МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторних робіт студентів з дисципліни

"***ТЕОРІЯ АЛГОРИТМІВ***"

для студентів І курсу денної форми навчання

Напрям підготовки – комп'ютерні науки

Одеса 2014

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторних робіт студентів з дисципліни

“***Теорія алгоритмів***”

для студентів І курсу денної форми навчання

Напрям підготовки – комп’ютерні науки

Затверджено на засіданні
методичної комісії факультету
протокол № _________
від “___”_____________2014р.

Одеса 2014

Методичні вказівки до виконання лабораторних робіт студентів з дисципліни “Теорія алгоритмів”, для студентів І курсу денної форми навчання. Напрям підготовки – комп’ютерні науки / Укладачі: Шпінарева І.М.., к.ф.м.н, доц.– Одеса, ОДЕКУ, 2014. - 53 с.

ЗМІСТ

Передмова 5
Список літератури 7
Лабораторна робота № 1_ Аналіз алгоритмів та алгоритмічні стратегії 8
Лабораторна робота № 2_Прості та покращені алгоритми сортування 17
Лабораторна робота № 3 Рекурсія . Задача пошуку. Лінійний та бінарний пошук 27
Лабораторна робота № 4 Реалізація пошуку з використанням хеш-таблиць . 36
Індивідуальне завдання_ Фундаментальні алгоритми на графах і деревах 43
ДОДАТОК А Програмний код двійкового дерева пошуку 57
ДОДАТОК В Хеш-таблиця 64

Передмова

Методичні вказівки призначені для студентів I курсу денної форми навчання. Мета виконання лабораторних робіт – закріплення теоретичного лекційного матеріалу та придбання практичних навичок програмування мовою Java базових алгоритмів та їх подальше застосування для вирішенні різних задач, засвоєння основних результатів та методів теорії алгоритмів, теорії рекурсивних функцій та теорії складності обчислень.

Дисципліна «Теорії алгоритмів» є нормативною дисципліною у напрямі бакалаврської підготовки «Комп'ютерні науки».

Внаслідок вивчення даної дисципліни студенти повинні **знати** основні відомості з аналізу алгоритмів і алгоритмічних стратегій та фундаментальні алгоритми (алгоритми сортування, злиття та пошуку, комбінаторні алгоритми, рекурсивні алгоритми,і алгоритми на графах і деревах).

Вони повинні **вміти** застосовувати алгоритми сортування, злиття та пошуку, комбінаторні алгоритми, рекурсивні алгоритми та алгоритми на графах і деревах та програмування алгоритмів мовою Java.

Методичні вказівки містять рекомендації по вивченню розділів дисципліни, контрольні запитання та завдання. Всі лабораторні роботи підкріплені прикладами розв'язання типових задач на ПЕОМ.

Під час підготовки до лабораторної роботи студент повинен вивчити відповідний теоретичний матеріал за конспектом лекцій і літературою, що рекомендована викладачем, розібрати приклади розв'язання задач, наведених у даних методичних вказівках, а також відповісти на контрольні питання. Кожна лабораторна робота містить перелік тем, які повинні бути розглянуті та знання, які є необхідними для рішення поставленої задачі. Виконанню лабораторної роботи передує практичне заняття з відповідної теми. На практичних заняттях розглядаються алгоритми рішення задач для того, щоб під час лабораторної роботи студенти складали програму, користуючись розглянутими алгоритмами.

На початку лабораторної роботи викладач проводить співбесіду за результатами якої студент отримує, або не отримує допуск до виконання лабораторної роботи. Якщо студент не отримав допуску, він залишається на заняттях, але не виконує лабораторної роботи на комп'ютері. Замість цього він вивчає теоретичний матеріал за даною темою, щоб відповісти на питання викладача та отримати допуск до виконання роботи.

За кожну лабораторну роботу студент отримує оцінки: за виконання та за захист роботи. Максимальні бали з кожної лабораторної роботи встановлюються ведучим викладачем. На першому занятті студенти отримують графік контролюючих заходів з дисципліни: перелік контролюючих заходів, терміни виконання, бали за кожний вид робіт.

Список літератури

1. Б.Н.Иванов, Дискретная математика. Алгоритмы и программы. Москва, 2001.
2. Уоррен Г. (H.Warren) Алгоритмические трюки для программистов, М.: Издательский дом "Вильямс", 2003. - 288 с.: ил.
3. Котов В.М., Волков И.А., Лапо А.И., Информатика. Методы алгоритмизации., Мн.; 2000. - 300с.: ил.
4. Окулов С.М., Программирование в алгоритмах – М.: БИНОМ. Лаборатория знаний, 2002.–341 с: ил.
5. Глушаков С.В. Программирование на Java 2: Изд.2-е.- Харьков: Фолио, 2003. – 536 с. – (Учебный курс).
6. А.В. Картузов, Д.В. Николенко. Программируем на языке Java – СПб: Наука и техника, 2001. – 192 стр. с ил.
7. Любош Бруга. Java по-быстрому. Практический экспресс-курс – СПб: Наука и техника, 2006. – 384 с. :ил.
8. Аккуратов Е.Е. Знакомьтесь: Java. Самоучитель. – М.: Изд. Дом «Вильямс», 2006. – 256с.: ил.

Додаткова

9. Кормен Т., Лейзерсон Ч., Ривест Р.Л., Штайн К. Алгоритмы: построение и анализ, 2-е издание. : Пер. с англ. – М. : Издательский дом «Вильямс», 2005. – 1296 с.
10. Левитин А. В. Алгоритмы: введение в разработку и анализ. : Пер. с англ. – М. : Издательский дом «Вильямс», 2006. – 576 с.
11. Robert Lafore Data Structures & Algorithms in Java Second Edition–2003. – 801с.
12. Карпов Ю.Г., Трифонов П.В. Сложность алгоритмов и программ.
13. Сборник "Зимняя школа Харьков 2011".

Ресурси мережі Інтернет

14. Теорія алгоритмів. [Електронний ресурс]. – Режим доступу: http://talg.weebly.com/.

Лабораторна робота № 1

Тема: «Аналіз алгоритмів та алгоритмічні стратегії»

Мета роботи

Метою лабораторної роботи є отримання практичних навичок визначати часову складність алгоритму. Познайомитися з алгоритмами на НСД, НСК, алгоритм Євклида, алгоритм "Решето Ератосфена". Навчитися будувати алгоритми з мінімальною часовою складністю для вирішення поставлених задач.

Завдання до лабораторної роботи

Використовуючи алгоритми, розглянуті на практичному занятті, скласти програму розрахунку заданих величин та провести аналіз ефективності реалізованих алгоритмів.

Методичні вказівки

Лабораторна робота спирається на знання й уміння, отримані при вивченні наступних тем лекційного курсу:

- Поняття алгоритму, властивості. Алгоритмічні системи.
- Поняття складності обчислення. Функція складності обчислень (за часом).
- Класи складності.Опис класів P і NP. Приклади NP-повних задач. Проблема перебору (P = NP?). Застосування теорії NP-повноти для аналізу складності завдань.

Тому під час підготовки до лабораторної роботи рекомендується повторити зазначені розділи дисципліни.

Наведемо нижче декілька важливих визначень, які слід пам’ятати під час виконання лабораторної роботи.

Основними мірами обчислювальної складності алгоритмів є:

- часова складність, яка характеризує час, необхідний для виконання алгоритму на даній машині; цей час, як правило, визначається кількістю операцій, які потрібно виконати для реалізації алгоритму;

– ємнісна складність, яка характеризує пам'ять, необхідну для виконання алгоритму.

Часова та ємнісна складність тісно пов'язані між собою. Обидві є функціями від розміру вхідних даних.
Надалі обмежимося тільки аналізом часової складності.
Складність алгоритму описується функцією f(n), де n - розмір вхідних даних. Важливе теоретичне і практичне значення має класифікація алгоритмів, яка бере до увагу швидкість зростання цієї функції.

Асимптотичні позначення

– «О велике». Цей клас складається з функцій, що ростуть не швидше функції f .

Визначення. Кажуть, що f(n) = O (g(n)), якщо існує така константа c > 0, що для будь-якого n виконується нерівність: |f(n)| <= c |g(n)|.

Оскільки і розмір вхідних даних, і кількість операцій є величинами невід'ємними (а фактично - додатними), модулі можна опустити.

– «Омега велике». Клас функцій, що ростуть принаймні так само швидко як функція **f**.

Визначення. Кажуть, що f(n) = Ω (g(n)), якщо існує така константа c > 0, що для будь-якого n виконується нерівність: f(n) ≥ c g(n).

– Через $\Theta(g)$ («тета велике») ми позначаємо клас функцій, що ростуть з тією ж швидкістю що і функція **f**.

Визначення. Кажуть, що f(n) = Θ (g(n)), якщо існує такі константи $c_1 > 0$, $c_2 > 0$, що для будь-якого n виконується нерівність: c_1 g(n)≤ f(n) ≤c_2 g(n).

Виділяють такі основні класи алгоритмів:

- логарифмічні: f(n) = O ($\log_2 n$);
- лінійні: f(n) = O (n);
- поліноміальні: f(n) = O (n^m); тут m −натуральне число, більше від одиниці; при m = 1 алгоритм є лінійним;
- експоненційні: f(n) = O (a^n); a − натуральне число, більше від одиниці.

Для однієї й тієї ж задачі можуть існувати алгоритми різної складності.

Приклади аналізу алгоритмів

Проаналізуємо декілька простих алгоритмів, на основі яких розглянемо усі важливі етапи, які зазвичай виконуються при аналізі подібних алгоритмів.

Приклад 1. Розглянемо задачу пошуку найбільшого елементу в списку з n

чисел. Для простоти припустимо, що цей список реалізований у вигляді масиву. Нижче приведений псевдокод алгоритму.

Алгоритм MaxElement$(A[0 \ldots n-1])$
// Вхідні дані: масив дійсних чисел $(A[0 \ldots n-1])$
// Вихідні дані: повертається значення найбільшого елементу масиву A
$maxval \leftarrow A[0]$
$for\ i \leftarrow 1\ to\ n-1\ do$
    $if\ A[i] > maxval$
    $maxval \leftarrow A[i]$
$return\ maxval$

Очевидно, що в цьому алгоритмі розмір вхідних даних треба оцінювати по кількості елементів в масиві, тобто числом n . Операції, що виконуються найчастіше, знаходяться у внутрішньому циклі $\boldsymbol{for}$ алгоритму. Таких операцій дві: порівняння $A[i] > maxval$ та привласнення $maxval \leftarrow A[i]$. Яку з них вважати базовою? Оскільки порівняння виконується на кожному кроці циклу, а привласнення – ні, логічно вважати, що основною операцією алгоритму є операція привласнення. (Зверніть увагу, що для будь-якого масиву розміром n кількість операцій порівняння в даному алгоритмі постійна. Тому не требо окремо розглядати ефективність алгоритму для гіршого, середнього і кращого випадків).

Позначимо через $C(n)$ кількість виконуваних в алгоритмів операцій порівняння та спробуємо вивести формулу, що виражає їх залежність від розміру вхідних даних n . Відомо, що за один цикл у алгоритмі виконується одна операція порівняння. Цей процес повторюється для кожного значення змінної циклу i , яке змінюється від 1 до $n-1$ включно. Тому для $C(n)$ отримаємо наступну суму:

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1\ (O(n).$$

Приклад 2. Розглянемо задачу знаходження максимального елемента в матриці

Цю задачу можна вирішити за допомогою приведеного нижче нескладного алгоритму.

Алгоритм MaxElements$(A[0 \dots n-1][0 \dots n-1])$

// Вхідні дані: масив дійсних чисел $(A[0 \dots n-1][0 \dots n-1])$

// Вихідні дані: повертається значення **max**

```
Max=A[0][0];
for (i = 0 ; i <  n ; i + +)
  for (j = 0 ; j <  n; j + +){
    if (Max < A[i][j])
      Max = A[i][j] ;}
```

У цьому алгоритмі, як і у попередньому, розмір вхідних даних цілком природно оцінювати по кількості елементів у масиві, тобто числом $n * n$. Оскільки в найглибше вкладеному внутрішньому циклі алгоритму виконується тільки одна операція порівняння двох елементів, її і вважатимемо основною операцією цього алгоритму.

Позначимо через $C_m(n)$ кількість виконуваних в алгоритмів операцій порівняння та спробуємо вивести формулу, що виражає їх залежність від розміру вхідних даних n^2. Змінна зовнішнього циклу i послідовно приймає значення від 0 до $n-1$. При цьому змінна циклу i послідовно набуває значень від 0 до $n-1$. Внутрішній цикл кожного разу повторюється наново при кожному виконанні зовнішнього циклу. При цьому при кожному повторі внутрішнього циклу у алгоритмі виконується одна операція порівняння.

