МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Навчальний посібник

Укладач Ю.Є. Грудзинський, ст. викладач каф. АЕП

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньою програмою"Автоматизація та комп’ютерно-інтегровані технології кібер-енергетичних систем"
спеціальності 151 "Автоматизація та комп’ютерно-інтегровані технології",
освітньопрофесійною програмою

Електронне мережне навчальне видання

Київ

НТУУ "КПІ ім. Ігоря Сікорського"

2022

Рецензент	*Аушева Н.М.*, д.т.н., професор, зав. каф. ЦТЕ ННІАТЕ КПІ ім. Ігоря Сікорського
Відповідальний редактор	*Баган Т.Г.*, к.т.н, доцент каф. АЕП ННІАТЕ КПІ ім. Ігоря Сікорського

Гриф надано Методичною радою НТУУ "КПІ імені Ігоря Сікорського"
(протокол № 3 від 01.12.2022 р.)
за поданням Вченої ради Навчально-наукового інституту атомної і теплової енергетики
(протокол № 2 від 26 вересня 2022 р.)

Посібник «Алгоритми та структури даних: Навчальний посібник» призначений для здобувачів ступеня бакалавра освітньопрофесійної програми "Автоматизація та комп’ютерно-інтегровані технології кібер-енергетичних систем" за спеціальністю 151 "Автоматизація та комп’ютерно-інтегровані технології". У посібнику приділено увагу питанням рекурсії, роботі з нестандартними масивами, сортуванню та пошуку; роботі з абстрактними структурами даних типу списків, стеків та черг; роботі з деревами та хешуванням. У посібнику наведено велику кількість готових алгоритмів і програм на мові С, а також, завдань для самостійної роботи студента, що полегшує засвоєння матеріала.

Посібник буде також корисним для програмістів-початківців, студентів, які вивчають комп'ютерні науки, так і для всіх, хто працює з обчислювальною технікою і бажає зрозуміти, як створювати високоефективні алгоритми.

Реєстр № НП 22/23-208. Обсяг 9,7 авт. арк.

Національний технічний університет України
"Київський політехнічний інститут імені Ігоря Сікорського"
проспект Перемоги, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

© НТУУ "КПІ ім. Ігоря Сікорського", 2022

Зміст

1. ОСНОВИ АЛГОРИТМІЗАЦІЇ

1.1 Визначення алгоритму

Алгоритм - кінцева послідовність точно визначених дій, що призводять до однозначного вирішення поставленого завдання [1]**.** Алгоритм визначає кроки, згідно з якими завдання буде виконуватися. Ніхто не стане писати алгоритми для виконання примітивних завдань або створювати інструкції, щоб отримати доступ до четвертого елементу масиву (мається на увазі, що ви вже володієте мінімальними навичками програмування і знаєте, як це робиться). Як правило, алгоритми пишуться тільки для складних завдань, наприклад у тому випадку, коли потрібно знайти найкоротший шлях через мережу з сотень вулиць або відшукати найкращий варіант інвестицій для оптимізації прибутку. При цьому всі дані, необхідні для вирішення завдання, організовуються особливим чином в так звану структуру (структурний тип даних). Це може бути масив, зв'язний список, дерево, граф, мережа або щось більш хитромудре. Важливо розуміти, що **алгоритми не можуть існувати без структур даних.**

Головна особливість будь-якого алгоритму - формальне виконання, що дозволяє виконувати задані дії (команди) не тільки людині, але і різним технічним пристроям (виконавцям). Процес складання алгоритму називається алгоритмізацією. Алгоритми можуть бути задані: словесно, таблично, графічно (за допомогою схем алгоритмів). Словесний завдання описує алгоритм за допомогою слів і пропозицій. Табличне завдання служить для представлення алгоритму у формі таблиць і розрахункових формул. Графічне завдання, або схема алгоритму - спосіб подання алгоритму за допомогою геометричних фігур, які називаються блоками. Послідовність блоків і сполучних ліній утворюють схему алгоритму. Опис алгоритмів за допомогою схем алгоритмів є найбільш наочним і поширеним способом завдання алгоритмів (у процедурному програмуванні). Схема алгоритму розташовується зверху вниз і зліва направо. Лінії з'єднання окремих блоків показують напрямок процесу обробки інформації в схемі. Кожний такий напрямок називається гілкою. Алгоритм незалежно від його структури завжди має по одному блоку «Початок» і «Кінець». Його гілки повинні в кінці зійтися, і за якою б гілкою не було розпочато рух, алгоритм завжди має привести до блоку «Кінець».

Щоб зрозуміти роботу алгоритму, недостатньо просто розглянути його кроки. Потрібно з'ясувати ще кілька важливих моментів:

• **Поведінка алгоритму.** Чи знаходить він найкраще з можливих або просто гарне рішення? Чи може бути кілька найкращих рішень? Чи є сенс віддати перевагу одному з них?

• **Швидкість алгоритму.** Швидкий він чи ні? Чи уповільнює роботу тільки з деякими вхідними даними?

• **Вимоги до пам'яті алгоритму**. Скільки комп'ютерних ресурсів йому необхідно? Чи є такий обсяг прийнятним? Чи потрібні алгоритму мільйони терабайт пам'яті, якими комп'ютер не володіє (принаймні, зараз)?

• **Основні методи, використовувані в алгоритмі**. Чи можна задіяти їх повторно для вирішення подібних завдань?

1.2 Властивості алгоритму

Властивостями алгоритму є дискретність, визначеність, виконуваність, скінченність, результативність і масовість.

Дискретність алгоритму означає, що його виконання зводиться до виконання окремих дій (кроків) у певній послідовності. Причому, кожна команда алгоритму повинна виконуватися за скінченний проміжок часу.

Визначеність (або детермінованість) алгоритму означає, що для заданого набору значень початкових (вхідних) даних алгоритм однозначно визначає порядок дій виконавця і результат цих дій. Алгоритм не повинен містити команди, які можуть сприйматися виконавцем неоднозначно, наприклад, «Узяти дві-три ложки цукру», «Трохи підігріти молоко», «Вимкнути світло через кілька хвилин», «Поділити число x на одне з двох даних чисел a або b», тощо. Крім того, в алгоритмах недопустимі ситуації, коли після виконання чергової команди виконавцю неясно, яку команду він повинен виконувати наступною.

Виконуваність алгоритму означає, що алгоритм, призначений для певного виконавця, може містити тільки команди, які входять до системи команд цього виконавця. Так, наприклад, алгоритм для виконавця «Учень першого класу» не може містити команду «Побудуй бісектрису даного кута», хоча така команда може бути в алгоритмі, який призначений для виконавця «Учень восьмого класу».

Зазначимо, що виконавець повинен лише вміти виконувати кожну команду

зі своєї системи команд, і не важливо, розуміє він її, чи ні. Говорять, що виконання алгоритмів виконавцем носить формальний характер: виконавець може не розуміти жодну з команд, може не знати мети виконання алгоритму, і все одно отримає результат. Так, наприклад, верстат з програмним керуванням не розуміє жодної з команд, яку він виконує, але завдяки своїй конструкції успішно виготовляє деталі.

Скінченність алгоритму означає, що його виконання закінчиться після скінченної (можливо, досить великої) кількості кроків і за скінченний час при будь-яких допустимих значеннях початкових даних. Наведені вище послідовності команд є скінченними, а наступна послідовність команд — нескінченна:

1. Взяти число 2.

2. Помножити взяте число на 10.

3. Додати до одержаного числа 5.

4. Якщо одержане число додатне, то виконати команду 3, якщо ні, то припинити виконання алгоритму.

Результативність алгоритму означає, що після закінчення його виконання обов'язково одержуються результати, які відповідають поставленій меті. Результативними вважаються також алгоритми, які визначають, що дану задачу не можна розв'язати, або дана задача не має розв'язків при заданому наборі початкових даних.

Масовість алгоритму означає, що алгоритм може бути застосований до цілого класу однотипних задач, для яких спільними є умова та хід розв'язування, і які відрізняються тільки початковими даними.

Таким, наприклад, є алгоритм розв'язування квадратного рівняння, який дозволяє знайти дійсні корені квадратного рівняння з довільними дійсними коефіцієнтами, або визначити, що при певних значеннях коефіцієнтів рівняння не має дійсних коренів. Масовим також є, наприклад, алгоритм побудови бісектриси довільного кута з використанням циркуля і лінійки.

Однак, крім масових алгоритмів, складаються і застосовуються алгоритми, які не є масовими. Таким, наприклад, є алгоритм розв'язування конкретного квадратного рівняння, наприклад, $2x^2 + 5x + 2 = 0$ або алгоритм приготування конкретного салату (наприклад, грецького) на конкретну кількість осіб.

Хороший алгоритм повинен бути правильним, надійним і ефективним. Якщо він не справляється з поставленим завданням, від нього мало користі. Немає сенсу в ньому і тоді, коли він видає неправильний результат.

Якщо алгоритм не є надійним, використовувати його в програмі стає небезпечно. По-справжньому гарний код повинен бути простим, інтуїтивно зрозумілим і витонченим. Тільки в цьому випадку можна отримати правильний результат або внести необхідні зміни в структуру програми. Якщо алгоритм складний і заплутаний, виникнуть проблеми не тільки з його реалізацією, але і з усуненням можливих помилок. Під сумнів можна буде поставити і сам результат: адже якщо алгоритм незрозумілий, як дізнатися, правильно він працює чи ні?

Більшість розробників багато часу приділяють ефективності алгоритмів. І не дарма. Правильний результат, легкість реалізації і можливість виправити помилки - все це зведеться до нуля, якщо алгоритм закінчить свою роботу тільки через сім років або зажадає обсяг пам'яті більше того, що є в комп'ютері.

Щоб перевірити ефективність алгоритму, програмісти задаються питанням: як зміниться продуктивність коду, якщо скоригувати розмірність завдання? Якщо подвоїти кількість змінних в алгоритмі, як зміниться час його роботи? А якщо звести початкову кількість змінних в четвертий степінь, чи не стане отриманий алгоритм виконуватися кілька років?

У такому ж ключі слід міркувати про витрату оперативної пам'яті і інших ресурсах, необхідних для роботи алгоритму. Варто потурбуватися і над продуктивністю алгоритму при різних умовах. Який найгірший варіант? Наскільки велика ймовірність його виникнення? Якщо запустити алгоритм з використанням великого обсягу довільних даних, то якою буде його середня продуктивність?

Щоб отримати уявлення про те, як складність завдання впливає на продуктивність, програмісти користуються поняттям «асимптотична складність».

1.3 Асимптотична складність алгоритму

Асимптотична складність (продуктивність) визначається функцією, яка вказує, наскільки погіршується робота алгоритму з ускладненням поставленого завдання. Таку функцію записують у круглих дужках, позначаючи великою літерою *O*.

Наприклад $O(N^2)$ означає, що в міру збільшення кількості вхідних даних час роботи алгоритму (використання пам'яті або інший вимірюваний параметр) зростає квадратично. Якщо даних стане вдвічі більше, продуктивність алгоритму погіршиться приблизно в чотири рази. При збільшенні кількості вхідних даних в три рази вона стане менше в дев'ять разів.

Існують п'ять основних правил спрощення при розрахунку асимптотичної складності довільного алгоритму:

1. Якщо для математичної функції f алгоритму необхідно виконати певні дії f(N) разів, то для цього йому знадобиться зробити O(f(N)) кроків.

2. Якщо алгоритм виконує одну операцію, що складається з O(f(N)) кроків, а потім другу операцію, що включає O(g(N)) кроків, то загальна продуктивність алгоритму для послідовного виконання функцій f і g складе O(f(N)+g(N)).

3. Якщо алгоритм виконує O(f(N)+g(N)) кроків і область значень N функції f(N) більша, ніж у функції g(N), то асимптотичну складність можна спростити до значення O(f(N)).

*4. Якщо алгоритму всередині кожного кроку O(f(N)) однієї операції доводиться виконувати ще O(g(N)) кроків іншої операції, то загальна продуктивність алгоритму складе O(f(N)*g(N)).*

*5. Постійними множниками(константами) під O можна знехтувати. Якщо C є константою, то O(C*f(N)) або O(f(C*N)) можна записати просто як O(f(N)).*

Нижче наведено кілька прикладів, які полегшать розуміння цих правил.

Правило 1

Якщо для математичної функції f алгоритму необхідно виконати певні дії f(N) разів, то для цього йому знадобиться зробити O(f(N)) кроків.

Приклад 1.1. Алгоритм функції знаходження найбільшого цілого числа в масиві.

```
int FindLargest(int array[], int max)
{
        int largest = array[0];              // O(1)
        for (int i = 1; i < max; i++) {          // O(N)
                if (array[i] > largest)
                        largest = array[i];
```

```
            }
        return largest;                    // O(1)
    }
```

В якості вхідного параметра алгоритм *FindLargest* використовує масив цілих чисел, результат повертається у вигляді одного цілого числа. На самому початку змінній *largest* присвоюється значення першого елемента масиву. Потім алгоритм перебирає елементи які залишилися і порівнює значення кожного з них з *largest*. Якщо він знаходить більшу величину, то прирівнює *largest* до неї і після закінчення циклу повертає найбільше знайдене значення.

Алгоритм перевіряє кожен з *N* елементів масиву за один раз, тому його продуктивність становить *O(1)+O(N)+O(1)*.

Правило 2

Якщо алгоритм виконує одну операцію, що складається з O(f(N)) кроків, а потім другу, що вимагає O(g(N)) кроків, то загальна продуктивність алгоритму для послідовного виконання функцій f і g буде O(f(N)+ g(N)).

Повернемося до алгоритму FindLargest. На цей раз звернемо увагу, що кілька рядків не включені в цикл. У Приклад 1.1 в коментарях праворуч наведено порядок часу виконання операторів програми.

Отже, наведений алгоритм виконує один крок перед циклом і ще один після нього. Кожен з них має продуктивність *O(1)* (це одноразові дії), тому загальний час роботи алгоритму складе *O(2+N)*.

Правило 3

Якщо алгоритму необхідно зробити O(f(N)+g(N)) кроків і область значень N функції f(N) більше, ніж у функції g(N), то асимптотичну складність можна спростити до O(f(N)).

У попередньому прикладі 1.1 ми з'ясували, що час роботи алгоритму FindLargest визначається виразом *O(2+N)*. Якщо параметр *N* почне зростати, його значення значно перевищить постійну величину *2* і попередній вираз можна буде спростити до *O(N)*.

Ігнорування менших функцій дозволяє знехтувати невеликими частинами алгоритму, щоб зосередити увагу на асимптотичну поведінку головної частини алгоритму, яка виявляється при ускладненні завдання. Іншими словами час,

витрачений алгоритмом на побудову простих структур даних перед виконанням об'ємного обчислення, є несуттєвим у порівнянні з тривалістю основних розрахунків.

Правило 4

*Якщо алгоритму всередині кожного кроку O(f(N)) однієї операції доводиться виконувати ще O(g(N)) кроків іншої операції, то загальна продуктивність алгоритму складе O(f(N)*g(N)).*

Приклад 1.2. Алгоритм функції, який визначає, чи містяться в масиві елементи, що повторюються. (Варто зазначити, що цей варіант не є найефективнішим способом виявлення дублікатів).

```
bool ContainsDuplicates (int array [], int max) {
// цикл по всіх елементах масиву
for (int i  = 0;  i < max; i++)            // O(N)
        for (int j = 0; j < max; j++)              // O(N)
        // перевіряємо, чи є два елементи дублікатами.
        if (i != j)
           if (array [i] == array [j])
              return true;
// Якщо ми дійшли до цього рядка, то дублікатів немає
return false;                              // O(1)
}
```

Алгоритм містить два цикли, один з яких є вкладеним. Зовнішній цикл перебирає всі елементи масиву *N*, виконуючи *O(N)* кроків. Усередині кожного такого кроку внутрішній цикл повторно переглядає всі *N* елементів масиву, здійснюючи ті ж *O(N)* кроків. Отже, загальна продуктивність алгоритму складе $O(N*N) = O(N^2)$.

Правило 5

*Постійними множниками (константами) під O можна знехтувати. Якщо C є константою, то O(C*f(N)) або O(f(C*N)) можна записати як O(f(N)).*

Знову подивимося на Приклад 1.2 алгоритму ContainsDuplicates і звернемо увагу на внутрішній цикл, який представлений умовою *if*. В рамках цієї умови визначається, чи рівні один одному індекси *i* та *j*. Якщо ні - тоді порівнюються

величини *array[i]* і *array[j]*, в разі їх збігу повертається значення *true*.

Нехтуючи додатковим кроком в операторі *return* (як правило, він виконується один раз), припустимо, що спрацьовують обидва оператора *if* (а так і відбувається в більшості випадків), тоді внутрішній цикл буде пройдений за *O(2N)* кроків. Отже, загальна продуктивність алгоритму складе $O(N*2N) = O(2N^2)$. Останнє правило дозволяє знехтувати коефіцієнтом *2* і записати продуктивність алгоритму у вигляді $O(N^2)$.

Дійсно, припустимо, ми збільшимо *N* в два рази, тобто будемо оперувати значенням *2N*. Тепер, якщо підставити це значення у вираз $2N^2$, вийде наступне: $2*(2N)^2 = 2 * 4N^2 = 8N^2$. Це і є наша величина $2N^2$, тільки помножена на *4*. Таким чином, час роботи алгоритму збільшиться в чотири рази.

Тепер оцінимо продуктивність алгоритму, використовуючи спрощений за правилом вираз $O(N^2)$. При підстановці в нього *2N* отримаємо наступне: $(2N)^2 = 4N^2$. Тобто наша початкова величина N^2зросла в чотири рази, як і час роботи алгоритму.

З усього вищесказаного випливає, що незалежно від того, будемо ми використовувати розгорнуту формулу $2N^2$ або обмежимося просто N^2, результат залишиться тим самим: збільшення складності завдання в два рази сповільнить роботу алгоритму в чотири рази. Таким чином, важливою тут є не константа *2*, а той факт, що час роботи збільшується разом зі збільшенням кількості вводів N^2 у чотири рази.

1.4 Різновиди функцій асимптотичної складності

1.4.1 Константна продуктивність *O(1)*

Алгоритм з продуктивністю *O(1)* виконується за один і той же відрізок часу, незалежно від складності завдання. Як правило, мова йде про обмежене коло команд, оскільки за *O(1)* часу неможливо навіть переглянути всі вхідні дані.

Наприклад, в якийсь момент, алгоритму швидкого сортування необхідно вибрати з масиву чисел те, яке буде дорівнювати середньому значенню. Тобто, якщо у нас є масив цілих чисел від 1 до 100, найкращим значенням виявиться число 50. У наведеному нижче алгоритмі показаний загальний підхід до вирішення цього завдання.

```
int DividingPoint (int array [], max) {
        int number1 = array [0];
        int number2 = array [max];
        int number3 = array [max / 2] ;
        if (number1 > number2 && number1 < number3)
                return number1;
        else if (number2 > number1 && number2 < number3)
                return number2;
        else
                return number3;
}
```

Подібний алгоритм вибирає три величини (на початку, кінці і середині масиву), порівнює їх між собою і повертає середнє значення. Не факт, що вибір виявиться найкращим, але цілком імовірно, що гіршим також не буде.

Оскільки алгоритм виконує лише кілька певних кроків, його продуктивність дорівнює *O(1)*, а час роботи не залежить від кількості вводів *N*.

1.4.2 ***Log(N)***

Алгоритм з продуктивністю *O(log(N))* ділить кількість розглянутих ним елементів на фіксований коефіцієнт при кожному кроці. На Рисунок 1.1 показано впорядковане повне бінарне дерево. Бінарне воно тому, що кожна вершина має не більше двох гілок. Повний тому, що кожен рівень (крім останнього) цілком заповнений, а всі вершини на останньому рівні згруповані в лівій частині. І, нарешті, впорядковане тому, що значення кожної вершини порівнянно по величині зі значенням лівого дочірнього вузла і не перевищує значення правого дочірнього вузла.

Наступний псевдокод описує пошук певного елемента по дереву, зображеному на Рисунок 1.1.

```
Node: FindItem (Integer: target_value)
        Node: test_node = <корінь дерева>
        Do Forever
                // Якщо ми випадаємо з дерева, значить, шуканої величини немає.
                If (test_node = null)
```

```
        Return null
        If (target_value = test_node.Value) Then
            // test_node містить шукану величину.
            // Це і є та вершина, яку ми шукали.
            Return test_node
        Else If (target_value <test_node.Value) Then
            // Переходимо по лівій гілці.
            test_node = test_node.LeftChild
        Else
            // Переходимо по правій гілці.
            test_node = test_node.RightChild
        End If
    End Do
End FindItem
```

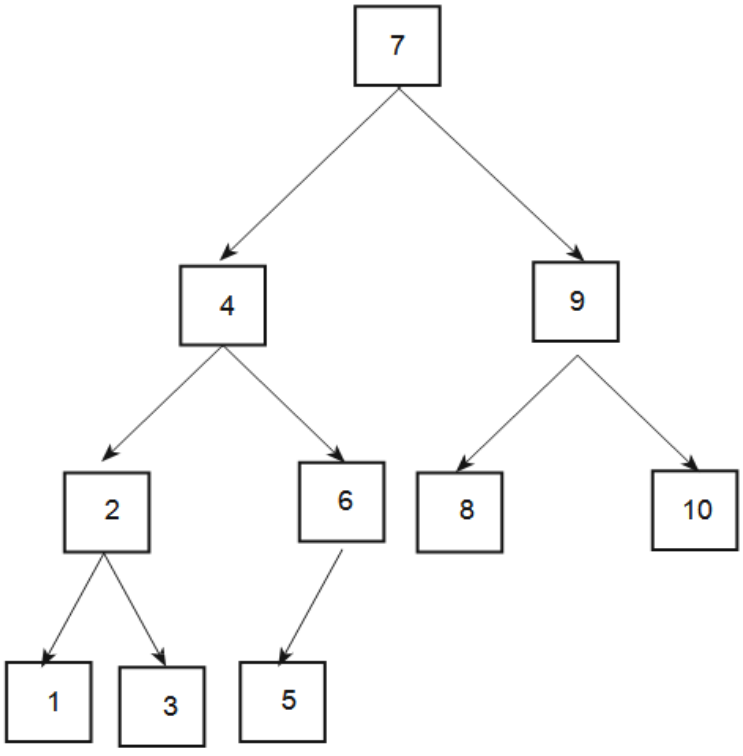


Рисунок 1.1 Приклад впорядкованого повного бінарного дерева

Алгоритм оголошує змінну *test_node* і присвоює їй початкове значення, розташоване у корені дерева. Потім запускається нескінченний цикл.

Якщо змінна *test_node* дорівнює *null*, шуканої величини на дереві немає - і алгоритм повертає *null*. *Null* - особлива величина, яку можна використовувати в

якості змінної для вказівки на об'єкт, наприклад на вершину або на дерево. Її значення можна перевести як «ця змінна ні на що не вказує».

Якщо *test_node* містить шукану величину, то змінна є вершиною, яку ми шукаємо, і алгоритм повертає її.

Коли шукана величина *target_value* менше величини *test_node*, алгоритм прирівнює *test_node* до дочірньої вершини зліва, тобто до *LeftChild*. (При *test_node,* що знаходиться на листі (внизу) дерева, *LeftChild* дорівнює *null*, а алгоритм виправляє положення під час наступного проходження циклу).

Якщо величина *test_node* не дорівнює *target_value* і не менше її, значить, вона більше. У цьому випадку алгоритм вважає значення *test_node* рівним значенню дочірньої вершини справа, тобто *RightChild*. (І знову, якщо *test_node* розташовується знизу дерева, то *RightChild* дорівнює *null*, а алгоритм виправляє становище при наступному проходженні циклу)

Таким чином змінна *test_node* поступово проходить по дереву зверху вниз і, нарешті, або виявляє шукану величину, або випадає з дерева (*test_node* дорівнює *null*).

У цьому алгоритмі важливо знати, наскільки низько переміститься *test_node,* поки не знайде *target_value* або не випаде з дерева. Якщо пощастить, шукана величина знайдеться відразу - наприклад, число 7 (див. Рисунок 1.1) виявиться за один крок. Перевірка невеликої частини дерева може знадобитися і в тому випадку, коли потрібне значення розташовується не в кореневій вершині, як, наприклад, число 4. Найгіршим варіантом вважається той, при якому алгоритму доведеться проводити пошук по всьому дереву - від кореневої вершини до низу.

Наступне важливе питання - наскільки високим є дерево. Повне завершене бінарне дерево висотою H включає приблизно 2^H вершин. Таким чином, N вершин складуть дерево висотою $H = log_2(N)$. Оскільки алгоритм рухається по дереву зверху вниз в гіршому (і близькому до нього) випадку, а саме дерево має висоту приблизно $log_2(N)$, робота алгоритму займе час, що дорівнює $O(log_2(N))$.

На цьому етапі виявляється цікава особливість. Як відомо, можна перетворити логарифм з основою A в логарифм з основою B, використовуючи формулу

$$log_B x = log_A x / log_A B$$

Якщо покласти $B = 2$, то заміна основи логарифму на A в логарифмі виразу $O(\log_2(N))$ буде виглядати наступним чином:

$O(\log2(N)) = O(\log A(N) / \log A2)$.

Значення $1/\log_A 2$ постійно при будь-якому A, і оскільки асимптотична продуктивність алгоритму O нехтує величинами, помноженими на константу, то $O(\log_2(N))$ буде дорівнює $O(\log_A(N))$ для будь-якого логарифма з основою A. З цієї причини час роботи можна записати просто, як $O(\log(N))$ не вказуючі основу логарифма.

Логарифмічна функція $\log(N)$ зі збільшенням N зростає порівняно повільно, тому алгоритми з продуктивністю $O(\log(N))$ зазвичай досить швидкі. В цьому і полягає їх головна користь.

1.4.3 *N*

Алгоритм *FindLargest* (див. Правило 1 пункту 1.3) має продуктивність $O(N)$. Функція N зростає швидше, ніж $\log(N)$, але все ж не так швидко, тому більшість подібних алгоритмів демонструє на практиці хорошу продуктивність.

1.4.4 *N*log(N)*

Припустимо, що алгоритм перебирає всі елементи в поставленому завданні, а потім в окремому циклі виконує з елементом якусь операцію $O(\log(N))$.

У цьому випадку продуктивність алгоритму визначається виразом $O(N*\log(N))$. Як варіант, завдання може виглядати так: алгоритм виконує операцію $O(\log(N))$ і на кожному її кроці щось робить з кожним елементом.

Припустимо, у нас є відсортоване дерево, що містить N елементів (як описано вище) і такої ж розмірності масив. Треба дізнатися, які з елементів масиву присутні в дереві. Один із способів з'ясувати це - здійснити циклічне проходження по величинам масиву, задіявши описаний раніше метод пошуку по дереву. В процесі роботи алгоритм перевірить N елементів масиву і виконає $\log(N)$ кроків по дереву для кожного з них, так що загальна продуктивність буде $O(N*\log(N))$.

Для багатьох алгоритмів сортування, заснованих на порівнянні елементів, час роботи якраз і становить $O(N*\log(N))$.

1.4.5 *N²*

Алгоритм, який спочатку перебирає всі вхідні дані, а потім передивляється

їх ще раз для кожного одиничного значення, має продуктивність $O(N^2)$. Наприклад, алгоритм *ContainsDuplicates* (див. Правило 4 пункту 1.3) виконується саме за такий час. Можливі й інші степені N, припустимо $O(N^3)$ або $O(N^4)$.Однак вони є значно повільнішими, ніж $O(N^2)$.

Кажуть, алгоритм має многочленний час роботи, якщо він включає в себе многочлен N: наприклад $O(N)$, $O(N^2)$, $O(N^6)$ і т.д. У будь-якому випадку подібні завдання можна вирішити. А ось експоненціальний і факторіальний час роботи, описані нижче, зростає дуже швидко, тому алгоритми з вказаною продуктивністю можуть бути застосовані тільки для невеликої кількості вхідних даних.

1.4.6 2^N

Експонентні функції, такі як 2^N, зростають блискавично і тому корисні для вирішення лише обмеженого кола завдань. Зазвичай, за допомогою алгоритмів з подібним часом роботи шукається оптимальний набір вхідних даних.

Як приклад розглянемо наступну задачу. Припустимо, у нас є рюкзак певної вантажопідйомності і набір речей, кожна з яких має свою масу і значення корисності. Необхідно наповнити рюкзак предметами так, щоб їх загальне значення корисності в заповненому рюкзаці виявилося найбільшим.

Завдання може здатися простим, але всі відомі алгоритми для знаходження найкращого рішення жадають розгляду кожної можливої комбінації предметів. Якщо врахувати, що будь-який предмет здатний перебувати тільки в двох станах (перебувати в рюкзаку або поза ним), то, помноживши кількість станів на кількість предметів, отримаємо $2*2*...*2 = 2^N$ варіантів вибору.

Іноді немає необхідності перевіряти всі можливі комбінації. Наприклад, якщо перший предмет заповнює собою весь рюкзак, то додавати до нього ще щось вже не має сенсу. Однак в більшості випадків не вдається виключити достатню кількість варіантів, щоб значно звузити область пошуку.

При вирішенні завдань з експоненціальним часом роботи часто користуються евристиками - алгоритмами, які дають хороші результати, але не гарантують, що вони будуть найкращими.

1.4.7 $N!$

Функція $N!$ використовується для роботи з цілими числами більше 0 і визначається формулою $N! = 1*2*3*...*N$. Вона зростає набагато швидше, ніж

експоненціальна функція 2^N.

В алгоритмах з факторіальним часом роботи, як правило, шукається оптимальний розподіл вхідних даних. Наприклад, у торгового представника є список міст. Його завдання - скласти маршрут таким чином, щоб відвідати кожен населений пункт один раз і повернутися в відправну точку, подолавши мінімальну відстань.

Якщо міст небагато, то все просто, але якщо список довгий, завдання ускладнюється на порядок. Найбільш очевидний спосіб вирішення - перебрати всі можливі варіанти маршруту. Так, за допомогою алгоритму можна вибрати *N* прилеглих населених пунктів для першого міста. Після цього у вас залишиться *N-1* можливих пунктів, які ще потрібно відвідати, потім *N-2* і т. Д. Отже, загальна кількість варіантів маршрутів складе *N * (N-1) * (N-2) * ... * 1 = N!*

У таблиці нижче представлені значення для функцій, які визначають час роботи алгоритмів, описаних вище. Вона допоможе оцінити, наскільки швидко зростають ці функції.

Таблиця 1.1 Величини О-функцій для різних вхідних даних

n	$log_2 N$	*N*	N^2	2^N	*N!*
1	0	1	1	2	1
5	2,32	5	25	32	625
10	3,32	10	100	1·024	$1*10^9$
15	3,90	15	225	$3,3*10^4$	$2,9*10^{16}$
20	4,32	20	400	$1*10^6$	$5,24*10^{24}$
50	5,64	50	2500	$1,1*10^{15}$	$1,8*10^{83}$
100	6,64	100	$1*10^4$	$1,3*10^{30}$	$1*10^{198}$
1·000	9,96	1·000	$1*10^6$	$1,1*10^{301}$	–
10·000	13,28	$1*10^4$	$1*10^8$	–	–
100·000	16,60	$1*10^5$	$1*10^{10}$	–	–

На рисунку нижче наведені відповідні графіки. Для наочності масштаб деяких з них був змінений, але і так зрозуміло, яка вони себе ведуть при збільшенні *n*: функції десяткового логарифма, лінійних і навіть показникових зростають з помірною швидкістю, в той час як експоненціальні і факторіальні роблять це надзвичайно швидко.

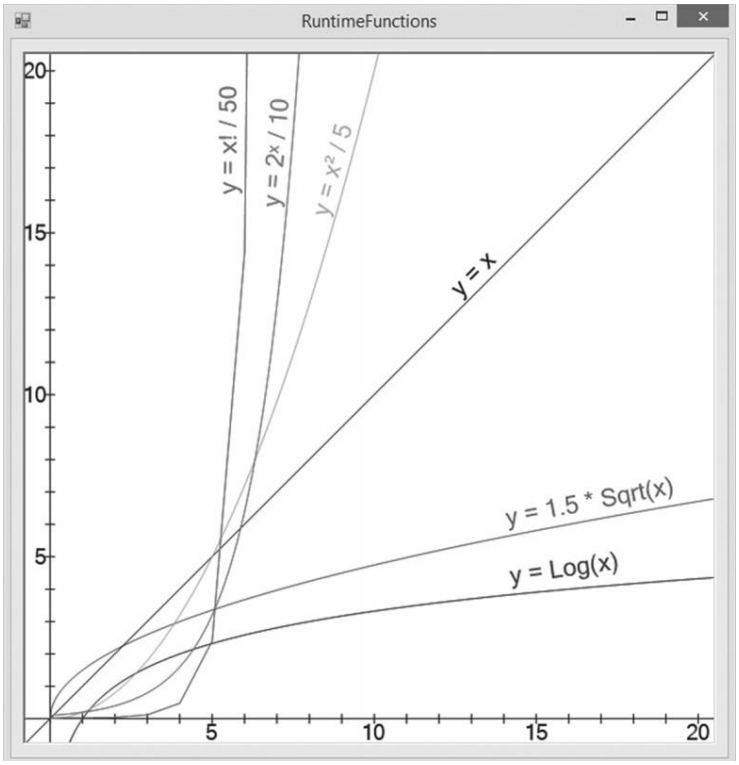


Рисунок 1.2 Графіки зростання різних O-функцій

1.5 Практичні рекомендації

Теорія дозволяє зрозуміти, як змінюється робочий цикл алгоритму, проте не менш важливі і практичні моменти. Наприклад, при аналізі алгоритму вважається, що всі кроки алгоритму виконуються за однакову кількість часу, навіть якщо це не відповідає дійсності. Взяти хоча б створення і видалення нових об'єктів - на них потрібно набагато більше часу, ніж на переміщення цілих чисел з однієї частини масиву в іншу. В такому випадку алгоритм з масивами буде працювати краще, ніж той, що використовує солідну кількість об'єктів і виглядає краще в асимптотичному плані.

Крім того, багато середовищ програмування надають доступ до функцій операційної системи, які є більш ефективними, ніж основні технології алгоритмізації. Наприклад, частина алгоритму, яка виконує сортування, вимагає переміщення елементів масиву на одну позицію вниз, щоб перед ними можна було вставити новий елемент. Цей досить повільний процес призводить до того, що робота алгоритму визначається як $O(N^2)$. Однак багато програм можуть використовувати особливу функцію, що переміщує зразу блок пам'яті (наприклад, функція *memmove()* у стандартній бібліотеці C/C++). Таким чином, замість переміщення одного елемента при проходженні через масив програма змушує функцію пересувати цілий блок елементів, значно прискорюючи процес.

Навіть якщо алгоритм має оптимальну асимптотичну складність, для збільшення продуктивності допустимо використовувати інструменти, які пропонує середовище програмування. Деякі з них здатні виконувати ті ж завдання, які розглядаються в цьому курсі. Наприклад, багато бібліотек містять програми сортування та відмінно підходять для роботи з масивами даних. Так, стандартна бібліотека мови Сі містить реалізацію функції *qsort()*, швидкість роботи якої навряд чи вдасться перевершити у власному коді.

Подібні специфічні бібліотеки можуть бути доступними для широкого загалу і вельми корисні при виконанні певних завдань. Наприклад, ефективно використовувати бібліотеку аналізу мережі, замість того щоб самому розробляти деякі інструменти, і заощадити час, припустимо, на побудові дерев або сортуванні. Звичайно, ніщо не заважає виконувати ці завдання власноруч, але звернутися до готової бібліотеки набагато простіше.

Якщо в інструментах програмування є функції, які замінять роботу будь-якого алгоритму, обов'язково скористайтеся ними. Так, найчастіше, можна домогтися кращої продуктивності і витратити менше сил на відлагоджування.

Нарешті, для дуже великих завдань найефективніший алгоритм не завжди є найшвидшим. Припустимо, ми сортуємо довгий список чисел. У цьому випадку алгоритм швидкого сортування спрацює добре. Але якщо у нас всього три числа, простий ряд операторів *If*, можливо, виявить себе куди краще. Навряд чи матиме значення, впорається програма з завданням за 1 або за 2 мс. Якщо ви не збираєтеся виконувати сортування багаторазово, розумніше вдатися до простого алгоритму, який легше піддається відлагоджуванню, а не до складного, який заощадить всього 1 мс.

Якщо ви користуєтеся бібліотеками, які аналогічні тим, що описані вище, можливо, вам не доведеться розробляти всі алгоритми самостійно, але розуміти принцип їх дії вкрай необхідно, щоб отримати якомога більше користі з інструментів, які застосовуються для їх реалізації. Наприклад, якщо відомо, що реляційні бази даних працюють з B-деревами і чимось подібним для зберігання індексів, то буде ясно, наскільки важливі є коефіцієнти попереднього розподілу та заповнення.

Розуміння алгоритмів може стати в нагоді і в інших ситуаціях. Можливо, вам

не знадобиться сам алгоритм сортування злиттям, але буде корисний його метод «розділяй і володарюй» для вирішення завдань з параллельними процесами.

1.6 Контрольні завдання і запитання

Зірочкою позначено завдання підвищеної складності.

1. У прикладі 1.2 було описано алгоритм ContainsDuplicates, що має час роботи $O(N^2)$. Розглянемо його покращену версію.

```
Bool ContainsDuplicates(int array[])
    //Цикл з усіх елемент ів масиву, крім останнього.
    for (i = 0; <найбільший індекс > - 1; i++) {
        //Цикл елементів після елемента i.
        for (j = i++; <найбільший індекс>; j++) {
            // Перевіряємо, чи є два елементи дублікатами.
            if (array[i] == array[j])
                return true;
        }
    }

    // Якщо ви дійшли цього рядка, то дублікатів немає.
    return false;
} // ContainsDuplicates
```

Визначте час роботи нової версії алгоритму.

2. У Таблиця 1.1 показано взаємозв'язок між складністю задачі N і продуктивністю алгоритмів, описаних різними функціями. Цю ж залежнсть можна продемонструвати ще одним способом — визначити максимальну складнсть задач, яку здатний виконати комп'ютер з певною швидкістю протягом конкретного часу.

Припустимо, що комп'ютер робить 1 млн. кроків алгоритму за 1 с. Алгоритм виконується протягом часу $O(N^2)$. У такому разі через 1 годину машина вирішить завдання, у якого $N = 60\,000$ (оскільки $60\,000^2$ =3 600 000 с – це час, потрібний для вирішення задачі розміром N). Створити таблицю, в якій буде представлено найбільший розмір задачі N, яку зможе виконати комп'ютер для кожної із функцій таблиці за: секунду, хвилину, годину, день, тиждень, місяць.

3. Іноді константи, якими нехтують в асимптотичному позначенні, відіграють дуже важливу роль. Уявіть, що два різні алгоритми виконують однакове завдання.

Для роботи першого потрібно *1500N* кроків, а другому потрібно $30N^2$ кроків. При яких значенях *N* ви надасте перевагу тому, чи іншому алгоритму?

4*. У вас є два алгоритми. Один складається з $\frac{N^3}{75} - \frac{N^2}{4} + N + 10$ кроків, а другий з $\frac{N}{2} + 8$ кроків. Яким має бути значення *N*, щоб ви вибрали перший/другий алгоритм?

5. Програма бере у якості вхідних параметрів *N* літер і генерує з них усі можливі пари літер. Наприклад, з літер ABCD генеруються пари AB, AC, AD, BC, BD, CD (розуміється, що AB та BA – це одна пара). Яким буде час роботи цього алгоритму?

6. Як ви вважаєте: чи може бути алгоритм без структури даних? Чи може існувати структура даних без алгоритму? Навести відповідні приклади.

7. Розглянемо два алгоритми фарбування паркану.

```
Algorithm1()
    For i = 0 To <кількість дощок у паркані> - 1
        <Фарбуємо дошку під номером i>
    Next i
End Algorithm1

Algorithm2(Integer: first_board, Integer: last_board)
    If (first_board == last_board) Then
        // Маємо лише одну дошку. Фарбуємо лише її.
        <Фарбуємо дошку first_board.>
    Else
        // Дощок більше однієї, ділимо їх на дві групи і фарбуємо рекурсивно.
        Integer: middle_board = (first_board + last_board) / 2
        Algorithm2(first_board, middle_board)
        Algorithm2(middle_board, last_board)
    End If
End Algorithm2
```

Який час роботи цих алгоритмів? (*N* – кількість дощок у паркані).

Який варіант кращий?

2 ОСНОВИ РЕКУРСИВНИХ АЛГОРИТМІВ

2.1 Основні визначення

Рекурсивним називається спосіб побудови об'єкта (поняття, системи, опис дії), в якому визначення об'єкта включає аналогічні об'єкти (поняття, системи, дії) у вигляді складових частин [1]. Приклади рекурсії у житті:

- *«У попа була собака, він її любив.*
  Вона з'їла шматок м'яса, він її убив.
  Каменем придавив, і на камені написав:
  «У попа була собака ...»
  (Дитяча лічилка)
- *Я хочу Вам написати, що я хочу Вам написати, що я хочу Вам написати. . . (З листа пацієнта психіатра)*
- *«Я знаю, що ти знаєш, що я знаю»*
  (Кінокомедія італійського кінорежисера Альберто Сордо, 1982).

Всі подібні приклади рекурсії помічають її головну властивість: деяка частина системи відтворює сама себе. Тут же зазначається особливість такого підходу: рекурсія не властива буденному сприйняттю, безумовна рекурсія є нескінченною і безглуздою. Обмежена рекурсія з певною кількістю рівнів - парадоксальна і небезпечна. Наприклад, *рефлексія* (від лат. Reflexio - звернення назад) - самопізнання, самоспостереження, образне уявлення самого себе в третій особі (див., наприклад, «Поєдинок» О. Купріна).

В ході рекурсії алгоритм *F* викликає сам себе. Якщо він робить це безпосередньо, то рекурсія називається прямою:

```
F(n)   (рекурсивний алгоритм)
       If (n=n0) then        (перевірка зупинки рекурсії)
            F ← C            (пряме обчислення F при n=n0)
       else
            F ← ...F(m)...   (рекурсивний виклик, m < n)
        Return (F)
End F
```

Якщо алгоритм *F* викликає себе через інший алгоритм *G* - рекурсія називається непрямою:

```
F(n)   (перша непряма рекурсивна функція)
       If (n=n0) then        (перевірка зупинки рекурсії)
          F ← Cf    (пряме обчислення F при n=n0)
       else
          F ← ...G(k)...     (рекурсивний виклик, k<n)
       Return(F)
End.

G(m)                         (друга непряма рекурсивна функція)
       If (m=m0) then        (перевірка зупинки рекурсії)
             G ← Cg          (пряме обчислення G при m=0)
       else
             G ← F(j)        (рекурсивний виклик, j<m)
       Return(G)
End.
```

Рекурсія може бути і множинною (виклик здійснюється кілька разів всередині роботи самого алгоритму *F*).

На перший погляд поняття здається дещо складним, оскільки людина не мислить рекурсивно. Наприклад, займаючись фарбуванням паркану, ми почнемо з одного краю і будемо рухатися до іншого. Навряд чи кому прийде в голову розділити паркан на дві частини (ліву і праву), а потім фарбувати кожну половину окремо (а ті, в свою чергу - ще навпіл!). Проте багато завдань, які рекурсивні за своєю природою, а їх структура і рішення легко відстежуються за допомогою саме рекурсивного алгоритму. Такі, наприклад, програми, що будують дерева і проводять операції на них.

Ми познайомимося з деякими рекурсивними алгоритмами. І хоча вони корисні вже самі по собі, набагато важливіше розуміти закладений в них принцип.

Усвідомивши його, ви почнете помічати рекурсію в багатьох ситуаціях, що мають відношення до програмування. Слід мати на увазі, що рекурсія - не завжди найкращий спосіб вирішення завдань, в деяких випадках вона знижує продуктивність програми, в зв'язку з чим ми розглянемо ще і способи видалення рекурсії.

2.2 Базові види рекурсії

2.2.1 Обчислення факторіала

Факторіал числа *N* визначається через функцію *factorial(N)*, яку можна задати рекурсивно наступними формулами:

$$\begin{cases} 0! = 1, \\ N! = N(N-1)! \end{cases}$$

Тоді розрахунок *3!* буде виглядати так:

*3! = 3*2! = 3*2*1! = 3*2*1*0! = 3*2*1*1*

Це визначення призводить до простого рекурсивного алгоритму.

Приклад 2.1 Рекурсивний алгоритм обчислення факторіала

```
Factorial (n)
    If (n =0) Then
       Return 1
    else
       Return n * Factorial(n - 1)
End Factorial
```

Якщо вхідне значення *n = 0*, то алгоритм повертає *1*, що відповідає першому рівнянню, яке визначає функцію *factorial*. В іншому випадку код повертає добуток числа *n* на факторіал *(n – 1)*, згідно з другим рівнянням.

Звідси випливають дві важливі характеристики, яким повинні відповідати всі рекурсивні алгоритми:

- кожен раз при новому запуску алгоритм повинен знижувати складність завдання (розмірність задачі), а в кінці викликає себе останній раз, щоб розв'язати найпростіший (тривіальний) варіант. У наведеному прикладі розрахунок *n!* перетворився в розрахунок *(n - 1)!*, з подальшим множенням на число *n*;
- рекурсія повинна бути обмеженою. У розглянутому випадку вхідний параметр

n зменшується з кожним викликом, поки не стає рівним *0*. В цей момент алгоритм повертає *1* - і рекурсивні виклики припиняються.

Недолік вищенаведеного алгоритму тільки один: при виконанні функції *factorial* з вхідним параметром *-1*, рекурсія ніколи не зупиниться. Щоб уникнути цього, потрібно змінити перший оператор в алгоритмі, наприклад, на такий *If (n ≤ 0) Then Return 1*. Тоді алгоритм прийме наступний вигляд.

Приклад 2.2 Безпечний рекурсивний алгоритм обчислення факторіала числа

```
Factorial (n)
    If (n ≤0) Then
        Return 1
    else If (n =0) Then
        Return 1
    else
        Return n * Factorial(n - 1)
End Factorial
```

Тут вже при виклику з негативним параметром повернеться *1* (хоча, з точки зору математики, і це неправильно! Негативні числа факторіала не мають).

В цілому аналізувати час роботи рекурсивних алгоритмів непросто, але не в даному випадку. При вхідному значенні *N* алгоритм викликає сам себе *N + 1* раз, щоб виконати розрахунок *N!, (N - 1)!, (N - 2)!, ..., 0!*. Тут немає нічого складного, тому загальна продуктивність складає *O(N)*.

Оскільки алгоритм викликає сам себе *N + 1* раз, максимальна глибина рекурсії дорівнює *O(N)*. У деяких середовищах програмування вона може бути обмежена розміром виділеного програмі стека, а це тягне за собою певні проблеми. Тому, при використанні рекурсивних алгоритмів у програмі, програміст повинен стежити за розміром виділеного програмі стека і змінити його відповідно до максимально можливої глибини рекурсії.

2.2.2 Числа Фібоначі

Числа Фібоначі визначаються наведеними нижче рівняннями.

Fibonacci (0) = 0

Fibonacci (1) = 1

Fibonacci (n) = Fibonacci (n – 1) + Fibonacci (n – 2), для n > 1

Наприклад, перші 12 чисел Фібоначчі - це 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 і 89. Згідно з визначенням, отримуємо наступний алгоритм:

Приклад 2.3 Рекурсивний алгоритми обчислення чисел Фібоначчі (невдалий)

```
Fibonacci (n)
    If (n ≤1) Then
      Return n
    else
      Return Fibonacci(n - 1) + Fibonacci(n - 2)
End Fibonacci
```

Якщо вхідне значення *n* дорівнює *0* або *1*, на виході повертається *0* або *1*. (Якщо воно менше 1, повертається саме вхідне значення) У разі коли *n* більше *1*, алгоритм викликає сам себе для вхідних значень *n - 1* і *n - 2*, підсумовує їх, а потім повертає результат.

Наведений алгоритм легкий для розуміння, але працює досить повільно. Так, для розрахунку *Fibonacci (6)* програма повинна визначити *Fibonacci (5)* і *Fibonacci (4)*, але перед розрахунком *Fibonacci (5)* їй знадобиться встановити *Fibonacci (4)* і *Fibonacci (3)*, тобто *Fibonacci (4)* визначається двічі. При великих *N* рекурсія *Fibonacci (N)* стає досить глибокою, а одні і ті ж значення доводиться знаходити безліч разів, змушуючи програму працювати протягом довгого часу. Такий ефект виникає тому, що рекурсивний виклик в алгоритмі є *множинним*.

На рисунку нижче показано дерево викликів алгоритму Фібоначчі для визначення *Fibonacci (6)*. Кожна вершина - це і є виклик, а число що стоїть в ній - вхідний параметр. У кореневому вузлі алгоритм звертається до *Fibonacci (5)* і *Fibonacci (4)*. Зверніть увагу, що внизу дерево заповнене дубльованими викликами: наприклад, *Fibonacci (0)* розраховується п'ять разів, а *Fibonacci (1)* - вісім.

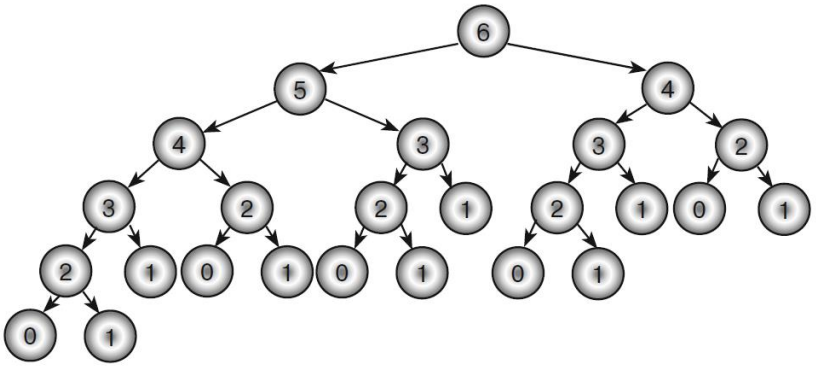


Рисунок 2.1 Дерево викликів в алгоритмі Фібоначчі для n=6

Оцінити час роботи цього алгоритму складніше, ніж факторіального, оскільки він є множинно-рекурсивним. Припустимо, що *T(N)* - продуктивність алгоритму Фібоначчі з вхідним значенням *N*. Якщо *N> 1*, то код розраховує *Fibonacci(N - 1)* і *Fibonacci(N - 2)*, потім на додатковому кроці підсумовує їх і повертає результат. Таким чином, *T(N) = T(N-1) + T(N-2) + 1*. Якщо проігнорувати константу 1 в кінці формули, то отримаємо визначення функції *Fibonacci*, тобто саме від неї залежить підсумковий час роботи алгоритму. Асимптотична оцінка складності роботи рекурсивного алгоритму Фібоначчі, як можна бачити з рисунка, є експоненціальною. На практиці час роботи такого алгоритму Фібоначчі зростає дуже швидко. Зазвичай при значеннях *N* понад 44 він вже стає непрактичним.

2.3 Рекурсивні структури даних

По аналогії з рекурсивним викликом функції, існують структури даних, що теж допускають рекурсивне визначення: елемент структури даних містить один або кілька покажчиків на елементи такого ж типу. Формально це відповідає тому факту, що у визначенні структурованого типу міститься покажчик на себе самого. Приклад рекурсивного визначення структур даних список і дерево наведено нижче:

<список> ::= <елемент> | <елемент> <елемент> <список>
<елемент> ::= <значення> | NULL

<дерево> ::= <корінь> | <ліве піддерево> | <праве піддерево> | NULL
<ліве піддерево> ::= <дерево>
<праве піддерево> ::= <дерево>
<корінь> ::= <значення>

Рисунок 2.2 Приклад двох рекурсивних структур даних: списка і дерева

2.4 Особливості програмної реалізації рекурсивних алгоритмів

Процес програмної реалізації рекурсивних алгоритмів має, в порівнянні з ітераційними алгоритмами, специфічні особливості, а саме: необхідність організації спеціальної структури даних і обслуговування рекурсивних викликів. При цьому організація «правильної» структури даних лежить на плечах програміста, в той час як механізм обслуговування рекурсивних викликів забезпечується самою мовою програмування, точніше, компілятором цієї мови, і підтримується машинними командами комп'ютера.

Особливості розробки структур даних. Оскільки рекурсивний алгоритм задає в часі послідовність звернень до одного й того ж фрагменту програмного коду, то застосована структура даних повинна забезпечити збереження тих комірок (змінних) пам'яті і масивів, які будуть задіяні алгоритмом після повернення з рекурсивного виклику. В рамках можливостей сучасних мов процедурного програмування найчастіше використовується локальна структура даних. При цьому способі організації даних в тілі рекурсивної функції або процедури описуються усі необхідні для роботи функції структури даних, а створення копій цих структур при рекурсивному виклику покладається на засоби мови програмування і компілятора. Такий підхід призводить, особливо при наявності масивів, до великих витрат пам'яті в області програмного стека, розмір якого, взагалі кажучи, обмежений. Цей спосіб призводить до того, що загальна структура даних і її розмір, відповідні поточній глибині рекурсії (послідовності рекурсивних викликів), буде створюватися механізмом обслуговування рекурсивного виклику безпосередньо під час виконання програмного коду.

Модель програмного стека. Огляд механізму організації рекурсивних викликів ми почнемо з моделі програмного стека. Будемо припускати, що операційна система підтримує спеціальну область оперативної пам'яті - програмний стек, використовуючи спеціальний регістр або елемент пам'яті - покажчик стека, який зберігає адресу деякої комірки (слова) в цій області. Дві спеціальні команди - «записати в стек» і «читати із стека» організовують роботу з цією областю так, що ми вважаємо, що маємо справу зі стеком як класичною структурою, хоча зазначені операції і змінюють адресу покажчика стека. Наприклад, команда «записати в стек» зменшує поточну адресу і записує за цією

адресою вміст свого операнда. Такий підхід гарантує, що будь-яка операція зі стеком не призводить до перезапису попереднього його вмісту, і дозволяє вважати, що трудомісткість виконання зазначених команд не залежить від обсягу інформації, що зберігається. Хоча наші команди неявно використовують поточний покажчик вершини програмного стека, тим не менш, ми розглядаємо їх як команди з одним операндом, що вказує елементарну комірку пам'яті або регістр, вміст яких або поміщається в стек, або зчитується в них з вершини стека. Наведемо покроковий опис цих команд.

1. «Записати в стек» <операнд>. Команда зменшує адресу покажчика стека на довжину операнда і поміщає вміст операнда, починаючи з отриманої адреси.

2. «Читати із стека» <операнд>. Команда зчитує інформацію, розташовану за покажчиком стека, в операнд і збільшує покажчик на довжину операнда.

