

Звіт

Лабораторна робота №3

“Дослідження структур даних динамічний масив та зв’язний список”

Студента групи ДA-12

Краковича Павла Дмитровича

Київ – 2022

1. Мета роботи

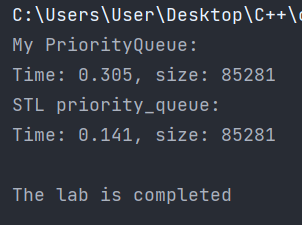
Ознайомитись і дослідити структуру даних бінарна купа. Набути навичок реалізації абстрактної структури даних черга з пріоритетом на основі бінарної купи мовою програмування С++. Ознайомитись з механізмом перевантаження операторів та дослідити особливості сортування купою (пірамідального сортування). Порівняти власну реалізацію пріоритетної черги з готовим бібліотечним рішенням STL.

2. Варіант роботи - №2

3. Хід виконання роботи

№1

* Перед ботом з’являються вороги в зоні видимості. Кожен ворог характеризується рівнем здоров’я, можливим уроном, скілом гравця тощо. Бот за певним критерієм повинен обрати найбільш оптимальну для себе ціль та знищити її, після цього перейти до наступної цілі. Потрібно врахувати, що впродовж гри можуть з’являтися нові вороги з більш високим пріоритетом.
* Результат роботи програми:



* Лістинг(код) програми:
  + main.cpp

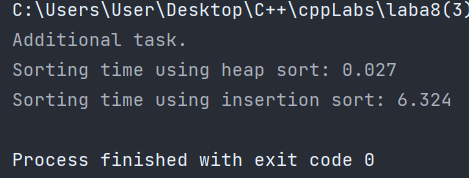
#include <queue>  
#include <cstdlib>  
#include <ctime>  
#include <iostream>  
#include "PriorityQueue.h"  
  
using namespace std;  
  
template <typename T>  
  
float testPriorityQueueSpeed(T&& priorityQueue)  
{  
 const int iters = 100000;  
 clock\_t timeStart = clock();  
 for (int i = 0; i < 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(nullptr));  
 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;  
}  
  
int main() {  
 testPriorityQueue();  
 return 0;  
}

* + PriorityQueue.h

//  
// Created by User on 12.06.2022.  
//  
  
#ifndef LABA8\_PRIORITYQUEUE\_H  
#define LABA8\_PRIORITYQUEUE\_H  
  
#include <vector>  
  
using namespace std;  
  
struct Data  
{  
 float distance;  
 float health;  
  
 Data()  
 : distance(static\_cast<float>(rand()))  
 , health(static\_cast<float>(rand()))  
 {}  
  
  
  
 bool operator < (const Data& secondUnit) const {  
 if (distance != secondUnit.distance) {  
 return distance < secondUnit.distance;  
 }  
 return health < secondUnit.health;  
 }  
 bool operator == (const Data& secondUnit) const {  
 return distance == secondUnit.distance;  
 }  
 bool operator != (const Data& secondUnit) const {  
 return distance != secondUnit.distance;  
 }  
 bool operator > (const Data& secondUnit) const {  
 if (distance != secondUnit.distance) {  
 return distance > secondUnit.distance;  
 }  
 return health > secondUnit.health;  
 }  
};  
  
  
struct PriorityQueue {  
  
 int Size;  
 vector<Data> queue;  
  
 PriorityQueue():Size(0) {}  
  
 int size() const {  
 return Size;  
 }  
  
 bool isEmpty() const {  
 return size() == 0;  
 }  
  
 static int getParent(int index) {  
 return (index - 1) / 2;  
 }  
  
 static int getLeftChild(int index) {  
 return (2 \* index + 1);  
 }  
  
 static int getRightChild(int index) {  
 return (2 \* index + 2);  
 }  
  
 int getMaxChildIndex(int index) {  
 if(queue.at(getLeftChild(index)) < queue.at(getRightChild(index))) {  
 return getRightChild(index);  
 }  
 return getLeftChild(index);  
 }  
  
 static bool isRoot(int index) {  
 return index <= 0;  
 }  
  
 bool isLeaf(int index) const {  
 return getLeftChild(index) >= size();  
 }  
  
 void siftUp(int index) {  
 if (isRoot(index)) {  
 return;  
 }  
 Data& self = queue[index];  
 Data& parent = queue[getParent(index)];  
 if (!(self < parent)) {  
 swap(self, parent);  
 siftUp(getParent(index));  
 }  
 }  
   
