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

Вариант: 1

Студент гр. 8309 Носов А.С.

Преподаватель Тутуева А.В.

Санкт-Петербург

2019

Оглавление

Постановка задачи. Описание реализуемого класса и методов

Необходимо создать шаблонный класс, в котором будет 7 методов доступных пользователю. К каждому методу в классе необходимо создать Unit-тест, который будет проверять правильность работы методов.

Описание методов:

1. void print(): Выводит дерево псевдографикой в консоль.
2. bool contains(TKEY key): Проверка наличия в дереве необходимого ключа.
3. void insert(TKEY key, TDATA data): Вставка в красно-черное дерево как в двоичное дерево, после вызывается метод insertFixup().
4. void remove(TKEY key): Удаление узла, если узел черный вызывается removeFixup().
5. void clear(): Полное удаление дерева.
6. void getKeys(): Вывод списка ключей.
7. void getValue(): Вывод списка данных.
8. void clearLock(nodeRBT<TKEY, TDATA> *&head): Вывод размера списка.
9. void inrDatas(nodeRBT<TKEY, TDATA> *current): Центрированный обход для данных.
10. void inrKeys(nodeRBT<TKEY, TDATA> *current): Центрированный обход для ключей.
11. void nwInsert(nodeRBT<TKEY, TDATA> *&current, nodeRBT<TKEY, TDATA> *&nwNode()): Реверсивная вставка узла.
12. void insertFixup(nodeRBT<TKEY, TDATA> *&nwNode): Балансировка дерева после вставки узла:
    - если узел красный, то перекрашивается его предок и прапредок.
    - если узел черный, то поворачиваем ветви налево или направо в зависимости от предка.
    - если узел черный, то перекрашиваются его предок и прапредок и выполняется поворот.
13. void leftRotate (nodeRBT<TKEY, TDATA> *current): Поворот налево.
14. void rightRotate (nodeRBT<TKEY, TDATA> *current): Поворот направо.
15. nodeRBT<TKEY, TDATA> *dlFind(nodeRBT<TKEY, TDATA> *current, TKEY key): Рекурсивный поиск узла для удаления.
16. nodeRBT<TKEY, TDATA> *treeMinimum(nodeRBT<TKEY, TDATA> *current): Проход до корней по левым ветвям.
17. void transplant(nodeRBT*& current, nodeRBT*& additional): Заменяем одно поддерево, являющееся дочерним по отношению к своему родителю, другим поддеревом.
18. void removeFixup(nodeRBT*& current): Балансировка дерева после удаления черного узла:
    - если брат, то перекрашиваем его в черный, а предка в красный. Поворачиваем налево или направо в зависимости от предка.
    - если у брата два потомка черные, то перекрашиваем брата в красный.
    - если у брата один потомок черный, то делаем второго потомка черный, а брата красным и выполняем поворот.
    - брат присваивает цвет потомка, а его красим в черный вместе с правым или левым потомком брата, выполняем поворот.
19. bool find(nodeRBT* current, TKEY key): Рекурсивный поиск элемента для метода contains.
20. int print_in_massive(nodeRBT * tree, int is_left, int offset, int depth, char s[30][255]): Рисует дерево.

Оценка временной сложности каждого метода

Временная сложность оценена с помощью модульных тестов.

1. void print(): O(N+K).

2. bool contains(TKEY key): O(logN)
3. void insert(TKEY key, TDATA data): 1) O(1) – вставка корня. 2) O(logN + logK)
4. void remove(TKEY key): 1) O(1) – удаление красного узла. 2) O(logN) – удаление черного узла.
5. void clear(): O(N)
6. void getKeys(): O(N)
7. void getValue(): O(N)
8. void clearLock(nodeRBT<TKEY, TDATA> *&head): O(N)
9. void inrDatas(nodeRBT<TKEY, TDATA> *current): O(N)
10. void inrKeys(nodeRBT<TKEY, TDATA> *current): Центрированный обход для ключей.
11. void nwInsert(nodeRBT<TKEY, TDATA> *&current, nodeRBT<TKEY, TDATA> *&nwNode()): O(logN)
12. void insertFixup(nodeRBT<TKEY, TDATA> *&nwNode): O(N)
13. void leftRotate (nodeRBT<TKEY, TDATA> *current): O(1)
14. void rightRotate (nodeRBT<TKEY, TDATA> *current): O(1)
15. nodeRBT<TKEY, TDATA> *dlFind(nodeRBT<TKEY, TDATA> *current, TKEY key): O(logN)
16. nodeRBT<TKEY, TDATA> *treeMinimum(nodeRBT<TKEY, TDATA> *current): O(N)
17. void transplant(nodeRBT*& current, nodeRBT*& additional): O(N)
18. void removeFixup(nodeRBT*& current): O(N)
19. bool find(nodeRBT* current, TKEY key): O(logN)
20. int print_in_massive(nodeRBT * tree, int is_left, int offset, int depth, char s[30][255]): O(N+K)

