МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Алгоритмы и струтуры данных»

Тема: Двоичные деревья

Вариант 2

Студент гр. 0302 _____ Устинов Г.А.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург

2021

Цель работы

Изучить структуру данных Двоичная куча, алгоритмы работы с ней, а также написать собственную реализацию структуры.

Задание

Вариант 2

Реализовать класс двоичного дерева в соответствии со своим вариантом.

Отчет по лабораторной работе должен содержать:

1. Титульный лист с указанием варианта.

2. Постановка задачи.

3. Описание реализуемого класса и методов.

4. Оценка временной сложности каждого метода.

5. Описание реализованных unit-тестов.

6. Пример работы.

7. Листинг.

Наличие unit-тестов ко всем реализуемым методам и предусмотренные исключительные ситуации являются обязательными требованиями.

Структуры очередь и стэк из стандартных библиотек использовать нельзя, реализуем свои.

Список методов, которые реализует каждый вариант:

1. bool contains(int); // поиск элемента в дереве по ключу

2. void insert(int); // добавление элемента в дерево по ключу. Должен работать за O(logN)

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)

Варианты

№ Варианта	Структура данных
1	двоичное дерево поиска
2	двоичная куча

Выполнение работы

Классы Stack и Queue

Для реализации структур данных Стек и Очередь, были разработаны два класса — *Stack* и *Queue*. У данных классов есть предок — абстрактный шаблонный класс *LinearContainer*, который описывает тип, хранящий данные в последовательно соединённых узлах. Предполагается, что и стэк и очередь удаляют элементы из начала, и различаются только методом добавления элементов в список: очередь добавляет элементы в конец, а стек — в начало.

Таким образом, в программе используются собственные шаблонные типы для очереди и стека, которые определяют общий интерфейс взаимодейсвия с контейнером для типа *T*:

- *void push(T)* — добавляет элемент в очередь или стек.
- *void pop()* - удаляет элемент из очереди или стека.
- *T& peek()* - возвращяет ссылку на значение элемента, который будет удалён при следующем вызове *pop()*.
- *size_t size()* - возвращает размер очереди или стека.
- *bool is_empty()* - возвращает *true*, если контейнер пуст, иначе — *false*.

Класс BinaryHeap

class BinaryHeap

Тип	Имя	Описание
std::shared_ptr<Node>	*root_*	Указатель на корень кучи
size_t	*count*	Количество элементов в куче
size_t	*height_*	Высота кучи
HeapType	*type_*	Тип кучи — максимальная или минимальная

Класс, реализующий функциональность двоичной кучи — двоичного дерева, которое заполныется слева на право, и каждый родитель больше своих детей. Хранит двоичное дерево узлов, и позволяет производить операции вставки, удаления и поиска по нему. В пространстве имён класса определены типы, упрощающие его реализацию:

Тип *enum HeapType* — неизменяемая константа кучи, которая определяет, является ли она максимальной или минимальной, то есть производится ли сортировка элементов по возрастанию или убыванию.

Класс *Iterator* — абстрактный класс, базовый класс для итераторов, по двоичной куче. Хранит указатель на абстрактный контейнер-очередь *LinearContainer* и указатель на текущий узел. Конктретный тип контейнера определяется конкретной реализацией итератора. Определяет функции для перемещения по куче — *next()*, для проверки, существования следующего элемента последовательности — *has_next()*, а также оператор получения значения *operator*()* и преобразования к логическому типу *operator bool()*.

Класс *BreadthFirstIterator* — итератор для обхода кучи в ширину. Наследник класса *Iterator*, использующий для реализации обхода в ширину очередь, в которую на каждом шаге добавляются дети текузего элемента.

Класс *DepthFirstIterator* — итератор, реализующий прямой оюход в глубину по куче. Наследует от класса *Iterator* и в качестве контейнера для потомков узла использует стек.

Структура *Node* — элемент кучи. Имеет указатель на родителя, указатели на двоих потомков — левого и правого, поле для хранения целочисленного значения — ключа — и логический флаг, указывающий, является ли этот узел корневым.

Структура *RouteCode* — вспомогательная структура для хранения кода пути и его длины. Используется для объединения данных о пути и передачи их в функцию, которая восстанавливает путь из его кода и проходит его.

Структура *TraverseResult* — вспомогательная структура, которая хранит указатель на предка элемента, до которого был проложен путь, и ссылку на указатель, который должен хранить его адресс. Используется функцией прохождения пути для возврата объединённых данных.

