МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ЛАБОРАТОРНАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: АВЛ Дерево

Студент гр. 7383 _____ Рудоман В.А.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург
2018

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель: Реализовать АВЛ дерево и основные функции для работы с ним. Возможность использовать программу как средство обучения через демонстрацию.

Задача должна быть решена с помощью возможностей языка программирования C++; Программа должна иметь дружественный для пользователя интерфейс.

1. АЛГОРИТМ РАБОТЫ ПРОГРАММЫ

Описание алгоритма вставки:

Элементы дерева вставляют как в обычное упорядоченное дерево (справа от элемента – элементы большие, а слева от элемента элементы меньшие). Но есть отличие от обычного упорядоченного бинарного дерева. По возвращению из рекурсии дерево преобразуется таким образом чтобы максимальное различие высоты между двумя его любыми ветвями было не более чем 1 – это обеспечивает логарифмическую сложность. Эти преобразования происходят если у элемента дерева `balance` не лежит в пределах [-1;1]. Здесь различаются 4 разных ситуации:

1) нарушение высоты в случае: лево-лево

2) нарушение высоты в случае: лево-право

3) нарушение высоты в случае: право-право

4) нарушение высоты в случае: право-лево

Все эти ситуации разрешаются с помощью левых и правых поворотов деревьев. Повороты БД изображены на рис. 1.

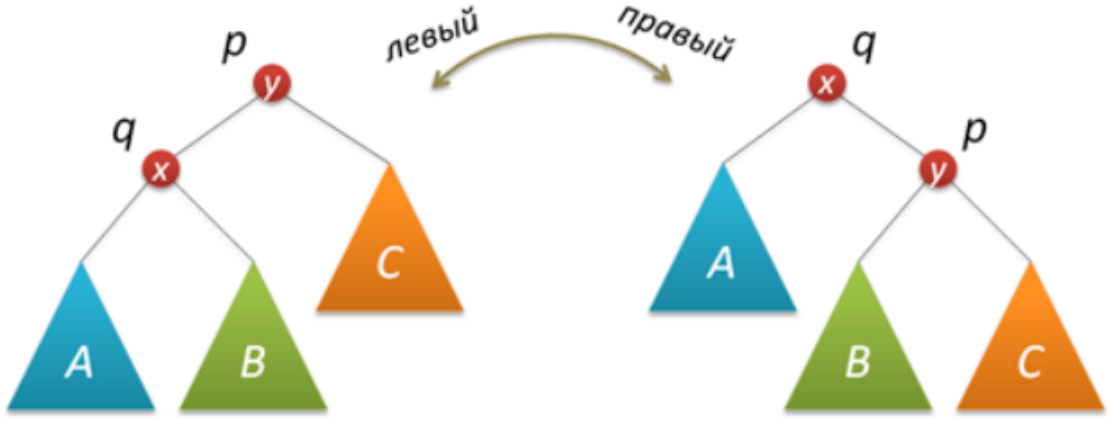


Рисунок 1 – повороты БД

Картинка описывает повороты бинарного дерева.

Описание алгоритма поиска элемента в АВЛ_дереве:

Поиск элемент в АВЛ_дереве происходит как в обычном упорядоченном дереве. Если мы узел, в котором мы находимся больше заданного значения

переходим к правому сыну, если меньше заданного значения переходим к левому сыну. Алгоритм заканчивает свою работу в 2-х случаях:

1) Если значения узла, в котором мы находимся равно заданному значению
2) Если мы дошли до листа и не встретили элемента равного нашему

Алгоритм удаления элемента из АВЛ дерева:

Находим заданный для удаления элемент в дереве. Создаем новое дерево с корнем со значением минимального элемента в правом сыне исходного дерева. Левый сын нового – левый сын старого. Правый сын нового – старый правый сын без минимального элемента. По выходу из рекурсии обновляем высоту и балансировку деревьев и при необходимости корректируем высоту дерева.

2. ОПИСАНИЕ ФУНКЦИЙ И СТРУКТУР ДАННЫХ

1) `Class Head_AVL_Tree` – реализована работа с АВЛ-дерево.
   A) Методы класса:
      a) `void insert(Type)` – вставка элемента в АВЛ-дерево.
      b) `void print_tree()` – вывод АВЛ-дерева на экран
      c) `bool is_contain(Type)` – возвращает true, если заданный элементы содержится в АВЛ-дереве.
      d) `Head_AVL_Tree()` – конструктор класса. Инициализирует данные.
      e) `~Head_AVL_Tree()` – деструктор класса. Очищает память выделенную под АВЛ-дерево.
      f) `void remove(Type)` – удаление заданного элемента.
   B) Данные класса:
      a) `class Node_AVL_Tree<Type>* head` – содержит указатель на АВЛ-дерево.
2) `class Node_AVL_Tree` – Реализована работа с элементами АВЛ-дерева.
   A) Методы класса:
      a) `bool is_contain(Type)` - возвращает true, если заданный элементы содержится в АВЛ-дереве.
      b) `int set_height()` – устанавливает высоту данного элемента АВЛ-Дерева.
      c) `int get_balance()` – Получает значение баланса для заданного элемента АВЛ-дерева.
      d) `void print_tree(int)` – выводит на экран АВЛ-дерево
      e) `class Node_AVL_Tree<Type>* insert(Type)` – вставка в АВЛ-дерево
      f) `class Node_AVL_Tree<Type>* left_rotate()` – левое вращение

g) `class Node_AVL_Tree<Type>* right_rotate()` – правое вращение

h) `class Node_AVL_Tree<Type>* make_balance()` – установка баланса путем вращения дерева вокруг элемента

i) `Node_AVL_Tree()` – конструктор класса. Инициализирует данные.

