Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут ім. Ігоря Сікорського»

Інститут прикладного системного аналізу

Кафедра системного проектування

Алгоритми та структури даних

Лабораторна робота №3

“Дослідження структур даних бінарна купа

та пріоритетна черга”

Аносов Ілля

ДА-21

Київ

2023

**Мета роботи:**

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

**ВАРІАНТ 1**

**Завдання**

1. Створити структуру Data для зберігання об’єктів з характеристиками відповідно до обраної задачі:  Обрати певний власний критерій для визначення більш пріоритетного об’єкту на основі всіх його характеристик

 Перевантажити оператор “менше” (operator<)

 Перевантажити оператор “дорівнює” (operator==)

2. Реалізувати незростаючу (max-heap) бінарну купу:

2.1Створити структуру BinaryHeap, яка буде являти собою бінарне дерево для роботи бінарної купи. Для зберігання бінарного дерева можна використати власний 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 впорядкування купи:

 siftUp(index) – просіяти елемент вверх по дереву

 siftDown(index) – просіяти елемент вниз по дереву

3. Реалізувати пріоритетну чергу на основі бінарної купи.

3.1 Створити структуру PriorityQueue, яка буде містити в собі BinaryHeap.

3.2 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
* clean() – очистити чергу

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

5. Реалізувати сортування купою на базі бінарної купи.

5.1 Створити структуру HeapSort на базі бінарної купи яка працює з типом даних int. Реалізувати потрібні методи:

* buildHeap(int\* array, int arraySize) – побова купи з вхідного масива
* heapSort(int\* array, int arraySize) – сортування масива, всередині буде викликатись buildHeap

5.2 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 (різні довжини масивів, найкращий / середній / найгірший випадки для сортування).

Зображення, що містить текст

Автоматично згенерований опис

prorityQueue.h

#pragma once

#include <vector>

#include <ctime>

using namespace std;

struct Data {

    int curiosity;

    int usefulness;

    int difficulty;

    int deadline;

    Data() {

        curiosity = rand() % 100;

        usefulness = rand() % 100;

        difficulty = rand() % 100;

        deadline = rand() % 21;

    }

    Data(int curiosity, int usefulness, int difficulty, int deadline) :

        curiosity(curiosity),

        usefulness(usefulness),

        difficulty(difficulty),

        deadline(deadline)  {}

    bool operator<(const Data& second) const {

        if (this->deadline != second.deadline)

            return this->deadline > second.deadline;

        if (this->difficulty != second.difficulty)

            return this->difficulty < second.difficulty;

        if (this->usefulness != second.usefulness)

            return this->usefulness < second.usefulness;

        return this->curiosity < second.curiosity;

    }

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

        if (this->deadline != second.deadline)

            return false;

        if (this->difficulty != second.difficulty)

            return false;

        if (this->usefulness != second.usefulness)

            return false;

        return true;

    }

    bool operator>(const Data& second) const {

        return second < \*this;

    }

};

struct MaxHeap {

private:

    int maxChildIndex(int first, int second);

public:

    void swap(int first, int second);

    vector<Data> dataVector;

    void siftUp(int index);

    void siftDown(int index);

    int getParent(int index);

    int getLeftChild(int index);

    int getRightChild(int index);

};

struct PriorityQueue {

private:

    MaxHeap heap;

public:

    void push(Data data);

    Data top();

    void pop();

    void clean();

    bool empty();

    int size();

};

struct HeapSort {

private:

    int\* arrayToChange;

    int maxChildIndex(int first, int second);

    void siftDown(int index, int arraySize);

    void swap(int first, int second);

    int getParent(int index);

    int getLeftChild(int index);

    int getRightChild(int index);

public:

    void buildHeap(int\* array, int arraySize);

    void sortHeap(int\* array, int arraySize);

};

MaxHeap.cpp

#include "priorityQueue.h"

int MaxHeap::getParent(int index) {

    return (index - 1) / 2;

