**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра CАПР**

отчет

**по лабораторной работе №1**

**по дисциплине «Алгоритмы и структуры данных»**

**Вариант 1**

**Тема: Ассоциативный массив на основе красно-черного дерева**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Квитко Д.В. |
| Преподаватель |  | Тутуева А.В |

Санкт-Петербург

2021

## Постановка задачи

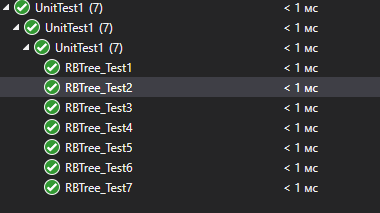
Реализовать шаблонный ассоциативный массив (map) на основе красно-черного дерева.

## Описание и оценка временной сложности методов

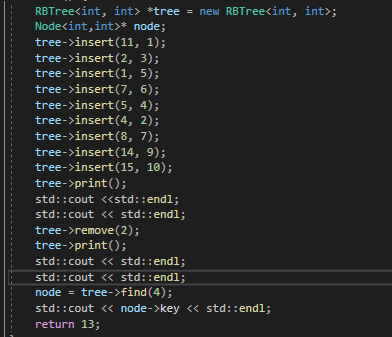
|  |  |  |
| --- | --- | --- |
| Название метода | Описание | Временная сложность |
| void left\_rotate(Node<Key, Value>\* curret) | Левый поворот | O(lg(n)) |
| void right\_rotate(Node<Key, Value>\* curret); | Правый поворот | O(lg(n)) |
| void insert(Key key, Value value); | Вставка элемента | O(lg(n)) |
| void Insert\_fixup(Node<Key, Value>\* newNode); | Восстановление свойств после вставки | O(lg(n)) |
| void remove(Key key); | Удаление элемента | O(h) |
| void Remove\_fixup(Node<Key, Value>\* x); | Восстановление свойств после удаления | O(h) |
| Node<Key, Value>\* find(Key key) | Поиск элемента по ключу | O(lg(n)) |

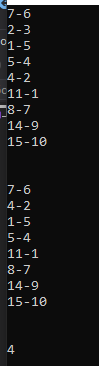
## Описание реализованный unit-тестов

|  |  |
| --- | --- |
| Название теста |  |
| RBTree\_Test1 | Проверяет вставку элементов в дерево |
| RBTree\_Test2 | Проверяет удаление элемента в дереве |
| RBTree\_Test3 | Проверяет поиск элемента (элемент есть в дереве) |
| RBTree\_Test4 | Проверяет поиск элемента (элемента нет в дереве) |
| RBTree\_Test5 | Проверяет на вызов исключение, при попытке вывести список ключей пустого дерева |
| RBTree\_Test6 | Проверяет на вызов исключение, при попытке вывести список значений пустого дерева |
| RBTree\_Test7 | Проверка метода очистки дерева |



## Пример работы





## Листинг

**BRTree.h**

#ifndef BRTree\_H

#define BRTree\_H

#include"NodeBRTree.h"

#include"stack.h"

#include"List.h"

template <typename Key, typename Value>

class RBTree

{

private:

Node<Key, Value>\* nil = new Node<Key, Value>();

void Insert\_fixup(Node<Key, Value>\* newNode);

void Remove\_fixup(Node<Key, Value>\* x);

void left\_rotate(Node<Key, Value>\* curret);

void right\_rotate(Node<Key, Value>\* curret);

public:

RBTree();

~RBTree();

Node<Key, Value>\* Root;

void insert(Key key, Value value); // добавление элемента с ключом и значением

void remove(Key key); // удаление элемента дерева по ключу

Node<Key, Value>\* find(Key key)// поиск элемента по ключу

{

if (Root == nullptr) {

throw "No such element exists";

}

else {

Node<Key, Value>\* iter;

iter = Root;

for (;;) {

if (key <= iter->key) {

if (iter->key == key) {

return iter;

}

if (iter->Lnext == nil) {

throw "No such element exists";

}

iter = iter->Lnext;