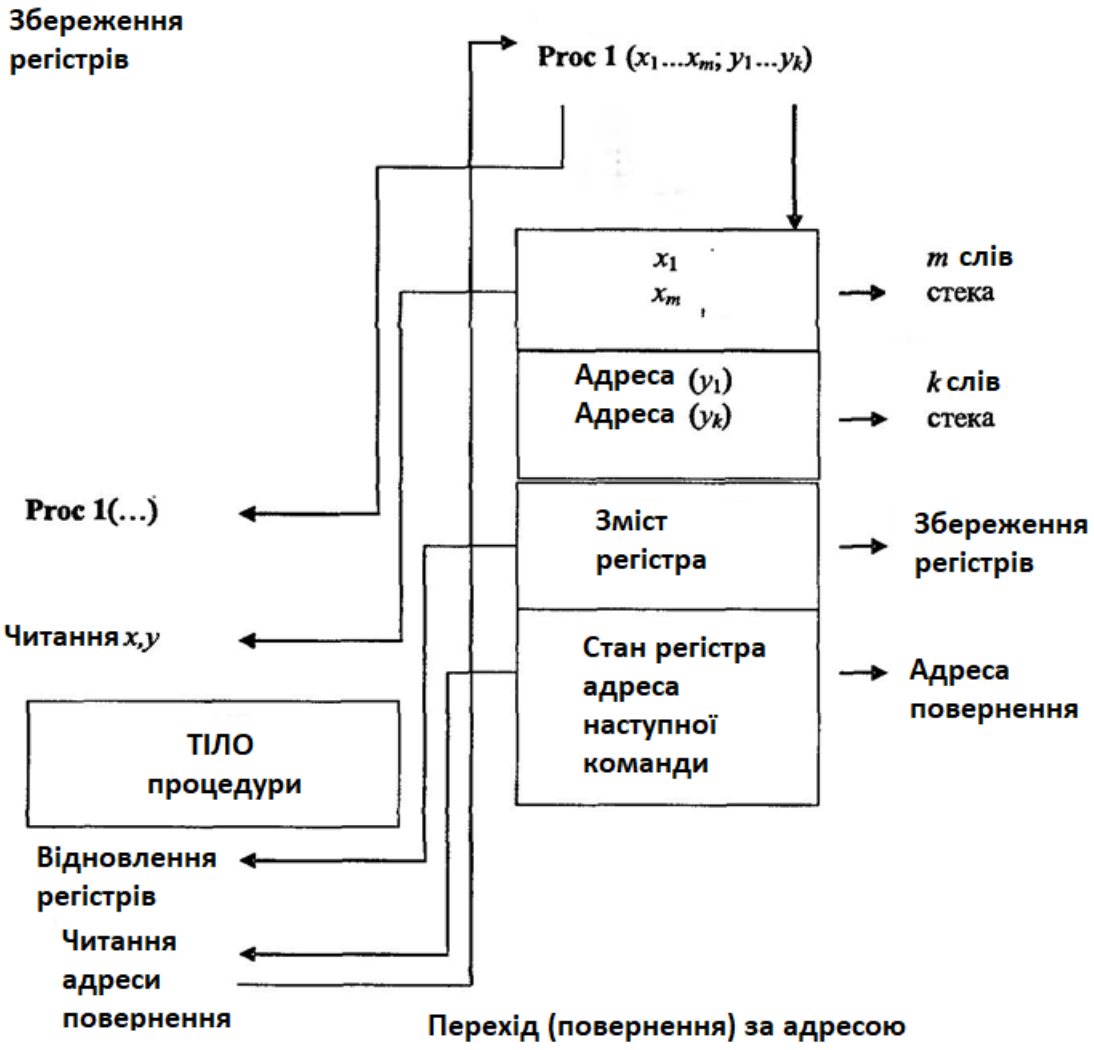


Рисунок 2.3 Механізм виклику процедури з використанням стека

Механізм виклику процедури з використанням стека (рисунок вище). Перш ніж переходити до вивчення механізму рекурсивного виклику, розглянемо

звичайний механізм в мовах процедурного програмування, що забезпечує передачу управління на деяку процедуру і повернення (по її завершенню) в точку виклику. Нагадаємо, що під процедурою розуміється поіменований фрагмент програми, до якого можна звертатися по його імені з можливістю передачі ряду аргументів (фактичних параметрів) як за значенням, так і за посиланням (адресою) для повернення отриманих результатів в точку виклику. На відміну від функції, процедура не повертає значення у своєму імені. Механізм має забезпечити збереження регістрів процесора, оскільки при поверненні в точку виклику ми повинні відновити їх значення; крім того, процедурі повинні бути певним чином передані фактичні параметри.

Для забезпечення цих дій ми і будемо використовувати описаний вище програмний стек і обслуговуючі його команди. Оскільки значення повинні поміщатися в програмний стек в порядку, зворотному їх вибірці, то спочатку ми розміщуємо в стек адресу повернення в точку виклику, потім значення необхідних регістрів і вже далі передаємо у функцію фактичні параметри. Схематично цей механізм проілюстрований на Рисунок 2.3 - ми вважаємо, що програма, яка викликає, сама зберігає регістри, а процедура, що викликається, сама відновлює їх перед передачею керування (поверненням) головній програмі.

При виклику процедури програма, що викликає, розміщує в стеку адресу повернення, стан необхідних регістрів процесора, адреси повернених значень і передані параметри, користуючись командою «записати в стек». Після цього виконується перехід за адресою на процедуру, що викликається, яка і "витягує" із стеку передані фактичні параметри (командою «читати з стека»), виконує необхідні обчислення і розміщує результати за вказаними в стеку адресами. При завершенні роботи викликана процедура відновлює значення регістрів, виштовхує із стека адресу повернення і здійснює перехід за цією адресою, повертаючи управління в точку свого виклику (див. Рисунок 2.3)

Відзначимо, що в даний час практично всі мови програмування для персональних комп'ютерів, більш коректно – компілятори, які їх обслуговують, підтримують механізм рекурсивного виклику, заснований на програмному стеку. У світлі вищесказаного цей механізм було б більш правильно назвати механізмом рекурсивного виклику-повернення. Викладений тут механізм є нехай наближеною

до дійсності, але все ж моделлю механізму організації рекурсивного виклику. Компілятори реальних мов програмування високого рівня можуть мати власні особливості організації такого механізму.

2.5 Видалення рекурсії

Рекурсія сприяє кращому розумінню деяких проблем: наприклад для роботи з бінарним деревом і обчисленням факторіала вона дає прості і витончені рішення. На жаль, є у рекурсії і свої недоліки - іноді її використання очевидно, але неефективно (як в задачах обчислення чисел Фібоначчі і Ханойської вежі).

Так, рекурсивний алгоритм генерування чисел Фібоначчі вимагає від програми багаторазового розрахунку одних і тих же величин. Це уповільнює роботу настільки, що розрахунок понад 44 значень стає непрактичним. Інші алгоритми використовують дуже глибокі рівні рекурсії і призводять до того, що програма вичерпує свій стек викликів.

Щоб запобігти подібним проблемам, можна зробити декілька кроків в плані реструктуризації рекурсивного алгоритму таким чином, щоб перетворити його в ітераційний. Це може допомогти поліпшити його продуктивність.

2.5.1 Видалення хвостової рекурсії

Видалення хвостової рекурсії відбувається в тому випадку, коли в останній операції перед поверненням алгоритм викликає сам себе. Як приклад розглянемо застосування факторіального алгоритму Приклад 2.2.

Насамперед алгоритм перевіряє, чи треба йому викликати самого себе чи можна повернути значення 1. Якщо рекурсивний виклик потрібен, програма помножує поточний результат на *n* і повертає його.

Рекурсивну версію алгоритму можна перетворити в нерекурсивну за допомогою циклу, всередині якого будуть виконуватися будь-які необхідні завдання початкового коду. Перед закінченням циклу алгоритму потрібно встановити для параметрів значення, використовувані під час рекурсивного виклику. У разі факторіального алгоритму це значення потрібно відстежити за допомогою змінної. Коли цикл повторюється, параметри встановлюються для рекурсивного виклику - і алгоритм виконує те, що робив рекурсивний виклик.

Кінець циклу пов'язаний з умовою, яка закінчувала рекурсію. Для

факторіального алгоритму це *n = 0*. Коли алгоритм рекурсивно викликає сам себе, він зменшує параметр *n* на *1*. Це також повинна робити і нерекурсивна версія до закінчення циклу.

У наступному прикладі представлена нерекурсивна версія факторіального алгоритму:

Приклад 2.4 Нерекурсивна версія факторіального алгоритма

```
Factorial (n)

    // Створюємо змінну для відстеження значення, що повертається.
    // Прирівнюємо її значення до 1, щоб можна було помножити на неї
    // результат.
    // (Результат дорівнює 1, якщо ми не входимо в цикл.).
    Integer: result = 1

    // Запускаємо цикл, що керується умовою зупинки рекурсії.
    While (n ≠ 0)

        // Зберігаємо результат для даного «рекурсивного» виклику.
        result = result * n
        // Готуємося до "рекурсії"
        n = n – 1
    Loop

    // Повертаємо накопичений результат.
    Return result

End Factorial
```

Алгоритм здається довшим тільки через коментарі.

Видалити хвостову рекурсію досить просто, тому деякі компілятори діють подібним чином автоматично, щоб скоротити вимоги до стекового простору.

Звичайно, головна проблема факторіального алгоритму пов'язана не з глибиною рекурсії, а з тим, що результати стають занадто великими і їх важко зберігати в типах даних з фіксованим розміром. Для деяких алгоритмів видалення хвостової рекурсії є корисним і зазвичай покращує продуктивність, оскільки перевірка умови циклу *While* здійснюється швидше, ніж виклик рекурсивного методу.

2.5.2 Зберігання проміжних значень

В алгоритмі Фібоначчі хвостова рекурсія не використовується. Недолік цієї програми полягає в тому, що вона повторно розраховує занадто багато проміжних значень, а це забирає час. Справитися з проблемою можна, якщо запам'ятовувати

значення в міру їх обчислення.

Приклад 2.5 Розрахунок чисел Фібоначчі алгоритмом поліноміальної складності

```
Integer: FibonacciValues[100]  (глобальний масив)
    // Найбільше обчислене значення на даний момент.
    Integer: MaxN       (глобальна змінна)
    // встановлюємо значення Fibonacci[0] и Fibonacci[1].
    InitializeFibonacci()
        FibonacciValues[0] = 0
        FibonacciValues[1] = 1
        MaxN = 1
    End InitializeFibonacci
    // Повертаємо n-е число Фібоначчі.
    Fibonacci(n)
    // Якщо ми ще не обчислювали дане значення, знаходимо його.
    If (MaxN < n) Then
        FibonacciValues[n] = Fibonacci(n - 1) + Fibonacci(n - 2)
        MaxN = n
    End If

    // Повертаємо отримане значення.
    Return FibonacciValues[n]
End Fibonacci
```

На самому початку алгоритм оголошує глобально видимий масив *FibonacciValues* для зберігання розрахованих попередніх величин чисел Фібоначчі. Змінна *MaxN* відстежує найбільше значення *n*, для якого *Fibonacci(n)* вже знаходиться в масиві.

Потім спрацьовує алгоритм *InitializeFibonacci*. Він встановлює перші два числа Фібоначчі перед тим, як програма звернеться до функції *Fibonacci*. А вже ця функція порівнює *MaxN* зі своїм вхідним параметром *n*. Якщо *n*-е число Фібоначчі ще не визначено, алгоритм рекурсивно викликає сам себе, щоб його розрахувати, потім зберігає отримане значення в масиві *FibonacciValues* і оновлює *MaxN*.

В кінці програма повертає значення, яке зберігається в масиві *FibonacciValues*. До цього моменту вона вже знає, що воно було там раніше або його помістили туди попередні виконання коду.

У даній версії алгоритму кожне число Фібоначчі розраховується **один раз**, а потім просто витягується з масиву. В результаті немає необхідності визначати проміжні величини величезну кількість разів. Якщо вихідний алгоритм Приклад 2.3 знаходив *Fibonacci (44)* приблизно за 1 хв (і це була межа його можливостей), то вдосконалена версія здатна взятися за великі значення, наприклад *Fibonacci (92)* і виконується майже миттєво. Однак розрахувати *Fibonacci (93)* навіть у неї не вийде, оскільки результат перевищує ціле число довжиною 64 біта (навіть тип *long long* в мові Сі вже не підтримує такі величезні значення) і слід переходити на типи з плаваючою комою.

Збереження проміжних величин підвищує продуктивність програми, але не видаляє зовсім рекурсію.

2.6 Контрольні завдання і запитання

1. Напишіть програму, що реалізує вихідний факторіальний рекурсивний алгоритм (Приклад 2.2) мовою С.

2. Напишіть програму, що реалізує вихідний рекурсивний алгоритм обчислення чисел Фібоначчі (Приклад 2.3) мовою С.

3. Напишіть рекурсивну програму мовою С для розв'язання задачі про вісім ферзів (розставити на шахівниці вісім ферзів так, щоб вони не били один одного).

4. Удоскональте програму з попередньої вправи: відстежте, скільки ферзей може атакувати певну позицію на дошці. Додаючи чергову фігуру, ігноруйте позиції, для яких значення не дорівнює 0. Як виконані модифікації вплинуть на розміщення одного ферзя та на загальний час роботи алгоритму?

5. Зверніть увагу, що в задачі про вісім ферзів у кожному ряду на шахівниці повинна стояти лише одна фігура. Удосконаліть програму з вправи 3 так, щоб кожен виклик методу EightQueens шукав позицію для нового ферзя лише у наступному ряду. Як це вплине на кількість можливих положень фігури та загальний час роботи алгоритму?

6. Напишіть програму, що реалізує вихідний факторіальний нерекурсивний алгоритм (Приклад 2.4) мовою С.

7. Напишіть програму, що реалізує вихідний алгоритм обчислення чисел Фібоначчі із значеннями, що зберігаються (Приклад 2.5), мовою С.

8. Напишіть програму, що реалізує вихідний нерекурсивний алгоритм

обчислення чисел Фібоначчі мовою С.

9. Нерекурсивний алгоритм Фібоначчі розраховує числа до необхідної межі, а потім знаходить потрібне значення в масиві. Насправді алгоритм може розраховувати менші значення Фібоначчі лише тоді, коли вони потрібні. Це займе трохи більше часу, але вам не знадобиться розміщувати повноцінний масив. Напишіть програму мовою С, що реалізує нерекурсивний алгоритм Фібоначчі з даним методом.

3 МАСИВИ І РОБОТА З НИМИ

3.1 Основні визначення

Масив — це велика кількість неперервної області пам'яті, доступ до якої здійснюється за допомогою індексів (їх кількість вказує на розмірність масиву) [1]. Його легко уявити у вигляді сукупності комірок, в яких програма зберігає значення (рисунок нижче).

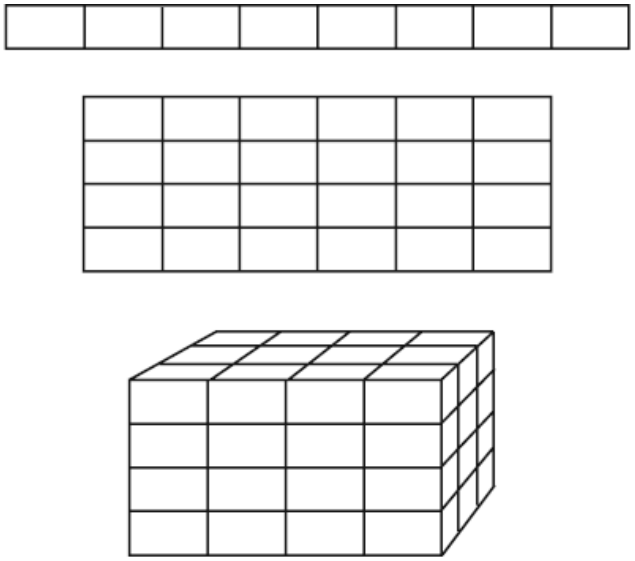


Рисунок 3.1 Приклади одно-, дво- і тривимірного масивів

Масиви є найбільш широко використовуваними структурами даних і передбачені у всіх високорівневих мовах програмування. Масив складається з елементів одного типу, званого **базовим**, тому структура масиву однорідна. Базовий тип, у свою чергу, може бути як скалярним, так і структурованим або абстрактним, тобто елементами масиву можуть бути числа, символи, рядки, структури, списки, дерева, у тому числі і інші масиви. Число елементів масиву фіксовано, тому обсяг займаної масивом пам'яті залишається незмінним. З кожним елементом масиву пов'язаний один (для одновимірного масиву) або декілька індексів (для багатовимірного масиву). Вони однозначно визначають місце елемента в масиві і забезпечують прямий та швидкий доступ до цього елемента. Індекси масиву відносяться до певного порядкового типу, тому індекси можна обчислювати. Це забезпечує, з одного боку, гнучкість обробки елементів масиву, з іншого боку, створює небезпеку виходу за межі масиву, якщо не передбачені відповідні засоби контролю (що і відбувається у мовах Сі та С++).

В залежності від числа індексів розрізняють **одновимірні** і **багатовимірні**

масиви. Допустима кількість індексів масиву (розмірність масиву) і діапазони зміни їх значень встановлюються мовою програмування, транслятори з мови програмування можуть уточнювати ці значення. За стандартом мови Сі розмірність масиву не може перевищувати *31* виміру, а нумерація індексів починається завжди з нуля, тобто індекс змінюється від *0* до *n-1*, де *n* - кількість значень індексу (розмір індексу).

Областями застосування масивів є:

- числові масиви в обчислювальних задачах;
- матрична алгебра, екстраполяція, інтерполяція;
- таблиці - масиви з елементами типу «запис»;
- керуючі та інформаційні таблиці в операційних системах, трансляторах, системах управління базами даних (СУБД);
- представлення інших структур: графів, дерев, тощо.

Ми розглянемо технології алгоритмізації, які можна задіяти для прискореної роботи з масивами, а також для створення масивів з ненульовими нижніми межами і економії обсягу пам'яті.

При створенні масиву програма, як правило, задає його розмірність і межі цієї розмірності. Наприклад, у наступному коді на мові програмування Сі оголошується масив *numbers*, що має 10 рядків і 20 стовпців.

int numbers [10] [20];

У Сі межі масиву відраховуються з нуля, тому індекси рядків в прикладі можуть набувати значень від 0 до 9, а індекси стовпців - від 0 до 19.

Крім усього іншого, компілятор при створенні масиву довільної розмірності, неявно виділяє достатню кількість **неперервної одномірної пам'яті** для зберігання даних масиву. Така пам'ять виглядає як довгий ряд байтів, з кожним з яких пов'язуються вказані індекси масиву. В одновимірному масиві призначення індексу відбувається просто: індекс *i* відповідає комірці *i*. У двовимірному аналогічна операція здійснюється одним із двох способів: шляхом розгортання масиву по рядках або по стовпцях.

При розгортанні по рядках програма прив'язує перший рядок елементів масиву до першого набору комірок пам'яті, другий рядок - до наступного набору комірок, тощо. По одному рядку за раз, поки всі елементи масиву не будуть

приведені у відповідність з комірками пам'яті. Іншими словами, при такому розміщенні, швидше за все змінюються останні індекси масиву (приклад - масив *B* розміру *mxn*: $b_{0,0}$, $b_{0,1}$, $b_{0,2}$, ..., $b_{0,n-1}$, $b_{1,0}$, $b_{1,1}$, $b_{1,2}$, ..., $b_{1,n-1}$, ..., $b_{m-1,0}$, $b_{m-1,1}$, $b_{m-1,2}$, ... , $b_{m-1,n-1}$). Такий тип розміщення елементів масиву в пам'яті реалізовано в мовах програмування C/C++ і Паскаль.

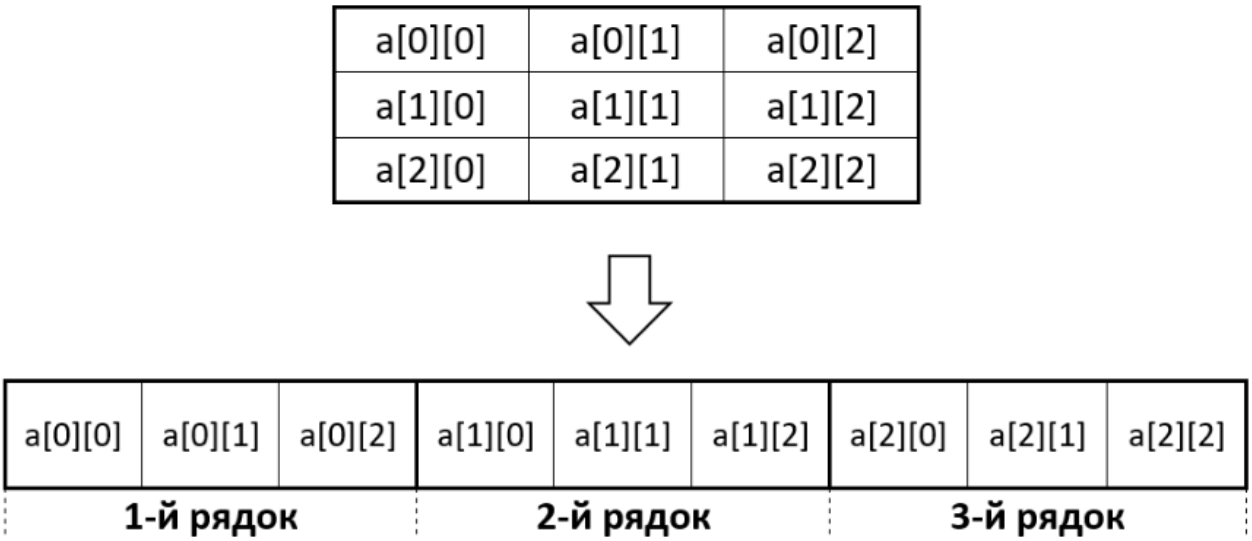


Рисунок 3.2 Розміщення елементів двовимірного масиву у комірках пам'яті по рядках

При розгортанні за стовпцями з першим набором комірок пам'яті буде асоціюватися перший стовпець елементів масиву, з другим – другий, тощо. Іншими словами, при такому розміщенні, швидше за все змінюються перші індекси масиву (приклад - масив *B* розміру *mxn*): $b_{0,0}$, $b_{1,0}$, $b_{2,0}$, ..., $b_{m-1,0}$, $b_{0,1}$, $b_{1,1}$, $b_{2,1}$, ..., $b_{m-1,1}$, ..., $b_{0,n-1}$, $b_{1,n-1}$, $b_{2,n-1}$, ... , $b_{m-1,n-1}$). Такий тип розміщення елементів масиву в пам'яті реалізовано в мові програмування Фортран.

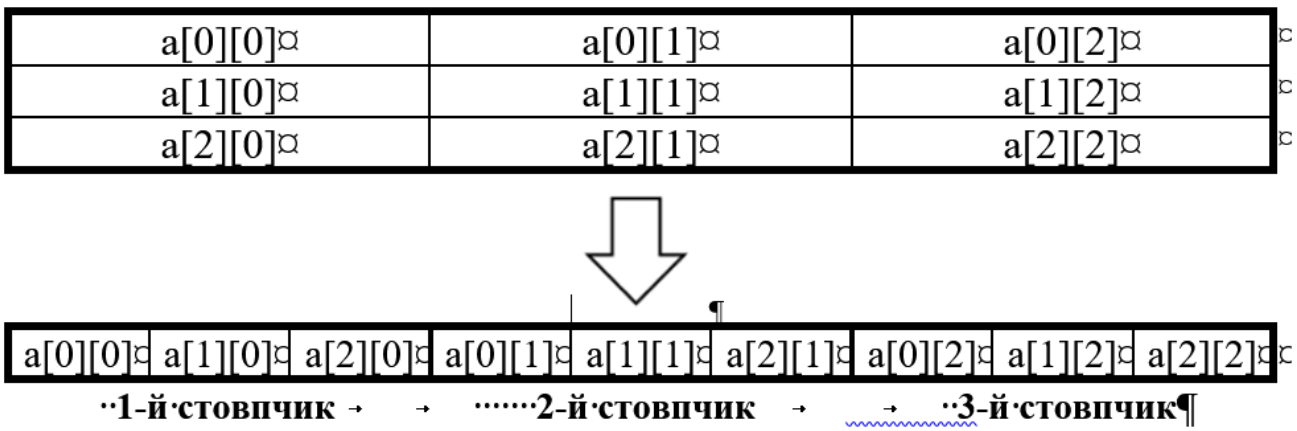


Рисунок 3.3 Розміщення елементів двовимірного масиву у комірках пам'яті по стовпцях

Цей принцип розгортання елементів масиву в лінійну пам'ять можна перенести і на масиви з більшою розмірністю. Наприклад, для зберігання тривимірного масиву по рядках програмі знадобиться зіставити перший двовимірний шар масиву з нульовим третім індексом. Розміщення відбувається при цьому пошарово, по рядках, тобто другий шар буде відповідати третьому індексу під номером 1, тощо. Зазвичай, робота програми не залежить від розміщення записів масиву до комірок пам'яті, тому переживати з цього приводу не варто. Досить того, що код працює з елементами масиву, а як вони зберігаються в пам'яті програмісту знати і необов'язково. Проте, при програмуванні на низькорівневих мовах типу Асемблер (де відсутній вбудований тип масиву), а також для розуміння процесу створення власних структур даних з подальшим застосуванням в масивах спеціального призначення (наприклад, трикутних) корисно добре розуміти механізм розміщення багатовимірного масиву в пам'яті.

Вся інформація, необхідна для управління масивом, задається при його оголошенні в програмі. Оголошення містить ім'я масиву, тип елементів, який однозначно визначає довжину елемента, діапазони зміни індексів або число значень індексів, якщо нижні межі індексів фіксовані. Таким чином, загальна кількість елементів масиву і розмір пам'яті для масиву повністю визначаються при оголошенні масиву. Транслятор виділяє необхідну кількість пам'яті і будує керуючий блок масива - дескриптор, або інформаційний вектор, наприклад, такого змісту (для одновимірного масиву):

- адреса початку масиву $A_н$**;**
- тип елемента (довжина елемента масиву L);
- нижня границя індекса $i_н$**;**
- верхня границя індекса $i_в$**.**

У разі багатовимірного масиву для кожного індексу задається значення нижньої і верхньої межі або ж для кожного рядка (кожного індексу) створюється свій дескриптор.

Цієї інформації достатньо як для доступу до елементів масиву, так і для контролю над тим, щоб значення індексів не виходили за встановлені діапазони (в тих мовах програмування, де це передбачено, наприклад в Паскалі). Доступ до будь-якого i-го елемента в одновимірному масиві здійснюється за адресою:

$$A_i = A_0 + i * L, \quad (3.1)$$

де A_0 - початок масиву, тобто адреса нульового елемента;

i - індекс елемента;

L - розмір елемента в байтах.

Нехай в нас є n-вимірний масив A (у якого число елементів по i-му виміру дорівнює m_i, а **індексація по всіх вимірах починається з нуля)**, представлений у пам'яті у вигляді лінійного масиву B. Тоді елементу $a_{i_1,i_2,\ldots,i_n}$ вихідного масиву A буде відповідати елемент масиву $b_к$ з **індексом**, рівним:

$$к = ((\ldots(i_1*m_2 + i_2)*m_3 + i_3)*m_4 + \ldots + i_{n-1})*m_n + i_n\ , \quad (3.2)$$

а адреса в пам'яті такого елемента $b_к$ лінійного масиву B буде, відповідно:

$$B_к = B_0 + k * L \quad (3.3)$$

де B_0 - початок масиву, тобто адреса нульового елемента;

k - індекс елемента з формули (3.2);

L - розмір елемента в байтах (для всіх елементів обох масивів однаковий!)

3.2 Одновимірні масиви

3.2.1 Пошук елемента

Якщо елементи масиву не відсортовані, то використовується лінійний, або повний пошук. При цьому розглядається кожен елемент масиву, поки не знаходиться шуканий елемент, або не стає очевидною його відсутність.

Приклад 3.1 Алгоритм лінійного пошуку в масиві

```
int indexOf (int array[], int target, int length) {
        for (int i = 0; i < length; i++)
                if (array[i] == target)  return i;

        // Якщо цільового елемента в масиві немає повертаємо відповідну
        // ознаку
        return -1;
}
```

У найгіршому разі цільовий елемент може бути найостаннішим в масиві. Якщо всього елементів N, то алгоритм закінчить пошук, пройшовшись по кожному з них. Таким чином, час його роботи складе $O(N)$. Те ж саме станеться, якщо цільового елемента в масиві немає. Щоб перевірити кожен елемент в масиві по черзі, в середньому знадобиться $N/2$ кроків, що також дорівнює $O(N)$.

3.2.2 Знаходження мінімальної, максимальної і середньої величин

У числовому масиві нерідко потрібно знайти мінімальну, максимальну або середню величину.

Приклад 3.2 Пошук мінімуму в лінійному масиві

```
int FindMinimum (int array[], int length) {
        int minimum;
        minimum = array[0];
        for (int i = 1; length – 1; i++)
                if (array[i] < minimum)  minimum = array[i];
        return minimum;
}
```

Приклад 3.3 Пошук максимуму в лінійному масиві

```
int FindMaximum (int array[], int length) {
        int maximum;
        maximum = array[0];
        for (int i = 1; length – 1; i++)
                if (array[i] > maximum)  maximum = array[i];
        return maximum;
}
```

Приклад 3.4 Пошук середнього в лінійному масиві

```
double FindAverage (int array[], int length) {
        int total=0;
        for (int i = 0; Length – 1; i++)
                total = total + array[i];
        return (double) total / length;
}
```

Як і у випадку з пошуком певного значення в масиві, всі вищенаведені три алгоритма повинні переглянути кожен елемент масиву, тому час їх роботи складе $O(N)$.

3.3 Динамічні масиви високої розмірності

Для того, щоб зберігати значення в динамічному багатовимірному масиві, запакуємо їх в одновимірний по рядках. Такий масив при цьому повинен бути досить великим, щоб вмістити всі елементи. Так, якщо є багатовимірний масив

розмірності N_1 x N_2 x…x N_m, то краще задати лінійний масив з такою ж кількістю елементів:

*values[N_1*N_2*…*N_m]*

Коли відомі індекси елемента $a_{i_1,i_2,\ldots,i_m}$ багатовимірного масиву, необхідно знайти його індекс в одновимірному масиві *values*. Для цього слід використовувати формулу (3.2). Покажемо на прикладі тривимірного масиву, як це робиться.

Нехай у нас є тривимірний масив розмірністю *4x4x3* (висота * рядок * стовпець). Нам потрібен елемент з індексами *(1,1,2)*. Щоб співвіднести елемент *(1,1,2)* тривимірного масиву з індексом *k* лінійного масиву *values*, підставимо в формулу (3.2) відповідні значення індексів і розмірностей:

*к = ((1*4 + 1)*3 + 2 = 17,*

Значення 17 і буде індексом елемента *(1,1,2)* в лінійному масиві *values*.

Тепер розглянемо приклад двох реалізацій одного і того ж алгоритму на мові Сі для створення і обробки двовимірного динамічного масиву. На цих прикладах можна буде побачити, як працюють з багатовимірними динамічними масивами як за допомогою виключно арифметики покажчиків, так і за допомогою індексів.

Приклад 3.5 Робота з елементами двовимірного динамічного масиву тільки з використанням адресної арифметики

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main(void) {
    int     j,  /* лічильник рядків*/
    i,  /* лічильник стовпців */
    m, /* поточне число рядків в масиві */
    n,  /* поточне число стовпців в масиві */
    *pi = NULL; /* покажчик на сам двовимірний динамічний масив */

    setlocale (LC_ALL, "RUS");
    printf ("Введіть число рядків і стовпців:");
    scanf ("%d%d", &m, &n);

    /* Виділення пам'яті під масив з одночасним контролем наявності
    вільної пам'яті */
    if ( !(pi = (int*) malloc (m*n*sizeof(int))) ) {
        printf("Для такого масива не вистачає пам'яті!");
```

```
            getch();        /* затримка! */
            exit(1);
        }

        /* Заповнення масиву числами */

        for (j=0; j<m; ++j)
            for (i=0; i<n; ++i)
                *((int*) pi+j*n+i) = (j*n)+i+1;  /* +1, так як індексація з
                                                    нуля! */

        /* Друк  масиву */
         for (j=0; j<m; ++j) {
            for (i=0; i<n; ++i) /* друк поточного j-го рядка масиву */
                printf ("%3d", *((int*) pi+j*n+i));
            printf ("\n"); /* перехід до друку нового рядка */
         }

        getch();        /* затримка!*/

        free (pi);

        exit(0);

}
```

Приклад 3.6 Реалізація двовимірного динамічного масиву за допомогою покажчиків і робота з ним за допомогою індексів

```
int main(void) {

        int     j,  /* лічильник рядків */
        i,  /* лічильник стовпців */
        m, /* поточне число рядків в масиві */
        n,  /* поточне число стовпців в масиві */
        **pi = NULL; /* покажчик на сам двовимірний динамічний масив */

        setlocale (LC_ALL, "RUS");
        printf ("Введіть число рядків і стовпців:");
        scanf ("%d %d", &m, &n);

        /* Виділення пам'яті під масив покажчиків на рядки */
        pi = (int**) malloc (m*sizeof(int*));
        if ( !pi ) {
            printf("Для такого масиву не вистачає пам'яті!");
            getch();        /* затримка! */
            exit(1);
        }

        /* Виділення пам'яті під рядки масиву */
        for (j = 0; j < m; j++) {
            pi[j] = (int*) malloc(n * sizeof(int));
            if ( !pi[j] ) {
                printf("Для такого масиву не вистачає пам'яті!");
```

```
                getch();   /* затримка! */
                exit(1);
            }
        }

        /* Заповнення масиву числами */
        for (j=0; j<m; ++j)
            for (i=0; i<n; ++i)
                pi[j][i] = (j*n)+i+1;       /* +1, так як індексація з нуля! */

        /* Друк массиву */
        for (j=0; j<m; ++j) {
            for (i=0; i<n; ++i)       /* друк поточного j-го рядка масиву */
                printf ("%3d", pi[j][i]);
            printf ("\n");     /* перехід до друку нового рядка */
        }

        getch();       /* затримка! */
        for (j = 0; j < m; j++)
            free (pi[j]);  /* вивільнення пам'яті під рядки */
        free (pi);   /* вивільнення пам'яті під масив покажчиків на рядки */
        exit(0);

}
```

На закінчення, слід зазначити, що робота програми з Приклад 3.5 здійснюється швидше, ніж програми з Приклад 3.6. Це відбувається через те, що:

- в першому прикладі лише одне звернення до "купи" при виділенні пам'яті, а в другому число звернень збігається з числом рядків динамічного масиву плюс один;
- операції індексації масиву працюють повільніше, ніж пряма адресна арифметика покажчиків.

3.4 Вільні масиви

Вільними називають двовимірні масиви, розмір кожного рядка в яких може бути різним. Оскільки стандарти мови програмування не допускають такого виду масивів, то створюється симбіоз масивів: одновимірний масив покажчиків, число елементів *n* якого дорівнює числу рядків змінної довжини, і *n* одновимірних масивів різної довжини. Таким чином, масив покажчиків має фіксовану довжину, значить, фіксоване і число рядків вільного масиву. А вже пам'ять під кожен рядок вільного масиву виділяється динамічно.

Отже, при створенні такого масиву потрібно *(n + 1)* звернень до "купи" для отримання динамічної пам'яті. Після того як потреба у вільному масиві відпала,

необхідно звільнити займану ним пам'ять, що знову-таки зажадає *(n + 1)* звернень до "купи".

На рисунку нижче зображена структура зберігання вільного масиву з *n* рядками. Масив покажчиків *U* має *n* елементів, кожен з яких містить адреси векторів - рядків масиву.

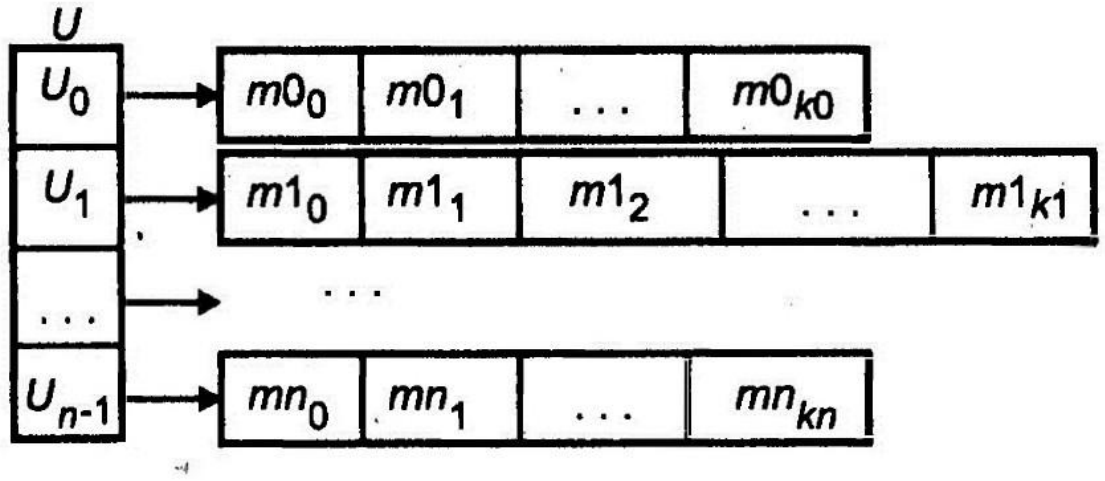


Рисунок 3.4 Структура зберігання вільного масиву

Фактично, вільний масив являє собою лінійний масив покажчиків на лінійні динамічні масиви даних.

Для управління вільним масивом може створюватися дескриптор, в найпростішому випадку - це покажчик на масив покажчиків. Крім того, створюється допоміжний лінійний масив, який буде зберігати відповідні довжини рядків. За аналогічною схемою можуть бути створені і двовимірні масиви з однаковими довжинами рядків. Вільні масиви часто використовуються для представлення інших, більш складних структур.

Приклад 3.7 Приклад реалізації вільного масиву на мові С

```
#define _CRT_SECURE_NO_WARNINGS
#include "stdlib.h"
#include "stdio.h"
#include "conio.h"
#include "math.h"
#include "locale.h"
#include "string.h"
#include "windows.h"

int main() {
    int **a; /* масив покажчиків на рядки */
```

```
    int *m; /* масив кількості елементів в рядках */

    int     i,    /* індекс рядка */
            j,    /* індекс стовпчика */
            n;  /* кількість рядків масиву */


    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    printf("Введіть кількість рядків: ");
    scanf("%d", &n);

    a = (int**)malloc(n * sizeof(int*));     // масив покажчиків на рядки
    m = (int*)malloc(n * sizeof(int));       // масив кількості елементів в
                                             // рядках

    // Введення елементів масиву з одночасним створенням рядків
    for (i = 0; i<n; i++) {
            printf("Введіть кількість елементів рядка %d: ", i);
            scanf("%d", &m[i]);

            // створення поточного рядка лінійного масиву
            a[i] = (int*)malloc(m[i] * sizeof(int));

            for (j = 0; j<m[i]; j++) {
                    printf("a[%d][%d]= ", i, j);
                    scanf("%d", &a[i][j]);
            }
            printf("\n\n");
    }

    // Виведення елементів масиву
    for (i = 0; i<n; i++) {
            for (j = 0; j<m[i]; j++)
                    printf("%3d ",a[i][j]);
            printf("\n");
    }

    for (j = 0; j < m[i]; j++)
            free(a[j]);    // звільнення пам'яті під рядки масиву


    /* звільнення пам'яті під масив покажчиків на рядки і масив кількості
       елементів  в рядках */
    free(a);
    free(m);
}
```

Результат виконання цього прикладу показано на Рисунок 3.5.

Таке представлення масивів має свої переваги і недоліки. До недоліків можна

віднести потребу в додатковій пам'яті під масив покажчиків і розмірів рядків та багаторазове звернення до "купи" для отримання пам'яті під рядки. До переваг можна віднести наступне:

- при роботі з дуже великими масивами отримання великих суцільних областей може стати неможливим; в той же час отримання пам'яті під кожен рядок змінного розміру цілком можливе;
- з'являється можливість обробляти великі масиви даних, що зберігаються в файлах, розміщуючи в оперативній пам'яті тільки ті рядки, які спільно обробляються в даний момент;
- застосування вільних масивів, що мають рядки різної довжини, дозволяє більш ефективно використовувати пам'ять;
- при сортуванні масиву по рядках (не елементів в рядку, а самих рядків) замість переставляння рядків в пам'яті можна переставляти тільки елементи масиву покажчиків, в той час як самі рядки залишаються на місці.

```
Введіть кількість рядків: 3
Введіть кількість елементів рядка 0: 4
a[0][0]= 0
a[0][1]= 1
a[0][2]= 2
a[0][3]= 3


Введіть кількість елементів рядка 1: 2
a[1][0]= -1
a[1][1]= -2


Введіть кількість елементів рядка 2: 3
a[2][0]= 4
a[2][1]= 5
a[2][2]= 6


  0   1   2   3
 -1  -2
  4   5   6
Press any key to continue . . .
```

Рисунок 3.5 Результат виконання прикладу з вільним масивом

3.5 Трикутні масиви

Розглянемо, наприклад, матрицю суміжності, яка представляє зв'язки між вузлами мережі. В якості останньої можуть виступати літаки, що літають між

різними аеропортами. Якщо відбувається політ від аеропорту ***i*** до аеропорту ***j***, то елемент масиву *connected[i, j]* приймає значення *1*. Припустимо також, що існує зворотний маршрут від аеропорту ***j*** до аеропорту ***i***, це дає нам елемент *connected[j, i] = connected[i, j]*. Виходить, що немає необхідності зберігати *connected[i, j]* і *connected [j, i]* одночасно, оскільки вони однакові. І отже, програма може заощадити пам'ять, перетворивши матрицю суміжності в трикутний масив (Рисунок 3.6).

У трикутної матриці A_{n*n} всі елементи вище або нижче головної діагоналі є нульовими. Якщо нульовими є елементи вище головної діагоналі (нижня трикутна матриця), то ненульовими будуть елементи $a_{11}; a_{21}, a_{22}; ,a_{31}, a_{32}, a_{33}; \ldots; a_{n1}, a_{n2}, \ldots, a_{nn}$.

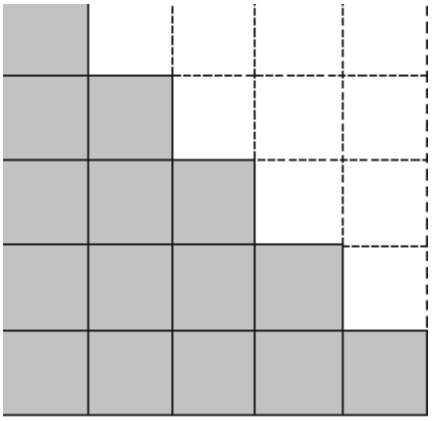


Рисунок 3.6 Приклад трикутного масиву

Таку матрицю доцільно зберігати у вигляді одновимірного масиву (вектора) *B* з числом елементів

$$m = n*(n + 1)/2. \qquad (3.4)$$

Тоді індекс i_b у лінійному масиві *B* для елемента a_{ij} нижньої трикутної матриці A (Рисунок 3.6), $i = 0, 1,..., n-1$ та $j = 0, 1,..., i$ визначається по формулі (індексація з нуля!):

$$i_b = \sum_{k=1}^{i} k + j = \frac{(i^2+i)}{2} + j, \qquad (3.5)$$

Діапазон зміни індекса i_b у масиві *B* буде становити:

$$i_b = 0,\ 1,\ \ldots,\ n* \frac{(n^2+n)}{2} - 1 \qquad (3.6)$$

Якщо у матриці *A* нульовими є елементи, розташовані нижче головної діагоналі (верхня трикутна матриця, Рисунок 3.7), то в масиві *B* ненульові елементи розмістяться в послідовності $a_{11}, a_{12}, \ldots . a_{1n}; a_{22}, a_{23}, \ldots a_{2n}; a_{nn}$, а доступ до елементу a_{ij} в цьому випадку буде здійснюватися за лінійним індексом i_b, обчислюваному за більш складною формулою (індексація як і раніше, з нуля!).

Індекс i_b у лінійному масиві B для елемента a_{ij} верхньої трикутної матриці A (Рисунок 3.7), $i = 0, 1,..., n-1$ та $j = i, i+1,..., n-1$ визначається по формулі (індексація з нуля!):

$$i_b = n * i + j - \sum_{k=1}^{i} k = n*i + j - \frac{(i^2+i)}{2}, \quad (3.7)$$

$$i_b = 0, 1, \dots, n*\frac{(n+1)}{2} - 1 \quad (3.8)$$

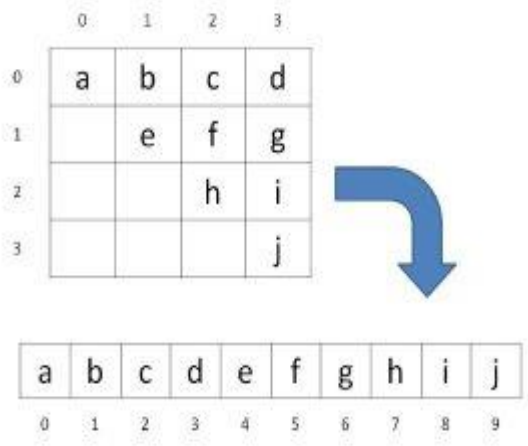


Рисунок 3.7 Приклад організації зберігання верхньої трикутної матриці

Приклад реалізації алгоритму введення даних в верхній трикутний масив і подальшого виведення їх з масиву на екран на мові Сі:

Приклад 3.8 Обробка і зберігання верхньої трикутної матриці

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "math.h"
#include "string.h"
#include "locale.h"
#include "windows.h"
#define INDEX(I,J,N)   (N)*(I)+(J)-((I)*(I)+(I))/2  // макрос обчислення індексу
                                                    // в лінійному масиві,
                                                    // формула (3.7)

int main(void) {

    int   j,      /* лічильник стовпців*/
          i,      /* лічильник рядків */
          m,      /* число елементів в масиві */
          n,      /* поточна розмірність масиву */
          k = 1, /* номер елемента в лінійному масиві */
          *pTriM = NULL;  /* покажчик на лінійний трикутний масив */
```

```
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    printf("Введіть розмірність трикутного масиву:");
    scanf("%d", &n);
    m = n*(n + 1) / 2; // число елементів в лінійному масиві, формула (3.4)

    /* Виділення пам'яті під масив з одночасним контролем наявності
       вільної пам'яті */
    if (!(pTriM = (int*)malloc(m * sizeof(int)))) {
        printf("Для такого масиву не вистачає пам'яті!");
        return 1;
    }

    /* Заповнення масиву порядковими номерами елементів в лінійному
       масиві */
    for (i = 0; i < n; ++i)
        for (j = i; j < n; ++j)
            pTriM[INDEX(i, j, n)] = k++;

    /* Друк масиву */
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j)
            printf("%d ", (j >= i) ? pTriM[INDEX(i, j, n)] : 0);
        printf("\n");
    }

    free(pTriM);
    return 0;
}
```

3.6 Розріджені масиви

Безумовно, вільні та трикутні масиви дозволяють програмі істотно економити пам'ять. Але якщо відомо, що елементів масиву буде зовсім мало, ці витрати можна скоротити ще значніше.

Приклад 3.9

Повернемося до матриці суміжностей з авіалініями, де *1* в запису *[i, j]* позначає політ між містами *i* та *j*. Міст може бути всього *600*, а з'єднані авіалініями будуть тільки *100* з них. В такому випадку у масиві з *36000* елементів виявиться лише *200* ненульових значень. Навіть з урахуванням того, що польоти будуть симетричними (на кожен політ з *i* в *j* припадатиме політ з *j* в *i* і ви збережете їх в трикутному масиві, з приблизно *18000* елементів ненульовими виявляться тільки 100. Це означає, що майже 99% масиву не використовується. Це в нас будуть великі негаразди!

Приклад 3.10

Рішення систем лінійних рівнянь з великою кількістю невідомих в наукових і технічних додатках. Розглянемо систему таких рівнянь:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases} \quad (3.9)$$

Нехай $m = n$, тобто число рівнянь дорівнює числу невідомих. В цьому випадку система буде мати єдине рішення. Припустимо, що матриця коефіцієнтів A системи матиме такий вигляд (як воно найчастіше і буває!), як на рисунку нижче:

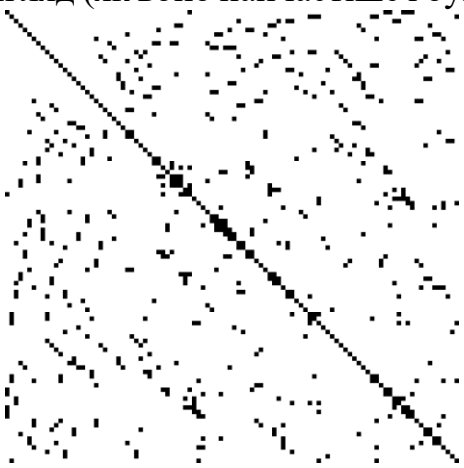


Рисунок 3.8 Розріджена матриця A системи з ненульовими елементами

Як видно з прикладів, **розрідженими** називаються матриці, що містять велику кількість нульових елементів. Подання таких матриць у вигляді двовимірних масивів призводить до нераціонального використання пам'яті і неефективності виконуваних над ними операцій. Тому, для зберігання таких матриць використовуються спеціальні схеми.

Перший і найбільш простий спосіб полягає в наступному. Створюється двовимірний масив розміром *(n x 3)*, де *n* - число ненульових елементів вихідної матриці. Перші два елементи кожного рядка містять індекси рядка і стовпця ненульового елемента, а третій - сам ненульовий елемент (Рисунок 3.9, слід відмітити, що для цього приклада індексація масиву починається не з 0, а з 1!). Якщо ж значення елементів вихідної матриці нецілого типу, то замість одного двомірного масиву *(n x 3)* створюються три одновимірних масива з *n* елементами кожний. Обробка таких масивів особливих труднощів не викликає.

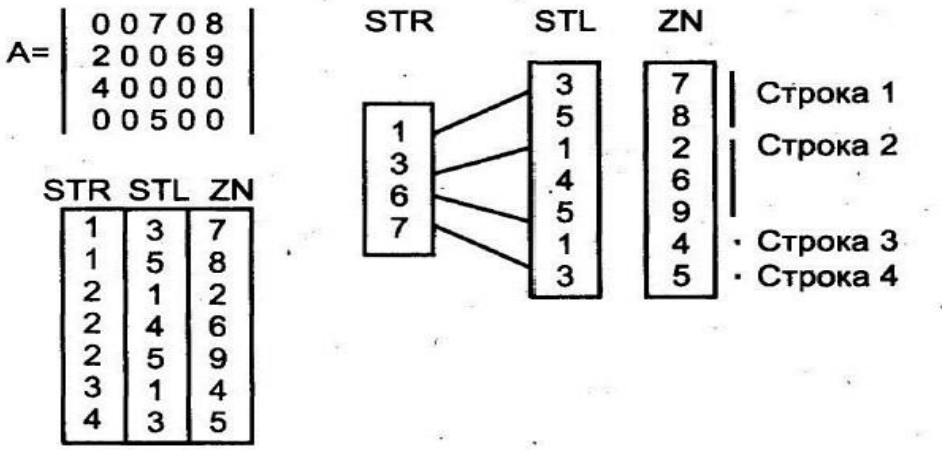


Рисунок 3.9 Два варіанти подання розрідженої матриці

Є можливість ще дещо скоротити обсяг необхідної пам'яті. Створюються три окремі масиви: масив *STR* по числу рядків вихідної матриці; масиви *STL* і *ZN* по числу ненульових елементів вихідної розрідженої матриці. У масив *STL* послідовно заносяться індекси стовпців ненульових елементів вихідної матриці, спочатку для першого рядка, потім для другого, тощо. Індекси стовпців кожного *i*-го рядка утворюють *i*-у групу. Кожен *i*-й елемент масиву *STR* містить індекс початку *i*-ї групи в масиві *STL*. У масив *ZN* заносяться значення відповідних ненульових елементів (на Рисунок 3.9 праворуч).

Алгоритми обробки розрідженої матриці при такому поданні трохи ускладнюються, так як число ненульових елементів в рядках вихідної матриці різне, а масив *STL* не містить ознак кінця груп. Тому для переходу з одного рядка матриці на іншу потрібно використовувати інформацію з масиву *STR*.

Ще більш ефективним і економним є спосіб подання розрідженої матриці з використанням багатозв'язних списків з динамічним отриманням пам'яті для ненульових елементів матриці, кожен з яких має наступну структуру:

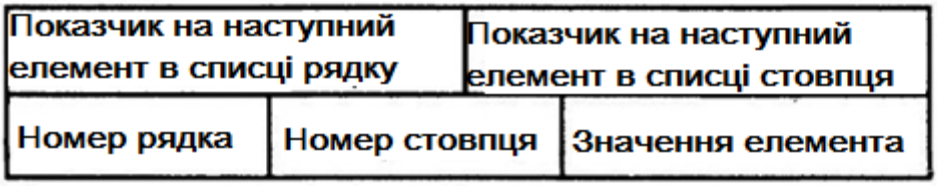


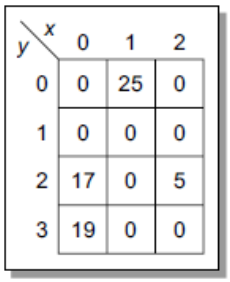
Рисунок 3.10 Структура елемента багатозв'язного списку розрідженої матриці

Тут для кожного рядка і кожного стовпця будуються зв'язні циклічні списки. Ненульовий елемент a_{ij} вихідної матриці потрапляє в обидва списки - *i*-й циклічний

список для *i*-го рядка і *j*-й циклічний список для *j*-го стовпця. Тому кожен елемент структури має містити, як мінімум, два покажчика - покажчик на наступний елемент у списку рядка і покажчик на наступний елемент у списку стовпця.

Кожен список починається з дескриптора рядка (стовпця). Вузол списку, який являє собою ненульовий елемент матриці, використовує всі п'ять полів структури елемента. У дескрипторі рядків/стовпців списку використовується тільки одне поле - покажчик на рядок у списку рядка або покажчик на стовпчик - в списку стовпця.

Приклад такої матриці наведено на малюнку нижче:



y \ x	0	1	2
0	0	25	0
1	0	0	0
2	17	0	5
3	19	0	0

Рисунок 3.11 Приклад розрідженої матриці

Стовпці і рядки в такій матриці можуть бути представлені у вигляді багатозв'язного списку, як це показано на малюнку вище. Оскільки кожен із значущих елементів такої матриці знаходиться рівно в одному рядку і одному стовпці, він буде відображатися в обох потоках рівно один раз. Структура кожного вузла такого списку буде містити значення самого елемента матриці, зберігати покажчик на наступний вузол (ненульовий елемент матриці) в рядку, зберігати покажчик на наступний вузол (ненульовий елемент матриці) в стовпчику, зберігати значення індексів для ненульового елемента масиву. Проте, як і в разі двоспрямованих списків, ми можемо також добавити до структури і відповідний покажчик попереднього вузла для кожного потоку в двоспрямованих списках, які представляють розріджену матрицю.

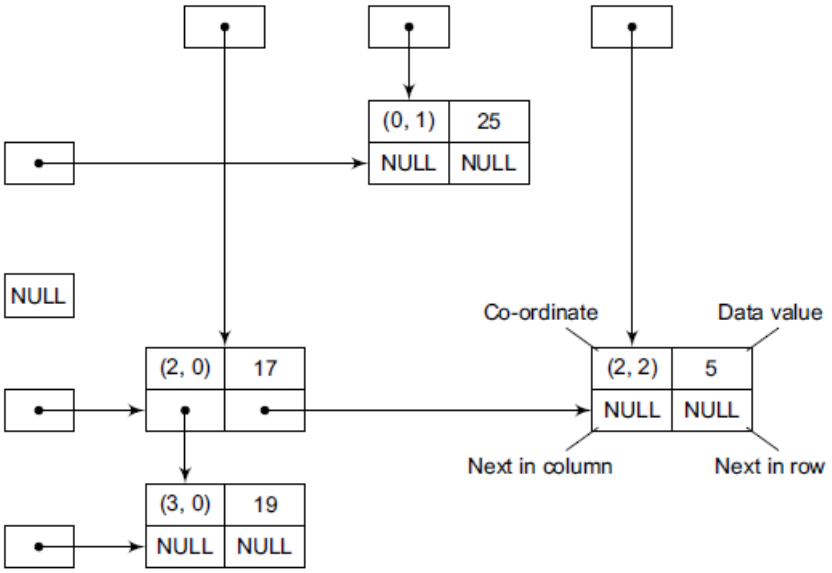


Рисунок 3.12 Представлення розрідженої матриці з рисунку 3.11 у вигляді двоспрямованого списку

3.7 Контрольні завдання і запитання

1. Напишіть алгоритм та програму розрахунку вибіркової дисперсії для одновимірного масиву, що містить N елементів, яка визначається наступним рівнянням:

$$S = \frac{1}{N}\sum_{i=0}^{N-1}(x_i - \bar{x})^2$$

де $\bar{x}$ - середня величина значень в масиві.

