Mitschriften zur Einführung

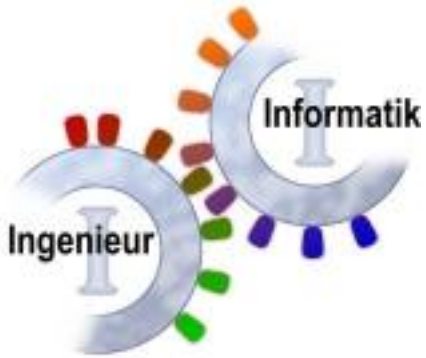
Algorithmen und Datenstrukturen

Zur Vorlesung an der Otto-von-Guericke-Universität Magdeburg gehalten von Prof. Dr.-Ing. Klaus Tönnies 2002/2003

Version 0.9-20030710-α

Stefan Schumacher, `stefan@net-tex.de`

10. Juli 2003



http://www.net-tex.de/uni

Inhaltsverzeichnis

Abbildungsverzeichnis 5

Algorithmenverzeichnis 6

1 Vorlesung 7

2 Algorithmen 9
- 2.1 Algorithmus . . . 9
- 2.2 Paradigmen . . . 9
  - 2.2.1 Imperatives Paradigma . . . 9
  - 2.2.2 Applikatives Paradigma . . . 9
  - 2.2.3 Vergleich Imperatives und Applikatives Paradigma . . . 10
  - 2.2.4 Logisches Paradigma . . . 10

3 Aufwand und Effizienz 11
- 3.1 Analysetypen . . . 11
- 3.2 Vorgehen zur Aufwandsabschätzung . . . 11

4 Abstrakte Datentypen (ADT) 12
- 4.1 Abstrakt und Konkret . . . 12
- 4.2 Abstrakte Datentypen und Java . . . 12
- 4.3 verkettete Liste . . . 13
  - 4.3.1 Operationen auf dem Listenelement: . . . 14
  - 4.3.2 Operationen auf der Liste: . . . 14
  - 4.3.3 public class Node . . . 15
  - 4.3.4 Aufwand der Liste: . . . 15
- 4.4 doppelt verkettete Liste . . . 16
  - 4.4.1 Einfügen von Elementen . . . 16
  - 4.4.2 Löschen von Elementen . . . 16
- 4.5 Stack . . . 17
  - 4.5.1 ArrayStack . . . 17
  - 4.5.2 Stack als Liste . . . 17
- 4.6 Vektor . . . 19
- 4.7 Liste . . . 19
  - 4.7.1 ADT Position . . . 19
- 4.8 ADT Sequenz . . . 20
  - 4.8.1 Sequenz als doppelt verkettete Liste . . . 20
  - 4.8.2 Sequenz als Feld . . . 20
  - 4.8.3 Sequenz als Mischimplementierung . . . 20
- 4.9 ADT Iterator . . . 20
- 4.10 ADT Container . . . 21

5 Bäume **22**
5.1 binärer Baum . . . 22
5.2 Methoden auf Baum . . . 23
5.3 Aufwand . . . 23
5.4 Tiefe des Baumes . . . 24
5.5 Höhe des Baumes . . . 24
5.6 Traversierung . . . 24
5.6.1 Preorder Traversierung . . . 25
5.6.2 Postorder Traversierung . . . 25
5.6.3 Inorder Traversierung . . . 25
5.7 Binäre Suchbäume . . . 27
5.8 Eulertour . . . 27
5.9 Datenstruktur für Bäume . . . 28
5.9.1 Vektoren für binäre Bäume . . . 28
5.9.2 Speicheraufwand . . . 28
5.9.3 binärer Baum als Zeigerstruktur . . . 28
5.9.4 beliebiger Baum als Zeigerstruktur . . . 29
5.10 Binären Baum kopieren . . . 29
5.11 Transformation in binären Baum . . . 29
5.12 binärer Baum als Vektor . . . 31
5.13 AVL-Baum . . . 31
5.14 (2-4)-Baum . . . 32
5.15 Rot-Schwarz Baum . . . 33

6 Graphen **35**
6.1 Darstellung . . . 35
6.2 Interface . . . 36
6.3 Datenstruktur . . . 37
6.4 Traversierung von Graphen . . . 38

7 Lexikon **40**
7.1 Hashtabellen . . . 40
7.1.1 Hashfunktion . . . 40
7.2 Beispielcodes . . . 41
7.3 Kollisionsverarbeitung . . . 42
7.4 Geordnete Lexika . . . 43
7.5 Sprunglisten . . . 43

