Практическая работа №2. Работа с ассоциативными контейнерами

Задание 2.1

Постройте сбалансированное дерево (используйте класс map библиотеки STL), которое содержит значения V по ключам K (таблица 2.1). Постройте функции поиска элемента по значению и по ключу. Постройте функцию вывода содержимого дерева с помощью итераторов. Постройте функцию filter(), которая принимает предикат P и возвращает новое дерево с объектами, для которых предикат принимает истинное значение (для всех вариантов условие предиката: значение поля V выше некоторого порога *threshold*, в случае хранения нескольких полей достаточно проверить одно из них).

Примечание: *В этом задании не требуется создавать класс дерева, нужно использовать класс map из библиотеки STL и написать отдельно требуемые функции (не методы класса).*

Код 2.1. Пример работы с контейнером map

```
//красно-черное (сблансированное) дерево map, есть интерфейс доступа к
значению по ключу

using namespace std;

#include <map>
#include <iostream>

int main()
{

   map<string, int> marks;
   marks["Petrov"] = 5;
   marks["Ivanov"] = 4;
   marks["Sidorov"] = 5;
   marks["Nikolaev"] = 3;
   marks["Abramov"] = 4;
   marks["Fedorov"] = 5;
   marks["Kuznetsov"] = 4;

   cout << "\nMap:\n";
   //итератор пробегает по map
   map<string, int>::iterator it_m = marks.begin();
   while (it_m != marks.end())
   {
```

```
            //перемещение по списку с помощью итератора, нет операции [i]
            cout << "Key: " << it_m->first << ", value: " << it_m->second <<
"\n";
            it_m++;
    }
}
```

Таблица 2.1. Ключи и хранимая в ассоциативном контейнере map информация

Вариант	Ключ K	Хранимая информация
1.	Адрес	«Объект жилой недвижимости». V: цена квартиры
2.	Название	«Сериал». V: рейтинг
3.	Название	«Смартфон». V: цена
4.	Фамилия и имя	«Спортсмен». V: количество медалей.
5.	Фамилия и имя	«Врач». V: рейтинг (вещественное число от 0 до 100) количество медалей
6.	Международный код	«Авиакомпания». V: количество обслуживаемых линий
7.	Название	«Книга». V: тираж
8.	Номер в небесном каталоге	«Небесное тело». V: расчётная масса в миллиардах тонн
9.	Название	«Населённый пункт». V: численность населения
10.	Имя или псевдоним исполнителя, название альбома	«Музыкальный альбом». V: количество проданных экземпляров
11.	Название фильма	«Фильм». V: доход

12.	Название производителя, имя модели	«Автомобиль». V: цена
13.	Регистрационный номер автомобиля	«Автовладелец». V: фамилия, имя
14.	Название, год постройки	«Стадион». V: вместимость
15.	Название, город	«Спортивная Команда». V: число побед, поражений, ничьих, количество очков
16.	Номер карты	«Пациент». V: группа крови
17.	Фамилия и имя	«Покупатель». V: средняя сумма чека
18.	Фамилия и имя	«Школьник». V: дата рождения
19.	Фамилия и имя	«Человек». V: адрес
20.	Название	«Государство». V: численность населения
21.	Адрес	«Сайт». V: количество посетителей в сутки.
22.	Название	«Программа». V: разработчик
23.	Производитель, модель	«Ноутбук». V: размер экрана, количество ядер, объем оперативной памяти
24.	Марка, диаметр колеса	«Велосипед». V: тип, наличие амортизаторов
25.	Фамилия и имя	«Программист». V: уровень (число от 1 до 10)

26.	Псевдоним	«Профиль в соц.сети». V: количество друзей
27.	Псевдоним	«Супергерой». V: суперсила
28.	Производитель, модель	«Фотоаппарат». V: размер матрицы, количество мегапикселей
29.	Полный адрес	«Файл». V: дата последнего изменения
30.	Производитель, название	«Самолет». V: дальность полета, максимальная скорость

Задание 2.2

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

Код 2.2. 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 2.3

Постройте шаблон сбалансированного дерева. Используйте его для хранения объектов класса C по ключам K в соответствии с таблицей (2.3). Переопределите функцию вывода содержимого дерева с помощью итераторов (в порядке возрастания / убывания ключей). Добавьте функции поиска элемента по ключу, значению.