Описание реализованных unit-тестов

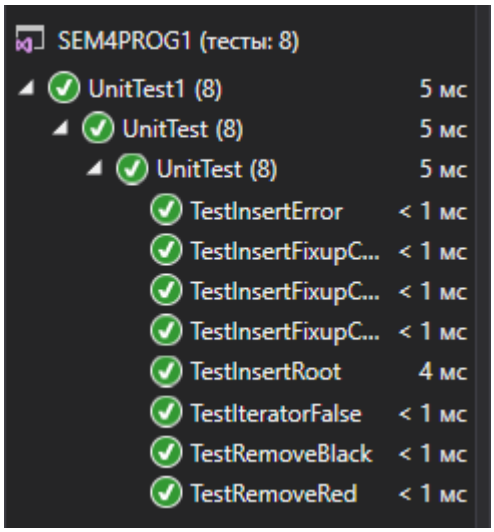
Модульные тесты нужны для сравнительно эффективного тестирования программы, удобный как для разработчика, так и другого пользователя.

Я использовал предустановленный в Visual Studio шаблон для написания unit-тестов. Созданная мною среда для каждого теста была примерно одна и та же, в основном отличаясь лишь наличием того или иного метода.

Ключевым элементов всех моих тестов было наличие сравнение практического и теоретического результата с помощью Assert::AreEqual().

