

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

**Український державний університет науки і технологій**

Кафедра «Комп’ютерні інформаційні технології»

**Лабораторна робота №3  
з дисципліни «Якість програмного забезпечення та тестування»**

**на тему:**

**«**Проектування структур даних ефективних за показниками обчислювальної складності.»

Виконав:

студент гр. ПЗ2011

Проценко Р.О.

Прийняла:

Стаднік А. В.

Дніпро, 2023

**Тема.** Проектування структур даних ефективних за показниками обчислювальної складності.

**Мета роботи.** Вивчити комбінаторно-ймовірнісні методи визначення показників обчислювальної складності алгоритмів та отримати навички застосування цих методів при вирішенні задач розробки ефективних структур даних.

**Завдання**

Розробити об’єктно-орієнтовану програму, що реалізує три структури даних, визначивши для них операції додавання, видалення, пошуку елементів.

Для тексту програми на заданій мові програмування виконати пошук усіх лексем класу, визначеного варіантом. Знайдені лексеми занести в розроблені структури, виключаючи повтор елементів.

Виконати ймовірнісно-комбінаторну оцінку функцій для роботи зі структурами, на основі якої зробити висновок про оптимальність структури з точки зору найменшої обчислювальної складності.

**Індивідуальне завдання**

|  |  |  |  |
| --- | --- | --- | --- |
| Структура | Операція додавання | Операція видалення | Операція пошуку |
| *Двозв’язний впорядкований список* | У позицію, що визначається порядком | За ключем з відповідної позиції | Шляхом послідовного перебору елементів |
| *Бінарне дерево з послідовним представленням* | У позицію, що залежить від значення елементів | За ключем з відповідної позиції | Перехід до наступного елементу в залежності від значення ключа: до лівого нащадка 2\*і+1, до правого нащадка 2\*і+2 |
| *Одномірний динамічний розширюваний масив* | У кінець | За ключем з відповідної позиції | Шляхом послідовного перебору елементів |

**Інформація про структури даних**

***Двозв’язний впорядкований список:***

Двозв’язний впорядкований список є структурою даних, що складається з вузлів, де кожен вузол має посилання на попередній та наступний вузол у послідовності. Впорядкований означає, що елементи у списку розташовані за зростанням (або за спаданням) значень.

*Доцільність застосування:*

*Додавання та видалення:*

Двозв’язні списки ефективно обробляють операції вставки та видалення в будь-якій позиції списку, оскільки не вимагають перебудови всієї структури.

*Пошук:*

Пошук може бути менш ефективним порівняно з масивами, але впорядкований список може використовуватися для швидкого вставлення та видалення з середини списку.

***Бінарне дерево з послідовним представленням:***

Бінарне дерево - це ієрархічна структура даних, де кожен вузол може мати максимум два нащадки, лівого та правого. Послідовне представлення використовує масив для зберігання вузлів так, що вузли розташовані послідовно за їхніми індексами.

*Доцільність застосування:*

*Швидкий пошук:*

Пошук в бінарному дереві є швидким завдяки можливості використання бінарного пошуку.

*Вставка та видалення:*

Ефективність вставки та видалення залежить від правильної структури дерева.

***Одномірний динамічний розширюваний масив:***

Одномірний динамічний розширюваний масив - це структура даних, яка використовується для зберігання послідовності елементів, і може автоматично збільшувати свій розмір при необхідності.

*Доцільність застосування:*

*Швидкий доступ до елементів:*

Масиви надають константний час доступу до будь-якого елементу за індексом.

*Ефективна вставка/видалення в кінці:*

Динамічні масиви зазвичай мають швидкий час вставки та видалення елементів в кінці.

**Критерії вибору мови програмування**

Для реалізації обраних структур та їх операцій була обрана мова програмування джава. Джава є мовою програмування, спроектованою для підтримки об'єктно-орієнтованого програмування, що сприяє створенню чіткої та модульної структури коду.

Також важливим чинником є підтримка автоматичного збору сміття, що спрощує управління пам'яттю та забезпечує автоматичне вивільнення ресурсів, покращуючи надійність та ефективність програм.

Окрім цього джава має широку ком’юніті спільноту та багато інформації і вже готових рішень різноманітних подібних задач.

