|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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 ИКБО-10-21 | Смольников А.Б. |
| Принял преподаватель | Филатов А.С. |

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

Москва 2022

# **Цель работы**

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

# **Постановка задачи**

* 1. **Задача 1**

Вариант 2 (22). Вид дерева: идеально сбалансированное из n узлов (не AVL).

1. Реализовать операции общие для вариантов c 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 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.

Таблица 1. Задания варианта №22

|  |  |
| --- | --- |
| Задание 1 | Значение информационной части - Целое число  Определить количество листьев с положительными значениями  Определить, сколько узлов дерева содержат заданное число.  Увеличить значения узлов вдвое, обходя дерево алгоритмом в ширину. |

# **Решение**

* 1. **Теоретическое введение**

Нелинейные структуры данных – такие структуры данных, отображающие сложные отношения между элементами структуры. Пример нелинейной структуры данных граф, и дерево, в частности.

Граф - топологическая модель, состоящая из множества вершин и множества ребер, соединяющих эти вершины. Дерево – связный граф, в котором нет циклов. Это означает, что из любой вершины можно попасть в любую, притом единственным путем. Деревья часто используются для организации иерархической структуры данных, например, при создании двоичных деревьев поиска или кучи, в этом случае одну вершину дерева называют корнем. Лист – вершина, не имеющая потомков. Обход дерева – систематический просмотр всех вершин, при котором каждая вершина встречается один раз.

Бинарное дерево – то дерево, в котором каждая вершина имеет два потомка (правые и левые поддеревья – left и right).

Идеально сбалансированное дерево – то дерево, в котором высоты поддеревьев отличаются не более, чем на 1.

Дерево выражений - такое дерево, которое в своем корне хранит число (если операторов нет), или оператор, связные вершины на расстоянии 1 хранят операторы (оператором может быть выражение). При этом каждое поддерево данного дерева является деревом выражений.

Бинарное дерево поиска – такое бинарное дерево, которое в левом узле хранит значение, меньшее значения потомка, а в правом – значение, большее значения потомка. Данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

Обход дерева – алгоритм, в котором рассматриваются все вершины дерева.

Обход в глубину – рекурсивный метод обхода, в котором посещается корень дерева, левое поддерево и правое поддерево (в нужном порядке).

Обход в ширину – метод обхода, в котором используется очередь. Начиная с корня добавляем текущий узел в конец стека, далее пока очередь не кончится, итеративно посещаем каждую вершину из начала очереди, добавляя в ее конец дочерние вершины.

**Ответы на четные вопросы:**

*Какова степень сильноветвящегося дерева?* Если сильноветвящееся дерево содержит n-1 ключей и n указателей, то степень узлов n>2.

*Как рассчитать длину пути в дереве?* Длина пути дерева определяется как сумма длин путей ко всем его вершинам. Рассчитывается по рекурсивной формуле: Длина пути = длина пути в левом поддереве + длина пути в правом поддереве + количество узлов в дереве – 1.

*Может ли дерево быть пустым?* Да, ровно, как и любое множество. Такое дерево является nullptr.

*Дайте определение алгоритму обхода.* Алгоритм обхода – такой алгоритм, который осуществляет посещение каждой вершины дерева один раз.

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

|  |  |  |  |
| --- | --- | --- | --- |
| *Индекс* | *key* | *left* | *right* |
| 1 | 12 | 7 | 3 |
| 2 | 15 | 8 | NULL |
| 3 | 4 | 10 | NULL |
| 4 | 10 | 5 | 9 |
| 5 | 2 | NULL | NULL |
| 6 | 18 | 1 | 4 |
| 7 | 7 | NULL | NULL |
| 8 | 14 | 6 | 2 |
| 9 | 21 | NULL | NULL |
| 10 | 5 | NULL | NULL |

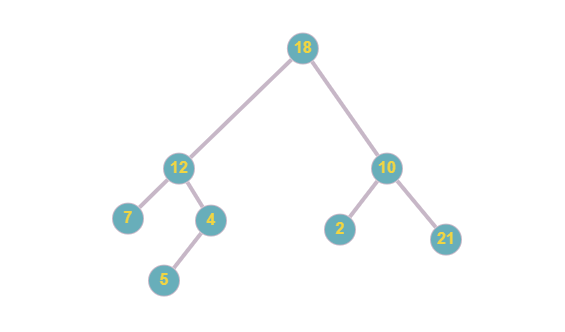


Рисунок 1.

