Seminararbeit Red-Black Trees

Herr Yannik Höll

6. Oktober 2021

Inhaltsverzeichnis

Glossar

Array TODO. 2, 17

Benchmark Test für die Zeitkomplexität und Speicherkomplextät von Software. 2, 3

BT Binary Tree. 2, 3, 10

Datenstruktur Strukturen, die Daten effizient speichern und einen effizienten Zugriff auf diese erlauben [San08, S. VIII]. 2, 3

Generic TODO. 2, 5

Heap TODO. 2, 5, 8, 11

Macro TODO. 2, 5, 6, 12

Pseudocode TODO. 2, 10

Quelltext Gesamtheit der Anweisungen eines Computerprogramms. 2, 3, 6

RBT Red Black Tree. 2, 3, 8, 9, 11

Rekursion Überbegriff für Programme oder Funktionen, die sich selbst aufrufen oder durch sich selbst definiert sind. [Sed90, S. 52] (z.B.

$$n! := \begin{cases} n \cdot (n-1)!, & \text{wenn } n \neq 0 \\ 1, & \text{wenn } n = 0 \end{cases}, \ n \in \mathbb{N}$$

). 2, 7

Rekursionstiefe TODO. 2, 11, 17

Stack TODO. 2, 7, 11, 17

Stackoverflow TODO. 2, 7, 11

Struct TODO. 2, 5–7, 11

Traversierung TODO. 2, 10

Union TODO. 2, 5

Worst Case Konfiguration der Daten in einer Datenstruktur, bei der sie am schlechtesten arbeitet (z.B. besonders viel Zeit zur Datenverarbeitung benötigt oder besonders viel Speicher benötigt). 2, 3

Worst-Case-Höhe TODO. 2, 6

1 Einleitung

Eine Möglichkeit Daten so geordnet zu speichern sind so genannte Baumdatenstrukturen. Diese Speichern Werte geordnet nach einem bestimmten Schlüssel. Eine sehr einfache Baumimplemntierung ist der Binary Tree (BT) welcher jeweils nur 2 Abzweigungen pro Knoten besitzt (rigorose Baumdefinition in Kapitel 2). Leider hat diese naiive Variante der Datenstruktur einige Probleme, welche vorallem beim Einfügen der Daten in geordneter Reigenfolge entstehen. Um diese Probleme 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 Red Black Trees (RBTs).

Diese Arbeit beschreibt wie sich RBT 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 und den Implementierungen einzugehen. Zudem werden die Ergenisse von Benchmark analysiert, welche zeigen sollen, dass RBTs tatsächlich besser Laufzeiteigenschaften haben als die Standimplementierung der BTs. Außerdem wird auf die Speicher- und Zeitkomplexität der beiden Datenstrukturen eingegangen und wie sich vorallem der RBT im Worst Case verhält.

2 Definition Binary Tree & Red Black Tree

3 Implementierung

Wie schon in den vorhergenden Kapiteln beschrieben, handelt es sich bei den Bäumen um eine generische 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 man 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 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 aus Void-Pointer (`void*`) implementieren, sodass man die Daten beispielsweise auf dem Heap ablegen kann und mithilfe eines Casts den Pointer der Daten (z.B. `int*`) zu `void*` umwandeln kann. Das erlaubt es, beliebige Datentypen und sogar Struct 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 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). 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 Macro zu definieren, die den Typen der Schlüssel definiert. Zusätzlich kann man noch eine 2. und 3. Macro definieren, die die Kleiner-Als- und Ist-Gleich-Operatoren definieren. Das ermöglicht es beliebige Datentypen als Schlüssel zu verwenden (sogar Structs), solange man die Vergleichsmacros definieren 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 selben Quelltext nicht meherere verschidene Varianten von Schlüsseldatentypen nutzen kann.

3.2 Knoten als Struct

Wie schon in Kapitel 2 beschrieben, sind Bäume nichts anders als ein 3-Tupel (l, k, r). l und r sind die Unterbäume und k ist die Wurzel des Baumes (hier als k bezeichnet wegen Schlüssel $\widehat{=}$ Key). 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 [1]. Im `struct Node` haben wir somit den generischen Schlüssel `T *key` und zugehörigen Wert `void *value`. Zusätzlich speichern wir Pointer zum linken und rechten Child, welche die Wurzeln der entsprechenden Unterbäume sind. Zusätzlich wird auch noch ein Pointer zum Parent gespiechert, da man diesen ziemlich oft in den Algorithmen zum Einfügen und Löschen von Knoten benötogt und man so den Quellcode etwas vereinfachen kann.

