Seminarausarbeitung

Transform-and-Conquer

Alexander Sittig
winf8045

abgegeben am
04.06.2008

Inhaltsverzeichnis

1. Grundlagen

Transform-and-Conquer baut auf der grundlegenden Idee auf, ein Problem durch Überführung in ein Anderes zu lösen. Für diese Überführung gibt es verschiedene Ansätze:

- *instance simplification* hier wird das Problem in eine einfacher zu lösende Darstellung überführt z.B. Presorting einer Liste, Gaußsches Eliminationsverfahren und weitere
- *representation change* hier wird das Problem in eine andere Repräsentation überführt z.B. Darstellung von Mengen als 2-3 Bäume, heaps und heapsort
- *problem reduction* hier wird ein Problem in ein anderes Problem für das bereits ein Algorithmus bekannt ist überführt. Diese Technik wird in Bereichen der Linearen Programmierung und zum Vereinfachen von Problemen in Graphen genutzt

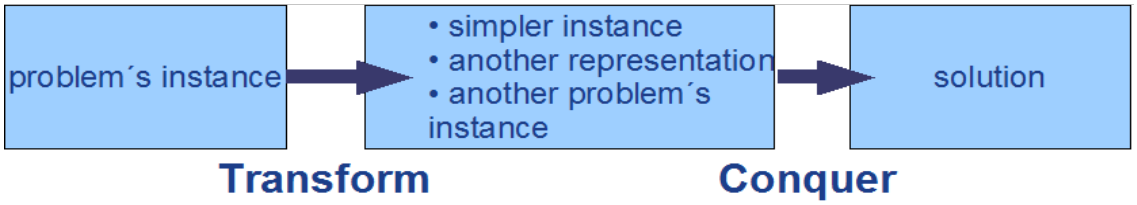


Abbildung 1: Struktur Transform-and-Conquer

Wie in der Abildung 1 zu sehen ist, kann man die Schritte „Transform“ z.B. das Sortieren einer Liste und das „Conquer“ z.B. das Bestimmen der gesuchten Lösung voneinander trennen.

2. Presorting

Als erstes wollen wir uns die sehr einfache aber auch sehr bildliche Möglichkeit des Presorting anhand zweier Beispiele genauer ansehen. Hier nutzt man den Vorteil, dass auf sortierten Arrays viele Fragen schneller beantwortet werden können.

Überprüfung auf Eindeutigkeit

Man hat ein unsortiertes Array mit N Elementen

34	12	35	48	15	32	67	41	95	38	85	43	63

Jetzt will man überprüfen, dass jedes Element nur einmal im Array vorkommt. Hierfür muss man im Brute-Force Ansatz jedes Element mit den n-1 verbleibenden Elementen vergleichen.

Wenn man jedoch ein sortiertes Array mit N Elementen hat.

12	15	32	34	35	38	41	43	48	63	67	85	95

Muss man nur n-1 Elemente mir ihrem Nachfolger vergleichen.

Vergleich: Bei einem unsortierten Array benötigt man n * (n-1) = n^2 – n Vergleiche -> $O(n^2)$

Bei einem sortierten Array benötigt man n -1 Vergleiche -> O(n)

Fazit: Für das Sortieren eines Arrays nötigt man mit z.B. dem Mergesort O(n log n) Zeit + O(n) Zeit für die Überprüfung = O(n log n) Gesamtzeit wenn man das unsortierte Array erst sortiert und anschließend überprüft, die direkte Überprüfung benötigte $O(n^2)$. Daraus schließt man, dass sich in diesem Fall das Vorsortieren lohnt.

Suche nach einem Element

Man hat wie im letzten Beispiel wieder ein unsortiertes Array mit N Elementen und man sucht ein bestimmtes Element. Im schlechtesten Fall muss man jetzt alle N Elemente mit dem gesuchten Element vergleichen um festzustellen, dass das Element nicht enthalten ist -> O(n). Im Durchschnitt muss man n/2 Elemente vergleichen bis man das Element gefunden hat -> auch O(n)

Wenn das Array sortiert ist kann man mit der Binärsuche in O(log n) das Element finden oder feststellen, dass es nicht enthalten ist.

Vergleich: Bei einem unsortierten Array benötigt man n/2 Vergleiche -> O(n)

Bei einem sortierten Array benötigt man $\log_2$ n Vergleiche -> O(log n)

Fazit: Da man für das Sortieren O(n log n) Zeit benötigt und die direkte Suche nur O(n) Zeit braucht, ist es bei einer Suche nicht sinnvoll, das Array erst zu sortierten und dann nach dem Element zu suchen. Jedoch ist es bei mehreren Suchen auf dem selben Array durchaus vernünftig erst zu sortieren und dann zu suchen. Da der Sortiervorgang nur einmal durchgeführt werden muss.

3. Suchbäume

Ein Suchbaum ist eine auf Bäumen basierende Datenstruktur mit dem Ziel in ihnen gespeicherte Elemente effizient zu durchsuchen.

Folgende Operationen werden unterstützt

- *insert* Einfügen eines Element
- *delete* Löschen eines Element
- *find* Ein Element finden

Binärbäume

Bei einem Binärbaum hat jeder Knoten einen Schlüssel k und max. zwei Kindknoten. Im Linken Kindknoten sind alle Schlüssel < k und im Rechten sind alle Schlüssel > k.

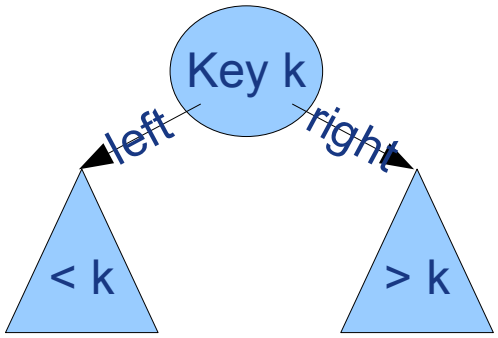


Abbildung 2: Aufbau Binärbaum

Motivation für Balancierte Bäume

Im besten Fall hat man bei einem einfachen Binärbaum schon einen optimalen Baum. In diesem Beispiel wurde für die Liste 24,15,58,12,20,30,70 ein Baum erzeugt. Man sieht, dass sich bei jedem Vergleich die Restmenge halbiert – deshalb hat man eine Such-, Lösch- und Einfügelaufzeit von O(log n)

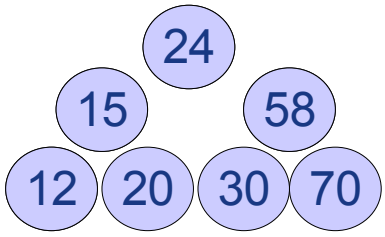


Abbildung 3: Best Case

Im schlechtesten Fall degeneriert der Baum aber zu einer Linearen Liste. Im Beispiel wurde ein Baum für die Liste 12,15,20,24,30,58,70 erstellt. Da sich nach jeden Vergleich die Restmenge nur um ein Element verkleinert, hat man nur eine Such-, Lösch- und Einfügelaufzeit von O(n)

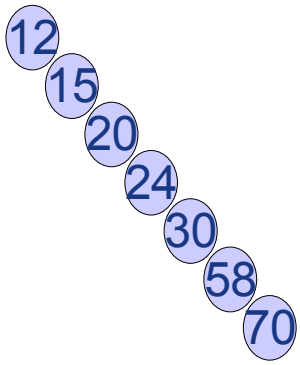


Abbildung 4: Worst Case

Balancierte Bäume

Bei Balancierten Bäumen verhindert man das Entstehen einer Linearen Liste durch bestimmte Rahmenbedingungen. Im folgenden wird die Idee der AVL- und der 2-3 Bäume vorgestellt.

3.1 AVL-Bäume

Bei AVL- Bäume wurden 1962 von den russischen Wissenschaftlern G.M. **A**delson-**V**elsky und E.M. **L**andis entwickelt. Definition:

- Ist ein Binärbaum (Jeder Knoten hat max. 2 Kindknoten und einen Schlüssel)
- An jeden Knoten wird zusätzlich die Längendifferenz zwischen linken und rechten Teilbaum gespeichert – genannt „*balance factor*“
- Als „*balance factor*“ ist nur -1, 0, 1 erlaubt
- Die Höhe des leeren Baums beträgt -1

In der Abbildung 5 sieht man links einen AVL Baum und rechts einen Baum der die AVL Bedingung, dass es nur Balance Faktoren zwischen -1 und 1 erlaubt sind, nicht erfüllt. Wie man jetzt aus diesem Baum wieder einen AVL Baum macht, werden wir im folgenden sehen.

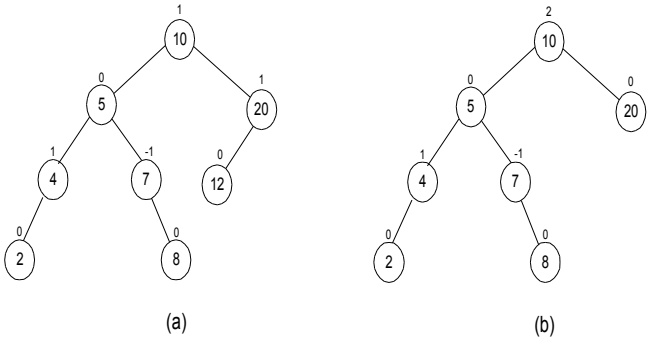


Abbildung 5: links AVL Baum - rechts kein AVL Baum

Es gibt 4 so genannte *rotations* um einen Baum wieder auszubalancieren.

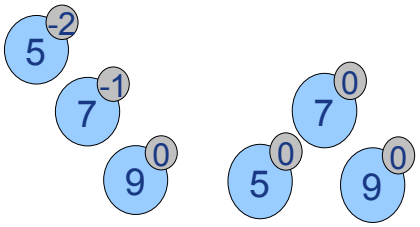


Abbildung 6: L-Rotation - L(5)

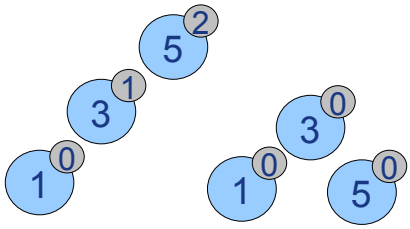


Abbildung 7: R-Rotation - R(5)

Die ersten beiden Rotationen sind die Einfachdrehung nach Link oder Rechts.

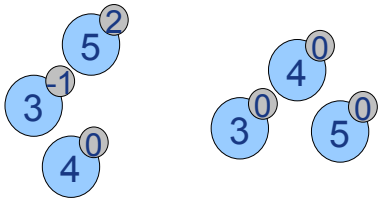


Abbildung 8: LR Rotation - LR(5)

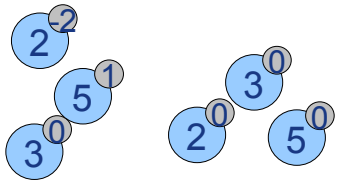


Abbildung 9: RL Rotation - RL(2)

Die zweiten beiden Rotationen sind die Doppeldrehungen Links Rechts oder Rechts Links

Beispiel: Erstellung eines AVL-Baums für 5,6,8,3,2,4,7

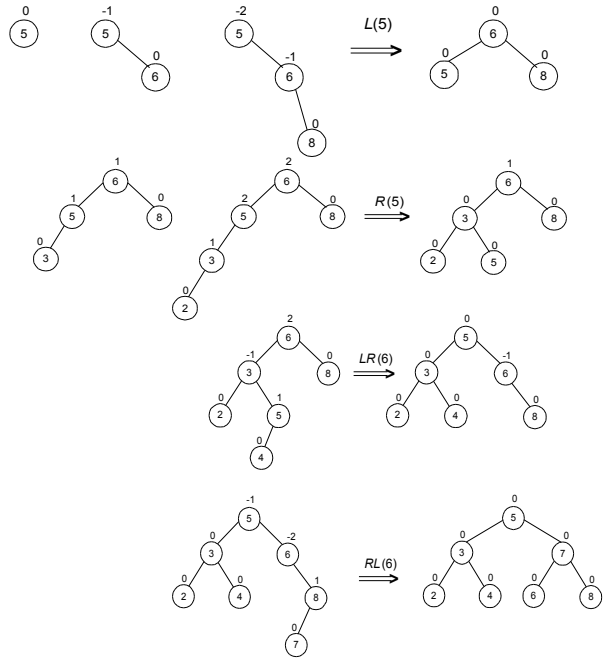


Abbildung 10: Aufbauen eines AVL Baum

Laufzeitabschätzung:

Die Höhe eines AVL beträgt maximal: $1{,}44 \log_2(n+2) - 1{,}3277$

Die Formel wurde aus dem Buch: The Design & Analysis of Algorithms – Anany Levitin Seite 218 übernommen

Durch die logarithmische Höhe des AVL Baums laufen die Operationen Einfügen, Löschen und Suchen auch in logarithmischer Zeit -> O(log n)

3.2 2-3 Bäume

Eine zweite Variante für Balancierte Bäume sind die 2-3 Bäume. Diese wurden 1970 von John Hopcraft entwickelt. Definition:

- Keine Binärbäume sondern Mehrwegbäume
- Zwei Knotentypen
  - Mit einem Schüssel und zwei Kindknoten wie bei Binärbäumen genannt *2-Node*
  - Mit zwei Schlüsseln und drei Kindknoten genannt *3-Node*
- Es wird nur in den Blättern eingefügt und gelöscht
- Die Informationen werden in allen Knoten gespeichert

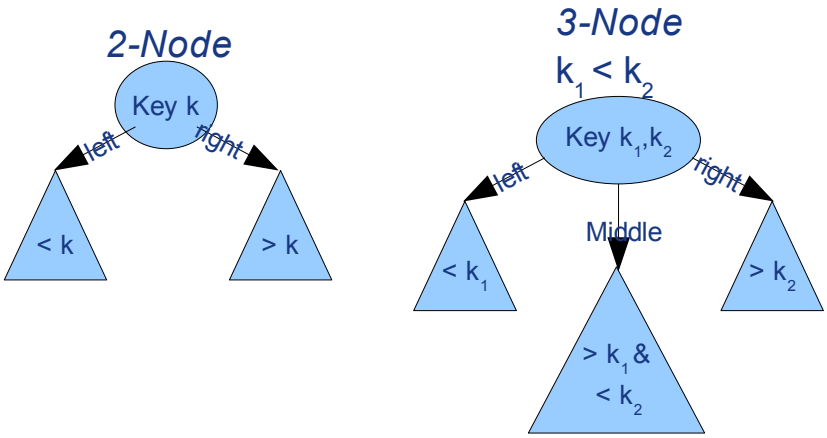


Abbildung 11: Knotentypen 2-3 Baum

Einfügen in einem 2-3 Baum

1. Finde das Blatt in dem eingefügt werden soll
2. Wenn das Blatt ein *2-Node* ist, erstelle ein *3-Node* mit dem vorhandenen Schlüssel und dem einzufügenden Schlüssel
3. Wenn Blatt ein *3-Node ist* -> teile Knoten
4. Sollte der Eltern Knoten(P) bereits ein *3-Node* sein, dann führe Teilungsregel für innere Knoten durch, siehe Abbildung 13

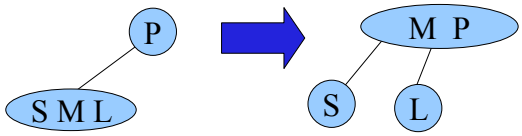


Abbildung 12: Teilen eines Blattes

Teilen eines inneren Knotens, die Buchstaben a-d beschreiben die Teilbäume die an den Knoten hängen

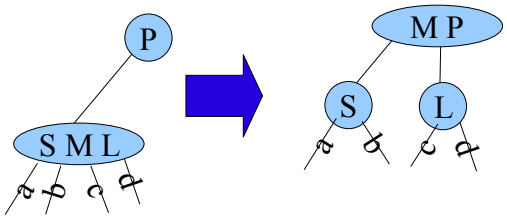


Abbildung 13: Teilen eines inneren Knotens

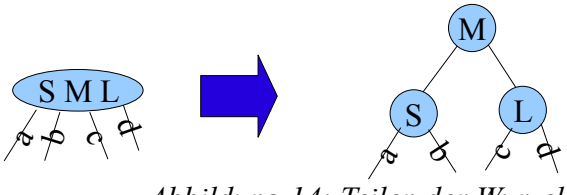


Abbildung 14: Teilen der Wurzel

Löschen in einem 2-3 Baum

Beim Einfügen hat man die Knoten aufgeteilt und beim Löschen muss man die Knoten wieder zusammenfügen.

Da nur in den Blättern gelöscht werden darf, aber auch Einträge gelöscht werden sollen, die nicht in den Blättern stehen, muss man diese mit ihrem Nachfolger(Blatt) vertauschen.

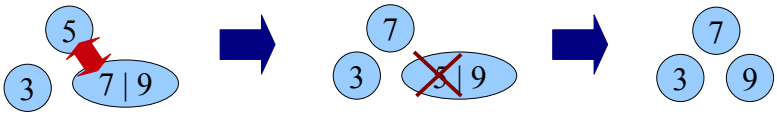


Abbildung 15: Tauschen und Löschen

Ist das Element in einem *3-Node,* dann löscht man einfach den Schüssel aus dem Knoten und man hat einen *2-Node*.

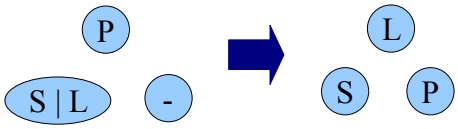


Abbildung 16: Umverteilen

Bei einen *2-Node* als Elternknoten, einem *3-Node* und einem *2-Node,* den man Löschen will, muss man, wie in Abbildung 16 zu sehen, die Knoten umverteilen.

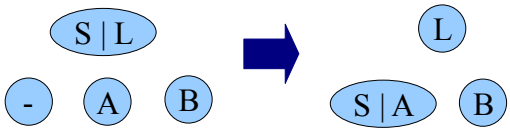


Abbildung 17: Zusammenfassen

Letztes Beispiel fürs Löschen ist ein *3-Node* als Elternknoten und drei 2-Knoten als Kinder wovon einer gelöscht werden soll. Hier muss man die Knoten zusammenfassen.

Laufzeitabschätzung

Die Höhe eines 2-3 Baumes beträgt maximal $\log_2(n+1) - 1$

Die optimale Höhe eines 2-3 Baumes erkauft man sich mit der Möglichkeit, dass ein Knoten zwei Datenelemente speichern kann. Deshalb hat man bei der Suche im schlechtesten Fall 2* Höhe des Baumes Vergleiche, wenn alle Elemente auf dem Weg der Suche *3-Nodes* sind. Suche: max. 2* ($\log_2(n+1) - 1$) Vergleiche -> O(log n). Gleiches gilt fürs Einfügen und Löschen, wobei hier noch die Anzahl der Teilungsoperationen oder Zusammenfassoperationen mitgezählt werden müssen. Man bleibt aber in O(log n) Zeit.

4. Heap

Ist eine auf Bäumen basierende Datenstruktur. Definition:

- Binärbaum

- Ein Schlüssel pro Knoten
- Beide Kindknoten sind kleiner oder kleiner gleich dem Schlüssel
- Der Baum ist komplett. Das heißt: Jeder Knoten hat zwei Kindknoten außer die Knoten auf der untersten Ebene rechts.

Konstruktion eines Heaps

Am Anfang wird die Baumstruktur mit den Schüsseln in gegebener Reihenfolge initialisiert.

Dann wird beginnend beim Knoten rechts unten die Heap Bedingung (alle Kindknoten < Schlüssel) überprüft, sollte die Bedingung nicht erfüllt sein, werden die Knoten getauscht und die Bedingung erneut geprüft.

Beispiel: Konstruktion eines Heaps für 2, 9, 7, 6, 5, 8

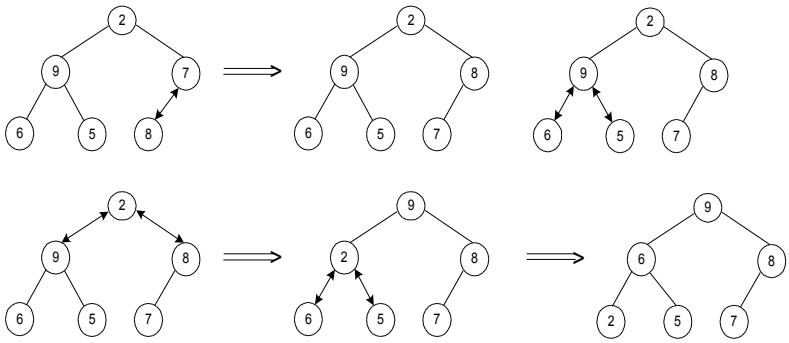


Abbildung 18: Heap Konstruktion

Laufzeitabschätzung

Beim Einfügen eins neuen Elements wird die Anzahl der Tauschoperationen durch die Höhe des Baumes beschränkt und da der Baum laut Definition ein dichter Baum ist -> O(log n)

Beim Erstellen eines Heaps ist die Laufzeit O(n)

4.1 Heapsort

Der erste Schritt des Heapsort ist die Konstruktion eines Heaps für das gegebene Array

Der zweite Schritt ist n mal das größte Element aus dem Heap zuentfernen.

Laufzeitabschätzung

Konstruktion eines Heap für n Elemente -> O(n)

n mal das größte Element löschen -> n * O(log n) -> O(n log n)

Gesamtlaufzeit: O(n) + O(n log n) = O(n log n)

5. Wörterbuchproblem

Für ein Wörterbuch fordert man, dass die Operationen:

- Einfügen
- Löschen
- Suchen

effizient (in O(log n)) implementiert sind.

Ein 2-3 Baum erfüllt diese Anforderungen.

UML-Diagramm für die Implementierung eines Wortbuches mit einem 2-3 Baum in Java

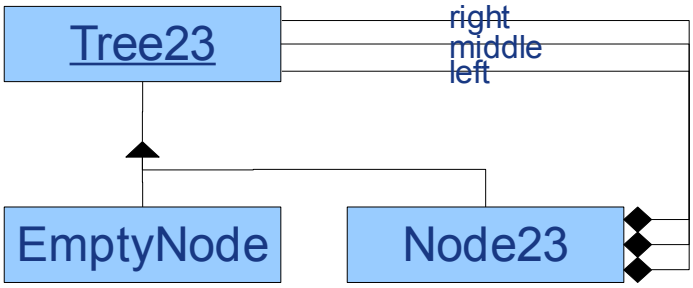
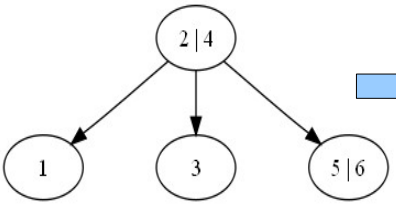


Abbildung 19: UML Diagramm 2-3 Baum

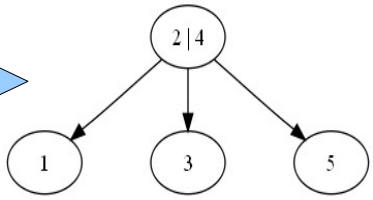
Beispiel für das Löschen in einem 2-3 Baumes mit der erstellten Klasse

Liste: 1;2;3;4;5;6

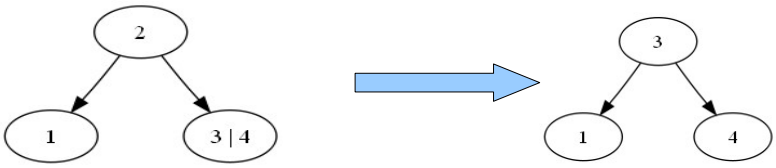
Löschen der 6



Löschen der 5



Löschen der 2



6. Literaturverzeichnis

I. Levitin, Anany. Introduction to the design & analysis of algorithms ISBN 0321358287