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. 8309 _____ Иванов Д.К.

Преподаватель _____ Тутуева А.В

Санкт-Петербург
2020

Исходная формулировка задания:

Реализовать шаблонный ассоциативный массив (map) на основе красно-черного дерева. Наличие unit-тестов ко всем реализуемым методам является обязательным требованием.

Цель работы:

Научиться реализации ассоциативного массива на основе красно-черного дерева.

Постановка задачи:

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 того или иного метода.

Организация данных:

Название	Описание работы метода	Оценка временной сложности
void print()	Вывод дерева при помощи псевдографики в консоль	O (N+K)
bool contains(const TKEY key)	Проверка: существует ли в дереве нужный ключ.	O (logN)
void insert(const TKEY key, const TDATA data)	Вставка в красно-черное дерево, как в бинарное дерево, после вызываем метод insertFixup()	O (1) – вставка корня O (logN*N)
void remove(const TKEY key)	Производим удаление узла в красно-черном дереве, как и в бинарном, если узел был черным вызываем метод removeFixup	O (1) – в случае если удаляем красный узел O (N) – в случае удаления черного узла
void clear()	Удаление красно-черного дерева	O(N)
void getKeys()	Вывод списка ключей.	O(N)
void getValue()	Вывод списка данных.	O(N)
void clearLock(nodeRBT<TKEY,TDATA> *&head)	Вывод размера списка	O(N)

void inrDatas(nodeRBT<TKEY, TDATA>* current)	Центрированный обход данных для списка данных.	O(N)

void inrKeys(nodeRBT<TKEY, TDATA>* current)	Центрированный обход данных для списка ключей.	O(N)
void nwInsert(nodeRBT<TKEY, TDATA>*& current, nodeRBT<TKEY, TDATA>*& nwNode()	Реверсивная вставка узла в красно-черное дерево	O(logN)
void insertFixup(nodeRBT<TKEY ,TDATA>*& nwNode)	Редактируем КР дерево после вставки нового узла по трем пунктам: 1. Если дядя красный, то перекрашиваем его, родителя и деда. 2. Если дядя черный производим поворот ветвей налево или направо (в зависимости от того является родитель левым или правым) 3. Также входит во второй случай, но перекрашивается родитель и дед. Происходит поворот налево или направо.	O (N)
void leftRotate(nodeRBT<TKEY, TDATA>* current)	Поворот налево	O (1)
void rightRotate(nodeRBT<TKEY , TDATA>* current)	Поворот направо	O (1)
nodeRBT<TKEY, TDATA>* dlFind(nodeRBT<TKEY, TDATA>*current,TKEY key)	Рекурсивный поиск узла для удаления	O (logN)
nodeRBT<TKEY, TDATA>* treeMinimum(nodeRBT<TKEY , TDATA>* current)	Проход до конца по левым ветвям дерева.	O (N)
void transplant(nodeRBT<TKEY, TDATA>*& current, nodeRBT<TKEY, TDATA>*& additional)	Замена одного поддерева, являющегося дочерним по отношению к своему родителю, другим поддеревом.	O (N)

void removeFixup(nodeRBT<TKEY, TDATA>*& current)	Редактируем КР дерево после удаления черного узла по 4 пунктам: 1. Если брат перекрашиваем его в черный, а родителя в красный. Производим поворот налево или направо (в зависимости от того каким потомком является наш узел). 2. Если у брата два потомка черные перекрашиваем брата в красный 3. Если у брата один потомок черный делаем второго потомка тоже черным и	O(N)
	брата красим в красный и производим поворот 4. Брату присваиваем цвет родителя, а его красим в черный вместе с правым или левым потомком брата, далее делаем поворот.	
bool find(nodeRBT<TKEY, TDATA>* current, TKEY key)	Рекурсивный поиск элемента для метода contains.	O (logN)
int print_in_massive(nodeRBT <TKEY, TDATA>* tree, int is_left, int offset, int depth, char s[30][255])	Создание дерева с помощью массива char.	O (N+K)
class bftIteratorKeys : public Iterator<TKEY>	Итератор для доступа к ключам в ассоциативном массиве	-
class bftIteratorData : public Iterator<TDATA>	Итератор для доступа к данным в ассоциативном массиве	-

Название Unit-теста	Описание работы
TEST_METHOD(TestInsertRoot)	Проверяем вставку корня дерева

TEST_METHOD(TestInsertError)	Проверяем ошибку при вводе уже существующего ключа
TEST_METHOD(TestInsertFixupCase1)	Проверка редактирования после вставки 1 случай.
TEST_METHOD(TestInsertFixupCase2)	Проверка редактирования после вставки 2 случай.
TEST_METHOD(TestInsertFixupCase3)	Проверка редактирования после вставки 3 случай.
TEST_METHOD(TestRemoveRed)	Проверка удаления красного узла дерева.
TEST_METHOD(TestRemoveBlack)	Проверка удаления черного узла дерева
TEST_METHOD(TestIteratorFalse)	Проверка вывода ошибки при создании итератора для пустого Ассоциативного массива

Код программы

```
#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();


	//method which better don't touch      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);
                }
```

```
//Adding a new item to a red-black tree (No Root) 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);
                }
                }
        }


        //We fix broken rules when inserting a new item      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;
                }

//Just  left  Rotete  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;
        }

    //just right Rotate       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;
}


        ı for an item in the tree nodeRBT<TKEY, TDATA>*
{       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;
}
```