**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. 9309 |  | Серов А.В. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2021

Оглавление

[1 Постановка задачи 3](#_Toc74103469)

[2 Цель работы 3](#_Toc74103470)

[3 Описание классов и методов, оценка временной сложности 3](#_Toc74103471)

[Описание классов 3](#_Toc74103472)

[Описание методов 3](#_Toc74103473)

[4 Текст программы 5](#_Toc74103474)

[5 Юнит-тестирование 16](#_Toc74103475)

[6 Пример 21](#_Toc74103476)

[7 Вывод 21](#_Toc74103477)

# Постановка задачи

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

**Список методов:**

insert(ключ, значение) // добавление элемента с ключом и значением

remove(ключ) // удаление элемента дерева по ключу

find(ключ) // поиск элемента по ключу

clear // очищение ассоциативного массива

get\_keys // возвращает список ключей

get\_values // возвращает список значений

print // вывод в консоль

# Цель работы

Реализация и изучение шаблонного ассоциативного массива (map) на основе красно-черного дерева.

# Описание классов и методов, оценка временной сложности

## Описание классов

1. **RBTree** – класс красно-черного дерева содержащий поле root, которое хранит в себе указатель на корень дерева, так же класс содержит подклассы: Node, dftIterator.
2. **Node -** представляет из себя узел дерева, который содержит указатели на двух потомков и родителя, а также на цвет, ключ и значение. Второй подкласс является итератором для обхода дерева в глубину из предыдущих лабораторных работ, но с урезанным функционалом: поскольку итератор используется только внутри класса, он не имеет внешнего интерфейсного класса и метода has\_next. Также dftIterator содержит подкласс Stack с шаблонными методами push и pop.

## Описание методов

1. **Конструктор Node** - заполняет узел дерева начальными значениями, позволяет установить цвет узла, значение и ключ. Временная сложность: O(1).
2. **Конструктор Stack** - заполняет указатель на головной элемент стэка значением nullptr.
3. **Метод push** – метод подкласса Stack, который проталкивает элемент в стэк.
4. **Метод pop** – метод Stack, который извлекает элемент из стэка с возвращением значения иформационного поля структуры элемента стэка.
5. **Деструктор Stack** – с помощью метода pop извлекает все элементы из стэка.
6. **Конструктор dftIterator** – данный конструктор выделяет память под стэк, а также устанавливает указатель cur на корень дерева.
7. **Метод next** – осуществляет перемещение по дереву, возвращая указатель на пройденный элемент. Если следующего элемента нет, то указатель cur принимает значение nullptr. Если указатель на момент запуска метода уже был в этом значении, то метод вернет nullptr.
8. **Деструктор dftIterator** - освобождает выделенную под стэк память.
9. **Конструктор RBTree** устанавливает значение указателя на корень как nullptr.
10. **Метод** **insert -** включает новый элемент в дерево. Если дерево пустое, то алгоритм просто устанавливает новый элемент как корень, в противном случае происходят вызовы алгоритмов insert\_rec и fix\_insertion.
11. **Методы rotate\_l** и **rotate\_r** - осуществляют левый и правый повороты соответственно. Входной элемент до исполнения алгоритма является нижним в паре поворачиваемых элементов.
12. **Метод insert\_rec –** в данном методе выполняется поиск места для нового элемента в дереве. Если в дереве уже есть элемент с ключом, совпадающим с ключом включаемого элемента, то будет выброшено исключение. Включаемый элемент всегда красный. Возвращает указатель на включенный элемент.
13. **Метод fix\_insertion –** делает баланс дерева при включении в него нового элемента. Рекурсивен, т.к. должен обрабатывать не только включаемый элемент, но и элементы по всей высоте дерева, если баланс текущего элемента нарушает свойства дерева выше.

