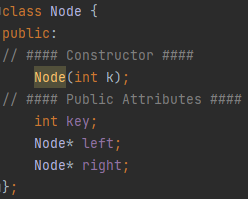
ALGOS – Programmierbeispiel 2

# Beschreibung der Datenstruktur:

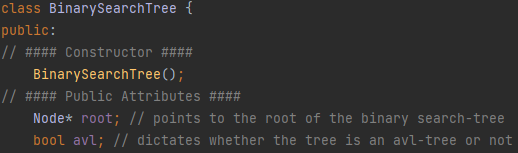
## Node:



Die Datenstruktur zu einem Node wurde ganz der Angabe entlehnt, „key“ speichert einen Schlüssel in Form eines Integers, die Node-Pointer „left“ und „right“ zeigen auf die potentiellen Nachfolger des Nodes.

Der Constructor initialisiert ein Node-Objekt mit dem Schlüsselwert „k“, während die Node-Pointer „left“ und „right“ auf nullptr zeigen.

## Binary-Search Tree:



Der Binary-Search Tree beinhaltet einen Node-Pointer namens „root“ der wie der Name schon sagt auf den Root-Node des BST zeigt. Zusätzlich hierzu haben wir einen Boolean „avl“ als Attribut gewählt, welcher darüber Auskunft gibt, ob der Baum ausreichend balanciert ist, um als AVL-Baum zu gelten.

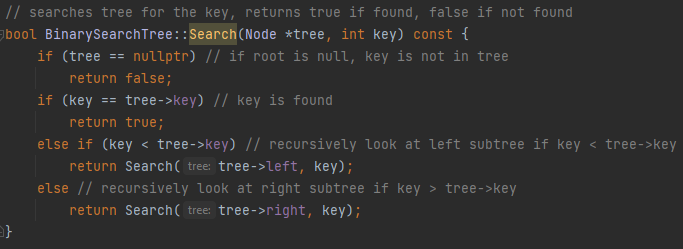
Eine weitere Überlegung war es, mit einem Integer die Anzahl der Knoten im Baum mitzuführen, die bei der Berechnung des durchschnittlichen Schlüsselmittelwerts hilfreich wäre. Zur Übung haben wir dies aber sein lassen und stattdessen eine rekursive Methode zu diesem Zweck entwickelt.

Der Constructor initialisiert den Node-Pointer „root“ auf nullptr und den Boolean „avl“ standardmäßig auf „true“, da ein leerer binärer Baum technisch gesehen als AVL-Baum gilt. Beim Einlesen der Knoten wird die Balance der Knoten überprüft.

# Beschreibung der Methoden:

## Erstellen und Suche eines Knotens:

### Search:



#### Beschreibung:

Diese Funktion sucht rekursiv in einem BST nach einem Knoten mit angegebenen Key. Sollte der übergebene Root-Node = nullptr sein, so kann die Suche erfolglos abgebrochen werden (es wird false returniert). Dann wird der übergebene gesuchte Schlüsselwert mit dem Key des betrachteten Nodes verglichen, sind sie gleich, wurde das Element gefunden und es wird true returniert, sonst wird die Funktion rekursiv aufgerufen, je nachdem ob der gesuchte Wert kleiner oder größer ist, mit dem linken oder rechten Teilbaum.

#### Aufwandsabschätzung:

Hier kann zwischen einer erfolgreichen und einer erfolgslosen Suche unterschieden werden, außerdem kann es sich bei dem BST um einen balancierten Baum handeln, oder auf der anderen Seite um eine entartete Liste:

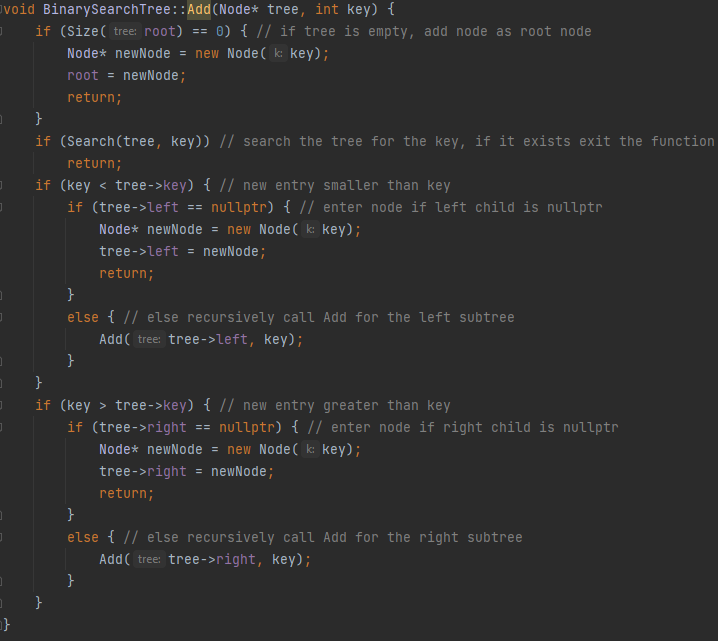
Best-Case (gefunden): O(1) (Element ist Root-Node)

Worst-Case (gefunden): O(log n) bei balanciertem Baum, O(n) entartete Liste

Best-Case (nicht gefunden): O(1) (Root-Node ist nullptr)

Worst-Case (nicht gefunden): O(log n) bei balanciertem Baum, O(n) entartete Liste

### Add:



#### Beschreibung:

Diese Funktion fügt einen neuen Knoten rekursiv zum BST hinzu, ist dieser leer, so wird ein neuer Knoten als Root-Node erstellt (konstante Laufzeit). Sonst wird die Search-Funktion aufgerufen (O(log n) … O(n)), um zu schauen, ob bereits ein Node mit angegebenen Key im Baum liegt. Ist das der Fall wird returniert, ohne einen neuen Node hinzugefügt zu haben.

Anschließend wird der Baum rekursiv durchlaufen, bis eine Stelle ohne Nachfolger gefunden wurde, an die der Knoten eingefügt werden kann. Die Laufzeit hängt hier von der Balance des Baumes ab.

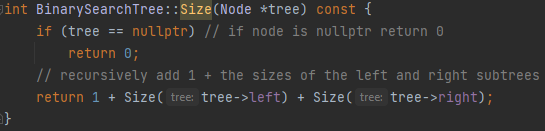
#### Aufwandsabschätzung:

Best-Case: O(log n) … balancierter Baum (abgesehen von konstantem Einfügen eines Root-Nodes)

Worst-Case: O(n) … entartete Liste

## Ermittlung der statistischen Werte:

### Size:



#### Beschreibung:

Die Methode bekommt als Parameter den Root-Node eines BST übergeben und ermittelt rekursiv die Anzahl von Knoten in diesem BST. Die Abbruchbedingung ist hier gegeben, wenn man bei einem Root-Node ankommt, der nullptr ist. Da diese selbst nicht mehr ein gültiger Knoten sind, gehen diese bei Berechnung der Größe nicht ein, aus diesem Grund wird hier 0 zurückgegeben.

Ansonsten wird die Größe eines Baumes durch 1 (Root-Node) + die Größe des linken und rechten Teilbaums berechnet.

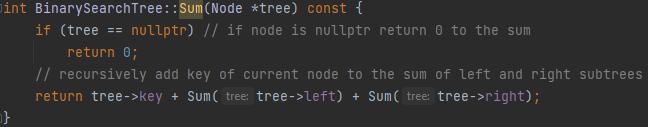
#### Aufwandsabschätzung:

Da bei diesem Beispiel die Balance des Baumes nicht gewährleistet ist, kann man hier zwischen einem Best-Case (ausgeglichener Baum) und einem Worst-Case (ausgeartete Linked List) unterscheiden:

Best-Case: O(log n)

Worst-Case: O(n)

### Sum:



#### Beschreibung:

Diese Methode funktioniert ganz analog zur Size-Methode, hier wird bloß der Schlüsselwert aufsummiert, anstatt der Anzahl der Knoten.