Проект с юнит-тестами был реализован в одном решении с основной программой

Пример работы



Листинг

Head.h

```
#pragma once
#include <stdio.h>
#include <iostream>

using namespace std;

template<typename T>
class Iterator
{
public:

	virtual T next() = 0;
	virtual bool has_next() = 0;

};

template <typename TKEY, typename TDATA>
class AArrey
{
public:
	AArrey();
	~AArrey();

	void print()
	{
		char s[30][255];
		for (int i = 0; i < 30; i++)
			sprintf_s(s[i], "%200s", "|");

		print_in_massive(Root, 0, 0, 0, s);
```

```
	for (int i = 0; i < 30; i++)
		cout << s[i] << endl;
}

bool contains(const TKEY key)
{
	if (Root == NIL)
	{
		return false;
	}
	if (Root->key == key)
	{
		return true;
	}
	if (Root->key > key)
	{
		return find(Root->left, key);
	}
	else
	{
		return find(Root->right, key);
	}
}

void insert(const TKEY  key, const TDATA data)
{
	if (contains(key) == true)
	{
		throw invalid_argument("Error - This key is already in memory!");
	}

	nodeRBT<TKEY, TDATA>* nwNode = new nodeRBT<TKEY, TDATA>;
```

```
        nwNode->key = key;
        nwNode->data = data;
        nwNode->left = NIL;
        nwNode->right = NIL;
        nwNode->parent = NIL;

        if (Root == NIL)
        {
                Root = nwNode;
        }
        else
        {
                nwNode->color = true; // true -> red
                nwInsert(Root, nwNode);
        }
        insertFixup(nwNode);
}

void remove(const TKEY key)
{

        nodeRBT<TKEY, TDATA>* dlNode = dlFind(Root, key);
        if (dlNode == NIL)
        {
                throw out_of_range("Error - This key isn't in memory!");
        }
        nodeRBT<TKEY, TDATA>* additional = dlNode;
        nodeRBT<TKEY, TDATA>* current;
        bool dlNodeColor = additional->color;
        if (dlNode->left == NIL)
        {
                current = dlNode->right;
                transplant(dlNode, dlNode->right);
```

```
	}
	else if (dlNode->right == NIL)
	{
		current = dlNode->left;
		transplant(dlNode, dlNode->left);
	}
	else
	{
		additional = treeMinimum(dlNode->right);
		dlNodeColor = additional->color;
		current = additional->right;
		if (additional->parent == dlNode)
		{
			current->parent = additional;
		}
		else
		{
			transplant(additional, additional->right);
			additional->right = dlNode->right;
			additional->right->parent = additional;
		}
		transplant(dlNode, additional);
		additional->left = dlNode->left;
		additional->left->parent = additional;
		additional->color = dlNode->color;
	}
	delete dlNode;
	if (dlNodeColor == false) removeFixup(current);

}

void clear()
{
```

```
        if (Root != NIL) clearLock(Root);
        Root = NIL;
    }

    void getKeys()
    {
        cout << "key on as.array\n";
        inrKeys(Root);
    }

    void getValue()
    {
        cout << "data on as.array\n";
        inrDatas(Root);
    }

private:

    //first Key,second Data
    template<typename T, typename U>
    struct nodeRBT
    {
        nodeRBT* parent = nullptr;
        nodeRBT* left = nullptr;
        nodeRBT* right = nullptr;
        T key;
        U data;
        bool color = false;//false - black true - red
    };
    nodeRBT<TKEY, TDATA>* NIL;
    nodeRBT<TKEY, TDATA>* Root;

    void clearLock(nodeRBT<TKEY, TDATA> * &head)
```

```
{
	if (head == NIL) return;
	clearLock(head->left);
	clearLock(head->right);
	delete head;

}

//Centered tree walk for data
void inrDatas(nodeRBT<TKEY, TDATA> * current)
{

	if (current == NIL)
	{
		return;
	}
	inrDatas(current->left);
	cout << current->data << endl;
	inrDatas(current->right);
}

//Centered tree walk for keys
void inrKeys(nodeRBT<TKEY, TDATA> * current)
{

	if (current == NIL)
	{
		return;
	}
	inrKeys(current->left);
	cout << current->key << endl;
	inrKeys(current->right);
}
```

```
void nwInsert(nodeRBT<TKEY, TDATA> * &current, nodeRBT<TKEY, TDATA> * &nwNode)
{
	if (nwNode->key <= current->key)
	{
		if (current->left == NIL)
		{
			current->left = nwNode;
			nwNode->parent = current;
		}
		else
		{
			nwInsert(current->left, nwNode);
		}
	}
	if (nwNode->key > current->key)
	{
		if (current->right == NIL)
		{
			current->right = nwNode;
			nwNode->parent = current;
		}
		else
		{
			nwInsert(current->right, nwNode);
		}
	}
}

void insertFixup(nodeRBT<TKEY, TDATA> * &nwNode)
{
	while (nwNode->parent->color == true)
	{
```

```
                if (nwNode->parent == nwNode->parent->parent->left)
                {
                        nodeRBT<TKEY, TDATA>* gUncle = nwNode->parent->parent-
>right;
                        if (gUncle->color == true)
                        {
                                nwNode->parent->color = false;
                                gUncle->color = false;
                                nwNode->parent->parent->color = true;
                                nwNode = nwNode->parent->parent;
                        }
                        else
                        {
                                if (nwNode == nwNode->parent->right)
                                {
                                        nwNode = nwNode->parent;
                                        leftRotate(nwNode);
                                }
                                nwNode->parent->color = false;
                                nwNode->parent->parent->color = true;
