Seminararbeit zum Thema Splaybaum

Andreas Windorfer, q8633657

31. März 2020

Inhaltsverzeichnis

1 Einleitung

Der Splaybaum ist eine von Daniel D. Sleator und Robert E. Tarjan vorgestellte Datenstruktur, die sich bei vielen Praxisanwendungen sehr effizient einsetzen lässt. In dieser Seminararbeit wird zunächst der Aufbau des Splaybaumes thematisiert. Dazu wird kurz der binäre Suchbaum vorgestellt, da dieser die Basis des Splaybaumes liefert. Im Anschluss wird der Operationssatz des Splaybaumes vorgestellt. Außerdem wird noch auf das Laufzeitverhalten beim Splaybaum eingegangen, vor allem im Bezug auf Operationsfolgen.

2 Aufbau des Splaybaum

Der Splaybaum ist ein binärer Suchbaum mit einem speziellem Operationssatz.

2.1 Binärer Suchbaum

Bei einem binären Suchbaum kann jeder Knoten sowohl einen linken, als auch einen rechten Kindknoten besitzen, weitere sind nicht möglich. Mit Ausnahme der Wurzel besitzt jeder Knoten genau einen Vaterknoten. Natürlich muss jeder Knoten eine Information enthalten, anhand derer er identifiziert werden kann. Diese Information wird als "Schlüssel" bezeichnet. Für jeden Knoten des Baumes gilt, dass alle sich in seinem linken Teilbaum befindlichen Schlüssel kleiner als der eigene sind, alle Schlüssel des rechten Teilbaumes sind größer. Damit diese Anforderung eingehalten werden kann, muss eine totale Ordnung auf der Menge der Schlüssel gegeben sein. Abbildung 1 zeigt einen Suchbaum, der diese Anforderung einhält und einen der sie verletzt.

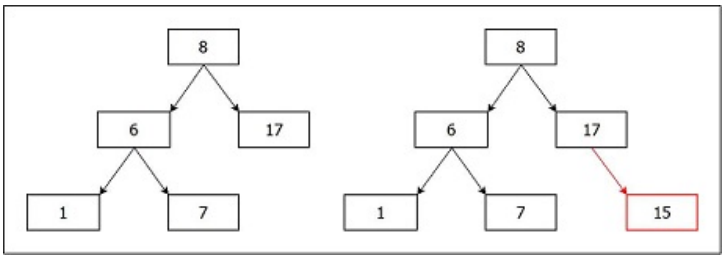

Abbildung 1: Links ist ein gültiger binärer Suchbaum, rechts nicht.

2.1.1 Suchen beim binären Suchbaum

Beim Suchen eines Schlüssels im binären Suchbaum, läuft man den Pfad von der Wurzel bis zum jeweiligen Knoten ab. Ist der gesuchte Schlüssel kleiner

als der des aktuell betrachteten Knotens, wählt man als nächstes den linken Nachfolger aus. Ist der gesuchte Schlüssel größer, wählt man den rechten Nachfolger. Dieses Vorgehen iteriert man solange bis man beim gesuchten Element angelangt ist. Stellt man fest, dass ein benötigter Nachfolger gar nicht vorhanden ist, kann es keinen Knoten mit dem gesuchten Schlüssel im Baum geben. Abbildung 2 stellt einen solchen Suchvorgang dar.

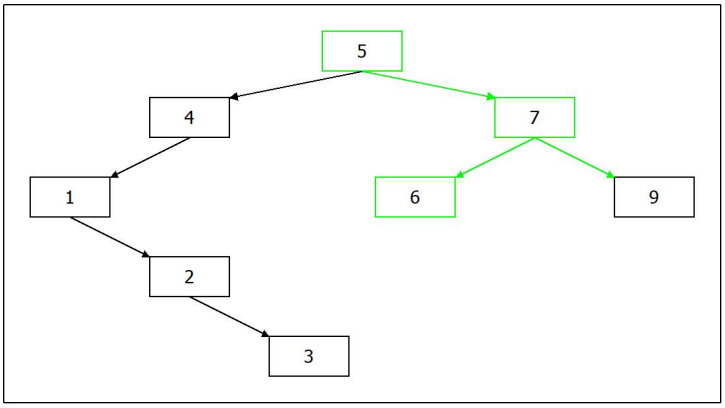

Abbildung 2: Beispielhafter Suchpfad. Hier für den Schlüssel 6.

2.1.2 Einfügen in den binären Suchbaum

Zunächst muss der richtige Einfügestelle gefunden werden. Dazu wird eine Suche nach dem einzufügenden Schlüssel durchgeführt. Falls der Schlüssel noch nicht im Baum vorhanden ist, wird man irgendwann an ein Ende der Datenstruktur gelangen, hier ist dann die passende Einfügestelle.

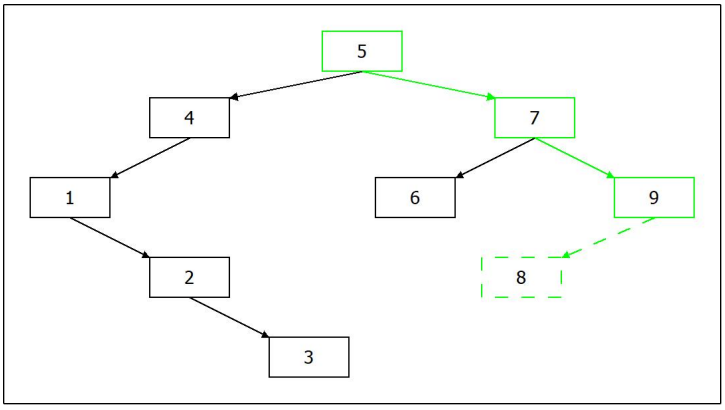

Abbildung 3: Einfügen eines Elementes mit Schlüssel 8.

Die Stelle an der ein Element eingefügt wird, hängt also nicht nur vom eigenen Schlüssel, sondern vor allem auch von den bereits im Baum vorhandenen

Schlüsseln ab. Dies führt dazu, dass Bäume mir der gleichen Schlüsselmenge, in Abhängigkeit von der Einfügereihenfolge, völlig unterschiedliche Strukturen und auch Höhen haben können. Abbildung 4 stellt dies dar. Es ist sofort einleuchtend, dass die Struktur eines Suchbaumes, wesentlichen Einfluss auf die Laufzeit seiner Operationen hat.

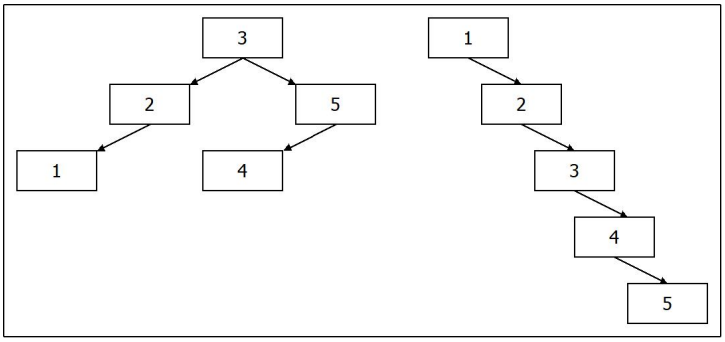

Abbildung 4: Einfügereihenfolge links: 3,2,1,5,4. Rechts: 1,2,3,4,5

2.2 Besonderheit beim Aufbau des Splaybaum

Jede gültige Darstellung eines Splaybaum ist auch eine gültige Darstellung eines binären Suchbaumes, und umgekehrt. Jedoch gehört der Splaybaum zu den selbstorganisierenden Datenstrukturen. Dies bedeutet beim Splaybaum, dass häufig verwendete Elemente weit oben stehen und somit schnell gefunden werden können. Wie das erreicht wird, wird klar wenn man sich mit seinem Operationssatz beschäftigt, was im folgenden Abschnitt passiert.

