|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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 № 4** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Нелинейные структуры данных. Бинарное дерево»** | |
|  | |
| Выполнил студент группы ИНБО-03-22 | Алтухов А.В. |
| Принял преподаватель | Филатов А.С. |

|  |  |  |
| --- | --- | --- |
| Лабораторная работа выполнена | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись студента)* |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись руководителя)* |

Москва 2023

# **Цель работы**

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

# **Постановка задачи**

Разработать программу в соответствии с требованиями варианта. Выполнить реализацию средствами ООП.

**Для вариантов с 1 по 7**

Вид дерева: идеально сбалансированное из n узлов (не AVL).

1. Реализовать операции общие для вариантов с 1 по 7
   1. Создать идеально сбалансированное бинарное дерево из n узлов. Структура узла дерева включает: информационная часть узла, указатель на левое и указатель на правое поддерево. Информационная часть узла определена вариантом
   2. Отобразить дерево на экране, повернув его на экране против часовой стрелки.
2. Реализовать операции варианта.
3. Разработать программу, демонстрирующую выполнение всех операций.
4. Составить отчет, отобразив в нем описание выполнения всех этапов разработки, тестирования и код всей программы со скриншотами результатов тестирования.

Вариант №2. Условие задания:

|  |  |
| --- | --- |
| Значение информационной части | Операции варианта |
| Целое число | 1. Определить количество листьев с положительными значениями. 2. Определить, сколько узлов в дереве содержат заданное число. 3. Увеличить значение узлов вдвое, обходя дерево алгоритмом в ширину |

# **Решение задачи**

Идеально сбалансированное бинарное дерево из n узлов — это бинарное дерево поиска, в котором все n элементов упорядочены таким образом, что высота дерева является оптимальной. Для достижения этой оптимальной структуры, узлы должны быть распределены равномерно и сбалансированно по всем уровням дерева.

Самый оптимальный случай для идеально сбалансированного бинарного дерева достигается, когда количество узлов n является степенью двойки (n = 2^k), где k - натуральное число. В таком случае, каждый уровень дерева будет полностью заполнен, и высота дерева будет минимальной.

Каждый элемент дерева – это узел, который я решил описать классом Node.

|  |
| --- |
| class Node {  public:  Node\* parent = nullptr;  Node\* left = nullptr;  Node\* right = nullptr;  int value;  Node(int value) {  this->value = value;  }  }; |

Чтобы управлять деревом был создан класс Tree, внутри которого будут созданы методы для управления. Они будут описаны ниже.

|  |
| --- |
| class Tree {  private:  Node\* root;  public:  Tree() {  this->root = nullptr;  }  Tree(int value) {  root = new Node(value);  }  } |

Для того, чтобы вывести на экран идеально сбалансированное бинарное дерево поиска, перевернутое против часов стрелки, был написан приватный метод printTreeRec, который получает ссылку на корень дерева, текущий уровень узла и символ, который будет отображаться перед значением каждого узла. Функция ничего не возвращает

|  |
| --- |
| void printTreeRec(Node\* root, int level, char prefix) {  if (root != nullptr) {  printTreeRec(root->right, level + 1, '/');  for (int i = 0; i < level; i++) {  cout << " ";  }  cout << prefix << "--" << root->value << endl;  printTreeRec(root->left, level + 1, '\\');  }  } |

Чтобы не озадачивать рядового пользователя вводом разный параметров и он мог в основной программе вывести дерево на экран, был написан открытый метод printTree.

|  |
| --- |
| void printTree() {  printTreeRec(root, 0, ' ');  } |

Чтобы найти место для нового узла дерева был написан закрытый рекурсивный метод insertRec, который получает ссылку на текущий узел и значение для информационной части нового узла.

|  |
| --- |
| void insertRec(Node\* currentNode, int value) {  if (value < currentNode->value) {  if (!currentNode->left) {  currentNode->left = new Node(value);  currentNode->left->parent = currentNode;  }  else {  insertRec(currentNode->left, value);  }  }  else if (value > currentNode->value) {  if (!currentNode->right) {  currentNode->right = new Node(value);  currentNode->right->parent = currentNode;  }  else {  insertRec(currentNode->right, value);  }  }  //если значение равно, новый узел не создается  } |

Но поскольку предыдущий метод является закрытым, нам нужно создать открытый метод для рядового пользователя. Для этого был создан открытый метод insert, который получает значение для информационной части нового узла и начинает поиск от корневого узла, если он есть, а если его нету, то создает его.

