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
ТЕМА: ДВОИЧНЫЕ ДЕРЕВЬЯ

Студент гр. 9302 _____ Кузнецов В.А.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург
2020

Постановка задачи.

Реализовать структуру данных двоичная куча и методы для работы с ней.

Описание реализуемого класса и методов.

Были реализованы следующие классы:

1. Class List – собственная реализация стека в виде списка. Необходимо для работы итератора. Имеет следующие поля и методы:
    a. Private class Node – приватный внутренний класс для описания одного элемента списка;
    b. Private Node* begin – начало списка;
    c. Private Node* end – конец списка;
    d. int size – размер списка (количество элементов);
    e. Node* getEnd() – возвращает последний элемент в списке;
    f. void pushBack(int n) – вставляет элемент в конец списка;
    g. void popBack() – удаляет последний элемент списка;
2. Class Heap – класс, реализующий двоичную кучу со следующими полями и методами:
    a. Private const int LENGTH – размер массива значений элементов, так как массив задается статически;
    b. Private int* arr – массив элементов;
    c. Private int size – действительное количество элементов в куче;
    d. Private class Iterator – внутренний класс для итератора двоичной кучи. Имеет поля и методы:
        i. Private int* arr – указатель на массив arr внешнего класса;
        ii. Private int* size – указатель на количество элементов кучи;
        iii. Private List* stack – стек пройденных элементов;
        iv. Private int pos – позиция итератора в массиве элементов;
        v. Private char type – тип итератора: ‘d’ – с обходом в глубину и ‘b’ с обходом в ширину;
        vi. Iterator operator++() – перегрузка префиксного инкремента, переход итератора к следующему элементу;
        vii. Iterator operator--() – перегрузка префиксного декремента, переход итератора к предыдущему элементу;
        viii. Iterator operator+(const int& c) – перегрузка бинарного плюса, переход итератора на с шагов вперед;
        ix. Iterator operator-(const int& c) – перегрузка бинарного минуса, переход итератора на с шагов назад;
        x. Int val() – возвращает значение элемента кучи, на котором итератор находится;
    e. Private void heapify(int i) – пересобирает элементы кучи, начиная с i-ой позиции, чтобы восстановить свойства кучи при удалении ее элементов.
    f. Private int getPos(int n) – возвращает позицию элемента со значением n;

g. Bool contains(int n) – поиск элемента со значением n;
h. Void insert(int n) – вставляет элемент со значением n в кучу;
i. Void remove(int n) – удаляет элемент со значением n;
j. Iterator create_dft_iterator – возвращает итератор с методом обхода в глубину;
k. Iterator create_bft_iterator – возвращает итератор с методом обхода в ширину;

Оценка временной сложности каждого метода.

В табл. 1 приведена временная сложность для каждого метода.

Таблица 1 – Временная сложность методов

Функция (оператор)	Сложность
pushback()	O(1)
popBack()	O(1)
++ для DFS	O(n)
++ для BFS	O(1)
--	O(1)
contains, getPos	O(n)
insert	O(log(n))
remove	O(log(n))
heapify	O(log(n))

Описание реализованных Unit-тестов.

Для проверки реализованных методов были написаны unit-тесты:

1. test_getEnd – тестирование функции getEnd на списке List длины 3;
2. test_pushBack – тестирование функции pushBack на списке List длины 3;
3. test_popBack – тестирование функции popBack на списке List длины 3 путем удаления каждого элемента;
4. test_heap_methods – тестирование методов класса Heap на куче из 6 элементов;
5. test_DFT – тестирование итератора обхода в глубину;
6. test_BFT – тестирование итератора обхода в ширину.

Все тесты были пройдены успешно.

Пример работы.

Работа программы приведена на рис. 1 и 2, где print – функция печати элементов кучи, реализованная только для наглядного представления работы.

Рисунок 1 – Демонстрационный код

```cpp
int main(){
    auto heap = new Heap();
    heap->insert(3);
    heap->insert(2);
    heap->insert(0);
    heap->insert(4);
    heap->insert(1);
    heap->insert(6);
    heap->insert(5);
    heap->print();
    std::cout << std::endl;
    heap->remove(5);
    heap->print();
    std::cout << std::endl;
    for (auto iterDFT = heap->create_dft_iterator(); iterDFT.val() != 1; ++iterDFT) {
        std::cout << iterDFT.val() << " ";
    }
    std::cout << std::endl;
    for (auto iterBFT = heap->create_bft_iterator(); iterBFT.val() != 3; ++iterBFT) {
        std::cout << iterBFT.val() << " ";
    }
    return 0;
}
```

Рисунок 2 – Результат работы

```
6 3 5 2 1 0 4
6 3 4 2 1 0
6 3 2 1 4
6 3 4 2 1
```