Таким чином, отримаємо:

$$C_m = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 = n * n = n^2 = O(n^2)$$

Задача знаходження максимального елемента в матриці має складність $O(n^2)$.

Приклад 3. Розглянемо задачу перевірки єдиності елементів. Іншими словами, треба переконатися, що усі елементи масиву різні. Цю задачу можна вирішити за допомогою приведеного нижче нескладного алгоритму.

Алгоритм Elements$(A[0 \dots n-1])$

// Вхідні дані: масив дійсних чисел $(A[0 \dots n-1])$

// Вихідні дані: повертається значення **true**, якщо усі елементи масиву A

// різні, та **false**у іншому випадку.

```
boolean l=true;
for( i = 0 ; i < n - 2; i + +){
```

```
for(j = i + 1; j < n - 1; j + +){
  if(A[i] = A[j])  l=false;}}
return  l;
```

У цьому алгоритмі, як і у попередньому, розмір вхідних даних цілком природно оцінювати по кількості елементів у масиві, тобто числом n. Оскільки в найглибше вкладеному внутрішньому циклі алгоритму виконується тільки одна операція порівняння двох елементів, її і вважатимемо основною операцією цього алгоритму. Зверніть увагу, що кількість операцій порівняння буде залежати не тільки від загального числа n елементів в масиві, але і від того, чи є в масиві однакові елементи, і якщо є, то на яких позиціях вони розташовані. Обмежимося розглядом найгіршого випадку.

За визначенням найгірший випадок вхідних даних відповідає такому масиву елементів, при обробці якого кількість операцій порівнянь $C_w(n)$ буде максимальною серед усієї сукупності вхідних масивів розміром n .Після аналізу внутрішнього циклу алгоритму приходимо до висновку, що найгірший випадок вхідних даних (тобто коли цикл виконується від початку до кінця, а не завершується достроково) може виникнути при обробці масивів двох типів: а) в яких немає однакових елементів; б) в яких два однакові елементи знаходяться поруч и розташовані у самому кінці масиву. В подібних випадках при кожному повторі внутрішнього циклу у алгоритмі виконується одна операція порівняння. При цьому змінна циклу j послідовно набуває значень від $i+1$ до $n-1$. Внутрішній цикл кожного разу повторюється наново при кожному виконанні зовнішнього циклу. При цьому змінна зовнішнього циклу i послідовно приймає значення від 0 до $n-2$. Таким чином, отримаємо:

$$C_w = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] =$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} =$$

Зверніть увагу, що цей результат можна було легко передбачити: у розглянутому алгоритмі у найгіршому випадку для масиву, який складається з n елементів, потрібно порівняти між собою усі $\frac{(n-1)n}{2}$ різних пар елементів.

Приклад 4. Розглянемо алгоритм Решето Ератосфена. Решето Ератосфена – це алгоритм знаходження простих чисел до заданого числа n. У процесі виконання даного алгоритму поступово відсіваються складені числа, кратні простим, починаючи з 2.

Цю задачу можна вирішити за допомогою приведеного нижче нескладного алгоритму.

```
import java.util.*;
public class Sieve{

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("input number: ");
    int n = sc.nextInt();
    int a[]=new int[n+1];
for(int i=0; i<n; i++){
        a[i] = i;    }
    a[1]=0;
    for(int s=2; s<n; s++){
        if(a[s]!=0){
            for(int j=s*2; j<n; j+=s){
                a[j]=0;            }
        }
    }
    for(int i=0; i<n; i++){
        if(a[i]!=0){
         System.out.print(a[i]+" ");
        }
    }
    }
}
```

Оцінимо складність алгоритму. Перше викреслення вимагає n / 2 дій, друге– n / 3, третє – n / 5 і т. д. За формулою Мертенса

$$\sum_{p\le n}\frac{n}{p} = n\sum_{p\le n}\frac{1}{p} \approx n\left(\frac{1}{2}+\sum_{k=2}^{\frac{n}{\ln n}}\left[\frac{1}{k\ln k}\right)\approx\right] n\left(\frac{1}{2}+\int_{2}^{\frac{n}{\ln n}}\left[\frac{1}{k\ln k}\right) =\right] n\ln\ln n$$

.

Так що для решета Ератосфена буде потрібно O (n log log n) операцій. Споживання пам'яті ж складе O (n).

План аналізу ефективності алгоритмів:

1. оберіть параметр (чи параметри), по якому буде оцінюватися розмір вхідних даних алгоритму;
2. визначите основну операцію алгоритму (як правило, вона знаходиться в найглибше вкладеному внутрішньому циклі алгоритму);
3. перевірте, чи залежить кількість виконуваних операцій тільки від розміру вхідних даних. Якщо воно залежить і від інших чинників, розгляньте при необхідності, як змінюється ефективність алгоритму для найгіршого, середнього та найкращого випадків;
4. запишіть суму, що виражає кількість виконуваних основних операцій алгоритму;
5. використовуючи стандартні формули і правила підсумовування, спростіть отриману формулу для кількості основних операцій алгоритму. Якщо це неможливо, визначте хоча б порядок зростання.

Контрольні питання

1. У чому полягає задача аналізу алгоритмів?
2. Які два види ефективності розглядаються у даному курсі?
3. У чому вимірюється час роботи алгоритму?
4. Як відбувається оцінка вхідних даних?
5. Які існують випадки вхідних даних?
6. Що таке порядок зростання алгоритмів? Які класи зростання ви знаєте?
7. У чому полягає асимптотичний аналіз алгоритмів?
8. Які асимптотичні позначення ви знаєте?

Завдання до лабораторної роботи

1. Побудувати алгоритм множення двох натуральних чисел без використання операції множення.

2. Побудувати алгоритм знаходження n-го простого числа.

3. Побудувати алгоритм знаходження біноміального коефіцієнта Cn^{k} для цілих n та k, де $0 \leq k \leq n$. Використайте наступні співвідношення:

$C_n^{0} = C_n^{n} = 1, C_{n+1}^{k+1} = C_n^{k+1} + C_n^{k}$.

4. Побудувати алгоритм знаходження всіх натуральних розв'язків нерівності $x^2 + y^2 < n$ для заданого натурального числа n.

5. Побудувати алгоритм, який для введеного цілого числа R знайде кількість точок з цілочисельними координатами, які лежать всередині кулі з радіусом R і центром в початку координат.

6. Побудувати алгоритм, який для заданого цілого числа x шукатиме найбільш близикий до $\sqrt{x}$ дріб n/m , де $m<100$.

7. Побудувати алгоритм для знаходження кількості щасливих білетів із шестизначними номерами. Білет вважається щасливим, якщо сума перших трьох цифр дорівнює сумі трьох останніх.

8. Побудувати алгоритм знаходження суми $1/0! + 1/1! + 1/2! + \ldots + 1/n!$, складність якого була б лінійною.

9. Є 25 золотих монет. Всі вони мають однакову вагу, за винятком однієї монети з дефектом, яка важить менше інших. Розробіть алгоритм, що визначає дефект за три зважування. Яка максимальна кількість монет, для яких можна визначити монету з дефектом не більш ніж за три зважування на вагах з чашками?

10. Три місіонери і три канібали знаходяться на одному березі річки; човен може перевезти двох чоловік. Всі вони хочуть перебратися через річку. Не можна допустити, щоб на одному березі знаходилася група місіонерів з численнішою групою канібалів. Розробіть процедуру, що дозволяє всім шістьом перебратися через річку.

11. Побудувати алгоритм, який здійснює обхід шахівниці конем. При чому кінь ніколи не буває двічі на одній клітці.

12. Побудувати алгоритм знаходження найбільшої спільної підпослідовності двох послідовностей.

13. Побудувати алгоритм, який набиратиме суму N копійок з набору монет 1, 2, 5,10, 25 та 50 копійок.

Прилади, устаткування та інструменти

Для виконання лабораторної роботи використовується ПЕОМ з установленим пакетом Sun Microsystems JDK 1.5 і вище та інтегрованим середовищем розробки BlueJ або Eclipse. Для написання програми на Java може бути використаний будь-який текстовий редактор, наприклад, Notepad, WordPad в MS Windows і ін.

Правила техніки безпеки та охорони праці

Правила техніки безпеки при виконанні лабораторної роботи регламентуються «Правилами техніки безпеки при роботі в комп'ютерній лабораторії».

Порядок проведення лабораторної роботи

Для виконання роботи кожен студент повинен:

1. Відповісти на контрольні питання та пройти усне опитування за теоретичним матеріалом лабораторної роботи;
2. Пройти інструктаж за правилами охорони праці;
3. Отримати варіант завдання у викладача;
4. Скласти алгоритм розв’язання задачі;
5. Записати код програми на комп’ютері;
6. Відкомпілювати програму та виправити всі помилки;
7. Запустити програму на виконання;
8. Отримати результати роботи програми і показати їх викладачу;
9. Підготувати і захистити звіт до лабораторної роботи.

Оформлення і захист звіту

Підготовлений до захисту звіт до лабораторної роботи повинен містити:

1. титульний лист, де вказані номер і назва лабораторної роботи, відомості про виконавця;
2. номер варіанта роботи та текст завдання;
3. відповіді на контрольні запитання до лабораторної роботи;
4. текст програми алгоритмічною мовою Java;
5. лістинг результатів виконання програми.

Лабораторна робота № 2

Тема: «Прості та покращені алгоритми сортування»

Мета роботи

Метою лабораторної роботи є отримання практичних навичок програмування алгоритмів сортування. Загальні принципи побудови алгоритмів сортування. Вивчення рекурсивні алгоритми сортування: пірамідальне сортування, сортування злиттям, швидке сортування. Навчитися визначати часову складність алгоритму сортування.

Завдання до лабораторної роботи

Використовуючи алгоритми, розглянуті на практичному занятті, скласти програму розрахунку заданих величин.

Методичні вказівки

Лабораторна робота спирається на знання й уміння, отримані при вивченні наступних тем лекційного курсу:

– Прості алгоритми сортування: обмін, вибір, вставка.
– Покращені алгоритми сортування –сортування Шелла, сортування Хоара (швидке сортування), сортування злиттям.
– Аналіз обчислювальної складності алгоритмів сортування

Тому під час підготовки до лабораторної роботи рекомендується повторити зазначені розділи дисципліни.

Розглянемо фрагменти програм, де використовуються алгоритми сортувань масивів:

Бульбашкова сортування

```
public static void bubbleSort(int[] data) {
 int n = data.length;
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < n-i; j++) {
            if (data[j] > data [j+1]) {
                int tmp = data[j]; data[j] = data[j+1];
     data[j+1] = tmp;
```

```
            }        }
    }}
```

Сортування двійковими вставками

```
public static void binInsertSort(int[] data) {
    int n = data.length;      // Длина массива
    for (int i = 1; i < n; i++) {
        int c = data[i];      // Вставляемое значение
        // Организация поиска места для вставки значения c
        int low = 0, high = i;
 // Inv : (low <= high) && место для c - внутри data[low:high]
        while (low < high) {
            int m = (low+high) >> 1;
            // low <= m < high
            if (data[m] < c) low = m+1; else high = m;
        }
        // Найдено место вставки - low
        // Сдвигаем элементы в сторону больших индексов.
        for (int j = i-1; j >= low; j--) {
            data[j+1] = data[j];
        }
        // Заносим значение на найденное место
        data[low] = c;
    }
}
```

Сортування Шелла

```
public static void ShellSort(int[] data) {
    int n = data.length;    // Длина массива
    int step = n;           // Шаг поисков и вставки
    int i, j;
    do {
        // Вычисляем новый шаг
        step = step / 3 + 1;
/ / Виробляємо сортування простими вставками з заданим кроком
        for (i = step; i < n; i++) {
            int c = data[i];
            for (j = i-step; j >= 0 && data[j] > c; j -= step) {
                data[j+step] = data[j];
            }
            data[j+step] = c;
        }
```

```
} while (step != 1);}
```

Сортування злиттям

Сортування злиттям сортує заданий масив $A[0 \dots n-1]$ шляхом розділення його на дві половини $A\left[0 \dots \left[\frac{n}{2}\right]-1\right]$ і $A\left[\left[\frac{n}{2}\right] \dots n-1\right]$, рекурсивного сортування кожної половини і злиття двох відсортованих половин в один відсортований масив

Основний недолік сортування злиттям – необхідність додаткової пам`яті, кількість якої лінійна пропорційно розміру вхідних даних. Сортування злиттям можливе і без залучення додаткової пам`яті, але така економія пам`яті істотно її ускладнює і дає в результаті значно більший постійний множник у формулі для часу роботи.

