AVL Tree documentation

Idea:

This project is a dictionary based on AVL Tree data structure. The node on which the structure bases has 3 pointers: for the parent node and both children - the left and right one. What is more, there are operators such as “<<” and “[ ]” implemented for printing the tree in order and indexing it, respectively. Also, “()” is an operator responsible for printing the tree and “ =” is the assignment operator.
The program shows graphical representation of the outcoming structure of the AVL Tree.

Class design:

The Dictionary template class contains of structure Node which is the basic element of the AVL tree. Node has three pointers - left, right and parent and two variables: id and data of type key and info, respectively. Another element of the structure is a constructor of a single node. Node is a private element of a TreeDictionary, together with the pointer of type Node to root of the tree and all the methods in which there are basic functionality functions for a tree like: rotations, balancing and printing. The class itself is shown below:

```
class TreeDictionary {
      struct Node {
            key id;
            info data;
            Node *parent;
            Node *left;
            Node *right;
            int height;
      };
      Node *root;
...
```

Methods used:

- `void add(key keyy, info data, Node *node)` – adds new element to the tree
- `void remove(key keyy)` – removes node of a given key
- `Node *newNode(key keyy, info data)`– creates new Node of given key and info
- `void erase(Node *node)` – removes all elements of the tree
- `Node *add(key keyy, info data, Node *node)` – adds new node to the tree
- `Node *rotateRight(Node *&node)`
- `Node *RotateLeft(Node *&node)`
- `Node *rotateLeftTwice(Node *&node)`
- `Node *rotateRightTwice(Node *&node)` – rotations used for balancing the tree
- `Node *findMinInSubTree(Node *node)` – returns the pointer to the minimal value of the key in tree
- `Node *findMaxInSubTree(Node *node)` – returns the pointer to the maximal value of the key in tree
- `Node *remove(key keyy, info data, Node *node)`– removes a node from the tree
- `int getBalance(Node *node)` – used to ensure if the nodes balance factor is sustained
- `void display(Node *node, int free)` – used to print the tree
- `Iterator operator=(const Iterator &iterator1)` – assignment operator
- `Iterator &operator++()` - moves iterator element by one to the next element
- `const Iterator operator++(int)`– moves iterator element to the next
- `Iterator &operator--()` – moves iterator by one to previous element
- `const Iterator operator--(int)` – moves element to previous one
- `bool operator==(Iterator iterator1) const` – checks if particular elements are equal
- `bool operator!=(Iterator iterator1) const` - checks if particular elements are different
- `Iterator operator+(int length)` – moves the element by the value of length in increasing direction
- `Iterator operator-(int length)` – moves the element by the value of length in decreasing direction
- `friend ostream &operator<<(ostream &o, const Iterator &iter)` – operator used to print the tree
- `TreeIterator find(const key &keyy)`– finds particular element in the tree
- `void print()` – prints the tree
- `info operator[](key keyy)` – used for indexing the tree

- `key operator()(info data)` – allows to access ikey of a given info from the tree
- `friend bool operator==(TreeDictionary &tree1, TreeDictionary &tree2)` – comparison operator
- `TreeDictionary& operator=(const TreeDictionary &tree)-` assignment operator