**Возможные случаи:**

1. Обрабатываемый элемент – новый корень (в этом случае перекрашивает элемент в черный и завершает работу);
2. Родитель обрабатываемого элемента – черный (завершает работу);
3. Дядя элемента – красный (в этом случае родитель и дядя элемента перекрашиваются в черный, а дедушка – в красный, затем алгоритм балансировки применяется к дедушке элемента);
4. Элемент является правым потомком, а его родитель – левым / элемент является левым потомком, а родитель – правым (в этом случае дедушка перекрашивается в красный, обрабатываемый элемент – в черный, выполняется лево-правый/право-левый поворот и алгоритм завершает работу).
5. Элемент и его родитель являются правыми/левыми потомками (родитель перекрашивается в черный, а дедушка – в красный, выполняется левый/правый поворот от родителя, алгоритм завершает работу).
6. **Метод getNode** – выполняет рекурсивный поиск элемента по ключу, возвращая указатель на элемент. Если элемента не существует, то возвращает nullptr.
7. **Метод getMin** – выполняет рекурсивный поиск минимального (самого левого) элемента в поддереве, возвращает указатель на элемент.
8. **Метод remove** - если дерево пустое, то кидает исключение. Иначе вызывает getNode для поиска удаляемого элемента. Если элемент не найден – метает исключение. Если у найденного элемента есть оба потомка, вызывает getMin и копирует значения информационного наполнения, найденного в getMin элемента в удаляемый элемент, а getMin устанавливает как удаляемый. После вызывает метод
9. **Метод deleting**

**Обрабатываемые случаи:**

1. Если элемент красный, то элемент просто удаляется.
2. Если у элемента есть один потомок, то элемент удаляется, а на его место встает потомок, которые перекрасится в черный цвет.
3. Если элемент не имеет потомков и стоит в корне, то элемент просто удалится и дерево станет пустым.(если ни один из случаев не подошел, то вызывается метод fix\_deleting, который обрабатывает оставшиеся случаи, после чего удаляемый элемент исключается из дерева. Метод fix\_deleting. Рекурсивно обрабатывает все случаи, когда удаляемый элемент не имеет потомков (нумерация продолжена))
4. Обрабатываемый элемент в корне дерева (эта проверка протекает только при рекурсивном вызове fix\_deleting с соответствующим флагом и нужна для некоторых случаев рекурсивной обработки элемента, уже не являющегося удаляемым);
5. Родитель обрабатываемого элемента – черный, а брат – красный (тогда родитель перекрашивается в красный, брат – в черный, затем выполняется левый/правый поворот для брата, в зависимости от того, является ли он левым/правым потомком, после чего алгоритм вызывается заново для того же элемента);
6. Родитель черный, потомки брата элемента – тоже черные (тогда брат элемента перекрашивается в красный, а алгоритм вызывается уже для родителя элемента, заново проверяя все случаи);
7. Родитель красный, брат и его потомки – черные (родитель перекрашивается в черный, брат – в красный, после чего алгоритм завершает работу).
8. Элемент является левым/правым потомком, правый/левый потомок брата – черный (выполняется правый/левый поворот от левого/правого потомка брата, этот потомок перекрашивается в черный, а брат – в красный, после чего алгоритм запускается заново для того же элемента);
9. Элемент является левым/правым потомком, правый/левый потомок брата – красный (выполняется левый/правый поворот от брата, брат перекрашивается в цвет своего бывшего родителя, родитель перекрашивается в черный, правый/левый потомок брата – тоже в черный, после чего алгоритм завершает свою работу);
10. **Метод find -** если дерево пустое, то метод исключения. В противном случае вызывает getNode. Если возвращаемое значение – nullptr, то так же метод исключение. В противном случае возвращает значение найденного элемента.
11. **Методы get\_keys** и **get\_values** - возвращают строку, заполненную списком ключей/значений элементов. Для формирования строки используют dftIterator. Кидают исключение, если дерево пустое.
12. **Метод print** - выводит на экран в виде списка ключей и значений дерево, посредством использования dftIterator. Кидает исключение, если дерево пустое.
13. **Метод clear** - вызывает remove для корня дерева до тех пор, пока дерево не опустеет.
14. **Деструктор RBTree** - вызывает метод clear.

# Текст программы

// 1 programm Serov 9309

#ifndef RB\_TREE\_H

#define RB\_TREE\_H

#include <iostream>

#include <string>

#include <stdexcept>

using namespace std;

enum Color

{

BLACK,

RED

};

class RBTree

{

class Node

{

public:

Node(bool IsRed, int Key, char Value)

{

if (IsRed)

color = RED;

else

color = BLACK;

key = Key;

value = Value;

parent = nullptr;

right = nullptr;

left = nullptr;

}

Color color;

int key;

char value;

Node\* parent;

Node\* right;

Node\* left;

};

Node\* root;

class dftIterator

{

class Stack

{

struct Elem

{

Node\* inf;

Elem\* next;

};

Elem\* head;

public:

Stack() { head = nullptr; }

void push(Node\* In)

{

Elem\* Insert = new Elem;

Insert->inf = In;

Insert->next = head;

head = Insert;

}

Node\* pop()

{

if (!head)

return nullptr;

Node\* ans = head->inf;

