Seminararbeit Red-Black Trees

Herr Yannik Höll

17. Oktober 2021

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Das geordnete Speichern von Daten ist in der heutigen Zeit eine sehr wichtige Aufgabe, vor allem in Betracht auf immer größer werdende Datensätze. Ebenso ist es wichtig, schnell diese Daten gezielt und effizient zu durchsuchen. Dafür benötigt man optimierte Datenstrukturen.

Eine Möglichkeit, Daten geordnet nach einem Schlüssel zu speichern, sind so genannte Baumdatenstrukturen. Eine sehr einfache Baumimplementierung ist der BT, welcher jeweils nur 2 Abzweigungen pro Knoten besitzt. Leider hat diese naive Variante der Datenstruktur einige Probleme, welche vor allem beim Einfügen der Daten in geordneter Reigenfolge entstehen. Um diese zu umgehen, kann man die Algorithmen zum Einfügen und Entfernen neuer Datensätze so anpassen, dass die Baumstruktur performanter wird. Ein möglicher besserer Ansatz sind die RBTs.

Diese Arbeit beschreibt, wie sich RBTs von normalen BTs unterscheiden. Es wird ausführlich beschrieben, wie normale BTs funktionieren und wie man ihre Algorithmen erweitert, um RBTs zu erhalten. Dazu wurden jeweils beide Datenstrukturen in der Programmiersprache **C** implementiert. Auf den jeweiligen Quelltext wird auch eingegangen, um auf bestimmte Schwierigkeiten und Besonderheiten in den Implementierungen einzugehen. Zudem werden die Ergenisse von Benchmark analysiert, welche zeigen sollen, dass RBTs tatsächlich bessere Laufzeiteigenschaften haben als die Standimplementierung der BTs. Außerdem wird auf die Zeitkomplexität der beiden Datenstrukturen eingegangen und wie sich vor allem der RBT im Worst Case verhält. Die Codeauschnitte in dieser Arbeit sind aus Platzgründen gekürzt. Meistens wurde die Fehlerbehandlung ganz entfernt. Wenn der Leser den gesamten Quellcode sehen möchte, muss er in die Implementierung in **C** schauen.

2 Definition Binary Tree & Red Black Tree

Bäume ganz allgemein sind anders als z.B. Arrays (ARRs) und Listen keine linearen Datenstrukturen, in denen alle Daten hintereinander liegen, sondern haben Verzweigungen, die durch bestimmte Eigenschaften in den Daten zustande kommen. Eine formal mathematische Definition von Bäumen lautet wie folgt:

Definition 1. *Baum*
Ein Baum T ist eine endliche Menge an Knoten mit folgenden Eigenschaften:

- *Es gibt einen speziellen Knoten k, den man Root (engl. Wurzel) nennt.*
- *Alle anderen Knoten sind in $n \geq 0$ disjunkte Mengen $T_1, \cdots, T_n$ unterteilt, die selbst auch wieder Bäume sind und die man Unterbäume nennt.*

Durch diese Definition kann man direkt das Verhältnis zwischen der Root und den anderen Knoten erkennen. Die Root ist den anderen Knoten übergeordnet. Zudem handelt es sich um eine rekursive Definition, weil der Baum mithilfe von sich selbst definiert wird. Das soll nochmal verdeutlichen, dass es sich grundlegend um eine rekursive Datenstruktur handelt. Alle Algorithmen, die man auf Bäume anwendet, können rekursiv definiert werden, was in den folgenden Kapiteln aufgezeigt wird. [Knu68, S. 308]

2.1 Binary Tree

Der Binary Tree (engl. Binärbaum) ist eine spezielle Art von Baum, für den nach Definition 1 einfach $n \leq 2$ gelten muss. D.h., für jeden Knoten im Baum gibt es maximal 2 Unterbäume. Man kann diesen Baum jetzt als 3-Tupel (l, k, r) definieren, wobei l und r den linken bzw. rechten Unterbaum darstellen und k die Root. Ich spreche hier über sortierte Binary Trees bzw. Binary Search Trees, werde diese aber ab sofort trotzdem einfach als Binary Tree bezeichnen. Eine entscheidende Eigenschaft ist, dass alle Knoten, die sich in l befinden kleiner als die Root k sind und alle Knoten in r größer. Das ist auch der Grund, warum man von geordneten Bäumen spricht. [San08, S. 147]

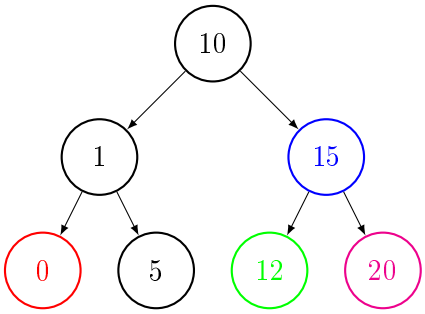

Abbildung 1: Beispielhafte Abbildung eines Binärbaums

2.1.1 Nomenklatur spezieller Knoten

In Abbildung 1 ist ein Beispiel eines Binary Trees zu sehen. Nun bekommen bestimmte Knoten in diesen Bäumen Namen, damit man einfach über sie sprechen kann, ohne immer eine konkrete Lagebeschreibung machen zu müssen. In diesem Baum ist der Knoten mit der Nummer 10 die Root. Wenn man die Pfeile von der 10 aus entlanggeht, gelangt man zu den Knoten 1 und 15. Diese werden als Children (engl. Kinder) von 10 bezeichnet. Genauso sind 0 und 5 die Children von 1; 12 und 20 die Children von 15. Umgekehrt bezeichnet man 10 als Parent (engl. Elternteil) von 1 und 15. Die Knoten 0, 5, 12 und 20 sind die Leaves (engl. Blätter) des Baums. Das sind genau die Knoten, die keine Children haben. Grundsätzlich ist es so, dass jeder Knoten im Baum einen Parent hat außer die Root. Alle Knoten, die den gleichen Parent haben, bezeichnet man als Siblings (engl. Geschwister). In unserem Beispiel sind 0 und 5, 12 und 20 und 1 und 15 Siblings. Der Parent eines Parent ist der Grandparent (engl. Großelternteil). Also ist beispielsweise die 10 der Grandparent von der 0. [Knu68, S.311]

Da die folgenden Benennungen nun ein bisschen komplizierter werden, kommen nun die Farben zum Einsatz. Von der 0 ausgehend ist der Sibling des Parents (blau) der Uncle (engl. Onkel). Das wäre also hier die 15. Und die Children vom Uncle werden als Niece (engl. Nichte) (grün, die 12) und Nephew (engl. Neffe) (magenta, die 20) bezeichnet. Die Niece ist immer der Knoten, der in dieselbe Richtung zeigt, wie der Knoten auf den man sich bezieht. Der Nephew ist der Knoten, der in die entgegengesetzte Richtung zeigt. Diese Unterscheidung wird später in einem Algorithmus wichtig werden.

2.1.2 Weiter Definitionen

Man bezeichnet einen Knoten als **Leaf** (engl. Blatt), wenn er keine Children besitzt. [Sed90, S. 36] Dementsprechend bezeichnet man einen Knoten als **Halfleaf** (engl. Halbblatt), wenn er nur ein Child hat.

Als **Path** (engl. Pfad) in einem Baum bezeichnet man eine Folge von Knoten, die paarweise verschieden sind und 2 Knoten im Baum (oder allgemein in einem Graphen) verbindet. [Sed90, S. 36]

2.1.3 Algorithmen

Die **Suche** im BT ist direkt durch seine Eigenschaften zu definieren. Da alle Knoten im linken Unterbaum kleiner sind und alle Knoten rechts größer, kann man einfach rekursiv eine Suche wie folgt

definieren:

Algorithm 1 Suche im BT

```
1: procedure SEARCH(node, key)
2:     if node == NULL then
         return NULL
3:     if node → key == key then
         return node
4:     else if node → key < key then
         return Search(node → left, key)
5:     else
         return Search(node → right, key)
```

`node` ist ein Pointer auf einen Knoten im Baum, `key` ist der zu suchende Schlüssel. `node→right` und `node→left` sind jeweils Pointer zum rechten und linken Unterbaum.

Letztendlich funktioniert der Algorithmus so, dass man bei der Root beginnt. Als erstes überprüft man, ob man bei einem NULL Pointer angekommen ist, welcher in dieser Arbeit als der Sentinel-Wert für einen leeren Knoten verwendet wird. Das zeigt an, dass der Schlüssel im Baum nicht vorhanden ist, und es wird direkt NULL ausgegeben. Dann wird abgefragt, ob der aktuelle Knoten `node` bereits den richtigen Schlüssel hat. Wenn das der Fall ist, dann wird er ausgegeben. Ansonsten wird die Suche beim linken bzw. rechten Unterbaum fortgesetzt, je nachdem ob `key` kleiner oder größer als `node→key` ist. Das wird durch einen rekursiven Aufruf der `Search`-Funktion realisiert. [Sed90, S. 203]

Das **Einfügen** ist im Grunde dasselbe wie die Suche, nur dass man diese durchführt, bis man `NULL` erreicht. An dieser Stelle wird dann der neue Knoten eingefügt. Da man keinen Pointer auf den Parent hat, muss man diesen jeweils zwischenspeichern. [Sed90, S. 205] Da der Algorithmus fast genauso wie die Suche definiert ist, wird hier auf Pseudocode verzichtet.

