**Лесечко Олеся(ДА-02)**

Лабораторна робота №3

“Дослідження структур даних пріоритетна черга та купа”

**Мета роботи:** Ознайомитись і дослідити структуру даних бінарна купа. Набути навичок реалізації абстрактної структури даних черга з пріоритетом на основі бінарної купи мовою програмування С++. Ознайомитись з механізмом перевантаження

операторів та дослідити особливості сортування купою (пірамідального сортування). Порівняти власну реалізацію пріоритетної черги з готовим бібліотечним рішенням STL.**Варіант: Задача 1**

Впродовж тижня студент отримує багато домашніх завдань. Кожне завдання характеризується цікавістю, рівнем корисності предмета, важкістю виконання, дедлайном тощо. Студент за певним критерієм повинен обрати найбільш оптимальне завдання та почати його виконувати, після цього перейти до наступного завдання. Потрібно врахувати, що впродовж тижня у студента можуть з’являтися нові завдання з більш високим пріоритетом.

Хід виконання роботи:

A) УМОВА

1. Створити структуру для зберігання об’єктів з характеристиками відповідно до обраної задачі:

• Обрати певний власний критерій для визначення більш пріоритетного об’єкту на основі його характеристик

• Перевантажити оператор “менше” (operator<) для порівняння двох об’єктів за критерієм.

• Перевантажити оператор “дорівнює” (operator==) для визначення рівності двох об’єктів за цим самим критерієм

2. Реалізувати пріоритетну чергу на основі незростаючої бінарної купи:

2.1 Створити структуру PriorityQueue, яка буде містити в собі бінарне дерево для роботи бінарної купи. Для зберігання бінарного дерева можна використати звичайний динамічний масив, що вже був реалізований у минулих роботах.

2.2 Реалізувати основні методи переміщення по бінарному дереву:

• getParent(index) знайти індекс батьківського вузла

• getLeftChild(index) знайти індекс лівого дочірнього вузла

• getRightChild(index) знайти індекс правого дочірнього вузла

2.3 Реалізувати внутрішні методи для підтримки властивості незростаючої (max-heap) купи:

• siftUp(index) просіяти елемент вверх по дереву

• siftDown(index) просіяти елемент вниз по дереву

2.4 Реалізувати основні методи для роботи пріоритетної черги:

• push(object) додати новий елемент в чергу

• top() отримати верхній елемент з черги

• pop() видалити верхній елемент з черги

• size() знайти кількість елементів в черзі

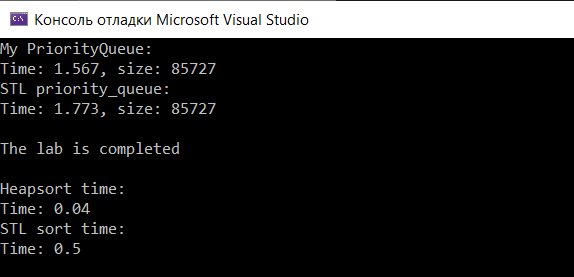
• empty() перевірити чергу на пустоту

3. Провести тестування, використавши вказану нижче функцію testPriorityQueue(). Перевірити правильність та швидкість роботи, порівнявши з готовим бібліотечним рішенням STL priority\_queue.

4. Продумати реалізацію сортування купою (пірамідального сортування) на базі бінарної купи, вміти обгрунтовано пояснити та показувати в коді.

Додаткові завдання: Реалізувати сортування купою, порівняти його з готовим бібліотечним рішенням STL std::sort() або з іншими алгоритмами сортувань, реалізованими в минулих роботах, – провести тестування з замірами часу на різних вхідних даних, зробити аналіз та висновки.

B) СКРІНШОТ РЕЗУЛЬТАТУ



С) ЛІСТИНГ ПРОГРАМИ (КОД)

**main.cpp**

#include <iostream>

#include "tests.h"

using namespace std;

int main() {

testPriorityQueue();

testSortPriorityQueue();

return 0;

}

**Data.h**

#pragma once

#include <iostream>

struct Data {

int interest; // інтерес до завдання від 1 до 10

int utility; // користь від завдання від 1 до 10

int complexity; // скільки днів потрібно для виконання завдання

int deadline; // днів до встановленого терміну

Data() {

interest = rand() % 10;

utility = rand() % 10;

complexity = rand() % 10;

deadline = rand() % 10;

}

Data(int tmpInterest, int tmpUtility, int tmpComplexity, int tmpDeadline) {

interest = tmpInterest;

utility = tmpUtility;

complexity = tmpComplexity;

deadline = tmpDeadline;

}

};

bool operator<(const Data& first, const Data& second);

bool operator>(const Data& first, const Data& second);

bool operator==(const Data& first, const Data& second);

bool myCompare(Data x, Data y);

**PriorityQueue.h**

#pragma once

#include <stdlib.h>

#include "Data.h"

struct binaryTree {

Data Data;

int Parent = -1;

int LeftChild = -1;

int RightChild = -1;

};

struct PriorityQueue {

int capacity;

int realsize;

binaryTree\* innerArray;

PriorityQueue() {

capacity = 1;

realsize = 0;

innerArray = new binaryTree[capacity];

}

~PriorityQueue() {

delete[] innerArray;

}

//методи переміщення по бінарному дереву

int getParent(int index);

int getLeftChild(int index);

int getRightChild(int index);

//методи для роботи пріоритетної черги

void push(Data Data);

Data top();

Data pop();

int size();

bool empty();

void print();

//методи для підтримки властивості незростаючої(max - heap) купи

void siftUp(int index);

void siftDown(int index);

