КАФЕДРА №

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №6

АВЛ - ДЕРЕВЬЯ ПОИСКАИ

по курсу: Структуры и алгоритмы обработки данных

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. №

	подпись, дата	инициалы, фамилия

Санкт-Петербург 2022

Цель работы

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

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 4

4	Вывести глубину самого верхнего листа дерева (maxh) и самого нижнего листа (ов) дерева (minh), а так же их значения. Удалить элементы и перебалансировать дерево. Процедуру повторять до тех пор, пока не выполнится условие maxh = minh	В ширину

Листинг

main.cpp

```
#include <iostream>
using namespace std;

#include <iomanip>


#include "tree.h"

#define RANDOM_MIN -100
#define RANDOM_MAX 100

// проверка ввода
double read_double(){
        double x;
  while ( (scanf("%lf",&x) ) != 1 ) {
    printf("Неверное введенное значение, попробуйте еще: ");
    while(getchar() != '\n');
  }
  return x;
}

// рандом
```

```
int random_int(int a, int b) {
  return a + (rand() % ( b - a + 1 ) );
}

// вывод и ввод элементов меню
int menu() {
  while (true) {
    cout << "1) Вывести дерево" << endl;
    cout << "2) Добавить элемент" << endl;
    cout << "3) Добавить несколько элементов (random)" << endl;
    cout << "4) Удалить элемент" << endl;
    cout << "5) Поиск" << endl;
    cout << "6) Обход дерева (В ширину)" << endl;
    cout << "7) Вывод глубин листов (minh, maxh)" << endl;
    cout << "8) Выровнять листы (minh == maxh)" << endl;
    cout << "0) Выход" << endl;
    cout << "[menu] > ";
    int id = read_double();
    if (0 <= id <= 8) {
      return id;

    } else {
      cout << "Этого нет в меню" << endl;
    }
  }
}

int main() {
  // смена кодировки
  system("chcp 65001"); // для VS заменить на setlocale(LC_ALL, "Russian");

  Tree tree;

  int menu_i;
  while (true) {
    menu_i = menu();

    switch (menu_i) {
      case (0):
        return 0;
        break;

      case (1):
        tree.show();
        break;

      case (2): {
        double a;
        while (true) {
          cout << "Новый элемент: ";
          a = read_double();
          if (!tree.append(a)) {
```

```
      cout << "Такой элемент уже существует." << endl;
    } else break;
  }
  break;
}

case (3): {
  cout << "Количество новых элементов: ";
  double a = read_double();
  for (int i = 0; i < a; i++) {
    tree.append(random_int(RANDOM_MIN, RANDOM_MAX));
  }
  break;
}

case (4): {
  cout << "Удаляемый элемент: ";
  double a = read_double();
  tree.remove_elem(a);
  break;
}

case (5): {
  cout << "Элемент: ";
  double a = read_double();
  Node* find_element = tree.find(a);
  if (find_element != NULL) {
    tree.print_recursion(find_element, NULL, false);
  } else cout << "Такого элемента не существет." << endl;
  break;
}

case (6): {
  tree.bfsearch();
  cout << endl;
  break;
}

case (7): {
  vector<double*> v = tree.get_height();
  int min_id = 0;
  int max_id = 0;
  cout << "Все листы: ";
  for (int i = 0; i < v.size(); i++) {
    cout << setw(4) << v.at(i)[0] << ":" << v.at(i)[1] << " ";
    if (v.at(i)[1] > v.at(min_id)[1])
      min_id = i;
    if (v.at(i)[1] < v.at(max_id)[1])
      max_id = i;
  }
  cout << endl;
  cout << "minh = " << v.at(min_id)[0] << ":" << v.at(min_id)[1] << endl;
```

```
      cout << "maxh = " << v.at(max_id)[0] << ":" << v.at(max_id)[1] << endl;
      break;
    }

    case (8): {
      vector<double*> v;
      int min_id;
      bool ok;
      while (true) {
        v = tree.get_height();

        min_id = 0;
        ok = true;
        for (int i = 0; i < v.size(); i++) {
          if (v.at(i)[1] < v.at(min_id)[1]) {
            ok = false;
            min_id = i;
          }
        }

        if (ok) break;

        for (int i = 0; i < v.size(); i++) {
          if (v.at(min_id)[1] != v.at(i)[1]) {
            tree.remove_elem(v.at(i)[0]);
          }
        }
      }
      break;
    }

  }
 }

 return 0;
}
```