Сортування методом злиття в асимптотичній межі поводиться як $\Theta(n \lg n)$, що краще, ніж $\Theta(n^2)$, але процедура злиття, яка використовується в цьому алгоритмі, не працює без додаткової пам`яті.

//Алгоритм злиття впорядкованих масивів

```
public static int[] merge(int[] a, int[] b) {
    int na = a.length,
        nb = b.length,
        nc;
    int[] c = new int[nc = na + nb];
    int ia = 0,
        ib = 0,
        ic = 0;
    while (ia < na && ib < nb) {
        if (a[ia] < b[ib])
            c[ic++] = a[ia++];
        else
            c[ic++] = b[ib++];
    }
    while (ia < na) c[ic++] = a[ia++];
    while (ib < nb) c[ic++] = b[ib++];
    return c;
}
```

Пірамідальна сортування

```
public static void heapSort(int[] data) {
```

```
        int n = data.length;   // Довжина масиву
        buildPyramid:          // Побудова піраміди
        for (int i = 1; i < n; i++) {
                int c = data[i];
              int p = i, q;
            do { q = p;
                 if ((p = (q-1) >> 1) >= 0 && data[p] < c)
data[q] = data[p];
                 else {
                     data[q] = c;
                     continue buildPyramid;
                 }
            } while (true);
        }

        meltPyramid:       // Поступове руйнування піраміди
        for (int i = n-1; i > 0; i--) {
            int c = data[i];
            data[i] = data[0];
            int q, p = 0;
            do { q = p;
                 p = (q << 1) | 1;
                 if (p >= i) { // Вышли за границу пирамиды
                     data[q] = c;
                     continue meltPyramid;
                 }
                 if (p < i-1 && data[p+1] > data[p]) p++;
                 if (data[p] > c) data[q] = data[p];
                 else {
                     data[q] = c;
                     continue meltPyramid;
                 }
            } while (true);
        }
    }
```

Швидке сортування (Хоара)

Швидке сортування– важливий алгоритм сортування, ґрунтований на методі декомпозиції. На відміну від сортування злиттям, яке розділяє елементи масиву відповідно до їх положення в масиві, швидке сортування розділяє елементи масиву відповідно до їх значень. Конкретно, воно виконує

$A[0 \dots n -$

перестановку елементів цього масиву для отримання розбиття, коли усі елементи до деякої позиції s не перевищують елементу $A[s]$, а елементи після позиції s не менше $A[s]$. В силу важливості ролі $A[s]$ елементу він називається опорним

Алгоритм швидкого сортування сортує n чисел «на місці», але час його роботи в найгіршому випадку рівний $\Theta(n^2)$. Проте, в середньому цей алгоритм виконується за час $\Theta(n \lg n)$ і на практиці по продуктивності перевершує алгоритм пірамідального сортування

```
public static void quickSort(int[] data) {
    quickSort(data, 0, data.length-1);
}
private static void quickSort(int[] data, int from, int to) {
    if (to-from < 50)
// Небольшие фрагменты быстрее сортировать методом простых
вставок
        simpleSort(data, from, to);
    else {
        int c = data[from];    // Выбираем некоторый элемент
/ Распределяем элементы массива на значения меньшие и большие c.
        int low = from, high = to+1;
        // Inv: (low <= high) && data[from:(low-1)] <= c &&
data[high:to] >= c
        while (low < high) {
            while (low < high && data[--high] >= c) ;
            data[low] = data[high];
            while (low < high && data[++low] <= c) ;
            data[high] = data[low];
        }
        // Вставляем элемент на свое место
        data[low] = c;
  // Независимо сортируем верхнюю и нижнюю половины массива
        quickSort(data, from, low-1);
        quickSort(data, low+1, to);
    }}
```

Цифрове сортування

```
public static void digitSort(int[] data) {
    int n = data.length;
    int[] data2 = new int[n];
    for (int step = 0; step < 8; step++) {
 // step - номер "цифры"
 // Сортировка "подсчетом" по цифре с заданным номером
        countSort(data, step, data2);
        int[] temp = data; data = data2; data2 = temp;
    }
}
private static void countSort (int[] src, int nDig, int[] dest)
{
    int n = src.length;
    int[] count = new int[16];
    // 1. Подсчет
    nDig <<= 2;
    for (int i = 0; i < n; i++) {
        count[(src[i] & (0xF << nDig)) >> nDig]++;
    }
    // 2. Суммирование
    for (int i = 1; i < 16; i++) {
        count[i] += count[i-1];
    }
    // 3. Расстановка
    for (int i = n-1; i >= 0; i--)
 {
   dest[--count[(src[i] & (0xF << nDig)) >> nDig]] = src[i];
  }
}
```

Приклад програми

Дан масив data = {2,9,6,4,8,3,9,3,6,8}. Виконаємо сортування масиву методом Хоара.

Нижче представлений код програми.

```
import java.util.*;
public class sort {
// Метод простих вставок
     public static void simpleSort(int[] data,int from, int to) {
             int i, j;
            for (i = from; i < to+1; i++) {
                  int c = data[i];
               for (j = i-1; j >= from && data[j] > c; j--) {
                      data[j+1] = data[j];
```

```
                }
                data[j+1] = c;
            }
        }
// Метод швидкого сортування
    public static void quickSort(int[] data) {
        quickSort(data, 0, data.length-1);
    }

    private static void quickSort(int[] data, int from, int to)
{
        if (to-from < 50)
// Невеликі фрагменти швидше сортувати методом простих вставок
      simpleSort(data, from, to);
        else {
            int c = data[from];    // Вибираємо деякий елемент
/ / Розподіляємо елементи масиву на значення менші і більші с.
            int low = from, high = to+1;
//Inv: low <= high)&&data[from:(low-1)] <= c&&data[high:to] >= c
            while (low < high) {
                while (low < high && data[--high] >= c) ;
                data[low] = data[high];
                while (low < high && data[++low] <= c) ;
                data[high] = data[low];
            }
            // Вставляємо елемент на своє місце.
            data[low] = c;
/ / Незалежно сортуємо верхню і нижню половини масиву.
            quickSort(data, from, low-1);
            quickSort(data, low+1, to);
        }

    }

public static void main(String[]args){
    int n=10;
    int data[]={2,9,6,4,8,3,10,5,7,1};
    for (int i = 0; i < n; i++)
        System.out.print(data[i]+" ");
    quickSort(data);
    System.out.println();
    for (int i = 0; i < n; i++)
        System.out.print(data[i]+" ");
}
}
```

В результаті виконання лабораторної роботи студент повинен порівняти алгоритми сортування. Наприклад, за результатами експериментів з випадковим масивом обчислити кількість перестановок елементів (таб.2.1)

Таблиця 2.1 – Кількість перестановок елементів (

	n = 25	n = 1000	n = 100000
Сортування Шелла	50	7700	2 100 000
Сортування простими вставками	150	240 000	2.5 млрд.

Контрольні питання

1. У чому полягає задача сортування?
2. Визначте групи алгоритмів сортування.
3. Опишіть сутність алгоритмів групи елементарного сортування (вибору, вставки, бульбашки, Шелла).
4. Опишіть сутність групи алгоритмів порозрядного сортування, їх особливості, приклад.
5. У чому полягає ідея алгоритму сортування вставками?
6. У чому полягає ідея алгоритму сортування вибором?
7. У чому полягає ідея алгоритму сортування злиттям?
8. У чому полягає ідея алгоритму швидкого сортування?
9. У чому полягає ідея алгоритму пірамідальне сортування?

Завдання до лабораторної роботи

Згенеруйте одновимірний масив з n елементів, де n вводиться з клавіатури. n може приймати значення 100, 1000, 10000.

Необхідно впорядкувати даний масив двома способами і порівняти тимчасову складність даних алгоритмів сортування.

Порахуйте кількість порівнянь і перестановок.

Варіанти алгоритмів задані в таблиці 2.2.

Таблиця 2.2 – Варіанти завдань

№ варіанту	Алгоритм1	Алгоритм2
1	бульбашкове сортування	цифрове сортування
2	сортування простими вставками	швидке сортування (Хоара)
3	сортування двійковими вставками	пірамідальне сортування
4	сортування простими вставками	алгоритм злиття
5	бульбашкове сортування	швидке сортування (Хоара)
6	сортування Шелла	пірамідальне сортування
7	сортування двійковими вставками	цифрове сортування
8	швидке сортування (Хоара)	сортування Шелла
9	бульбашкове сортування	алгоритм злиття
10	сортування Шелла	цифрове сортування
11	сортування двійковими вставками	швидке сортування (Хоара)
12	сортування простими вставками	пірамідальне сортування
13	бульбашкове сортування	сортування Шелла

Прилади, устаткування та інструменти

Для виконання лабораторної роботи використовується ПЕОМ з установленим пакетом Sun Microsystems JDK 1.5 і вище та інтегрованим середовищем розробки BlueJ або Eclipse. Для написання програми на Java може бути використаний будь-який текстовий редактор, наприклад, Notepad, WordPad в MS Windows і ін.

Правила техніки безпеки та охорони праці

Правила техніки безпеки при виконанні лабораторної роботи регламентуються «Правилами техніки безпеки при роботі в комп'ютерній лабораторії».

Порядок проведення лабораторної роботи

Для виконання роботи кожен студент повинен:

1. Відповісти на контрольні питання та пройти усне опитування за теоретичним матеріалом лабораторної роботи;
2. Пройти інструктаж за правилами охорони праці;
3. Отримати варіант завдання у викладача;
4. Скласти алгоритм розв’язання задачі;
5. Записати код програми на комп’ютері;
6. Відкомпілювати програму та виправити всі помилки;
7. Запустити програму на виконання;
8. Отримати результати роботи програми і показати їх викладачу;
9. Підготувати і захистити звіт до лабораторної роботи.

Оформлення і захист звіту

Підготовлений до захисту звіт до лабораторної роботи повинен містити:

1. титульний лист, де вказані номер і назва лабораторної роботи, відомості про виконавця;
2. номер варіанта роботи та текст завдання;
3. відповіді на контрольні запитання до лабораторної роботи;
4. текст програми алгоритмічною мовою Java;
5. лістинг результатів виконання програми.

Лабораторна робота № 3

Тема: «Рекурсія . Задача пошуку. Лінійний та бінарний пошук»

Мета роботи

Отримання практичних навичок програмування задач лінійного пошуку у масиві та бінарне дерево пошуку. Вивчення властивості бінарного дерева пошуку. Отримання практичних навичок находження наступного та попереднього елементів бінарного дерева пошуку. Вивчення рекурсивні алгоритми вставки, пошуку та видалення у бінарному дереві пошуку.

Завдання до лабораторної роботи

Побудувати бінарне дерево пошуку. І виконати операції :

- вставка і видалення елементів;
- пошук елемента по ключу;
- пошук мінімального елементу;
- пошук максимального елементу.

Методичні вказівки

Лабораторна робота спирається на знання й уміння, отримані при вивченні наступних тем лекційного курсу:

- Алгоритми лінійного та бінарного пошуку.
- Пошук у бінарних деревах.
- Рекурсія. Особливості рекурсивних програм.

Тому під час підготовки до лабораторної роботи рекомендується повторити зазначені розділи дисципліни.

Наведемо нижче декілька важливих визначень, які слід пам’ятати під час виконання лабораторної роботи та перелік важливих методів, що дозволяють полегшити обробку дерева і можуть бути використані при створенні програми лабораторної роботи.

У алгоритмах пошуку нас цікавить процес перегляду списку у пошуках деякого конкретного елементу, що називається цільовим.

Алгоритм лінійного пошуку послідовно переглядає по одному елементу списку, починаючи з першого, до тих пір, поки не знайде цільовий елемент. Очевидно, що чим далі в списку знаходиться конкретне значення ключа, тим більше часу піде на його пошук. Це слід пам'ятати при аналізі алгоритму.

У алгоритмі послідовного пошуку два найгірші випадки. У першому випадку цільовий елемент стоїть в списку останнім. У другому його зовсім немає в списку. Припустимо, що усі ключові значення в списку унікальні, і тому якщо збіг стався в останньому записі, то усі попередні порівняння були невдалими. Проте алгоритм проробляє усі ці порівняння доки не дійде до останнього елементу. В результаті буде виконано N порівнянь, де N – число елементів в списку.

Бінарне дерево пошуку – структура даних, яка підтримує багато операцій з динамічними множинами, включаючи пошук елементу, мінімального і максимального значення, попереднього і подальшого елементу, вставку і видалення.

