

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

**Український державний університет науки і технологій**

Кафедра «Комп’ютерні інформаційні технології»

**Лабораторна робота №3**

**з дисципліни «Якість програмного забезпечення»**

**на тему:**

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

Виконала:

студентка гр. ПЗ2011

Костюк Анастасія Сергіївна

Прийняв:

Куроп’ятник О.С.

Дніпро, 2023

# Лабораторна робота № 3

**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.

## Зміст роботи:

1) виконати підготовку до роботи:

− ознайомитися з теоретичними відомостями;

− підготувати зразки даних (див. пояснення у пункті «Завдання»), придатних для подальшого аналізу;

− вивчити базові методи визначення показників обчислювальної складності алгоритмів;

2) виконати проектування альтернативних структур даних та операцій над ними на абстрактному та логічному рівнях;

3) виконати проектування структур даних на рівні представлення: розробити програму, що формує альтернативні структури даних та забезпечує необхідні операції над ними;

4) розрахувати показники обчислювальної складності операцій обробки структурованих даних;

5) обґрунтований вибір структур даних за допомогою комбінаторно-ймовірнісних методів;

6) оцінити якість виконання практичної частини роботи з представленням результатів у висновках.

## Зміст звіту:

1) вид і номер роботи, дисципліна, інформація про виконавця (група, П. І. Б.);

2) тема, мета лабораторної роботи;

3) постановка задачі і матеріали для визначення типового наповнення майбутніх структур;

4) інформацію про структури даних:

− можливі структури даних та аргументи про доцільність їх застосування;

− проекти трьох обраних структур даних на абстрактному рівні;

5) критерії вибору мови програмування, що була використана для реалізації обраних структур та їх операцій;

6) текст розробленої програми формування та використання альтернативних структур даних з достатнім рівнем коментованості;

7) розрахунок показників обчислювальної складності обробки структурованих даних, наведений у наступному порядку:

− визначення випадків кращої, середньої та гіршої обчислювальної складності операцій обробки даних;

− підрахунок кількості виконаних операторів для операцій обробки даних у вигляді таблиць 8, 9;

− розрахунок загальної кількості виконаних операторів для операцій обробки даних;

− розрахунок обчислювальної складності операцій обробки даних та їх представлення у вигляді таблиці 11.

8) аналіз результатів з обґрунтуванням вибору структури даних. Обґрунтування вибору структури даних має спиратись на проведені раніше дослідження. Аналіз результатів повинен формувати здатність об’єктивного оцінювання виконаної роботи для виявлення її переваг та недоліків.

9) висновки з оцінкою якості виконаної роботи

## Постановка задачі:

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.

Мову програмування, на якій буде представлено текст програми для лексичного розбору, обирає студент самостійно за погодженням з викладачем.

Особливості виконання операцій:

|  |  |  |  |
| --- | --- | --- | --- |
| Структура | Операція додавання | Операція видалення | Операція пошуку |
| 1. Однозв’язний впорядкований список | У позицію, що визначається порядком | За ключем з відповідної позиції | Шляхом послідовного перебору елементів |
| 5. Стек зі зв’язним представленням | У кінець | З кінця | Шляхом послідовного перебору елементів. Якщо елемент не рівний шуканому, він вивантажується в допоміжну структуру. Після того, як елемент було знайдено або перебирання закінчилося, елементи допоміжної структури повертаються у стек в початковому порядку |
| 7. Стек з послідовним представленням | У кінець | З кінця | Шляхом послідовного перебору елементів. Якщо елемент не рівний шуканому, він вивантажується в допоміжну структуру. Після того, як елемент було знайдено або перебирання закінчилося, елементи допоміжної структури повертаються у стек в початковому порядку |

1. **Інформація про структури даних** 
   1. **Однозв’язний впорядкований список**

Опис:

* class Node – Клас, що представляє собою вузол списку, з полями data (значення вузла: ціле число) та next (вказівник на наступний вузол).
* class OrderedLinkedList – Клас, що реалізує операції додавання, видалення та пошуку елементів у впорядкованому однозв'язному списку.

Операції додавання та видалення елементів реалізовані таким чином, щоб зберігати порядок.

Пошук елемента виконується шляхом обходу списку та порівняння значень.