Код 2.3. Класс бинарного дерева поиска

```
#include <iostream>

using namespace std;
//узел
template<class T>
class Node
{
protected:
    //закрытые переменные Node N; N.data = 10 вызовет ошибку
    T data;

    //не можем хранить Node, но имеем право хранить указатель
    Node* left;
    Node* right;
    Node* parent;

    //переменная, необходимая для поддержания баланса дерева
    int height;
public:
    //доступные извне переменные и функции
    virtual void setData(T d) { data = d; }
    virtual T getData() { return data; }
    int getHeight() { return height; }

    virtual Node* getLeft() { return left; }
    virtual Node* getRight() { return right; }
    virtual Node* getParent() { return parent; }

    virtual void setLeft(Node* N) { left = N; }
    virtual void setRight(Node* N) { right = N; }
    virtual void setParent(Node* N) { parent = N; }

    //Конструктор. Устанавливаем стартовые значения для указателей
    Node<T>(T n)
    {
           data = n;
           left = right = parent = NULL;
           height = 1;
    }

    Node<T>()
```

```
    {
        left = NULL;
        right = NULL;
        parent = NULL;
        data = 0;
        height = 1;
    }


    virtual void print()
    {
        cout << "\n" << data;
    }

    virtual void setHeight(int h)
    {
        height = h;
    }

    template<class T> friend ostream& operator<< (ostream& stream, Node<T>& N);
};

template<class T>
ostream& operator<< (ostream& stream, Node<T>& N)
{
    stream << "\nNode data: " << N.data << ", height: " << N.height;
    return stream;
}
template<class T>
void print(Node<T>* N) { cout << "\n" << N->getData(); }

template<class T>
class Tree
{
protected:
    //корень - его достаточно для хранения всего дерева
    Node<T>* root;
public:
    //доступ к корневому элементу
    virtual Node<T>* getRoot() { return root; }

    //конструктор дерева: в момент создания дерева ни одного узла нет, корень смотрит
в никуда
    Tree<T>() { root = NULL; }

    //рекуррентная функция добавления узла. Устроена аналогично, но вызывает сама себя
- добавление в левое или правое поддерево
    virtual Node<T>* Add_R(Node<T>* N)
    {
        return Add_R(N, root);
    }

    virtual Node<T>* Add_R(Node<T>* N, Node<T>* Current)
    {
```

```
            if (N == NULL) return NULL;
            if (root == NULL)
            {
                  root = N;
                  return N;
            }

            if (Current->getData() > N->getData())
            {
                  //идем влево
                  if (Current->getLeft() != NULL)
                        Current->setLeft(Add_R(N, Current->getLeft()));
                  else
                        Current->setLeft(N);
                  Current->getLeft()->setParent(Current);
            }
            if (Current->getData() < N->getData())
            {
                  //идем вправо
                  if (Current->getRight() != NULL)
                        Current->setRight(Add_R(N, Current->getRight()));
                  else
                        Current->setRight(N);
                  Current->getRight()->setParent(Current);
            }
            if (Current->getData() == N->getData())
                  //нашли совпадение
                  ;
            //для несбалансированного дерева поиска
            return Current;
      }

      //функция для добавления числа. Делаем новый узел с этими данными и вызываем нуж-
ную функцию добавления в дерево
      virtual void Add(int n)
      {
            Node<T>* N = new Node<T>;
            N->setData(n);
            Add_R(N);
      }

      virtual Node<T>* Min(Node<T>* Current=NULL)
      {
            //минимум - это самый "левый" узел. Идём по дереву всегда влево
            if (root == NULL) return NULL;

            if(Current==NULL)
                  Current = root;
            while (Current->getLeft() != NULL)
                  Current = Current->getLeft();

            return Current;
      }
```

```
	virtual Node<T>* Max(Node<T>* Current = NULL)
	{
		//минимум - это самый "правый" узел. Идём по дереву всегда вправо
		if (root == NULL) return NULL;

		if (Current == NULL)
			Current = root;
		while (Current->getRight() != NULL)
			Current = Current->getRight();

		return Current;
	}

	//поиск узла в дереве. Второй параметр - в каком поддереве искать, первый - что
искать
	virtual Node<T>* Find(int data, Node<T>* Current)
	{
		//база рекурсии
		if (Current == NULL) return NULL;

		if (Current->getData() == data) return Current;

		//рекурсивный вызов
		if (Current->getData() > data) return Find(data, Current->getLeft());

		if (Current->getData() < data) return Find(data, Current->getRight());


	}

	//три обхода дерева
	virtual void PreOrder(Node<T>* N, void (*f)(Node<T>*))
	{
		if (N != NULL)
			f(N);
		if (N != NULL && N->getLeft() != NULL)
			PreOrder(N->getLeft(), f);
		if (N != NULL && N->getRight() != NULL)
			PreOrder(N->getRight(), f);
	}

	//InOrder-обход даст отсортированную последовательность
	virtual void InOrder(Node<T>* N, void (*f)(Node<T>*))
	{
		if (N != NULL && N->getLeft() != NULL)
			InOrder(N->getLeft(), f);
		if (N != NULL)
			f(N);
		if (N != NULL && N->getRight() != NULL)
			InOrder(N->getRight(), f);
	}

	virtual void PostOrder(Node<T>* N, void (*f)(Node<T>*))
	{
```

```
            if (N != NULL && N->getLeft() != NULL)
                PostOrder(N->getLeft(), f);
            if (N != NULL && N->getRight() != NULL)
                PostOrder(N->getRight(), f);
            if (N != NULL)
                f(N);
        }
    };

int main()
{
        Tree<double> T;
        int arr[15];
        int i = 0;
        for (i = 0; i < 15; i++) arr[i] = (int)(100 * cos(15 * double(i+1)));
        for (i = 0; i < 15; i++)
            T.Add(arr[i]);

        Node<double>* M = T.Min();
        cout << "\nMin = " << M->getData() << "\tFind " << arr[3] << ": " <<
T.Find(arr[3], T.getRoot());

        void (*f_ptr)(Node<double>*); f_ptr = print;
        cout << "\n-----\nInorder:";
        T.InOrder(T.getRoot(), f_ptr);
        char c; cin >> c;
        return 0;
}
```