Какая структура используется в алгоритме обхода дерева методом в «ширину»? Используется структура «Очередь». Она же – std::list, она же – двунаправленный список

Какая структура используется в не рекурсивном обходе дерева методом в «глубину»? Используется структура «Очередь». Она же – std::list, она же – двунаправленный список

Для каждого заданного арифметического выражения постройте бинарное дерево выражений:

1. a+b-c\*d+e

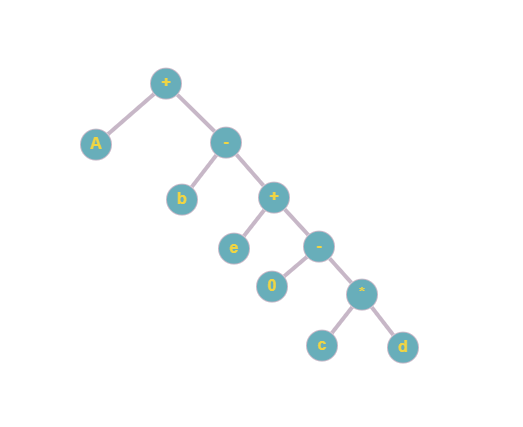


Рисунок 2.

1. /a-b\*c d

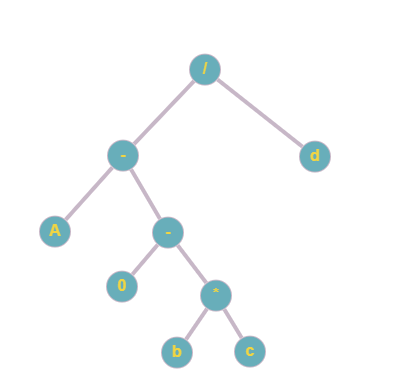


Рисунок 3.

1. a b c d / - \*

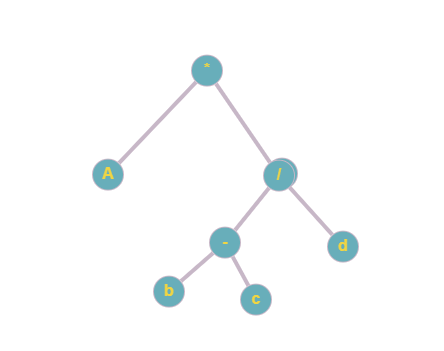


Рисунок 4.

1. - / + a b c d e

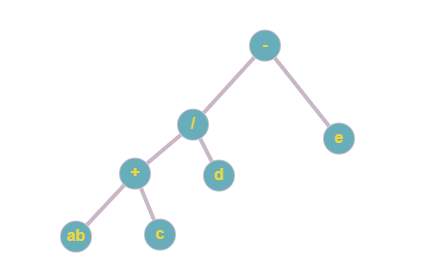


Рисунок 5.

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

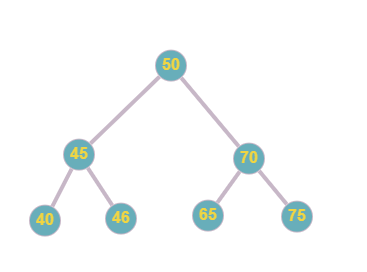


Рисунок 6.