j) `~Node_AVL_Tree()` – деструктор класса. Очищает память выделенную под АВЛ-дерево.

k) `class Node_AVL_Tree<Type>* remove(Type)` - Функция удаления заданного элемента из дерева.

l) `class Node_AVL_Tree<Type>* remove_min()` – Функция удаления минимального элемента.

m) `class Node_AVL_Tree<Type>* find_min()` – Нахождение минимального элемента в дереве.

B) Данные класса:

a) `int height` – определяет высоту дерева.

b) `int balance` – определяет баланс дерева.

c) `Type data` – данные дерева.

d) `class Node_AVL_Tree<Type>* left` – левый сын

e) `class Node_AVL_Tree<Type>* right-` правый сын

3) Вспомогательные функции:

A) `void make_deep(int, int)` - вспомогательная функция для вывода АВЛ-дерева на экран.

B) `void make_hight(int)` – вспомогательная функция для вывода АВЛ-дерева на экран.

3. ТЕСТИРОВАНИЕ

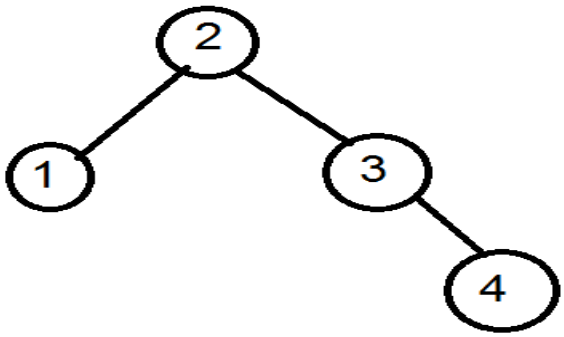
Тестирование:

Входные данные: 1 2 3

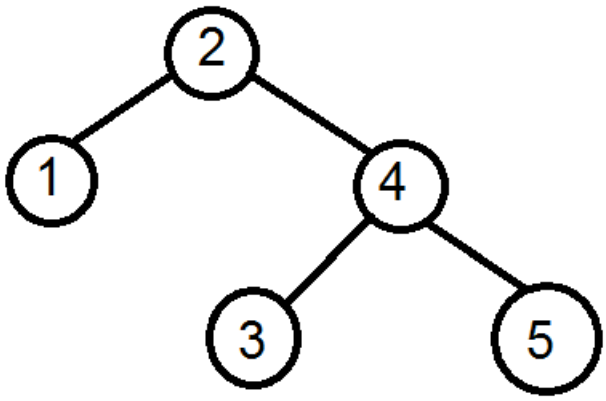
Рассматриваемые входные данные в обычном БД расположились бы линейно, что увеличило бы количество итераций в поисковом алгоритме.

Допустим, на добавление подано число 2. Тогда программа выведет сообщение “element already in tree”.

Рассмотрен другой случай. На вход подано число 4, дерево будет выглядеть следующим образом:



Для более подробного разбора теста добавлено еще одно значение: 5



Произошел поворот вправо вокруг 3

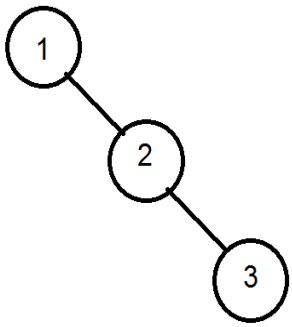
Всего различаются 4 различных случая для вставки элемента:

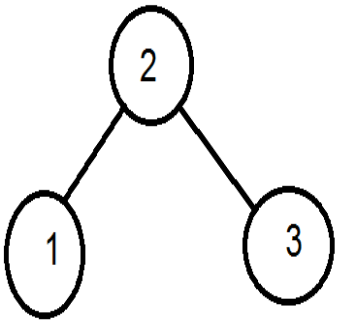
На следующих рисунках: левое дерево – исходное дерево.

правое дерево – видоизменённое дерево.

