# AVL tree based dictionary - an EADS project

Maciej Marcinkiewicz

January 2020

## 1 Introduction

The aim of the project is to implement an AVL tree based dictionary in the C++ programming language with use of template feature. As it is dictionary, the main class has two generic parameters – `Key` and `Info`. We assume that `Key` is unique and if new data with already existing key is being aded, `Info` is replaced with new value. Additional requirement is to print the structure in a way that root is on the left hand side of screen and leaves are on the right hand side.

## 2 Overview of funtions

### 2.1 Constructors and destructor

```
Tree()
```

Default constructor.

```
~Tree()
```

Destructor.

```
Tree(const Tree<Key, Info> &other)
```

Copy constructor.

```
Node(const Key &key, const Info &info)
```

Constructor for `Node`. Creates new `Node` object with given key and information.

```
~Node()
```

Destructor for `Node`.

### 2.2 Overloaded operators

```
Tree<Key, Info> &operator=(const Tree<Key, Info> &other)
```

Assignment operator.

```
bool operator==(const Tree<Key, Info> &other)
```

Comparison operator. Returns `true` if compared trees contain the same elements.

```
bool operator!=(const Tree<Key, Info> &other)
```

Comparison operator. Returns `false` if compared trees contain the same elements.

## 2.3 Operations on tree

```
int Height(Node *node) const
```

Returns height of Node. If Node is equal to nullptr returns -1.

```
int GetBalance(Node *node) const
```

Returns balance factor of Node. If Node is equal to nullptr returns -1.

```
int UpdateHeight(int leftHeight, int rightHeight) const
```

Returns updated high for Node. leftHeight and rightHeight are heights of node's children.

```
Node *SingleRotateLeft(Node *node)
Node *SingleRotateRight(Node *node)
Node *DoubleRotateLeft(Node *node)
Node *DoubleRotateRight(Node *node)
```

These methods perform operations of nodes rotation.

```
Node *GetMinimalKey(Node *node) const
```

Returns Node with smallest key value.

## 2.4 Modifiers

```
public: void Insert(const Key &key, const Info &info)
private: Node *Insert(Node *parent,
                      const Key &key, const Info &info)
```

Add new element with given data. If key exists in tree, method replaces info.

```
public: void Remove(const Key &key);
private: Node *Remove(Node *node, const Key &key)
```

If key exists in tree, method removes the entry with this key.

```
public: bool Clear()
private: void Clear(Node *node)
```

Removes all nodes. If tree was empty before, method returns false.

## 2.5 Printing

```
public: void PrintInorder() const
private: void PrintInorder(Node *node) const
```

Prints tree within in-order traversal.

```
public: void PrintPreorder() const
private: void PrintPreorder(Node *node) const
```

Prints tree within pre-order traversal.

```
public: void PrintPostorder() const
private: void PrintPostorder(Node *node) const
```

Prints tree within post-order traversal.

```
public: void PrintVisually() const
private: void PrintVisually(Node *node) const
```

Prints tree's visual representation.

**Example**

```
                                (11, 1)
                                /
                         (10, 1)
                        /
                 (9, 1)
                       \
                 /      (7, 1)
                /
root---> (3, 1)
                \
                 (-1, 1)
                         \
                         (-12, 1)
```

### 2.6 Other methods

```
bool IsEmpty() const
```

Returns `true` if tree contains no nodes.

```
int Size() const
```

Returns private field `size` which contains number of nodes in tree.

```
bool Search(const Key &key) const
Node *Search(Node *node, const Key &key) const
```

Returns `true` if `key` is in the tree.

```
void CopyTree(Node *&newTree, Node *copiedTree)
```

Auxiliary method used by copy constructor and assignment operator. Copies tree to given root `newTree`.

```
void CompareTrees(Node *left, Node *right, bool &isEqual)
```

Auxiliary method used by comparison operators. If trees are not the same saves `false` value to `isEqual` variable.

## 3 Decisions and changes

My conception did not have any bigger changes. On the beginning I wanted to implement methods only for single rotations. If double rotations were needed, I would perform them within `Insert` and `Remove`. However adding double rotation methods increased readability of code.

# 4 Testing

## 4.1 Introduction

For testing purpose I used Catch2 framework. To check code for memory leaks I used Valgrind. No memory leaks were found after usage of all implemented features of class.

## 4.2 Structure of tests

**Empty tree**

- Printing tree
- Checking if tree is empty
- Using removal methods
- Using `Search` method

**Filling tree and printing**

- Adding nodes
- In-order printing
- Pre-order printing
- Post-order printing
- Visual printing

**Removing nodes from tree**

- Removing chosen nodes
- Removing all nodes

**Comparison operators**

- Equality operator
- Inequality operator

**Copying trees**

- Copying with copy constructor
- Copying with assignment operator

**Other methods**

- Size method
- IsEmpty method
- Search method