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

Zusätzlich existiert ein weiter Struct für den Baum selbst, welcher allerdings nur als Handle für die Funktionen dient. Er speichert die Wurzel und die Anzahl der eingefügten Knoten. Diese wird benötigt, damit man die Worst-Case-Höhe des Baums berechnen kann [2].

```
#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 aus wird lediglich ein Pointer zu ihr ausgegeben. Das Löschen des Baumes wird durch `free_tree` implementiert. Dieses ist auch 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.

```
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 in der Implementierung auf Rekursion verzichtet und immer entweder ein iterativer Ansatz verwendet oder ein selbst implementierter Stack, um Stackoverflows zu vermeiden.

Die Funktion, die die Suche implementiert, akzeptiert eine Pointer zum Baum, den zu suchenden Schlüssel und einen Pointer, in der der Pointer des gefundenen Knotens geschrieben werden kann, falls 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` beginnend bei der Wurzel. Anschließen 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 geschiet, in dem `current` entweder das linke oder rechte Child des aktuellen Knotens zugewisen 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;
}
```

```
12
13 *node = current;
14 return RB_TREE_SUCCESS;
```

3.4 Einfügen von Knoten

3.4.1 Einfügen des neuen Knotens

Eine weiter wichtige Operation ist das Einfügen von Daten in den Baum. Dabei müssen die zwei Eigenschaften der Datenstruktur erhalten bleiben, 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 einem 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).

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

Die Implementierung sorgt erst 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 das 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 den 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 Variable zwischenspeichern.

```
1 struct Node *previous = NULL;
2 struct Node *current  = rbtree->root;
3
4 while (current != NULL) {
5     previous = current;
6     current  = (TLESS(*key, *(previous->key))) ? previous->left : previous->right;
7 }
```

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 Heap-Speicher korrekt initialisiert wird und die Farbe auf rot gesetzt. 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 Wurzel 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 Wurzel 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 Wurzel 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

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 der Fall 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 Rotations 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;
```

```
12
13 child->left = start_node;
14 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.

3.5 Löschen von Knoten

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 1 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 Wurzel des Unterbaums (der ganze Baum ist auch ein Unterbaum von sich selbst). Und `VISIT` stellt dabei eine 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-Stack zu verwenden, wurde ein Software-Stack `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 Stack geholt. Diese Vorgehen simuliert das verhalten einer rekusiven Funktion, jedoch besteht bei sehr großer Rekursionstiefe nicht die Möglichkeit eines Stackoverflow. Die anderen Traversierungsarten wurden auf die selbe 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 Wurzel des Unterbaums als besucht. So kann man Bottom-Up-Algorithmen implementieren, die bei den Blättern beginnen und die Wurzel des Baums als letztes behandeln.

Ein sehr praktischen Bespiel dafür ist das Freigeben der RBTs. Dabei müssen zuerst die Unterbäume aus dem Heap gelöscht werden und danach die Wurzel, weil man sonst die Unterbäume zwischenspeichern müsste, da die Pointer zu ihnen in der Wurzel gespeichert sind. Wenn man diese aber nun als erster 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 Knoten-Struct 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 die sie nicht zu lang werden.) Das liegt, dass die Funktionen über den Ausgabewert Fehler an den Nutzer zurückgeben. Diese Fehler wurden als Macro definiert und durch sie können verschiedene Arten unterschieden werden.

```
#define RB_TREE_SUCCESS              0U
#define RB_TREE_OUT_OF_MEM           1U
#define RB_TREE_KEY_ERROR            2U
#define RB_TREE_NULL_ERROR           3U
#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 Program 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` Macro deaktiviert hat.

4 Implementierungsvarianten

5 Benchmarks

6 Speicher- und Zeitkomplexität

7 Verwendung

Fußnoten

[1]Theoretisch gibt es auch eine Möglichkeit die Baumstruktur als Array von Knoten darzustellen und bessere Cacheladezeiten zu bekommen. Das größte Problem beim 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 zu stande ("The order and contiguity of storage allocated by successive calls to malloc() is unspecified."[IEE]). Das verlangsamt das Programm. Man könnte als Array Implementierung eine spezielle Möglichkeit nutzen, Indizes für das Strukturieren des Arrays in Parent und Child zu nutzen. Die Root würde den Index 0 erhalten. Die linken und rechten Kinder 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 und es gibt eine eindeutige Parent-Child-Beziehung. 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 Array unbesetzt bleiben. Deswegen habe ich mich dagegen entschieden und habe die verkettete Variante der Datenstruktur implementiert.

[2]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 Stacks 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.

Literatur

[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.