ГУАП

КАФЕДРА № 53

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Старший преподаватель |  |  |  | Суетина Т.А. |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРОТОРНОЙ РАБОТЕ №6 |
| «Красно-черные деревья» |
| по курсу: ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 5836 |  |  |  | Храмченко Д.А. |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2020

1. **Цель работы**: получить теоретические знания по основным операциям со сбалансированным деревьям поиска: вставка узла, удаление узла, поиск заданного элемента в дереве по ключу и практически реализовать полученные знания.
2. **Задание**:

Написать функцию, реализующую малый правый поворот при добав-лении узла в дерево.

Написать функцию, реализующую малый левый поворот при добавле-нии узла в дерево.

Написать функцию, реализующую малый правый поворот при удалении узла.

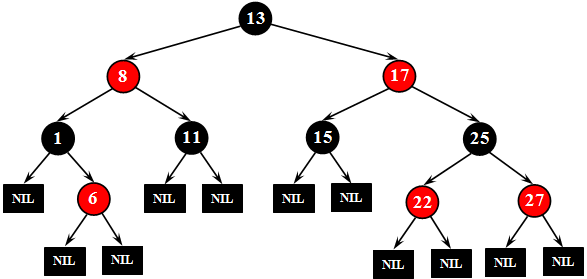
Написать функцию, реализующую большой левый поворот при удалении узла.

**Методические рекомендации**

Бинарные деревья работают лучше всего, когда они сбалансированы, когда длина пути от корня до любого из листьев находится в определенных пределах, связанных с числом вершин. Красно-черные деревья являются одним из способов балансировки деревьев. Название происходит от стандартной раскраски узлов таких деревьев в красный и черный цвета. Цвета вершин используются при балансировке дерева.

**Красно-черное дерево** ( *Red-Black-Tree, RB-Tree* ) – это бинарное дерево со следующими свойствами ( рис. 1):

* каждая вершина должна быть окрашена либо в черный, либо в красный цвет;
* корень дерева должен быть черным;
* листья дерева должны быть черными и объявляться как NIL-вершины (NIL-узлы, то есть "виртуальные" узлы, наследники узлов, которые обычно называют листьями; на них "указывают" NULL указатели);
* каждый красный узел должен иметь черного предка;
* на всех ветвях дерева, ведущих от его корня к листьям, число черных вершин одинаково.



**Рис. 1.** Красно-черное дерево

Количество черных вершин на ветви от корня до листа называется *черной высотой дерева*. Перечисленные свойства гарантируют, что самая длинная ветвь от корня к листу не более чем вдвое длиннее любой другой ветви от корня к листу.

Над красно-черными деревьями можно выполнять все те же основные операции, что и над бинарными деревьями.

Приведем функции следующих операций над красно-черными деревьями: создание дерева, печать (просмотр) дерева, обход дерева, проверка пустоты дерева и удаление дерева.

//создание красно-черного дерева

void Make\_RBTree(RBTree\*\* Node, int n){

int Data;

while (n > 0) {

cout << "Введите значение ";

cin >> Data;

Insert\_Node(Node, Data);

n--;

}

}

//добавление узла в красно-черное дерево

void Insert\_Node(RBTree\*\* Node,int Data) {

RBTree \*\*Curent, \*Parent, \*New\_Node;

Curent = Node;

Parent = NIL;

// Поиск местоположения

while (\*Curent != NIL) {

Parent = (\*Curent);

Curent = Data < (\*Curent)->Data ? &((\*Curent)->Left) : &((\*Curent)->Right);

}

// Создание нового узла

New\_Node = new RBTree();

New\_Node->Data = Data;

New\_Node->Parent = Parent;

New\_Node->Left = NIL;

New\_Node->Right = NIL;

New\_Node->color = RED;

// Вставка элемента в дерево

if(Parent != NIL){

if (Data < Parent->Data) Parent->Left = New\_Node;

else Parent->Right = New\_Node;

}

else (\*Curent) = New\_Node;

Insert\_Fixup(Node, New\_Node);

}

// Поддержка баланса дерева после вставки нового элемента

void Insert\_Fixup(RBTree\*\* Node,RBTree\* New\_Node){

RBTree\* Current = New\_Node;

// Проверка свойств дерева

while (Current != \*(Node) && Current->Parent->color == RED){

// если есть нарушение

if (Current->Parent == Current->Parent->Parent->Left) {

RBTree \*ptr = Current->Parent->Parent->Right;

if (ptr->color == RED) {

Current->Parent->color = BLACK;

ptr->color = BLACK;

Current->Parent->Parent->color = RED;

Current = Current->Parent->Parent;

}

else {

if (Current == Current->Parent->Right) {

// сделать Current левым потомком

Current = Current->Parent;

Rotate\_Left(Node,Current);

}

// перекрасить и повернуть

Current->Parent->color = BLACK;