3 Operationen des Splaybaum

Natürlich kann man bei einem Splaybaum Elemente suchen, einfügen und löschen. Zusätzlich zu diesen Operationen ist es möglich zwei Splaybäume zu vereinigen, oder einen Splaybaum in zwei Bäume aufzuteilen. Alle genannten Operationen verwenden intern eine Hilfsoperation splay, die nun als erstes vorgestellt wird.

3.1 Splay

Ziel der splay(key i, tree s) Operation ist es nicht nur einen ausgewählten Knoten zur neuen Wurzel des Baumes zu machen. Zusätzlich wird die Tiefe der Knoten auf dem Zugriffspfad in etwa halbiert. Siehe dazu auch die Abbildungen 5 und 6 aus [3] von Daniel Dominic Sleator and Robert Endre Tarjan. Dazu wird der Operation splay ein Wert i übergeben. Zunächst wird nach einem Knoten x gesucht, dessen Schlüssel i entspricht. Dieser Suchvorgang

läuft ab, wie bei Abschnitt 2.1.1 erklärt. Ist die Suche erfolgreich, bezeichnet x den Knoten mit Schlüssel i. Ist ein solcher Knoten nicht im Baum enthalten, bezeichnet x den letzten Knoten, auf dem beim Suchen zugegriffen wurde, so dass dann entweder der Knoten mit dem nächst kleineren oder dem nächsten größerem Schlüssel verwendet wird. Abbildung 7 zeigt dies. Nun wird x Schritt für Schritt durch Rotationen nach oben befördert. Dabei werden sechs Fälle unterschieden, wobei jeweils zwei symmetrisch sind. Im folgenden werden nun zunächst die möglichen Rotationen vorgestellt, bevor dann die Arbeitsweise von splay als Ganzes gezeigt wird. Damit zum Ende von splay wieder eine valide Suchbaumstruktur vorhanden ist, ist es wichtig, dass die ursprüngliche Rechts-Links-Struktur erhalten bleibt. Für alle Elemente des Suchbaumes muss also paarweise gelten, das Element, dass sich vor splay weiter rechts befindet, muss dies nach splay immer noch tun. An den folgenden beigefügten Abbildungen, wird man leicht erkennen, dass die Rechts-Links-Beziehung bei jedem Einzelschritt erhalten bleibt.

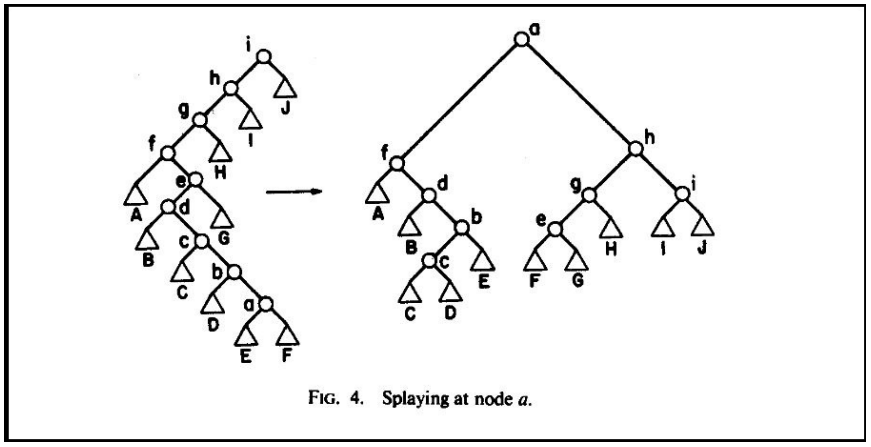

FIG. 4. Splaying at node *a*.

Abbildung 5: Beispiel 1. Die Tiefen der Knoten halbieren sich in etwa. [3]

3.1.1 Zig bzw. zag Rotation

Eine zig Rotation wird durchgeführt, wenn x direkter linker Nachfolger der Wurzel y ist. Dabei wird x zur Wurzel. y wird zum rechten Nachfolger von x. Sollte x bereits einen rechten Nachfolger gehabt haben, wird dieser samt seiner eventuell vorhanden Teilbäume links an y angehängt. Dies ist immer möglich, da an dieser Stelle zuvor x angefügt war. Man spricht bei diesem Vorgang auch davon, dass rechts über y rotiert wird.
Eine zag Rotation ist der symmetrische Fall, wenn x direkter rechter Nachfolger der Wurzel ist. Er ist in Abbildung 9 dargestellt.

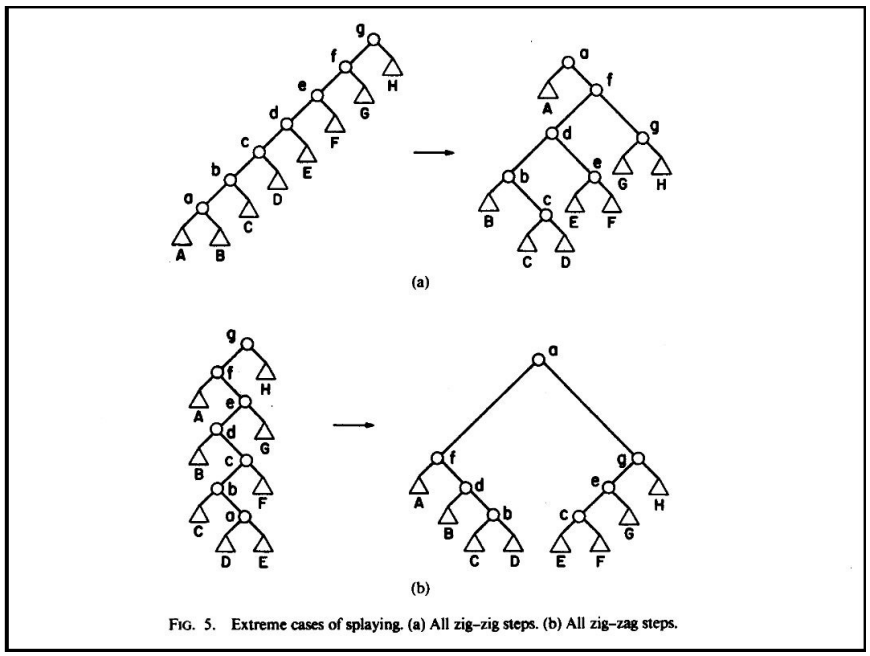

Abbildung 6: Beispiel 2. Die Tiefen der Knoten halbieren sich in etwa. [3].

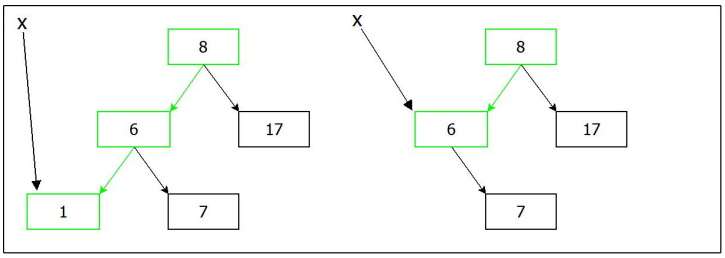

Abbildung 7: Beispielhafte Belegung von x, mit i = 1

3.1.2 Zig-zag bzw. zag-zig Rotation

Die Fälle in den x direkt unter der Wurzel liegt, werden also mit zig bzw. zag behandelt. Liegt x nicht direkt unter der Wurzel, so muss auch der Vater von x einen Vater haben. Dieser wird hier als z bezeichnet. Ist x ein linker Nachfolger und y ein rechter Nachfolger wird zig-zag angewendet. Hierbei wird zunächst eine zig-Rotation angewendet, also rechts über y rotiert. Nun befindet sich x an der Stelle, die zuvor y besetzte. x ist also rechter Nachfolger von z, deshalb wird nun mit zag links über z rotiert.
Eine zag-zig Rotation ist der symmetrische Fall, wenn x rechter Nachfolger eines Knoten ist, der selbst linker Nachfolger ist. Er ist in Abbildung 11 dargestellt.