Листинг.

```
#include <iostream>

class List {
    class Node {
    public:
        int n;
        Node* next;
        Node* prev;
        Node() {
            n = NULL;
            next = nullptr;
            prev = nullptr;
        }
        Node(int n) {
            this->n = n;
            next = nullptr;
            prev = nullptr;
        }
    };
    Node* begin;
    Node* end;
public:
    int size;
    List() {
        begin = new Node();
        end = new Node();
        size = 0;
    }
    Node* getEnd(){
        if (size == 0) {
            return nullptr;
        }
        else if (size == 1) {
            return begin;
        }
        return end;
    }
    void pushBack(int n) {
        if (size == 0) {
            begin->n = n;
        }
        else if (size == 1) {
            end->n = n;
            end->prev = begin;
            begin->next = end;
        }
        else {
            auto node = new Node(n);
            auto end = this->end;
            end->next = node;
            node->prev = end;
            this->end = node;
        }
        size++;
    }
    void popBack() {
        if (size == 1) {
            begin->n = NULL;
            size--;
        }
        else if (size == 2) {
```

```
            begin->next = nullptr;
            end->prev = nullptr;
            end->n = NULL;
            size--;
        }
        else if (size > 2) {
            end->prev->next = nullptr;
            end = end->prev;
            size--;
        }
    }
};

class Heap {
private:
    const int LENGTH = 10000;
    int* arr;
    int size;
    class Iterator {
        int* arr;
        int* size;
        List* stack;
        int pos;
        char type;
    public:
        Iterator(int* arr, int* size, List* stack, char type, int pos = 0) {
            this->arr = arr;
            this->size = size;
            this->stack = stack;
            this->type = type;
            this->pos = pos;
            if (stack->size == 0)
                stack->pushBack(0);
        };
        ~Iterator() = default;
        Iterator operator++() {
            int newPos = pos;
            if (type == 'd') {
                if (newPos * 2 + 1 < *size) {
                    newPos = newPos * 2 + 1;
                    stack->pushBack(newPos);
                }
                else {
                    while (((newPos - 1) / 2) * 2 + 2  != newPos + 1 || newPos + 1 >=
*size) {
                        newPos = (newPos - 1) / 2;
                        if (newPos == 0) {
                            return *this;
                        }
                    }
                    ++newPos;
                    stack->pushBack(newPos);
                }
            }
            else {
                if (newPos + 1 >= *size)
                    return *this;
                newPos++;
                stack->pushBack(newPos);
            }
            pos = newPos;
            return *this;
        }
        Iterator operator--() {
            if (pos == 0)
```

```
                return *this;
            pos = stack->getEnd()->prev->n;
            stack->popBack();
            return *this;
        }
        Iterator operator+(const int& c) {
            auto iter = *(new Iterator(arr, size, stack, type, pos));
            for (int i = 0; i < c; i++) {
                ++iter;
            }
            return iter;
        }
        Iterator operator-(const int& c) {
            auto iter = *(new Iterator(arr, size, stack, type, pos));
            for (int i = 0; i < c; i++) {
                --iter;
                if (iter.pos == 0)
                    return iter;
            }
            return iter;
        }
        Iterator operator+() const{
            return *this;
        }
        Iterator operator-() const{
            return *this;
        }
        int val() {
            return arr[pos];
        }
    };
    void heapify(int i) {
        int left = 2 * i + 1;
        int right = 2 * i + 2;
        int el;
        if (left < size) {
            if (arr[i] < arr[left]) {
                el = arr[i];
                arr[i] = arr[left];
                arr[left] = el;
                heapify(left);
            }
        }
        if (right < size) {
            if (arr[i] < arr[right]) {
                el = arr[i];
                arr[i] = arr[right];
                arr[right] = el;
                heapify(right);
            }
        }
    }
    int getPos(int n) {
        if (n > arr[0])
            return -1;
        for (int i = 0; i < size; i++) {
            if (n == arr[i])
                return i;
        }
        return -1;
    }
public:
    Heap() {
        arr = new int[LENGTH];
        size = 0;
```

```
    }
    bool contains(int n) {
        if (n > arr[0])
            return false;
        for (int i = 0; i < size; i++) {
            if (n == arr[i])
                return true;
        }
        return false;
    }
    void insert(int n) {
        int i = size;
        arr[i] = n;
        int parent = (i - 1) / 2;
        while (parent >= 0 && arr[parent] < arr[i] && i > 0) {
            int el = arr[i];
            arr[i] = arr[parent];
            arr[parent] = el;
            i = parent;
            parent = (i - 1) / 2;
        }
        size++;
    }
    void remove(int n) {
        int pos = getPos(n);
        if (pos != -1) {
            int el = arr[pos];
            arr[pos] = arr[size - 1];
            size--;
            heapify(pos);
        }
    }
    void print() {
        for (int i = 0; i < size; i++) {
            std::cout << arr[i] << " ";
        }
    }
    Iterator create_dft_iterator() {
        return *(new Iterator(arr, &size, new List(), 'd'));
    }
    Iterator create_bft_iterator() {
        return *(new Iterator(arr, &size, new List(), 'b'));
    }
};

int main(){
    auto heap = new Heap();
    heap->insert(3);
    heap->insert(2);
    heap->insert(0);
    heap->insert(4);
    heap->insert(1);
    heap->insert(6);
    heap->insert(5);
    heap->print();
    std::cout << std::endl;
    heap->remove(5);
    heap->print();
    std::cout << std::endl;
    for (auto iterDFT = heap->create_dft_iterator(); iterDFT.val() != 0; ++iterDFT) {
        std::cout << iterDFT.val() << " ";
    }
    std::cout << std::endl;
    for (auto iterBFT = heap->create_bft_iterator(); iterBFT.val() != 0; ++iterBFT) {
        std::cout << iterBFT.val() << " ";
```

```
    }
    return 0;
}
```