8 Sortieren und Suchen **45**
8.1 Stabilitiät . . . 45
8.2 Bubble Sort . . . 45
8.3 Selection Sort . . . 45
8.4 Binary Search . . . 45
8.5 Merge Sort . . . 45

9 Mustererkennung **47**
9.1 Brute Force . . . 47
9.2 Boyer Moore . . . 47
9.3 Knuth Morris Pratt . . . 48

10 Exceptions **49**
10.1 Die Klasse Throwable . . . 49

11 Entwurfsmuster **50**
11.1 Adaption . . . 50
11.2 Divide and Conquer . . . 50
11.3 Brute Force . . . 50
11.4 Greedy . . . 50
11.5 Dynamic Programming . . . 50

12 Entwurfstechniken **51**
12.1 Zerlegungskonzept . . . 51
12.2 Überprüfung von Algorithmen . . . 52
12.2.1 Validierung . . . 52
12.2.2 Verifikation . . . 52
12.2.3 Spezifikation . . . 53
12.2.4 Backus-Naur-Form . . . 53
12.3 Visualisierung von Algorithmen . . . 53
12.4 Pseudocode . . . 56
12.5 Prioritätsschlange . . . 56
12.5.1 Kompositionsobjekte . . . 56
12.5.2 Comparator . . . 56
12.6 Heap . . . 57
12.7 Heap als Vektor . . . 57

13 Anmerkungen **58**

Abbildungsverzeichnis

4.1 einfach verkettete Liste . . . 14
4.2 Einfügen in einfach verkettete Liste . . . 14
4.3 Löschen in einfach verketteter Liste . . . 15
4.4 doppeltverkettete Liste . . . 16
4.5 Einfügen in doppeltverkettete Liste . . . 16
4.6 Löschen in doppeltverketteter Liste . . . 16
4.7 Stack . . . 17
5.8 Prätraversierung . . . 25
5.9 Posttraversierung . . . 26
5.10 binärer Baum als Zeigerstruktur . . . 29
5.11 Baum als Zeigerstruktur . . . 29
5.12 Transformation in binären Baum . . . 31
5.13 Restrukturierung eines Nicht-AVL Baumes . . . 31
5.14 Beispiel (2-4)-Baum . . . 32
5.15 Split eines (2-4)-Knotens . . . 32
5.16 Schlüssellöschen im (2-4)-Baum . . . 33
5.17 Schwarz-Rot Präsentation eines (2-4)-Baumes . . . 33
5.18 Rotation im Rot-Schwarz-Baum . . . 34
5.19 Umfärben 1 im Rot-Schwarz-Baum . . . 34
5.20 Umfärben 2 im Rot-Schwarz-Baum . . . 34
6.21 Beispielgraph . . . 39
6.22 Beispielgraph . . . 39
7.23 Prinzip der Hashfunktion . . . 41
12.24 Struktogramm / Nassi Schneidermann Diagram . . . 54
12.25 Programmablaufplan . . . 55

Algorithmenverzeichnis

1 even() . . . 10
2 Public Class node . . . 15
3 ArrayStack . . . 18
4 depth() . . . 24
5 height() . . . 24
6 heightRek() . . . 25
7 preorder() . . . 25
8 postorder() . . . 26
9 inorder() . . . 26
10 EulerTour(Tree T, node V) . . . 27
11 public class BTNode . . . 30
12 clone(T1,T2,v1,v2) . . . 30
13 Shift() . . . 42
14 pseudocode-pattern . . . 56

1 Vorlesung

Webseite zur Vorlesung : http://www.isg.cs.uni-magdeburg.de/ead/index.html
Webseite zur Übung : http://www-ai.cs.uni-magdeburg.de/ bluemel/ws-02-03/index.html
Webseite zu Tutorien : http://www-ai.cs.uni-magdeburg.de/ tutoren/

Meine Seite mit einigen Lösungen zu EAD: http://www.net-tex.de/uni/index.html

Meine Belegarbeit zum Rot-Schwarz Baum : http://www.uni-magdeburg.de/steschum/rsb/

