Міністерство освіти і науки України

Дніпропетровський національний університет імені Олеся Гончара

Кафедра електронних обчислювальних машин

ПОСІБНИК ДО ВИВЧЕННЯ КУРСУ
«JAVA-ТЕХНОЛОГІЇ ТА МОБІЛЬНІ ПРИСТРОЇ»
АЛГОРИТМИ І СТРУКТУРИ ДАНИХ

Ухвалено Вченою радою факультету фізики, електроніки та комп’ютерних систем

як навчально-методичний посібник

Дніпропетровськ

2016

УДК 004.432/[004.424+004.422.6]

Рецензенти: д-р техн. наук, проф. В. В. Гнатушенко

канд. техн. наук, доц. О. В. Реута

Посібник до вивчення курсу «Java-технології та мобільні пристрої». Алгоритми і структури даних. / О. В. Спірінцева, В. В. Герасимов. — ДНУ ім. О. Гончара, 2015. — 98 с.

Розглянуто основні структури даних та алгоритми їх обробки на платформі Java. Основною метою посібника є формування бази теоретичних та практичних знань для студентів Дніпропетровського національного університету імені Олеся Гончара спеціальності «Комп'ютерні системи та мережі». за напрямом підготовки 6.050102 "Комп'ютерна інженерія".

Затверджено Вченою радою факультету фізики, електроніки та комп'ютерних систем, протокол № 19 від 31 травня 2015 р.

Навчальне видання

Ольга Володимирівна Спірінцева

Володимир Володимирович Герасимов

Посібник до вивчення курсу
«JAVA-ТЕХНОЛОГІЇ ТА МОБІЛЬНІ ПРИСТРОЇ».
Алгоритми і структури даних

Підписано до друку | Формат 60 х 84/16 | Папір друкарський
Друк плоский. Ум. друк. арк. 2,79 | Ум. фарбовідб. 2,79 | Обл. – вид. арк. 3,66
Наклад 50 пр. Зам. №
РВВ ДНУ, 49010, м. Дніпропетровськ, пр. Гагаріна, 72

З М І С Т

В С Т У П

Вивчення алгоритмів і структур даних — основа будь-якого комп'ютерного курсу, не тільки для програмістів і тих, хто вивчає обчислювальну техніку. Кожен, хто користується комп'ютером, бажає, щоб він працював швидше і вирішував масштабніші задачі. Це стосується будь-якої області наукових досліджень, інженерних розрахунків і сучасних програмних систем: від моделювання системи N тіл в фізиці до розшифровки генетичного коду в молекулярній біології; від системи архітектурного проектування до моделювання літаків; від систем керування базами даних до механізмів пошуку в Інтернеті.

При написанні комп'ютерних програм досить часто реалізується метод, який уже був розроблений раніше для вирішення якої-небудь задачі. Такий метод часто не залежить від конкретної мови програмування. Саме метод, а не комп'ютерна програма, описує кроки, які необхідно виконати для вирішення задачі. Термін **алгоритм** застосовується в обчислювальній техніці для описання кінцевого, детермінованого і ефективного методу вирішення задачі, який можна реалізувати у вигляді комп'ютерної програми. Алгоритм можна визначити, описавши процедуру для вирішення задачі на звичайній мові, у вигляді схеми або комп'ютерної програми.

Більшість корисних алгоритмів вимагають організації даних, які використовуються в обчисленнях. Така організація називається **структурами даних**, і ці структури також є основними об'єктами вивчення в обчислювальній техніці. Алгоритми і структури даних працюють в тісній взаємодії. Існує думка, що структури даних існують як побічні або кінцеві продукти алгоритмів, і їх вивчення необхідне для розуміння алгоритмів. Прості алгоритми можуть вимагати складних структур даних, і навпаки, складні алгоритми можуть використовувати прості структури даних.

Коли комп'ютер використовується для вирішення задачі, то звичайно існує декілька можливих підходів. Для невеликих задач майже не має різниці, який з підходів використовувати, якщо вони всі правильно вирішують задачу. Однак у

випадку великих задач швидко усвідомлюється необхідність методів, які ефективно використовують процесорний час і пам’ять. Основною причиною вивчення алгоритмів є те, що ця дисципліна в принципі дозволяє значно економити ресурси — аж до того, що стає можливим виконувати задачі, які інакше були б недоступні. Ретельна розробка алгоритму — неймовірно ефективна частина процесу вирішення великої задачі в будь-якій прикладній області.

При розробці великої або складної комп’ютерної програми необхідно витратити багато зусиль на розуміння і чітке визначення вирішуваної задачі, керування її складністю і розкладання на менші підзавдання, які неважко реалізувати. Досить часто після такого розкладання необхідні алгоритми реалізуються елементарно. Однак в більшості випадків існує декілька алгоритмів, і вибір одного з них дуже важливий, так як значна частина системних ресурсів буде витрачена на виконання цих алгоритмів.

Використання чужих розробок в комп’ютерних системах постійно розширюється, тому слід очікувати не тільки того, що прийдеться використовувати уже розроблені алгоритми, а й того, що реалізовувати доведеться тільки невелику їх частину. Наприклад, бібліотеки Java містять реалізації величезної кількості фундаментальних алгоритмів. Однак реалізація простих варіантів базових алгоритмів допомагає краще зрозуміти їх і тому ефективніше використовувати, а також осмислено налаштовувати складні версії із бібліотек. І, що важливіше, іноді виникає необхідність в самостійній реалізації базових алгоритмів. Звичайно така необхідність виникає, якщо доводиться починати роботу в нових обчислювальних середовищах (як апаратних, так і програмних) з новими можливостями, які не враховуються в старих реалізаціях.

Лабораторна робота № 1. Знайомство з мовою програмування Java та інтегрованим середовищем розробки NetBeans

Мета роботи: вивчити базові структури мови програмування Java та інтерфейс середовища NetBeans.

КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Основи мови програмування Java

Концепція мови Java була створена Джеймсом Гослінгом (James Gosling), який розпочав роботу над цим проектом в 1991 р. Мова створювалася на основі наступних 5 принципів проектування:

1. Проста, об'єктно-орієнтована і звична. Java містить в собі невелике ядро узгоджених основних понять, які швидко засвоюються. Спочатку мова моделювалася на основі популярної в той час мови C++, тому програмісти могли без зусиль перейти на Java. Крім цього, мова наслідувала парадигмі об'єктної орієнтованості: системи складаються з інкапсульованих об'єктів, які взаємодіють шляхом передачі один одному повідомлень.

2. Надійна і безпечна. У мову включені перевірки на етапі компіляції та виконання, що забезпечує швидке виявлення помилок. Крім цього до мови включені функції безпечного доступу до мережі і файлів, щоб робота розподілених додатків не могла бути порушена в результаті вторгнення або пошкодження.

3. Нейтральна до архітектури і кросплатформенна. Однією з основних переваг Java є її переносимість. Додатки без зусиль переносяться з однієї платформи на іншу з мінімальними змінами або взагалі без змін. Девіз "Написано один раз — працює скрізь!", що супроводжував випуск Java 1.0 в 1995р., відноситься до міжплатформенних переваг мови.

4. Високопродуктивна. Додатки виконуються швидко і ефективно завдяки різним функціям нижнього рівня, таким як можливість роботи інтерпретатора Java незалежно від середовища виконання і використання програми автоматичного чищення пам'яті для звільнення невживаної пам'яті.

5. Така, що інтерпретується, забезпечує створення потоків, і динамічна. При використанні Java вихідний код, написаний розробником, компілюється в проміжну форму, що інтерпретується, відому під назвою байт-код. Поняття "набір команд байт-коду" відноситься до машинної мови, яка використовується віртуальною машиною Java (JVM — Java Virtual Machine). При наявності відповідного інтерпретатора цю мову можна транслювати в машинний код платформи, на якій він виконується. Можливість створення декількох потоків підтримується, головним чином, засобами класу `Thread`, що забезпечує одночасну роботу кількох завдань. Мова і система етапу виконання є динамічними в тому сенсі, що додатки під час виконання можуть пристосовуватися до змін робочого середовища.

Платформа Java — програмна платформа, яка складається з таких двох частин:

• Віртуальна машина Java (JVM). JVM являє собою підсистему, що виконує команди, які генеруються компілятором Java. JVM можна інтерпретувати як екземпляр середовища виконання Java (JRE), вбудований в різні програмні продукти, такі як веб-браузери, сервери і операційні системи.

• Інтерфейс прикладного програмування (API) Java. Це попередньо підготовлений код, організований у вигляді пакетів коду для подібних завдань. Наприклад, в пакети Applet і AWT входять класи для створення шрифтів, меню і кнопок.

Пакет для розробки додатків Java, або JDK (Java Development Kit), відноситься до Java SE Edition, в той час як для інших пакетів використовується позначення "SDK", яке є скороченням загального терміна "software development kit" (пакет програм для розробки додатків). Наприклад, Java EE SDK.

Змінні

У Java існують такі типи даних:

- примітивні типи даних,
- посилальні типи даних.

У мові Java всі змінні повинні бути оголошені перед тим, як вони будуть використовуватися.

Оголошення змінних в Java:

```
int x = 1;
int y = 2;
```

При оголошенні змінної в наступній послідовності зазначаються:

• тип даних (в даному прикладі `int` — змінна містить ціле число),

• ім'я змінної (в даному прикладі імена — `x` і `y`),

• початкове значення змінної або, іншими словами, ініціалізація змінної.

У даному прикладі змінним `x` і `y` привласнені значення `1` і `2`. Однак це не є обов'язковою умовою при оголошенні змінної.

Якщо потрібно оголосити кілька змінних одного типу, то це можна зробити одним рядком, вказавши імена змінних через кому.

Правила іменування змінних в Java:

• Ім'я змінної має починатися з літери (маленької) і складатися з букв (Unicode), цифр і символу підкреслення «_». Технічно можливо почати ім'я змінної з «$» або «_», проте це заборонено угодою з оформлення коду в Java (Java Code Conventions). Крім того, символ долара "$", за угодою, ніколи не використовується взагалі. Відповідно до угоди ім'я змінної повинно починатися саме з маленької літери (з великої літери починаються імена класів). Пробіли при іменуванні змінних не допускаються.

• Ім'я змінної не повинно бути ключовим або зарезервованим словом мови Java.

• Ім'я змінної чутливе до регістру: `newVariable` і `newvariable` — різні імена.

• При виборі імені змінних, слід використовувати повні слова замість загадкових абревіатур. Це зробить ваш код більш зручним для читання і розуміння. У багатьох випадках це також зробить ваш код самодокументованим.

• Якщо вибране вами ім'я змінної складається тільки з одного слова — запишіть його маленькими буквами. Якщо воно складається з більш ніж одного слова, то відокремлюйте кожне наступне слово в імені змінної великою літерою. Наприклад: `superCounter, myCat`.

• Якщо змінна зберігає постійне значення, то кожне слово слід писати великими літерами і відокремлювати за допомогою символу підкреслення. Приклад: `static final int NUMBER_OF_HOURS_IN_A_DAY = 24`.

У Java існує 8 примітивних типів даних:

• `byte` (цілі числа, 1 байт);

• `short` (цілі числа, 2 байти);

• `int` (цілі числа, 4 байти);

• `long` (цілі числа, 8 байтів);

• `float` (дійсні числа, 4 байти);

• `double` (дійсні числа, 8 байтів);

• `char` (символ Unicode, 2 байти);

• `boolean` (значення істина / неправда, 1 байт).

Ці 8 типів служать основою для всіх інших типів даних. Примітивні типи мають явний діапазон допустимих значень:

`byte` — діапазон допустимих значень від -128 до 127;

`short` — діапазон допустимих значень від -32768 до 327678;

`int` — діапазон допустимих значень від -2147483648 до 2147483647.

Тип `int` використовується частіше при роботі з цілочисельними даними, ніж `byte` і `short`, навіть якщо їх діапазону вистачає. Це відбувається тому, що при вказуванні значень типу `byte` і `short` у виразах, їх тип все одно автоматично підвищується до `int` при обчисленні.

`long` — діапазон допустимих значень від -9223372036854775808 до +9223372036854775807. Тип зручний для роботи з великими цілими числами.

`float` — діапазон допустимих значень від ~ $1{,}4*10^{-45}$ до ~ $3{,}4*10^{38}$.

Зручний для використання, коли не потрібно особливої точності в дробовій частині числа.

`double` — діапазон допустимих значень від ~ $4{,}9*10^{-324}$ до ~ $1{,}8*10^{308}$. Математичні функції такі як `sin(), cos(), sqrt()` повертають значення `double`.

`char` — символьний тип даних являє собою один 16-бітний Unicode символ.

Він має мінімальне значення `'\u0000'` (або 0), і максимальне значення `'\uffff'` (або 65535 включно). Символи `char` можна задавати також за допомогою відповідних чисел.

`boolean` — призначений для зберігання логічних значень. Змінні цього типу можуть приймати тільки одне з двох можливих значень — `true` або `false`.

У посилальні типи входять всі класи, інтерфейси, масиви.

Також існують класи-оболонки: `Byte, Short, Integer, Long, Float, Double, Character, Boolean, String`.

На відміну від примітивних типів вони пишуться з великої літери. Ці типи відповідають примітивним типам, однак є посилальними. Їх класи містять методи для перетворення типів, а також інші методи і константи, корисні при роботі з примітивними типами даних.

В якості типу також виступає будь-який створений нами клас при створенні екземпляру класу.

Оператори розгалуження і циклу

Операторів розгалуження в Java два — `if` і `switch`. Перший дозволяє піти програмою по одному з декількох напрямків, другий дозволяє зробити вибір між великим числом варіантів (два, три, чотири, ...).

Конструкція **`if-else`** дозволяє залежно від деякого логічного значення виконувати різні частини програмного коду. В узагальненій формі цей оператор записується таким чином:

```
if (логічний вираз) оператор 1; [Else оператор2;]
```

Оператор `switch` забезпечує ясний спосіб перемикання між різними частинами програмного коду в залежності від значення однієї змінної або виразу. Загальна форма цього оператора така:

```
switch (вираз)
{
case значення1: break;
case значення2: break;
case значенняN: break;
```

```
default:
}
```

Оператор `switch` працює таким чином. Обчислене значення виразу порівнюється з усіма значеннями, зазначеними в операторах `case`. Якщо при цьому знаходиться оператор `case` із значенням, яке збігається зі значенням виразу, управління передається коду, що стоїть за ним (після двокрапки). Якщо ж значенню виразу не відповідає жодному з операторів `case`, управління передається коду, розташованому після ключового слова `default`. Відзначимо, що оператор `default` необов'язковий. У випадку, коли жоден з операторів `case` не відповідає значенню виразу і в `switch` відсутній оператор `default`, виконання програми продовжується з оператора, наступного за оператором `switch`.

Циклів в Java три види — `while`, `do-while` і `for`. Перші два слід використовувати тоді, коли точно невідомо, скільки разів цикл повинен виконуватись. Цикл `for` використовуємо тоді, коли число, скільки разів наш код повинен повторюватися, відомо.

Конструкція **`for`** має наступний вигляд:

```
for (ініціалізація; умова; ітерація) {
// Тіло циклу
}
```

Ініціалізація — перший параметр, який містить змінну і її початкове значення. За допомогою цієї змінної буде обчислюватися кількість повторень циклу.

Умова — другий параметр, містить деякий логічний вираз — умова, при якому буде виконуватися цикл.

Ітерація — третій параметр, вираз, що змінює змінну (задану в ініціалізації) після кожного кроку циклу. Зміна відбувається тільки у разі виконання умови.

Конструкція **`while`** має наступний вигляд:

```
while (логічний_вираз) {
// Тіло циклу
}
```

Конструкція `while` виконує вираз у фігурних дужках до тих пір, поки логічний вираз має істинне значення (`true`). Цей параметр є умовою виконання циклу.

Істинність логічного виразу (умови) перевіряється перед кожним кроком циклу, в тому числі і перед першим. `While` так само називають оператором з передумовою.

На відміну від оператора `while, do-while` є оператором постумови, який спочатку виконує тіло циклу, а потім здійснює перевірку умови. Таким чином, тіло циклу виконається завжди хоча б один раз.

Конструкція **`do - while`** має наступний вигляд:

```
do {
// Тіло циклу
} while (логічний_вираз);
```

Як видно з конструкції, тіло циклу виконується перед перевіркою умови. Згодом тіло циклу повториться у випадку, якщо логічний вираз був істинним. Оператор `do-while` зручно застосовувати, якщо деякі дії необхідно виконати, принаймні, один раз, а їх повторення буде залежати від умови.

Класи і об'єкти

Java, як і будь-яка інша об'єктно-орієнтована мова програмування, має засоби побудови класів і об'єктів. Кожен об'єкт в Java має *тип;* ним є той клас, до якого належить даний об'єкт. У кожному класі є члени двох видів — поля і методи:

- Полями називаються змінні, що містять дані класу і його об'єктів. У них зберігаються результати обчислень, виконуваних методами даного класу.
- Методи містять виконуваний код класу.

Методи складаються з операторів. Ці оператори, а також спосіб виклику методів, визначають процес виконання програми.

Так може виглядати оголошення простого класу, що представляє точку на площині:

```
class Point {
public double x, y;
}
```

Клас `Point` містить два поля з координатами x і y точки, і в ньому немає

жодного методу (звичайно ж, в поточній реалізації). Подібне оголошення класу визначає, як будуть виглядати об'єкти, створені на його основі, а також задає поведінку об'єктів за допомогою ряду інструкцій. Креслення набуває найбільшу цінність після того, як до нього додаються технічні завдання та інструкції.

Члени класу можуть мати різні права доступу. Оголошення полів x і y класу `Point` з ключовим словом `public` означає, що будь-який метод програми, що отримав доступ до об'єкта `Point`, зможе прочитати або змінити ці поля. Дозволяється обмежити доступ до даних і надавати його лише методам самого класу або пов'язаних з ним класів.

Об'єкти створюються за допомогою виразів, в яких використовується ключове слово `new`. Створені на основі визначення класу об'єкти часто називають *екземплярами* даного класу.

У мові Java створювані об'єкти розміщуються в області системної пам'яті, яка називається *купою (heap)*. Доступ до будь-якого об'єкта здійснюється за допомогою *посилання на об'єкт* — замість самого об'єкта в змінних міститься лише посилання на нього. Коли посилання не відноситься ні до якого об'єкту, воно дорівнює `null`.

Зазвичай між самим об'єктом і посиланням на нього не робиться особливих відмінностей і можна сказати — передати методу об'єкт, насправді маючи на увазі — передати методу посилання на об'єкт.

Повертаючись до наведеного вище класу `Point`, давайте припустимо, що ми розробляємо графічний додаток, в якому необхідно стежити за безліччю точок. Кожна точка представляється окремим об'єктом `Point`. Ось як може виглядати створення і ініціалізація об'єктів `Point`:

```
Point lowerLeft = new Point();
Point upperRight = new Point();
Point middlePoint = new Point();
lowerLeft.x = 0.0;
lowerLeft.y = 0.0;
upperRight.x = 1280.0;
upperRight.y = 1024.0;
```

```
middlePoint.x = 640.0;
middlePoint.y = 512.0;
```

Кожен об’єкт класу `Point` має власну копію полів `x` і `y`. Наприклад, зміна поля x об’єкта `lowerLeft` ніяк не впливає на значення x об’єкта `upperRight`. Поля об’єктів іноді називають *змінними екземпляра (instance variables),* оскільки в кожному об’єкті (примірнику) класу міститься окрема копія цих полів.

Методи і параметри

У мові Java операції над класом здійснюються за допомогою методів класу — інструкцій, які виконуються над даними об’єкта, щоб отримати потрібний результат. У методах часто використовуються такі деталі реалізації класу, які повинні бути приховані від всіх інших об’єктів. Дані ховаються в методах і стають недоступними для всіх інших об’єктів — в цьому полягає основний сенс інкапсуляції даних.

Кожен метод має нуль або більше параметрів. Метод може повертати значення або оголошуватися з ключовим словом `void`, яке означає, що метод нічого не повертає. Оператори методу містяться в блоці між фігурними дужками `{ i }`, які слідують за іменем методу і оголошенням його *сигнатури.* Сигнатурою називається ім’я методу, супроводжуване числом і типом його параметрів. Можна удосконалити клас `Point` і додати до нього простий метод `clear`, який виглядає так:

```
public void clear() {
    x = 0; y = 0;
}
```

Метод `clear` не має параметрів, оскільки в дужках `( i )` після його імені нічого немає; крім того, цей метод оголошується з ключовим словом `void`, оскільки він не повертає ніякого значення. Усередині методу дозволяється пряме іменування полів і методів класу — можна просто написати x і y, без посилання на конкретний об’єкт.