Основні операції у бінарному дереві пошуку виконуються за час, пропорційний його висоті. Для повного бінарного дерева з n вузлами ці операції виконуються за час $\Theta(\lg n)$ у найгіршому випадку. Математичне очікування висоти побудованого випадковим чином бінарного дерева дорівнює $O(\lg n)$, так що усі основні операції над динамічною множиною в такому дереві виконуються в середньому за час $\Theta(\lg n)$.

Бінарне дерево пошуку може бути представлене за допомогою зв'язаної структури даних, в якій кожен вузол є об'єктом `Node`. На додаток до полів ключа і супутніх даних, кожен вузол містить поля `leftChild`, `rightChild` , які вказують на лівий, правий дочірні вузли . Якщо дочірній вузол відсутній, відповідне поле містить значення . Єдиний вузол, вказівник `root`– це кореневий вузол дерева.

```
class Node
{
public int iData; // data item (key)
public double dData; // data item
public Node leftChild; // this node’s left child
public Node rightChild; // this node’s right child
```

```
public void displayNode() // display ourself
{
System.out.print('{');
System.out.print(iData);
System.out.print(", ");
System.out.print(dData);
System.out.print("} ");
}
} // end class Node
```

Ключі у бінарному дереві пошуку зберігаються особливим чином, щоб у будь-який момент задовольняти наступній властивості бінарного дерева пошуку: якщо x – вузол бінарного дерева пошуку, а вузол y знаходиться в лівому піддереві x, то $key[y] \le key[x]$. Якщо вузол y знаходиться в правому піддереві x, то $key[x] \le key[y]$.

Вставка елементу. Алгоритм вставки елементу починає свою роботу в корені дерева і проходить по низхідному шляху, переміщаючись вліво або управо залежно від результату порівняння ключів, до тих пір, поки не наштовхнеться на незайняту позицію лівого або правого спадкоємця елементу.

```
public void insert(int id, double dd)
     {
     Node newNode = new Node(); // make new node
     newNode.iData = id; // insert data
     newNode.dData = dd;
     if(root==null) // no node in root
     root = newNode;
     else // root occupied
     {
     Node current = root; // start at root
     Node parent;
     while(true) // (exits internally)
     {
     parent = current;
     if(id < current.iData) // go left?
     {
     current = current.leftChild;
     if(current == null) // if end of the line,
     { // insert on left
     parent.leftChild = newNode;
```

```
return;
}
} //
else // or go right?
{
     current = current.rightChild;
     if(current == null) // if end of the line
     { // insert on right
     parent.rightChild = newNode;
     return;
     }
     } // end else go right
     } // end while
     } // end else not root
     } //
```

Пошук. Для пошуку вузла із заданим ключем у бінарному дереві пошуку використовується наступний алгоритм: для кожного вузла x на шляху вниз, його ключ $key[x]$ порівнюється з шуканим ключем k. Якщо ключі однакові, пошук завершується. Якщо k менше $key[x]$, пошук триває в лівому піддереві x; якщо більше – то пошук переходить в праве піддерево. Відвідані при рекурсивному пошуку вузли утворюють низхідний шлях від кореня дерева, так що час роботи цього алгоритму рівний $O(h)$, де h – висота дерева.

```
public Node find(int key) // find node with given key
     {
     Node current = root; // start at root
     while(current.iData != key)
     {
     if(key < current.iData)
     current = current.leftChild;
     else // or go right?
     current = current.rightChild;
     if(current == null)
     return null;   }
     return current;
     } // end find()
```

Видалення елементу. Алгоритм видалення вузла працює по-різному, залежно від того – чи є у елемента, що видаляється, дочірні вузли:

– Якщо у вузла, що видаляється, немає дочірніх елементів, то ми просто змінюємо його батьківський вузол, замінюючи в ньому вказівник на елемент, що видаляється, значенням NIL .

– Якщо у вузла, що видаляється, тільки один дочірній елемент, то ми видаляємо вузол, зв'язуючи батьківський елемент і дочірній елемент вузла, що видаляється.

– Якщо у вузла, що видаляється, два дочірні елементи, то знаходимо вузол, що йде за ним, у якого немає лівого дочірнього вузла, прибираємо його з позиції, де він знаходився раніше, шляхом створення нового зв'язку між його батьком і нащадком, і замінюємо ним вузол, що видаляється.

Час роботи алгоритму видалення у всіх трьох випадках, для бінарного дерева пошуку заввишки h складає $O(h)$.

```
public boolean delete(int key) // delete node with given key
     {
     Node current = root;
     Node parent = root;
     boolean isLeftChild = true;
     while(current.iData != key) // search for node
     {
     parent = current;
     if(key < current.iData) // go left?
     {
     isLeftChild = true;
     current = current.leftChild;
     }
     else // or go right?
     {
     isLeftChild = false;
     current = current.rightChild;
     }
     if(current == null) // end of the line,
     return false; // didn't find it
     } // end while
     if(current.leftChild==null &&
     current.rightChild==null)
     {
     if(current == root) // if root,
          root = null; // tree is empty
     else if(isLeftChild)
          parent.leftChild = null; // disconnect
```

```
else // from parent
     parent.rightChild = null;
     }
     else if(current.rightChild==null)
     if(current == root)
     root = current.leftChild;
     else if(isLeftChild)
     parent.leftChild = current.leftChild;
     else
     parent.rightChild = current.leftChild;
     else if(current.leftChild==null)
     if(current == root)
     root = current.rightChild;
     else if(isLeftChild)
     parent.leftChild = current.rightChild;
     else
     parent.rightChild = current.rightChild;
     else          {
     Node successor = getSuccessor(current);
     // connect parent of current to successor instead
     if(current == root)
     root = successor;
     else if(isLeftChild)
     parent.leftChild = successor;
     else
     parent.rightChild = successor;
     successor.leftChild = current.leftChild;
     } // end else two children
return true;
} // end delete
```

Код програми представлений в додатку А.

Контрольні питання

1. Що таке пошук?
2. Що називаються ключем пошуку?
3. Які відомі методи пошуку?
4. Який алгоритм пошуку є найбільш ефективним?
5. Чим відрізняються пошук в масиві від пошуку в списку?
6. У чому полягаю метод лінійного пошуку?
7. Що представляю собоя двійкове дерево?
8. У чому полягає вставка вузла в дерево?

9. У чому полягає видалення вузла з дерева?
10. Які особливості видалення елемента з деревовидної структури даних?
11. У чому полягає пошук в дереві?
12. Що таке «проходження дерева»? Які можливі варіанти проходження дерева?

Варіанти завдань

№	Вставка	Пошук	Видалення
1	41, 71, 25, 24, 60, 68, 26, 4, 77, 12, 52, 62, 21, 31, 30	68, 12, 30, min	41, 25, 60
2	47, 40, 22, 61, 95, 9, 93, 39, 43, 38, 59, 25, 60, 71, 32	38, 60, 32 , max	47, 22, 61
3	46, 26, 2, 79, 76, 99, 41, 37, 51, 6, 97, 35, 93, 10, 21	97, 10, 35, min	46, 99, 26
4	48, 52, 15, 54, 79, 56, 46, 73, 65, 94, 9, 85, 7, 33, 8	73, 85, 8 , max	48,79, 15
5	41, 27, 44, 13, 17, 37, 65, 33, 40, 84, 59, 69, 7, 60, 67	69, 33, 60 , min	41, 65, 27
6	53, 13, 66, 55, 63, 24, 77, 82, 44, 40, 18, 86, 15, 68, 30	82, 40, 30 , max	53, 66, 13
7	81, 42, 57, 87, 25, 52, 6, 32, 91	42, 91, 25, min	42, 91, 25, 81
8	44, 50, 11, 49, 23, 15, 87, 1, 27	11, 44, 49, max	11, 44, 49, 50
9	58, 78, 23, 47, 4, 80, 44, 56, 93	58, 23, 80, min	58, 23, 80, 4
10	54, 29, 97, 37, 55, 4, 34, 96, 98	29, 54, 37, max	29, 54, 37, 98
11	62, 81, 34, 4, 82, 41, 79, 96, 84	81, 34, 62, min	81, 34, 62, 96
12	56, 23, 36, 22, 87, 44, 25, 73, 11	23, 56, 36, max	23, 56, 36

Прилади, устаткування та інструменти

Для виконання лабораторної роботи використовується ПЕОМ з установленим пакетом Sun Microsystems JDK 1.5 і вище та інтегрованим середовищем розробки BlueJ або Eclipse. Для написання програми на Java може бути використаний будь-який текстовий редактор, наприклад, Notepad, WordPad в MS Windows і ін.

Правила техніки безпеки та охорони праці

Правила техніки безпеки при виконанні лабораторної роботи регламентуються «Правилами техніки безпеки при роботі в комп'ютерній лабораторії».

Порядок проведення лабораторної роботи

Для виконання роботи кожен студент повинен:

1. Відповісти на контрольні питання та пройти усне опитування за теоретичним матеріалом лабораторної роботи;
2. Пройти інструктаж за правилами охорони праці;
3. Запустити на комп'ютері інтегроване середовище розробки BlueJ;
4. Отримати варіант завдання у викладача;
5. Скласти алгоритм розв’язання задачі;
6. Записати код програми на комп’ютері;
7. Відкомпілювати програму та виправити всі помилки;
8. Запустити програму на виконання;
9. Отримати результати роботи програми і показати їх викладачу;
10. Підготувати і захистити звіт до лабораторної роботи.

Оформлення і захист звіту

Підготовлений до захисту звіт до лабораторної роботи повинен містити:

1. титульний лист, де вказані номер і назва лабораторної роботи, відомості про виконавця;
2. номер варіанта роботи та текст завдання;
3. відповіді на контрольні запитання до лабораторної роботи;
4. текст програми алгоритмічною мовою Java;
5. лістинг результатів виконання програми.

6.

Лабораторна робота № 4

Тема: «Реалізація пошуку з використанням хеш-таблиць»

Мета роботи

Отримання навичок роботи з процедурою хешування. Розв'язання колізій у хеш-таблицях. Розв'язання колізій за допомогою ланцюжків. Розв'язання колізій за допомогою відкритої адресації. Алгоритм пошуку ключа у хеш-таблицях. Алгоритм видалення з хеш-таблиці. Алгоритм пробування.

Завдання до лабораторної роботи

Використовуючи алгоритми, розглянуті на практичному занятті, скласти програму розрахунку заданих величин.

Методичні вказівки

Лабораторна робота спирається на знання й уміння, отримані при вивченні наступних тем лекційного курсу:

- Пошук у відсортованому масиві.
- Реалізація пошуку з використанням хеш-таблиць.

Тому під час підготовки до лабораторної роботи рекомендується повторити зазначені розділи дисципліни.

Наведемо нижче декілька важливих визначень, які слід пам'ятати під час виконання лабораторної роботи та перелік важливих методів, що дозволяють полегшити обробку хеш-таблиць і можуть бути використані при створенні програми лабораторної роботи.

Хеш-таблиця є ефективною структурою для реалізації словників. Хоча на пошук елементу в хеш-таблиці може в найгіршому випадку знадобитися стільки ж часу, що і в зв'язаному списку, а саме $\Theta(n)$, на практиці хешування

виключно ефективно. Враховуючи різні припущення, математичне очікування часу пошуку елементу в хеш-таблиці складає $O(1)$.

Для прискорення доступу до даних можна використовувати попереднє їх впорядкування у відповідності зі значеннями ключів. При цьому дані організуються у вигляді таблиці за допомогою хеш-функції h, яка використовується для „обчислення” адреси за значенням ключа.

Побудова хеш-функції методом ділення полягає у відображенні ключа k в один з елементів шляхом отримання залишку від ділення k на m, тобто хеш-функція має вид $h(k) = k \bmod m$, де m – розмір таблиці.

```
public int hashFunc(int key) // hash function
{
return key % arraySize;
}
```

Проте виникає утруднення, коли результати хешування двох різних елементів співпадають. Це називається колізією.

Для розв’язання колізій використовуються різноманітні методи, які в основному зводяться до методів „ланцюжків” і „відкритої адресації”.

Розв`язання колізій в хеш-таблицях за допомогою ланцюжків

При використанні методу ланцюжків усі елементи, хешовані в один і той же елемент, об'єднуються в зв'язаний список. Елемент i містить вказівник на заголовок списку усіх елементів, хеш-значення ключа яких дорівнює i; якщо таких елементів немає, елемент містить значення .

