ГУАП

# КАФЕДРА №43

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| старший преподаватель |  | 16.11.2023 |  | С.А. Рогачев |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ № 6 |
| АВЛ – ДЕРЕВЬЯ ПОИСКА |
| по курсу: АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4232 |  | 16.11.2023 |  | М.В.Куриш |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2023

1. **Цель работы**

Целью работы является изучение деревьев поиска и получение практических навыков их использования.

1. **Задание на лабораторную работу**

Разработать на языке программирования высокого уровня программу, которая должна выполнять следующие функции: − добавлять элементы в сбалансированное дерево поиска; − удалять элементы из сбалансированного дерева поиска; − искать элементы в дереве поиска с выводом количества шагов, за которое осуществляется поиск; − выводить дерево на экран (любым способом доступным для восприятия); − выводить список, соответствующий обходу вершин, в соответствии с вариантом задания; − осуществлять операцию, заданную в таблице 6. Количество элементов и порядок их ввода при создании сбалансированного дерева поиска определяется по согласованию с преподавателем.

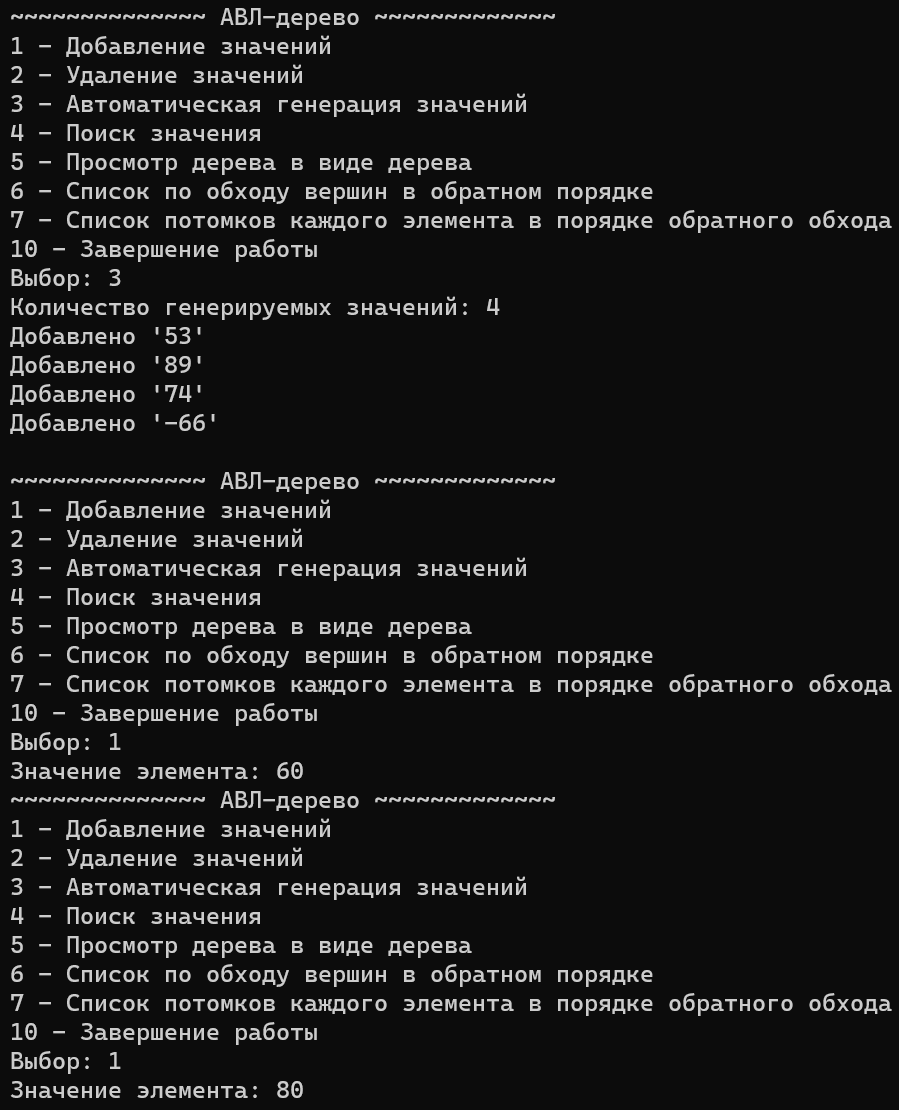
Вариант 22.

