****

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«МИРЭА – Российский технологический университет»**

# РТУ МИРЭА

Отчет по выполнению практического задания 7.1

**Тема: НЕЛИНЕЙНЫЕ СТРУКТУРЫ**

Дисциплина: Структуры и алгоритмы обработки данных

Выполнил студент Королихин В.Н.

группа ИКБО-21-23

**Москва 2024**

**Цель работы:** освоить приёмы работы с бинарными деревьями. Получить умения и навыки разработки и реализации операций над структурой данных - бинарное дерево.

ЗАДАНИЕ 1

**Формулировка задачи**

Составить программу создания двоичного дерева поиска и реализовать процедуры для работы с деревом согласно варианту.

Вариант 15:

* Тип значения узла: целое,
* Тип дерева: Красно-чёрное дерево.

Реализовать алгоритмы:

1. Вставка элемента,
2. Симметричный обход,
3. Обход в ширину,
4. Найти сумму значений листьев,
5. Найти среднее арифметическое всех узлов

**Математическая модель решения (описание алгоритма)**

Класс RedBlackTree содержит корневой узел, методы 1-5 и вспомогательные методы.

Узлы представлены структурой Node. Эта структура, помимо стандартных полей узла бинарного дерева, имеет поле bool color – цвет узла.

Реализация методов:

1. Создаётся новый узел, содержащий нужное значение.
2. Этот узел окрашивается в красный цвет.
3. Используя стандартный обход бинарного дерева поиска ищем место, куда можно вставить новый элемент (узел),
4. Если вставляем в корень дерева, то его надо перекрасить в черный цвет,
5. Иначе, если найденный родитель красный, вызывается вспомогательный метод void fixInsert(Node\*& node).
6. Пока node не является корнем, окрашен в красный и имеет красного родителя, выполняются следующие пункты,
7. Обновляются указатели parent и grandparent на соответствующие значения.
8. Определяется указатель Node\* uncle, “дядя” элемента node.
9. Если uncle тоже красный, то grandparent перекрашивается в красный, а parent и uncle – в чёрный, указателю node присваивается указатель grandparent, возвращаемся к пункту 1.5.1)
10. Если uncle не красный, то проверяем, является ли node правым потомком своего родителя.
11. Если является правым, то осуществляем левый поворот вокруг parent, а затем правый поворот вокруг grandparent
12. Если является левым, то сразу осуществляем правый поворот вокруг grandparent
13. Создаётся рекурсивный метод void inOrderTraversalHelper(Node\* root),
14. Если root – пустой, то выполняем команду return
15. Вызываем рекурсивно метод inOrderTraversalHelper с аргументом root->left
16. Выводим число, хранящееся в текущем узле root
17. Вызываем рекурсивно метод inOrderTraversalHelper с аргументом root->right.
18. Создаётся метод void levelOrderTraversalHelper (Node\* root),
19. Если root – пустой, то выполняем команду return
20. Создаем очередь q
21. Заносим в очередь root
22. Пока очередь непустая создаётся указатель Node\* current
23. Указателю current присваивается первый элемент очереди q, затем этот элемент из очереди удаляется.
24. Выводим число, хранящееся в текущем узле current
25. Если левый потомок узла current не является пустым, то он заносится в очередь
26. Если правый потомок узла current не является пустым, то он заносится в очередь
27. Возвращаемся к пункту 3.5).
28. Создаётся метод int sumOfLeavesHelper (Node\* root),
29. Если root – пустой, то возвращаем 0,
30. Если и левый, и правый потомки узла root являются пустыми, т.е. root – листок, то возвращаем число, хранящееся в нем,
31. Рекурсивно вызывается метод sumOfLeavesHelper с аргументом root->left,
32. Рекурсивно вызывается метод sumOfLeavesHelper с аргументом root->right,
33. Результаты рекурсивных вызовов складываются и возвращаются.
34. Создаётся метод double findAverageHelper(Node\* root)
35. Если root – пустой, то возвращаем 0.0,
36. Инициализируются int-переменные sum и count,
37. Вызывается метод calculateSumAndCount, которому в качестве аргументов передаются по ссылке переменные sum и count
38. Возвращается результат sum / count

Описание вспомогательного метода calculateSumAndCount

1. Создаётся рекурсивный метод void calculateSumAndCount(Node\* root, int& sum, int& count),
2. Если root – пустой, то выполняем команду return,
3. Добавляем значение текущего узла в сумму sum,
4. Инкрементируем счётчик count,
5. Вызываем рекурсивно текущий метод для левой и правой веток.