Класс двоичной кучи также определяет несколько публичных методов, которые реализуют алгоритмы его обработки:

- *bool contains(int val)* — возвращает *true*, если ключ *val* находится в куче и *false* в противном случае. Использует вспомогательный метод *find_in_subtree()* для корня дерева для поиска нужного элемента, а потому скорость работы равно скорости работы *find_in_subtree()* - N.
- *void insert(int val)* — добавляет элемент со значением *val* в кучу. Изначально производится добавление элемента в «конец» кучи — согласно правилу заполнения кучи на самое левое свободное место на последнем слое. Если пронумеровать элементы на слое слева на право, начиная с нуля, то номер элемента будет «кодом», которому можно однозначно сопоставить путь до этого элемента, если представить «код» в виде двоичного числа. Каждый разряд этого числа будет

кодировать переход к потомку очередного узла: «0» - переход к левому потомку, «1» - к правому, от старшего разряда к младшему. Зная номер элемента его высоту, можно перейти к нему за время logN, где N — количество элементов в куче. Функция вычисляет код элемент за последующим, получая количество элементов на послежнем слое по формуле $i = N - (2^h - 1)$, где h — высота дерева. Длина кода равна высоте дерева. Если же последний слой полон (проверяется сравнением количества элементов со значением $2^{(h+1)} - 1$), то кодом нужного места берётся «0», его длинна на еденицу больше высоты дерева. После вставки элемента на этот слой, куча приводится к своему инварианту — каждый родитель больше своих детей. Для этого используется закрытая функция *hapify()*. Так-как работа состоит в последовательном вызове двух вспомогательных методов *traverse_by_code()* и *heapify()*, то скорость работы зависит от их скоростей, которые равны logN.

- *void remove(int val)* — удаляет элемент по ключу. Ищется узел, хранящий нужное значеие, после чего он меняется местами с последним узлом, и производится его удаление. Для элемента, который был помещён на место удалённого, вызывается функция *heapify()*. Подобно *insert()*, последовательно вызвает *traverse_by_code()* и *heapify()*. Поскольку в худшем случае (при удалении корня кучи) нужно дважды пройти вертикально по дереву (получить последний элемент в куче и сдвинуть бывший последний элемент снова вниз), то скорость этого метода равно logN.
- *BreadthFirstIterator create_bft_iterator()* - создаёт итератор для обхода в ширину, привязанный к куче, для которой вызван этот метод.

- *DepthFirstIterator create_dft_iterator()* - создаёт итератор для прямого обхода кучи в глубину, привязанный к той куче, для которой вызван этот метод.

Также класс определяет закрытые вспомогательные методы для выполнения нужных операций:

- *std::shared_ptr<Node> find_in_subtree(std::shared_ptr<Node> subroot, int val)* — рекурсивно находит значение *val* в поддереве, на котрое указывает *subroot*. Сначала проверяется равеноство значения в *subroot* и *val*, если они равны, то возвращается *subroot*. Иначе, если только значение не должно находится выше в куче, чем *subroot*, проверяются последовательно его левое и правое поддеревья. Результат возвращается. Если функция возвращает *nullptr*, значит элемент не был найден. Так-как производится обход по всем поддеревьям, пока не будет найден элемент, скорость работы зависит от количства элементов в куче, то есть N.
- *TraverseResult traverse_by_code(RouteCode code)* — восстанавливает маршрут из его кода и проходит его. Возвращает указатель на родитель найденного элемента и ссылку на один из его указателей-детей, в котором должен храниться нужный элемень. Так как не требуется вести поиск, путь известен, то максимальное количество переходов для достижения пути — глубина дерева, то есть logN, где N — количество элементовв куче.
- *void swap_nodes(std::shared_ptr<Node> first, std::shared_ptr<Node> second)* — меняет местами два узла, меняя их указатели на родителей и детей. Если в результате обмена сменился корень у дерева, значение *root_* обновляется.