* 1. **Функции задания №1**

Для организации записей используется класс idealBalanceTree, который создает дерево в соттветствии с заданием и управляет им. Внутри него определена структура записи – node, включающая в себя информационную часть узла и указатели на левое и правое поддеревья.

|  |
| --- |
| class idealBalanceTree {  protected:  struct node {  int key;  node \*left;  node \*right;  };  static node \*root;  int size = 0;  public:  idealBalanceTree();  void printTree(node \*element= nullptr, std::string prefix = "", bool root = true);  void generateTree(int length);  void addElement(int key,node\* &el = root);  int height(node \*element = root);  //personal task:  // count the number of positive elements in the tree  int countPositiveElements(node \*element = root);  //count the number of elements with given variable  int countElementsWithVariable(int variable, node \*element = root);  //multuply all elements with 2 using BFS  void multiplyAllElementsWith2(node \*element = root);  }; |

Конструктор idealBalanceTree () осуществляет инициализацию корневого элемента дерева значением nullptr:

|  |
| --- |
| idealBalanceTree::idealBalanceTree() {  root = nullptr;  } |

Метод printTree() выводит дерево, повернув его против часовой стрелки на 90 градусов, слева направо. Для реализации разделения информации используется строка – префикс, которая в процессе обхода в глубину растет в длине.ц

|  |
| --- |
| void idealBalanceTree::printTree(node \*element , std::string prefix, bool root) {  if (root) {  element = this->root;  }  if(!element) return;  if(element->right){  printTree(element->right, prefix + (root ? " " : "│ "), false);  }  std::cout << prefix << (root ? "└── " : "├── ") << element->key<</\* " ("<< height(element) << ")"<<\*/std::endl;  if(element->left){  printTree(element->left, prefix + (root ? " " : "│ "), false);  }  } |

Метод addElement() осуществляет добавление узла в дерево с учетом идеальной балансировки дерева. Из ключа создается узел, при помощи метода определения высоты поддерева (см. ниже) выбирается нужное поддерево, в которое нужно добавить новый элемент.

|  |
| --- |
| void idealBalanceTree::addElement(int key, node\* &el) {  node\* newNode = new node;  newNode->key = key;  newNode->left = nullptr;  newNode->right = nullptr;  size++;  if(el== nullptr){  el=newNode;  return;  }  if(el->left== nullptr){  el->left=newNode;  return;  }else if (el->right== nullptr) {  el->right = newNode;  return;  }else{  //вызвать функцию с условием, что максимальная и минимальная глубина дерева отличаются не более чем на 1  if (height(el->left) < height(el->right)) {  addElement(key, el->left);  } else {  addElement(key, el->right);  }  return;  }  } |

Метод height() определяет высоту текущего поддерева при помощи рекурсивного вызова этого же метода с попутным инкрементированием значения высоты.

|  |
| --- |
| int idealBalanceTree::height(idealBalanceTree::node \*element) {  if(element== nullptr){  return 0;  }  int leftHeight = height(element->left);  int rightHeight = height(element->right);  if(leftHeight>rightHeight){  return leftHeight+1;  }else{  return rightHeight+1;  }  } |

Метод generateTree() случайно генерирует значения в диапазоне от -30 до 30 при помощи вихря Мерсенна, вызывает метод добавления значения в дерево.

|  |
| --- |
| void idealBalanceTree::generateTree(int length) {  std::random\_device rd;  std::mt19937 gen(rd());  std::uniform\_int\_distribution<> dis(-30, 30);  for (int i = 0; i < length; i++) {  addElement(dis(gen));  }  this->size=length;  } |

Метод countPositiveElements() реализует подсчет положительных узлов дерева, обходя само дерево в глубину, инкрементируя счетчик и возвращая его значение.

|  |
| --- |
| int idealBalanceTree::countPositiveElements(node \*element) {  if(element== nullptr){  return 0;  }  int count = 0;  if(element->key>0){  count++;  }  count+=countPositiveElements(element->left);  count+=countPositiveElements(element->right);  return count;  } |

Метод countElementsWithVariable() считает количество узлов, содержащих заданное значение, обходя дерево в глубину.

|  |
| --- |
| int idealBalanceTree::countElementsWithVariable(int variable, node \*element) {  if(element== nullptr){  return 0;  }  int count = 0;  if(element->key==variable){  count++;  }  count+=countElementsWithVariable(variable,element->left);  count+=countElementsWithVariable(variable,element->right);  return count;  } |

Метод multiplyAllElementsWith2() умножает каждое значение информационной части узлов на два, обходя дерево в ширину. Обход реализован при помощи очереди, использован тип данных list. Пока очередь не пуста, обходим узел, добавляя его дочерние элементы в конец очереди, выполняем умножение. Удаляем изначальный узел из очереди.