Середовище розробки NetBeans

IDE NetBeans дозволяє швидко і легко розробляти настільні, мобільні та веб-додатки Java, а також додатки HTML5 з використанням технологій HTML, JavaScript і CSS. IDE також надає багатофункціональні набори засобів для розробників PHP і C/C++. Це безкоштовне ПЗ з відкритим вихідним кодом, яке має велике співтовариство користувачів і розробників по всьому світу.

NetBeans IDE — безкоштовне інтегроване середовище розробки для всіх платформ Java:

- ✓ Java ME — комплект Java Micro Edition (мікровидання), призначений для програмування "тонких апаратних клієнтів". Тобто пристроїв, що мають малі ресурси — надолонних комп'ютерів, стільникових телефонів, мікроконтролерів, смарткарт. Стара назва J2ME.
- ✓ Java SE — комплект Java Standard Edition (стандартне видання), призначений для програмування "товстих клієнтів". Тобто пристроїв, що володіють досить великими ресурсами — звичайних комп'ютерів. Стара назва J2SE.
- ✓ Java EE — комплект Java Enterprise Edition, призначений для написання серверного програмного забезпечення. Стара назва J2EE.

Основними перевагами NetBeans є:

1. Найкраща підтримка новітніх технологій Java

У порівнянні з іншими середовищами IDE середовище IDE NetBeans забезпечує висококласну комплексну підтримку новітніх технологій Java і останніх удосконалень стандартів Java. Це перше безкоштовне середовище IDE, яке підтримує JDK 8, JDK 7, Java EE 7, включаючи відповідні удосконалення HTML5 і JavaFX 8.

Завдяки редактору Java, який постійно вдосконалюється, різноманітним можливостям і широкому спектру інструментів, шаблонів і зразків, IDE NetBeans встановлює стандарт розробки за допомогою новітніх готових технологій.

2. Швидке інтелектуальне редагування коду

IDE — це набагато більше, ніж просто текстовий редактор. У редакторі NetBeans обробляються відступи ліній, зіставляються слова і дужки, а також виді-

ляється вихідний код як синтаксично, так і семантично. Він також надає шаблони коду, поради з написання коду та інструменти реорганізації.

3. Легке і ефективне управління проектами

Отримати чітке уявлення про великі додатки з тисячами файлів і папок і мільйонами рядків коду — це непросте завдання. IDE NetBeans пропонує різні представлення даних, від декількох вікон проектів до корисних засобів настройки та ефективного управління додатками, що забезпечує швидкий і зручний перехід за рівнями ієрархії даних, а також надає засоби управління версіями за допомогою готової інтеграції Subversion, Mercurial і Git.

При приєднанні до проекту нових розробників вони зможуть зрозуміти структуру програми завдяки чіткій організації коду.

4. Швидка розробка користувальницького інтерфейсу

З допомогою редакторів і функцій перетягування в середовищі IDE можна швидко і ефективно розробляти графічні інтерфейси для додатків Java SE, HTML5, JavaEE, PHP, C/C++ і Java ME. NetBeans GUI Builder автоматично обробляє правильні інтервали між рядками і вирівнювання для додатків Java SE, підтримуючи редагування усередині рядка. Конструктор графічного інтерфейсу настільки інтуїтивний і простий у використанні, що його застосовували як прототип графічних інтерфейсів під час демонстрацій для клієнтів.

5. Створення коду без помилок

NetBeans надає засоби статичного аналізу, зокрема інтеграцію з широко використовуваним інструментом `FindBugs`, для виявлення та виправлення поширених проблем в коді Java. Крім того, відладчик NetBeans дозволяє розміщувати точки зупинки у вихідному коді, додавати контрольні модулі для полів, виконувати код в покроковому режимі із заходом в методи, створювати знімки і відстежувати виконання.

NetBeans Profiler надає експертну допомогу для оптимізації швидкодії програми та використання пам’яті, спрощує побудову надійних і масштабованих додатків Java SE, JavaFX і Java EE. IDE NetBeans включає візуальний відладчик для додатків Java SE, що дозволяє виконувати налагодження користувальницьких ін-

терфейсів, не звертаючись до вихідного коду.

6. Підтримка декількох мов

IDE NetBeans забезпечує чудову підтримку для розробників на C/C++ і PHP, надаючи вичерпний набір редакторів та інструментів для відповідних платформ і технологій. Крім того, в IDE є редактори та інструменти для XML, HTML, Groovy, Javadoc, JavaScript і JSP. Редактор є розширюваним, тому можна підключити підтримку для безлічі інших мов.

7. Підтримка декількох платформ

NetBeans може бути встановлена на всіх операційних системах з підтримкою Java — від Windows і Linux до Mac OS X.

8. Велика кількість модулів, наданих спільнотою

Спільнота NetBeans є великою і активною. Безліч користувачів постійно розробляють нові модулі, адже IDE NetBeans є розширюваним середовищем і має API з обширною документацією.

ПРАКТИЧНА ЧАСТИНА

Створення проекту в NetBeans

Створимо за допомогою середовища NetBeans додаток Java. Для цього запустимо інтегроване середовище розробки (IDE) NetBeans і виберемо в головному меню Файл / Створити проект. У діалозі виберемо Java / Додаток Java / Далі (рис. 1).

Створення нового проекту. Крок 1. Введіть ім’я та шлях до папки проекту. Пункт «Створити головний клас» відповідає за створення класу додатку, що запускається і містить функцію `main`. Після чого можна натискати кнопку «Готово» (рис. 2).

Крок 2. На рис. 3 показано, як виглядає редагування вихідного коду програми в середовищі NetBeans.

У лівому верхньому вікні — Projects показується дерево проектів. У нашому випадку це дерево для проекту JavaApplication1. Це вікно може бути використано для одночасного показу довільного числа проектів. За замовчуванням усі дерева

згорнуті, і потрібні вузли слід розгортати клацанням по вузлу з плюсом або подвійним клацанням по відповідному імені.

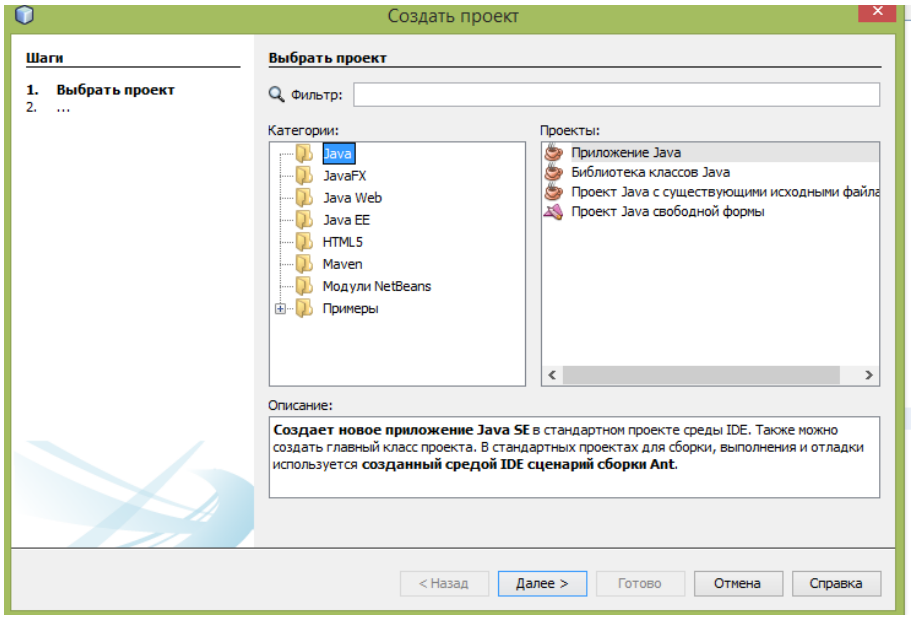


Рис. 1. Створення додатку в NetBeans

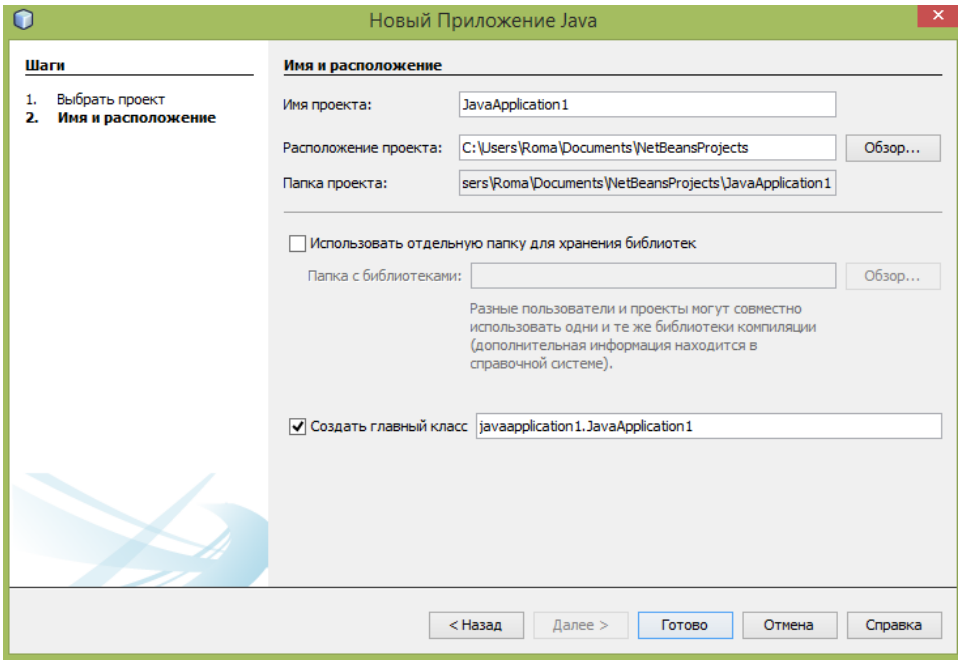


Рис. 2. Завдання імені проекту та головного класу

У правому вікні — Source показується вихідний код проекту.

У лівому нижньому вікні — Navigator показується список імен членів класу додатку — імена змінних і підпрограм. Подвійне клацання по імені призводить до

того, що у вікні редактора вихідного коду відбувається перехід на те місце, де задана відповідна змінна або підпрограма.

Розглянемо, як виглядає згенерований вихідний код нашого додатку Java (рис. 3).

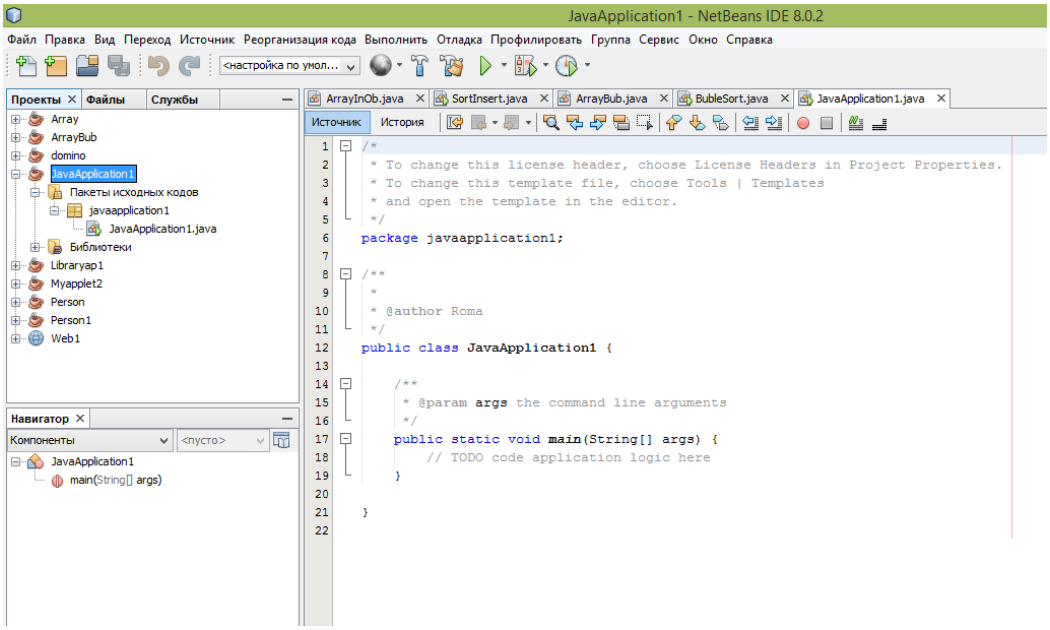


Рис. 3. Створення додатку

Спочатку йде багаторядковий коментар `/* ... */`. Потім оголошується, що наш клас буде знаходитися в пакеті javaapplication1.

Після цього йде багаторядковий коментар `/** ... */`, призначений для автоматичного створення документації по класу. У ньому присутня інструкція завдання метаданих за допомогою виразу `author` — інформація про автора проекту для утиліти створення документації javadoc. Метадані — це якась інформація, яка не відноситься до роботи програми і не включається до неї при компіляції, але супроводжує програму і може бути використана іншими програмами для перевірки прав на доступ до неї або її поширення, перевірки сумісності з іншими програмами, вказівки параметрів для запуску класу і т.п. В даному місці вихідного коду ім'я користувача береться середовищем розробки з операційної системи. Далі слідує оголошення класу `JavaApplication1`, який є головним класом програми. У ньому можуть бути оголошені дві загальнодоступні (public) підпрограми. Першою з них є конструктор:

```
public JavaApplication1() {}
```

Його ім’я збігається з іменем класу. Він займається створенням об’єктів типу `JavaApplication1`. Зазвичай такий конструктор викликається з методу main, і з його допомогою створюється всього один об’єкт. Але, взагалі кажучи, таких об’єктів у простих програмах може і не створюватися, як це і відбувається в нашому випадку. Всі класи і об’єкти додатку викликаються і управляються з методу main, який оголошений далі і виглядає наступним чином:

```
public static void main (String[] args) {}
```

Він є методом класу і тому для його роботи немає необхідності в створенні об’єкта, що є екземпляром класу `JavaApplication1`. Хоча якщо цей об’єкт і створюється, це відбувається під час роботи методу `main`. Метод `main` є головним методом додатку і управляє роботою програми, що запускається. Він автоматично викликається при запуску програми. Параметром `args` цього методу є масив рядків — `String[]`. Це параметри командного рядка, які передаються в додаток при його запуску.

Після закінчення виконання методу `main` додаток завершує свою роботу.

При оголошенні будь-якого методу в Java спочатку вказується модифікатор видимості, який вказує права доступу до методу, потім інші модифікатори, після чого слідує тип значення, що повертається методом. Якщо модифікатор видимості не вказаний, вважається, що це `private` — закритий, приватний, що не дозволяє доступ до методу з інших класів.

Конструктори становлять особливий випадок — в них ім’я типу та ім’я методу збігаються. У Java вони не вважаються методами, хоча в інших мовах програмування такого тонкого розмежування не робиться.

Далі слідує ім’я методу, після чого в круглих дужках іде список параметрів (аргументів), що передаються в даний метод при його виклику. Після цього в фігурних дужках йде тіло методу, тобто його реалізація — пишеться той алгоритм, який буде виконуватися при виклику методу.

У мові Java, як і в C/C++, підпрограми завжди є підпрограмами-функціями, які повертають яке-небудь значення. Якщо треба написати підпрограму-

процедуру, в якій не треба повертати ніякого значення, в C/C++/Java користуються підпрограмами-функціями з типом значення `void`, що повертається — пустота, порожній простір. Як і відбувається у випадку нашого методу `main`.

Середовище NetBeans створює заготовки методів — у них є пусте тіло. Для здійснення методом будь-якої діяльності слід дописати свій власний код. Напишемо традиційний приклад — виведення повідомлення — Hello, world!. Для цього замість коментаря

```
// TODO code application logic here (описати тут логіку роботи програми)
```

напишемо рядок виведення тексту

```
System.out.println ("Hello, world!");
```

Клас `System` — система, що має поле `out`, вивід. Це об'єкт, призначений для підтримки виводу. У нього є метод `println`, призначений для виведення тексту в режимі консолі.

Компіляція файлів проекту і запуск програми

Для збірки проекту слід вибрати в меню середовища розробки **Виконати/Зібрати проект** (або клавіша F11, або на панелі інструментів іконка з молотком). При цьому відбувається компіляція тільки тих файлів проекту, які були змінені в процесі редагування після останньої збірки.

Пункт **Виконати/Очистити і зібрати проект** (або комбінація клавіш Shift+F11, або на панелі інструментів іконка з молотком і віником) видаляє всі вихідні файли проекту (очищає папки build і dist), після чого заново компілюються всі класи проекту.

Пункт **Виконати/Створити документацію (…)** запускає створення документації по проекту. При цьому з вихідних кодів класів проекту вибирається інформація, занесена в документаційні коментарі `/** ... */`, і на її основі створюється гіпертекстовий HTML-документ.

Пункт **Виконати / Виконати файл** "Main.java" компілює вибраний файл проекту — в нашому випадку файл Main.java, в якому зберігаються вихідні коди

класу `Main`.

Для того щоб запустити зкомпільований додаток з середовища розробки, слід вибрати в меню середовища розробки **Виконати / Запустити проект** (клавіша F6, або на панелі інструментів іконка з зеленим трикутником). Після запуску нашого проекту у вихідній консолі, яка знаходиться в нижній частині вікна проекту, з'явиться службова інформація про хід компіляції і запуску.

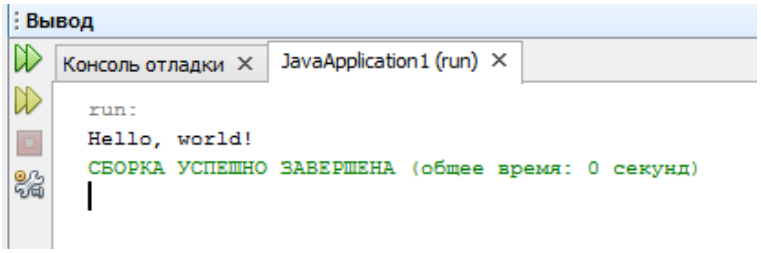


Рис. 4. Інформація про хід компіляції і запуску у вихідній консолі

До неї ж здійснюється виведення методів `System.out.print` і `System.out.println`. Метод `System.out.print` відрізняється від методу `System.out.println` тільки тим, що в `println` після виведення здійснюється автоматичний перехід на новий рядок, а в `print` триває вивід в той же рядок консолі.

ПРАКТИЧНЕ ЗАВДАННЯ

1. Створити новий проект NetBeans «MyFirstApplication». У методі `main` реалізувати множення і ділення двох чисел з дійсним типом даних. Числа вводяться з клавіатури, для цього використовуйте приклад наступного коду:

```
public class MyFirstApplication {
public static String getString() throws IOException
{
InputStreamReader isr = new InputStreamReader (System.in);
BufferedReader br = new BufferedReader (isr);
String s = br.readLine();
return s;
}
```

```
public static float getfloat() throws IOException
{
String s = getString();
return Float.parseFloat(s);
}
public static void main(String[] args) throws IOException
{
float a1 = getfloat(); // Читання дійсного числа з клавіатури
}
```

Більш докладно розберіться з представленим кодом самостійно.

2. У методі `main` за допомогою оператора розгалуження `if` порівняти два числа і показати результат порівняння.

3. За допомогою оператора циклу `for` збільшити суму і різницю двох чисел в 10 і 5 разів відповідно. Використовуйте числа цілочисельного типу, для цього додайте в програму відповідний метод `getint()`.

4. Створити окремі методи ділення, множення, суми і різниці для дійсних чисел. Перевірити вірність роботи методів.

5. Створити найпростіший калькулятор з використанням створених методів. Для створення меню калькулятора використовуйте оператор `switch`. Перевірити на правильність і протестувати створений калькулятор.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке Java?
2. Вкажіть основні принципи Java.
3. Які існують типи даних Java?
4. Які існують оператори розгалуження і циклу?
5. З чого складається клас?
6. Що таке поля і методи класу?
7. Області видимості методів і полів з модифікатором `private`.
8. Області видимості методів і полів з модифікатором `public`.
9. Області видимості методів і полів з модифікатором `protected`.

10. Що таке об’єкт класу і де розміщується об’єкт після його створення?
11. Що таке NetBeans, Java EE, SE, ME?
12. Що таке компілятор?