tree.h

```
#include <iostream>
using namespace std;
#include <vector>

// узел дерева
struct Node {
 double elem;     // содержимое узла
 int height;   // высота узла
 Node *left = NULL;   // указатель на меньшего потомка
 Node *right = NULL;  // указатель на большего потомка
};

// для вывода деревая
struct Trunk {
```

```
    Trunk *prev;
    string str;

    Trunk(Trunk *prev, string str)
    {
        this->prev = prev;
        this->str = str;
    }
};

#include "balance.h"

class Tree {
public:
  Tree();

  bool append(double);
  void remove_elem(double);
  void show();
  Node* get_root();
  vector<double*> get_height();
  void get_height_recursion(Node* ptr, int);
  Node* find(double);
  Node* balance_node(Node* ptr);
  // void PostOrder(Node* ptr);
  void bfsearch();
  void print_recursion(Node* ptr, Trunk *prev, bool isLeft);

private:
  Node *tree = NULL;
  vector<double*> height_vector;
  vector<Node*> width_queue;

};

// конструктор
Tree::Tree() {}

// доавление элемента
bool Tree::append(double elem) {
  bool ok = true;
  if (tree == NULL) {
    tree = new Node;
    tree -> elem = elem;
    tree -> height = 1;
  } else {
    Node *root = tree;
    int height = 1;
    while (true) {
      if (tree -> elem == elem) {
        ok = false;
        break;
```

```
      } else if (tree -> elem < elem) {
        if (tree -> right != NULL) {
          tree = tree -> right;
        } else {
          tree -> right = new Node;
          tree -> right -> height = height;
          tree -> right -> elem = elem;
          break;
        }
      } else {
        if (tree -> left != NULL) {
          tree = tree -> left;
        } else {
          tree -> left = new Node;
          tree -> left -> height = height;
          tree -> left -> elem = elem;
          break;
        }
      }
      height++;
    }
    tree = root;
    tree = balance_node(tree);
  }

  if (ok)
    return true;
  else
    return false;
}

// удаление дерева
void Tree::remove_elem(double elem) {
  tree = remove(tree, elem);
}

// балансировка всего дерева
Node* Tree::balance_node(Node *ptr) {
  if (ptr -> left != NULL) {
    ptr -> left = balance_node(ptr -> left);
  }
  if (ptr -> right != NULL) {
    ptr -> right = balance_node(ptr -> right);
  }
  ptr = balance(ptr);
  return ptr;
}

// вернуть указатель дерева
Node* Tree::get_root() {
  return tree;
}
```

```
// поиск элемента
Node* Tree::find(double elem) {
  Node* find_elem = tree;
  int steps = 0;
  if (tree != NULL) {
    while (true) {
      steps++;
      if (find_elem -> elem < elem) {
        if (find_elem -> right != NULL) {
          find_elem = find_elem -> right;
        } else break;
      } else if (find_elem -> elem > elem) {
        if (find_elem -> left != NULL) {
          find_elem = find_elem -> left;
        } else break;
      } else {
        cout << "Для поисика потребовалось " << steps << " шагов." << endl;
        return find_elem;
      }
    }
  }
  return NULL;
}

// обход дерева в ширину
void Tree::bfsearch() {
  width_queue.clear();
  width_queue.push_back(tree);

  Node* buf = NULL;
  while (width_queue.size() > 0) {
    buf = width_queue.at(0);
    width_queue.erase(width_queue.begin());
    cout << buf -> elem << " ";

    if (buf -> left)
      width_queue.push_back(buf -> left);

    if (buf -> right)
      width_queue.push_back(buf -> right);
  }

}
//
// // обход дерева (Обратный)
// void Tree::PostOrder(Node* ptr) {
//   if (ptr == NULL) return;
//   PostOrder(ptr -> left);
//   PostOrder(ptr -> right);
//   cout << ptr -> elem << " ";
// }
```

```
vector<double*> Tree::get_height() {
  height_vector.clear();
  get_height_recursion(tree, 1);
  return height_vector;
}

// получить листы дерева
void Tree::get_height_recursion(Node* ptr, int height = 1) {
  if (ptr != NULL) {
    if (ptr -> left == NULL && ptr -> right == NULL) {
      double* arr = new double[2];
      arr[0] = ptr -> elem;
      arr[1] = height;
      height_vector.push_back(arr);
    }

    else {
      if (ptr -> left != NULL) get_height_recursion(ptr -> left, height + 1);
      if (ptr -> right != NULL) get_height_recursion(ptr -> right, height + 1);
    }
  }
}

// показать дерево
void Tree::show() {
  print_recursion(tree, NULL, false);
}

// Вспомогательная функция для печати ветвей бинарного дерева
void showTrunks(Trunk *p) {
  if (p == nullptr) {
    return;
  }

  showTrunks(p->prev);
  cout << p->str;
}

// вывод дерева (рекурсивно)
void Tree::print_recursion(Node* ptr, Trunk *prev, bool isLeft) {
  if (ptr != NULL) {
    string prev_str = "    ";
    Trunk *trunk = new Trunk(prev, prev_str);

    print_recursion(ptr->right, trunk, true);

    if (!prev) {
      trunk->str = "———";
    } else if (isLeft) {
      trunk->str = ".———";
```

```
      prev_str = "   |";
    } else {
      trunk->str = "`———";
      prev->str = prev_str;
    }

    showTrunks(trunk);
    cout << " " << ptr->elem << endl;

    if (prev) {
      prev->str = prev_str;
    }
    trunk->str = "   |";

    print_recursion(ptr->left, trunk, false);
  }
}
```