- *void heapify(std::shared_ptr<Node> node)* — восстанавливает инвариант кучи после её изменения. Вызывается для добавленного элемента, или для элемента, изменившего свою позицию. Сдвигает элемент вверх или вниз, пока тот не окажется на нужном месте. Для сдвига применяет методы *sift_up()* и *sift_down()*. В худшем случае, сдвинуть элемент можно из крайнего верхнего или нижнего положения в противоположное, при этом количество перестановок равно глубине дерева, то есть logN.
- *void sift_up()* - сдвигает элемент наверх, пока над ним есть элементы, которые должны быть под ним.
- *void sift_down()* - сдвигает элемент вниз, пока под ним есть элементы, которые должны быть над ним.
- *bool is_highter(…)* - перегруженный метод, который проверяет, находится ли элемент или значение выше того, с которым сравнивают. При сравнении проверяет, к какому типу относится куча, к максимальнмому, или к минимальному.
- *bool is_lower(…)* - подобно *is_higher()*, проверяет, находится ли элемент или узел ниже того, с которым сравнивают, основываясь на типе кучи.
- *void recalculate_height()* - вызывается при изменении кучи для пересчёта значения высоты на основании значения *count_*.

  когда левая граница больше правой.

Тестирование

Введены тесты для публичных типов, которые были написаны для выполнения этой работы. Инвариант и корректность остальных типов контролируется теми типами, которые их используют.

Стэк и очередь тестируются практически идентично — добавление и удавление как в пустой так и непустой контейнер.

Куча тестируется как пустая, так и непустая, причём для обоих своих типов — для максимальной и для минимальой. Описаны тесты для всех публичных методов типа.

Для тестирования используется фреймворк googletest.

Пример работы

Следующий код служит примером работы с разработанными типами.

```
for (int i = 0; i < 10; ++i)
    heap.insert(i);
auto it = heap.create_vft_iterator();
while (it) {
    cout << *it;
    it.next();
}
```

Разработанный программный код см. в приложении А.

Выводы

Был изучен и руализован тип двоичной кучи, а также методы его обработки. Также, были установленна схожесть структур данных стека и очереди, а также способы применения типа однонаправленного списка для их реализации.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл **src/linear_container.hh**

```
#pragma once
#ifndef CONTAINER_HH
#define CONTAINER_HH

#include <memory>
#include <stdexcept>

namespace data_structs {

template<class T>
class LinearContainer {
public:
   LinearContainer() : head_(nullptr), tail_(nullptr), count_(0) {}
   virtual ~LinearContainer() = default;

   virtual void push(T data) = 0;
   void pop();

   T& peek();

   size_t size() { return count_; }
   bool is_empty() { return count_ == 0; }
protected:
   template<class U>
   struct Node;

   std::shared_ptr<Node<T>> head_;
   std::shared_ptr<Node<T>> tail_;
   size_t count_;
};


template<class T>
template<class U>
struct LinearContainer<T>::Node {
   Node(U data) : next(nullptr), data(data) {}

   std::shared_ptr<Node> next;
   U data;
};


template<class T>
void LinearContainer<T>::pop()
{
   if (count_ == 0)
```

```
        return;
    head_ = head_->next;
    if (!head_)
        tail_ = head_;
    --count_;
}


template<class T>
T& LinearContainer<T>::peek()
{
    if (count_ == 0)
        throw std::logic_error("Peeking on empty queue.");
    return head_->data;
}

}   // namespace data_structs

#endif  // CONTAINER_HH
```

Файл **src/stack.hh**

```
#pragma once
#ifndef STACK_HH
#define STACK_HH

#include <memory>
#include <stdexcept>

#include "linear_container.hh"

namespace data_structs {

template<class T>
class Stack : public LinearContainer<T> {
public:
    Stack() = default;
    ~Stack() = default;

    void push(T data);
};


template<class T>
void Stack<T>::push(T data)
{
    std::shared_ptr<typename LinearContainer<T>::Node<T>>
        node(new typename LinearContainer<T>::Node<T>(data));
    if (this->count_ == 0)
        this->tail_ = node;
    else
```

```
        node->next = this->head_;
    this->head_ = node;
    ++this->count_;
}

}   // namespace data_structs

#endif  // STACK_HH
```

Файл **src/queue.hh**

```
#pragma once
#ifndef QUEUE_HH
#define QUEUE_HH

#include <memory>
#include <stdexcept>

#include "linear_container.hh"

namespace data_structs {

template<class T>
class Queue : public LinearContainer<T> {
public:
    Queue() = default;
    ~Queue() = default;

    void push(T data);
};


template<class T>
void Queue<T>::push(T data)
{
    std::shared_ptr<typename LinearContainer<T>::Node<T>>
        node(new typename LinearContainer<T>::Node<T>(data));
    if (this->count_ == 0)
        this->head_ = node;
    else
        this->tail_->next = node;
    this->tail_ = node;
    ++this->count_;
}

}   // namespace data_structs

#endif  // QUEUE_HH
```