Лабораторна робота № 2. Знайомство з масивами і рядковими об’єктами

Мета роботи: знайомство з найпоширенішою структурою даних — масивом. Вивчення ролі масивів в об’єктно-орієнтованому програмуванні та набуття навичок роботи з ними.

КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Найпоширеніша структура даних — масив, яка підтримується в більшості мов програмування. Внаслідок своєї популярності масив є гарною відправною точкою для знайомства зі структурами даних і вивчення їх ролі в об’єктно-орієнтованому програмуванні.

Масив — це структура даних, у якій зберігаються величини однакового типу.

Розмір або довжина масиву — це загальна кількість елементів в масиві. Розмір масиву задається при створенні масиву і не може бути змінений надалі. Не можна прибрати елементи з масиву або додати їх туди, але можна існуючим елементам привласнити нові значення.

Індекс початкового елемента — 0, наступного за ним — 1 і т. д. Індекс останнього елемента в масиві на одиницю менше, ніж розмір масиву.

У Java масиви є об’єктами. Це означає, що ім’я, яке дається кожному масиву, лише вказує на адресу якогось фрагмента даних у пам’яті. Крім адреси в цій змінній нічого не зберігається. Індекс масиву, фактично, вказує на те, наскільки треба відступити від початкового елемента масиву в пам’яті, щоб дістатися до потрібного елемента.

Щоб створити масив, треба оголосити для нього підходяще ім’я, а потім з цим іменем зв’язати потрібний фрагмент пам’яті, де і будуть один за одним збері-

гатися значення елементів масиву.

2.1 Створення масиву

Створення масивів вимагає використання оператора `new`, наприклад:

```
int intArray[]; // Оголошення посилальної змінної типу масив
intArray = new int[3]; // Створення масиву цілого типу
// Створення двовимірного масиву дійсного типу
float[][] flArray = new float[2][3];
// Створення масиву і ініціалізація
double[] a = {0.01, -3.4, 2.89, 4.5, -6.7};
// Створення і ініціалізація двовимірного масиву цілих змінних
int[][] arrayOfInts = {{32, 87, 3, 589}, {12, 1076, 2000, 8}, {622,
127, 77, 955}};
```

Оператор `[]` повідомляє компілятору, що ім'я належить об'єкту масиву, а не звичайній змінній. Оскільки масив є об'єктом, його ім'я (`intArray` в попередньому коді) містить *посилання* на масив. Сам масив зберігається десь в пам'яті, а `intArray` містить тільки адресу блоку даних.

Масиви Java мають три важливі переваги перед масивами інших мов:

- програмісту не обов'язково вказувати розмір масиву при його оголошенні;
- будь-який масив в мові Java є змінною — це означає, що його можна передати як параметр методу і використовувати в якості значення, що повертається методом;
- розмір масиву в будь-який момент часу можна дізнатися через спеціальну змінну `length`, наприклад:

```
int len = intArray.length;
```

2.2 Звернення до елементів масиву

При зверненні до елементу масиву необхідно вказати його індекс у квадратних дужках. Цей синтаксис також використовується в багатьох інших мовах програмування:

```
temp = intArray[3]; // Отримання вмісту четвертого елемента масиву
intArray[7] = 66; // Вставка значення 66 у восьмий осередок
```

2.3 Рядкові об’єкти

Для роботи з послідовностями символів в Java передбачені тип об’єктів `String` і мовна підтримка при їх ініціалізації. Клас `String` надає різноманітні методи для роботи з об’єктами `String`.

Приклад літералів типу `String` — зокрема, коли в програмі з’являється оператор такого вигляду:

```
System.out.println ("Hello, world");
```

Компілятор Java насправді створює об’єкт `String`, присвоює йому значення зазначеного літералу і передає його в якості параметра методу `println`.

Об’єкти типу `String` відрізняються від масивів тим, що при їх створенні не потрібно вказувати розмір. Створення нового об’єкта `String` і його ініціалізація виконуються всього одним оператором, як показує наступний приклад:

```
class StringDemo {
static public void main (String args[]) {
String myName = " Petronius";
myName = myName + " Arbiter";
System.out.println ("Name = " + myName);
} }
```

Ми створюємо об’єкт `String` з іменем `myName` і ініціалізуємо його рядковим літералом. Вираз з оператором конкатенації +, який слідує за ініціалізацією, створює новий об’єкт `String` з новим значенням. Нарешті, значення `myName` виводиться в стандартний вихідний потік. Результат роботи наведеної вище програми буде таким:

```
Name = Petronius Arbiter
```

Крім знака + в якості оператора конкатенації можна використовувати оператор += як скорочену форму, в якій назва змінної розміщується в лівій частині оператора.

Об’єкти `String` містять метод `length`, який повертає кількість символів в рядку. Символи мають індекси від `0` до `length()-1`.

Об’єкти `String` є незмінними, тобто доступними тільки для читання. Вміст об’єкта `String` ніколи не змінюється. Коли в програмі зустрічаються оператори

такого вигляду:

```
str = "redwood";
// ... Зробити що - небудь зі str ...
str = "oak";
```

то другий оператор присвоювання задає нове значення *посилання на об'єкт,* а не *вміст* рядка. При кожному виконанні операції, яка на перший погляд змінює вміст об'єкта (наприклад, +=), насправді виникає новий об'єкт `String`, також доступний тільки для читання, — тоді як вміст вихідного об'єкта `String` залишається незмінним. Змінна буде вказувати на новий об'єкт `String`, а старий об'єкт `String` буде покинуто і він стане "сміттям". Клас `StringBuilder` дозволяє створювати рядки зі змінним вмістом і уникнути утворення такого "сміття".

Найпростіший спосіб порівняти два об'єкти `String` і з'ясувати, чи збігається їх вміст, полягає у використанні методу `equals`:

```
if (oneStr.equals (twoStr))
     ...
```

Приклад програми з використанням масиву

Спочатку ми розглянемо традиційну процедурну версію, а потім перейдемо до об'єктно-орієнтованого аналогу. Для цього створимо проект в NetBeans під назвою «ArrayProject» з класом `Array`.

```
class Array
{
public static void main(String[] args)
{
long[] arr; // Посилання на масив
arr = new long[100]; // Створення масиву
int nElems = 0; // Кількість елементів
int j; // Лічильник циклу
long searchKey; // Ключі шуканого елемента
arr[0] = 77; // Вставка 5 елементів
arr[1] = 99;
arr[2] = 44;
```

```
arr[3] = 55;
arr[4] = 22;
nElems = 5; // Масив містить 5 елементів
for (j = 0; j <nElems; j ++) // Вивід елементів
System.out.print(arr[j] + " ");
System.out.println("");
searchKey = 22; // Пошук елемента з ключем 22
for(j = 0; j<nElems; j++) // Для кожного елемента
if(arr[j] == searchKey) // Значення знайдено?
break; // Так - вихід з циклу
if(j == nElems) // Досягнуто останній елемент?
System.out.println("Can not find" + searchKey); // Так
else
System.out.println("Found " + searchKey); // Ні
searchKey = 55; // Видалення елемента з ключем 55
for(j = 0; j<nElems; j++) // Пошук видаляємого елемента
if(arr[j] == searchKey)
break;
for(int k = j; k<nElems-1; k++) // Переміщення наступних елементів
arr[k] = arr[k + 1];
nElems-- ; // Зменшення розміру
for(j = 0; j <nElems; j ++) // Вивід елементів
System.out.print(arr[j] + " ");
System.out.println("");
} } // Кінець класу Array
```

Програма створює масив з іменем «arr», поміщає до нього 5 елементів даних, виводить всі елементи, шукає елемент з ключем 22, видаляє елемент з ключем 55 і виводить 4 елементи, що залишилися. Результат роботи програми виглядає так:

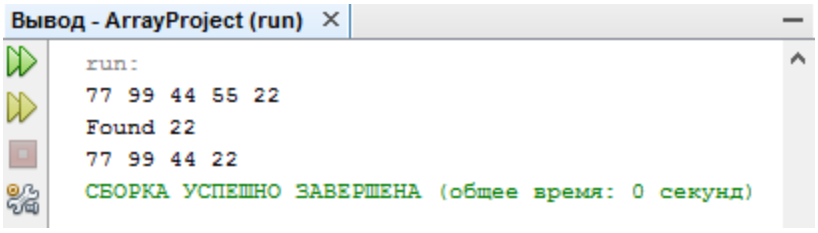


Рис. 5. Результат роботи класу Array

У масиві зберігаються дані типу `long`. Цей тип обраний для того, щоб більш наочно показати, що це дані. Тип `int` використовується для значень індексів. Примітивний тип обраний для спрощення коду. У загальному випадку дані, що зберігаються в структурі даних, складаються з декількох полів, тому для їх подання зазвичай використовуються об'єкти замість примітивних типів.

Розподіл програми на класи

Перехід на об'єктно-орієнтований стиль відбуватиметься у два етапи.

1. На першому етапі ми відокремимо структуру даних (масив) від решти коду програми. Частина програми стає користувачем цієї структури.

2. На другому етапі буде вдосконалено взаємодію між структурою зберігання даних і її користувачем.

Розподіл програми на класи надає численні переваги. Які класи? Власне структура даних і інша частина програми, що використовує цю структуру даних. Поділ програми на два класи прояснює її функціональність, спрощує її проектування і розуміння (а також модифікацію і супровід коду в реальних проектах).

У класі `Array` масив використовувався як структури даних, але він розглядався просто як елемент мови. У новій версії масив буде інкапсульований в класі з іменем `LowArray`. Клас надаватиме методи, за допомогою яких об'єкти інших класів (`LowArrayApp` в даному випадку) зможуть звертатися до масиву. Ці методи забезпечують взаємодію між `LowArray` і `LowArrayApp`.

```
class LowArray
{
private long[] a; // Посилання на масив a
public LowArray(int size) // Конструктор
{ a = new long[size]; } // Створення масиву
public void setElem(int index, long value) // Запис елемента
{ a[index] = value; }
public long getElem(int index) // Читання елемента
{ return a[index]; }
} // Кінець класу LowArray
```

```
class LowArrayApp
{
public static void main(String[] args)
{
LowArray arr; // Посилання
arr = new LowArray(100); // Створення об’єкта LowArray
int nElems = 0; // Кількість елементів у масиві
int j; // Змінна циклу
arr.setElem(0, 77); // Вставка 5 елементів
arr.setElem(1, 99);
arr.setElem(2, 44);
arr.setElem(3, 55);
arr.setElem(4, 22);
nElems = 5; // Масив містить 5 елементів
for(j = 0; j<nElems; j++) // Вивід елементів
System.out.print(arr.getElem(j) + " ");
System.out.println("");
int searchKey = 26; // Пошук елемента
for(j = 0; j <nElems; j++) // Для кожного елемента
if(arr.getElem(j) == searchKey) // Значення знайдено?
break;
if(j == nElems) // Hi
System.out.println("Can not find " + searchKey);
else // Так
System.out.println("Found " + searchKey);
// Видалення елемента з ключем 55
for(j = 0; j <nElems; j++) // Пошук елемента, що видаляється
if(arr.getElem(j) == 55)
break;
for(int k = j; k<nElems; k++) // Переміщення наступних елементів
arr.setElem(k, arr.getElem(k + 1));
nElems--; // Зменшення розміру
for(j = 0; j<nElems; j++) // Вивід вмісту
System.out.print(arr.getElem(j) + " ");
System.out.println("");
```

```
} } // Кінець класу LowArrayApp
```

Результат виконання програми lowArray.java нагадує результати попередньої, не враховуючи того, що перед видаленням елемента з ключовим значенням 55 ми намагаємося знайти неіснуюче ключове значення 26 (рис. 6).

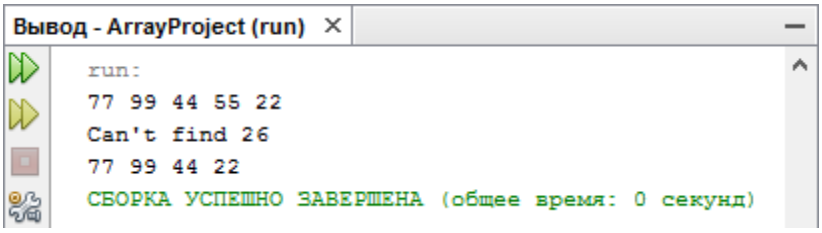


Рис. 6. Результат роботи класів `LowArray` і `LowArrayApp`

У програмі клас `LowArray` фактично являє собою «обгортку» для звичайного масиву Java. Масив прихований від навколишнього світу усередині класу: поле даних оголошено з ключовим словом `private`, так що воно доступне тільки для методів класу `LowArray`. Клас `LowArray` містить три методи: `setElem()` і `getElem()`, які виконують вставку і вибірку елемента відповідно, а конструктор створює порожній масив заданого розміру.

Інший клас `LowArrayApp` створює об’єкт класу `LowArray` і використовує його для зберігання і обробки даних. Вважайте, що `LowArray` — інструмент, а `LowArrayApp` — користувач інструменту.

Клас, використовуваний для зберігання об’єктів даних (як клас `LowArray`), іноді називається контейнерним класом (або просто контейнером). Як правило, контейнер не тільки зберігає дані, але й надає методи для звернення до них.

ПІДСУМКИ

Створення масиву — двоступеневий процес. По-перше, необхідно розподілити пам’ять для зберігання масиву і привласнити її змінній масиву. Таким чином, в Java всі масиви є такими, що динамічно розподіляються. Як тільки масив створений і пам’ять для нього розподілена, до конкретного елементу масиву можна звертатися, вказуючи його індекс у квадратних дужках. Індекс масиву почина-

ється з нуля.

Розглянуті раніше масиви — одномірні. Бувають ще багатовимірні. Наприклад, двовимірні масиви можна порівняти з будинком, але не з одноповерховим, а, наприклад, з п’ятиповерховим. У цьому випадку додається ще один індекс для обліку поверхів.

При оголошенні змінної багатовимірного масиву для вказівки кожного додаткового індексу використовують окремий набір квадратних дужок. Такий масив також створюється ключовим словом `new`.

ЗАВДАННЯ

В класі `LowArray` створити методи:

- для пошуку елементів в масиві по заданому значенню;
- для видалення значень в масиві по заданому значенню;
- для виведення вмісту масиву;
- для знаходження максимального і мінімального елемента в масиві;
- для підрахунку кількості парних і непарних значень;
- для заміни всіх непарних значень масиву на вказане значення.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке масив?
2. Як створюється масив (шаблон)?
3. Що повідомляє компілятору оператор `[]`?
4. Назвіть переваги масивів в Java перед іншими мовами.
5. Як дізнатися розмір масиву?
6. Що необхідно вказати при зверненні до елементу масиву?
7. Чим відрізняються об’єкти типу `String` від масивів?
8. Що роблять методи `length` і `equals`?
9. Якими є масиви в Java?
10. Що передбачає перехід на об’єктно-орієнтована мову?
11. Вкажіть переваги поділу програми на класи.

Лабораторна робота № 3. Найпростіші алгоритми сортування

Мета роботи: закріплення навичок роботи з масивами і освоєння найпростіших алгоритмів сортування.

КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Коли у вашій бази даних накопичиться скільки-небудь значний обсяг даних, неминуче виникне необхідність в їх сортуванні.

Сортування даних також може стати необхідним підготовчим кроком перед пошуком. Так як сортування відіграє важливу роль при роботі з даними і при цьому може бути досить витратною за часом операцією, на цю тему велися об'ємні дослідження в галузі інформатики. Були розроблені різні методи сортування. У цій лабораторній роботі ми розглянемо три найпростіших алгоритми:

- ✓ сортування методом вибору;
- ✓ бульбашкове сортування;
- ✓ сортування методом вставки.

3.1 Бульбашкове сортування

Алгоритм полягає в повторюваних проходах по сортованого масиву. За кожен прохід елементи послідовно порівнюються попарно і, якщо порядок в парі невірний, виконується обмін елементів. Проходи по масиву повторюються до тих пір, поки на черговому проході не виявиться, що обміни більше не потрібні і відповідно — масив відсортований. При проході алгоритму елемент, що стоїть не на своєму місці, «спливає» до потрібної позиції як бульбашка у воді, звідси і назва алгоритму.

Алгоритм сортування методом бульбашки: порівнюємо поточний і наступний елементи масиву. Якщо поточний елемент більший, ніж наступний, то міняємо їх місцями (рис. 7).

В результаті останнім елементом в масиві у нас виявиться найбільший елемент.

Рис. 7. Алгоритм бульбашкового сортування, 1-й крок

Другий крок сортування методом бульбашки — повторюємо вищевказані дії для частини масиву, починаючи з 1 позиції до N-1. В результаті передостанній елемент в масиві теж буде на своєму, "відсортованому" місці.

На наступних кроках сортування методом бульбашки вищевказані дії повторюються для частини масиву, починаючи з 1 позиції до N-2 (крок 3), а потім для діапазону 1..N-3 і так далі до діапазону 1..2.

Після завершення останнього кроку масив буде відсортований за зростанням (рис. 8).

Рис. 8. Процес сортування масиву за зростанням елементів

Середнє число порівнянь і обмінів мають квадратичний порядок росту $O(N^2)$. В О-синтаксисі використовується прописна буква О; вважається, що вона позначає «порядок». На підставі цього синтаксису можна (дуже умовно) оцінити різні складності: O(1) — відмінно, O(log N) — добре, O(N) — непогано, $O(N^2)$ — погано. Звідси можна зробити висновок, що алгоритм бульбашки досить повільний, проте він простий і його можна поліпшити простими засобами.

По-перше, розглянемо ситуацію, коли на якому-небудь з проходів не відбулося жодного обміну. Це означає, що всі пари розташовані в правильному порядку, так що масив вже відсортований. Одне з можливих поліпшень алгоритму полягає в запам'ятовуванні, чи проводився на даному проході який-небудь обмін. Якщо ні — алгоритм закінчує роботу.

Процес поліпшення можна продовжити, якщо запам'ятовувати не тільки сам факт обміну, а й індекс останнього обміну k. Дійсно: всі пари сусідніх елементів з індексами, більшими k, вже розташовані в потрібному порядку. Подальші проходи можна закінчувати на індексі k, замість того щоб рухатися до встановленої заздалегідь верхньої межі і.

Реалізація бульбашкового сортування на мові Java

Створимо проект в NetBeans «BubbleSort» з двома класами `ArrayBub` і `BubbleSortApp` (головний клас).

```
public class ArrayBub // Бульбашкове сортування
{
private long[] a; // Посилання на масив a
private int nElems; // Кількість елементів даних
public ArrayBub(int max) // Конструктор
{
a = new long[max]; // Створення масиву
nElems = 0; // Поки немає жодного елемента
}
public void insert(long value) // Вставка елемента в масив
{
a[nElems] = value; // Власне вставка
```

```
nElems++; // Збільшення розміру
}
public void display() // Вивід вмісту масиву
{
for(int j = 0; j<nElems; j++) // Для кожного елемента
System.out.print(a[j] + " "); // Вивід
System.out.println("");
}
public void bubbleSort()
{
int out, in;
for(out = nElems-1; out>1; out-- ) // Зовнішній цикл (зворотний)
for(in = 0; in <out; in++) // Внутрішній цикл (прямий)
if(a[in]> a[in + 1]) // Порядок порушений?
swap(in, in + 1); // Поміняти місцями
}
private void swap(int one, int two)
{
long temp = a[one];
a[one] = a[two];
a[two] = temp;
} } // Кінець класу ArrayBub

public class BubbleSortApp
{
public static void main(String[] args)
{
int maxSize = 100; // Розмір масиву
ArrayBub arr; // Посилання на масив
arr = new ArrayBub(maxSize); // Створення масиву
arr.insert(77); // Вставка 8 елементів
arr.insert(99);
arr.insert(44);
arr.insert(55);
arr.insert(22);
arr.insert(88);
```

```
arr.insert(11);
arr.insert(13);
arr.display(); // Вивід елементів
arr.bubbleSort(); // Бульбашкове сортування елементів
arr.display(); // Повторний вивід
} } // Кінець класу BubbleSortApp
```

В результаті сортування найменший елемент повинен знаходитися на початку масиву (індекс 0), а найбільший — в кінці (індекс nElems-1). Лічильник циклу `out` зовнішнього циклу `for` починає перебір з кінця масиву (в позиції nElems - 1) і послідовно зменшується в процесі перебору. Елементи з індексами, більшими `out`, знаходяться на своїх остаточних місцях. Змінна `out` зміщується вліво після кожного проходу, щоб алгоритм не чіпав вже відсортовані елементи. Лічильник `in` починається з початку масиву і послідовно збільшується при кожній ітерації внутрішнього циклу аж до `out`. Внутрішній цикл порівнює два осередки, на які вказують `in` і `in + 1`, і якщо елемент в комірці `in` більший ніж елемент в комірці `in + 1` — міняє їх місцями.