}

int MaxHeap::getLeftChild(int index) {

    return 2 \* index + 1;

}

int MaxHeap::getRightChild(int index) {

    return 2 \* index + 2;

}

int MaxHeap::maxChildIndex(int first, int second) {

    if (dataVector[first] < dataVector[second])

        return second;

    return first;

}

void MaxHeap::swap(int first, int second) {

    Data copy = dataVector[first];

    dataVector[first] = dataVector[second];

    dataVector[second] = copy;

}

void MaxHeap::siftUp(int index) {

    if (dataVector.size() - 1 < index || index == 0)

        return;

    int parent = getParent(index);

    if (dataVector[index] < dataVector[parent])

        return;

    swap(parent, index);

    siftUp(parent);

}

void MaxHeap::siftDown(int index) {

    int leftChild = getLeftChild(index);

    int rightChild = getRightChild(index);

    int maxChild = leftChild;

    if (leftChild >= dataVector.size())

        return;

    if (rightChild < dataVector.size())

        maxChild = maxChildIndex(leftChild, rightChild);

    if (dataVector[maxChild] < dataVector[index])

        return;

    swap(maxChild, index);

    siftDown(maxChild);

}

PriorityQueue.cpp

#include "priorityQueue.h"

void PriorityQueue::push(Data data) {

    heap.dataVector.push\_back(data);

    heap.siftUp(heap.dataVector.size() - 1);

}

Data PriorityQueue::top() {

    return heap.dataVector.front();

}

void PriorityQueue::pop() {

    heap.swap(0, heap.dataVector.size() - 1);

    heap.dataVector.pop\_back();

    heap.siftDown(0);

}

void PriorityQueue::clean() {

    heap.dataVector.clear();

}

bool PriorityQueue::empty() {

    return heap.dataVector.empty();

}

int PriorityQueue::size() {

    return heap.dataVector.size();

}

HeapSort.cpp

#include "priorityQueue.h"

int HeapSort::getParent(int index) {

    return (index - 1) / 2;

}

int HeapSort::getLeftChild(int index) {

    return 2 \* index + 1;

}

int HeapSort::getRightChild(int index) {

    return 2 \* index + 2;

}

int HeapSort::maxChildIndex(int first, int second) {

    if (arrayToChange[first] < arrayToChange[second])

        return second;

    return first;

}

void HeapSort::swap(int first, int second) {

    int copy = arrayToChange[first];

    arrayToChange[first] = arrayToChange[second];

    arrayToChange[second] = copy;

}

void HeapSort::siftDown(int index, int arraySize) {

    int leftChild = getLeftChild(index);

    int rightChild = getRightChild(index);

    int maxChild = leftChild;

    if (leftChild >= arraySize)

        return;

    if (rightChild < arraySize)

        maxChild = maxChildIndex(leftChild, rightChild);

    if (arrayToChange[maxChild] < arrayToChange[index])

        return;

    swap(maxChild, index);

    siftDown(maxChild, arraySize);

}

void HeapSort::buildHeap(int\* array, int arraySize) {

    arrayToChange = array;

    int N = (arraySize) / 2 - 1;

    for (int i = N; i >= 0; i--) {

        siftDown(i, arraySize);

    }

}

void HeapSort::sortHeap(int\* array, int arraySize) {

    int currentSize = arraySize;

    buildHeap(array, arraySize);

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

        swap(0, currentSize - 1);

        currentSize--;

        siftDown(0, currentSize);

    }

}

Main.cpp

#include "priorityQueue.h"

#include <algorithm>

#include <iostream>

#include <queue>

#include <cstdlib>

#define N 30000

void bubbleSort(int\* arrSort, const int arrLength)

{

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

        for (int j = 0; j < arrLength - i - 1; j++)

            if (arrSort[j] > arrSort[j + 1])

                swap(arrSort[j], arrSort[j + 1]);

    return;