Описание вспомогательного метода rotateLeft

1. Создаётся метод void rotateLeft(Node\*& node),
2. Создается указатель child, который указывает на правого ребенка узла node.
3. Устанавливается правый ребенок узла node в качестве левого поддерева правого ребенка child.
4. Если новое правое поддерево существует, обновляется ссылка на родителя этого поддерева, указывая на текущий узел node.
5. Родитель правого ребенка child устанавливается как родитель текущего узла node.
6. Если текущий узел является корнем дерева, то правый ребенок child становится новым корнем дерева. В противном случае, обновляются ссылки на родителя, чтобы правый ребенок child стал соответствующим ребенком родителя.
7. Левый ребенок правого ребенка child устанавливается как текущий узел node.
8. Родитель текущего узла node обновляется, устанавливая его как правого ребенка для узла child.

Описание вспомогательного метода rotateRight

Создаётся метод void rotateRight (Node\*& node),

1. Создается указатель child, который указывает на левого ребенка узла node.
2. Устанавливается левый ребенок узла node в качестве правого поддерева левого ребенка child.
3. Если новое левое поддерево существует, обновляется ссылка на родителя этого поддерева, указывая на текущий узел node.
4. Родитель левого ребенка child устанавливается как родитель текущего узла node.
5. Если текущий узел является корнем дерева, то левый ребенок child становится новым корнем дерева. В противном случае, обновляются ссылки на родителя, чтобы левый ребенок child стал соответствующим ребенком родителя.
6. Правый ребенок левого ребенка child устанавливается как текущий узел node.
7. Родитель текущего узла node обновляется, устанавливая его как левого ребенка для узла child.

**Код программы**

class RedBlackTree {

struct Node {

int data;

bool color; // 0 - черный, 1 - красный

Node\* parent;

Node\* left;

Node\* right;

// Конструктор

Node(int value) : data(value), color(1), parent(nullptr), left(nullptr), right(nullptr) {}

};

// Корень

Node\* root;

// Вспомогательная функция: левое вращение

void rotateLeft(Node\*& node)

{

Node\* child = node->right;

node->right = child->left;

if (node->right != nullptr)

node->right->parent = node;

child->parent = node->parent;

if (node->parent == nullptr)

root = child;

else if (node == node->parent->left)

node->parent->left = child;

else

node->parent->right = child;

child->left = node;

node->parent = child;

}

// Вспомогательная функция: правое вращение

void rotateRight(Node\*& node)

{

Node\* child = node->left;

node->left = child->right;

if (node->left != nullptr)

node->left->parent = node;

child->parent = node->parent;

if (node->parent == nullptr)

root = child;

else if (node == node->parent->left)

node->parent->left = child;

else

node->parent->right = child;

child->right = node;

node->parent = child;

}

// Вспомогательная функция: разрешение противоречий в дереве при вставке

void fixInsert(Node\*& node)

{

Node\* parent = nullptr;

Node\* grandparent = nullptr;

while (node != root && node->color

&& node->parent->color) {

parent = node->parent;

grandparent = parent->parent;

if (parent == grandparent->left) {

Node\* uncle = grandparent->right;

if (uncle != nullptr

&& uncle->color) {

grandparent->color = 1;

parent->color = 0;

uncle->color = 0;

node = grandparent;

}

else {

if (node == parent->right) {

rotateLeft(parent);

node = parent;

parent = node->parent;