Пошук в хеш-таблиці з ланцюжками переповнення здійснюється наступним чином. Спочатку обчислюється адреса за значенням ключа. Потім здійснюється послідовний пошук в списку, який зв’язаний з обчисленим адресом.

```
public Link find(int key) // find link
{
int hashVal = hashFunc(key); // hash the key
Link theLink = hashArray[hashVal].find(key); // get link
return theLink; // return link
}
```

Час, необхідний для вставки елемента в найгіршому випадку дорівнює $O(1)$. Процедура вставки виконується дуже швидко, оскільки передбачається, що елемент, що вставляється, відсутній в таблиці. Час роботи пошуку в найгіршому випадку пропорційний довжині списку.

```
public void insert(Link theLink) // insert a link
{
int key = theLink.getKey();
int hashVal = hashFunc(key); // hash the key
hashArray[hashVal].insert(theLink); // insert at hashVal
}
```

Розв`язання колізій в хеш-таблицях за допомогою відкритої адресації

При використанні методу відкритої адресації усі елементи зберігаються безпосередньо в хеш-таблиці. Хеш-таблиця з прямою адресацією представляє собою узагальнення звичайного масиву. Можливість прямої індексації елементів звичайного масиву дозволяє доступ до довільної позиції в масиві за час O(1). При пошуку елементу систематично перевіряються елементи таблиці до тих пір, поки не буде знайдений шуканий елемент або доки не буде встановлено, що його немає в таблиці.

```
public DataItem find(int key) // find item with key
// (assumes table not full)
{
int hashVal = hashFunc1(key); // hash the key
int stepSize = hashFunc2(key); // get step size
while(hashArray[hashVal] != null) // until empty cell,
{ // is correct hashVal?
if(hashArray[hashVal].getKey() == key)
return hashArray[hashVal]; // yes, return item
hashVal += stepSize; // add the step
hashVal %= arraySize; // for wraparound
}
return null; // can't find item
}
```

Для обчислення послідовності досліджень колізії для відкритої адресації зазвичай використовуються три методи: лінійне дослідження, квадратичне дослідження і подвійне хешування:

- Метод лінійного дослідження використовує хеш-функцію

,

де i набуває значення від 0 до $m-1$ включно.

- Квадратичне дослідження використовує функцію виду

,

де h' – допоміжна хеш-функція, c_1 і $c_2 \neq 0$ – допоміжні константи, а i набуває значень від 0 до $m-1$ включно. Цей метод працює істотно краще за лінійне дослідження, але для того, щоб дослідження охоплювало усі елементи, потрібен вибір спеціальних значень c_1, c_2 і r.

– Подвійне хешування використовує хеш-функцією виду

,

де h_1 і h_2 – допоміжні хеш-функції.

Наприклад

```
public int hashFunc1(int key)
{
return key % arraySize;
}
public int hashFunc2(int key)
{
return 5 - key % 5;
}
```

Для виконання вставки при відкритій адресації відбувається послідовна перевірка або дослідження елементу хеш-таблиці до тих пір, поки не буде знайдений порожній елемент, в який поміщається ключ, що вставляється.

```
public void insert(int key, DataItem item)
// (assumes table not full)
{
int hashVal = hashFunc1(key); // hash the key
int stepSize = hashFunc2(key); // get step size
while(hashArray[hashVal] != null &&
hashArray[hashVal].getKey() != -1)
{
hashVal += stepSize; // add the step
hashVal %= arraySize; // for wraparound
}
hashArray[hashVal] = item; // insert item
} // end insert()
```

Процедура вилучення з таблиці.

```
public DataItem delete(int key) // delete a DataItem
{
int hashVal = hashFunc1(key); // hash the key
int stepSize = hashFunc2(key); // get step size
while(hashArray[hashVal] != null) // until empty cell,
{
if(hashArray[hashVal].getKey() == key)
{
DataItem temp = hashArray[hashVal]; // save item
hashArray[hashVal] = nonItem; // delete item
return temp; // return item
}
hashVal += stepSize; // add the step
hashVal %= arraySize; // for wraparound
}
return null; // can't find item
} // end delete()
```

Контрольні питання.

1. Що таке хеш-функція?
2. Яки переваги мають алгоритмі використовуючі хеширування?
3. Що таке колізії?
4. Як пов'язані хеш-функції та асоциативні масиви?
5. Яким чином можна зменьшити кількість колізій?

Варіанти завдань

№ варі-анту	Спосіб вирішення колізій	Хеш-функція 2
1	Простий список	
2	Впорядкований список	
3	За допомогою масиву (відкрита адресація)	Метод лінійного дослідження
4	За допомогою масиву (відкрита адресація)	Квадратичне дослідження
5	За допомогою масиву (відкрита адресація)	Подвійне хешування
6	Простий список	
7	Впорядкований список	
8	За допомогою масиву (відкрита адресація)	Метод лінійного

		дослідження
9	За допомогою масиву (відкрита адресація)	Квадратичне дослідження
10	За допомогою масиву (відкрита адресація)	Подвійне хешування

Прилади, устаткування та інструменти

Для виконання лабораторної роботи використовується ПЕОМ з установленим пакетом Sun Microsystems JDK 1.5 і вище та інтегрованим середовищем розробки BlueJ або Eclipse. Для написання програми на Java може бути використаний будь-який текстовий редактор, наприклад, Notepad, WordPad в MS Windows і ін.

Правила техніки безпеки та охорони праці

Правила техніки безпеки при виконанні лабораторної роботи регламентуються «Правилами техніки безпеки при роботі в комп'ютерній лабораторії».

Порядок проведення лабораторної роботи

Для виконання роботи кожен студент повинен:

1. Відповісти на контрольні питання та пройти усне опитування за теоретичним матеріалом лабораторної роботи;
2. Пройти інструктаж за правилами охорони праці;
3. Отримати варіант завдання у викладача;
4. Скласти алгоритм розв'язання задачі;
5. Записати код програми на комп'ютері;
6. Відкомпілювати програму та виправити всі помилки;
7. Запустити програму на виконання;
8. Отримати результати роботи програми і показати їх викладачу;
9. Підготувати і захистити звіт до лабораторної роботи.

Оформлення і захист звіту

Підготовлений до захисту звіт до лабораторної роботи повинен містити:

1. титульний лист, де вказані номер і назва лабораторної роботи, відомості про виконавця;
2. номер варіанта роботи та текст завдання;
3. відповіді на контрольні запитання до лабораторної роботи;

4. текст програми алгоритмічною мовою Java;
5. лістинг результатів виконання програми.

Індивідуальне завдання

Тема: «Фундаментальні алгоритми на графах і деревах»

Мета роботи

Метою лабораторної роботи є отримання практичних навичок програмування алгоритмів на графах.

Завдання до лабораторної роботи

Використовуючи алгоритми, розглянуті на практичному занятті, скласти програму розрахунку заданих величин.

Методичні вказівки

Індивідуальне робота та спирається на знання й уміння, отримані при вивченні наступних тем лекційного курсу:

– - Пошук найкоротших шляхів та оптимальних маршрутів у графах. Алгоритм Дейкстри.

Тому під час підготовки до роботи рекомендується повторити зазначені розділи дисципліни.

Наведемо нижче декілька важливих визначень, які слід пам’ятати під час виконання роботи

З формальної точки зору граф є впорядкованою парою $G = (V, E)$ множин, перша з яких складається з вершин або вузлів графа, а друга – з його ребер. Ребро зв`язує між собою дві вершини.Ребро графа, що зв`язує дві вершини A і B позначається як AB . В цьому випадку говорять, що A примикає до B або що ці дві вершини сусідні.

Граф може бути орієнтованим або ні. Ребра неорієнтованого графа (який часто називається просто графом) можна проходити в обох напрямах. В цьому випадку ребро -— це неврегульована пара вершин, його кінців. У орієнтованому графові, або орграфі, ребра є впорядкованими парами вершин: перша вершина – це початок ребра, а друга – його кінець.

Маршрут у графі – це послідовність вершин x_1, x_2,…, x_n така, що для кожного i=1,2,…,n−1 вершини x_i і x_{i+1} з'єднані ребром. Ці n−1 ребер називаються ребрами маршруту. Говорять, що маршрут проходить через них, а число n−1 називають довжиною маршруту

У зваженому графові або орграфі кожному ребру приписано число, що називається вагою ребра. При зображенні графа вага записується поряд з ребром. При роботі з орієнтованим графом ми вважаємо вагу ребра ціною проходу по ньому. Вартість шляху по зваженому графові дорівнює сумі вагів усіх ребер шляху. Найкоротший шлях у зваженому графові – це шлях з мінімальною вагою, навіть якщо число ребер в дорозі можна зменшити.

Інформацію про графи або орграфи можна зберігати двома способами: у вигляді матриці суміжності або у вигляді списку суміжних вершин.

Матриця суміжності забезпечує швидкий доступ до інформації про ребра графа, проте якщо в графі мало ребер, то ця матриця міститиме значно більше порожніх елементів, ніж заповнених. Довжинасписку суміжних вершин пропорційна числу ребер в графі, проте при користуванні списком час отримання інформації про ребро збільшується

Матриця суміжності A графа з числом вершин записується у вигляді двовимірного масиву розміром $N \times N$. У кожному елементі цього масиву записано значення 0 за винятком лише тих випадків, коли з вершини u у вершину V_i веде ребро, і тоді в елементі записується значення 1. Говорячи більш строго

$$A[i,j] = \begin{cases} 1, \text{якщо } v_i v_j \in E \\ 0, \text{якщо } v_i v_j \notin E \end{cases}, \text{для всіх } i \text{ та } j \text{ від } 1 \text{ до } N.$$

Приклад

Пусть дани графи неорієнтований (рис.5.1 а)та орієнтований (рис.5.1 б)

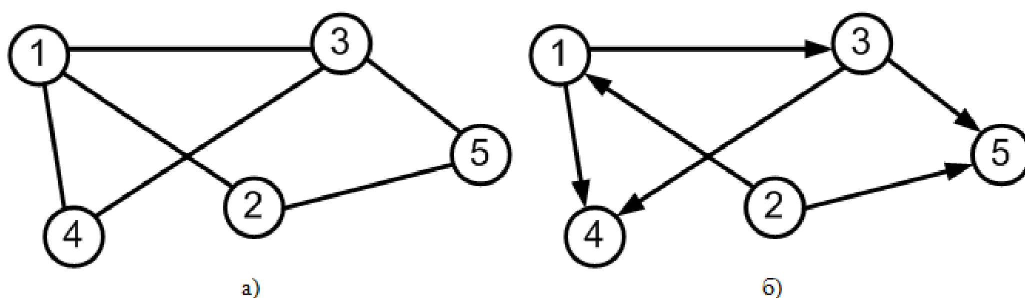


Рисунок 5.1 – Приклад а) неорієнтованого та б) орієнтованого графу

Матриці суміжності для відповідних графів:

	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	0	1
3	1	0	0	1	1
4	1	0	1	0	0
5	0	1	1	0	0

	1	2	3	4	5
1	0	0	1	1	0
2	1	0	0	0	1
3	0	0	0	1	1
4	0	0	0	0	0
5	0	0	0	0	0

Найбільш ефективний алгоритм пошуку найкоротших шляхів на графах є **алгоритм Дейкстра.** Нехай G = (V, E) – зважений орієнтований граф, $w(v_i, v_j)$ – вага дуги (v_i,v_j). Почавши з вершини а, знаходимо відстань від а до кожної із суміжних із нею вершин. Вибираємо вершину, відстань від якої до вершини а найменша; нехай це буде вершина v*. Далі знаходимо відстані від вершини а до кожної вершини суміжної з v* вздовж шляху, який проходить через вершину v*. Якщо для якоїсь із таких вершин ця відстань менша від поточної, то замінюємо нею поточну відстань. Знову вибираємо вершину, найближчу до а та не вибрану раніше; повторюємо процес.

Алгоритм Флойда використовує матрицю *A(n,n)*, в якій знаходяться довжини найкоротших шляхів:

$A_{ij} = C_{ij}$, якщо $i \neq j$;

$A_{ij} = 0$, якщо $i = j$;

$A_{i\,j} = \infty$, якщо відсутній шлях з вершини *i* у вершину *j*.

Над матрицею *A* виконується *n* ітерацій. Після *k*-ої ітерації A_{ij} містить значення найменшої довжини шляху з вершини *i* у вершину *j*, причому шлях не проходить через вершини з номерами великими *k*.

Обчислення на *k* -ій ітерації виконується за формулою:

$$A^{k}_{ij} = min\ (\ A^{\ k-1}_{ij},\ A^{\ k-1}_{ik},\ A^{\ k-1}_{kj}\ ).$$

Верхній індекс *k* означає значення матриці *A* після *k*-ої ітерації.