Datum	Thema der Vorlesung	Folien (PDF)
14.10.2002	Was ist Informatik? Geschichte der Informatik, Algorithmus, Programm	Ad1.pdf
17.10.2002	Anwendung SQRT, Objektorientiertes Design, JAVA	Ad2.pdf
21.10.2002	Pseudocode, Kontrollstrukturen, Schleifen, Datentypen, Deklaration, Rekursion	AD03.pdf
24.10.2002	Kontrollstrukturen, Schleifen, Rekursion, Felder	AD04.pdf
28.10.2002	Algorithmenparadigmen, Applikative Algorithmen, Rekursion	AD05.pdf
04.11.2002	Imperatives Paradigma	AD06.pdf
07.11.2002	Logisches Paradigma	AD07.pdf
11.11.2002	Aufwand und Effizienz, O-Notation, Beispiel: Aufwandsabschätzung	AD08.pdf
14.11.2002	O-Notation, Omega-Notation, Theta-Notation, Binäre Suche, Bubble Sort	AD09.pdf
18.11.2002	Abstrakte Datentypen, Interfaces, Stack, Exceptions, Vererbung	AD10.pdf
21.11.2002	Stack, Casting, Beispiel: Region Growing	AD11.pdf
25.11.2002	ADT Queue, Implementierung der Queue, Klassen- und Objektmethoden	AD12.pdf
28.11.2002	Queue-Beispielanwendung: Postamt	AD13.pdf
02.12.2002	Verkettete Listen, Beispiel: Digitale Landkarte	AD14.pdf
05.12.2002	Stack als Liste, Queue als einfach verkettete Liste, ADT Deque	AD15.pdf
09.12.2002	ADT Vektor, Beispiel: Geometrievektoren	AD16.pdf
12.12.2002	ADT Liste, ADT Position, Beispiel: Bauvorhaben	AD17.pdf
16.12.2002	ADT Sequenz, Insertion Sort, ADT Iterator, Container	AD18.pdf
19.12.2002	Bäume, Binärer Baum, BSP Baum	AD19.pdf
10.01.2003	Bäume - Grundlegende Methoden, Traversierung	AD20.pdf
13.01.2003	Binäre Suchbäume, Euler-Tour, Abstrakte Klassen	AD21.pdf
16.01.2003	Klausur	
20.01.2003	Datenstrukturen zur Repräsentation von Bäumen	AD22.pdf
23.01.2003	Entwurfstechniken, Programmierwettbewerb	AD23.pdf
27.01.2003	Validierung von Programmen	AD24.pdf
30.01.2003	Prioritätsschlangen, Comparator	AD25.pdf

1.03.2003	Heap, Prioritätsschlange als Heap, Heap als Vektor, bin. Baum als Vektor, HeapTree	AD26.pdf
03.04.2003	Operationen auf Heap, Heap-Sort, In-Place-Heap-Sort	AD27.pdf
07.04.2003	ADT Dictionary, Hash-Tabellen	AD28.pdf
10.04.2003	Lexikon, Kollisionsverarbeitung, Implementierung, Double Hashing	AD29.pdf
14.04.2003	Geordnete Lexika, Look-Up Table, Sprunglisten	AD30.pdf
17.04.2003	Suchbäume, Binäre Suchbäume, Einfügen und Löschen	AD31.pdf
24.04.2003	AVL-Bäume	AD32.pdf
28.04.2003	Nichtbinäre Suche, (2,4)-Baum	AD33.pdf
05.05.2003	Einfügen und Löschen im (2,4)-Baum, Einführung Rot-Schwarz-Baum	AD34.pdf
08.05.2003	Einfügen und Löschen im Rot-Schwarz-Baum	AD35.pdf
12.05.2003	Sortierverfahren, Merge Sort	AD36.pdf
15.05.2003	Quick Sort	AD37.pdf
22.05.2003	Bucket Sort, Lexikographische Suche, Prune-and-Search	AD38.pdf
26.05.2003	Pattern Matching, Boyer-Moore-Algorithmus	AD39.pdf
02.06.2003	Pattern Matching, Knuth-Morris-Pratt Algorithmus	AD40.pdf
05.06.2003	Graphen, Einführung	AD41.pdf
12.06.2003	Datenstrukturen für Graphen	AD42.pdf
16.06.2003	Traversierung	AD43.pdf
23.06.2003	Gerichtete Graphen	AD44.pdf
26.06.2003	Kürzeste Wege	AD45.pdf
30.06.2003	Graphensuche	AD46.pdf

2 Algorithmen

2.1 Algorithmus

Ein Algorithmus ist eine präzise endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer Schritte.
endlich - der Algorithmus muss terminieren (wie auch immer, d.h er kann z.B. auch eine Fehlermeldung zurückgeben)
Terminierung heisst, das der Algorithmus nach endlich vielen Schritten abbricht.

