# Binary Tree

## Grundaufbau:

### Einfügen neuer Daten:

Jede Node besteht aus einem Wert und 2 Pointern (rechts und links). Im Falle, dass die gesuchte/neue Nummer kleiner ist als die Nummer der derzeitigen Node wird dasselbe nochmal für die Node rechts (Pointer, der right heißt) (Im Falle, dass der Wert größer als die Nummer wird der Linke Pointer genommen). Dies wird so lange wiederholt, bis der passende Pointer ein Null Pointer ist.

Ein Bild, das Text, Schrift, Reihe, Screenshot enthält.

Automatisch generierte Beschreibung

Diese Funktion hat im Worstcase einen Aufwand von O(n), da in diesem Fall alle Werte immer größer oder kleiner als der vorherige Wert sind (also bereits sortiert sind) und somit einer verketteten Liste gleichkommen. Der average Case dieser Funktion hat einen Aufwand von O(log(n)) da man mit jeder Node die Anzahl an möglichen werten halbiert.

### Minimum und Maximum:

Diese Rekursiven Funktionen rufen sich so lange selbst auf, bis sie den Baum bis ganz unten ans Ende gelaufen sind und nur die rechten oder nur die linken Pointer genommen haben.

Ein Bild, das Text, Schrift, Reihe, Screenshot enthält.

Automatisch generierte Beschreibung

Aufwand hierbei ist wie bei der Funktion davor im Worstcase O(n) und im average Case O(log(n))

### Nodecounter:

Die Nodecounter Funktion ruft rekursiv jede Node auf und schaut, ob sie eine anhängende Node hat anschließend addiert er immer +1 für jede Node dazu.

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

Automatisch generierte Beschreibung

Der Worstcase gleicht hier dem average Case und ist O(n) da jede Node 1-mal aufgerufen wird.

### Sum:

Wie der NodeCounter schon ruft die Sum Funktion jede einzelnen Node auf. Dieses Mal addiert diese Funktion jedoch immer die Werte der Nodes zusammen und bildet somit eine Summe.

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

Automatisch generierte Beschreibung

Wie bei der Funktion davor beträgt der Aufwand im worst- und average Case O(n)

### Delete Tree:

Die Funktion hat denselben Aufbau wie die 2 davor nur mit dem Unterschied, dass diese die zugehörige Node löscht.

Ein Bild, das Text, Schrift, Screenshot, weiß enthält.

Automatisch generierte Beschreibung

Aufwand liegt in Average und Worstcase bei O(n)

### Check\_if\_subtree:

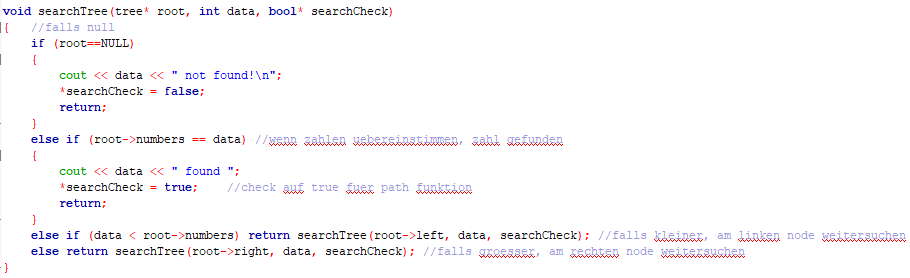
Bei dieser Funktion wird überprüft ob 2 angegebene Werte einen Subtree bilden oder nicht. Die funktioniert, indem man von der Node sich immer näher dem wert annähert und alle Anhängsel des Startingpoints überprüft. Im Falle, dass der Wert gefunden wurde, returned diese Funktion true.

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

Automatisch generierte Beschreibung

Der Aufwand dieser Funktion beträgt im Worstcase O(n) (alle Werte in einer Liste) und im AverageCase O(n/x) da je nach Eingabeparameter der Funktion insgesamt weniger Werte überprüft werden müssen.

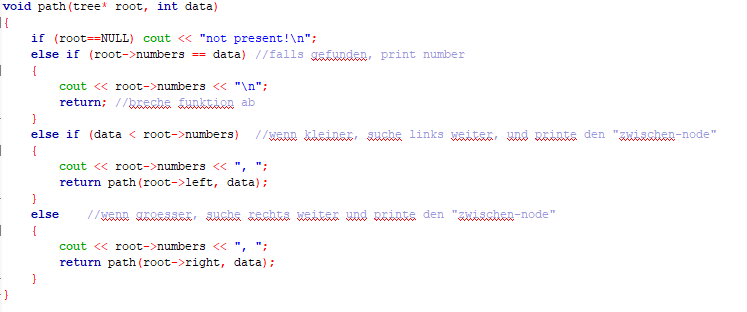
### searchTree:

Diese Funktion sucht im Binary Tree rekursiv den Value, welcher vom User als Input verlangt wird. Sollte der root NULL betragen ist die suche fehlgeschlagen und bricht die Funktion ab. Wenn der Input und der Wert des Knotens übereinstimmen wird der searchCheck boolean als true gesetzt, ein „found“ geprinted und die Funktion schließlich beendet. Ansonsten wird jenachdem ob der Wert größer oder kleiner als der Knoten ist, entweder rechts oder links die Funktionerneut aufgerufen. 

Im schlechtesten Fall, sollte der Baum eine verkettete Liste sein, beträgt der Aufwand O(n), und im Durchschnitt liegt der Aufwand bei O(h) h = Höhe / Height des Baums

### Path

Die Path-Funktion funktioniert vom Prinzip exakt gleich wie searchTree und dient lediglich zur Formatierung des Programms. Hier wird lediglich nur der „Zwischenknoten“ in die Konsole geprinted, um den Weg zu visualisieren, den die Funktion geht. Path wird nur aufgerufen, sollte die searchCheck variable als true markiert sein.



Da die Funktion gleich fungiert wie searchTree hat diese ebenfalls im schlechtesten Fall einen Zeitaufwand von O(n), und im average Case O(h).