**Текст розробленої програми**

package lr3;  
  
public class DoublyOrderedLinkedList {  
 private static class Node {  
 int data;  
 Node next;  
 Node prev;  
  
 public Node(int data) {  
 this.data = data;  
 }  
 }  
  
 private Node head;  
  
 public DoublyOrderedLinkedList() {  
 this.head = null;  
 }  
  
 public void add(int data) {  
 Node newNode = new Node(data);  
 if (head == null || data < head.data) {  
 newNode.next = head;  
 head = newNode;  
 } else {  
 Node current = head;  
 while (current.next != null && current.next.data < data) {  
 current = current.next;  
 }  
 newNode.next = current.next;  
 current.next = newNode;  
 }  
 }  
  
 public void remove(int key) {  
 Node current = head;  
 Node prev = null;  
 while (current != null && current.data != key) {  
 prev = current;  
 current = current.next;  
 }  
 if (current != null) {  
 if (prev != null) {  
 prev.next = current.next;  
 } else {  
 head = current.next;  
 }  
 }  
 }  
  
 public boolean search(int key) {  
 Node current = head;  
 while (current != null && current.data != key) {  
 current = current.next;  
 }  
 return current != null;  
 }  
}

public class BinaryTree {  
 private final int[] array;  
 private int size;  
  
 public BinaryTree(int capacity) {  
 this.array = new int[capacity];  
 this.size = 0;  
 }  
  
 public void add(int value) {  
 if (size < array.length) {  
 array[size] = value;  
 int index = size;  
 int parentIndex = (index - 1) / 2;  
  
 while (index > 0 && array[index] > array[parentIndex]) {  
 int temp = array[index];  
 array[index] = array[parentIndex];  
 array[parentIndex] = temp;  
  
 index = parentIndex;  
 parentIndex = (index - 1) / 2;  
 }  
  
 size++;  
 }  
 }  
  
 public void remove(int key) {  
 int index = -1;  
 for (int i = 0; i < size; i++) {  
 if (array[i] == key) {  
 index = i;  
 }  
 }  
   
 if (index != -1) {  
 array[index] = array[size - 1];  
 size--;  
  
 while (true) {  
 int leftChild = 2 \* index + 1;  
 int rightChild = 2 \* index + 2;  
 int maxIndex = index;  
  
 if (leftChild < size && array[leftChild] > array[maxIndex]) {  
 maxIndex = leftChild;  
 }  
  
 if (rightChild < size && array[rightChild] > array[maxIndex]) {  
 maxIndex = rightChild;  
 }  
  
 if (index == maxIndex) {  
 break;  
 }  
  
 int temp = array[index];  
 array[index] = array[maxIndex];  
 array[maxIndex] = temp;  
  
 index = maxIndex;  
 }  
 }  
 }  
  
 public boolean search(int key) {  
 for (int i = 0; i < size; i++) {  
 if (array[i] == key) {  
 return true;  
 }  
 }  
 return false;  
 }  
}  
  
package lr3;  
  
import java.util.Arrays;  
  
public class DynamicArray {  
 private int[] array;  
 private int size;  
  
 public DynamicArray() {  
 this.array = new int[100];  
 this.size = 0;  
 }  
  
 public void add(int value) {  
 if (size == array.length) {  
 int newCapacity = array.length \* 2;  
 array = Arrays.*copyOf*(array, newCapacity);  
 }  
 array[size] = value;  
 size++;  
 }  
  
 public void remove(int key) {  
 int index = -1;  
 for (int i = 0; i < size; i++) {  
 if (array[i] == key) {  
 index = i;  
 }  
 }  
   
 if (index != -1) {  
 for (int i = index; i < size - 1; i++) {  
 array[i] = array[i + 1];  
 }  
 size--;  
 }  
 }  
  
 public boolean search(int key) {  
 for (int i = 0; i < size; i++) {  
 if (array[i] == key) {  
 return true;  
 }  
 }  
 return false;  
 }  
}

**Розрахунок показників обчислювальної складності**

Двозв’язний впорядкований список:

Для **операції додавання**:

* *Найкращий випадок –* Додавання вузла у початок списку.
* *Середній випадок –* Додавання вузла у середину списку, для якого необхідно виконати обхід половини списку та операцію вставки.
* *Найгірший випадок –* Додавання у кінець списку, для якого необхідно виконати обхід усього списку.

Для **операції видалення**:

* *Найкращий випадок –* Видалення з початкової позиції.
* *Середній випадок –* Видалення з середини списку, для якого необхідно виконати обхід половини вузлів та операцію видалення.
* *Найгірший випадок –* Видалення з кінцевої позиції списку, для якого необхідно виконати обхід усіх вузлів та видалити елемент.