|  |
| --- |
| void insert(int value) {  if (!root) {  root = new Node(value);  }  else {  insertRec(root, value);  }  } |

Чтобы построить идеально сбалансированное дерево был написан закрытый метод buildBalancedTree, который на вход получает диапазон значений, для которого строится дерево. Функция возвращает указатель на корневой узел построенного дерева.

|  |
| --- |
| Node\* buildBalancedTree(int start, int end) {  if (start > end)  return nullptr;  int mid = (start + end) / 2;  Node\* newNode = new Node(mid);  newNode->left = buildBalancedTree(start, mid - 1);  if (newNode->left)  newNode->left->parent = newNode;  newNode->right = buildBalancedTree(mid + 1, end);  if (newNode->right)  newNode->right->parent = newNode;  return newNode;  } |

Но поскольку предыдущий метод является закрытым, нам нужно создать открытый метод для рядового пользователя. Для этого был создан открытый метод balancedTree, который получает на вход количество узлов.

|  |
| --- |
| void balancedTree(int n) {  root = buildBalancedTree(1, n);  } |

Чтобы найти количество положительных листьев в дереве был написан закрытый метод countPositiveLeavesRec, который получает ссылку на текущий узел, а возвращает целое число.

|  |
| --- |
| int countPositiveLeavesRec(Node\* currentNode) {  if (currentNode == nullptr) {  return 0;  }  if (currentNode->left == nullptr && currentNode->right == nullptr && currentNode->value > 0) {  return 1;  }  int leftCount = countPositiveLeavesRec(currentNode->left);  int rightCount = countPositiveLeavesRec(currentNode->right);  return leftCount + rightCount;  } |

Но поскольку предыдущий метод является закрытым, нам нужно создать открытый метод для рядового пользователя. Для этого был создан открытый метод countPositiveLeaves, который ничего не получает.

|  |
| --- |
| int countPositiveLeaves() {  return countPositiveLeavesRec(root);  } |

Чтобы определить сколько узлов дерева содержат заданное число был написан публичный метод getUzel, который а качестве аргумента получает значение, которое необходимо найти, а возвращает целое число.

|  |
| --- |
| int getUzel(int value) {  int res = 0;  Node\* current = root;  while (current) {  if (current->value == value)  return res;  else if (value < current->value) {  if (current->left)  current = current->left;  else break;  }  else {  if (current->right)  current = current->right;  else break;  }  res++;  }  return -1;  } |

Для того, чтобы увеличить значение всех узлов вдвое, был написан публичный метод doubleNodeValues, которому не требуются аргументы и он ничего не возвращает.

|  |
| --- |
| void doubleNodeValues() {  queue<Node\*> nodes;  if (!root) return;  nodes.push(root);  while (!nodes.empty()) {  Node\* current = nodes.front();  nodes.pop();  // Проверяем, что у текущего узла есть хотя бы один потомок  if (current->left || current->right) {  // Увеличиваем значение узла вдвое  current->value \*= 2;  }  if (current->left)  nodes.push(current->left);  if (current->right)  nodes.push(current->right);  }  } |

При запуске программы пользователю предоставляется автоматически созданное идеально сбалансированное дерево (рис. 1), после чего ему выводится список операций и предлагают выбрать, какую осуществить (рис. 2).

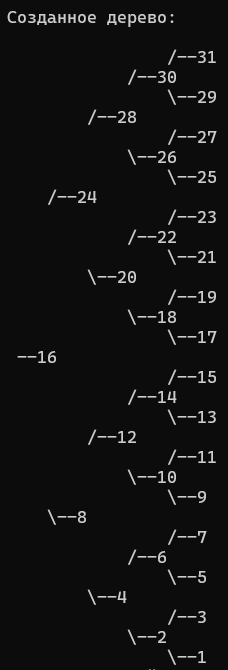
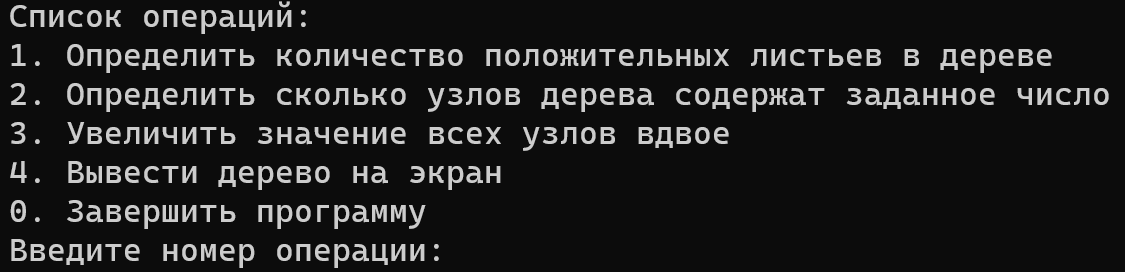