1) Balance == 2, right->balance == 1

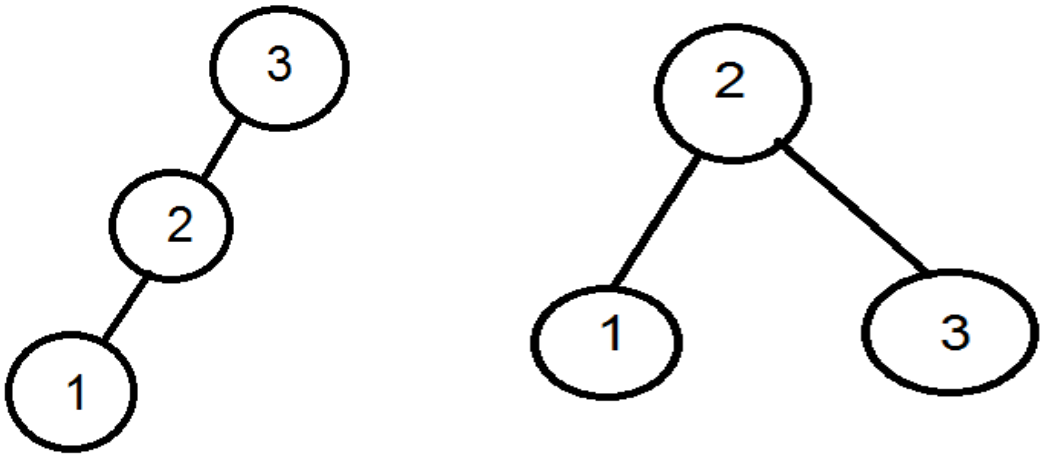
   Решается с помощью левого поворота вокруг 1





2) Balance == -2, left->balance == -1

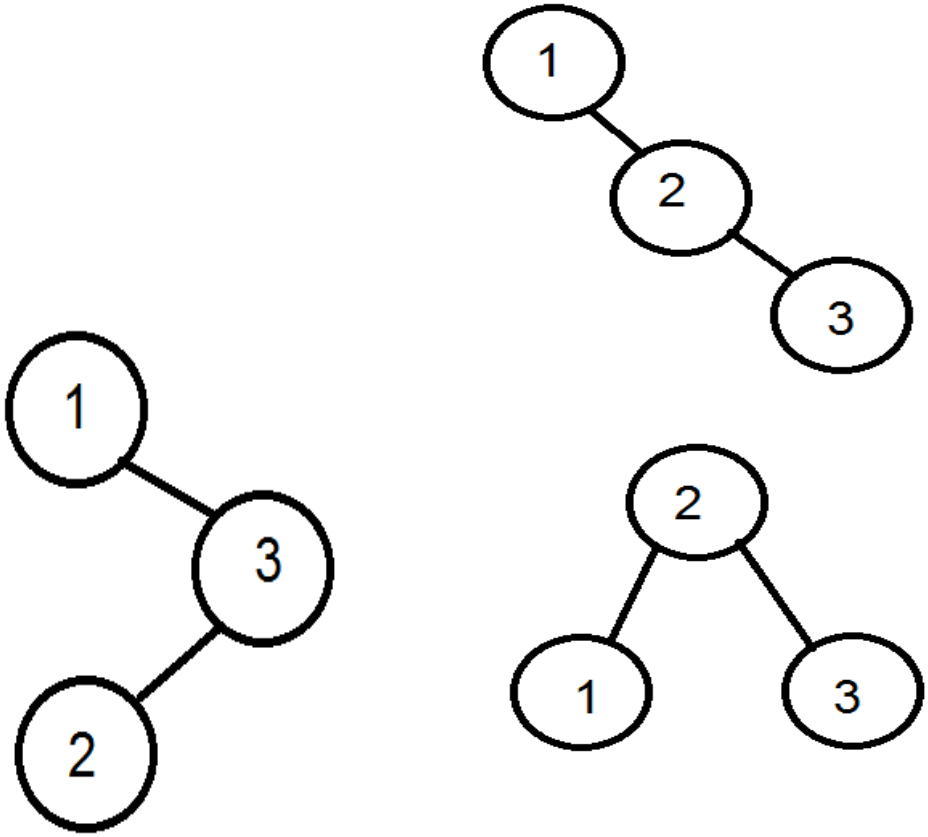
Решается с помощью правого поворота вокруг 3



