Министерство образования и науки Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования

**«ПЕРМСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ПЕРМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

Кафедра “Информационные технологии и автоматизированные системы”

**ОТЧЕТ**

**по лабораторной работе №20**

Дисциплина: “Основы теории алгоритмов и структуры данных”

Тема: Бинарные деревья

Вариант №17

Выполнил:

Студент группы РИС-20-2б

Пономарев Егор Витальевич

Проверила:

Доцент кафедры ИТАС

Полякова О.А.

**Пермь, 2021**

**Цель работы**

Реализовать бинарное дерево

**Постановка задачи**

1. Сформировать идеально сбалансированное бинарное дерево, тип информационного поля указан в варианте.

  2. Распечатать полученное дерево.

  3. Выполнить обработку дерева в соответствии с заданием, вывести полученный результат.

  4. Преобразовать идеально сбалансированное дерево в дерево поиска.

  5. Распечатать полученное дерево.

**Анализ задачи**

**(1)**

1. Создание идеально сбалансированного дерева:

|  |
| --- |
| p = Tree(n, p); |

1. Печать дерева на консоли:

|  |
| --- |
| PrintTree(p, n); |

1. Нахождение среднего арифметического среди элементов дерева:

|  |
| --- |
| cout << "\nSred\_Arifm: " << sred\_arifm << endl; |

1. Образование дерева поиска:

|  |
| --- |
| for (int i = 0; i < n1; ++i)  {  cin >> in;  tree.insert(in);  } |

1. Печать дерева поиска на консоли в порядке возрастания:

|  |
| --- |
| tree.print\_tree(tree.m\_root); |

**(2)**

**tree\_elem\* p** - указатель на корень дерева

int in - вводимые в дерево поиска элементы

int n1 - кол-во элементов в дереве поиска

tree\_elem\* r - элемент идеально сбалансированного дерева

int nl, nr; - кол-во элементов в левой и правой частях дерева

tree\_elem\* curr - текущий элемент

int m\_size; - размер дерева

tree\_elem\* m\_root; - указатель на корень дерева

int data; - параметр элемента дерева

tree\_elem\* left; - левый потомок вершины дерева

tree\_elem\* right; - правый потомок вершины дерева

**(3)**

Данные представлены в виде идеально сбалансированного и поискового деревьев

**(4)**

Ввод в идеально сбалансированное дерево производится следующим образом:

|  |
| --- |
| r = new tree\_elem; cin >> r->data; |

Ввод в дерево поиска производится следующим образом:

|  |
| --- |
| for (int i = 0; i < n1; ++i)  {  cin >> in;  tree.insert(in);  } |

Вывод в идеально сбалансированном дереве производится следующим образом:

|  |
| --- |
| PrintTree(p, n); |

Вывод в дереве поиска производится следующим образом:

|  |
| --- |
| tree.print\_tree(tree.m\_root); |

**(5)** Поставленные задачи стоит решать следующим образом:

1. Формирование идеально сбалансированного дерева:

|  |
| --- |
| tree\_elem\* Tree(int n, tree\_elem\* p) {  tree\_elem\* r;  int nl, nr;  if (n == 0) { p = NULL; return p; }  nl = n / 2;  nr = n - nl - 1;  r = new tree\_elem;  cin >> r->data;  r->left = Tree(nl, r->left);  r->right = Tree(nr, r->right);  p = r;  return p; } |

Объявляется элемент и на каждой рекурсивной итерации его поле с данными заполняется. Сначала заполняется левое плечо дерева размером nl, пока оно не станет равно 0. Далее заполняется правое плечо, размер которого равен левому, либо на единицу меньше. Таким образом, к концу второй рекурсии создается идеально сбалансированное дерево и возвращается указатель на корень дерева.

1. Печать идеально сбалансированного дерева и подсчет среднего арифметического:

|  |
| --- |
| if (p)  {  PrintTree(p->left, level + 1);  for (int i = 0; i < level; i++) cout << " ";  cout << p->data << endl;  sred\_arifm += (p->data / (float)n);  PrintTree(p->right, level + 1);  } |

Сначала рекурсивно с конца печатается левая часть дерева, далее вызывается функция от правого плеча, и также печатаются ее элементы. Вместе с печатью к вычисляется среднее арифметическое элементов