}

rotateRight(grandparent);

swap(parent->color, grandparent->color);

node = parent;

}

}

else {

Node\* uncle = grandparent->left;

if (uncle != nullptr

&& uncle->color) {

grandparent->color = 1;

parent->color = 0;

uncle->color = 0;

node = grandparent;

}

else {

if (node == parent->left) {

rotateRight(parent);

node = parent;

parent = node->parent;

}

rotateLeft(grandparent);

swap(parent->color, grandparent->color);

node = parent;

}

}

}

root->color = 0;

}

// Вспомогательная функция: помощник для симметричного обхода

void inOrderTraversalHelper(Node\* root) {

if (!root) return;

// Сначала обходим левое поддерево

inOrderTraversalHelper(root->left);

// Посещаем текущий узел

cout << root->data << " ";

// Затем обходим правое поддерево

inOrderTraversalHelper(root->right);

}

// Вспомогательная функция: помощник для обхода в ширину

void levelOrderTraversalHelper(Node\* root) {

if (root == nullptr) {

return;

}

// Очередь для хранения узлов на текущем уровне

queue<Node\*> q;

// Помещаем корень в очередь

q.push(root);

while (!q.empty()) {

// Извлекаем узел из очереди

Node\* current = q.front();

q.pop();

// Обрабатываем текущий узел

cout << current->data << " ";

// Добавляем левое поддерево в очередь

if (current->left != nullptr) {

q.push(current->left);

}

// Добавляем правое поддерево в очередь

if (current->right != nullptr) {

q.push(current->right);

}

}

}

// Вспомогательная функция: помощник для нахождения суммы листьев

int sumOfLeavesHelper(Node\* root) {

if (root == nullptr) {

return 0; // Если узел пустой, возвращаем 0

}

// Если узел является листом

if (root->left == nullptr && root->right == nullptr) {

return root->data;

}

// Рекурсивно суммируем листья в левом и правом поддеревьях

return sumOfLeavesHelper(root->left) + sumOfLeavesHelper(root->right);

}

// Вспомогательная функция: помощник для подсчета суммы значений узлов и их количества

void calculateSumAndCount(Node\* root, int& sum, int& count) {

if (root == nullptr) {

return;

}

// Добавляем значение текущего узла в сумму

sum += root->data;

// Увеличиваем счетчик узлов

count++;

// Рекурсивно обходим левое и правое поддерево

calculateSumAndCount(root->left, sum, count);

calculateSumAndCount(root->right, sum, count);

}

// Вспомогательная функция: помощник для вычисления среднего арифметического

double findAverageHelper(Node\* root) {

if (root == nullptr) {

return 0.0; // Если дерево пустое, среднее равно 0

}

int sum = 0;

int count = 0;

// Вычисляем сумму и количество узлов

calculateSumAndCount(root, sum, count);

// Возвращаем среднее арифметическое

return static\_cast<double>(sum) / count;

}

// Вспомогательная функция: помощник для вывода Красно-Черного дерева

void printHelper(Node\* root, string indent, bool last)

{

if (root != nullptr) {

cout << indent;

if (last) {

cout << "R----";

indent += " ";

}

else {

cout << "L----";

indent += "| ";

}

string sColor

= (root->color) ? "Красный" : "Черный";

cout << root->data << "(" << sColor << ")"

<< endl;

printHelper(root->left, indent, false);

printHelper(root->right, indent, true);

}

}

public:

// Конструктор

RedBlackTree() : root(nullptr) {}

// Метод вставки

void insert(int key)

{

Node\* node = new Node(key);

Node\* parent = nullptr;

Node\* current = root;

while (current != nullptr) {

parent = current;

if (node->data < current->data)

current = current->left;

else

current = current->right;

}

node->parent = parent;

if (parent == nullptr)

root = node;

else if (node->data < parent->data)

parent->left = node;

else

parent->right = node;

fixInsert(node);

}

// Метод симмтеричного обхода

void inOrderTraversal() {

if (root == nullptr) {

cout << "Дерево пусто." << endl;