Для наочності перестановка елементів виконується окремим методом `swap()`. Взагалі кажучи, використовувати окремий метод `swap()` в реальній програмі небажано, тому що виклик функції пов’язаний із додатковими, нехай і невеликими витратами часу. Якщо ви пишете власну реалізацію сортування, краще виконуйте перестановку «на місці».

3.2 Сортування вибором

Алгоритм сортування методом вибору:

1. Знайти мінімальне значення в поточному списку.
2. Знайдене мінімальне значення міняється з елементом на першій позиції.

3. Повторюємо сортування, виключивши з розгляду вже відсортований перший елемент, тобто, починаючи з другої позиції.

Послідовність кроків при n = 7 показана на рис. 9.

Алгоритм не використовує додаткової пам’яті: всі операції відбуваються "на місці". Алгоритм сортування методом вибору перевершує бульбашкове сортуван-

ня за характеристиками — кількість необхідних перестановок скорочується з $O(N^2)$ до $O(N)$. На жаль, кількість порівнянь залишається рівною $O(N^2)$.

Рис. 9. Приклад сортування вибором

Реалізація сортування методом вибору на мові Java

Створимо проект в NetBeans «SelectSort» с двома класами `ArraySel` і `SelectSortApp` (головний клас). Лістинг проекту SelectSort схожий на лістинг BubbleSort, хіба що контейнерний клас називається `ArraySel` замість `ArrayBub`, а метод `bubbleSort()` замінений методом `selectSort()`. Ось як він виглядає:

```
public void selectSort()
{
int out, in, min;
for(out = 0; out<nElems-1; out++) // Зовнішній цикл
{
min = out; // Мінімум
for(in = out+1; in <nElems; in++) // Внутрішній цикл
if(a[in] <a[min]) // Якщо значення min більше,
min = in; // Значить, знайдений новий мінімум
swap(out, min); // Поміняти їх місцями
} }
```

Зовнішній цикл (змінна `out`) починає перебір з початку масиву (індекс 0) і просувається до більших значень індексів. Внутрішній цикл (змінна `in`) починається з `out` і також рухається вправо. У кожній новій позиції `in` порівнюються елементи `a[in]` і `a[min]`. Якщо `a[in]` менше, то `min` присвоюється значення `in`. Наприкінці внутрішнього циклу `min` вказує на мінімальне значення поточного проходу, і алгоритм переставляє елементи масиву, на які посилаються `out` і `min`.

3.3 Сортування методом вставки

В будь-якому алгоритмі сортування зі збільшенням розміру сортованого масиву збільшується і час сортування. Основною перевагою алгоритму сортування вставками є можливість сортувати масив у міру його отримання. Тобто, маючи частину масиву, можна починати його сортувати. У паралельному програмуванні така особливість відіграє не останню роль.

Сортований масив можна розділити на дві частини — відсортована частина і невідсортована. На початку сортування перший елемент масиву вважається відсортованим, все інші — не відсортовані. Починаючи з другого елемента масиву і закінчуючи останнім, алгоритм вставляє не відсортований елемент масиву в потрібну позицію у відсортованій частині масиву. Таким чином, за один крок сортування відсортована частина масиву збільшується на один елемент, а не відсортована частина масиву зменшується на один елемент.

Розглянемо приклад сортування за зростанням масиву з 7 чисел (табл. 1). Вхідний масив: **3 3 7 1 2 5 0.**

На кожному кроці сортування порівнюється поточний елемент з усіма елементами у відсортованій частини. У підсумку на шостому кроці ми отримуємо відсортований за зростанням масив.

Реалізація сортування методом вставки на мові Java

Створимо проект в NetBeans «InsertSort» з двома класами `ArrayIns` і `InsertSortApp` (головний клас). Лістинг проекту InsertSort схожий на лістинг SelectSort і BubbleSort, хіба що контейнерний клас називається `ArrayIns`, а метод

`selectSort()` і `swap (out, min)` замінений методом `insertionSort()`. Ось як він виглядає:

Таблиця 1

Сортування методом вставки

Крок	**Відсортована частина масиву**							**Поточний елемент**	**Вставка**
1	3							3	false
2	3	3						7	false
3	3	3	7					1	true
4	1	3	3	7				2	true
5	1	2	3	3	7			5	true
6	1	2	3	3	5	7		0	true
-	0	1	2	3	3	5	7	-	-

```
public void insertionSort()
{
int in, out;
for(out = 1; out <nElems; out++) // out - розділовий маркер
{
long temp = a[out]; // Зкопіювати позначений елемент
in = out; // Почати переміщення з out
while(in > 0 && a[in - 1] >= temp) // Ще не знайдений менший елемент
{
a[in] = a[in - 1]; // Перемістити елемент вправо
 --in; // Перейти на одну позицію вліво
}
a[in] = temp; // Вставити позначений елемент
} }
```

У зовнішньому циклі `for` лічильник починає з позиції 1 і рухається вправо. Він позначає крайній лівий не відсортований елемент. У внутрішньому циклі `while` лічильник `in` починає з позиції `out` і рухається вліво — або поки `temp` не стане менше елемента масиву, або коли подальший зсув стане неможливим. При кожному проході по циклу `while` наступний відсортований елемент зсувається на

одну позицію вправо.

ПІДСУМКИ

✓ У всіх алгоритмах сортування, описаних в цьому розділі, дані зберігаються в масиві.

✓ Сортування засновано на порівнянні ключів елементів даних в масиві і переміщенні елементів (а вірніше, посилань на них), поки вони не будуть розташовуватися в потрібному порядку.

✓ Всі алгоритми цієї роботи виконуються за час $O(N^2)$. Тим не менше, деякі з них можуть працювати помітно швидше інших.

✓ Бульбашкове сортування — найменш ефективний, але найпростіший алгоритм.

✓ З усіх видів сортування зі складністю $O(N^2)$, описаних в цій лабораторній роботі, найчастіше використовується сортування методом вставки.

✓ Жоден з описаних вище алгоритмів сортування не вимагає додаткових витрат пам'яті за межами вихідного масиву (тільки одна тимчасова змінна).

ЗАВДАННЯ

1. У проекті BubbleSort перепишіть метод `main()` так, щоб він створював великий масив і заповнював його даними. Для генерування випадкових чисел можна скористатися наступним фрагментом коду:

```
long n =(long)(java.lang.Math.random()*(maxSize - 1));
```

Також доопрацювати можливість введення розміру масиву користувачем.

2. У проекті BubbleSort змініть метод `BubbleSort()` так, щоб він сортував в порядку убування чисел.

3. Додайте в клас `ArraySel` програми SelectSort метод з іменем `moda()`, який дозволяє знайти в масиві моду. Модою називається елемент ряду, який зустрічається найчастіше. Масив попередньо відсортувати, після чого застосувати метод мода.

4. Додайте в клас `ArrayIns` програми InsertSort метод з іменем `median()`,

який повертає медіану масиву і поміняти його з останнім елементом. Медіаною називається елемент ряду, який ділить його на дві рівні частини: в одній знаходяться елементи, що не менше медіани, в іншій — не більше медіани.

5. Змініть метод `insertionSort()` в програмі InsertSort так, щоб він підраховував кількість копіювань і порівнянь в ході сортування, а потім виводив отримані результати. Для підрахунку порівнянь необхідно розбити надвоє складну умову у внутрішньому циклі `while`.

КОНТРОЛЬНІ ПИТАННЯ

1. Які дві основні операції сортування?
2. Які операції чергуються в алгоритмі бульбашкового сортування?
3. Скільки виконується порівнянь при N елементах бульбашкового сортування?
4. Яке з простих сортувань краще?
5. Наведіть алгоритми сортування методом вибору та вставки.

Лабораторна робота № 4. Робота зі стеком

Мета роботи: знайомство зі стеком, вивчення його інтерфейсу та набуття навичок роботи з ним.

КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Стеки, черги та пріоритетні черги є більш абстрактними сутностями, ніж масиви і багато інших структур даних. Вони визначаються, насамперед, своїм інтерфейсом: набором дозволених операцій, які можуть виконуватися з ними.

Стек (Stack) — структура даних з дисципліною доступу до елементів "останнім прийшов — першим вийшов" (Last-In-First-Out, LIFO).

Багато мікропроцесорів мають стекову архітектуру. При виклику методу адресу повернення і аргументи заносяться в стек, а при виході вони витягуються зі стека. Операції зі стеком вбудовані в мікропроцесор. Стекова архітектура також

використовувалась в деяких старих калькуляторах. Замість того, щоб вводити арифметичний вираз з круглими дужками, користувач зберігав проміжні результати в стеку.

Для пояснення ідеї стека найкраще скористатися аналогією. Багато людей складають листи стопкою на журнальному столику. Коли з'явиться вільна хвилина, вони обробляють пошту, що нагромадилася зверху вниз. Спочатку вони відкривають лист, що знаходиться на вершині стопки, і виконують необхідну дію — оплачують рахунок, викидають лист і т. д. Розібравшись з першим листом, вони переходять до наступного конверту, який тепер виявляється зверху стопки, і розбираються з ним. Зрештою вони добираються до нижнього листа, який тепер виявляється верхнім.

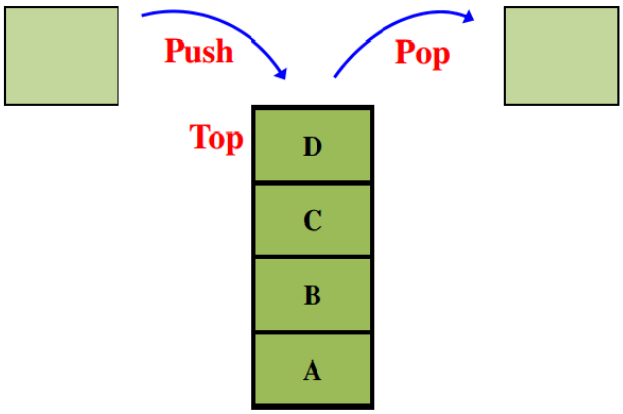


Рис. 10. Стек

Принцип «почати з верхнього листи» відмінно працює за умови, що вся пошта може бути оброблена за розумний час. В іншому випадку виникає небезпека того, що листи в нижній частині стопки не будуть проглядатися місяцями і відповідні рахунки будуть прострочені.

Основні операції зі стеком:

- ✓ вставка (занесення) елемента до стеку;
- ✓ вилучення зі стека — виконуються тільки на вершині стека, тобто з його верхнім елементом.

4.1 Реалізація стека на мові Java

Створимо проект в NetBeans «Stack» з двома класами `StackX` і `StackApp` (головний клас).

```
class StackX
{
private int maxSize; // Розмір масиву
private long[] stackArray;
private int top; // Вершина стека
public StackX(int s) // Конструктор
{
maxSize = s; // Визначення розміру стека
stackArray = new long[maxSize]; // Створення масиву
top = -1; // Поки немає жодного елемента
}
public void push(long j) // Розміщення елемента на вершині стека
{ stackArray[++top] = j; }
public long pop() // Вилучення елемента з вершини стека
{ return stackArray[top--]; }
public long peek() // Читання елемента з вершини стека
{ return stackArray[top]; }
public boolean isEmpty() // True, якщо стек порожній
{ return(top == -1); }
public boolean isFull() // True, якщо стек повний
{ return(top == maxSize-1); }
} // Кінець класу StackX

class StackApp
{
public static void main(String[] args)
{
StackX theStack = new StackX(10); // Створення нового стека
theStack.push(10); // Занесення елементів в стек
theStack.push(20);
theStack.push(30);
theStack.push(40);
while(!theStack.isEmpty()) // Поки стек не стане порожнім
```

```
{// Видалити елемент зі стеку
long value = theStack.pop();
System.out.print(value); // Вивід вмісту
System.out.print(" ");
}
System.out.println("");
} } // Кінець класу StackApp
```

Метод `main()` класу `StackApp` створює стек для зберігання 10 елементів, заносить до нього 4 елементи, а потім виводить всі елементи, витягуючи їх зі стека, поки він не спорожніє.

Метод `push()` збільшує `top`, щоб змінна вказувала на осередок, що знаходиться безпосередньо над поточним осередком, і зберігає в ньому елемент даних. Зверніть увагу: `top` збільшується до вставки елемента.

Метод `pop()` повертає значення, що знаходиться на вершині стека, після чого зменшує `top`. В результаті елемент, що знаходиться на вершині стека, фактично видаляється; він стає недоступним, хоча саме значення залишається в масиві (до тих пір, поки в клітинку не буде занесений інший елемент).

Метод `peek()` просто повертає верхнє значення, не змінюючи стан стека.

Методи `isEmpty()` і `isFull()` повертають `true`, якщо стек порожній або повний відповідно. Для порожнього стека змінна `top` містить `-1`, а для повного — `maxSize - 1`.

Обробка помилок

Єдиного підходу до обробки помилок стека не існує. Наприклад, що має відбуватися при занесенні елемента в заповнений стек або при спробі вилучення елемента з порожнього стека?

Відповідальність за обробку таких помилок покладається на користувача класу. Перш ніж вставляти елемент, користувач повинен перевірити, чи залишилися в стеку вільні комірки:

```
if(!theStack.isFull()) insert(item);
else System.out.print("Can’t insert, stack is full");
```

Заради простоти коду ми виключили перевірку з `main()` (до того ж в цій простій програмі ми знаємо, що стек не заповнений, бо щойно його самі ініціалізували). Перевірка порожнього стека виконується в методі `main()` при виклику `pop()`. Багато класів стеків виконують внутрішню перевірку таких помилок в методах `push()` і `pop()`. Таке рішення вважається кращим. У Java клас стеку, що виявив помилку, зазвичай ініціює виключення, яке може бути перехоплено і оброблено користувачем класу.

4.2 Приклад використання стека при перестановці літер у слові

Цей приклад вирішує дуже просте завдання: перестановка літер у слові. Створимо проект в NetBeans «Reverse» з трьома класами `StackX`, `Reverser` і `ReverseApp` (головний клас).

```
class Reverser
{
private String input; // Початковий рядок
private String output; // Оброблений рядок
public Reverser(String in) // Конструктор
{input = in; }
public String doRev() // Перестановка символів
{
int stackSize = input.length(); // Визначення розміру стека
StackX theStack = new StackX(stackSize); // Створення стека
for(int j = 0; j <input.length(); j++)
{
char ch = input.charAt(j); // Читання символу з вхідного потоку
theStack.push(ch); // Занесення в стек
}
output = "";
while(!theStack.isEmpty())
{
char ch = (char)theStack.pop(); // Вилучення символу зі стека
output = output + ch; // Приєднання до вихідного рядку
}
return output;
```

```
} } // Кінець класу Reverser

class ReverseApp
{
public static void main(String[] args) throws IOException
{
String input, output;
while(true)
{
System.out.print("Enter a string:");
System.out.flush();
input = getString(); // Читання рядка з клавіатури
if(input.equals("")) // Завершення, якщо рядок пустий
break;
// Створення об'єкта Reverser
Reverser theReverser = new Reverser(input);
output = theReverser.doRev();
System.out.println("Reversed:" + output);
} }
public static String getString() throws IOException
{
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
String s = br.readLine();
return s;
} } // Кінець класу ReverseApp
```

Клас `Reverser` переставляє символи вхідного рядка. Його найважливіший компонент метод `doRev()` вибирає розмір стека відповідно до довжини вхідного рядка. У методі `main()` ми отримуємо рядок від користувача, створюємо об'єкт `Reverser` з передачею цього рядка в аргументі конструктора, викликаємо метод `doRev()` об'єкта і виводимо повернене значення, в якому і міститься рядок з переставленими символами.

ПІДСУМКИ

✓ Стеки використовуються для спрощення деяких операцій з області програмування.

✓ У стеку доступний елемент, який був вставлений останнім.

✓ Основні операції зі стеком — вставка (занесення) елемента на вершину стека і вилучення елемента, що знаходиться на вершині.

ЗАВДАННЯ

1. У проекті Stack перепишіть метод `main()` так, щоб було додано 8 елементів і виведено на екран 4 останніх.

2. У проекті Stack перепишіть метод `main()` так, щоб користувач міг вводити кількість чисел для роботи зі стеком.

3. Створити проект, в якому введений рядок записується в стек, після чого витягнуті значення поміщаються в масив парних і непарних значень. Вивести ці масиви.

4. Створити проект, в якому вводиться рядок, що містить символи і числа, який розбивається на два стека: перший містить символи, другий числа. Організувати виведення вмісту обох стеків.

5. Створити проект, в якому введений рядок розбивається посимвольно в два стека: перший містить голосні, другий — приголосні. Організувати виведення вмісту обох стеків.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке стек і за яким принципом він працює?

2. У стек заносяться числа 70, 50, 40 і 10, після чого зі стека витягуються три елементи. Яке число залишиться в стеку?

3. Які операції виконуються зі стеком?

4. Назвіть методи класу `StackX` і що вони роблять?

Лабораторна робота № 5. Черга

Мета роботи: освоєння навичок роботи з чергами, вивчення черг як допоміжного інструментарію програміста на концептуальному рівні.

КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Черга — це структура даних, яка нагадує стек, але з черги першим витягується елемент, вставлений першим (FIFO, First-In-First-Out), тоді як зі стеку першим витягується елемент, вставлений останнім (LIFO, Last-In-First-Out).

Черги використовуються для моделювання реальних ситуацій очікування: клієнтів у банку, вильоту літаків або передачі даних по Інтернету.

Рис. 11. Демонстрація роботи черги

В операційній системі вашого комп’ютера (і в мережі) трудяться різні черги, які непомітно виконують свої обов’язки. У черзі друку завдання чекають звіль-

нення принтера. Дані, що вводяться з клавіатури, теж зберігаються в черзі. Якщо ви працюєте в текстовому редакторі, а комп'ютер на короткий час відволікся на виконання іншої операції, натискання клавіш не будуть втрачені; вони очікують у черзі, доки у редактора не з'явиться вільний час для їх отримання. Черга забезпечує зберігання натискань клавіш у вихідній послідовності до моменту обробки.

Існують дві основні операції з чергою:

- ✓ вставка елементу в кінець черги;
- ✓ витягування елементу з початку черги.

`Rear` — це маркер, який переміщається при вставці нового елементу, тобто вказує на новий елемент у кінці черги.

`Front` — це маркер, який переміщається при витягуванні елемента з черги, що знаходиться на початку черги. `Front` знаходиться нижче `Rear`, і тому у нього менший індекс, але це не завжди так.

5.1 Реалізація черги на мові Java

Створимо проект в NetBeans «Queue» з двома класами `Queue` і `QueueApp` (головний клас).

```
public class Queue
{
private int maxSize;
private long[] queArray;
private int front;
private int rear;
private int nItems;
public Queue(int s) // Конструктор
{
maxSize = s;
queArray = new long[maxSize];
front = 0;
rear = -1;
nItems = 0;
}
```

```
public void insert(long j) // Вставка елемента в кінець черги
{
if(rear == maxSize - 1) // Циклічне перенесення
rear = -1;
queArray[++rear] = j; // Збільшення rear і вставка
nItems++; // Збільшення кількості елементів
}
public long remove() // Витягування елемента з початку черги
{
long temp = queArray[front ++]; // Вибірка і збільшення front
if(front == maxSize) // Циклічне перенесення
front = 0;
nItems-- ; // Зменшення кількості елементів
return temp;
}
public long peekFront() // Читання елемента на початку черги
{ return queArray[front]; }
public boolean isEmpty() // true, якщо черга порожня
{ return (nItems == 0); }
public boolean isFull() // true, якщо черга заповнена
{ return (nItems == maxSize); }
public int size() // Кількість елементів у черзі
{ return nItems; }
} // Кінець класу Queue

public class QueueApp {
public static void main(String[] args)
{
Queue theQueue = new Queue(5); // Черга з 5 осередків
theQueue.insert(10); // Вставка 4 елементів
theQueue.insert(20);
theQueue.insert(30);
theQueue.insert(40);
theQueue.remove(); // Витягування 3 елементів
theQueue.remove(); //(10, 20, 30)
theQueue.remove();
```

```
theQueue.insert(50); // Вставка ще 4 елементів
theQueue.insert(60); //(З циклічним переносом)
theQueue.insert(70);
theQueue.insert(80);
while(!theQueue.isEmpty())    // Витягування і друк
{                             // всіх елементів
long n = theQueue.remove(); //(40, 50, 60, 70, 80)
System.out.print(n);
System.out.print(" ");
}
System.out.println("");
} }
```

У запропонованій реалізації в полях класу `Queue` зберігаються не тільки індекси початку і кінця черги, а й поточна кількість елементів `nItems`.