Для **операції пошуку**:

* *Найкращий випадок –* Пошук неіснуючого елементу, що менший за початковий елемент.
* *Середній випадок –* Пошук існуючого елементу у середині списку, при якому необхідно виконати перегляд половини вмісту структури даних.
* *Найгірший випадок –* Пошук неіснуючого елементу, який більший за кінцевий елемент, при якому необхідно здійснити перегляд усього вмісту структури даних.

Бінарне дерево з послідовним представленням:

Для **операції додавання**:

* *Найкращий випадок –* Додавання коли дерево порожнє.
* *Середній випадок –* Додавання коли дерево заповнене на половину.
* *Найгірший випадок –* Додавання коли дерево заповнене повністю.

Для **операції видалення**:

* *Найкращий випадок –* Видалення листку або крайнього елементу.
* *Середній випадок –* Видалення з середини дерева.
* *Найгірший випадок –* Видалення кореню дерева.

Для **операції пошуку**:

* *Найкращий випадок –* Шуканий елемент є коренем.
* *Середній випадок –* Пошук в бінарному дереві.
* *Найгірший випадок –* Пошук в дереві, яке вироджується у лінійний список.

Одномірний динамічний розширюваний масив:

Для **операції додавання**:

* *Найкращий випадок –* Коли є достатньо місця у поточному масиві.
* *Середній випадок –* Коли є достатньо місця у поточному масиві.
* *Найгірший випадок –* Якщо масив повністю заповнений і відбувається подвоєння розміру.

Для **операції видалення**:

* *Найкращий випадок –* Видалення останнього елементу.
* *Середній випадок –*  Видалення середнього елементу.
* *Найгірший випадок –* Видалення першого елементу.

Для **операції пошуку**:

* *Найкращий випадок –* Шуканий елемент є першим елементом масиву.
* *Середній випадок –* Елемент знаходиться в середині масиву.
* *Найгірший випадок –* Шуканий елемент є останнім у масиві, або його немає взагалі.

**Підрахунок кількості виконаних операторів при обробці двозв’язного впорядкованого списку**

|  |  |  |  |
| --- | --- | --- | --- |
| Операції та оператори | Кількість виконаних операторів | | |
| Найкращий випадок | Середній випадок | Найгірший випадок |
| Додавання елементу | | | |
| public void add(int data) {  Node newNode = new Node(data);  if (head == null || data < head.data) {  newNode.next = head;  head = newNode;  } else {  Node current = head;  while (current.next != null && current.next.data < data) {  current = current.next;  }  newNode.next = current.next;  current.next = newNode;  }  } | 1 1 1 1 0  0 0 0  0 0 | 1 1  0 0 1 1 n/2+1 n/2  1 1 | 1 1 0 0 1 1 n n-1  1 1 |
| **Разом** | 4 | 7 + n | 5+2n |
| Видалення | | | |
| public void remove(int key) {  Node current = head;  Node prev = null;  while (current != null && current.data != key) {  prev = current;  current = current.next;  }  if (current != null) {  if (prev != null) {  prev.next = current.next;  } else {  head = current.next;  }  } } | 1 1 1 0 0  1 1 0 1 1 | 1 1 n/2+1 n/2 n/2  1 1 1 0 0 | 1 1 n n-1 n-1  1 1 1 0 0 |
| **Разом** | 9 | 6+3n/2 | 3+3n |
| Пошук елементу | | | |
| public boolean search(int key) {  Node current = head;  while (current != null && current.data != key) {  current = current.next;  }  return current != null;  } | 1 1 0  1 | 1 n/2+1 n/2  1 | 1 n n-1  1 |
| **Разом** | 3 | 3+n | 1+2n |