Ein **deterministischer Ablauf** ist ein Algorithmus der eine eindeutige Folge der erinzelnen Schritte festlegt.
Ein **determinierendes Ergebnis** ist ein eindeutiges Ergebnis das bei vorgegebener Eingabe zurückgegeben wird.

2.2 Paradigmen

2.2.1 Imperatives Paradigma

Der Algorithmus ist eine Folge von Befehlen die eventuell mehrfach durchlaufen werden ('Wasserfall').
Die Summe der Befehle kennzeichnet die Semantik des Algorithmus.
Das Resultat der Befehlsausführung ist die Ausführung aller Befehle.
Hauptsächliche Kontrollkonstrukte sind Schleifen und Abfragen.
Jede Anweisung ändert den Zustand von Variablen des Algorithmus.
Eingabe: Vorbedingung VOR.
Ausgabe: Nachbedingung NACH
Algorithmus: VOR <Anweisungsfolge>NACH
<Anweisungsfolge>::= <Anweisung>—<Anweisung><Anweisungsfolge>
Jede Anweisung a_i hat eine Vorbedingung v_i und eine Nachbedingung n_i.
Für eine Folge von Anweisungen $a_1, a_2, \ ... \ , a_n$ die hintereinander ausgeführt wird, wird die Nachbedingung n_i zur Vorbedingung v_{i+1}.
Für jeden Anweisungstyp müssen Vor- und Nachbedingungen gefunden werden.
Die Transformationen von Vor- in Nachbedingungen müssen für alle Anweisungen der Folge ausgeführt werden.

2.2.2 Applikatives Paradigma

Merkmal ist der wiederholte Aufruf der Funktion in sich selbst durch sich selbst auf ein Teilproblem bezogen.
Erfordert mindestens ein Kontrollkonstrukt um zu beenden.
Erste Funktion ist die Semantik des Algorithmus, Funktion ist ein Term von Termen, Wert der Funktion ist die jeweilige Auswertung des Terms
Endrekursion : spezielle Form der Rekursion, lässt sich leicht in Iteration umwandeln,

da die Anzahl der Schritte bekannt ist.

$$f(x) = \begin{cases} g(x)\ falls\ p(x) \\ \\ f(r(x))\ sonst \end{cases} \qquad f(x) = \begin{cases} x\ falls\ x<2 \\ \\ even(x-2)\ sonst \end{cases}$$

Algorithm 1 even()

```
if n < 2 then
  return(n)
else
  even(n − s)
end if
```

2.2.3 Vergleich Imperatives und Applikatives Paradigma

	Applikativ	Imperativ
Auswertung	Termauswertung	Zustandsänderungen
Elementare Konstrukte	Terme von Unbestimmten	Terme von Variablen
Komplexe Konstrukte	Menge von Funktionen	Anweisungsfolge
	Fallunterscheidung	bedingte Anweisung
	Rekursion	Schleife
Resultat	Wert der ersten Funktion	Endzustand

2.2.4 Logisches Paradigma

Folge von Sätzen. Eine Menge von Aussagen und Aussageformen.
Algorithmus beschreibt einen Sachverhalt.
Ein Resultat wird durch eine Anfrage an das System von Sachverhalten erzeugt.
Basiseinheit sind elementare Aussagen
Es gibt eine Menge von Regeln über den Aussagen
Es wird nur dann ein Ergebnis erzeugt wenn die Anfrage aufgrund der vorhandenen Regeln und Fakten beantwortet werden kann.
Ein Algorithmus ist die Deduktion von Fakten aus eine Menge von Fakten und Regeln.
Resultat eines deduktiven Algorithmus D: ist die Menge aller aus den Regeln und Fakten ableitbaren Fakten.

