|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | МИНОБРНАУКИ РОССИИ | | | | Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |   Институт Информационных технологий | |
|  | |
| Кафедра Математического обеспечения и стандартизации информационных технологий | |
|  |  |
|  |  |

|  |  |
| --- | --- |
| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 7.1** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Балансировка дерева поиска»** | |
|  | |
| Выполнил студент группы ИНБО-12-23 | Албахтин И.В. |
| Принял ассистент | Муравьёва Е.А. |

Москва 2024

**Оглавление**

[1. Цель работы 3](#_Toc181783644)

[2. Задание №1 3](#_Toc181783645)

[2.1 Постановка задачи 3](#_Toc181783646)

[3. Задание №2 11](#_Toc181783647)

[3.1 Постановка задачи 11](#_Toc181783648)

[3.2 Описание алгоритмов операции 12](#_Toc181783649)

[3.3 Код программы 17](#_Toc181783650)

[3.4 Тестирование программы 22](#_Toc181783651)

[4. Задание №3 26](#_Toc181783652)

[4.1 Постановка задачи 26](#_Toc181783653)

[4.2 Код программы 26](#_Toc181783654)

[4.3 Тестирование программы 31](#_Toc181783655)

[5. Вывод 33](#_Toc181783656)

# **Цель работы**

Получение умений и навыков разработки и реализаций операций над структурой данных бинарное дерево.

# **Задание №1**

## **2.1 Постановка задачи**

Ответить на вопросы:

1. Что определяет степень дерева?

- Степень дерева – максимальная степень его узлов

2. Какова степень сильноветвящегося дерева?

- Степень сильно ветвящегося дерева — это максимальная степень его узлов. Обычно такая степень превышает 2. Степень сильноветвящегося дерева произвольная.

3. Что определяет путь в дереве?

- Путь в дереве – последовательность узлов от корня до узла.

4. Как рассчитать длину пути в дереве?

-Длина пути в дереве – сумма длин всех его рёбер. Длина внутреннего пути может быть определена по следующей рекурсивной формуле: Длина внутреннего пути = Длина внутреннего пути в левом поддереве + длина внутреннего пути в правом поддереве + количество узлов в дереве – 1.

5. Какова степень бинарного дерева?

- Степень бинарного дерева всегда равна 2, так как у каждого узла может быть не более двух дочерних узлов..

6. Может ли дерево быть пустым?

- Да, дерево может быть пустым, что означает, что в нём нет ни одного узла (корень равен NULL). Бинарное (двоичное) дерево поиска может быть либо пустым, либо оно обладает таким свойством, что корневой элемент имеет большее значение узла, чем любой элемент в левом поддереве, и меньшее или равное, чем элементы в правом поддереве.

7. Дайте определение бинарного дерева?

- Бинарное дерево (Двоичное дерево) — иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками.

8. Дайте определение алгоритму обхода.

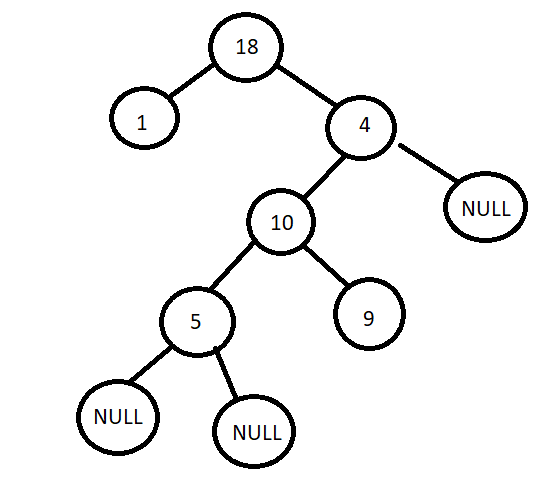
- Алгоритм обхода дерева — это метод посещения всех узлов дерева в определённом порядке, например, в глубину или в ширину. обход графа — это переход от одной его вершины к другой в поисках свойств связей этих вершин. Связи (линии, соединяющие вершины) называются направлениями, путями, гранями или ребрами графа. Вершины графа также именуются узлами.  
Двумя основными алгоритмами обхода графа являются поиск в глубину (Depth-First Search, DFS) и поиск в ширину (Breadth-First Search, BFS).

9. Что такое высота дерева? Приведите рекуррентную зависимость для вычисления высоты дерева.

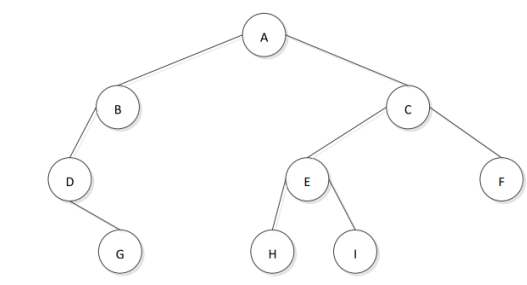
- Высота дерева – это количество рёбер между корнем и максимальным уровнем. Рекуррентная зависимость:

Высота дерева = 1 + max(высота левого поддерева, высота правого поддерева).

