МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ



**Український державний університет науки і технологій**

Кафедра «Комп’ютерні інформаційні технології»

**Лабораторна робота №4  
з дисципліни «Якість програмного забезпечення та тестування»**

**на тему: «Проектування структур даних з урахуванням програмно-апаратних середовищ їх використання»**

Виконав: студент гр. ПЗ2011

Білоус Олег Вікторович

Перевірив: доцент КІТ

Куроп’ятник О. С.

Дніпро, 2023

Лабораторна робота №4

**Тема:** Проектування структур даних з урахуванням програмно-апаратних середовищ їх використання.

**Мета:** Вивчити статистичні методи визначення показників часової ефективності алгоритмів та набути практичних навичок застосування цих методів при вирішенні задач розробки структур даних.

1. **Постановка задачі**

**Варіант 2**

Провести розрахунки середньої переваги однієї структури даних над іншою та співвідношення областей переваги для двох найкращих структур даних із минулої роботи.

1. **Текст програми**

#include <iostream>

#include <windows.h>

#include <vector>

using namespace std;

struct Node {

int data;

Node\* prev;

Node\* next;

Node(int val) : data(val), prev(nullptr), next(nullptr) {}

};

class OrderedList {

private:

Node\* head;

Node\* tail;

public:

OrderedList() : head(nullptr), tail(nullptr) {}

// Додавання елемента

void insert(int val) {

Node\* newNode = new Node(val);

if (head == nullptr) {

head = tail = newNode;

}

else if (head->data >= val) {

newNode->next = head;

head->prev = newNode;

head = newNode;

}

else if (tail->data <= val) {

newNode->prev = tail;

tail->next = newNode;

tail = newNode;

}

else {

Node\* temp = head->next;

while (temp->data < val) {

temp = temp->next;

}

newNode->prev = temp->prev;

newNode->next = temp;

temp->prev->next = newNode;

temp->prev = newNode;

}

}

// Видалення елемента

void remove(int val) {

if (head == nullptr) {

std::cout << "Список порожній" << std::endl;

return;

}

if (head->data == val) {

Node\* temp = head;

head = head->next;

if (head != nullptr) {

head->prev = nullptr;

}

else {

tail = nullptr;

}

delete temp;

}

else if (tail->data == val) {

Node\* temp = tail;

tail = tail->prev;

tail->next = nullptr;

delete temp;

}

else {

Node\* temp = head->next;

while (temp != nullptr && temp->data != val) {

temp = temp->next;

}

if (temp != nullptr) {

temp->prev->next = temp->next;

temp->next->prev = temp->prev;

delete temp;

}

else {

std::cout << "Елемент не знайдено" << std::endl;

}

}

}

// Пошук за значенням

bool search(int val) {

Node\* temp = head;

while (temp != nullptr) {

if (temp->data == val) {

return true;

}

temp = temp->next;

}

return false;

}

};

struct NodeQ {

int data;

NodeQ\* next;

NodeQ(int val) : data(val), next(nullptr) {}

};

class Queue {

private:

NodeQ\* front;

NodeQ\* rear;

public:

Queue() : front(nullptr), rear(nullptr) {}

// Додавання елемента в кінець черги

void enqueue(int val) {

NodeQ\* newNode = new NodeQ(val);

if (front == nullptr) {

front = rear = newNode;

}

else {

rear->next = newNode;

rear = newNode;

}

}

// Видалення елемента з початку черги

void dequeue() {

if (front == nullptr) {

std::cout << "Queue is empty" << std::endl;

return;

}

NodeQ\* temp = front;

front = front->next;

delete temp;

}

// Пошук елемента та перестановка в кінець черги

void searchAndMoveToEnd(int val) {

if (front == nullptr || front->next == nullptr) {

return;

}

NodeQ\* prev = nullptr;

NodeQ\* current = front;

while (current != nullptr && current->data != val) {

prev = current;

current = current->next;