Для обчислення A^{k}_{ij} проводиться порівняння величини A^{k-1}_{ij} (тобто вартість шляху від вершини *i* до вершини *j* без участі вершини *k* або іншої вершини з вищим номером) з величиною $A^{k-1}_{ik} + A^{k-1}_{kj}$ (вартість шляху від вершини *i* до вершини *k* плюс вартість шляху від вершини *k* до вершини *j*). Якщо шлях через вершину *k* дешевше, ніж A^{k-1}_{ij}, то величина A^{k}_{ij} змінюється

Приклад. Розглянемо граф змішаного типу, зображений на рис. 5.2 а, де кожне неорієнтоване ребро розглядається як пара протилежно направлених дуг рівної ваги. Матриця суміжності надана на рис. 5.2б. Потрібно знайти всі найкоротші шляхи від вершини x_1 до всієї решти вершин алгоритмом Дейкстри.

КРОК 1. Привласнимо $L(x_1)=0$, $L(x_i)=\infty$ для всіх x_i, окрім x_1 . Покладемо $p=x_1$.

Перша ітерація.

КРОК 2. Знайдемо пряме відображення для поточної даної вершини:

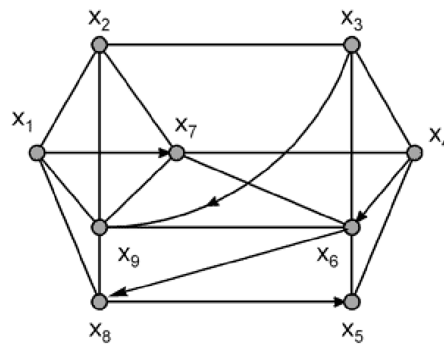
$\Gamma(p)=\Gamma(x_1)=\{x_2,x_7,x_8,x_9\}$. Всі вершини, що входять в пряме відображення мають тимчасові позначки, тому перерахуємо їх значення:

$L(x_2)=\min[L(x_2),L(x_1)+c(x_1,x_2)]=\min[\infty,0+10]=10$;

$L(x_7)=\min[\infty,0+3]=3$;

$L(x_8)=\min[\infty,0+6]=6$;

$L(x_9)=\min[\infty,0+12]=12$.



а

C=

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
x_1		10					3	6	12
x_2	10		18				2		13
x_3		18		25		20			7
x_4			25		5	16	4		
x_5				5		10			
x_6			20		10		14	15	9
x_7		2		4		14			24
x_8	6				23	15			5
x_9	12	13				9	24	5	

б

Рисунок 5.2 – Приклад пошуку найкоротшого шляху:а) граф; б) матриця вагів дуг

КРОК 3. На даному кроці ітерації маємо наступні тимчасові позначки вершин:

$L(x_2)=10$, $L(x_3)=\infty$,

$L(x_7)=3$, $L(x_4)=\infty$,

$L(x_8)=6$, $L(x_5)=\infty$,

$L(x_9)=12$, $L(x_6)=\infty$.

Очевидно, що мінімальну позначку, яка дорівнює 3, має вершина x_7.

КРОК 4. За наступну поточну позначку приймаємо вершину x_7, тобто $p=x_7$, а її позначка стає постійною, $L(x_7)=3^+$.

КРОК 5. Оскільки не всі вершини графу мають постійні позначки, переходимо до кроку 2.

Друга ітерація.

Граф з поточними значеннями позначок вершин надано на рис.5.3.

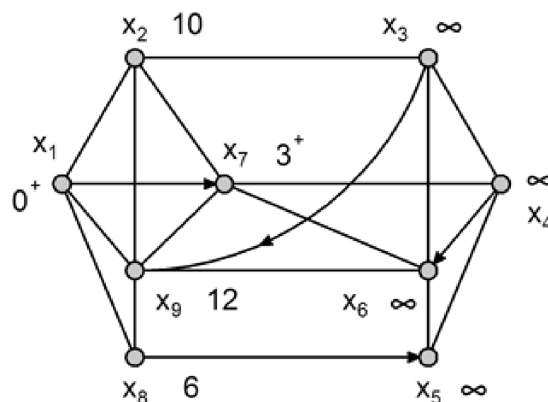


Рисунок 5.3 – Позначки в кінці першої ітерації

КРОК 2. Знаходимо $\Gamma(x_7)=\{x_2,x_4,x_6,x_9\}$. Позначки всіх вершин тимчасові, отже перераховуємо їх значення:

$L(x_2)=\min[10,3+2]=5$,

$L(x_4)=\min[\infty,3+4]=7$,

$L(x_6)=\min[\infty,3+14]=17$,

$L(x_9)=\min[12,3+24]=12$.

КРОК 3. На даному кроці ітерації маємо наступні тимчасові позначки вершин:

$L(x_2)=5$, $L(x_3)=\infty$,

$L(x_4)=7$, $L(x_5)=\infty$,

$L(x_6)=17$, $L(x_8)=6$, $L(x_9)=12$.

Очевидно, що мінімальну позначку, рівну 5, має вершина x_2 .

КРОК 4. За наступну поточну позначку приймаємо вершину x_2, тобто $p=x_2$, а її позначка стає постійною, $L(x_2)=5^+$.

КРОК 5. Оскільки не всі вершини графа мають постійні позначки, переходимо до кроку 2.

Третя ітерація.

Граф з поточними значеннями позначок вершин надано на рис. 5.4.

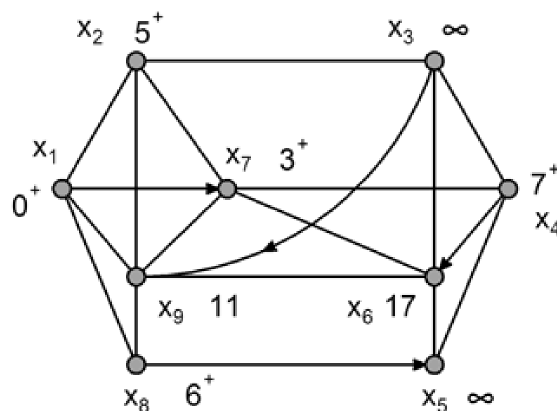


Рисунок 5.4 – Позначки в кінці другої ітерації

КРОК 2. Знаходимо $\Gamma(x_2)=\{x_1,x_3,x_7,x_9\}$. Помітки вершин x_3 і x_9 тимчасові, отже перераховуємо їх значення:

$L(x_3)=\min[\infty,5+18]=23$,

$L(x_9)=\min[12,5+13]=12$.

КРОК 3. На даному кроці ітерації маємо наступні тимчасові помітки вершин:

$L(x_3)=23$, $L(x_4)=7$, $L(x_5)=\infty$,

$L(x_6)=17$, $L(x_8)=6$, $L(x_9)=12$.

Очевидно, що мінімальну мітку, рівну 6, має вершина x_8.

КРОК 4. За наступну поточну помітку приймаємо вершину x_8, тобто $p=x_8$, а її мітка стає постійною, $L(x_8) = 6^+$.

КРОК 5. Не всі вершини графа мають постійні помітки, тому переходимо до кроку 2.

Четверта ітерація.

КРОК 2. Знаходимо $\Gamma(x_8)=\{x_1,x_5,x_6,x_9\}$. Помітки вершин x_5, x_6 і x_9 тимчасові, отже, перераховуємо їх значення:

$L(x_5)=\min[\infty,6+23]=29$,

$L(x_6)=\min[17,6+15]=17$,

$L(x_9)=\min[12,6+5]=11$.

КРОК 3. На даному кроці ітерації маємо наступні тимчасові позначки вершин:

$L(x_3)=23$, $L(x_4)=7$,

$L(x_5)=29$, $L(x_6)=17$, $L(x_9)=11$.

Очевидно, що мінімальну позначку, рівну 7 має верші на x_4 .

КРОК 4. За наступну поточну мітку приймаємо вершину x_4, тобто $p=x_4$, а її мітка стає постійною, $L(x_4) = 7^+$.

КРОК 5. Оскільки не всі вершини графа мають постійні позначки, переходимо до кроку 2.

П'ята ітерація.

КРОК 2. Знаходимо $\Gamma(x_4)=\{x_3,x_5,x_6,x_7\}$. Позначки вершин x_3, x_5 і x_6 тимчасові, отже, перераховуємо їх значення:

$L(x_3)=\min[23,7 + 25]=23$,

$L(x_5)=\min[29,7 + 5]=12$,

$L(x_6)=\min[17,7 +1 6]=17$.

КРОК 3. На даному кроці ітерації маємо наступні тимчасові позначки вершин:

$L(x_3) = 23$, $L(x_5) = 29$,

$L(x_6) = 17$, $L(x_9) = 11$.

Очевидно, що мінімальну позначку, рівну 11 має вершина x_9 .

КРОК 4. За наступну поточну позначку приймаємо вершину x_9, тобто $p=x_9$, а її мітка стає постійною, $L(x_9) = 11^+$.

КРОК 5. Оскільки не всі вершини графа мають постійні позначки, переходимо до кроку 2.

Шоста ітерація.

КРОК 2. Знаходимо $\Gamma(x_9)=\{x_1,x_2,x_6,x_7,x_8\}$. Саме вершина x_6 тимчасова, отже перераховуємо її значення: $L(x_6)=\min[17,11+9]=17$.

КРОК 3. На даному кроці ітерації маємо наступні тимчасові позначки вершин: $L(x_3) = 23$, $L(x_5) = 12$, $L(x_6) = 17$.

Очевидно, що мінімальну позначку, рівну 12 має вершина x_5 .

КРОК 4. За наступну поточну мітку приймаємо вершину x_5, тобто $p = x_5$, а її мітка стає постійною, $L(x_5) = 12^+$.

КРОК 5. Оскільки не всі вершини графа мають постійні мітки, переходимо до кроку 2.

Сьома ітерація.

КРОК 2. Знаходимо $\Gamma(x_5) = \{x_4, x_6\}$. Позначка вершини x_6 тимчасова, отже, перераховуємо її значення:

$L(x_6)=\min [17,12 + 10]=17$.

КРОК 3. На даному кроці ітерації маємо наступні тимчасові позначки:

$L(x_3) = 23$, $L(x_6) = 17$.

Очевидно, що мінімальну позначку, рівну 17 має вершина x_6 .

КРОК 4. За наступну поточну позначку приймаємо вершину x_6, тобто $p=x_6$, а її позначка стає постійною, $L(x_6) = 17^+$.

КРОК 5. Оскільки не всі вершини графа мають постійні позначки, переходимо до кроку 2.

Восьма ітерація.

КРОК 2. Знаходимо $\Gamma(x_6)=\{x_3,x_5,x_7,x_8,x_9\}$. позначка вершини x_3 тимчасова, отже, перераховуємо її значення: $L(x_3)=\min[23,17 + 20]=23$.

КРОК 3. На даному кроці ітерації маємо одну тимчасову позначку вершини: $L(x_3)=23$, яка стає постійною.

КРОК 4. Всі вершини мають постійні мітки, тому алгоритм закінчений. Для знаходження найкоротшого шляху між вершинами, наприклад, x_2 і початковою x_1 послідовно використовуємо співвідношення:

$$L(x'_2)+c(x'_2,x_2)=L(x_2) = 5,$$

де вершина x'_2 – це вершина, безпосередньо передуюча x_2 в найкоротшому шляху від x_1 до x_2 . Єдиною такою вершиною є вершина x_7 . Далі відношення застосовуємо другий раз:

$$L(x'_7)+c(x'_7,x_7)=L(x_7)=3.$$

Єдиною такою вершиною є вершина x_1 . Тому найкоротший шлях від x_1 до x_2 є перелік вершин: x_1, x_7, x_2.

Вершина x_1, є базою та надає всі найкоротші шляхи від x_1 , які надано на рис.5.5.

Дев’ята ітерація. Кінець алгоритму.