|  |  |  |
| --- | --- | --- |
| **№ вар.** | **Задание** | **Порядок обхода** |
| 22 | Для каждого узла дерева A, вывести список потомков в порядке обхода. | Обратный |

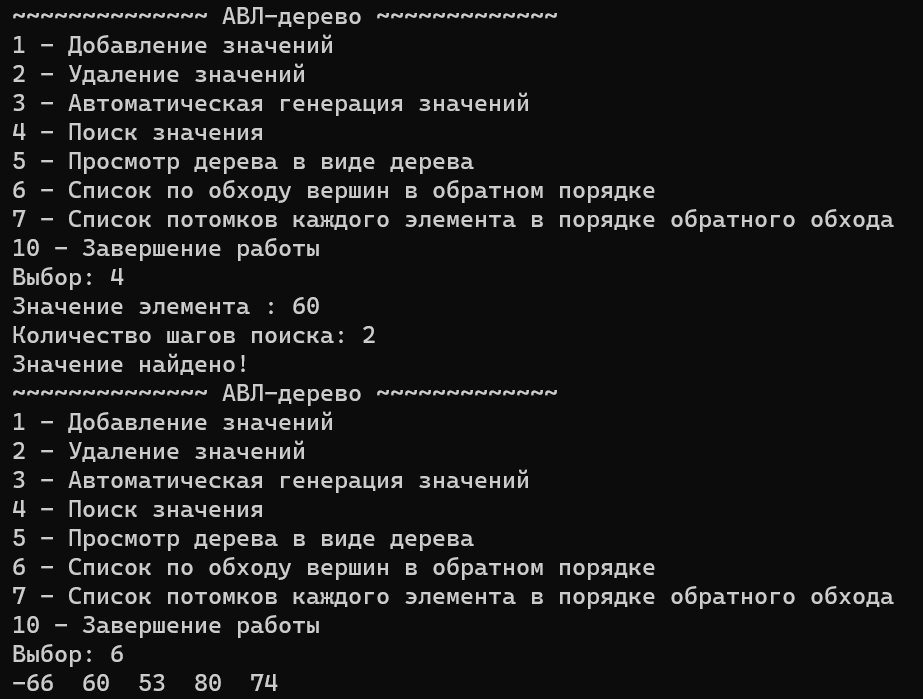
1. **Листинг программы**

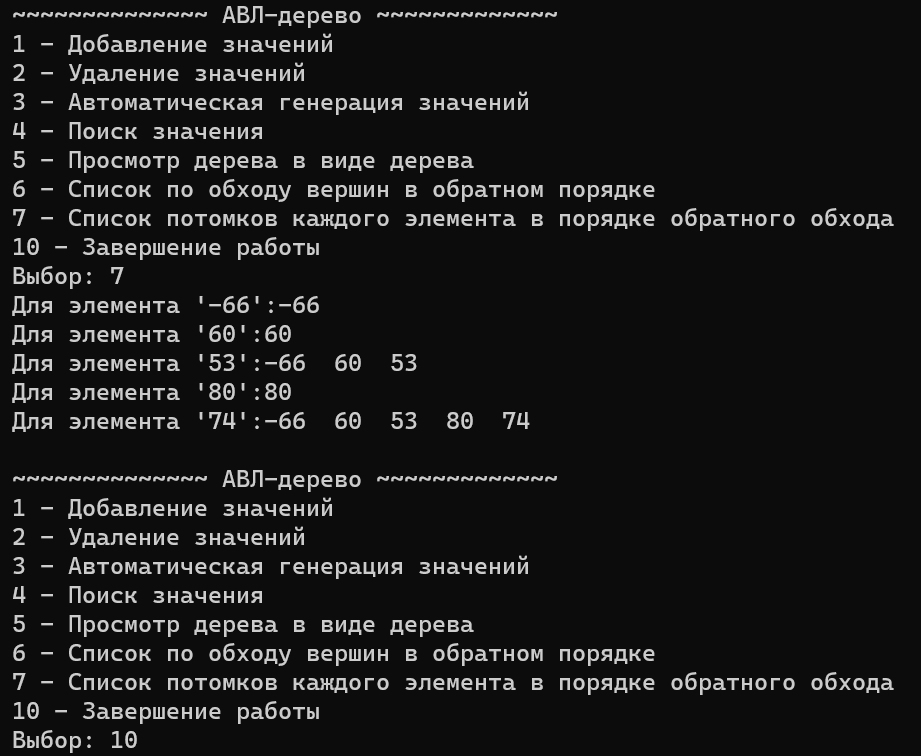
|  |
| --- |
| #include <iostream>  #include <random>  using namespace std;  int intInput(string welcome) {  int a;  cout << welcome;  cin >> a;  while (!cin) {  cout << "\x1B[31mНеверный ввод. (Ожидается число)\033[0m\n";  cin.clear();  cin.ignore(999, '\n');  cout << "Введите заново: ";  cin >> a;  }  return a;  }  class Tree {  class Node {  public:  int data;  Node\* left;  Node\* right;  int height;  Node(int data) {  this->data = data;  this->left = nullptr;  this->right = nullptr;  this->height = 1;  }  };  Node\* root;  int height(Node\* node) {  return node == nullptr ? 0 : node->height;  }  int bFactor(Node\* node) {  return height(node->right) - height(node->left);  }  void updateHeight(Node\* node) {  int heightL = height(node->left);  int heightR = height(node->right);  node->height = (heightL > heightR ? heightL : heightR) + 1;  }  Node\* leftTurn(Node\* node) {  Node\* childRight = node->right;  node->right = childRight->left;  childRight->left = node;  updateHeight(node);  updateHeight(childRight);  return childRight;  }  Node\* rightTurn(Node\* node) {  Node\* childLeft = node->left;  node->left = childLeft->right;  childLeft->right = node;  updateHeight(node);  updateHeight(childLeft);  return childLeft;  }  Node\* balance(Node\* node) {  updateHeight(node);  if (bFactor(node) == 2)  {  if (bFactor(node->right) < 0)  node->right = rightTurn(node->right);  return leftTurn(node);  }  if (bFactor(node) == -2)  {  if (bFactor(node->left) > 0)  node->left = leftTurn(node->left);  return rightTurn(node);  }  return node;  }  Node\* findNearest(Node\* node, bool rightChild) {  if (rightChild) {  while (node->left != nullptr)  node = node->left;  }  else {  while (node->right != nullptr)  node = node->right;  }  return node;  }  void show(const std::string& prefix, const