МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Алгоритмы и структуры данных»

Тема: Красно-черное дерево

Студент гр. 9381 _____ Авдеев Илья

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Изучить структуру БДП: красно-чёрное дерево.

Задание.

Варинат 27

По заданной последовательности элементов Elem построить структуру данных. Для построенной структуры данных проверить, входит ли в неё элемент e типа Elem, и если входит, то в скольких экземплярах. Добавить элемент e в структуру данных. Предусмотреть возможность повторного выполнения с другим элементом.

Описание

Красно-черное дерево - это еще одна форма сбалансированного бинарного поискового дерева. Впервые оно было представлено в 1972 году как еще одна разновидность сбалансированного бинарного дерева. Время поиска, вставки или удаления узла для красно-черного дерева является логарифмической функцией от числа узлов.

Данный тип деревьев отличается свойствами:

1.Узел может быть либо красным, либо чёрным и имеет двух потомков;

2.Корень — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный;

3.Все листья, не содержащие данных — чёрные.

4.Оба потомка каждого красного узла — чёрные.

5.Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

Благодаря этим ограничениям, путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего и дерево примерно сбалансировано. Операции вставки, удаления и поиска требуют в худшем случае времени, пропорционального длине дерева, что позволяет красно-чёрным деревьям быть более эффективными в худшем случае, чем обычные двоичные деревья поиска.

Пусть для красно-чёрного дерева T число чёрных узлов от корня до листа равно B. Тогда

кратчайший возможный путь до любого листа содержит B узлов и все они чёрные. Более длинный возможный путь может быть построен путём включения красных узлов. Однако, благодаря п.4 в дереве не может быть двух красных узлов подряд, а согласно пп. 2 и 3, путь начинается и кончается чёрным узлом. Поэтому самый длинный возможный путь состоит из 2B-1 узлов, попеременно красных и чёрных.

Описание алгоритма функции int find(int value)

Сначала проводится проверка наличия элемента структуры. Если ее нет, то функция возвращает 0.

Создается возвращаемое значение res типа int равное 0;

В цикле проводится сравнение содержимого узла с элементом value, если они равны к i прибавляется один. Если содержимое больше узла рассмотрим левое плече, иначе правое.

Выводы

Было освоена структура данных Бинарное дерево поиска: Красно-черное дерево. Освоены функции работы с ним. В ходе выполнения работы была реализована функция поиска элемента и вставки в структуру.