2. Створіть алгоритм та програму розрахунку стандартного відхилення вибірки для одновимірного масиву чисел, що визначається як квадратний корінь дисперсії.

3. Створіть алгоритм та програму видалення елементу з лінійного динамічного масиву.

4. Трикутні масиви іноді називаються нижніми трикутними, оскільки всі значення зберігаються у нижній лівій половині. Напишіть алгоритм та програму отримання з нижнього трикутного масиву верхнього трикутного масиву зі значеннями у правому верхньому кутку. Який зв'язок між рядком та стовпцем для кожного елемента такого масиву?

5. Припустимо, що вам потрібно провести діагональ у прямокутному (не квадратному) масиві з лівого верхнього кута у правий нижній. Напишіть алгоритм та програму, яка прирівнює елементи масиву вздовж діагоналі або під нею до 1, а

над діагоналлю - до 0.

6. Діагональ у прямокутному масиві починається з правого верхнього та йде до лівого нижнього кутка. Розробіть алгоритм та програму, яка надає елементам уздовж діагоналі або над нею значення 1, а під діагоналлю - 0.

7. Напишіть алгоритм та програму, де значення кожного елемента прямокутного масиву — це відстань до найближчого краю масиву.

8. Напишіть алгоритм та програму для додавання двох трикутних масивів.

9. Розробіть алгоритм та програму для множення двох трикутних масивів.

10. Напишіть алгоритм та програму для додавання двох розріджених масивів.

4 СОРТУВАННЯ

4.1 Загальні поняття

Сортування слід розуміти як процес перегрупування однотипних елементів структури даних в деякому певному порядку [1]. Алгоритм сортування визначається як алгоритм, що розміщує елементи якоїсь однотипної послідовності даних (масив, список, файл) в певному порядку, який може бути або числовим порядком, або лексикографічним порядком, або будь-яким іншим заданим користувачем порядком. Мета сортування - полегшити подальший пошук, оновлення, виключення, включення елементів в структуру даних. На відсортованих даних легше визначити, чи є пропущені елементи, чи всі елементи перевірені, легше знайти загальні елементи двох однотипних структур, злити їх воєдино. Сортування є важливим засобом для прискорення роботи практично будь-якого алгоритму, в якому потрібно часте звертання до певних елементів структури даних.

Розроблено безліч алгоритмів сортування, проте немає алгоритму, який був би найкращим в будь-якому випадку. Ефективність алгоритму сортування може залежати від ряду факторів, таких, як:

- числа сортованих елементів;
- діапазону і розподілу значень сортованих елементів;
- ступеню початкової відсортованості елементів;
- характеристик алгоритму (складність, вимоги до пам'яті, тощо);
- місця розміщення елементів (в оперативній пам'яті (масив) або на диску (файл)).

Залежно від останнього фактора всі методи сортування розбивають на два класи: *внутрішнє сортування* (сортування масивів) і *зовнішнє сортування* (сортування файлів, або сортування послідовностей). Зовнішнє сортування застосовується, коли весь обсяг сортованих даних не може одночасно розміститися в пам'яті комп'ютера.

Сортовані елементи часто являють собою записи даних певної структури. Кожен запис має поле ключа, за значенням якого здійснюється сортування, і поля даних. При розгляді алгоритмів сортування нас цікавить тільки поле ключа, тому

інші поля сортованих структур, як і у випадку зі списками і деревами, розглядатися не будуть.

Метод сортування називається *стійким*, якщо в процесі сортування відносне розташування елементів з однаковими (рівними) ключами не змінюється. Стійкість сортування бажана, якщо мова йде про елементи, вже відсортовані по деяким іншим критеріям (властивостям), що не впливають на ключ, за яким зараз здійснюється сортування. Наведемо приклад:

Приклад 4.1 Поняття стійкості сортування на прикладі викладання товару на полицю магазину

Скажімо, в магазин завозять однаковий товар, нехай це будуть торти. Усі торти заносяться в якийсь масив послідовно, за часом завоза. При цьому ключем є ціна торта.

Так як торти продукт, що швидко псується, продати його треба як можна швидше, тобто це мета магазину. Але в той же час, так як торти - це не найдешевші предмети в магазині, покупець буде прагнути купити торт дешевше.

У магазин торти завозили протягом усього тижня, і під кінець тижня треба продати, що залишилося, як можна швидше. Для визначення тортів з мінімальною ціною ми сортуємо масив тортів стійким сортуванням по зростанню ключа (ціни), і отримуємо спочатку найдешевші, а якщо є торти з однаковою ціною, то першим буде торт, який прибув в магазин раніше (отже продати його треба швидше, так як зіпсується він раніше). Якби для сортування по ціні було вибрано нестійкий алгоритм сортування, то магазин міг вчинити не дуже раціонально і продати торти, які могли б ще полежати.

Основними вимогами до алгоритмів сортування, як і до будь-яких алгоритмів, є вимоги по пам'яті і часу виконання. Це передбачає, що внутрішнє сортування (елементів масиву) виконується на місці, без передачі їх в результуючий масив. Гарною мірою ефективності алгоритму сортування за часом може бути число необхідних порівнянь ключів C і число пересилань M, які залежать від числа сортованих елементів n.

Записи даних можуть бути відсортовані на основі якоїсь властивості. Таке поле (поля) записів або властивостей називається ключем сортування. Досить часто, при виконанні реальних додатків, виникає необхідність сортування масиву записів з використанням декількох ключів. Так відбувається зазвичай тоді, коли одного ключа не достатньо, щоб однозначно ідентифікувати запис. Наприклад, у великій організації, ми можемо захотіти впорядкувати список співробітників на основі їх відділів, а потім, в межах кожного відділу, відповідно до їх імен в

алфавітному порядку.

Іншими прикладами сортування по декількох ключах можуть бути:

- телефонні довідники, в яких імена відсортовані за місцем розташування, категорії (бізнес або громадяни), а потім і в алфавітному порядку;
- в бібліотеці, інформація про книги може бути відсортована в алфавітному порядку на основі назв, а потім по іменах авторів;
- адреси клієнтів можуть бути відсортовані за назвою населеного пункту клієнта, а потім і по вулиці.

Таким чином, повне сортування і його ключ може бути визначене за допомогою двох або більше ключів часткових сортувань. В такому випадку, перший ключ називається первинним ключем сортування, а другий і наступні - вторинними ключами сортування.

Приклад 4.2 Приклад сортування по декількох ключах, первинний ключ Кафедра, вторинний ключ - ПІБ

П.І.Б.	Кафедра	Група	Телефон
Кублій Л.І.	АТЕП	ТА-09	098-12345678
Владіміров В.М.	АЕС	ТР-21	093-57093531
Ногін М.В.	ТП	ТФ-12	073-38987654
Цуканова А.О.	АТЕП	ТО-09	096-54123456
Іванов І.А.	АЕС	ТР-22	093-50123455

Відсортована таблиця:

П.І.Б.	Кафедра	Група	Телефон
Владіміров В.М.	АЕС	ТР-21	093-57093531
Іванов І.А.	АЕС	ТР-22	093-50123455
Кублій Л.І.	АТЕП	ТА-09	098-12345678
Цуканова А.О.	АТЕП	ТО-09	096-54123456
Ногін М.В.	ТП	ТФ-12	073-38987654

4.2 Алгоритми внутрішнього сортування швидкості O(N^2)

Швидкість $O(N^2)$ - це не дуже хороший показник, проте для відносно малих масивів (менш 10 000 елементів) він є достатньо швидким. До того ж, ці алгоритми досить прості, тому **для зовсім малих масивів** (менше 1000 елементів) можуть працювати навіть ефективніше, ніж більш складні алгоритми. Наскільки малим повинен бути такий масив, залежить від вашої системи. Зазвичай, декілька тисяч елементів.

4.2.1 Сортування бульбашкою

Бульбашкове сортування передбачає наступне: якщо масив не відсортований, суміжні елементи в ньому можуть знаходитися один відносно другого в невірному положенні. Через це алгоритм повинен пройти по масиву кілька разів, міняючи місцями всі неправильні пари. Іллюструє цей алгоритм рисунок нижче. Перший масив виглядає здебільшого відсортованим, але, пройшовшись по ньому, алгоритм виявить, що пара 6 - 3 знаходиться в неправильному положенні (число 6 має слідувати після 3). Алгоритм поміняє знайдені елементи місцями і отримає другий масив, в якому в неправильному положенні виявиться пара 5 - 3. Після її виправлення утворюється третій масив, де невірна пара 4 - 3. Помінявши її елементи місцями, алгоритм сформує четвертий масив, зробить ще один кінцевий прохід і, не знайшовши пар, що стоять в неправильному положенні, зупиниться.

У бульбашкового сортування невпорядкований елемент 3 як би повільно «спливає» на правильну позицію, звідси і специфічну назву методу. Кожне проходження через масив ставить на потрібне місце як мінімум один елемент. У масиві, наведеному на Рисунок 4.1, при першому проходженні на правильній позиції виявляється число 6, при другому - 5, при третьому – 3, тощо [2].

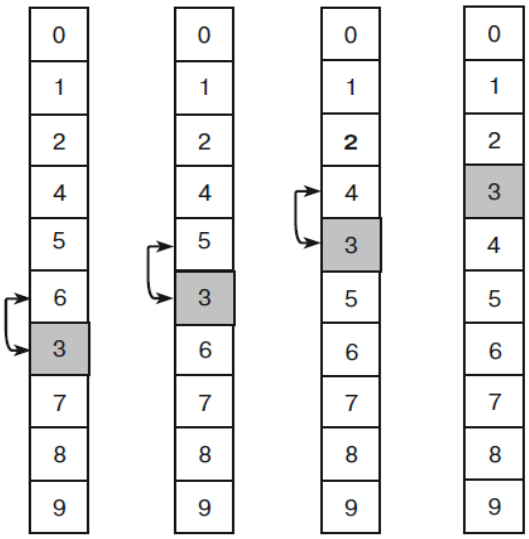


Рисунок 4.1 Ілюстрація методу бульбашкового сортування

Приклад 4.3 Алгоритм сортування бульбашкою на мові С

```
int sortBubble (int *arr, /* покажчик на сортований масив */
        int n   /* розмір масиву */ )  {
```

```
    int j,   /* лічильник циклу */
    notSorted=1,        /* ознака невпорядкованості */
    temp; /* тимчасова проміжна змінна при обміні */

    if (n < 1) return 1; /* масиву немає, сортувати нічого */

    while (notSorted) { // цикл виконується за наявності не впорядкованості

        notSorted = 0;      // скидання ознаки невпорядкованості

       // проходження по масиву в пошуках невпорядкованих пар
        for (j=0; j < n-1; j++) {

            // якщо пара знайдена - міняємо елементи пари місцями
            if (arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            notSorted = 1;      // встановлюємо ознаку
                                // невпорядкованості
            }

        }

    }

    return 0;

}
```

Якщо припустити, що в масиві міститься *N* елементів і хоча б один з них переміщається і займає своє місце в результаті однократного перегляду значень, то алгоритм може зробити не більше *N* проходжень. (Всі *N* знадобляться, коли масив спочатку відсортований у зворотному порядку.) Кожне таке проходження включає *N* кроків, звідси загальний час роботи алгоритму - $O(N^2)$.

Сортування бульбашкою відноситься до стійких алгоритмів сортування.

4.2.2 **Сортування вставкою**

Сортування вставкою є прикладом дуже простого алгоритму сортування, в якому відсортований масив (або список) будується по одному елементу за один раз. Гравці в бридж знайомі з цією технікою сортування, так як вона зазвичай використовується для замовлення колоди карт під час гри.

Основною ідеєю алгоритму є та, що він вставляє кожен елемент на своє місце в остаточному списку (звідси і назва). Алгоритм сортування вставкою працює наступним чином:

- масив значень для сортування ділиться на дві групи. Перша група зберігає

відсортовані значення, а друга - містить невідсортовані значення;

- алгоритм сортування буде виконуватися до тих пір, поки ми маємо хоч один елемент в невідсортованій частині масива значень;
- нехай спочатку масив значень нараховує *n* елементів. Вважаємо, що елемент масиву з індексом *0* (перший елемент) відразу знаходиться у відсортованій частині. Інші елементи знаходяться в невідсортованій частині. Перший елемент невідсортованої частини має індекс 1 (другий елемент масиву);
- під час кожної ітерації алгоритму, перший елемент *i* (номер ітерації) в невідсортованій частині порівнюємо по черзі з усіма елементами відсортованої частини і в разі виявлення серед відсортованих елементів елемента з більшим значенням ключа, ніж порівнюваний (індекс *j*), зсовуємо всі елементи масиву в місці вставки вправо на одну позицію до *i*, а на місце, що звільнилося, вставляємо наш елемент *i* із невідсортованої частини.

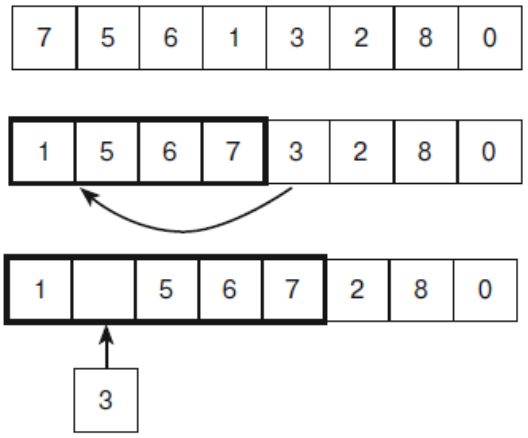


Рисунок 4.2 Ілюстрація роботи алгоритму сортування вставкою

Переміщення масиву може зайняти якийсь час. Адже якщо новий індекс елемента повинен бути *j*, код повинен зрушити все елементи між індексами *j* і *i* на одну позицію вправо, щоб звільнити місце.

На малюнку вище показано основні кроки алгоритму. Верхній фрагмент являє собою вхідний невідсортований масив. На середньому фрагменті перші чотири елементи (вони обведені товстою лінією) вже відсортовані, і алгоритм готується додати до них наступний елемент із значенням 3. Алгоритм переглядає вже відсортовані елементи зліва, поки не визначить, що число 3 потрібно вставити перед числом 5. На нижньому фрагменті видно, що алгоритм пересунув значення 5, 6 і 7 вправо, щоб звільнити місце для значення 3. Після запису числа 3 на нове

місце, він продовжить основний цикл, щоб упорядкувати наступний елемент зі значенням 2.

Приклад 4.4 Алгоритм сортування вставками на С

```
int sortInsert (int *arr, int n) {
    int i, j, temp;
    if (n < 1)
    return 1;      /* масиву немає, сортувати нічого */
    for (i=1; i < n; i++) { // основний цикл
        temp = arr[i];
        j = i-1;
        while (j>=0 && temp <= arr[j]) { // внутрішній цикл зсуву
            arr[j+ 1]= arr[j];
            j--;
        }
        arr[j+ 1]= temp;
    }
    return 0;
}
```

Якщо в масиві міститься *N* елементів, алгоритм розглядає всі. Для кожної позиції *i* він повинен знайти попередні відсортовані елементи і визначити нове місце для сортованого значення. Потім йому потрібно пересунути елементи між цим положенням і положенням індексу *i* на одну позицію вправо. Так, якщо елемент *i* повинен зміститися на позицію *j*, потрібно спочатку *j* кроків, а потім ще *(i–j)* кроків для обміну елементів. В результаті загальна кількість кроків дорівнюватиме *i*.

Якщо підсумувати кроки, необхідні для позиціонування всіх елементів, загальний час роботи складе $1 + 2 + 3 + ... + N = N * (N + 1) / 2$, звідси продуктивність алгоритму — $O(N^2)$.

Сортування вставкою відноситься до стійких алгоритмів сортування.

4.3 Алгоритми внутрішнього сортування швидкості O(N*log(N))

Всі прямі методи сортування фактично пересувають кожен елемент на кожному кроці на одну позицію. Тому вони вимагають порядку $O(N^2)$ таких кроків. Звідси випливає, що в основу будь-яких поліпшень повинен бути покладений принцип переміщення елементів на кожному кроці на максимальну відстань.

4.3.1 **Швидке сортування**

Швидке сортування - широко використовуваний алгоритм сортування, розроблений Чарльзом Хоаром в 1960 році. Швидке сортування здатне виконати в середньому *O(N*log(N))* порівнянь при сортуванні масиву з *N* елементів. Проте в гіршому випадку він може мати квадратичний час роботи $O(N^2)$. Зазвичай, алгоритм швидкого сортування виконується швидше, ніж інші *O(N*log(N))* алгоритми, так як его ефективна реалізація може звести до мінімуму ймовірність того, що стане потрібен квадратичний час. Швидке сортування також відоме як сортування Хоара.

Швидке сортування є істотно поліпшеним варіантом алгоритму сортування за допомогою прямого обміну (сортування бульбашкою), відомого, в тому числі, своєю низькою ефективністю. Принципова відмінність полягає в тому, що в першу чергу проводяться перестановки елементів на найбільшій можливій відстані і після кожного проходу елементи діляться на дві незалежні групи. Цікавий факт: поліпшення найнеефективнішого прямого методу сортування дало в результаті один з найбільш ефективних поліпшених методів.

Загальна ідея алгоритму швидкого сортування полягає в наступному:

1. Вибрати з масиву елемент, що назвемо опорним. Це може бути будь-який з елементів масиву або ж число, обчислене на основі значень елементів. Від вибору цього числа сильно залежить ефективність алгоритму.
2. Порівняти всі інші елементи з опорним і переставити їх у масиві так, щоб розбити масив на дві неперервні частини (підмасиви), які слідують одна за одною: «менші за опорний», «рівні або більші за опорний».
3. Рекурсивно впорядкувати підмасиви «менших» і «більших» значень, якщо довжина підмасиву більше одиниці. Базовий випадок рекурсії буде мати місце, коли підмасив має нуль або один елемент, так як в цьому випадку він вже відсортований.

Для коректності алгоритму значення опорного елемента має знаходитися між максимальним і мінімальним (включно) значеннями в масиві (підмасиві). Відомі стратегії:

- вибирати постійно один і той же елемент, наприклад, середній або останній по положенню;

- вибирати елемент з випадково обраним індексом;
- вибирати середнє арифметичне значень елементів масиву (підмасиву).

Приклад 4.5 Приклад алгоритму знаходження опорного елемента в масиві, що сортується

Приклад алгоритму знаходження опорного елемента в масиві, що сортується, з одночасним розбиттям масиву на два підмасива (відрізка). Нехай змінна *loc* містить індекс опорного елемента в масиві, змінна *right* - індекс правої границі, змінна *left* - індекс лівої границі. Тоді алгоритм працює наступним чином:

1. Встановити *loc = left* і рівним індексу першого елемента в масиві. Встановити *right* рівним індексу останнього елемента масиву. Тобто, для Сі, *loc = left = 0, right = N-1* (де *N* - кількість елементів в масиві).
2. Почнемо з елементу, на який вказує *right* і скануємо масив справа наліво, порівнюючи кожен елемент на шляху з елементом, на який вказує *loc*. Тобто, елемент *array[loc]* при цьому повинен залишатися меншим, ніж елемент array*[right]:*
   a) якщо це так, то просто продовжуємо сканування, поки *right* не стане рівним *loc*. Якщо індекс *right* стане рівним індексу *loc*, це буде означати, що положення опорного елемента вибрано вірно;
   b) однак, якщо в якийсь момент виконається умова *array[loc] > array[right]*, то слід поміняти місцями ці два елементи масиву і перейти до кроку 3;
   c) встановити *loc = right.*
3. Почати з елемента, на який вказує *left* і сканувати масив зліва направо, порівнюючи кожен елемент на шляху з елементом, на який вказує *loc*. Тобто, елемент *array[loc]* повинен бути більшим, ніж *array[left]:*
   a) якщо це так, то просто продовжуємо сканування, поки *left* не стане рівним *loc*. Якщо індекс *left* стане рівним індексу *loc,* це означає, що опорний елемент знаходиться в правильному положенні;
   b) однак, якщо в якийсь момент, виконається умова *array[loc] <array[left],* то слід поміняти місцями ці два елементи масиву і перейти до кроку 2;
   c) встановити *loc = left.*

Роботу цього алгоритму проілюструємо рисунками нижче. Нехай вихідний

несортований масив виглядає наступним чином:

Рисунок 4.3 Вихідний несортований масив елементів

Кроки алгоритму з Приклад 4.5 показані на малюнках нижче.

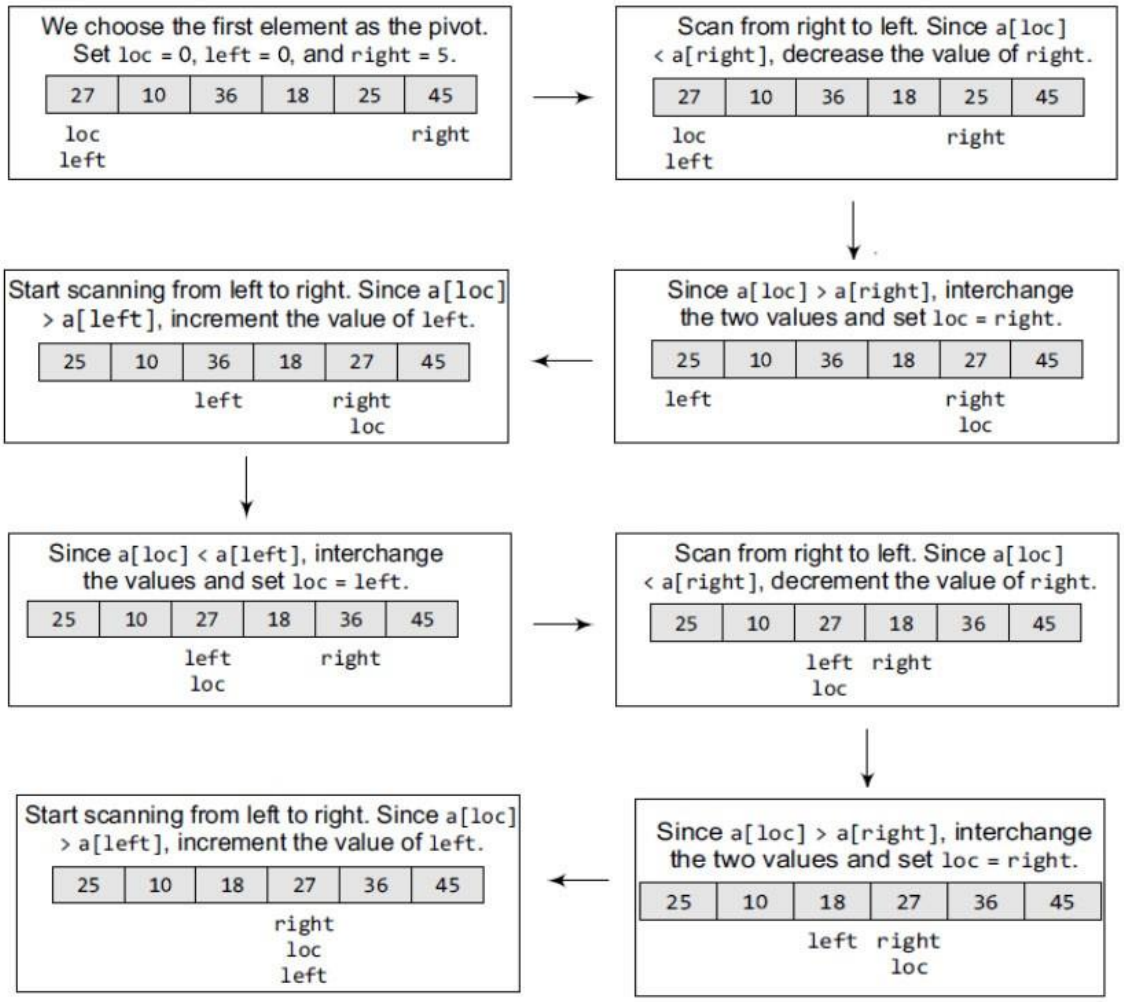


Рисунок 4.4 Ілюстрація знаходження опорного елемента і поділу масиву на дві частини

Як видно з малюнка вище, *left = right*, тому алгоритм завершується тим, що опорним елементом стає елемент 27. Всі елементи менші ніж 27 розміщені перед ним, а всі елементи більші за 27 - розміщено після нього.

Далі, лівий підмасив, що містить 25, 10, 18 і правий підмасив, що містить 36 та 45 упорядковуються рекурсивно таким же чином.

Сама реалізація алгоритму швидкого сортування на мові Сі може виглядати наступним чином:

Приклад 4.6 Реалізація алгоритму швидкого сортування

```
/*      Допоміжна функція, що реалізує алгоритм пошуку опорного елемента
 і поділу сортованого масиву на дві частини */

void partition (int *arr, int beg, int end, int *loc) {
        int flag,       // флаг закінчення поділу
        temp, // проміжна змінна для обміну значень в поточному підмасиві
        left, right;    // лівий та правий рухомі індекси

        // ініціалізація алгоритму, відповідно до п. 1 Приклад 4.5
        left = *loc = beg;
        right = end;
        flag = 0;

        // виконуємо, поки не "розбили" масив на два підмасива
        while (!flag)  {

        // Йдемо справа наліво по масиву, відповідно до п. 2 Приклад 4.5
        while (arr[*loc] <= arr[right] && *loc != right)
                right--;
        if (*loc == right)   /* розбиття масиву і робота алгоритму закінчена */
                flag = 1;
        else if (arr[*loc] > arr[right]) {
                temp = arr[right];
                arr[right] = arr[*loc];
                arr[*loc] = temp;
                *loc = right;
        }

        // тепер рухаємося зліва направо по масиву, згідно п. 3 Приклад 4.5
        if (!flag)  {
                while (arr[*loc] >= arr[left] && *loc != left)
                        left++;
                if (*loc == left)      /* розбиття масиву і робота алгоритма
                                          закінчена */
                        flag = 1;
                else if (arr[*loc] < arr[left]) {
                        temp = arr[left];
                        arr[left] = arr[*loc];
                        arr[*loc] = temp;
                        *loc = left;
                }
                }
        }   // кінець циклу while (!flag)

        return;

}


/* Основна функція, яка, власне і реалізує в собі алгоритм швидкого
```

```
сортування */
void sortQuick (int *arr, int beg, int end) {
    int loc;

    if (beg < end) {
        partition (arr, beg, end, &loc);
        sortQuick (arr, beg, loc-1);
        sortQuick (arr, loc+ 1, end);
    }

    return;

}
```

Зауважимо, що алгоритм швидкого сортування є нестійким.

Оцінимо швидкість виконання алгоритму. Функція поділу масиву, яка просто перебирає елементи масиву один раз, використовує *O(N)* часу. У кращому випадку, кожен раз, коли ми розбиваємо масив, ми ділимо його на дві майже рівні частини. Тобто, рекурсивний виклик обробляє підмасиви приблизно половинного розміру. В цьому випадку, може бути зроблено тільки $log_2(N)$ рекурсивних викликів, перш ніж ми досягнемо підмасива розміром 1 елемент. Тому, результуюча складність всього алгоритму визначиться добутком часу перебору всього масиву *O(N)* на час виконання рекурсивних викликів $O(log_2(N))$ при кожному розбитті. Таким чином, ми і отримуємо оцінку *O(N*log(N))*.

Практично, ефективність швидкого сортування залежить від елемента, який був обраний в якості опорного. При невдалому виборі опорного елемента швидкість алгоритму складе $O(N^2)$. Найгірший випадок відбувається, коли масив вже відсортований за даним ключем (в порядку зростання або зменшення) і крайній лівий елемент обраний як опорний. Саме тому багато реалізацій швидкого сортування намагаються випадковим чином вибирати опорний елемент. Рандомізована версія алгоритму швидкого сортування завжди має алгоритмічну складність *O(N*log(N))*.

4.3.2 **Сортування Шелла**

Сортування Шелла винайдено Дональдом Шеллом в 1959 році. Це алгоритм сортування, який є узагальненням різновиду сортування вставкою. При аналізі алгоритму сортування вставкою можна помітити, що:

- по-перше, сортування вставкою працює добре, коли вхідні дані майже відсортовані;

- по-друге, сортування вставкою вельми неефективно використовувати, так як воно переміщує значення тільки на одну позицію за один раз.

Сортування Шелла вважається поліпшенням сортування вставкою, так як воно при роботі порівнює елементи масиву, розділені проміжком в кілька позицій. Це дозволяє елементу здійснювати великі стрибки в напрямку своєї очікуваної позиції в масиві. У Шелла елементи сортуються за кілька проходів і в кожному проході елементи беруться все з меншими і меншими проміжками між ними. Останній етап сортування Шелла є простим варіантом вставки. Але на той час, коли ми дійшли до останнього етапу, елементи вже "майже відсортовані", і отже, алгоритм забезпечить хорошу продуктивність.

Якщо взяти сценарій алгоритму, при якому найменший елемент зберігається в іншому кінці масиву, то сортування такого масиву "бульбашкою" або вставкою буде виконуватися зі швидкістю $O(N^2)$. З іншого боку, алгоритм Шелла першими переміщує малі значення на велику відстань всього лише за допомогою декількох порівнянь і обмінів.

Щоб уявити собі, яким чином працює сортування Шелла, слід виконати такі дії:

- крок 1: Уявімо елементи лінійного масиву у вигляді двовимірної таблиці і станемо сортувати стовпці таблиці (за допомогою сортування вставкою);
- крок 2: Повторимо крок 1, щоразу збільшуючи висоту (довжину) колонок таблиці з одночасним зменшенням числа колонок в таблиці. В кінцевому підсумку, ми прийдемо знову до лінійного масиву (єдина колонка таблиці, кожен рядок якої містить єдиний елемент), але вже з практично відсортованим вмістом.

Зверніть увагу, що ми тільки візуалізуємо масив таблицею, сам же алгоритм виконує його сортування прямо на місці.

Приклад 4.7 Приклад упорядкування масиву сортуванням Шелла

Маємо не відсортований масив на малюнку нижче

```
63,  19,  7,  90,  81,  36,  54,  45,  72,  27,  22,  9,  41,  59,  33
```

Рисунок 4.5 Початковий невідсортований масив

Розташуємо елементи шуканого масиву у вигляді таблиць нижче і відсортуємо в них кожен стовпець:

63	19	7	90	81	36	54	45
72	27	22	9	41	59	33	

Невідсортований

63	19	7	9	41	36	33	45
72	27	22	90	81	59	54	

Відсортований за стовпцями

Тепер масив виглядає так:

63, 19, 7, 9, 41, 36, 33, 45, 72, 27, 22, 90, 81, 59, 54

Повторюємо крок 1 з меншим числом стовпців, але з більш довгими стовпцями:

63	19	7	9	41
36	33	45	72	27
22	90	81	59	54

Невідсортований

22	19	7	9	27
36	33	45	59	41
63	90	81	72	54

Відсортований за стовпцями

Масив тепер виглядає так:

22, 19, 7, 9, 27, 36, 33, 45, 59, 41, 63, 90, 81, 72, 54

Повторюємо крок 1 з ще меншим числом стовпців:

22	19	7
9	27	36
33	45	59
41	63	90
81	72	54

Невідсортований

9	19	7
22	27	36
33	45	54
41	63	59
81	72	90

Відсортований за стовпцями

Сам масив виглядає так:

9, 19, 7, 22, 27, 36, 33, 45, 54, 41, 63, 59, 81, 72, 90

В кінці отримуємо невідсортований масив з однієї колонки:

9
19
7
22
27
36
33
45
54
41
63
59
81
72
90

Відсортувавши і її - отримаємо результат - повністю відсортований початковий масив:

7, 9, 19, 22, 27, 33, 36, 41, 45, 54, 59, 63, 72, 81, 90

Графічна іллюстрація переміщень елементів масиву в алгоритмі Шелла наведена на наступному рисунку:

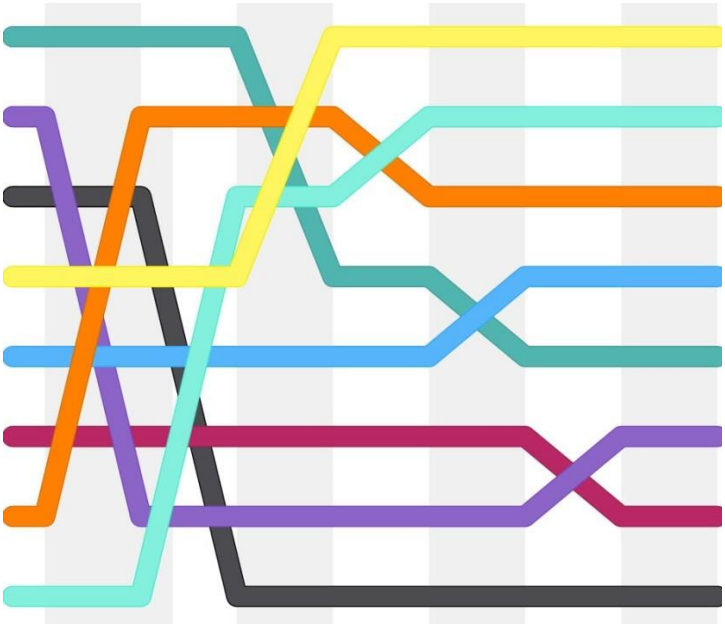


Рисунок 4.6 Процес перестановки елементів в сортуванні Шелла

Приклад 4.8 Приклад реалізації алгоритму сортування Шелла

```
int sortShell (int *arr, int n) {
        int flag,       /* флаг необхідності продовжувати сортування */
        gapSize,        /* довжина рядка віртуальної таблиці */
        i,      /* лічильник числа "стрибків" в масиві */
        temp; /* проміжна змінна при обміні */

        // ініціалізація
        flag = 1;
        gapSize = n;
        if (n < 1)      /* немає масиву, нічого сортувати */
                return 1;

        /* працюємо, поки довжина рядка віртуальної таблиці більше 1 або
         сортування не закінчено */
        while (flag == 1 || gapSize > 1) {

                flag =0;
                gapSize = (gapSize + 1) / 2;  /* на кожному проході ділимо
                                                 довжину рядка  наполовину */

                /* зміщуємося по масиву на довжину рядка */
```

```
                for (i = 0; i < (n - gapSize); i++)
                        if (arr[i + gapSize] < arr[i]) {
                                temp = arr[i + gapSize];
                                arr[i + gapSize] = arr[i];
                                arr[i] = temp;
                                flag = 1;
                        }
        }
        return 0;
}
```

Алгоритм сортування Шелла є нестійким.

4.4 Алгоритми внутрішнього сортування, швидші за *O(N*log(N))*

Найшвидшому алгоритму, що використовує порівняння елементів сортованого масиву виду $a_i <> a_j$,, для сортування N елементів потрібно, як мінімум $O(N*log(N))$ часу. Розглянуті в розділах 4.3 сортування (швидке і Шелла)) в очікуваному випадку досягають такої межі, тому може здатися, що межа швидкості досягнута. Але якщо загострити увагу на словах «алгоритм, що використовує порівняння» і згадати, що в математиці відношення порядку (впорядкованість елементів множини) може бути побудовано і на інших принципах, то зазначена продуктивність перестає бути межею. Далі, познайомимося з двома алгоритмами, де сортування здійснюється за час, який значно менше, ніж $O(N*log(N))$.

4.4.1 Сортування підрахунком

Подібний метод зручно використовувати в тому випадку, якщо сортувати доводиться цілі числа, що лежать у відносно невеликому діапазоні. Застосування сортування підрахунком доцільно лише тоді, коли сортовані числа мають, або їх можна відобразити в діапазон можливих значень, який досить малий у порівнянні з множиною, що сортується. Наприклад, нам потрібно упорядкувати 1 млн цілих чисел, значення яких знаходяться в діапазоні від 0 до 1000. Основна ідея полягає в тому, щоб встановити кількість елементів масиву з певним значенням, а потім скопіювати це значення по порядку потрібну кількість разів назад в масив.

Роботу алгоритму можна описати таким чином. Нехай у нас є вхідний масив елементів (цілих чисел) A, що сортуються. Створимо допоміжний масив лічильників $C\ [0..k-1]$, що спочатку буде складатися з нулів, а потім послідовно

будемо читати елементи вхідного масиву *A* і для кожного *A[i]* збільшувати відповідний *C[A[i]]* на одиницю. В кінці роботи алгоритму досить пройти по масиву лічильників *C*, і для кожного індексу з масиву $j \in \{0, \dots, k-1\}$ в масив *A* послідовно записати число *j C[j] раз*. Сортування закінчено!

Приклад 4.9 Приклад реалізації алгоритму сортування підрахунком на мові С

```
int sortCount (int *a,              /* вхідний масив, що сортується */
               int n)         { /* розмір масиву */


    int *c,          /* масив лічильників */
    i, j,     /* лічильники циклів */
    b,        /* допоміжний індекс для вхідного масиву */
    size;   /* розмір масиву лічильників */
    if (n < 1)
    return 1;

    // визначаємо розмір масиву лічильників
     size = a[0];
    for (i = 0; i < n; i++)
            if (size < a[i])
                    size = a[i];
    size++;       /* розмір масиву (тому, що рахуємо від нуля!)  */

    c = (int*) calloc(size, sizeof(int));       /* створюємо масив лічильників */
    if (!c)
            return 2;  // не вистачає пам'яті для роботи

    // заповнення масиву лічильників
    for (i = 0; i < n; i++)
            c[a[i]]++;

    // заповнення вхідного масиву відсортованими значеннями
    b = 0;
    for (j = 0; j < size; j++)
            for (i = 0; i < c[j]; i++) {
                    a[b] = j;
            b++;
    }

    free (c);
```

```
    return 0;
}
```

Якщо використовувана мова програмування не ініціалізує масив C при створенні так, щоб він містив всі нулі, алгоритм витратить на це додатково *size* кроків, після чого виконає ще N кроків, щоб порахувати значення з масиву *A*.

На завершення роботи алгоритм копіює значення назад у вихідний масив *A* (кожне один раз), тому дана частина процесу займе ще *N* кроків. Звідси загальний час роботи алгоритму складе *O(2*N + size) = O(N + size)*. Якщо size відносно малий в порівнянні з *N*, отримана продуктивність виявиться набагато краще *O(N*log(N))*, яку демонструють алгоритми розділу 4.3.

Алгоритм сортування підрахунком є стійким.

4.4.2 **Блокове (корзинне) сортування**

Алгоритм блокового, або, як його ще називають *корзинного сортування*, ділить елементи сортованого масиву на блоки так, щоб всі елементи в кожному наступному по порядку блоку були завжди більше (або менше), ніж у попередньому, а потім сортує кожен з блоків окремо з допомогою рекурсивного виклику самого себе і приєднує в кінці послідовно вміст відсортованих блоків до вихідного масиву.

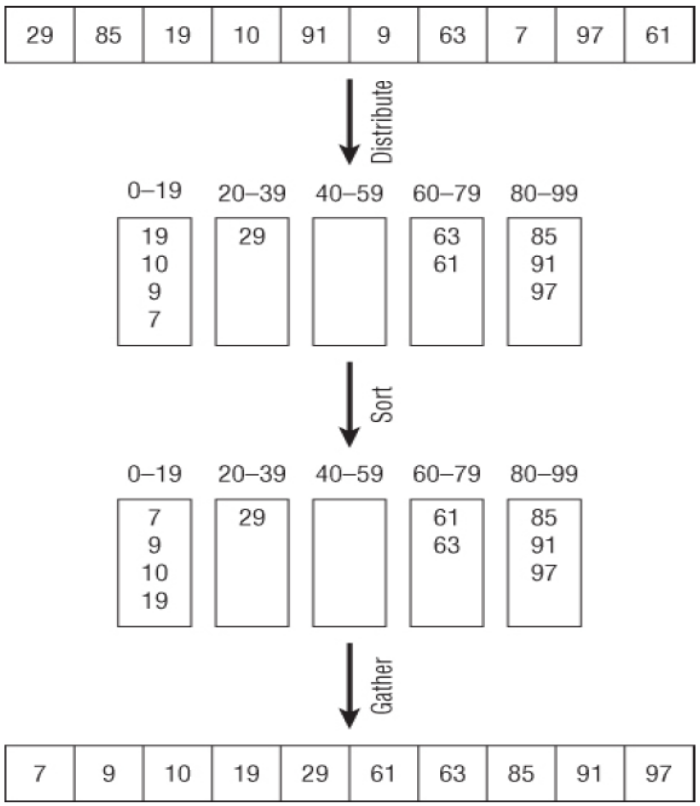


Рисунок 4.7 Етапи блокового сортування з цілими числами

Як приклад візьмемо масив, зображений на рисунку зверху. Він містить 10 елементів зі значеннями від 0 до 99. На першому етапі алгоритм ділить елементи на блоки, які в нашому випадку складаються з 20 значень: від 0 до 19, від 20 до 39, тощо. На другому етапі алгоритм сортує кожен блок , а на третьому об'єднує їх значення для побудови результату сортування.

Аналогічно можна зробити і під час сортування масивів чисел з плаваючою точкою:

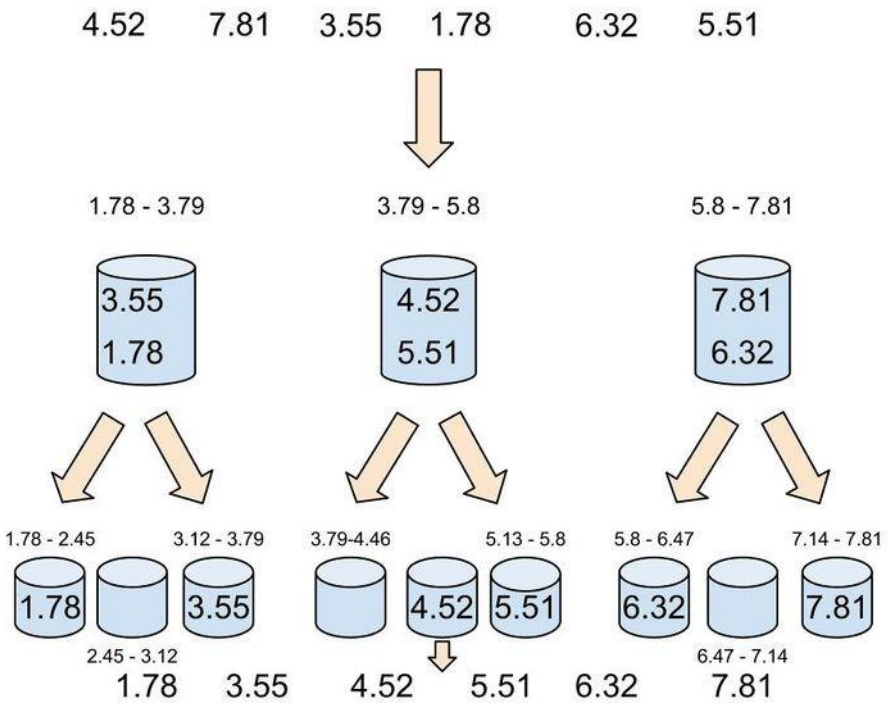


Рисунок 4.8 Етапи блокового сортування з плаваючими числами

Блоками можуть виступати стеки, зв'язні списки, черги, масиви або будь-які інші структури даних.

Якщо при використанні *M* блоків значення в масиві з *N* елементів розподілені відносно рівномірно і їх діапазон ділиться порівну, можна очікувати, що в кожен блок потрапить *N / M* елементів. Блокове сортування вимагає *O(M)* додаткової пам'яті і знання про природу даних, що сортуються, яке виходить за рамки функцій «переставити» і «порівняти». Складність алгоритму при цьому складе *O(N)* на вдалих вхідних даних.

У наступному псевдокоді представлений високорівневий алгоритм, який демонструє цей метод.

Приклад 4.10 Алгоритм блочного сортування

*int sortBucket (double *arr, int n)*

< Створюємо блоки.>

< Розподіляємо елементи по блоках.>

< Сортуємо блоки.>

< Збираємо елементи блоків в вихідний масив.>

End sortBucket

Щоб розбити *N* рівномірно розподілених елементів по блоках, алгоритму знадобиться *N* кроків, якщо не враховувати перенесення елемента в блок. Зазвичай таке перетворення займає фіксований відрізок часу. Припустимо, в масиві знаходяться цілі числа від 0 до 99, як в прикладі на **Ошибка! Источник ссылки не найден.**. Нам треба перенести елемент зі значенням *v* в блок номер *⌊v/20⌋*. Цей номер можна розрахувати за певний час, таким чином, для розподілу всіх елементів знадобиться *O(N)* кроків. Якщо таких блоків *M*, для сортування кожного з них знадобиться *F(N/M)* кроків, де *F* — функція часу роботи алгоритму, який ми використовуємо при сортуванні блоків. Виходить, що загальний час сортування всіх блоків складе *O(M*F(N/M))*.

Щоб зібрати всі відсортовані значення назад в масив, треба виконати знову *N* основних кроків. При підсумовуванні всіх етапів отримаємо наступне: *O(N) + O(M*F(N/M)) + O(N) = O(N + M*F(N/M))*. Якщо *M* є фіксованою часткою *N*, то *N/M*, а отже, і *F(N/M)* - константи. Тоді формула спрощується до *O(N + M)*.

На практиці число *M* має становити відносно велику частку *N*, щоб алгоритм міг добре працювати. Якщо у нас 10 млн записів і 10 блоків, то в кожному з них буде в середньому по 1 млн записів. На відміну від сортування підрахунком, продуктивність блокового сортування не залежить від кількості використовуваних блоків.

Недоліки алгоритму: сильно деградує при великій кількості мало відмінних один від одного елементів, тобто, якщо розподіл сортуємих елементів у вхідному наборі відмінен від рівномірного, або ж при виборі невдалої функції отримання номера блоку по вмісту елемента.

Алгоритм блокового сортування є стійким.

4.5 Використання функції *qsort()* мови С

У бібліотеці стандартних функцій мови С є функція *qsort()* (файл заголовків *stdlib.h*), призначена для сортування масивів елементів різної природи. Метод

сортування, використовуваний функцією - швидке сортування Хоара. Ця функція має абстрактний інтерфейс, який дозволяє використовувати її з сортованими масивами будь-якого типу.

Функція має наступний прототип:

```
void qsort (void *base,
            size_t nmemb,
            size_t size,
            int (*compar)(const void *, const void *));
```

Перший аргумент є покажчиком на початок сортованого масиву. Стандарт Сі допускає приведення покажчика на будь-який тип до типу покажчик на *void*. Це дозволяє першому фактичному аргументу функції *qsort()* посилатися на масив елементів будь-якого типу.

Другий аргумент являє собою кількість сортованих елементів в масиві *base*. Прототип перетворює це значення до типу *size_t***.** Тип *size_t* є цілочисельним типом даних, який повертається операцією *sizeof()* і який визначено в стандартних заголовних файлах.

Оскільки функція *qsort()* перетворює свій перший аргумент в покажчик типу *void*, вона втрачає інформацію про розмір кожного елемента масиву. Для компенсації цього слід явно вказати функції *qsort()* розмір об'єкта даних. Саме для цього послуговує *третій аргумент*. Наприклад, якщо виконується сортування масиву типу *double*, в якості третього аргументу використовується вираз *sizeof(double).*

Нарешті, *qsort()* для визначення порядку сортування вимагає передачі покажчика на *функцію порівняння*. Функція порівняння повинна приймати два аргументи: покажчики на два порівнюваних елемента. Вона повертає ціле число більше нуля, якщо перший елемент повинен слідувати за другим, нуль, якщо елементи однакові і цілочисельне значення, менше за нуль, якщо другий елемент повинен слідувати за першим. Функція *qsort()* використовує функцію порівняння, передаючи їй значення покажчиків, які обчислює на основі іншої інформації, що надається.

Формат функції порівняння задається останнім аргументом прототипу функції *qsort()*:

*int (*compar)(const void*, const void*)*

У прототипі визначено, що останній аргумент являє собою покажчик на функцію, яка повертає значення типу *int* і приймає два аргументи. Кожен з них є покажчиком на константний тип *const void*. Ці два покажчика посилаються на порівнювані елементи.

У прикладі нижче ілюструється метод визначення функції порівняння і застосування функції *qsort()*. Програма створює масив випадкових значень дійсних чисел, а потім сортує його:

Приклад 4.11 Застосування функції *qsort()*

```
#define _CRT_SECURE_NO_WARNINGS
#include "stdlib.h"
#include "stdio.h"
#include "conio.h"
#include "math.h"
#include "locale.h"
#include "string.h"
#include "windows.h"

void fillArray(double ar[], int n);
void showArray(const double ar[], int n);
int myComp(const void *p1, const void *p2);

int main(void) {
        double vals[] = { -10.5, 5, 1, 4, 3, 2, -9.9, 6, 7, 8 };

        SetConsoleCP(1251);
        SetConsoleOutputCP(1251);


        puts("Список рандомних чисел:");
        showArray(vals, sizeof(vals) / sizeof(double));
        qsort(vals, sizeof(vals) / sizeof(double), sizeof(double), myComp);
        puts("\n Відсортований список:");
        showArray(vals, sizeof(vals) / sizeof(double));
        return 0;
}


void showArray(const double ar[], int n) {
        int index;
        for (index = 0; index < n; index++) {
                printf("%10.4g ", ar[index]);
```

```
            if (index % 6 == 5)
                putchar('\n');
        }
        if (index % 6 != 0)
            putchar('\n');
        return;
}


/* реалізація функції порівняння за зростанням */
int myComp(const void * p1, const void * p2) {

        /* для доступу до значень потрібні покажчики на тип double */
        const double *a1 = (double*)p1;
        const double *a2 = (double*)p2;

        if (*a1 < *a2)
            return -1;
        else if (*a1 == *a2)
            return 0;
        else
            return 1;
}
```

Як вже говорилося, прототип *qsort()* задає інтерфейс функції порівняння:

*int (*compar)(const void *, const void *)*

В ньому вказано, що останній аргумент є покажчиком на функцію, яка повертає значення типу *int* і приймає два аргументи. Кожен з них являє собою покажчик на тип *const void*. Програма Приклад 4.11 містить прототип функції *myComp()*, який відповідає наступному:

*int myComp(const void *p1, const void *p2);*

Згадаймо з курсу програмування, що ім'я функції, коли використовується в якості аргументу, є покажчиком на неї. Тому функція *myComp* відповідає прототипу *compar*.

Функція *qsort()* передає функції порівняння адреси двох елементів, які необхідно порівняти. У нашій програмі адреси двох порівнюваних значень типу *double* присвоюються змінним *p1* і *p2*. Зверніть увагу, що перший аргумент функції *qsort()* посилається на масив в цілому, а два аргументи функції порівняння посилаються на два елементи цього масиву. Тут виникає проблема. Щоб виконати

порівняння вказуваних значень, необхідно розіменувати покажчик. Оскільки значення мають тип *double*, слід виконати розіменування покажчика на тип *double*. Однак функція *qsort()* вимагає наявності покажчиків на тип *void*. Вихід зі становища полягає в оголошенні покажчиків відповідного типу всередині функції *myComp()* з їх подальшою ініціалізацією значеннями, переданими в якості аргументів.

Коротше кажучи, *qsort()* і функція порівняння для узагальненості використовують покажчики на *void*. Внаслідок цього функції *qsort()* необхідно явно вказати розмір кожного елемента масиву, а при визначенні функції порівняння *myComp()* перетворити аргументи-покажчики в покажчики на тип даних, які функція повинна обробляти.

Покажчики на *void* в мовах C і C++ обробляються по-різному. Обидві мови дозволяють присвоїти покажчик на будь-який тип даних змінній типу *void* *. Наприклад, в Приклад 4.11 при виконанні функції *qsort()* значення типу *double** присвоюється покажчику на *void**. Однак синтаксис мови C++, додатково вимагає приведення типів, коли покажчик *void** присвоюється покажчику іншого типу. При цьому в C подібної вимоги немає. Наприклад, у функції *myComp()* з прикладу для покажчика *p1* типу *void** таке приведення типів реалізовано:

```
const double *a1 = (double*) p1;
```

У мові C таке приведення типів необов'язково, тоді як в мові C++ воно необхідно. Оскільки приведення типів може бути застосовано в обох мовах, має сенс виконувати його завжди. Це спростить адаптацію програми для мови C++.

4.6 Зовнішнє сортування

Зовнішнє сортування є різновидом методу сортування, який здатний обробляти великі і надвеликі обсяги даних [3]. Воно застосовується, коли дані, які потрібно відсортувати, не поміщаються цілком в оперативну пам'ять комп'ютера і, отже, в процесі сортування виникає необхідність використання більш повільної пам'яті (як правило, магнітний диск або навіть магнітна стрічка). Один з можливих алгоритмів зовнішнього сортування буде пояснено нижче.

Приклад 4.12 Алгоритм зовнішнього двофазного сортування злиттям

Нехай нам необхідно впорядкувати 700 МБ даних з використанням тільки 100 МБ оперативної пам'яті. Кроки для сортування наведені нижче.

- Крок 1: Зчитування 100 МБ даних в оперативну пам'ять і сортування цих даних, використовуючи будь-який звичайний алгоритм сортування з раніше вже вивчених.
- Крок 2: Запис відсортованих даних назад на носій.
- Крок 3: Повторити кроки 1 і 2, поки всі дані (в кожній з 100 Мб частини на носії) НЕ відсортуються. На цьому перша фаза сортування закінчується. Всі ці сім відсортованих частин, тепер необхідно об'єднати в один вихідний файл.
- Крок 4: Починається друга фаза сортування. Зчитують перші 10 МБ кожної відсортованої частини і називають їх вхідними буферами. Отже, тепер у нас є 70 МБ даних в оперативній пам'яті. Виділяється ще 30 МБ пам'яті, яка залишилася (зі 100), для буфера виводу.
- Крок 5: Виконується злиття даних з усіх вхідних буферів в один вихідний. Якщо в будь-який момент часу, вихідний буфер заповнюється, його вміст записується на диск (стрічку) у вигляді частини остаточно відсортованого файлу, а сам буфер очищається і готується до нового злиття. Якщо який-небудь з семи вхідних буферів стає порожнім, в нього зчитуються наступні 10 Мбайт пов'язаної з ним 100 МБ відсортованої раніше частини вхідного файлу. Якщо ж 100 МБ частина повністю вичерпана, відповідний вхідний буфер позначається як невикористовуваний і в подальшому злитті участі вже не бере. На цьому друга фаза сортування завершується.

На малюнку нижче проілюстровано цей алгоритм:

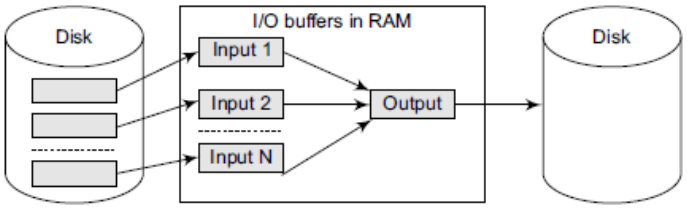


Рисунок 4.9 Ілюстрація алгоритму зовнішнього сортування двофазним злиттям

З наведеного вище прикладу, ми можемо тепер уявити узагальнений алгоритм сортування злиттям для зовнішнього сортування. Якщо обсяг даних, що підлягають сортуванню, перевищує обсяг доступної для роботи пам'яті в K раз, то розбиваємо вхідний масив даних, що сортується, на K частин. Ці частини відсортовуються окремо і зберігаються на носії, а потім над ними виконується

процедура злиття.

