

Алгоритми та структури даних

**Лабораторна робота №7**

**“Дослідження структур даних пріоритетна черга та купа”**

Робота : Лещинського Б.Д. группа КА-07

08.04.2021

Варіант 13

**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.

**Хід виконання роботи:**

**Завдання:**

Обрати одну із запропонованих задач та реалізувати пріоритетну чергу для ефективнного вирішення цієї задачі (обрано варіант з чергою пацієнтів).

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.

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.

**Код програми:**

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <queue>

using namespace std;

struct Data {

int PatientId;

int SurvivingChance;

int Degree;

Data() {

PatientId = rand() % 1001;

SurvivingChance = rand() % 100;

Degree = rand() % 10;

}

bool operator <(const Data other ) const {

return (this->SurvivingChance + this->Degree) < (other.SurvivingChance + other.Degree);

}

bool operator >(const Data other) const {

return !(\*this < other);

}

bool operator ==(const Data other) const {

return (this->SurvivingChance + this->Degree) == (other.SurvivingChance + other.Degree);

}

};

struct PriorityQueue {

Data\* heapArr = new Data[1];

int entities = 0;

int Size = 1;

int getParent(int ind) {

return (ind - 1) / 2;

}

int getLeftChild(int ind) {

return ind \* 2 + 1;

}

int getRightChild(int ind){

return ind \* 2 + 2;

}

void siftUp(int ind) {

while(ind!=0 && heapArr[ind] > heapArr[getParent(ind)]){

Data swap = heapArr[getParent(ind)];

heapArr[getParent(ind)] = heapArr[ind];

heapArr[ind] = swap;

ind = getParent(ind);

}

}

void siftDown(int ind) {

while (ind < entities / 2) {

int maxInd = getLeftChild(ind);

if (getRightChild(ind) < entities && heapArr[getRightChild(ind)] > heapArr[getLeftChild(ind)]) {

maxInd = getRightChild(ind);

}

if (heapArr[ind] > heapArr[maxInd] || heapArr[ind] == heapArr[maxInd]) {

return;

}

Data swap = heapArr[maxInd];

heapArr[maxInd] = heapArr[ind];

heapArr[ind] = swap;

ind = maxInd;

}

}

void push(Data patient) {

if (entities + 1 == Size) {

Data\* newArr = new Data[Size \* 2];

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

newArr[i] = heapArr[i];

}

delete[] heapArr;

heapArr = newArr;

Size \*= 2;

}

heapArr[entities] = patient;

siftUp(entities);

entities++;

}

Data top() {

return heapArr[0];

}

void pop() {

Data swap = heapArr[0];

heapArr[0] = heapArr[entities - 1];

heapArr[entities - 1] = swap;

entities--;

siftDown(0);

}

int size() {

return entities;

}

bool empty() {

return entities == 0;

}

};

template <typename T>

float testPriorityQueueSpeed(T&& priorityQueue)

{

const int iters = 100000;

clock\_t timeStart = clock();

for (int i = 0; i < iters; i++)

{

int insertDataAmount = rand() % 6 + 5;

for (int j = 0; j < insertDataAmount; j++)

{

priorityQueue.push(Data());

}

priorityQueue.top();

priorityQueue.pop();

}

clock\_t timeEnd = clock();

float time = (float(timeEnd - timeStart)) / CLOCKS\_PER\_SEC;

return time;

}

bool testPriorityQueue()

{

srand(time(NULL));

const int iters = 20000;

PriorityQueue myPriorQueue;

priority\_queue<Data> stlPriorQueue;

bool isDataEqual = true;

for (int i = 0; i < iters; i++)

{

int insertDataAmount = rand() % 6 + 5;

for (int j = 0; j < insertDataAmount; j++)

{

Data randData = Data();

myPriorQueue.push(randData);

stlPriorQueue.push(randData);

}

if (!(myPriorQueue.top() == stlPriorQueue.top()))

{

isDataEqual = false;

cerr << "Comparing failed on iteration " << i << endl << endl;

break;

}

int removeDataAmount = rand() % insertDataAmount;

for (int j = 0; j < removeDataAmount; j++)

{

myPriorQueue.pop();

stlPriorQueue.pop();