10. Изобразите бинарное дерево, корень которого имеет индекс 6, и которое представлено в памяти таблицей вида



11. Укажите путь обхода дерева по алгоритмы: прямой, обратный, симметричный



- Прямой порядок – от корня к левой ветви, затем к правой: A B D G C E H I F

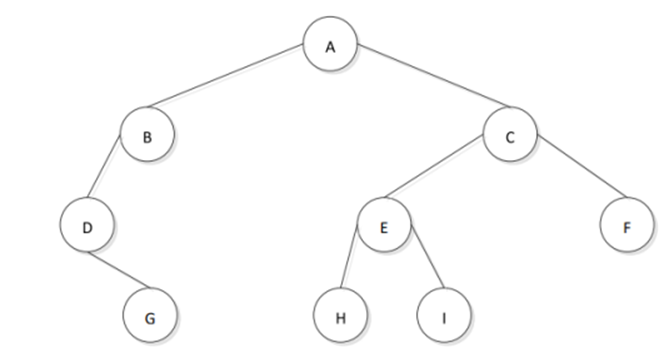
Обратный порядок – проходится левая ветвь, затем правая, затем корень: G D B H I E F C A

Симметричный порядок – дерево проходится, начиная с левой ветви вверх к корню, затем к правой ветви: D G B A H E I C F

12. Какая структура используется в алгоритме обхода дерева методом в «ширину»?

- Для обхода дерева методом в ширину используется очередь. Обход в ширину подразумевает, что сначала мы посещаем корень, затем, слева направо, все ветви первого уровня, затем все ветви второго уровня и т.д. Пусть мы находимся в корне дерева. Далее необходимо посетить всех наследников корня. Таким образом, нужно засунуть в контейнер сначала узел, затем его наследников, при этом узел далее должен быть обработан первым. То есть, элемент, который вошёл первым должен быть обработан первым.

13. Выведите путь при обходе дерева в «ширину». Продемонстрируйте использование структуры при обходе дерева.

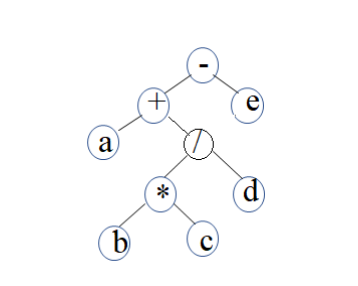


Путь при обходе дерева в «ширину»: A B C D E F G H I

14. Какая структура используется в не рекурсивном обходе дерева методом в «глубину»?

- Для не рекурсивного обхода дерева методом в глубину используется стек. Поиск по глубине легко реализуется с помощью стека, в том числе рекурсивно (через стек вызовов), в то время как поиск по ширине легко реализуется с помощью очереди, в том числе и рекурсивно.

15. Выполните прямой, симметричный, обратный методы обхода дерева выражений.



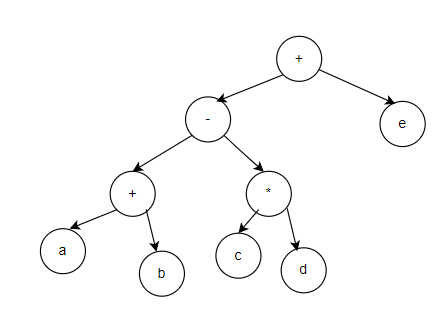
Прямой метод обхода: -+a/\*bcde

Симметричный метод обхода: a+b\*c/d-e

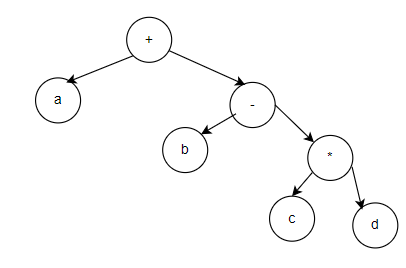
Обратный метод обхода: abc\*d/+e-

16. Для каждого заданного арифметического выражения постройте бинарное дерево выражений:

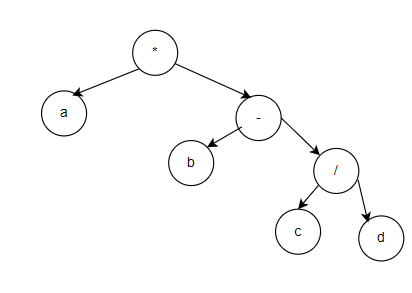
1) a+b-c\*d+e,



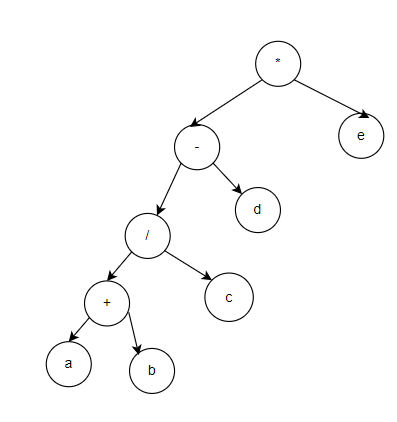
2) /a-b\*c d



3) a b c d / - \*



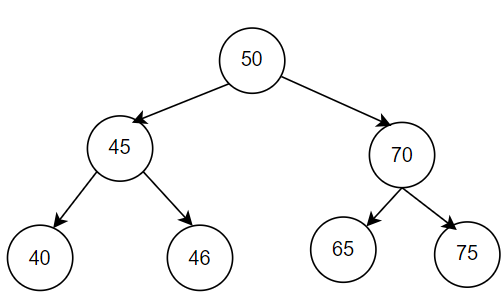
4) \* - / + a b c d e



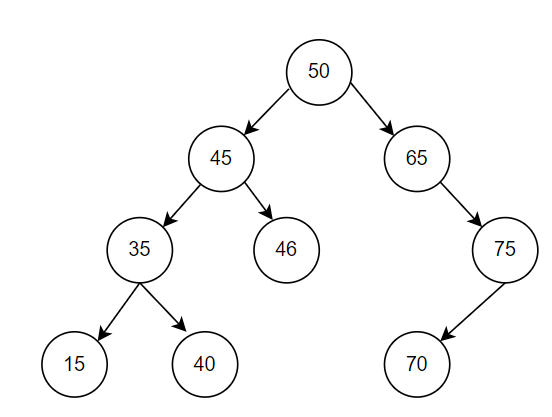
17. В каком порядке будет проходиться бинарное дерево, если алгоритм обхода в ширину будет запоминать узлы не в очереди, а в стеке?

- Если использовать стек для обхода в ширину, порядок будет аналогичен обратному обходу в глубину. Если заменить очередь стеком при реализации поиска в ширину, вместо этого вы будете выполнять поиск в глубину. Поиск в ширину найдет кратчайший путь в невзвешенном графике, когда он впервые встретит искомый элемент. Если ребра имеют веса, то вам нужно будет либо выполнить исчерпывающий поиск, чтобы найти все пути к элементу, либо использовать вместо этого алгоритм, подобный алгоритму Джикстры. Поиск в глубину не обязательно найдет кратчайший путь, когда он впервые найдет искомый элемент. Поэтому нужно будет выполнить поиск по всем путям.

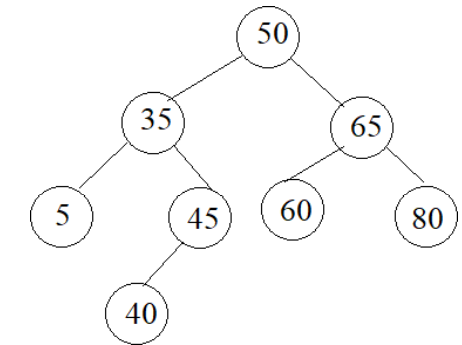
18. Постройте бинарное дерево поиска, которое в результате симметричного обхода дало бы следующую последовательность узлов: 40 45 46 50 65 70 75

- 

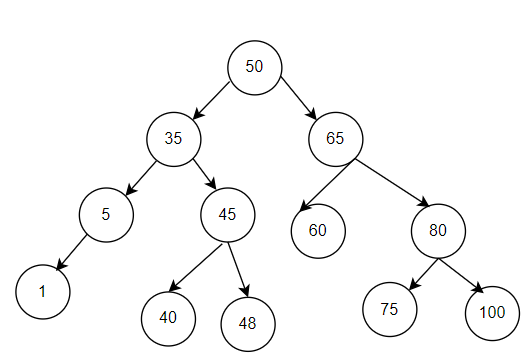
19. Приведенная ниже последовательность получена путем прямого обхода бинарного дерева поиска: 50 45 35 15 40 46 65 75 70. Постройте это дерево.



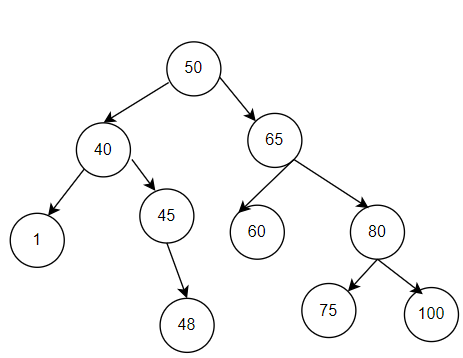
20. Дано следующее бинарное дерево поиска