|  |
| --- |
| void idealBalanceTree::multiplyAllElementsWith2(node \*element) {  if(element== nullptr){  return;  }  std::list <node\*> queue;  queue.push\_back(element);  while(!queue.empty()){  node\* current = queue.front();  queue.pop\_front();  if(current== nullptr){  continue;  }else{  current->key\*=2;  if(current->left){  queue.push\_back(current->left);  }  if(current->right){  queue.push\_back(current->right);  }  }  }  } |

Функция testTree() выполняет автоматизированное тестирование программы, проверяя все операции, реализованные в классе idealBalanceTree.

|  |
| --- |
| int testTree(){  system("chcp 65001");  idealBalanceTree tree;  tree.generateTree(11);  std::cout<< "Дерево: " << std::endl;  tree.printTree();  std::cout<< "Высота дерева: " << tree.height() << std::endl;  std::cout<<"Количество положительных элементов: "<<tree.countPositiveElements()<<std::endl;  std::cout<<"Количество элементов с заданным значением: "<<tree.countElementsWithVariable(5)<<std::endl;  tree.multiplyAllElementsWith2();  std::cout<<"Дерево после умножения всех элементов на 2: "<<std::endl;  tree.printTree();  return 0;  } |

* 1. **Интерфейс**

При запуске программы пользователю предоставляется выбор одного из предложенных заданий, указанных в постановке задачи. Любой другой введенный символ спровоцирует завершение программы.

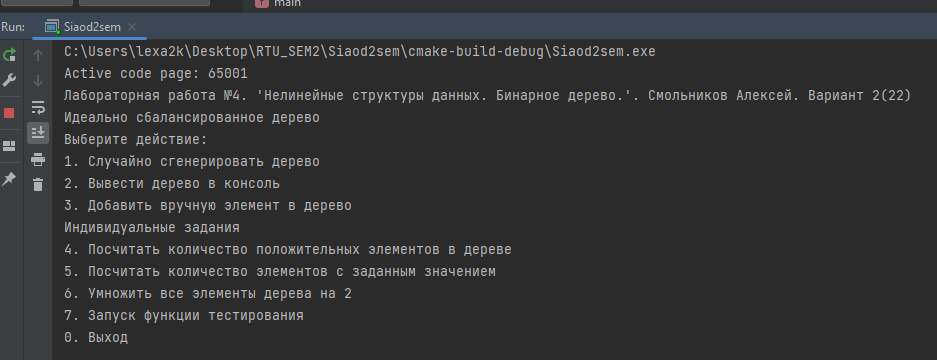


Рисунок 1. Интерфейс программы

# **Тестирование**

Пункт 1, метод generateTree() случайно генерирует дерево с диапазоном значения от -30, до 30. Сгенерировав дерево, а затем выведя его, удостоверимся, что создание дерева работает успешно, все числовые значения находятся в диапазоне.

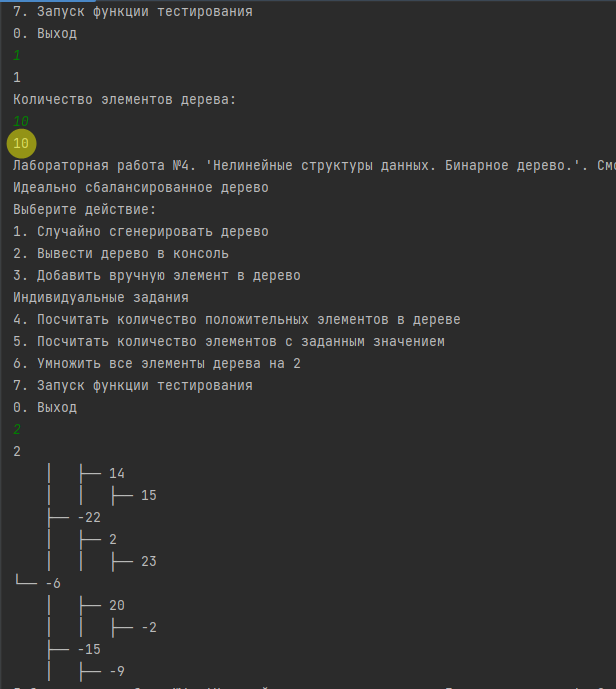


Рисунок 1. Генерация дерева

Пункт меню под номером 2 отвечает за вывод дерева, повернув его на 90 градусов. В соответствии с рисунком 1, дерево выводится корректно в нужном виде.