Elem\* Del = head;

head = head->next;

delete Del;

return ans;

}

~Stack()

{

while (head)

pop();

}

};

Node\* cur;

Stack\* s;

public:

dftIterator(Node\* Root)

{

cur = Root;

s = new Stack;

}

Node\* next()

{

if (!cur)

return nullptr;

if (cur->right)

s->push(cur->right);

Node\* ans = cur;

if (cur->left)

cur = cur->left;

else

cur = s->pop();

return ans;

}

~dftIterator() { delete s; }

};

void rotate\_l(Node\* pivot)

{

Node\* P = pivot->parent;

Node\* GP = P->parent;

Node\* leftC = pivot->left;

pivot->parent = GP;

if (GP)

{

if (GP->left == P)

GP->left = pivot;

else

GP->right = pivot;

}

else

root = pivot;

P->parent = pivot;

pivot->left = P;

if (leftC)

leftC->parent = P;

P->right = leftC;

}

void rotate\_r(Node\* pivot)

{

Node\* P = pivot->parent;

Node\* GP = P->parent;

Node\* rightC = pivot->right;

pivot->parent = GP;

if (GP)

{

if (GP->left == P)

GP->left = pivot;

else

GP->right = pivot;

}

else

root = pivot;

P->parent = pivot;

pivot->right = P;

if (rightC)

rightC->parent = P;

P->left = rightC;

}

void fix\_insertion(Node\* newNode)

{

Node\* P = newNode->parent;

if (!P)

{

newNode->color = BLACK;

root = newNode;

return;

}

if (P->color == BLACK)

return;

Node\* GP = P->parent;

Node\* U;

if (GP->right == P)

U = GP->left;

else

U = GP->right;

Color U\_col;

if (!U)

U\_col = BLACK;

else

U\_col = U->color;

if (U\_col == RED)

{

GP->color = RED;

P->color = BLACK;

U->color = BLACK;

fix\_insertion(GP);

}

else

{

if (P->right == newNode)

{

if (GP->left == P)

{

rotate\_l(newNode);

rotate\_r(newNode);

newNode->color = BLACK;

GP->color = RED;

}

else

{

rotate\_l(P);

P->color = BLACK;

GP->color = RED;

}

}

else

{

if (GP->right == P)

{

rotate\_r(newNode);

rotate\_l(newNode);

newNode->color = BLACK;

GP->color = RED;

}

else

{

rotate\_r(P);

P->color = BLACK;

GP->color = RED;

}

}

}

}

void fix\_deleting(Node\* pivot, bool mode)

{

Node\* P = pivot->parent;

if (mode && !P)

{

root = pivot;

return;

}

Node\* S;

if (P->left == pivot)

S = P->right;

else

S = P->left;

Node\* Sl = S->left;

Node\* Sr = S->right;

Color Sl\_col, Sr\_col;

if (!Sl)

Sl\_col = BLACK;

else

(Sl\_col = Sl->color);

if (!Sr)

Sr\_col = BLACK;

else

(Sr\_col = Sr->color);

if (S->color == RED)

{

if (P->left == pivot)

rotate\_l(S);

else

rotate\_r(S);

S->color = BLACK;

P->color = RED;

fix\_deleting(pivot, 0);

}

else if (P->color == BLACK && Sr\_col == BLACK && Sl\_col == BLACK)

{

S->color = RED;

fix\_deleting(P, 1);

}

else if (P->color == RED && Sr\_col == BLACK && Sl\_col == BLACK)

{

P->color = BLACK;

S->color = RED;

}

else if (P->left == pivot && Sr\_col == BLACK)

{

rotate\_r(Sl);

Sl->color = BLACK;

S->color = RED;

fix\_deleting(pivot, 0);

}

else if (P->left == pivot && Sr\_col == RED)

{

rotate\_l(S);

S->color = P->color;

P->color = BLACK;

Sr->color = BLACK;

}

else if (P->right == pivot && Sl\_col == BLACK)

{

rotate\_l(Sr);

Sr->color = BLACK;

S->color = RED;

fix\_deleting(pivot, 0);

}

else if (P->right == pivot && Sl\_col == RED)

{

rotate\_r(S);

S->color = P->color;

P->color = BLACK;

Sl->color = BLACK;

}

}

void deleting(Node\* delNode)

{

if (delNode->color == RED)

{

if (delNode->parent->right == delNode)

delNode->parent->right = nullptr;

else

delNode->parent->left = nullptr;

delete delNode;