**Підрахунок кількості виконаних операторів при обробці бінарного дерева з послідовним представленням**

|  |  |  |  |
| --- | --- | --- | --- |
| Операції та оператори | Кількість виконаних операторів | | |
| Найкращий випадок | Середній випадок | Найгірший випадок |
| Додавання елементу | | | |
| public void add(int value) {  if (size < array.length) {  array[size] = value;  int index = size;  int parentIndex = (index - 1) / 2;  while (index > 0 && array[index] > array[parentIndex]) {  int temp = array[index];  array[index] = array[parentIndex];  array[parentIndex] = temp;   index = parentIndex;  parentIndex = (index - 1) / 2;  }  size++;  }  } | 1 1 1 1 1 0 0 0  0 0  1 | 1 1 1 1 n/2+1 n/2 n/2 n/2  n/2 n/2  1 | 1 1 1 1 n n-1 n-1 n-1  n-1 n-1  1 |
| **Разом** | 6 | 6+3n | 6n |
| Видалення | | | |
| public void remove(int key) {  int index = -1;  for (int i = 0; i < size; i++) {  if (array[i] == key) {  index = i;  break;  }  }   if (index != -1) {  array[index] = array[size - 1];  size--;  while (true) {  int leftChild = 2 \* index + 1;  int rightChild = 2 \* index + 2;  int maxIndex = index;  if (leftChild < size && array[leftChild] > array[maxIndex]) {  maxIndex = leftChild;  }  if (rightChild < size && array[rightChild] > array[maxIndex]) {  maxIndex = rightChild;  }  if (index == maxIndex) {  break;  }  int temp = array[index];  array[index] = array[maxIndex];  array[maxIndex] = temp;  index = maxIndex;  }  } } | 1 n n 1    1 1 1 1 1 1 1 1 1  1 1  1 1  0 0 0 0 | 1 n/2 n/2 1    1 1 1 n/2 n/2 n/2 n/2 n/2 n/2  n/2 n/2  n/2 1  n/2 n/2 n/2 n/2 | 1 1 1 1    1 1 1 n n n n n n  n n  n 1  n n n n |
| **Разом** | 15+2n | 6 + 15n/2 | 8 + 13n |
| Пошук елементу | | | |
| public boolean search(int key) {  for (int i = 0; i < size; i++) {  if (array[i] == key) {  return true;  }  }  return false;  } | 1 1 1   0 | n/2 n/2 1   0 | n n 1 |
| **Разом** | 3 | 1 + n | 1 + 2n |

**Підрахунок кількості виконаних операторів при обробці** **динамічного розширюваного масиву**

|  |  |  |  |
| --- | --- | --- | --- |
| Операції та оператори | Кількість виконаних операторів | | |
| Найкращий випадок | Середній випадок | Найгірший випадок |
| Додавання елементу | | | |
| public void add(int value) {  if (size == array.length) {  int newCapacity = array.length \* 2;  array = Arrays.*copyOf*(array, newCapacity);  }  array[size] = value;  size++;  } | 1 0 0  1 1 | 1 0 0  1 1 | 1 1 n  1 1 |
| **Разом** | 3 | 3 | 4+n |
| Видалення | | | |
| public void remove(int key) {  int index = -1;  for (int i = 0; i < size; i++) {  if (array[i] == key) {  index = i;  break;  }  }  if (index != -1) {  for (int i = index; i < size - 1; i++) {  array[i] = array[i + 1];  }  size--;  }  } | 1 n n 1    1 1 1  1 | 1 n/2 n/2 1    1 n/2 n/2  1 | 1 1 1 1    1 n n  1 |
| **Разом** | 6+2n | 4+2n | 6+2n |
| Пошук елементу | | | |
| public boolean search(int key) {  for (int i = 0; i < size; i++) {  if (array[i] == key) {  return true;  }  }  return false;  } | 1 1 1   0 | n/2 n/2 1   0 | n n 1 |
| **Разом** | 3 | 1 + n | 1 + 2n |

**Розрахунок обчислювальної складності операцій обробки структур даних**

Визначимо обчислювальну складність операцій для розроблених структур за комбінаторно-імовірнісним методом.

Припустимо, що для деякої структури даних реалізовано операцій. Тоді ймовірність використання будь-якої з операцій дорівнює:

,

де  – ймовірність виконання -ої операції обробки даних, – кількість виконань -ої операції обробки даних. У (1) повинні бути враховані усі реалізовані операції обробки структури даних.

Під час обробки наведеного тексту лексичним аналізатором операція додавання у таблицю ідентифікаторі буде виконана 15 разів, а операція пошуку – 32. Тоді розрахуємо ймовірність використання операцій згідно з (1):

де  – ймовірність виконання операції додавання,  – ймовірність виконання операції пошуку, *P-* – ймовірність виконання операції видалення.

