**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №3**

**по дисциплине «Алгоритмы и структуры данных»**

**Вариант № 2**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 9301 |  | Русанова К.В. |
| Преподаватель |  | Тутуева А. В. |

Санкт-Петербург

2020

### Постановка задачи. Описание реализуемых алгоритмов

Реализовать класс двоичной кучи, а также следующие методы:

1. bool contains(int); // поиск элемента в дереве по ключу
2. void insert(int); // добавление элемента в дерево по ключу.
3. void remove(int); // удаление элемента дерева по ключу
4. Iterator create\_dft\_iterator(); // создание итератора, реализующего один из методов обхода в глубину (depth-first traverse)
5. Iterator create\_bft\_iterator() // создание итератора, реализующего методы обхода в ширину (breadth-first traverse)

### Оценка временной сложности

Табл. 1

|  |  |  |
| --- | --- | --- |
| **Название метода** | **Сложность (в среднем)** | **Сложность (в худшем)** |
| contains | O(n) | O(n) |
| insert | O(log(n)) | O(n) |
| remove | O(n) | O(n) |

### Реализованные unit-тесты

1. TestInsert1 — Проверка добавления одного эл-та в кучу
2. TestInsert1To5 — Проверка добавления эл-тов с ключами 1-5
3. TestInsert5To1 — Проверка добавления эл-тов с ключами 5-1
4. TestContainsT — Проверка наличия с истинным исходом
5. TestContainsF — Проверка наличия с ложным исходом
6. TestRemoveLast — Проверка удаления последнего эл-та
7. TestRemove — Проверка удаления эл-та
8. TestRemoveFirst — Проверка удаления первого эл-та
9. TestRemoveError — Проверка ошибки при удалении
10. TestBFTError — Проверка окончания обхода в ширину
11. TestDFTError — Проверка окончания обхода в глубину
12. TestDFT1 — Проверка обхода в глубину
13. TestDFT2 — проверка обхода в глубину

### Пример работы

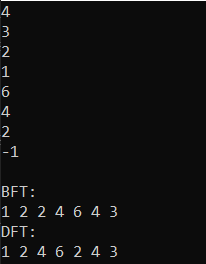


Рис. — Пример работы

Лист. — Программа для демонстрации примера работы

#include <iostream>

#include "BinaryHeap.h"

int main()

{

int in;

BinaryHeap h=BinaryHeap();

std::cin >> in;

while (in != -1)

{

h.insert(in);

std::cin >> in;

}

std::cout << std::endl;

std::cout <<"BFT:"<< std::endl;

Iterator\* h\_bft\_iterator = h.create\_bft\_iterator();

while (h\_bft\_iterator->has\_next())

{

std::cout << h\_bft\_iterator->next() << " ";

}

std::cout << std::endl;

std::cout << "DFT:" << std::endl;

Iterator\* h\_dft\_iterator = h.create\_dft\_iterator();

while (h\_dft\_iterator->has\_next())

{

std::cout << h\_dft\_iterator->next() << " ";

}

}

### Листинг

Лист. — Заголовочный файл

#pragma once

#include <stdlib.h>

#include "Iterator.h"

#include "dualList.h"

class BinaryHeap

{

private:

int count;

int Maxcount;

int\* Heap; //array for heap

public:

BinaryHeap(int\* Heap=nullptr, int Maxcount = 1, int count = 0) {

this->count = count;

this->Maxcount = Maxcount;

this->Heap = (int\*)malloc(sizeof(int)\*Maxcount);

};

void siftDown(int);

void siftUp(int);

bool contains(int); // search for an element in the heap by key

void insert(int); // add an item to the heap by key

void remove(int); // delete an item to the heap by key

Iterator\* create\_dft\_iterator(); // depth-first traverse iterator

Iterator\* create\_bft\_iterator(); // breadth-first traverse iterator

class dft\_Iterator : public Iterator // depth-first traverse

{

public:

dft\_Iterator(int\* start, int max) {

Stack = new dualList();

size = max;

current = start;

Icurrent = 0;

Stack->push\_back(Icurrent);

if ((Icurrent + 1) \* 2 < size) //add right branch

Stack->push\_back((Icurrent + 1) \* 2);