 void siftDown(int index) {  
 if (isLeaf(index)) {  
 return;  
 }  
  
 int maxChildIndex;  
 if (getLeftChild(index) + 1 < Size) {  
 maxChildIndex = getMaxChildIndex(index);  
 }  
 else {  
 maxChildIndex = getLeftChild(index);  
 }  
  
 if (queue[maxChildIndex] < queue[index]) {  
 return;  
 }  
 swap(queue[maxChildIndex], queue[index]);  
 siftDown(maxChildIndex);  
 }  
  
 void push(const Data& data) {  
 queue.push\_back(data);  
 Size++;  
 siftUp(Size - 1);  
 }  
  
 Data top() {  
 return queue.front();  
 }  
  
 void pop() {  
 if (isEmpty()) {  
 return;  
 }  
 iter\_swap(queue.begin(), queue.end()-1);  
 Size--;  
 queue.erase(queue.end() - 1);  
 siftDown(0);  
 }  
};  
  
#endif //LABA8\_PRIORITYQUEUE\_H

№2(Додаткове 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, зробити аналіз та висновки.
* Результат роботи програми:



* Лістинг(код) програми:
  + main.cpp

#include <iostream>  
#include "heapSort.h"  
#include "insertionSort.h"  
#include <ctime>  
using namespace std;  
  
void randomPermutation(int \*arr, int size) {  
 for(int j,i=size-1;i>=0;i--) {  
 j = rand() % size;  
 swap(arr[i],arr[j]);  
 }  
}  
  
int main()  
{  
 cout << "Additional task." << endl;  
 srand(time(nullptr));  
 int size = 50000;  
 int \*arr = new int[size];  
 for (int i = 0; i <size; i++) {  
 arr[i]=i+1;  
 }  
 randomPermutation(arr,size);  
  
 clock\_t HeapTimeStart = clock();  
 heapSort(arr, size);  
 clock\_t HeapTimeEnd = clock();  
 float HeapTime = (float(HeapTimeEnd - HeapTimeStart)) / CLOCKS\_PER\_SEC;  
 cout << "Sorting time using heap sort: " << HeapTime << endl;  
 randomPermutation(arr,size);  
  
 clock\_t StlTimeStart = clock();  
 insertionSort(arr, size);  
 clock\_t StlTimeEnd = clock();  
 float StlTime = (float(StlTimeEnd - StlTimeStart)) / CLOCKS\_PER\_SEC;  
 cout << "Sorting time using insertion sort: " << StlTime << endl;  
}

* + heapSort.h

//  
// Created by User on 12.06.2022.  
//  
  
#ifndef TASK\_2\_HEAPSORT\_H  
#define TASK\_2\_HEAPSORT\_H  
  
using namespace std;  
  
  
  
int getLeftChild(int index) {  
 return (2 \* index + 1);  
}  
  
int getRightChild(int index) {  
 return (2 \* index + 2);  
}  
  
bool isLeaf(int index, int size) {  
 return getLeftChild(index) >= size;  
}  
  
int getMaxChildIndex(int index, int\* arr) {  
 if(arr[getLeftChild(index)] < arr[getRightChild(index)]) {  
 return getRightChild(index);  
 }  
 return getLeftChild(index);  
}  
  
void siftDownHeap(int\* arr, int size, int index) {  
 if (isLeaf(index,size)) {  
 return;  
 }  
  
 int maxChildIndex;  
 if (getLeftChild(index) + 1 < size) {  
 maxChildIndex = getMaxChildIndex(index,arr);  
 }  
 else {  
 maxChildIndex = getLeftChild(index);  
 }  
  
 if (maxChildIndex < index) {  
 return;  
 }  
 swap(arr[maxChildIndex], arr[index]);  
 siftDownHeap(arr,size,maxChildIndex);  
}  
  
void heapSort(int\* arr,int size) {  
 for (int i = size / 2 - 1; i >= 0; i--) {  
 siftDownHeap(arr, size, i);  
 }  
  
 for (int i = size - 1; i > 0; i--) {  
 // Move current root to end  
 swap(arr[0], arr[i]);  
 // call max heapify on the reduced heap  
 siftDownHeap(arr, i, 0);  
 }  
}  
  
