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

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Алгоритмы и структуры данных»

Тема: Ассоциативный массив

Студент гр. 8301 _____ Пчёлко В.А.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург

2020

Оглавление

Цель работы

Реализовать шаблонный ассоциативный массив (map) на основе красно-черного дерева на C++.

1 Ход работы

1.1 Постановка задачи

Реализовать на основе красно-черного дерева шаблонный класс - ассоциативный массив (map), реализовать следующие функции и написать unit тесты:
Список методов:

1. insert(ключ, значение) // добавление элемента с ключом и значением
2. Delete(ключ) // удаление элемента дерева по ключу
3. find(ключ) // поиск элемента по ключу
4. clear // очищение ассоциативного массива
5. get_keys // возвращает список ключей
6. get_values // возвращает список значений
7. treeprint // вывод в консоль

1.2 Описание пользовательских типов данных

Класс Map состоит из вложенного класса Node с полями: T key (Ключ по которому хранится значение) и T1 value (Значение которое хранится по определённому ключу) bool color (цвет ячейки необходимый для дальнейшей проверки сбалансированности), Node* parent(указатель на родителя ячейки), Node* left и Node* right (Указатели на левые и правые ячейки(детей)), а также и собственные поля Node* Top (Вершина дерева), Node* Leaf(Обозначение пустого листа). Реализованный мною класс Map основан на такой структуре данных как красно-чёрное дерево.

Красно-черное дерево требует соблюдения ряда правил. При добавлении и удалении элементов происходит балансировка дерева.

Класс содержит следущие методы:

- Констуктор – реализованы конструкторы по умолчанию для вложенного класса Node и для самого класса Map.
- Деструктор – реализован деструктор, вызывающий метод clear().
- *insert(T key,T1 value)* – функция добавления элемента в дерево по ключу. Добавление происходит точно так же как и в бинарном дереве. Дальше следует проверка на соблюдение 5 свойств, иначе происходит перебалансировка.

- *Delete(T key)* – функция удаления элемента по ключу.
- *find(T key)* – функция получения значения по ключу.
- *clear()* – функция, по одному удаляющая элементы постфиксным обходом дерева.
- *get_keys()* – функция, возвращающая список ключей.
- *get_values()* – функция, возвращающая список значений.
- *treeprint()*– функция вывода дерева.
- *Ряд вспомогательных методов*

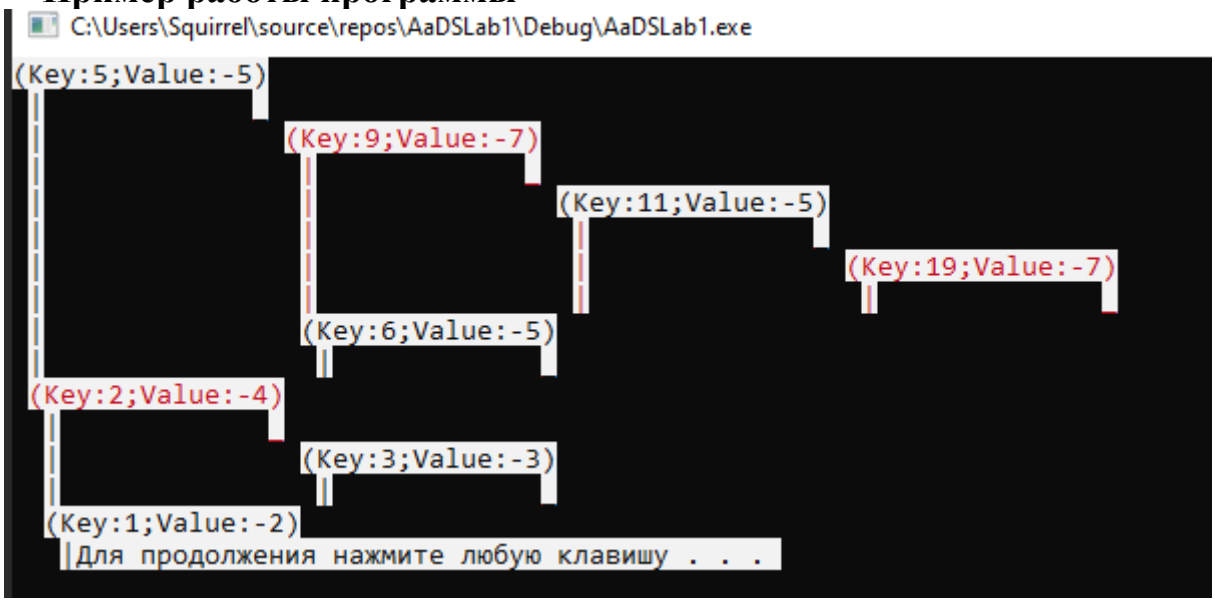
1.3 Оценка временной сложности методов

метод	Временная сложность
Delete(T key)	O(log n)
insert(T key,T1 value)	O(log n)
find(T key)	O(log n)
clear()	O(n)
get_keys()	O(n)
get_values()	O(n)
treeprint()	O(n)