Current->Parent->Parent->color = RED;

Rotate\_Right(Node,Current->Parent->Parent);

}

}

else {

RBTree \*ptr = Current->Parent->Parent->Left;

if (ptr->color == RED) {

Current->Parent->color = BLACK;

ptr->color = BLACK;

Current->Parent->Parent->color = RED;

Current = Current->Parent->Parent;

}

else {

if (Current == Current->Parent->Left) {

Current = Current->Parent;

Rotate\_Right(Node,Current);

}

Current->Parent->color = BLACK;

Current->Parent->Parent->color = RED;

Rotate\_Left(Node,Current->Parent->Parent);

}

}

}

(\*Node)->color = BLACK;

}

//поворот узла Current влево

void Rotate\_Left(RBTree\*\* Node,RBTree \*Current) {

RBTree \*ptr = Current->Right;

Current->Right = ptr->Left;

if (ptr->Left != NIL) ptr->Left->Parent = Current;

if (ptr != NIL) ptr->Parent = Current->Parent;

if (Current->Parent != NIL) {

if (Current == Current->Parent->Left)

Current->Parent->Left = ptr;

else

Current->Parent->Right = ptr;

}

else {

(\*Node) = ptr;

}

ptr->Left = Current;

if (Current != NIL) Current->Parent = ptr;

}

//поворот узла Current вправо

void Rotate\_Right(RBTree\*\* Node,RBTree \*Current) {

RBTree \*ptr = Current->Left;

Current->Left = ptr->Right;

if (ptr->Right != NIL) ptr->Right->Parent = Current;

if (ptr != NIL) ptr->Parent = Current->Parent;

if (Current->Parent != NIL) {

if (Current == Current->Parent->Right)

Current->Parent->Right = ptr;

else

Current->Parent->Left = ptr;

}

else {

(\*Node) = ptr;

}

ptr->Right = Current;

if (Current != NIL) Current->Parent = ptr;

}

//печать красно-черного дерева

void Print\_RBTree(RBTree\* Node, int l){

int i;

if (Node != NIL) {

Print\_RBTree(Node->Right, l+1);

for (i=0; i< l; i++) cout << " ";

if (Node->color == RED)

SetConsoleTextAttribute(hStd,FOREGROUND\_RED);

cprintf ("%4ld", Node->Data);

SetConsoleTextAttribute(hStd,atr);

Print\_RBTree(Node->Left, l+1);

}

else cout << endl;

}

//прямой обход красно-черного дерева

void PreOrder\_RBTree(RBTree\* Node){

if (Node != NIL) {

printf ("%3ld",Node->Data);

PreOrder\_RBTree(Node->Left);

PreOrder\_RBTree(Node->Right);

}

}

//обратный обход красно-черного дерева

void PostOrder\_RBTree(RBTree\* Node){

if (Node != NIL) {

PostOrder\_RBTree(Node->Left);

PostOrder\_RBTree(Node->Right);

printf ("%3ld",Node->Data);

}

}

//симметричный обход красно-черного дерева

void SymmetricOrder\_RBTree(RBTree\* Node){

if (Node != NIL) {

PostOrder\_RBTree(Node->Left);

printf ("%3ld",Node->Data);

PostOrder\_RBTree(Node->Right);

}

}

//проверка пустоты красно-черного дерева

bool Empty\_RBTree(RBTree\* Node){

return ( Node == NIL ? true : false );

}

//освобождение памяти, выделенной под красно-черное дерево

void Delete\_RBTree(RBTree\* Node){

if (Node != NIL) {

Delete\_RBTree(Node->Left);

Delete\_RBTree(Node->Right);

delete(Node);

}

}

1. **Код программы:**

//! класс дерева

class RBtree {

struct node\_st { node\_st\* p1, \* p2; int value; bool red; }; // структура узла

node\_st\* tree\_root; // корень

int nodes\_count; // число узлов дерева

private:

node\_st\* NewNode(int value); // выделение новой вешины

void DelNode(node\_st\*); // удаление вершины

node\_st\* Rotate21(node\_st\*); // вращение влево

node\_st\* Rotate12(node\_st\*); // вращение вправо

void BalanceInsert(node\_st\*\*); // балансировка вставки

bool BalanceRemove1(node\_st\*\*); // левая балансировка удаления

bool BalanceRemove2(node\_st\*\*); // правая балансировка удаления

bool Insert(int, node\_st\*\*); // рекурсивная часть вставки

bool Remove(node\_st\*\*, int); //рекурсивная часть удаления

};

// выделение новой вешины

RBtree::node\_st\* RBtree::NewNode(int value)

{

nodes\_count++;

node\_st\* node = new node\_st;

node->value = value;

node->p1 = node->p2 = 0;

node->red = true;

return node;

}

// удаление вершины

void RBtree::DelNode(node\_st\* node)

{

nodes\_count--;

delete node;