Node\* node, bool isLeft)  {  if (node != nullptr)  {  cout << prefix;  cout << (isLeft ? "|--> " : "L--> ");  cout << node->data << endl;  show(prefix + (isLeft ? "| " : " "), node->left, true);  show(prefix + (isLeft ? "| " : " "), node->right, false);  }  }  void showTraversalList(Node\* node, bool hideMainRoot = true) {  if (node == nullptr)  return;  showTraversalList(node->left, false);  showTraversalList(node->right, false);  if (!hideMainRoot)  cout << node->data << " ";  }  void showEachTraversalList(Node\* node) {  if (node == nullptr)  return;  showEachTraversalList(node->left);  showEachTraversalList(node->right);  cout << "Для элемента '" << node->data << "':";  showTraversalList(node);  cout << endl;  }  Node\* add(Node\* node, int newVal)  {  if (!node) return new Node(newVal);  if (newVal < node->data)  node->left = add(node->left, newVal);  else  node->right = add(node->right, newVal);  return balance(node);  }  Node\* findmin(Node\* node)  {  return node->left ? findmin(node->left) : node;  }  Node\* removemin(Node\* node)  {  if (node->left == 0)  return node->right;  node->left = removemin(node->left);  return balance(node);  }  Node\* remove(Node\* node, int newVal)  {  if (!node) return 0;  if (newVal < node->data)  node->left = remove(node->left, newVal);  else if (newVal > node->data)  node->right = remove(node->right, newVal);  else  {  Node\* q = node->left;  Node\* r = node->right;  delete node;  if (!r) return q;  Node\* min;  if (height(r) <= height(q)) {  min = findmin(r);  min->right = removemin(r);  min->left = q;  }  else {  min = findmin(q);  min->left = removemin(q);  min->right = r;  }  return balance(min);  }  return balance(node);  }  public:  Tree() {  this->root = nullptr;  }  void add(int newVal) {  root = add(root, newVal);  }  void remove(int val) {  root = remove(root, val);  }  int find(int val) {  int stepsCount = 0;  Node\* curr = root;  while (curr != nullptr) {  if (curr->data == val) {  cout << "Количество шагов поиска: " << stepsCount << endl;  return true;  }  if (val < curr->data)  