1.4 Описание реализованных unit-тестов

В тестах реализованных для класса Map мы проверили функцию добавления элемента в дерево с помощью функции insert() и удаляем с помощью функции remove(), и проверяем их с помощью функции find() которая возвращает нам значение по ключу или бросает исключение. Так же мои unit-тесты затрагивают такие методы как get_keys() и get_values(), которые возвращают списки ключей и значений, а также метод очистки дерева clear().

1.5 Пример работы программы



1.6 Код программы

Map.h

```
#define RED true
#define BLACK false

#include"List.cpp"
#include<string>
#include<Windows.h>

using namespace std;

template<typename T, typename T1>
class Map {
public:
	class Node
	{
	public:
		Node(bool color = RED, T key = T(), Node* parent = NULL, Node* left = NULL,
Node* right = NULL, T1 value = T1()) :color(color), key(key), parent(parent), left(left),
right(right), value(value) {}
		T key;
		T1 value;
		bool color;
		Node* parent;
		Node* left;
		Node* right;
	};

	~Map() {
		if (this->Top != NULL)
			this->clear();
		Top = NULL;
		delete Leaf;
		Leaf = NULL;
	}

	Map(Node* Top = NULL, Node* Leaf = new Node(0)) :Top(Top), Leaf(Leaf) {}

	void treeprint();

	void  insert(T key, T1 value);

	List<T>* get_keys();
	List<T1>* get_values();
	T1 find(T key);
	void clear();

	void Delete(T key);

private:
	Node* Top;
	Node* Leaf;

	void deleteNode(Node* current);
	void clear_tree(Node* tree);

	void ListKeyOrValue(int mode, List<T>* list);
	void KeyOrValue(Node* tree, List<T>* list, int mode);
	Node* minimum(Node* node);

	Node* maximum(Node* node);
```

```
        Node* grandparent(Node* cur);

        Node* uncle(Node* cur);

        Node* sibling(Node* cur);

        Node* find_key(T key);
        //all print function
        void Drawline(short x, short y, short old_y);
        void go(short x, short y);
        void Color(int front, int back);
        int numberDigits(int a);
        void ConsoleOut(Node *tree, short &Y, short &Xright);

        //balance tree
        void BalanceTree(Node* node);


        //Rotation
        void left_rotate(Node* node);
        void right_rotate(Node* node);
                                                };
```