Метод `insert()` передбачає, що черга ще не заповнена. У загальному випадку при вставці спочатку збільшується індекс `rear`, а потім новий елемент вставляється в клітинку, на яку посилається його нове значення. Але якщо значення `rear` вже досягло вершини масиву `(maxSize-1)`, то перед виконанням вставки воно повинно повернутися до нижньої межі. Для цього полю `rear` присвоюється значення -1, щоб при збільшенні воно стало рівним 0 (нижня межа масиву). Код методу завершується збільшенням кількості елементів `nItems`.

Метод `remove()` передбачає, що черга не порожня. Витягування завжди починається з отримання значення в комірці `front` і збільшення `front`. Якщо значення `front` при цьому виходить за межу масиву, воно повертається до 0. На час перевірки цієї можливості значення, що повертається, зберігається в тимчасовій змінній. Код методу завершується зменшенням кількості елементів `nItems`.

Метод `peek()` повертає значення з комірки `front`. Деякі реалізації також дозволяють читати значення з нижньої межі масиву; в таких випадках методам зазвичай присвоюються імена `peekFront()` і `peekRear()` (або просто `front()` і `rear()`).

Методи `isEmpty()`, `isFull()` і `size()` повертають значення, що залежить

від поля `nItems`. Перші два методи порівнюють його з `0` і `maxSize`, а третій повертає поточне значення.

Ефективність черг: вставка і витягування елементів черги, як і елементів стека, виконуються за час O(1).

5.2 Дек

Дек (deque) являє собою двосторонню чергу. І вставка, і видалення елементів можуть виконуватися з обох кінців. Відповідні методи можуть називатися `insertLeft() / insertRight()` і `removeLeft() / removeRight()`.

Якщо обмежитися тільки методами `insertLeft()` і `removeLeft()` (або їх еквівалентами для правого кінця), дек працює як стек. Якщо ж обмежитися методами `insertLeft()` і `removeRight()` (або протилежної парою) — він працює як черга.

За своєю гнучкістю деки перевершують і стеки, і черги; іноді вони використовуються в бібліотеках класів-контейнерів для реалізації обох різновидів.

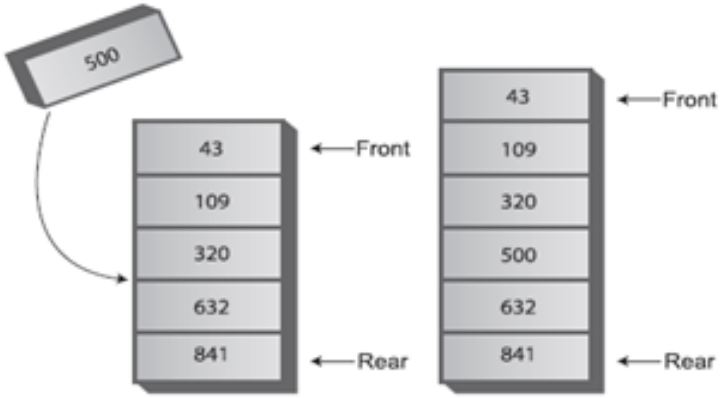
Тим не менш використовуються вони рідше стеків або черг.

5.3 Пріоритетні черги

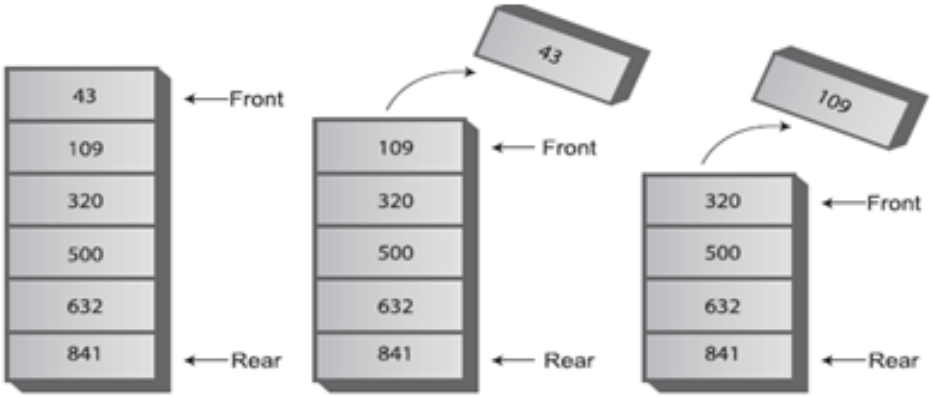
Пріоритетна черга є більш спеціалізованою структурою даних, ніж стек або черга, однак і вона несподівано часто виявляється корисною. У пріоритетній черзі, як і у звичайній, є початок і кінець, а елементи витягуються від початку. Але у пріоритетній черзі елементи упорядковуються по ключу, так що елемент з найменшим (в деяких реалізаціях — найбільшим) значенням ключа завжди знаходиться на початку. Нові елементи вставляються в позиціях, які зберігають порядок сортування.

Пріоритетні черги (як і звичайні) часто використовуються в комп'ютерних системах. Скажімо, в операційній системі з витісняючою багатозадачністю програми можуть розміщуватися в пріоритетній черзі, щоб високопріоритетна програма першою отримала процесорний час для її виконання. При роботі з пріоритетними чергами досить часто потрібно отримати доступ до елементу з наймен-

шим значенням ключа (що може становити найекономічний або найшвидший спосіб виконання якої-небудь операції) — тобто найпріоритетнішим є елемент з найменшим ключем.



Вставка нового елементу в пріоритетну чергу



Витягання двох елементів з пріоритетної черги

Рис. 12. Демонстрація роботи пріоритетної черги

5.4 Реалізація пріоритетної черги на мові Java

```
public class PriorityQ
{
// Елементи масиву упорядковано за значенням ключа
// від максимума (0) до мінімуму (maxSize-1)
private int maxSize;
private long[] queArray;
private int nItems;
public PriorityQ(int s) // Конструктор
{
maxSize = s;
```

```
queArray = new long[maxSize];
nItems = 0;
}
public void insert(long item) // Вставка елемента
{
int j;
if(nItems == 0) // Якщо черга порожня,
queArray[nItems ++] = item; // вставляємо в осередок 0
else // Якщо черга містить елементи
{
for(j = nItems-1; j>= 0; j--) // Перебір у зворотньому напрямку
{
if(item> queArray[j]) // Якщо новий елемент більший,
queArray[j+1] = queArray[j]; // зрушити вгору
else // Якщо менший — зрушення припиняється
break;
}
queArray[j + 1] = item; // Вставка елемента
nItems++;
} }
public long remove() // Витягування мінімального елемента
{ return queArray[--nItems]; }
public long peekMin() // Читання мінімального елемента
{ return queArray[nItems-1]; }
public boolean isEmpty() // true, якщо черга порожня
{ return (nItems == 0); }
public boolean isFull() // true, якщо черга заповнена
{ return (nItems == maxSize); }
} // Кінець класу PriorityQ

public class PriorityQApp
{
public static void main(String[] args) throws IOException
{
PriorityQ thePQ = new PriorityQ(5);
thePQ.insert(30);
```

```
thePQ.insert(50);
thePQ.insert(10);
thePQ.insert(40);
thePQ.insert(20);
while(!thePQ.isEmpty())
{
long item = thePQ.remove();
System.out.print(item + " "); // 10, 20, 30, 40, 50
}
System.out.println("");
} }
```

Метод `insert()` перевіряє, чи містить черга хоча б один елемент; якщо елементи відсутні, елемент вставляється в клітинку з індексом 0. В іншому випадку він починає з вершини масиву зрушувати існуючі елементи вгору, поки не знайде місце, що підходить для вставки нового елемента. Потім елемент вставляється, а значення `nItems` збільшується. Якщо існує ймовірність того, що пріоритетна черга заповнена, слід перевірити цю можливість методом `isFull()` перед викликом `insert()`.

Поля `front` і `rear` в цій реалізації не потрібні (на відміну від звичайної черги), тому що, як уже говорилося раніше, значення `front` завжди дорівнює `nItems-1`, а значення `rear` завжди дорівнює `0`.

Метод `remove()` гранично простий: він зменшує `nItems` і повертає елемент, що знаходиться на вершині масиву.

Метод `peekMin()` влаштований аналогічно, якщо не враховувати того, що він не зменшує `nItems`.

Методи `isEmpty()` і `isFull()` перевіряють, чи містить поле `nItems` значення `0` або `maxSize` відповідно.

Ефективність пріоритетних черг: у пріоритетній черзі вставка виконується за час O(N), а витягування — за час O(1).

ПІДСУМКИ

✓ Черги та пріоритетні черги зазвичай використовуються для спрощення деяких операцій з області програмування.

✓ У цих структурах даних доступний тільки один елемент, який був вставлений першим.

✓ Основні операції з чергою — вставка елемента в кінець черги і витягування елемента з початку черги.

✓ Черга може бути реалізована у формі циклічної черги (на базі масиву), в якій здійснюється циклічний перехід індексів від кінця масиву до початку.

✓ У пріоритетній черзі доступний елемент з найменшим (або іноді з найбільшим) значенням ключа.

✓ Основні операції з пріоритетною чергою — вставка елемента в порядку сортування та видалення елемента з найменшим значенням ключа.

✓ Ці структури даних реалізуються на базі масивів або інших структур (наприклад, зв'язаних списків).

ЗАВДАННЯ

1. Протестуйте проект Queue. Виконайте удосконалення проекту:

1) Організувати введення n елементів черги.

2) Створити чергу з n елементів (заповнення виконати функцією `random`).

3) Витягти з черги перші n елементів і вивести їх на екран.

2. У класі `Queue` створити метод `clear()` для видалення всіх елементів черги. Протестувати роботу даного методу.

3. Напишіть метод класу `Queue` для виведення вмісту черги. Вміст черги має виводитися від першого вставленого елементу до останнього, а користувач не повинен бачити, що послідовність переривається на кордоні масиву. Будьте уважні і простежте за тим, щоб один елемент і вміст порожньої черги виводилися коректно незалежно від положення `front` і `rear`.

4. Створіть клас `Deque` за описом деків (двосторонніх черг). Клас повинен містити методи `insertLeft()`, `insertRight()`, `removeLeft()`, `removeRight()`, `isEmpty()`. Також в ньому повинна бути реалізована підтримка циклічного переносу індексів, за аналогією з чергами.

5. Змініть всі методи класу `Queue`, щоб була реалізована черга без лічильника елементів. Продемонструвати роботу класу. Зростає при цьому ефективність методів `insert()` і `remove()`?

6. Створити проект, в якому, маючи рядок, потрібно сформувати чергу з

кожного третього символу, після чого витягти елементи.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке черга і які основні операції виконуються з чергою?
2. Що таке пріоритетна черга?
3. Порівняйте ефективність пріоритетної і звичайної черги.
4. У пріоритетну чергу заносяться числа 20, 50, 40 і 70, після чого витягуються 2 елементи. Які числа залишаться в пріоритетній черзі?
5. Для чого зручно використовувати пріоритетну чергу?
6. Чим відрізняється пріоритетна черга від звичайної?
7. Назвіть методи класу `Queue` і що вони роблять?
8. Назвіть методи класу `PriorityQ` і що вони роблять?

Лабораторна робота № 6. Зв'язані списки

Мета роботи: освоєння навичок роботи зі зв'язаними списками, засвоєння принципів використання їх як структур даних замість масивів.

КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

У комп'ютерній науці зв'язаний список є однією з основних структур даних, які використовуються в програмуванні. Він складається з послідовності вузлів, кожен з яких містить довільні поля даних і одне або два посилання (Links), що вказують на наступний і/або попередній вузли. Зв'язаний список є самостійним посилальним типом даних, оскільки він містить покажчик або посилання на інші дані того ж типу. Зв'язані списки дозволяють вставку і видалення вузлів в будь-якому місці в списку за постійний час, але не допускають довільного доступу. Існує кілька різних типів зв'язаних списків: зв'язані списки, двосторонні списки, сортовані списки, двозв'язні списки, списки з ітераторами (для вирішення проблеми довільного доступу до елементів списку).

6.1 Будова зв'язаного списку

У зв'язаному списку кожен елемент даних вбудовується в спеціальний

об’єкт, званий елементом списку (класу, на основі якого створюються такі об’єкти, часто присвоюється ім’я `Link`). Оскільки список містить багато однотипних елементів, для них зручно створити окремий клас, відмінний від класу самого зв’язаного списку. Кожен елемент (тобто об’єкт `Link`) містить посилання на наступний елемент списку; поле, в якому це посилання зберігається, зазвичай називається `next`. Об’єкт списку також містить посилання на перший елемент `first`. Відносини між об’єктами в цій архітектурі представлені на рис. 13.

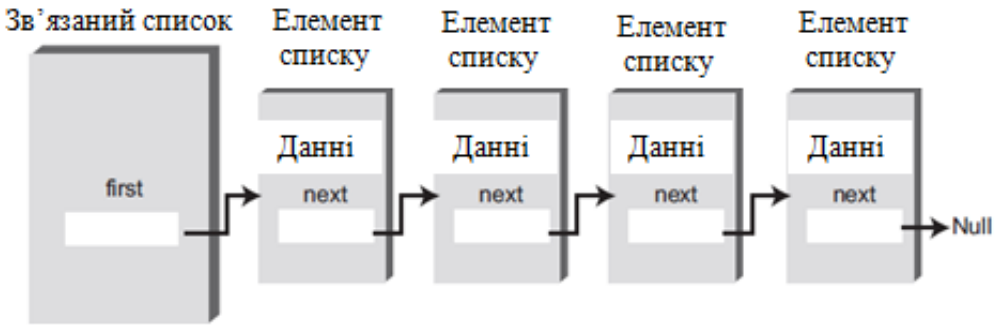


Рис. 13. Зв’язаний список

Далі наводиться частина визначення класу `Link`. У полях класу зберігаються дані і посилання на наступний елемент списку:

```
class Link
{
public int iData; // Дані
public double dData; // Дані
public Link next; // Посилання на наступний елемент списку
}
```

Подібні визначення класів іноді називаються самопосилальними (selfreferential), оскільки об’єкт класу містить поле (`next` в даному випадку) з посиланням на об’єкт того ж типу. У типовому додатку полів даних може бути набагато більше. Скажімо, запис про працівника фірми може містити поля імені, адреси, номери соціального страхування, посади, зарплати, а також інші атрибути. Замість окремих полів часто використовується об’єкт класу, що містить всі дані:

```
class AnotherLink
{
```

```
public inventoryItem iI; // Об'єкт з даними
public AnotherLink next; // Посилання на наступний елемент списку
}
```

6.2 Реалізація простого зв'язаного списку на мові Java

У першому прикладі проекту linkedLists представлений простий зв'язаний список.

Цей різновид списку підтримує наступні операції:

✓ Вставка елемента на початок списку.

✓ Видалення елемента з початку списку.

✓ Перебір списку для виведення вмісту.

Але перш ніж розбирати код проекту linkedLists, розглянемо найважливіші складові класів `Link` і `LinkList`.

Повне визначення класу `Link` виглядає так:

```
class Link
{
public int iData; // Дані
public double dData; // Дані
public Link next; // Наступний елемент у списку
public Link(int id, double dd) // Конструктор
{
iData = id; // Ініціалізація даних
dData = dd; //('Next' автоматично присвоюється null)
}
public void displayLink() // Вивід вмісту елемента
{ System.out.print("{" + iData + ", " + dData + "}"); }
}
```

Конструктор ініціалізує дані. Ініціалізувати поле `next` не потрібно, тому що до нього автоматично записується `null` при створенні. Значення `null` вказує на те, що посилання не вказує ні на що — до того, як елемент буде зв'язаний з іншими елементами, справа буде саме такою.

Поля даних `Link` (`iData` і т. д.) оголошені відкритими (`public`). Якщо ого-

лосити їх приватними, нам доведеться надавати відкриті методи для роботи з ними; це приведе до зайвого коду, а лістинг стане довшим і буде гірше читатися. Звичайно, в більш серйозній програмі всі поля даних класу `Link` варто оголосити з ключовим словом `private`.

Клас `LinkList` містить всього один елемент даних: посилання на перший елемент списку. Посилання зберігається в полі з іменем `first`. Це єдина інформація про місцезнаходження елементів, яка зберігається в списку. Інші елементи відслідковуються по ланцюжку посилань `next`, починаючи з елементу `first`.

```
class LinkList
{
private Link first; // Посилання на перший елемент списку
public void LinkList() // Конструктор
{ first = null; // Список поки не містить елементів }
public boolean isEmpty() // true, якщо список порожній
{ return (first == null); }
// ... Інші методи
}
```

Конструктор `LinkList` ініціалізує `first` значенням `null`. Коли `first` містить `null`, список не містить жодного елемента. Якби у списку був хоча б один елемент, в `first` зберігалося б посилання на нього. Метод `isEmpty()` використовує цей факт для перевірки відсутності елементів у списку.

Лістинг проекту linkedLists:

```
public class LinkList
{
...
// Вставка елементу в початок списку
public void insertFirst(int id, double dd)
{// Створення нового елемента
Link newLink = new Link(id, dd);
newLink.next = first; // NewLink - > старе значення first
first = newLink; // First - > newLink
}
public Link deleteFirst() // Видалення першого елементу
```

```
{//(Передбачається, що список не порожній)
Link temp = first; // Збереження посилання
first = first.next; //Вилучення: first-> посилання на другий елемент
return temp; // Метод повертає посилання на видалений елемент
}
public void displayList()
{
System.out.print("List(first  - > last):");
Link current = first; // Від початку списку
while(current != null) // переміщення до кінця списку
{
current.displayLink(); // Вивід даних
current = current.next; // Перехід до наступного елементу
}
System.out.println("");
} }
public class LinkListApp
{
public static void main(String[] args)
{
LinkList theList = new LinkList(); // Створення нового списку
theList.insertFirst(33, 5.99); // Вставка чотирьох елементів
theList.insertFirst(38, 2.99);
theList.insertFirst(26, 6.99);
theList.displayList(); // Вивід вмісту списку
while(!theList.isEmpty()) // Поки залишаються елементи,
{
Link aLink = theList.deleteFirst(); // Видалення елемента
System.out.print("Deleted"); // Вивід видаленого елемента
aLink.displayLink();
System.out.println("");
}
theList.displayList(); // Вивід вмісту списку
} }
```

6.3 Двосторонні списки

Двосторонній список схожий на звичайний зв’язаний список з однією додатковою можливістю: у ньому зберігається посилання не тільки на перший, але і на останній елемент.

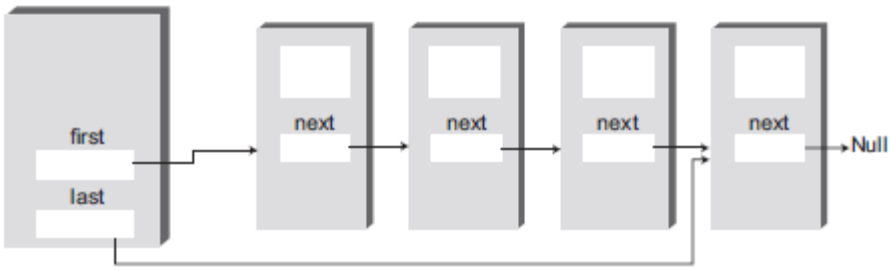


Рис. 14. Двосторонній список

Наявність посилання на останній елемент дозволяє вставляти нові елементи не тільки на початку, але і в кінці списку. Звичайно, новий елемент можна вставити в кінці і звичайного однозв’язного списку, перебираючи весь його вміст до останнього елемента, але таке рішення неефективне.

6.4 Реалізація двостороннього списку на мові Java

Створимо проект в NetBeans «firstlastlist» з трьома класами `Link`, `FirstLastList` і `FirstLastApp`.