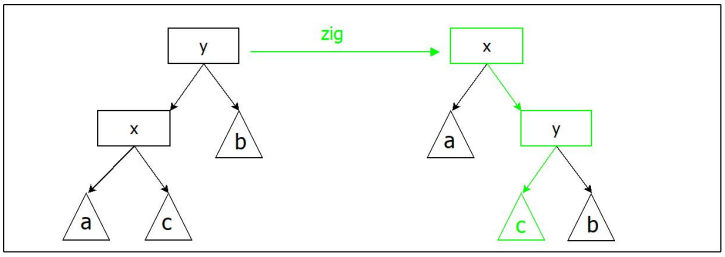

Abbildung 8: Zig Rotation

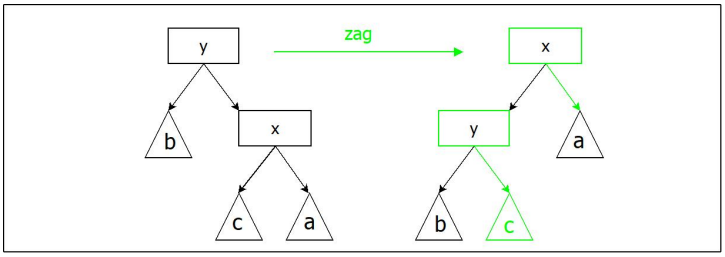

Abbildung 9: Zag Rotation

3.1.3 Zig-zig bzw. zag-zag Rotation

Nun bleiben noch die Fälle über, bei denen sowohl y, als auch x linke bzw. rechte Nachfolger sind. Hier wird dann mit Zig-zig bzw. Zag-zag gearbeitet. Wichtig ist, dass als Erstes die oberen Knoten rotiert werden. Die Abbildungen 12 und 13 demonstrieren dies.

3.1.4 Beispielhafte Algorithmen für splay

In Abbildung 15 wird eine Java Implementierung angegeben, die sich genau an die obigen Beschreibungen hält. Die Algorithmen in Abbildung 17 und 16 entstammen der Arbeit [2] von Daniel Dominic Sleator und Robert Endre Tarjan, in der sie den von ihnen entwickelten Splaybaum vorstellten. Ersterer Algorithmus arbeitet ebenfalls bottom-up. Das heißt, er ordnet den Baum von unten nach oben neu an. Der zweite arbeitet top-down, also von oben nach unten. Die beiden bottom-up Varianten benötigen eine Möglichkeit, auf den Vater eines Knoten zugreifen zu können. Dies kann über einen zusätzlichen Zeiger pro Knoten umgesetzt werden.

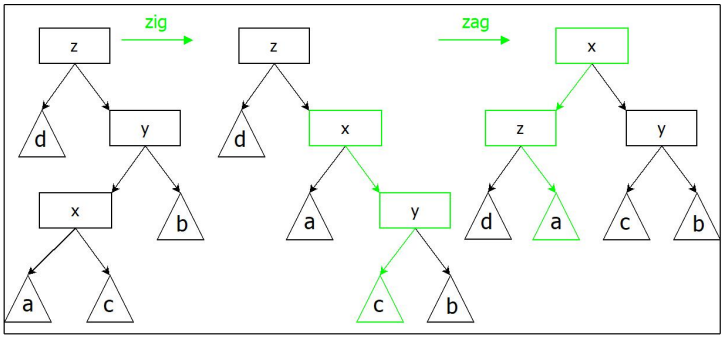

Abbildung 10: Zig-zag Rotation

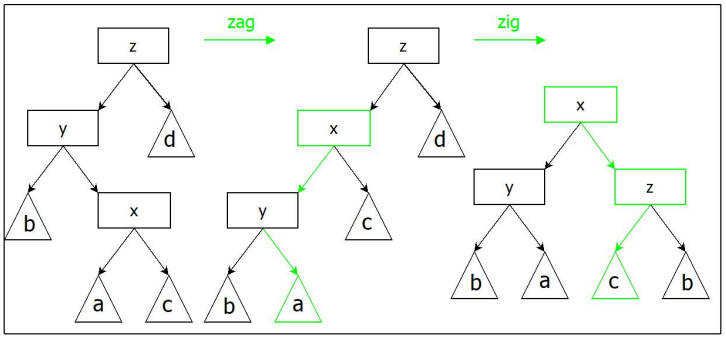

Abbildung 11: Zag-zig Rotation

```
public Node splay(int i, Node t) {
        //suchen
        Node x = t;
        while (x != null && i != x.item) {
            if (i < x.item)
                x = x.left;
            else
                x = x.right;
        }
        //Baum nach oben neu anordnen
        while (x.parent != null){
            Node xp = x.parent;
            if (xp.left == x && xp.parent == null)
                zig(x);
            else if (xp.right == x && xp.parent == null)
                zag(x);
            else if (xp.left == x && xp.parent.right  == xp)
                zigZag(x);
            else if (xp.right == x && xp.parent.left  == xp)
                zagZig(x);
            else if (xp.left == x && xp.parent.left  == xp)
                zigZig(x);
            else if (xp.right == x && xp.parent.right  == xp)
                zagZag(x);
        }
        return x;
    }
    private void zig(Node x){
      Node y = x.parent;
      Node z = null;
      if (y != null)
          z = y.parent;
      y.left = x.right;
      if (x.right != null)
        x.right.parent = y;
      if (z != null && z.left == y)
          z.left = x;
      else if (z != null && z.right == y)
          z.right = x;
      x.parent = z;
      x.right = y;
      y.parent = x;
    }
```

```
 private void zag(Node x){
  Node y = x.parent;
  Node z = null;
  if (y != null)
      z = y.parent;
  y.right = x.left;
  if (x.left != null)
    x.left.parent = y;
  if (z != null && z.left == y)
      z.left = x;
  else if (z != null && z.right == y)
      z.right = x;
  x.parent = z;
  x.left = y;
  y.parent = x;
}
 private void zigZag(Node x){
    zig(x);
    zag(x);
}
 private void zagZig(Node x){
    zag(x);
    zig(x);
}
private void zigZig(Node x){
    zig(x.parent);
    zig(x);
}
private void zagZag(Node x){
    zag(x.parent);
    zag(x);
}
```


Abbildung 15: Eine Java Implementierung von splay.

