**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»**

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

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Баязитов О.О. |
| Преподаватель |  | Тутуева А.В |

Санкт-Петербург

2021

## Постановка задачи

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

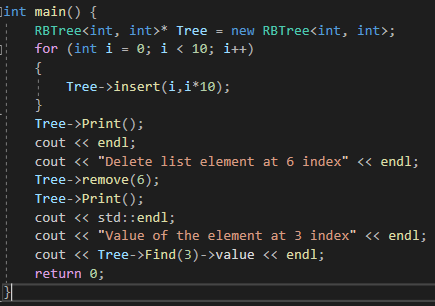
## Описание и оценка временной сложности методов

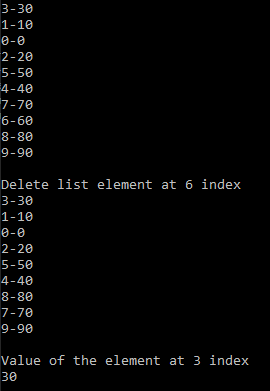
|  |  |  |
| --- | --- | --- |
| Название метода | Описание | Временная сложность |
| void Left\_Rotate(Node<Key, Value>\* x) | Левый поворот | O(lg(n)) |
| void Right\_Rotate(Node<Key, Value>\* x) | Правый поворот | O(lg(n)) |
| void insert(Key key, Value value) | Вставка элемента | O(lg(n)) |
| void 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 |  |
| Test1 | Вставка элемента в дерево |
| Test2 | Удаление элемента в дереве |
| Test3 | Проверка поиска элемента |
| Test4 | Проверка поиска элемента |
| Test5 | Вывод ключей пустого дерева |
| Test6 | Вывод пустого дерева |
| Test7 | 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)

{

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

{

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

{

Node<Key, Value>\* y;

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

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

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

y->color = 'b';

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

newNode = newNode->parent->parent;

}

else {

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

{

newNode = newNode->parent;

Left\_Rotate(newNode);

}

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

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

Right\_Rotate(newNode->parent->parent);

}

}

else {

Node<Key, Value>\* y;

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

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

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

y->color = 'b';

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

newNode = newNode->parent->parent;

}

else {

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

{

newNode = newNode->parent;

Right\_Rotate(newNode);

}

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

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

Left\_Rotate(newNode->parent->parent);

}

}

}

Root->color = 'b';

}

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

{

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

{

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

{

Node<Key, Value>\* w;

w = x->parent->Rnext;

if (w->color == 'r')

{

w->color = 'b';

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

Left\_Rotate(x->parent);

w = x->parent->Rnext;

}

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

{

w->color = 'r';

x = x->parent;

}

else

{

if (w->Rnext->color == 'b')

{

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

w->color = 'r';

Right\_Rotate(w);

w = x->parent->Rnext;

}

w->color = x->parent->color;

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

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

Left\_Rotate(x->parent);

x = Root;

}

}

else

{

Node<Key, Value>\* w;

w = x->parent->Lnext;

if (w->color == 'r')

{

w->color = 'b';

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

Right\_Rotate(x->parent);

w = x->parent->Lnext;

}

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

{

w->color = 'r';

x = x->parent;

}

else

{

if (w->Lnext->color == 'b')

{

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

w->color = 'r';

Left\_Rotate(w);

w = x->parent->Lnext;

}

w->color = x->parent->color;

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

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

Right\_Rotate(x->parent);

x = Root;

}

}

}

x->color = 'b';

}

void 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;

}

if (y != nil) {

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;

}

void 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;

}

if (y != nil) {

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;

}

public:

RBTree()

{

Root = nil;

}

~RBTree()

{

Clear();

nil = nullptr;

Root = nullptr;

}

Node<Key, Value>\* Root;

void insert(Key key, Value value) //adding an item with a key and a 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;

}

}

}

}

void remove(Key key) //deleting a tree element by key

{

Node<Key, Value>\* y;

Node<Key, Value>\* Z;

Node<Key, Value>\* X;

Z = Find(key);

y = Z;

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

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->color;

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->color = Z->color;

}

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

Remove\_fixup(X);

}

}

Node<Key, Value>\* Find(Key key)//item search by key

{

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)

{

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;

}

void Clear() //clearing an associative array

{

Preorder(Root);

delete nil;

nil = nullptr;

Root = nullptr;

}

void Preorder(Node<Key, Value>\* node) //wood cleaning

{

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

Preorder(node->Lnext);

Preorder(node->Rnext);

delete node;

}

List<Key>\* Get\_Keys() //returns a list of 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;

}

List<Value>\* Get\_Values() //returns a list of 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;

}

void Print() //console output

{

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;

}

}

};

#endif

**List.h**

#ifndef List\_H

#define List\_H

template <typename Data>

class List

{

public:

List()

{

head = nullptr;

tail = nullptr;

size = 0;

}

~List()

{

Clear();

}

void Reset\_List()

{

tail = nullptr;

head = nullptr;

}

unsigned int Get\_Size()

{

return size;

}

void Push\_Back(Data date)

{

if (size == 0)

{

head = new ListNode(date);

tail = head;

}

else {

tail->next = new ListNode(date);

tail = tail->next;

}

size++;

}

void Push\_Front(Data date)

{

if (size == 0)

{

head = new ListNode(date);

tail = head;

}

else {

head = new ListNode(date, head);

}

size++;

}

void 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--;

}

void Pop\_Front()

{

if (size == 0)

{

return;

}

if (size == 1)

{

delete head;

Reset\_List();

}

else {

ListNode\* current = head;

head = head->next;

delete current;

}

size--;

}

void 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;

}

}

void Clear()

{

while (size != 0)

{

Pop\_Front();

}

}

bool IsEmpty()

{

if (size != 0) {

return 0;

}

return 1;

}

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;

}

}

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 color;

Node\* Lnext;

Node\* Rnext;

Node\* parent;

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

{

this->key = key;

this->value = value;

this->color = color;

this->parent = parent;

this->Lnext = Lnext;

this->Rnext = Rnext;

};

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

{

this->color = color;

this->parent = parent;

this->Lnext = Lnext;

this->Rnext = Rnext;

};

~Node()

{

value = NULL;

key = NULL;

color = 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()

{

Head = nullptr;

Tail = nullptr;

Size = 0;

}

~Stack()

{

Clear();

}

void Clear()

{

while (Size != 0)

{

Pop\_Front();

}

}

bool IsEmpty()

{

if (Size != 0)

{

return 0;

}

return 1;

}

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

{

if (Size == 0)

{

Head = new StackNode(date);

Tail = Head;

}

else

{

Head = new StackNode(date, Head);

}

Size++;

}

void Pop\_Front()

{

if (Size == 0)

{

return;

}

if (Size == 1)

{

delete Head;

Reset\_List();

}

else

{

StackNode\* current = Head;

Head = Head->next;

delete current;

}

Size--;

}

void 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--;

}

void Reset\_List()

{

Head = nullptr;

Tail = nullptr;

}

};

#endif