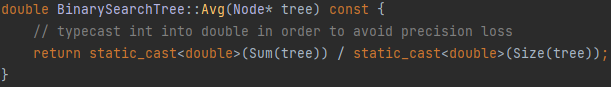
#### Aufwandsabschätzung:

Da bei diesem Beispiel die Balance des Baumes nicht gewährleistet ist, kann man hier zwischen einem Best-Case (ausgeglichener Baum) und einem Worst-Case (ausgeartete Linked List) unterscheiden:

Best-Case: O(log n)

Worst-Case: O(n)

### Avg:



#### Beschreibung:

Diese Methode berechnet den Mittelwert aller Schlüsselwerte eines Baums, hierzu werden „Sum“ und „Size“ benötigt. Der explizite Typecast zu double verhindert einen Genauigkeitsverlust bei der Berechnung des arithmetischen Mittels.

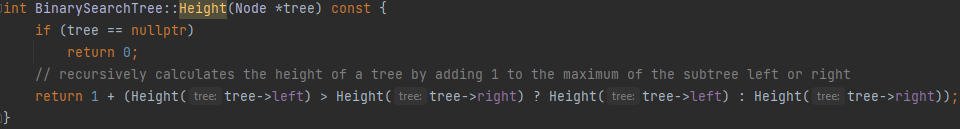
#### Aufwandsabschätzung:

Analog zur „Sum“ und “Size“ Methode lässt sich die asymptotische Laufzeit auch hier ermitteln.

Best-Case: O(log n)

Worst-Case: O(n)

### Height:



#### Beschreibung:

Hier wird rekursiv die Höhe eines BST ermittelt, indem die größere Höhe/Tiefe der beiden Teilbäume + 1 (wegen Root-Node) zurückgegeben wird. Die rekursive Abbruchbedingung findet bei den Knoten = nullptr statt, die nicht in die Höhe miteinfließen.

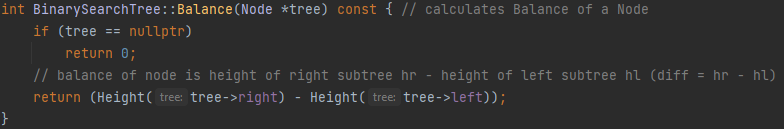
#### Aufwandsabschätzung:

Hier wird wieder zwischen den zwei Extrema eines komplett ausgewogenen binären Baums und einer entarteten Liste unterschieden.

Best-Case: O(log n)

Worst-Case: O(n)

### Balance:



#### Beschreibung:

Die Balance-Funktion verwendet zur Berechnung des Balance-Faktors eines Nodes die Height-Funktion. Wie in den Vorlesungsfolien berechnen wir die Balance durch die Differenz der Höhe des rechten Teilbaums weniger der Höhe des linken Teilbaums. Die Balance wird wieder für jeden Knoten rekursiv ermittelt mit der Abbruchbedingung, in der der übergebene Root-Node nullptr ist.

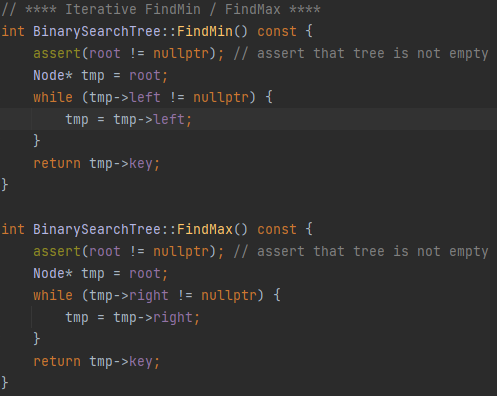
#### Aufwandsabschätzung:

Diese Methode ruft im Falle eines Baumes mit Root ungleich nullptr die rekursive Funktion Height auf, aus diesem Grund herrscht hier die gleiche asymptotische Laufzeit wie in den anderen Methoden.

Best-Case: O(log n)

Worst-Case: O(n)

### Iterative FindMin / FindMax:



#### Beschreibung:

Die Funktionen FindMin und FindMax haben wir grundsätzlich erstmals iterativ implementiert, da wir uns hierfür die Eigenschaften eines BST zunutze machen können. Da das kleinste/größte Element auch zugleich das am weitest linke/rechte Element ist, ist es so nicht notwendig den gesamten Baum rekursiv zu durchsuchen. Stattdessen wird iterativ der linke/rechte Pfad bis zum ersten Vorkommnis eines Knotens = nullptr traversiert.