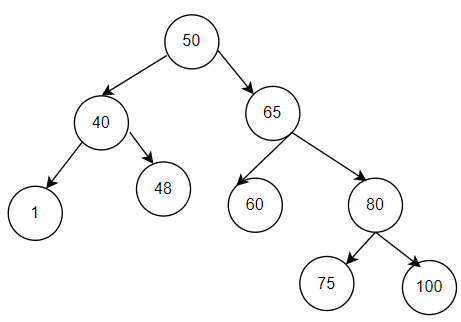
1. после включения узлов 1, 48, 75, 100



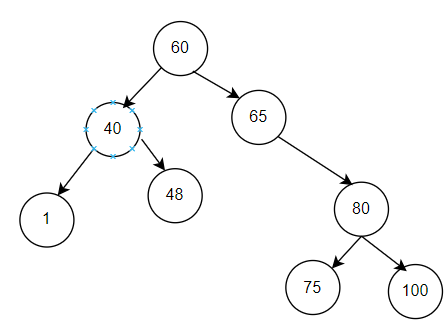
1. после удаления узлов 5, 35



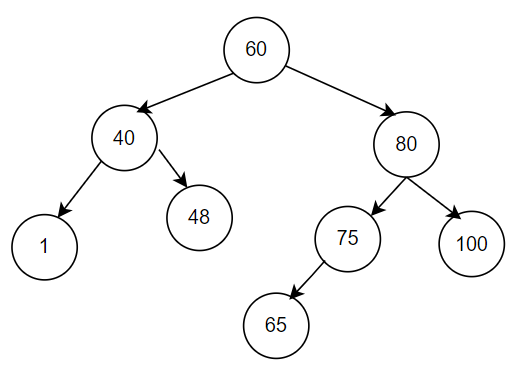
1. после удаления узла 45



1. после удаления узла 50



1. после удаления узла 65 и вставки его снова



# **Задание №2**

## **3.1 Постановка задачи**

Разработать программу бинарного дерева поиска и выполнить операции в соответствии с требованиями варианта. Варианты заданий представлены в таблице 1. Методы, которые должны быть реализованы в независимости от варианта: • включение элемента в дерево; • поиск ключа в дереве; • удаление ключа из дерева; • отображение дерева. Совет. Выполните реализацию средствами ООП, операции будут методами класса.

Таблица 1. Варианты заданий

|  |  |  |
| --- | --- | --- |
| № | Значение информационной части | Операции варианта |
| 3 | Символьное значение | Вернуть самый левый узел дерева  Определить длину пути (количество ребер) от  корня до ближайшего узла с заданным значением  Найти максимальное значение среди значений листьев  дерева. |

## 

## **3.2 Описание алгоритмов операции**

Программа состоит из двух классов:

* Node: представляет узел бинарного дерева поиска, хранящий значение ключа и ссылки на левое и правое поддеревья.
* BinarySearchTree: содержит методы для работы с бинарным деревом поиска, такие как добавление элемента, поиск элемента, поиск минимального элемента, удаление элемента и другие операции.

1. Класс Node

|  |
| --- |
| class Node {  public:      int key;      Node\* left;      Node\* right;      Node(int value) : key(value), left(nullptr), right(nullptr) {}  }; |

**Назначение**: Класс Node представляет узел бинарного дерева поиска. Каждый узел имеет:

* key — значение узла.
* left и right — указатели на левого и правого потомков (по умолчанию равны nullptr).

2. Класс BinarySearchTree

|  |
| --- |
| class BinarySearchTree {  private:      Node\* root; |

**Назначение**: root хранит указатель на корневой узел дерева. Это начальная точка для всех операций, связанных с обходом и модификацией дерева.

3. Включение элемента в дерево (метод insert)

|  |
| --- |
| Node\* insert(Node\* node, int key) {      if (node == nullptr) return new Node(key);      if (key < node->key)          node->left = insert(node->left, key);      else if (key > node->key)        node->right = insert(node->right, key);      return node;  } |

**Алгоритм**:

1. Если узел пустой (node == nullptr), создаём новый узел с заданным key и возвращаем его.
2. Если значение key меньше значения текущего узла, рекурсивно вставляем его в левое поддерево.
3. Если значение key больше, рекурсивно вставляем его в правое поддерево.
4. После вставки возвращаем текущий узел для связи дерева.

**Использование**: Метод позволяет вставить элемент в нужное место в соответствии с правилами бинарного дерева поиска.

4. Поиск элемента в дереве (метод search)

|  |
| --- |
| Node\* search(Node\* node, int key) {      if (node == nullptr || node->key == key) return node;      if (key < node->key)          return search(node->left, key);      else          return search(node->right, key);  } |

**Алгоритм**:

1. Если узел пустой (node == nullptr) или значение key совпадает со значением узла (node->key == key), значит или элемент не найден, или найден, возвращаем node.
2. Если key меньше значения узла, рекурсивно ищем в левом поддереве.
3. Если key больше, рекурсивно ищем в правом поддереве.