};

int next();

bool has\_next();

~dft\_Iterator() {

delete Stack;

delete current;

}

private:

dualList\* Stack;

int\* current;

int Icurrent;

int size;

};

class bft\_Iterator : public Iterator //breadth-first traverse

{

public:

bft\_Iterator(int\* start,int max) {

current=start;

size = max;

Icurrent = 0;

};

int next();

bool has\_next();

~bft\_Iterator() {

delete current;

}

private:

int\* current;

int Icurrent;

int size;

};

~BinaryHeap() {

delete Heap;

};

};

Лист. 3 — Файл с реализованными методами

#include "BinaryHeap.h"

#include <iostream>

//parent=(i - 1) / 2

//left=2 \* i + 1

//right=(i + 1) \* 2

void BinaryHeap::siftDown(int i) {

int j,left,right,ToSwap;

while ((2 \* i + 1) < count) //while we can go down

{

left = 2 \* i + 1; //left child

right = (i + 1) \* 2; //right child

j = left;

if ((right < count) && (Heap[right] < Heap[left])) //find min of child

j = right;

if (Heap[i] <= Heap[j]) //if the minimum child is equal to the element

break;

ToSwap = Heap[j];

Heap[j] = Heap[i];

Heap[i] = ToSwap;

i = j;

}

}

void BinaryHeap::siftUp(int i) {

int ToSwap;

while (Heap[i]<Heap[(i-1)/2]) //while the parent is more

{

ToSwap = Heap[(i - 1) / 2];

Heap[(i - 1) / 2] = Heap[i];

Heap[i] = ToSwap;

i = (i - 1) / 2; //go to parent

}

}

void BinaryHeap::insert(int add)

{

if (count+1 >= Maxcount) //if cann't add

{

int\* p;

p = (int\*)realloc(Heap, (Maxcount \* 2 + 1)\*sizeof(int));

if (!p) {

throw std::out\_of\_range("Allocation error");

return;

}

else {

Heap = p;

Maxcount = Maxcount \* 2 + 1;

}

} count++;

Heap[count - 1] = add;

siftUp(count - 1);

}

bool BinaryHeap::contains(int find) {

int i = 0, ToSwap;;

while ((i < count) && (Heap[i] != find)) {

i++;

}

if (Heap[i] != find)

return false; //if not found

else

return true; //if found

}

void BinaryHeap::remove(int del) {

int ToSwap,i = 0;

while ((i < count) && (Heap[i] != del))// if contains

i++;

if (Heap[i] != del) {

throw std::out\_of\_range("Element doesn't exist"); //error

return;

}

ToSwap = Heap[count-1];

Heap[count - 1] = Heap[i];

Heap[i] = ToSwap;

count--;

siftDown(i);

}

Iterator\* BinaryHeap::create\_bft\_iterator() {

return new bft\_Iterator(Heap,count);

}

bool BinaryHeap::bft\_Iterator::has\_next() {

return Icurrent < size;

}

int BinaryHeap::bft\_Iterator::next() {

if (!has\_next()) {

throw std::out\_of\_range("No more elements");

}

int temp = current[Icurrent];

Icurrent++;

return temp;

}

Iterator\* BinaryHeap::create\_dft\_iterator() {

return new dft\_Iterator(Heap, count);

}

bool BinaryHeap::dft\_Iterator::has\_next() {

return (Stack->at(Stack->get\_size() - 1) != 0)||((size<3)&&(Icurrent<size)&&(Icurrent>-1));

}

int BinaryHeap::dft\_Iterator::next() {

if (!has\_next()) {

throw std::out\_of\_range("No more elements");

}

int temp = current[Icurrent];

if (size<3)

Icurrent++;

else{

if ((Icurrent + 1) \* 2 < size) //add right branch

Stack->push\_back((Icurrent + 1) \* 2);

if (Icurrent\*2+1 < size) //if haven't reached the end of the branch go down

{

Icurrent = Icurrent \* 2 + 1;

}else{ //if reach the end of the branch go right

Icurrent = Stack->at(Stack->get\_size() - 1);

Stack->pop\_back();

}

}

return temp;

}