}

else {

cout << "Симметричный обход дерева: ";

inOrderTraversalHelper(root);

cout << endl;

}

}

// Метод обхода в ширину

void levelOrderTraversal() {

if (root == nullptr) {

cout << "Дерево пусто." << endl;

}

else {

cout << "Обход дерева в ширину: ";

levelOrderTraversalHelper(root);

cout << endl;

}

}

// Метод вывода суммы листьев

void sumOfLeaves() {

if (root == nullptr) {

cout << "Дерево пусто." << endl;

}

else {

cout << "Сумма листьев равна ";

cout << sumOfLeavesHelper(root);

cout << endl;

}

}

// Метод вывода среднего арифм. узлов

void findAverage() {

if (root == nullptr) {

cout << "Дерево пусто." << endl;

}

else {

cout << "Среднее арифметическое всех узлов равно ";

cout << findAverageHelper(root);

cout << endl;

}

}

// Метод вывода дерева

void printTree()

{

if (root == nullptr)

cout << "Дерево пусто." << endl;

else {

cout << "Дерево:" << endl;

printHelper(root, "", true);

}

}

};

**Результаты тестирования**

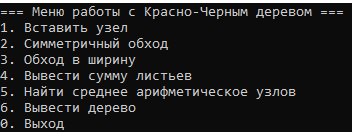


Рисунок 1 – Внешний вид пользовательского меню

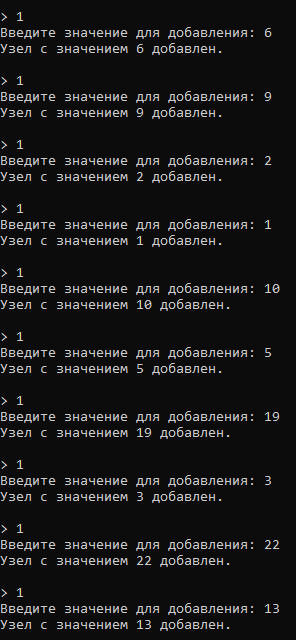


Рисунок 2 – Заполнение дерева 10-ю записями

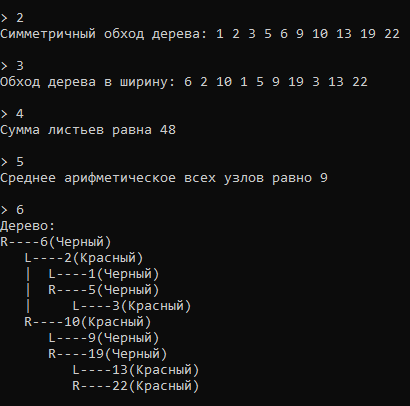


Рисунок 3 – Работа всех требуемых методов

ВЫВОДЫ

Тщательное исследование проблемы позволило найти оптимальное решение для организации работы с деревом поиска. Использование структуры данных, такой как двоичное дерево, позволило эффективно хранить и обрабатывать данные, обеспечив быструю вставку, удаление и поиск элементов. Двоичные деревья поиска позволяют поддерживать баланс между временем выполнения операций и использованием памяти.

Применение алгоритмов балансировки, таких как вращения, улучшило производительность программы, обеспечив сохранение свойств дерева и эффективную работу с ним. Это привело к значительному сокращению времени выполнения операций вставки и поиска, что напрямую влияет на общую скорость работы программы.

Программа отличается простотой и надежностью. Простота реализации позволяет легко вносить изменения и улучшения, а также облегчает поддержку системы в дальнейшем. Эффективность программы гарантирована за счет правильного выбора структуры данных и использования алгоритмов, соответствующих задаче.

ИНФОРМАЦИОННЫЕ ИСТОЧНИКИ

1. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд., 2016.

2. Документация по языку С++ [Электронный ресурс]. URL: https://docs.microsoft.com/ruru/cpp/cpp/ (дата обращения 01.09.2021).

3. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 01.09.2021).