**Использование**: Метод проверяет наличие элемента в дереве, возвращая node.

5. Удаление элемента из дерева (метод deleteNode)

|  |
| --- |
| Node\* deleteNode(Node\* node, int key) {      if (node == nullptr) return node;      if (key < node->key)          node->left = deleteNode(node->left, key);      else if (key > node->key)          node->right = deleteNode(node->right, key);      else {          if (node->left == nullptr) {              Node\* temp = node->right;              delete node;              return temp;          } else if (node->right == nullptr) {              Node\* temp = node->left;              delete node;              return temp;          }          Node\* temp = findMin(node->right);          node->key = temp->key;          node->right = deleteNode(node->right, temp->key);      }      return node;  } |

**Алгоритм**:

1. Если узел пустой (node == nullptr), элемент не найден, возвращаем node.
2. Если key меньше значения узла, рекурсивно удаляем в левом поддереве.
3. Если key больше, удаляем в правом поддереве.
4. Если key совпадает с текущим узлом:
   * Если у узла нет левого потомка, удаляем его, заменяя правым поддеревом.
   * Если нет правого потомка, удаляем его, заменяя левым поддеревом.
   * Если у узла есть оба потомка, ищем минимальный узел в правом поддереве, копируем его значение и удаляем его из правого поддерева.

**Использование**: Метод удаляет элемент, перестраивая дерево по правилам бинарного поиска.

6. Нахождение минимального узла (метод minValueNode)

|  |
| --- |
| Node\* findMin(Node\* node) {      while (node && node->left != nullptr)          node = node->left;      return node;  } |

**Алгоритм**:

1. Начинаем с текущего узла и идём по левым потомкам до тех пор, пока не найдём самый левый (самый маленький) узел.

**Использование**: Метод используется для нахождения минимального элемента при удалении узла с двумя потомками.

7. Возврат самого левого листа (метод findLeftmost)

|  |
| --- |
| Node\* findLeftmost(Node\* node) {      if (node == nullptr) return nullptr;      while (node->left != nullptr)          node = node->left;      return node;  } |

**Алгоритм**:

1. Если узел пустой, возвращаем nullptr.
2. Пока у узла есть левый потомок, переходим к нему.
3. Возвращаем node

**Использование**: Метод возвращает крайний левый лист в дереве.

8. Подсчёт длины пути от корня до ближайшего узла с заданным значением (метод pathLength)

|  |
| --- |
| int pathLength(Node\* node, int key, int length = 0) {      if (node == nullptr) return -1;      if (node->key == key) return length;      if (key < node->key)          return pathLength(node->left, key, length++);      else          return pathLength(node->right, key, length++);  } |

**Алгоритм**:

1. Если узел пустой, возвращаем -1.
2. Если значение key совпадает со значением узла (node->key == key), возвращаем length.
3. Если значение узла меньше значения key, рекурсивно ищем длину пути по левому поддереву.
4. Если значение узла больше значения key, рекурсивно ищем длину пути по правому поддереву.

**Использование**: Метод подсчитывает длину пути до ближайшего узла с заданным значением.

9. Нахождение максимального значения среди значений листьев дерева (метод maxLeafValue)

|  |
| --- |
| int maxLeafValue(Node\* node) {      if (node == nullptr) return INT\_MIN;      if (node->left == nullptr && node->right == nullptr) return node->key;      return max(maxLeafValue(node->left), maxLeafValue(node->right));  } |

**Алгоритм**:

1. Если узел пустой, возвращаем INT\_MIN.
2. Если указатели на левый и правый узлы пустые, возвращаем текущий значение key.
3. Рекурсивный возврат максимального из максимального значения левого и правого потомков.

**Использование**: Метод находит максимальное значение среди значений листьев дерева.

10. Отображение дерева (метод inorder)

|  |
| --- |
| void inorder(Node\* node) {      if (node != nullptr) {          inorder(node->left);          cout << node->key << " ";          inorder(node->right);      }  } |

**Алгоритм**:

* 1. Рекурсивно обходим левое поддерево.
  2. Выводим значение узла.
  3. Рекурсивно обходим правое поддерево.