Лістинг проекта «firstlastlist»:

```
public class Link {
public long dData; // Дані
public Link next; // Наступний елемент у списку
public Link(long dd) // Конструктор
{ dData = dd; }
public void displayLink() // Вивід вмісту елемента
{ System.out.print(" " + dData); }
}
public class FirstLastList {
private Link first; // Посилання на перший елемент
private Link last; // Посилання на останній елемент
```

```
public FirstLastList() // Конструктор
{
first = null; // Список поки не містить елементів
last = null;
}
public boolean isEmpty() // true, якщо список порожній
{ return first == null; }
public void insertFirst(long dd) // Вставка елемента в
{                                // початок списку
Link newLink = new Link (dd); // Створення нового елемента
if (isEmpty()) // Якщо список порожній,
last = newLink; // NewLink < -  last
newLink.next = first; // NewLink  - > старе значення first
first = newLink; // First  - > newLink
}
public void insertLast(long dd) // Вставка елемента в
{                               // кінець списку
Link newLink = new Link(dd); // Створення нового елемента
if(isEmpty()) // Якщо список порожній,
first = newLink; // First  - > newLink
else
last.next = newLink; // Старе значення last  - > newLink
last = newLink; // NewLink < -  last
}
public long deleteFirst() // Видалення першого елементу списку
{// Передбачається, що список не порожній
long temp = first.dData;
if (first.next == null) // Якщо тільки один елемент
last = null; // Null < -  last
first = first.next; // First  - > старе значення next
return temp;
}
public void displayList()
{
System.out.print("List(first  - > last):");
Link current = first; // Від початку списку
```

```
while(current != null) // Переміщення до кінця списку
{
current.displayLink(); // Вивід даних
current = current.next; // Перехід до наступного елементу
}
System.out.println("");
} }

public class FirstLastApp {
public static void main(String[] args)
{
FirstLastList theList = new FirstLastList();
theList.insertFirst(3); // Вставка в початок списку
theList.insertFirst(2);
theList.insertFirst(1);
theList.displayList(); // Вивід вмісту списку
theList.insertLast(4); // Вставка в кінець списку
theList.insertLast(5);
theList.insertLast(6);
theList.displayList(); // Вивід вмісту списку
theList.deleteFirst(); // Видалення перших двох елементів
theList.deleteFirst();
theList.displayList(); // Повторний вивід даних
} }
```

Клас двостороннього списку називається `FirstLastList`. Як згадувалося раніше, два поля даних, `first` і `last`, містять посилання відповідно на перший і останній елементи списку. Якщо список містить всього один елемент, то на нього посилаються обидва поля, а за відсутності елементів обидва поля містять `null`. Клас містить новий метод `insertLast()` для вставки нового елемента в кінець списку.

Методи вставки і видалення аналогічні методами одностороннього списку. Втім, обом методам вставки доводиться перевіряти особливий випадок, коли список порожній до виконання вставки. Інакше кажучи, якщо виклик `isEmpty()` повертає `true`, то метод `insertFirst()` повинен присвоїти `last` посилання на но-

вий елемент, а метод `insertLast()` повинен присвоїти це посилання `first`.

При вставці в початок списку методом `insertFirst()` в поле `first` заноситься посилання на новий елемент; при вставці в кінець списку методом `insertLast()` посилання на новий елемент заноситься в поле `last`. При видаленні з початку списку також існує особливий випадок з видаленням останнього елемента в списку (у полі `last` в цьому випадку необхідно записати `null`). На жаль, перехід від звичайного списку до двостороннього не спрощує видалення останнього елемента, адже у вашому розпорядженні немає посилання на передостанній елемент, в полі `next` якого необхідно записати `null`.

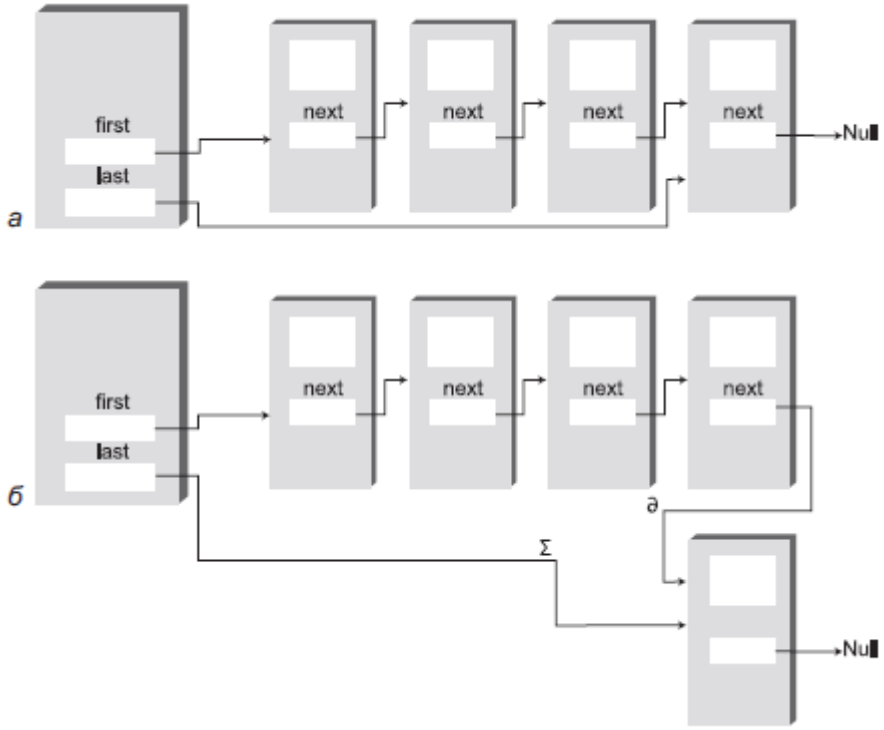


Рис. 15. Вставка нового елемента в кінець списку:
а — до вставки; б — після вставки

6.5 Реалізація стека на базі зв'язаного списку

Створимо проект в NetBeans «LinkStack» з чотирма класами `Link` (наведено вище), `LinkList, LinkStack` і головним класом `LinkStackApp`.

Лістинг проекту «LinkStack»:

```
public class LinkList {
private Link first; // Посилання на перший елемент у списку
```

```
public LinkList() // Конструктор
{ first = null; } // Список поки не містить елементів
public boolean isEmpty() // true, якщо список порожній
{ return(first == null); }
public void insertFirst(long dd) //Вставка елемента в початок списку
{// Створення нового елемента
Link newLink = new Link(dd);
newLink.next = first; // NewLink  - > старе значення first
first = newLink; // First  - > newLink
}
public long deleteFirst() // Видалення першого елемента
{//(Передбачається, що список не порожній)
Link temp = first; // Збереження посилання
first = first.next; //Вилучення: first-> посилання на другий елемент
return temp.dData; // Метод повертає дані видаленого елемента
}
public void displayList()
{
Link current = first; // Від початку списку
while(current != null) // Переміщення до кінця списку
{
current.displayLink(); // Вивід даних
current = current.next; // Перехід до наступного елементу
}
System.out.println("");
} }

public class LinkStack {
private LinkList theList;
public LinkStack() // Конструктор
{ theList = new LinkList(); }
public void push(long j) // Розміщення елемента на вершині стека
{ theList.insertFirst(j); }
public long pop() // Витягування елемента з вершини стека
{ return theList.deleteFirst(); }
public boolean isEmpty() // true, якщо стек порожній
```

```
{ return(theList.isEmpty()); }
public void displayStack()
{
System.out.print("Stack(top  - > bottom):");
theList.displayList();
} }

public class LinkStackApp {
public static void main(String[] args)
{
LinkStack theStack = new LinkStack(); // Створення стека
theStack.push(1); // Вставка елементів
theStack.push(2);
theStack.push(3);
theStack.displayStack(); // Вивід вмісту стека
theStack.push(4); // Вставка елементів
theStack.push(5);
theStack.displayStack(); // Вивід вмісту стека
theStack.pop(); // Витягування елементів
theStack.pop();
theStack.displayStack(); // Вивід вмісту стека
} }
```

Метод `main()` в класі `LinkStackApp` працює тільки з класом `LinkStack`, а клас `LinkStack` працює тільки з класом `LinkList`. Якісь безпосередні взаємодії між `main()` і класом `LinkList` відсутні.

А саме, коли метод `main()` викликає операцію `push()` класу `LinkStack`, цей метод в свою чергу викликає метод `insertFirst()` класу `LinkList` для виконання фактичної вставки даних. Аналогічним чином `pop()` викликає `deleteFirst()` для видалення елемента, а `displayStack()` викликає `displayList()` для виведення вмісту стека. З точки зору користувача класу, який програмує `main()`, використання реалізації `LinkStack` нічим не відрізняється від використання реалізації на базі масиву.

ПІДСУМКИ

✓ Зв'язаний список складається з одного об'єкта `linkedList` і деякого числа об'єктів `Link`.

✓ Об'єкт `linkedList` містить посилання на перший елемент списку (якому часто присвоюється ім'я `first`).

✓ Кожен об'єкт `Link` містить дані і посилання на наступний елемент списку (якому часто присвоюється ім'я `next`).

✓ Якщо поле `next` дорівнює `null`, це є ознакою кінця списку.

✓ Щоб вставити елемент на початку зв'язаного списку, необхідно зберегти в поле `next` нового елемента посилання, що зберігалося раніше в полі `first` списку, а в поле `first` — посилання на новий елемент.

✓ Щоб видалити елемент на початку списку, необхідно присвоїти `first` посилання на елемент `first.next`.

✓ Щоб переміщатися по зв'язаному списку, необхідно почати з елемента `first`, а потім послідовно переходити за посиланням `next` до наступного елементу списку.

✓ Пошук елемента із заданим ключем здійснюється переміщенням по списку.

✓ Коли елемент буде знайдений, користувач може вивести його вміст, видалити або виконати з ним іншу операцію — наприклад, можна вставити новий елемент до або після нього.

✓ Двосторонній список зберігає посилання на останній елемент списку (яке часто називається `last`) поряд з посиланням на перший елемент.

✓ У двосторонньому списку можлива вставка елементів в кінці списку.

ЗАВДАННЯ

1. Доопрацювати метод `main()` класу `LinkListApp`:

1) Організувати вставку певної кількості елементів, заданих користувачем.

2) Вставка заповнюється методом `random()`.

3) Створити метод `display()` в класі `LinkList` для виведення певної кіль-

кості елементів списку і застосувати його в `main()`.

2. Доповнити клас `LinkList` методами для пошуку і видалення елемента із заданим значенням ключа.

3. Доповнити клас `FirstLastList` методом `replaceLast()` для заміни останнього елемента зі списку.

4. Створити метод `insertAfter()` в класі `FirstLastList` для пошуку заданого елемента і вставки після нього нового. Операція складна, тому що в цій ситуації необхідно змінити чотири посилання. Перш за все слід знайти елемент із заданим ключем; пошук виконується так само, як у методі `find()`. Потім, якщо позиція вставки знаходиться не в кінці списку, необхідно створити два зв'язки між новим і наступним елементом, і ще два — між поточним і новим елементом. Якщо новий елемент повинен вставлятися в кінці списку, то його поле `next` повинно містити `null`, а поле `last` — посилання на новий елемент.

5. Створити проект, в якому реалізується чергу на базі зв'язаного списку.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке зв'язний список?

2. Що таке елемент списку?

3. Як представлені відносини між об'єктами зв'язаного списку?

4. Що таке самопосилальний клас?

5. Які різновиди операцій підтримує простий зв'язний список?

6. Чому не потрібно ініціалізувати поле `next` класу `Link`?

7. Чим відрізняється двосторонній зв'язний список від простого?

8. Що означає команда `newLink.next = first` в коді методу `insertFirst()`?

9. Припустимо, поле `current` вказує на передостанній елемент однозв'язного списку. Як виглядає команда для видалення останнього елемента списку?

Лабораторна робота № 7. Двійкові дерева

Мета роботи: освоєння навичок роботи з двійковими деревами, вивчення їх переваг перед масивами і зв’язаними списками.

КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

В ідеалі розробникам програмного забезпечення хотілося б мати структуру даних із швидкими операціями вставки/видалення, як у зв’язаному списку, і швидким пошуком, як у впорядкованому масиві. Дерева мають обидві ці характеристики; крім того, це одна з найцікавіших структур даних.

Дерево складається з вузлів, з’єднаних ребрами (рис. 16). У такому поданні дерева вузли позначаються кружками, а ребра - лініями, що з’єднують ці кружки.

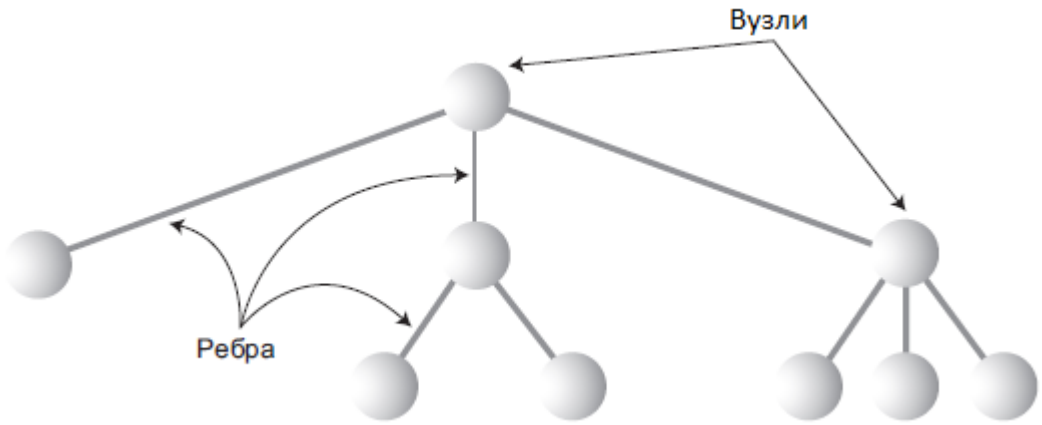


Рис. 16. Загальне дерево

Ребра представляють відносини між вузлами. Спрощено кажучи, програма може легко (і швидко) перейти від вузла до вузла, якщо між ними є сполучна лінія. Більше того, переходи між вузлами можливі тільки по з’єднувальних лініях. У загальному випадку переміщення відбувається тільки в одному напрямку: від кореневого вузла вниз. У програмах, написаних на Java, ребра звичайно представляються посиланнями.

Двійкове дерево є різновидом дерев. В цьому дереві вузли мають не більше двох нащадків. Більш загальні дерева, в яких вузли можуть мати більше двох нащадків, називаються багатоколійними деревами.

Два нащадки кожного вузла двійкового дерева називаються лівим і правим нащадком залежно від позиції на зображенні дерева (рис. 17). Число нащадків вузла двійкового дерева не обов’язково повинно бути рівним 2. Вузол може мати тільки лівого або тільки правого нащадка або не мати нащадків взагалі (листовий вузол).

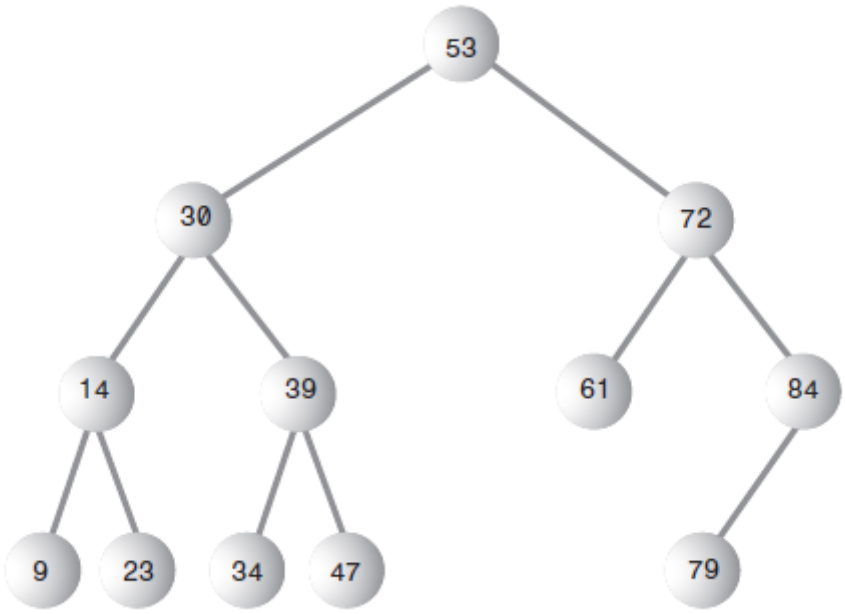


Рис. 17. Двійкове дерево

Визначальна характеристика дерева двійкового пошуку: ключ лівого нащадка вузла повинен бути меншим, ніж у батька, а ключ правого нащадка — більшим або рівним ключу батька.

Деякі дерева є незбалансованими, тобто більшість вузлів зосереджено з одного чи з іншого боку кореня (рис. 18). Незбалансованими також можуть бути окремі піддерева.

Дерева стають незбалансованими через порядок вставки елементів даних. Якби ключі вставлялися випадковим чином, то і дерево було б більш-менш збалансованим. Однак при генеруванні зростаючої (11, 18, 33, 42, 65 і т. д.) або спадаючої послідовності всі значення будуть відповідно правими або лівими нащадками, а дерево стане незбалансованим.

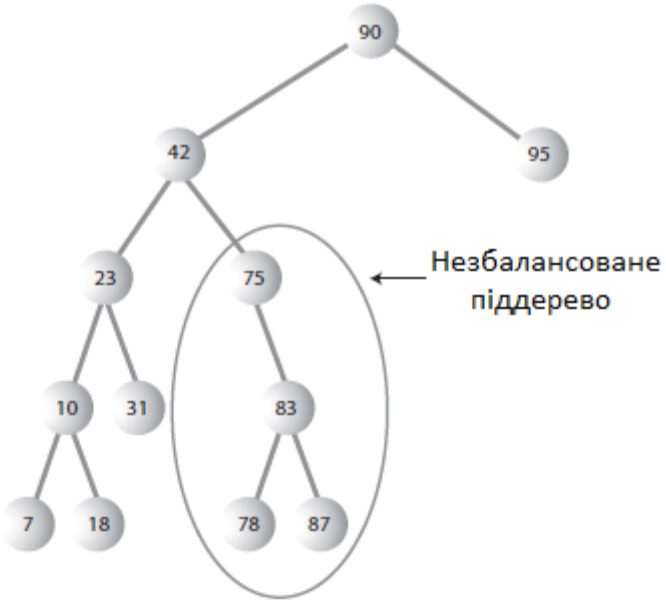


Рис. 18. Незбалансоване дерево

7.1 Реалізація двійкового дерева на мові Java

Для початку нам знадобиться клас для представлення об’єктів вузлів. Клас містить дані, що представляють збережені об’єкти (наприклад, описи працівників для бази даних відділу кадрів), а також посилання на кожного з двох нащадків поточного вузла. Визначення класу виглядає так:

```
class Node
{
int iData; // Дані, використовувані як ключ
double dData; // Інші дані
Node leftChild; // Лівий нащадок вузла
Node rightChild; // Правий нащадок вузла
public void displayNode() { }
}
```

Нам також знадобиться клас для представлення не окремих вузлів, а всього дерева. Цей клас буде називатися `Tree`. Він містить тільки одне поле: змінну `Node`, у якій зберігається корінь дерева. Поля для інших вузлів не потрібні, оскільки доступ до них здійснюється через кореневий вузол. Клас `Tree` містить ряд методів для пошуку, вставки і видалення вузлів, різних видів обходу і виведення вмісту дерева. «Скелет» класу виглядає так:

```
class Tree
{
private Node root; // Єдине поле даних
public Node find(int key) { }
public void insert(int id, double dd) { }
public boolean delete(int key) { }
// Інші методи
}
```

Нарешті, із створеним деревом необхідно виконувати операції. Назвемо цей клас `TreeApp`:

```
class TreeApp
{
public static void main(String[] args) { }
}
```

7.1.1 Пошук вузла — метод `void find(int key)`

Пошук вузла з заданим ключем — найпростіша з основних операцій з деревами. Вузли в дереві двійкового пошуку відповідають об'єктам, що містять інформацію. Це можуть бути об'єкти `person` з ключовим полем табельного номеру, а також полями імені, адреси, телефону, зарплати і т. д. Або об'єкти, що представляють деталі машин, з ключовим полем коду деталі і полями поточної кількості, ціни і т.д.

Розглянемо схему пошуку певного значення (57) в двійковому дереві (рис. 19).

На рис. 19 переміщення стрілки починається від кореневого вузла. Програма порівнює ключ 57 з ключем кореневого вузла, рівним 63. Бажаний ключ менше, тому програма знає, що шуканий вузол знаходиться в лівій частині дерева — або це лівий нащадок кореневого вузла, або один з його нащадків. Ключ лівого нащадка кореневого вузла дорівнює 27; порівняння 57 і 27 показує, що шуканий вузол належить правіше під деревом вузла 27. Стрілка переходить до вузла 51 — корінь цього піддерева. Значення 57 більше 51, тому процес пошуку спочатку переходить направо до 58, а потім наліво до 57. Цього разу ключ 57 збігається з шу-

каним значенням — вузол успішно знайдений.