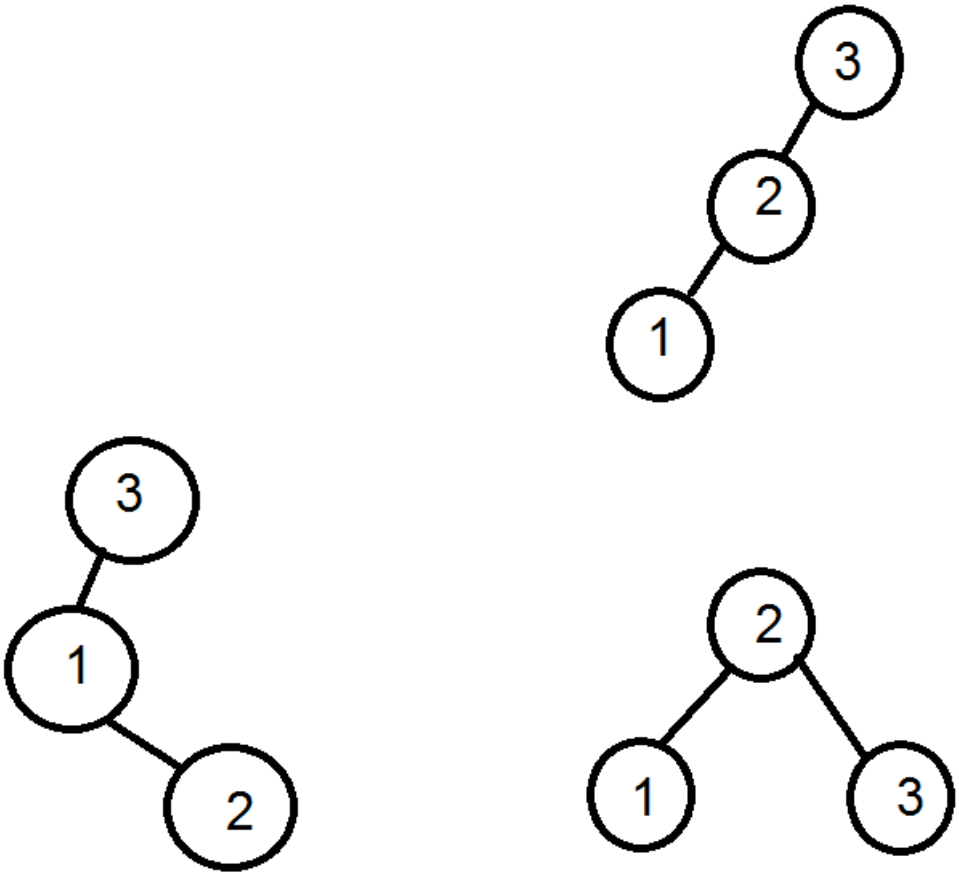
3) Balance == 2 right->balance == -1

Решается с помощью двух поворотов:

Сначала вокруг 3, затем вокруг 1

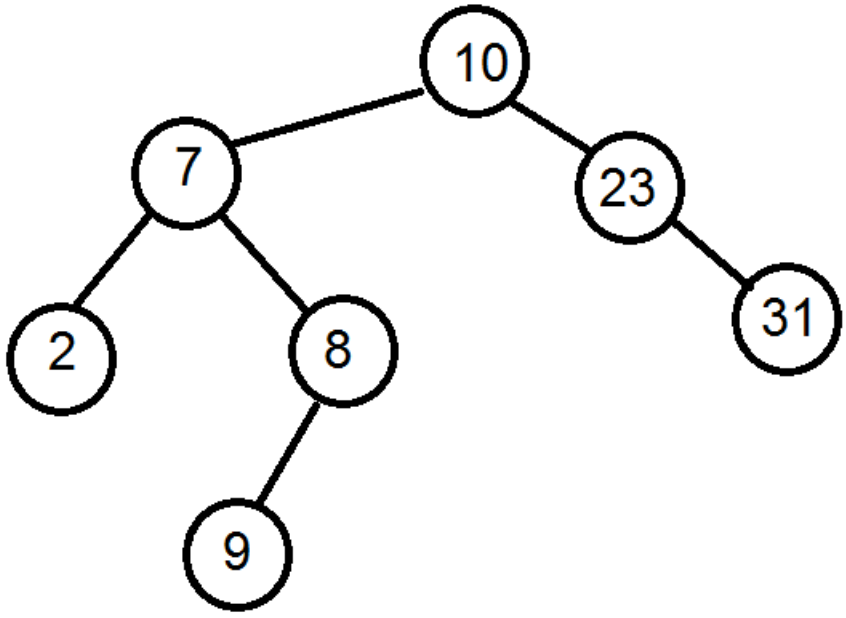


Balance == -2, left->balance == 1



Теперь протестируется удаление элемента из АВЛ дерева:

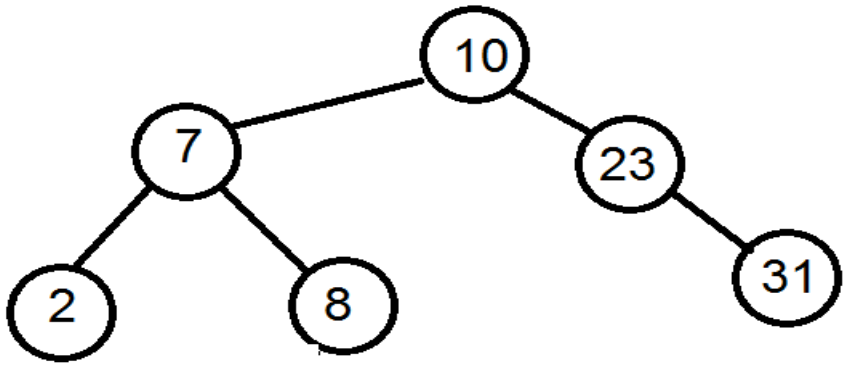
Допустим имеется АВЛ дерево:



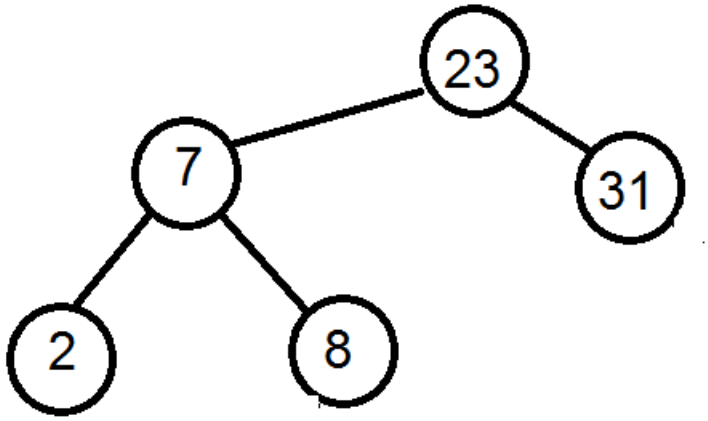
При попытке удалить элемент, который не содержится в дереве (например, 123) – будет выведено сообщение: `tree isn`t have this element.`

При удаление терминального элемента (например, 9) – элемент просто удалиться из дерева, и дерево по возвращению из рекурсии будет восстанавливать свою высоту в каждом элементе и балансироваться.

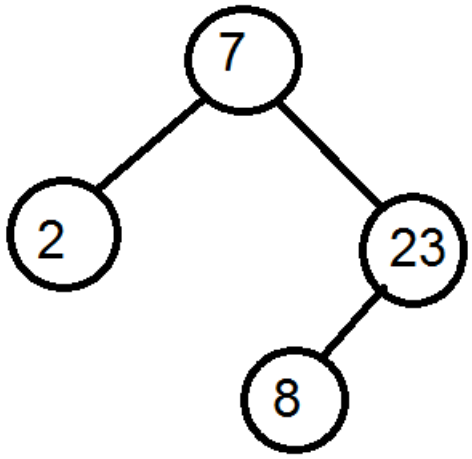
Дерево после удаления: 9



Дерево после удаления корня: 10



Дерево после удаления: 31



Таким образом, в тестирование рассмотрены различные случаи удаления из дерева.

Приведены некоторые ошибки, которые могут возникнуть при работе программы. Данные ошибки приведены в табл. 1.

Таблица 1 – Реакция программы на некорректные данные

Возможные ошибки	Реакция программы на некорректные входные данные
Неверно указано название файла	`ошибка открытия файла`
Не поступает ни одного значение из файла	`no one element to tree. Program will end`
Ошибка ввода при вводе элемента в дерево на добавление	`input error. May be it`s not a number or it`s very big number`
Ошибка ввода при вводе значения на предложение о продолжение программы	`input error. May be it`s not a number or it`s very big number`
Попытка добавить элемент в дерево, который уже находиться там	`element already in tree`
Попытка вывести пустое дерево	`Tree is empty`
Попытка удалить элемент, которого нет в дереве	tree isn`t have this element

ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС

Для удобства работы с программой реализовано меню выбора действий. Меню представлено на рис. 1.

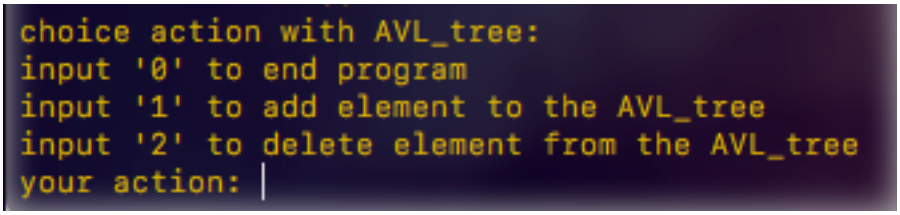


Рисунок 1 – Пользовательское меню

Выбор «1» означает добавить элемент в дерево. Выбор «2» означает удалить элемент из дерева. Выбор «0» выход из программы.

При выборе «1» для начала программа ищет в дереве заданный элемент и если его нет, то добавляет в дерево, иначе элемент в дерево не добавляется так как уже присутствует в нем. Процесс поиска элемента представлен на рис. 2.

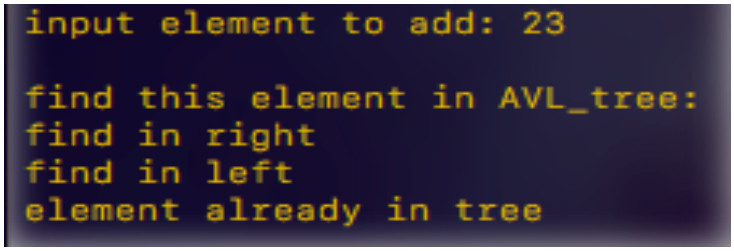


Рисунок 2 – процесс поиска заданного элемента

При выборе «2» в пользовательском меню для начала программа, как и при вставке элемента в дерево, определяет присутствие заданного элемента в АВЛ дереве. Если элемента в дереве нет, то и удалять из него нечего. Если элемент присутствует в дереве то происходит его удаление.

Также после совершения каждого из действий выводится измененное дерево в виде псевдографики. Его изображение представлено на рис. 3.