Map.cpp

```
#include "map.h"

template<typename T, typename T1> void  Map<T, T1>::insert(T key, T1 value)
{

        if (this->Top != NULL) {
                Node* node = NULL;
                Node* parent = NULL;
                for (node = this->Top; node != Leaf; )// Search leaf for new element
                {
                        parent = node;
                        if (key < node->key)
                                node = node->left;
                        else if (key > node->key)
                                node = node->right;
                        else if (key == node->key)
                                throw std::out_of_range("key is repeated");
                }
                node = new Node(RED, key, Leaf, Leaf, Leaf, value);
                node->parent = parent;
                if (parent != Leaf) {
                        if (key < parent->key)
                                parent->left = node;
                        else
                                parent->right = node;
                }
                BalanceTree(node);
        }
        else {
                this->Top = new Node(BLACK, key, Leaf, Leaf, Leaf, value);
        }
}

template<typename T, typename T1> List<T>* Map<T, T1>::get_keys() {
        List<T>* list = new List<T>();
        this->ListKeyOrValue(1, list);
        return list;
```

```
}

template<typename T, typename T1> List<T1>* Map<T, T1>::get_values() {
	List<T1>* list = new List<T1>();
	this->ListKeyOrValue(2, list);
	return list;
}

template<typename T, typename T1> T1 Map<T, T1>::find(T key) {
	Node* node = Top;
	while (node != Leaf && node->key != key) {
		if (node->key > key)
			node = node->left;
		else
			if (node->key < key)
				node = node->right;
	}
	if (node != Leaf)
		return node->value;
	else
		throw std::out_of_range("Key is missing");
}

template<typename T, typename T1> void Map<T, T1>::Delete(T key)
{
	Node* needed = Top;
	// находим узел с ключом key
	while (needed->key != key) {
		if (needed->key < key)
			needed = needed->right;
		else
			needed = needed->left;
	}
	deleteNode(needed);
}

template<typename T, typename T1> void Map<T, T1>::deleteNode(Node* current) {
	Node* fixed, *changed;


	// delete node current from tree

	if (!current || current == Leaf) return;


	if (current->left == Leaf || current->right == Leaf) {
		/* changed has a Leaf node as a child */
		changed = current;
	}
	else {
		/* find tree successor with a Leaf node as a child */
		changed = current->right;
		while (changed->left != Leaf) changed = changed->left;
	}

	/* fixed is changed's only child */
	if (changed->left != Leaf)
		fixed = changed->left;
	else
		fixed = changed->right;

	/* remove changed from the parent chain */
	fixed->parent = changed->parent;
	if (changed->parent)
```

```
                if (changed == changed->parent->left)
                        changed->parent->left = fixed;
                else
                        changed->parent->right = fixed;
        else
                Top = fixed;

        if (changed != current) {
                current->key = changed->key;
                current->value = changed->value;
        }

        if (changed->color == BLACK)
                BalanceTree(fixed);
}

template<typename T, typename T1> void Map<T, T1>::clear() {
        this->clear_tree(this->Top);
        this->Top = NULL;
}

template<typename T, typename T1> void Map<T, T1>::clear_tree(Node* tree) {
        if (tree != Leaf) {
                clear_tree(tree->left);
                clear_tree(tree->right);
                delete tree;
        }
}

template<typename T, typename T1> typename Map<T, T1>::Node* Map<T, T1>::minimum(Node* node)
//finding minimal element
{
        while (node->left != Leaf)
        {
                node = node->left;
        }
        return node;
}

template<typename T, typename T1> typename Map<T, T1>::Node* Map<T, T1>::maximum(Node*
node)//finding max element
{
        while (node->right != Leaf)
        {
                node = node->right;
        }
        return node;
}

template<typename T, typename T1> typename Map<T, T1>::Node* Map<T, T1>::grandparent(Node*
cur)//finding  grandparent
{
        if ((cur != Leaf) && (cur->parent != Leaf))
                return cur->parent->parent;
        else
                return Leaf;
}

template<typename T, typename T1> typename Map<T, T1>::Node* Map<T, T1>::uncle(Node* cur)
//finding uncle
{
        Node* cur1 = grandparent(cur); //assisting node
        if (cur1 == Leaf)
                return Leaf; // No grandparent means no uncle
```

```
	if (cur->parent == cur1->left)
		return cur1->right;
	else
		return cur1->left;
}

template<typename T, typename T1> typename Map<T, T1>::Node* Map<T, T1>::sibling(Node* cur)
{
	if (cur == cur->parent->left)
		return cur->parent->right;
	else
		return cur->parent->left;
}

template<typename T, typename T1> void Map<T, T1>::ListKeyOrValue(int mode, List<T>* list) {
	if (this->Top != Leaf)
		this->KeyOrValue(Top, list, mode);
	else
		throw std::out_of_range("Tree empty!");
}