**Использование**: Метод выводит элементы в порядке возрастания.

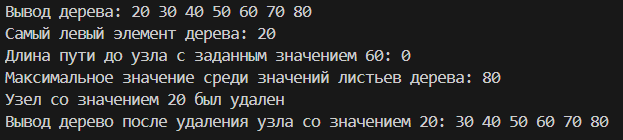
## **3.3 Код программы**

*Листинг 1 – Исходный код программы для Задания 2*

|  |
| --- |
| #include <iostream>  #include <algorithm>  #include <Windows.h>  using namespace std;  class Node {  public:      int key;      Node\* left;      Node\* right;      Node(int value) : key(value), left(nullptr), right(nullptr) {}  };  class BinarySearchTree {  private:      Node\* root;      Node\* insert(Node\* node, int key) {          if (node == nullptr) return new Node(key);          if (key < node->key)              node->left = insert(node->left, key);          else if (key > node->key)              node->right = insert(node->right, key);          return node;      }      Node\* search(Node\* node, int key) {          if (node == nullptr || node->key == key) return node;          if (key < node->key)              return search(node->left, key);          else              return search(node->right, key);      }      Node\* findMin(Node\* node) {          while (node && node->left != nullptr)              node = node->left;          return node;      }      Node\* deleteNode(Node\* node, int key) {          if (node == nullptr) return node;          if (key < node->key)              node->left = deleteNode(node->left, key);          else if (key > node->key)              node->right = deleteNode(node->right, key);          else {              if (node->left == nullptr) {                  Node\* temp = node->right;                  delete node;                  return temp;              } else if (node->right == nullptr) {                  Node\* temp = node->left;                  delete node;                  return temp;              }              Node\* temp = findMin(node->right);              node->key = temp->key;              node->right = deleteNode(node->right, temp->key);          }          return node;      }      void inorder(Node\* node) {          if (node != nullptr) {              inorder(node->left);              cout << node->key << " ";              inorder(node->right);          }      }      Node\* findLeftmost(Node\* node) {          if (node == nullptr) return nullptr;          while (node->left != nullptr)              node = node->left;          return node;      }      int pathLength(Node\* node, int key, int length = 0) {          if (node == nullptr) return -1;          if (node->key == key) return length;          if (key < node->key)              return pathLength(node->left, key, length++);          else              return pathLength(node->right, key, length++);      }      int maxLeafValue(Node\* node) {          if (node == nullptr) return INT\_MIN;          if (node->left == nullptr && node->right == nullptr) return node->key;          return max(maxLeafValue(node->left), maxLeafValue(node->right));      }  public:      BinarySearchTree() : root(nullptr) {}      void insert(int key) {          root = insert(root, key);      }      bool search(int key) {          return search(root, key) != nullptr;      }      void deleteNode(int key) {          root = deleteNode(root, key);      }      void display() {          inorder(root);          cout << endl;      }      int getLeftmostValue() {          Node\* leftmost = findLeftmost(root);          return (leftmost != nullptr) ? leftmost->key : -1;      }      int pathLengthTo(int key) {          return pathLength(root, key);      }      int findMaxLeafValue() {          return maxLeafValue(root);      }  };  int main() {      SetConsoleCP(1251);      SetConsoleOutputCP(1251);      BinarySearchTree bst;      bst.insert(50);      bst.insert(30);      bst.insert(20);      bst.insert(40);      bst.insert(70);      bst.insert(60);      bst.insert(80);      cout << "Вывод дерева: ";      bst.display();      cout << "Самый левый элемент дерева: " << bst.getLeftmostValue() << endl;      int searchKey = 60;      cout << "Длина пути до узла с заданным значением " << searchKey << ": " << bst.pathLengthTo(searchKey) << endl;      cout << "Максимальное значение среди значений листьев дерева: " << bst.findMaxLeafValue() << endl;      int key = 20;      bst.deleteNode(key);      cout << "Узел со значением " << key << " был удален" << endl;      cout << "Вывод дерево после удаления узла со значением 20: ";      bst.display();      return 0;  } |

## 

## **3.4 Тестирование программы**



**Рисунок 1. Тестирование всех методов задания №2**

# **Задание №3**

## **4.1 Постановка задачи**

Разработать приложение, которое использует сбалансированное дерево поиска (СДП), и выполнить операции в соответствии с требованиями варианта.

Методы, которые должны быть реализованы в независимости от варианта:

• включение элемента в дерево;

• удаление ключа из дерева;

• поиск ключа в дереве с возвратом записи из файла;

• вывод дерева в форме дерева (с отображением структуры дерева)

Таблица 2. Варианты заданий

|  |  |  |  |
| --- | --- | --- | --- |
| Вариант | Тип дерева | Значение информационной части | Операции варианта |
| 3 | Красно-черное дерево | Символьное значение | Найти среднее арифметическое всех узлов.  Найти длину пути от корня до заданного значения.  Вывести все узлы дерева, используя прямой обход. |