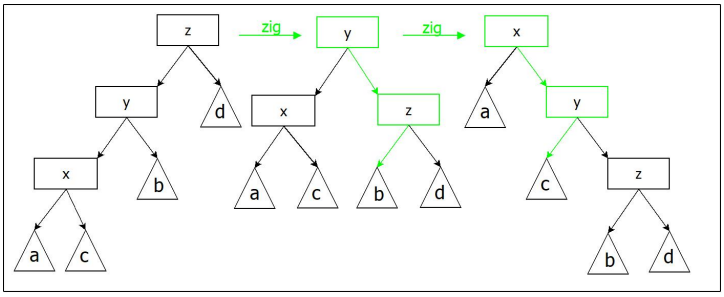

Abbildung 12: Zig-zig Rotation

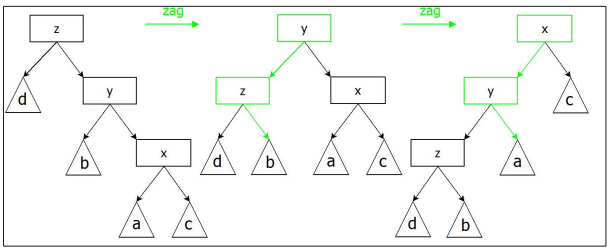

Abbildung 13: Zag-zag Rotation

Um den Algorithmus aus Abbildung 17 nachvollziehen zu können, ist es wichtig zu wissen, dass dieser den Zeiger auf x nicht in jedem Schritt mitschreibt. Ein solcher ist zum Ende der Operation sowieso nicht vorhanden. Die Zeiger von x auf seine Teilbäume werden nur einmal zum Schluss gesetzt. Zum Markieren der Teilbäume während der Operation, werden die Zeiger r und l verwendet. Außerdem wird bei ZigZig bzw. ZagZag auf den Zwischenschritt verzichtet.

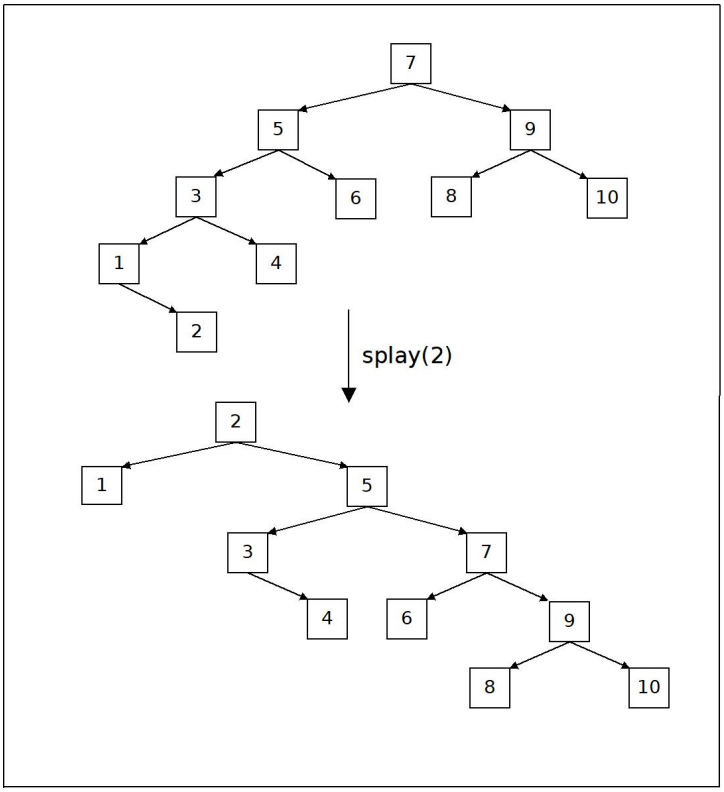

Abbildung 14: Splay Operation. Hier wird splay(2, t) durchgeführt.

```
global tree dummy;
dummy := create tree node;

tree function splay(item i, tree s);
        string state;
        tree l, r, lp, rp;
        if s=null → return null fi;
        left(dummy) := null; right(dummy) := null;
        state := "N";       l := r := dummy;
        do s≠null and i > item (s) →
                if state≠"L" → right(l) := s; lp := l; state := "L"
                 | state="L" → right(l) := left(s); right(lp) := s;
                               left(s) := l; state := "N"
                fi;
                l := s; s := right(s)
         | s≠null and i < item(s) →
                if state≠"R" → left(r) := s; rp := r; state := "R"
                 | state="R" → left(r) := right(s); left(rp) := s;
                               right(s) := r; state := "N"
                fi;
                r := s; s := left(s);
        od;
        if s≠null → right(l) := left(s); left(r) := right(s)
         | s=null →
                if r=dummy → s := l; right(lp) := left(l)
                 | r≠dummy → s := r; left(rp) := right(r) fi
        fi;
        left(s) := right(dummy); right(s) := left(dummy);
        return s
end splay;
```


Abbildung 16: Eine mögliche top-down Pseudoimplementierung von splay.[2]

Das in Abbildung 17 dargestellte Verfahren, merkt sich mit Hilfe eines zusätzlichen Knoten, die beiden späteren Nachfolger des gesuchten Knotens. Die Zeiger r und l merken sich die Stellen an denen zuletzt nach recht bzw. links verzweigt wurde. Diese Stellen werden benötigt, um eventuell vorhandene Teilbäume des gesuchten Knotens, korrekt versetzen zu können. Existiert kein passender Knoten, wird wenn möglich, der nächst Kleinere verwendet, ansonsten der nächst Größere.

```
tree function rotateleft(tree a);
        tree b;
        b := right(a);
        right(a), parent(left(b)) := left(b), a;
        left(b), parent(a) := a, b;
        return b
end rotateleft;

tree function rotateright(tree a); [analogous to rotateleft]

procedure splay(tree x);
        string state;
        tree l, r, y, z;
        state="N";
        l, r, z := left(x), right(x), x;
        y := parent(x);
        do y≠null →
                if right(y)=z →
                        right(y), parent(l) := l, y;
                        z, l, y := y, y, parent(y);
                        if state≠"L" → state := "L"
                         | state="L" → l := rotateleft(l);
                                        state := "N" fi
                 | left(y)=z →
                        left(y), parent(r) := r, y;
                        z, r, y := y, y, parent(y);
                        if state≠"R" → state := "R"
                         | state="R" → r := rotateright(r);
                                        state := "N" fi
                fi
        od;
        left(x), parent(l) := l, x;
        right(x), parent(r) := r, x;
        parent(x) := null
end splay;
```

Abbildung 17: Eine mögliche bottom-up Pseudoimplementierung von splay. [2]

3.2 Suchen

Die Operation suchen(key i, tree t) ist nun sehr einfach umzusetzen. Es wird zunächst splay(i, t) aufgerufen und dann der Wert der neuen Wurzel mit i verglichen. Ist ein Knoten mit Schlüssel i im Baum vorhanden, so muss er nach splay(i, t) die Wurzel sein und eine Referenz auf die Wurzel wird zurückgegeben. Ist der Schlüssel nicht im Baum vorhanden wird null zurückgegeben.

3.3 Aufteilen

Bei aufteilen(key i, tree t) wird zunächst suchen (i, t) ausgeführt. Im Anschluss wird ein Teilbaum von der Wurzel abgetrennt. Ist der Schlüssel der Wurzel größer als i wird der linke Teilbaum abgetrennt, ansonsten der Rechte. Es werden dann Referenzen auf die beiden resultierenden Bäume zurückgegeben.

3.4 Vereinigen

Damit bei vereinigen (tree t1, tree t2) wieder ein Splaybaum entsteht muss gelten, dass der kleinste gespeicherte Schlüssel in t2 größer ist als der größte in t1. Bei der Operation wird zunächst suchen(KeyMax, t1) durchgeführt, wobei KeyMax für den größten möglichen Schlüssel steht. Nach der suchen Operation, ist der Knoten mit dem größten Schlüssel an der Wurzel. Dieser Knoten kann natürlich auch keinen rechten Teilbaum haben, so dass man den Splaybaum t2 nun einfach rechts anfügen kann. Zusätzlich gibt es eine Operation vereinigen(tree t1, key i, tree t2) mit einem dritten Parameter vom Typ key. Für diesen Parameter muss gelten, dass er größer bzw. kleiner als jeder in t1 bzw. t2 vorkommende Schlüssel ist. Dann wird ein Knoten mit Schlüssel i erzeugt und t1 und t2 werden einfach angehängt.