У наведеному прикладі був використаний один прохід злиття. Але якщо відношення сортованих даних і оперативної пам'яті особливо велике, використовується багатопрохідне сортування. Ми можемо спочатку об'єднати тільки першу половину відсортованих частин, а потім - другу половину, і, нарешті, зливаємо дві відсортовані половини разом, і т.д. Точне число проходів залежить від наступних факторів:

- величини коефіцієнта *K*;
- фізичних характеристик носія, такі як швидкість передачі даних, час доступу, і т.д.

4.6.1 Застосування зовнішнього сортування

Зовнішнє сортування часто використовується для оновлення файлів баз даних з файлу транзакцій. Наприклад, оновлення файлу СТУДЕНТИ на основі даних про нових студентів, рубіжному і поточному контролі знань, відрахуваннях.

Воно також використовується в застосуваннях баз даних для виконання таких операцій, як проекції і з'єднання. Проекція означає вибір підмножини полів зі структури записів, а з'єднання - об'єднує інформацію із записів двох файлів по деякому загальному для них полю, створюючи новий файл з новою структурою записів. Зовнішнє сортування також використовується для видалення дубльованих записів з наборів даних.

4.7 Контрольні завдання і запитання

1. Цикл *For i (*Приклад 4.4*)*, який використовується в алгоритмі, що сортує вставкою, переглядає індекси від 0 до останнього у масиві. Що станеться, якщо початковий індекс дорівнюватиме 1? Чи зміниться час роботи алгоритму?

2. Напишіть програму, яка реалізує швидке сортування на місці.

3. Для швидкого сортування найгіршим випадком є той, при якому на початку сортування елементи відсортовані у прямому або зворотному порядку, або містять багато дублікатів. Перших двох проблем можна позбутися, якщо вибрати випадкові опорні елементи. Як уникнути третьої?

4. Напишіть програму, яка реалізує алгоритм блочного сортування рекурсивно. Дозвольте користувачеві визначити кількість елементів, їх максимальне значення та кількість блоків.

5. Уявіть, що значення масиву *counts* знаходяться в діапазоні від 100 000 до 110 000, у ньому 110 001 запис та кількість елементів відносно не багата. Як модифікувати сортування підрахунком, щоб досягти хорошої продуктивності в цій ситуації?

6. Якщо масив містить N елементів, що охоплюють діапазон від 0 до M – 1, що станеться з блоковим сортуванням під час використання M блоків?

7. Який з алгоритмів сортування добре працюватиме, а який ні для наступних наборів даних:

- 10 значень із плаваючою комою;
- 1000 цілих чисел;
- 1000 імен;
- 100 000 цілих чисел зі значеннями від 0 до 1000;
- 100 000 цілих чисел із значеннями від 0 до 1 млрд;
- 100000 імен;
- 1 млн значень із плаваючою комою;
- 1 млн імен;
- 1 млн цілих чисел із рівномірним розподілом;
- 1 млн цілих чисел із нерівномірним розподілом?

5 ЗВ’ЯЗНІ СПИСКИ

5.1 Поняття зв'язного списку і його організація

Раніше, при вивченні такої структури зберігання інформації, як масиви, ми з'ясували, що масив являє собою одно- чи багатовимірну сукупність елементів даних одного типу, в яких елементи зберігаються в сусідніх зв'язаних областях пам'яті. Оголошуючи масив, ми повинні вказати розмір масиву, який буде задавати максимальну кількість збережених в ньому елементів. Наприклад, якщо ми оголошуємо масив *int marks [10]*, то такий масив може зберігати максимум до 10 елементів даних, але не більше того. Але що, якщо ми заздалегідь не знаємо число елементів? Звичайно і в цьому випадку, можна скористатися динамічно створюваними масивами з використанням доступу до них за допомогою покажчиків (як в п. 3.3), але тоді ми не зможемо довільно змінювати розмір такого масиву шляхом додавання нових і видалення елементів, які стали непотрібними, без необхідності пересувати з кожною зміною новий масив в нову область зв'язаної пам'яті в "купі". А якщо в якийсь момент часу вільної зв'язаної області пам'яті потрібного розміру в наявності в "купі" вже не виявиться, незважаючи на велику кількість областей меншого розміру (ефект фрагментації купи)? Для більш ефективного використання пам'яті в "купі", бажано зберігати елементи масиву в довільних місцях "купи", а не використовувати під них лише зв'язані області пам'яті. Таким чином, стає потрібною така структура даних, яка одночасно знімає обмеження на максимальну кількість збережених елементів в одному об'єкті і умов зберігання, для того щоб створювані програми виходили більш ефективними.

Зв'язний список якраз і являє собою структуру даних, яка вільна від вищезгаданих обмежень. Зв'язний список не зберігає свої елементи в послідовних комірках пам'яті, і користувач може як додати будь-яку кількість нових елементів до нього в довільний момент, так і видалити елементи, які стали непотрібними зі списку без необхідності виділення нових зв'язних областей в "купі" і перезапису списку на нове місце. Проте, на відміну від масиву, зв'язний список не допускає **прямого (випадкового)** доступу до даних. Елементи в зв'язному списку можуть бути доступні тільки в послідовному порядку. Але на відміну від масиву, вставка і видалення елемента списку може бути виконана в будь-який момент в реальному

часі.

Зв'язаний список, являє собою лінійну сукупність елементів даних, які називаються вузлами і так званий дескриптор (або покажчик на початок) списку [1]. Кожен з вузлів містить покажчик (адресу) наступного вузла в списку. Останній вузол у списку містить *NULL*-покажчик, який і сигналізує про кінець списку.

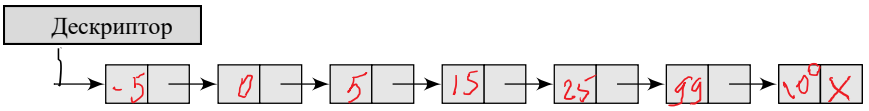


Рисунок 5.1 Структура зв'язного списку

Фізично список реалізується як сукупність дескриптора списку і елементів, розміщених в оперативній пам'яті або на зовнішніх запам'ятовуючих пристроях (ЗЗП) і зв'язаних один з одним у ланцюжок за допомогою покажчиків. Дескриптор зазвичай реалізується у вигляді структури і може містити наступну інформацію про список:

- тип структури;
- назву списку;
- покажчик (адреса) початку списку;
- поточне число елементів;
- опис елемента (довжина, тип, тощо).

Замість дескриптора, при можливості, можна використовувати тільки покажчик.

Приклад 5.1 Типовий дескриптор лінійного односпрямованого списку

```
typedef struct pNode {  // опис вузла списку
        char data[12];
        pNode* next;
};

struct dsList {                 // опис дескриптора списку
        pNode* start;
        int sizeList;
}
```

Зв'язаний список є абстрактною структурою даних, яка в свою чергу, часто використовується для реалізації інших, більш складних абстрактних структур даних. Таким чином, він виступає в якості будівельного блоку для реалізації структур даних, таких як стеки, черги і їх варіації. Зв'язний список може сприйматися як поїзд або послідовність вузлів, в яких кожен вузол містить один

або кілька полів даних і покажчик на наступний вузол.

На Рисунок 5.1, ми можемо бачити зв'язний список, в якому кожен вузол містить дві частини: ціле число і покажчик на наступний вузол. Ліва частина вузла, яка і містить дані, може включати в себе простий тип даних, масив або структуру. Права частина вузла містить покажчик на наступний вузол (або адресу наступного вузла в послідовності). Останній вузол не матиме за собою ніякого наступного вузла, так що він буде зберігати спеціальне значення, зване *NULL*. На Рисунок 5.1 нульовий покажчик представлений як X. При програмуванні, ми зазвичай визначаємо *NULL* як *0*. Отже, *NULL* покажчик вказує на кінець списку. Так як у зв'язаному списку кожен вузол містить покажчик на інший вузол, який має той же тип, він відноситься до рекурсивних типів даних.

Зв'язні списки містять дескриптор, який зберігає адресу першого вузла в списку. Ми можемо обійти весь список за допомогою дескриптора, який містить адресу першого вузла; адресна частина першого вузла в свою чергу, зберігає адресу наступного за ним вузла. Використовуючи цю техніку, окремі вузли списку утворюють ланцюжок вузлів. Якщо дескриптор в якості адреси першого вузла містить *NULL*, то зв'язний список порожній і не містить вузлів.

В мові програмування C, ми можемо реалізувати вузол зв'язного списку з прикладу вище, використовуючи наступний тип:

```
struct node {
        int data;
        struct node *next;
};
```

Тепер подивимося, як зв'язний список зберігається в пам'яті. Для того, щоб сформувати зв'язний список, нам потрібна структура під назвою вузол, який має два поля, *data* і *next*. *data* буде зберігати інформаційну частину, а *next* буде зберігати адресу наступного вузла в послідовності.

У цьому прикладі (Рисунок 5.2), *START = 1*, так що перший вузол списку зберігається за адресою 1. Перший вузол містить дані *data = 'H'*, які є символ *'H'*. Відповідний *next* зберігає адресу наступного вузла, що дорівнює 4. Таким чином, ми будемо бачити адресу 4, щоб витягти наступний елемент даних, тощо. Ми повторюємо цю процедуру до тих пір, поки не досягнемо позиції, де наступний вузол в поле *next* містити *NULL*, так як це означає кінець зв'язного списку. Коли ми

закінчимо перегляд зв'язного списку, ми побачимо, що наш список в наведеному вище прикладі зберігає символи, які, якщо скласти їх разом, утворюють слово *HELLO*.

Зверніть увагу, що на Рисунок 5.2 показується блок пам'яті, з адресами від 1 до 10. Заштрихована частина містить дані інших застосувань. Пам'ятаємо, що вузли зв'язного списку не обов'язково повинні бути в зв'язаних комірках пам'яті. У нашому прикладі вузли для зв'язного списку, зберігаються за адресами 1, 4, 7, 8 і 10.

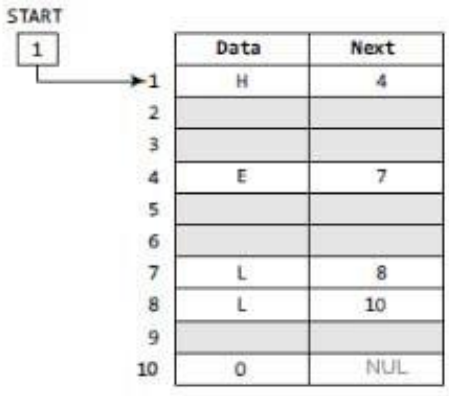


Рисунок 5.2 Дескриптор START вказує перший елемент зв'язного списку в пам'яті

Візьмемо інший приклад, щоб побачити, як два зв'язних списки зберігаються разом в пам'яті комп'ютера. Наприклад, учні школи хочуть вибрати вивчення предметів: біологію або інформатику. Тепер у нас буде два зв'язаних списки, по одному для кожного предмета. Кожен із списків буде містити номери всіх учнів, які вибрали для вивчення відповідний предмет.

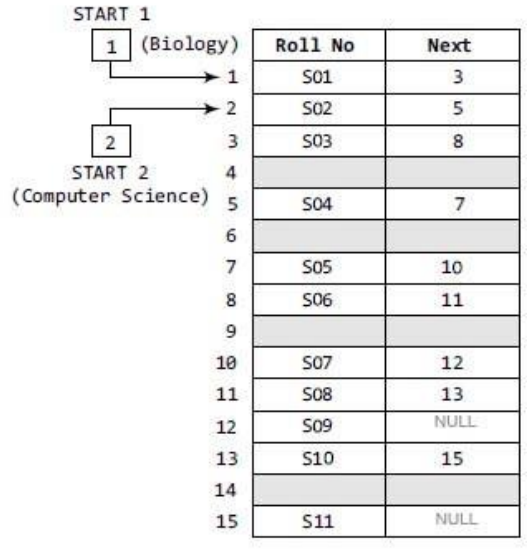


Рисунок 5.3 Два зв'язних списки одночасно розташовані в одній області пам'яті

5.2 Односпрямовані списки

Односпрямований список є найпростішим типом зв'язного списку, в якому кожен вузол містить деякі дані і покажчик на наступний вузол того ж списку (Рисунок 5.1). Говорячи про те, що вузол містить покажчик на наступний, ми маємо на увазі, що вузол зберігає адресу наступного вузла в послідовності. Односпрямований список дозволяє виконувати обхід даних тільки в один бік.

5.2.1 Навігація в списку

Навігація в зв'язному списку означає доступ до вузлів списку, щоб виконати деяку обробку інформації з цих вузлів. Зв'язний список завжди містить дескриптор, який зберігає адресу першого вузла списку. Кінець списку позначається *NULL* або *0* в поле *Next* останнього вузла. Для обходу зв'язного списку, будемо використовувати ще одну покажчикову змінну *ptr*, яка показує на вузол списку, доступний для обробки в даний момент. Алгоритм для обходу зв'язного списку показаний в наступному прикладі.

Приклад 5.2

```
ptr = descriptor.start;      /* ініціалізація */
while (ptr != NULL) {    /* обхід  списку */
        ApplyProcess (ptr->data);        /* обробка вузла */
        ptr = ptr -> next;   /* перевстановлення поточного покажчика вузла на
                                новий вузол */
}
```

У цьому алгоритмі ми спочатку ініціалізуємо *ptr* початковою адресою, що зберігається в дескрипторі списку. Так що тепер, *ptr* вказує на перший вузол зв'язного списку. Потім переходимо до виконання циклу обходу списку, який повторюється до тих. пір, поки ми не зустрінемо останній вузол (в якому значення *ptr = NULL*). Всередині циклу обходу ми застосовуємо процес обробки вузла (наприклад, друк) до поточного вузла, тобто до вузла, на який вказує *ptr*. І далі, ми переходимо до наступного вузла списку, присвоївши змінній *ptr* адресу цього вузла, яка зберігається в полі *next* поточного вузла.

Напишемо тепер програму для підрахунку і друку числа вузлів в односпрямованому списку.

Приклад 5.3 Підрахунок числа вузлів у списку

```
count = 0;
ptr = descriptor.start;      /* ініціалізація */
```

```
while (ptr != NULL) {      /* обхід списку */
        count++;     /* обробка вузла */
        ptr = ptr -> next;   /* перевстановлення поточного покажчика вузла на
                                наступний вузол */
}

printf ("Число вузлів списку = %d", count);
```

5.2.2 Пошук у списку

Пошук в зв'язному списку означає знаходження вузла з наперед заданими властивостями / ознаками. Як вже говорилося, зв'язний список складається з вузлів, які розділені на дві частини: інформаційну і адресну. Таким чином, пошук призначений для виявлення того, чи присутнє в інформаційній частині вузлів списку задане значення, чи ні. Якщо воно присутнє, то алгоритм повертає адресу вузла, який містить таке значення.

Приклад 5.4 Алгоритм пошуку в односпрямованому списку

```
pos= NULL;/* адреса вузла списку з заданим значення */

ptr = descriptor.start;              /* ініціалізація */

while (ptr != NULL)                  /* обхід списку */
        if (val == ptr -> data) {    /* знайшли потрібний вузол? */
                pos = ptr;
                break;
        }
        else
                ptr = ptr -> next;   /* перевстановлення поточного покажчика
                                        вузла на новий вузол */
```

Перед початком пошуку додатково ініціалізуєм покажчикову змінну *pos*, яка буде після закінчення пошуку містити адресу знайденого вузла з шуканим значенням. Потім, при виконанні обходу циклу, порівнюємо дані кожного вузла зі змінною *val*, яка містить необхідне нам *значення*. Якщо пошук успішний, тобто *значення* було знайдено, то адресу цього вузла зберігаємо в *pos* і виходимо з циклу обходу. Однак, якщо пошук буде безуспішним, в *pos* залишиться встановлене нами значення *NULL*, яке буде вказувати на те, що в даному списку потрібне нам *значення* відсутнє. Робота даного алгоритму проілюстрована на Рисунок 5.4. Тут ми шукаємо вузол списку, що містить значення *val = 4*.

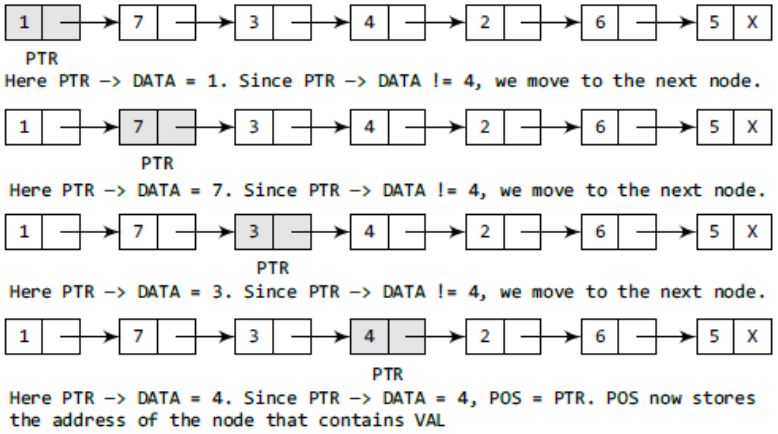


Рисунок 5.4 Робота алгоритму пошуку в односпрямованому списку

5.2.3 Вставка нового вузла в односпрямований список

Покажемо, як новий вузол додається у вже існуючий зв'язний список. Ми розглянемо чотири випадки такої вставки.

Випадок 1: Новий вузол вставляється **у верхню частину (початок)** списку.

Випадок 2: Новий вузол вставляється **в кінець списку**.

Випадок 3: Новий вузол вставляється **після заданого вузла**.

Випадок 4: Новий вузол вставляється **перед заданим вузлом**.

Перед тим, як описувати алгоритми для виконання операцій вставки у всіх цих чотирьох випадках, давайте спочатку обговоримо важливий термін, який називається переповненням. Переповнення є станом, який виникає, коли в "купі" буде відсутня вільна пам'ять, що виділяється при створенні нового вузла. При виникненні переповнення "купи" програма повинна видати відповідне повідомлення.

Вставка вузла на початку списку. Розглянемо зв'язний список, показаний на Рисунок 5.5. Припустимо, що ми хочемо додати новий вузол із значенням 9 в якості першого вузла списку. Тоді такі зміни повинні будуть здійснені наступними послідовними кроками:

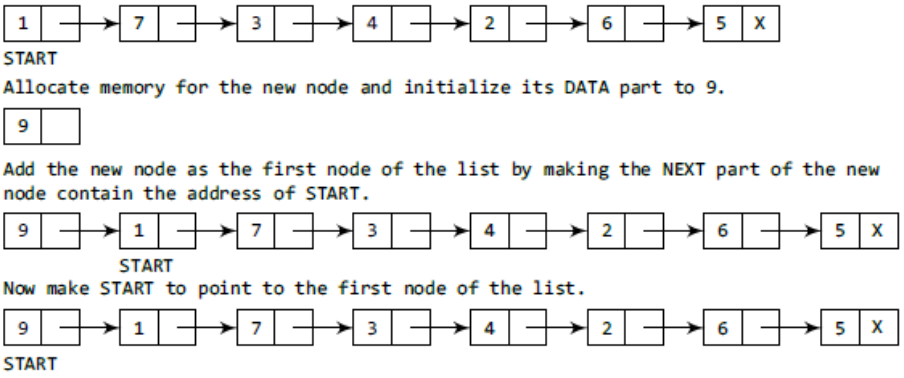


Рисунок 5.5 Вставка вузла на початку списку

В Приклад 5.5 Алгоритм вставки нового вузла на початку списку показаний відповідний алгоритм для вставки нового вузла на початок списку. На початку ми перевіряємо, чи доступний нам потрібний обсяг пам'яті в "купі" для нового вузла списку. Якщо вільна пам'ять вичерпана, закінчуємо виконання алгоритму з видачею відповідного повідомлення у функцію, що викликає. В іншому випадку, якщо вільна пам'ять доступна, то виділяємо місце для нового вузла. Встановлюємо його поле *data* у відповідність зі значенням змінної *val* і ініціалізуємо поле *next* нового вузла адресою першого вузла списку, який зберігався в дескрипторі списку. У дескрипторі списку змінюємо адресу *start* на адресу новоствореного вузла списку. Тепер, дескриптор списку буде показувати на новостворений вузол в якості першого вузла списку.

Приклад 5.5 Алгоритм вставки нового вузла на початку списку

```
NewNode = (<TypeNode>*) malloc (sizeof(<TypeNode>)); /* виділення пам'яті
                                                        під новий вузол */

if (!NewNode)       /* якщо пам'яті в "купі" немає */
        return -1;  /* завершення алгоритму з видачею відповідного
                       повідомлення до функції, що викликала */
else {
        NewNode -> next = descriptor.start; /* встановлюємо адресу наступного
                                               вузла */
        NewNode -> data = val;          /* присвоюємо значення інформаційній
                                           частині */
        descriptor.start = NewNode; /* перевстановлюємо дескриптор на новий
                                       вузол */
}

return 0;       /* нормальне завершення функції додавання вузла */
```

Вставка вузла в кінець списку. Припустимо, ми хочемо додати новий вузол зі значенням 9 в кінець списку. Тоді в даному списку слід виконати такі зміни:

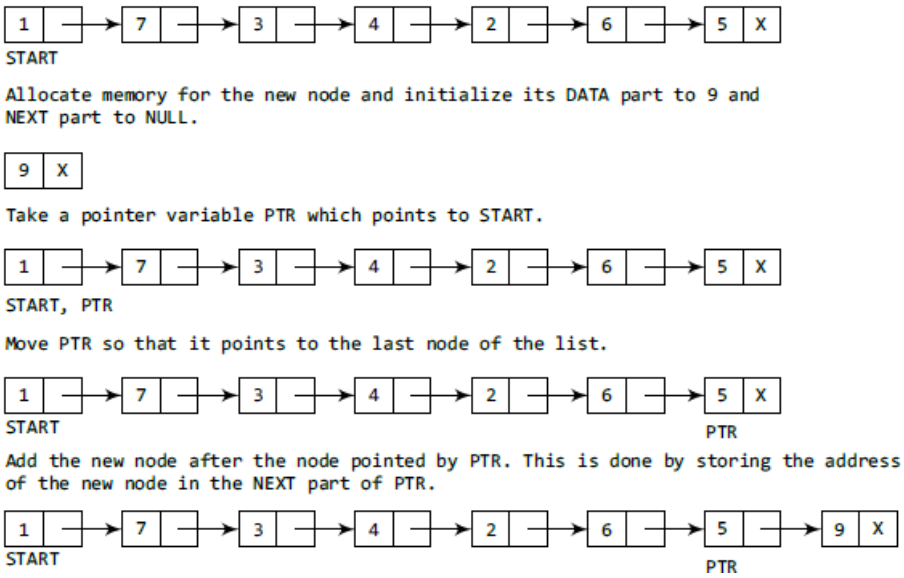


Рисунок 5.6 Вставка вузла в кінець списку

В Приклад 5.6 наведено алгоритм вставки нового вузла в кінці зв'язного списку. На початку ми знову перевіряємо, чи доступний обсяг пам'яті в "купі" для нового вузла списку. У разі відсутності пам'яті - завершуємо алгоритм. При наявності вільної пам'яті виконуємо алгоритм далі. Полю *next* новоствореного вузла слід присвоїти значення *NULL*, з тим, щоб вузол став термінальним, а інформаційному полю *data* присвоїти значення *val*. Далі беремо змінну-покажчик *ptr* і ініціалізуємо її адресою початку списку з дескриптора. Тобто, *ptr* тепер вказує на перший вузол зв'яззного списку. На наступному етапі проходимо в циклі по всіх вузлах списку, поки не досягнемо останнього вузла (значення *NULL* в полі *next*). Після того, як ми досягнемо останнього вузла, на останньому кроці ми змінюємо поле *next* останнього вузла таким чином, щоб воно показувало на новий вузол.

Приклад 5.6 Алгоритм додавання вузла в кінець списку

```
NewNode = (<TypeNode>*) malloc (sizeof(<TypeNode>)); /* виділення пам'яті
                                                        під новий вузол */

if (!NewNode)       /* якщо пам'яті в "купі" немає */
      return -1;    /* завершення алгоритму з видачею відповідного
                       повідомлення в функцію, що викликає */
NewNode -> next = NULL; /* встановлюємо ознаку кінця списку */
NewNode -> data = val;  /* присвоюємо значення інформаційної
                           частини */

/* ініціалізуємо початок обходу списку */
ptr = descriptor.start;
if (!ptr) {
   descriptor.start = NewNode;
   return 0;
   }
```

```
/* проходимо список до кінця */
while (ptr->next != NULL)
        ptr = ptr -> next;
ptr -> next = NewNode;  /* останній вузол в списку перевстановлюємо на
                                вузол,  що додається */
return 0;      /* нормальне завершення функції додавання вузла */
```

Вставка нового вузла після заданого. Розглянемо зв'язний список, показаний на Рисунок 5.7. Припустимо, що ми хочемо додати новий вузол із значенням 9 після вузла зі значенням 3.

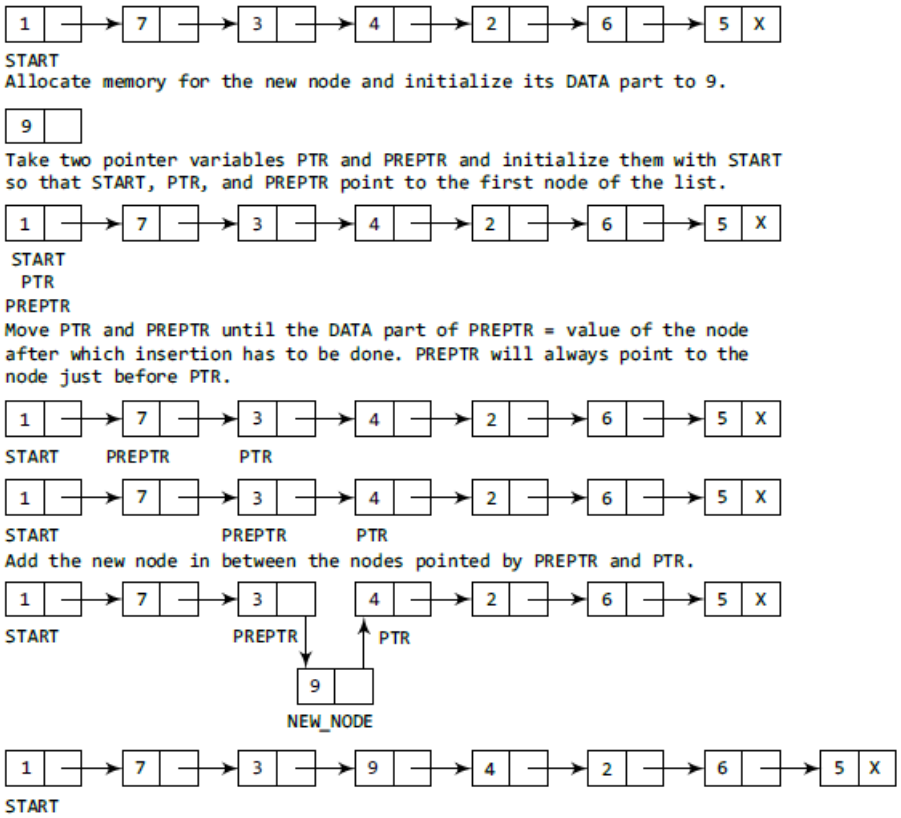


Рисунок 5.7 Вставка вузла в список ПІСЛЯ вузла з заданим значенням

Подивимося на алгоритм, показаний в Приклад 5.7. Після виконання попередніх кроків (створення і перевірки нового вузла), ми візьмемо змінну-покажчик *ptr* і проініціалізуємо її адресою початку списку з дескриптора. Тобто, *ptr* тепер вказує на перший вузол зв'язного списку. Далі беремо іншу змінну *PrePtr* - покажчик, який будемо використовувати для зберігання адреси вузла, що передує вузлу, на який вказує *ptr*. Спочатку *PrePtr* ініціалізуємо тією ж адресою, що і *ptr*. Так що тепер, *ptr*, *PrePtr* і *descriptor.start* - всі вказують на перший вузол зв'язного списку.

Здійснюємо в списку пошук вузла зі значенням інформаційної частини рівної змінній *num* (в нашому випадку *num = 3*). Нам необхідно знайти вузол з таким

значенням інформаційної частини, оскільки новий вузол повинен бути вставлений після нього. Після того, як ми досягнемо цього вузла, на наступних етапах, ми змінимо покажчики *next* знайденого в списку і додамо вузол таким чином, щоб новий вузол вставлявся після потрібного вузла. У разі, якщо вузол в списку з заданим значенням *num* не буде знайдений - припиняємо виконання алгоритму з видачею відповідного повідомлення і звільняємо раніше виділену пам'ять під новий вузол.

Приклад 5.7 Алгоритм додавання нового вузла ПІСЛЯ заданого

```
if (!descriptor.start)        /* якщо список пустий */
        return -2;     /* завершення алгоритму з видачею відповідного
                          повідомлення у функцію, що викликала */
NewNode = (<TypeNode>*) malloc (sizeof(<TypeNode>)); /* виділення пам'яті
                                                        під новий вузол */
if (!NewNode)        /* якщо пам'яті в "купі" немає */
        return -1;     /* завершення алгоритму з видачею відповідного
                          повідомлення у функцію, що викликала */

NewNode -> data = val;  /* присвоюємо значення інформаційній частині */
ptr = descriptor.start;     /* ініціалізуємо початок обходу списку */
PrePtr = ptr;

/*Прохід по списку до його кінця*/
do {
        PrePtr = Ptr;
        ptr = ptr -> next;   /* просуваємо покажчик на наступний вузол */
} while ((PrePtr -> data != num) && ptr);/* шукаємо в списку вузол із
                                            значенням num поки не кінець
                                            списку */

/* якщо значення num в списку немає */
if (!ptr && (PrePtr -> data != num)) {
        free (NewNode);    /* перед виходом - поверни пам'ять! */
        return -3; /* завершення алгоритму з видачею відповідного
                      повідомлення у функцію, що викликала */
}

/* додаємо новий елемент списку після заданого num */
PrePtr -> next = NewNode;  /* розриваємо список для вставляння нового
                              елемента в кінець "лівої половини" списку */
NewNode -> next = ptr;  /* дописуємо адресу початку "правої половини"
                           списка у новий елемент */
return 0;      /* нормальне завершення функції додавання вузла */
```

Вставка нового вузла перед заданим. Розглянемо зв'язний список, показаний на рисунку нижче. Припустимо, що ми хочемо додати до нього новий

вузол із значенням 9 ПЕРЕД вузлом, що містить значення 3.

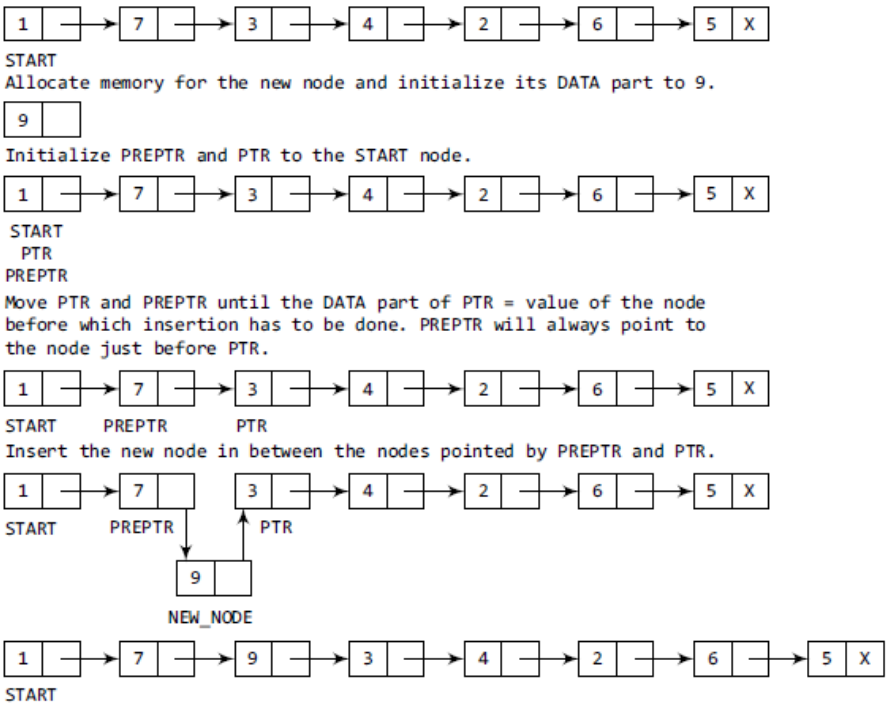


Рисунок 5.8 Вставка нового вузла у список ПЕРЕД вузлом, що містить задане значення

Алгоритм буде дещо відрізнятися від попереднього алгоритму.

Приклад 5.8 Алгоритм додавання нового вузла ПЕРЕД заданим

```
if (!descriptor.start)        /* якщо список пустий */
        return -2;     /* завершення алгоритму з видачею відповідного
                          повідомлення у функцію, що викликала */
NewNode = (<TypeNode>*) malloc (sizeof(<TypeNode>)); /* виділення пам'яті
                                                        під новий  вузол */
if (!NewNode)         /* якщо пам'яті в "купі" немає */
        return -1;     /* завершення алгоритму з видачею відповідного
                          повідомлення у функцію, що викликала */


/* Ініціалізація перед початком роботи */
NewNode -> data = val;  /* присвоюємо значення інформаційній
                           частині зновствореного вузла */
ptr = descriptor.start;      /* ініціалізуємо початок обходу списку */
PrePtr = ptr;


/* Проходження по списку */
do {
        PrePtr = Ptr;
        ptr = ptr -> next;   /* просуваємо покажчик на наступний вузол */
        /******* Відмінності від алгоритму з Приклад 5.7 **********/
        if (!ptr) break;       /* якщо досягнуто кінець списку, а значення num
                                  так і не знайдено */
```

```
} while (ptr -> data != num);    /* шукаємо в списку вузол із значенням пит */

/* якщо значення пит в списку немає */
if (!ptr && (ptr -> data != num)) {
        free (NewNode);    /* перед виходом - поверни пам'ять! */
        return -3;     /* завершення алгоритму з видачею відповідного
                          повідомлення у функцію, що викликала */
}


/*  Вставка нового елемента перед потрібним вузлом num  */
PrePtr -> next = NewNode;  /* розриваємо список для вставляння нового
                              елемента і в кінець "лівої половини" дописуємо
                              адресу нового елемента */
NewNode -> next = ptr;  /* дописуємо адресу початку "правої половини"
                           списка в новий елемент */
return 0;      /* нормальне завершення функції додавання вузла */
```

5.2.4 Видалення вузла з односпрямованого списку

Тут ми обговоримо, як вузол видаляється з уже існуючого зв'язного списку. Ми розглянемо три випадки, і подивимося, як виконується видалення у кожному конкретному випадку.

Випадок 1: Видаляється **перший вузол списку**.

Випадок 2: Видаляється **останній вузол списку**.

Випадок 3: Видаляється **вузол після заданого вузла**.

Перед тим, як описувати алгоритми цих трьох випадків, визначимо новий важливий термін під назвою **спустошення списку** (*Underflow List*). Спустошення виникає тоді, коли ми намагаємося видалити вузол із зв'язного списку, який **вже** порожній. Це відбувається, коли поле дескриптора списку *descriptor.start = NULL* або коли немає більше вузлів для видалення. Слід пам'ятати про те, що, коли ми видаляємо вузол із зв'язного списку, ми насправді повинні звільнити пам'ять, зайняту цим вузлом і повернути її в пул вільних адрес "купи" для використання в подальшому. В іншому випадку у нас почнуть виникати "витоки" пам'яті, що може привести до аварійного завершення програми в зв'язку з нестачею доступної пам'яті в "купі".

Видаляється перший вузол списку. Коли ми хочемо видалити вузол на початку списку, то слід виконати зміни в зв'язному списку так, як показано на малюнку нижче:

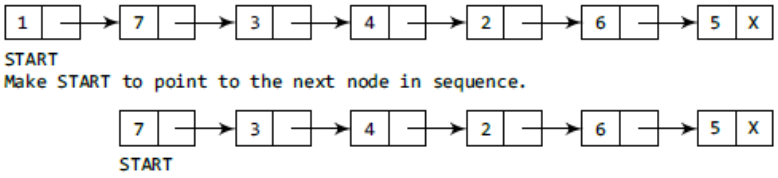


Рисунок 5.9 Видалення першого вузла односпрямованого списку

В прикладі нижче наведено алгоритм подібного видалення. На першому кроці ми перевіряємо, чи існує зв'язний список, чи ні. Якщо *descriptor.start = NULL*, то це означає, що список і так вже порожній, видаляти нічого й управління передається до функції, що викликала.

Приклад 5.9 Алгоритм видалення першого вузла з односпрямованого списку

```
if (descriptor.start == NULL)  /* якщо список пустий */
        return -2;     /* завершення алгоритму з видачею відповідного
                          повідомлення у функцію, що викликала */

else {
        ptr = descriptor.start;      /* зберігаємо адресу
                                        видаляємого вузла */
         descriptor.start = descriptor.start -> next;/* "обрізаємо" список зліва*/
        free (ptr);     /* повертаємо пам'ять в пул "купи " */
}

return 0;      /* нормальне завершення функції видалення вузла */
```

Якщо список ще не порожній, то ми використовуємо змінну-покажчик *ptr*, яку встановлюємо на перший вузол списку. На наступному кроці *descriptor.start* присвоюється адреса наступного вузла в послідовності, тим самим поле вказує на новий перший елемент списку. І нарешті, звільняємо пам'ять, зайняту вже видаленим вузлом.

Видаляється останній вузол списку. Припустимо, ми хочемо видалити останній вузол із зв'язного списку. В цьому випадку необхідно буде виконати зміни, наведені на рисунку нижче:

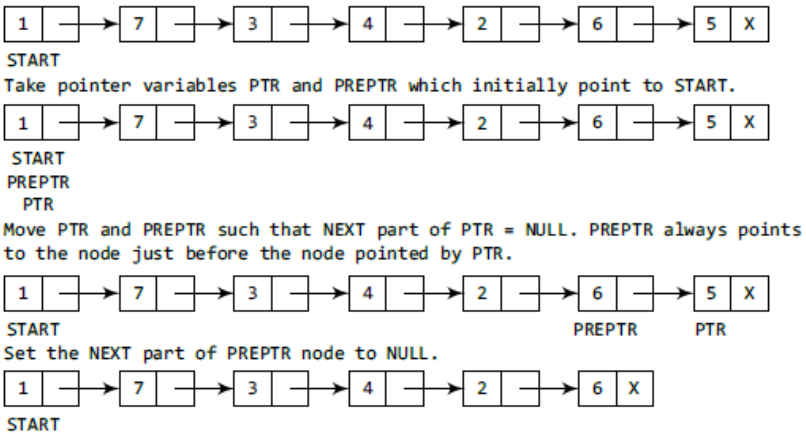


Рисунок 5.10 Видалення останнього вузла односпрямованого списку

В Прикладі 6.9 наведено алгоритм для видалення останнього вузла зі зв'язного списку. На другому кроці ми візьмемо змінну-покажчик *ptr* і ініціалізуємо її значенням *descriptor.start*. Тобто, *ptr* тепер вказує на перший вузол зв'язного списку. Далі беремо іншу змінну *PrePtr* - покажчик, який будемо використовувати для зберігання адреси вузла, що передує вузлу, на який вказує *ptr*. Спочатку *PrePtr* ініціалізуємо тією ж адресою, що і *ptr*. Так що тепер, *ptr*, *PrePtr* і *descriptor.start* - всі вказують на перший вузол зв'язного списку.

Потім приступаємо до проходження списку з метою пошуку його кінця. Як тільки *ptr -> next = NULL* - ми досягли останнього вузла списку, який і потрібно видалити. При цьому, покажчик *PrePtr* буде містити адресу попереднього вузла, який стане останнім. Пам'ять під об'єкт, на який вказує *ptr*, повертаємо в пул "купи".

Приклад 5.10 Алгоритм видалення останнього вузла з односпрямованого списку

```
if (descriptor.start == NULL)  /* якщо список пустий */
        return -2;     /* завершення алгоритму з видачею відповідного
                          повідомлення у функцію, що викликала */
ptr = descriptor.start;    /* ініціалізуємо поточні покажчики для
                              початку обходу списку */
PrePtr = ptr;

/* обходимо список до самого кінця */
do {
        PrePtr = Ptr;
        ptr = ptr -> next;   /* просуваємо покажчик на наступний вузол */
} while (ptr -> next != NULL);  /* шукаємо кінець списку */

PrePtr -> next = NULL;  /* "Обрізаємо" список */
free (ptr);     /* повертаємо пам'ять в пул "купи"*/
```

```
return 0;      /* нормальне завершення функції видалення вузла */
}
```

Видалення вузла після заданого. Припустимо, ми хочемо видалити вузол в списку після вузла, що містить значення в інформаційному полі рівне 4. Тоді в зв'язному списку необхідно буде виконати наступні зміни, наведені на рисунку нижче:

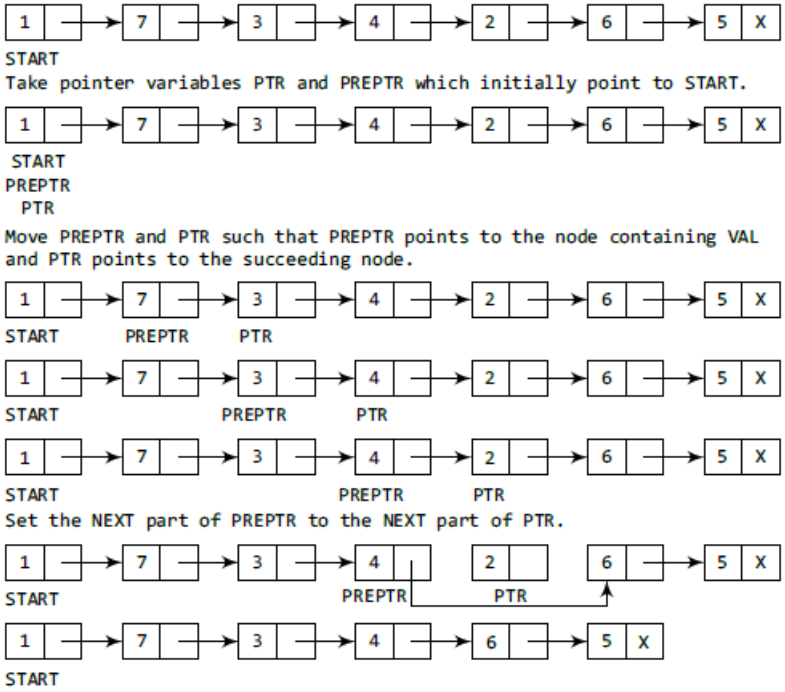


Рисунок 5.11 Видалення вузла в односпрямованому списку після заданого

В прикладі нижче наведено алгоритм для видалення вузла після заданого в зв'язному списку.

Приклад 5.11 Алгоритм видалення вузла в списку після заданого

```
if (!descriptor.start)        /* якщо список пустий */
        return -2;     /* завершення алгоритму з видачею відповідного
                          повідомлення у функцію, що викликала */
ptr = descriptor.start;      /* ініціалізуємо поточні покажчики
                                для початку обходу списку */
PrePtr = ptr;

/* обходимо список у пошуках заданого вузла */
do {
        PrePtr = Ptr;
        ptr = ptr -> next;   /* просуваємо поточний покажчик на наступний
                                вузол */
} while ((PrePtr -> data != num) && ptr);/* шукаємо в списку вузол із
                                            значенням num поки не кінець
                                            списку */
```

```
/* якщо значення num в списку немає */
if (!ptr && (PrePtr -> data != num)) {
        return -3;     /* завершення алгоритму з видачею відповідного
                          повідомлення у функцію, що викликала */
}

/* Видаляємо знайдений вузол */
PrePtr -> next = ptr -> next;
free (ptr);     /* повертаємо пам'ять видаленого вузла в пул
                   "купи"*/
return 0;       /* нормальне завершення функції додавання вузла */
```

Після виконання попереднього кроку перевірки існування списку ми візьмемо змінну-покажчик *ptr* і проініціалізуємо її адресою початку списку з дескриптора. Тобто, *ptr* тепер вказує на перший вузол зв'язного списку. Далі беремо іншу змінну *PrePtr* - покажчик, який будемо використовувати для зберігання адреси вузла, що передує вузлу, на який вказує *ptr*. Спочатку *PrePtr* ініціалізується тією ж адресою, що і *ptr*. Так що тепер, *ptr, PrePtr* і *descriptor.start* - всі вказують на перший вузол зв'язного списку.

Здійснюємо в списку пошук вузла зі значенням інформаційної частини рівної змінній *num* (в нашому випадку *num = 4*). Нам необхідно знайти вузол з таким значенням інформаційної частини, оскільки наступний після нього вузол списку повинен бути видаленим. Після того, як ми досягнемо цього вузла, на наступних етапах, ми змінимо покажчик *next* вузла *PrePtr* таким чином, щоб він вказував на наступний вузол після видаляємого. У разі, якщо вузол в списку з заданим значенням *num* знайдений не буде - припиняємо виконання алгоритму з видачею відповідного повідомлення.

На завершення, звільняємо пам'ять, займану видаленим нами вузлом.

5.3 Двоспрямовані списки

Двоспрямований список є більш складним типом зв'язного списку, який містить в кожному вузлі два покажчика: покажчик на наступний вузол і покажчик на попередній вузол в ланцюжку вузлів. Таким чином, вузол складається щонайменше з трьох частин: інформаційної, покажчика на наступний і покажчика на попередній вузол:

Приклад 5.12 Структура вузла двоспрямованого списку

```
struct node {
        int data;
        struct node *next;
        struct node *prev;
};
```

Двоспрямований список виглядає наступним чином:

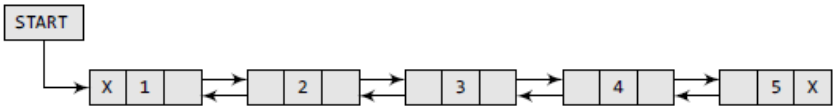


Рисунок 5.12 Вигляд двоспрямованого списку

Поле *prev* першого вузла і поле *next* останнього вузла будуть містити *NULL*. Поле *prev* використовується для зберігання адреси попереднього вузла, що дозволяє реалізувати перегляд списку в зворотному напрямку. Ми бачимо, що двоспрямований список вимагає більше місця в пам'яті під вузол і більш складних базових операцій маніпулювання списком. Проте, двоспрямований список забезпечує легкість маніпулювання елементами списку, оскільки він підтримує покажчики на вузли в обох напрямках (вперед і назад). Наявність двох покажчиків замість одного надає кілька переваг. Найбільш важлива з них полягає в тому, що переміщення по списку стає можливим в обох напрямках. Це спрощує роботу зі списком, зокрема, вставку і видалення вузлів. Крім цього, користувач може переглядати список в будь-якому напрямку. Ще одна перевага має значення тільки при деяких несправностях. Оскільки весь список можна пройти не лише за прямим, але і за зворотним посиланням, то в разі, якщо якийсь із покажчиків стане невірним, цілісність списку можна відновити за іншим ланцюгом посилань.

Для ще більшої ефективності списку використовується наступна структура дескриптора списку:

- тип структури;
- назву списку;
- покажчик (адреса) початку списку;
- покажчик (адреса) кінця списку;
- поточне число елементів;
- опис елемента (довжина, тип і т.п.).

Тобто до дескриптора списку додається покажчик кінця списку (END). Для відсортованих списків це дозволяє домогтися більшої ефективності пошуку. Алгоритми для роботи з двоспрямованими та односпрямованими списками дуже схожі, за винятком того, що перші повинні виконувати додаткові дії для керування ще одним набором посилань.

5.4 Циклічні списки

У циклічному зв'язному списку, останній вузол містить покажчик на перший вузол списку. Використовуються як циклічний односпрямований, так і циклічний двоспрямований списки. Здійснюючи навігацію по такому списку, ми можемо почати в будь-якому вузлі і проходити по списку в будь-якому напрямку, вперед або назад, поки ми не досягнемо того ж вузла, з якого ми розпочали. Таким чином, циклічний зв'язний список не має ні початку, ні кінця. Умовно, початком циклічного списку вважається вузол за адресою, вказаної в дескрипторі списку. На наступному малюнку наведено приклад циклічного односпрямованого списку.

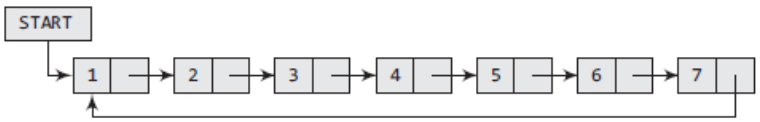


Рисунок 5.13 Циклічний односпрямований список

Єдиним недоліком циклічного зв'язного списку є складність навігації. Справа в тому, що в циклічному списку відсутній останній вузол і, отже, немає значення *NULL* в адресній частині будь-якого з вузлів списку. Це ускладнює індикацію моменту завершення перегляду списку.

Зате циклічний список може бути дуже корисний у тому випадку, коли в рамках виконання завдання потрібно нескінченне число разів проходити через послідовність будь-яких елементів і виконувати пов'язані з ними завдання. Циклічні списки широко використовуються в багатозадачних ОС для обслуговування задач. Так, ОС здатна повторювати цикл завдань, щоб по черзі запускати і зупиняти кожну з них, виділяючи задане число квантів часу процесора для її виконання. Якщо почався новий процес, він може бути доданий в будь-яке місце списку, наприклад за дескриптором, що дозволить йому відразу-ж і запуститися.

Ще один приклад використання циклічного списку - це Інтернет. Коли ми

працюємо в Інтернеті, ми можемо використовувати кнопку *Назад* і кнопку *Вперед* браузера для переходу до попередніх / наступних сторінок. Для цього структура циклічного списку використовується в браузері для підтримки послідовності відвіданих Web сторінок.

Алгоритми для роботи з циклічними списками повинні враховувати відсутність термінальних вузлів в списку, для чого використовується маркування вузлів. Це ускладнює їх реалізацію.

5.5 Багатозв'язні списки

В односпрямованому зв'язному списку кожен вузол містить посилання лише на наступний елемент, в двоспрямованому - на наступний і попередній. Це необхідно для того, щоб забезпечити два способи переміщення - прямий і зворотний. Однак до структури вузлів списку можна додавати і інші посилання, щоб передбачити додаткові способи переміщення і в інших напрямках.

Припустимо, ми сконструювали структуру *Planet* для зберігання інформації про планети Сонячної системи і призначили їй поле з ім'ям *Distance*, в якому містяться дані про відстань від планети до Сонця. З огляду на це список буде організований (відсортований) у наступному порядку: Меркурій, Венера, Земля, Марс, Юпітер, Сатурн, Уран, Нептун (і Плутон, при бажанні). Аналогічним чином можна додати і інші поля, щоб впорядкувати планети по їх масі (поле *weight*), діаметру (поле D), тощо. Кожен ланцюжок вузлів списку, відсортований за новим полем, буде визначатися новим набором адресних покажчиків списку і буде називатися потоком. З одним потоком працювати легко, якщо уявити його як простий односпрямований або двоспрямований список, але візуалізувати всі потоки одночасно досить складно. На Рисунок 5.14 показаний зв'язний список планет з трьома потоками. Посилання, позначені тонкою лінією, вказують на поток сортування по віддаленості від Сонця, пунктирною – на поток по масі, а товстої – на поток по діаметру.

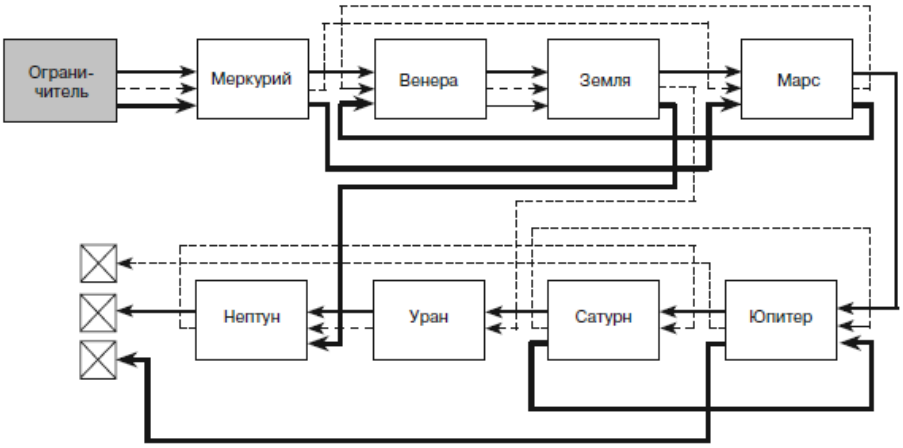


Рисунок 5.14 Приклад багатозв'язного односпрямованого списку

Таким чином, можна бачити, що в багатозв'язних списках, кожен вузол може мати *n* чи *2n* покажчиків на інші вузли, де *n* - число потоків списку. двоспрямований список є окремим випадком багатозв'язного списку при *n=1*. Однак, на відміну від двоспрямованого списку, вузли в багатозв'язних списках можуть бути чи не бути двоспрямованими окремо для кожного з потоків.

5.6 Контрольні завдання і запитання

1. У розділі 5.2.3 наводяться алгоритми вставки швидкості *O(N)*. Якщо ви збережете іншу змінну *bottom*, що буде розміщена у дескрипторі і вказує на останній елемент списку, то зможете додавати елементи у кінець списку за час *O(1)*. Напишіть відповідний код. Як він ускладнює інші алгоритми, які знаходять, видаляють чи додають елемент на початок (кінець) списку? Реалізуйте алгоритм видалення елемента з такого списку.

2. Розробіть алгоритм та програму для знаходження найбільшого елемента в несортованому односпрямованому списку з елементами, що містять цілі числа.

3. Напишіть алгоритм та програму, щоб додати елемент до початку двонаправленого списку.

4. Створіть алгоритм та програму для додавання елемента до кінця двонаправленого списку.

5. Напишіть алгоритм та програму, яка видаляє певний елемент з двонаправленого списку. Подайте цей процес у вигляді малюнка.

6. Припустимо, у вас є сортований двонаправлений список з іменами. Чи можна покращити алгоритм, розпочавши пошук з покажчика не на перший елемент, а на останній? Чи змінить це час його роботи?

7. Напишіть алгоритм та програму вставки елемента у сортований двонаправлений список, де покажчики на перший і останній елементи містять, відповідно, мінімально та максимально можливі значення.

8. Реалізуйте алгоритм та програму, яка визначає, чи відсортовано зв'язний список.

9. Напишіть програму, яка створює багатопотоковий список планет (див. п. 5.5). Дозвольте користувачеві вибрати потрібний перемикач або пункт списку, щоб відсортувати планети по різних потоках.

6 ЧЕРГА І СТЕК

Стеки і черги - відносно прості структури даних, які зберігають об'єкти в порядку «першим прийшов, першим вийшов» або «останнім прийшов, першим вийшов». По мірі необхідності їх можна розширювати, щоб додавати додаткові елементи, чи скорочувати для видалення елементів, подібно зв'язним спискам. Для створення стеків і черг найчастіше використовуються зв'язні списки, хоча можливе використання й масивів.

Стеки і черги ідеально підходять для моделювання реальних ситуацій - каналів обслуговування в банку, супермаркеті або операційній системі. Однак найчастіше в них зберігають об'єкти, які будуть в подальшому оброблятися за допомогою інших алгоритмів, наприклад мережевих алгоритмів найкоротшого шляху.