**Аналізу та підрахунок кількості використань операторів обробки структур даних**

|  |  |  |
| --- | --- | --- |
| Текст програми | Кількість використань операцій | |
| Операція додавання | Операція пошуку |
| public class DynamicArray {  private int[] array;  private int size;   public DynamicArray() {  this.array = new int[100];  this.size = 0;  }   public void add(int value) {  if (size == array.length) {  int newCapacity = array.length \* 2;  array = Arrays.*copyOf*(array, newCapacity);  }  array[size] = value;  size++;  }   public void remove(int key) {  int index = -1;  for (int i = 0; i < size; i++) {  if (array[i] == key) {  index = i;  }  }   if (index != -1) {  for (int i = index; i < size - 1; i++) {  array[i] = array[i + 1];  }  size--;  }  }   public boolean search(int key) {  for (int i = 0; i < size; i++) {  if (array[i] == key) {  return true;  }  }  return false;  } } | 1 1 1  1  1  1      1 1 1  1      1 1 | 1 1  1 2 1 2  3 1    4 3 2   1 4 3  1    4 3 |
| **Разом** | 12 | 37 |

Сума операцій видалення дорівнює сумі операцій додання = 12.

Правильність розрахунку перевіряється за умовою + + *P-*= 1.

Для двозв’язного впорядкованого списку:

Ŝ = 0.2\*4 + 0.6\*9 + 0.2\*3 = 6.8

S = 0.2\*(7 + 12) + 0.6\*(6+3\*12/2) + 0.2\*(3 + 12) = 17,6

Š = 0.2\*(5+2\*12) + 0.6\*(3\*12+3) + 0.2\*(1+2\*12) = 34,2

Для бінарного дерева з послідовним представленням:

Ŝ = 0.2\*6 + 0.6\*(15 + 12\*2) + 0.2\*3 = 25.2

S = 0.2\*(6+3\*12) + 0.6\*(6+ 15\*12/2) + 0.2\*(1+12)= 68.6

Š = 0.2\*6\*12 + 0.6\*(8+13\*12) + 0.2\*(1 + 2\*12) = 117.6

Для динамічного розширюваного масиву:

Ŝ = 0.2\*3 + 0.6\*(6+2\*12) + 0.2\*3 = 19.2

S = 0.2\*3 + 0.6\*(4+2\*12) + 0.2\*(1+12) = 20

Š = 0.2\*(4+12) + 0.6\*(6+2\*12) + 0.2\*(1+2\*12) = 26.2

Результати розрахунку обчислювальної складності

|  |  |  |  |
| --- | --- | --- | --- |
| Структура даних | Обчислювальна складність | | |
| Найкращий випадок | Середній випадок | Найгірший випадок |
| Двозв’язний впорядкований список | 6.8 | 17.6 | 34.2 |
| Бінарне дерево з послідовним представленням | 25.2 | 68.6 | 117.6 |
| Динамічний розширюваний масив | 19.2 | 20 | 26.2 |

**Аналіз результатів**

Розрахунок обчислювальної складності структур даних надає важливу інформацію щодо ефективності їх використання в конкретних сценаріях. Аналіз результатів для розглянутих структур даних може бути таким:

***Двозв’язний впорядкований список:***

*Найкращий випадок:* Додавання в початок списку має найменшу обчислювальну складність, що робить його оптимальним для цього випадку.

*Середній та найгірший випадок:* Обчислювальна складність зростає, але всі операції залишаються в прийнятних межах.

***Бінарне дерево з послідовним представленням:***

*Найкращий випадок:* Вставка в порожнє дерево найефективніша, але призводить до високої складності у середньому та найгіршому випадках.

*Середній та найгірший випадок:* Обчислювальна складність досить висока, зокрема у найгіршому випадку, що робить цю структуру менш ефективною для використання в даній задачі.

***Одномірний динамічний розширюваний масив:***

*Найкращий випадок:* Додавання у поточний масив без необхідності розширення є оптимальним.

*Середній та найгірший випадок:* При розширенні масиву виникає висока обчислювальна складність, особливо у найгіршому випадку.

Двозв’язний впорядкований список виглядає як оптимальний вибір, оскільки дозволяє ефективно виконувати операції додавання, видалення та пошуку у більшості випадків. Бінарне дерево з послідовним представленням має високу обчислювальну складність у найгіршому та середньому випадках, що робить його менш оптимальним для даної задачі.Одномірний динамічний розширюваний масив є ефективним для деяких випадків, але має високу обчислювальну складність при розширенні.

Ураховуючи ці обчислення, вибір двозв’язного впорядкованого списку на мові програмування Java може бути виправданим для оптимізації часу виконання операцій вставки, видалення та пошуку у вказаному контексті.

**Висновки**

Проектування ефективних структур даних є ключовим етапом в розробці алгоритмів, і вивчення показників обчислювальної складності є необхідною складовою для досягнення цієї ефективності.