curr = curr->left;  else  curr = curr->right;  stepsCount++;  }  cout << "Количество шагов поиска: " << stepsCount << endl;  return false;  }  void show()  {  show("", root, false);  }  void showTraversalList() {  showTraversalList(root, false);  cout << endl;  }  void showEachTraversalList() {  showEachTraversalList(root);  cout << endl;  }  };  void generateValues(Tree\* myTree, int count) {  random\_device rd;  mt19937 mt(rd());  uniform\_int\_distribution<int> uniNumbers(-100, 100);  while (count != 0) {  int added = uniNumbers(mt);  myTree->add(added);  cout << "Добавлено '" << added << "'\n";  count--;  }  cout << endl;  }  int main()  {  setlocale(LC\_ALL, "RUS");  Tree myTree;  bool flag1 = true;  while (flag1) {  cout << "~~~~~~~~~~~~~~ АВЛ-дерево ~~~~~~~~~~~~~\n";  cout << "1 - Добавление значений\n";  cout << "2 - Удаление значений\n";  cout << "3 - Автоматическая генерация значений\n";  cout << "4 - Поиск значения\n";  cout << "5 - Просмотр дерева в виде дерева\n";  cout << "6 - Список по обходу вершин в обратном порядке\n";  cout << "7 - Список потомков каждого элемента в порядке обратного обхода\n";  cout << "10 - Завершение работы\n";  switch (intInput("Выбор: ")) {  case 1:  myTree.add(intInput("Значение элемента: "));  break;  case 2:  myTree.remove(intInput("Значение элемента: "));  break;  case 3:  generateValues(&myTree, intInput("Количество генерируемых значений: "));  break;  case 4:  if (myTree.find(intInput("Значение элемента : ")))  cout << "Значение найдено!\n";  else  cout << "Значение НЕ найдено.\n";  break;  case 5:  myTree.show();  break;  case 6:  myTree.showTraversalList();  break;  case 7:  myTree.showEachTraversalList();  break;  case 10:  flag1 = false;  break;  default:  cout << "\x1B[31mНеверный выбор пункта меню\033[0m\nПовторите попытку\n";  break;  }  }  } |

1. **Контрольный пример**

****

****

****

****

1. **Выводы**

Изучены деревья поиска и получены практические навыки их использования. На практике реализован обратный обход двоичного дерева, а также визуализация дерева в удобном для восприятия виде.