}

void chooseSort(int\* parray, const int arrayLength)

{

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

    {

        int min = parray[i];

        int nMin = i;

        int sumMin = 0;

        do {

            sumMin += min % 10;

            min /= 10;

        } while (min > 0);

        for (int j = i + 1; j < arrayLength; j++) {

            int a = parray[j];

            int sumA = 0;

            do {

                sumA += a % 10;

                a /= 10;

            } while (a > 0);

            if (sumA < sumMin)

            {

                sumMin = sumA;

                nMin = j;

            }

            else if (sumA == sumMin && parray[j] < parray[nMin]) {

                sumMin = sumA;

                nMin = j;

            }

        }

        swap(parray[i], parray[nMin]);

    }

    return;

}

int main()

{

    srand(time(NULL));

    PriorityQueue prior;

    testPriorityQueue();

    //sort

    int sort1[N], sort2[N], sort3[N], sort4[N];

    for (int i = 0; i < N; i++) {

        int rnd = rand() % 1000;

        sort1[i] = rnd;

        sort2[i] = rnd;

        sort3[i] = rnd;

        sort4[i] = rnd;

    }

    HeapSort heapSort;

    cout << "Sort test for 30 000 elements:" << endl << endl;

    //

    cout << "Average case:" << endl;

    clock\_t timeStart = clock();

    heapSort.sortHeap(sort1, N);

    clock\_t timeEnd = clock();

    cout << "HeapSort: " << (float(timeEnd - timeStart)) / CLOCKS\_PER\_SEC << "c" << endl;

    timeStart = clock();

    bubbleSort(sort2, N);

    timeEnd = clock();

    cout << "BubbleSort: " << (float(timeEnd - timeStart)) / CLOCKS\_PER\_SEC << "c" << endl;

    timeStart = clock();

    chooseSort(sort3, N);

    timeEnd = clock();

    cout << "ChooseSort: " << (float(timeEnd - timeStart)) / CLOCKS\_PER\_SEC << "c" << endl;

    timeStart = clock();

    sort(sort4, sort4 + N);

    timeEnd = clock();

    cout << "Std sort: " << (float(timeEnd - timeStart)) / CLOCKS\_PER\_SEC << "c" << endl << endl;

    //

    cout << "Best case:" << endl;

    timeStart = clock();

    heapSort.sortHeap(sort1, N);

    timeEnd = clock();

    cout << "HeapSort: " << (float(timeEnd - timeStart)) / CLOCKS\_PER\_SEC << "c" << endl;

    timeStart = clock();

    bubbleSort(sort2, N);

    timeEnd = clock();

    cout << "BubbleSort: " << (float(timeEnd - timeStart)) / CLOCKS\_PER\_SEC << "c" << endl;

    timeStart = clock();

    chooseSort(sort3, N);

    timeEnd = clock();

    cout << "ChooseSort: " << (float(timeEnd - timeStart)) / CLOCKS\_PER\_SEC << "c" << endl;

    timeStart = clock();

    sort(sort4, sort4 + N);

    timeEnd = clock();

    cout << "Std sort: " << (float(timeEnd - timeStart)) / CLOCKS\_PER\_SEC << "c" << endl << endl;

    //

    cout << "Worst case:" << endl;

    for (int i = 0; i < N - 1; i++) {

        sort1[N - 1 - i] = i;

        sort2[N - 1 - i] = i;

        sort3[N - 1 - i] = i;

        sort4[N - 1 - i] = i;