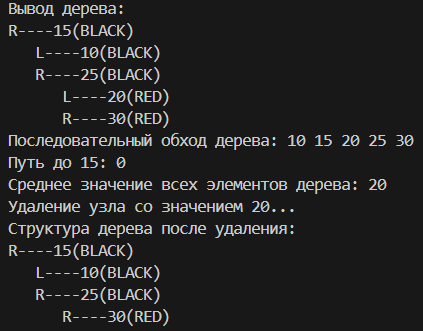
## 

## **4.2 Код программы**

*Листинг 3 – Исходный код программы для Задания 3*

|  |
| --- |
| #include <iostream>  #include <sstream>  #include <fstream>  #include <string>  #include <Windows.h>  using namespace std;  enum Color { RED, BLACK };  struct Node {  string value;  Color color;  Node \*left, \*right, \*parent;  Node(string value) : value(value), color(RED), left(nullptr), right(nullptr), parent(nullptr) {}  };  class RedBlackTree {  private:  Node\* root;  void rotateLeft(Node\* x);  void rotateRight(Node\* x);  void fixInsert(Node\* x);  void fixDelete(Node\* x);  Node\* deleteNode(Node\* root, string value);  Node\* searchTree(Node\* node, const string& val);  void inorderTraversal(Node\* node);  int sumValues(Node\* node);  int countNodes(Node\* node);  int pathLength(Node\* node, const string& val, int path);  void printTree(Node\* node, string indent, bool last);  Node\* minimum(Node\* node);  public:  RedBlackTree() : root(nullptr) {}  void insert(const string& value);  void remove(const string& value);  bool search(const string& val);  void inorder();  double averageValue();  int pathToValue(const string& val);  void printTree();  };  // Левый поворот  void RedBlackTree::rotateLeft(Node\* x) {  Node\* y = x->right;  x->right = y->left;  if (y->left != nullptr) {  y->left->parent = x;  }  y->parent = x->parent;  if (x->parent == nullptr) {  root = y;  } else if (x == x->parent->left) {  x->parent->left = y;  } else {  x->parent->right = y;  }  y->left = x;  x->parent = y;  }  // Правый поворот  void RedBlackTree::rotateRight(Node\* x) {  Node\* y = x->left;  x->left = y->right;  if (y->right != nullptr) {  y->right->parent = x;  }  y->parent = x->parent;  if (x->parent == nullptr) {  root = y;  } else if (x == x->parent->right) {  x->parent->right = y;  } else {  x->parent->left = y;  }  y->right = x;  x->parent = y;  }  // Восстановление баланса после вставки  void RedBlackTree::fixInsert(Node\* x) {  while (x != root && x->parent->color == RED) {  if (x->parent == x->parent->parent->left) {  Node\* y = x->parent->parent->right;  if (y && y->color == RED) {  x->parent->color = BLACK;  y->color = BLACK;  x->parent->parent->color = RED;  x = x->parent->parent;  } else {  if (x == x->parent->right) {  x = x->parent;  rotateLeft(x);  }  x->parent->color = BLACK;  x->parent->parent->color = RED;  rotateRight(x->parent->parent);  }  } else {  Node\* y = x->parent->parent->left;  if (y && y->color == RED) {  x->parent->color = BLACK;  y->color = BLACK;  x->parent->parent->color = RED;  x = x->parent->parent;  } else {  if (x == x->parent->left) {  x = x->parent;  rotateRight(x);  }  x->parent->color = BLACK;  x->parent->parent->color = RED;  rotateLeft(x->parent->parent);  }  }  }  root->color = BLACK;  }  // Вставка узла  void RedBlackTree::insert(const string& value) {  Node\* node = new Node(value);  Node\* y = nullptr;  Node\* x = root;  while (x != nullptr) {  y = x;  if (node->value < x->value)  x = x->left;  else  x = x->right;  }  node->parent = y;  if (y == nullptr) {  root = node;  } else if (node->value < y->value) {  y->left = node;  } else {  y->right = node;  }  if (node->parent == nullptr) {  node->color = BLACK;  return;  }  if (node->parent->parent == nullptr)  return;  fixInsert(node);  }  // Восстановление баланса после удаления  void RedBlackTree::fixDelete(Node\* x) {  while (x != root && (!x || x->color == BLACK)) {  if (x == x->parent->left) {  Node\* w = x->parent->right;  if (w->color == RED) {  w->color = BLACK;  x->parent->color = RED;  rotateLeft(x->parent);  w = x->parent->right;  }  if ((!w->left || w->left->color == BLACK) && (!w->right || w->right->color == BLACK)) {  w->color = RED;  x = x->parent;  } else {  if (!w->right || w->right->color == BLACK) {  if (w->left)  w->left->color = BLACK;  w->color = RED;  rotateRight(w);  w = x->parent->right;  }  w->color = x->parent->color;  x->parent->color = BLACK;  if (w->right)  w->right->color = BLACK;  rotateLeft(x->parent);  x = root;  }  } else {  Node\* w = x->parent->left;  if (w->color == RED) {  w->color = BLACK;  x->parent->color = RED;  rotateRight(x->parent);  w = x->parent->left;  }  if ((!w->right || w->right->color == BLACK) && (!w->left || w->left->color == BLACK)) {  w->color = RED;  x = x->parent;  } else {  if (!w->left || w->left->color == BLACK) {  if (w->right)  w->right->color = BLACK;  w->color = RED;  rotateLeft(w);  w = x->parent->left;  }  w->color = x->parent->color;  x->parent->color = BLACK;  if (w->left)  w->left->color = BLACK;  rotateRight(x->parent);  x = root;  }  }  }  if (x)  x->color = BLACK;  }  // Удаление узла  void RedBlackTree::remove(const string& value) {  Node\* z = searchTree(root, value);  if (!z) return;  Node\* y = z;  Node\* x;  Color yOriginalColor = y->color;  if (z->left == nullptr) {  x = z->right;  if (x) x->parent = z->parent;  if (z->parent == nullptr) {  root = x;  } else if (z == z->parent->left) {  z->parent->left = x;  } else {  z->parent->right = x;  }  } else if (z->right == nullptr) {  x = z->left;  if (x) x->parent = z->parent;  if (z->parent == nullptr) {  root = x;  } else if (z == z->parent->left) {  z->parent->left = x;  } else {  z->parent->right = x;  }  } else {  y = minimum(z->right);  yOriginalColor = y->color;  x = y->right;  if (y->parent == z) {  if (x) x->parent = y;  } else {  if (x) x->parent = y->parent;  y->parent->left = x;  y->right = z->right;  z->right->parent = y;  }  if (z->parent == nullptr) {  root = y;  } else if (z == z->parent->left) {  z->parent->left = y;  } else {  z->parent->right = y;  }  y->parent = z->parent;  y->color = z->color;  y->left = z->left;  z->left->parent = y;  }  delete z;  if (yOriginalColor == BLACK && x) fixDelete(x);  }  Node\* RedBlackTree::searchTree(Node\* node, const string& val) {  if (node == nullptr || node->value == val) {  return node;  }  if (val < node->value) {  return searchTree(node->left, val);  }  return searchTree(node->right, val);  }  bool RedBlackTree::search(const string& val) {  return searchTree(root, val) != nullptr;  }  int RedBlackTree::pathToValue(const string& val) {  return pathLength(root, val, 0);  }  int RedBlackTree::pathLength(Node\* node, const string& val, int path) {  if (!node) return -1;  if (node->value == val) return path;  int leftPath = pathLength(node->left, val, path + 1);  if (leftPath != -1) return leftPath;  return pathLength(node->right, val, path + 1);  }  void RedBlackTree::inorderTraversal(Node\* node) {  if (!node) return;  inorderTraversal(node->left);  cout << node->value << " ";  inorderTraversal(node->right);  }  void RedBlackTree::inorder() {  inorderTraversal(root);  cout << endl;  }  int RedBlackTree::sumValues(Node\* node) {  if (!node) return 0;  return stoi(node->value) + sumValues(node->left) + sumValues(node->right);  }  int RedBlackTree::countNodes(Node\* node) {  if (!node) return 0;  return 1 + countNodes(node->left) + countNodes(node->right);  }  double RedBlackTree::averageValue() {  int sum = sumValues(root);  int count = countNodes(root);  return count ? static\_cast<double>(sum) / count : 0.0;  }  void RedBlackTree::printTree(Node\* node, string indent, bool last) {  if (node != nullptr) {  cout << indent;  if (last) {  cout << "R----";  indent += " ";  } else {  cout << "L----";  indent += "| ";  }  cout << node->value << "(" << (node->color == RED ? "RED" : "BLACK") << ")" << endl;  printTree(node->left, indent, false);  printTree(node->right, indent, true);  }  }  void RedBlackTree::printTree() {  printTree(root, "", true);  }  Node\* RedBlackTree::minimum(Node\* node) {  while (node->left != nullptr) {  node = node->left;  }  return node;  }  int main() {  SetConsoleCP(1251);  SetConsoleOutputCP(1251);  RedBlackTree tree;  tree.insert("10");  tree.insert("20");  tree.insert("15");  tree.insert("30");  tree.insert("25");    cout << "Вывод дерева:" << endl;  tree.printTree();  cout << "Последовательный обход дерева: ";  tree.inorder();  string value = "15";  cout << "Путь до " << value << ": " << tree.pathToValue(value) << endl;  cout << "Среднее значение всех элементов дерева: " << tree.averageValue() << endl;  cout << "Удаление узла со значением 20..." << endl;  tree.remove("20");  cout << "Структура дерева после удаления:" << endl;  tree.printTree();  return 0;  } |

## **4.3 Тестирование программы**



**Рисунок 2. Тестирование всех методов задания №3**

# **5. Вывод**

В ходе выполнения задания я разработал приложение, использующее сбалансированное красно-чёрное дерево поиска для эффективного хранения и управления данными. Я создал структуру узлов дерева, обеспечивающую возможность быстрого доступа и балансировки при изменении данных. Были реализованы основные операции, такие как вставка, удаление и поиск ключей, а также методы для вывода дерева и поддержания его структуры. В процессе тестирования приложение было проверено на различных сценариях, включая крайние случаи, чтобы убедиться в корректности работы балансировки. Результаты тестирования показали устойчивое время выполнения операций, что подтвердило эффективность применения красно-чёрного дерева для управления данными с минимальным временем доступа.