  
#endif //TASK\_2\_HEAPSORT\_H

* + insertionSort.h

//  
// Created by User on 20.06.2022.  
//  
  
#ifndef MAIN\_CPP\_INSERTIONSORT\_H  
#define MAIN\_CPP\_INSERTIONSORT\_H  
  
  
void insertionSort(int \*arr,int size) {  
 for(int i=1;i<size;i++) {  
 for(int j=i-1;j>=0;j--) {  
 if( arr[j]>arr[j+1]) {  
 swap(arr[j],arr[j+1]);  
 }  
 }  
 }  
}  
  
  
#endif //MAIN\_CPP\_INSERTIONSORT\_H

Висновки

В процесі виконання лабораторної роботи №3 я ознайомився, дослідив та реалізував структуру даних «Бінарна купа». Набув на її основі навичок з реалізації абстрактної структури даних «Черга з пріоритетом».

Додатково, для написання черги, навчився перенавантажувати оператори в мові C++ та виконавши додаткове завдання, дослідив особливості та реалізував сортування купою.

Під час виконання додаткового завдання я використав свою реалізацію сортування вставками для порівняння з сортуванням купою.

Звісно, сортування купою набагато ефективніше, тому час сортування так сильно відрізняється. Для випадкового тасування я використав свою реалізацію алгоритму «Фішера-Йетса» з минулих лабораторних робіт.

Контрольні запитання

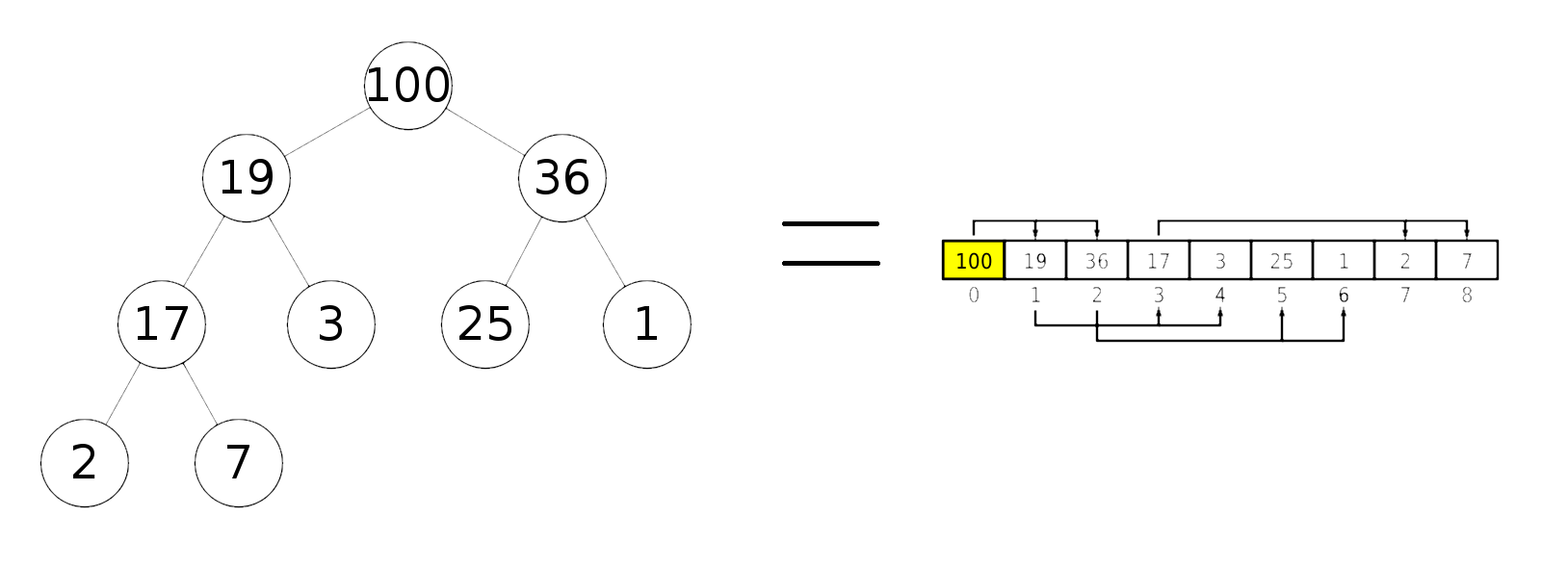
1) Навіщо потрібна структура даних пріоритетна черга? Чим вона відрізняється від звичайної черги?