Die letzte wichtige Operation ist das **Löschen** von Knoten. Diese ist wohl die komplizierteste, weil der naive Ansatz, einfach den Knoten zu entfernen, nicht funktioniert. Die einzigen Knoten, die man entfernen kann, ohne dass der Baum in 2 Teile zerfällt, sind die Leaves. Deswegen muss man diesen Knoten mit einem Leaf austauschen. Dabei muss aber die Inorder-Traversierungsreihenfolge erhalten bleiben (Def. siehe Kapitel 3.6). Damit muss man mit dem Leaf tauschen, welches den nächstkleineren bzw. -größeren Schlüssel hat. Um den Knoten, der gelöscht werden soll zu finden, kann man die vorhin definierte Suche verwenden. Nach dem Austausch ist der zu löschende Knoten ein Leaf oder Halfleaf und kann sicher entfernt werden. Im Falle eines Halfleafs muss dann noch das Child des entfernten Knotens als Child des entsprechenden Parents festgelegt werden.

Algorithm 2 Löschen im BT

```
1: procedure DELETE(root, key)
2:     node ← Search(root, key)
3:     if node == NULL then
         return
4:     node ← SwapToLeaf(node)
5:     if IsHalfLeaf(node) then
6:         SetChild(GetParent(node), GetChild(node))
7:     FREE(node)
```

[Sed90, S. 210f]

Genaue Implementierungen der hier im Pseudocode nicht gezeigten Funktionen (*IsHalfLeaf*, *GetParent*, *GetChild*, *SetChild*) werden dann im Kapitel 3 genauer beleuchtet.

Als letztes wird hier noch ein Algorithmus besprochen, der zwar mithilfe des allgemeinen BT definiert werden kann, aber erst in den Algorithmen des RBT seine Anwendung findet. Die Rede ist von den **Baumrotationen**, die die Struktur des Baumes direkt verändern, ohne die Inorder-Reihenfolge durcheinanderzubringen. Hier gibt es zwei Möglichkeiten, eine Rotation nach links bzw. nach rechts.

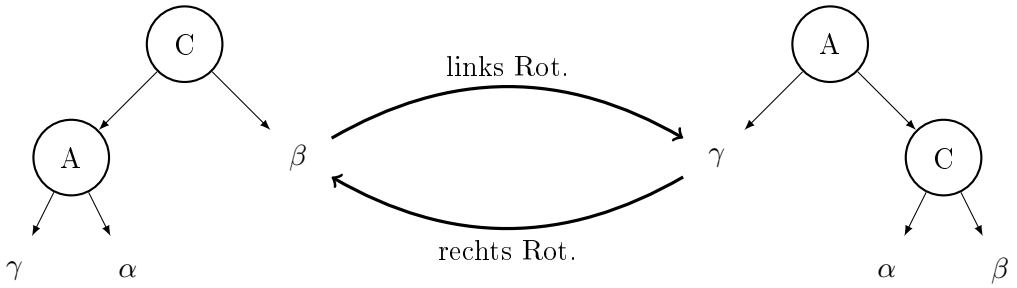

Abbildung 2: Baumrotationen Beispiel

In der Abbildung 2 kann man diese Algorithmen angewandt auf einen Unterbaum sehen. α, β und γ sind in diesem Fall beliebige Unterbäume oder könnten auch leer sein. Man kann erkennen, dass durch den Algorithmus die Inorder-Reihenfolge erhalten bleibt. Die beiden Operationen sind Inversen der jeweils anderen. Außerdem wird ein Knoten eine Ebene im Baum nach oben verschoben und dessen Parent um eine nach unten. Ein Unterbaum wechselt die Parents. [CLRS09, S. 313]

2.2 Red Black Tree

Die Motivation für RBTs ist der Worst Case von den BTs. Wenn man hintereinander einfach geordnet Elemente in diese einfügt, dann wird immer nur eins der beiden Children besetzt, je nach Ordnung der Daten. Das führt zu einer Datenstruktur, die eher einer linearen Liste gleicht und deswegen schlechtere Zeitkomplexität besitzt. Dieses Problem wird gelöst, indem man den Baum nach Einfügen und Entfernen von Knoten balanciert, d.h., die Anzahl der Knoten in jedem Path von der Root zu einem Leaf gleichgroß zu halten.

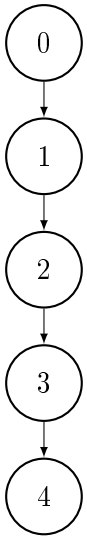

Abbildung 3: Worst Case BT

Grundsätzlich hat der Red Black Tree auch alle Eigenschaften eines normalen Binary Trees. Jedoch erhält jeder Knoten eine zusätzliche Eigenschaft, die üblicherweise als Farbe bezeichnet wird. Dabei kann ein Knoten rot oder schwarz sein.

Nun werden zusätzlich Regeln eingeführt, die ein Baum erfüllen muss, damit er ein Red Black Tree ist.

Definition 2. *Eigenschaft RBT*

1. *Die Root ist immer schwarz.*

2. *Die Children eines roten Knotens sind schwarz.*

3. *Jeder Path zwischen der Root und einem Leaf hat dieselbe Anzahl an schwarzen Knoten.*

Die 2. Regel ist äquivalent zur Aussage, dass nicht 2 rote Knoten aufeinander folgen dürfen. [Sed90, S. 220f]

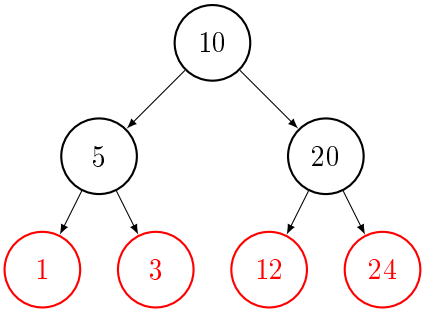

Abbildung 4: Valider RBT

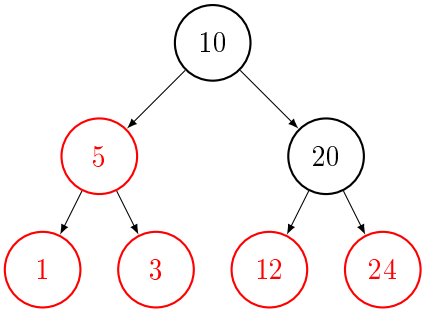

Abbildung 5: Kein RBT; Erfüllt Eigenschaften 2 und 3 nicht

Abbildung 6: Beispiel RBT

Das Balancieren funktioniert im Grunde so, dass man nachdem Knoten in den Baum eingefügt bzw. aus dem Baum entfernt wurden nur dafür sorgen muss, dass diese 3 Regeln wieder erfüllt werden. Dafür muss man die Algorithmen des normalen BT um einen Teil erweitern, der überprüft, ob irgendeine der Eigenschaften nicht erfüllt ist und dann anschließend müssen Knoten umgefärbt und Baumrotationen durchführen werden, bis das wieder für alle 3 der Fall ist.

2.2.1 Erweiterung der Algorithmen

Die **Suche** im Baum muss nicht verändert werden, da die Eigenschaften keinen Einfluss auf sie haben. [Sed90, S. 221]

Colorflip ist ein zusätzlicher Algorithmus, der dann beim Balancieren gebraucht wird. Er ändert einfach die Farben des Parents, Grandparents und des Uncles vom Knoten, auf den er angewandet wird, von rot nach schwarz bzw. umgekehrt.

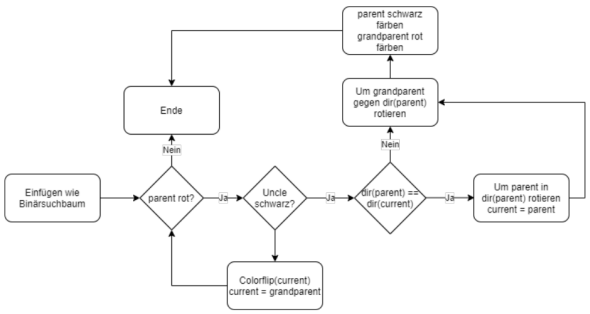

Abbildung 7: Programmablaufplan Einfügen

Beim **Einfügen** ist die erste Neuerung, dass die neuen Knoten immer rot sind. Also kann nur Eigenschaft 2 invalidiert werden, nachdem dieser Algorithmus durchgeführt wurde. Das passiert genau dann, wenn der Parent des eingefügten Knotens rot war. Man erweitert den Standardalgorithmus nun um eine weitere Funktion, die diesen Fall überprüft und dann nach bestimmten Kriterien Rotationen und Colorflips durchführt, bis der Baum wieder ein valider RBT ist.

Der Algorithmus wird iterativ auf den Baum solange angewandt, bis der Parent des

aktuellen Knotens nicht rot ist. Begonnen wird beim neu eingefügten Knoten. Der **1. Fall** tritt ein, wenn der Uncle des behandelten Knotens rot ist. Dann wird einfach ein Colorflip ausgeführt und man setzt den Algorithmus beim Grandparent fort. Im **2. Fall** ist der Uncle schwarz und der aktuelle Knoten zeigt in dieselbe Richtung wie der Parent. Dann wird eine Rotation ausgeführt in die Richtung, in die der Parent zeigt. Eine Linksrotation wenn er nach links zeigt und ansonsten eine Rechtsrotation. Das Schöne hier ist nun, dass dieser zweite Fall direkt zum **3. Fall** führt. Er kann jedoch auch erreicht werden, wenn der Parent und der aktuelle Knoten in unterschiedliche Richtungen zeigen. Dann wird in die Richtung des Parents um den Grandparent rotiert, der Parent schwarz und der Grandparent rot gefärbt und der Algorithmus wird beendet. So braucht man maximal 2 Rotationen oder $h/2$ Colorflips (h ist die Länge des längsten Paths von der Root zu einem Leaf oder wird auch Höhe des Baums genannt).