1. Образование дерева поиска:

|  |
| --- |
| void binary\_tree::insert(int key) {  tree\_elem\* curr = m\_root;  while (curr && curr->data != key)  {  if (curr->data > key && curr->left == NULL)  {  curr->left = new tree\_elem(key);  ++m\_size;  return;  }  if (curr->data < key && curr->right == NULL)  {  curr->right = new tree\_elem(key);  ++m\_size;  return;  }  if (curr->data > key)  curr = curr->left;  else  curr = curr->right;  } } |

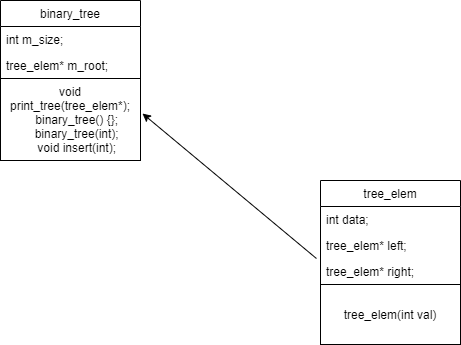
При добавлении элемента в дерево поиска все начинается с корневой вершины. Проверяется существование корня дерева и несовпадение с уже существующими данными, которые поступают в аргументе метода. Если все выполняется проверяется следующее: Если данные текущей вершины больше входного значения и левого потомка не существует, то входное значение уходит в левое плечо вершины. Иначе, оно уходит в правое плечо. Путешествие по дереву, пока не найдено место для входного элемента, происходит так: Если входное значение меньше текущей вершины, то переходим к следующей вершине, являющейся левым потомком. Иначе, в правого потомка

1. Печать дерева поиска:

|  |
| --- |
| void binary\_tree::print\_tree(tree\_elem\* curr) {  if (curr)  {  print\_tree(curr->left);  cout << curr->data << " ";  print\_tree(curr->right);  } } |

Происходит вывод дерева поиска в порядке возрастания. Поступает корень дерева, идет рекурсия по левому плечу дерева до достижения нулевого указателя, потом по правому плечу до достижения нулевого указателя. Так собираются данные с вершин с конца левого плеча (от меньшего к большему). Подобно перечисленному ранее собираются данные с правого плеча

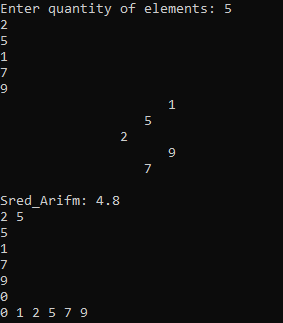
**UML-диаграмма**



**Код на C++**

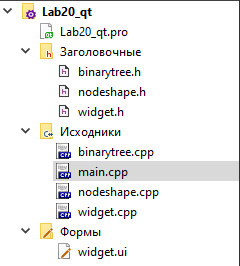
|  |
| --- |
| #include <iostream> using namespace std;  int n;   // binary search tree class tree\_elem { public:  int data;  tree\_elem\* left;  tree\_elem\* right;  tree\_elem(int val)  {  left = NULL;  right = NULL;  data = val;  }  tree\_elem()  {   } };  class binary\_tree { public:  tree\_elem\* m\_root; private:  int m\_size; public:  void print\_tree(tree\_elem\*);  binary\_tree() {};  binary\_tree(int);  void insert(int); };  binary\_tree::binary\_tree(int key) {  m\_root = new tree\_elem(key);  m\_size = 1; }  void binary\_tree::print\_tree(tree\_elem\* curr) {  if (curr)  {  print\_tree(curr->left);  cout << curr->data << " ";  print\_tree(curr->right);  } }    void binary\_tree::insert(int key) {  tree\_elem\* curr = m\_root;  while (curr && curr->data != key)  {  if (curr->data > key && curr->left == NULL)  {  curr->left = new tree\_elem(key);  ++m\_size;  return;  }  if (curr->data < key && curr->right == NULL)  {  curr->right = new tree\_elem(key);  ++m\_size;  return;  }  if (curr->data > key)  curr = curr->left;  else  curr = curr->right;  } }  tree\_elem\* Tree(int n, tree\_elem\* p) {  tree\_elem\* r;  int nl, nr;  if (n == 0) { p = NULL; return p; }  nl = n / 2;  nr = n - nl - 1;  r = new tree\_elem;  cin >> r->data;  r->left = Tree(nl, r->left);  r->right = Tree(nr, r->right);  p = r;  return p; }  float sred\_arifm = 0; void PrintTree(tree\_elem\* p, int level) {  if (p)  {  PrintTree(p->left, level + 1);  for (int i = 0; i < level; i++) cout << " ";  cout << p->data << endl;  sred\_arifm += (p->data / (float)n);  PrintTree(p->right, level + 1);  } }     int main() {  tree\_elem\* p = 0;  cout << "Enter quantity of elements: ";  cin >> n;  p = Tree(n, p);  PrintTree(p, n);  cout << "\nSred\_Arifm: " << sred\_arifm << endl;   int in, n1; cin >> in >> n1;  binary\_tree tree(in);  for (int i = 0; i < n1; ++i)  {  cin >> in;  tree.insert(in);  }  tree.print\_tree(tree.m\_root);   return 0; } |