}

else if (delNode->left || delNode->right)

{

Node\* child;

if (delNode->left)

child = delNode->left;

else

child = delNode->right;

child->parent = delNode->parent;

child->color = BLACK;

if (!child->parent)

root = child;

else

{

if (delNode->parent->right == delNode)

delNode->parent->right = child;

else

delNode->parent->left = child;

}

delete delNode;

}

else if (delNode == root)

{

root = nullptr;

delete delNode;

}

else

{

fix\_deleting(delNode, 0);

if (delNode->parent->left == delNode)

delNode->parent->left = nullptr;

else

delNode->parent->right = nullptr;

delete delNode;

}

}

Node\* getMin(Node\* cur)

{

while (cur->left)

cur = cur->left;

return cur;

}

Node\* getNode(int k, Node\* cur)

{

if (!cur)

return nullptr;

if (cur->key == k)

return cur;

if (cur->key < k)

return getNode(k, cur->right);

if (cur->key > k)

return getNode(k, cur->left);

}

Node\* insert\_rec(int IncludeKey, char IncludeValue, Node\* Parent)

{

if (IncludeKey == Parent->key)

throw invalid\_argument("");

if (IncludeKey > Parent->key)

{

if (!Parent->right)

{

Parent->right = new Node(1, IncludeKey, IncludeValue);

Parent->right->parent = Parent;

return Parent->right;

}

else

return insert\_rec(IncludeKey, IncludeValue, Parent->right);

}

else

{

if (!Parent->left)

{

Parent->left = new Node(1, IncludeKey, IncludeValue);

Parent->left->parent = Parent;

return Parent->left;

}

else

return insert\_rec(IncludeKey, IncludeValue, Parent->left);

}

}

public:

RBTree() { root = nullptr; }

void insert(int Key, char Value)

{

if (!root)

root = new Node(0, Key, Value);

else

{

Node\* newNode;

try { newNode = insert\_rec(Key, Value, root); }

catch (exception& exception) { throw invalid\_argument("Element with such key has been already included\n"); }

fix\_insertion(newNode);

}

}

void remove(int Key)

{

if (!root)

throw runtime\_error("Tree is empty\n");

Node\* delNode = getNode(Key, root);

if (!delNode)

throw runtime\_error("Node isn't exist\n");

if (delNode->left && delNode->right)

{

Node\* temp = delNode;

delNode = getMin(delNode->right);

temp->key = delNode->key;

temp->value = delNode->value;

}

deleting(delNode);

}

char find(int Key)

{

if (!root)

throw runtime\_error("Tree is empty\n");

Node\* ans = getNode(Key, root);

if (!ans)

throw runtime\_error("Node isn't exist\n");

return ans->value;

}

void clear()

{

while (root)

remove(root->key);

}

string get\_keys()

{

if (!root)

throw runtime\_error("Tree is empty\n");

string ans;

dftIterator\* Iterator = new dftIterator(root);

Node\* Cur = Iterator->next();

while (Cur)

{

ans += to\_string(Cur->key);

ans += ' ';

Cur = Iterator->next();

}

delete Iterator;

return ans;

}

string get\_values()

{

if (!root)

throw runtime\_error("Tree is empty\n");

string ans;

dftIterator\* Iterator = new dftIterator(root);

Node\* Cur = Iterator->next();

while (Cur)

{

ans += Cur->value;

ans += ' ';

Cur = Iterator->next();

}

delete Iterator;

return ans;

}

void print()

{

if (!root)

throw runtime\_error("Tree is empty\n");

dftIterator\* Iterator = new dftIterator(root);

Node\* Cur = Iterator->next();

while (Cur)

{

cout << Cur->key << ' ' << Cur->value << endl;

Cur = Iterator->next();

}

delete Iterator;

}

~RBTree()

{

clear();

}

};

#endif

# Юнит-тестирование

TEST\_METHOD(insert)

{

string expected;

RBTree T;

T.insert(40, 'G');

expected = "G ";

Assert::AreEqual(expected, T.get\_values());

T.insert(50, 'F');

expected = "G F ";

Assert::AreEqual(expected, T.get\_values());

T.insert(70, 'H');

expected = "F G H ";

Assert::AreEqual(expected, T.get\_values());

T.insert(20, 'B');

expected = "F G B H ";

Assert::AreEqual(expected, T.get\_values());

T.insert(30, 'A');

expected = "F A B G H ";

Assert::AreEqual(expected, T.get\_values());

T.insert(10, 'C');

expected = "F A B C G H ";