}

else {

if (iter->key == key) {

return iter;

}

if (iter->Rnext == nil) {

throw "No such element exists";

}

iter = iter->Rnext;

}

}

}

}

void transplant(Node<Key, Value>\* U, Node<Key, Value>\* V);

void clear(); // очищение ассоциативного массива

void preorder(Node<Key, Value>\* node);

List<Key>\* get\_keys(); // возвращает список ключей

List<Value>\* get\_values(); // возвращает список значений

Node<Key, Value>\* get\_nil();

void print(); // вывод в консоль

};

#endif

**functionList.h**

#include"List.h"

#include"NodeBRTree.h"

template <typename Data>

List<Data>::List()

{

head = nullptr;

tail = nullptr;

size = 0;

}

template <typename Data>

List<Data>::~List()

{

clear();

}

template <typename Data>

void List<Data>::reset\_list()

{

tail = nullptr;

head = nullptr;

}

template <typename Data>

unsigned int List<Data>::get\_size()

{

return size;

}

template <typename Data>

void List<Data>::push\_back(Data date)

{

if (size == 0) {

head = new ListNode(date);

tail = head;

}

else {

tail->next = new ListNode(date);

tail = tail->next;

}

size++;

}

template <typename Data>

void List<Data>::push\_front(Data date)

{

if (size == 0) {

head = new ListNode(date);

tail = head;

}

else {

head = new ListNode(date, head);

}

size++;

}

template <typename Data>

void List<Data>::pop\_back() {

if (size == 0) return;

if (size == 1) {

delete head;

reset\_list();

}

else {

Node\* current = head;

while (current->next != tail) {

current = current->next;

}

current->next = nullptr;

delete tail;

tail = current;

}

size--;

}

template <typename Data>

void List<Data>::pop\_front() {

if (size == 0) {

return;

}

if (size == 1) {

delete head;

reset\_list();

}

else {

ListNode\* current = head;

head = head->next;

delete current;

}

size--;

}

template <typename Data>

void List<Data>::print\_to\_console() {

if (size == 0) {

return;

}

else {

unsigned int index = get\_size();

ListNode\* current = head;

while (index != 0) {

std::cout << current->value << " ";

current = current->next;

index--;

}

std::cout << std::endl;

}

}

template <typename Data>

void List<Data>::clear()

{

while (size != 0)

{

pop\_front();

}

}

template <typename Data>

bool List<Data>::isEmpty() {

if (size != 0) {

return 0;

}

return 1;

}

template <typename Data>

Data List<Data>::at(unsigned int index)

{

if (index >= size) {

throw std::out\_of\_range("Index is greater than list size");

}

else {

ListNode\* current = head;

unsigned int counter = 0;

while (counter != index) {

current = current->next;

counter++;

}

return current->data;

}

}

**functionBRTree.h**

#include"List.h"

#include"NodeBRTree.h"

#include"stack.h"

#include <stdexcept>

#include<iostream>

template <typename Key, typename Value>

RBTree<Key, Value>::RBTree()

{

Root = nil;

}

template <typename Key, typename Value>

RBTree<Key, Value>::~RBTree()

{

clear(Root);

nil= nullptr;

Root = nullptr;

}

template <typename Key, typename Value>

Node<Key,Value>\* RBTree<Key, Value>::get\_nil()

{

return nil;

}

template <typename Key, typename Value>

void RBTree<Key, Value>::preorder(Node<Key, Value>\* node) {

if (node == nullptr || node == nil) return;

preorder(node->Lnext);

preorder(node->Rnext);

delete node;

}

template <typename Key, typename Value>

void RBTree<Key, Value>::clear() {

preorder(Root);

delete nil;

nil = nullptr;

Root = nullptr;

}

template <typename Key, typename Value>

void RBTree<Key, Value>::left\_rotate(Node<Key, Value>\* x) {

Node<Key, Value>\* y;

y = x->Rnext;

x->Rnext = y->Lnext;

if (y->Lnext != nil) {

y->Lnext->parent = x;

}

y->parent = x->parent;

if (x->parent == nil) {

Root = y;