}

}

int myQueueSize = myPriorQueue.size();

int stlQueueSize = stlPriorQueue.size();

float stlTime =

testPriorityQueueSpeed<priority\_queue<Data>>(priority\_queue<Data>());

float myTime = testPriorityQueueSpeed<PriorityQueue>(PriorityQueue());

cout << "My PriorityQueue:" << endl;

cout << "Time: " << myTime << ", size: " << myQueueSize << endl;

cout << "STL priority\_queue:" << endl;

cout << "Time: " << stlTime << ", size: " << stlQueueSize << endl << endl;

if (isDataEqual && myQueueSize == stlQueueSize)

{

cout << "The lab is completed" << endl << endl;

return true;

}

cerr << ":(" << endl << endl;

return false;

}

// heapsort

int getParent(int ind) {

return (ind - 1) / 2;

}

int getLeftChild(int ind) {

return ind \* 2 + 1;

}

int getRightChild(int ind) {

return ind \* 2 + 2;

}

void siftDown(int\* arr, int arrSize, int ind) {

while (ind < arrSize / 2) {

int maxInd = getLeftChild(ind);

if (getRightChild(ind) < arrSize && arr[getRightChild(ind)] > arr[getLeftChild(ind)]) {

maxInd = getRightChild(ind);

}

if (arr[ind] > arr[maxInd] || arr[ind] == arr[maxInd]) {

return;

}

int swap = arr[maxInd];

arr[maxInd] = arr[ind];

arr[ind] = swap;

ind = maxInd;

}

}

void HeapSort(int\* arr, int arrSize) {

//building heap:

for (int i = arrSize / 2 - 1; i >= 0; i--) {

//siftDown for each vertice that has at least one successor

siftDown(arr, arrSize, i);

}

//sorting itself:

for (int i = 0; i < arrSize - 1; i++) {

int swap1 = arr[0];

arr[0] = arr[arrSize - 1 - i];

arr[arrSize - 1 - i] = swap1;

siftDown(arr, arrSize - i - 1, 0);

}

}

bool arrEqual(int\* arr1, int\* arr2, int arraySize) {

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

if (arr1[i] != arr2[i]) {

cout << "Not equal on iteration: " << i << endl;

return false;

}

}

cout << "The given arrays are equal";

return true;

}

int main(){

double sTime, fTime, resultTime = 0;

srand(time(NULL));

testPriorityQueue();

int arr1[100000];

int arr2[100000];

/\*for (int i = 0; i < 100000; i++) {

arr1[i] = arr2[i] = rand();

}\*/

for (int i = 0; i < 100000; i++) {

arr1[i] = arr2[i] = rand(); //i//(i % 500 == 0) ? i + 50 : i; <--- different cases of starting data

}

cout << "Testing handmade heapSort against STL stable\_sort() function on the array \nfilled with the same random elements (size 100000 elem.): \n";

cout << endl;

sTime = double(clock());

HeapSort(arr1, 100000);

fTime = double(clock());

resultTime += (fTime - sTime) / CLOCKS\_PER\_SEC;

cout << "Heap array sorting time: " << resultTime << endl;

cout << endl;

sTime = double(clock());

stable\_sort(begin(arr2), end(arr2));//sort(begin(arr2), end(arr2));

fTime = double(clock());

resultTime = (fTime - sTime) / CLOCKS\_PER\_SEC;

cout << "STL sort() time: " << resultTime << endl;

cout << endl;

cout << "Checking arrays` equality elementwise after sortings: \n";

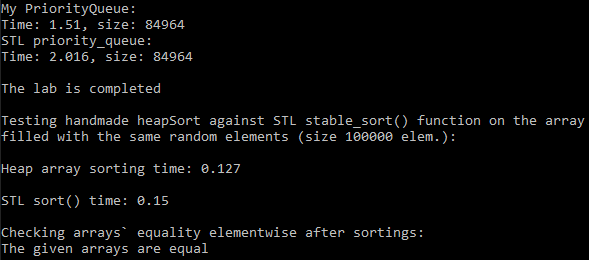
arrEqual(arr1, arr2, 20);

cout << endl;

return 0;

}

**Результати роботи програми:**



Висновки:

Я ознайомився і дослідив структуру даних бінарна купа. Набув навичок реалізації абстрактної структури даних черга з пріоритетом на основі бінарної купи мовою програмування С++. Ознайомився з механізмом перевантаження операторів та дослідив особливості сортування купою (пірамідального сортування). Порівняти власну реалізацію пріоритетної черги з готовим бібліотечним рішенням STL (stable\_sort()).