3.5 Einfügen

Bei einfügen (key i, tree t) wird zunächst aufteilen(i, t) ausgeführt. Im Anschluss wird ein neuer Knoten mit Schlüssel i erzeugt, an den die beiden entstandenen Bäume angehängt werden.

3.6 Löschen

Bei löschen (key i, tree t) wird zunächst suchen(i, t) ausgeführt. Wurde ein Knoten mit Schlüssel i gefunden, kann dieser entfernt werden. Im Anschluss muss dann noch vereinigen auf den beiden entstandenen Bäumen ausgeführt werden.

4 Komplexität

Anders als andere Suchbäume macht der Splaybaum keine besondere Zusicherung bezüglich seiner maximalen Höhe. Fügt man eine sortierte Folge von n Schlüsseln in den anfangs leeren Baum ein, so erhält man einen Baum mit Höhe n. Abbildung 18 stellt dies an der Schlüsselfolge 1 bis 5 dar. Da der Knoten mit dem nächst kleineren Wert zuletzt jeweils als Wurzel eingefügt wurde, endet der Suchvorgang zum Beginn der Einfügeoperation bereits an der Wurzel. Es kommt durch den Aufruf von splay(i, t) innerhalb von einfügen(i, t) also zu keiner Veränderung des Baumes. Wie beim gewöhnlichen binären Suchbaum erhält man bei einer solchen Konstellation eine Liste, an der jedoch oben anstatt unten angefügt wird.
Außerdem soll noch die Speicherplatzkomplexität erwähnt werden. Da pro gespeichertem Schlüssel ein Knoten benötigt wird, ist diese $O(n)$.

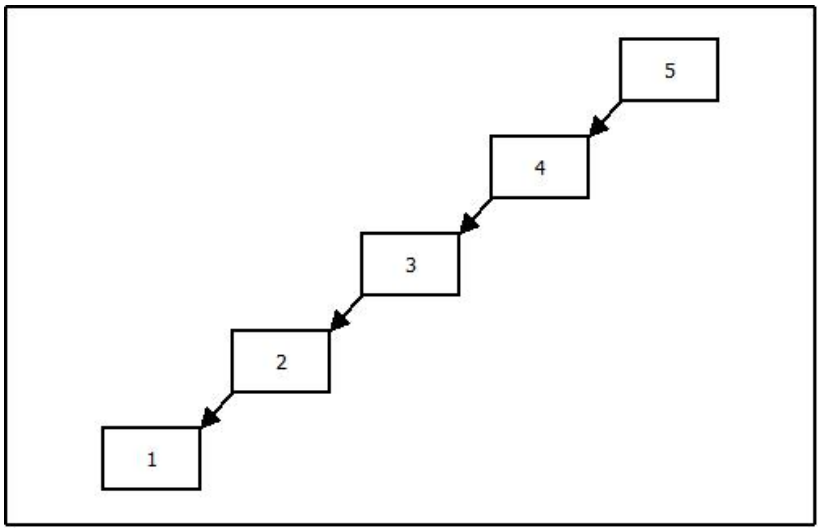

Abbildung 18: Splaybaum in den eine aufwärts sortierte Folge von Schlüsseln eingefügt wurde.

4.1 Einzeloperationen

Nun benötigt die splay Operation pro Baumebene eine Rotation, damit der gewünschte Knoten die Wurzelposition erreicht. Da die Kosten einer Rotation unabhängig von der Position im Baum sind, ergibt sich als Laufzeitkomplexität für splay $O(n)$. Für jede vorgestellte Operation, mit Ausnahme der Vereinigung mit drei Parametern, gilt, dass nach einem internen Aufruf von splay lediglich noch lokal bei der Wurzel gearbeitet wird. Auch diese Operationen haben also Laufzeit $O(n)$. Bei vereinigen(tree s1, key i, tree s2) wird lediglich ein Knoten erzeugt und beschrieben. Die Operation hat also

Laufzeitkomplexität $O(1)$.

4.2 Amortisierte Laufzeitanalyse

Die Stärke des Splaybaum kommt erst zum Vorschein, wenn man eine amortisierte Laufzeitanalyse durchführt. Zunächst soll diese jedoch erstmals vorgestellt werden.

4.2.1 Was ist eine amortisierte Laufzeitanalyse ?

Bei einer amortisierten Laufzeitanalyse geht es darum eine möglichst niedrige, obere Schranke für die durchschnittliche Laufzeit einer Operationsfolge im schlechtesten Fall zu ermitteln. Zum Beispiel hat die splay Operation einzeln betrachtet Laufzeit $O(n)$. Nimmt man nun eine Folge von m Operationen an, ergibt sich $O(mn)$. Mit Hilfe einer amortisierten Analyse ist es jedoch gelungen, eine niedrigere obere Schranke zu finden.

4.2.2 Bankkontomethode

Die Bankkontomethode ist eine von drei Methoden, die bei der amortisierten Analyse verwendet werden. Die Idee dabei ist es, günstige Einzeloperationen die teureren subventionieren zu lassen. Dazu zahlen die günstigen Operationen auf ein gedachtes Bankkonto ein, von dem teurere dann abheben können. Zu jeder i-ten von insgesamt n Einzeloperationen o_i sind also nicht nur die tatsächlichen Kosten c_i zur Durchführung entscheidend, sondern auch noch die amortisierten Kosten a_i, welche die Auswirkung auf das Bankguthaben G beinhalten. Es gilt $a_i = c_i + G_i - G_{i-1}$. Wenn nun nach jeder durchgeführten Operation, dass Guthaben >= 0 ist, wurde die gewählte obere Laufzeitschranke durchgängig eingehalten. Es muss also $\sum_{i=1}^{n}(a_i - c_i) \geq 0$ für alle $n \in$ N gelten. Damit das funktionieren kann, muss es natürlich einen Zusammenhang zwischen der Anzahl bereits durchgeführter günstiger Operationen und den tatsächlichen Kosten der teuren Operationen geben.
Als Beispiel wird häufig ein Stack verwendet, welcher zusätzlich zu push und pop noch popAll anbietet. Bei popAll wird solange pop wiederholt, bis der Stack leer ist. Sei n die Anzahl der bereits durchgeführten Operationen. Die teuerste Operation ist popAll mit $O(n)$, da hier bis zu n mal pop durchgeführt werden muss. Push und pop können in konstanter Zeit durchgeführt werden. Ohne amortisierte Analyse würde man für n Operationen $O(n^2)$ als obere Schranke angeben, da man jeweils den worst-case verwenden muss. Für eine Analyse nach Bankkontomethode legt man nun fest, dass z.B die Kosten für push $c_(push) = x$ und pop $c_(pop) = y$ GE (Geldeinheit) kosten. Außerdem wird festgelegt, dass push y GE auf das Guthaben G einzahlt. Damit hat push konstante amortisierte Kosten $a_(push)$ von $x + y$ GE. Im Gegenzug darf pop y GE vom Konto entnehmen, was amortisierte Kosten

$a_(pop)$ von $y - y = 0$ GE ergibt. Wichtig ist die Beobachtung ($B1$), dass bei einem Stack, der direkt nach der Erzeugung zunächst mal leer ist, genau so viele GE am Konto liegen, wie Elemente am Stack sind. Wenn pop ausgeführt werden kann sind also auch immer ausreichend GE am Konto. Da die Kosten für pop 0 GE sind, ist die teuerste der drei Operationen in diesem Beispiel nun push mit x GE. Eine Folge von n Operationen kostet also maximal nx, mit konstantem x. Damit kann man $O(n)$ als asymptotische Laufzeitschranke verwenden. Da für den Wert des Bankkontos nach jeder Operation ≥ 0 gilt, hat der gedachte Stack nach jeder abgeschlossenen Operation maximal gleich viel Zeit benötigt, wie der reale Stack, so dass auch für diesen $O(n)$ angenommen werden darf. Abbildung 19 veranschaulicht das Beispiel nochmal.