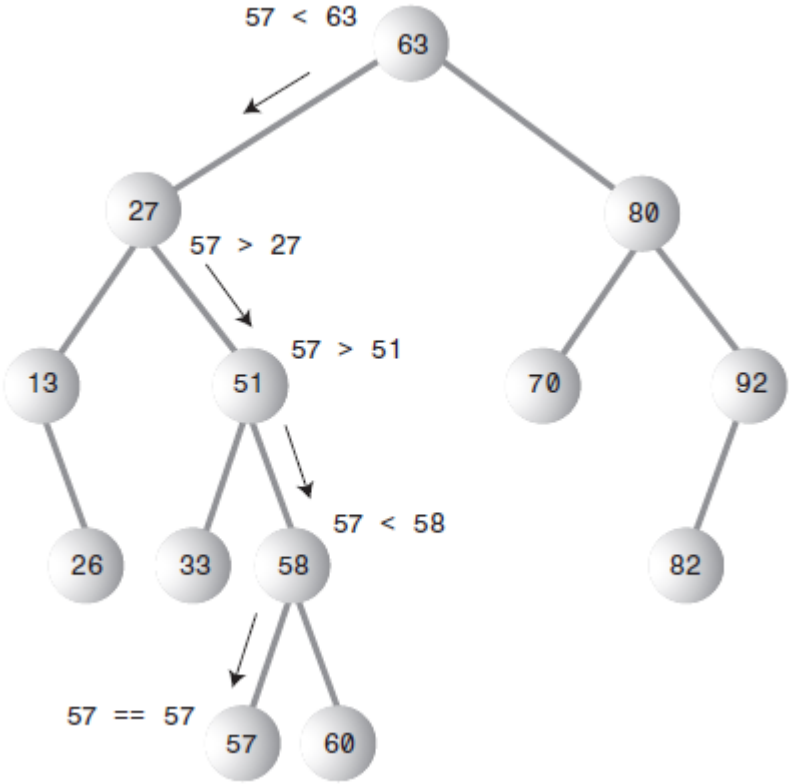


Рис. 19. Пошук в двійковому дереві

Код функції пошуку `find()`, оформленої у вигляді методу класу `Tree`:

```
public Node find(int key) // Пошук вузла з заданим ключем
{//(Передбачається, що дерево не порожнє)
Node current = root; // Почати з кореневого вузла
while(current.iData != key) // Ще не знайдено
{
if(key < current.iData) // Рухатися наліво?
current = current.leftChild;
else
current = current.rightChild; // Або направо?
if(current == null) // Якщо нащадка немає,
return null; // Пошук завершився невдачею
}
return current; // Елемент знайдений
}
```

Для зберігання вузла, що зараз перевіряється, використовується змінна `current`. Шукане значення зберігається в аргументі `key`. Пошук починається з кореневого вузла (це необхідно, оскільки в дереві безпосередньо доступний тільки цей вузол), тобто на початку роботи `current` присвоюється посилання на кореневий вузол. Потім у циклі `while` шукане значення `key` порівнюється зі значенням поля `iData` (ключового поля) поточного вузла. Якщо значення `key` менше, то `current` присвоюється посилання на лівого нащадка, а якщо більше (або дорівнює) — то посилання на правого нащадка вузла.

Якщо посилання `current` стає рівним `null`, значить, знайти наступного нащадка не вдалося; перебір досяг кінця дерева, шуканий вузол не знайдений, а отже, не існує. Метод повідомляє про цей факт, повертаючи `null`.

Якщо умову циклу `while` порушено, це означає, що шуканий вузол був успішно знайдений. Метод повертає вузол, щоб код, що викликав `find()`, зміг звернутися до полів цього вузла.

7.1.2 Вставка вузла — метод `void insert(int id, double dd)`

Щоб вставити вузол, необхідно спочатку знайти місце для його вставки. Метод відстежує вузол від кореня до вузла, який стане батьком нового вузла. Коли батько буде знайдений, новий вузол вставляється як лівий або правий нащадок залежно від того, чи буде ключ нового вузла менше або більше батьківського ключа.

Процедура вставки вузла починається з пошуку позиції для вставки. Розглянемо схему вставки певного значення (45) в двійковому дереві, яка показана на рис. 20.

Значення 45 менше 60, але більше 40 — переходимо до вузла 50. Тепер потрібно рухатися наліво, тому що 45 менше 50, але у вузла 50 немає лівого нащадка; його поле `leftChild` дорівнює `null`. При виявленні `null` алгоритм вставки вважає, що місце для приєднання нового вузла знайдено. Додаток створює новий вузол з ключем 45 і зв’язує його з вузлом 50 в якості лівого нащадка, як показано на рис. 20, б.

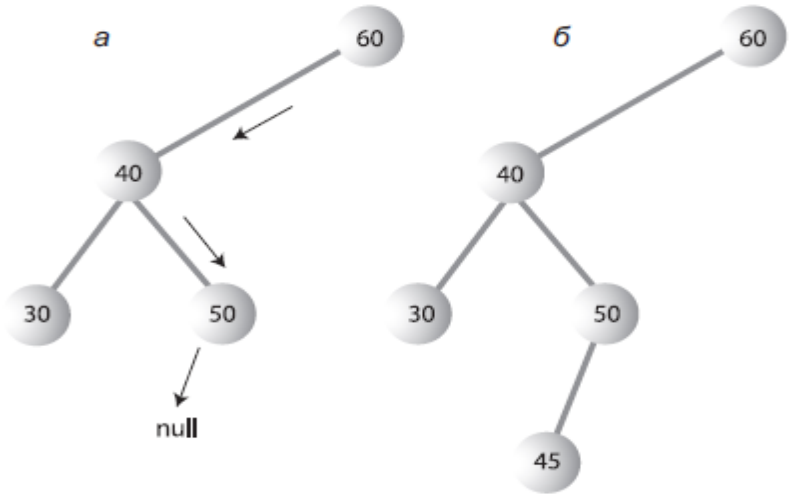


Рис. 20. Вставка вузла в двійкове дерево: а) до вставки; б) після вставки

Код функції `insert()`:

```
public void insert(int id, double dd)
{
Node newNode = new Node(); // Створення нового вузла
newNode.iData = id; // Вставка даних
newNode.dData = dd;
if(root == null) // Кореневий вузол не існує
root = newNode;
else // Кореневий вузол зайнятий
{
Node current = root; // Почати з кореневого вузла
Node parent;
while(true) //(Внутрішній вихід з циклу)
{
parent = current;
if(id <current.iData) // Рухатися наліво?
{
current = current.leftChild;
if(current == null) // Якщо досягнутий кінець ланцюжка
{// Вставити зліва
parent.leftChild = newNode;
return;
```

```
} }
else // Або направо?
{
current = current.rightChild;
if(current == null) // Якщо досягнутий кінець ланцюжка,
{// Вставити праворуч
parent.rightChild = newNode;
return;
} } } } }
```

Спочатку метод `insert()` створює новий вузол на основі даних, переданих в аргументах. Далі він повинен визначити місце для вставки нового вузла. Використовується приблизно такий же код, як і при пошуку вузла.

Шуканим значенням є елемент, який передається в аргументі `id`. Цикл `while` використовує умову `true`, тому що виявлення вузла з тим же значенням, що у `id`, ігнорується; вузол зі співпадаючим ключем інтерпретується так, як ніби його ключ був більше шуканого.

Місце для вставки нового вузла завжди знаходиться успішно (якщо в системі вистачить пам'яті); коли це відбудеться, новий вузол включається в дерево, а виконання циклу `while` завершується командою `return`.

У новій змінній `parent` (батько `current`) зберігається останній відмінний від `null` вузол, відвіданий при переборі. Зберігання цього вузла необхідно, тому що для перевірки того, що попереднє значення `current` не мало відповідного нащадка, `current` присвоюється `null`. Не зберігши `parent`, метод втратить поточну позицію в дереві. Вставка нового вузла здійснюється зміною відповідного покажчика на нащадка в `parent` (останній відвіданий вузол, відмінний від `null`). У разі безуспішного пошуку лівого нащадка `parent` новий вузол приєднується до `parent` як лівий нащадок, а при пошуку правого нащадка — відповідно як правий нащадок.

7.1.3 Видалення вузла — метод `public boolean delete(int key)`

Видалення починається з пошуку видаляємого вузла — процедура пошуку

вже була продемонстрована раніше в методах `find()` і `insert()`. Коли вузол буде знайдений, необхідно розглянути три можливі випадки:

1. Вузол, що видаляється, є листовим (не має нащадків).

2. Вузол, що видаляється, має одного нащадка.

3. Вузол, що видаляється, має двох нащадків.

Видалення вузлів — вельми непроста операція. Вона настільки складна, що деякі програмісти вважають за краще обходитися без неї. Вони включають в клас `node` нове поле логічного типу з іменем виду `isDeleted`. Щоб видалити вузол, вони просто привласнюють цьому полю значення `true`. Інші операції — такі, як `find()`, перш ніж працювати з вузлом, перевіряють це поле і переконуються в тому, що вузол не позначений як видалений. При такому підході видалення вузла не змінює структуру дерева. Звичайно, це також означає, що пам'ять може заповнюватися «видаленими» вузлами.

Такий підхід виглядає компромісно, але він може бути придатним при відносно невеликій кількості вилучень з дерева. Наприклад, якщо дані колишніх працівників залишаються в базі даних відділу кадрів назавжди.

Код методу для видалення вузла представлений нижче:

```
public boolean delete(int key) // Видалення вузла з заданим ключем
{//(Передбачається, що дерево не порожнє)
Node current = root;
Node parent = root;
boolean isLeftChild = true;
while(current.iData != key) // Пошук вузла
{
parent = current;
if(key < current.iData) // Рухатися наліво?
{
isLeftChild = true;
current = current.leftChild;
}
else // Або направо?
{
```

```
isLeftChild = false;
current = current.rightChild;
}
if(current == null) // Кінець ланцюжка
return false; // Вузол не знайдений
}
// Вузол, що видаляється, знайдений
// Якщо вузол не має нащадків, він просто видаляється.
if(current.leftChild == null &&
current.rightChild == null)
{
if(current == root) // Якщо вузол є кореневим,
root = null; // Дерево очищається
else if(isLeftChild)
parent.leftChild = null; // Вузол від’єднується від батька
else
parent.rightChild = null;
}
// Якщо немає правого нащадка, вузол замінюється лівим піддерево
else if(current.rightChild == null)
if(current == root)
root = current.leftChild;
else if(isLeftChild)
parent.leftChild = current.leftChild;
else
parent.rightChild = current.leftChild;
// Якщо немає лівого нащадка, вузол замінюється правим піддеревом
else if(current.leftChild == null)
if(current == root)
root = current.rightChild;
else if(isLeftChild)
parent.leftChild = current.rightChild;
else
parent.rightChild = current.rightChild;
else {// Є два нащадка, вузол замінюється наступником
// Пошук наступника для вузла, що видаляється(current)
```

```
Node successor = getSuccessor(current);
// Батько current зв’язується з посередником
if(current == root)
root = successor;
else if(isLeftChild)
parent.leftChild = successor;
else
parent.rightChild = successor;
}
// Наступник зв’язується з лівим нащадком current
return true; // Ознака успішного завершення
}
// Метод повертає вузол з наступним значенням після delNode.
// Для цього він спочатку переходить до правого нащадку, а потім
// відстежує ланцюжок лівих нащадків цього вузла.
private Node getSuccessor(Node delNode)
{
Node successorParent = delNode;
Node successor = delNode;
Node current = delNode.rightChild; // Перехід до правого нащадку
while(current != null) // Поки залишаються ліві нащадки
{
successorParent = successor;
successor = current;
current = current.leftChild; // Перехід до лівого нащадку
}
// Якщо наступника немає
if(successor != delNode.rightChild) // правим нащадком,
{// Створити зв’язки між вузлами
successorParent.leftChild = successor.rightChild;
successor.rightChild = delNode.rightChild;
}
return successor;
}
```

7.2 Обхід дерева

Обходом дерева називається відвідування всіх його вузлів в певному порядку. На практиці обхід використовується не так часто, як пошук, вставка і видалення вузлів. Одна з причин цього полягає в тому, що алгоритми обходу не відрізняються швидкістю. Однак обхід дерева буває корисним у деяких обставинах, і він представляє інтерес з теоретичної точки зору. Також обхід є простішою дією, ніж видалення.

Існують три простих алгоритми обходу дерева: прямий (preorder), симетричний (inorder) і зворотний (postorder). Для дерев двійкового пошуку найчастіше застосовується алгоритм симетричного обходу, тому розглянемо цей алгоритм.

7.2.1 Симетричний обхід

При симетричному обході двійкового дерева всі вузли перебираються в порядку зростання ключів. Якщо вам потрібно створити відсортований список даних двійкового дерева — це одне з можливих рішень.

Найпростіший спосіб обходу заснований на використанні рекурсії. При виклику рекурсивного методу для обходу всього дерева в аргументі передається вузол. Спочатку цим вузлом є корінь дерева. Метод повинен виконати тільки три операції:

1. Виклик самого себе для обходу лівого піддерева вузла.

2. Відвідування вузла.

3. Виклик самого себе для обходу правого піддерева вузла.

Під відвідуванням вузла мається на увазі виконання деякої операції: виведення даних, запис у файл і т.д. Обхід працює з будь-яким двійковим деревом, а не тільки з деревами двійкового пошуку. Алгоритм обходу не звертає уваги на значення ключів, його цікавить лише наявність у вузла нащадків. Код метода `inOrder()`:

```
private void inOrder(Node localRoot)
{
if(localRoot != null)
```

```
{
inOrder(localRoot.leftChild);
System.out.print(localRoot.iData + " ");
inOrder(localRoot.rightChild);
} }
```

У вихідному виклику методу в аргументі передається кореневий вузол root: `inOrder (root)`. Далі метод діє самостійно, рекурсивно викликаючи самого себе до тих пір, поки не залишиться вузлів для обходу.

ПІДСУМКИ

- ✓ Дерева складаються з вузлів, з'єднаних ребрами.
- ✓ Кореневим вузлом (або коренем) називається вузол верхнього рівня дерева. Кореневий вузол не має батька.
- ✓ У двійковому дереві вузол має не більше двох нащадків.
- ✓ У дереві двійкового пошуку ключі всіх вузлів, які є лівими нащадками вузла А, менше ключа А; ключі всіх вузлів, що є правими нащадками вузла А, більше ключа А (або дорівнюють йому).
- ✓ Вузли відповідають об'єктам даних, збережених у дереві.
- ✓ Ребра зазвичай подаються у програмах посиланнями на нащадків вузла (а іноді й на батька).
- ✓ У процесі пошуку вузла алгоритм порівнює шукане значення з ключем вузла і переходить до лівого нащадку цього вузла, якщо шукане значення менше, або до правого, якщо воно більше.
- ✓ При виконанні вставки алгоритм спочатку знаходить місце для нового вузла, а потім змінює посилання на нащадка у батьківському вузлі, щоб воно вказувало на вставлений вузол.
- ✓ Якщо вузол не має нащадків, то для його видалення достатньо записати null в поле посилання на нащадка в його батьківському вузлі.
- ✓ Щоб видалити вузол з одним нащадком, слід записати посилання на його нащадка в поле батьківського вузла.
- ✓ Щоб видалити вузол з двома нащадками, слід замінити його наступни-

ком.

✓ Вузли зі співпадаючими значеннями ключів можуть створити проблеми в масивах, тому що при пошуку буде знайдений тільки перший з них.

✓ Дерева можуть представлятися в пам'яті комп'ютера у вигляді масиву, хоча представлення з посиланнями є більш поширеним.

ЗАВДАННЯ

1. Створити проект «BinaryTree» з реалізації роботи двійкового дерева. Лістинги і опис класів представлено вище. Програма повинна виконувати наступні дії:

- Створення дерева.
- Додавання елемента до дерева.
- Пошук елемента.
- Видалення елемента.

2. Освоїти, а також більш докладно розібрати і опрацювати метод видалення вузла.

3. Створити метод для пошуку мінімального значення в дереві.

4. Створити метод для пошуку максимального значення в дереві.

5. Вивчити і розібратися з методом `displayTree()` — метод для відображення дерева. Використовувати даний метод у програмі «BinaryTree». Код представлений нижче.

```
public void displayTree()
{
Stack globalStack = new Stack();
globalStack.push(root);
int nBlanks = 32;
boolean isRowEmpty = false;
System.out.println("........................");
while(isRowEmpty == false)
{
Stack localStack = new Stack();
isRowEmpty = true;
```

```
for(int j = 0; j < nBlanks; j++)
System.out.print(" ");
while(globalStack.isEmpty() == false)
{
Node temp = (Node)globalStack.pop();
if(temp != null)
{
System.out.print(temp.iData);
localStack.push(temp.leftChild);
localStack.push(temp.rightChild);
if(temp.leftChild != null ||
temp.rightChild != null)
isRowEmpty = false;
}
else
{
System.out.print(" -  - ");
localStack.push(null);
localStack.push(null);
}
for(int j=0; j<nBlanks*2 - 2; j++)
System.out.print(" ");
}
System.out.println();
nBlanks /= 2;
while(localStack.isEmpty()==false)
globalStack.push(localStack.pop());
}
System.out.println(".........................");
}
```

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке дерево і з чого воно складається?
2. Що таке двійкове і багатоколійне дерево?
3. Що таке лівий і правий нащадок?

4. Коли двійкове дерево називається деревом пошуку?

5. Чи всі дерева є двійковими деревами?

6. Є повне двійкове дерево з 20 вузлами. Передбачається, що корінь знаходиться на рівні 0. Скільки вузлів знаходиться на рівні 4?

7. Що перевіряє алгоритм пошуку в дереві, переміщаючись від вузла до вузла?

8. Опишіть схему пошуку значення в дереві.

9. З чого починається вставка вузла в дереві?

10. Як можна обійтися без операції видалення?

11. Що таке обхід?

12. Які алгоритми обходу існують?

13. Яка послідовність дій при симетричному обході.

Лабораторна робота № 8. Хеш-таблиці

Мета роботи: освоєння навичок роботи з хеш-таблицями, релізація основних дій з ними: пошуку, запису та читання елементу.

КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Хеш-таблицею називається структура даних, яка зберігає пари ключ-значення в форматі, який забезпечує дуже швидку вставку, пошук і видалення.

Для користувача хеш-таблиці звернення до даних відбувається практично миттєво. Все робиться настільки швидко, що комп’ютерні програми часто використовують хеш-таблиці при необхідності зробити вибірку з десятків тисяч елементів менш ніж за секунду. Хеш-таблиці за швидкістю значно перевершують дерева.

У хеш-таблиць також є свої недоліки. Вони реалізуються на базі масивів, а масиви важко розширити після створення. У деяких різновидів хеш-таблиць швидкодія катастрофічно падає при заповненні таблиці, тому програміст повинен досить точно уявляти, скільки елементів даних буде зберігатися в таблиці.

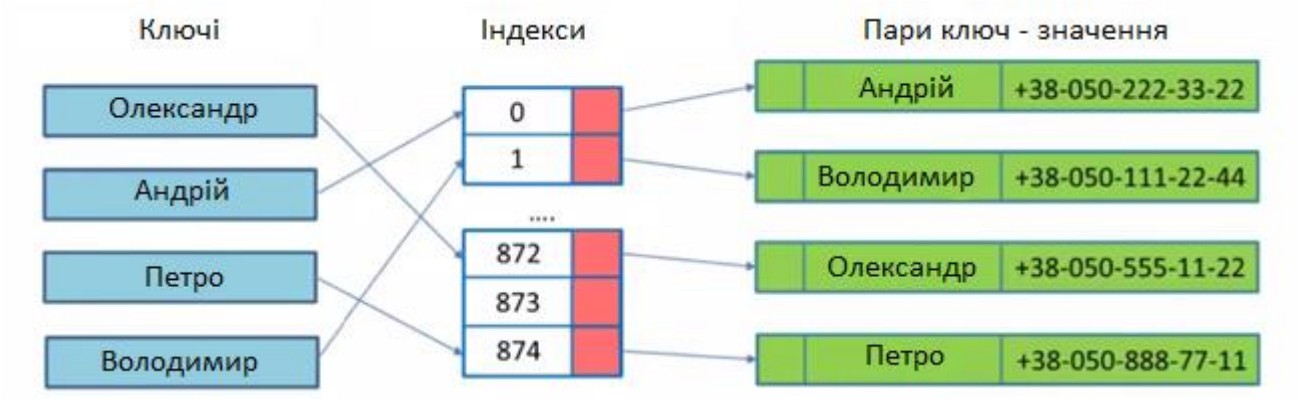


Рис. 21. Приклад хеш-таблиці

Крім того, при роботі з хеш-таблицями не існує зручного способу перебору елементів у певному порядку (скажімо, від менших до більших). Якщо вам необхідна така можливість, пошукайте іншу структуру даних.