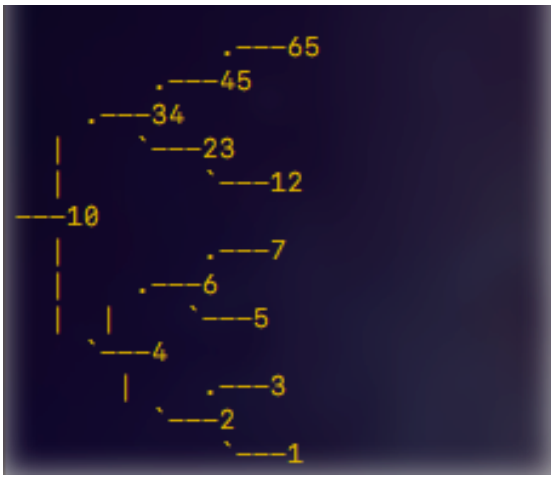


Рисунок 3 – изображение дерева

ЗАКЛЮЧЕНИЕ

В работе изучена структура данных сбалансированное дерево, на примере АВЛ-дерева. Сбалансированные деревья позволяют выполнять вставку, удаление и поиск заданного элемента с логарифмической асимптотической сложностью. В отличие от деревьев. Маловероятно, что дерево из N элементов будет иметь высоту N и все же такая вероятность возможна. Тогда вставка, поиск и удаление элементов будет происходить с линейной сложностью, и такие деревья будут уступать по скорость элементарному линейному списку.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1) Вирт Н. “Алгоритмы и структуры данных”
2) Кнут Д. “Искусство программирования”
3) Род Стивенсон “Алгоритмы”

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ФАЙЛА MAIN.CPP

```
#include <iostream>
 #include <fstream>
 #include "avl_tree.h"

 typedef int Type;
 #define FROM_FILE // input from file

 int main(int argc, char* argv[])
 {
     Head_AVL_Tree<Type> head;
     #ifdef FROM_FILE
     std::ifstream fin;
     if(argc == 2) // если файл как аргумент командной строки
         fin.open(argv[1], std::ios::in);
     else{
         std::string file_name;
         std::cout << "input filename" << std::endl;
         std::cin >> file_name;
         fin.open(file_name, std::ios::in);
     }
     if(!fin){
         std::cout << "ошибка открытия файла" << std::endl;
         return 0;
     }
     #endif

     std::string main_str;
     #ifdef FROM_FILE
         getline(fin, main_str);
         std::cout << std::endl << "create AVL_Tree from file " <<
std::endl; // считываем входную строку
     #else
         std::cout << "input your element of AVL_tree: " << std::endl;
         getline(std::cin, main_str);
     #endif

     int count_num = 0;
     for(size_t i = 0; i < main_str.size(); i++){
         int temp = 0;
         bool is_num = false;
         for(size_t j = i; isdigit(main_str[j]); j++, i++){ // create
tree from string
             temp = temp*10 + main_str[j]-'0';
             is_num = true;
         }
         if(is_num){
             count_num++;
```

```
            std::cout << "add to the tree: " << temp << std::endl;
            head.insert(temp);
            std::cout << "tree after added " << temp << std::endl;

            std::cout << std::endl;
           head.EnterPrintTree();
        }

     }
    //head.print_tree();

     if(!count_num){
         std::cout << "no one element to tree. Program will end" <<
std::endl;
         exit(0);
     }
     if(argc != 2){
         std::cout << "choice action with AVL_tree: " << std::endl <<
"input '0' to end program " << std::endl <<
                "input '1' to add element to the AVL_tree" << std::endl
<<
                "input '2' to delete element from the AVL_tree" <<
std::endl;
     }

     int action = 0;
     if(argc != 2)
      std::cout << "your action: ";
     std::cin >> action;
     if(std::cin.fail()){
         std::cout << "input error. May be it`s not a number or it`s very
big number" << std::endl;
         return 0;
     }

     while(1){
         switch (action) {
             case 0:
                 #ifdef FROM_FILE
                     fin.close(); // закрываем файл
                 #endif
                 return 0;
                 break;
             case 1:
                 Type insert_tree_element;
                 if(argc != 2)
                     std::cout << "input element to add: ";
                 std::cin >> insert_tree_element;
                 if(argc == 2)
                     std::cout << "add to tree new element: " <<
insert_tree_element  << std::endl;
```

```
                if(std::cin.fail()){
                    std::cout << "input error. May be it`s not a number
or it`s very big number" << std::endl;
                    return 0;
                }
                std::cout << std::endl << "find this element in
AVL_tree: " << std::endl;
                if(!head.is_contain(insert_tree_element)){
                    std::cout << "tree does not have this element" <<
std::endl << std::endl;
                    head.insert(insert_tree_element);
                    std::cout << "tree after add new element:" <<
std::endl;
                    head.EnterPrintTree();
                }
                else
                    std::cout << "element already in tree" << std::endl
<< std::endl;
                if(argc != 2)
                     std::cout << "input next action: ";
                std::cin >> action;
                      if(std::cin.fail()){
                    std::cout << "input error. May be it`s not a number
or it`s very big number" << std::endl;
                    return 0;
                }
                break;
            case 2:
                if(argc != 2)
                    std::cout << "input element to delete: " <<
std::endl;
                Type element_to_delete;
                std::cin >> element_to_delete;
                if(std::cin.fail()){
                    std::cout << "input error. May be it`s not a number
or it`s very big number" << std::endl;
                    return 0;
                }
                      std::cout << "delete from tree: " <<
element_to_delete << std::endl;
                std::cout << std::endl << "find this element in
AVL_tree: " << std::endl;
                if(!head.is_contain(element_to_delete)){
                    std::cout << "tree does not have this element" <<
std::endl << std::endl;
                }
                else{
                    head.remove(element_to_delete);
                    std::cout << std::endl << "tree after delete: " <<
element_to_delete << std::endl;
                    head.EnterPrintTree();
```

```
                }
            if(argc != 2)
                std::cout << "input next action: ";
                std::cin >> action;
             if(std::cin.fail()){
                    std::cout << "input error. May be it`s not a number
or it`s very big number" << std::endl;
                    return 0;
                }
                break;
              default:
                std::cout << "It`s not an action. Repeat please" <<
std::endl;
                if(argc != 2)
                std::cout << "input next action: ";
                std::cin >> action;
                if(std::cin.fail()){
                    std::cout << "input error. May be it`s not a number
or it`s very big number" << std::endl;
                    return 0;
                }
                break;
          }
     }
     return 0;
 }
```