Рисунок 1. Меню. Выбор способа создания дерева

  
Рисунок 2. Меню. Список и выбор операции

# **Тестирование программы**

В следующих тестах будем иметь дело с готовым красивым деревом для удобства. Попробуем выбрать операцию, которая выводит дерево на экран (рис. 3).

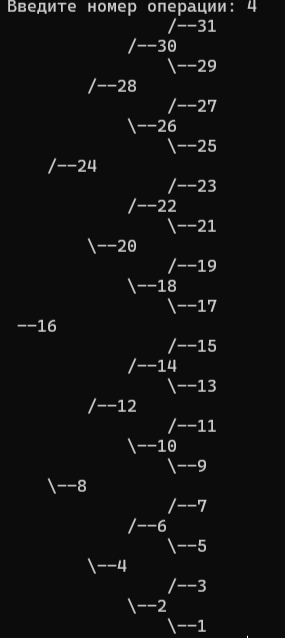


Рисунок 3. Вывод дерева на экран

Взглянем на дерево и посчитаем количество листьев и узнаем сколько узлов содержат в себе заданное число, например 1. Количество 16, а узлов, которые содержат в себе 1, равно 4. Проверим, как с этим справилась программа и вызовем сразу две операции – первую и вторую (рис. 4). Как мы можем видеть, программа справилась с расчетами.

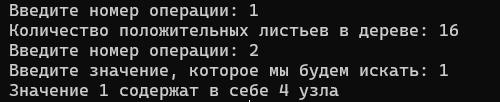


Рисунок 4. Вызов операций 1 и 2

Теперь попробуем умножить значения всех узлов вдвое (рис. 5). Действительно, значения всех узлов увеличилось вдвое.

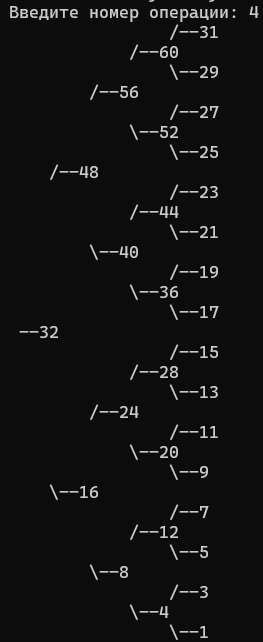


Рисунок 5. Увеличение значения всех узлов вдвое

Повторим 2 операцию для нового дерева, с удвоенными узлами (рис. 6). Будем находить количество узлов которые содержат в себе 4. До этого мы видели, что значение 4 было на 3 уровне и значит всего 2 узла содержали в себе это значение. После удвоения всех узлов значение 4 находится на 4 уровне, а это значит, что теперь 3 узла содержат в себе 4. Увидим, что это так, значит программа работает корректно.

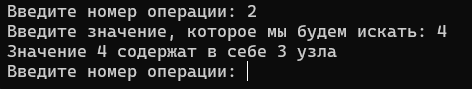


Рисунок 6. Вызов операции 2

Протестируем 4 операцию. Так как в нашем дереве нет отрицательных листьев, следовательно количество положительных листьев должно быть равно в целом количеству листьев. Операция выполняется корректно (рис. 7).

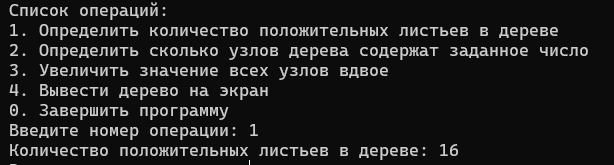


Рисунок 7. Определения количества положительных листьев

# **Вывод**

В результате выполнения практической работы были изучены сбалансированные бинарные деревья и были получены умения и навыки разработки и реализации операций над структурой данных, реализующих сбалансированное бинарное дерево. Была разработана программа в соответствии с требованиями варианта, с использованием средств ООП.