Файл **src/binary_heap.hh**

```
#pragma once
#ifndef BINARY_HEAP_HH
#define BINARY_HEAP_HH

#include <ostream>
#include <memory>

#include "stack.hh"
#include "queue.hh"

namespace data_structs {

class BinaryHeap {
public:
   /*
    * Determines the sorting order for the nodes.
    */
   enum HeapType {kMinHeap, kMaxHeap};

   class Iterator;
   class BreadthFirstIterator;
   class DepthFirstIterator;

   BreadthFirstIterator create_bft_iterator();
   DepthFirstIterator create_dft_iterator();

   BinaryHeap(HeapType type)
       : root_(nullptr), count_(0), height_(0), type_(type)
   {}
   BinaryHeap(BinaryHeap& other);
   BinaryHeap(BinaryHeap&& other);

   ~BinaryHeap() = default;

   bool contains(int val);
   /*
    * What if every node in the heap would have its own code?
    * Let the code be a sequence of edges codes. And edges codes will
    * be 0 if it is descending to the left and 1 if it is descending
    * to the right:
    *        root
    *      /     \         Something like
    *    0        1         that, I gues..?
    *   / \      /
    * 00   01  10
    * Knowing the code of a node we could easily find its position
    * just following it. And this will help us find the position
    * to insert a new element. The amount of edge traversions to the
    * parent of future node equals height - 1 if the last layer of
    * nodes is incomplete and just height otherwise.
```

```
     * Let the amount of elements on the last layer be 0 if the last
     * layer is complete. Otherwise it is count - (pow(2, height) - 1).

     * The code of the future element will be equal to that number.
     * Length of the code equals height of the tree if the last layer
     * is incomplete and height + 1 otherwise.
     */
    void insert(int val);
    /*
     * Similalry, lets find the last element. Its code will always
     * be count - (pow(2, height)) and is always of height lenght.
     */
    void remove(int val);

    bool is_empty() { return count_ == 0; }
    size_t count() { return count_; }
    size_t height() { return height_; }
    HeapType type() { return type_; }

    friend std::ostream &operator<<(std::ostream &os, BinaryHeap &h);
private:
    struct Node;
    /*
     * The code of the binary route and its lenght.
     * Mind that route with code 0 is a root, processing this route
     * is not supported. Be careful!
     */
    struct RouteCode {
        size_t code;
        size_t len;
    };
    /*
     * Pointer to the parent of the node and a reference to the pointer

     * of the parent where an end point of the route would have been
     * stored if the end point would have existed. Might be a valid
     * pointer though, if the route leads to an existing node.
     */
    struct TraverseResult {
        TraverseResult(std::shared_ptr<Node> &end_point)
            : ref(end_point)
        {}

        std::shared_ptr<Node> parent;
        std::shared_ptr<Node> &ref;
    };

    // Recursively finds val in the subtree pointed by subroot
    std::shared_ptr<Node> find_in_subtree(std::shared_ptr<Node>
subroot, int val);

    /*
```

```
    * Gets code, translates it to a binary route through
    * the heap, traverses it and returns reference to the pointer
    * that would hold that code.
    */
   TraverseResult traverse_by_code(RouteCode code);

   void swap_nodes(std::shared_ptr<Node> first, std::shared_ptr<Node>
second);

   void heapify(std::shared_ptr<Node> node);
   void sift_up(std::shared_ptr<Node> node);
   void sift_down(std::shared_ptr<Node> node);
   /*
    * Tells if the node with the given value should be higher in the
heap
    * then comparative node.
    */
   bool is_higher(int val, std::shared_ptr<Node> comparative);
   bool is_higher(std::shared_ptr<Node> node, std::shared_ptr<Node>
comparative);
   /*
    * Tells if the node with the given value should be lower in the
heap
    * then comparative node.
    */
   bool is_lower(int val, std::shared_ptr<Node> comparative);
   bool is_lower(std::shared_ptr<Node> node, std::shared_ptr<Node>
comparative);

   void recalculate_height();

   std::shared_ptr<Node> root_;
   size_t count_;
   size_t height_;
   HeapType type_;
};


struct BinaryHeap::Node {
   Node(int val)
       : parent(nullptr), left(nullptr), right(nullptr),
       data(val), root(false)
   {}
   Node(const Node& other)
       : parent(nullptr), left(nullptr), right(nullptr),
       data(other.data), root(other.root)
   {}

   std::shared_ptr<Node> parent;
   /*
    * As the heap is packed from left to right, if the node has no
    * left child, it has no right either.
```

```
    */
   std::shared_ptr<Node> left;
   std::shared_ptr<Node> right;
   int data;
   bool root;
};


class BinaryHeap::Iterator {
public:
   Iterator(std::shared_ptr<LinearContainer<std::shared_ptr<Node>>>
container, std::shared_ptr<Node> &node)
       : container_(container), cur_(node)
   {}
   virtual ~Iterator() = default;

   int operator*();
   operator bool() const { return bool(cur_); }

   void next();
   bool has_next() { return !container_->is_empty(); }
   friend std::ostream &operator<<(std::ostream &os, BinaryHeap &h);
protected:
   virtual void enqueue() = 0;

   std::shared_ptr<LinearContainer<std::shared_ptr<Node>>> container_;

   std::shared_ptr<Node> cur_;
};


class BinaryHeap::BreadthFirstIterator : public BinaryHeap::Iterator {

public:
   BreadthFirstIterator(BinaryHeap& heap)
       :
Iterator(std::shared_ptr<LinearContainer<std::shared_ptr<Node>>>(new
Queue<std::shared_ptr<Node>>()), heap.root_)
   {
       enqueue();
   }
protected:
   void enqueue();
};


class BinaryHeap::DepthFirstIterator : public BinaryHeap::Iterator {
public:
   DepthFirstIterator(BinaryHeap& heap)
       :
Iterator(std::shared_ptr<LinearContainer<std::shared_ptr<Node>>>(new
Stack<std::shared_ptr<Node>>()), heap.root_)
```

```
    {
        enqueue();
    }
protected:
    void enqueue();
};

}   // namespace data_structs

#endif  // BINARY_HEAP_HH
```