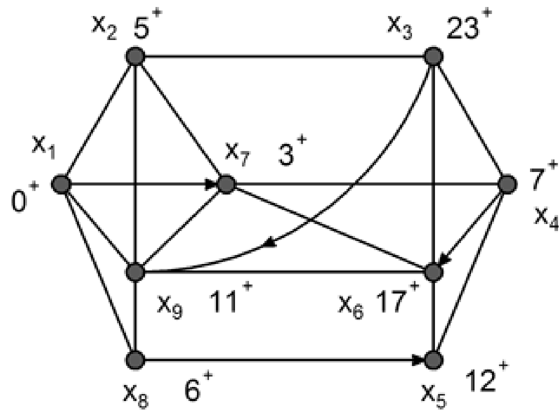


Рисунок 5.5 – Вершина x_1 є базовою вершиною

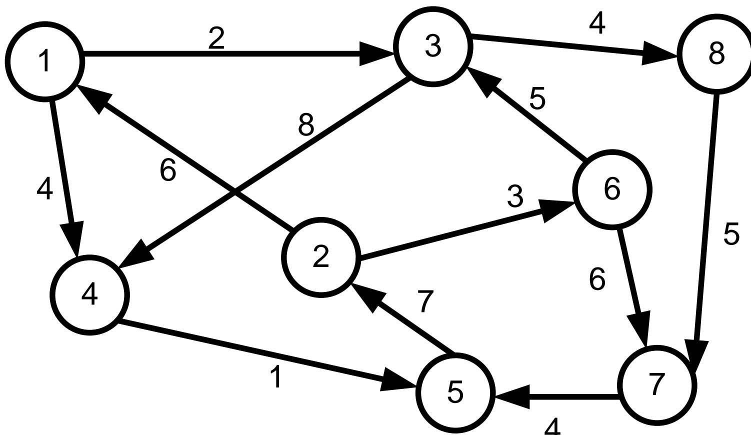
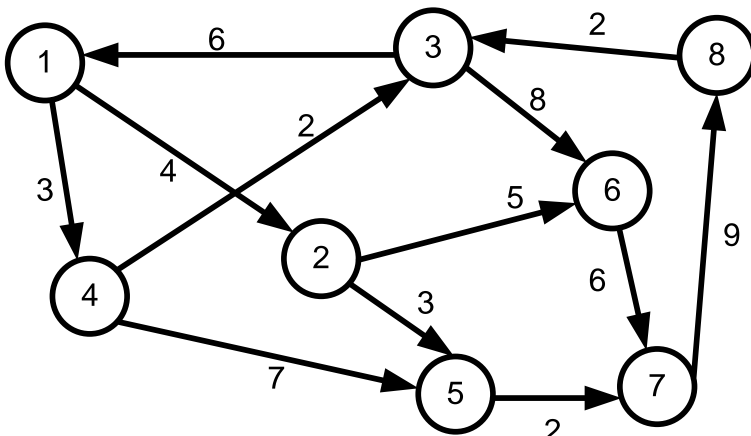
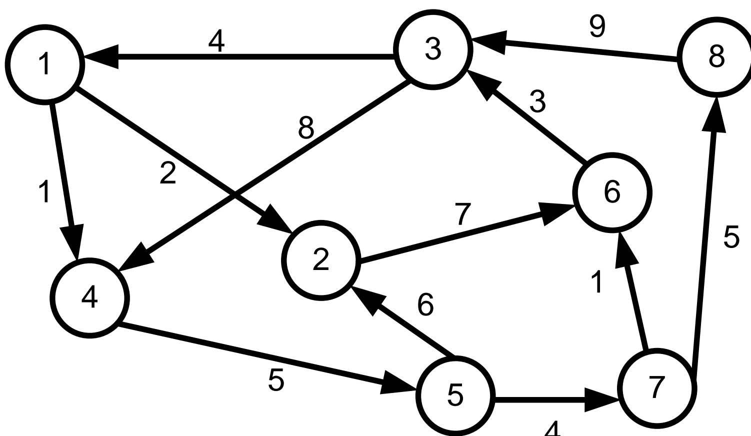
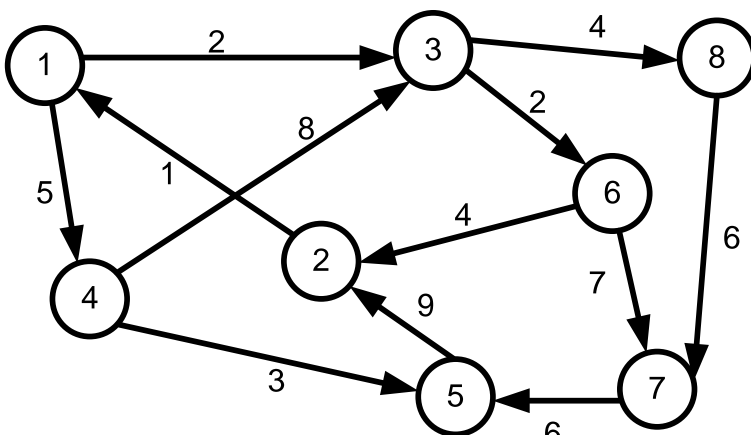
Контрольні питання

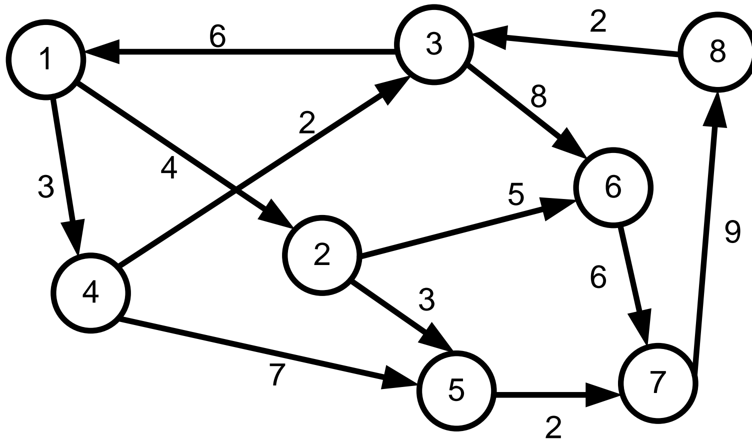
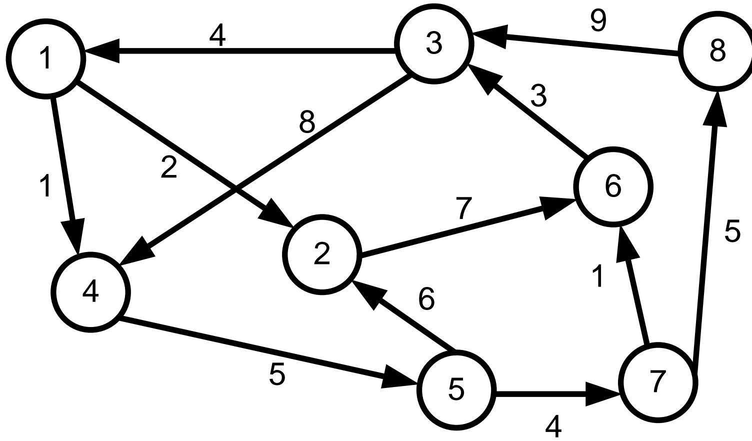
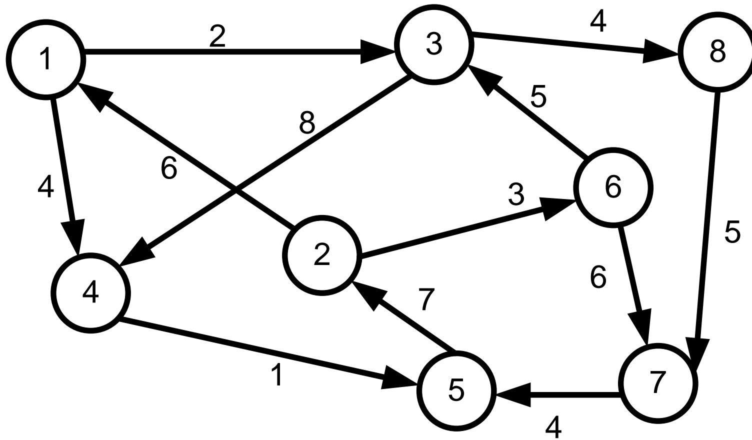
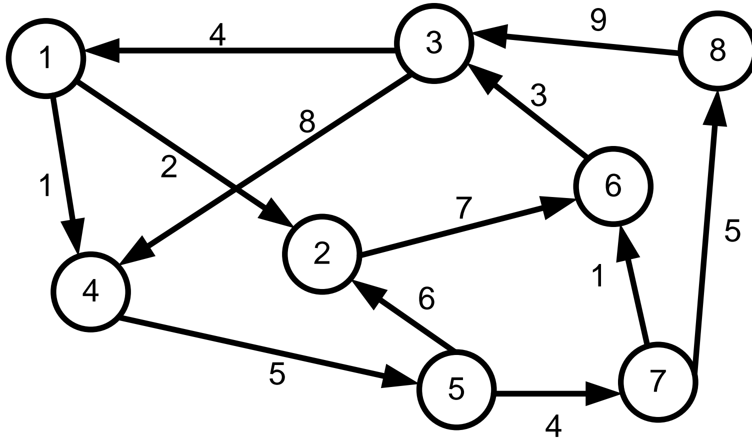
1. Що таке граф? Які види графів ви знаєте?
2. У якому виді зберігаються графи у пам'яті комп'ютера?
3. Що таке обхід графу?
4. Як відбувається обхід графа у глибину?
5. Як відбувається обхід графа по рівнях?
6. Як працює алгоритм Дейкстри?
7. Як працює алгоритм Флойда?

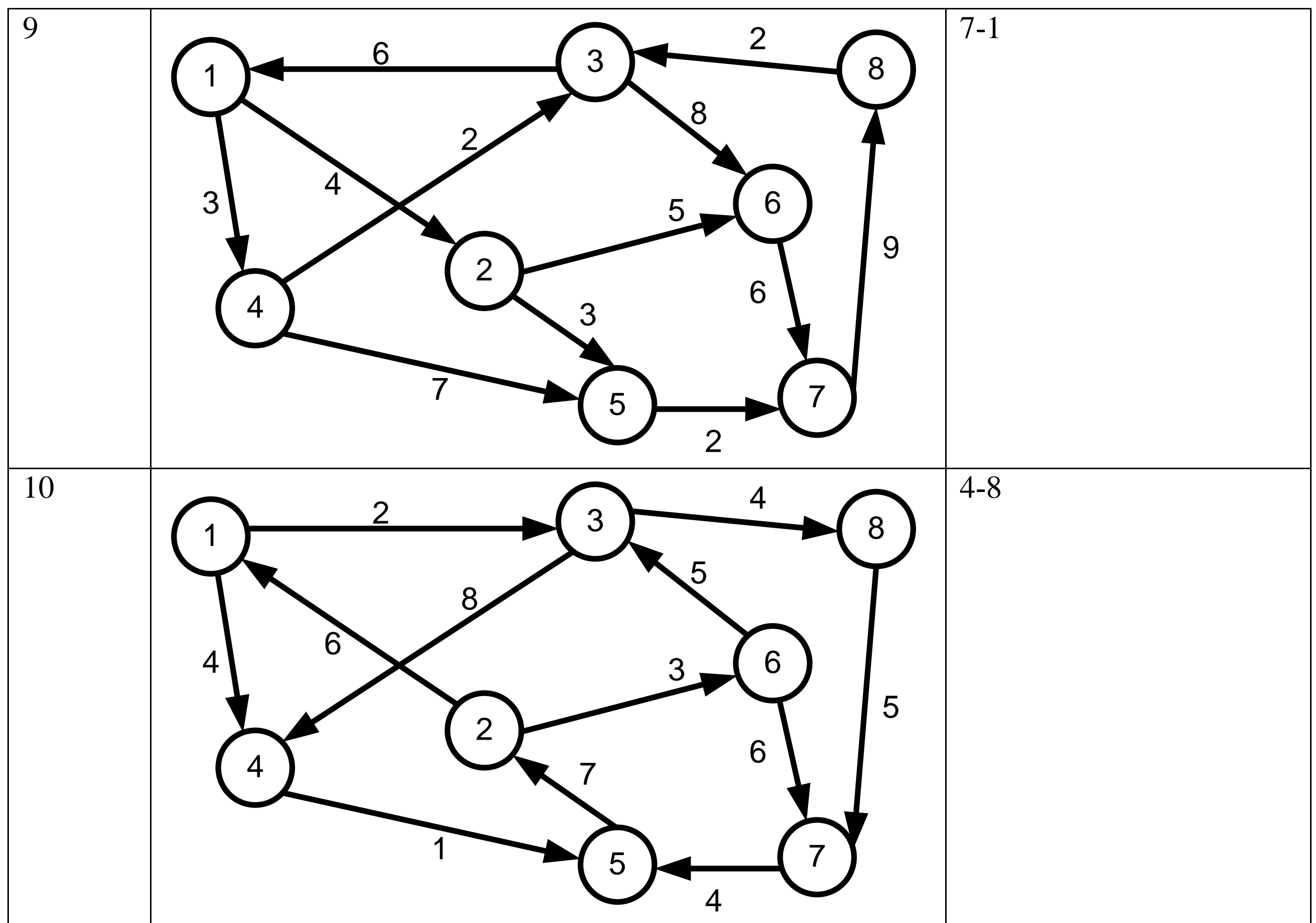
Варіанти завдань

Побудувати матрицю суміжності і знайти найкоротші шляхи з вершини N (для алгоритму Дейкстри) за допомогою алгоритму Дейкстри, Флойда в наступному графі

Варіант	Граф	Вершина N

1		1-7
2		2-8
3		5-3
4		6-1

5		1-8
6		2-8
7		6-1
8		5-1



9		7-1
10		4-8

Порядок проведення індивідуальної роботи

Для виконання роботи кожен студент повинен:

1. Відповісти на контрольні питання та пройти усне опитування за теоретичним матеріалом лабораторної роботи;
2. Отримати варіант завдання у викладача;
3. Скласти алгоритм розв'язання задачі;
4. Записати код програми на комп'ютері;
5. Відкомпілювати програму та виправити всі помилки;
6. Запустити програму на виконання;
7. Отримати результати роботи програми і показати їх викладачу;
8. Підготувати і захистити звіт до індивідуальної роботи.