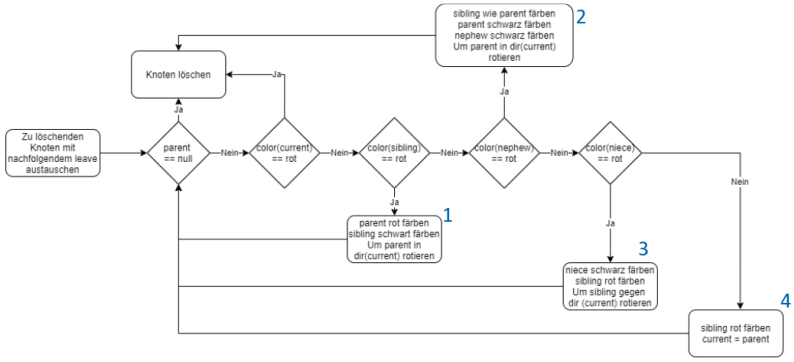

Abbildung 8: Programmablaufplan Löschen

Der letzte Algorithmus, der angepasst werden muss, ist das **Löschen** von Knoten im Baum. Es muss wieder einen Algorithmus geben, der den Baum nach dem Entfernen eines Knotens wieder zu einem validen RBT macht.

Nun kommt es zu einer Verletzung der Regeln, wenn man einen schwarzen Knoten entfernt, weil dann nicht mehr alle Paths dieselbe Anzahl von ihnen haben. Der folgende Algorithmus ist wieder als Ablaufplan in Abbildung 8 dargestellt. Auch hier gibt es wieder eine Fallunterscheidung. Diesmal jedoch mit 4 Fällen. Diese sind auch in der Abbildung entsprechend markiert. Als erstes wird einmal überprüft, ob die Root entfernt wird (also, ob der Knoten überhaupt einen Parent hat) oder ob er rot ist. In diesen Fällen ist der Algorithmus sofort fertig und man kann sicher löschen.

Anderfalls muss nun das Rebalancieren beginnen. Wenn der Sibling des aktuellen Knotens rot ist, wird der Parent rot gefärbt, der Sibling schwarz und es wird in Richtung des aktuellen Knotens um den Parent rotiert. Wenn der Sibling rot ist, werden dann Niece und Nephew überprüft. Wenn der Nephew rot ist, dann wird der **2. Fall** behandelt. In diesem Fall wird der Sibling rot eingefärbt, der Parent und Nephew schwarz und es wird in Richtung des aktuellen Knotens um den Parent rotiert. Das Besondere hier ist, dass nach diesem Fall der Baum balanciert ist und das Entfernen des Knotens durchgeführt werden kann. Sonst geht es damit weiter, dass die Farbe der Niece überprüft wird. Ist sie rot, dann wird der Algorithmus für den **3. Fall** ausgeführt. Hier wird die Niece schwarz gefärbt, der Sibling rot und dann um den Sibling in Richtung des aktuellen Knotens rotiert. Und wenn alle oben genannten Bedingungen nicht zutreffen, sind alle genannten Knoten schwarz und es wird einfach der Sibling rot gefärbt und der Algorithmus beim Parent fortgesetzt. Das ist der **4. Fall**. Die 4 Abbildungen unten illustrieren die Unterbäume für die 4 Fälle. Der Knoten, der mit x bezeichnet ist, ist der zu entfernende Knoten. Es ist ein Leaf oder Halfleaf per Definition.

Als kurze Erklärung, warum die Algorithmen so definiert sind, muss man zuerst das Problem aufzeigen. Im Unterbaum des zu löschenden Knotens fehlt ein schwarzer Knoten. Nun gibt es 2 Ansätze, dieses Problem zu lösen. Entweder man versucht einen neuen schwarzen Knoten zu erzeugen, damit die

Balance wieder hergestellt wird. Das sind die Fälle 1, 2, 3. Allerdings muss dabei sichergestellt werden, dass die anderen Regeln des RBT danach weiterhin gelten. Oder man probiert aus dem benachbarten Unterbaum einen schwarzen Knoten zu entfernen (Fall 4). Hier besteht nun das Problem, dass sich die Anzahl der schwazen Knoten der Pfade, die durch den Parent und dessen Sibling führen, unterscheiden. Also muss man auch einen schwarzen Knoten aus dem Unterbaum des Siblings des Parents entfernen. Und so weiter bis man an der Root ankommt. Auch hier kann man zeigen, dass die Anzahl der Ausführungen der einzelnen Algorithmen für die Fälle begrenzt ist, der gesamte Algorithmus also terminiert.

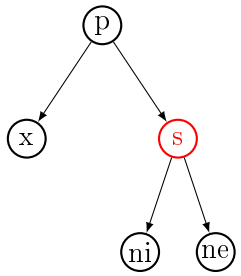

Abbildung 9: 1. Fall

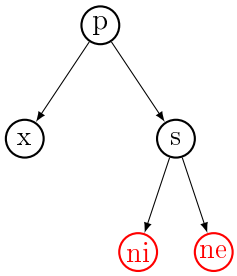

Abbildung 10: 2. Fall

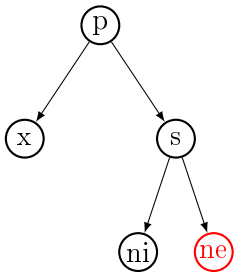

Abbildung 11: 3. Fall

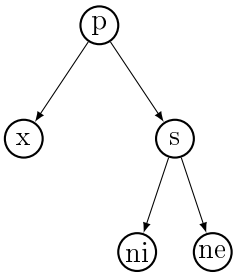

Abbildung 12: 4. Fall

[YTD]

3 Implementierung

Wie schon in den vorhergenden Kapiteln beschrieben, handelt es sich bei den Bäumen um eine Datenstruktur, in die der Nutzer beliebig Daten mit einem bestimmten Schlüssel einfügen kann.

Die Implementierung stellt deswegen 3 Funktionen bereit, mit denen man nach einem bestimmten Schlüssel im Baum suchen kann, man einen neuen Schlüssel zusammen mit einem Datensatz einfügen kann und einen Schlüssel und den Datensatz wieder aus dem Baum löschen kann.

3.1 Generics in C

Bei der Implementierung in **C** gab es dabei einige Schwierigkeiten, die man lösen musste. Es beginnt damit, dass **C** keine objekt-orientierte Programmiersprache ist und keine eingebaute Möglichkeit für Generics (GNRs) hat. Nun kann man dieses Problem auf verschiedene Weisen lösen. Den Wert, der bei jedem Knoten des Baums gespeichert werden soll, lässt sich ganz simple als Void-Pointer (`void*`) implementieren, sodass man die Daten beispielsweise auf dem Heap (HP) ablegen kann und mithilfe eines Casts den Pointer der Daten (z.B. `int*`) zu `void*` umwandeln kann. Das erlaubt es, beliebige Datentypen und sogar Structs in den Baum einzufügen. Diese Variante ist möglich, da die Nutzerdaten auf die Suche nach einem Knoten keinen Einfluss haben.

Anders ist es bei der Implementierung der Schlüssel. Hier muss sichergestellt sein, dass diese untereinander vergleichbar sind, sodass man die Suche, wie im Kapitel 2 beschrieben, durchführen kann. Man könnte Gebrauch von Unions (UNSs) in **C** machen, in denen man die numerischen Datentypen als mögliche Schlüssel anbietet.

```
union RBTreeKey {
  char c;
  short s;
  int i;
  long long l;
  float f;
  double d;
}
```

Zusätzlich müsste man dazu noch angeben, welchen Datentyp man in seinem Code nutzt (z.B. mithilfe einer Macro (MC)). Dieser Ansatz ist allerdings sehr unflexibel, weil man auf die Datentypen, die im `union RBTreeKey` vom Programmierer festgelegt sind, beschränkt ist.

Ein bessere Ansatz ist es, eine MC zu definieren, die den Typen der Schlüssel enthält. Zusätzlich kann man noch eine 2. und 3. MC erstellen, die die Kleiner-Als- und Ist-Gleich-Operatoren definieren. Das ermöglicht es, beliebige Datentypen als Schlüssel zu verwenden (sogar Structs), solange man die Vergleichsmacros richtig in der MC angeben kann.

```
#define T int
#define TEQUAL(x, y) ((x) == (y))
#define TLESS(x, y) ((x) < (y))
```

Genau diese Implementierung wurde auch gewählt. Im Usercode müssen nur der Typ `T` und die Vergleichsoperationen `TLESS` und `TEQUAL` definiert werden. Einziger Nachteil ist, dass man im selbe Quelltext nicht meherere verschiedene Varianten von Schlüsseldatentypen nutzen kann.