ПРИЛОЖЕНИЕ Б
ИСХОДНЫЙ КОД ФАЙЛА AVL_TREE.H

```
#ifndef AVL_TREE_H // for include only one time
 #define AVL_TREE_H

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

 void make_deep(int, int);
 void make_hight(int);

 template <class Type>
 class Head_AVL_Tree;

 template <class Type>
 class Node_AVL_Tree{
 public:
     friend class Head_AVL_Tree<Type>; // дружим с головой
     bool is_contain(Type, int);
     int set_height();
     int set_balance();
     void print_tree();
     void print_tree(int);
     class Node_AVL_Tree<Type>* insert(Type);
     class Node_AVL_Tree<Type>* remove(Type);
     class Node_AVL_Tree<Type>* remove_min();
     class Node_AVL_Tree<Type>* find_min();
     class Node_AVL_Tree<Type>* left_rotate();
     class Node_AVL_Tree<Type>* right_rotate();
     void printTree(Trunk *, bool);
     class Node_AVL_Tree<Type>* make_balance();
     Node_AVL_Tree();
     ~Node_AVL_Tree();
 private:
     int height;
     int balance;
     Type data;
     class Node_AVL_Tree<Type>* left;
```

```
    class Node_AVL_Tree<Type>* right;
};

template <class Type>
class Head_AVL_Tree{
public:

    Head_AVL_Tree();
    ~Head_AVL_Tree();
    void insert(Type);
    void EnterPrintTree();
    bool is_contain(Type);
    void remove(Type);
private:
    class Node_AVL_Tree<Type>* head;
};

template <class Type>
Node_AVL_Tree<Type>::Node_AVL_Tree(){
    left = nullptr;
    right = nullptr;
}

template <class Type>
Node_AVL_Tree<Type>::~Node_AVL_Tree(){ // очищаем память под дерево
    if(left)
        delete left;
    if(right)
        delete right;
}

template <class Type>
bool Node_AVL_Tree<Type>::is_contain(Type desired, int depth){
    if(data == desired)
        return true;
    if(left && data > desired){
        std::cout << "find in left" << std::endl;
        if(left->is_contain(desired, depth+1))
            return true;
    }
    if(right && data < desired){
        std::cout << "find in right" << std::endl;
        return right->is_contain(desired, depth+1);
    }
```

```
    return false;
}

template <class Type>
void Node_AVL_Tree<Type>::print_tree(){ // печатает информацию об узлач
дерева
    std::cout << data << " ";
    std::cout << height << " ";
    std::cout << balance << std::endl;
    if(left)
        left->print_tree();
    if(right)
        right->print_tree();
}

template <class Type>
void Node_AVL_Tree<Type>::print_tree(int deep){ // beautiful print tree
    make_hight(deep);
    std::cout << data << std::endl;
    if(right){
        right->print_tree(deep+1);
    }
    if(left){
        left->print_tree(deep+1);
    }
}

template <class Type>
class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::remove(Type to_remove){
    if(data == to_remove){
        if(!left && !right){
            delete this;
            return nullptr;
        }
        if(!right){
            delete this;
            return left;
        }
        class Node_AVL_Tree<Type>* new_root;
        new_root = right->find_min();
        right = right->remove_min();
        new_root->left = left;
        new_root->right = right;
        new_root->height = set_height();
```

```
        new_root->balance = set_balance();
        return new_root->make_balance();
    }
    if(data < to_remove)
        right = right->remove(to_remove);
    if(data > to_remove)
        left = left->remove(to_remove);
    height = set_height();
    balance = set_balance();
    return make_balance();
}

 template <class Type>
 class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::find_min(){
    return left?left->find_min():this;
 }

 template <class Type>
 class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::remove_min(){
    if(!left){
        class Node_AVL_Tree<Type>* temp = right;
        this->right = nullptr;
        delete this;
        return temp;
    }
    left = left->remove_min();
    height = set_height();
    balance = set_balance();
    return make_balance();
 }

 template <class Type>
 class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::insert(Type value){ //
вставка
    if(value >= data){
        if(!right){
            right = new Node_AVL_Tree<Type>; // находим нужный узел для
вставки как в обычном упорядоченном дереве
            right->data = value;
            right->height = 1;
        }
        else
            right = right->insert(value);
    }
```

```
    if(value < data){
        if(!left){
            left = new Node_AVL_Tree<Type>;
            left->data = value;
            left->height = 1;
        }
        else