6.1 Поняття стекової структури

Стек — це структура даних, в якій елементи додаються і видаляються у порядку «останнім прийшов, першим пішов». Через таку поведінку їх іноді називають списками *LIFO*, або просто *LIFO* (від англ. *Last in first out*).

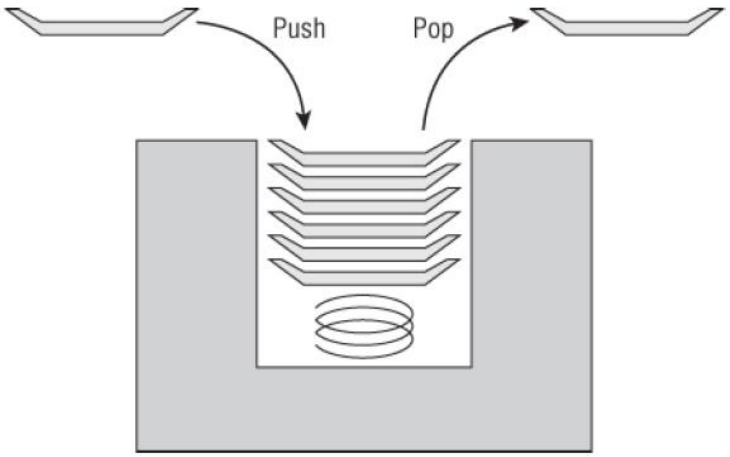


Рисунок 6.1 Стопка тарілок, як приклад стековой структури

Стек легко уявити у вигляді стопки книг, що лежить на столі: можна додати або прибрати книгу зверху, але не можна вийняти її з середини або знизу, не зруйнувавши всю конструкцію. Ще один хороший приклад - стос тарілок на кухні (рисунок вище). Додати тарілку зверху - легко, знизу - неможливо. Зняти тарілку зверху - легко, знизу - неможливо. Тарілки можна додавати і знімати тільки на / з вершини стопки.

Таким чином, стек являє собою динамічно змінюваний набір елементів, в

якому запис і вибірка елементів проводиться лише з одного кінця [1]. Тобто вибірка елементів із стека завжди здійснюється в порядку, зворотному їх засиланню.

Області застосування стека:

- реалізація механізму виклику функцій і передача їм параметрів;
- трансляція (синтаксичний і семантичний аналізи, генерація кодів і т.д.);
- реалізація механізму рекурсії;
- реалізація управління динамічною пам'яттю, тощо.

Стеки можуть бути представлені в пам'яті або у вигляді масиву (вектора), або у вигляді зв'язного списку. При векторному поданні під стек відводиться суцільна область пам'яті, досить велика, щоб в ній можна було помістити максимальне число елементів, що визначається розв'язуваним завданням. Граничні адреси цієї області є параметрами фізичної структури стека - вектора. В процесі заповнення стека місце останнього введеного елемента (його адреса) знаходиться в покажчику вершини стека. Якщо покажчик вийде за верхню межу стека, то стек вважається переповненим і включення нового елемента стає неможливим. Тому для векторного стека треба відводити досить велику пам'ять, розмір якої визначається максимально можливою ємністю стека. Проте якщо векторний стек в процесі своєї роботи заповнюється тільки частково, то пам'ять використовується неефективно. Так як під стек відводиться фіксований обсяг пам'яті, а кількість елементів усе-ж таки змінна, то кажуть, що стек в векторній пам'яті - це *напівстатична структура даних*. Зазвичай в стеку елементи мають один і той самий тип, тому обробка такого стека досить проста.

Багато сучасних ЕОМ містять у своїй конструкції апаратні стеки або засоби роботи зі стеками. Однак навіть у цьому випадку при розробці програм часто доводиться використовувати свої власні програмні стеки. Стек, представлений як вектор, має вигляд, показаний на рисунку нижче:

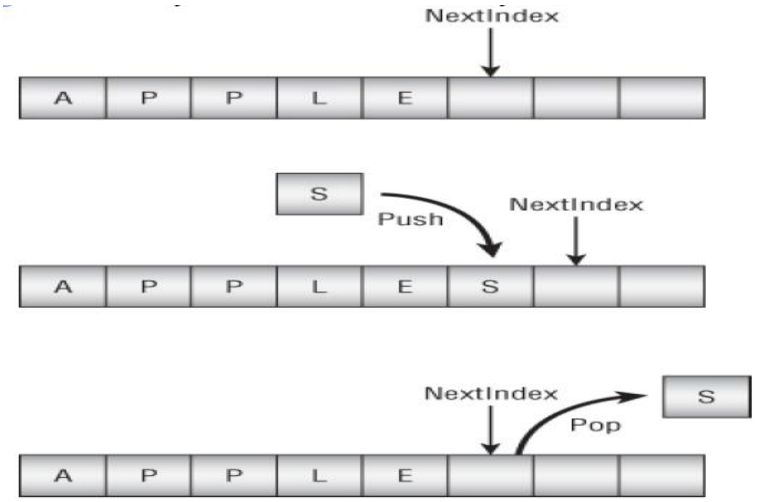


Рисунок 6.2 Представлення стека у вигляді вектора

Другий спосіб реалізації стека - це реалізація його структури за допомогою зв'язного списка. При списковому представленні стека пам'ять під дескриптор і під кожен елемент стека виділяється динамічно; включення і вибірка елементу здійснюються на початку списка, який є одночасно і *вершиною стека (*Рисунок 6.3). Переповнення стека в цьому випадку не відбувається, або відбувається досить рідко (лише при переповненні самої області динамічної пам'яті програми), проте алгоритми обробки складніші, а час обробки збільшується, оскільки операції включення та вибірки елементів пов'язані зі зверненням до операційної системи для отримання або звільнення пам'яті.

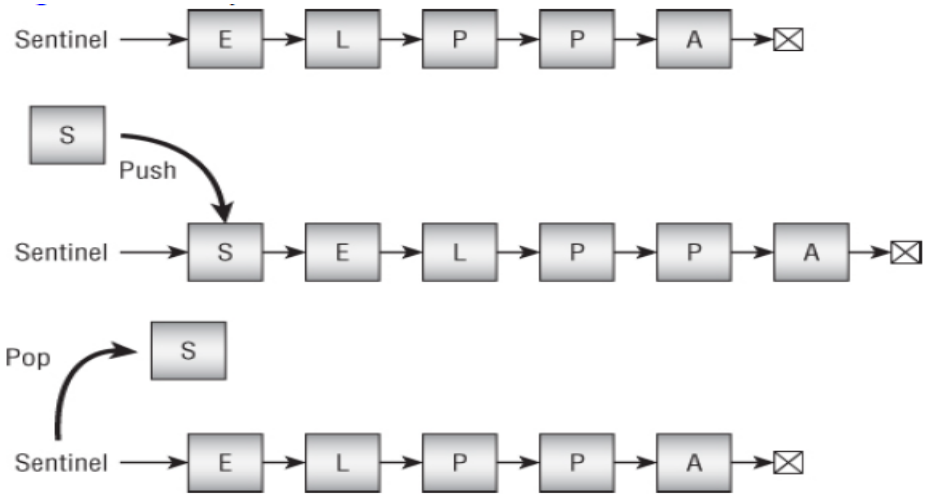


Рисунок 6.3 Представлення стека у вигляді зв'язного списку

Векторний стек не потребує додаткової пам'яті для зберігання посилань між елементами, але в ньому нераціонально використовується місце під елементи. Розмір відведеної пам'яті під векторний стек залежить від програми і від того, чи відома заздалегідь максимальна кількість елементів. Якщо такої інформації немає - при необхідності можна змінити розмір динамічного масиву, але це займе

додатковий час на його перезапис. Якщо в масиві *N* елементів, то при зміні його розміру знадобиться виконати *O(N)* кроків, щоб скопіювати вже присутні елементи на нове місце. Просте резервування місця для додаткових елементів може бути вкрай неефективним. Припустимо, що в якийсь момент алгоритму знадобиться зберегти 1000 елементів в стеку, хоча більшу частину часу йому потрібно всього кілька. В такому випадку масив стеку буде витрачати невиправдано багато ресурсів пам'яті. Якщо ж ми знаємо, що в стеці завжди буде зберігатися лише кілька десятків елементів і необхідна висока швидкість при звертанні до нього, то має сенс будовувати його саме на основі масиву. В більшості ж випадків стекову структуру слід будувати на основі списків.

6.2 Операції над стеком

Основними операціями над стеком є:

- створення стека;
- включення елемента в стек;
- отримання елемента зі стека;
- видалення стека.

6.2.1 Створення стека

Для векторного стека в пам'яті необхідно побудувати дескриптор і відвести пам'ять для масива під елементи стека виходячи з максимально можливої їх кількості [2]. Це може бути виконано або транслятором при компіляції програми, або шляхом отримання динамічної пам'яті в ході виконання програми. У будь-якому випадку при ініціалізації дескриптора в нього обов'язково заносяться покажчики початку і кінця стека (перше і останнє значення індексів масиву), а в покажчику вершини стека встановлюється ознака *NULL*.

Для стека спискової структури створення стека зводиться до побудови тільки дескриптора, в ньому достатньо наявності лише покажчика на початок списку, який одночасно є і покажчиком вершини стека, так як включення і вибірка елемента здійснюються лише з початку списку. Початково цей покажчик встановлюється в *NULL*, тим самим позначаючи, що стек порожній. Розглянемо створення стека службовців на основі зв'язного списку з інформаційним елементом – записом службовця зі структурою, що наведена нижче:

Приклад 6.1 Створення стека службовців на основі зв'язного списку

```
struct Date {
        int Den; /* день */
        int Mes; /* місяць*/
        int God; /* рік */
};

struct sluzh {
        char Fam[30];
        char Im[30];
        char Otch[30];
        char Pol;
        int      Vozr;
        struct Date Rozhd;
        struct Date Post;
        char Spec[5];
        char SemPol;
};

typedef struct tStackNode {        /* структура вузла стека */
        struct sluzh data;   /* дані про службовця */
tStackNode *next;  /* покажчик на наступний елемент стека */
};

typedef struct tSD {         /* формат дескриптора стека */
        tStackNode *top;   /* покажчик вершини стека */
        int size;        /* поточний розмір стека (в елементах) */
};

tSD* CreateStack(void) {
        tSD *sd;
        sd = (tSD*) malloc(sizeof(tSD));
        if (!sd)
                return NULL;   /* стек не  створений */
        else {  /* стек створений */
                sd -> top = NULL;    /* ініціалізація вершини стека */
                sd -> size = 0;       /* ініціалізація розміру стека */
                return sd;
        }
}
```

6.2.2 Включення елемента у стек

У разі стека на основі масиву, якщо немає переповнення стека, новий елемент включається на вершину стека при одночасному контролі границі масива, при переповненні - відмова від включення.

У разі стека спискової структури переповнення, як правило, не буває, тим не менш, перевірка необхідна. Включення зводиться до отримання динамічної пам'яті

під елемент і занесенню туди даних. Очевидно, що в покажчик наступного елемента стека в новому елементі заноситься попереднє значення покажчика на вершину стека (початок списку) з дескриптора, а в покажчик вершини стека в дескрипторі - адреса нового елемента.

Приклад 6.2 Створення стека службовців на основі зв'язного списку

```
int PushStack (tSD *sd, const sluzh *data) {
        tStackNode *sn;     /* покажчик на новий елемент */
        if (!sd)        /* стек не існує! */
                return -2;
        sn = (tStackNode*) malloc (sizeof(tStackNode)); /* створення нового
                                                           вузла списку */
        if (!sn)        /* не вистачає пам'яті (стек повний) */
                return -1;
        sn -> data = *data;         /* занесення даних про службовця в стек */
        sn -> next = sd -> top;     /* встановлення покажчика на наступний
                                       елемент у новому вузлі */
        sd -> size++;       /* зміна поточного розміру стека */
        sd -> top = sn;     /* встановлення покажчика на вершину стека в
                               дескрипторі */
        return 0;
}
```

6.2.3 Вибірка елемента зі стека

Операція можлива тільки тоді, коли стек не порожній. Вона означає, що вибирається значення з елемента, що знаходиться на вершині стека, а сам елемент виключається зі стека. Виключення елемента з векторного стека зводиться до зсуву покажчика вершини (індекса масиву) на один елемент до початку стека. При виключенні елемента зі спискового стека покажчик на вершину стека в дескрипторі замінюється на адресу наступного елемента, а пам'ять колишнього початкового елемента звільняється.

Приклад 6.3 Вибірка елемента зі стека на основі зв'язного списку

```
int PopStack (tSD *sd, sluzh *data) {
        tStackNode *pDel; /* адреса "виштовхуємого" вузла */
        if (!sd)        /* стек не існує */
                return -2;
        if (!sd -> top)         /* стек вже порожній (underflow) */
                 return -3;
        else {  /* вибірка елемента з-верхівки стека */
```

```
            *data = sd -> top -> data;        /* передаються дані з елемента */
            pDel = sd -> top;  /* зберігається адреса "виштовхуємого"
                                  елемента */
            sd -> top = sd -> top ->next; // вершиною стека стає наступний
                                          // елемент
            sd -> size--; /* зміна поточного розміру стека */
            free (pDel);  /* звільняється пам'ять "виштовхнутого"
                             елемента */
            return 0;
        }
    }
```

6.2.4 Видалення стека

Зводиться до звільнення динамічної пам'яті, отриманої в процесі створення і обробки векторного стека; якщо ж елементи і дескриптор такого стека були побудовані транслятором, то - до очищення елементів списку в масиві та встановлення ознаки «порожньо» в дескрипторі.

Якщо стек створювався на основі спискової структури, то перед видаленням дескриптора стека слід спочатку видалити всі елементи стека по черзі, від верхівки стека і до першого елемента. В іншому випадку будуть виникати так звані *витоки пам'яті*.

Приклад 6.4 Видалення стека на основі зв'язного списку

```
int DestroyStack (tSD *sd) {
        tStackNode *pDel; /* адреса тимчасового елемента стека */
        if (!sd)        /* стек не існує */
            return -2;
        // якщо стек не порожній - спершу слід видалити всі його елементи!
        while (sd -> top) { /* обхід елементів стека */
            pDel = sd -> top;  /* збереження адреси вершини стека для
                  видалення */
             sd -> top = sd -> top -> next;  /* "обрізання" верхівки стека */
            free (pDel);  /* видалення колишньої верхівки стека*/
        }
        free (sd);      /* звільняється пам'ять дескриптора стека */
        return 0;
    }
```

6.3 Поняття черги

Черга— це змінюваний упорядкований (чи ні) набір елементів. Додавання

елемента в чергу проводиться з одного кінця (хвоста черги, *REAR*), а вибірка - з іншого кінця (голови черги, *FRONT*) відповідно до правила "Першим прийшов - першим пішов" (*FIFO: First Input - First Output*). Така черга є *простою чергою* без пріоритетів (рисунок нижче). Часто використовуються *черги з пріоритетами*, в них більш пріоритетні елементи включаються ближче до голови черги, вибірка здійснюється, як правило, з голови черги.

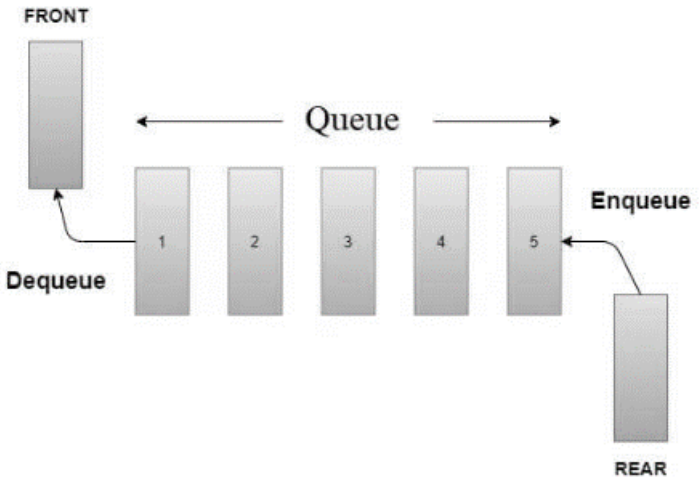


Рисунок 6.4 Схема простої черги

Пояснимо поняття черг з використанням аналогій, наведених нижче:

- люди на ескалаторі. Люди, які вступили на ескалатор першими будуть першими і при сходженні з нього;
- черга в очікуванні маршрутки. Перша людина, що стоїть в черзі, буде і першою, що зайде в маршрутку;
- черга за квитками біля вікна каси. Перша людина в черзі отримає квиток першою і таким чином, буде першою, яка покине чергу, тощо.

У всіх цих прикладах ми бачимо, що елемент в голові черги обслуговується першочергово. Те ж саме відноситься до структури даних черга.

Черги знаходять широке застосування в операційних системах (черги завдань, буфера введення-виведення, буфер введення з клавіатури), в мережах ЕОМ, при моделюванні реальних процесів і т.ін.

Черги можуть мати векторну або списковую структуру зберігання, в свою чергу векторна структура може займати статичну або динамічну пам'ять. Черга векторної структури через обмеженість елементів має властивість переповнення, коли хвіст черги досягне кінця вектора. В цьому випадку додавання елементів стає неможливим, навіть якщо в початковій частині вектора будуть вільні місця з-під

забраних елементів. Для усунення такого недоліку утворюють кільцеві черги. При досягненні кінця вектора нові елементи додаються на вільні місця з початку вектора. Тут також можливо переповнення, коли хвіст наздожене голову. Якщо ж голова наздожене хвіст, то черга виявляється порожньою.

6.4 Операції з простою чергою, реалізованої у вигляді списку

Реалізація черги у вигляді масиву ефективна тоді, коли максимальний розмір черги маленький або відомий заздалегідь. Але якщо розмір масиву під чергу не може бути визначено заздалегідь (а таких випадків більшість), іншої альтернативи, ніж використання представлення черги у вигляді зв'язного списку, не існує.

У такій черзі, кожен елемент складається з двох частин, одна з яких зберігає дані, а інша - адресу наступного елементу. Покажчик *START* дескриптора черги використовується як *FRONT*. Крім того, використовується ще один покажчик у дескрипторі, званий *REAR*, який буде зберігати адресу останнього елемента в черзі. Всі додавання (постановка) нових елементів в чергу будуть виконуватися з хвоста черги, а всі читання (з одночасним видаленням) елементів з черги будуть виконуватися з голови черги. Якщо *FRONT = REAR = NULL*, то це вказує на те, що черга порожня.

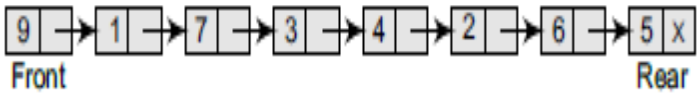


Рисунок 6.5 Реалізація простої зв'язної черги

Основними операціями над чергою є:

- створення і видалення черги;
- включення в чергу нового елемента;
- вибірка елемента з черги.

При їх виконанні використовуються такі допоміжні операції, як перевірка наявності елементів в черзі і перевірка переповнення черги.

6.4.1 Створення черги

Як і у випадку зі стеком, створення черги зводиться до побудови тільки дескриптора черги. У ньому присутні покажчик на голову черги *front*, покажчика на хвіст черги *rear* і поточний розмір черги *size*. Покажчики *front* і *rear* встановлюється в *NULL*, поле *size* встановлюється в нуль, позначаючи, що черга

порожня. Розглянемо створення черги зі структурою даних запису службовця, як у Приклад 6.1 на основі зв'язного списку.

Приклад 6.5 Створення простої черги на основі зв'язного списку

```
typedef struct tQueueNode { /* структура вузла черги */
        struct sluzh data;   /* дані про службовця */
        tQueueNode *next;        /* покажчик на наступний елемент черги */
};

typedef struct tQD {        /* формат дескриптора черги */
        tQueueNode *front;       /* покажчик голови черги */
        tQueueNode *rear;        /* покажчик хвоста черги */
        int size;       /* поточний розмір черги (в елементах) */
};

tQD* CreateQueue(void) {

        tQD *qd;

        qd = (tQD*) malloc(sizeof(tQD));

        if (!qd)
                return NULL;        /* черга не створена */
        else {          /* черга створена */
                qd -> front = qd -> rear = NULL;      /* ініціалізація голови і
                                                         хвоста черги */
                qd -> size = 0;      /* ініціалізація розміру черги */
                return qd;
        }
}
```

6.4.2 Видалення черги

Зводиться до звільнення динамічної пам'яті, отриманої в процесі створення і обробки черги. Перед видаленням дескриптора черги слід видалити всі елементи черги по черзі, починаючи з голови черги і до її хвоста. В іншому випадку будуть виникати *витоки пам'яті*.

Приклад 6.6 Видалення черги на основі зв'язаного списку

```
int ReleaseQueue (tQD *qd) {

        tQueueNode *pDel;        /* адреса тимчасового елемента черги*/

        if (!qd)        /* черга не існує */
                return -2;

        /* якщо черга не порожня - спершу видалити всі її елементи! */
        while (qd -> front) {       // обхід елементів черги, починаючи з голови
                pDel = qd -> front;         /* збереження адреси голови черги
                                               для подальшого видалення */
```

```
        qd -> front = qd -> front -> next;      // "обрізання" голови черги
        free (pDel);  /* видалення колишньої голови черги */
    }

    free (qd);     /* звільняється пам'ять дескриптора черги */
    return 0;

}
```

6.4.3 Включення в чергу

У разі черги спискової структури переповнення, як правило, не буває, тим не менш, перевірка необхідна. Включення в чергу зводиться до отримання динамічної пам'яті під новий елемент, занесенню в нього даних і додавання елемента в хвіст черги. Очевидно, що в покажчик наступного елемента черги в старому хвості заноситься значення покажчика на новостворений елемент (новий хвіст черги).

Приклад 6.7 Включення в чергу на основі зв'язного списку

```
int InQueue (tQD *qd, sluzh const* const data) {

    tQueueNode *qn; /* покажчик на новий елемент */

     if (!qd)       /* черга не існує! */
        return -2;

    qn = (tQueueNode*) malloc (sizeof(tQueueNode)); /* створення нового
                                                       вузла списку */
    if (!qn)       /* не вистачає пам'яті (динамічна пам'ять переповнена) */
        return -1;

    qn -> data = *data;        /* занесення даних про службовця
                                  в елемент черги */
    qn -> next = NULL;         /* встановлення ознаки хвоста черги */
    if (qd -> front)       /* якщо черга не порожня, присвоювання
                              колишньому хвосту черги адреси нового  хвоста */
        qd -> rear -> next = qn;

    qd -> size++;          /* зміна поточного розміру черги */
    qd -> rear = qn;  /* встановлення покажчика на хвіст черги
                         в дескрипторі */
    if (!qd -> front)      /* встановлення покажчика на голову черги
                              в дескрипторі, якщо черга була порожня */
        qd -> front = qd -> rear;
    return 0;

}
```

6.4.4 Виключення (читання) елемента з черги

Операція можлива тільки тоді, коли черга не порожня. Вона означає, що

вибирається значення з елемента, що знаходиться в голові черги, а сам елемент виключається з черги. Покажчик на голову черги в дескрипторі замінюється на адресу наступного елемента, а пам'ять під колишню голову черги звільняється.

Приклад 6.8 Виключення (читання) елемента з черги на основі зв'язного списку

```
int ExQueue (tQD *qd, sluzh *data) {
        tQueueNode *pDel; // тимчасова адреса голови черги, що видаляється
         if (!qd)        /* черга не існує */
                return -2;
        if (!qd -> front)      /* черга порожня (underflow) */
                return -3;
        /* читання і видалення голови черги */
        else {
                *data = qd -> front -> data;      /* передаються дані з елемента */
                pDel = qd -> front;          /* тимчасово зберігається адреса
                                                старої голови черги */
                qd -> front = qd -> front -> next;  /* головою черги стає наступний
                                                       елемент */
                qd -> size--; /* зміна поточного розміру черги */
                free (pDel);  звільняється пам'ять з-під колишньої голови черги
                return 0;
        }
}
```

6.5 Черга з пріоритетом

Черга з пріоритетом є структурою даних черга, в якій кожному елементу присвоюється пріоритет. Пріоритет елемента буде використовуватися для визначення порядку, в якому будуть оброблятися елементи. Загальні правила обробки елементів в черзі з пріоритетом:

елемент з більш високим пріоритетом обробляється перед елементом з більш низьким пріоритетом;

два елементи з однаковим пріоритетом обробляються за принципом перший прийшов - першим обробляється.

Чергу з пріоритетом можна розглядати як модифіковану чергу, в якій для обслуговування вибирається перший елемент з найвищим пріоритетом з головної частини черги. Сам пріоритет елемента може бути встановлений на основі різних факторів. Черги пріоритетів широко використовуються в операційних системах

для виконання процесів з найвищим пріоритетом у першу чергу. Пріоритет процесу може бути встановлений на основі необхідного процесорного часу для повного завершення процесу. Наприклад, якщо є три процеси, причому першому процесу необхідно 5 мкс, щоб завершитися, другому процесу необхідно 4 мкс, а третій процес потребує 7 мкс, то другий процес матиме найвищий пріоритет, і таким чином буде взятий з черги завдань і виконаний першим. Проте, час центрального процесора не є єдиним чинником, який визначає пріоритет, а є лише одним з кількох чинників. Іншим фактором може бути важливість одного процесу в порівнянні з іншим. Наприклад, якщо ми повинні запустити два процеси одночасно, де один процес займається бронюванням онлайн замовлення, а другий – друком файлу на принтері, то, очевидно, що онлайн бронювання є більш важливим і має бути виконано в першу чергу.

Є два способи реалізації черги з пріоритетами. Можна використовувати вже відсортований список для зберігання елементів. У цьому випадку, коли потрібно витягти елемент з черги, буде вилучатись знову головний елемент, у якого буде максимальний пріоритет після сортування. В цьому випадку при додаванні нового елемента в чергу трудомісткість алгоритму складе *O(N)*, а при вилученні елемента *O(1)*. При іншому підході до організації черги можна використовувати невідсортований зв'язний список елементів. В цьому випадку додавання нового елемента в чергу завжди робиться в хвості черги. Тоді кожен раз, коли елемент повинен бути взятий з черги на обробку, проводиться пошук в черзі першого елемента з найвищим пріоритетом. При такому підході трудомісткість складе *O(1)* при додаванні елемента і *O(N)* при вилученні елемента з черги.

Черги з пріоритетом можуть бути представлені з використанням масивів або зв'язних списків. Коли черга з пріоритетом реалізується з використанням зв'язного списку структура кожного елемента списку повинна містити наступну інформацію:

- данні елемента черги;
- пріоритет елемента;
- адресу наступного елемента черги.

Якщо ми використовуємо упорядкований зв'язаний список, то елемент з більш високим пріоритетом буде передувати елементу з більш низьким

пріоритетом. Розглянемо чергу пріоритетів, показану на наступному малюнку:

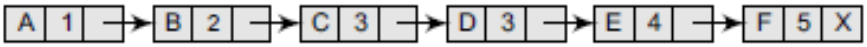


Рисунок 6.6 Реалізація черги з пріоритетом

Менше значення поля пріоритету означає більш високий пріоритет. Наприклад, якщо є два елементи A і B, де A має пріоритет 1, а B має пріоритет 2, то A буде оброблений перед B, так як має більш високий пріоритет, ніж B.

Черга пріоритетів на вищенаведеному рисунку є чергою з відсортованих пріоритетів, що має шість елементів. З черги, ми не можемо зрозуміти, чи був доданий A перед E, або навпаки, тому що список вже не працює на основі перший прийшов - перший пішов. Тут, елемент з більш високим пріоритетом, передує елементу з більш низьким пріоритетом. Проте, ми можемо з упевненістю сказати, що C був вставлений в черзі до D, тому що, коли два елементи мають однаковий пріоритет елементи обробляються за принципом перший прийшов - перший пішов.

На мові С структура елемента черги матиме такий вигляд:

Приклад 6.9 Структура елемента черги з пріоритетом

```
typedef struct tQueuePNode {    /* структура вузла черги */
        struct sluzh data;   /* дані про службовця */
        int prior;      /* пріоритет елемента */
        tQueuePNode *next;       /* покажчик на наступний елемент черги */
};
```

а структура дескриптора черги з пріоритетом на С матиме такий вигляд:

Приклад 6.10 Структура дескриптора черги з пріоритетом

```
typedef struct tQPD {       /* формат дескриптора черги */
        tQueuePNode *front;      /* покажчик голови черги */
        tQueuePNode *rear;       /* покажчик хвоста черги */
        int size;       /* поточний розмір черги (в елементах) */
};
```

Створення, видалення черги і читання (вилучення) елемента з черги з пріоритетом виконується так само, як і у випадку простої черги (див. розділ 6.4), але з урахуванням того, що змінився тип елемента черги і тип дескриптора. Щоб встановити елемент у відсортовану чергу з пріоритетом необхідно в алгоритмі враховувати три випадки:

- черга порожня (елемент додається в початок черги);

- черга не порожня і в черзі немає елементів, пріоритет у яких нижче, ніж у елемента, який додається (елемент додається в хвіст черги);
- черга не порожня і в черзі є елементи з пріоритетом нижче, ніж у того, який додається (елемент стає в середину черги).

Алгоритм додавання елемента в чергу з пріоритетами показаний на наступному прикладі:

Приклад 6.11 Встановлення у відсортовану чергу з пріоритетом

```
int InQueueP (tQPD *qd, sluzh const* const data, int prior) {

tQueuePNode *qn,/* покажчик на новий елемент */
                *ptr, *prePtr;      /* покажчики на поточний і передуючий йому
                                       елементи  черги */
if (!qd)        /* черга не існує! */
        return -2;

/*************** Створення нового елементу черги ***********/
qn = (tQueuePNode*) malloc (sizeof(tQueuePNode)); /* створення нового
                                                     елементу черги */
 if (!qn)       /* черга переповнена (не вистачає динамічної пам'яті) */
        return -1;

qn -> data = *data;         /* занесення даних про службовця в елемент черги */
qn -> prior = prior;        /* занесення даних про пріоритет в елемент черги */
qd -> size++;       /* відразу збільшуємо лічильник числа елементів черги */

/*  Випадок 1. Черга порожня, елемент додаємо в початок */
if (!qd -> front) {
        qd -> front =qd -> rear = qn;  /* перевизначаємо голову і хвіст черги
                                          в дескрипторі */
        qn -> next = NULL;          /* термінуємо чергу */
        return 0;
}
/************* Кінець випадку 1 ****************************/


/*    Інші два випадки  */

/* здійснюємо навігацію по черзі, поки не знайдемо елемент зі значенням
   поля prior більшим за задане, або не досягнемо останнього елемента
   черги */
ptr = prePtr = qd -> front; /* ініціалізуємо початок обходу черги */
do {
        prePtr = ptr;
        ptr = ptr -> next;   /* просуваємо покажчик на наступний елемент */
        if (!ptr) break;        /* якщо досягнуто хвіст черги, а значення
                                   (prior + 1)         так і не знайдено –
```

```
                                        завершуємо цикл */
} while (ptr -> prior <= prior); /* шукаємо в списку перший елемент
                                    зі значенням поля пріоритету (prior + 1) */


/* Випадок 2. Якщо в черзі немає елементів зі значенням пріоритету
більшим ніж prior, елемент додаємо в хвіст черги  */
if (!ptr) {
        qn -> next = NULL;      /* встановлення ознаки хвоста черги */
        prePtr -> next = qn;    // попередній останній елемент черги
                                // перевизначається на елемент,
                                // що додається */
        qd -> rear = qn;    /* перевизначати хвіст черги в дескрипторі */
}
/******************** Кінець випадку 2 ******************************/


/************** Випадок 3. Знайдено елемент черги з полем prior більшим
                    за задане. Додаємо новий елемент в середину черги *****/
else {
        prePtr -> next = qn; /* попередній елемент в черзі перевизначати на
                                    той, що додається */
        qn -> next = ptr;  /* доданий в чергу елемент перевизначаємо
                                на    наступний */
}
/******************** Кінець випадку 3 ******************************/

return 0;

}
```

6.6 Кільцева черга

В лінійних чергах, обговорюваних досі, включення в чергу може бути виконано тільки в хвіст черги, а виключення елемента - тільки з голови черги. Подивимося на наступну чергу, для представлення якої будемо використовувати масив, а не зв'язаний список, як раніше:

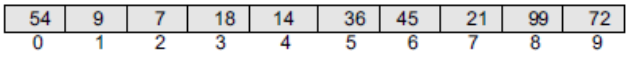


Рисунок 6.7 Лінійна черга, реалізована масивом

Видно, що тут *FRONT = 0* і *REAR = 9*. Як видно з малюнка, все повністю заповнено. Тепер, якщо виникне необхідність поставити в чергу новий елемент, то ця операція стане неможливою: в масиві, яким представлена наша черга, немає більше вільного місця. Розглянемо тепер ситуацію, в якій зроблені два послідовних читання елементів з черги. Черга після цього стане виглядати так, як показано на наступному малюнку:

		7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9

Рисунок 6.8 Векторна черга після вилучення двох елементів

Зараз *FRONT = 2* і *REAR = 9*. Припустимо, що ми тепер хочемо поставити новий елемент в чергу, показану на Рисунок 6.8. Навіть незважаючи на те, що на початку масиву з'явилися два доступних місця, умова переповнення черги, як і раніше залишається, тому що включення в чергу нових елементів ми виконуємо тільки з хвоста. Це є головним недоліком лінійної векторної черги, реалізованої у вигляді масиву.

Щоб вирішити цю проблему, можна використовувати два підходи. По-перше, перемістити елементи масиву черги вліво, так щоби простір, що звільнився у хвості можна було зайняти і використовувати ефективно. Але такий підхід може зайняти дуже багато часу, особливо, коли розмір черги (масиву) досить великий. Другий підхід полягає у використанні кільцевої черги. У кільцевій черзі, значення першого індексу виникає відразу після проходження останнього індексу. Концептуально, кільцеву чергу можливо представити у вигляді наступних малюнків:

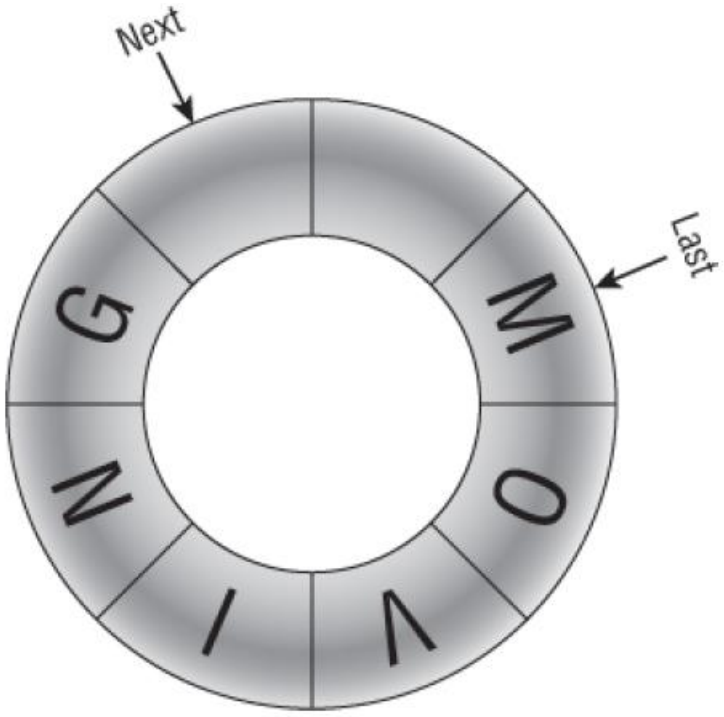


Рисунок 6.9 Приклад кільцевої черги

Алгоритми реалізації операцій включення і виключення елементів з кільцевої черги будуть складнішими, тому що в них необхідно буде врахувати і запобігати можливій "накладці" голови і хвоста черги при її заповненні, через те,

що хвіст і голова черги будуть переміщатися "по колу" в міру включення / виключення елементів з черги. Це є недоліком подібних алгоритмів.

6.7 Контрольні завдання і запитання

Зірочкою позначено завдання підвищеної складності.

1. Розробіть алгоритм та програму для отримання вихідного стека, в якому елементи вхідного стека розташовані у зворотному порядку

2. Напишіть програму, яка робить сортування вставкою для стека.

3. Виконуючи сортування вставкою в стеку, алгоритм для кожного елемента переміщує в тимчасовий стек спочатку невідсортовані елементи, потім деякі відсортовані, а далі повертає все у вихідний стек. Чи дійсно йому потрібно повертати всі елементи у вихідний стек? Чи можна удосконалити роботу алгоритму, змінивши цей крок? Як це вплине на час роботи алгоритму?

4. Створіть програму, що реалізує деку (двосторонню чергу).

5*. Уявіть, що на "Новій Пошті" клієнти вишиковуються в одну чергу, яка обслуговується кількома операторами. Новий клієнт займає місце в кінці черги і просувається до її початку, де необхідну операцію проводить оператор, що звільнився на даний момент. Змоделюйте чергу «з кількома головами» на основі звичайної, яку обслуговують кілька операторів.

Створіть програму, подібну до тієї, інтерфейс якої зображено на малюнку 5.11, щоб зімітувати чергу «з кількома головами». Дозвольте користувачеві встановлювати кількість операторів, час між клієнтами, час перебування кожного клієнта у черзі, а також швидкість симуляції. Запустіть модель із введеними параметрами, щоб побачити, як поводиться черга. Як кількість операторів вплине на середній час очікування?

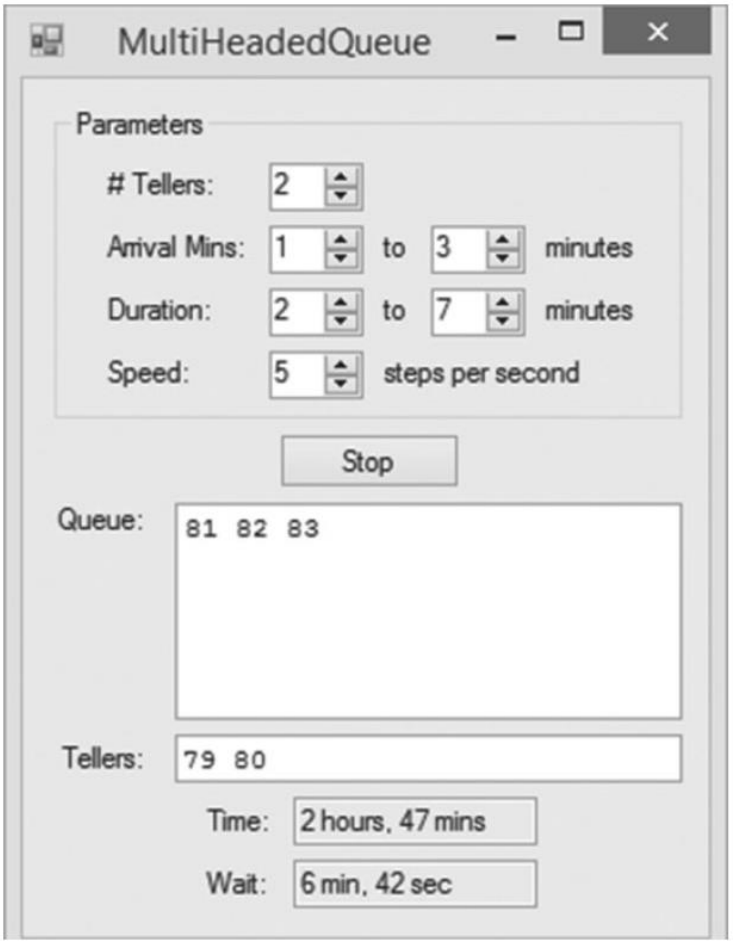


Рисунок 6.10 Вікно програми MultiHeadedQueue

6. Напишіть програму, яка здійснює сортування вставкою для черги.

7. Побудуйте алгоритм та програму черги з пріоритетом на основі купи, використовуючи масив, в якому не потрібно змінювати розміри. Призначте йому фіксований розмір, можливо, 100 елементів, а потім стежте за тими елементами, з якими працює програма. (Підказка: використовуйте два масиви (один для рядкових значень, інший для відповідних пріоритетів), розташуйте елементи відповідно до їх пріоритетів.)

8. Визначте час роботи, необхідний для додавання елементів у чергу з пріоритетом на основі купи. Проаналізуйте, скільки часу потрібно для видалення.

9. Напишіть програму, яка реалізує швидке сортування зі стеками.

10. Реалізуйте алгоритм швидкого сортування, замінивши стеки на черги. Проаналізуйте переваги та недоліки такого перетворення.

7 ФАЙЛИ ТА ЇХ ОРГАНІЗАЦІЯ

В даний час більшість організацій збирають великі обсяги даних в тій чи іншій формі. Наприклад, параметри роботи системи керування технологічною лінією, дані про студентів і співробітників університету, дані про клієнтів банку та їх транзакції, тощо. Всі ці дані до недавнього часу традиційно зберігалися у вигляді паперових документів, але обробка цих документів завжди була хаотичним і важким завданням. Аналогічно, наукові експерименти і білінг зв'язку також генерують величезні обсяги інформації.

Тому для того, щоб ефективно працювати з наявними даними з різних джерел постала необхідність зберігати дані у комп'ютерах у вигляді блоків інформації, які називаються файлами. У комп'ютерній термінології, *файл* являє собою *сукупність корисних даних, об'єднаних єдиним призначенням, збережених під одним ім'ям на постійному носії* (вінчестер, флешка, компакт-диск, тощо) [1].

7.1 Ієрархія даних

Кожен файл містить дані, які можуть бути організовані ієрархічно, щоб систематизувати інформацію. Ієрархія даних включає в себе елементи даних, такі як поля, записи, файли і каталоги. Ці терміни визначені нижче:

Поле даних *є елементарною одиницею, яка зберігає єдиний факт*. Поле даних зазвичай характеризується своїм типом і розміром. Наприклад, ім'я студента є полем даних, яке зберігає ім'я одного студента. Це поле має тип масиву символів чи рядка, і його розмір може бути встановлений на максимум від 2 до чорт-зна скільки символів, в залежності від вимог. Якщо ми візьмемо довжину поля занадто малою, то втратимо можливість зберігати довгі імена. Якщо ж взяти довжину поля з великим надлишком "на про всяк випадок" - то велика кількість байт у нас буде витрачено даремно в більшості випадків, що призведе до перевитрати місця на диску (пам'яті).

Запис *являє собою сукупність пов'язаних між собою за змістом окремих полів даних, яка розглядається як єдине ціле з точки зору застосування*. Наприклад, запис студента може містити поля даних, такі як ім'я, адреса, номер телефону, назву навчальної групи, оцінки, тощо. Запис в мові Сі добре подається структурним типом даних.

***Файл** являє собою набір пов'язаних запи*сів. Наприклад, якщо є 25 студентів в одній групі, то файл буде включати до себе 25 записів. Всі ці пов'язані записи зберігаються в *одному файлі*. Крім того, ми можемо мати файл всіх співробітників, що працюють в організації, файл всіх клієнтів компанії, файл всіх постачальників, тощо.

***Каталог (директорія, папка)** зберігає в одному місці пов'язані між собою файли*. Каталог організовує інформацію таким чином, щоб користувачі могли легко знайти її. Наприклад, каталог навчальних груп однієї кафедри, каталог кафедр на факультеті, тощо.

7.2 Атрибути файлу

Кожен файл в комп'ютерній системі зберігається у відповідному каталозі. При цьому, кожен файл забезпечується списком атрибутів, що пов'язані з ним, які надають інформацію про файл як самій операційній системі, так і прикладній програмі, яка обробляє файл (наприклад, як його можна використовувати).

Програма, яка повинна отримати доступ до файлу, переглядає запис каталогу, щоб прочитати атрибути цього файлу. Наприклад, якщо користувач намагається записати в файл, який був позначений "тільки для читання", то програма виведе відповідне повідомлення для користувача про те, що він намагається виконати заборонену операцію над цим файлом.

Частина файлових атрибутів називаються *прихованими*. При виконанні команд перегляду змісту папки (каталогу) в ОС, файли, для яких встановлено прихований атрибут, не відображатимуться.

Ім'я файлу. Це рядок символів, що однозначно визначає файл в деякому просторі імен файлової системи, зазвичай званому каталогом. Імена файлів будуються за правилами, прийнятим в тій чи іншій файловій і операційній системі. Багато систем дозволяють призначати імена як звичайним файлам, так і каталогам, і спеціальним об'єктам (символьним посиланням, блоковим пристроям, тощо).

Ім'я файлу є частиною повного імені файлу, який також називають повним або абсолютним шляхом до файлу. Повне ім'я може включати наступні компоненти:

- протокол або спосіб доступу (*http, ftp, file,* тощо);
- ім'я або адреса комп'ютера, вузла мережі (*wikipedia.org, 207.142.131.206,*

\\MYCOMPUTER, SYS: тощо);

- пристрій зберігання, диск (*C :, /, SYSLIB*, тощо);
- шлях до каталогу *(/usr/bin, \TEMP, [USR.LIB.SRC]*, тощо);
- власне ім'я файлу, яке може містити його розширення *(.txt, .exe, .COM*, тощо);
- версія або номер ревізії (*perl6, ping6*, тощо).

Ім'я файлу необхідно для того, щоб до файлу міг звернутися користувач. В одному каталозі не може бути двох файлів з однаковими іменами (деякі файлові системи чутливі до регістру, що залишає можливість для створення файлів, імена яких відрізняються лише регістром символів).

Ім'я файлу зазвичай складається з двох частин, між якими ставиться крапка:

- назва (до точки, часто також називають просто ім'ям);
- розширення імені.

Наприклад в MS-DOS ім'я файлу не може перевищувати 8 символів плюс до трьох символів розширення імені, в Windows до 255 символів.

Покажчик позиції. Це покажчик, який вказує на байтову позицію в файлі, з якої буде виконуватися наступна операція читання/запису.

Тип файлу. Цей атрибут вказує на те, чи є файл текстовим або двійковим. У *текстовому файлі*, числа (цілі або з плаваючою точкою) зберігаються у вигляді рядка символів. Це дозволяє людині безпосередньо читати інформацію з файла, але робота з текстовими файлами сильно сповільнюється через їх надлишковий обсяг і необхідність виконання операцій перетворення при читанні / запису. У *двійковому файлі*, з іншого боку, дані зберігаються неперетвореними у тому ж самому двійковому вигляді, в якому вони представлені в пам'яті комп'ютера. Це прискорює роботу з файлом і інформація, що зберігається в файлі, займає менше місця. Платою за це є відсутність можливості для людині безпосередньо читати вміст таких файлів.

Метод доступу до файла. Він вказує на те, чи можна отримати доступ до довільного запису файлу або доступ до записів файлу організований чітко послідовно, як на магнітофонній стрічці. В режимі *послідовного доступу*, записи зчитуються чітко один за одним. Тобто, якщо 25 записів про студентів однієї групи зберігаються у послідовному файлі *STUDENT*, а нам необхідно прочитати запис

19-го студента, ми повинні будемо спочатку прочитати записи перших 18 студентів, навіть якщо вони нам і не потрібні. У файлі з прямим доступом, записи можуть бути доступні в будь-якому порядку.

Флаги атрибутів файлу. Файл може мати шість додаткових атрибутів, прикріплених до нього. Ці атрибути, як правило, зберігаються в одному байті, причому кожен біт, являє собою визначення окремого атрибута. Якщо якийсь біт встановлений в '1', то це означає, що відповідний атрибут ввімкнено. Наступна таблиця показує список атрибутів і їх положення в байті атрибутів файлу.

Таблиця 7.1 Список файлових атрибутів

Attribute	Attribute Byte
Read-Only	00000001
Hidden	00000010
System	00000100
Volume Label	00001000
Directory	00010000
Archive	00100000

Якщо системний файл встановлюється як прихований і тільки для читання, то його атрибут-байт може бути визначений як 00000111. Далі, в цьому пункті, наведемо повний список атрибутів файлу. *Зверніть увагу, що каталог розглядається як спеціальний файл в операційній системі*. Таким чином, всі ці атрибути застосовні до файлів, а також до каталогів єдиним чином.

Read Only. Файл позначений "тільки для читання". Його дані не можуть бути видалені або змінені. Наприклад, якщо зроблена спроба видалити або змінити вміст файлу "тільки для читання" з'явитися повідомлення "відмовлено в доступі" на екрані.

Hidden. Файл, позначений як прихований не відображається в списку файлів каталогу при його перегляді.

System. Файл, позначений як системний, вказує на те, що це важливий файл, який використовується операційною системою, і прикладні програми не повинні його змінювати або видаляти з диска. По суті, це "більш серйозний" атрибут файлу, який використовується часто разом з атрибутом *Read Only*.

Volume label. Кожному розділу (тому, диску) носія присвоюється мітка для його ідентифікації. Мітка може бути призначена під час форматування / створення

носія або пізніше за допомогою різних інструментів, таких як команда *LABEL* операційної системи.

Directory. У списку файлів поточного каталогу файли і підкаталоги поточного каталогу відрізняються один від одного скинутим / встановленим атрибутом Directory. Це означає, що файли, які мають атрибут *Directory* встановленим, фактично не є файлами, а є підкаталогами, які в свою чергу можуть містити один або кілька файлів.

Archive. Архівний біт використовується в якості зв'язку між програмами, які модифікують файли і тими, які використовуються для резервного копіювання файлів. Більшість програм резервного копіювання дозволяють користувачеві виконувати інкрементне резервне копіювання. Інкрементне резервне копіювання вибирає тільки ті файли для резервного копіювання, які були змінені з моменту останнього резервного копіювання.

Коли програма резервного копіювання створює резервну копію файлу, або, іншими словами, коли програма архівує файл, вона очищає біт архіву (встановлює його рівним нулю). Надалі, якщо будь-яка програма змінює файл - вона вмикає архівний біт (встановлює його в 1). Таким чином, всякий раз, коли програма резервного копіювання запускається, вона перевіряє біт архіву файлу, щоб дізнатися, чи був файл змінений з моменту її останнього запуску. Програма резервного копіювання буде архівувати тільки ті файли, які були змінені.

7.3 Текстові та двійкові файли

Текстовий файл, також відомий як плоский файл або файл ASCII, структурується як послідовність рядків символів алфавіту, цифр, спеціальних символів, тощо. Дані в текстовому файлі, числові і нечислові, зберігаються використовуючи відповідний код ASCII для кожного збереженого символу. Кінець текстового файлу позначається шляхом розміщення в останньому байті файлу спеціального символу, званого маркером кінця файлу *0x1A = 26 (Ctrl-Z).* Текстові файли зчитуються завжди лише до вказаного символу. Інформація після нього в текстовому режимі зчитана не буде. Інформація в текстовому файлі зазвичай розбивається на рядки і в кінці кожного рядка знаходиться символ кінця рядка *'\n'*. В кінці останнього рядка цей символ може бути відсутнім. *Текстові файли завжди послідовні*.

Двійковий (бінарний) файл містить довільний тип даних, закодований в двійковій формі для комп'ютерного зберігання і цілей обробки. Двійковий файл може містити і текст, що не розбивається на рядки. Двійковий файл зберігає дані в форматі, який аналогічний формату, в якому ці ж самі дані зберігаються в оперативній пам'яті. Таким чином, двійковий файл (у більшості випадків) не доступний для читання людиною, і для нього необхідна спеціалізована програма, що дозволяє побачити вміст.

Бінарні файли містять інформацію про форматування, яке розуміють лише певні програми або процеси. Для людей цілком можливо читати текстові файли, які містять тільки текст ASCII, в той час як бінарні файли стає можливим прочитати людині тільки за допомогою відповідних програм, які здатні перетворювати вміст двійкового файлу в вид, що розуміється людиною. Наприклад, тільки Microsoft Word може інтерпретувати інформацію форматування в документі Word.

При необхідності, текстові файли також можна обробляти як бінарні, але ніякого перетворення інформації при цьому проводитися не буде.

7.4 Базові файлові операції

Повний перелік базових файлових операцій представлено на малюнку нижче. Розглянемо їх детальніше.

Creation (Створення / відкриття файлу). Файл створюється / відкривається функцією відкриття файлу *fopen(),* шляхом зазначення його імені та режиму роботи з файлом. Файл відкривається відповідно до режиму роботи для запису / читання вмісту. Після завершення роботи з файлом, файл закривається функцією *fclose()*.

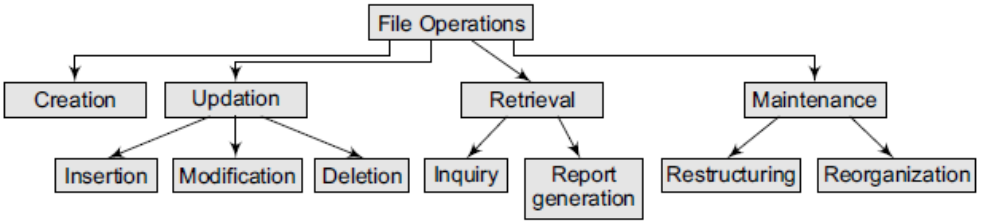


Рисунок 7.1 Діаграма зв'язку базових файлових операцій

Updation (Оновлення файлу). Оновлення файлу означає зміну вмісту файлу, щоб відобразити поточну картину реальності. Файл може бути оновлений наступними способами:

insertion (вставляння) нового запису в файл. Наприклад, якщо новий студент вступає до університету, нам потрібно додати новий запис в файл *STUDENT*;

deletion (видалення) існуючого запису. Наприклад, якщо студент завершує навчання в університеті, його запис повинен бути видалений з файлу *STUDENT;*

modification (зміна) існуючого запису. Наприклад, якщо студент змінює навчальну групу в процесі навчання, то виправленням імені групи буде модифікація існуючої записи про студента, а саме, поля *Группа*.

Слід зауважити, що в бібліотеці мови С відсутні готові функції, які виконують всі три вищевказані операції оновлення і програміст повинен писати свої, на базі існуючих функцій введення / виведення мови С.

Retrieval (Отримання інформації з файлу). Ці операції означають отримання необхідних даних із зазначеного файлу. Інформація може бути отримана з файлу або для відповіді на запит (*Inquiry*) або для формування звіту (*Report generation*). Запит, зазвичай, формує невеликий обсяг даних, в той час як генерація звіту може формувати значні обсяги даних з файлу. Для отримання даних з файлу в бібліотеці функцій введення / виведення мови Сі існує декілька функцій, що дозволяють працювати з файлом як у побайтовому, так і по блочному режимах як з текстовими, так і з двійковими файлами.

Maintenance (Збереження файлу). Операції включають в себе реструктуризацію (*Restructuring*) або реорганізацію (*Reorganization*) зберігання файлу для підвищення продуктивності роботи програм, які здійснюють обробку цього файлу. Перебудова структури файлу зберігає організацію зберігання файлу без зміни, а змінює лише структурні аспекти файлу (наприклад, змінює ширину поля записи файлу або додає / видаляє поля в запису). З іншого боку, реорганізація файлу може включати в себе зміну всієї схеми організації доступу до файлу на диску. У бібліотеці функцій введення / виводу мови Сі відсутні готові функції, які виконують вищевказані дії і завдання їх забезпечення лягає цілком і повністю на плечі програміста, використовуючи існуючі функції введення / виводу бібліотеки.

7.5 Організація файлів

Ми вже знаємо, що файл являє собою набір пов'язаних записів. Основною проблемою в управлінні файлами є спосіб, в якому записи організовані всередині файлу, оскільки саме спосіб істотно впливає на продуктивність системи.

Організація записів означає логічне розташування записів у файлі, а не їхнє фізичне розташування всередині файлу, який зберігається на носії.

