

Задание 3.1

Постройте очередь с приоритетами на основе адаптера `priority_queue`. Типы приоритетов и значений соответствуют пункту 2 задания №1. Выведите элементы очереди в порядке убывания приоритета.

Код 3.1. Пример работы с адаптером “очередь с приоритетом”

```
using namespace std;

#include <iostream>
#include <queue>

template<typename T>
void print_queue(T& q) {
    while (!q.empty()) {
        cout << q.top() << " ";
        q.pop();
    }
    std::cout << '\n';
}

int main() {
    priority_queue<int> q;

    for (int n : {1, 8, 5, 6, 3, 4, 0, 9, 7, 2})
        q.push(n);

    print_queue(q); }
```

Задание 3.2

Используйте шаблон класса `Heap` (бинарная куча, пирамида) для хранения объектов в соответствии с пунктом 2 задания №1 (используется упорядоченность по приоритету, в корне дерева – максимум). Реализуйте функцию удаления корня дерева `ExtractMax()`. Выведите элементы `Heap` в порядке убывания приоритета с её помощью. Добавьте операцию удаления произвольного элемента (требуется просеивание вверх или вниз?).

Код 3.2. Класс `Heap` (куча, пирамида)

```
#include <iostream>
using namespace std;
```

Продолжение кода 3.2

```
//узел дерева
template <class T>
class Node
{
private:
    T value;
public:
    //установить данные в узле
    T getValue() { return value; }
    void setValue(T v) { value = v; }

    //сравнение узлов
    int operator<(Node N)
    {
        return (value < N.getValue());
    }
    int operator>(Node N)
    {
        return (value > N.getValue());
    }
    //вывод содержимого одного узла
    void print()
    {
        cout << value;
    }
};

template <class T>
void print(Node<T>* N)
{
    cout << N->getValue() << "\n";
}

//куча (heap)
template <class T>
class Heap
{
private:
```

Продолжение кода 3.2

```
//массив
Node<T>* arr;
//сколько элементов добавлено
int len;
//сколько памяти выделено
int size;
public:
    //доступ к вспомогательным полям кучи и оператор
индекса
    int getCapacity() { return size; }
    int getCount() { return len; }

    Node<T>& operator[](int index)
    {
        if (index < 0 || index >= len)
            ;//?
        return arr[index];}
//конструктор
Heap<T> (int MemorySize = 100)
{
    arr = new Node<T>[MemorySize];
    len = 0;
    size = MemorySize;
}
//поменять местами элементы arr[index1],
arr[index2]
void Swap(int index1, int index2)
{
    if (index1 <= 0 || index1 >= len)
        ;
    if (index2 <= 0 || index2 >= len)
        ;
    //здесь нужна защита от дурака

    Node<T> temp;
    temp = arr[index1];
    arr[index1] = arr[index2];
```

Продолжение кода 3.2

```
arr[index2] = temp;
}

//скопировать данные между двумя узлами
void Copy(Node<T>* dest, Node<T>* source)
{
    dest->setValue(source->getValue());
}

//функции получения левого, правого дочернего
элемента, родителя или их индексов в массиве
Node<T>* GetLeftChild(int index)
{
    if (index < 0 || index * 2 >= len)
        ;
    //здесь нужна защита от дурака
    return &arr[index * 2 + 1];
}
Node<T>* GetRightChild(int index)
{
    if (index < 0 || index * 2 >= len);
    //здесь нужна защита от дурака
    return &arr[index * 2 + 2];
}
Node<T>* GetParent(int index)
{
    if (index <= 0 || index >= len)
        ;
    //здесь нужна защита от дурака

    if (index % 2 == 0)
        return &arr[index / 2 - 1];
    return &arr[index / 2];
}
```

Продолжение кода 3.2

```
int GetLeftChildIndex(int index)
{
    if (index < 0 || index * 2 >= len)
        ;
    //здесь нужна защита от дурака
    return index * 2 + 1;
}

int GetRightChildIndex(int index)
{
    if (index < 0 || index * 2 >= len)
        ;
    //здесь нужна защита от дурака

    return index * 2 + 2;
}

int GetParentIndex(int index)
{
    if (index <= 0 || index >= len)
        ;
    //здесь нужна защита от дурака
    if (index % 2 == 0)
        return index / 2 - 1;
    return index / 2;
}

//просеять элемент вверх
void SiftUp(int index = -1)
{
    if (index == -1) index = len - 1;
    int parent = GetParentIndex(index);
    int index2 = GetLeftChildIndex(parent);
    if (index2 == index) index2 =
GetRightChildIndex(parent);
    int max_index = index;

    if (index < len && index2 < len && parent >= 0)
```

Продолжение кода 3.2

```
{
    if (arr[index] > arr[index2])
        max_index = index;
    if (arr[index] < arr[index2])
        max_index = index2;
}
if (parent < len && parent >= 0 &&
arr[max_index]>arr[parent])
{
    //нужно просеивание вверх
    Swap(parent, max_index);
    SiftUp(parent);
}
}
//добавление элемента - вставляем его в конец
массива и просеиваем вверх
template <class T>
void push(T v)
{
    Node<T>* N = new Node<T>;
    N->setValue(v);
    push(N);
}
template <class T>
void push(Node<T>* N)
{
    if (len < size)
    {
        Copy(&arr[len], N);
        len++;SiftUp();}}
//перечислить элементы кучи и применить к ним
функцию
void Straight(void(*f) (Node<T>*))
{
    int i;
    for (i = 0; i < len; i++)
    {
```

Окончание кода 3.2

```
        f(&arr[i]);
    }
}

//перебор элементов, аналогичный проходам бинарного
дерева
void InOrder(void(*f) (Node<T>*), int index = 0)
{
    if (GetLeftChildIndex(index) < len)
        PreOrder(f, GetLeftChildIndex(index));
    if (index >= 0 && index < len)
        f(&arr[index]);
    if (GetRightChildIndex(index) < len)
        PreOrder(f, GetRightChildIndex(index));
}
};
int main()
{
    Heap<int> Tree;

    Tree.push(1);
    Tree.push(-1);
    Tree.push(-2);
    Tree.push(2);
    Tree.push(5);
    Tree.push(6);
    Tree.push(-3);
    Tree.push(-4);
    cout << "\n-----\nStraight:";
    void(*f_ptr) (Node<int>*); f_ptr = print;
    Tree.Straight(f_ptr);
    return 0;
}
```

Задание 3.3

Используйте шаблон класса `FibonacciHeap` (фибоначчиева пирамида) для хранения объектов в соответствии с пунктом 2 задания №1 (используется упорядоченность по приоритету, на верхнем уровне структуры максимум). Реализуйте функцию удаления корня дерева `ExtractMax()`. Выведите элементы пирамиды в порядке убывания приоритета с её помощью. Замените указатели, дающие начало и конец связного списка элементов каждого уровня, на класс `list` библиотеки STL.