}

if (current != nullptr) {

if (current == front) {

front = front->next;

rear->next = current;

rear = current;

rear->next = nullptr;

}

else if (current != rear) {

prev->next = current->next;

rear->next = current;

rear = current;

rear->next = nullptr;

}

}

}

};

void testAdd() {

OrderedList orderedList;

Queue queue;

LARGE\_INTEGER frequency;

LARGE\_INTEGER time\_start, time\_finish;

QueryPerformanceFrequency(&frequency);

srand(static\_cast<unsigned int>(time(nullptr)));

double sumS = 0;

double sumR = 0;

for (int i = 0; i < 64; ++i) {

int randomNumber = 10000000 + rand() % 90000000;

QueryPerformanceCounter(&time\_start);

orderedList.insert(randomNumber);

Sleep(10);

QueryPerformanceCounter(&time\_finish);

double timeList = static\_cast<double>(time\_finish.QuadPart - time\_start.QuadPart) / frequency.QuadPart;

QueryPerformanceCounter(&time\_start);

queue.enqueue(randomNumber);

Sleep(10);

QueryPerformanceCounter(&time\_finish);

double timeQueue = static\_cast<double>(time\_finish.QuadPart - time\_start.QuadPart) / frequency.QuadPart;

double difference = (timeList - timeQueue) / max(timeList, timeQueue);

sumS += difference;

if (timeList > timeQueue) {

sumR += 1;

}

else {

sumR -= 1;

}

}

sumS = (sumS / 64) \* 100;

sumR = (sumR / 64) \* 100;

std::cout << "SumS: " << sumS << std::endl;

std::cout << "SumR: " << sumR << std::endl;

}

void testRemove() {

OrderedList orderedList;

Queue queue;

LARGE\_INTEGER frequency;

LARGE\_INTEGER time\_start, time\_finish;

QueryPerformanceFrequency(&frequency);

srand(static\_cast<unsigned int>(time(nullptr)));

double sumS = 0;

double sumR = 0;

std::vector<int> addedNumbers;

for (int i = 0; i < 64; ++i) {

int randomNumber = 10000000 + rand() % 90000000;

orderedList.insert(randomNumber);

queue.enqueue(randomNumber);

addedNumbers.push\_back(randomNumber);

}

for (int i = 0; i < 64; ++i) {

int numberToRemove = addedNumbers[i];

QueryPerformanceCounter(&time\_start);

orderedList.remove(numberToRemove);

Sleep(10);

QueryPerformanceCounter(&time\_finish);

double timeList = static\_cast<double>(time\_finish.QuadPart - time\_start.QuadPart) / frequency.QuadPart;

QueryPerformanceCounter(&time\_start);

queue.searchAndMoveToEnd(numberToRemove);

Sleep(10);

QueryPerformanceCounter(&time\_finish);

double timeQueue = static\_cast<double>(time\_finish.QuadPart - time\_start.QuadPart) / frequency.QuadPart;

double difference = (timeList - timeQueue) / max(timeList, timeQueue);

sumS += difference;

if (timeList > timeQueue) {

sumR += 1;

}

else {

sumR -= 1;

}

}

sumS = (sumS / 64) \* 100;

sumR = (sumR / 64) \* 100;

std::cout << "SumS: " << sumS << std::endl;

std::cout << "SumR: " << sumR << std::endl;

}

void testSearch() {

OrderedList orderedList;

Queue queue;

LARGE\_INTEGER frequency;

LARGE\_INTEGER time\_start, time\_finish;

QueryPerformanceFrequency(&frequency);

srand(static\_cast<unsigned int>(time(nullptr)));

double sumS = 0;

double sumR = 0;

std::vector<int> addedNumbers;

for (int i = 0; i < 64; ++i) {

int randomNumber = 10000000 + rand() % 90000000;

orderedList.insert(randomNumber);

queue.enqueue(randomNumber);

addedNumbers.push\_back(randomNumber);