#### Aufwandsabschätzung:

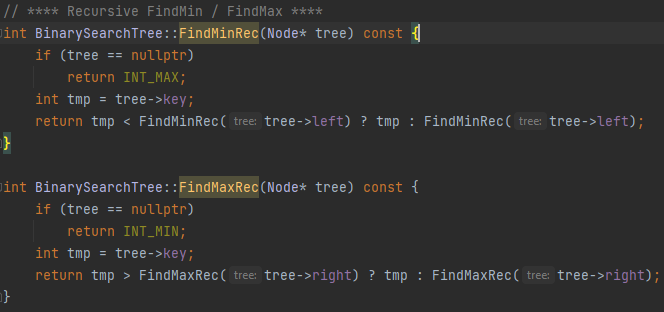
Hier wird zwischen einem perfekt balancierten Baum bzw. einer entarteten Liste als Extremfälle unterschieden.

Best-Case: O(1) (entartete Liste und kleinstes bzw. größtes Element ist am Anfang)

Average Case: O(log n) (evtl. Balancierter Baum -> Äste haben log n Tiefe, n … Anzahl Knoten)

Worst-Case: O(n) (entartete Liste und kleinstes bzw. größtes Element ist am Ende)

### Recursive FindMin / FindMax:



#### Beschreibung:

Hier noch eine rekursive Alternative zu den obigen Funktionen, allerdings glaube ich nicht, dass sie wirklich clean implementiert wurden. Der linke bzw. rechte Ast des Baums wird rekursiv durchgelaufen und das Minimum des jetzigen Knotens bzw. des Nachfolgers returniert. Bei der Abbruchbedingung war ich mir nicht wirklich sicher, wie man das sauberer bewerkstelligen kann, habe es aber so gemacht, dass bei einem Knoten = nullptr mit dem maximal bzw. minimal möglichen Integer-Wert verglichen wird.

#### Aufwandsabschätzung:

Hier ähnlich wie bei der iterativen Implementierung, da ja nicht der gesamte Baum, sondern lediglich der linke bzw. rechte Ast durchgegangen wird. Die Länge dieser Äste kann bei einer entarteten Liste 0 oder n sein, bei einem balancierten Baum log n.

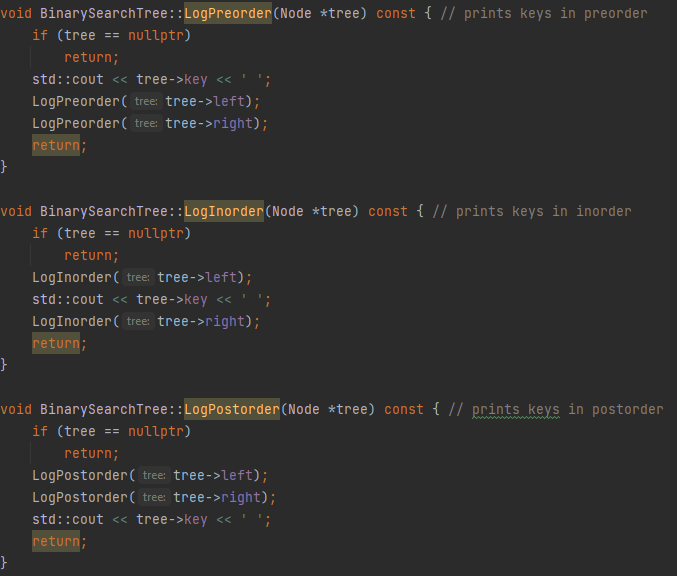
Best-Case: O(1) (entartete Liste und kleinstes bzw. größtes Element ist am Anfang)

Average Case: O(log n) (evtl. Balancierter Baum -> Äste haben log n Tiefe, n … Anzahl Knoten)

Worst-Case: O(n) (entartete Liste und kleinstes bzw. größtes Element ist am Ende)

## Logging:

### LogPreorder / LogInorder / LogPostorder:



#### Beschreibung:

Alle drei Funktionen durchlaufen den Baum rekursiv und geben dabei die Schlüsselwerte der Knoten aus, lediglich die Reihenfolge der Ausgabe und der rekursiven Aufrufe ist vertauscht.

#### Aufwandsabschätzung:

Best-Case ist wieder ein balancierter Baum, Worst-Case eine entartete Liste:

Best-Case: O(log n)

Worst-Case: O(n)