3 Aufwand und Effizienz

Aufwand ist der verbrauchte Speicherplatz und Rechenzeit
Abschätzung der benötigten:
Rechenzeit:

- Jede Operation benötigt eine abstrakte Zeiteinheit
- Operationen: arithmetische/logische Operationen, Vergleich, Zuweisung, Feldadressierung, Methodenaufruf/-rückkehr, Referenzauflösung

Speicher:

- Speicherplatz der primitiven Datentypen
- deklarierte Variablen in Objektmethoden
- Elemente in einem Array
- Methoden und deklarierte Variablen in Objekten

3.1 Analysetypen

Best Case:
Einschätzung des besten Falles, kaum reell zutreffend, eigentlich ist dies der Abbruch des Algorithmus durch eine Exception
Average Case:
Aufwand bei Durchschnittsverteilung der Eingaben daher ist statistische Kenntnis über Verteilung der Eingaben nötig
Worst Case:
Der schlechteste Fall wird betrachtet
leichter berechenbar, reeller und sicherer, da alle Möglichkeiten abgedeckt werden
In der Regel die Aufsummierung der erforderlichen Schritte eines Algorithmus, wobei eigentlich nur der grösste Anteil des Aufwands von Bedeutung ist, bei $n^3 + n^2 + 3n$ ist nur n^3 von Interesse.

3.2 Vorgehen zur Aufwandsabschätzung

- Suche der Parameter
- Suche der am häufigsten ausgeführten Operation (verschachtelte Schleifen / innere Schleife)
- Berechnung der Anzahl der Schleifenschritte

Die Wahl des Algorithmus hängt ab von der
- Aufwandsabschätzung
- vom realistischen n
- vom Verhältnis zwischen Worst Case und Average Case

4 Abstrakte Datentypen (ADT)

4.1 Abstrakt und Konkret

- Zusammenfassung vom Eigenschaften und Operationen einer Datenstruktur
- Unabhängig von ihrer Realisierung (Kapselung / Information Hiding)
- Datenstrukturen ohne Kenntnis der Implementierung nutzbar
- gibt die Schnittstelle für alle darauf implementierten Datentypen an
- die konkrete Implementierung ist nicht Teil des ADT
- ADT wird nur über die Schnittstelle verwendet

Eine beschreibt das Verhalten des Datentyps, also alle Funktionen die auf dem Datentyp definiert sind und die Definitions und Wertebereiche der Funktionen. WB und DB sind durch (andere Datentypen) gekennzeichnet, der ADT definiert eine neue Sorte über das Verhalten bekannter Sorten.

4.2 Abstrakte Datentypen und Java

- Objekte
  - Abstraktion eines Gegenstands aus dem Anwendungsgebiet und/oder dessen Implementierung
  - kann Daten des Gegenstandes (auch gekapselt) speichern
  - wird durch Attribute und Methoden beschrieben
- Klasse
  - Beschreibung einer Menge von (nahezu) gleichen Objekten
  - Objekte sind die Instanz einer Klasse
  - konkrete Klasse besitzt Objekte
  - abstrakte Klasse besitzt keine Objekte
- Attribute
  - beschreiben die Datenstruktur einer Klasse (Variablen)
- Methoden
  - *Funktionen* und *Prozeduren* die in einer *Klasse* enthalten sind
  - legen die Reaktionen eines *Objekts* auf eintreffende *Nachrichten* fest
  - ein *Objekt* befindet sich in einem bestimmten Zustand (*Attribute*) der durch *Methoden* verändert werden kann
- Vererbung
  - Unterklasse übernimmt Datenstruktur (*Attribute*) und das Verhalten (*Attribute*) von einer/mehreren existierenden überklassen
  - `class Unterklasse extends Überklasse`
- Nachrichten

  - Mitteilungen an *Objekte* um Funktionen dynamisch zu starten

- Konstruktor

  - Methode zur *Initialisierung* eines *Objekts* der zugehörigen *Klasse*
  - Rückgabewert ist immer der eigene Klassentyp

- new

  - erzeugt ein neues *Objekt* und alloziert entsprechenden Speicher
  - benötigt den entsprechenden *Konstruktor*

    ```
    Klasse klassenobjekt = new Klasse();
    ```

- Konstruktorverkettung

  - *Objekte* einer Unterklasse erfordern die *Konstruktoren* aller überklassen
  - `super()`

    ruft dazu den entsprechenden Standardkonstruktor der jeweiligen überklasse auf

- Interfaces

  - umfassen abstrakte öffentliche Methoden sowie Konstantendefinitionen
  - `interface Name [extends ... ... ...]`
    ```
    {
     Methodendeklarationen
    }
    ```
  - *Interfaces* können über

    ```
    extends
    ```

    mehrfach erben

- Implementierung

  - *Klassen* implementieren Schnittstellen mit
  - `class Klasse implements Interface`

