**Beschreibung der Funktionen**

**Einlesen und Erstellen des Baumes**

Das Erstellen des Baumes wurde rekursiv implementiert. Bei jeder neuen Node wird der Baum durchlaufen bis an der passenden Stelle ein freier Platz gefunden wird. Also bei jedem Funktionsaufruf wird, neben dem neuen Eintrag, die Wurzel des Baumes übergeben, wobei danach gecheckt wird ob die Stelle frei ist, was bedeuten würde das Ziel, also der Einfügeplatz wäre erreicht. Ansonsten wird der einzufügende Wert mit dem Wert der derzeitigen Wurzel verglichen und falls der neue Wert größer ist die Funktion erneut aufgerufen, aber mit dem rechten Teilbaum der derzeitigen Wurzel. Umgekehrtes für den Fall, dass der Wert kleiner ist. Hierbei ist explizit der Fall, dass die Werte gleich sind ausgeschlossen, da dann kein Einfügen erfolgen soll.

**Löschen des Baumes**

Da für den Baum manuell Speicher allokiert wird, muss dieser auch wieder freigegeben werden. Hierbei wird eine Funktion verwendet die als Parameter einen Knoten des Baumes übergeben wird, wobei es sich beim ersten Aufruf um die Wurzel des Baumes handelt. Es wird geprüft ob die Wurzel überhaupt noch einen Wert hat. Falls doch werden jeweils für den linken und rechten Teilbaum überprüft ob diese Werte beinhalten und falls sich dies bestätigt wird die Funktion für den jeweiligen Teilbaum rekursiv aufgerufen.

**Statistiken berechnen**

**Minimum:**

* Da der niedrigste Wert des Baumes an der links untersten Wurzel liegt, muss der Baum lediglich immer wieder nach links traversiert werden, bis der linkeste untere Knoten erreicht wird. Als Funktion umgesetzt bedeutet das, dass die Wurzel beim Erstaufruf übergeben wird und geprüft wird ob ein linker Teilbaum existiert. Wenn ja dann wird die Funktion erneut mit dem linken Teilbaum als Wurzel aufgerufen. Wenn nein dann beinhaltet der derzeitige Knoten den niedrigsten Wert des Baumes.

**Maximum:**

* Die Berechnung des Maximums erfolgt ident wie die Berechnung des Minimums, wobei statt dem linken Teilbaum der rechte Teilbaum genommen wird, da der rechteste untere Knoten den größten Wert beinhaltet.

**Durchschnitt:**

* Für die Berechnung des Durchschnittes werden zwei Werte benötigt. Einerseits die Anzahl der Knoten und andererseits die Summe aller Werte im Baum. Da beide dieser Werte verlangen jeden Knoten des Baumes zu besuchen schauen diese relativ ähnlich aus, also eigentlich ident, wobei beim Zählen der Knoten theoretisch einfach als Wert immer 1 genommen wird, statt dem eigentlichen Wert. Ähnlich wie bei den bisherigen Funktionen wird beim Erstaufruf die Wurzel des Baumes übergeben. Danach wird überprüft ob die derzeitige Wurzel noch ein Knoten ist, wobei dies die Abbruchbedingung der Funktion ist, da es ja nicht tiefer geht. Andernfalls wird für den linken und rechten Teilbaum die Funktion aufgerufen weiter traversiert.

**Balance Berechnen**

* Um die Balance zu berechnen wird die Root Node des Baumes gebraucht und ebenfalls ein boolean die Information zu speichern ob der Baum auch ein AVL Baum ist. Als erstes wird überprüft ob das aktuelle root nicht null ist, wenn ein leerer baum übergeben wird oder man bei einem Ast ganz unten ankommt da die Funktion rekursive ist.  Dann wird das linke Ast des Baumes angeschaut gefolgt vom rechten, die Balance wird dann mit dem rechten Minus den Linken return wert der vorherigen Funktionen gerechnet wird. Es wird dann überprüft ob der Betrag der Balance größer ist oder Gleich 2 beträgt wenn das der Fall ist wird “AVL violation!” angezeigt. Es wird dann auch für jedem Knoten die Bance ausgegeben. Die Funktion gibt den Balance Wert + 1 des größeren Balance Wert des linken und rechten Baumes.

**SubTree Suche**

* **Fall 1 Element:** Wenn die erste Node des Baumes keine Linke und Rechte node besitzt ist es eine einfache Zahlen Suche diese wurde ebenfalls rekursiv implementiert. Da es verlangt ist den Weg zu dieser Node zu speichern, braucht die Funktion das root element des Baumes, ein vektor welches per Referenz übergeben wird um die besuchte Elemente aufzunehmen und das Gesuchte Wert. Als erstes wird überprüft ob das aktuelle root element null ist wenn dies der Fall ist wird die wird null zurückgegeben, wenn das gesuchte Wert kleiner ist als dem Wert des aktuellen Element wird das aktuelle Wert im Vektor gespeichert und die Funktion dieselbe Funktion wieder aufgerufen mit dem linken node als root. Das gleiche passiert gespiegelt wenn das gesuchte Wert größer ist als das vom aktuellen root element. Wenn das Gesuchte Element den gleichen Wert aufweist wie das aktuelle Element wird das Element zurückgegeben.
* **Fall n > 1 Elemente:** Falls der SubTree mehr als ein Element aufweist wird zuerst das erste Element des Subtrees gesucht wenn dieser vorhanden ist wird diese Node für das 1. Baum als root node übergegeben an der Funktion searchSubTree mit dazu den root des zweiten Baumes und ein boolean Wert per referenz um zu festzustellen um einem Ergebnis zurück zu bekommen. zuerst wird überprüft ob die Rekursion unterbrochen werden muss da ein Element nicht gefunden worden ist. Dann wird überprüft ob das aktuelle Element Kinder besitzt wenn dies der Fall ist wird ein Wert für mögliches error handling zurückgegeben. Dann wird überprüft ob das linke Kind nicht null ist, wenn dies der Fall ist wird von dem Linken Kind aus wie oben beschrieben nach dem Wert des linken Kind vom SubTree gesucht. Falls dieser nicht gefunden wird heißt das dass das 2. Tree nicht ein SubTree vom ersten ist, error wird auf true gesetzt und die rekursion wird bis nach oben gestoppt. Falls das Element aber gefunden wird wird das aktuelle root des 1. Baumes auf dem Element gesetzt welches gerade gefunden worden ist. Auf der gleichen Idee ist ebenfalls die Suche nach rechts aufgebaut zwischen den zwei Seiten wird aber noch geprüft ob error nicht true ist um den abbruch der Rekursion zu beschleunigen.

**Komplexität der Funktionen**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Best Case | Average Case | Worst Case |
| Einfügen einer Node | O(1) | O(log2(n)) | O(n) |
| Minimum | O(1) | O(log2(n)) | O(n) |
| Maximum | O(1) | O(log2(n)) | O(n) |
| Average | O(4n) | O(4n) | O(4n) |
| CheckAVL | O(4n) | O(4n) | O(4n) |
| SearchSubTree | O(1) | O(n\*log2(n)) | O(n\*log2(n)) |
| SearchKey | O(1) | O(log2(n)) | O(n) |