Так як вибір відповідного способу організації файлу є проектним рішенням, то воно повинно бути вибрано виходячи з пріоритету досягнення оптимальної роботи при найбільш вірогідному використанні файлу. Таким чином, наступні міркування слід мати на увазі, перш ніж вибирати відповідний спосіб організації файлів:

- швидкість доступу до одного або декількох записів файлу;
- простота модифікації одного або декількох записів файлу, не порушуючи швидкості доступу до інших записів;
- ефективне (не надмірне) зберігання записів у файлі;
- використання надмірності для забезпечення цілісності даних файлу.

Хоча можна виявити, що деякі з цих вимог знаходяться в суперечності один з одним, це робота розробника: знайти хороший компроміс між суперечливими вимогами, щоб отримати ефективне вирішення проблеми. Наприклад, легкість додавання записів може бути погіршена для того, щоб отримати більш швидкий доступ до даних в операціях читання / запису.

7.5.1 **Послідовна організація**

У послідовно-організованому файлі записи зберігаються чітко в тому порядку, в якому вони були введені. Тобто, перший введений запис буде і першим записом у файлі, другий введений запис - буде другим записом у файлі, і так далі. В результаті, нові записи додаються тільки в кінець файлу.

Послідовні файли можуть бути прочитані тільки послідовно, починаючи з першого запису. Послідовна організація файлу є основним способом організації великих та надвеликих масивів записів у файл. Рисунок 7.2 показує *n* пронумерованих записів від *0* до *n-1*, що зберігаються в послідовному файлі.

Після того, як ми починаємо зберігати записи в файлі, ми не можемо робити будь-які зміни в них. Ми навіть не зможемо просто видаляти записи з послідовного файлу. У разі, коли потрібно видалити або оновити один або кілька записів, ми повинні повністю замінити всі записи шляхом створення нового файлу і перезапису в нього інформації зі старого з внесенням відповідних змін.

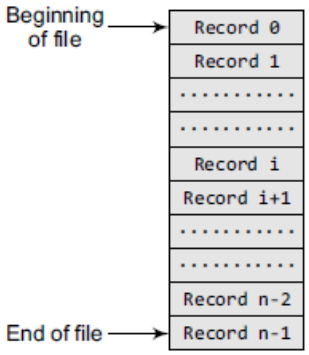


Рисунок 7.2 Послідовна організація інформації у файлі

У послідовній організації файлів, всі записи мають однаковий розмір і структуру. Кожне поле в межах кожного запису має фіксований розмір. Записи упорядковано на основі значення одного поля або комбінації двох або більше полів. Це поле/поля називаються ключем запису. Кожен ключ однозначно ідентифікує запис у файлі. Таким чином, кожен запис має різне значення для ключового поля. Записи можуть бути відсортовані в порядку зростання або зменшення ключа.

Таблиця 7.2 Характеристики послідовної організації файлів

Особливості	Переваги	Недоліки
• записи записуються в тому ж порядку, в якому і вводяться • записи зчитуються і записуються чітко послідовно • видалення або оновлення однієї або декількох записів файлу призводить до необхідності перезапису всього файлу з внесеними окремими виправленнями на нове місце • всі записи мають однаковий розмір, однакову структуру і однаковий формат полів • всі записи в файлі відсортовано за значенням ключа • в основному використовуються для створення звітів або послідовного читання	• простота та зручність в роботі • відсутні додаткові накладні витрати на зберігання • послідовні файли можуть зберігатися на магнітних стрічках і дисках • добре підходить для пакетної обробки • добре підходить для зберігання великих обсягів даних і резервування інформації	• записи можуть бути прочитані тільки послідовно.Якщо необхідний *n*-ий запис, перед ним доведеться прочитати усі попередні (*n-1*) записи • не підтримує операції оновлення. Необхідно створювати новий файл з перезаписом в нього всієї інформації зі старого, навіть з мінімальними виправленнями • не може бути використано для інтерактивних додатків

Послідовні файли зазвичай використовуються для створення звітів або для зберігання великої кількості послідовних даних (наприклад, показники датчиків температури, тиску, вологості, розміщених на об'єкті управління). Послідовні

файли можуть легко зберігатися, як на магнітних стрічках, так і на дисках. У бібліотеці функцій введення / виводу мови С є багато функцій для обробки файлів з послідовним доступом. У Таблиця 7.2 наведено характеристики, переваги та недоліки послідовної організації файлів.

7.5.2 **Прямий доступ**

Прямий доступ забезпечує ефективний спосіб безпосереднього доступу до довільних записів файлу. При такому способі доступу, записи впорядковуються у файлі по їх відносному ключу. Це означає, що індекс запису являє собою розташування запису щодо початку файлу (як індекс в масиві). Індекс запису приймає значення в діапазоні від *0* до *N-1*, де *N* - число записів у файлі. Наприклад, запис з індексом *0* - перший запис у файлі. Всі записи у файлі з прямим доступом мають фіксовану однакову довжину.

Таким чином, при прямому доступі, записи організовані в порядку зростання відносного номера (індексу) запису. Такий файл можна розглядати як один з вимірів таблиці, записаної на диску, в якому індекс запису є номером рядка в таблиці. Файли з прямим доступом можуть бути використані як для довільного, так і для послідовного доступу до даних. Для послідовного доступу, досить читати / писати записи один за одним.

Relative record number	Records stored in memory
0	Record 0
1	Record 1
2	FREE
3	FREE
4	Record 4
.....	.....
98	FREE
99	Record 99

Рисунок 7.3 Зберігання інформації в файлі з прямим доступом

Файли з прямим доступом забезпечують підтримку тільки одного ключа - індексу запису. Цей ключ повинен бути числовим і повинен приймати значення від 0 до максимального числа записів *N-1*. Це означає, що при створенні файлу повинно бути відразу виділено достатньо місця, щоб розмістити всі *N* записів,

навіть якщо в даний момент необхідності в зберіганні такого числа записів і немає.

На Рисунку 7.3 показано схематичне представлення файлу з прямим доступом, в якому було виділено достатньо місця для зберігання 100 записів. Як видно з малюнка, далеко не всі місця в файлі зайняті даними. Місця, відмічені як *FREE*, не мають інформації в записах файлу, що зберігаються. Таким чином, кожен запис у файлі або зберігає інформацію, або позначається як вільний.

Прямий доступ організації файлу забезпечує довільний доступ саме до того запису файлу, який необхідний в даний момент. Якщо записи мають фіксовану довжину, і ми знаємо базову адресу файлу і довжину запису, то адресу будь-якого запису в файлі можна отримати за такою формулою:

*Адреса n-го запису = Базова адреса першого запису + (n-1) * довжина_запису (8.1)*

Таблиця 7.3 Характеристики організації файлів з прямим доступом

Особливості	Переваги	Недоліки
• забезпечує ефективний спосіб доступу до окремих записів файлу • індекс запису пропорційний величині зсуву вибраного запису відносно початку файлу • записи в файлі з прямим доступом мають фіксовану довжину • файли з прямим доступом можуть бути використані як для довільної, так і для послідовної вибірки даних • кожне місце в таблиці записів файлу або зберігає запис або позначено як *FREE*	• простота обробки • якщо відомий відносний індекс необхідного запису, то доступ до запису може бути надано практично відразу • прямий доступ до записів робить процес обробки записів в таких файлах набагато швидше, ніж при послідовній організації • дозволяє виконувати операції оновлення записів в файлі на місці, без необхідності переписування файлу на нове місце • забезпечує як випадковий, так і послідовний доступ до записів файлу з низькими накладними витратами • нові записи можуть бути легко додані на вільні місця в файлі, без необхідності додавання їх тільки в кінець, або переписування файлу на нове місце • добре підходить для інтерактивних додатків	• використання прямої організації доступу обмежено виключно використанням фізичних пристроїв зберігання з прямою організацією доступу (магнітні диски, флешки) • записи можуть бути тільки фіксованої довжини • для довільного доступу до запису індекс запису повинен бути відомий заздалегідь (обчислений або визначений)

У таблиці вище підсумовано інформацію по характеристикам файлів з прямим методом доступу. У бібліотеці функцій введення / виводу мови С є кілька

функцій для обробки файлів з прямим доступом, такі як *fseek(), fsetpos(), fread(), fwrite()*. Або можна використовувати комбінації пошукових функцій *fseek(), fsetpos()* і побайтового читання / запису *fgetc()/fputc()*.

7.5.3 Індексно-послідовний доступ

Індексно-послідовна організація файлів використовується для швидкого отримання даних, що зберігаються. Записи в індексно-послідовному файлі мають фіксовану довжину і кожен запис однозначно ідентифікується ключовим полем (як і у випадку з файлом прямого доступу). Крім того, для кожного такого файлу підтримується файл з індексною таблицею (називається індексним файлом, або просто - індексом), яка зберігає ключі і адреси всіх записів основного файлу. Тобто для кожного послідовного файлу, ми маємо свою власну таблицю індексів. Цей тип організації файлів називається індексно-послідовною організацією, тому що фізично самі записи можуть зберігатися в будь-якому місці, але таблиця індексів зберігає адреси початку блоків, де зберігаються ці записи. Концептуально, індексно-послідовний файл можна уявити собі так, як показано на рисунку нижче:

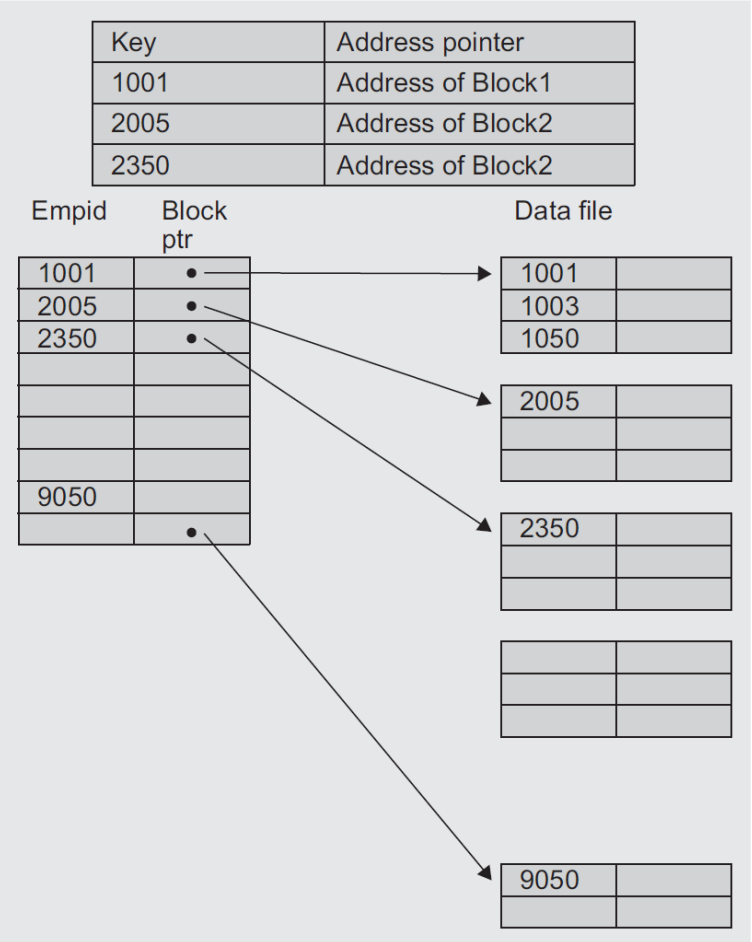


Рисунок 7.4 Індексно-послідовна організація файлу

Індексно-послідовні файли можуть мати різні структури. Однак в будь-якому випадку вони складаються з двох функціонально різних частин: області індексів і області даних. Для структури індексно-послідовних файлів характерно те, що записи файлу утворюють впорядковані по зростанню ключів послідовні групи, кожна з яких починається з запису із найменшим в групі ключем і закінчується записом з найбільшим ключем. В індексній області індекси груп утворюють ієрархічну деревоподібну структуру (дерево пошуку), що складається з декількох рівнів, для отримання доступу до записів файлу. Кожен кінцевий вузол (лист) дерева пошуку містить один з ключів - найменший чи найбільший - однієї послідовної групи і адресу початку цієї групи на пристрої прямого доступу, за яким забезпечується прямий доступ до групи. Для послідовної обробки записів файлу досить здійснити лівий спадний (прямий) обхід дерева індексів в індексному файлі і далі в кожному кінцевому вузлі (листі дерева індексів) послідовно обробити записи відповідної групи області даних. Для прямого доступу до запису із заданим ключем спочатку здійснюється пошук по дереву відповідної групи (вузла) в індексній області, потім - послідовний пошук потрібного запису всередині групи області даних. Можлива і послідовна обробка файлу, починаючи з цього запису.

Область даних, в свою чергу, складається з двох частин: *основної області* і *області переповнення (області вільної пам'яті)*. Основна область призначена для розміщення послідовних груп записів, область переповнення використовується для розміщення даних, коли при додаванні нових записів вже не вистачає місця у відповідній послідовній групі. Таким чином, область переповнення призначена для розширення послідовних груп.

Конкретна структура індексно-послідовного файлу залежить від типу носія і операційної системи, яка підтримує роботу з таким файлом. У бібліотеці функцій введення / виведення мови Сі відсутні функції, призначені для роботи з файлами індексно-послідовної організації, і тому завдання обробки таких файлів покладається на програміста або на використання готових бібліотек незалежних розробників. У Таблиця 7.4 наведено характеристики, переваги та недоліки індексно-послідовної організації файлу.

Таблиця 7.4 Характеристики індексно-послідовної організації файлів

Особливості	Переваги	Недоліки
• забезпечує швидкий доступ до даних • записи області даних і індексу мають фіксовану довжину • індекс індексної області зберігає адресу першого запису в групі області даних • для послідовної обробки записів файлу досить здійснити лівий спадний обхід дерева індексів в індексному файлі, а в досягнутому кінцевому вузлі індексу послідовно обробити записи відповідної групи області даних • для прямого доступу до запису із заданим ключем спочатку здійснюється пошук по дереву відповідної групи (вузла) в індексній області, потім - послідовний пошук потрібного запису всередині групи області даних. Можлива і послідовна обробка решти файлу, починаючи з цього запису • індексно-послідовні файли добре працюють в тих ситуаціях, коли необхідна зручність обробки послідовного доступу при високій продуктивності прямого доступу до даних файла	• ключовим удосконаленням є те, що індекси малі і можуть бути швидко знайдені, що дозволяє отримати доступ тільки до невеликих за розміром груп даних • підтримка програм, що вимагають як пакетного, так і інтерактивного режиму обробки • записи можуть бути доступні як послідовно, так і в довільному порядку • дозволяє проводити оновлення записів в тому ж файлі, без перезапису на нове місце	• індексно послідовні файли можуть бути збережені тільки на пристроях з прямим доступом (магнітні диски і флешки) • потрібно мати додатковий простір для зберігання індексів • збільшуються накладні витрати на їх обробку (при порівнянні з прямим доступом) • обробка цих файлів є більш складною, ніж обробка послідовних файлів і файлів з прямим доступом • підтримка записів тільки фіксованої довжини

7.6 Робота з файлами у мові С

З огляду на те, що функції роботи з файлами детально вивчаються в курсі програмування на мові С, тут тільки в загальних рисах розглянемо операції над файлами.

У мові С відсутні вбудовані в мову засоби введення і виведення. Всі операції, пов'язані з введенням / виведенням виконуються за допомогою функцій стандартної бібліотеки С. Бібліотечні функції введення / виведення дають можливість читати з файлів і з пристроїв і писати дані в файли і пристрої. Вони підтримують три рівня введення-виведення: введення / виведення потоку;

введення / виведення нижнього рівня; введення / виведення для консолі і порту.

Розглянемо коротко перші два рівня.

7.6.1 Введення / виведення потоку

При введенні-виведенні потоку всі дані розглядаються як потік байтів. З точки зору програміста потік - це або файл, або пристрій (клавіатура, екран і т.п.). Функції введення / виведення дозволяють виконувати над потоками наступні дії:

- відкриття і закриття потоку;
- читання з потоку і запис в потік;
- буферизацію потоку.

Відкриття та закриття потоку. Відкриття здійснюється перед виконанням операцій введення / виведення. Функція відкриття файлу *fopen()* повертає в програму покажчик потоку (покажчик на структуру типу *FILE*) або *NULL* при неможливості відкрити файл. Покажчик потоку використовується потім усіма іншими функціями введення / виведення. Після завершення обробки потік закривається функцією *fclose()*.

Читання з потоку і запис в потік. Вони виконуються різними способами. Можливі форматні і неформатні введення і виведення. При *форматному введенні* (функціями типу *scanf()* або *fscanf()*) послідовність символів зчитується з потоку, перетворюється в заданий формат представлення даних і розміщується в оперативній пам'яті по заданих адресах (іменах зміних). В одній і тій же функції можуть бути задані декілька форматів, що забезпечує введення даних досить складної структури. При *форматному виведенні* (функції типу *printf*() або *fprintf()*) виконуються зворотні дії: дані, що зберігаються в різних областях пам'яті (змінних програми), перетворюються із заданих типів представлення даних в символьний формат і виводяться в потік як послідовність байтів. При цьому в послідовність може додаватися програмістом деяка додаткова текстова інформація і керуючі символи.

Буферизація потоку. Полягає в тому, що при відкритті потоку система управління файлами операційної системи пов'язує з потоком область пам'яті, звану *буфером*. При операції читання блок даних з файлу (пристрою) спершу розміщується у вхідному буфері, і надалі, при потребі дані читаються у програму

із вхідного буфера, а не із файлу. Коли дані із вхідного буфера оброблені (буфер стає пустим), у буфер із вхідного потоку (файлу) передається наступний блок даних. При операціях запису дані із програми спершу розміщуються у вихідному буфері, а далі вміст буфера переписується у файл чи пристрій (буфер вивантажується) по:

- мірі заповнення вихідного буфера;
- при закритті потоку,
- успішному завершенню програми, якщо потік не був попередньо цією програмою самостійно закритий.

Буферизація підвищує ефективність введення / виведення, так як операційна система за одну операцію передає великі блоки даних.

Операції введення / виведення можуть виконуватися в *текстовому* або *бінарному режимі*, який встановлюється при відкритті потоку. У текстовому режимі при введенні даних комбінація керуючих символів «повернення каретки» (*CR*, код 13) і «новий рядок» (*LF*, код 10) перетворюється в один символ нового рядка *'\n'* (*LF*, код 10). При запису в потік здійснюється зворотне перетворення, тобто символ нового рядка *'\n'* замінюється послідовністю символів *CR* і *LF*.

7.6.2 **Введення / виведення нижнього рівня**

Функції введення/виведення нижнього рівня, на відміну від функцій введення / виведення потоку, виконують тільки небуферізовані і неформатовані введення / виведення. Вони не входять до складу бібліотеки введення / виведення мови С і прямо викликають залежні засоби операційної системи, тому програми, що використовують такі функції, можуть не переноситися на інші платформи. Користування ними не рекомендується без необхідного технічного обґрунтування!

При відкритті файлу з ним зв'язується дескриптор файлу, який потім використовується всіма іншими функціями. Введення і виведення здійснюються блоками, довжина блоку і адреса буфера введення-виведення (області пам'яті в програмі) задаються при звертанні до відповідної функції. Введення / виведення, як і при введенні / виведенні потоку, може проводитися в текстовому або в двійковому режимі відкриття файлу.

7.7 Контрольні завдання і запитання

1. У файлі створити неупорядковану таблицю успішності студентів. Ввести файл в оперативну пам'ять, відсортувати по прізвищам методом вибору і записати назад у файл.

2. У файлі створити неупорядковану таблицю успішності студентів. Ввести файл в оперативну пам'ять, відсортувати по прізвищам методом вставки і записати назад у файл.

3. У файлі створити неупорядковану таблицю успішності студентів. Ввести файл в оперативну пам'ять, відсортувати по прізвищам швидким сортуванням і записати назад у файл.

4. У файлі створити неупорядковану таблицю успішності студентів. Ввести файл в оперативну пам'ять, відсортувати по середньому балу методом Шелла і записати назад у файл.

5. У файлі створити неупорядковану таблицю успішності студентів. Ввести файл в оперативну пам'ять, відсортувати по середньому балу бульбашковим сортуванням і записати назад у файл.

6. У файлі створити неупорядковану таблицю успішності студентів. Ввести файл в оперативну пам'ять, відсортувати по середньому балу блоковим сортуванням і записати назад у файл.

8 ДЕРЕВА

8.1 Поняття дерева і термінологія

У цій темі ми розглянемо такі високорекурсивні структури даних, як дерева. Вони можуть використовуватися для відображення ієрархії системи управління, організації та моделювання процесів прийняття рішень. У вигляді дерева зручно зберігати структуру організації компанії або каталог запчастин, з яких складається, скажімо, автомобіль.

Граф називається **деревом**, якщо він зв'язний і ациклічний (тобто не містить циклів). Отже, дерево складається з *вершин* (вузлів), які містять дані і з'єднуються *гілками* [1]. Гілки зображують у вигляді ліній, що ведуть від батьківської вершини до дочірньої (рисунок нижче).

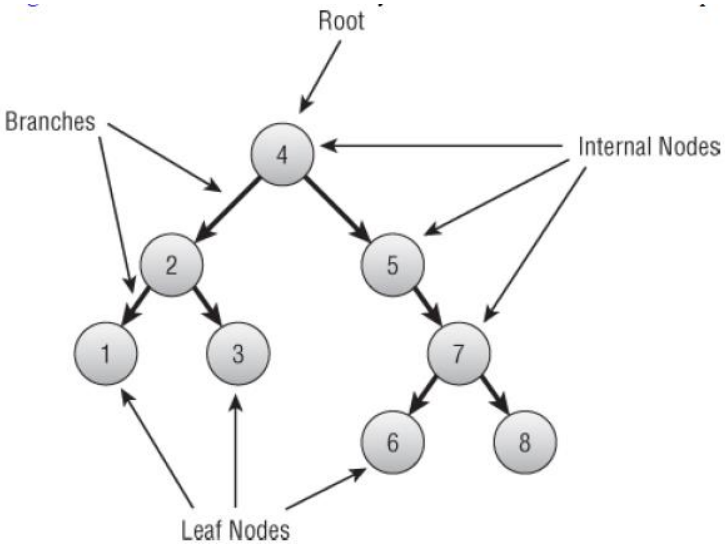


Рисунок 8.1 Деревовидна структура даних

У кожної вершини, за винятком кореневої, є одна батьківська вершина. Дві вершини із загальним батьком іноді називають *сестрами*, дочірні вершини дочірніх вершин - *нащадками*, а батьківські вершини батьківських вершин - *предками*.

Залежно від типу дерева біля вершини може бути різна кількість нащадків. Це число називається *ступенем вершини*. Його максимальне значення визначає *степінь дерева* в цілому. Наприклад, дерево другого степеня є *бінарним*, у кожної його вершини може бути максимум два дочірніх вузла. Вершина, яка не має дочірніх вузлів, називається *термінальною (листовою)*; якщо вершина містить хоча б один дочірній вузол – вона називається *внутрішньою*.

У дерев в програмуванні корінь зазвичай розташовується вгорі, а гілки

спрямовані вниз (Рисунок 8.1). Згідно всього вищесказаного, можна стверджувати, що деревом є:

- вершина лише з одним коренем;
- коренева вершина, поєднана гілками з одним або декількома меншими деревами.

Рівень (глибина) вершини — це відстань від вершини до кореня. Для кореня вона буде дорівнювати 0.

Гілка дерева – з'єднання від одної вершини до іншої.

Висота вершини — число гілок на найдовшому низхідному шляху від заданої вершини до листа, тобто до низу дерева. Висота самого дерева дорівнює висоті кореневої вершини.

Піддеревом дерева *T* з коренем *R* є вершина *R* з усіма її нащадками. Так, на Рисунок 8.1 піддерево з коренем у вершині 5 - це дерево, що містить вершини 5, 7, 6 і 8.

В *упорядкованому дереві* розташування дочірніх вершин має значення. Багато алгоритмів по-різному розглядають ліву і праву дочірні вершини. У *невпорядкованому* дереві розташування дочірніх вершин не грає ролі. (Як правило, дерева впорядковані, навіть якщо це не особливо важливо для алгоритму. Вся справа в тому, що дочірні вершини або гілки зберігаються в масиві або іншій структурі даних, яка передбачає сортування.)

Для будь-яких двох вершин *перший (найменший)* загальний предок - це вузол-предок, розташований найближче до них. Щоб його знайти, потрібно пройти шлях від однієї з вершин до кореня до знаходження спільного батьківського вузла з другою вершиною. На Рисунок 8.1 для вершин 3 і 5 першим спільним предком буде корінь 4. Слід зазначити, що таким предком може виявитися і одна з двох вершин. На тому ж малюнку для вершин 5 і 6 це буде 5.

Зверніть також увагу, що між будь-якими двома вершинами дерева існує *унікальний шлях*, який не слідує по одній гілці дерева двічі. Він бере початок з першої вершини, проходить вгору по дереву до найменшого спільного предка, а потім спускається до другої вершини.

У *завершеному дереві* (Рисунок 8.2) внутрішня вершина або взагалі не має дочірніх вершин, або їх кількість дорівнює ступеню дерева. Наприклад, в

завершеному бінарному дереві у кожного вузла може бути два дочірніх або жодного. Дерево, зображене на Рисунок 8.1, незавершене, оскільки у вершини 5 є тільки одна дочірня вершина 7.

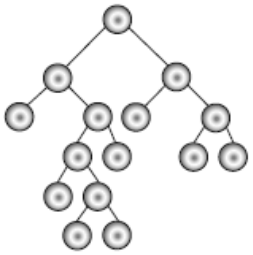


Рисунок 8.2 Приклад завершеного дерева

У *повного дерева* кожен рівень є повним, за винятком, можливо, нижнього, де всі вершини зсуваються вліво. На Рисунок 8.3 показано повне бінарне дерево. Воно незавершене, оскільки третя вершина на другому рівні має лише одну дочірню.

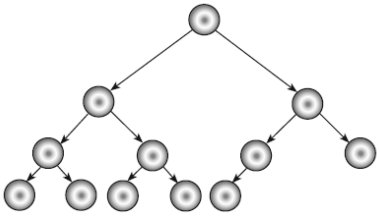


Рисунок 8.3 Приклад повного дерева

Ідеальне дерево — завершене дерево, все листя якого знаходиться на одному рівні, заповнюючи рівень повністю (Рисунок 8.4). Іншими словами, в ньому присутні всі вершини, допустимі для його висоти

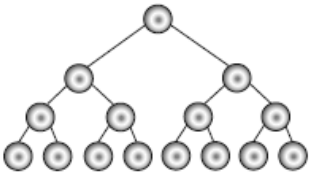


Рисунок 8.4 Приклад ідеального дерева

Всі згадані вище терміни для зручності зведемо в єдину таблицю нижче.

Таблиця 8.1 Терміни, що застосовуються при описі дерев

Термін	Значення
Брат (сестра)	Дві вершини дерева, що мають загального батька
Вершина (вузол)	Об'єкт дерева, що містить дані і з'єднаний з іншими такими ж об'єктами за допомогою гілок

Термін	Значення
Вершина внутрішня	Вершина, яка містить як мінімум одну дочірню вершину
Вершина дочірня	Вершина, поєднана з батьківської вершиною і, як правило, розташована на рівень нижче її
Вершина термінальна (листова)	Вершина, яка не містить дочірніх вершин
Гілка	Частина дерева, що з'єднує вершини
Висота	Стосовно до вершини – це найдовший шлях, що веде від неї вниз по дереву до термінальної (листової) вершини. Стосовно до дерева – те саме, що і висота кореня
Глибина (рівень) вершини	Це відстань від вершини до кореня. У корені вона буде дорівнювати 0
Дерево бінарне	Дерево другого ступеня, кожна вершина якого має не більше двох дочірніх вузлів
Дерево повне	Дерево, в якому кожен рівень є повним, крім, можливо, самого нижнього, на якому всі вершини повинні бути зсунуті вліво
Дерево ідеальне	Повне дерево, в якому все листя знаходяться на одному рівні, заповнюючи рівень повністю
Дерево завершене	Дерево, в якому кожна вершина містить нуль дочірніх вершин або їх кількість дорівнює ступеню дерева
Дерево впорядковане	Дерево, для якого важливе сортування всіх дочірніх вузлів
Корінь	Сама верхня вершина дерева, яка не має батьківських вершин
Лист	Термінальна вершина
Піддерево	Вершина дерева з усіма її нащадками

Термін	Значення
Нащадок	Дочірні вершини дочірніх вершин, тощо. Вниз по дереву до листя
Предок	Батьківська вершина батьківської вершини, тощо. Вгору по дереву до кореня
Предок перший (найменший) загальний	Один і той же предок для двох будь-яких вершин, який розташований до них найближче
Батько	Вершина, поєднана з дочірніми вершинами за допомогою гілок і, як правило, розташована на рівень вище них. Кожен вузол дерева, крім кореня, має тільки одного батька
Ступінь	Стосовно до вершини - загальна кількість її дочірніх вершин. По відношенню до дерева - максимальний ступінь будь-якої з його вершин
Рівень	Відстань між даною вершиною і коренем дерева

8.2 Типи дерев

Існує шість основних типів дерев:

- дерева загального типу;
- ліси;
- бінарні дерева;
- бінарні дерева пошуку;
- дерева виразів;
- турнірні дерева.

8.2.1 **Загальні дерева**

Загальні дерева являють собою структури даних, які зберігають елементи в ієрархічному порядку. Найвища вершина дерева є кореневою і кожна вершина, за винятком кореня, має батька. Вершина в загальному дереві (за винятком термінальних вершин) може мати нуль або більше піддерев. Загальні дерева, які мають 3 дочірніх дерева для кожної вершини називаються потрійними (тернарними) деревами. Кількість піддерев в загальному дереві для будь-якої вершини може бути змінною. Наприклад, вершина може мати одне піддерево, в той час як деякі інші вершини можуть мати 3, 4 і більше піддерев.

Хоча загальні дерева можуть бути представлені у вигляді абстрактних типів, завжди існує проблема, коли інше піддерево додається до вершини, яка вже має максимальну кількість піддерев. Навіть алгоритми пошуку, обходу, додавання і видалення вершин стають набагато складнішими, оскільки для кожної вершини є вже не дві можливості, а безліч.

Для подолання цих труднощів, загальні дерева перетворюються в бінарні, операції з якими виконуються набагато простіше. Загальне дерево при перетворенні в бінарне не може в кінцевому вигляді бути добре або повністю сформованим, але переваги такого перетворення дозволяють програмісту використовувати алгоритми для процесів, які використовуються для бінарних дерев з незначними змінами.

8.2.2 **Ліс**

Ліс є незв'язним об'єднанням дерев. Безліч непересічних дерев (або ліс) отримують шляхом видалення кореня і ребра, що з'єднують кореневу вершину до вершин на рівні 1.

Ми вже бачили, що кожна вершина дерева є коренем деякого піддерева. Таким чином, всі піддерева безпосередньо під вершиною утворюють ліс, як показано на малюнку нижче.

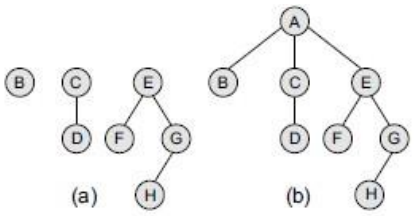


Рисунок 8.5 Ліс і його відповідне дерево

Ліс також може бути визначений як упорядкований набір з нуля або більше загальних дерев. У той час як загальне дерево повинно мати корінь, ліс може бути і порожнім, тому що він за визначенням є множиною, а множини можуть бути порожні.

Ми можемо перетворити ліс в дерево, додавши одну вершину в якості кореневої вершини дерева. Наприклад, ліва частина Рисунок 8.5 показує ліс, а права частина малюнка показує відповідне цьому лісу дерево.

8.2.3 **Бінарні дерева**

У бінарному дереві, кожна вершина має 0, 1, або максимум 2 нащадків.

Вершина, яка має нуль нащадків називається листовою або термінальною вершиною. Кожна вершина бінарного дерева містить наступні елементи:

- - елемент даних;
- - лівий покажчик, який вказує на лівого нащадка;
- - правий покажчик, який вказує на правого нащадка.

Адреса кореневої вершини (*root*) міститься в дескрипторі дерева. Якщо *root* = *NULL*, то це означає, що дерево порожнє. Рисунок нижче показує бінарне дерево:

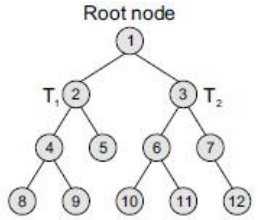


Рисунок 8.6 Бінарне дерево

На малюнку вище, *R (root)* є кореневою вершиною і містить два піддерева T_1 і T_2, звані лівим і правим піддеревом *R*.

Зверніть увагу, що ліве піддерево кореневої вершини складається з вершин: **2, 4, 5, 8, і 9**. Аналогічним чином, праве піддерево кореневої вершини складається з вузлів: **3, 6, 7, 10, 11** та **12**.

У дереві, коренева вершина 1 має два нащадка: 2 і 3. Вершина 2 має два нащадка: 4 і 5. Вершина 4 має два нащадка: 8 і 9. Вершина 5 не має нащадків і є листом дерева. Вершина 3 має два нащадка: 6 і 7. Вершина 6 має два нащадка: 10 і 11. Нарешті вершина 7 має тільки одного нащадка: 12. Вершина 12 є листом.

Бінарне дерево є рекурсивною структурою даних по визначенню, так як кожен вузол в дереві містить ліве піддерево і праве піддерево. Навіть окремі листи дерева містять пусте ліве піддерево і пусте праве піддерево. Дивлячись на Рисунок 8.6, можна побачити, що вершини 5, 8, 9, 10, 11, і 12 не мають нащадків, тобто можна сказати, що їх нащадками є пусті піддерева.

***Повне бінарне дерево*.** Повне бінарне дерево являє собою бінарне дерево, яке задовольняє двом умовам. По-перше, в повному бінарному дереві, кожен рівень, за винятком, можливо, останнього повністю заповнений. По-друге, всі вузли останнього рівня зсуваються в ліве положення, наскільки це можливо, як показано на малюнку нижче.

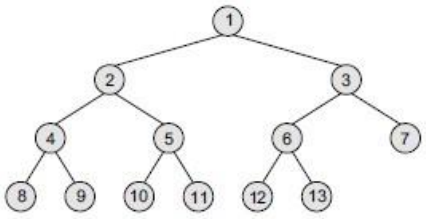


Рисунок 8.7 Повне бінарне дерево

У повному бінарному дереві T_n, існує рівно n вершин і рівень r в T може мати не більше 2^r вершин.

На малюнку вище, рівень *0* має $2^0 = 1$ вершину, рівень *1* має $2^1 = 2$ вершини, рівень *2* має $2^2 = 4$ вершини, рівень *3* має *6* вершин, що меньше, ніж максимум $2^3 = 8$ вершин.

На цьому ж малюнку показано, що дерево *T* має рівно 13 вершин. Вони були навмисно помічені від 1 до 13 таким чином, щоб можна було легко знайти батьківську вершину, праву дочірню вершину, і ліву дочірню вершину для даної вершини. Алгоритм нумерації вершин може бути визначений таким чином: *якщо K є номером батьківської вершини, то номер його лівого нащадка може бути розрахований як 2*K, а номер його правого нащадка може бути розрахований як 2*K+1*. Наприклад, нащадки вершини *4* матимуть номери *8 = (2*4)* і *9 = (2*4+1)*. Так само, номер батьківської вершини нащадка *K* може бути обчислений як ціла частина (антьє) *[K/2]*. Для тієї ж вершини *4*, його батьківська вершина може бути обчислена як *4/2 = 2*.

Висота повного бінарного дерева *T*, яке має рівно *n* вершин визначається як:

$$H_n = [log_2(n+1)] \qquad (8.1)$$

Завершене (розширене) бінарне дерево. Бінарне дерево *T* називається завершеним (розширеним) бінарним деревом, якщо кожна вершина в дереві не має жодного нащадка або має рівно двох нащадків. На Рисунку 8.8 показано, як звичайне бінарне дерево перетворюється в завершене.

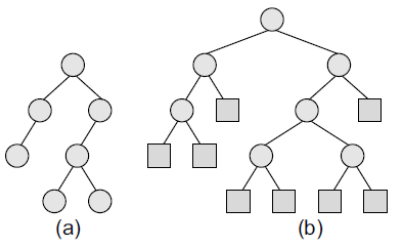


Рисунок 8.8 (a) Бінарне дерево (b) Завершене бінарне дерев

У завершеному дереві вершини, що мають двох нащадків, називаються

внутрішніми вершинами, а вершини, що не мають нащадків - називаються зовнішніми вершинами. На малюнку вище внутрішні вершини позначені колами, а зовнішні вершини позначені квадратами.

Для перетворення бінарного дерева в завершене, кожне пусте піддерево замінюється новою вершиною. Початкові вершини в дереві є внутрішніми вершинами, а додані вершини називаються зовнішніми вершинами.

8.2.4 Бінарне дерево пошуку.

Бінарне дерево пошуку, також відоме як впорядковане бінарне дерево, являє собою варіант бінарного дерева, в якому вершини розташовані в певному порядку, згідно впорядкування своїх ключів. Концепція бінарних дерев пошуку і різних операцій, виконуваних на них, буде обговорюватися далі в курсі.

8.2.5 Дерево розбору (дерево синтаксичного аналізу).

Синтаксичний аналіз (жарг. *парсинг* (запозичене з англ. терміну *parsing*)) в інформатиці - процес зіставлення лінійної послідовності лексем (слів, токенів) природної або формальної мови з його формальною граматикою. Результатом зазвичай є дерево розбору (синтаксичне дерево). Зазвичай застосовується спільно з лексичним аналізом. Синтаксичний аналізатор (жарг. *парсер*) - це програма або частина програми, що виконує синтаксичний аналіз.

В ході розбору початковий текст перетворюється в структуру даних, зазвичай - в бінарне дерево, зване деревом розбору, яке відображає синтаксичну структуру вхідної послідовності і добре підходить для подальшої обробки.

Приклад 8.1

Розглянемо парсинг наступного алгебраїчного виразу:

$$Exp = (a - b) + (c * d) \qquad (8.2)$$

Цей вираз може бути представлено за допомогою дерева розбору, як показано на малюнку нижче:

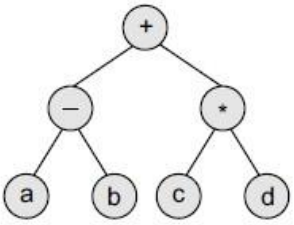


Рисунок 8.9 Дерево розбору, відповідне виразу (8.2)

Приклад 8.2

Побудувати відповідне бінарне дерево розбору для виразу

$$Exp = a + b / c * d – e \qquad (8.3)$$

Рішення. Використовуючи пріоритет операцій, перепишемо цей вираз у вигляді, в якому він буде виконуватися:

$$Exp = ((a + ((b/c)*d)) – e)$$

Звідси відразу можна побудувати відповідне йому бінарне дерево розбору:

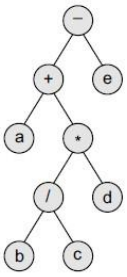


Рисунок 8.10 Дерево розбору для виразу (8.3)

8.2.6 Турнірні дерева (дерево прийняття рішень).

Хай відомо, що в турнірі, скажімо, з шахів, бере участь одночасно *n* гравців. Для того, щоб виявити переможця серед всіх цих гравців, грається попарно кілька партій одночасно протягом одного туру. Як правило, проводиться кілька турів.

У 1-му турі, грається ряд партій, в яких два гравці грають один проти одного. Кількість партій, які будуть грати в 1-му турі, буде залежати від кількості гравців. Наприклад, якщо є 8 гравців, що беруть участь в турнірі, в 1-му турі буде зіграно 4 партії. Тоді у 2-му турі, переможці 1-го туру будуть грати один проти одного. Аналогічним чином, в 3-му турі, переможці 2-го туру будуть грати один проти одного, і той, хто виграє 3-й тур - оголошується переможцем. Турнірні дерева використовуються для подання цієї концепції.

У турнірному дереві (також званому деревом прийняття рішень), кожна зовнішня вершина являє собою гравця, а кожна внутрішня вершина являє переможця партії між гравцями, представленими її дочірніми вершинами. Ці турнірні дерева виявляють переможця на кожному рівні (турі) і дозволяють виявити Головного переможця змагань. У турнірному дереві, коренева вершина визначає переможця, як на рисунку нижче:

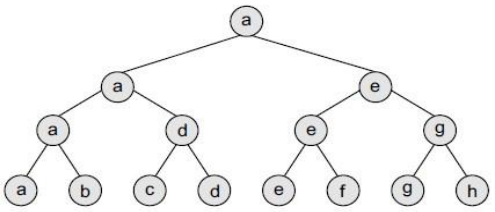


Рисунок 8.11 Турнірне дерево. Літерами від *a* до *h* позначені гравці

8.3 Властивості бінарного дерева

Бінарні дерева використовуються в багатьох алгоритмах. По-перше, вони легкі для розуміння, а по-друге, більшість завдань базується на двійковому виборі. Ось основні властивості бінарних дерев.

1. Кількість гілок B в бінарному дереві з N вершин знаходиться за формулою:

$$B = N - 1. \qquad (8.4)$$

2. Кількість вершин N в ідеальному бінарному дереві висотою H розраховується наступним чином:

$$N = 2^{H+1} - 1. \qquad (8.5)$$

3. Якщо в ідеальному бінарному дереві N вершин, його висота визначається як:

$$H = log_2(N + 1) - 1. \qquad (8.6)$$

4. Кількість термінальних вершин L в ідеальному бінарному дереві висотою H обчислюється за формулою

$$L=2^H, \qquad (8.7)$$

а кількість внутрішніх вершин I - за формулою

$$I = N - L = 2^H - 1. \qquad (8.8)$$

Звідси випливає, що в ідеальному бінарному дереві кількість термінальних і внутрішніх вершин практично однакова:

$$I = L - 1. \qquad (8.9)$$

8.4 Побудова бінарного дерева із загального дерева

Для використання алгоритмів обробки тільки бінарного дерева необхідно вміти перетворювати дерева загального вигляду до *бінарних*. Правила перетворення загального дерева в бінарне дерево наведені нижче.

Правило 1: Корінь бінарного дерева = корінь загального дерева

Правило 2: У будь-якій вершині дерева відсікаються всі гілки, крім крайньої

лівої, відповідної старшим дочірнім вершинам.

Правило 3: З'єднуються горизонтальними лініями всі дочірні вершини одного батька на кожному рівні.

Правило 4: Старшою (лівою) дочірньою вершиною в будь-якій вершині отриманої структури буде вершина, яка перебуває під даною (якщо вона є).

Приклад 8.3 Перетворити в бінарне дерево дерево наступної структури:

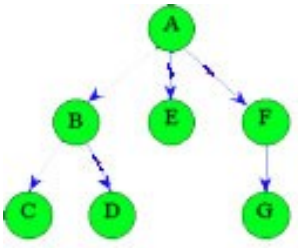


Рисунок 8.12 Дерево, яке слід перетворити до бінарного

Рішення.

Крок 1: Вершина А є коренем загального дерева, вона так і залишиться коренем бінарного дерева відповідно до Правила 1.

Крок 2: Відсікаємо всі гілки в вихідному дереві, крім крайньої лівої, відповідно до Правила 2.

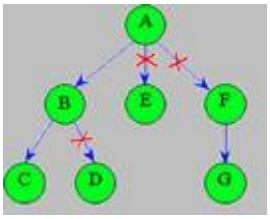


Рисунок 8.13 Крок 2

Крок 3: З'єднуємо горизонтальними лініями всі дочірні вершини одного з батьків, відповідно до Правила 3.

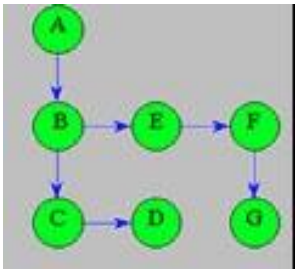


Рисунок 8.14 Крок 3

Крок 4: Перемальовувати отриману структуру відповідно до Правила 4.

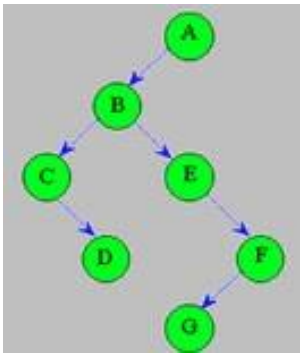


Рисунок 8.15 Крок 4

Дерево після четвертого кроку і буде остаточним бінарним деревом вихідного дерева.

8.5 Представлення бінарного дерева у пам'яті комп'ютера

У зв'язаному поданні бінарного дерева, кожна вершина буде складатися з трьох частин: елемент даних, покажчик на лівий вузол і покажчик на правий вузол. Таким чином, в мові програмування С бінарне дерево будується зі структурою вершини, вказаною нижче на прикладі:

Приклад 8.4 Структура вершини бінарного дерева у мові Сі

```
struct BTnode {
    struct BTnode *left;    /* адреса лівого нащадка */
    int data;               /* дані, що зберігаються на вершині */
    struct BTnode *right;   /* адреса правого нащадка */
    };
```

Кожне бінарне дерево має свій дескриптор, що містить покажчик *root* на кореневий елемент (самий верхній елемент) дерева. Якщо *root = NULL*, то дерево є порожнім. Розглянемо схематичне представлення бінарного дерева, наведеного на рисунку нижче:

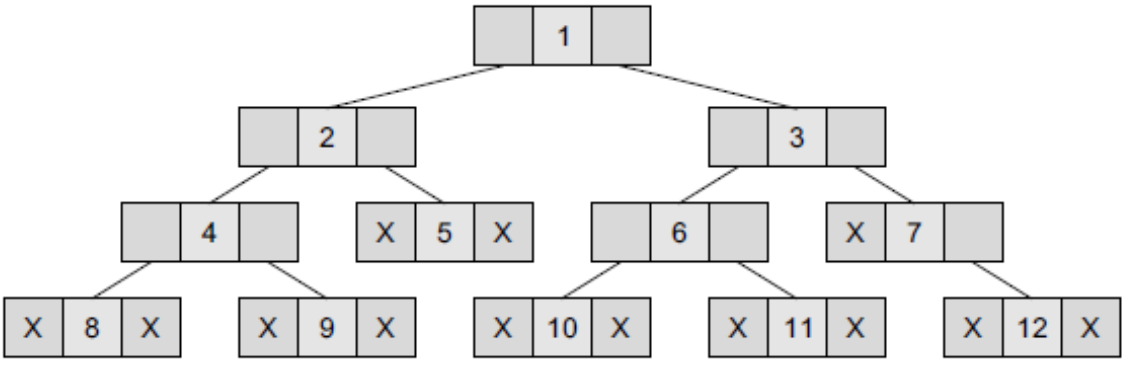


Рисунок 8.16 Схематичне представлення бінарного дерева в пам'яті

На малюнку, ліва частина в вершинах використовується, щоб вказати на ліву дочірню вершину, тобто в ній зберігається адреса лівої дочірньої вершини. Середня частина вершини використовується для зберігання даних. І, нарешті,

права частина використовується, щоб вказати на праву дочірню вершину, тобто в ній зберігається адреса правої дочірньої вершини. Порожні піддерева представлені з використанням символу *X* (значення *NULL*).

Якщо взяти бінарне дерево на рисунку нижче,

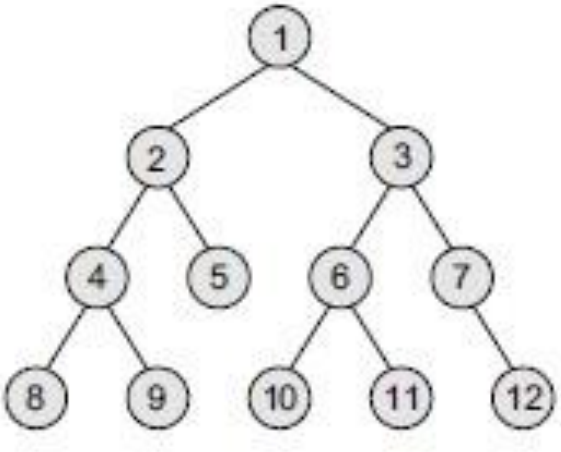


Рисунок 8.17 Початкове бінарне дерево

то в оперативній пам'яті комп'ютера воно може бути представлене з використанням зв'язаного списку таким чином (тут -1 позначає пустий покажчик):

		LEFT	DATA	RIGHT
ROOT 3	1	–1	8	–1
	2	–1	10	–1
→	3	5	1	8
	4			
	5	9	2	14
	6			
	7			
	8	20	3	11
	9	1	4	12
	10			
	11	–1	7	18
	12	–1	9	–1
	13			
	14	–1	5	–1
→	15			
15 AVAIL	16	–1	11	–1
	17			
	18	–1	12	–1
	19			
	20	2	6	16

Рисунок 8.18 Представлення бінарного дерева зв'язним списком

Приклад 8.5 Побудувати в пам'яті бінарне дерево, яке зберігає імена членів родини:

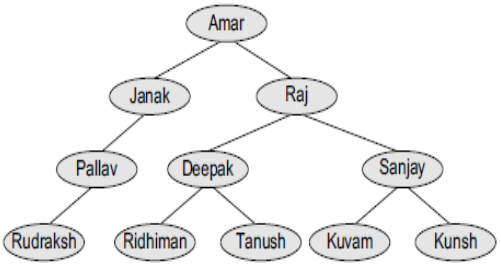
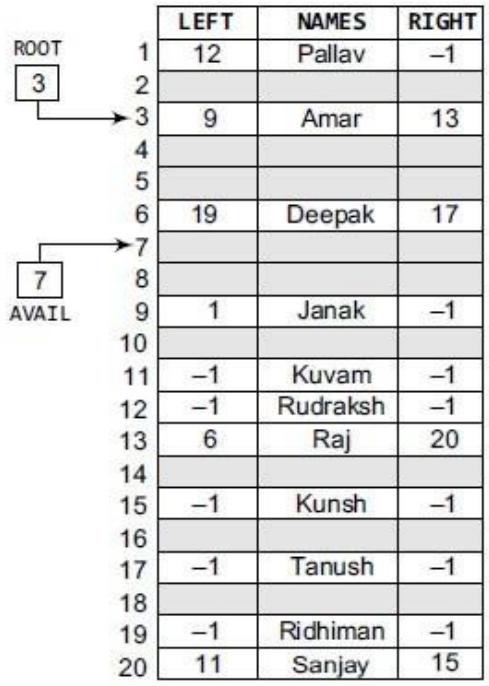


Рисунок 8.19 Генеалогічне дерево родини



	LEFT	NAMES	RIGHT
1	12	Pallav	–1
2			
3	9	Amar	13
4			
5			
6	19	Deepak	17
7			
8			
9	1	Janak	–1
10			
11	–1	Kuvam	–1
12	–1	Rudraksh	–1
13	6	Raj	20
14			
15	–1	Kunsh	–1
16			
17	–1	Tanush	–1
18			
19	–1	Ridhiman	–1
20	11	Sanjay	15

Рисунок 8.20 Представлення генеалогічного дерева родини в пам'яті

8.6 Використання бінарних дерев

Бінарні дерева використовуються для зберігання простих і складних даних. Під простими даними ми розуміємо числові і символьні значення, а під складними даними розуміємо дані типу масив або структура. Бінарні дерева широко використовуються в інформаційних технологіях для:

- подання інших типів структурованих даних, таких як хеш-таблиці, множини, графи і карти чого-небудь;
- представлення самобалансуючих дерев. Наприклад, самобалансуючі червоно-чорні дерева використовується в задачах планування роботи ядра операційної системи на багатоядерних процесорах;
- представлення збалансованого варіанту бінарного дерева - *B-дерева*. B-дерева часто використовуються для зберігання деревовидних структур даних на диску. Вони широко застосовуються і для зберігання індексів проіндексованих файлів;
- зберігання вторинних індексів в базах даних, де наявність додаткового індексу прискорює операції вибору даних;
- побудови компіляторів;
- проектування баз даних;
- файлової системи каталогів операційних систем;

- зберігання інформації і пошуку в таблицях символів.

8.7 Траверс (обхід) бінарних дерев

Траверсом (обходом) бінарного дерева називається систематична процедура обробки кожної вершини в дереві рівно один раз. На відміну від лінійних структур даних, в яких елементи проходяться послідовно, дерево являє собою нелінійну структуру даних, в якій обхід елементів дерева може відбуватися *різними шляхами*. Існують різні алгоритми для обходу дерев. Ці алгоритми розрізняються залежно від порядку відвідування вершин дерева. У цьому пункті ми обговоримо ці алгоритми.

8.7.1 Обхід в прямому порядку

При такому обході алгоритм спочатку обробляє вершину, потім її лівий дочірній вузол, а після правий. Алгоритм прямого обходу також відомий як алгоритм *NLR (Node-Left-Right)*. Розглянемо дерево, представлене на малюнку нижче, і пояснимо хід виконання процедури обходу:

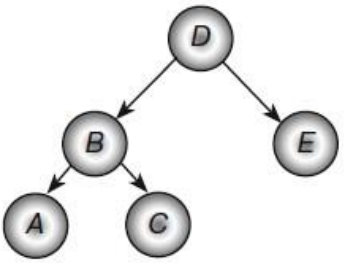


Рисунок 8.21 Початкове дерево для обходу в прямому порядку

Насамперед алгоритм звернеться до кореня і виведе значення D. Потім він переміститься до лівої дочірньої вершині кореня, виведе B і розгляне вже її ліву дочірню вершину, тобто A. Більше дочірніх вершин немає, тому алгоритм повернеться в вершину B і пройде до його правої дочірньої вершині, в нашому випадку до C. У неї теж немає нащадків, отже, відбудеться ще одне повернення до вершини B. Раз дочірніх вершин у вершини B більше немає, то програма підніметься вгору по дереву до кореня D і пройде до його правої дочірньої вершини E. Не знайшовши у неї нащадків, алгоритм знову повернеться до кореня, констатує відсутність будь-яких інших дочірніх вершин і завершить обхід. В результаті порядок обходу буде виглядати так: ***D, B, A, C, E***.

Алгоритм вивчає (проходить) вершини в одному порядку, а результати своєї роботи виводить в іншому. У наступному списку представлені кроки, які

виконуються під час прямого обходу для дерева, зображеного на Рисунок 8.21:

1. *Пройти D.*
2. *Вивести D.*
3. *Пройти B.*
4. *Вивести B.*
5. *Пройти A.*
6. *Вивести A.*
7. *Пройти B.*
8. *Пройти C.*
9. *Вивести C.*
10. *Пройти B.*
11. *Пройти D.*
12. *Пройти E.*
13. *Вивести E.*
14. *Пройти D.*

Рекурсивна реалізація даного алгоритму представлена нижче, у вигляді програми:

Приклад 8.6 Алгоритм прямого обходу бінарного дерева

```
typedef struct BTnode {
        struct BTnode *left;
        int data;
        struct BTnode *right;
};

void preOrder (BTnode* tree) {
        if (tree) {  /* якщо вершина існує */
                printf ("%d\n", tree-> data);/* обробка даних на вершині дерева */
                preOrder (tree -> left);    /* "пірнаємо" в ліве піддерево вузла */
                preOrder (tree -> right); /* "пірнаємо" в праве піддерево вузла */
        }
        return;
}
```

Обхід в прямому порядку використовується не тільки для бінарних дерев, а й для тих, що мають більш високу ступінь. Незмінним залишається загальне правило: перш за все вивчається сама вершина, а потім її дочірні вершини, починаючи з лівої.