Под пунктом 3 расположен метод addElement(), добавляющий узел с пользовательской информацией. На рисунке 2 прослеживается создание дерева с 10 элементами, и добавлением в него еще одного со значением 999999. Видно, что компонент добавления работает корректно.

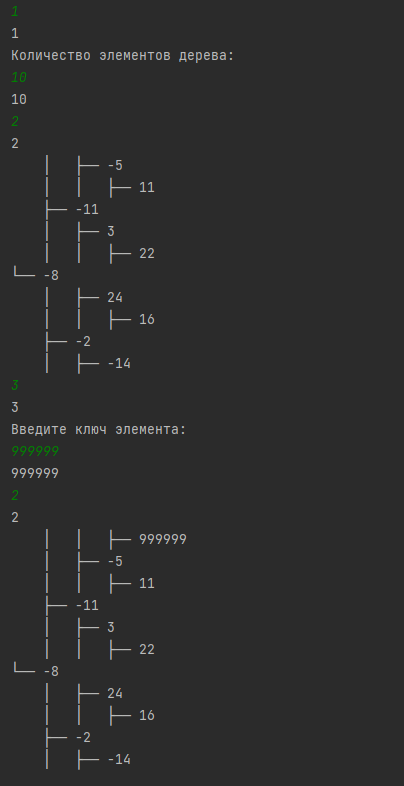


Рисунок 2. Добавление элемента в дерево

В задании 4 ведется подсчет элементов дерева, хранящих положительные числа. На рисунке 3 видно, что подсчет верный, в дереве 5 элементов:

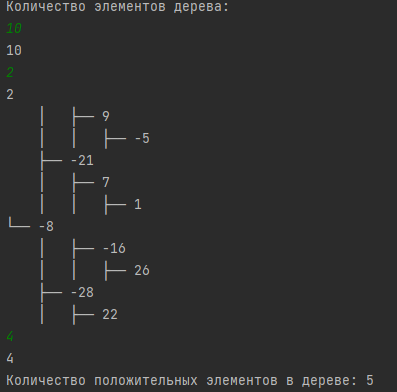


Рисунок 3. Подсчет положительных элементов

Пункт меню 5 реализует подсчет количества элементов с заданным значением. Из рисунка 4 видно, что алгоритм работает корректно:

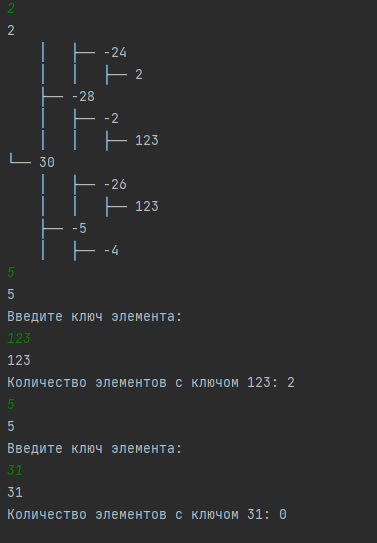


Рисунок 4. Подсчет элементов по значению

Пункт меню под номером 6 реализует умножение каждого элемента дерева на 2, при этом обходя по ширине. Из рисунка 5 видно, что метод multiplyAllElementsWith2 работает корректно:

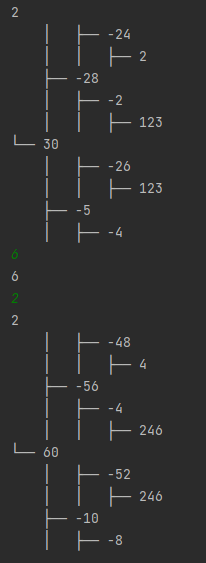


Рисунок 5. Умножение дерева на 2

Пункт 7 запускает автоматизированную функцию тестирования класса дерева. Из рисунка 6 видно, что все компоненты работают корректно:

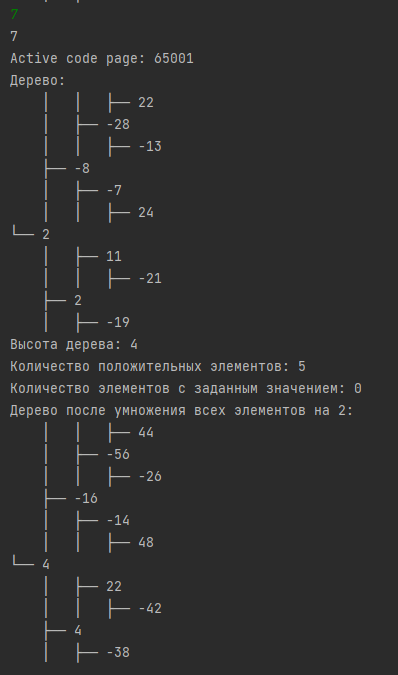


Рисунок 6. Код функции тестирования

# **Выводы**

В результате выполнения работы были:

1. Получены умения и навыки разработки и реализации операций над структурой данных «Бинарное дерево»
2. Разработана программа в соответствии с заданием варианта
3. Выполнена реализация средствами ООП.

# **Исходный код программы**

main.cpp

|  |
| --- |
|  |
| //  // Created by lexa2k on 19.10.2022.  //  #include <iostream>  #include "tree.h"  using namespace std;  idealBalanceTree::node\* idealBalanceTree::root = nullptr;  //create console menu  int main() {  system("chcp 65001");  int choice=-1;  idealBalanceTree tree;  while (choice!=0) {  cout<<"Лабораторная работа №4. 'Нелинейные структуры данных. Бинарное дерево.'. Смольников Алексей. Вариант 2(22)"<<endl;  cout<<"Идеально сбалансированное дерево"<<endl;  cout<<"Выберите действие:"<<endl;  cout<<"1. Случайно сгенерировать дерево"<<endl;  cout<<"2. Вывести дерево в консоль"<<endl;  cout<<"3. Добавить вручную элемент в дерево"<<endl;  cout<<"Индивидуальные задания"<<endl;  cout<<"4. Посчитать количество положительных элементов в дереве"<<endl;  cout<<"5. Посчитать количество элементов с заданным значением"<<endl;  cout<<"6. Умножить все элементы дерева на 2"<<endl;  cout<<"7. Запуск функции тестирования"<<endl;  cout<<"0. Выход"<<endl;  cin>>choice;  switch (choice) {  case 1:{  int length;  cout<<"Количество элементов дерева:"<<endl;  cin>>length;  tree.generateTree(length);  break;  }  case 2:{  tree.printTree();  break;  }  case 3:{  int key;  cout<<"Введите ключ элемента:"<<endl;  cin>>key;  tree.addElement(key);  break;  }  case 4:{  cout<<"Количество положительных элементов в дереве: "<<tree.countPositiveElements()<<endl;  break;  }  case 5:{  int key;  cout<<"Введите ключ элемента:"<<endl;  cin>>key;  cout<<"Количество элементов с ключом "<<key<<": "<<tree.countElementsWithVariable(key)<<endl;  break;  }  case 6:{  tree.multiplyAllElementsWith2();  break;  }  case 7:{  testTree();  break;  }  case 0:{  break;  }  default:{  cout<<"Wrong input"<<endl;  break;  }  }  }  return 0;  } |

tree.h

|  |
| --- |
| //  // Created by lexa2k on 19.10.2022.  //  #ifndef SIAOD2SEM\_TREE\_H  #define SIAOD2SEM\_TREE\_H  #include <string>  //create class for ideal balanced tree where leafs are on the same level  class idealBalanceTree {  protected:  struct node {  int key;  node \*left;  node \*right;  };  static node \*root;  int size = 0;  public:  idealBalanceTree();  void printTree(node \*element= nullptr, std::string prefix = "", bool root = true);  void generateTree(int length);  void addElement(int key,node\* &el = root);  int height(node \*element = root);  //personal task:  // count the number of positive elements in the tree  int countPositiveElements(node \*element = root);  //count the number of elements with given variable  int countElementsWithVariable(int variable, node \*element = root);  //multuply all elements with 2 using BFS  void multiplyAllElementsWith2(node \*element = root);  };  int testTree();  #endif //SIAOD2SEM\_TREE\_H |