Але якщо вам не потрібно перебирати елементи в певному порядку, а розмір бази даних можна спрогнозувати заздалегідь, хеш-таблиці не мають собі рівних за швидкістю і зручністю.

8.1 Хешування

Хешування — перетворення за певним алгоритмом вхідного масиву даних довільної довжини у вихідний бітовий рядок фіксованої довжини.

Хешування в найпростішому уявленні, це — спосіб перетворення будь-якої змінної/об’єкта в унікальний код після застосування будь-якої формули/алгоритму до їх властивостей. Справжня функція хешування повинна слідувати наступному правилу: хеш-функція повинна повертати однаковий хеш-код кожного разу, коли вона застосована до однакових або рівних об’єктів. Іншими словами, два однакових об’єкта повинні повертати однакові хеш-коди.

Хеш-функції — функції, які перетворюють число з більшого діапазону в число з меншого діапазону. Менший діапазон відповідає індексам масиву. Масив, в який вставляються дані з використанням хеш-функції, називається хеш-таблицею.

Наступний приклад допоможе зрозуміти, як працює схема перетворення великого діапазону в малий. Припустимо, числа з діапазону від 0 до 199 (представ-

лені змінною `largeNumber`) повинні відображатися в діапазон від 0 до 9 (змінна `smallNumber`). Скорочений діапазон складається з 10 чисел (змінна `smallRange`). Верхня межа «великого» діапазону для нас не істотна (аби числа не перевищували максимальне значення змінної). Вираз для виконання перетворення на мові Java виглядає так:

```
smallNumber = largeNumber % smallRange;
```

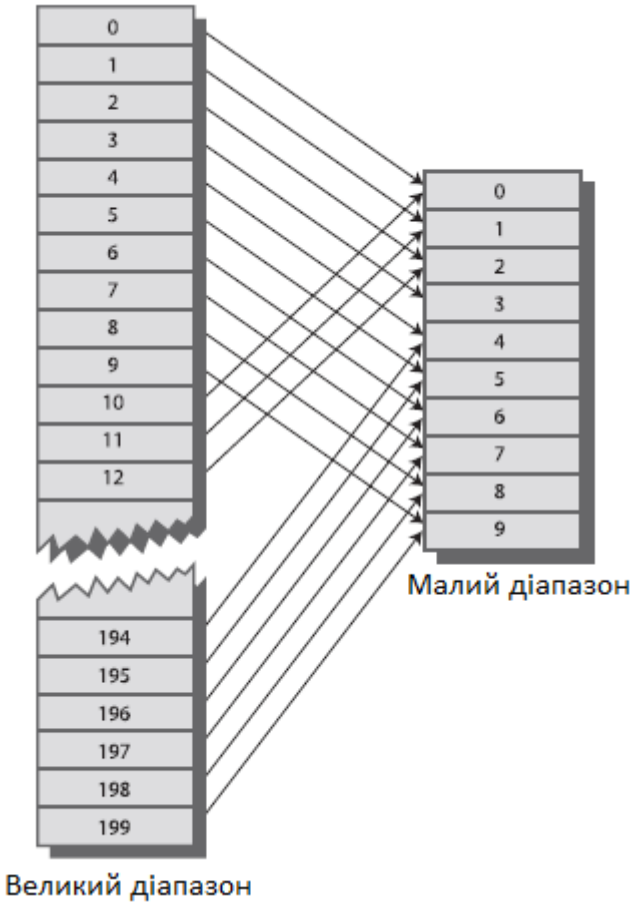


Рис. 22. Перетворення діапазону

Залишки від ділення будь-якого числа на 10 завжди лежать в діапазоні від 0 до 9; наприклад, результат виразу 13 % 10 дорівнює 3, а результат 157 % 10 дорівнює 7.

Діапазон 0 ÷ 199 таким чином перетвориться в діапазон 0 ÷ 9, тобто відбувається стиснення з коефіцієнтом 20:1.

Аналогічний вираз може використовуватися для стиснення дуже великих чисел, які однозначно представляють кожне слово англійської мови, в індекси ма-

сиву:

```
arrayIndex = hugeNumber % arraySize;
```

8.2 Колізії

Колізія (Collision) — це збіг значень хеш-функції для двох різних ключів (рис. 23).

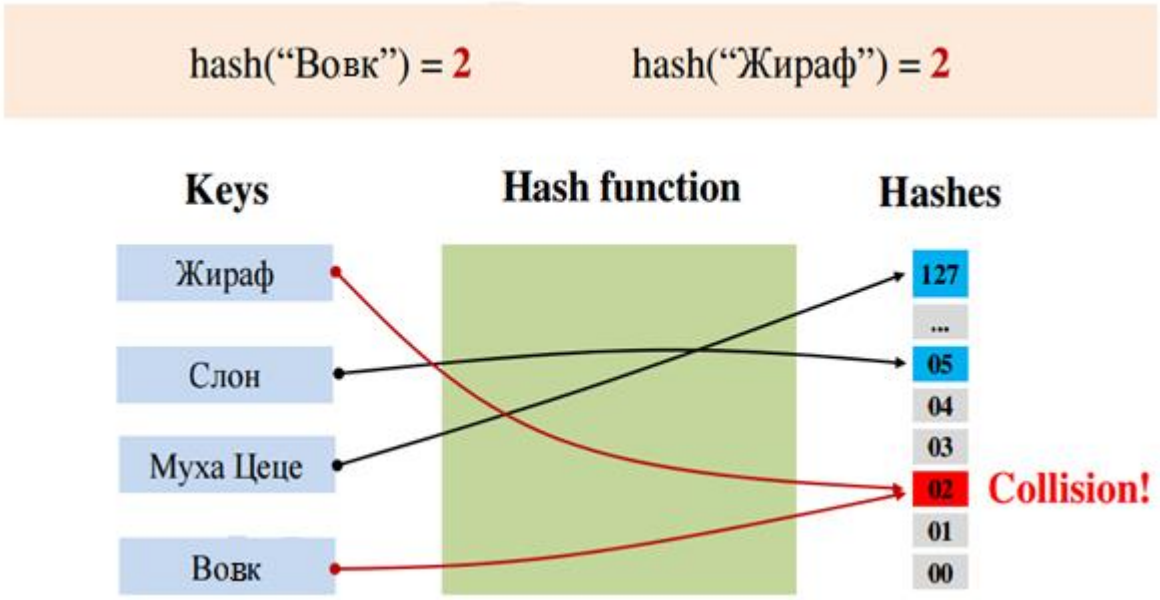


Рис. 23. Колізія в хеш-таблиці

Одне з можливих рішень при виникненні колізії полягає в систематизованому пошуку порожньої комірки і вставці нового елементу до неї (замість індексу, отриманого в результаті застосування хеш-функції). Таке рішення називається відкритою адресацією. Якщо слово «Вовк» хешується в індекс 02, а ця комірка вже зайнята словом «Жираф», можна спробувати вставити «Вовк» в іншу комірку, наприклад 01.

Для того щоб зрозуміти як працює хеш-таблиця слід розглянути алгоритми додавання, пошуку і видалення елемента в таблиці.

8.3 Метод `find()`

Програмний код `find()` виглядає так:

```
public DataItem find(int key) // Пошук елемента із заданим ключем
```

```
//(Метод припускає, що таблиця не заповнена)
{
int hashVal = hashFunc(key); // Хешування ключа
while(hashArray[hashVal] != null) // Поки не буде знайдена порожня
                                  // клітинка
{// Ключ знайдено?
if(hashArray[hashVal].getKey() == key)
return hashArray[hashVal]; // Так, повернути елемент
++hashVal; // Перехід до наступної комірки
hashVal %= arraySize; // При досягненні кінця таблиці
} // Відбувається повернення до початку
return null; // Елемент не знайдено
}
```

Метод `find()` спочатку викликає `hashFunc()`, щоб хешувати шуканий ключ для отримання індексу `hashVal`. Метод `hashFunc()` застосовує оператор % до шуканого ключа і розміру масиву.

Потім в умові `while` метод `find()` перевіряє, чи порожній осередок з даними індексом (`null`). Якщо комірка не порожня, метод перевіряє, чи містить вона шуканий ключ. Якщо перевірка дає позитивний результат, то `find()` повертає елемент, якщо ні — збільшує `hashVal` і повертається до початку циклу `while`, щоб перевірити, чи зайнята наступна комірка.

Перебираючи осередки масиву, `hashVal` зрештою дістанеться до кінця. Коли це відбудеться, перебір повинен повернутися до початку. Для цього можна було б скористатися командою `if`, яка обнуляє змінну `hashVal` при досягненні розміру масиву. Однак для досягнення того ж ефекту простіше застосувати оператор `%` до `hashVal` та розміру масиву.

8.4 Метод `insert()`

Наведений нижче метод `insert()` використовує схожий алгоритм для визначення позиції, в якій повинен знаходитися елемент даних, але замість конкретного елемента він шукає порожню комірку або видалений елемент (ключ дорівнює -1). Виявивши порожню комірку, `insert()` розміщує в ній новий елемент.

```
public void insert(DataItem item) // Вставка елемента даних
//(Метод припускає, що таблиця не заповнена)
{
int key = item.getKey(); // Отримання ключа
int hashVal = hashFunc(key); // Хешування ключа
// Поки не буде знайдена порожня клітинка або -1,
while(hashArray[hashVal]!= null && hashArray[hashVal].getKey()!= -1)
{
++hashVal; // Перехід до наступної комірки
hashVal %= arraySize; // При досягненні кінця таблиці
} // Відбувається повернення до початку
hashArray[hashVal] = item; // Вставка елемента
}
```

8.5 Метод `delete()`

Наступна реалізація методу `delete()` використовує код, аналогічний `find()`, для пошуку існуючого елемента. Виявивши елемент, `delete()` замінює його спеціальним елементом даних `nonItem` з наперед визначеним ключем -1.

```
public DataItem delete(int key) // Видалення елемента даних
{
int hashVal = hashFunc(key); // Хешування ключа
while(hashArray[hashVal] != null)
{// Ключ знайдено?
if(hashArray[hashVal].getKey() == key)
{
DataItem temp = hashArray[hashVal]; // Тимчасове збереження
hashArray[hashVal] = nonItem; // Видалення елемента
return temp; // Метод повертає елемент
}
++hashVal; // Перехід до наступної комірки
hashVal %= arraySize; // При досягненні кінця таблиці
} // Відбувається повернення до початку
return null; // Елемент не знайдено
}
```

8.6 Реалізація хеш-таблиці з лінійним пробіруванням на мові Java

Алгоритм лінійного пробірування послідовно шукає порожні клітинки. Якщо при спробі вставки елемента з'ясовується, що осередок 1 зайнятий, ми переходимо до осередку 2, потім до осередку 3 і т. д. Індекс послідовно збільшується до тих пір, поки не буде знайдена порожня клітинка. Процедура пошуку називається «лінійним пробіруванням», тому що вона заснована на лінійній перевірці послідовності осередків.

В NetBeans створимо проект «HashTable» з 3 класами: `DataItem`, `HashTable` і `HashTableApp`.

Лістинг програми:

```
public class DataItem
{//(Даних може бути і більше)
private int iData; // Дані(ключ)
public DataItem(int ii) // Конструктор
{ iData = ii; }
public int getKey()
{ return iData; }
} // Кінець класу DataItem

class HashTable
{
private DataItem[] hashArray; // Масив осередків хеш-таблиці
private int arraySize;
public HashTable(int size) // Конструктор
{
arraySize = size;
hashArray = new DataItem[arraySize];
nonItem = new DataItem(-1); // Ключ видаленого елемента -1
}
public void displayTable()
{
System.out.print("Table:");
for(int j = 0; j <arraySize; j ++)
{
```

```
if(hashArray[j] != null)
System.out.print(hashArray[j].getKey() + " ");
else
System.out.print("**");
}
System.out.println("");
}
public int hashFunc(int key)
{ return key % arraySize; // Хеш-функція }
public void insert(DataItem item) // Вставка елемента даних
{ ... }
public DataItem delete(int key) // Видалення елемента даних
{ ... }
public DataItem find(int key) // Пошук елемента із заданим ключем
{ ... }
} // Кінець класу HashTable

class HashTableApp
{
public static void main(String[] args) throws IOException
{
DataItem aDataItem;
int aKey, size, n, keysPerCell;
// Введення розмірів
System.out.print("Enter size of hash table:");
size = getInt();
System.out.print("Enter initial number of items:");
n = getInt();
keysPerCell = 10;
// Створення таблиці
HashTable theHashTable = new HashTable(size);
for(int j = 0; j <n; j++) // Вставка даних
{
aKey =(int)(java.lang.Math.random()*keysPerCell*size);
aDataItem = new DataItem(aKey);
theHashTable.insert(aDataItem);
```

```
}
while(true) // Взаємодія з користувачем
{
System.out.print("Enter first letter of show, insert, delete, or
find:");
char choice = getChar();
switch(choice)
{
case 's':
theHashTable.displayTable(); break;
case 'i':
System.out.print("Enter key value to insert:");
aKey = getInt();
aDataItem = new DataItem(aKey);
theHashTable.insert(aDataItem); break;
case 'd':
System.out.print("Enter key value to delete:");
aKey = getInt();
theHashTable.delete(aKey); break;
case 'f':
System.out.print("Enter key value to find:");
aKey = getInt();
aDataItem = theHashTable.find (aKey);
if (aDataItem! = null)
{
System.out.println("Found" + aKey);
}
else
System.out.println("Could not find" + aKey); break;
default:
System.out.print("Invalid entry \n");
} } }
public static String getString() throws IOException
{
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
```

```
String s = br.readLine();
return s;
}
public static char getChar() throws IOException
{
String s = getString();
return s.charAt(0);
}
public static int getInt() throws IOException
{
String s = getString();
return Integer.parseInt(s);
} } // Кінець класу HashTableApp
```

Метод `main()` класу `HashTableApp` реалізує користувальницький інтерфейс з можливостями виведення вмісту хеш-таблиці (команда s), вставки (i), видалення (d) або пошуку елемента (f).

Спочатку програма пропонує користувачеві ввести розмір хеш-таблиці та кількість елементів у ній. Хеш-таблиця може мати будь-який розмір, від десятка елементів до 10 000. Не використовуйте команду s для таблиць, що містять більше декількох сотень елементів; виведення вмісту займе дуже багато часу.

Змінна `keysPerCell` в методі `main()` задає відношення розмірів діапазону ключів і масиву. У лістингу їй присвоюється значення 10. Це означає, що для таблиці з розміром 20 значення ключів лежать в діапазоні від 0 до 200.

8.7 Згортка

Одна з розумних хеш-функцій заснована на розбитті ключа на групи цифр з подальшим підсумовуванням груп. Таке рішення гарантує, що хеш-код буде залежати від кожної цифри вихідних даних. Кількість цифр у групі повинна відповідати розміру масиву. Іншими словами, для масиву з 1000 елементів кожна група повинна складатися з трьох цифр.

Припустимо, що ви хочете хешувати 9-розрядні номери соціального страхування для лінійного пробірування. Якщо розмір масиву дорівнює 1000, число з 9 цифр ділиться на три групи з трьох цифр. Так, для коду 123456789 обчислюється

ключ 123 + 456 + 789 = 1368. Оператор % усікає отриману суму, щоб максимальне значення індексу становило 999. У нашому прикладі 1368 % 1000 = 368. Якби розмір масиву дорівнював 100, то ключ з 9 цифр довелося б розділити на чотири групи з двох цифр і одну групу з однієї цифри: 12 + 34 + 56 + 78 + 9 = 189, і 189 % 100 = 89.

Коли розмір масиву кратний 10, роботу цієї схеми легко зрозуміти. Але як було показано для інших хеш-функцій, оптимальний розмір масиву повинен бути простим числом. Реалізація цієї схеми залишається для самостійної роботи.

ПІДСУМКИ

- ✓ Хеш-таблиця створюється на базі масиву.
- ✓ Діапазон значень ключів зазвичай більше розміру масиву.
- ✓ Хеш-функція перетворює значення ключа в індекс масиву.
- ✓ Типовим прикладом бази даних, який ефективно реалізується у формі хеш-таблиці, є словник, наприклад, англійської мови.
- ✓ Хешування ключа в уже заповнену комірку масиву називається колізією.
- ✓ Існує дві основні схеми вирішення колізій: відкрита адресація і метод ланцюжків.
- ✓ При відкритій адресації елементи даних, які хешуються в заповнену комірку масиву, розміщуються в іншому осередку.
- ✓ При лінійному зондуванні зміщення завжди дорівнює одиниці. Таким чином, якщо обчислений хеш-функцією індекс масиву дорівнює x, то пробірування переходить до осередків x, x + 1, x + 2, x + 3 і т. д.
- ✓ Кількість кроків, необхідних для виявлення елемента, називається довжиною пробірування.
- ✓ При лінійному зондуванні в хеш-таблиці з’являються безперервні послідовності заповнених осередків, звані первинними групами. Наявність таких груп знижує ефективність хешування.

ЗАВДАННЯ

1. Виконати налагодження і тестування всіх операцій в проекті hashTable.
2. У проекті hashTable в методі `main()` організувати заповнення хеш-

таблиці вручну.

3. При лінійному зондуванні час безуспішного пошуку зв’язано з розміром груп. Знайдіть середній розмір групи для 30 елементів в 60 осередках (коефіцієнт заповнення 0,5). Ізольований осередок (тобто осередок, по обидві сторони від якого розташовуються вільні комірки) вважається групою розміру 1. Щоб знайти середнє значення, можна підрахувати кількість осередків у кожній групі, а потім розділити суму на кількість груп.

4. Напишіть хеш-функцію, що реалізує описаний метод. Програма повинна працювати з довільним розміром масиву і довжини ключа. Використовуйте лінійне пробірування.

5. Напишіть метод `rehash()` для програми hash.java. Метод повинен викликатися з `insert()` для переміщення всієї хеш-таблиці в масив приблизно вдвічі більшого розміру в ситуаціях, коли коефіцієнт заповнення перевищує 0,5.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке хеш-таблиця?
2. Що таке хешування і хеш-функція?
3. Які недоліки хеш-таблиць?
4. У яких випадках використовуються хеш-таблиці?
5. Наведіть приклад хеш-таблиці. Опишіть, як вона працює.
6. Як виконується перетворення великого діапазону масиву в малий?
7. Що таке колізія? Наведіть приклад.
8. Яке рішення колізій існує?
9. Опишіть метод `find()`.
10. Опишіть метод `insert()`.
11. Опишіть метод `delete()`.
12. Що таке алгоритм лінійного пробірування?

Л І Т Е Р А Т У Р А

1. Гудрич М. Т. Структуры данных и алгоритмы в Java / М. Т. Гудрич, Р. Тамассия; Пер. с англ. А. М. Чернухо. — Мн.: Новое знание, 2003. — 671 с.
2. Лафоре Р. Структуры данных и алгоритмы в Java. Классика Computers Science. 2-е изд. — СПб.: Питер, 2013. — 704 с.
3. Монахов В. В. Язык программирования Java и среда NetBeans / В. В. Монахов: 3-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2011. — 704 с.
4. Седжвик Р. Алгоритмы на Java / Седжвик Р., Уэйн К.: 4-е изд., Пер. с англ. — М.: ООО "И. Д. Вильямс", 2013. — 848 с.
5. Сьерра К. Изучаем Java / Сьерра К., Бейтс Б.: 2-е изд., Пер. с англ. — М.: Эксмо, 2012. — 720 с.
6. Шилдт Г. Java. Полное руководство, 8-е изд.: Пер. с англ. — М.: ООО "И.Д. Вильямс", 2012. — 1104 с.
7. Эккель Б. Философия Java / Б. Эккель: 4-е изд, Пер. с англ. — СПб.: Питер, 2009. — 640 с.
8. Oracle Technology Network for Java Developers [Електронний ресурс]. Режим доступу: **http://www.oracle.com/technetwork/java/index.html**
9. Welcome to NetBeans [Електронний ресурс]. Режим доступу: **https://netbeans.org/**