8.7.2 **Симетричний обхід**

При симетричному обході алгоритм обробляє ліву дочірню вершину, потім її саму і тільки після цього праву дочірню вершину. Алгоритм симетричного обходу також відомий як алгоритм *LNR (Left-Node-Right).*

Працюючи з деревом, зображеним на Рисунок 8.21, програма почне рухатися з кореня, переміститься відразу до лівої дочірньої вершини B, а через неї до лівої дочірньої вершини A. У цій вершині вже немає лівого нащадка, тому програма виведе A і, не знайшовши правого нащадка , повернеться до батьківської вершини B, щоб вивести її. Після цього алгоритм пройде до правої дочірньої вершини C, у якій також відсутній лівий нащадок. Вивівши C, програма упевниться, що немає і правого нащадка, і знову повернеться до батьківської вершини B. Оскільки робота з лівою частиною дерева вже закінчена, алгоритм підніметься до кореня D, виведе його і звернеться до правої частини з дочірньою вершиною E. У цієї вершини немає лівого дочірнього вузла, значить, алгоритм зразу виведе E і, не знайшовши правого дочірнього вузла, повернеться до кореня D. Підсумковий порядок симетричного обходу буде таким: ***A, B, C, D, E***.

Слід звернути увагу на те, що виведені (оброблені) вершини розташовуються у відсортованому порядку. Впорядковані дерева якраз і будуються таким чином, щоб симетричний обхід дав на виході відсортовані значення.

У наступному списку представлені кроки, які виконуються під час симетричного обходу для дерева, зображеного на Рисунок 8.21:

1. *Пройти D.*
2. *Пройти B.*
3. *Пройти A.*
4. *Вивести A.*
5. *Пройти B.*
6. *Вивести B.*
7. *Пройти C.*
8. *Вивести C.*
9. *Пройти B.*
10. *Пройти D.*
11. *Вивести D.*

12. *Пройти E.*
13. *Вивести E.*
14. *Пройти D.*

Рекурсивна реалізація описаного алгоритму представлена нижче у вигляді C - програми [2]:

Приклад 8.7 Алгоритм симетричного обходу бінарного дерева

```
typedef struct BTnode {
        struct BTnode *left;
        int data;
        struct BTnode *right;
};
void inOrder (BTnode* tree) {
        if (tree) {  /* якщо вершина існує */
                inOrder (tree -> left);      /* "пірнаємо" в ліве піддерево вузла */
                printf ("%d\n", tree-> data);/* обробка даних на вершині дерева */
                inOrder (tree -> right);   /* "пірнаємо"в праве піддерево вузла */
        }
        return;
}
```

На відміну від обходу в прямому порядку, для дерева зі ступенем більше 2 визначити симетричний обхід непросто. Можна змусити алгоритм обробити одну половину дочірніх вершин, а потім вершину і дочірні вершини, які залишилися, але таке рішення буде нетиповим.

8.7.3 Обхід у зворотному порядку

Алгоритм обходу в зворотному порядку також відомий як алгоритм *LRN (Left – Right - Node).*

В даному випадку алгоритм обробляє спочатку лівий дочірній вузол вершини, потім правий і тільки після цього саму вершину. Як і в попередніх обходах, алгоритм почне розглядати дерево, зображене на Рисунок 8.21, з кореня, переміститься у ліву частину до дочірньої вершини B, а через неї - до лівої дочірньої вершини A. Не знайшовши подальших нащадків, він виведе A, повернеться до батьківської вершини B і перейде до правої дочірньої вершини C. У неї теж немає нащадків, тому програма виведе саму вершину C і знову звернеться до батьківської вершини B. Оскільки робота з дочірніми вершинами на цьому рівні закінчена, алгоритм виведе B, підніметься до кореня D і пройде до правої дочірньої

вершини E. Не виявивши пов'язаних з нею нащадків, алгоритм зразу виведе E і повернеться до D. Раз обхід дочірніх вершин закінчено, залишається вивести D і завершити траверс. В цілому порядок проходження алгоритму буде таким: ***A, C, B, E, D***.

У наступному списку представлені кроки, які виконуються під час зворотного обходу для дерева, зображеного на Рисунок 8.21:

1. *Пройти D.*
2. *Пройти B.*
3. *Пройти A.*
4. *Вивести A.*
5. *Пройти B.*
6. *Пройти C.*
7. *Вивести C.*
8. *Пройти B.*
9. *Вивести B.*
10. *Пройти D.*
11. *Пройти E.*
12. *Вивести E.*
13. *Пройти D.*
14. *Вивести D.*

Рекурсивна реалізація описаного алгоритму представлена нижче у вигляді C - програми:

Приклад 8.8 Алгоритм обходу у зворотному порядку бінарного дерева

```
typedef struct BTnode {
        struct BTnode *left;
        int data;
        struct BTnode *right;
};

void postOrder (BTnode* tree) {
        if (tree) {  /* якщо вершина існує */
                postOrder (tree -> left);  /* "пірнаємо" в ліве піддерево вузла */
                postOrder (tree -> right);/* "пірнаємо" в праве піддерево вузла */
                printf ("%d\n", tree-> data);/* обробка даних на вершині дерева */
        }
        return;
}
```

Обхід в зворотному порядку також легко застосувати до дерев ступеня більше 2: алгоритм повинен пройти по всім дочірнім вузлам вершини справа наліво перед тим, як обробити її саму.

8.7.4 Обхід в ширину

Здійснюючи обхід в ширину, алгоритм обробляє всі вершини дерева на поточному рівні в порядку зліва направо, а потім переходить до вершин наступного рівня. У випадку з деревом, зображеним на Рисунок 8.21, програма насамперед звернеться до кореня і виведе D, потім перейде до наступного рівня і виведе вершини B і E, а на завершальному етапі опуститься на нижній рівень і виведе A і C. Повний обхід буде виглядати так: ***D, B, E, A, C***.

Цей алгоритм також називають *BFT (breadth-first traversal)* алгоритмом обходу. Даний алгоритм не дотримується структури дерева, на відміну від алгоритмів, розглянутих вище. У використовуваному прикладі немає дочірньої гілки від вершини E до вершини A, тому неясно, як алгоритм до неї перейде. Щоб вирішити це завдання, потрібно додати дочірні вершини в чергу (яку створюємо додатково!), а потім обробити їх, коли завершиться обробка батьківського рівня.

Приклад такого алгоритму на мові C наведено нижче:

Приклад 8.9 Алгоритм обходу бінарного дерева в ширину

```
typedef struct tQueueNode { /* структура вузла черги вершин*/
        BTnode data;        /* дані елемента черги - вершини дерева */
        tQueueNode *next; /* наступна дочірня вершина */
};

typedef struct tQD {        /* дескриптор черги */
        tQueueNode *front;
        tQueueNode *rear;
        int size;
};

typedef struct BTnode {    /* вершина дерева */
        struct BTnode *left;
        int data;
        struct BTnode *right;
};

typedef struct tBTD {       /* дескриптор дерева */
        int size;       /* число вершин */
        BTnode *root;       /* корінь дерева */
};
```

```
tQD* CreateQueue(void);  /* прототип створення простої черги,
                            (самі алгоритми див. розділ 6.4 */
int InQueue (tQD*, BTnode*); /* прототип постановки вершини дерева
                                в чергу */
int ExQueue (tQD*, BTnode*); /* прототип вилучення вершини дерева
                                з черги */
int ReleaseQueue (tQD*)  /* прототип видалення черги */

int depthFirst(tBTD *dsTree) {
        tQD dsChildren;
        BTnode *node;

        // Створюємо чергу для зберігання дочірніх вершин при подальшій
        // обробці.
        dsChildren = CreateQueue();
        if (!dsChildren)      /* немає пам'яті в купі */
                return -1;

        // Ставимо корінь в чергу
        InQueue (dsChildren, dsTree -> root);

        // Обробляємо чергу, поки вона не стане порожньою.
        while (dsChildren -> size) {
                // Отримуємо наступну вершину з черги.
                if (!ExQueue (dsChildren, node))        /* внутрішня помилка */
                        return -2;

                < обробляємо вершину.>          /* тут знаходяться оператори
                                                   обробки даних на вершині node*/
                // Додаємо дочірні вершини вузла node у чергу.
                If (node -> left)
                        if (!InQueue (dsChildren, node -> left))
                                return -1;
                If (node -> right)
                        if (!InQueue (dsChildren, node -> right)
                                return -1;
        }

        if (!ReleaseQueue (dsChildren))
                return -2;
        else
                return 0;
}
```

Спершу алгоритм створює чергу і розміщує в ній кореневу вершину. Потім починається цикл, який працює до тих пір, поки черга не спорожніє. В середині цикла програма видаляє з черги першу вершину, обробляє її, а після додає в чергу її дочірні вершини. У такому випадку всі елементи розглядаються в порядку

«Першим прийшов, першим пішов», тобто вершини поточного рівня будуть оброблені повністю, перед тим як почнеться обробка будь-яких дочірніх вершин. Оскільки алгоритм ставить в чергу відразу лівий дочірній вузол вершини, а потім правий, на певному рівні всі вершини обробляються в порядку зліва направо. (Цей факт можна довести за допомогою індукції.)

8.7.5 **Час виконання обходу**

Алгоритми, які виконують обхід дерева симетрично, в прямому і зворотному порядках, слідують вниз до термінальних вершин, повертаючись вгору до кореня в міру розгортання рекурсивних викликів. Після того як алгоритм пройшов по вершині і повернувся до її батьків, він більше до цієї вершини не звертається, тобто відвідує її всього один раз. Таким чином, для дерева з *N* вершин час роботи цих алгоритмів складе *O(N)*.

Згадані перші три обходи не вимагають додаткового простору в пам'яті / на диску, оскільки вони використовують структуру дерева для відстеження свого розташування в ньому. Проте їх глибина рекурсії дорівнює висоті дерева, і якщо дерево виявиться дуже високим, ймовірно переповнення стека.

При обході в ширину вершини обробляються по мірі їх постановки в чергу. Оскільки кожна вершина поміщається туди лише один раз, для дерева з *N* вершин час роботи алгоритму буде рівним *O(N)*.

Цей алгоритм не використовує рекурсію, а значить, не залежить від її глибини, але потребує додаткового місця в пам'яті для побудови черги. У найгіршому випадку, якщо дерево є ідеальним бінарним, тобто на його нижньому рівні знаходиться практично половина всіх вершин, в черзі виявиться *O(N/2) = O(N)* вершин.

У загальному вигляді можна уявити, що дерево довільного рівня складається з одного кореневого вузла, а всі інші є дочірніми. Тоді черга створюється з *N - 1* вершин, але вимога до необхідного місця в пам'яті залишається колишньою – *O(N)*.

8.8 Впорядковані бінарні дерева

Вершини упорядкованих дерев (бінарних дерев пошуку) (рисунок нижче) структуровані таким чином, що симетричний обхід обробляє їх у відсортованому порядку. Іншими словами, значення кожної батьківської вершини більше значення

її лівого дочірнього вузла і менше правого (або дорівнює йому).

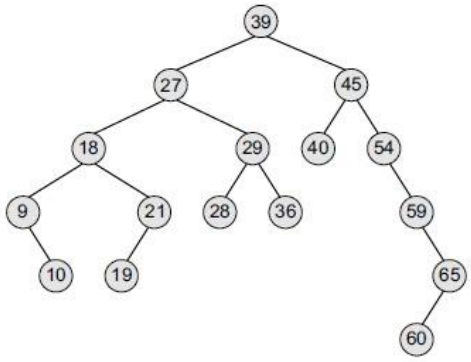


Рисунок 8.22 Бінарне дерево пошуку

У бінарному дереві пошуку (*BST - Binary Search Tree*), всі вершини в лівому піддереві мають значення, менші, ніж у кореневої вершини. Відповідно, всі вершини в правому піддереві мають значення або рівне, або більше, ніж у кореневої вершини. Те ж правило застосовується до кожного піддерева в дереві. (Зверніть увагу, що бінарне дерево пошуку може містити, або не містити повторювані значення в вершинах, в залежності від реалізації дерева).

Як видно з Рисунок 8.22 кореневою вершиною є 39. Ліве піддерево кореневої вершини складається з вершин **9, 10, 18, 19, 21, 27, 28, 29, і 36**. Всі ці вершини мають менші значення, ніж коренева. Праве піддерево кореневої вершини складається з вузлів **40, 45, 54, 59, 60, і 65**. Побудова будь-якого піддерева в такому дереві також підкоряється такому ж рекурсивному правилу. Наприклад, в лівому піддереві кореневої вершини **39**, вершина **27** є коренем, а всі елементи в її лівому піддереві (**9, 10, 18, 19, 21**) меньші, ніж **27**, в той час як всі вершини в її правому піддереві (**28, 29** і **36**) більші, ніж значення кореня **27**.

Так як вузли в бінарному дереві пошуку впорядковані, необхідний час для пошуку елемента в дереві значно знижується. Всякий раз, коли ми шукаємо елемент, ми не виконуємо прохід по ВСЬОМУ дереву. У кожній вершині, ми отримуємо підказку про те, яке піддерево слід використовувати для подальшого пошуку. Наприклад, в даному дереві, якщо ми повинні шукати вершину зі значенням ключа 29, то ми знаємо, що ми повинні переглядати тільки ліве піддерево. Якщо таке значення присутнє в дереві, це можливо тільки для лівого піддерева, так як 29 менше, ніж 39 (значення кореневого вузла). Ліве піддерево має кореневу вершину зі значенням 27. Оскільки 29 більше, ніж 27, тепер ми станемо рухатися в правому піддереві, де і знайдемо елемент. Таким чином, середній час

роботи пошукової операції є $O(log_2N)$, так як на кожному кроці, ми виключаємо половину поточного піддерева із процесу пошуку. Через таку високу ефективність пошуку елементів, бінарні дерева пошуку широко використовуються в інформаційних системах, де необхідні дані індексуються деяким значенням ключа.

Бінарні дерева пошуку також дозволяють прискорити операції вставки і видалення вершин. Такі дерева мають перевагу в швидкості обробки швидко змінюваних даних і при роботі в умовах реального часу.

Бінарні дерева пошуку вважаються більш ефективними структурами зберігання даних в порівнянні з відсортованими лінійними масивами і зв'язними списками. У відсортованому масиві, пошук може також займати $O(log_2N)$ часу, але вставки і видалення являються набагато тривалішими операціями. В зв'язних списках операції вставки і видалення елементів виконуються простіше, але операції пошуку займають більший час, пропорційний $O(N)$.

Проте, в найгіршому випадку, і бінарне дерево пошуку може також дати $O(N)$ часу для пошуку елемента. Найгірший випадок може статися, коли дерево являє собою лінійний ланцюжок вершин, як показано на малюнку нижче:

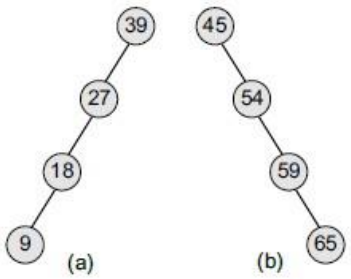


Рисунок 8.23 Ліво- (a) і право- (b) перекошені дерева

Такі бінарні дерева називають *виродженими*. Для запобігання виникнення подібних ситуацій використовуються спеціальні алгоритми додавання (видалення) вершин при виконанні операцій над деревом.

8.9 Операції над бінарними деревами пошуку

Бінарні дерева пошуку (далі просто дерева) використовуються для подання даних і для маніпуляції ними. Конкретне подання даних і виконання над ними операцій залежать від розв'язуваної прикладної задачі. Ми будемо розглядати загальні операції над деревами, які можуть виконуватися при вирішенні будь-яких завдань. До цих операцій можна віднести наступні:

- створення дерева;

- пошук (локалізація) елемента в дереві;
- включення елемента в дерево;
- видалення елемента з дерева;
- видалення бінарного дерева.

Ці операції розглядаються нижче стосовно бінарних дерев пошуку, розміщених в динамічній пам'яті.

8.9.1 Створення дерева

Для управління деревом як динамічною структурою необхідна наявність дескриптора. Якщо дескриптор побудований в динамічній пам'яті, то доступ до дерева здійснюється через ланцюжок покажчиків, як показано на рисунку нижче:

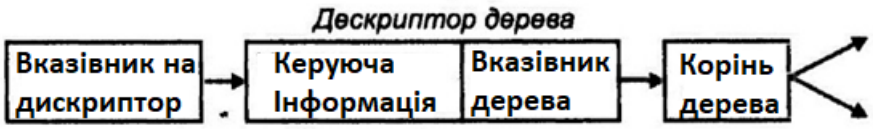


Рисунок 8.24 Доступ до дерева за допомогою дескриптора

У найпростішому випадку (як виняток) дескриптор не використовується, доступ до дерева здійснюється за покажчиком, оголошеному в програмі як змінна.

Алгоритм створення бінарного дерева зводиться до створення і ініціалізації дескриптора. На мові С це виглядає так:

Приклад 8.10 Алгоритм створення бінарного дерева

```
typedef struct BTnode {    /* вершина дерева */
        struct BTnode *left;
        int data;
        struct BTnode *right;
};

typedef struct tBST {                /* дескриптор дерева */
        int size;       /* число вершин */
        BTnode *root;       /* корінь дерева */
};

tBST* createTree (void) {

        tBST *dsBST;
        dsBST = (tBST*) malloc(sizeof(tBST));
        if (!dsBST) return NULL; /* немає доступної пам'яті в купі,
                                    дескриптор не створений */
        dsBST -> root = NULL;
        dsBST -> size = 0;
        return dsBST;
}
```

8.9.2 **Пошук вершини в бінарному дереві**

Алгоритм пошуку вершини використовується для пошуку в дереві вершини з заданим значенням або констатації факту її відсутності.

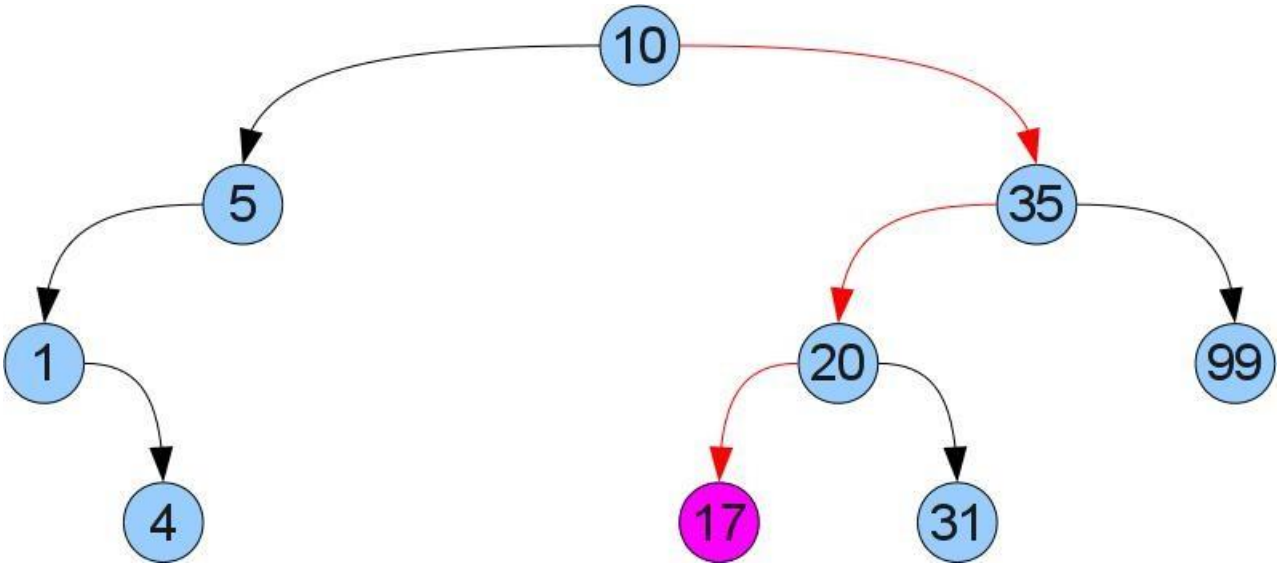


Рисунок 8.25 Алгоритм пошуку значення 17 в бінарному дереві пошуку

Процес пошуку починається з кореневої вершини. Алгоритм перевіряє, чи є бінарне дерево пошуку порожнім. Якщо воно порожнє, то значення, яке ми шукаємо відсутнє в дереві. Таким чином, алгоритм пошуку завершується шляхом повернення значення *NULL* в якості знайденої вершини до програми, що викликала. Якщо ж є вершини в дереві, то алгоритм перевіряє, чи є ключове значення поточної вершини рівним значенню пошуку. Якщо ТАК, алгоритм повертає в викликану програму адресу знайденої вершини і завершується. Якщо НІ, то алгоритм перевіряє, чи є значення для пошуку менше, ніж значення поточної вершини, і в цьому випадку він повинен рекурсивно викликати сам себе з лівої дочірньої вершини в якості параметра. У разі, якщо значення пошуку більше, ніж значення ключа поточної вершини, алгоритм повинен рекурсивно викликати себе з правої дочірньої вершини.

Алгоритм пошуку на мові С може виглядати так:

Приклад 8.11 Алгоритм пошуку в бінарному дереві пошуку

```
typedef struct BTnode {    /* вершина дерева */
        struct BTnode *left;
        int data;
        struct BTnode *right;
};
```

```
BTnode *searchElement (BTnode *root, int key) {
        if  (root -> data == key || !root)
                return (root);
        else if (key < root -> data)
                return (searchElement(root -> left, key));
        else
                return (searchElement(root -> right, key));
}
```

8.9.3 Включення елемента в бінарне дерево пошуку

Функція включення (вставки) використовується для додавання нової вершини із заданим значенням ключа в правильну позицію бінарного дерева пошуку.

Алгоритм включення в дерево окремих елементів складається з трьох дій:

- пошук місця включення;
- отримання динамічної пам'яті для вершини і створення зв'язку вершини з деревом за допомогою покажчика;
- занесення даних елемента до новоствореної вершини.

Додавання вершини в правильну позицію означає, що нова вершина не повинна порушувати властивості бінарного дерева пошуку. На рисунку нижче показаний приклад вставки ключа 42 в бінарне дерево пошуку:

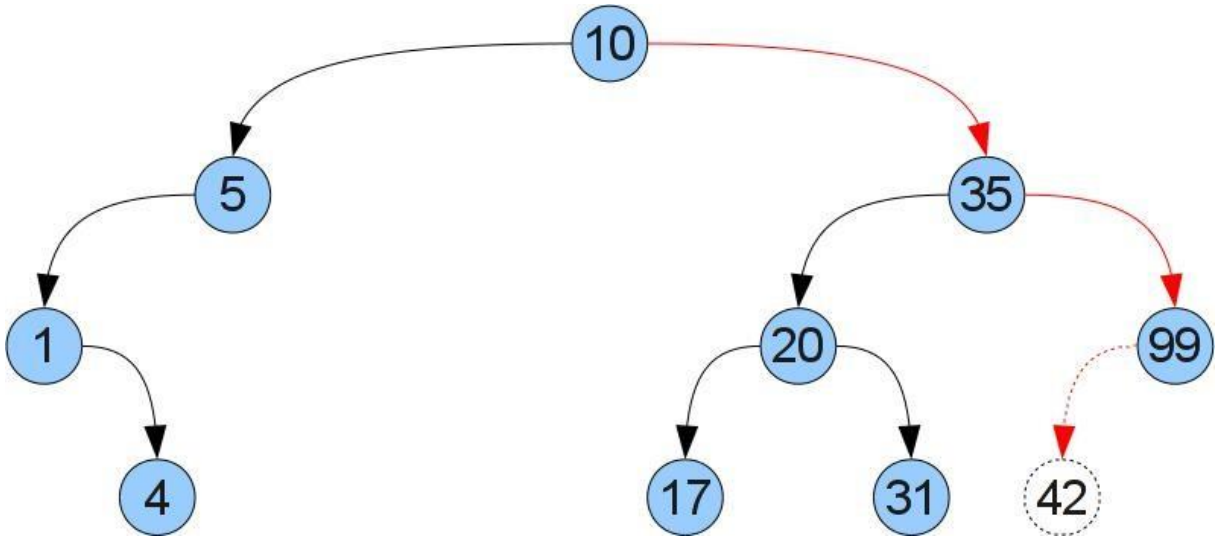


Рисунок 8.26 Приклад вставки ключа 42 в бінарне дерево пошуку

В прикладі нижче показано алгоритм включення заданого значення ключа в бінарне дерево пошуку:

Приклад 8.12 Алгоритм включення заданого значення в бінарне дерево пошуку

```
typedef struct BTnode {    /* вершина дерева */
        struct BTnode *left;
        int data;
        struct BTnode *right;
};

typedef struct tBST {        /* дескриптор дерева */
        int size;        /* число вершин */
        BTnode *root;        /* корінь дерева */
};


BTnode* insertTree (tBST *dsTree, BTnode *subTree, int key) {
        BTnode *tempNode;        /* тимчасова вершина */

        /* місце вставки знайдено (поточна вершина вже термінальна),
        виконується створення вершини і її заповнення */
        if (!subTree) {
                subTree = (BTnode*) malloc(sizeof(BTnode));
                if (!subTree)  // не вистачає місця в "купі" під нову вершину
                        return NULL;
                subTree -> data = key;
                subTree -> left = subTree -> right = NULL;
                dsTree -> size++;
                return (subTree);
                }


        // місце вставки ще не знайдено,

        // 1. Продовжуємо пошук місця вставки по лівій гілці
        else if (key < subTree -> data)
                tempNode = insertTree(dsTree, subTree -> left, key);
        // 2. Продовжуємо пошук місця вставки по правій гілці
        else
                tempNode = insertTree(dsTree, subTree -> right, key);


        return tempNode;

}
```

Вихідний код для функції вставки схожий з кодом функції пошуку. Це тому, що ми повинні спочатку знайти правильну позицію в дереві для вставки вершини, і лише потім додати вершину в знайдену позицію. Функція вставки змінює структуру дерева. Тому, коли функція вставки викликається рекурсивно, то

функція повинна повертати покажчик на нову вершину дерева.

На першому кроці алгоритму, функція вставки перевіряє, чи є поточна вершина дерева термінальною, тобто чи є переданий в функцію покажчик піддерева *subTree* нульовим. Якщо ТАК, то алгоритм просто додає нову вершину з ініціалізацією її полів і зміною розміру дерева в дескрипторі, інакше він порівнює значення переданого ключа *key* зі значенням ключа поточної вершини, а потім опускається на вершину вниз вліво або вправо, в залежності від результату порівняння.

Функція вставки продовжує рухатися вниз, проходячи по ходу свого руху всі рівні бінарного дерева, поки не досягне листової вершини. У цьому місці додається нова вершина, дотримуючись правил побудови бінарних дерев пошуку. Тобто, якщо значення ключа нової вершини більше або дорівнює значенню ключа поточної вершини - нова вершина вставляється в правому піддереві, інакше - в лівому піддереві.

Функція вставки вимагає часу виконання, в найгіршому випадку пропорційному двійковому логарифму висоти дерева $O(log_2 H)$.

8.9.4 Видалення вершини з бінарного дерева пошуку

Функція видалення вершини є найскладнішою з операцій над BST. Вся справа в тому, що властивості двійкового дерева пошуку не повинні порушитися при видаленні однієї з вершин. Існує три різних випадки видалення вершин з BST (в залежності від наявності у вершини дочірніх нащадків), кожен з яких ми розглянемо окремо.

Випадок 1: видалення вершини, яка не має нащадків.

Подивимося на бінарне дерево пошуку, наведене на рисунку нижче. Якщо необхідно видалити вершину 78, ми можемо просто видалити цю вершину без будь-якого питання. Це найпростіший випадок видалення.

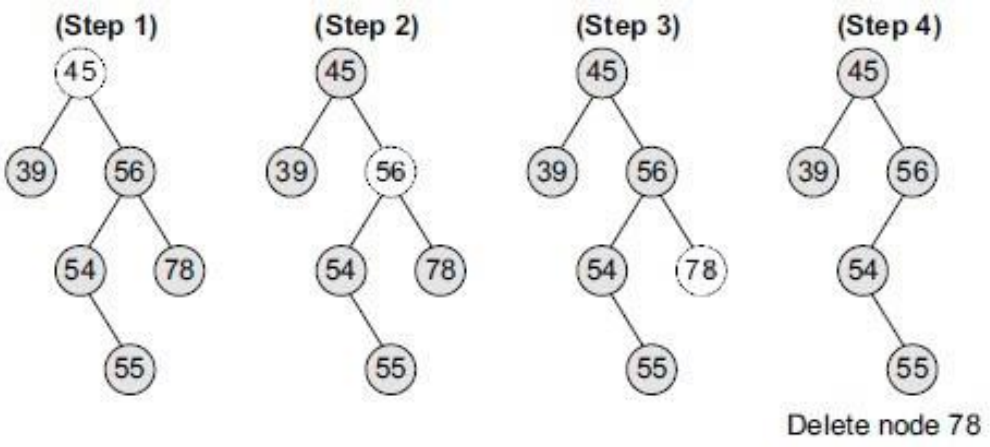


Рисунок 8.27 Приклад видалення вершини 78, яка не має нащадків

Випадок 2: видалення вершини з одним нащадком.

Щоб впоратися з таким завданням єдиний нащадок вершини, що видаляється, встановлюється в якості нащадка її батьківської вершини. Іншими словами, видалена вершина в дереві замінюється її нащадком. Наприклад, видалимо вершину 5 з дерева нижче:

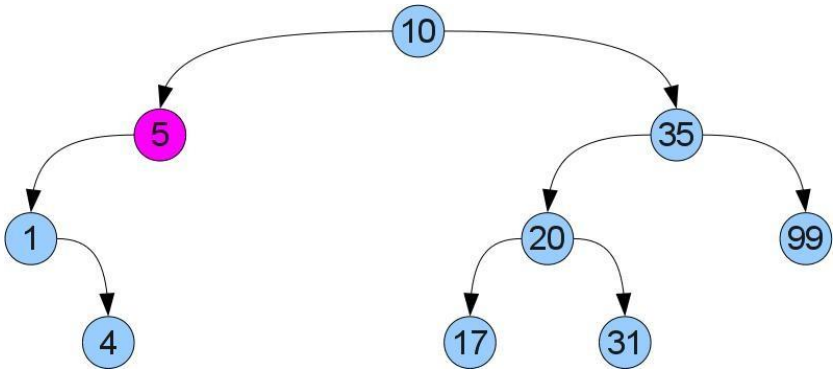


Рисунок 8.28 Видалити вершину 5 з бінарного дерева пошуку

Результат операції такого видалення наведено на наступному рисунку:

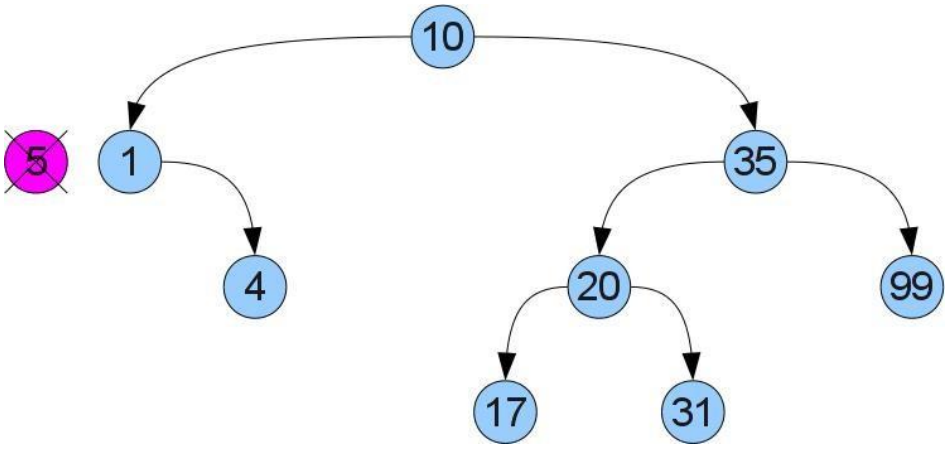


Рисунок 8.29 Бінарне дерево пошуку після видалення з нього вершини 5

Випадок 3: видалення вершини з двома нащадками.

Видаляємо знову вершину 5, але трохи з іншого дерева:

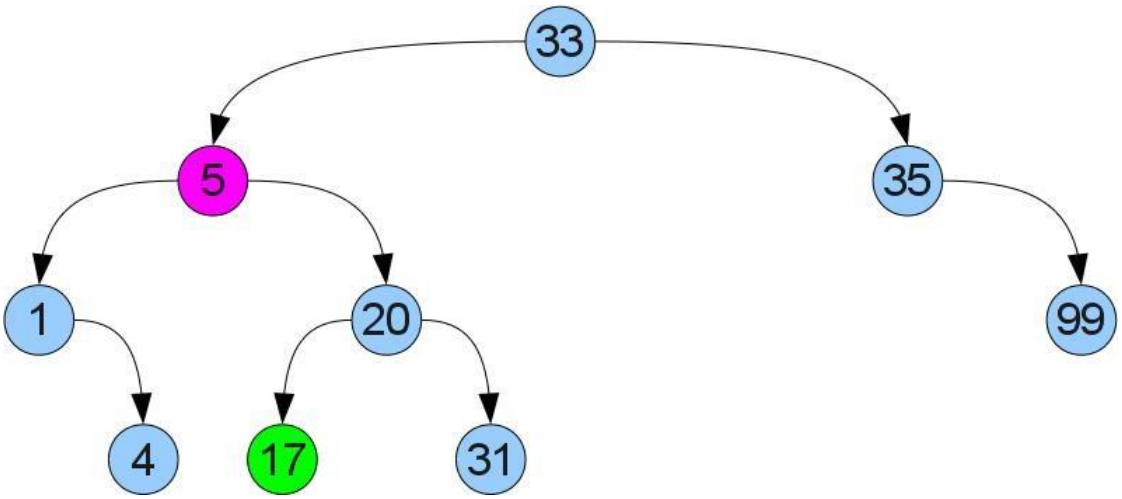


Рисунок 8.30 Видалення вершини 5 з двома нащадками

Зробимо тепер по-іншому: знайдемо в правому піддереві мінімум. Ясно, що його можна знайти, якщо почати в правому нащадку вузла 5 і йти до упору вліво, по лівим гілкам. Оскільки у знайденому мінімумі немає лівого нащадка, можна вирізати його за аналогією з випадком 1 і вставити його замість видаляємої вершини. Через те що він був мінімальним в правому піддереві, властивість впорядкованості не порушується:

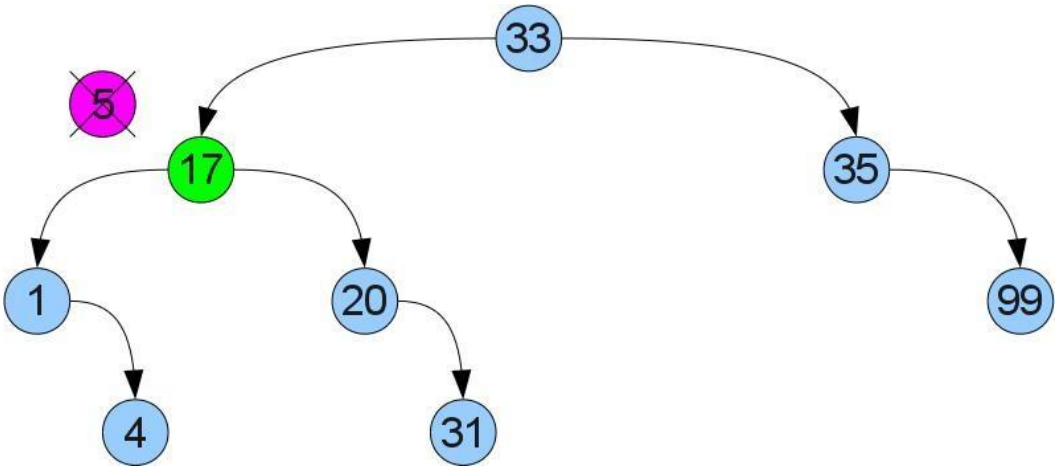


Рисунок 8.31 Бінарне дерево пошуку після видалення з нього вершини 5

Алгоритм, який реалізує в собі всі три вищевказаних випадки, наведено у прикладі нижче:

Приклад 8.13 Алгоритм видалення вершини дерева

```
typedef struct BTnode {    /* вершина дерева */
        struct BTnode *left;
        int data;
        struct BTnode *right;
};

typedef struct tBST {              /* дескриптор дерева */
        int size;                  /* кількість вершин */
        BTnode *root;              /* корінь дерева */
```

```
};


// рекурсивна функція знаходження мінімального елемента у
// дереві (піддереві)
BTnode *findSmallestElement(BTnode *tree)  {
        if (!tree || !tree -> left)
                return tree;
        else
                return findSmallestElement(tree -> left);
}



// функція видалення елемента
BTnode *deleteElement(tBST* dsTree, BTnode *subTree, int key) {

        BTnode      *current,     /* адреса поточної вершини */
                    *parent,      /* адреса батьківської вершини */
                    *temp;              /* адреса тимчасової вершини */
        // випадок виродженого дерева
        if (!subTree)
                return (subTree);

        // пошук вершини, що видаляється, в лівому (правому) піддереві
        parent = current = subTree;
        while (current && key != current–>data)  {
                parent = current;
                current = (key < current –> data) ? current –> left : current–>right;
        }

        if (!current)  return NULL;  // вершина, що видаляється, в дереві
відсутня


        /***************** Вершина, що видаляється, знайдена! ***********/

        //  Випадок 3, присутність двох нащадків у вершини, що видаляється
        if  (current -> left && current -> right)  {

                /* пошук мінімального елемента у правому піддереві з переносом
                   його даних на місце даних вершини, що видаляється */
                current -> data = findSmallestNode(current -> right) -> data;

                //  видалення мінімального елемента у правому піддереві
                return deleteElement (current -> right, current -> data);
        }

        // Випадок 1, у вершини, що видаляється, відсутні нащадки
        else if (!current -> left && !current -> right)  {

                // якщо вершина, що видаляється стоїть зліва від батьківської
```

```
            if (parent -> left == current)
                parent - > left = NULL;

            // якщо справа від батьківської
            else
                parent -> right = NULL;

            temp = current;
            dsTree -> size--;
            free (current);
            return temp;
        }

        // Випадок 2, один нащадок у вершини, що видаляється

        else if  (current -> left)    // якщо нащадок лівий
            parent -> left = current -> left;
        else                          // якщо нащадок правий
            parent -> right = current -> right;

        temp = current;
        dsTree -> size--;
        free (current);
        return temp;

}
```

8.9.5 Видалення бінарного дерева

Щоб видалити дерево бінарного пошуку з пам'яті, ми спершу повинні видалити всі вершини в лівому піддереві, а потім видалити всі вершини в правому піддереві. Після цього можна видалити і дескриптор дерева. Алгоритм в прикладі нижче дає рекурсивну процедуру повного видалення бінарного дерева пошуку.

Приклад 8.14 Алгоритм повного видалення бінарного дерева пошуку

```
typedef struct BTnode {    /* вершина дерева */
        struct BTnode *left;
        int data;
        struct BTnode *right;
};


int deleteTree (BTnode *tree)  {
        if (tree)  {
                deleteTree (tree -> left);
                deleteTree (tree -> right);
                free (tree);
                return 0;
                }
        else
```

```
        return 1;
}
```

8.10 Контрольні завдання і запитання

1. Розробити схеми алгоритмів для бінарного дерева пошуку:

- для усіх видів обходу з розділу 8.7;
- для створення дерева і пошуку потрібної вершини в ньому (розділи 8.9.1 та 8.9.2);
- для включення вершини в дерево і видалення дерева в цілому(розділи 8.9.3 та 8.9.5);
- для видалення вершини з дерева (розділ 8.9.4).

2. Розробити схему алгоритма і програму для алгоритма побудови бінарного дерева із загального дерева (розділ 8.4).

3. Чи може ідеальне бінарне дерево містити парну кількість вершин?

4. Ідеальне дерево вважається повним та завершеним, але не всі повні та завершені дерева ідеальні. Накресліть дерево, яке є повним та завершеним, але не ідеальним.

5. Доведіть, що кількість гілок B у бінарному дереві з N вершин обчислюється за формулою $B = N - 1$. Використовуйте індукцію.

6. Доведіть факт із вправи 5 без використання індукції.

7. Як виглядатиме обхід у прямому порядку для дерева, зображеного на наступному рисунку:

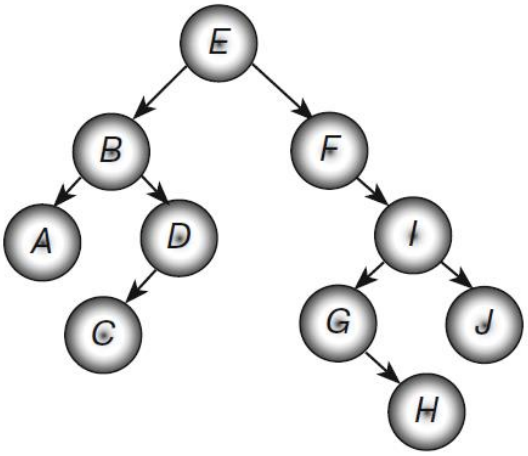


Рисунок 8.32

8. Здійсніть симетричний обхід дерева, зображеного на рисунку вище.

9. Проаналізуйте обхід у зворотному порядку для дерева, представленого на

рисунку вище.

10. Здійсніть обхід в ширину для дерева, зображеного на рисунку вище.

11. Напишіть програму, яка здійснює всі види обходу з вправ 7–10 для дерева, представленого на Рисунок 8.32.

12. Що станеться, якщо використовувати стек замість черги в алгоритмі обходу двійкового дерева в ширину? Спробуйте згенерувати такий обхід рекусивно.

13. Накресліть дерева розбору для наступних математичних виразів:

1) (15 / 3) + (24 / 6);

*2) 8 * 12 – 14 * 32;*

3) 1/2+1/4+1/20;

*4) (36 * 2) / (9 * 32);*

*5) 5! / ((5 - 3)! * 3!);*

6) sin^2 (45°).

9 ПОШУК

9.1 Загальні відомості

У темі СОРТУВАННЯ ми дізналися про те, як упорядковувати дані, і з'ясували, що алгоритми швидкого сортування та сортування Шелла дозволяють легко впоратися з великим обсягом даних, а блокове сортування і сортування підрахунком показують непогану продуктивність тільки при певних умовах. Зараз мова піде про алгоритми, які допомагають знаходити конкретні дані в раніше відсортованому масиві.

Пошук *— звуження деякої кількості даних з метою виявлення підмножини даних, що відповідає критеріям пошуку* [1].

Алгоритми пошуку призначені для пошуку конкретного значення в наборі даних. Якщо значення присутні в наборі, то пошук вважається успішним, і процес пошуку дає місце розташування цього значення в наборі даних. Інакше, якщо значення відсутнє у наборі, процес пошуку відображає відповідне повідомлення і у цьому випадку пошук називається невдалим. Для пошуку ми використовуємо ключ елемента даних.

Існують два види ключів - *внутрішні* і *зовнішні*. Наприклад, ми хочемо знайти дані в записах, що пов'язані з заданим ключем (по прізвищу знайти номер телефону). Такий ключ називається *внутрішнім ключем* або *вбудованим ключем*. А такий вид пошуку буде називатися *внутрішнім пошуком*. Однак, ми можемо організувати окрему таблицю ключів, яка включає в себе крім самих ключів покажчики на записи, які зберігають інформацію, що відноситься до окремих ключів. Самі ж записи з даними будемо зберігати в іншому файлі. Такі ключі будуть називатися *зовнішніми ключами*, а цей вид пошуку, де інформаційна частина таблиці зберігається в додатковому файлі - буде називатися *зовнішнім пошуком*. Прикладом такого пошуку є пошук ключів в індексних файлах баз даних.

Всі алгоритми пошуку діляться на:

- пошук у невпорядкованих наборах даних (лінійний пошук);
- пошук в упорядкованих наборах даних (бінарний та інтерполяційний пошуки).

9.2 Лінійний пошук

Лінійний пошук, також званий *послідовним* пошуком, є дуже простим методом, використовуваним для пошуку у довільному наборі даних конкретних значень. Він працює шляхом порівняння ключа пошуку з ключем кожного елемента набору, одного за іншим, послідовно, поки не буде знайдено потрібний елемент. Лінійний пошук в основному використовується для пошуку у не впорядкованих наборах даних.

Лінійний пошук циклічно проходить по набору даних, намагаючись відшукати цільовий елемент. На рисунку нижче таким елементом є число 77.

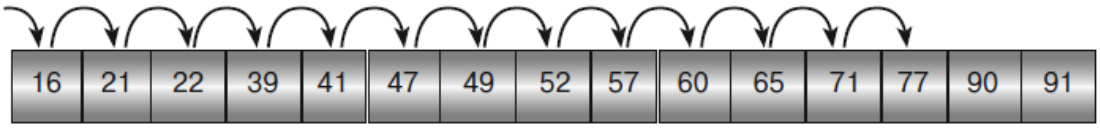


Рисунок 9.1 Ілюстрація лінійного пошуку

На відміну від бінарного і інтерполяційного пошуку, лінійний може працювати не тільки з масивами, але і зі зв'язними списками, де можна запросто перестрибнути з однієї частини в іншу. Списки можуть бути і несортованими, але з відсортованими алгоритм економить значну частину часу: він зупиняється, дійшовши до елемента зі значенням, що перевищує цільове, і не веде подальший пошук того, чого явно не існує.

Алгоритм лінійного пошуку в зв'язних списках виглядає так, як у Приклад 5.4, а алгоритм лінійного пошуку в масиві, як у Приклад 3.1.

Наведені алгоритми, можливо, пройдуть по всьому набору даних, перш ніж виявлять, що цільового елемента немає. У гіршому випадку це займе *O(N)* часу. Лінійний пошук набагато повільніший, ніж інші види пошуку. Головна його перевага полягає в тому, що він працює зі зв'язними списками і несортованими масивами.

9.3 Бінарний пошук

Бінарний пошук являє собою алгоритм пошуку, який працює тільки з попередньо відсортованими наборами. Механізм бінарного пошуку може бути краще зрозумілий за допомогою аналогії з телефонним довідником. Коли ми шукаємо конкретне прізвище в довіднику, спочатку відкриваємо довідник з середини, а потім вирішуємо, чи варто шукати прізвище в першій частині

довідника або в другій частині. Знову ж таки, ми відкриваємо деякі сторінки в середині, і весь процес повторюється до тих пір, поки ми, нарешті, не знайдемо необхідне прізвище.

Візьмемо іншу аналогію. Як ми знаходимо слова в словнику? Спочатку відкриваємо словник десь посередині. Потім ми порівнюємо перше слово на цій сторінці з потрібним нам, значення якого ми шукаємо. Якщо потрібне слово знаходиться до першого слова на сторінці, ми далі дивимося в першій половині словника, інакше ми дивимося в другій половині. Знову ж таки, ми відкриваємо середню сторінку в першій половині словника і порівнюємо перше слово на цій сторінці з потрібним словом і повторюємо ту ж процедуру, поки, нарешті, не отримаємо потрібне слово. Той же механізм застосовується і в алгоритмі бінарного пошуку.

У бінарному пошуку алгоритм відстежує найменший і найбільший індекси елементів масиву - *min* і *max*. Спочатку робимо їх рівними індексу першого і останнього елемента масива відповідно. Потім алгоритм розраховує середній індекс, що знаходиться між ними, тобто *mid*. Якщо значення ключа пошуку менше *array[mid]*, алгоритм встановлює значення *max* в *mid*, щоб почати новий пошук в лівій половині масиву; якщо ключ пошуку більше - встановлює *min* в *mid* і новий пошук ведеться в правій половині. Якщо ж цільове значення дорівнює *array[mid]*, алгоритм повертає його індекс.

На рисунку нижче зображений бінарний пошук числа 77:

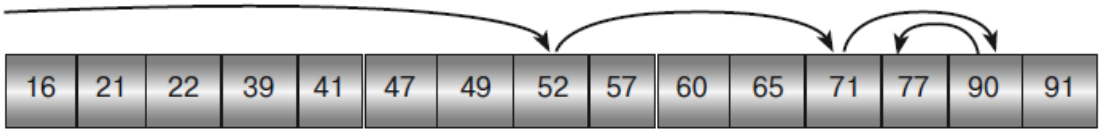


Рисунок 9.2 Ілюстрація бінарного пошуку

Сам алгоритм представлено нижче:

Приклад 9.1 Алгоритм бінарного пошуку

```
int binarySearch(int *arr, int n, int key) {

    int   beg,   /* початок пошуку */
          end,   /* кінець пошуку */
          mid,   /* середина масиву пошуку (місце розбиття
                    масиву пошуку) */
          pos;   /* результат пошуку */

    if (n < 1)   /* якщо масиву немає - шукати нічого */
```

```
        return -1

    // ініціалізація пошуку
    beg = 0;
    end = n-1;
    pos = -2;     /* елемент не знайдений */

    // пошук елемента
    while (beg <= end) {      // виконувати, поки існує підмасив пошуку

        mid = (beg + end)/2;
        if (arr[mid] == key) {
            pos = mid;
            break;
        }

        else if (arr[mid] > key)
            end = mid - 1;
        else
            beg = mid + 1;
    }

    // повернення результату пошуку
    return pos;

}
```

На кожному кроці даний алгоритм ділить масив елементів, серед яких може міститися цільовий, навпіл. Якщо кількість елементів масиву N, то після проходження максимум $O(log_2 N)$ кроків в частині масиву, де може розташовуватися цільовий елемент, залишиться тільки одне значення. Таким чином алгоритм знайде шукане або виявить, що дії не принесли належного результату. Це означає, що алгоритм має час роботи $O(log_2 N)$.

9.4 Інтерполяційний пошук

Інтерполяційний пошук, також відомий як *екстраполяційний*, є методом пошуку, який знаходить задане значення у попередньо відсортованому масиві. Концепція інтерполяційного пошуку аналогічна тому, як ми шукаємо прізвища в телефонній книзі, яка впорядкована. Наприклад, при пошуку прізвища "Амбросимов" в телефонній книзі, ми знаємо, що таке прізвище буде перебувати десь на початку, тому застосування техніки бінарного пошуку шляхом ділення книги на дві половинки кожен раз, вже не є гарною ідеєю. Ми повинні почати сканування перших значень вже на найпершому проході.

На кожному кроці інтерполяційного пошуку обчислюється залишковий

простір (масив) пошуку ключа. Розрахунок проводиться на основі тільки значень, що залишилися в межах залишкового простору і значення ключа пошуку. Значення, знайдене в розрахованій позиції порівнюється зі значенням розшукуваного ключа. Якщо два значення рівні, то пошук закінчений.

Проте, в разі, якщо значення не рівні, то в залежності від результату порівняння, залишковий простір пошуку зводиться до частини до або після розрахованої раніше позиції. Таким чином, ми бачимо, що інтерполяційний пошук схожий на двійковий метод пошуку. Проте, важлива відмінність між цими двома методами в тому, що бінарний пошук завжди вибирає середнє значення в просторі, що залишився. Відкидається половина значень на основі порівняння між розрахованим значенням і пошуковим значенням. А при інтерполяційному пошуку, використовується інтерполяція, щоб знайти елемент поблизу розшукуваного, а потім лінійний пошук вже використовується для пошуку точного співпадання.

Бінарний пошук кожен раз шукає цільовий елемент масиву в середині поділу, а інтерполяційний намагається ще прискорити процес - він намагається вгадати розташування цільового елемента в масиві за його значенням. Припустимо, в масиві міститься 100 елементів зі значеннями від 1 до 100. Якщо наша мета - знайти число 77, то його потрібно шукати в районі останньої третини масиву, десь поруч з індексом 70. Загальний розподіл чисел не завжди дозволяє отримати результат зі 100% точністю, але результат може виявитися досить близьким до шуканого.

На рисунку нижче зображений інтерполяційний пошук ключа 77:

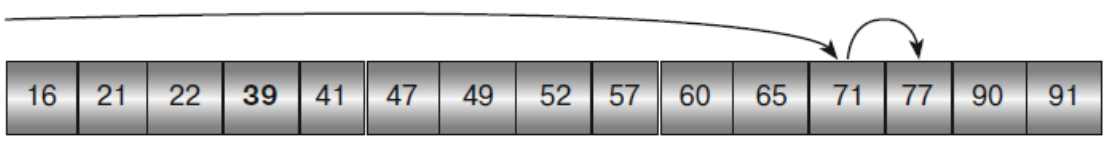


Рисунок 9.3 Ілюстрація інтерполяційного пошуку

Алгоритм такого методу пошуку може виглядати так:

Приклад 9.2 Алгоритм інтерполяційного пошуку

```
int interpolSearch(int arr[], int n, int key)  {
        int mid,      /* місце розбиття масиву пошуку */
            low,      /* нижня границя пошуку */
            high;     /* верхня границя пошуку */
```

```
    if (n < 1)      /* якщо масиву немає, то і шукати нічого */
        return -1;

    // ініціалізація пошуку
    low = 0;
    high = n-1;

    // пошук елемента
    while (low <= high)  {     // виконувати, поки існує підмасив пошуку

        /* визначаємо границю розбиття! (саме тут може бути використано алгоритм розбиття, що найліпшим чином відповідає властивостям пошукових значень) */

        mid = low + (high – low)*((key – arr[low]) / (arr[high] – arr[low]));

        if (key == arr[mid])
            return mid;  /* повертаємо значення індексу ключа */

        if (key < arr[mid])
            high = mid – 1;
        else
            low = mid + 1;

    }

    // елемент не знайдено
    return –2;
}
```

На практиці, інтерполяційний пошук часто працює швидше бінарного, так як з обчислювального боку їх відрізняють лише використані арифметичні операції: інтерполювання - в інтерполяційному пошуку та поділ на два - в бінарному, а швидкість їх обчислення відрізняється незначно. З іншого боку, при використанні інтерполяційного пошуку слід мати на увазі, що пошуковий алгоритм використовує таку важливу властивість даних, як однорідність розподілу значень. Якщо значення ключів на інтервалі пошуку розміщені нерівномірно, то швидкість роботи алгоритма може стати рівною, або навіть меншою за швидкість бінарного пошука.

Ключем може бути не тільки номер (число), а й наприклад, текстовий рядок, тоді стає зрозуміла аналогія з телефонною книгою: якщо ми шукаємо ім'я в телефонній книзі, що починається на «А», його потрібно починати шукати на початку, але ніяк не в середині інтервала пошуку. В принципі, ключем може бути все що завгодно, тому що ті ж рядки, наприклад, запросто кодуються посимвольно,

в найпростішому випадку символ можна закодувати значенням від 1 до 33 (тільки кирилиця), або наприклад, від 1 до 26 (тільки латинський алфавіт), тощо.

Інтерполяція може проводитися на основі функції, апроксимуючої розподіл значень, або набору кривих, що виконують апроксимацію на окремих ділянках. У цьому випадку пошук може завершитися за кілька перевірок. Переваги цього методу полягають у зменшенні запитів на читання з повільної зовнішньої пам'яті (такої, наприклад, як жорсткий диск), якщо запити відбуваються часто.

Часто аналіз і побудова аппроксимуючих кривих не потрібні, показовий випадок тут - коли всі елементи відсортовані за зростанням. В такому списку, мінімальне значення буде за індексом *0*, а максимальне за індексом *N-1*. В цьому випадку апроксимуючу криву можна прийняти за пряму і застосовувати лінійну інтерполяцію.

Якщо дані в наборі розподілені дуже нерівномірно і ми шукаємо найгірше цільове значення, даний алгоритм буде мати продуктивність *O(N)*. Якщо розподіл приблизно однаковий - очікувана продуктивність складе $O(\log_2(\log_2 N))$.

9.5 Використання функції *bsearch()* мови Сі

У бібліотеці стандартних функцій мови Сі є функція *bsearch()* (файл заголовку *stdlib.h*), призначена для пошуку елементів в масивах різної природи. Метод пошуку, використовуваний функцією - бінарний пошук. Ця функція має абстрактний інтерфейс, який дозволяє використовувати її з пошуковими масивами будь-якого типу.