Операції:

* void addElement(int value) – Додає новий елемент зі значенням value у впорядкований список.
* void removeElement(int key) – Видаляє елемент із списку за ключем key. Перевіряє наявність елемента з ключем key у списку.
* bool searchElement(int key) – Перевіряє наявність елемента з ключем key у списку.
  1. **Стек за зв’язним представленням**

Опис:

* class Node – Клас, що представляє собою вузол списку, з полями data (значення вузла: ціле число) та next (вказівник на наступний вузол).
* class OrderedLinkedList – Клас, який реалізує операції push, pop та search для стеку із зв'язним представленням.

Операції:

* void push(int value) – Додає новий елемент на вершину стеку зі значенням value.
* int pop() – Видаляє та повертає елемент з вершини стеку.
* bool find(int value) const – Пошук елемента у стеці.
  1. **Стек із послідовним представленням**

Опис:

* class OrderedLinkedList – Клас, що використовує вектор для послідовного зберігання елементів стеку.

Операції:

* void push(int data) – Додає новий елемент на вершину стеку зі значенням data.
* int pop() – Додає новий елемент на вершину стеку зі значенням data..
* bool searchElement(int key) – Пошук елемента з ключем key у стеці.

**Стек** – різновид лінійного списку, структура даних, яка працює за принципом «останнім прийшов – першим пішов». Всі операції в стеку можна проводити тільки з одним елементом, який знаходиться на верхівці стеку та був введений в стек останнім.

Операції зі стеком.

* push («заштовхнути елемент»): елемент додається в стек та розміщується в його верхівці. Розмір стеку збільшується на одиницю. При перевищенні розміру стека граничної величини, відбувається переповнення стека (англ. stack overflow);
* pop («виштовхнути елемент»): отримує елемент з верхівки стеку. Він видаляється зі стеку і його місце в верхівці стеку займає наступний за ним відп. до LIFO, а розмір стеку зменш. на одиницю. При намаганні «виштовхнути» елемент з вже пустого стеку, відбувається ситуація «незаповнення» стеку.
* isEmpty: перевірка наявності ел. в стеку; результат: істина, коли стек порожній;
* isFull: перевірка заповненості стека. Результат: істина, коли додавання нового елементу неможливе;
* clear: звільнити стек (видалити усі елементи);
* top: отримати верхній елемент (без виштовхування);
* size: отримати розмір (кількість елементів) стека;
* swap: поміняти два верхніх елементи місцями.

**Список** - лінійна структура даних, яка представляє собою послідовність елементів, де кожен елемент має посилання на наступний.

Операції включають:

* insert: вставка елемента у вказане місце списку.
* delete: видалення елемента зі списку.
* search: пошук елемента у списку.
* size: отримання кількості елементів у списку.
* isEmpty: перевірка порожнечі списку.

Список може бути однозв'язним (кожен елемент має посилання на наступний) або двозв'язним (є посилання і на попередній, і на наступний елемент).

1. **Критерії вибору мови програмування**

* Підтримка об'єктно-орієнтованого програмування.

Оскільки використовуються класи для реалізації структур даних, мова повинна підтримувати об'єктно-орієнтований підхід. С++ є такою мовою.

* Зручність роботи з динамічною пам'яттю.

Оскільки використовуються зв’язані структури, мова повинна забезпечувати зручність управління динамічною пам'яттю. C++ забезпечує ефективний механізм управління пам'яттю через new/delete.

* Ефективність та продуктивність.

Для розрахунку показників обчислювальної складності важлива продуктивність та можливість ефективної реалізації операцій. C++ відома своєю високою продуктивністю. Python міг забезпечити зручний синтаксис, але може бути менш ефективним у порівнянні з C++.

1. **Текст програми**

Проектування структур даних на рівні представлення полягає у визначенні самих структур даних та їх операцій засобами мови програмування.

* 1. Приклад реалізації операцій над елементами однозв’язного впорядкованого списку на мові C++:

#include <iostream>

class Node {

public:

int data;

Node\* next;

Node(int value) : data(value), next(nullptr) {}

};

class OrderedLinkedList {

private:

Node\* head;

public:

OrderedLinkedList() : head(nullptr) {}

void addElement(int value) {

Node\* newNode = new Node(value);

if (!head || value < head->data) {

newNode->next = head;

head = newNode;

}

else {

Node\* current = head;

while (current->next && current->next->data < value) {

current = current->next;

}

newNode->next = current->next;

current->next = newNode;

}

}

void removeElement(int key) {

Node\* current = head;

if (current && current->data == key) {

head = current->next;

delete current;

return;

}

while (current->next && current->next->data != key) {

current = current->next;

}

if (current->next) {

Node\* temp = current->next;

current->next = current->next->next;

delete temp;