3.2 Knoten als Struct

Wie schon in Kapitel 2 beschrieben, sind Bäume nichts anderes als ein 3-Tupel (l, k, r). l und r sind die Unterbäume und k ist die Root des Baumes. Jeder Knoten enthält somit seinen Schlüssel und eine

Referenz auf den linken und rechten Unterbaum. In **C** wird der Baum nun als verkettete Liste von Knoten dargestellt. Im `struct Node` wird der generische Schlüssel `T *key` und zugehörige Wert `void * value` gespeichert. Zusätzlich gibt es jeweils einen Pointer zum linken und rechten Child, welche die Roots der entsprechenden Unterbäume sind. Zusätzlich wird auch noch ein Pointer auf den Parent gespeichert, da man diesen ziemlich oft in den Algorithmen zum Einfügen und Löschen von Knoten benötigt und man so den Quellcode etwas vereinfachen kann.

Natürlich gibt es im Knoten auch noch ein Feld, welches die Farbe speichert. Diese ist später wichtig, weil sie zum Balancieren des Baumes benötigt wird. Dieses findet während des Einfügens und Löschens neuer Knoten in den Baum statt. Die Werte, die die Farbe annehmen kann, werden durch die MCs `RB_TREE_RED` und `RB_TREE_BLACK` definiert.

Zusätzlich existiert ein weiterer Struct für den Baum selbst, welcher allerdings nur als Handle für die Funktionen dient. Er speichert die Root und die Anzahl der eingefügten Knoten. Diese wird benötigt, damit man die Worst-Case-Höhe des Baums berechnen kann [1].

```c
#define RB_TREE_RED     1U
#define RB_TREE_BLACK   0U

struct Node;

struct Node {
    void *value;

    T *key;

    struct Node *left;
    struct Node *right;
    struct Node *parent;

    uint8_t color;
};

struct RBTree {
    struct Node *root;

    size_t node_count;
};
```

Zusätzlich existieren 2 Helfer-Funktionen, die jeweils eine Instanz von diesem Struct für den Nutzer erstellen und auch wieder freigeben. Der Baum wird durch die `create_tree` Funktion auf dem Heap abgespeichert und es wird lediglich ein Pointer zu ihm ausgegeben. Das Löschen des Baumes wird durch `free_tree` implementiert. Dieses ist auf keinen Fall trivial, weil es Bottom-Up durchgeführt werden muss und somit nicht durch den Nutzer selbst implementiert werden sollte.

Die Signatur dieser Funktionen sieht man im unteren Listing.

```c
struct RBTree* create_tree();
void free_tree(struct RBTree *rbtree);
```

3.3 Suche nach Knoten

Eine der wichtigsten Operation, auf die auch später das Einfügen und das Löschen von Knoten aufbaut, ist die Suche im Baum. Diese ist normalerweise als rekursiver Algorithmus definiert, lässt sich aber auch ziemlich einfach iterativ implementieren. Grundsätzlich wurde im Code dieser Arbeit auf Rekursion

verzichtet und entweder ein iterativer Ansatz verwendet oder ein selbst implementierter Stack (ST), um Stackoverflows zu vermeiden.

Die Funktion, die die Suche implementiert, akzeptiert einen Pointer zum Baum, den zu suchenden Schlüssel und einen Pointer, in den der Pointer des gefundenen Knotens geschrieben werden kann, wenn vorhanden.

```
uint8_t search_node(struct RBTree* rbtree, T* key, struct Node** node);
```

Der Algorithmus selbst speichert den aktuellen Knoten in der Variable `struct Node* current`. Diese wird mit der Root initialisiert. Anschließend wird iterativ entweder der linke oder der rechte Unterbaum besucht, abhängig davon, ob der Schlüssel, nachdem gesucht wird, kleiner oder größer als der Schlüssel des aktuellen Knotens ist. Wenn er kleiner ist, wird der linke Unterbaum besucht, sonst der Rechte. Dies geschieht, in dem `current` entweder das linke oder rechte Child des aktuellen Knotens zugewiesen wird.

Die `while`-Schleife bricht ab, wenn der Schlüssel gefunden wurde oder das nächste Child `NULL` ist. Im letzteren Fall wird ein Fehlercode returned und dem Ausgabe Pointer `NULL` zugewiesen, weil der Schlüssel nicht im Baum vorhanden ist. Ansonsten kann man der Ausgabe einfach `current` zuweisen und den Erfolgswert returnen, der anzeigt, dass es keinen Fehler gab (mehr dazu in Kapitel 3.7).

```
struct Node *current  = rbtree->root;

while (current != NULL) {
    if (TEQUAL(*(current->key), *(key))) break;
    current = (TLESS(*key, *(current->key))) ? current->left : current->right;
}

if (current == NULL) {
    *node = NULL;
    return RB_TREE_KEY_ERROR;
}

*node = current;
return RB_TREE_SUCCESS;
```

3.4 Einfügen von Knoten

3.4.1 Einfügen des neuen Knotens

Eine weitere wichtige Operation ist das Einfügen von Daten in den Baum. Dabei müssen die zwei Eigenschaften der Datenstruktur erhalten bleiben. Diese sind die sortierte Reihenfolge und, dass der Baum ein valider RBT ist. Das Listing unten zeigt die Signatur der Funktion, die das Einfügen durchführt. Sie akzeptiert einen Pointer zu dem Baumstruct `struct RBTree* rbtree`, einen Pointer zum Schlüssel des neuen Knotens `T* key` und optional Daten, die auch im Knoten gespeichert werden sollen `void* value` (dieser Wert kann auch `NULL` sein).

```
uint8_t insert_node(struct RBTree* rbtree, T* key, void* value);
```

Die Implementierung sorgt zuerst dafür, dass der neue Knoten an die richtige Stelle im Baum eingefügt wird und stellt danach (wenn nötig) sicher, dass es immer noch ein valider RBT ist.

Das Einfügen des Knotens in den Baum kann nun analog zur Suche implementiert werden. Der Unterschied liegt darin, dass man den Baum durchsucht, bis man bei `NULL` ankommt. Beim Suchen war der Fehlerfall, dass es keinen Knoten mit dem zu suchenden Schlüssel gab, aber während des Einfügens

ist das die Annahme, die getroffen wird. Der Schlüssel, den der Nutzer neu hinzufügen will, sollte noch nicht im Baum enthalten sein. Damit ist der erreichte `NULL`-Pointer nach der Logik des Baums genau die Stelle, an der der neue Knoten mit dem neuen Schlüssel eingefügt werden muss.

Der Grund, warum hier der Pointer zum Vorgänger-Knoten `previous` zusätzlich gespeichert werden muss ist, dass `NULL` nicht auf eine valide Structinstanz zeigt, sondern lediglich anzeigt, dass es keinen Knoten an dieser Stelle gibt. Somit kann man auch nicht den Parent von `NULL` abfragen und man muss diese Information in einer zustätzlichen Variablen zwischenspeichern.

```
struct Node *previous = NULL;
struct Node *current  = rbtree->root;

while (current != NULL) {
    previous = current;
    current  = (TLESS(*key, *(previous->key))) ? previous->left : previous->right;
}
```

Nun wird eine neue Instanz von `struct Node` erstellt. Das erledigt die Helfer-Funktion `_create_node`, welche den neuen Knoten auf dem Heap abspeichert und den Pointer auf ihn ausgibt. In ihr wird auch direkt sichergestellt, dass der neu allozierte HP-Speicher korrekt initialisiert und die Farbe auf rot gesetzt wird. Der neue Knoten muss dann an den Parent vom erreichten `NULL`-Pointer entweder links oder rechts angehangen werden.

```
struct Node *new_node = _create_node(key, value);
if (TLESS(*key, *(previous->key))) {
    previous->left = new_node;
} else {
    previous->right = new_node;
}
new_node->parent = previous;
```

Es gibt auch noch den Spezialfall, dass ein Knoten in einen noch leeren RBT eingefügt werden soll. Hier muss dann der Pointer auf die Root im `struct RBTree` gesetzt werden. Deswegen wird, bevor der oben angebene Algorithmus ausgeführt wird noch überprüft, ob der Nutzer den leeren Baum als Eingabe in die Funktion gegeben hat. Das Gute ist, dass man in diesem Fall auch gar nicht den Baum durchsuchen muss, sondern sofort weiß, dass der neue Knoten die Root selbst ist. Es darf allerdings nicht vergessen werden, dass durch `_create_node` die Farbe des neuen Knotens auf rot gesetzt wurde. Sie muss deswegen noch zu schwarz geändert werden, weil die Root des RBT immer schwarz sein muss.

```
if (rbtree->root == NULL) {
  rbtree->root = new_node;
  rbtree->root->color = RB_TREE_BLACK;
}
```

3.4.2 Balancieren des Baums (Einfügen)

Nach dem Einfügen in den Baum kann es dazu kommen, dass die Regeln des RBTs verletzt werden. Dieser Fall tritt dann ein, wenn der Parent des neuen Knotens rot ist, denn dann sind 2 aufeinanderfolgende rote Knoten im Baum, was nicht sein darf (siehe 2).

Wenn der obige Fall eintritt, muss einer der in Kapitel 2 beschriebenen Algorithmen ausgeführt werden, damit der Baum wieder alle Eigenschaften erfüllt und ein valider RBT wird. Das hat den Nebeneffekt, dass der Baum dabei besser im Durchschnitt ausbalanciert wird. In der Implementierung wurden Helferfunktionen implementiert, die den Colorflip und die Rotationen am Baum durchführen.