    }

    timeStart = clock();

    heapSort.sortHeap(sort1, N);

    timeEnd = clock();

    cout << "HeapSort: " << (float(timeEnd - timeStart)) / CLOCKS\_PER\_SEC << "c" << endl;

    timeStart = clock();

    bubbleSort(sort2, N);

    timeEnd = clock();

    cout << "BubbleSort: " << (float(timeEnd - timeStart)) / CLOCKS\_PER\_SEC << "c" << endl;

    timeStart = clock();

    chooseSort(sort3, N);

    timeEnd = clock();

    cout << "ChooseSort: " << (float(timeEnd - timeStart)) / CLOCKS\_PER\_SEC << "c" << endl;

    timeStart = clock();

    sort(sort4, sort4 + N);

    timeEnd = clock();

    cout << "Std sort: " << (float(timeEnd - timeStart)) / CLOCKS\_PER\_SEC << "c" << endl;

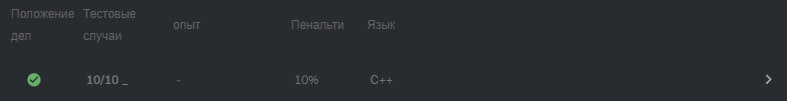
}

**Додаткове завдання:**

1. Розв’язати задачу “Пошук k-го найменшого елемента в масиві”:

1.1 Використати реалізацію бінарної купи для вирішення задачі

1.2 Перевірити своє рішення на тестах в системі:



#include <iostream>

#include <vector>

using namespace std;

int getParent(int index) {

    return (index - 1) / 2;

}

int getLeftChild(int index) {

    return 2 \* index + 1;

}

int getRightChild(int index) {

    return 2 \* index + 2;

}

int maxChildIndex(vector<int>\* arrayToChange, int first, int second) {

    if (arrayToChange->at(first) < arrayToChange->at(second))

        return second;

    return first;

}

void swap(vector<int>\* arrayToChange, int first, int second) {

    int copy = arrayToChange->at(first);

    arrayToChange->at(first) = arrayToChange->at(second);

    arrayToChange->at(second) = copy;

}

void siftDown(vector<int>\* arrayToChange, int index, int arraySize) {

    int leftChild = getLeftChild(index);

    int rightChild = getRightChild(index);

    int maxChild = leftChild;

    if (leftChild >= arraySize)

        return;

    if (rightChild < arraySize)

        maxChild = maxChildIndex(arrayToChange, leftChild, rightChild);

    if (arrayToChange->at(maxChild) < arrayToChange->at(index))

        return;

    swap(arrayToChange, maxChild, index);

    siftDown(arrayToChange, maxChild, arraySize);

}

void buildHeap(vector<int>\* input, int arraySize) {

    int N = arraySize / 2 - 1;

    for (int i = N; i >= 0; i--) {

        siftDown(input, i, arraySize);

    }

}

void sortHeap(vector<int>\* input, int arraySize, int k) {

    //int currentSize = arraySize;

    buildHeap(input, k);

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

        if (input->at(i) < input->at(0)) {

            swap(input, 0, i);

            siftDown(input, 0, k);

        }

    }

}

int kthSmallest(vector<int> input, int k)

{

    sortHeap(&input, input.size(), k);

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

        cout << input[i] << " ";

    }

    return input[0];

}

**Висновок:**

В результаті виконання лабораторної роботи було створено структури Купа, Пріоритетний список та сортування купою (Пірамідальне сортування). Також було перевантажено оператори «менше», «дорівнює» та «більше» для структури, що зберігає домашні завдання. Також виконано додаткове завдання на знаходження k-го найменшого елемента, де для отримання шуканого значення використовується пірамідальне сортування перших k елементів та почергове порівняння елементів з більшим індексом з наступною зміною місцями та просіюванням. При порівнянні нового сортування з попередньо створеними наглядно видно різницю швидкості, де готове бібліотечне рішення виконується швидше за все, а пірамідальне сортування лише трохи відстає від нього. При збільшені кількості елементів час їх виконання збільшується в рази повільніше. В свою чергу сортування бульбашкою та вставкою навіть для малої кількості елементів вже демонструють гірші показники, а зі збільшенням розмірів масиву – різниця з попередніми алгоритмами значно більше відчувається та збільшується в геометричній прогресії.