}

for (int i = 0; i < 64; ++i) {

int numberToSearch = addedNumbers[i];

QueryPerformanceCounter(&time\_start);

bool foundInList = orderedList.search(numberToSearch);

Sleep(10);

QueryPerformanceCounter(&time\_finish);

double timeList = static\_cast<double>(time\_finish.QuadPart - time\_start.QuadPart) / frequency.QuadPart;

QueryPerformanceCounter(&time\_start);

queue.searchAndMoveToEnd(numberToSearch);

Sleep(10);

QueryPerformanceCounter(&time\_finish);

double timeQueue = static\_cast<double>(time\_finish.QuadPart - time\_start.QuadPart) / frequency.QuadPart;

double difference = (timeList - timeQueue) / max(timeList, timeQueue);

sumS += difference;

if (timeList > timeQueue) {

sumR += 1;

}

else {

sumR -= 1;

}

// Перевірка, чи елемент знайдено в списку і черзі

if (foundInList != (timeQueue > 0)) {

std::cout << "Mismatch in search results for number: " << numberToSearch << std::endl;

}

}

sumS = (sumS / 64) \* 100;

sumR = (sumR / 64) \* 100;

std::cout << "SumS: " << sumS << std::endl;

std::cout << "SumR: " << sumR << std::endl;

}

int main() {

testAdd();

testRemove();

testSearch();

return 0;

}

1. **Порівняння результатів**

Для оцінки параметрів S та R був використан клас із бібліотеки windows.h QueryPerformanceCounter. Він дозволив розрахувати час виконання операцій додавання, пошуку та видалення у дані структури.

Також, для того, щоб не відбулася помилка ділення на нуль, бо операція може виконатися настільки швидко, що таймер поверне 0, у проміжок, час якого вимірювався була додана команда Sleep(10), яка додавала зайвий константний час до виконання операцій. Так як вимірюється відносне значення і цей час константний для обох структур, він не буде впливати на вихідний результат і допоможе уникнути помилки ділення на нуль.

Весь розрахунок проводиться у методах драйверах, тому на виході, у консолі, були отримані готові дані, які й будуть занесені у таблицю.

Таблиця 1

Результати розрахунку показників обчислювальної складності

|  |  |  |
| --- | --- | --- |
| Операція | Показники обчислювальної складності | |
| Оцінка S- | Оцінка R |
| Додавання | -0.19 % | 46.87 % |
| Видалення | 0.28 % | 54.68 % |
| Пошук | -0.83 % | 39.06 % |

В операції додавання перевагу має список. Вона дуже мала, але помітна.

У операції видалення краще себе показує черга, так як значення S додатне а R > 50%.

Але і в операції пошуку краще себе показує список.

Із наведених даних, можна дійти висновку, що список є більш ефективною структурою ніж черга, хоча ця перевага досить невелика.

**Висновок**

Під час виконання лабораторної роботи було проведено порівняльний аналіз ефективності двох структур даних: списку і черги. Результати розрахунків параметрів обчислювальної складності (S і R) показали, що у більшості операцій, таких як додавання, видалення і пошук, список має деяку перевагу над чергою.

У операції додавання, список показав невелику перевагу, хоча ця перевага була дуже мала (0.19% за S), але все одно помітна. У випадку операції видалення, список виявився ефективнішим, з показниками S близькими до нуля та R більшими за 50%. Проте, у операції пошуку, список також виявився кращим варіантом, хоча перевага була менш помітною (S -0.83% та R 39.06%).

Загалом, результати дослідження свідчать про те, що у даному контексті список є більш оптимальним вибором для багатьох операцій порівняно з чергою. Однак, важливо враховувати, що в реальних ситуаціях вибір структури даних також залежить від конкретних завдань і вимог проекту, і цей аналіз може бути використаний для прийняття обґрунтованих рішень щодо вибору структури даних для конкретного застосування.