Übung 2 : Treecheck

1. AVL-Balance an jedem Knoten ausgeben:

```
//checks if AVL-conditions are met
//returns statistics about the current node and its subtree to the node above
nodeStatistics treeNode::getStats()
{
    nodeStatistics rightStats, leftStats, curStats;

    //get statistics of the subNodes
    //if subNode does not exists use default values (listed in headerStats.h)
    if(rightNode != nullptr)
    {
        rightStats = rightNode->getStats();
    }

    if(leftNode != nullptr)
    {
        leftStats = leftNode->getStats();
    }

    //set max depth of the current subtree
    curStats.depth = rightStats.depth > leftStats.depth ? rightStats.depth + 1 : leftStats.depth + 1;

    curStats.total += key + leftStats.total + rightStats.total;
    curStats.counter += leftStats.counter + rightStats.counter + 1;

    curStats.AVL = rightStats.AVL && leftStats.AVL ? true : false;

    int AVLbalance = rightStats.depth - leftStats.depth;

    if(AVLbalance < -1 || AVLbalance > 1)
    {
        std::cout << "bal(" << key << ") = " << AVLbalance << " (AVL violation!)\n";
        curStats.AVL = false;
    }
    else
    {
        std::cout << "bal(" << key << ") = " << AVLbalance << "\n";
    }

    return curStats;
}
```

Funktionsbeschreibung:

Die Funktion ruft rekursiv für jedes „Kind“ eines Elements sich selbst auf. Sobald man ein Blatt erreicht, werden die Daten des Blattes erfasst und die AVL-Balance wird ausgegeben. Diese Daten werden an das übergeordnete Element returniert, das ebenfalls seine Informationen zu den Daten der „Child-Nodes“ hinzufügt und diese an seinen „Parent“ übergibt.

Abbruchbedingung:

Sobald ein Blatt des Baumes erreicht ist, ruft sich die Funktion nicht mehr selbst auf und arbeitet von hinten nach vorne alle Elemente des gegebenen Baumes durch.

Rückgabewert:

Um diese Funktion nur einmal durchlaufen zu müssen werden mehrere Werte gleichzeitig erhoben und zurückgegeben in Form von folgendem struct:

```
struct nodeStatistics{

    int depth = 1;
    int total = 0;
    int counter = 0;
    bool AVL = true;

} typedef nodeStatistics;
```

Depth: Tiefe des bisher durchlaufenen Baumes (+1 pro „Level“)

Total: Summe aller bisher durchlaufenen Schlüsselwerte

Counter: Anzahl aller bisher durchlaufenen Schlüsselwerte

AVL: Flag um festzustellen, ob der bisher durchlaufene Baum ein gültiger AVL Baum ist

Aufwandsabschätzung:

Jede Node im Baum wird genau ein einziges Mal traversiert. Wir haben einen Aufwand von O(n) wobei n die Anzahl der Nodes des Baumes entspricht.

2. Suche nach einem Wert im Baum:

```
bool treeNode::searchForKey(int key)
{
    if(this->key > key)
    {
        if(leftNode == nullptr)
        {
            return false;
        }
        else
        {
            return leftNode->searchForKey(key);
        }
    }
    else if(this-> key < key)
    {
        if(rightNode == nullptr)
        {
            return false;
        }
        else
        {
            return rightNode->searchForKey(key);
        }
    }

    return true;
}
```

Funktionsbeschreibung:

Der Schlüsselwert des Elements wird mit dem gesuchten Wert verglichen. Falls der Wert größer ist, wird die Funktion rekursiv mit den rechten „Kind“ des Elements aufgerufen. Falls es kein rechtes Kind gibt, wissen wir, dass der Wert im Baum nicht existiert. Dasselbe Prinzip gilt für den Fall, dass der gesuchte Wert kleiner als der Schlüsselwert ist.

Abbruchbedingung:

Die Funktion ruft sich nicht selbst auf, wenn der Schlüsselwert dem gesuchten Wert entspricht (in diesem Fall returniert sie true) oder wenn der gesuchte Wert nicht im Baum enthalten sein kann (sie returniert false).

Aufwandsabschätzung:

Wenn der gegebene Baum ein gültiger AVL-Baum ist haben wir einen Aufwand von O(log n), da die maximale Anzahl der Aufrufe der Tiefe des Baumes entspricht.

Bei einem ungültigen AVL-Baum haben daher einen Aufwand von O(n).

3. Suche nach einem Unterbaum im Baum

Funktionsbeschreibung:

1.Schritt: wir suchen im Hauptbaum die Node, deren Schlüsselwert der gegeben Node des Unterbaums entspricht (Funktionsweise siehe 2.).

Wenn wir diese Node nicht finden, ist der Unterbaum nicht enthalten.

```
if(this->key > subRoot->getKey())
{
    if(leftNode == nullptr)
    {
        return false;
    }
    else
    {
        return leftNode->searchForSubTree(subRoot);
    }
}
else if(this->key < subRoot->getKey())
{
    if(rightNode == nullptr)
    {
        return false;
    }
    else
    {
        return rightNode->searchForSubTree(subRoot);
    }
}
```

2.Schritt: Sobald wir die entsprechende Node gefunden haben, wird dieselbe Funktion für die beiden „Kinder“ des Unterbaumes aufgerufen (zusätzlich wird hier noch abgeprüft, ob diese Kinder existieren, oder sie überhaupt im Hauptbaum enthalten sein können).

```
if(key == subRoot->getKey())
{
    bool rightSubTree, leftSubTree;

    if(subRoot->getRightNode() == nullptr)
    {
        rightSubTree = true;
    }
    else if(rightNode == nullptr && subRoot->getRightNode() != nullptr)
    {
        rightSubTree = false;
    }
    else
    {
        rightSubTree = rightNode->searchForSubTree(subRoot->getRightNode());
    }

    if(subRoot->getLeftNode() == nullptr)
    {
        leftSubTree = true;
    }
    else if(leftNode == nullptr && subRoot->getLeftNode() != nullptr)
    {
        leftSubTree = false;
    }
    else
    {
        leftSubTree = leftNode->searchForSubTree(subRoot->getLeftNode());
    }

    if(leftSubTree && rightSubTree)
    {
        return true;
    }

    return false;
}
```

Abbruchbedingung:

1. Der Wert des Unterbaumes ist nicht im Hauptbaum vorhanden → Funktion bricht ab und returniert false.
2. Man findet ein Blatt des Unterbaums im Hauptbaum → Funktion bricht ab und returniert true.

Aufwandsabschätzung:

Hier entspricht der maximale Aufwand O(n) wobei n der Anzahl der Elemente des Unterbaumes entspricht.