balance.h

```
unsigned char height(Node* p) {
        return p? p -> height: 0;
}

int bfactor(Node* p) {
        return height(p -> right) - height(p -> left);
}

void fixheight(Node* p) {
        unsigned char hl = height(p->left);
        unsigned char hr = height(p->right);
        p -> height = (hl > hr? hl: hr) + 1;
}

// поворот узла (право)
Node* rotateright(Node* p) {
        Node* q = p->left;
        p->left = q->right;
        q->right = p;
        fixheight(p);
        fixheight(q);
        return q;
}

// поворот узла (лево)
Node* rotateleft(Node* q) {
        Node* p = q -> right;
        q -> right = p -> left;
        p -> left = q;
        fixheight(q);
        fixheight(p);
        return p;
```

```
}

Node* balance(Node* p) {
        fixheight(p);
        if (bfactor(p) == 2) {
          if (bfactor(p -> right) < 0)
                  p->right = rotateright(p -> right);
          return rotateleft(p);
        }
        if (bfactor(p) == -2) {
          if (bfactor(p -> left) > 0)
                  p -> left = rotateleft(p -> left);
          return rotateright(p);
        }
        return p;
}

////////////////

Node *findmin(Node *p) {
        return p->left? findmin(p->left): p;
}

Node* removemin(Node* p) {
        if( p->left==0 )
          return p->right;
        p->left = removemin(p->left);
        return balance(p);
}

Node* remove(Node* p, int k) {
        if( !p ) return 0;
        if( k < p->elem )
          p->left = remove(p->left,k);
        else if( k > p->elem )
          p->right = remove(p->right,k);
        else {
          Node* q = p->left;
          Node* r = p->right;
          delete p;
          if( !r ) return q;
          Node* min = findmin(r);
          min->right = removemin(r);
          min->left = q;
          return balance(min);
        }
        return balance(p);
}
```

Результат работы

Вывод

Мы изучили деревья поиска и получили практические навыки их использования.