Файл **src/binary_heap.cpp**

```
#include "binary_heap.hh"

#include <cmath>
#include <memory>
#include <stdexcept>

#include "stack.hh"

namespace data_structs {

BinaryHeap::BreadthFirstIterator BinaryHeap::create_bft_iterator()
{
    return BreadthFirstIterator(*this);
}

BinaryHeap::DepthFirstIterator BinaryHeap::create_dft_iterator()
{
    return DepthFirstIterator(*this);
}


BinaryHeap::BinaryHeap(BinaryHeap& other)
    : BinaryHeap(other.type_)
{
    BreadthFirstIterator it = other.create_bft_iterator();
    while (it) {
        insert(*it);
        it.next();
    }
}

BinaryHeap::BinaryHeap(BinaryHeap&& other)
    : count_(other.count_), height_(other.height_), type_(other.type_)
{
    root_.swap(other.root_);
}
```

```
bool BinaryHeap::contains(int val)
{
   if (!root_)
       return false;
   std::shared_ptr<Node> cur = root_;
   return find_in_subtree(root_, val) != nullptr;
}

void BinaryHeap::insert(int val)
{
   if (count_ == 0) {
       root_ = std::shared_ptr<Node>(new Node(val));
       root_->root = true;
       height_ = 0;
   } else {
       RouteCode code;
       // if the last layer is full
       if (count_ == std::pow(2, height_ + 1) - 1) {
           code.code = 0;
           code.len = height_ + 1;
       } else {
           code.code = count_ - (std::pow(2, height_) - 1);
           code.len = height_;
       }
       TraverseResult end_point = traverse_by_code(code);
       end_point.ref = std::shared_ptr<Node>(new Node(val));
       end_point.ref->parent = end_point.parent;
       heapify(end_point.ref);
   }
   ++count_;
   recalculate_height();
}

void BinaryHeap::remove(int val)
{
   if (!root_)
       return;
   std::shared_ptr<Node> sought = find_in_subtree(root_, val);
   if (!sought) {
       return;
   } else if (count_ == 1) {
       root_.reset();
       count_ = 0;
       return;
   }
   RouteCode code = {count_ - (size_t)std::pow(2, height_), height_};
   TraverseResult res = traverse_by_code(code);
   std::shared_ptr<Node> last = res.ref;
   swap_nodes(sought, last);
   (sought->parent->right && sought->parent->right == sought
       ? sought->parent->right : sought->parent->left).reset();
```

```
    heapify(last);
    --count_;
    recalculate_height();
}


std::ostream &operator<<(std::ostream &os, BinaryHeap &h)
{
    BinaryHeap::BreadthFirstIterator it = h.create_bft_iterator();
    while (it || it.has_next()) {
        os << *it << ": ";
        if (it.cur_->left)
            os << it.cur_->left->data;
        if (it.cur_->right)
            os << ", " << it.cur_->right->data;
        os << "\n";
        it.next();
    }
    return os;
}


std::shared_ptr<BinaryHeap::Node>
BinaryHeap::find_in_subtree(std::shared_ptr<Node> subroot, int val)
{
    std::shared_ptr<Node> sought = nullptr;
    if (subroot->data == val) {
        sought = subroot;
    } else if (!is_higher(val, subroot)) {
        if (subroot->left)
            sought = find_in_subtree(subroot->left, val);
        if (!sought && subroot->right)
            sought = find_in_subtree(subroot->right, val);
    }
    return sought;
}


BinaryHeap::TraverseResult
BinaryHeap::traverse_by_code(BinaryHeap::RouteCode code)
{
    // Encode a route
    Stack<int> route;
    for (size_t i = 0; i < code.len; ++i) {
        route.push(code.code & 1);
        code.code = code.code >> 1;
    }
    // Traverse the route
    std::shared_ptr<Node> cur = root_;
    while (route.size() > 1) {
        cur = route.peek() == 0 ? cur->left : cur->right;
        route.pop();
```

```
    }
    TraverseResult res(route.peek() == 0 ? cur->left : cur->right);
    res.parent = cur;
    return res;
}


void BinaryHeap::swap_nodes(std::shared_ptr<Node> first,
std::shared_ptr<Node> second)
{
    if (first == second)
        return;
    if (first->parent) {
        (first->parent->right == first
            ? first->parent->right : first->parent->left) = second;
    }
    if (second->parent) {
        (second->parent->right == second
            ? second->parent->right : second->parent->left) = first;
    }
    first->parent.swap(second->parent);
    if (first->left)
        first->left->parent = second;
    if (second->left)
        second->left->parent = first;
    first->left.swap(second->left);
    if (first->right)
        first->right->parent = second;
    if (second->right)
        second->right->parent = first;
    first->right.swap(second->right);
    bool tmp = second->root;
    second->root = first->root;
    first->root = tmp;
    if (first->root)
        root_ = first;
    else if (second->root)
        root_ = second;
}


void BinaryHeap::heapify(std::shared_ptr<Node> node)
{
    // If node's got no children, it can be only upsifted
    if (!node->left) {
        sift_up(node);
    } else if (node->root) {
        sift_down(node);
    } else {
        if (is_higher(node, node->parent))
            sift_up(node);
        else
```

```
            sift_down(node);
    }
}

void BinaryHeap::sift_up(std::shared_ptr<Node> node)
{
    while (!node->root && is_higher(node, node->parent)) {
        swap_nodes(node, node->parent);
    }
}

void BinaryHeap::sift_down(std::shared_ptr<Node> node)
{
    while (node->left) {
        std::shared_ptr<Node> to_swap;
        if (node->right && is_lower(node->left, node->right))
            to_swap = node->right;
        else
            to_swap = node->left;
        if (is_lower(node, to_swap)) {
            swap_nodes(node, to_swap);
        } else {
            break;
        }
    }
}


bool BinaryHeap::is_higher(int val, std::shared_ptr<Node> comparative)

{
    if (type_ == kMinHeap)
        return val < comparative->data;
    else
        return val > comparative->data;
}

bool BinaryHeap::is_higher(std::shared_ptr<Node> node,
std::shared_ptr<Node> comparative)
{
    return is_higher(node->data, comparative);
}


bool BinaryHeap::is_lower(int val, std::shared_ptr<Node> comparative)
{
    if (type_ == kMinHeap)
        return val > comparative->data;
    else
        return val < comparative->data;
}
```

```
bool BinaryHeap::is_lower(std::shared_ptr<Node> node,
std::shared_ptr<Node> comparative)
{
   return is_lower(node->data, comparative);
}


void BinaryHeap::recalculate_height()
{
   if (count_ < 2)
       height_ = 0;
   else
       height_ = (size_t)std::floor(std::log2l(count_));
}


int BinaryHeap::Iterator::operator*()
{
   if (!*this)
       throw std::logic_error("Dereferencing past-the-end iterator.");

   return cur_->data;
}


void BinaryHeap::Iterator::next()
{
   if (!cur_)
       throw std::out_of_range("Traversing with past-the-end
iterator.");
   if (container_->is_empty()) {
       cur_ = nullptr;
   } else {
       cur_ = container_->peek();
       container_->pop();
       enqueue();
   }
}


void BinaryHeap::BreadthFirstIterator::enqueue()
{
   if (!cur_)
       return;
   if (cur_->left)
       container_->push(cur_->left);
   if (cur_->right)
       container_->push(cur_->right);
}


void BinaryHeap::DepthFirstIterator::enqueue()
```

```
{
   if (!cur_)
       return;
   if (cur_->right)
       container_->push(cur_->right);
   if  (cur_->left)
       container_->push(cur_->left);
}

}   // namespace data_structs
```

Файл **test/lab3-queue-test.cpp**

```
#include "../src/queue.hh"

#include <stdexcept>

#include <gtest/gtest.h>

using namespace data_structs;

TEST(Base, QueueCreationTest)
{
   Queue<int> q;
   ASSERT_EQ(q.size(), 0);
   ASSERT_TRUE(q.is_empty());
   ASSERT_THROW(q.peek(), std::logic_error);
   ASSERT_NO_THROW(q.pop());
}

TEST(Base, QueueEmptyInsert)
{
   Queue<int> q;
   ASSERT_NO_THROW(q.push(0));
   ASSERT_EQ(q.size(), 1);
   ASSERT_FALSE(q.is_empty());
   ASSERT_EQ(q.peek(), 0);
}


class QueueTest : public ::testing::Test {
public:
   void SetUp()
   {
       for (int i = 0; i < 10; ++i)
           q_.push(i);
   }
protected:
   Queue<int> q_;
};
```

```
TEST_F(QueueTest, PushTest)
{
   ASSERT_NO_THROW(q_.push(10));
   ASSERT_EQ(q_.size(), 11);
   ASSERT_FALSE(q_.is_empty());
   ASSERT_EQ(q_.peek(), 0);
}

TEST_F(QueueTest, PopTest)
{
   for (int i = 0; i < 10; ++i) {
       ASSERT_EQ(q_.peek(), i);
       ASSERT_NO_THROW(q_.pop());
   }
   ASSERT_TRUE(q_.is_empty());
}
```