            left = left->insert(value);
        }
    height = set_height();
    balance = set_balance();
    return make_balance(); // балансировке по возврату из рекурсии
 }

 template <class Type>
 class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::right_rotate(){
    std::cout << "right rotate around element: " << this->data <<
std::endl;
    Node_AVL_Tree<Type>* temp;
    temp = left;
    left = temp->right;
    this->height = this->set_height();
    this->balance = this->set_balance();
    if(temp->left){
        temp->left->height = temp->left->set_height();
        temp->left->balance = temp->left->set_balance();
    }
    temp->right = this;
    temp->height = temp->set_height();
    temp->balance = temp->set_balance();
    return temp;
 }

 template <class Type>
 class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::left_rotate(){
    std::cout << "left rotate around element: " << this->data <<
std::endl;
    Node_AVL_Tree<Type>* temp;
    temp = right;
    right = temp->left;
    this->height = this->set_height();
    this->balance = this->set_balance();
    if(temp->right){
        temp->right->height = temp->right->set_height();
```

```
        temp->right->balance = temp->right->set_balance();
    }
    temp->left = this;
    temp->height = temp->set_height();
    temp->balance = temp->set_balance();
    return temp;
}

template <class Type>
class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::make_balance(){
    Node_AVL_Tree<Type>* temp;
    temp = this;
    if(balance == 2){
        if(right->balance == -1)
            temp->right = right->right_rotate(); // случай право право
        temp = left_rotate(); // случай лево лево
    }
    if(balance == -2){
        if(left->balance == 1)
            temp->left = left->left_rotate(); // случай право лево
        temp = right_rotate(); // случай право право
    }
    return temp;
}

template <class Type>
int Node_AVL_Tree<Type>::set_height(){ // установка высоты
    if(!left && !right) // if sheet
         return 1;
    if(!left)
        return (right->height + 1);
    if(!right)
        return (left->height + 1);
    if(left->height >= right->height)
        return (1 + left->height);
    if(left->height < right->height)
        return (1 + right->height);
    return 0; // чтобы компилятор не ругался
}

template <class Type>
int Node_AVL_Tree<Type>::set_balance(){ // установка баланса в вершине
    if(!left && !right) // if sheet
         return 0;
```

```
    if(!left)
        return right->height;
    if(!right)
        return (left->height * (-1));
    return (right->height - left->height);
}

template <class Type>
Head_AVL_Tree<Type>::Head_AVL_Tree(){
    head = nullptr;
}

template <class Type>
Head_AVL_Tree<Type>::~Head_AVL_Tree(){
    delete head;
}



template <class Type>
void Head_AVL_Tree<Type>::insert(Type value){
    if(!head){
        Node_AVL_Tree<Type>* temp = new Node_AVL_Tree<Type>;
        temp->data = value;
        temp->height = 1;
        head = temp;
        return;
    }
    head = head->insert(value);
}

template <class Type>
bool Head_AVL_Tree<Type>::is_contain(Type desired){
    if(!head)
        return false;
    if(head->data == desired){
     std::cout << "this element is root" << std::endl;
        return true;
    }
    if(head->left && head->data > desired){
        std::cout << "find in left " << std::endl;
        return head->left->is_contain(desired, 1);
    }
    if(head->right && head->data < desired){
```

```
            std::cout << "find in right " << std::endl;
            return head->right->is_contain(desired, 1);
        }
        return false;
    }

    template <class Type>
    void Head_AVL_Tree<Type>::remove(Type to_remove){
        head = head->remove(to_remove);
    }



// Helper function to print branches of the binary tree
void showTrunks(Trunk *p) {
    if (p == nullptr)
        return;
    showTrunks(p->prev);
    std::cout << p->str;
}

template <class Type>
void Node_AVL_Tree<Type>::printTree(Trunk *prev, bool isLeft){
    if (this == nullptr)
        return;

    std::string prev_str = "    ";
    Trunk *trunk = new Trunk(prev, prev_str);
    left->printTree(trunk, true);

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

    showTrunks(trunk);
    std::cout << data << std::endl;
    if (prev)
```

```
        prev->str = prev_str;

    trunk->str = "   |";
    right->printTree(trunk, false);
}

template <class Type>
void Head_AVL_Tree<Type>::EnterPrintTree(){
    head->printTree(nullptr, false);
}


 #endif
```