ПРИЛОЖЕНИЕ А

СОДЕРЖАНИЕ ФАЙЛА Source.cpp

```
#include <iostream>
#include <time.h>
#include <conio.h>

class RBtree
{
	struct node
	{
		node* left, * right;
		int value;
		bool red;
	};
	node* tree_root;
	int size;
private:
	node* make_node(int value);
	void del_node(node*);
	void clear(node*);
	node* rotate_right(node*);
	node* rotate_left(node*);
	void balance_insert(node**);
	bool balance_remove_case1(node**);
	bool balance_remove_case2(node**);
	bool insert(int, node**);
	bool getmin(node**, node**);
	bool remove(node**, int);
	void print_tree(node*& ptr, int u);
public:
	RBtree();
	~RBtree();
	void clear();
	int find(int);
	void insert(int);
	void remove(int);
	void print();
};


RBtree::RBtree()
```

```
{
	tree_root = 0;
	size = 0;
}

RBtree::~RBtree()
{
	clear(tree_root);
}


RBtree::node* RBtree::make_node(int value)
{
	size++;
	node* n = new node;
	n->value = value;
	n->left = n->right = NULL;
	n->red = true;
	return n;
}


void RBtree::del_node(node* node)
{
	size--;
	delete node;
}


void RBtree::clear(node* node)
{
	if (!node) return;
	clear(node->left);
	clear(node->right);
	del_node(node);
}


RBtree::node* RBtree::rotate_left(node* n)
{
	node* right = n->right;
	node* rleft = right->left;
```

```
	right->left = n;
	n->right = rleft;
	return right;
}


RBtree::node* RBtree::rotate_right(node* n)
{
	node* left = n->left;
	node* lright = left->right;
	left->right = n;
	n->left = lright;
	return left;
}


void RBtree::balance_insert(node** root)
{
	node* left, * right, * lleft, * lright;
	node* node = *root;
	if (node->red) return;
	left = node->left;
	right = node->right;
	if (left && left->red)
	{
		lright = left->right;
		if (lright && lright->red)
			left = node->left = rotate_left(left);
		lleft = left->left;
		if (lleft && lleft->red)
		{
			node->red = true;
			left->red = false;
			if (right && right->red)
			{
				lleft->red = true;
				right->red = false;
				return;
			}
			*root = rotate_right(node);
			return;
		}
```

```
	}

	if (right && right->red)
	{
		lleft = right->left;
		if (lleft && lleft->red)
			right = node->right = rotate_right(right);
		lright = right->right;
		if (lright && lright->red)
		{
			node->red = true;
			right->red = false;
			if (left && left->red)
			{
				lright->red = true;
				left->red = false;
				return;
			}
			*root = rotate_left(node);
			return;
		}
	}
}

bool RBtree::balance_remove_case1(node** root)
{
	node* n = *root;
	node* left = n->left;
	node* right = n->right;
	if (right && left->red)
	{
		left->red = false; return false;
	}
	if (right && right->red)
	{
		n->red = true;
		right->red = false;
		n = *root = rotate_left(n);
		if (balance_remove_case1(&n->left)) n->left->red = false;
		return false;
	}
	unsigned int mask = 0;
```

```
	node* rleft = right->left;
	node* rright = right->right;
	if (rleft && rleft->red) mask |= 1;
	if (rright && rright->red) mask |= 2;
	switch (mask)
	{
	case 0:
		right->red = true;
		return true;
	case 1:
	case 3:
		right->red = true;
		rright->red = false;
		right = n->right = rotate_right(right);
		rright = right->right;
	case 2:
		right->red = n->red;
		rright->red = n->red = false;
		*root = rotate_left(n);
	}
	return false;
}

bool RBtree::balance_remove_case2(node** root)
{
	node* n = *root;
	node* left = n->left;
	node* right = n->right;
	if (right && right->red) { right->red = false; return false; }
	if (left && left->red)
	{
		n->red = true;
		left->red = false;
		n = *root = rotate_right(n);
		if (balance_remove_case2(&n->right)) n->right->red = false;
		return false;
	}
	unsigned int mask = 0;
	node* lleft = left->left;
	node* lright = left->right;
	if (lleft && lleft->red) mask |= 1;
	if (lright && lright->red) mask |= 2;
```

```
	switch (mask)
	{
	case 0:
		left->red = true;
		return true;
	case 2:
	case 3:
		left->red = true;
		lright->red = false;
		left = n->left = rotate_left(left);
		lleft = left->left;
	case 1:
		left->red = n->red;
		lleft->red = n->red = false;
		*root = rotate_right(n);
	}
	return false;
}


int RBtree::find(int value)
{
	node* n = tree_root;
	int i = 0;
	while (n)
	{
		if (n->value == value)
			i++;
		n = n->value > value ? n->left : n->right;
	}
	return i;
}



bool RBtree::insert(int value, node** root)
{
	node* n = *root;
	if (!n) *root = make_node(value);
	else
	{
		if (value == n->value) return true;
```

```
            if (insert(value, value < n->value ? &n->left : &n->right)) return
true;
            balance_insert(root);
      }
      return false;
}



bool RBtree::getmin(node** root, node** res)
{
      node* node = *root;
      if (node->left)
      {
            if (getmin(&node->left, res)) return balance_remove_case1(root);
      }
      else
      {
            *root = node->right;
            *res = node;
            return !node->red;
      }
      return false;
}


bool RBtree::remove(node** root, int value)
{
      node* t, * node = *root;
      if (!node) return false;
      if (node->value < value)
      {
            if (remove(&node->right, value))    return balance_remove_case2(root);
      }
      else if (node->value > value)
      {
            if (remove(&node->left, value))     return balance_remove_case1(root);
      }
      else
      {
            bool res;
            if (!node->right)
```

```
            {
                  *root = node->left;
                  res = !node->red;
            }
            else
            {
                  res = getmin(&node->right, root);
                  t = *root;
                  t->red = node->red;
                  t->left = node->left;
                  t->right = node->right;
                  if (res) res = balance_remove_case2(root);
            }
            del_node(node);
            return res;
      }
      return 0;
}


void RBtree::insert(int value)
{
      insert(value, &tree_root);
      if (tree_root) tree_root->red = false;
}

void RBtree::remove(int value)
{
      remove(&tree_root, value);
}


void RBtree::clear()
{
      clear(tree_root);
      tree_root = 0;
}


void RBtree::print_tree(node*& ptr, int u)
{
      if (ptr == nullptr)
```

```
	{
		u++;
		for (int i = 0; i < u - 1; ++i) std::cout << "\t";
		std::cout << "NULL\n";
		return;
	}
	else
	{
		print_tree(ptr->right, ++u);
		for (int i = 0; i < u - 1; ++i) std::cout << "\t";
		std::cout << ptr->value << "(";
		if (ptr->red)
			std::cout << "red";
		else
			std::cout << "black";
		std::cout << ")\n";
		u--;
	}
	print_tree(ptr->left, ++u);
}

void RBtree::print()
{
	print_tree(tree_root, 0);
}

int main()
{
	int n = 1, i;
	RBtree tree;

	setlocale(LC_ALL, "ru");

	srand(time(0));
	switch (2)
	{
	case 1:
		while (n != NULL)
		{
			std::cin >> n;
			tree.insert(n);
		}
```

```
            break;
        case 2:
            for (int i = 0; i < 20; i++)
            {
                n = rand() % 20 + rand() % 20;
                tree.insert(n);
            }
            break;
        default:
            break;
        }

        tree.print();
        while (1)
        {

            std::cout << "Найти элемент: ";
            std::cin >> n;

            std::cout << std::endl << tree.find(n) << std::endl;

            std::cout << "Влючить его? y - да, n - закончить\n";
            int k = _getch();
            if (k == 'y')
            {
                tree.insert(n);
                tree.print();
            }
            else
                break;
        }
        tree.clear();

        getchar();
        return 0;
}
```