# **Исходный код программы**

|  |
| --- |
| #include <iostream>  #include <queue>  #include <cmath>  using namespace std;  class Node {  public:  Node\* parent = nullptr;  Node\* left = nullptr;  Node\* right = nullptr;  int value;  Node(int value) {  this->value = value;  }  };  class Tree {  private:  Node\* root;  public:  Tree() {  this->root = nullptr;  }  Tree(int value) {  root = new Node(value);  }  void printTree() {  printTreeRec(root, 0, ' ');  }  void insert(int value) {  if (!root) {  root = new Node(value);  }  else {  insertRec(root, value);  }  }  int countPositiveLeaves() {  return countPositiveLeavesRec(root);  }  void doubleNodeValues() {  queue<Node\*> nodes;  if (!root) return;  nodes.push(root);  while (!nodes.empty()) {  Node\* current = nodes.front();  nodes.pop();  if (current->left || current->right) {  current->value \*= 2;  }  if (current->left)  nodes.push(current->left);  if (current->right)  nodes.push(current->right);  }  }  int getUzel(int value) {  int res = 0;  Node\* current = root;  while (current) {  if (current->value == value)  return res;  else if (value < current->value) {  if (current->left)  current = current->left;  else  break;  }  else {  if (current->right)  current = current->right;  else  break;  }  res++;  }  return -1;  }  // Идеально сбалансированное дерево из n узлов  void balancedTree(int n) {  root = buildBalancedTree(1, n);  }  private:  void insertRec(Node\* currentNode, int value) {  if (value < currentNode->value) {  if (!currentNode->left) {  currentNode->left = new Node(value);  currentNode->left->parent = currentNode;  }  else {  insertRec(currentNode->left, value);  }  }  else if (value > currentNode->value) {  if (!currentNode->right) {  currentNode->right = new Node(value);  currentNode->right->parent = currentNode;  }  else {  insertRec(currentNode->right, value);  }  }  }  Node\* buildBalancedTree(int start, int end) {  if (start > end)  return nullptr;  int mid = (start + end) / 2;  Node\* newNode = new Node(mid);  newNode->left = buildBalancedTree(start, mid - 1);  if (newNode->left)  newNode->left->parent = newNode;  newNode->right = buildBalancedTree(mid + 1, end);  if (newNode->right)  newNode->right->parent = newNode;  return newNode;  }  void printTreeRec(Node\* root, int level, char prefix) {  if (root != nullptr) {  printTreeRec(root->right, level + 1, '/');  for (int i = 0; i < level; i++) {  cout << " ";  }  cout << prefix << "--" << root->value << endl;  printTreeRec(root->left, level + 1, '\\');  }  }  int countPositiveLeavesRec(Node\* currentNode) {  if (currentNode == nullptr) {  return 0;  }  if (currentNode->left == nullptr && currentNode->right == nullptr && currentNode->value > 0) {  return 1;  }  int leftCount = countPositiveLeavesRec(currentNode->left);  int rightCount = countPositiveLeavesRec(currentNode->right);  return leftCount + rightCount;  }  };  int main() {  setlocale(LC\_ALL, "rus");  Tree tree = Tree();  tree.balancedTree(31);  cout << "\nСозданное дерево:\n\n";  tree.printTree();  cout << "Список операций:\n";  cout << "1. Определить количество положительных листьев в дереве\n";  cout << "2. Определить сколько узлов дерева содержат заданное число\n";  cout << "3. Увеличить значение всех узлов вдвое \n";  cout << "4. Вывести дерево на экран\n";  cout << "0. Завершить программу\n";  int choice = 100;  while (choice != 0) {  cout << "Введите номер операции: ";  cin >> choice;  switch (choice) {  case 1: {  int positiveLeavesCount = tree.countPositiveLeaves();  cout << "Количество положительных листьев в дереве: " << positiveLeavesCount << endl;  break;  }  case 2: {  cout << "Введите значение, которое мы будем искать: ";  int val;  cin >> val;  int uzel = tree.getUzel(val);  if (uzel == -1)  cout << "Значение не найдено\n";  else  cout << "Значение " << val << " содержат в себе " << uzel << " узла\n";  break;  }  case 3: {  tree.doubleNodeValues();  cout << "Значения всех узлов увеличены вдвое.\n";  break;  }  case 4:  tree.printTree();  break;  case 0:  break;  default:  cout << "Неверно выбрана операция\n";  break;  }  }  return 0;  } |