}

}

bool searchElement(int key) {

Node\* current = head;

while (current) {

if (current->data == key) {

return true;

}

current = current->next;

}

return false;

}

};

* 1. Приклад реалізації операцій над елементами стеку зі зв’язним представленням на мові С++

#include <iostream>

class Node {

public:

int data;

Node\* next;

Node(int value) : data(value), next(nullptr) {}

};

class LinkedStack {

private:

Node\* top;

public:

LinkedStack() : top(nullptr) {}

void push(int value) {

Node\* newNode = new Node(value);

newNode->next = top;

top = newNode;

}

int pop() {

if (top) {

int popped = top->data;

Node\* temp = top;

top = top->next;

delete temp;

return popped;

}

else {

return -1; // Assuming -1 as an indicator of an empty stack

}

}

bool search(int value)

{

Node\* temp = top;

LinkedStack auxiliaryStack;

while (temp){

if (temp->data == value) {

while (auxiliaryStack.top) {

push(auxiliaryStack.pop());

}

return true;

}

else{

auxiliaryStack.push(pop());

temp = top;

}

}

while (auxiliaryStack.top {

push(auxiliaryStack.pop());

}

return false;

}};

* 1. Приклад реалізації операцій над елементами стеку з послідовним представленням на мові С++

#include <iostream>

class SequentialStack {

private:

static const int maxSize = 100; // Максимальний розмір стеку

int elements[maxSize];

int top;

public:

SequentialStack() : top(-1) {}

void push(int data) {

if (top < maxSize - 1) {

elements[++top] = data;

}

else {

return -1;

}

}

int pop() {

if (top >= 0) {

return elements[top--];

}

else {

return -1; // Assuming -1 as an indicator of an empty stack

}

}

bool searchElement(int key) {

SequentialStack tempStack;

bool found = false;

while (top >= 0) {

int current = pop();

if (current == key) {

found = true;

}

tempStack.push(current);

}

while (tempStack.top >= 0) {

push(tempStack.pop());

}

return found;

}};

1. **Аналіз операцій над елементами структур даних**

Аналіз операцій полягає у визначенні найкращого, середнього та найгіршого випадків обробки даних у розроблених структурах. Результат аналізу операції над структурами, що реалізовані у прикладах наведені нижче.

Однозв’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.

Стек зі зв’язним представленням

Для **операції додавання**:

* *Найкращий випадок –* Додавання елемента на початок списку, тобто на вершину стеку.
* *Середній випадок –* Додавання елемента на початок списку, тобто на вершину стеку.
* *Найгірший випадок –* Додавання елемента на початок списку, тобто на вершину стеку.

Для **операції видалення**:

* *Найкращий випадок –* Видалення з вершини стеку.
* *Середній випадок –* Видалення з середини стеку.
* *Найгірший випадок –* Видалення з початку стеку.

Для **операції пошуку**:

* *Найкращий випадок –* Пошук елементу голови
* *Середній випадок –* Пошук елементу у середині стеку (будь-якому місці)
* *Найгірший випадок –* Пошук неіснуючого елементу

Стек з послідовним представленням:

Для **операції додавання**:

* *Найкращий випадок –* Додавання в кінець стеку
* *Середній випадок –* Додавання в кінець стеку
* *Найгірший випадок –* Додавання в кінець стеку

Для **операції видалення**:

* *Найкращий випадок –* коли в стеку немає жодного елементу
* *Середній випадок –*  видалення будь-якого елемента із стеку
* *Найгірший випадок –* видалення будь-якого елемента зі стеку

Для **операції пошуку**:

* *Найкращий випадок –* Пошук елементу в кінці стеку
* *Середній випадок –* Пошук елементу у середині стеку
* *Найгірший випадок –* Пошук неіснуючого елементу

1. **Розрахунок показників обчислювальної складності**

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

|  |  |  |  |
| --- | --- | --- | --- |
| Операції та оператори | Кількість виконаних операторів | | |
| Найкращий випадок | Середній випадок | Найгірший випадок |
| Додавання елементу | | | |
| Node\* newNode = new Node(value);  if (!head || value < head->data) {  newNode->next = head;  head = newNode;  }  else {  Node\* current = head;  while (current->next && current->next->data < value) {  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 + 2n/2 | 5+2n |
| Видалення | | | |
| Node\* current = head;  if (current && current->data == key) {  head = current->next;  delete current;  return; }  while (current->next && current->next->data != key) {  current = current->next; }  if (current->next) {  Node\* temp = current->next;  current->next = current->next->next;  delete temp;  } | 1 1 1 1 1  1 0  1 1 1 1 | 1 1 0 0 0  n/2+1 n/2  1 0 0 0 | 1 1 0 0 0  n n-1  1 0 0 0 |
| **Разом** | 9 | 4+2n/2 | 2+2n |
| Пошук елементу | | | |
| Node\* current = head;  while (current) {  if (current->data == key) {  return true;  }  current = current->next; }  return false; | 1 1 0 0  0  0 | 1 n/2+1 n/2 1  n/2  1 | 1 n n-1 0  n  1 |
| **Разом** | 2 | 4+3n/2 | 3n+2 |