**Скриншоты**

****

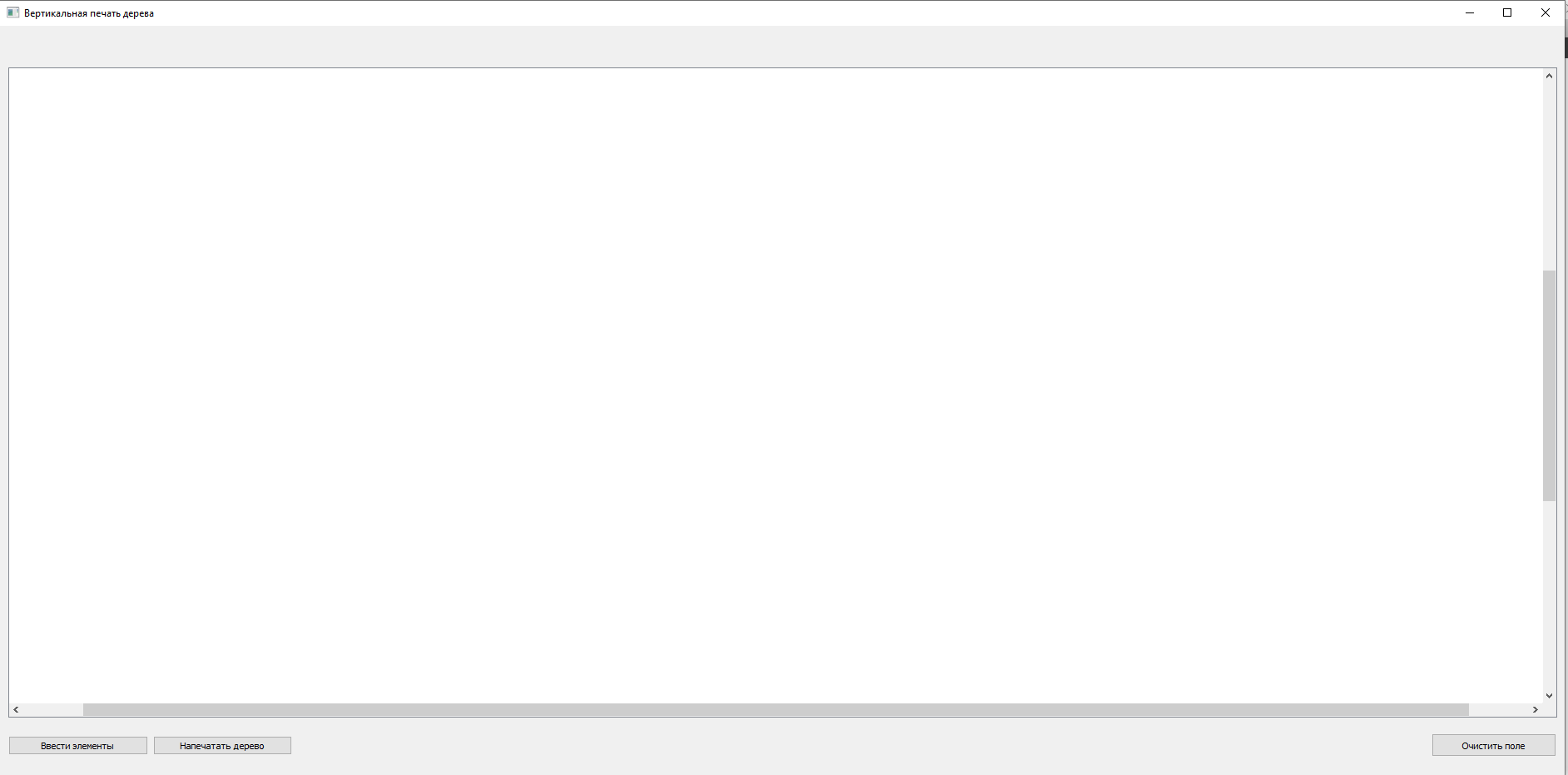
**Реализация на Qt**

Файлы проекта на Qt:

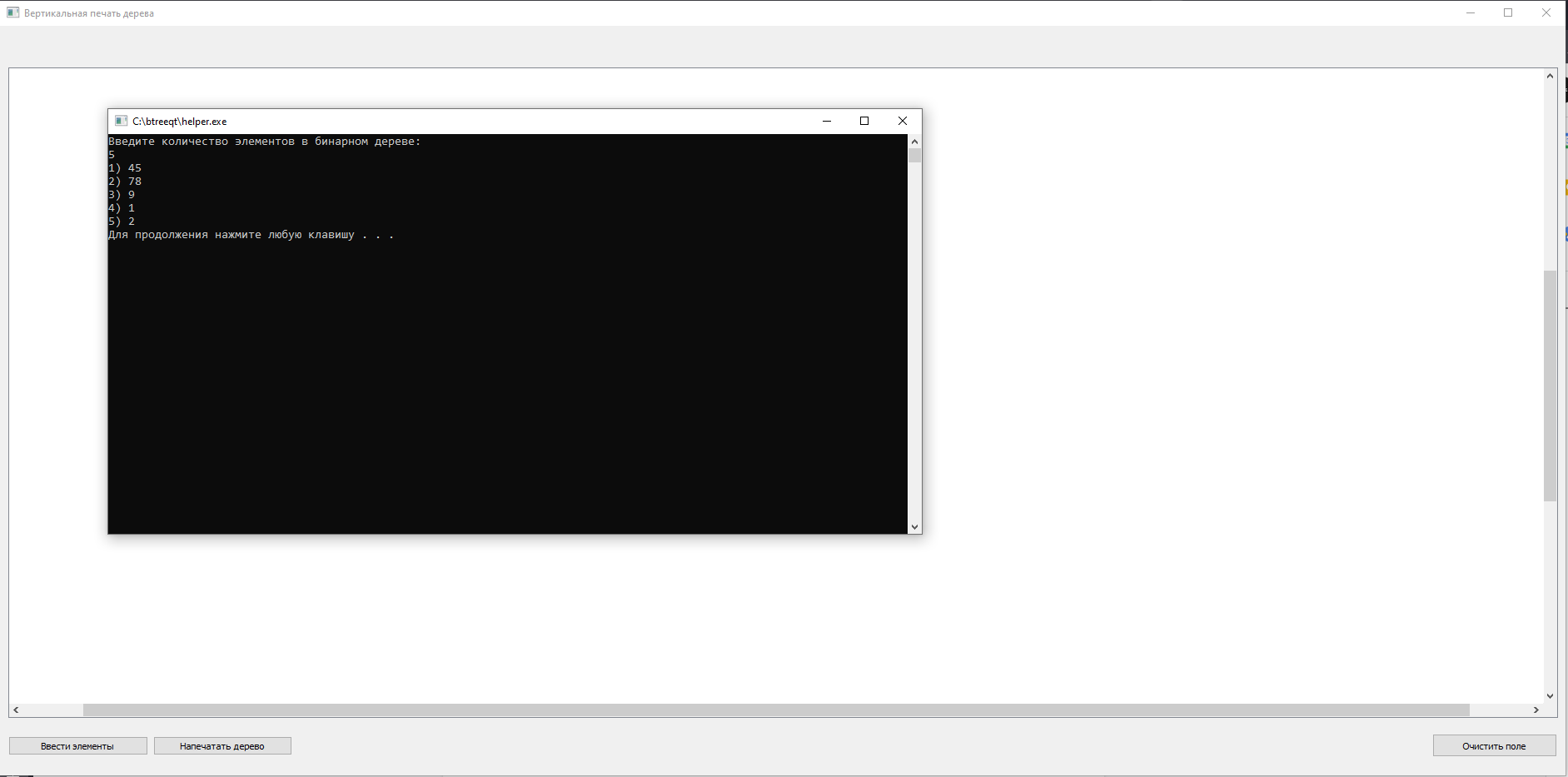


Вывод программы:

1. Начальное окно



1. Ввод элементов дерева:



1. Печать результата:

