Aufgabe 4:

Selbstanordnende Datenstrukturen

Team: 04 (Sean Pedersen, Uwe Krause)

Aufgabenaufteilung:

- Konzeption / Entwurf:
    - Listen: S. Pedersen
    - Bäume: U. Krause (generelles) /
        - Konzipierung der Bottom-Up Strategie mit 2 Stacks: Zusammen
- Implementation: (noch offen)

Quellenangaben:

- Vorlesungsfolien
- Splay Tree Visualization https://www.cs.usfca.edu/~galles/visualization/SplayTree.html
- D. D. Sleator und R. E. Tarjan, Self-Adjusting Binary Search Trees, AT&T Bell Laboratories, Murray Hill, NJ
- T. Ottmann, P. Widmayer, Algorithmen und Datenstrukturen, 3. Aufl., Spektrum

Bearbeitungszeitraum:

- jeweils 5 Stunden p.P. nach Vorstellung in der Vorlesung in eigenes Thema einarbeiten und mit vorhergehenden Aufgabenstellungen vergleichen, ggf. Parallelen ziehen.
- Baum: 5 Stunden einlesen in Paper von Sleator / Tarjan, extrahieren der Top-Down Strategie, dann feststellen, dass die Strategie das Problem umschifft und dass es vielleicht sinnvoller wäre, bei dieser Gelegenheit einen allgemeinen Ansatz zu verfolgen, auf den man eventuell später bei einem ähnlichen rekursiven Problem zurückgreifen kann.
- 3 Stunden p.P. gemeinsames Formulieren und konzipieren der Bottom-Up Splay Strategie mit 2 Stacks
- 5 Stunden p.P. gemeinsam für weitere Formulierungen des Entwurfs für Listen, den verbleibenden Baumoperationen und der Testumgebung.

Aktueller Stand:

- Konzept / Entwurf: Abgeschlossen.
- Implementation: (noch offen)

1. Definitionen

1.1. Baum (rekursive Definition)

(Unverändert aus Aufgabe 3 übernommen.)

Ein Baum ist ein rekursiv definierter abstrakter Datentyp. Hierbei gibt es genau einen Anfang (Wurzel) und eine endliche Anzahl Nachfolger (Teilbäume), welche wiederum Bäume sind.

Ein Sonderfall ist der leere Baum, der keinen Wert und keine Nachfolger hat.
Für ihn wird die Höhe 0 definiert.

Jeder nicht-leere Baum besteht aus einem Wert, einer Höhe im Baum, die sich über die Höhe der nachfolgenden Teilbäume definiert und Nachfolgern, welche wiederum Teilbäume sind.

- Ein Baum, der ausschließlich leere Bäume als Nachfolger hat wird “Blatt” genannt.
- Die Höhe entspricht der Maximalhöhe der nachfolgenden Teilbäume plus 1.

Alles andere ist kein Baum.

1.1.1. Unärbaum (Liste)

Eine Liste zeichnet sich dadurch aus, dass jede (Teil-)Liste exakt einen Nachfolger hat, welcher wiederum eine Liste ist. Die Höhe einer Liste ist somit äquivalent zu ihrer Größe, also der Anzahl von Elementen.

1.1.2. Binärbaum

(Unverändert aus Aufgabe 3 übernommen.)

Ein Binärbaum zeichnet sich zusätzlich dadurch aus, dass jeder (Teil-)Baum exakt 2 Nachfolger hat, welches wiederum Bäume sind[1].

1.1.3. Binärer Suchbaum

(Unverändert aus Aufgabe 3 übernommen.)

Ein binärer Suchbaum legt zusätzlich Regeln fest, wie die Nachfolger angeordnet sein müssen.

Die Teilbäume werden auf Grundlage eines Vergleichs ihrer Werte in der Struktur angeordnet. Teilbäume mit niedrigeren Wert werden als linker Nachfolger festgelegt und Teilbäume mit einem höheren oder gleichen Wert werden als rechter Nachfolger festgelegt.

[1] Von denen aber einer oder beide leere Bäume sein können.

Für alle Werte muss daher eine natürliche Ordnung bestehen oder definiert werden.

1.1.4. Splay-tree

In einem Splay-tree mit n Knoten haben die üblichen Suchbaumoperationen eine “amortisierte” Laufzeit von O(log n) pro Operation.
Mit “amortisiert” ist die durchschnittliche Laufzeit pro Operation im Vergleich zu einer Worst-Case Sequenz von Operationen gemeint.

Erreicht wird dies, indem die Struktur innerhalb des Baumes bei jedem Zugriff neu geordnet wird. Elemente auf die zugegriffen wird, werden entlang des Suchpfads näher an die Wurzel rotiert, um bei einer erneuten Suche schnellere absolute Zugriffszeiten zu erzielen.

2. Such- & Optimierungsstrategien

Unabhängig von der konkreten Datenstruktur gibt es verschiedene Such- und Optimierungsstrategien.

Die jeweilige Datenstruktur wird gemäß ihren definierten Regeln durchlaufen, solange bis der gewünschte Wert gefunden wird.

- Für die Liste wird als Resultat die Position des gesuchten Elementes in der (ggf. manipulierten) Struktur zurückgegeben.
- Für den Baum wird als Resultat die Höhe des gesuchten Elementes in der (ggf. manipulierten) Struktur zurückgegeben.

Bei optimierenden Strategien wird die Datenstruktur, in der gesucht wurde, gemäß der Strategie manipuliert und zusätzlich zum Suchresultat wird die manipulierte Datenstruktur zurückgegeben.

Es wird immer davon ausgegangen, dass der zu suchende Wert in der Datenstruktur vorhanden ist, fehlerhafte Anfragen werden (wie vorgegeben) nicht abgefangen.

2.1. Normale Suche

Die Datenstruktur in der gesucht wurde bleibt unverändert, die Position innerhalb der Struktur wird ermittelt.

2.2. Move-to-front (root)

Das gefundene Element wird an die erste Stelle der Datenstruktur verlegt.

Bei der Liste wird das Element von seiner aktuellen Position entfernt und an den Anfang der Struktur positioniert, indem es als Nachfolger die ursprünglichen Liste (ohne das entfernte Element) erhält. Die Position innerhalb der neuen Liste ist also immer 1.

Bei den Splaybäumen wird das gefundene Element mittels Rotationsoperationen solange “nach oben” rotiert, bis es an der Wurzelposition angelangt ist. Bei den Rotationsoperationen

werden die Höhen angepasst[2]. Die Höhe des Knotens mit dem gesuchten Wert entspricht nach den Rotationen der Gesamthöhe des Baums.

2.3. Transpose

Das gefundene Element wird um eine Position nach vorne (oder “oben”) verschoben.

Bei der Liste wird das Element mit seinem Vorgänger getauscht. Die Position innerhalb der neuen Liste entspricht also der alten Position minus Eins.

Bei dem Baum wird das gefundene Element mit einer einzelnen Rotationsoperation um eine Ebene nach oben verschoben. Bei den Rotationsoperationen werden die Höhen angepasst. Die Höhe des Knotens mit dem gesuchten Wert ergibt sich aus der Rotationsoperation.

3. Teilprobleme

3.1. Durchlaufen des Baums

Da die Auswahl der richtigen Rotationsrichtung der Splayrotationen immer in Abhängigkeit der Vorfahren des aktuellen Knotens durchgeführt werden, innerhalb der vorgegebenen Datenstruktur aber immer nur das aktuelle Element und seine Nachfolger sichtbar sind, ist es notwendig[3] den bereits gegangenen Pfad festzuhalten, damit auf dem “Rückweg” die richtigen Rotationen durchgeführt werden können.

Während der Baum auf der rekursiven Suche nach dem gesuchten Knoten (Knoten mit dem Wert, der gesucht oder gelöscht werden soll, oder der nächste leere Knoten, an dem der Wert angehangen werden soll) durchlaufen wird, wird der Pfad festgehalten, indem die Abzweigungsrichtungen (links, rechts) auf einem Stack gespeichert werden.
(Oder in einer beliebigen anderen Datenstruktur, die einen Lifo-Zugriff erlaubt[4].)

- Ist der Wert des aktuellen Elementes kleiner (bzw. größer) als das, auf das zugegriffen werden soll, so wird der Baum rekursiv nach links (bzw. rechts) durchlaufen.
  Der Datenstruktur, die den bisher gegangenen Pfad repräsentiert wird die Information hinzugefügt, in welche Richtung gegangen wurde (R/L).

---

[2] Die Höhen werden in der Datenstruktur gespeichert aufgrund der Vorgabe, dass die Baumstruktur aus den vorhergehenden Aufgaben zu verwenden ist.
Einer der besonderen Vorteile von Splaybäumen im Vergleich zu beispielsweise AVL-Bäumen ist, dass sie für die Reorganisation keine Höhen- oder Balance-informationen benötigen.

[3] Es gibt noch andere Möglichkeiten, beispielsweise ein Top-Down-Verfahren, welches den Baum beim Durchlaufen in Teilbäume mit kleineren und größeren Knoten als das gesuchte Element zerlegt und anschließend wieder neu zusammensetzt.

[4] In Erlang beispielsweise bietet sich die einfach verkettete Liste an, da das Hinzufügen und Entfernen am Anfang leicht ist und Fallunterscheidungen bequem mittels Patternmatching und Guards realisiert werden können. Eine Alternative für Programmiersprachen ohne diese Werkzeuge könnte ein String sein, der später mit Substring oder ähnlichem zerlegt wird.

- Ist der Wert des aktuellen Elements gleich dem, auf das zugegriffen werden soll, wird das Element entsprechend der Optimierungsstrategie mittels splaying ggf. verschoben.

3.2. Optimierungsstrategien

3.2.1. Keine

Es findet keine Reorganisation durch Rotation statt, der Baum bleibt unverändert.

3.2.2. Splay-to-root

Bei dem Zugriff auf den gesuchten Knoten endet der rekursive Abstieg.

Der Rückwegstack[5] wird vorbereitet, indem das oberste Element des Pfadstacks auf den Rückwegstack verschoben wird.

3.2.3. Transpose

Sobald das Element, auf das zugegriffen werden soll, gefunden wurde, wird eine (1) Rotation durchgeführt.

Die Richtung wird durch das letzte Element des Rückwegstacks vorgegeben.

3.3. Spreizen (splay)

Der Kern der Splay-trees ist die Splay-operation, welche Elemente, auf die zugegriffen wird mithilfe von Rotationen näher zur Wurzel bewegt.

In Ergänzung zu der Variante, das gewünschte Element Stück für Stück um eine Ebene näher zur Wurzel zu bewegen (“zig”-Rotation), wurden die “zig-zig” und die “zig-zag” Operationen definiert, die zwei zig-Rotationen (in Abhängigkeit der erforderlichen Richtungen) kombinieren.

Die Regeln können der Aufgabenstellung entnommen werden und werden der Vollständigkeit halber hier zitiert:

“Sei t ein binärer Suchbaum und x ein Schlüssel. Dann ist das Ergebnis der Operation Splay(t, x) der binäre Suchbaum, den man wie folgt erhält.

- Schritt 1: Suche nach x in t. Sei p der Knoten, bei dem die (erfolgreiche) Suche endet, falls x in t vorkommt, und sei p der Vater des Blattes, bei dem eine erfolglose Suche nach x in t endet, sonst.
- Schritt 2: Wiederhole die folgenden Operationen zig, zig-zig und zig-zag beginnend bei p solange, bis sie nicht mehr ausführbar sind, weil p Wurzel geworden ist.

[5] Abschnitt über das rekursive Splaying beschrieben.

- Fall 1: [p hat Vater φp und φp ist die Wurzel]
  Dann führe die Operation „zig“ aus, d.h. eine Rotation nach links oder rechts, die p zur Wurzel macht.
- Fall 2: [p hat Vater φp und Großvater φφp und p und φp sind beides rechte oder beides linke Söhne]
  Dann führe die Operation „zig-zig“ aus, d.h. zwei aufeinanderfolgende Rotationen in dieselbe Richtung, die p zwei Niveaus hinaufbewegen.
- Fall 3: [p hat Vater φp und Großvater φφp und einer der beiden Knoten p und φp ist linker und der andere rechter Sohn seines jeweiligen Vaters]
  Dann führe die Operation „zig-zag“ aus, d.h. zwei Rotationen in entgegengesetzte Richtungen, die p zwei Niveaus hinaufbewegen.”

Im Gegensatz zu einer reinen move-to-root-Strategie werden die Rotationen nicht immer strikt eine Ebene von unten nach oben durchgeführt.

3.3.1. Rekursives spreizen

Um für die zig-zig oder zig-zag Rotationen eine Ebene überspringen zu können, wird ein “Rückwegstack” eingeführt.

Bei jedem Schritt des Rückwegs werden Rückwegstack und Pfadstack untersucht:

- Pfadstack leer?
  - Rückwegstack leer?
    - Wurzel erreicht, keine Rotation,
      ENDE
  - Rückwegstack 1 oder 2 Element?
    - Rotation
    - Rückwegstack leeren
- Pfadstack hat mindestens 1 Element?
  - Rückweg leer oder 1 Element?
    - oberstes Element vom Pfadstack auf Rückwegstack verschieben
    - keine Rotation
  - Rückweg 2 Elemente?
    - Rotation
    - Rückwegstack leeren

Rückgabewert der Splayoperationen ist

- der ggf. veränderte Baum
- und der ggf. modifizierte Pfadstack
- und der ggf. modifizierte Rückwegstack

3.4. Rotation

Den Rotationsoperationen wird der aktuell betrachtete (Teil-) Baum und der Rückwegstack übergeben.

Auf Grundlage des Rückwegstacks wird entschieden, welche Rotationsoperation angewandt wird und in welche Richtung:

- 1 Elemente auf Stack?
  - L: zig-rechts
  - R: zig-links
- 2 Elemente auf Stack?
  - LL: zig-zig rechts
  - LR: zig-zag links
  - RL: zig-zag rechts
  - RR: zig-zig links

3.4.1. Rotationsoperationen

(Unverändert aus Aufgabe 3 übernommen.)

Teilprobleme der Rotation:

- Die Eingabe muss in relevante Teile zerlegt werden.
- Die Rotation erfolgt durch Neuzusammensetzung gemäß vorgegebener Rotationsregeln.

Zerlegung:

- Der Ursprungsbaum (A) und seine Teilbäume (L, R) werden jeweils in Wert, linker Nachfolger und rechter Nachfolger zerlegt.
  Es entstehen 3 isolierte Werte (A.w, L.w, R.w)
  und 6 Unter-Teilbäume (L, R, LL, LR, RL, RR)

Ein gemäß dieser Zerlegung aufgeteilter Ursprungsbaum (A) besteht aus:

A = (A.w, L = (L.w, LL, LR, L.h), R = (R.w, RR, RL, R.h), A.h)

3.4.1.1. Zig links

(Zig-links entspricht der Linksrotation des AVL-Baums und wurde daher leicht gekürzt aus Aufgabe 3 übernommen.)

Der Eingabebaum wird in seine für diese Operation relevanten Einzelteile zerlegt:

(A.w, L, (R.w, RL, RR, R.h), A.h)

Anschließend wird die [...] Ausgabe durch Zusammensetzen gebildet:

(R.w, (A.w, L, RL, neue_höhe[6]), RR, neue_höhe)

[6] Die Höhe berechnet sich aus dem Maximum aus der Höhe des linken Teilbaums und des rechten Teilbaums + 1

3.4.1.2. Zig rechts

(Zig-rechts entspricht der Rechtsrotation des AVL-Baums und wurde daher leicht gekürzt aus Aufgabe 3 übernommen.)

Der Eingabebaum wird in seine für diese Operation relevanten Einzelteile zerlegt:

```
(A.w, (L.w, LL, LR, R.h), R, A.h)
```

Anschließend wird die [...] Ausgabe durch Zusammensetzen gebildet:

```
(L.w, LL, (A.w, LR, R, neue_höhe), neue_höhe)
```

3.4.1.3. zig-zig

Die zig-zig Operationen entspricht einer Ausführung einer zig Rotation auf den aktuellen Knoten, gefolgt von einer weiteren Ausführung der gleichen zig-Rotation auf das Ergebnis der vorherigen Rotation.

3.4.1.4. zig-zag

Die zig-zag Operationen entspricht einer Ausführung einer zig Rotation auf den aktuellen Knoten, gefolgt von einer weiteren Ausführung einer zig-Rotation in Gegenrichtung auf das Ergebnis der vorherigen Rotation.

4. Operationen

4.1. Listenoperationen

4.1.1. create: ∅ → list

Gibt als Rückgabewert eine leere Erlang-Liste zurück. Die stets als Ende erhalten bleiben wird, da die leere Liste als Tail, das Ende der Liste signalisiert.

4.1.2. isEmpty: list → bool

Prüft ob die übergebene Liste äquivalent zum Rückgabewert von create/0, also der leeren Erlang-Liste ist.

4.1.3. isList: list → bool

Prüft ob der übergebene Parameter eine Erlang-Liste ist.

4.1.4. equal: list × list → bool

Prüft die beiden übergebenen Listen auf strukturelle Gleichheit. Zwei Listen sind gleich, wenn jedes Element den gleichen Inhalt und den gleichen Nachfolger hat.

4.1.5. laenge: list → int

Gibt die Länge der übergebenen Liste zurück, indem die Liste rekursiv durchlaufen wird und ein Zähler pro Rekursionsschritt um eins hochgezählt wird. Die leere Liste hat eine Länge von 0.

4.1.6. insert: list × elem → list

Fügt das übergebene Element an den Anfang der übergebenen Liste an und gibt anschließend die Liste mit dem eingefügten Element zurück.

4.1.7. delete: list × elem → list

Die übergebene Liste wird rekursiv durchlaufen bis das übergebene Element gefunden wird (aktuelles Element gleich übergebenen Element). Der Nachfolger des zu löschenden Elementes wird zum Nachfolger des Vorgängers des zu löschenden Elementes. Die Liste ohne das zu löschende Element wird zurückgegeben.

4.1.8. finds: list × elem → pos

Die übergebene Liste wird rekursiv durchlaufen, dabei wird ein Zähler in jedem Rekursionsschritt um eins hochgezählt, bis das aktuelle Element gleich dem übergebenen Element ist. Der Zähler wird zurückgegeben.

4.1.9. Find-Move-To-Front (findmf: list × elem → {pos,list})

Das zu entfernende Element wird mit delete/2 gelöscht.

Es wird ein Tupel zurückgegeben mit der Position 1 und dem übergebenen Element als Liste mit der von delete/2 zurückgegebenen Liste.

4.1.10. Find-Transpose (findtp: list × elem → {pos,list})

Die übergebene Liste wird rekursiv durchlaufen mit einem Look-Ahead von 1 (aktuelles & **nächstes** Element), dabei wird ein Zähler in jedem Rekursionsschritt um eins hochgezählt, bis das nächste Element gleich dem übergebenen Element ist. Nun wird das aktuelle Element mit dem nächsten Element vertauscht, also um eins nach vorne geschoben.

Es wird ein Tupel zurückgegeben mit dem Zähler (Position) und der vertauschten Liste.

4.1.11. retrieve: list × pos → elem

Die übergebene Liste wird rekursiv durchlaufen, dabei wird ein Zähler in jedem Rekursionsschritt um eins hochgezählt, bis der Zähler gleich der übergebenen Position ist. Nun wird das aktuelle Element zurückgegeben.

4.2. Baumoperationen

4.2.1. Erstellen (initBT: ∅ → btree)

(Leicht verändert aus Aufgabe 3 übernommen.)

Beim Erstellen eines Splay-Baums ist dieser initial leer, es wird also ein Baum erstellt, der der Definition eines leeren binären Baumes entspricht.

Dargestellt wird dies durch das Symbol der leeren Liste [ ]

4.2.2. Prüfen auf Leere (isEmptyBT: btree → bool)

(Unverändert aus Aufgabe 3 übernommen.)

Der Baum wird geprüft, ob der der Definition eines leeren Baumes entspricht.

- Ein Baum ist leer, wenn er der festgelegten Datenstruktur aus initBT entspricht.
- Alles andere ist kein leerer Baum.

4.2.3. Gleichheit (equalBT: btree × btree → bool)

(Unverändert aus Aufgabe 3 übernommen.)

Prüft auf strukturelle, nicht auf semantische Gleichheit.

“Es kann sein, dass auf dem gleichen Ursprungsbaum gleiche Operationen mit gleichen Argumenten (z.B. insert/delete) in einer anderen Reihenfolge zu inhaltsgleichen, aber strukturell ungleichen Bäumen führen.

Zwei Bäume sind strukturell gleich, wenn beginnend von den Wurzeln, jedes Knotenpaar den gleichen Wert, die gleiche Höhe und ebenfalls nach diesen Regeln gleiche Nachfolgeknoten hat.

Hierfür wird beginnend von den Wurzeln beider Bäume jedes Knotenpaar auf Gleichheit der Werte, der Höhenwerte und der beiden Nachfolger (rekursiv) miteinander verglichen.

Wird ein leerer Baum mit einem nichtleeren Baum verglichen, so sind diese ungleich.

Sobald eine Ungleichheit gefunden wurde, müssen die nachfolgenden Teilbäume nicht weiter betrachtet werden.”

4.2.4. Gültige Struktur (isBT: btree → bool)

(Leicht verändert aus Aufgabe 3 übernommen.)

Prüft einen übergebenen Baum darauf, ob die Definitionen erfüllt sind, die zum binären [...] Suchbaum führen.

Für jeden Knoten wird überprüft, ob ...

Baum
- er ein leerer Baum ist,
- oder ein gültiger Knoten mit mindestens einem Nachfolger.

Binärbaum
- er genau 2 Nachfolger hat.

Binärer Suchbaum
- der Wert eine ganze Zahl ist[7].
- der Wert des linken Nachfolge-Teilbaumes kleiner ist als der Wert des aktuellen Knotens.
- der Wert des linken Nachfolge-Teilbaumes größer[8] ist als der Wert des aktuellen Knotens.
- die Höhe des aktuellen Knotens der maximalen Höhe der beiden Nachfolgebäume plus 1 entspricht.

Splay Baum
- Der Splay Baum hat kein Merkmal innerhalb der Datenstruktur, das über die Vorgaben des binären Suchbaums hinausgeht.

4.2.5. Einfügen (insertBT: btree × elem → btree)

(Gekürzt aus Aufgabe 3 übernommen.)

Um in einen Baum einen Wert einzufügen werden der Baum und der einzufügende Wert benötigt.

Der Rückgabewert ist der ggf.[9] modifizierte Baum.

Beginnend von der Wurzel des Baumes aus, wird unter Berücksichtigung des einzufügenden Wertes (kleiner als aktueller Wert: links weiter, größer-gleich: rechts weiter), der Baum solange rekursiv durchlaufen bis ein leerer Baum gefunden wird, dieser wird nun durch den einzufügenden Wert ersetzt.

4.2.6. Löschen (deleteBT: btree × elem → btree)

Gelöscht wird wie in Aufgabe 3, Die Rotationen gemäß AVL-Regeln entfallen.

Um aus einem Baum einen Wert zu löschen, werden der Baum und der zu löschende Wert benötigt.

Der Rückgabewert ist der ggf. modifizierte Baum.

---

[7] Ein Vergleich mit einem Wert, der keine ganze Zahl ist, kann zwar erfolgreich sein, diese Regelverletzung wird aber in der Überprüfung des nächsten Knotens festgestellt.
Dieses Vorgehen vermeidet die doppelte Prüfung jedes Wertes.
[8] Da keine Duplikate erlaubt sind, nicht “größer oder gleich”.
[9] Wenn gültiges Argument, also kein Duplikat, eingefügt werden soll.

Analog zum Einfügen wird das zu löschende Element rekursiv gesucht, indem der Baum abhängig vom Wert des aktuellen Knoten rekursiv jeweils links oder rechts durchlaufen wird.

Hierbei wird für jeden Knoten geprüft, ob der Wert des aktuellen Knotens gleich dem zu entfernenden Wert ist.

- Ist der aktuelle Knoten leer, ist der zu entfernende Wert nicht im Baum gewesen.
  Der Baum wird unverändert zurückgegeben.
  Die Höhe hat sich nicht verändert.
- Wenn der zu löschende Wert kleiner als der Wert des aktuellen Knotens ist, wird der Wert rekursiv aus dem linken Teilbaum entfernt.
- Wenn der zu löschende Wert größer als der der Wert des aktuellen Knotens ist, wird der Wert rekursiv aus dem rechten Teilbaum entfernt
- Wenn der Wert des aktuellen Knotens gleich dem zu löschenden Wert ist, wird unterschieden:
  - ist der linke Teilbaum leer, wird der zu löschende Wert entnommen, indem der rechte Teilbaum an seine Stelle tritt.
    Die Höhe des Baumes hat sich hierdurch verändert.
  - ist der rechte Teilbaum leer, wird der zu löschende Wert entnommen, indem der linke Teilbaum an seine Stelle tritt.
    Die Höhe des Baumes hat sich hierdurch verändert.
  - Ansonsten muss ein geeigneter Ersatzwert gefunden werden.[10]
    - Die Suche nach dem Ersatzwert liefert den Ersatzwert, den um diesen Wert reduzierten Baum sowie die information, ob sich die Höhe durch die Entnahme oder Rotationen geändert hat.
    - Der so ermittelte Ersatzwert tritt anstelle des zu löschenden Werts. Der linke Nachfolger bleibt unverändert und der rechte Nachfolger ist der Nachfolger, aus dem der Ersatzwert entfernt wurde.
    - Wenn sich die Höhe des Unterbaums verändert hat, muss ggf. die Höhe des Baums angepasst werden.

4.2.6.1. Ermittlung des Ersatzwerts

Wenn ein Wert aus einem Baum entnommen wird, gibt es die Fälle, in denen der Wert ganz am Ende des Baumes ist und die Fälle, in denen der Wert mitten in der Struktur steckt.

Wird ein Wert entnommen, stellt sich die Frage, welcher Wert an die Stelle rutscht, an der der entnommene Wert vorher war.

- Gibt es keinen Nachfolger stellt sich die Frage nicht, der entnommene Wert wird durch einen leeren Knoten ersetzt, die Höhe des Baums aus dem entnommen wurde verändert sich dadurch.
- Gibt es nur einen Nachfolger (links oder rechts) wird dieser Nachfolger anstelle des entnommenen Knotens positioniert, die Höhe des Baums aus dem entnommen wurde verändert sich dadurch.

[10] WIe das geht wird gesondert beschrieben.

- Gibt es jedoch zwei Nachfolger, muss entschieden werden, wie auf möglichst effiziente Art ein geeigneter Ersatz gefunden werden kann, ohne dass die komplette Struktur des Baums verändert werden muss. Damit die Suchbaumeigenschaften erhalten bleiben, kommt hierfür nur der nächstgrößere oder nächstkleinere Wert infrage[11].
    - Der nächstgrößere Wert wird ermittelt, indem der rechte Nachfolger solange nach links durchlaufen wird, bis der kleinste Wert gefunden wird (linker Nachfolger ist leerer Baum).
      Dieser wird entnommen.

4.2.7. Finden (findSBT: btree × elem → int)

Der Baum wird wie bei den Teilproblemen beschrieben durchlaufen.

Sobald der aktuelle Knoten den gesuchten Wert enthält, wird die Höhe des Knotens zurückgegeben.

4.2.8. Finden move-to-root (findBT: btree × elem → {int,btree})

Der Baum wird wie bei den Teilproblemen beschrieben durchlaufen.

Sobald der aktuelle Knoten den gesuchten Wert enthält, beginnt das splaying, wie bei den Teilproblemen beschrieben.

Die neue Höhe ergibt sich aus den Rotationsoperationen und wird mit dem resultierenden Gesamtbaum zurückgegeben, sobald die Wurzel erreicht wurde.

4.2.9. Finden transpose (findTP: btree × elem → {int,btree})

Der Baum wird wie bei den Teilproblemen beschrieben durchlaufen.

Sobald einer der Kinder des aktuellen Knotens den gesuchten Wert enthält (Look-Ahead 1), wird der aktuell betrachtete Baum gemäß der entsprechenden Richtung, wie bei den Teilproblemen beschrieben, einmalig rotiert.

Die neue Höhe des gesuchten Werts ergibt sich aus der Rotation und wird in einer Hilfsvariable gesichert.

Oberhalb des veränderten Teilbaums wird der Baum auf dem Rückweg der Rekursion unverändert zusammengesetzt.

Die in der Hilfsvariable gespeicherte Höhe wird mit dem resultierenden Gesamtbaum zurückgegeben, sobald die Wurzel erreicht wurde.

[11] Würde stattdessen ein anderer Wert (z.B. der übernächstgrößere) genommen, wäre der nächstgrößere Wert als kleinerer Wert des rechten Unterbaums falsch platziert.

4.2.10. Ausgabe (printBT: btree × filename → dot)

(Unverändert aus Aufgabe 3 übernommen.)

Der Baum wird mit Hilfe des Programmes GraphViz dargestellt.

Hierfür muss eine Datei erzeugt werden die gemäß des GraphViz Standards geparst werden kann.

Eine einfache Ausprägung einer solchen Datei besteht aus einem Befehlssatz einleitender Metadaten (Header) und einem Block anzuzeigender Informationen (Body).

Der Block wird begrenzt durch eine öffnende geschweifte Klammer am Anfang und eine schließende geschweifte Klammer am Ende.

Die einzelnen Knoten des Baums werden strukturiert, indem für jeden Knoten der Wert, der Nachfolgewert und die Höhe des Nachfolgers gemäß der vorgegebenen Struktur in die Datei ausgegeben werden.

```
digraph avltree
```

Header

```
{
    // beliebig viele Wiederholungen, getrennt durch Zeilenumbruch
    // und (optionalem) Semikolon
    Wert -> Nachfolger_Wert [label = Nachfolger_Höhe];
}
```

Body

5. Testumgebung

Die Testumgebung dient zur Messung der Laufzeit der drei vorgestellten Optimierungsstrategien.

Um Rückschlüsse über die Laufzeit und somit Effizienz der Operationen ziehen zu können, werden “große” Mengen von Zahlen mittels der Util-Funktion randomliste/1 in einen leeren Binärbaum eingefügt.

Zusätzlich wird eine Liste mit zu suchenden Werten erstellt (mit Duplikaten), deren Werte in dem befüllten Binärbaum vorkommen, für die je eine Sequenz von Suchoperationen pro Optimierungsstrategie durchgeführt wird.

Die Laufzeiten werden gemessen und die resultierenden Bäume mittels printBT/1 ausgegeben.

Um Verfälschungen durch “Rekursionsbeiwerk” zu vermeiden, sollt anstatt nur einmal die Laufzeit der Schleife zu messen, die Laufzeit nur von den einzelnen ausgeführten Operationen gemessen werden.

<ToDo: Messungen durchführen, protokollieren und interpretieren.>