Оформлення і захист звіту

Підготовлений до захисту звіт до лабораторної роботи повинен містити:

1. титульний лист, де вказані номер і назва лабораторної роботи, відомості про виконавця;
2. номер варіанта роботи та текст завдання;
3. відповіді на контрольні запитання до лабораторної роботи;
4. текст програми алгоритмічною мовою Java;
5. лістинг результатів виконання програми.

ДОДАТОК А

Програмний код двійкового дерева пошуку

```
class Node
{
public int iData; // data item (key)
public double dData; // data item
public Node leftChild; // this node’s left child
public Node rightChild; // this node’s right child
public void displayNode() // display ourself
{
System.out.print('{');
System.out.print(iData);
System.out.print(", ");
System.out.print(dData);
System.out.print("} ");
}
} // end class Node


////////////////////////////////////////////////////////////////

import java.io.*;
import java.util.*;
public class Tree {
     private Node root; // first node of tree
     public Tree() // constructor
     { root = null; } // no nodes in tree yet
     public Node find(int key) // find node with given key
     {
     Node current = root; // start at root
     while(current.iData != key) // while no match,
     {
     if(key < current.iData) // go left?
     current = current.leftChild;
     else // or go right?
     current = current.rightChild;
     if(current == null) // if no child,
     return null; // didn’t find it
     }
     return current; // found it
     } // end find()
     public void insert(int id, double dd)
     {
     Node newNode = new Node(); // make new node
     newNode.iData = id; // insert data
     newNode.dData = dd;
     if(root==null) // no node in root
```

```
root = newNode;
else // root occupied
{
Node current = root; // start at root
Node parent;
while(true) // (exits internally)
{
parent = current;
if(id < current.iData) // go left?
{
current = current.leftChild;
if(current == null) // if end of the line,
{ // insert on left
parent.leftChild = newNode;
return;
}
} //
else // or go right?
{
    current = current.rightChild;
    if(current == null) // if end of the line
    { // insert on right
    parent.rightChild = newNode;
    return;
    }
    }
    }
    }
    }
    // -----------------------------------------------------
    public boolean delete(int key) // delete node
    { // (assumes non-empty list)
    Node current = root;
    Node parent = root;
    boolean isLeftChild = true;
    while(current.iData != key) // search for node
    {
    parent = current;
    if(key < current.iData) // go left?
    {
    isLeftChild = true;
    current = current.leftChild;
    }
    else // or go right?
    {
    isLeftChild = false;
    current = current.rightChild;
    }
    if(current == null) // end of the line,
    return false; // didn't find it
    } // end while
    if(current.leftChild==null &&
    current.rightChild==null)
```

```
{
if(current == root) // if root,
root = null; // tree is empty
else if(isLeftChild)
     parent.leftChild = null; // disconnect
     else // from parent
     parent.rightChild = null;
     }
     else if(current.rightChild==null)
     if(current == root)
     root = current.leftChild;
     else if(isLeftChild)
     parent.leftChild = current.leftChild;
     else
     parent.rightChild = current.leftChild;
     else if(current.leftChild==null)
     if(current == root)
     root = current.rightChild;
     else if(isLeftChild)
     parent.leftChild = current.rightChild;
     else
     parent.rightChild = current.rightChild;
     else //
     {
     Node successor = getSuccessor(current);
     if(current == root)
     root = successor;
     else if(isLeftChild)
     parent.leftChild = successor;
     else
     parent.rightChild = successor;
     successor.leftChild = current.leftChild;
     } // end else two children
return true; // success
} // end delete()
// -----------------------------------------------------
private Node getSuccessor(Node delNode)
{
Node successorParent = delNode;
Node successor = delNode;
Node current = delNode.rightChild; // go to right child
while(current != null) // until no more
{ // left children,
successorParent = successor;
successor = current;
current = current.leftChild; // go to left child
}
if(successor != delNode.rightChild) // right child,
{ // make connections
successorParent.leftChild = successor.rightChild;
successor.rightChild = delNode.rightChild;
}
return successor;   }
```

```
// -------------------------------------------------------
public void traverse(int traverseType)
{
switch(traverseType)
{
case 1: System.out.print("\nPreorder traversal: ");
preOrder(root);
break;
case 2: System.out.print("\nInorder traversal: ");
inOrder(root);
break;
case 3: System.out.print("\nPostorder traversal: ");
postOrder(root);
break;
}
System.out.println();
}
// -------------------------------------------------------
private void preOrder(Node localRoot)
{
if(localRoot != null)
{
System.out.print(localRoot.iData + " ");
preOrder(localRoot.leftChild);
preOrder(localRoot.rightChild);
}
}
// -------------------------------------------------------
private void inOrder(Node localRoot)
{
if(localRoot != null)
{
inOrder(localRoot.leftChild);
System.out.print(localRoot.iData + " ");
inOrder(localRoot.rightChild);
}
}
// -------------------------------------------------------
private void postOrder(Node localRoot)
{
if(localRoot != null)
{
postOrder(localRoot.leftChild);
postOrder(localRoot.rightChild);
System.out.print(localRoot.iData + " ");
}
}
// -------------------------------------------------------
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(' ');
            } // end while globalStack not empty
            System.out.println();
            nBlanks /= 2;
            while(localStack.isEmpty()==false)
            globalStack.push( localStack.pop() );
            } // end while isRowEmpty is false

    System.out.println("...................................");
            } // end displayTree()
            } // end class Tree


import java.io.*;
import java.util.*;
public class TreeApp {
    public static void main(String[] args) throws IOException
    {
    int value;
    Tree theTree = new Tree();
    theTree.insert(50, 1.5);
    theTree.insert(25, 1.2);
    theTree.insert(75, 1.7);
    theTree.insert(12, 1.5);
    theTree.insert(37, 1.2);
```

```
theTree.insert(43, 1.7);
theTree.insert(30, 1.5);
theTree.insert(33, 1.2);
theTree.insert(87, 1.7);
theTree.insert(93, 1.5);
theTree.insert(97, 1.5);
while(true)
{
System.out.print("Enter first letter of show, ");
System.out.print("insert, find, delete, or traverse: ");
int choice = getChar();
switch(choice)
{
case 's':
theTree.displayTree();
break;
case 'i':
System.out.print("Enter value to insert: ");
value = getInt();
theTree.insert(value, value + 0.9);
break;
case 'f':
System.out.print("Enter value to find: ");
value = getInt();
Node found = theTree.find(value);
if(found != null)
{
System.out.print("Found: ");
found.displayNode();
System.out.print("\n");
}
else
System.out.print("Could not find ");
System.out.print(value + '\n');
break;
case 'd':
System.out.print("Enter value to delete: ");
value = getInt();
boolean didDelete = theTree.delete(value);
if(didDelete)
System.out.print("Deleted " + value + '\n');
else
System.out.print("Could not delete ");
System.out.print(value + '\n');
break;
case 't':
System.out.print("Enter type 1, 2 or 3: ");
value = getInt();
theTree.traverse(value);
break;
default:
System.out.print("Invalid entry\n");
} // end switch
```

```
} // end while
} // end main()
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);
     }
     // -------------------------------------------------------
     } // end class TreeApp
```

ДОДАТОК В

Хеш-таблиця

```
The hashChain.java Program
// hashChain.java

import java.io.*;
////////////////////////////////////////////////////////////////
class Link
{ // (could be other items)
private int iData; // data item
public Link next; // next link in list
// -------------------------------------------------------------
public Link(int it) // constructor
{ iData= it; }
// -------------------------------------------------------------
public int getKey()
{ return iData; }
// -------------------------------------------------------------
public void displayLink() // display this link
{ System.out.print(iData + " "); }
} // end class Link

////////////////////////////////////////////////////////////////
class SortedList
{
private Link first; // ref to first list item
// -------------------------------------------------------------
public void SortedList() // constructor
{ first = null; }
// -------------------------------------------------------------
public void insert(Link theLink) // insert link, in order
{
int key = theLink.getKey();
Link previous = null; // start at first
Link current = first;
// until end of list,
while( current != null && key > current.getKey() )
{ // or current > key,
previous = current;
current = current.next; // go to next item
```

```
}
if(previous==null) // if beginning of list,
first = theLink; // first --> new link
else // not at beginning,
previous.next = theLink; // prev --> new link
theLink.next = current; // new link --> current
} // end insert()
// -----------------------------------------------------------------
public void delete(int key) // delete link
{ // (assumes non-empty list)
Link previous = null; // start at first
Link current = first;
// until end of list,
while( current != null && key != current.getKey() )
{ // or key == current,
previous = current;
current = current.next; // go to next link
}
// disconnect link
if(previous==null) // if beginning of list
first = first.next; // delete first link
else // not at beginning
previous.next = current.next; // delete current link
} // end delete()
// -----------------------------------------------------------------
public Link find(int key) // find link
{
Link current = first; // start at first
// until end of list,
while(current != null && current.getKey() <= key)
{ // or key too small,
if(current.getKey() == key) // is this the link?
return current; // found it, return link
current = current.next; // go to next item
}
return null; // didn't find it
} // end find()
// -----------------------------------------------------------------
public void displayList()
{
System.out.print("List (first-->last): ");
Link current = first; // start at beginning of list
while(current != null) // until end of list,
{
```

```
current.displayLink(); // print data
current = current.next; // move to next link
}
System.out.println("");
}
} // end class SortedList
////////////////////////////////////////////////////////////////
class HashTable
{
private SortedList[] hashArray; // array of lists
private int arraySize;
// -------------------------------------------------------------
public HashTable(int size) // constructor
{
arraySize = size;
hashArray = new SortedList[arraySize]; // create array
for(int j=0; j<arraySize; j++) // fill array
hashArray[j] = new SortedList(); // with lists
}
// -------------------------------------------------------------
public void displayTable()
{
for(int j=0; j<arraySize; j++) // for each cell,
{
System.out.print(j + ". "); // display cell number
hashArray[j].displayList(); // display list
}
}
// -------------------------------------------------------------
public int hashFunc(int key) // hash function
{
return key % arraySize;
}
// -------------------------------------------------------------
public void insert(Link theLink) // insert a link
{
int key = theLink.getKey();
int hashVal = hashFunc(key); // hash the key
hashArray[hashVal].insert(theLink); // insert at hashVal
} // end insert()
// -------------------------------------------------------------
public void delete(int key) // delete a link
{
int hashVal = hashFunc(key); // hash the key
```

```
hashArray[hashVal].delete(key); // delete link
} // end delete()
// -------------------------------------------------------------
public Link find(int key) // find link
{
int hashVal = hashFunc(key); // hash the key
Link theLink = hashArray[hashVal].find(key); // get link
return theLink; // return link
}
// -------------------------------------------------------------
} // end class HashTable


////////////////////////////////////////////////////////////////
class HashChainApp
{
public static void main(String[] args) throws IOException
{
int aKey;
Link aDataItem;
int size, n, keysPerCell = 100;
// get sizes
System.out.print("Enter size of hash table: ");
size = getInt();
System.out.print("Enter initial number of items: ");
n = getInt();
// make table
HashTable theHashTable = new HashTable(size);
for(int j=0; j<n; j++) // insert data
{
aKey = (int)(java.lang.Math.random() *
keysPerCell * size);
aDataItem = new Link(aKey);
theHashTable.insert(aDataItem);
}
while(true) // interact with user
{
System.out.print("Enter first letter of ");
System.out.print("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 Link(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");
} // end switch
} // end while
} // end main()
//-----------------------------------------------------------------
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);}} // end class HashChainApp
```

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторних робіт студентів з дисципліни

“ТЕОРІЯ АЛГОРИТМІВ”

для студентів І курсу денної форми навчання

Напрям підготовки – комп’ютерні науки

Укладачі: Шпінарева І.М., к.ф.-м.н.

Підп. до друку Формат 60x84/16 Папір офс.

Умовн. а.а. Тираж Замовл.

Одеський державний екологічний університет,
65016, м. Одеса, вул. Львівська, 15