TreeCheck - Protokoll

Programm dient zur Überprüfung von AVL Bäumen. Bäume werden durch beliebige .txt Datei eingelesen und erstellt. Durchgeführt durch “Tree::read()”:

```
while (getline(MyReadFile, readText))
{
    newKey = stoi(readText);
    checkNode(root, newKey);
}
```

.txt File wird Zeile für Zeile eingelesen (eine Zahl nach der anderen). String wird für Weiterverwendung in int umgewandelt.

```
void Tree::addNode(tnode* node, int newKey)
{
    if (newKey < node->key)
    {
        if (node->left == nullptr)
        {
            node->left = new tnode;
            node->left->key = newKey;
        }
        addNode(node->left, newKey);
    }
    if (newKey > node->key)
    {
        if (node->right == nullptr)
        {
            node->right = new tnode;
            node->right->key = newKey;
        }
        else
        {
        addNode(node->right, newKey);
        }
    }
}
```

“addNode()” vergleicht den Wert der eingefügt werden soll mit der Node die er übergeben bekommen hat. Sollte er keine freie Stelle für den neuen Wert bei der aktuellen Node finden, ruft sich addNode mit entsprechender linken oder rechten anhängenden Node auf.

Erster Aufruf passiert mit root-Node addNode(root, newKey).

Average- / Minimum- / Maximum- Search

Für die Suche nach MIN, MAX & AVG wurde ursprünglich ein einziger Durchgang geplant. Das fehlerfreie zusammenführen ist jedoch ohne return von allen Werten nicht möglich. Durch die Anzahl der Werte welche returned werden können (1) entsteht eine Limitation. Um diese Limitation umgehen zu können mussten die Werte getrennt von der Funktion gespeichert werden und die Funktion findAvgMinMax() bekommt Referenzen auf diese Werte übergeben.

```
void Tree::findAvgMinMax(::tnode* node, int& minValue, int& maxValue, int& totalValue, int& nodesCount)
```

```
nodesCount++;
totalValue += node->key;
if (minValue == NULL && maxValue == NULL) {
    maxValue = node->key;
    minValue = node->key;
}
else {
    if (node->key > maxValue)
        maxValue = node->key;
    if (node->key < minValue)
        minValue = node->key;
}
```

Anzahl der Knoten wird bei jedem durchgang erhöht. Sollten minValue und maxValue noch nicht gesetzt sein werden übernehmen diese den Wert des aktuellen Keys, ansonsten werden diese individuell abgeglichen und bei bedarf überschrieben.

Sollte es noch einen verweis auf ein weiteres Element in der Liste geben, so ruft sich findAvgMinMax selbst mit diesem Element erneut auf.

```
if (node->left != nullptr)
{
    findAvgMinMax(node->left, minValue, maxValue, totalValue, nodesCount);
}
if (node->right != nullptr)
{
    findAvgMinMax(node->right, minValue, maxValue, totalValue, nodesCount);
}
```

Einfache Suche

```
tnode* left, * right;
left = right = nullptr;

if (n == node->key) {
    path.emplace_back(node->key);
    return node;
}
if (n < node->key)
{
    if (node->left != nullptr) {
        left = search(node->left, n, path);
    }
}
if (n > node->key)
{
    if (node->right != nullptr) {
        right = search(node->right, n, path);
    }
}

if (left != nullptr) {
    path.emplace_back(node->key);
    return left;
}
if (right != nullptr) {
    path.emplace_back(node->key);
    return right;
}
return nullptr;
```

Die Suche innerhalb unseres Baums gestalltet sich sehr ähnlich der Berechnung von Min und Max. Solange der Key nicht gefunden wurde und es eine weitere Kante gibt, wird über diese eine angehängte Node durchsucht. Bei der Ankunft am Ende eines Ast wird ein nullptr zurückgegeben. Wird der Wert im Baum gefunden wird der korrespondierende Knoten zurückgegeben.

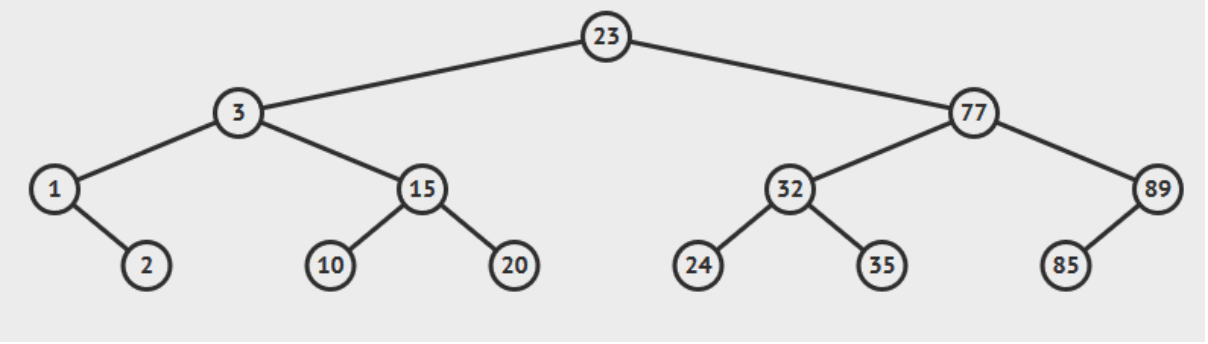
Subtree Suche

Beispiel:

Eingabe

```
23
77
3
15
10
20
1
2
77
23
32
89
85
24
35
```

Visuelle Repräsentation der Datenstruktur im Programm



O-Notation

void Tree::addNode(tnode* node, int newKey)	O(log n)
void Tree::deleteTree(tnode* node)	O(log n)
tnode* Tree::superSearch(std::string nodeToFind)	O(log n²) |1*|
void Tree::read(std::string filePath)	O(log n * n) |2*|
void Tree::findAvgMinMax(::tnode* node, int& minValue,...)	O(log n)
tnode* Tree::search(tnode* node, int n, std::vector<int>& path)	O(log n)
int Tree::calcBalance(tnode* node)	O(log n)
bool Tree::subTreeSearch(tnode* sub, tnode* parent)	O(log n²)
tnode* Tree::nodeSearch(tnode* node, int n)	O(log n)

|1*| = O(log n) * O(log n) || Supersearch(log n) * search(log n)
|2*| = O(log n) * O(n) || addNode(log n) * n (Größe des Baums)