**Підрахунок кількості виконаних операторів при обробці стеку зі зв’язним представленням**

|  |  |  |  |
| --- | --- | --- | --- |
| Операції та оператори | Кількість виконаних операторів | | |
| Найкращий випадок | Середній випадок | Найгірший випадок |
| Додавання елементу | | | |
| Node\* newNode = new Node(value);  newNode->next = top;  top = newNode; | 1 1 1 | 1 1 1 | 1 1 1 |
| **Разом** | 3 | 3 | 3 |
| Видалення | | | |
| if (top) {  int popped = top->data;  Node\* temp = top;  top = top->next;  delete temp;  return popped;  }  else {  return -1;  } | 1 1 1 1 1 1  0 0 | 1 0 0 0 0 0  1 1 | 1 0 0 0 0 0  1 1 |
| **Разом** | 6 | 3 | 3 |
| Пошук елементу | | | |
| Node\* temp = top;  LinkedStack auxiliaryStack;  while (temp){  if (temp->data == value) {  while (auxiliaryStack.top) {  push(auxiliaryStack.pop());  }  return true;  }  else{  auxiliaryStack.push(pop());  temp = top;  }  }  while (auxiliaryStack.top {  push(auxiliaryStack.pop());  }  return false; | 1 1  1 0 0 0  0  0 0 0   1 0  0 | 1 1  n/2+1 n/2 n/2+1 3n/2  1  n/2-1 3n/2 n/2   n/2+1 3n/2  1 | 1 1  n+1 n n+1 3n  0  n 0 0   n+1 3n  1 |
| **Разом** | 4 | 6 + 16n/2 | 6 + 11n |

**Підрахунок кількості виконаних операторів при обробці стеку із послідовним представленням**

|  |  |  |  |
| --- | --- | --- | --- |
| Операції та оператори | Кількість виконаних операторів | | |
| Найкращий випадок | Середній випадок | Найгірший випадок |
| Додавання елементу | | | |
| if (top < maxSize - 1) {  elements[++top] = data;  }  else {  return -1;  } | 1 1  0 0 | 1 1  0 0 | 1 1  0 0 |
| **Разом** | 2 | 2 | 2 |
| Видалення | | | |
| if (top >= 0) {  return elements[top--];  }  else {  return -1;  } | 1 1  0 0 | 1 0  1 1 | 1 0  1 1 |
| **Разом** | 2 | 3 | 3 |
| Пошук елементу | | | |
| SequentialStack tempStack;  bool found = false;  while (top >= 0) {  int current = pop();  if (current == key) {  found = true;  }  tempStack.push(current);  }  while (tempStack.top >= 0) {  push(tempStack.pop());  }  return found; | 1 1 1 0 0 0  0  1 0  1 | 1 1 n/2+1 n/2 n/2 1  2n/2  n/2+1 2n/2  1 | 1 1 n+1 n n 0  2n  n+1 2n  1 |
| **Разом** | 5 | 6 + 8n/2 | 5 + 8n |

1. **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.

Використаємо зразок програми на мові C++.

Під час обробки наведеного тексту лексичним аналізатором операція додавання у таблицю ідентифікаторі буде виконана 30 разів, а операція пошуку – 62. Тоді розрахуємо ймовірність використання операцій згідно з (1):