                                rightRotate(nwNode->parent->parent);
                        }
                }
                else
                {
                        nodeRBT<TKEY, TDATA>* gUncle = nwNode->parent->parent-
>left;
                        if (gUncle->color == true)
                        {
                                nwNode->parent->color = false;
                                gUncle->color = false;
                                nwNode->parent->parent->color = true;
                                nwNode = nwNode->parent->parent;
                        }
```

```
                else
                {
                        if (nwNode == nwNode->parent->left)
                        {
                                nwNode = nwNode->parent;
                                rightRotate(nwNode);
                        }
                        nwNode->parent->color = false;
                        nwNode->parent->parent->color = true;
                        leftRotate(nwNode->parent->parent);
                }
        }
    }
    Root->color = false;
}

void leftRotate(nodeRBT<TKEY, TDATA> * current)
{
    nodeRBT<TKEY, TDATA>* rCurrent = current->right;
    current->right = rCurrent->left;
    if (rCurrent->left != NIL)
    {
        rCurrent->left->parent = current;
    }
    rCurrent->parent = current->parent;
    if (current->parent == NIL)
    {
        Root = rCurrent;
    }
    else if (current->parent->left == current)
    {
        current->parent->left = rCurrent;
    }
```

```
        else
        {
                current->parent->right = rCurrent;
        }
        rCurrent->left = current;
        current->parent = rCurrent;
}

void rightRotate(nodeRBT<TKEY, TDATA> * current)
{
        nodeRBT<TKEY, TDATA>* lCurrent = current->left;
        current->left = lCurrent->right;
        if (lCurrent->right != NIL)
        {
                lCurrent->right->parent = current;
        }
        lCurrent->parent = current->parent;
        if (current->parent == NIL)
        {
                Root = lCurrent;
        }
        else if (current->parent->right == current)
        {
                current->parent->right = lCurrent;
        }
        else
        {
                current->parent->left = lCurrent;
        }
        lCurrent->right = current;
        current->parent = lCurrent;
}
```

```
	//Search for an item in the tree
	nodeRBT<TKEY, TDATA>* dlFind(nodeRBT<TKEY, TDATA> * current, TKEY key)
	{
		if (current == NIL)
		{
			return NIL;
		}
		if (current->key == key)
		{
			return current;
		}
		if (current->key > key)
		{
			return dlFind(current->left, key);
		}
		else
		{
			return dlFind(current->right, key);
		}
	}

	nodeRBT<TKEY, TDATA>* treeMinimum(nodeRBT<TKEY, TDATA> * current)
	{
		while (current->left != NIL)
		{
			current = current->left;
		}
		return current;
	}

	void transplant(nodeRBT<TKEY, TDATA> * &current, nodeRBT<TKEY, TDATA> *
&additional)
	{
```

```
		if (current->parent == NIL)
		{
			Root = additional;
		}
		else if (current == current->parent->left)
		{
			current->parent->left = additional;
		}
		else
		{
			current->parent->right = additional;
		}
		additional->parent = current->parent;
	}

	void removeFixup(nodeRBT<TKEY, TDATA> * &current)
	{
		while (current != Root && current->color == false)
		{
			if (current == current->parent->left)
			{
				nodeRBT<TKEY, TDATA>* brother = current->parent->right;
				if (brother->color == true)
				{
					brother->color = false;
					current->parent->color = true;
					leftRotate(current->parent);
					brother = current->parent->right;
					return;
				}
				if ((brother->left->color == false) && (brother->right-
>color == false))
				{
```

```
                brother->color = true;
                current = current->parent;
                return;
            }
            else if (brother->right->color == false)
            {
                brother->left->color = false;
                brother->color = true;
                rightRotate(brother);
                brother = current->parent->right;
                return;
            }
            brother->color = current->parent->color;
            current->parent->color = false;
            brother->right->color = false;
            leftRotate(current->parent);
            current = Root;
        }
        else
        {
            nodeRBT<TKEY, TDATA>* brother = current->parent->left;
            if (brother->color == true)
            {
                brother->color = false;
                current->parent->color = true;
                rightRotate(current->parent);
                brother = current->parent->left;
                return;
            }
            if (brother->right->color == false && brother->left->color
== false)
            {
                brother->color = true;
```

```
                        current = current->parent;
                        return;
                }
                else if (brother->left->color == false)
                {
                        brother->right->color = false;
                        brother->color = true;
                        leftRotate(brother);
                        brother = current->parent->left;
                        return;
                }