Таблица 2.3. Ключ и тип объекта, хранимого в контейнере АВЛ-дерево

Вари-ант	Ключ	Класс С
1.	Адрес	«Объект жилой недвижимости». Минимальный набор полей: адрес, тип (перечислимый тип: городской дом, загородный дом, квартира, дача), общая площадь, жилая площадь, цена.
2.	Название	«Сериал». Минимальный набор полей: название, продюсер, количество сезонов, популярность, рейтинг, дата запуска, страна.
3.	Название	«Смартфон». Минимальный набор полей: название, размер экрана, количество камер, объем аккумулятора, максимальное количество часов без подзарядки, цена.

4.	Фамилия и имя	«Спортсмен». Минимальный набор полей: фамилия, имя, возраст, гражданство, вид спорта, количество медалей.
5.	Фамилия и имя	«Врач». Минимальный набор полей: фамилия, имя, специальность, должность, стаж, рейтинг (вещественное число от 0 до 100).
6.	Международный код	«Авиакомпания». Минимальный набор полей: название, международный код, количество обслуживаемых линий, страна, интернет-адрес сайта, рейтинг надёжности (целое число от -10 до 10).
7.	Название	«Книга». Минимальный набор полей: фамилия (первого) автора, имя (первого) автора, название, год издания, название издательства, число страниц, вид издания (перечислимый тип: электронное, бумажное или аудио), тираж.
8.	Номер в каталоге	«Небесное тело». Минимальный набор полей: тип (перечислимый тип: астероид, естественный спутник, планета, звезда, квазар), имя (может отсутствовать), номер в небесном каталоге, удаление от Земли, расчётная масса в миллиардах тонн (для сверхбольших объектов допускается значение Inf, которое должно корректно обрабатываться).
9.	Название	«Населённый пункт». Минимальный набор полей: название, тип (перечислимый тип: город, посёлок, село, деревня), числовой код региона, численность населения, площадь.
10.	Имя или псевдоним исполните-	«Музыкальный альбом». Минимальный набор полей: имя или псевдоним исполнителя, название альбома, количество композиций, год выпуска, количество проданных экземпляров.