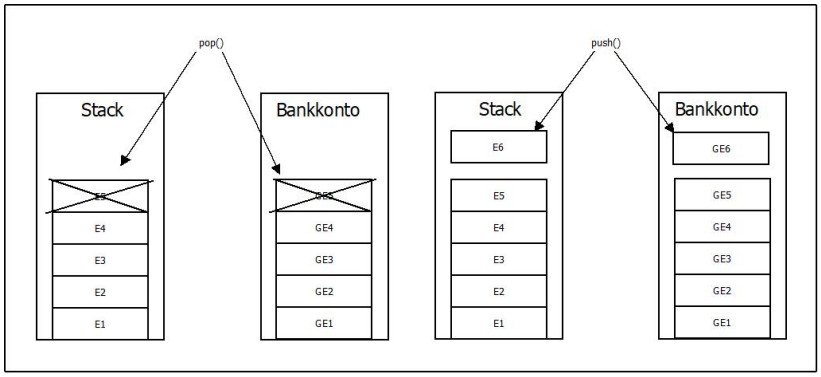

Abbildung 19: Bankkontomethode am Beispiel eines Stacks.

4.2.3 Potentialfunktionmethode

Wenn nicht anders angegeben verwendet dieser Abschnitt die gleichen Bezeichner wie der Abschnitt Bankkontomethode. Die Potentialfunktionmethode ist der Bankkontomethode sehr ähnlich. Bei ihr wird jedoch nicht mit einem gedachten Bankkonto gearbeitet, sondern es wird der aktuelle Zustand der Datenstruktur bewertet. Die Bewertung der Datenstruktur übernimmt die Potentialfunktion Φ. Man kann mit der Potentialfunktionmethode also beliebige Startzustände einer Datenstruktur verarbeiten, während man bei der Bankkontomethode einen Zustand benötigt für den $G_(DS) = 0$ gilt. Eine hohe Bewertung steht dabei für hohe potentielle Kosten. Das heißt Operationen die sich negativ auf den möglicherweise folgenden worst-case-Fall auswirken, verändern die Bewertung ins positive, und umgekehrt. Die amortisierten Kosten einer Operation werden mit $a_i = c_i + \Phi_i - \Phi_{i-1}$ angegeben. Man kann die amortisierten Gesamtkosten mit $(\sum_{i=1}^{n} c_i) + \Phi_n + \Phi_0$ zusammenfassen. Bis auf Φ_n und Φ_0 heben sich die Elemente gegenseitig auf. Die amortisierten Kosten sollen wieder als obere Schranke für die tatsächlichen

Kosten $\sum_{i=1}^{n} c_i$ genutzt werden. Also muss $\forall m \in \{0..n\} : \sum_{i=1}^{m} c_i \leq \sum_{i=1}^{m} a_i$ gelten. Dies erreicht man indem nur Potentialfunktionen verwendet werden, bei der für eine leere Struktur $\Phi_{leer} = 0$ und $\Phi_i \geq 0$ für alle i mit $0 \leq i \leq n$ gelten. Das entscheidende ist nun zu einer Datenstruktur eine geeignete Potentialfunktion Φ zu finden. Das soll wieder am Stack mit drei Operationen gezeigt werden.
Beim Stack bietet sich $\Phi_i = y * \textit{Anzahl der Elemente im Stack}$ nach der Operation i an. Dass die beiden Bedingungen $\Phi_i \geq 0$ für alle i mit $0 \leq i \leq n$ und $\Phi_0 = 0$ erfüllt sind, ist damit schon mal direkt ersichtlich. Als amortisierte Kosten der Operationen ergeben sich $x + \Phi_i - \Phi_{i-1} = x + y$ für push, $y + \Phi_i - \Phi_{i-1} = y - y = 0$ für pop. Für popAll ergibt sich (Anzahl Elemente am Stack) $* 0 = 0$. Am teuersten ist also push. Nun folgt eine Laufzeitabschätzung der amortisierten Kosten bei n Operationen im worstcase. $(\sum_{i=1}^{n} c_i) + \Phi_n = nx + ny = n(x + y)$. Man darf also $O(n)$ als obere Schranke für die amortisierten Kosten angeben und diese sind eine obere Schranke für die tatsächlichen kosten.

4.2.4 Amortisierte Kosten von splay

Zu jedem Schlüssel des Baumes i gehört ein Gewicht $iw(i)$, welches eine reelle Zahl > 1 darstellt. Jeder Knoten x ist Wurzel eines Teilbaumes t_x und hat ein Gesamtgewicht $tw(x) = \sum_{i \in t_x} iw(i)$. Der Wert von $tw(x)$ entspricht also der Summe aller Gewichte im jeweiligen Teilbaum. Der Rang $r(x)$ eines Knoten ist definiert durch $r(x) = log_2(tw(x))$. Sei T der Gesamtbaum, dann ist die Potentialfunktion durch $\phi = \sum_{n \in T} r(n)$ definiert. Damit gilt für eine leere Datenstruktur $\phi_(leer) = 0$. Außerdem gilt für jede andere Konstellation $\phi_(T) \geq 0$. Intuitiv ist sofort ersichtlich, dass man niedriges Potential erreicht, wenn der Baum gut balanciert ist und schwere Knoten weit oben sind. Im folgendem Lemma aus [3] von Daniel Dominic Sleator und Robert Endre Tarjan wird diese Rangeigenschaft verwendet: Wenn Knoten p Vater der Knoten s und t ist, mit $r(s) = r(t)$, dann gilt $r(p) > r(s)$. Das folgt aus $tw(s) \geq 2^{r(s)}$ und $tw(t) \geq 2^{r(t)}$, da $tw(p) \geq tw(s) + tw(t) \geq 2^{r(s)+1}$.

Lemma [3] : *Eine splay Operation mit Knoten x in einem Baum mit Wurzel v hat maximal amortisierte Kosten a von* $3(r(v) - r(x)) + 1$. *Dies wird auch als Zugriffslemma bezeichnet.*
Beweis Mit x wird der Knoten bezeichnet, der an splay übergeben wurde. y ist der Vater von x und z ist der Vater von y. $r()$, $tw()$ und ϕ liefern die Funktionswerte für die Struktur vor der Rotation, $r'()$, $tw'()$ und ϕ' die Werte für danach. Die Kosten für das Ausführen von Rotationen sind konstant und werden im folgenden mit 1 angegeben. Zunächst wird für jeden Rotationstyp gezeigt, dass die Kosten niedriger als $3(r'(x) - r(x)) + 1$ sind.