}

else if (x == x->parent->Lnext)

{

x->parent->Lnext = y;

}

else

{

x->parent->Rnext = y;

}

y->Lnext = x;

x->parent = y;

}

template <typename Key, typename Value>

void RBTree<Key, Value>::right\_rotate(Node<Key, Value>\* x) {

Node<Key, Value>\* y = new Node<Key, Value>;

y = x->Lnext;

x->Lnext = y->Rnext;

if (y->Rnext != nil) {

y->Rnext->parent = x;

}

y->parent = x->parent;

if (x->parent == nil) {

Root = y;

}

else if (x == x->parent->Rnext)

{

x->parent->Rnext = y;

}

else

{

x->parent->Lnext = y;

}

y->Rnext = x;

x->parent = y;

}

template <typename Key, typename Value>

void RBTree<Key, Value>::insert(Key key, Value value)

{

if (Root == nil) {

Node<Key, Value>\* kkk = new Node<Key, Value>(key, value, 'b', nil, nil, nil);

Root = kkk;

}

else {

Node<Key, Value>\* iter;

iter = Root;

for (;;) {

if (key <= iter->key) {

if (iter->Lnext == nil) {

iter->Lnext = new Node<Key, Value>(key, value, 'r', iter, nil, nil);

Insert\_fixup(iter->Lnext);

break;

}

iter = iter->Lnext;

}

else {

if (iter->Rnext == nil) {

iter->Rnext = new Node<Key, Value>(key, value, 'r', iter, nil, nil);

Insert\_fixup(iter->Rnext);

break;

}

iter = iter->Rnext;

}

}

}

}

template <typename Key, typename Value>

void RBTree<Key, Value>::Insert\_fixup(Node<Key, Value>\* newNode) {

while (newNode != Root && newNode->parent->colour == 'r')

{

if (newNode->parent == newNode->parent->parent->Lnext) {

Node<Key, Value>\* y;

y = newNode->parent->parent->Rnext;

if (y->colour == 'r') {

newNode->parent->colour = 'b';

y->colour = 'b';

newNode->parent->parent->colour = 'r';

newNode = newNode->parent->parent;

}

else {

if (newNode == newNode->parent->Rnext) {

newNode = newNode->parent;

left\_rotate(newNode);

}

newNode->parent->colour = 'b';

newNode->parent->parent->colour = 'r';

right\_rotate(newNode->parent->parent);

}

}

else {

Node<Key, Value>\* y;

y = newNode->parent->parent->Lnext;

if (y->colour == 'r') {

newNode->parent->colour = 'b';

y->colour = 'b';

newNode->parent->parent->colour = 'r';

newNode = newNode->parent->parent;

}

else {

if (newNode == newNode->parent->Lnext) {

newNode = newNode->parent;

right\_rotate(newNode);

}

newNode->parent->colour = 'b';

newNode->parent->parent->colour = 'r';

left\_rotate(newNode->parent->parent);

}

}

}

Root->colour = 'b';

}

template <typename Key, typename Value>

void RBTree<Key, Value>::Remove\_fixup(Node<Key, Value>\* x) {

while (x != Root && x->colour == 'b')

{

if (x == x->parent->Lnext) {

Node<Key, Value>\* w;

w = x->parent->Rnext;

if (w->colour == 'r') {

w->colour = 'b';

x->parent->colour = 'r';

left\_rotate(x->parent);

w = x->parent->Rnext;

}

if (w->Lnext->colour == 'b' && w->Rnext->colour == 'b') {

w->colour = 'r';

x = x->parent;

}

else

{

if (w->Rnext->colour == 'b') {

w->Lnext->colour = 'b';

w->colour = 'r';

right\_rotate(w);

w = x->parent->Rnext;

}

w->colour = x->parent->colour;

x->parent->colour = 'b';

w->Rnext->colour = 'b';

left\_rotate(x->parent);

x = Root;

}

}

else

{

Node<Key, Value>\* w;

w = x->parent->Lnext;

if (w->colour == 'r') {

w->colour = 'b';

x->parent->colour = 'r';

right\_rotate(x->parent);

w = x->parent->Lnext;