Das untere Listing zeigt einen Auszug aus der Funktion, welche die Baumrotation durchführt (nur die Linksrotation).

```
struct Node *child = start_node->right;
if (start_node == rbtree->root) rbtree->root = child;
child->parent = start_node->parent;

if (start_node->parent != NULL) {
  if (start_node->parent->left == start_node) start_node->parent->left = child;
  else start_node->parent->right = child;
}

start_node->right = child->left;
if (start_node->right) start_node->right->parent = start_node;

child->left = start_node;
start_node->parent = child;
```

Hier zeigt sich der Vorteil der Implementierung als Verkettung von Pointern. Die Rotation kann einfach durch das Austauschen von Child- und Parent-Pointern implementiert werden.

Die Funktion, die den Colorflip implementiert, wird hier nicht genauer betrachtet, weil sie tatsächlich nur das `color` Feld im Struct ändert.

Nachdem die Helferfunktionen besprochen wurden, kann nun endlich mit dem Rebalancieren begonnen werden. Die Funktion, die diesen entsprechenden Algorithmus dafür implementiert, heißt `fix_tree_insert`. Ihre Signatur befindet sich im unteren Listing.

```
void fix_tree_insert(struct Node *start_node, struct RBTree *rbtree)
```

`start_node` ist dabei der Knoten, der die Eigenschaften eines RBT verletzt, also der Knoten, der zuletzt eingefügt wurde.

Unten kann man nun die Implementierung der Funktion sehen. Als erstes werden 2 Pointer angelegt, die den aktuell betrachteten Knoten und seinen Parent speichern. Das ist wichtig, weil es passieren kann, dass der Algorithmus mehrere Schritte benötigt. Das ist auch der Grund, warum sich alles innerhalb einer `while`-Schleife befindet, nämlich damit solange rebalanciert wird, bis die Abbruchbedingung erreicht wird (siehe Kapitel 2).

Wie schon in Kapitel 2 beschrieben, gibt es verschiedene Fälle, die betrachtet werden müssen. Je nach Farbe des Uncle-Knotens und Richtung des Parents werden Colorflips und Rotationen durchgeführt. Dafür können hier nun die Helferfunktionen, die in den letzten Abschnitten beschrieben wurden, genutzt werden. Der Vorteil daran ist, dass diese auch gleich noch bestimmte Fehlerfälle abfangen, sodass man sich viel Codeduplizierung ersparen kann.

Als letztes wird noch die Farbe der Root auf `RB_TREE_BLACK` gesetzt, weil es vorkommen kann, dass sie am Ende rot ist. Die Eigenschaften von RBT schreiben jedoch vor, dass die Root immer schwarz sein muss. Hier wurde bewusst auf eine `if`-Abfrage verzichtet, um Instruktionen zu sparen.

```
struct Node *current = start_node;
struct Node *parent = start_node->parent;

while (parent != NULL && parent->color == RB_TREE_RED && current->color ==
    RB_TREE_RED) {
    if (parent->parent == NULL) break;

    struct Node *uncle = get_uncle(current);

    if (get_color(uncle) == RB_TREE_BLACK) {
```

```
        if (get_direction(parent) != get_direction(current)) {
            rotate(parent, get_direction(parent), rbtree);
            current = parent;
            parent = current->parent;
        } else {
            struct Node *grandparent = get_grandparent(current);
            rotate(grandparent, !get_direction(parent), rbtree);
            parent->color = RB_TREE_BLACK;
            grandparent->color = RB_TREE_RED;
            break;
        }
    } else if (get_color(uncle) == RB_TREE_RED) {
        color_flip(current);
        if (parent->parent == NULL) break;

        struct Node *uncle = get_uncle(current);
        current = get_grandparent(current);
        parent = current->parent;
    }
}

rbtree->root->color = RB_TREE_BLACK;
```

3.5 Löschen von Knoten

Die letzte Operation, die die Daten im Baum verändert, ist das Löschen von Knoten. Sie ist wohl auch der komplizierteste Algorithmus, weil sie sehr viele Fälle betrachtet und viele Grenzfälle abfangen muss.

In meiner Implementierung findet das Löschen in 3 Schritten statt. Als erstes wird der Knoten, der gelöscht werden soll, mit einem Leaf des Baumes ausgetauscht, falls er noch keines ist. Danach kann dieser sicher entfernt werden, ohne dass der Baum in mehrere Bäume zerfällt. Anschließend wird wieder die Balancierung durchgeführt. Alle diese Algorithmen wurden wieder in separate Funktionen extrahiert.

Die Funktion, die der Nutzer meiner Datenstruktur aufruft, um einen Knoten zu entfernen, heißt `delete_node`. Sie hat folgende Signatur:

```
uint8_t delete_node(struct RBTree* rbtree, T* key);
```

`T* key` ist hier der Schlüssel des Knotens, der gelöscht werden soll.

Es wird damit begonnen, dass der Knoten mit dem entsprechenden Schlüssel `key` gesucht wird. Dafür kann die `search`-Funktion verwendet werden, die im Kapitel 3.3 etabliert wurde.

```
struct Node *node_to_delete = NULL;
search_node(rbtree, key, &node_to_delete);
```

Als nächstes wird dann der Algorithmus durchgeführt, der den zu löschenden Knoten mit einem Blatt austauscht. Das Blatt, mit dem getauscht wird, ist so zu wählen, dass nach der Löschung die Inorder-Traversierungsreihenfolge korrekt ist. Das alles wird durch die Funktion `swap_to_leaf` implementiert. Sie gibt dann den Pointer zum Blatt aus `x`, welches entfernt werden kann.

```
struct Node *x = swap_to_leaf(node_to_delete);
```

Dann wird noch die `fix_tree_delete`-Funktion aufgerufen. Sie ist das Äquivalent zu `fix_tree_insert`, welches nach dem Einfügen den Baum balanciert, aber etwas komplizierter in ihrer Implementierung.

```
fix_tree_delete(x, rbtree);
```

Jetzt wird noch überprüft, ob die Root gelöscht werden soll, was bedeutet, dass sie der einzige Knoten im Baum ist. Hier müssen dann einige Felder im `rbtree`-Struct verändert werden. Es werden dann alle Pointer, die auf den Knoten zeigen, zu `NULL` geändert, was ihn effektiv aus dem Baum entfernt. Danach wird der zu löschende Knoten freigegeben.

```
if (rbtree->root == x) {
    _free_node(rbtree->root);
    rbtree->root = NULL;
    return RB_TREE_SUCCESS;
}

if (get_direction(x) == RB_TREE_LEFT_CHILD) x->parent->left = NULL;
else x->parent->right = NULL;

_free_node(x);
```

Es wurden in diesem einführenden Abschnitt einige Funktionen genannt, die noch als Blackbox betrachtet wurden. Auf deren Funktionsweise wird in den nächsten Kapiteln näher eingegangen.

3.5.1 Austauschen mit Blatt

Das Austauschen findet in der Funktion `swap_to_leaf` statt, deren Signatur sich im unteren Listing befindet.

```
struct Node* swap_to_leaf(struct Node *node_to_delete);
```

`node_to_delete` ist der Pointer zum Knoten, der mit einem Leaf getauscht werden soll. Das darf aber nicht irgendein Leaf sein, sondern das, welches den nächstkleineren bzw -größeren Schlüssel hat, damit nach dem Entfernen von `node_to_delete` die Inorder-Reihenfolge erhalten bleibt.

In der Funktion selbst muss nun der Knoten mit der oben genannten Eigenschaft gefunden werden. Dies geschiet, in dem man erst im linken Unterbaum von `node_to_delete` des Child sucht, welches sich am weitesten rechts befindet. Und falls es das nicht gibt, sucht man im rechten Unterbaum nach dem Knoten, der sich am weitesten links befindet. Das sind die Knoten, die in der Inorder-Reihenfolge direkt vor bzw. nach dem zu löschenden Knoten kommen. Das wird durch die Helfer-Funktionen `get_next_smallest` bzw. `get_next_largest` implementiert. Im unteren Listing kann man beispielhaft die Implementierung einer dieser Funktionen sehen:

```
void get_next_largest(struct Node *start, struct Node **next_largest)
{
    struct Node *current  = start;
    while (current->left != NULL) current = current->left;
    *next_largest = current;
}
```

Wenn keiner der beiden oberen Fälle eintritt, dann ist `note_to_delete` bereits ein Leaf und der Algorithmus gibt den Knoten selbst aus.

Ansonsten werden mithilfe der Funktion `swap` die Schlüssel und Daten des Leafs und von `note_to_delete` einfach ausgetauscht und es wird der Pointer zum Leaf ausgegeben. Unten kann man die Implementierung für den Fall des linken Unterbaums sehen. Die für den rechten Unterbaum kann man im Code nachschlagen.

```
struct Node *next_smallest =  NULL;
get_next_smallest(node_to_delete->left, &next_smallest);
```

```
leaf = next_smallest;
swap((void**)&node_to_delete->key, (void**)&next_smallest->key);
swap(&node_to_delete->value, &next_smallest->value);
```

3.5.2 Balancieren des Baumes (Löschen)

Ähnlich wie beim Einfügen in den Baum, muss auch nach dem Entfernen aus dem Baum rebalanciert werden. Die Signatur der Funktion, die den entsprechenden Algorithmus implementiert, ist so definiert:

```
void fix_tree_delete(struct Node *x, struct RBTree *rbtree);
```

`x` ist hier der Pointer auf den Knoten, der aus dem Baum gelöscht werden soll. Man kann hier annehmen, dass er ein Leaf oder Halfleaf ist, denn die `swap_to_leaf`-Funktion wird vorher auf `node_to_delete` angewandt, sodass immer nur Leaves oder Halfleaves gelöscht werden müssen.