zig bzw. zag Rotation: Da diese Rotation nur direkt unter der Wurzel ausgeführt wird, ist z nicht definiert. Es gilt also $\phi' - \phi = r'(x) + r'(y) - r(x) - r(y)$. Nach Abschluss der Rotation ist x die Wurzel, zuvor war es y, daraus folgt $r(y) = r'(x)$ und $\phi' - \phi = r'(y) - r(x)$. Durch $r'(x) \geq r'(y)$ ergibt sich $\phi' - \phi \leq r'(x) - r(x) \leq 3(r'(x) - r(x))$. Mit den Kosten für das durchführen der Rotation erhält man $1 + \phi' - \phi \leq 1 + r'(x) - r(x) \leq 3(r'(x) - r(x) + 1)$.
zigZig bzw. zagZag Rotation: Hier ist auch z definiert und es gilt $\phi' - \phi = r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$. Analog zum zig Fall gilt $r(z) = r'(x)$, also $\phi' - \phi = r'(y) + r'(z) - r(x) - r(y)$. Mit $r'(x) \geq r'(y), r'(x) \geq r'(z)$ und $r(x) \leq r(y)$, gilt $\phi' - \phi \leq 2(r'(x) - r(x))$. Hat sich der Rang von x erhöht, kann die Differenz von $3(r'(x) - r(x))$ zu $2(r'(x) - r(x))$ genutzt werden um die Rotation auszuführen. Gilt $r'(x) = r(x)$, muss auch $r'(x) = r(y) = r(z) = r(x)$ gelten, denn $r(x) \leq r(z)$, $r(x) \leq r(y)$, $r'(x) \geq r(z)$ und $r'(x) \geq r(y)$. Dass auch $r'(z) < r(x)$ gilt erkennt man gut am Zwischenschritt der zigZig Rotation, siehe Abbildung 12. Nach diesem Zwischenschritt ist x immer noch Wurzel des gleichen Baumes, wie zu Beginn. z ist bereits die Wurzel des Baumes, die es auch nach Ende der Rotation noch ist. Außerdem ist der Rang von y nach dem Zwischenschritt gleich $r'(x)$. Daraus folgt mit der vor dem Lemma beschriebenen Rangeigenschaft $r'(z) < r(x) = r(z)$. Aus $r'(x) = r(x)$, $r'(y) \leq r(y)$ und $r'(z) < r(z)$ ergibt sich aber $\phi' < \phi$, und mit dieser Differenz kann die Rotation bezahlt werden. Zusätzliche Kosten für das Ausführen der Rotation entstehen also nicht.
zigZag bzw. zagZig Rotation: Aus den gleichen Gründen wie bei zigZig gilt $\phi' - \phi = r'(y) + r'(z) - r(x) - r(y) \leq 2(r'(x) - r(x))$. In Abbildung 10 sieht man das im Teilbaum mit Wurzel y keine neuen Knoten hinzukommen, also gilt $r'(y) \leq r(y)$. Ebenfalls aus der Abbildung 10 kann man $r'(z) \leq r'(x)$ entnehmen, woraus $\phi' - \phi \leq r'(x) - r(x)$ folgt. Für $r'(x) \geq r(x)$ können die Kosten für die Ausführung nun wieder aus der Differenz zu $3(r'(x) - r(x))$ bezahlt werden. Für $r'(x) = r(x) = r(z) = r(y)$ gilt aufgrund der Rangeigenschaft von oben, dass mindestens einer der Knoten z und y im Rang gefallen sein muss. So dass man auch hier einen Kredit zum durchführen der Arbeit über hat. Auch der zigZag Fall benötigt also die zusätzliche KE nicht.
Für aufsummierte zigZig und zigZag Rotationen gilt also $a \leq 3(r(v) - r(x))$. Bei der letzten Rotation, die x zur Wurzel macht kommt dann 1 hinzu, somit ist man bei $a \leq 3(r(v) - r(x)) + 1$. Mit der bekannten Regel $log_x y - log_x z = log_x \frac{y}{z}$, ergibt das $a \leq 3(log_2 \frac{tw(v)}{tw(x)})$

Das Ergebnis dieser Analyse ist auch als Zugriffslemma bekannt:
Zugriffslemma [3]:
Die amortisierte Laufzeit der Splayoperation mit Wurzel t und Knoten x ist maximal $3(r(t) - r(x)) + 1 = O(log((t)/s(x))$

4.2.5 Amortisierte Kosten der anderen Operationen

Für die Analyse der anderen Operationen wird zu Beginn von einem Wald aus leeren Bäumen ausgegangen. Zu keinem Zeitpunkt ist ein Schlüssel in mehr als einem Baum enthalten. U steht für die Menge aller möglichen Schlüssel. T für die Menge aller im Wald enthaltenen Bäume. V ist die Menge der Schlüssel i mit $i \in U$ und $\forall(t \in T)(i \notin t)$. Als Potentialfunktion wird die Summe der Potentiale aller Bäume und der logarithmiertem Gewichte von aktuell nicht enthaltenen Gewichten verwendet. $\phi = \sum_{t \in T} \phi_t + \sum_{i \in V} log(i)$. Zu Beginn hat man also das kleinstmögliche Potential $p_0 = \sum_{i \in U} log(i) > 0$. Für einen Schlüssel i aus dem Baum t sei $i+$ der nächst größere Schlüssel in t, $i-$ der nächst kleinere. Ist $i-$ bzw. $i+$ nicht definiert gilt $w(i-) = \infty$ bzw. $w(i+) = \infty$. Sei außerdem $tw(i)$ die Gesamtgewichtsfunktion vor der Ausführung und $tw'(i)$ die Gesamtgewichtsfunktion nach der Ausführung. Das folgende Lemma aus [3] Daniel Dominic Sleator und Robert Endre Tarjan gibt nun Aufschluss über die genauen Kosten.

Lemma [3] : *Es sei W das Gesamtgewicht der an einer Operation beteiligten Bäume, dann gelten die amortisierten oberen Laufzeitschranken aus Abbildung 20.*

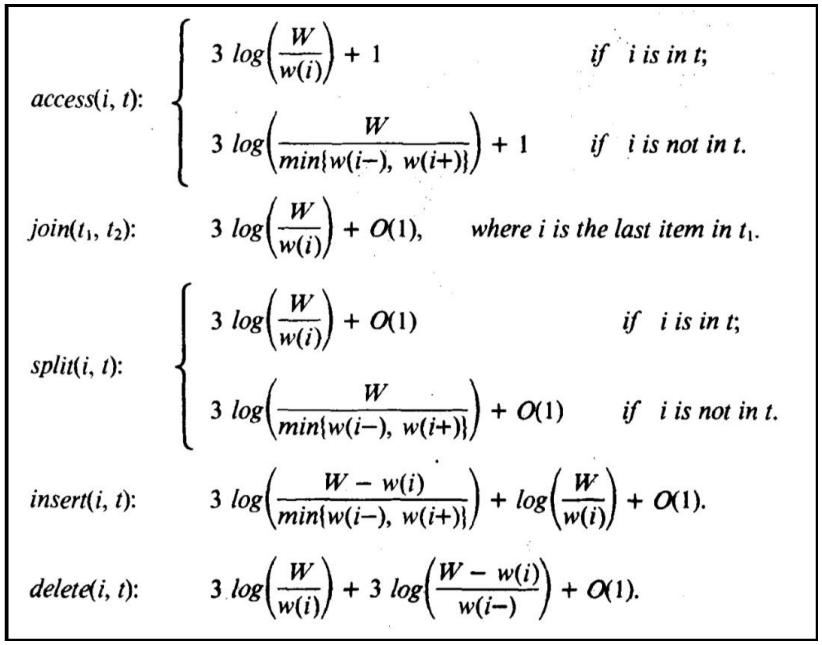

Abbildung 20: Amortisierte Kosten beim Splaybaum [3].

Beweis: Die Angeben für suchen und aufteilen ergeben sich direkt aus dem Zugriffslemma. Für vereinigen ergibt sich als Potentialänderung $log\frac{tw(t1)+tw(t2)}{tw(t1)} \leq$

3 $log(\frac{W}{tw(t1)})$. Außerdem gilt $tw(t1) \geq w(i)$. Bei einfügen hat man zunächst die Kosten von suchen. Da der eingefügte Schlüssel i zuvor mit $log(i)$ in der Gesamtpotentialfunktion enthalten war ergibt sich zusätzlich $log\frac{w(i)+tw(t)}{w(i)} = log(\frac{W}{w(i)})$. Die Schranke von löschen ergibt sich aus denen von suchen und verbinden.