tree.cpp

|  |
| --- |
| //  // Created by lexa2k on 19.10.2022.  //  #include "tree.h"  #include <iostream>  #include <random>  #include <list>  idealBalanceTree::idealBalanceTree() {  root = nullptr;  }  void idealBalanceTree::generateTree(int length) {  std::random\_device rd;  std::mt19937 gen(rd());  std::uniform\_int\_distribution<> dis(-30, 30);  for (int i = 0; i < length; i++) {  addElement(dis(gen));  }  this->size=length;  }  void idealBalanceTree::addElement(int key, node\* &el) {  node\* newNode = new node;  newNode->key = key;  newNode->left = nullptr;  newNode->right = nullptr;  size++;  if(el== nullptr){  el=newNode;  return;  }  if(el->left== nullptr){  el->left=newNode;  return;  }else if (el->right== nullptr) {  el->right = newNode;  return;  }else{  //вызвать функцию с условием, что максимальная и минимальная глубина дерева отличаются не более чем на 1  if (height(el->left) < height(el->right)) {  addElement(key, el->left);  } else {  addElement(key, el->right);  }  return;  }  }  int idealBalanceTree::height(idealBalanceTree::node \*element) {  if(element== nullptr){  return 0;  }  int leftHeight = height(element->left);  int rightHeight = height(element->right);  if(leftHeight>rightHeight){  return leftHeight+1;  }else{  return rightHeight+1;  }  }  void idealBalanceTree::printTree(node \*element , std::string prefix, bool root) {  if (root) {  element = this->root;  }  if(!element) return;  if(element->right){  printTree(element->right, prefix + (root ? " " : "│ "), false);  }  std::cout << prefix << (root ? "└── " : "├── ") << element->key<</\* " ("<< height(element) << ")"<<\*/std::endl;  if(element->left){  printTree(element->left, prefix + (root ? " " : "│ "), false);  }  }  //personal task:  // count the number of positive elements in the tree  int idealBalanceTree::countPositiveElements(node \*element) {  if(element== nullptr){  return 0;  }  int count = 0;  if(element->key>0){  count++;  }  count+=countPositiveElements(element->left);  count+=countPositiveElements(element->right);  return count;  }  //count the number of elements with given variable  int idealBalanceTree::countElementsWithVariable(int variable, node \*element) {  if(element== nullptr){  return 0;  }  int count = 0;  if(element->key==variable){  count++;  }  count+=countElementsWithVariable(variable,element->left);  count+=countElementsWithVariable(variable,element->right);  return count;  }  //multuply all elements with 2 using BFS  void idealBalanceTree::multiplyAllElementsWith2(node \*element) {  if(element== nullptr){  return;  }  std::list <node\*> queue;  queue.push\_back(element);  while(!queue.empty()){  node\* current = queue.front();  queue.pop\_front();  if(current== nullptr){  continue;  }else{  current->key\*=2;  if(current->left){  queue.push\_back(current->left);  }  if(current->right){  queue.push\_back(current->right);  }  }  }  }  int testTree(){  system("chcp 65001");  idealBalanceTree tree;  tree.generateTree(11);  std::cout<< "Дерево: " << std::endl;  tree.printTree();  std::cout<< "Высота дерева: " << tree.height() << std::endl;  std::cout<<"Количество положительных элементов: "<<tree.countPositiveElements()<<std::endl;  std::cout<<"Количество элементов с заданным значением: "<<tree.countElementsWithVariable(5)<<std::endl;  tree.multiplyAllElementsWith2();  std::cout<<"Дерево после умножения всех элементов на 2: "<<std::endl;  tree.printTree();  return 0;  } |