                brother->color = current->parent->color;
                current->parent->color = false;
                brother->left->color = false;
                rightRotate(current->parent);
                current = Root;
        }
    }
}

//search items
bool find(nodeRBT<TKEY, TDATA> * current, TKEY key)
{
    if (current == NIL)
    {
        return false;
    }
    if (current->key == key)
    {
        return true;
    }
    if (current->key > key)
    {
```

```
                return find(current->left, key);
        }
        else
        {
                return find(current->right, key);
        }
    }


    //This code better don't touch
    int print_in_massive(nodeRBT<TKEY, TDATA> * tree, int is_left, int offset, int
depth, char s[30][255])
    {
        char b[30];
        int width = 7;
        if (!tree) return 0;
        if (tree->color == false) sprintf_s(b, "B%05dB", tree->key);
        if (tree->color == true) sprintf_s(b, "R%05dR", tree->key);
        int left = print_in_massive(tree->left, 1, offset, depth + 1, s);
        int right = print_in_massive(tree->right, 0, offset + left + width,
depth + 1, s);

        for (int i = 0; i < width; i++)
            s[2 * depth][offset + left + i] = b[i];

        if (depth && is_left) {

            for (int i = 0; i < width + right; i++)
                s[2 * depth - 1][offset + left + width / 2 + i] = '-';

            s[2 * depth - 1][offset + left + width / 2] = '.';
            s[2 * depth - 1][offset + left + width + right + width / 2] =
'+';

        }
```

```
		else if (depth && !is_left) {


			for (int i = 0; i < left + width; i++)

				s[2 * depth - 1][offset - width / 2 + i] = '-';


			s[2 * depth - 1][offset + left + width / 2] = '.';

			s[2 * depth - 1][offset - width / 2 - 1] = '+';

		}

		return left + width + right;

	}


public: // for iterators


	class bftIteratorKeys : public Iterator<TKEY>

	{

	public:

		bftIteratorKeys(nodeRBT<TKEY, TDATA>* Root2, nodeRBT<TKEY, TDATA>* NIL2)

		{

			if (Root2 == NIL2)throw out_of_range("tree is empty!");

			nil = NIL2;

			current = new nodeQ<TKEY, TDATA>;

			current->link = Root2;

			current->next = nullptr;

			tail = current;

		}

		TKEY next() override

		{

			if (current == nullptr || current->link == nil) throw
out_of_range("The next element does not exist");

			TKEY data = current->link->key;

			if (current->link->left != nil)

			{

				tail->next = new nodeQ<TKEY, TDATA>;
```

```
                tail = tail->next;
                tail->link = current->link->left;
                tail->next = nullptr;
            }
            if (current->link->right != nil)
            {
                tail->next = new nodeQ<TKEY, TDATA>;
                tail = tail->next;
                tail->link = current->link->right;
                tail->next = nullptr;
            }
            nodeQ<TKEY, TDATA>* next = current->next;
            delete current;
            current = next;
            return data;
        }
        bool has_next()override
        {
            return current != nullptr;
        }

private:
        template<typename T, typename U>
        struct nodeQ
        {
            nodeRBT<T, U>* link;
            nodeQ* next;
        };
        nodeQ<TKEY, TDATA>* current;
        nodeQ<TKEY, TDATA>* tail;
        nodeRBT<TKEY, TDATA>* nil;
};
```

```
	Iterator<TKEY>* createBftIteratorKey()
	{
		return new bftIteratorKeys(Root, NIL);
	}



	class bftIteratorData : public Iterator<TDATA>
	{
	public:
		bftIteratorData(nodeRBT<TKEY, TDATA>* Root2, nodeRBT<TKEY, TDATA>* NIL2)
		{
			if (Root2 == NIL2)throw out_of_range("tree is empty!");
			nil = NIL2;
			current = new nodeQ<TKEY, TDATA>;
			current->link = Root2;
			current->next = nullptr;
			tail = current;
		}
		TDATA next() override
		{
			if (current == nullptr || current->link == nil) throw
out_of_range("The next element does not exist");
			TDATA data = current->link->data;
			if (current->link->left != nil)
			{
				tail->next = new nodeQ<TKEY, TDATA>;
				tail = tail->next;
				tail->link = current->link->left;
				tail->next = nullptr;
			}
			if (current->link->right != nil)
			{
```

```
                        tail->next = new nodeQ<TKEY, TDATA>;
                        tail = tail->next;
                        tail->link = current->link->right;
                        tail->next = nullptr;
                }
                nodeQ<TKEY, TDATA>* next = current->next;
                delete current;
                current = next;
                return data;
        }
        bool has_next()override
        {
                return current != nullptr;
        }

private:
        template<typename T, typename U>
        struct nodeQ
        {
                nodeRBT<T, U>* link;
                nodeQ* next;
        };
        nodeQ<TKEY, TDATA>* current;
        nodeQ<TKEY, TDATA>* tail;
        nodeRBT<TKEY, TDATA>* nil;
};
Iterator<TDATA>* createBftIteratorData()
{
        return new bftIteratorData(Root, NIL);
}
};
template<typename TKEY, typename TDATA>
AArrey<TKEY, TDATA>::AArrey()
```

```
{
	NIL = new nodeRBT<TKEY, TDATA>;
	Root = NIL;
}


template<typename TKEY, typename TDATA>
AArrey<TKEY, TDATA>::~AArrey()
{
	if (Root != NIL) {
		clearLock(Root);
	}
	delete NIL;
}
```