	ля, название альбома	
11.	Название фильма	«Фильм». Минимальный набор полей: фамилия, имя режиссёра, название, страна, год выпуска, стоимость, доход.
12.	Серийный номер	«Автомобиль». Минимальный набор полей: имя модели, название производителя, цвет, серийный номер, количество дверей, год выпуска, цена.
13.	Регистрационный номер автомобиля	«Автовладелец». Минимальный набор полей: фамилия, имя, регистрационный номер автомобиля, дата рождения, номер техпаспорта.
14.	Название, год постройки	«Стадион». Минимальный набор полей: название, виды спорта, год постройки, вместимость, количество арен.
15.	Название, город	«Спортивная Команда». Минимальный набор полей: название, город, число побед, поражений, ничьих, количество очков.
16.	Фамилия и имя	«Пациент». Минимальный набор полей: фамилия, имя, дата рождения, телефон, адрес, номер карты, группа крови.
17.	Фамилия и имя	«Покупатель». Минимальный набор полей: фамилия, имя, город, улица, номера дома и квартиры, номер счёта, средняя сумма чека.
18.	Фамилия и имя	«Школьник». Минимальный набор полей: фамилия, имя, пол, класс,

		дата рождения, адрес.
19.	Фамилия и имя	«Человек». Минимальный набор полей: фамилия, имя, пол, рост, возраст, вес, дата рождения, телефон, адрес.
20.	Название	«Государство». Минимальный набор полей: название, столица, язык, численность населения, площадь.
21.	Адрес	«Сайт». Минимальный набор полей: название, адрес, дата запуска, язык, тип (блог, интернет-магазин и т.п.), cms, дата последнего обновления, количество посетителей в сутки.
22.	Название	«Программа». Минимальный набор полей: название, версия, лицензия, есть ли версия для android, iOS, платная ли, стоимость, разработчик, открытость кода, язык кода.
23.	Производитель, модель	«Ноутбук». Минимальный набор полей: производитель, модель, размер экрана, процессор, количество ядер, объем оперативной памяти, объем диска, тип диска, цена.
24.	Марка, диаметр колеса	«Велосипед». Минимальный набор полей: марка, тип, тип тормозов, количество колес, диаметр колеса, наличие амортизаторов, детский или взрослый.
25.	Фамилия и имя	«Программист». Минимальный набор полей: фамилия, имя, email, skype, telegram, основной язык программирования, текущее место работы, уровень (число от 1 до 10).
26.	Псевдоним	«Профиль в соц.сети». Минимальный набор полей: псевдоним, адрес страницы, возраст, количество друзей, интересы, любимая цитата.
27.	Псевдоним	«Супергерой». Минимальный набор полей: псевдоним, настоящее имя,

		дата рождения, пол, суперсила, слабости, количество побед, рейтинг силы.
28.	Производитель, модель	«Фотоаппарат». Минимальный набор полей: производитель, модель, тип, размер матрицы, количество мегапикселей, вес, тип карты памяти, цена.
29.	Полный адрес	«Файл». Минимальный набор полей: полный адрес, краткое имя, дата последнего изменения, дата последнего чтения, дата создания.
30.	Производитель, название	«Самолет». Минимальный набор полей: название, производитель, вместимость, дальность полета, максимальная скорость.

Задание 2.4

Используйте шаблон класса Heap (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.

Код 2.4. Класс Heap (куча, пирамида)

```
#include <iostream>

using namespace std;

//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:
	//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];
		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];
	}

	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;
	}

	//просеить 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)
		{
			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 Add(T v)
	{
		Node<T>* N = new Node<T>;
		N->setValue(v);
		Add(N);
	}
```

```
	template <class T>
	void Add(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++)
		{
			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.Add(1);
	Tree.Add(-1);
	Tree.Add(-2);
	Tree.Add(2);
	Tree.Add(5);
	Tree.Add(6);
	Tree.Add(-3);
	Tree.Add(-4);
	Tree.Add(4);
	Tree.Add(3);

	cout << "\n-----\nStraight:";
	void(*f_ptr)(Node<int>*); f_ptr = print;
	Tree.Straight(f_ptr);

	char c; cin >> c;
	return 0;}
```