Файл **test/lab3-stack-test.cpp**

```
#include "../src/stack.hh"

#include <stdexcept>

#include <gtest/gtest.h>

using namespace data_structs;

TEST(Base, StackCreationTest)
{
   Stack<int> s;
   ASSERT_EQ(s.size(), 0);
   ASSERT_TRUE(s.is_empty());
   ASSERT_THROW(s.peek(), std::logic_error);
   ASSERT_NO_THROW(s.pop());
}

TEST(Base, StackEmptyInsert)
{
   Stack<int> s;
   ASSERT_NO_THROW(s.push(0));
   ASSERT_EQ(s.size(), 1);
   ASSERT_FALSE(s.is_empty());
   ASSERT_EQ(s.peek(), 0);
}


class StackTest : public ::testing::Test {
public:
   void SetUp()
```

```
    {
        for (int i = 0; i < 10; ++i)
            s_.push(i);
    }
protected:
    Stack<int> s_;
};

TEST_F(StackTest, PushTest)
{
    ASSERT_NO_THROW(s_.push(10));
    ASSERT_EQ(s_.size(), 11);
    ASSERT_FALSE(s_.is_empty());
    ASSERT_EQ(s_.peek(), 10);
}

TEST_F(StackTest, PopTest)
{
    for (int i = 9; i >= 0; --i) {
        ASSERT_EQ(s_.peek(), i);
        ASSERT_NO_THROW(s_.pop());
    }
    ASSERT_TRUE(s_.is_empty());
}
```