Ähnlich wie beim Einfügen gibt es auch hier wieder Fälle, die der Code abarbeiten muss, abhängig von der Farbe bestimmter Knoten. Der Algorithmus befindet sich in einer `while`-Schleife, weil es auch hier wieder passieren kann, dass es mehrere Durchläufe gibt, bevor der Baum balanciert ist.

Die ersten beiden Bediengungen `x->parent == NULL` und `get_color(x) == RB_TREE_RED` terminieren den Algorithmus sofort. Die erste Bediengung trifft genau dann zu, wenn die Root entfernt wird. Danach entsteht allerdings der leere Baum, der nach Definition balanciert ist, also muss man hier nichts mehr tun. Da die Eigenschaften von RBT beim Löschen nur dann verletzt werden, wenn man einen schwarzen Knoten entfernt, kann man auch sofort abbrechen, wenn die Farbe von `x` rot ist, was der zweiten Bediengung entspricht.

Alle anderen Zweige des `if`-Konstrukts entsprechen nun den 4 Fällen. Wie diese sich genau aufbauen, und warum sie so definiert sind, wurde bereits in Kapitel 2 beschrieben. Deswegen wird dieser Teil hier nicht noch einmal im Detail erklärt.

Was es noch anzumerken gibt ist, dass der Fall 2 ebenfalls zum Terminieren des Algorithmus führt, weswegen sich dort ein `break` am Ende des Codeblocks befindet. Es ist durch die Definition des Algorithmus sichergestellt, dass dieser immer terminiert, also irgendwann die Root erreicht oder den Fall 2 (siehe Kapitel 2).

Das untere Listing zeigt die gesamte Implementierung der Funktion.

```
while(1) {
    if (x->parent == NULL) {
        break;
    } else if (get_color(x) == RB_TREE_RED) { break; }
      else if (get_color(get_sibling(x)) == RB_TREE_RED) {
        x->parent->color = RB_TREE_RED;
        get_sibling(x)->color = RB_TREE_BLACK;
        rotate(x->parent, get_direction(x), rbtree);
    } else if (get_color(get_nephew(x)) == RB_TREE_RED) {
        get_sibling(x)->color = x->parent->color;
        x->parent->color = RB_TREE_BLACK;
        get_nephew(x)->color = RB_TREE_BLACK;
        rotate(x->parent, get_direction(x), rbtree);
        break;
    } else if (get_color(get_niece(x)) == RB_TREE_RED) {
        get_niece(x)->color = RB_TREE_BLACK;
        get_sibling(x)->color = RB_TREE_RED;
        rotate(get_sibling(x), !get_direction(x), rbtree);
    } else {
        get_sibling(x)->color = RB_TREE_RED;
```

```
21          x = x->parent;
22      }
23  }
24  x->color = RB_TREE_BLACK;
```

[YTD]

3.6 Generische Baumtraversierung

Wie schon in der Einleitung erwähnt, bieten BTs die Möglichkeit, Daten geordnet zu speichern. Nun benötigt man auch Möglichkeiten, auf die Daten in einer bestimmten Reihenfolge zuzugreifen. Grundsätzlich gibt es 3 Optionen, die man als Traversierung bezeichnet, nämlich Inorder-, Preorder- und Postorder-Traversierung. Alle diese Algorithmen sind rekursiv definiert. [Sed90, S. 44ff] Der untere Pseudocode stellt dar, wie sie funktionieren.

Algorithm 3 Traversierungs-Algorithmen

```
1:  procedure PREORDER(nodeptr, VISIT)
2:      if noteptr == NULL then return
3:      VISIT(noteptr)
4:      PREORDER(noteptr→left)
5:      PREORDER(noteptr→right)
6:  procedure INORDER(nodeptr, VISIT)
7:      if noteptr == NULL then return
8:      INORDER(noteptr→left)
9:      VISIT(noteptr)
10:     INORDER(noteptr→right)
11: procedure POSTORDER(nodeptr, VISIT)
12:     if noteptr == NULL then return
13:     POSTORDER(noteptr→left)
14:     POSTORDER(noteptr→right)
15:     VISIT(noteptr)
```

[Knu68, S. 318ff]

Dabei ist `noteptr` der Pointer der Root des Unterbaums (der ganze Baum ist auch ein Unterbaum von sich selbst). Und `VISIT` stellt dabei einen Funktionspointer dar, der einen `noteptr` als Parameter akzeptiert und ebenfalls als Argument in die Traversierungsfunktionen gegeben wird. So ist es dem Nutzer möglich, eine bestimmte Funktion auf alle Knoten anzuwenden.

```
1  uint8_t inorder_traversel(struct RBTree *rbtree, void (*visit)(struct Node*))
2  {
3      struct Node *current = rbtree->root;
4      struct Stack *stack = create_stack(_calc_worst_case_height(rbtree));
5
6      while (current != NULL || !is_stack_empty(stack)) {
7          if (current != NULL) {
8              push(stack, current);
9              current = current->left;
10         } else {
11             struct Node *stack_node;
```

```
            pop(stack, &stack_node);
            visit(stack_node);
            current = stack_node->right;
        }
    }
    free_stack(stack);
    return RB_TREE_SUCCESS;
}
```

Im oberen Listing kann man am Beispiel der Inorder-Traversierung erkennen, wie die Algorithmen iterativ implementiert wurden. Anstatt den Hardware-ST zu verwenden, wurde ein Software-ST `struct Stack *stack` implementiert. In diesen werden jeweils die noch zu besuchenden Unterbäume gepushed, und wenn der aktuelle Knoten `current NULL` ist, wird ein neuer Knoten aus dem ST geholt. Dieses Vorgehen simuliert das Verhalten einer rekursiven Funktion, jedoch besteht bei sehr großer Rekursionstiefe nicht die Möglichkeit eines Stackoverflow. Die anderen Traversierungsarten wurden auf dieselbe Weise implementiert.

Es kommt auf die genutzte Traversierungsart an, in welcher Reihenfolge man die Knoten besucht. Wenn man Inorder nutzt, dann bekommt man sie nach Schlüsseln aufsteigend sortiert. Das kann man z.B. dafür nutzen, die Schlüssel in geordneter Reihenfolge auszugeben. Bei der Postorder-Traversierung wird die Root des Unterbaums als letztes besucht. So kann man Bottom-Up-Algorithmen implementieren, die bei den Leaves beginnen und die Root des Baums als letztes behandeln.

Ein sehr praktischen Bespiel dafür ist das Freigeben der RBTs. Dabei müssen zuerst die Unterbäume aus dem HP gelöscht werden und danach die Root, weil man sonst die Unterbäume zwischenspeichern müsste, da die Pointer zu ihnen in der Root gespeichert sind. Wenn man diese aber nun als erstes freigibt, würde man keinen Zugriff mehr auf ihre Unterbäume haben.

```
void _free_node(struct Node *node)
{
    if (node->key != NULL) free(node->key);
    if (node->value != NULL) free(node->value);
    free(node);
}

void free_tree(struct RBTree *rbtree)
{
    postorder_traversel(rbtree, &_free_node);
    free(rbtree);
}
```

Hier wurde die `VISIT`-Funktion als `_free_node` implementiert, welche den Schlüssel und den Wert des besuchten Knotens freigibt und dann den Knotenstruct selbst.

3.7 Fehlerbehandlung

Wie man in vielen vorhergenden Listings bereits sehen konnte, haben die meisten Funktionen als Ausgabetypen `uint8_t`. (Die Fehlerbehandlung wurde aus den Listings jedoch meistens weggelassen, damit sie nicht zu lang werden.) Das liegt daran, dass die Funktionen über den Ausgabewert Fehler an den Nutzer zurückgeben. Diese Fehler wurden als MC definiert und durch sie können verschiedene Fehlerarten unterschieden werden.

```
#define RB_TREE_SUCCESS              0U
#define RB_TREE_OUT_OF_MEM           1U
#define RB_TREE_KEY_ERROR            2U
```

```
4 #define RB_TREE_NULL_ERROR          3U
5 #define RB_TREE_DUPLICATE_KEY_ERROR 4U
```

In meiner Implementierung repräsentiert die 0 immer, dass es keinen Fehler gab. `RB_TREE_OUT_OF_MEM` wird von Funktionen verwendet, die dynamisch Speicher allozieren und das Betriebssystem nicht mehr genug zur Verfügung stellt. Die einzige Funktion, die den Fehler nutzt, ist `insert_node`, nämlich genau dann, wenn kein neuer Knoten erstellt werden kann. Hierbei wird der Fehler allerdings nicht ausgegeben, sondern das Programm wird mit `exit(RB_TREE_OUT_OF_MEM)` beendet. Ein `RB_TREE_KEY_ERROR` wird von `search_node` ausgegeben, wenn der zu suchende Schlüssel im Baum nicht existiert. Funktionen, die Pointer als Argumente akzeptieren, geben einen `RB_TREE_NULL_ERROR` Fehler aus, wenn mindestens einer dieser Pointer `NULL` ist. Und ein `RB_TREE_DUPLICATE_KEY_ERROR` wird ausgegeben, wenn man `insert_node` einen Schlüssel übergibt, der bereits im Baum existiert und man doppelte Schlüssel und implizites Überschreiben mithilfe der `RB_TREE_DUPLICATE_KEYS` MC deaktiviert hat.