Assert::AreEqual(expected, T.get\_values());

T.insert(25, 'E');

expected = "F A B C E G H ";

Assert::AreEqual(expected, T.get\_values());

T.insert(5, 'D');

expected = "A B C D E F G H ";

Assert::AreEqual(expected, T.get\_values());

}

TEST\_METHOD(insert\_exception)

{

RBTree T;

bool b = 0;

T.insert(1, 'A');

try

{

T.insert(1, 'B');

Assert::IsTrue(0);

}

catch (exception& invalid\_argument)

{

b = 1;

}

Assert::IsTrue(b);

}

TEST\_METHOD(remove)

{

string expected;

RBTree T;

T.insert(40, 'G');

T.insert(50, 'F');

T.insert(70, 'H');

T.insert(20, 'B');

T.insert(30, 'A');

T.insert(10, 'C');

T.insert(25, 'E');

T.insert(5, 'D');

T.remove(5);

expected = "A B C E F G H ";

Assert::AreEqual(expected, T.get\_values());

T.insert(5, 'D');

T.remove(10);

expected = "A B D E F G H ";

Assert::AreEqual(expected, T.get\_values());

T.remove(5);

expected = "A B E F G H ";

Assert::AreEqual(expected, T.get\_values());

T.remove(25);

T.remove(20);

expected = "F A G H ";

Assert::AreEqual(expected, T.get\_values());

T.remove(40);

T.remove(30);

expected = "F H ";

Assert::AreEqual(expected, T.get\_values());

}

TEST\_METHOD(remove\_exception1)

{

RBTree T;

bool b = 0;

try

{

T.remove(1);

Assert::IsTrue(0);

}

catch (exception& runtime\_error)

{

b = 1;

}

Assert::IsTrue(b);

}

TEST\_METHOD(remove\_exception2)

{

RBTree T;

bool b = 0;

T.insert(1, 'A');

try

{

T.remove(2);

Assert::IsTrue(0);

}

catch (exception& runtime\_error)

{

b = 1;

}

Assert::IsTrue(b);

}

TEST\_METHOD(find)

{

RBTree T;

char expected = 'H';

T.insert(40, 'G');

T.insert(50, 'F');

T.insert(70, 'H');

Assert::AreEqual(expected, T.find(70));

}

TEST\_METHOD(find\_exception1)

{

RBTree T;

bool b = 0;

try

{

T.find(1);

Assert::IsTrue(0);

}

catch (exception& runtime\_error)

{

b = 1;

}

Assert::IsTrue(b);

}

TEST\_METHOD(find\_exception2)

{

RBTree T;

bool b = 0;

T.insert(1, 'A');

try

{

T.find(0);

Assert::IsTrue(0);

}

catch (exception& runtime\_error)

{

b = 1;

}

Assert::IsTrue(b);

}

TEST\_METHOD(get\_keys\_exception)

{

RBTree T;

bool b = 0;

try

{

T.get\_keys();

Assert::IsTrue(0);

}

catch (exception& runtime\_error)

{

b = 1;

}

Assert::IsTrue(b);

}

TEST\_METHOD(get\_values\_exception)

{

RBTree T;

bool b = 0;

try

{

T.get\_values();

Assert::IsTrue(0);

}

catch (exception& runtime\_error)

{

b = 1;

}

Assert::IsTrue(b);

}

TEST\_METHOD(print\_exception)

{

RBTree T;

bool b = 0;

try

{

T.print();

Assert::IsTrue(0);

}

catch (exception& runtime\_error)

{

b = 1;

}

Assert::IsTrue(b);

}

# Пример

**Код:**

#include "rb\_tree.h"

int main()

{

cout << "TEST\n\n";

RBTree T;

T.insert(4, 'G');

T.insert(1, 'F');

T.insert(7, 'H');

T.insert(2, 'B');

T.insert(3, 'A');

T.insert(10, 'C');

T.insert(20, 'E');

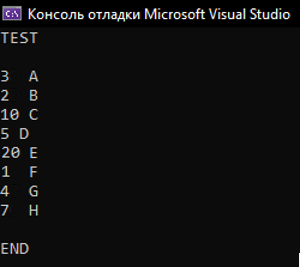
T.insert(5, 'D');

T.print();

cout << "\nEND\n";

}

**Результаты:**



# Вывод

При выполнении данной лабораторной работы были на практике применены знания, полученные при обучении, а именно: алгоритмы балансировки двоичных деревьев, которые увеличивают скорость работы алгоритмов различных классов.