main.cpp

```
#include "Head.h"
#include <iostream>

using namespace std;
int main()
{
	AArrey<string, double> myAA;

	cout << "test repos work!" << endl;
	myAA.insert("jan", 327.2);
	myAA.insert("feb", 368.2);
	myAA.insert("mar", 197.3);
	myAA.insert("apr", 178.4);
	myAA.insert("may", 100.0);
	myAA.insert("jun", 69.9);
	myAA.insert("jul", 32.3);
	myAA.insert("aug", 37.3);
	myAA.insert("sep", 19.0);
	myAA.insert("oct", 37.0);
	myAA.insert("nov", 73.2);
	myAA.insert("dec", 110.9);
	myAA.print();

	Iterator<string>* qitK = myAA.createBftIteratorKey();
	while (qitK->has_next())
	{
		cout << qitK->next() << " ";
	}
	myAA.getKeys();

	myAA.clear();
	try
	{
		Iterator<double>* qitD = myAA.createBftIteratorData();
```

```
		while (qitD->has_next())
		{
			cout << qitD->next() << " ";
		}
	}
	catch (out_of_range error)
	{
		cout << error.what();
	}

	myAA.getValue();

	return 0;
}
```

UnitTest1.cpp

```
#include "pch.h"
#include "CppUnitTest.h"
#include "main.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest
{
	TEST_CLASS(UnitTest)
	{
	public:

		TEST_METHOD(TestInsertRoot)
		{
			AArrey<double, int> myMap;
			myMap.insert(1.213, 34);
			Iterator<double>* qitK = myMap.createBftIteratorKey();
			Assert::AreEqual(1.213, qitK->next());
		}

		TEST_METHOD(TestInsertError)
		{
			AArrey<double, int> myMap;
			myMap.insert(1.213, 34);
			try
			{
				myMap.insert(1.213, 34);
			}
			catch (invalid_argument error)
			{
				Assert::AreEqual("Error - This key is already in memory!",
error.what());
			}
		}

		TEST_METHOD(TestInsertFixupCase3)
		{
			AArrey<float, int> myMap;
			myMap.insert(1.213, 34);
			myMap.insert(2.23, 33);
			//case 3
			myMap.insert(4.3, 32);
```

```
}

TEST_METHOD(TestInsertFixupCase1)
{
	AArrey<float, int> myMap;
	myMap.insert(2.213, 34);
	myMap.insert(1.23, 33);
	myMap.insert(3.3, 32);
	//case 1
	myMap.insert(4.1, 23);
}

TEST_METHOD(TestInsertFixupCase2)
{
	AArrey<float, int> myMap;
	myMap.insert(11, 34);
	myMap.insert(14, 33);
	myMap.insert(2, 32);
	myMap.insert(15, 23);
	myMap.insert(7, 21);
	myMap.insert(1, 32);
	myMap.insert(5, 23);
	myMap.insert(8, 21);
	//case 1 -> |case 2 |->case 3
	myMap.insert(4, 123);
}

TEST_METHOD(TestRemoveRed)
{
	AArrey<float, int> myMap;
	myMap.insert(11, 34);
	myMap.insert(14, 33);
	myMap.insert(2, 32);
	myMap.insert(15, 23);
	myMap.insert(7, 21);
	myMap.insert(1, 32);
	myMap.insert(5, 23);
	myMap.insert(8, 21);
	myMap.insert(4, 123);
	myMap.remove(2);
}

TEST_METHOD(TestRemoveBlack)
{
	AArrey<float, int> myMap;
	myMap.insert(11, 34);
	myMap.insert(14, 33);
	myMap.insert(2, 32);
	myMap.insert(15, 23);
	myMap.insert(7, 21);
	myMap.insert(1, 32);
	myMap.insert(5, 23);
	myMap.insert(8, 21);
	myMap.insert(4, 123);
	myMap.remove(14);
}

TEST_METHOD(TestIteratorFalse)
{
	AArrey<float, int> myMap;
	try
```

```
                {
                        Iterator<float>* qit = myMap.createBftIteratorKey();
                }
                catch (out_of_range error)
                {
                        Assert::AreEqual("tree is empty!", error.what());
                }


        }


    };
}
```