4 Implementierungsvarianten

4.1 Rekursive Implementierung

Wie schon häufig im Kapitel 3 erwähnt, kann man auch die Suche und die Traversierung direkt als rekursive Funktionen, die sich selbst aufrufen, implementieren. Das wurde auch durchgeführt. Im Kaptiel 5.3 kann man die Unterschiede bezüglich der Laufzeit der Programme sehen. Interssierte Leser können sich den Quelltext in der Datei `rbtree_h.c` ansehen. Auf eine genauere Diskussion der Implementierung wird hier verichtet.

4.2 Kekule Zahlen

Theoretisch gibt es auch eine Möglichkeit die Baumstruktur als ARR von Knoten darzustellen und bessere Cacheladezeiten zu bekommen. Das größte Problem bei der Implementierung als verkettete Datenstruktur ist, dass die einzelnen Knoten stark verstreut im Heap abgespeichert sind und somit es zu vielen Cache Misses kommt. Diese Streuung kommt durch die Implementierung von `malloc` in der **C** -Standart-Bibliothek zustande ("The order and contiguity of storage allocated by successive calls to malloc() is unspecified."[IEE]). Das verlangsamt das Programm.

Kekule-Zahlen sind eine spezielle Variante, das Array zu indizieren, um zwischen den Elementen eine konkrete Parent-Child-Beziehung herzustellen. Die Root würde den Index 0 erhalten. Die linken und rechten Children dann jeweils den Index $2 \cdot n+1$ und $2 \cdot n+2$ (n ist der Index des Parents). So erhält jeder Knoten einen eindeutigen Index. Leider ist der Red-Black-Tree kein vollständig balancierter Binary Tree, sodass es im Worst Case (Kapitel 6) passieren kann, dass sehr viel Speicher verschwendet wird, weil sehr viele Stellen im ARR unbesetzt bleiben. Deswegen habe ich mich dagegen entschieden und habe die verkettete Variante der Datenstruktur implementiert.

4.3 n-rote Knoten erlauben

Als Generalisierung von RBT könnte man sich einen n-RBT vorstellen, in dem es erlaubt ist, dass maximal n rote Knoten aufeinander folgen. ($n \in \mathbb{N}$). Ich vermute, dass dadurch die Zeit, die das Löschen und Einfügen in den Baum benötigen, verringert würde, weil die Fixup-Algorithmen nicht so oft ausgeführt werden müssten bzw. eher terminieren könnten.[Bay72, S. 301] Nachteil ist, dass die Länge eines farblich alternierenden Pfades im n-RBT $n+1$-mal so lang werden kann, wie die eines

auschließlich schwarzen Pfads. (Ohne Beweis). Das würde die Performance der Suche der Datenstruktur wieder verschlechtern. Da die Suche viel häufiger gebraucht wird als das Einfügen und Löschen, ist der n-RBT mit $n > 0$ keine gute Alternative zum 0-RBT. Ich konnte auch keine Implementierungen von ihm finden. Eine Implementierung und richtige Benchmarks müssten durchgeführt werden, um die hier aufgestellten Behauptungen zu belegen, was aber aus Platz und Zeitgründen nicht durchgeführt wurde.

5 Benchmarks

Nun da erklärt wurde, wie sich RBTs von BTs unterscheiden und beschrieben wurde, welche komplizierten Operationen durchgeführt werden müssen, damit der RBT balanciert wird, stellt sich natürlich die Frage, ob dadurch wirklich eine bessere Laufzeit erzielt wird. In den folgenden 2 Abschnitten geht es darum zu überprüfen, wie sich die Implementierung laufzeittechnisch verhält, wenn man eine große Anzahl an Knoten in die Datenstruktur einfügt. Es soll verglichen werden, ob die RBT tatsächlich schneller sind als die BT und ob meine Behauptung, dass iterativ schneller als rekursiv ist, zumindest in diesem Fall stimmt.

5.1 Durchführung

Für die Benchmarks wurde im **C** -Code eine `main`-Funktion erstellt, die die Anzahl an Knoten, mit denen getestest werden soll, als Kommandozeilenparameter erwartet. Anschließend werden zwei ARRs erstellt, die mit zufällig generierten Daten befüllt werden. Das erste enthält Daten, die in die Bäume als Schlüssel eingefügt werden sollen und das zweite Daten, nach denen gesucht wird.

Begonnen wird mit dem RBT. Er wird zuerst mit den Daten befüllt, danach wird nach den Schlüsseln im zweiten ARR gesucht. Dann wird seine Höhe bestimmt, welche ein sehr wichtiges Kriterium für die Laufzeit ist, und am Ende werden die Knoten alle wieder entfernt. Die Zeit, die die Operationen brauchen, wird direkt im Code gemessen und dann im CSV Format in den `stdout` ausgegeben. Dieselbe Prozedur wird anschließend auch noch für den BT durchgeführt. Das Testen mit verschiedenen Knotenanzahlen wurde durch ein Pythonscript automatisiert, welches das Programm kompiliert und dann mit verschiedenen Knotenanzahlen ausführt. Begonnen wird dabei bei 100.000 Knoten. Diese Anzahl wird in Schritten von 50.000 bis auf 2.000.000 erhöht. Damit zufällige Schwankungen ausgeschlossen werden, wird jeder Test mit einer bestimmten Anzahl 15 mal durchgeführt. Am Ende wird das arithmetische Mittel der Messwerte gebildet.

5.2 Red-Black-Tree vs. Binary Tree

Besonders wichtig ist es zu zeigen, dass unsere "Verbesserung"normaler Binary Trees tatsächlich performanter ist. In nebenstehender Abbildung kann man die Ergebnisse des Benchmarks erkennen. Die Graphen, die mit dem Index $_{rb}$ beschriftet sind, sind die des RBT und die anderen gehören zum BT. Man kann erkennen, dass alle Operationen beim RBT grundsätzlich schneller als die der Standardimplementierung sind. Der Unterschied zwischen den beiden Datenstrukturen ist nicht konstant, sondern nimmt mit steigender Anzahl an Knoten immer weiter zu.

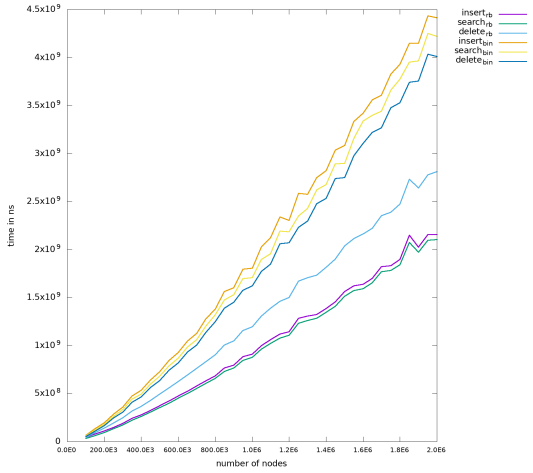

Abbildung 13: Vergleich Laufzeit RBT/BT

Den Grund dafür kann man sehen, wenn man die Höhen der beiden Bäume vergleicht. Man kann sehen, dass sich der RBT viel näher an der Höhe des vollständig balancierten Baumes befindet, als der BT. Das sorgt dafür, dass man beim RBT im Durchschnitt nicht so viele Vergleiche benötigt, bis man den Knoten erreicht, den man sucht oder einen `NULL`-Pointer, bei dem man einen neuen Knoten in den Baum einfügt. Das macht sich in den Benchmarks eindeutig bemerkbar.

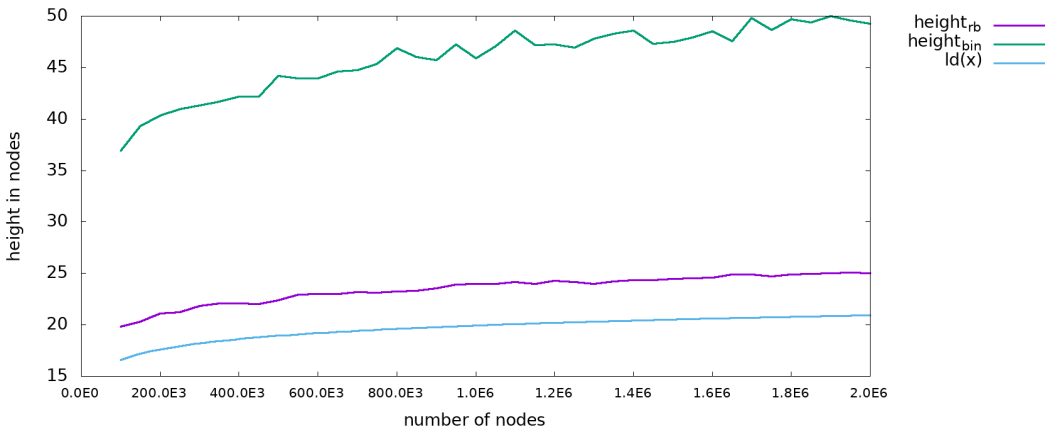

Abbildung 14: Vergleich der Höhen

Eine weitere interessante Beobachtung ist, dass die Suche und das Einfügen beim RBT ungefähr gleich schnell sind, während das Löschen etwas langsamer als diese beiden Operationen ist. Einen so starken Unterschied gibt es beim BT nicht. Meine Hypothese dafür ist, dass das Rebalancieren beim Löschen von Knoten sehr viel langsamer ist, als das Rebalancieren beim Einfügen, weil der Algorithmus häufig den ganzen Baum bis zur Root durchgeht.