}

// вращение влево

RBtree::node\_st\* RBtree::Rotate21(node\_st\* node)

{

node\_st\* p2 = node->p2;

node\_st\* p21 = p2->p1;

p2->p1 = node;

node->p2 = p21;

return p2;

}

// вращение вправо

RBtree::node\_st\* RBtree::Rotate12(node\_st\* node)

{

node\_st\* p1 = node->p1;

node\_st\* p12 = p1->p2;

p1->p2 = node;

node->p1 = p12;

return p1;

}

// балансировка вершины

void RBtree::BalanceInsert(node\_st\*\* root)

{

node\_st\* p1, \* p2, \* px1, \* px2;

node\_st\* node = \*root;

if (node->red) return;

p1 = node->p1;

p2 = node->p2;

if (p1 && p1->red) {

px2 = p1->p2;

if (px2 && px2->red) p1 = node->p1 = Rotate21(p1);

px1 = p1->p1;

if (px1 && px1->red) {

node->red = true;

p1->red = false;

if (p2 && p2->red) {

px1->red = true;

p2->red = false;

return;

}

\*root = Rotate12(node);

return;

}

}

// тоже самое в другую сторону

if (p2 && p2->red) {

px1 = p2->p1;

if (px1 && px1->red) p2 = node->p2 = Rotate12(p2);

px2 = p2->p2;

if (px2 && px2->red) {

node->red = true;

p2->red = false;

if (p1 && p1->red) {

px2->red = true;

p1->red = false;

return;

}

\*root = Rotate21(node);

return;

}

}

}

bool RBtree::BalanceRemove1(node\_st\*\* root)

{

node\_st\* node = \*root;

node\_st\* p1 = node->p1;

node\_st\* p2 = node->p2;

if (p1 && p1->red) {

p1->red = false; return false;

}

if (p2 && p2->red) {

node->red = true;

p2->red = false;

node = \*root = Rotate21(node);

if (BalanceRemove1(&node->p1)) node->p1->red = false;

return false;

}

unsigned int mask = 0;

node\_st\* p21 = p2->p1;

node\_st\* p22 = p2->p2;

if (p21 && p21->red) mask |= 1;

if (p22 && p22->red) mask |= 2;

switch (mask)

{

case 0:

p2->red = true;

return true;

case 1:

case 3:

p2->red = true;

p21->red = false;

p2 = node->p2 = Rotate12(p2);

p22 = p2->p2;

case 2:

p2->red = node->red;

p22->red = node->red = false;

\*root = Rotate21(node);

}

return false;

}

bool RBtree::BalanceRemove2(node\_st\*\* root)

{

node\_st\* node = \*root;

node\_st\* p1 = node->p1;

node\_st\* p2 = node->p2;

if (p2 && p2->red) { p2->red = false; return false; }

if (p1 && p1->red) { // случай 1

node->red = true;

p1->red = false;

node = \*root = Rotate12(node);

if (BalanceRemove2(&node->p2)) node->p2->red = false;

return false;

}

unsigned int mask = 0;

node\_st\* p11 = p1->p1;

node\_st\* p12 = p1->p2;

if (p11 && p11->red) mask |= 1;

if (p12 && p12->red) mask |= 2;

switch (mask) {

case 0:

p1->red = true;

return true;

case 2:

case 3:

p1->red = true;

p12->red = false;

p1 = node->p1 = Rotate21(p1);

p11 = p1->p1;

case 1:

p1->red = node->red;

p11->red = node->red = false;

\*root = Rotate12(node);

}

return false;

}

// рекурсивная часть вставки

bool RBtree::Insert(int value, node\_st\*\* root)

{

node\_st\* node = \*root;

if (!node)\* root = NewNode(value);

else {

if (value == node->value) return true;

if (Insert(value, value < node->value ? &node->p1 : &node->p2)) return true;

BalanceInsert(root);

}

return false;

}

// рекурсивная часть удаления

bool RBtree::Remove(node\_st\*\* root, int value)

{

node\_st\* t, \* node = \*root;

if (!node) return false;

if (node->value < value) {

if (Remove(&node->p2, value)) return BalanceRemove2(root);

}

else if (node->value > value) {

if (Remove(&node->p1, value)) return BalanceRemove1(root);

}

else {

bool res;

if (!node->p2) {

\*root = node->p1;

res = !node->red;

}

else {

res = GetMin(&node->p2, root);

t = \*root;

t->red = node->red;

t->p1 = node->p1;

t->p2 = node->p2;

if (res) res = BalanceRemove2(root);

}

DelNode(node);

return res;

}

return 0;

}

1. **Вывод:**

В ходе выполнения данной лабороторной работы я получила теоретические знания по основным операциям со сбалансированным деревьям поиска: вставка узла, удаление узла, поиск заданного элемента в дереве по ключу и практически реализовать полученные знания.