*Пріоритетна черга - це абстрактна структура даних на кшталт стека або черги, де кожен елемент має пріоритет. Елемент із більш високим пріоритетом знаходиться перед елементом із нижчим пріоритетом. Якщо елементи мають однакові пріоритети, вони розташовуються в залежності від своєї позиції в черзі. Приоритетні черги використовуються тоді, коли потрібно не лише зберігати дані, а й одразу сортувати їх.*

2) Які структури даних із вже відомих можна було б застосувати для вирішення поставлених задач, у чому їх недоліки?

*Аналогом для виконання даного завдання є комбінація динамічного масива, який сортується та елементи шукаються у ньому завдяки бінарному пошуку. Однак, додавання та видалення елементів буде виконуватися довше, ніж у приоритетній черзі, побудованій на бінарній купі.*

3) Що таке бінарне дерево? Намалюйте його, опишіть на ньому особливості бінарної купи.

*Бінарне, або двійкове дерево — це деревоподібна структура даних, що складається з вузлів, кожен з яких має не більше двох дочірніх елементів, званих лівим та правим вузлами. Дерево починається з одного вузла, відомого як корінь. Кожен елемент, що має хоч один дочірній елемент називається вузлом. Вузли бувають двох видів: лівий та правий. Елементи, які не мають дочірніх елементів назива.ться листками. *

*Бінарнка купа — це структура даних, що є масивом, який можна розглядати як двійкове дерево. Кожен вузол цього дерева відповідає певному елементу масива. На всіх рівнях, крім, можливо останнього, дерево повністю заповнене (заповнений рівень — такий, що містить максимально можливу кількість вузлів). Останній рівень заповнюється послідовно зліва направо до тих пір, доки в масиві не закінчаться елементи.*

*Основною ідеєю бінарної купи є те, що вона є бінарним деревом, яке задовільняє трьом правилам:*

* *Значення кожного вузла більше за значення його дочірніх вузлів*
* *Кількість елементів на кожному рівні дорівнює , де і – номер рівня (окрім останнього)*
* *Нові елементи в останній рівень додаються зліва направо.*

4) Яким чином бінарна купа представляється у вигляді масиву? Які ще є способи представлення бінарного дерева у пам’яті?

*Двійкова купа зазвичай видається як масив. Подання робиться так:*

*Кореневий елемент буде знаходиться за індексом 0, а i – рівень дерева.*

*(i-1)/2 – Повертає батьківський вузол до поточного вузла*

*(2\*i)+1 – Повертає лівий дочірній вузол поточного вузла*

*(2\*i)+2 – Повертає правильний дочірній вузол поточного вузла*

5) Поясніть процес побудови купи та просіювання елементів у незростаючій (max-heap) / неспадній (min-heap) бінарній купі.

*У Min-Heap ключ, присутній у вузлі батька, повинен бути меншим або рівним серед ключів, присутніх у всіх його дочірніх. Та сама властивість має бути рекурсивно істинною для всіх піддерев у цьому бінарному дереві. Під час просіювання вниз, найменший елемент має пріоритет.*

*У Max-Heap ключ, присутній у вузлі батька, повинен бути більшим або рівним серед ключів, присутніх у всіх його дочірніх. Та сама властивість має бути рекурсивно істинною для всіх піддерев у цьому бінарному дереві. Під час просіювання вниз, набільший елемент має пріоритет.*

*При цьому, у Min-Heap та Max-Heap в корені, відповідно знаходяться мінімальний та максимальний елементи дерева.*

6) Опишіть логіку роботи пірамідального сортування, його асимптотичну складність, переваги та недоліки.

*Пірамідальне сортування (HeapSort) — це метод сортування порівняння, заснований на такій структурі даних, як бінарна куча. Вона похожа на сортировку вибором, де ми спочатку шукаємо максимальний елемент і поміщаємо його в кінцець. Далі ми повторяємо ту же операцію поки у нас є елементи*

*Логіка роботи алгоритму:*

*1. Вибудовуємо елементи масиву у вигляді сортувального дерева:*

*2. Видаляємо елементи з кореня по одному за раз і перебудовуємо дерево N разів.*

*Переваги алгоритму*

* *Час роботи в найгіршому випадку — O(n\*log n)*

*Недоліки алгоритму*

* *Не може працювати з дублікатами даних без модифікації*
* *На майже відсортованих даних працює так само довго, як і на хаотичних даних;*
* *Складний в реалізації;*