5.3 Rekursiv vs. Iterativ

Eine wichtige Frage, die sich während der Implementierung gestellt hat ist, ob man die Operationen vom RBT iterativ oder rekursiv implementiert. Rekursion wäre zwar die erste Sache, an die man dabei denkt, weil die Definition von Bäumen auch meistens rekursiv ist. Außerdem werden auch alle Algorithmen rekursiv beschrieben. Das Problem daran ist, dass durch sehr große Rekursionstiefe eventuell Leistungsprobleme entstehen können und teilweise auch sehr viel Speicher nötig ist. Deswegen wurde, wie schon in vorhergenden Kapiteln beschrieben, eine iterative und eine rekursive Version der Datenstruktur erstellt, um diese zu vergleichen. Das Benchmark wurde wie im vorhergehenden Abschnitt 5.2 durchgeführt, nur wurden hier die 2 verschiedenen Versionen des Programms getestet. Dafür wurde das Pythonscript verändert, sodass es zuerst die iterative Variante kompiliert und die Tests ausführt und in eine Datei schreibt und anschließend das selbe für die Rekursive durchführt.

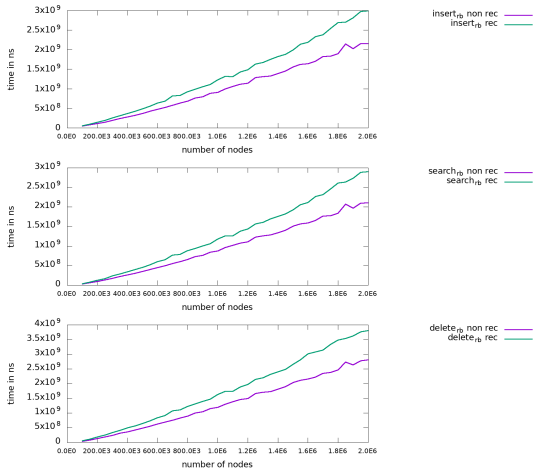

Abbildung 15: Vergleich Rekursive/Iterative Implementierungen

Die Ergebnisse sind in nachfolgenden Graphen in Abbildung 15 zu sehen. Es gibt jeweils einen Graph für die Suche, das Einfügen und das Löschen.

Man kann in allen drei Fällen erkennen, dass die iterative Variante (non rec) ab einer sehr großen Anzahl von Knoten konsistent schneller ist als die Rekursive (rec). Meine Vermutung ist, dass bei Bäumen mit einer sehr großen Baumhöhe sehr viele Stackframes aufgebaut und auch wieder abgebaut

werden müssen, wenn der rekursive Code ausgeführt wird. Bei meiner anderen Implementierung gibt es einen Softwarestack, welcher nur die nötigen Daten speichert und sich auch nicht vergrößern muss, weil er schon mit der richtigen Größe erstellt wird. Das könnte für den signifikanten Unterschied in der Laufzeit der beiden Varianten führen.

6 Zeitkomplexität

Der RBT hat im Normal Case (NC) die selbe Komplexität wie der BT. Dort gibt es keine Verbesserung. Was man jedoch sehen kann ist, dass im Worst Case (WC) die Komplexität von $O(ld(n))$ erhalten bleibt, während sie beim BT zu $O(n)$ wird. Das liegt daran, dass die Standardimplementierung der Binary Trees keine Balancierung vornimmt. D.h., wenn man Daten in sortierter Reigenfolge einfügt, erhält man eine Linked List, weil immer nur einer der Unterbäume besetzt wird, je nachdem ob die Daten aufsteigend oder absteigend sortiert sind. Die Eigenschaft, dass man nach jedem Vergleich ca. die Hälfte der Daten nicht mehr abarbeiten muss, geht dadurch verloren. Nun ist es so, dass man Baumstrukturen genau zum effizienten Suchen verwenden möchte. Aus diesem Grund ist es besonders wichtig, dass man die schnelle Suche auch im Worst Case beibehält.

Abbildung 16: Zeit-Komplexität

-	Suche	Einfügen	Löschen
Binary Tree (NC)	$O(ld(n))$	$O(ld(n))$	$O(ld(n))$
Binary Tree (WC)	$O(n)$	$O(n)$	$O(n)$
Red Black Tree (NC)	$O(ld(n))$	$O(ld(n))$	$O(ld(n))$
Red Black Tree (WC)	$O(ld(n))$	$O(ld(n))$	$O(ld(n))$

[Sed90, S.215, S. 227f]

Der WC beim RBT wird dann erreicht, wenn es einen alternierenden Path gibt, d.h., einen Path, in dem sich die Farben der Knoten immer abwechseln. Dieser kann die doppelte Länge im Vergleich zu einem Path erreichen, in dem sich nur schwarze Knoten befinden. Dieses Beispiel zeigt auch noch einmal, dass es sich bei RBT nicht um vollständig balancierte Bäume handelt. Sie sind lediglich balancierte Bäume, die den WC, zu einer linearen Datenstruktur zu werden, vermeiden. In der unteren Abbildung sind die Knoten, in denen eine dick gedruckte Linie **endet** jeweils rot und alle anderen schwarz.

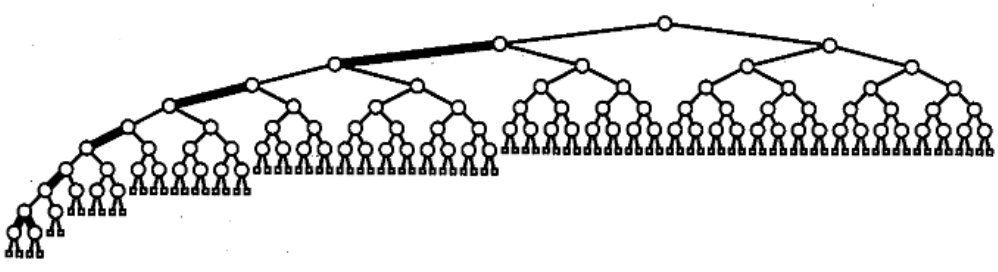

Abbildung 17: Worst Case RBT: Path mit alternierender Farbe

[Sed90, S. 228]

7 Fazit

In der Arbeit wurden die beiden Datenstrukturen RBT und BT erst theoretisch definiert und dann eine praktische Implementierung des RBT erfolgreich umgesetzt. Es wurde ebenfalls ein BT implementiert, worauf aber nicht direkt eingegangen wurde. Seine Implementierung wurde lediglich für das Benchmarking benötigt. Es wurden außerdem verschiedene Ansätze aufgezeigt, wie man einen RBT direkt umsetzten kann, und klar gemacht, warum welcher Ansatz gewählt wurde und welche Vor- und Nachteile das mit sich bringt.

Durch ausführliches Benchmarking konnte aufgezeigt werden, dass der RBT tatsächlich eine bessere Performance in Bezug auf die Laufzeit aufweist. Das wurde nochmal theoretisch mit der Diskussion der Zeitkomplexität untermauert.

Dem Leser wurde ein umfangreicher Überblick über das Thema geboten und Möglichkeiten, dieses weiter zu erkunden, indem er die noch offenen Fragen selbst versucht, zu beantworten.

Fußnoten

[1]Die Worst-Case-Höhe wird für eine kleine Optimierung benötigt. Im Kapitel 3.6 wird beschrieben, wie man die Bäume traversiert. Das ist normalerweise eine rekursive Operation, die ich jedoch iterativ implementiert habe. Dafür werden STs benötigt, die sich dynamisch vergrößern, falls sie überlaufen würden. Ich habe diese Stacks jedoch so implementiert, dass man sie mit einer initialen Größe erstellen kann. Die Worst-Case-Höhe des Baums gibt nun ungefähr die größte Rekursionstiefe an, welche bei mir der größten Anzahl an Elementen auf dem Stack entspricht. Somit kann ich dem Stack von Anfang an schon genug Platz geben, sodass er sich nicht so oft vergrößern muss, was mit einem Aufruf von `reallocarray()` verbunden. Dabei wird ein großer Speicherbereich kopiert, was das Programm unnötig verlangsamt, wenn man zu oft mehr Speicher im Stack benötigt.

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Seminararbeit selbständig und ohne unerlaubte Hilfe angefertigt und andere als die in der Seminararbeit angegebenen Hilfsmittel nicht benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Literatur

[Bay72] R. Bayer. Symmetric binary b-trees: Data structure and maintanance algorithms. *Acta Informatica*, 1:290–306, 1972.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[IEE] The open group base specifications issue 7, 2018 edition ieee std 1003.1-2017 (revision of ieee std 1003.1-2008). `https://pubs.opengroup.org/onlinepubs/9699919799/functions/malloc.html`, zuletzt besucht 29.09.2021.

[Knu68] Donald E. Knuth. *The Art of Computer Programming - Volume 1 Fundamental Algorithms.* Addison-Wesley Professional, 3 edition, 1968.

[San08] Kurt Mehlhorn & Peter Sanders. *Algorithms and Data Structures. The Basic Toolbox.* Springer Verlag, 2008.

[Sed90] R. Sedgewick. *Algorithms in C.* Addison-Wesley Professional, 1990.

[YTD] Red-black tree deletion - data structures and algorithms 34. `https://www.youtube.com/watch?v=bDT1woMULVw`, zuletzt besucht: 12.10.2021.