template<typename T, typename T1> void Map<T, T1>::KeyOrValue(Node* tree, List<T>* list, int
mode) {

	if (tree != Leaf) {
		KeyOrValue(tree->left, list, mode);
		if (mode == 1)
			list->push_back(tree->key);
		else
			list->push_back(tree->value);
		KeyOrValue(tree->right, list, mode);
	}
}

template<typename T, typename T1> typename Map<T, T1>::Node* Map<T, T1>::find_key(T key) {
	Node* node = this->Top;
	while (node != Leaf && node->key != key) {
		if (node->key > key)
			node = node->left;
		else
			if (node->key < key)
				node = node->right;
	}
	if (node != Leaf)
		return node;
	else
		throw std::out_of_range("Key is missing");
}
//all print function
template<typename T, typename T1> void Map<T, T1>::go(short x, short y) {
	HANDLE Out = GetStdHandle(STD_OUTPUT_HANDLE);
	COORD coord = { x,y };
	SetConsoleCursorPosition(Out, coord);
}
template<typename T, typename T1> void Map<T, T1>::Drawline(short x, short y, short old_y)
{
	while (old_y != y)
	{
		go(x, old_y);
		cout << "|";
		old_y++;
	}
}
template<typename T, typename T1> int Map<T, T1>::numberDigits(int a)
```

```
{
	int count = 0;
	if (a < 0)
		count++;
	while (a > 0)
	{
		a = a / 10;
		count++;
	}
	return count;
}
template<typename T, typename T1> void Map<T, T1>::ConsoleOut(Node *tree, short &Y, short
&Xright) {
	short old_y;
	short Xleft;
	if (tree != Leaf)
	{
		if (tree->color)
			Color(4, 15);
		else
			Color(0, 15);
		go(Xright, Y);
		cout << "(Key:" << tree->key << ";Value:" << tree->value << ")";
		old_y = Y;
		Xleft = Xright;
		Xright = Xright + numberDigits(tree->value) + numberDigits(tree->key) + 13;
		Y += 1;
		go(Xright, Y);
		if (tree->left != NULL) {
			if (tree->right == NULL)
				cout << "|";
		}
		else {
			if (tree->right != NULL) {
				cout << "|";
			}
		}
		if (tree->right != NULL) {
			cout << "_";
		}
		else
			cout << " ";
		Y += 1;
		Xright += 2;
		Xleft++;
		if (tree->right != NULL)
		{
			ConsoleOut(tree->right, Y, Xright);
		}
		if (tree->left != NULL)
		{
			Drawline(Xleft, Y, old_y + 1);
			ConsoleOut(tree->left, Y, Xleft);
		}
		else {
			return;
		}
	}
	else {
		return;
	}
}
template<typename T, typename T1> void Map<T, T1>::Color(int text, int background) {
	HANDLE Out = GetStdHandle(STD_OUTPUT_HANDLE);
```

```
	SetConsoleTextAttribute(Out, (WORD)((background << 4) | text));
}
template<typename T, typename T1> void Map<T, T1>::treeprint() {
	short down = 0;
	short right = 0;
	this->ConsoleOut(Top, down, right);
}

//fix before add
template<typename T, typename T1> void Map<T, T1>::BalanceTree(Node* node)
{
	Node* uncle;
	/* Current node is RED */
	while (node != this->Top && node->parent->color == RED)//
	{
		/* node in left tree of grandfather */
		if (node->parent == this->grandparent(node)->left)//
		{
			/* node in left tree of grandfather */
			uncle = this->uncle(node);
			if (uncle->color == RED) {
				/* Case 1 - uncle is RED */
				node->parent->color = BLACK;
				uncle->color = BLACK;
				this->grandparent(node)->color = RED;
				node = this->grandparent(node);
			}
			else {
				/* Cases 2 & 3 - uncle is BLACK */
				if (node == node->parent->right) {
					/*Reduce case 2 to case 3 */
					node = node->parent;
					this->left_rotate(node);
				}
				/* Case 3 */
				node->parent->color = BLACK;
				this->grandparent(node)->color = RED;
				this->right_rotate(this->grandparent(node));
			}
		}
		else {
			/* Node in right tree of grandfather */
			uncle = this->uncle(node);
			if (uncle->color == RED) {
				/* Uncle is RED */
				node->parent->color = BLACK;
				uncle->color = BLACK;
				this->grandparent(node)->color = RED;
				node = this->grandparent(node);
			}
			else {
				/* Uncle is BLACK */
				if (node == node->parent->left) {
					node = node->parent;
					this->right_rotate(node);
				}
				node->parent->color = BLACK;
				this->grandparent(node)->color = RED;
				this->left_rotate(this->grandparent(node));
			}
		}
	}
	this->Top->color = BLACK;
}
```

```
//Rotates
template<typename T, typename T1> void Map<T, T1>::left_rotate(Node* node)
{
	Node* right = node->right;
	/* Create node->right link */
	node->right = right->left;
	if (right->left != Leaf)
		right->left->parent = node;
	/* Create right->parent link */
	if (right != Leaf)
		right->parent = node->parent;
	if (node->parent != Leaf) {
		if (node == node->parent->left)
			node->parent->left = right;
		else
			node->parent->right = right;
	}
	else {
		this->Top = right;
	}
	right->left = node;
	if (node != Leaf)
		node->parent = right;
}

template<typename T, typename T1> void Map<T, T1>::right_rotate(Node* node)
{
	Node* left = node->left;
	/* Create node->left link */
	node->left = left->right;
	if (left->right != Leaf)
		left->right->parent = node;
	/* Create left->parent link */
	if (left != Leaf)
		left->parent = node->parent;
	if (node->parent != Leaf) {
		if (node == node->parent->right)
			node->parent->right = left;
		else
			node->parent->left = left;
	}
	else {
		this->Top = left;
	}
	left->right = node;
	if (node != Leaf)
		node->parent = left;
                                                }
```

Выводы

Реализовал ассоциативный массив на основе красно-черного дерева в C++.