};

**Data.cpp**

#include "Data.h"

bool operator<(const Data& first, const Data& second) {

if (first.interest != second.interest) {

return first.interest < second.interest;

}

if (first.utility != second.utility) {

return first.utility < second.utility;

}

if (first.complexity != second.complexity) {

return first.complexity < second.complexity;

}

return first.deadline < second.deadline;

}

bool operator>(const Data& first, const Data& second) {

if (first.interest != second.interest) {

return first.interest > second.interest;

}

if (first.utility != second.utility) {

return first.utility > second.utility;

}

if (first.complexity != second.complexity) {

return first.complexity > second.complexity;

}

return first.deadline > second.deadline;

}

bool operator==(const Data& first, const Data& second) {

if (first.deadline == second.deadline && first.interest == second.interest && first.complexity == second.complexity && first.utility == second.utility) {

return true;

}

return false;

}

bool myCompare(Data x, Data y) {

return x > y;

}

**PriorityQueue.cpp**

#include "PriorityQueue.h"

#include <iostream>

using namespace std;

int PriorityQueue::getParent(int index) {

if (index == 0) {

return -1;

}

return ((index - 1) / 2);

//return (int(floor((index \* 1.0 - 1) / 2)));

}

int PriorityQueue::getLeftChild(int index) {

return 2 \* index + 1;

}

int PriorityQueue::getRightChild(int index) {

return 2 \* index + 2;

}

void PriorityQueue::push(Data Data) {

if (realsize == capacity) {

capacity \*= 2;

binaryTree\* old = innerArray;

innerArray = new binaryTree[capacity];

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

innerArray[i] = old[i];

}

delete[] old;

}

innerArray[realsize].Data = Data;

innerArray[realsize].Parent = getParent(realsize);

innerArray[realsize].LeftChild = getLeftChild(realsize);

innerArray[realsize].RightChild = getRightChild(realsize);

siftUp(realsize);

realsize++;

}

Data PriorityQueue::top() {

if (empty()) {

return innerArray[0].Data;

}

return {0,0,0,0};

}

Data PriorityQueue::pop() {

if (empty()) {

Data deleted = innerArray[0].Data;

realsize--;

swap(innerArray[0].Data, innerArray[realsize].Data);

innerArray[realsize].Data = { 0,0,0,0 };

siftDown(0);

return deleted;

}

return {0, 0, 0, 0};

}

int PriorityQueue::size() {

return realsize;

}

bool PriorityQueue::empty() {

return (size() > 0);

}

void PriorityQueue::print() {

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

cout << "index: " << i << " " << "Data { " << innerArray[i].Data.interest << " " << innerArray[i].Data.utility << " " << innerArray[i].Data.complexity << " " << innerArray[i].Data.deadline << " }"<< endl;

}

}

void PriorityQueue::siftUp(int index) {

int idParent = innerArray[index].Parent;

while (innerArray[index].Data > innerArray[idParent].Data && index != 0) {

swap(innerArray[index].Data, innerArray[idParent].Data);

index = idParent;

idParent = innerArray[index].Parent;

}

}

void PriorityQueue::siftDown(int index) {

while (1) {

int left = innerArray[index].LeftChild;

int right = innerArray[index].RightChild;

int idSwap = 0;

if (2 \* index + 1 >= realsize) {

break;

}

if (innerArray[index].Data > innerArray[left].Data && innerArray[index].Data > innerArray[right].Data) {

break;

}

if (innerArray[right].Data < innerArray[left].Data) {

idSwap = left;

}

else {

idSwap = right;

}

swap(innerArray[index].Data, innerArray[idSwap].Data);

index = idSwap;

}

}

**sort.cpp**

#include "sort.h"

#include <iostream>

using namespace std;

int getParent(int index) {

if (index == 0) {

return -1;

}

return ((index - 1) / 2);

}

int getLeftChild(int index) {

return 2 \* index + 1;

}

int getRightChild(int index) {

return 2 \* index + 2;

}

void siftDown(int index,int realsize, Data\* arr) {

while (1) {

int left = getLeftChild(index);

int right = getRightChild(index);

int idSwap = 0;

if (2 \* index + 1 >= realsize) {

break;

}

if (arr[index] > arr[left] && arr[index] > arr[right]) {

break;

}

if (arr[right] < arr[left]) {

idSwap = left;

}

else {

idSwap = right;

}

swap(arr[index], arr[idSwap]);

index = idSwap;

}

}

void Heapsort(Data\* arr, int N) {

for (int i = N; i >= 0; i--) {

int left = getLeftChild(i);

int right = getRightChild(i);

if (right < N) {

siftDown(i, N, arr);

}

else if (left < N) {

siftDown(i, N, arr);

}

}

}

Висновки:

Ознайомилась з перезавантаженням операторів, зробила перезавантаження для порівняння двох об’єктів та для визначення рівності. Прийшла до висновку, що перезавантаження нам потрібно для того щоб спростити дії роботи структури, тобто завдяки перезавантаженню операторів ми можемо виконувати зі своєю структурою ті ж операції, що і з вбудованими типами даних.

Дізналася про чергу з пріоритетом, різниця між нею та звичайною чергою в тому, що кожен елемент має пріоритет. Реалізувала її в коді купою.

Також реалізувала в коді пірамідальне сортування, яке працює гарантовано за n log n при сортуванні n елементів, це сортування використовує бінарне дерево, де в будь-яка вершина більша за своїх нащадків. Недолік цього сортування в тому, що на майже відсортованих даних працює так само, як і на хаотичних.