Файл **test/lab3-bianry-heap-test.cpp**

```
#include "../src/binary_heap.hh"

#include <cmath>
#include <iostream>

#include <gtest/gtest.h>

using namespace data_structs;
using std::cout;
using std::endl;


TEST(Base, CreationTest)
{
    BinaryHeap heap(BinaryHeap::kMinHeap);
    ASSERT_EQ(heap.count(), 0);
    ASSERT_TRUE(heap.is_empty());
}

TEST(Base, AddOneElement)
{
    BinaryHeap heap(BinaryHeap::kMinHeap);
    heap.insert(0);
```

```
   cout << heap << endl;
   ASSERT_EQ(heap.count(), 1);
   ASSERT_FALSE(heap.is_empty());
}

TEST(Base, AddFewElements)
{
   BinaryHeap heap(BinaryHeap::kMinHeap);
   heap.insert(0);
   cout << heap << endl;
   heap.insert(1);
   cout << heap << endl;
   heap.insert(2);
   cout << heap << endl;
   heap.insert(3);
   cout << heap << endl;
   heap.insert(4);
   cout << heap << endl;
   ASSERT_EQ(heap.count(), 5);
   ASSERT_FALSE(heap.is_empty());
}

TEST(Base, AddNonSorted)
{
   BinaryHeap heap(BinaryHeap::kMinHeap);
   heap.insert(5);
   cout << heap << endl;
   heap.insert(3);
   cout << heap << endl;
   heap.insert(2);
   cout << heap << endl;
   heap.insert(4);
   cout << heap << endl;
   heap.insert(1);
   cout << heap << endl;
   ASSERT_EQ(heap.count(), 5);
   ASSERT_FALSE(heap.is_empty());
}


class MinBinaryHeapTest : public ::testing::Test {
public:
   MinBinaryHeapTest() : heap_(BinaryHeap::kMinHeap)
   {}
   void SetUp()
   {
       for (int i = 0; i < 20; ++i)
           heap_.insert(i);
   }
protected:
   BinaryHeap heap_;
};
```

```
TEST_F(MinBinaryHeapTest, Contains)
{
   for (int i = 0; i < (int)heap_.count(); ++i)
       ASSERT_TRUE(heap_.contains(i));
}

TEST_F(MinBinaryHeapTest, Remove)
{
   for (int i = 0; i < 20; ++i) {
       ASSERT_NO_THROW(heap_.remove(i));
       ASSERT_FALSE(heap_.contains(i));
   }
}

TEST_F(MinBinaryHeapTest, Insert)
{
   ASSERT_NO_THROW(heap_.insert(10));
   ASSERT_EQ(heap_.count(), 21);
}

TEST_F(MinBinaryHeapTest, BFI)
{
   BinaryHeap::BreadthFirstIterator iterator =
heap_.create_bft_iterator();
   cout << "{ ";
   for (int i = 0; i < (int)heap_.count(); ++i) {
       cout << *iterator << " ";
       ASSERT_EQ(*iterator, i);
       iterator.next();
   }
   cout << "}" << endl;
}

TEST_F(MinBinaryHeapTest, DFI)
{
   BinaryHeap::DepthFirstIterator iterator =
heap_.create_dft_iterator();
   cout << "{ ";
   while (iterator || iterator.has_next()) {
       cout << *iterator << " ";
       iterator.next();
   }
   cout << "}" << endl;
}

class MaxBinaryHeapTest : public ::testing::Test {
public:
   MaxBinaryHeapTest() : heap_(BinaryHeap::kMaxHeap)
   {}
   void SetUp()
   {
```

```
        for (int i = 20; i > 0; --i)
            heap_.insert(i);
    }
protected:
    BinaryHeap heap_;
};

TEST_F(MaxBinaryHeapTest, Contains)
{
    for (int i = 1; i <= (int)heap_.count(); ++i)
        ASSERT_TRUE(heap_.contains(i));
}

TEST_F(MaxBinaryHeapTest, Remove)
{
    for (int i = 1; i <= 20; ++i) {
        ASSERT_NO_THROW(heap_.remove(i));
        ASSERT_FALSE(heap_.contains(i));
    }
}

TEST_F(MaxBinaryHeapTest, Insert)
{
    ASSERT_NO_THROW(heap_.insert(10));
    ASSERT_EQ(heap_.count(), 21);
}

TEST_F(MaxBinaryHeapTest, BFI)
{
    BinaryHeap::BreadthFirstIterator iterator =
heap_.create_bft_iterator();
    cout << "{ ";
    for (int i = 20; i > 0; --i) {
        cout << *iterator << " ";
        ASSERT_EQ(*iterator, i);
        iterator.next();
    }
    cout << "}" << endl;
}

TEST_F(MaxBinaryHeapTest, DFI)
{
    BinaryHeap::DepthFirstIterator iterator =
heap_.create_dft_iterator();
    cout << "{ ";
    while (iterator || iterator.has_next()) {
        cout << *iterator << " ";
        iterator.next();
    }
    cout << "}" << endl;
}
```