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

**Зразок програми для аналізу та підрахунку кількості використань операторів обробки структур даних**

|  |  |  |  |
| --- | --- | --- | --- |
| Текст програми | Кількість використань | | |
| Операція додавання | Операція пошуку | Операція видалення |
| #include <iostream>  #include <ctime>  const int maxAttempts = /\* specify your value \*/;  bool nrButtonActive, acceptButtonActive, roundEnd;  int round, attempt, wins, field;  int pins[maxAttempts][4], answer[maxAttempts][4];  int passwordPin[4];  bool passwordHelp[2][4];  void initRound(int round) {  srand(time(0));  for (int i = 0; i < maxAttempts; ++i) {  for (int j = 0; j < 4; ++j) {  pins[i][j] = -1;  answer[i][j] = -1;  }  }  for (int i = 0; i < 4; ++i) {  bool isChosen = true;  while (isChosen) {  isChosen = false;  int a, b, c, d;  time\_t currentTime = time(0);  tm\* localTime = localtime(&currentTime);  a = localTime->tm\_hour;  b = localTime->tm\_min;  c = localTime->tm\_sec;  d = rand() % 100;  passwordPin[i] = (d) % 6;  for (int j = 0; j < i; ++j) {  if (i != j && passwordPin[j] == passwordPin[i]) {  isChosen = true;  break;  }  }  }  }  for (int i = 0; i < 4; ++i) {  pins[attempt][i] = -1;  }  nrButtonActive = false;  acceptButtonActive = false;  roundEnd = false;  round = round;  attempt = 0;  field = -1;  int selectedPin = -1;  }  int main() {  int currentRound = 1;  initRound(currentRound);  return 0;  } | 1 3 4 2 1 1  1 2  1 1       1   4 2 2      1                      1   1 1 | 1   4 3 3 3    3 1 1 1  2 2 2 2 2 1  3  3 6  1       3 3   1 1 1 2 1 1 1   1 1 2 |  |
| **Разом** | 30 | 62 | 30 |

Сума операцій видалення дорівнює сумі операцій додання = 30.

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

Для однозв'язного впорядкованого списку:

Ŝ = 0.25\*4 + 0.5\*2 + 0.25\*9 = 4,25

S = 0.25\*(7+2\*30/2) + 0.5\*(4+3\*30/2) + 0.25\*(4+2\*30/2) = 42,25

Š = 0.25\*(5+2\*30) + 0.5\*(3\*30+2) + 0.25\*(2+2\*30) = 77,75

Для стека зі зв'язним представленням:

Ŝ = 0.25\*3 + 0.5\*4 + 0.25\*6 = 4,25

S = 0.25\*3 + 0.5\*(6+ 16\*30/2) + 0.25\*3= 124,5

Š = 0.25\*3 + 0.5\*(6+11\*30) + 0.25\*3 = 169,5

Для стека із послідовним представленням:

Ŝ = 0.25\*2 + 0.5\*5 + 0.25\*2 = 3,5

S = 0.25\*2 + 0.5\*(6+8\*30/2) + 0.25\*3 = 64,25

Š = 0.25\*2 + 0.5\*(5+8\*30) + 0.25\*3 = 123,75

Результати розрахунку обчислювальної складності

|  |  |  |  |
| --- | --- | --- | --- |
| Структура даних | Обчислювальна складність | | |
| Найкращий випадок | Середній випадок | Найгірший випадок |
| Однозв’язний впорядкований список | 4,25 | 42,25 | 77,75 |
| Стек із зв’язним представленням | 4,25 | 124,5 | 169,5 |
| Стек із послідовним представленням | 3,5 | 64,25 | 123,75 |

1. **Аналіз результатів**

Розрахунок обчислювальної складності структур даних вказує на те, як кількість операцій залежить від об'єму даних, які обробляються. У контексті розглянутої задачі, де важлива оптимізація часу виконання операцій, було виявлено різну обчислювальну складність.

*У однозв'язному впорядкованому списку:*- Прийнята обчислювальна складність у найкращому випадку, але значно підвищується в середньому та найгіршому.   
- Є оптимальним варіантом, оскільки має помірну обчислювальну складність.

*У стеці із зв’язним представленням:*- Має дуже велику обчислювальну складність у всіх випадку.   
- Не є оптимальним варіантом, оскільки має не прийняту обчислювальну складність

*У стеці із послідовним представленням:*   
- Має меншу обчислювальну складність у найкращому випадках, висока ефективність для даної задачі. Але має дуже високу обчислювальну складність у найгіршому та середньому випадках.

Загальним є висновок, що стек зі зв’язним однозв’язний впорядкований список виглядає як оптимальний вибір, оскільки він дозволяє ефективно виконувати операції додавання, видалення та пошуку.

**Висновки**

В ході виконання лабораторної роботи було створено три структури даних, проведено дослідження обчислювальної складності для різних структур даних. За допомогою формул було отримано те, що для пошуку зарезервованих слів найкраще використовувати однозв’язний впорядкований список, так як він показує найкращі оптимальні показники . Його використання зменшує обчислювальну складність та забезпечує оптимальну продуктивність задачі.