}

if (w->Rnext->colour == 'b' && w->Lnext->colour == 'b') {

w->colour = 'r';

x = x->parent;

}

else

{

if (w->Lnext->colour == 'b') {

w->Rnext->colour = 'b';

w->colour = 'r';

left\_rotate(w);

w = x->parent->Lnext;

}

w->colour = x->parent->colour;

x->parent->colour = 'b';

w->Lnext->colour = 'b';

right\_rotate(x->parent);

x = Root;

}

}

}

x->colour = 'b';

}

template <typename Key, typename Value>

void RBTree<Key, Value>::remove(Key key) {

Node<Key, Value>\* y;

Node<Key, Value>\* Z;

Node<Key, Value>\* X;

Z = find(key);

y = Z;

char y\_o\_colour = y->colour;

if (Z->Lnext == nil) {

X = Z->Rnext;

transplant(Z, Z->Rnext);

}

else if (Z->Rnext == nil)

{

X = Z->Lnext;

transplant(Z, Z->Lnext);

}

else

{

y = y->Rnext;

while (y->Lnext != nil)

{

y = y->Lnext;

}

y\_o\_colour = y->colour;

X = y->Rnext;

if (y->parent == Z) {

X->parent = y;

}

else

{

transplant(y, y->Rnext);

y->Rnext = Z->Rnext;

y->Rnext->parent = y;

}

transplant(Z, y);

y->Lnext = Z->Lnext;

y->Lnext->parent = y;

y->colour = Z->colour;

}

if (y\_o\_colour == 'b') {

Remove\_fixup(X);

}

}

template <typename Key, typename Value>

void RBTree<Key, Value>::transplant(Node<Key, Value>\* U, Node<Key, Value>\* V)

{

if (U->parent == nil) {

Root = V;

}

else if (U == U->parent->Lnext) {

U->parent->Lnext = V;

}

else

{

U->parent->Rnext = V;

}

V->parent = U->parent;

}

template <typename Key, typename Value>

List<Key>\* RBTree<Key, Value>::get\_keys()

{

if (Root == nil) {

throw("There is no element");

}

stack<Key, Value> stackKey;

List<Key>\* listKey = new List<Key>;

stackKey.push(Root);

bool flag = true;

Node<Key, Value>\* temp = stackKey.head->date;

while (!stackKey.isEmpty()) {

listKey->push\_back(temp->key);

if (temp->Rnext != nil) {

if (flag) {

stackKey.pop\_front();

flag = false;

}

stackKey.push(temp->Rnext);

}

if (temp->Lnext != nil) {

temp = temp->Lnext;

}

else

{

if (flag) {

stackKey.pop\_front();

}

if (!stackKey.isEmpty()) {

temp = stackKey.head->date;

}

flag = true;

}

}

return listKey;

}

template <typename Key, typename Value>

List<Value>\* RBTree<Key, Value>::get\_values()

{

if (Root == nil) {

throw("There is no element");

}

stack<Key, Value> stackValue;

List<Key>\* listValue = new List<Key>;

stackValue.push(Root);

bool flag = true;

Node<Key, Value>\* temp = stackValue.head->date;

while (!stackValue.isEmpty()) {

listValue->push\_back(temp->value);

if (temp->Rnext != nil) {

if (flag) {

stackValue.pop\_front();

flag = false;

}

stackValue.push(temp->Rnext);

}

if (temp->Lnext != nil) {

temp = temp->Lnext;

}

else

{

if (flag) {

stackValue.pop\_front();

}

if (!stackValue.isEmpty()) {

temp = stackValue.head->date;

}

flag = true;

}

}

return listValue;

}

template <typename Key, typename Value>

void RBTree<Key, Value>::print() {

List<Key>\* ListKey = get\_keys();

List<Value>\* ListValue = get\_values();

for (int i = 0; i < ListKey->get\_size(); i++) {

std::cout << ListKey->at(i) << "-" << ListValue->at(i) << std::endl;