Функція має наступний прототип:

```
void *bsearch (
        const void *key,
        const void *buff,
        size_t num,
        size_t size,
        int (*compar)(const void *, const void *)
);
```

Перший аргумент *key* є покажчиком на ключ пошуку.

Другий аргумент *buff* являє собою покажчик на початок масиву, в якому виконується пошук. Стандарт ANSI C допускає приведення покажчика на будь-які дані до типу *покажчик на void*. Це дозволяє другому фактичному аргументу

функції *bsearch()* посилатися на масив елементів будь-якого типу.

Третій аргумент *num* являє собою кількість елементів в масиві пошуку. Прототип перетворює це значення до типу ***size_t.***

Четвертий аргумент *size* є цілочисельним типом даних, розміром елементу пошукового масиву в байтах, який можна визначити за допомогою операції *sizeof()*. Оскільки функція *bsearch()* перетворює свій другий аргумент в покажчик типу ***void***, вона втрачає інформацію про розмір кожного елемента масиву. Для компенсації цього недоліку слід явно вказати функції *bsearch()* розмір об'єкта даних. Саме для цього служить четвертий аргумент. Наприклад, якщо виконується пошук в масиві типу ***double***, у якості четвертого аргументу слід використати вираз *sizeof(double).*

Функція *bsearch()* для визначення порядку порівняння ключа пошуку зі значенням ключа елемента масиву вимагає передачі покажчика на функцію порівняння. Функція порівняння повинна приймати два аргументи, а саме покажчики на два порівнюваних елемента. Вона повертає додатнє ціле число, якщо перший елемент повинен слідувати за другим (тобто, є більшим), нуль, якщо елементи однакові, і від'ємне цілочисельне значення, якщо другий елемент повинен слідувати за першим (тобто, перший елемент є меншим). Функція *bsearch()* використовує функцію порівняння, передаючи їй значення покажчиків, які обчислює на основі іншої інформації, що надається.

Вид функції порівняння задається останнім аргументом прототипу функції *bsearch()*:

*int (*compar)(const void *, const void *)*

У прототипі визначено, що останній аргумент являє собою покажчик на функцію, яка повертає значення типу *int* і приймає два безтипові покажчики-аргументи. Кожен з них є покажчиком на тип *const void*. Ці два покажчика посилаються на порівнювані елементи.

І нарешті, результат пошуку, адреса першого елемента масиву, в якому було знайдено необхідне значення ключа, повертається в імені функції *bsearch()*, що вимагає його явного приведення у викликаній програмі до заданого типу елементів масиву пошуку.

Масив *buff* повинен бути обов'язково відсортований у порядку зростання,

щоб за наймолодшою адресою містився найменший елемент. Якщо масив не містить шуканого ключа-значення, повертається нульовий покажчик.

У прикладі нижче ілюструється метод визначення функції порівняння і застосування функції *bsearch()*. Наступна програма зчитує символи, що вводяться з клавіатури і визначає, чи входять вони в заданий алфавіт:

Приклад 9.3 Ілюстрація застосування функції *bsearch()*

```
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>

char *alpha = "abcdefghijklmnopqrstuvwxyz";
int comp(const void *ch, const void *s);

int main(void) {

        char ch;
        char *p;

        printf("Введіть символ: ");
        ch = getche();
        ch = tolower(ch);

        p = (char *) bsearch(&ch, alpha, strlen(alpha), sizeof(char), comp);
        if (p)
                printf(" %c знаходиться в алфавіті \n", *p);
        else
                printf("не входить в алфавіт\n");

        return 0;

}

/* Користувацька функція порівняння двох символів. */
int comp(const void *ch, const void *s) {
        return *(char *)ch - *(char *)s;
}
```

9.6 Контрольні завдання і запитання

1. Створіть алгоритм та програму, яка застосовує рекурсивний лінійний пошук. Які її переваги та недоліки порівняно з нерекурсивною версією?

2. Реалізуйте алгоритм та програму лінійного пошуку у відсортованих зв'язних списках.

3. Розробіть алгоритм та програму, що реалізує рекурсивний бінарний пошук. Які її переваги та недоліки порівняно з нерекурсивною версією?

4. Реалізуйте алгоритм та програму рекурсивного інтерполяційного пошуку.

Які її переваги та недоліки порівняно з нерекурсивною версією?

5. Якщо масиві містяться дублікати, вивчені нами алгоритми бінарного та інтерполяційного пошуку не гарантують повернення першого елемента. Спробуйте їх змінити таким чином, щоб вони могли гарантовано повернути перший цільовий елемент. Розробіть алгоритм та напишіть програму. Визначте час роботи змінених версій.

10 ХЕШУВАННЯ

10.1 Поняття таблиці

Кожен у своєму житті хоча б раз стикався з таблицями. Відомим прикладом є телефонна книга, яка містить записи, пов'язані з конкретними людьми, або підприємствами. *Ключем* (або входом в *таблицю*) є прізвище людини, найменування підприємства, тощо. Частина рядка таблиці, що залишилася після ключа, містить іншу інформацію (наприклад, адресу і номер телефону).

Проте, насправді цей приклад є прикладом *відсортованого* списку, а не таблиці в чистому вигляді. Різниця полягає в тому, що в списку елементи розташовані у вигляді послідовності. Існує перший елемент, другий, тощо і для кожного елемента (за винятком останнього) існує єдиний «наступний» елемент.

У випадку таблиці може і не існувати впорядкованості, застосованої до її елементів (рядків). Тобто відсутнє поняття "поруч". Невпорядковані по ключу таблиці виникають досить часто при виконанні операцій з даними в реляційних базах даних. Дуже знайомий приклад з життя - це таблиця перетворення різних одиниць виміру одна в одну, наприклад, в/із метричних одиниць виміру в/із англійські одиниці виміру.

45 kg	99.2 lbs		46 kg	101.4 lbs
47 kg	103.6 lbs		48 kg	105.8 lbs
49 kg	108 lbs		50 kg	110.2 lbs
51 kg	112.5 lbs		52 kg	114.7 lbs
53 kg	116.8 lbs		54 kg	119 lbs
55 kg	121.3 lbs		56 kg	123.5 lbs
57 kg	125.7 lbs		58 kg	127.9 lbs
59 kg	130.1 lbs		60 kg	132.3 lbs

Рисунок 10.1 Фрагмент таблиці перетворення кілограм у фунти

Виділеного ключа така таблиця не має. Абстрактний тип "таблиця" відображає той факт, що, взагалі кажучи, немає ніякого внутрішнього порядку серед рядків таблиці.

У той час, як ми маємо операцію траверсу (переміщення) всередині інших елементарних і абстрактних типів даних, не існує така операція для таблиць. Це означає, що немає ніякого способу, щоб дослідити вміст рядків таблиці, крім перебору рядків ВІД і ДО. Насправді не існує навіть операції визначення (без прямого перебору), скільки рядків містить така таблиця.

Раніше ми вивчили кілька структур даних, які можуть бути використані для ефективного розміщення інформації у пам'яті. Кращим вибором, з урахуванням наших поточних знань, є використання для цього бінарних дерев пошуку (особливо якщо вони збалансовані), що дає оцінку $O(log_2N)$ для операцій вставки, видалення і пошуку. Зараз ми розглянемо техніку, звану хешуванням, яка прагне зробити швидкість виконання цих операції постійною і незалежною від числа елементів. Це може здатися неможливим, але хешування дійсно дуже близьке до досягнення цієї мети. Тому хешування, поряд з бінарними деревами, надзвичайно широко застосовується при роботі з даними (і не тільки! Сучасні системи шифрування інформації теж активно використовують техніку хешування).

Хеш-таблиці дозволяють швидко шукати значення, оскільки не зберігають їх в попередньо відсортованому списку, а організують їх особливим чином і легко можуть *розрахувати* місцезнаходження потрібного елемента. Припустимо, у нас є невелика компанія з 20 співробітників і нам хотілося б отримувати інформацію про кожного з них по чотиризначному ідентифікаційному номеру (ІН). Один із способів збереження даних — сформувати масив зі 100 елементів, в якому ІН будуть відповідати індекси (позиції) *(N mod 100)*. Наприклад, співробітник з ІН 2190 буде розміщений в позиції 90, з ІН 2817 - в позиції 17, а з ІН 3078 - в позиції 78. Щоб знайти певного працівника, досить скористатися формулою *(ІН mod 100)* і переглянути відповідний елемент в масиві. Подібна операція займе час аж *O(1),* тобто на ділі виявиться швидшим, ніж найшвидший пошук, з числа відомих нам.

На практиці не все так просто. Якщо співробітників виявиться досить багато, то серед них знайдуться двоє, у яких ІН буде відповідати одному і тому ж значенню. Скажімо, ІН 2817 і 1317 займуть в таблиці одну позицію - 17. Але навіть незважаючи на цей недолік, виконане перетворення значень - вдалий початок! Воно є відправною точкою для роботи з хеш-таблицями.

10.2 Основи хеш-таблиць

Хеш-таблиця поєднує дані з комірками пам'яті [1]. Найчастіше такий зв'язок утворюється між значенням ключа, наприклад ІН або прізвищем, і набагато більшим записом, скажімо, інформацією про співробітника або клієнта. Через цю особливість хеш-таблиці іноді називають *асоціативними масивами*, або, висловлюючись менш офіційною мовою, *словниками*.

Перетворення значень ключа до виду, придатного для використання в хеш таблиці, називається *хешуванням* і описується особливими математичними *хеш-функціями*. Хешовані значення ключів часто схожі, тому хороші хеш-функції розподіляють їх таким чином, щоб вони розташовувалися в різних позиціях в таблиці.

Припустимо, ми хочемо знаходити записи про співробітників в хеш-таблиці за прізвищами. Якщо у нас працюють два співробітники - Іванов і Іванова, в ідеалі функція хешування повинна зв'язати їх прізвища з двома різними рядками. У подібних випадках часто генеруються кілька безглуздих значень, які виглядають так, ніби справжнє значення ключа було розбите на шматки.

Якщо в хеш-таблиці розміщується досить багато значень, рано чи пізно все одно знайдуться два ключа, які виявляться пов'язаними з однією і тією ж позицією в таблиці. Така ситуація називається *колізією*. Якщо вона відбудеться, нам знадобиться *політика розв'язання колізій*, яка визначає, що потрібно в цьому випадку робити.

Рисунок нижче показує хеш-таблицю, в якій кожен ключ з множини ключів ***K*** зіставляється з індексом в таблиці, що генерується за допомогою хеш-функції.

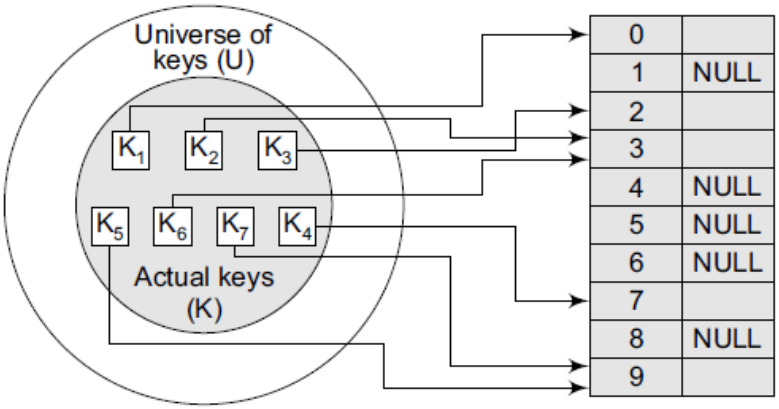


Рисунок 10.2 Приклад хеш-таблиці

Ключі K_2 і K_6, а також K_5 і K_7 демонструють виникнення колізії (конфлікту). Рядки таблиці з індексами 1, 4-6 і 8 є порожніми і готовими до розміщення нових записів.

Коефіцієнт заповнення хеш-таблиці відображає те, наскільки багато в ній записів з даними у порівнянні з її об'ємом. Цей показник впливає на ймовірність виникнення колізій. Більше шансів, що вони будуть мати місце при додаванні ключа в таблицю, заповнену на 95%, ніж на 10%.

Таким чином, для хеш-таблиці потрібні:

- структура для зберігання даних;
- функція хешування для зв'язування рядків структури зберігання з ключем даних, що зберігаються;
- політика розв'язання колізій, яка визначає, що потрібно робити, коли хешовані ключі співпадають.

Щоб бути корисною, хеш-таблиця повинна як мінімум додавати нові елементи і надавати доступ до збережених раніше. Стане в нагоді і здатність видаляти хешований ключ, яка, на жаль, іноді відсутня.

У міру роботи хеш-таблиця може повністю заповнитися або її коефіцієнт заповненності стає настільки близьким до 100%, що колізії будуть дуже ймовірні й постраждає продуктивність роботи з цією таблицею. У такому випадку знадобиться удосконалити алгоритм, щоб визначити, де і як змінюється розмір хеш-таблиці, і збільшити його. Іноді доопрацювання будуть потрібні, щоб таблицю зменшити і звільнити невикористаний простір, наприклад якщо вона розрахована на 1 млн записів, а у нас їх всього 100.

Найпростіший спосіб змінити розміри хеш-таблиці - створити нову таблицю бажаного розміру і *рехешувати* в неї всі елементи з вихідної структури. Його можна застосовувати майже до всіх таблиць, хоча є й інші варіанти здійснення тієї самої дії, наприклад пряме зв'язування.

10.3 Хеш-функції

Хеш-функція являє собою математичну формулу, яка при застосуванні до ключа обчислює ціле число, яке може бути використане в якості індексу для хеш-таблиці. Основною властивістю хорошої хеш-функції є те, що результати операції хешування ключів повинні бути рівномірно розподілені випадковим чином. Така функція видає унікальний набір цілих чисел в межах деякого прийнятного діапазону з метою зменшення числа колізій. На практиці не існує хеш-функції, яка б виключала колізії повністю. Гарна хеш функція здатна тільки звести до мінімуму число колізій, розподіляючи елементи більш-менш рівномірно по всьому масиву.

Якими ж властивостями повинна володіти хороша хеш-функція? Таких властивостей всього три.

Бюджетність. Витрати на обчислення хеш-функції повинні бути

невеликими, так щоб час хешування був меншим, ніж використання інших підходів до представлення даних. Наприклад, якщо алгоритм бінарного пошуку може шукати елемент у відсортованій таблиці з N елементів за час $O(\log_2 N)$, то хеш-функція повинна виконуватися ще швидше.

Детермінізм. Хеш-функція повинна бути детермінованою. Це означає, що одне і теж значення хеш-функції має генеруватися для одного і того ж значення ключа. Цей критерій виключає хеш-функції, які залежать від зовнішніх параметрів середовища виконання (наприклад, часу доби) і від адреси пам'яті хешованого об'єкта (так як адреса об'єкта може змінюватися в процесі обробки).

Однорідність. Гарна хеш-функція повинна генерувати індекс в таблиці по ключу якомога більш рівномірно по всій своїй області значень. Це означає, що ймовірність появи кожного значення хеш-функції в області значень повинна бути однаковою. Властивість однорідності дозволяє звести до мінімуму число колізій.

Хеш-функцій з заданими властивостями існує досить велика кількість і кожен день з'являються нові. Тут ми розглянемо чотири з них, досить широко використовуваних на практиці.

10.3.1 Обчислення залишку

Це найпростіший метод хешування цілого ключа K. Цей метод ділить націло ключ K на число M $(K > M)$, а потім використовує отриманий в результаті залишок. В цьому випадку, хеш-функція може бути задана у вигляді

$$h(K) = K \bmod M \qquad (10.1)$$

Метод залишку досить хороший для будь-якого значення M і, оскільки функція вимагає тільки одну операцію ділення, метод працює дуже швидко. Проте, для вибору оптимального значення M повинні бути прийняті додаткові запобіжні заходи.

Наприклад, припустимо, що M є парним числом, тоді $h(K)$ буде парним при парному K і $h(K)$ буде непарним, якщо K непарний. Якщо всі можливі ключі рівновірогідні, то це не є проблемою. Але якщо парні ключі зустрічаються частіше, ніж непарні, то обчислення залишку не буде генерувати хеш-значення рівномірно.

Як правило, краще всього вибирати M простим числом, тому що просте число збільшує ймовірність того, що генерація хеш-значення по ключу відбуватиметься рівномірно по всій області значень функції. M також не повинно

бути занадто близьким до степеня числа 2. Тому що у цьому випадку функція буде просто видавати молодші *K* біт двійкового подання ключа *K*.

Отримання залишку надзвичайно просто в реалізації. Наступний фрагмент коду ілюструє його:

Приклад 10.1 Реалізація на мові C хеш-функції отримання залишку

```
int h (int k) {
        int const m = 97; // просте число
        return (k % m);
}
```

Потенційним недоліком отримання залишку є те, що при використанні цього методу, послідовні ключі будуть генерувати і послідовні значення хеш-функції. З одного боку, це добре, тому що гарантує, що послідовні ключі не будуть перекриватися, а з іншого боку, це означає, що будуть займатися послідовні елементи таблиці, а не випадкові. А це вже (як ми побачимо далі) може призвести до значного погіршення продуктивності.

10.3.2 Мультиплікативний метод

Алгоритм мультиплікативного методу є наступним:

Крок 1: Вибрати константу A таку, що 0 <A <1.
Крок 2: Помножити ключ K на A.
Крок 3: Взяти дробову частину добутку кроку 2.
Крок 4: Помножити результат кроку 3 на розмір хеш-таблиці M.
Крок 5: Взяти цілу частину результату Крок 4.

Отже, хеш-функцію можна визначити як:

$$h(K) = [M * [K * A]], 0 < A < 1 \qquad (10.2)$$

Найголовніша перевага цього методу полягає в тому, що він працює практично з будь-яким значенням *A*. Хоча оптимальний вибір значення *A* все ж існує. Дональд Кнут в своїй знаменитій багатотомній праці, присвяченій алгоритмам, припустив, що найкращим вибором для *A* є вибір числа $\varphi = 0.6180339887$ ("золотий" переріз).

Приклад реалізації мультиплікативної хеш-функції наведено нижче:

Приклад 10.2 Реалізація на мові C мультиплікативної хеш-функції

```
int h (int k, int m) {
        const double fi = 0.6180339887;        // число Кнута
```

```
        int i;  // ціла частина від ділення дійсного числа
        return (m*modf(k*fi, &i));
}
```

Функція стандартної бібліотеки STL мови C *modf()* з прототипом

*double modf(double x, double * intptr);*

розбиває значення з плаваючою комою ***x*** на дробову та цілу частини. Кожна з цих частин має той самий знак, що і ***x***. Ім'я функції повертає дробову частину числа ***x*** зі знаком. Цілочисельна частина результату повертається у вигляді значення з плаваючою комою *intptr*.

10.3.3 Метод середини квадрата

Метод середини квадрата є хорошою хеш-функцією, яка працює всього за два кроки:

Крок 1: Обчислюється квадрат значення ключа K. Тобто, знаходимо K^2.
Крок 2: З отриманого цілого числа беремо R середніх цифр. Це і буде індекс в хеш-таблиці.

Алгоритм добре працює, тому що більшість або всі цифри цілочисельного подання ключа впливають на результат. Це відбувається тому, що всі цифри в цілочисельному поданні ключа сприяють породженню середніх цифр значення його квадрата. Таким чином, в результаті не домінує розподіл останніх цифр або перших цифр значення ключа. У методі середини квадрата, одні і ті ж *R* цифр повинні вибиратися для всіх ключів.

Приклад 10.3 Приклад розрахунку хеш-функції методом середини квадрата

Розрахувати значення хеш-функції для ключів *1234* і *5642* з використанням методу середини квадрата. Хеш-таблиця має 100 комірок пам'яті.

Рішення. Звернемо увагу на те, що хеш-таблиця має 100 комірок пам'яті, чиї індекси варіюються від 0 до 99. Це означає, що тільки дві цифри необхідні для подання ключа індексу в хеш-таблиці, так що $R = 2$. При $K = 1234$, $K^2 = 1522756$, $h(K) = h(1234) = 27$. При $K = 5642$, $K^2 = 31832164$, $h(5642) = 21$. Зауважимо, що в якості "середини числа" ми вибирали 3-ю і 4-у цифру, починаючи з правої сторони.

Приклад реалізації методу наведено нижче.

Приклад 10.4 Реалізація на мові Сі хеш-функції методу середини квадрата

```
// хеш-функція застосовується для вилучення 16 середніх двійкових цифр
// 32 - розрядного двійкового числа
int h (int k) {
        int     r=16, // число видобутих середніх двійкових цифр (розрядів)
                sqr,  // квадрат числа k
                shift; // величина зсуву вправо на задану кількість двійкових
                       розрядів (цифр)
        sqr = k*k;
        shift = (32 – r) / 2;
        return ((sqr >> shift) & 0xffff);
}
```

10.3.4 Хеш-функції для ключів – символьних рядків

Якщо ключі не є числами, то вони повинні бути перетворені в цілі числа перед застосуванням описаних вище хеш-функцій. Для цього є кілька способів. Наприклад, для рядка символів в якості двійкового числа може інтерпретуватися внутрішній двійковий код кожного символа рядка. Недоліком цього є те, що для більшості ЕОМ внутрішні коди всіх літер або цифр дуже схожі.

У методі злиття для створення деякого цілого числа теж використовуються коди ANSI кожного символа рядка. Так, код великої літери кирилиці 'І' має значення 0xB2, а малої літери 'в' значення 0xE2. Ключ «Іван» після злиття всіх кодів символів подається цілим числом 0xB2E2E0ED. Для приведення великого числа до прийнятного розміру далі може бути використаний метод середини квадрата.

10.4 Розв'язання колізій

Як вже говорилося раніше, яким би не був вибір хеш-функції, колізії будуть відбуватися кожного разу, коли хеш-функція відображає два різних ключі в одне й те саме місце хеш-таблиці. Очевидно, що два і більше записів не можуть бути збережені в одному і тому ж самому місці. Таким чином, при хешуванні, для розв'язання колізій слід використовувати додаткові спеціалізовані методи, що запобігають накладанню різних даних одне на одного. Розглянемо два з найбільш популярних методів розв'язання колізій у хешуванні:

- Метод відкритої адресації.

- Метод ланцюжків (пряме зв'язування).

10.4.1 **Метод ланцюжків (пряме зв'язування)**

У хеш-таблиці з прямим зв'язуванням (Рисунок 10.3) значення ключів зберігаються в спеціальних наборах записів, званих *блоками переповнення*. Кожен з елементів самої хеш-таблиці є фактично дескриптором (покажчиком) зв'язного списку, в якому знаходяться прив'язані до блоку переповнення елементи. У самій таблиці в позиції, що задається хеш-функцією, зберігається тільки значення покажчика адреси "прив'язаного" до цього місця блоку (так званий метод із *зовнішніми ланцюжками*). Тобто, хеш-таблиця є простою адресною таблицею і записів з інформаційними полями (даними) не містить.

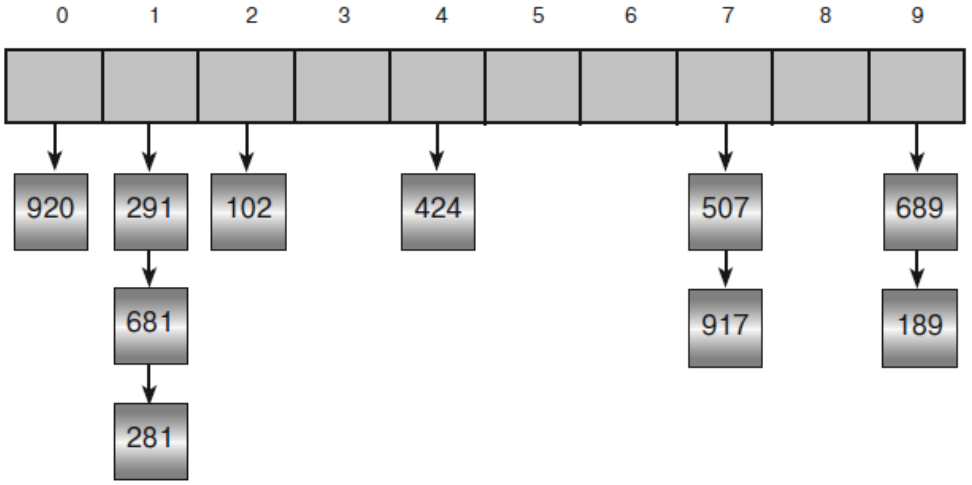


Рисунок 10.3 Ілюстрація методу ланцюжків

Спочатку всі покажчики обнулені. Будь-який запис, що надходить у хеш-таблицю за адресою *h(key)* розміщується у блоці переповнення (зв'язний список) для цієї адреси. Даний спосіб дозволяє створювати хеш-таблиці великих обсягів, що забезпечує простоту при розв'язанні колізій. Крім того, коли елементи таблиці мають змінну довжину або коли блоки зв'язних списків розміщуються на зовнішніх пристроях пам'яті, цей спосіб є найбільш прийнятним.

Зв'язні списки в оперативній пам'яті створюються у динамічній області пам'яті, коли для кожного елемента виділяється пам'ять і для кожної хеш-адреси створюється свій блок переповнення (ланцюжків).

Щоб додати ключ до хеш-таблиці, його потрібно спочатку співвіднести з відповідним блоком через функцію хешування, а потім вставити новий елемент у верхню частину (на початок) зв'язного списку. Кожна з цих двох операцій потребує *O(1)* кроків, тобто займе дуже мало часу. За визначенням, хеш-таблиця не повинна

містити дубльованих значень. Якщо в ній *B* блоків і *N* елементів, які розподілені досить рівномірно, то кожен зв'язний список блоку буде включати приблизно *N/B* елементів. Таким чином, щоб перевірити, чи присутній новий елемент в блоці, знадобиться ще *O(N/B)* кроків. Це означає, що для додавання елемента в хеш-таблицю в середньому знадобиться всього *2*O(1) + O(N/B) = O(N/B)* кроків.

Пошук елементів у хеш-таблиці пройде швидше, якщо зв'язані списки будуть містити ключі у відсортованому порядку. У цьому випадку алгоритм зробить висновок, що ключа немає, якщо дійде до значення ключа в списку більшого, ніж ключ запису, що додається. У цьому випадку алгоритм розв'язання колізій не стане переглядати список до кінця. Теоретично час його роботи складе *O(N/B),* але на практиці воно буде меншим.

Щоб знайти потрібний запис, програмі необхідно хешувати ключ запису і визначити, в якому з блоків він може міститися, а потім рухатися по зв'язному списку до тих пір, поки не буде досягнуто його кінець або не виявиться елемент зв'язного списку з таким значенням ключа. Як і у випадку з додаванням елемента, належить виконати максимум *O(N/B)* кроків.

Щоб видалити запис з хеш-таблиці з прямим зв'язуванням потрібно хешувати ключ запису для знаходження потрібного блоку, і виконати відповідну операцію в зв'язному списку. Для хешування знадобиться *O(1)*кроків, безпосередньо видалення займе *O(N/B)* кроків. Таким чином, загальний час складе *O(N/B).*

Хеш-таблиця з прямим зв'язуванням може розширюватися і стискатися в міру необхідності, тому нам не потрібно спеціально змінювати її розмір. Однак якщо зв'язні списки в блоках стануть занадто довгими, пошук і видалення елементів займуть багато часу. В цьому випадку необхідно буде збільшити саму хеш-таблицю, щоб створити в ній більше блоків. Оскільки при рехешуванні таблиці не треба проводити пошук дублікатів до кінця зв'язного списку в кожному блоці, повністю впоратися з операцією можна за час *O(N).*

Приклад 10.5 Приклад реалізації алгоритму методу ланцюжків на мові C

```
/* Структура вузла блоку переповнення хеш-таблиці */
typedef struct node_HT {
```

```
        int value;
        node_HT *next;
} node;

/* Ініціалізація комірок хеш-таблиці при її створенні. Операція вимагає O(N)
   часу */

void initializeHashTable (node* hash_table[],  /* масив покажчиків на блоки
                                                  ланцюжків хеш-таблиці  */
                          int m                /*  розмір хеш-масиву */)  {

        int i;
        for(i=0; i<m; i++)
                hash_table[i]=NULL;

        return;
}


/* Додавання нового елемента в хеш-таблицю. Елемент вставляється в
початок зв'язного списку блоку, покажчик на який зберігається в місці,
заданому функцією хешування h(k). Час роботи операції вставки є O(1),
новий ключ завжди додається в якості першого елемента списку, незалежно
від розміру зв'язного списку */

node* insertValue (node* hash_table[], int key, int& error_key) {

        node *new_node;
        new_node = (node*) malloc(sizeof(node));
        if (!new_node)       /* немає доступної пам'яті */
                return 0;

        error_key = 0;

        /* Перед тим, як іти далі, у цьому місці слід впевнитися у тому, що у
        зв'язному списку блока відсутній елемент з таким самим значенням
        ключа key. Якщо це не так – слід проінформувати викликаючу
        функцію встановивши error_key = 1 і завершити роботу поточної
        функції з кодом 0! */

        /* Вставка нового елемента */
        new_node -> value = key;
        new_node -> next = hash_ table[h(key)];     // встановлюємо на
                                                    // колишній перший
                                                    // елемент блоку
                                                    // переповнення
        hash_table[h(key)] = new_node;        // записуємо адресу нового
                                              // першого в  хеш-таблицю
        return new_node;
}
```

```
/* Пошук значення ключа в хеш-таблиці. Пошук проводиться у зв'язному
списку, покажчик на початок якого зберігається в місці, заданому функцією
хешування h(к). Якщо пошук успішний, функція повертає покажчик на вузол
зв'язаного списку; в іншому випадку вона повертає NULL. У гіршому випадку
час роботи операції пошуку пропорційний розміру зв'язаного списку блока. */

node* searchValue(node* hash_table[], int key) {

        node *ptr;
        ptr = hash_table[h(key)];
        while ((ptr!=NULL) && (ptr –> value != key))
                ptr = ptr –> next;
        if (ptr–>value == key)
                return ptr;
        else
                return NULL;
}


/* Видалення елемента з хеш-таблиці. Щоб видалити елемент із зв'язного
списку блоку хеш-таблиці, адреса початку якого зберігається в місці хеш-
таблиці, заданому хеш-функцією h(к), ми повинні знати адресу попередника
видаляємого елемента у блоці переповнення. Ми робимо це за допомогою
додаткового покажчика pSave. Час виконання операції видалення такий же,
як і для операції пошуку. Функція повертає 0 в разі успішного виконання
операції і 1 - якщо елемент із заданим ключем не знайдено. */

int deleteValue (node* hash_table[], int key) {

        node *pSave, *ptr;

        // ініціалізація пошуку
        pSave = NULL;
        ptr = hash_table[h(key)]; // визначаємо адресу блоку зберігання
                                  // елемента

        // пошук елемента, що видаляється, в блоці хеш-таблиці
        while ((ptr != NULL) && (ptr -> value != key)) {
                pSave = ptr;
                ptr = ptr -> next;
        }

        if (ptr) {
                pSave -> next = ptr -> next; // зв'язуємо попередній елемент
                                             // перед тим, що видаляємо, і
                                             // наступний елемент в блоці
                free (ptr);
                return 0;       // елемент видалено
        }

        else
```

```
            return 1;      // елемент не знайдений, операція не виконана
    }
```

10.4.2 Метод відкритої адресації

Ще один метод реалізації розв'язання коллізій у хеш-таблицях - *відкрита адресація*. У цьому методі використовується тільки одна хеш-таблиця з *N* елементами, блоки переповнення не створюються. Суть методу полягає в тому, щоб після входу в таблицю за адресою, що надає хеш-функція *h(k)*, повністю відмовитися від покажчиків і просто по деякому правилу (алгоритму) переглядати комірки таблиці, поки не будуть знайдені або вільна комірка при вставці нового елементу, або вже існуючий ключ - при пошуку елемента.

У різних видах відкритої адресації використовуються різні функції хешування. Неоднакова і політика розв'язання колізій, але в загальному випадку вона виглядає так: для кожного значення *h(k)* в хеш-таблиці підбирається кілька комірок, і якщо перша вже зайнята, алгоритм пробує використати другу, потім третю і так до тих пір, поки не знайде вільну комірку або не прийде до висновку, що такої немає.

Серія комірок, яку алгоритм підбирає для значення, називається *пробною послідовністю*. За її середньою довжиною добре оцінювати наповненість хеш таблиці. В ідеалі пробна послідовність повинна дорівнювати 1 або 2, більші числа говорять про заповненість хеш-таблиці і необхідність рехешування.

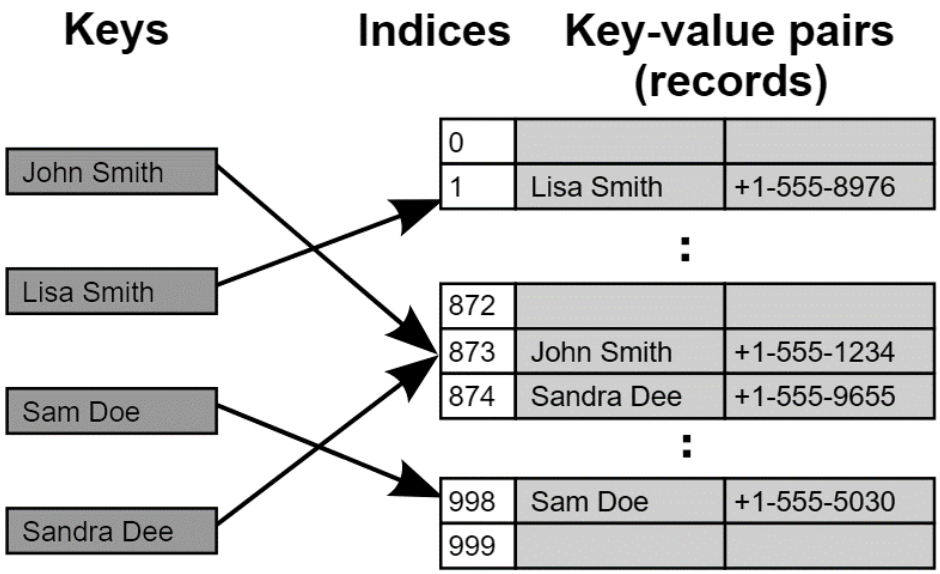


Рисунок 10.4 Приклад хеш-таблиці з відкритою адресацією і лінійним пробіруванням

Після того, як відбувається колізія, метод відкритої адресації або закритого хешування обчислює нову позицію з використанням пробної послідовності і запис, який викликав колізію, зберігається саме в цьому місці. У такому методі, всі значення, включаючи і інформаційні дані, зберігаються в комірці самої хеш-таблиці. Хеш-таблиця містить два типи значень: *сторожові значення* (наприклад, *-1*) і *значення даних*. Наявність сторожового значення вказує на те, що місце розташування не містить інформаційних даних в даний час, і може бути використано для зберігання нового значення даних.

Коли функція хешування відображає ключ на конкретну комірку таблиці, то в комірці, перш за все перевіряється сторожове значення. Якщо воно містить *-1*, то значення даних можуть бути збережені в цій комірці - вона порожня. Якщо сторожове значення відмінне від *-1* - комірка вже містить дані і повинні розглядатися інші комірки в прямому напрямку, щоб знайти вільну. Якщо не знайдено жодної вільної комірки - маємо переповнення таблиці (у цьому випадку рехешування таблиці стає обов'язковим!).

Процес перевірки комірок пам'яті в хеш-таблиці називається *пробіруванням*. Відкритий метод адресації може бути реалізований з використанням лінійного чи квадратичного пробірування, а також технології подвійного хешування.

Розглянемо роботу хеш-методу відкритої адресації з *лінійним пробіруванням*. У цьому методі, якщо якесь значення вже зберігається в місці, на яке вказує хеш-функція *h(k),* то наступною використовуваною хеш-функцією для вирішення колізії буде:

$$h(k,i) = [h'(k) + i] \% m, \qquad (10.3)$$

де m - розмір хеш-таблиці;

$h'(k) = k \% m;$

i - номер проби, який змінюється від 0 до $m-1$.

Таким чином, для даного ключа k, перше значення хеш-функції (10.3) буде якраз і складати $(k \% m)$, тому що в перший момент $i = 0$. Якщо місце вільно, значення зберігається в ньому, інакше друга проба ($i = 1$) згенерує нову адресу місця розташування $[h'(k) + 1] \% m$. Якщо і це місце розташування зайнято, то наступні проби будуть генерувати наступні адреси як:

$[h'(k) + 2] \% m,$

[h'(k) + 3] % m,

[h'(k) + 4] % m,

[h'(k) + 5] % m,

тощо, поки вільне місце не буде знайдено.

Приклад 10.6 Приклад використання лінійного пробірування

Розглянемо хеш-таблицю розміру *m = 10*. За допомогою лінійного пробірування, додамо в неї такі ключі 72, 27, 36, 24, 63, 81, 92, и 101. Тут хеш-функція буде мати вигляд *h'(k) = k % 10*.

Спочатку порожня хеш-таблиця буде виглядати так:

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

Крок 1 *k = 72*

h(72, 0) =(72 % 10 + 0) % 10 = (2) % 10 = 2

Так як *T[2]* вільно, вставляємо ключ 72 в це місце:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Крок 2 *k = 27*

h(27, 0) =(27 % 10 + 0) % 10 = (7) % 10 = 7

Так як *T[7]* вільно, вставляємо ключ 27 в це місце:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	27	–1	–1

Так буде тривати ще чотири кроки, поки ми не додамо в таблицю ключ 81:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	–1	36	27	–1	–1

Крок 7 *k = 92*

h(92, 0) =(92 %10 + 0) % 10 =(2) % 10 = 2

Місце *T[2]* вже зайнято, тому ми не зможемо зберегти 92 в позиції *T[2]*. Спробуємо знайти нове місце для розміщення. Це вже буде проба з індексом *i= 1.*

k = 92, h(92, 1) =(92 % 10 + 1) % 10 = (2 + 1) % 10 = 3

Місце *T[3]* теж зайнято, тому продовжуємо пробувати з індексом *i = 2:*

k = 92, h(92, 2) =(92 % 10 + 2) % 10 = (2 + 2) % 10 = 4

Місце *T[4]* знову зайнято, але ми не впадаємо у відчай! - пробуємо тепер *i=3*:

k = 92, h(92, 3) =(92 % 10 + 3) % 10 = (2 + 3) % 10 = 5

Voi la! Праця винагороджена! Місце *T[5]* вільно і ми з чистою совістю розміщуємо ключ 92 в цій комірці.

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	92	36	27	–1	–1

Крок 8 k = 101

h(101, 0) =(101 % 10 + 0) % 10 =(1) % 10 = 1

Місце *T[1]* зайнято і ми починаємо пробувати з індексу *i = 1:*

k = 101, h(101, 1) =(101 % 10 + 1) % 10 = (1 + 1) % 10 = 2

T[2] також зайнятий і ми починаємо повторювати пробірування з різними значеннями поки не згенеруємо незайняту адресу 8.

Процедура пошуку значення в хеш-таблиці така ж, як і для зберігання значення в хеш-таблиці. При пошуку значення в хеш-таблиці, пошуковий ключ хешується і ключ запису, що зберігається в цьому місці таблиці порівнюється з пошуковим значенням. Якщо збіг є, то операція пошуку завершується і успішна. Час пошуку в даному випадку буде становити *O(1)*. Якщо ключі не збігаються, то функція пошуку починає пробірування по вищеописаному алгоритму в хеш-таблиці, поки:

- значення не буде знайдено;
- функція пошуку зустріне порожню комірку в таблиці (зі сторожовим значенням *-1*), що вказує, що такого значення пошукового ключа в таблиці немає;
- функція пошуку закінчується, оскільки вона досягає кінця таблиці, а значення все ще не було знайдено.

У гіршому випадку, операції пошуку, можливо, доведеться зробити *m-1* порівнянь (де *m* - розмір хеш-таблиці), а час роботи алгоритму пошуку може відповідно зайняти *O(m)* часу. Найгірший випадок буде зустрічатися, коли після сканування всіх *m-1* елементів хеш-таблиці, потрібне значення буде присутнє на останньому місці або його взагалі немає в таблиці.

Таким чином, ми бачимо, що зі збільшенням числа колізій, відстань між індексом масиву, який обчислюється попередньо за допомогою хеш-функції і

фактичним місцем розташування елемента збільшується, тим самим збільшуючи і час пошуку.

Лінійне пробірування знаходить вільне місце, виконавши лінійний пошук в таблиці, починаючи з позиції *h(k)*. Перевага даного методу - в його простоті. Якщо необхідно, пробна послідовність пройде по кожній комірці масиву і вставить елемент у вільне місце, якщо воно ще залишилося. Але є і супутній недолік — так звана первинна кластеризація, яка проявляється в утворенні великих блоків зайнятих суміжних комірок і призводить до довгих пробних послідовностей. В результаті при додаванні нового елемента і його хешуванні до будь-якої комірки в кластері пробна послідовність змушена пройти через весь кластер, перш ніж знайде вільну комірку. Робочі характеристики лінійного пробірування чутливі і до розподілу вхідних значень.

У міру заповнення хеш-таблиці, формуються і ростуть кластери зайнятих послідовних комірок, пропорційно зростає і час, необхідний для пошуку. До всього цього, коли нове значення має бути вставлено в таблицю на зайняте місце - значення вставляється в кінці кластера, що знову-таки збільшує довжину вже зайнятого кластера. Як правило, вставка проводиться між двома кластерами, які відокремлені один від одного однією вільною коміркою. З лінійним пробіруванням зростають шанси на те, що наступні вставки відбудуться саме в це місце, тим самим потенційно різко збільшуючи довжину зайнятого кластера на величину набагато більшу, ніж один елемент. Чим більше число колізій, тим довші проби, які необхідні, щоб знайти вільне місце і тим меншою стає продуктивність хешування. Це явище і називається *первинною кластеризацією*. Щоб уникнути первинної кластеризації, використовуються більш складні методи, такі як квадратичне пробірування і подвійне хешування.

Видалення елемента. Відкрита адресація дозволяє відносно швидко додавати і знаходити елементи, якщо таблиця не надто повна, але не дає можливості видаляти їх так, як це робиться при прямому зв'язуванні в методі ланцюжків. Справа в тому, що один елемент таблиці може бути частиною пробної послідовності для іншого елемента, порушивши її, ми вже не зможемо знайти інший елемент, що розміщений далі у хеш-таблиці.

Припустимо, елементи *A* і *B* хеш-таблиці пов'язані з одним і тим самим індексом I_A. Спочатку, ми додамо в зазначену позицію елемент A, а потім використовуємо її у пробній послідовності, щоб перейти до другої позиції I_B і додати в неї елемент *B*. Тепер уявіть, що елемент вже A видалений (в комірку записано сторожове значення -1) і потрібно знайти елемент *B*. Звернувшись при пошуку *B* послідовно спочатку до індексу I_A, ми побачимо вже порожній запис, що буде означати завершення пошуку і тому помилково вирішимо, що елемент *B* відсутній!

Якщо позначати видаляємий елемент у комірці саме як видалений, а не робити запис комірки порожнім сторожовим елементом, зазначеної проблеми вдасться уникнути. Скажімо, в масиві, де містяться 32-бітові цілі числа зі знаком, комірку масиву, що не має значення, можна подавати як -2 147 483 648, а видалену - як +2 147 483 647. Тоді, додаючи новий елемент в хеш-таблицю, ми розмістимо його в порожній комірці, якщо знайдемо таку в пробній послідовності, а займаючись пошуком елемента, будемо вести його до тих пір, поки не виявимо відповідне значення.

Недолік такого підходу – заповнення з часом хеш-таблиці видаленими записами. Це не сповільнить вставку нових елементів, але негативно позначиться на пошуку вже існуючих. У гіршому випадку, якщо таблиця виявиться повністю заповнена поточними і видаленими елементами, щоб знайти потрібний елемент або зробити висновок про його відсутність, нам може знадобитися пошук по всій таблиці.

При великій кількості видалених елементів доцільно рехешувати поточні значення і скинути видалені комірки таблиці, щоб вони вже містили не спеціальне значення видалення, а сторожові значення. Якщо в таблиці з розумним коефіцієнтом заповнення міститься *N* елементів, це повинно зайняти *O(N)* часу.

Приклад 10.7 Приклад алгоритму на мові С при лінійному пробіруванні

/ Ініціалізація хеш-таблиці при її створенні. m - необхідний розмір хеш-таблиці. Хеш-таблиця зберігає у якості елементів даних цілі числа типу* int. *У разі успіху функція повертає адресу першого елемента таблиці. У разі відсутності необхідної пам'яті - функція поверне NULL. Операція вимагає O(N) часу */*

```
int* initializeHT (int m) {
        int     I,
```

```
        *ht;
    ht = (int*) malloc(m*sizeof(int));
    if (!ht)
        return NULL;
    for (i=0; i<m; i++)
        ht[i]=INT_MIN;    // ініціалізуємо порожні рядки мінімальним
                          // значенням типу int
    return ht;
}
```

/ Додавання нового елемента в хеш-таблицю. Елемент вставляється в позицію, зазначену хеш-функцією, або в позицію, виявлену пробіруванням. Час роботи операції вставки в кращому випадку є O(1), в гіршому - O(m). Функція у своєму імені повертає число:*
0 - за відсутності колізій;
число від 1 до m-1 - число проб при наявності колізій;
-1 - при відсутності вільного місця в таблиці/*

```
int insertValue (int ht[], int m, int key) {
    int     val,    // значення хеш-функції
            quantity,       // наповненість хеш-таблиці
            flag=0,         // флаг завершення пробірування, якщо 0
            i;      // поточний індекс в таблиці
    val =  key % m;  // обчислюємо хеш-функцію
    if ( ht[val] == INT_MIN ) {     // немає колізій - зразу розміщуємо
                                    // значення ключа в таблиці
        ht[val] = key;
        return 0;
    }
    // є колізія!
    else {
        flag = 1;
        // починаємо лінійне пробірування
        for (i = val +1, quantity = 1;
             quantity < m && flag;
             i = ++i % m, quantity++)
                if ( ht[i] == INT_MIN ) {  // якщо є вільне місце
                    ht[i] = key;
                    flag = 0;
                }
        if (!flag)
```

```
                return quantity;    // ключ розміщений, повертаємо
                                    // наповненість таблиці
        else
                return -1;    // ключ не розміщено, в таблиці немає місця
        }
}
```

/ Пошук значення ключа в хеш-таблиці. Пошук проводиться з місця, зазначеному хеш-функцією. При наявності колізії - запускаємо лінійне пробірування. Якщо пошук успішний, функція повертає індекс ключа в таблиці; в іншому випадку вона повертає -1. У гіршому випадку час роботи операції пошуку пропорційний розміру таблиці. */*

```
int  searchValue(int ht[], int m, int key) {

        int val,        // індекс пошукового ключа в таблиці (якщо є)
        quantity,       // наповненість таблиці
        flag,           // флаг відсутності ключа в таблиці, якщо 1
        i;              // поточний індекс в таблиці

        flag = 1;       // початково важаємо, що ключ відсутній
        val =  key % m;  // значення хеш-функції

        if ( ht[val] == key )       // якщо ключ знайдено одразу!
                flag = 0;

        else  {// ключ не знайдено, запускаємо лінійне пробірування
                for     (i = val +1, quantity = 1;
                        quantity < m && flag;
                        i = ++i % m, quantity++)
                                if ( ht[i] == key )  {  // ключ знайдено!
                                val = i;        // присвоюємо значення знайденного
                                                // індекса ключа в таблиці
                                flag = 0;       // скидаємо флаг пробірування
                                }

                if (flag)
                        return -1;    // ключ не знайдений

                else if (val != INT_MAX)
                        return val;   // ключ знайдений, повертаємо індекс
                                      // ключа в таблиці

                else
                        return -1;

        }

}
```

/ Видалення елемента з хеш-таблиці. Видалений елемент таблиці позначається як INT_MAX. Час виконання операції видалення такий же, як і в операції пошуку. Функція повертає 0 в разі успішного виконання операції і 1 - якщо елемент із заданим ключем не знайдено */*

```
int deleteValue(int ht[], int m, int key) {
        int found;
        found = searchValue(ht[], m, key);
        if (found > -1) {
                ht[found] = INT_MAX;
                return 0;
        }
        else
                return 1;
}
```

10.5 Застосування хешування

Одним з переваг техніки хешування є те, що хешування не вимагає місця під зберігання індексів, як у випадку інших структур даних. Крім того, хеш-таблиця забезпечує швидкий доступ до даних і їх швидке оновлення, тим самим хешування відмінно підходить при роботі з даними в режимах реального часу.

З іншого боку, основним недоліком хешування при вставці і пошуку даних є те, що йому, як правило, не вистачає локальності і послідовності розташування данних з послідовними ключами. Це робить вставку і пошук даних в хеш-таблиці в більшій мірі випадковими операціями, навіть при наявності послідовних ключів.

До того ж, вибір ефективної хеш-функції, найкращим чином адаптованої до типу ключа, є скоріше мистецтвом, ніж наукою. Це не рідко призводить до використання на практиці поганих для даного застосування хеш-функцій.

Проте, не дивлячись на недоліки, хеш-таблиці широко використовуються в тих ситуаціях, коли необхідно обробити великі обсяги даних за мінімальний час. Наведемо кілька типових прикладів використання хешування.

Хеш використовується для індексування баз даних. Багато систем управління базами даних зберігають окремий файл, відомий як індексний. Коли дані повинні бути вилучені з файлу, інформація про ключ спочатку шукається у відповідному файлі індексу, який посилається на точне місце розташування запису з данними у файлі бази даних. Ця інформація про ключ в індексному файлі часто

зберігається у вигляді хеш-таблиці, особливо для баз реального часу.

Техніка хешування використовується у файлових системах високопродуктивних сучасних ОС. Такі системи використовують два додаткових методи для підвищення ефективності доступу до своїх файлів. У той час як один з цих методів є кешування, яке зберігає найбільш вживану інформацію в оперативній пам'яті, інший метод - хешування, який знаходить місце розташування потрібного запису даних у файлі набагато швидше, ніж більшість інших методів.

Хешування використовується і для реалізації таблиці символів компілятора C++. Компілятор використовує таблицю символів, щоб вести облік всіх визначених користувачем ідентифікаторів у програмі C++. Хеш полегшує компілятору швидкий пошук імен змінних та інших атрибутів, пов'язаних з символами.

Хешування також широко використовується для пошукових систем Інтернету.

Ключі захисту ліцензій, ключі доступу до персональних даних та ключі шифрування також часто створюються з використанням хешування з елементів даних, які ніколи не змінюються: дати народження, імені, тощо.

Хешування використовується і при роботі з раніше вивченими нами розрідженими матрицями. Розріджена матриця являє собою двовимірний масив, в якому більшість елементів дорівнює 0. Ще одна можливість зберігання розрідженої матриці полягає в тому, щоб зберігати ненульові елементи в якості елементів одновимірного масиву. Тобто за допомогою хешування, можна зберігати двовимірний масив як одновимірний. За допомогою належним чином підібраних хеш-функцій можна одержати взаємно-однозначну відповідність між ненульовими елементами розрідженої матриці і елементами одновимірного хеш-масива.

Хешування використовується і в комп'ютерних іграх. Стан ігрового поля гри часто зберігається у файлі на диску за допомогою хешування.

У комп'ютерній графіці, центральною проблемою є зберігання об'єктів в сцені або поданні. Для цього ми організуємо наші дані шляхом хешування. Хеш може бути використаний для створення сітки відповідного розміру, що покриває графічну сцену. Сітка є нічим іншим, як двовимірним масивом, і існує взаємно-однозначна відповідність того, як слід переходити від двовимірного масиву до

одновимірного. Таким чином, ми зберігаємо сітку у вигляді одновимірного масиву, як вже було сказано на прикладі розріджених матриць. Всі точки, які знаходяться в одному вузлі сітки будуть зберігатися в одному і тому ж самому місці. Якщо комірка містить три точки (три кольори RGB), то ці три точки будуть збережені в одному запису. Відображення вузла сітки в комірку пам'яті здійснюється за допомогою хеш-функції. Основна перевага цього способу зберігання - швидке виконання графічних операцій, таких, наприклад, як пошук найближчого сусіда об'єкта сцени.

10.6 Контрольні завдання і запитання

1. Яким вимогам повинна відповідати хеш-функція? Що станеться, якщо якась із вимог буде проігнорована?

2. Порівняйте розв'язання колізій за допомогою методів ланцюжків і відкритої адресації. Вкажіть їх переваги та недоліки.

3. У чому полягає головний недолік лінійного пробірування?

4. Що таке явище первинної кластеризації? Які методи допомогають боротися з ним?

5. Який головний недолік при використанні метода відкритої адресації?

6. Напишіть програму для створення хеш-таблиці з прямим зв'язуванням.

7. Змініть програму з попередньої вправи так, щоб можна було використовувати сортовані зв'язні списки. Порівняйте середні довжини пробних послідовностей двох програм для випадку, коли в хеш-таблицях задіяно 10 блоків і до 100 елементів.

8. Накресліть графік середньої довжини пробної послідовності для програм з вправ 6 і 7, де йдеться про хеш-таблиці з 10 блоками та 50, 100, 150, 200 та 250 елементами. Який висновок можна зробити на основі даних?

9. Розробіть схему алгоритма та напишіть програму для створення хеш-таблиці з використанням відкритої адресації та лінійного пробірування.

10. Розробіть схему алгоритма та напишіть програму для створення хеш-таблиці з використанням відкритої адресації та квадратичного пробірування.

11. Розробіть схему алгоритма та напишіть програму для створення хеш-таблиці з використанням відкритої адресації та псевдовипадкового пробірування.

12. Розробіть схему алгоритма та напишіть програму для створення хеш-

таблиці з використанням відкритої адресації та подвійного хешування.

13. Лінійне пробірування завжди знаходить порожнє місце у хеш-таблиці, якщо воно існує. Однак квадратичне та псевдовипадкове пробірування, а також подвійне хешування можуть пропускати порожні записи та дійти хибного висновку про те, що таблиця заповнена. Як вибрати розмір таблиці N, щоб запобігти висновку про те, що хеш-таблиця заповнена, якщо це не так?

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Базова література:

1. Алгоритми та структури даних (комп'ютерний практикум): [Електронний ресурс] : навч. посіб. для студ. спеціальності 151 «Автоматизація та комп’ютерно-інтегровані технології» / Укладач: Ю. Є. Грудзинський; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 4,8 МБайт). – Київ : КПІ ім. Ігоря Сікорського, 2022. – 100 с.

2. Коротєєва Т.О. Алгоритми та структури даних. Навчальний посібник. – Львів: Видавництво Львівської політехніки, 2014. – 280 с.

Допоміжна література:

3. Stephens R. Essential Algorithms. A Practical Approach to Computer Algorithms Using Python and C#, 2nd Edition. – Indianapolis: Wilew, 2019. – p. 783.

4. Thareja R. Data Structures Using C, 2nd Edition. - New Delhi: Oxford University Press, 2014. – 557 p.

5. Bhasin H. Algorithms. Design and Analysis. - New Delhi: Oxford University Press, 2015. – 727 p.

6. Tiwari N.K., Agrawal J., Shandilya Sh.K. DATA STRUCTURES. - New Delhi: I.K. International Publishing House Pvt. Ltd., 2016. – 275 p.

ВИКОРИСТАНА ЛІТЕРАТУРА

[1] R. Stephens, Essential Algorithms. A Practical Approach to Computer Algorithms Using Python and C#, 2nd Edition ред., Indianapolis: John Wiley & Sons, Inc., 2019, p. 782.

[2] R. Thareja, Data Structures Using C, 2nd Edition ред., New Delhi: Oxford University Press, 2014, p. 557.

[3] Т. Коротєєва, Алгоритми та структури даних: Навчальний посібник, Львів: Видавництво Львівської політехніки, 2014, p. 280.