5 Weiterführende Eigenschaften des Splaybaum

Es kommt bei den folgenden Unterpunkten vor, dass Gewichte an die Knoten fest vergeben werden. Dies kann ohne Probleme gemacht werden da das Verhalten des Splaybaum durch die ausgeführten Operationen bestimmt wird. Da die Gewichte in den Operationen nicht verwendet wird, ist das Verhalten von den gewählten Werten unabhängig. So dass man sie frei zur Analyse verwenden kann. Die beiden Sätze in diesem Kapitel stammen aus [3] von Daniel Dominic Sleator und Robert Endre Tarjan.

5.1 Balance Satz [3]:

Beim diesem Satz geht es um die amortisierte Gesamtzugriffszeit auf einen Baum mit n Knoten und Wurzel r. Die Gesamtgewichtsfunktion eines Knoten mit Schlüssel i hat ein Minimum $w(i)$ und ein Maximum von $\sum_{i=1}^{n} w(i)$. Die Wert der Potentialfunktion kann sich deshalb maximal um $\sum_{i=1}^{n} log\frac{tw(r)}{w(i)}$ absenken.
Balance Satz [3]: *Es sei ein Splaybaum mit n Knoten geben auf den m mal zugegriffen wird. Dann gilt für die Gesamtzugriffszeit* $O((m+n)log\ n + m)$

Beweis: Es wird $\frac{1}{n}$ als Gewicht für jeden vorkommenden Schlüssel gewählt. Mit dem Zugriffslemma kann man die Kosten einer Einzeloperation mit $\leq 3\ log(n) + 1$, als Sequenz also $O(\text{m } log\ (n) + m)$, angeben. Damit erreicht man, dass man nun nicht mehr jeden Einzelschritt nachverfolgen muss, um den aktuellen Rang des Zugriffsknotens zu wissen. Addiert man dazu die maximale negative Potentialänderung $O\ (n\ log\ \frac{1}{\frac{1}{n}}) = O\ (n\ log\ n)$, erhält man die Behauptung.

Ein Suchbaum der immer bestmöglich in Balance ist, erreicht $O(m\ log(n))$. Da n konstant ist, ist der Splaybaum amortisiert betrachtet, asymptotisch genau so gut.

5.2 Statische Optimalität

Bei statischen binären Suchbäumen kommt es ohne explizite Updateoperation zu keiner Strukturänderung. Wenn die Zugriffswahrscheinlichkeiten

auf die Elemente eines solchen Suchbaumes bekannt sind, dann lässt sich ein optimaler statischer Suchbaum T_o bezüglich der Kosten für suchen konstruieren. Wenn also die Elemente E_1, E_2...,E_n vorhanden sind und diese Zugriffswahrscheinlichkeiten p_1, p_2...,p_n zugeordnet sind, dann lassen sich die Zugriffskosten für jeden möglichen Baum mit $C_t = \sum_{i=1}^{n} p_i(Ebene(E_i)+1)$ berechnen. Beim statisch optimalen Suchbaum ist dieser Wert dann minimal. Als Beispiel dienen Bäume mit den Schlüsseln 1, 2 und 3 und Wahrscheinlichkeiten von $p(1) = 0.1$, $p(2) = 0.7$ und $p(3) = 0.2$, so dass $p(1)+p(2)+p(3) = 1$ gilt. Es gibt fünf mögliche Suchbäume mit diesen Elementen, siehe Abbildung 21. Der statisch optimale Baum für diese Werte ist der ganz linke mit

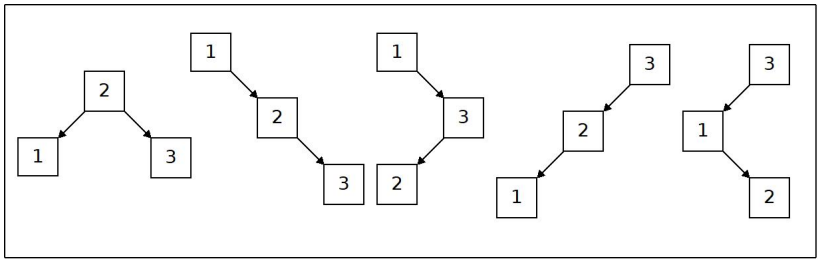

Abbildung 21: Fünf mögliche Suchbäume mit den Schlüsseln 1,2 und 3

$C_t = 1.1$. Der folgende Satz sagt nun aus, dass die Laufzeit eines Splaybaumes asymptotisch mindestens genau so gut ist, als die eines jeden statischen Suchbaumes, also die des statisch optimalen Suchbaumes eingeschlossen.

Satz zur statischen Optimalität [3]:
Es sei $q(i)$ die Anzahl der Zugriffe auf den Schlüssel i. Gilt für jeden Schlüssel i, $q(i) \geq 1$. Dann gilt für die Kosten k des Gesamtzugriffes $k = O(m + \sum_{i=1}^{n} q(i)log(\frac{m}{q(i)}))$.

Beweis: Es wird ein Gewicht $w(i)$ von $q(i)/m$ gewählt. Der Maximalwert für $tw(i)$ ist dann 1. Es wird das Zugriffslemma verwendet um die Kosten für einen Zugriff auf i mit $O(log(\frac{1}{tw(i)}))$ anzugeben. Da $q(i) \geq 1$, gilt $mq(i) \geq m \geq tw(i)$. Damit gilt $\frac{m}{q(i)} \geq \frac{1}{tw(i)}$ und es kann $O(m(log(\frac{m}{q(i)}))$ $= O(m)$ als Schranke für alle m Zugriffe verwendet werden. Das Potential kann sich maximal um $\sum_{i=1}^{n} log(\frac{m}{q(i)})$ vermindern. Aufaddieren ergibt dann die Behauptung.

Die Hauptidee des Beweisen ist es, die Gewichte $iw(x)$ der Elemente des Splaybaumes abhängig von der Zugriffswahrscheinlichkeit zu machen. Damit erreicht die Baumstruktur genau dann niedriges Potential, wenn die oft angefragten Elemente weit oben sind und der Baum gut balanciert ist.

Dass die bewiesene Schranke ausreicht um die asymptotische Schranke des optimalen statischen Suchbaumes einzuhalten, kann in der Arbeit [1] von Norman Abramson nachvollzogen werden.

5.3 Dynamische Optimalität

Dynamische binäre Suchbäume können ihre Struktur auch infolge von Zugriffen auf Elemente des Suchbaumes ändern. Der Splaybaum ist also ein dynamischer Suchbaum.
Sei A ein binärer Suchbaumalgorithmus mit folgenden Eigenschaften:
1. A startet bei Zugriff auf eines seiner Elemente x mit einer Suche ab der Wurzel.
2. A muss den Pfad von der Wurzel zum Element x komplett ablaufen.
3. A kann den Zugriff auf einen Knoten über einen Zeiger in $O(1)$ Zeit ausführen.
4. Rotationen kann A in $O(1)$ Zeit ausführen.
5. Zwischen denn Zugriffen auf einzelne Knoten kann A beliebige Rotationen durchführen.

Als dynamisch optimal gilt ein binärer Suchbaumalgorithmus, wenn er für jede Suchfolge asymptotisch mindestens genau so gut ist, wie jeder Suchalgorithmus A.
In [3] von Daniel Dominic Sleator und Robert Endre Tarjan wird vermutet, dass Splaybäume dynamisch optimal sind.

Literatur

[1] Norman Abramson. *Information theory and coding.* McGraw-Hill electronic sciences series. McGraw-Hill, New York, NY, 1963.

[2] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary trees. pages 235–245, 1983.

[3] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.