EADS LAB1
AVL tree
Done by Filip Korzeniewski

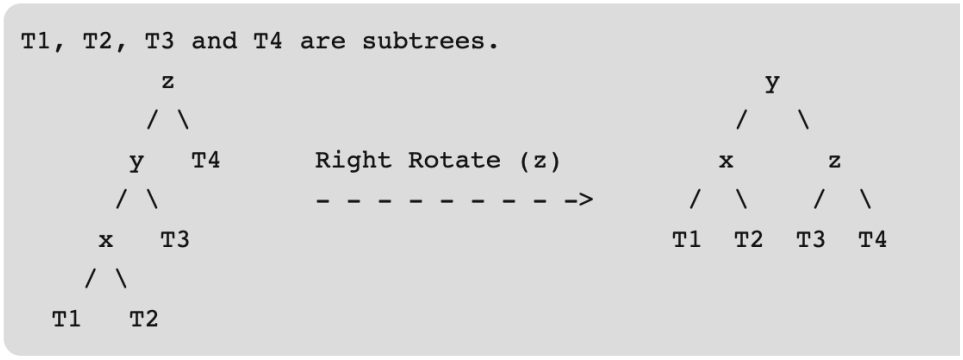
General scheme of project.

The task was to create AVLtree with typical functions. The functions are removing elements, adding elements, finding, getting data and some that made it easier implement. Generally, most of functions are done recursively. It is much easier to implement them this way.

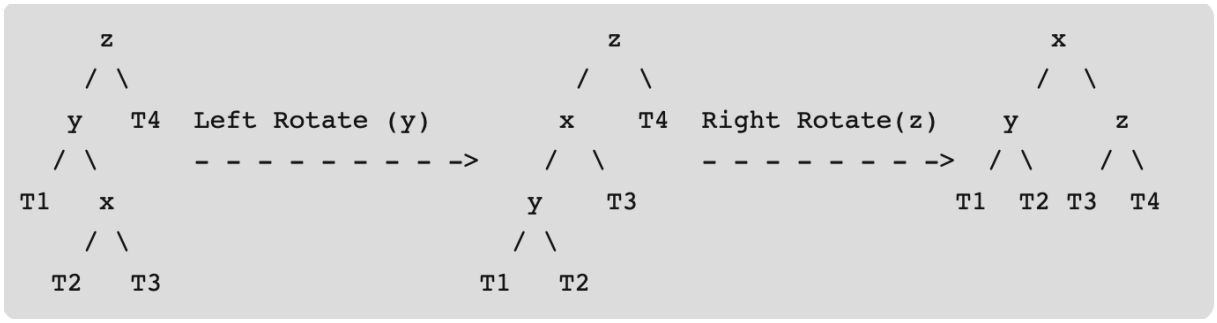
AVL tree is a kind of structure where everything is stored in a structure like binary tree but the difference of heights of every branch must not be higher than 1. The tree must be rebalanced then every time when the height starts to vary.

There are 4 different cases of rebalancing the tree:

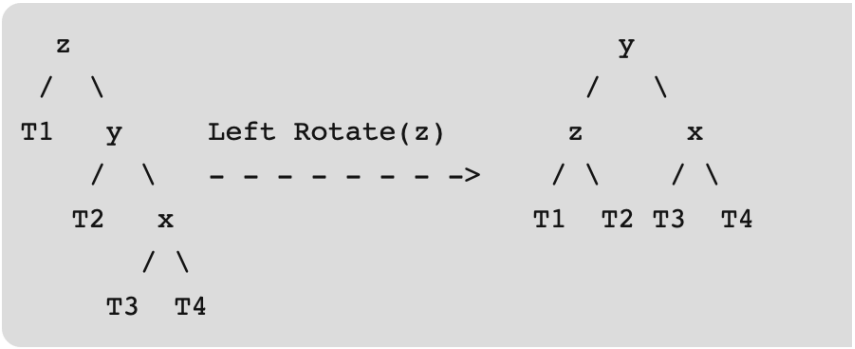
Left-left case:



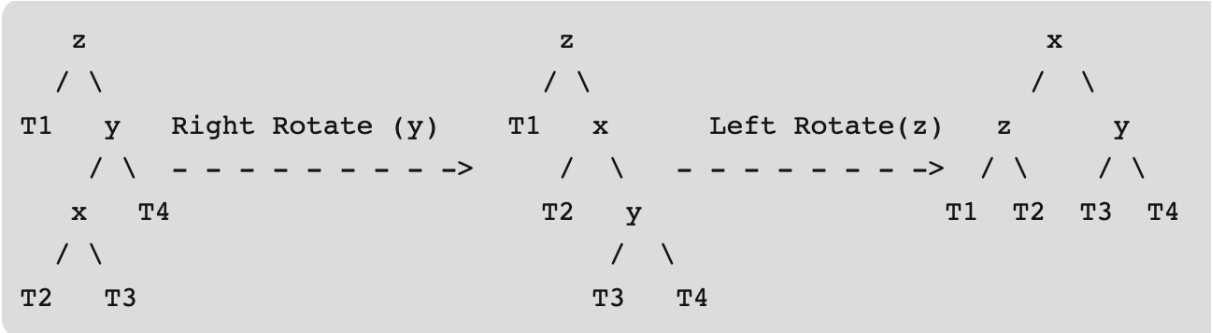
Left-right case:



Right-right case:



Right-left case:



The outline:

AVLtree:

```cpp
template <typename Key, typename Info>
class AVLtree{
private:
    struct Node{
        Key key;
        Info info;
        Node* left;
        Node* right;
        int height;
        Node(Key _key, Info _info): key(_key), info(_info),
            left(nullptr), right(nullptr), height(1) {};
    };

    Node* root;

    Node* mostLeft(Node* temp){Node* curr = temp;
        while(curr->left){
            curr = curr->left;
        } return curr;
    };
    int max(int a, int b){ return (a>b) ? a : b; };
    int height(Node* temp){ if(!temp){return 0;} return temp->height; };
        //if null doesn't crush, just returns 0
    int getBalance(Node* temp); //gets balance factor of subtree

    Node* RightRotate(Node* temp); //rotate right subtree around temp
    Node* LeftRotate(Node* temp); //rotate left substree around temp

    Node* insert(Node* temp, const Key& _key, const Info& _info);
        //recursively inserts node;
    Node* find(Node* temp, const Key& _key); //reccursively
    Node* remove(Node* &temp, const Key& _key);
    int updateHeight(Node* temp);

    void print(Node* temp, int space) const;
    void printIn(Node* temp) const;
    void clearAll(Node* temp);

    void copyTree(Node* temp);
    void copyTree(const AVLtree& x);
public:
    void insert(const Key& _key, const Info& _info);
    bool remove(const Key& _key);
    bool find(const Key& _key);

    void print() const;
    void printIn() const;
```

```cpp
    Node* root;

    Node* mostLeft(Node* temp){Node* curr = temp;
        while(curr->left){
            curr = curr->left;
        } return curr;
    };
    int max(int a, int b){ return (a>b) ? a : b; };
    int height(Node* temp){ if(!temp){return 0;} return
        temp->height; }; //if null doesn't crush, just returns 0
    int getBalance(Node* temp); //gets balance factor of subtree

    Node* RightRotate(Node* temp); //rotate right subtree around t
    Node* LeftRotate(Node* temp); //rotate left substree around te

    Node* insert(Node* temp, const Key& _key, const Info& _info);
        //recursively inserts node;
    Node* find(Node* temp, const Key& _key); //reccursively
    Node* remove(Node* &temp, const Key& _key);
    int updateHeight(Node* temp);

    void print(Node* temp, int space) const;
    void printIn(Node* temp) const;
    void clearAll(Node* temp);

    void copyTree(Node* temp);
    void copyTree(const AVLtree& x);
public:
    void insert(const Key& _key, const Info& _info);
    bool remove(const Key& _key);
    bool find(const Key& _key);

    void print() const;
    void printIn() const;

    void clearAll();

    AVLtree<Key, Info>& operator=(const AVLtree<Key, Info>& x);
    AVLtree<Key, Info>& operator+=(const AVLtree<Key, Info>& x);
    AVLtree<Key, Info> operator+(const AVLtree<Key, Info>& x);


    AVLtree(): root(nullptr){};
    AVLtree(const AVLtree<Key, Info>& x);
    ~AVLtree(){ clearAll(); };

};
```

Tests:

```cpp
void count_words_test();
void insert_remove_test();
void rotate_test();
void operator_constructor_test();
void find_test();
void map_compare_test();
```

Brief description

AVLtree stores data the way as it has been already claimed. Moreover, there was used a function count_words that counts words in file and returns it but the main usage of it was to insert all words of text file into AVLtree.

A few words about tests (comparison to map.h)

Tests covered many cases that class AVLtree could have failed. Furthermore, and what is the most important – they show working of whole project. Everything works as it should. Tests checks working of insertion and deletion, finding elements, rotations, working of operators and constructors and finally comparing the algorithm to the built-in one – map.

The final test checked and compared the time needed to proceed insertion into AVLtree class structure and using built-in map structure. It compared also time needed to find any element in structure. It turned out that insertion is a bit faster with an AVLtree algorithm.

Finding can be unreliable because of the fact that searched data can be stored in different “depth” in both algorithms. It is proven by the fact that for test where 2 words being looked for, word “where” was found in 2ms using map.h and 18ms using AVLtree while word “Muggle” in 20ms using map.h and 9ms using AVLtree class.

```
--------------------------
COMPARING WITH MAP.H TEST

there are 83187 words in text
Looking for "where" word:
AVL tree function:
found element:
key: where info: 98

For "map.h":
key: where info: 98

Measured time of inserting elements:
For AVLtree class: 213214
For "map.h": 211380

Measured time of finding a "where" word:
For AVLtree class: 18
For "map.h": 2
Looking for "Muggle" word:
AVL tree function:
found element:
key: Muggle info: 1

For "map.h":
key: Muggle info: 1

Measured time of finding an element:
For AVLtree class: 9
For "map.h": 20
Program ended with exit code: 0
```