}

}

**functionStack.h**

#include"BRTree.h"

#include"stack.h"

#include"NodeBRTree.h"

template <typename Key, typename Value>

stack<Key, Value>::stack() {

head = nullptr;

tail = nullptr;

size = 0;

}

template <typename Key, typename Value>

stack<Key, Value>::~stack() {

clear();

}

template <typename Key, typename Value>

void stack<Key, Value>::clear() {

while (size != 0)

{

pop\_front();

}

}

template <typename Key, typename Value>

bool stack<Key, Value>::isEmpty() {

if (size != 0) {

return 0;

}

return 1;

}

template <typename Key, typename Value>

void stack<Key, Value>::push(Node<Key, Value>\* date) {

if (size == 0) {

head = new stackNode(date);

tail = head;

}

else {

head = new stackNode(date, head);

}

size++;

}

template <typename Key, typename Value>

void stack<Key, Value>::pop\_front() {

if (size == 0) {

return;

}

if (size == 1) {

delete head;

reset\_list();

}

else {

stackNode\* current = head;

head = head->next;

delete current;

}

size--;

}

template <typename Key, typename Value>

void stack<Key, Value>::pop\_back() {

if (size == 0) return;

if (size == 1) {

delete head;

reset\_list();

}

else {

stackNode\* current = head;

while (current->next != tail) {

current = current->next;

}

current->next = nullptr;

delete tail;

tail = current;

}

size--;

}

template <typename Key, typename Value>

void stack<Key, Value>::reset\_list()

{

head = nullptr;

tail = nullptr;

}

**List.h**

#ifndef List\_H

#define List\_H

template <typename Data>

class List

{

public:

List();

~List();

void reset\_list();

void push\_back(Data); // adding to the end of the list

void push\_front(Data); // adding to the top of the list

void pop\_back(); // deleting the last element

void pop\_front(); // deleting the first element

unsigned int get\_size(); // getting the list size

void print\_to\_console(); // output list items to the console using a separator

void clear(); // deleting all list items

bool isEmpty(); // checking if the list is empty

Data at(unsigned int index);

private:

class ListNode {

public:

Data data;

ListNode\* next;

ListNode(Data data, ListNode\* next = nullptr)

{

this->data = data;

this->next = next;

};

~ListNode()

{

}

};

ListNode\* tail;

ListNode\* head;

unsigned int size;

};

#endif

**NodeBRTree.h**

#ifndef NodeBRTree\_H

#define NodeBRTree\_H

#include<iostream>

template <typename Key,typename Value>

class Node {

public:

Key key;

Value value;

char colour;

Node\* Lnext;

Node\* Rnext;

Node\* parent;

Node(Key key, Value value, char colour = 'b', Node\* parent = nullptr, Node\* Lnext = nullptr, Node\* Rnext = nullptr)

{

this->parent = parent;

this->key = key;

this->value = value;

this->colour = colour;

this->Lnext = Lnext;

this->Rnext = Rnext;

};

Node(char colour = 'b', Node\* parent = nullptr, Node\* Lnext = nullptr, Node\* Rnext = nullptr)

{

this->parent = parent;

this->colour = colour;

this->Lnext = Lnext;

this->Rnext = Rnext;

};

~Node()

{

value = NULL;

key = NULL;

colour = NULL;

Lnext = nullptr;

Rnext = nullptr;

parent = nullptr;

}

};

#endif

**stack.h**

#ifndef stack\_H

#define stack\_H

#include"BRTree.h"

#include"NodeBRTree.h"

template <typename Key, typename Value>

class stack

{

private:

class stackNode {

public:

Node<Key, Value>\* date;

stackNode\* next;

stackNode(Node<Key, Value>\* date = nullptr, stackNode\* next = nullptr)

{

this->date = date;

this->next = next;

};

~stackNode()

{

}

};

public:

stackNode\* head;

stackNode\* tail;

unsigned int size;

stack();

~stack();

void clear();

bool isEmpty();

void push(Node<Key, Value>\* date);

void pop\_front();

void pop\_back();

void reset\_list();

};

#endif