4.3 verkettete Liste

einfach verkettete Liste:
- besteht aus Elementen
- jedes Element verweist auf seinen Nachfolger

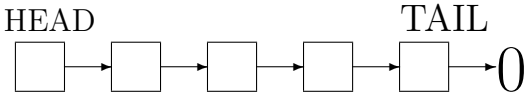


Abbildung 4.1: einfach verkettete Liste

head: zeigt auf das erste Element der Liste
tail: zeigt auf das letzte Element der Liste
next: zeigt auf das nächste Element in der Liste
element: Eigenschaften eines Listenelements

4.3.1 Operationen auf dem Listenelement:

- Rückgabe des Listeninhalts
- Einfügen bzw. Ändern des Inhalts
- Referenz zum nächsten Element

4.3.2 Operationen auf der Liste:

- Einfügen von Elementen:

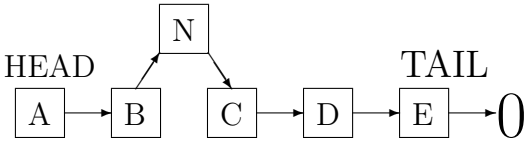


Abbildung 4.2: Einfügen in einfach verkettete Liste

1. neues Element N auf seinen neuen Nachfolger C zeigen lassen
2. Vorgänger B auf neues Element N zeigen lassen

- Einfügen von Elementen am Anfang der Liste:
1. neues Element N auf seinen neuen Nachfolger A zeigen lassen
2. Head auf neues Element N zeigen lassen

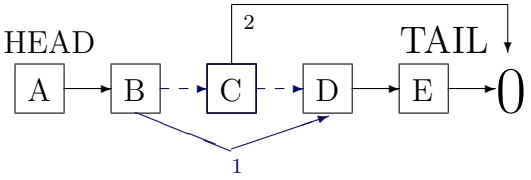


Abbildung 4.3: Löschen in einfach verketteter Liste

- Löschen von Elementen:
1. Referenz von B auf D setzen
2. Referenz von C auf 0 setzen, C wird vom Garbage Collector entsorgt

- Löschen von Head oder Tail:
analog, es müssen die entsprechende Head bzw. Tail Referenzen umgesetzt werden.

4.3.3 public class Node

Algorithm 2 Public Class node

```
public class Node { // Klasse für Listenelemente
    private Object element; //  Inhalt  des Listenelements
    private Node next; // Zeiger auf das nächste Listenelement
    public Node() { // Konstruktoren: leerer Knoten
    this(null,null); }
    public Node (Object e, Node n) { // Knoten mit Inhalt
        element = e; next = n; }
    public Object getElement() { // diverse Zugriffsmethoden
        return element; }
    public Node getNext() {
        return next; }
    public void setElement(Object newElem) {
        element=newElem; }
    public void setNext (Node newNext) {
        next = newNext; }
  }
```

4.3.4 Aufwand der Liste:

	Feld	Liste
Zugriff	Index: 0(1)	Pointer: O(n)
Einfügen	Verschieben: O(n)	Pointer umsetzen O(1)
Größe	statisch vordefiniert	dynamisch änderbar

4.4 doppelt verkettete Liste

- Listenelemente verweisen auf Nachfolger `next` und Vorgänger `prev`
- wird durch zwei Spezialknoten (`header` und `trailer`) abgeschlossen
- Einfügen und Entfernen an beiden Enden mit O(1)

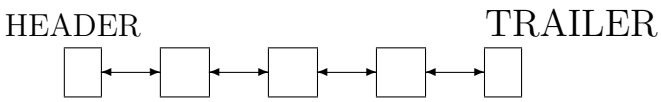


Abbildung 4.4: doppeltverkettete Liste

4.4.1 Einfügen von Elementen

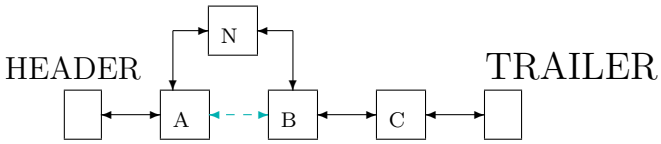


Abbildung 4.5: Einfügen in doppeltverkettete Liste

- N.next auf B, N.prev auf A
- B.prev auf N, A.next auf N

4.4.2 Löschen von Elementen

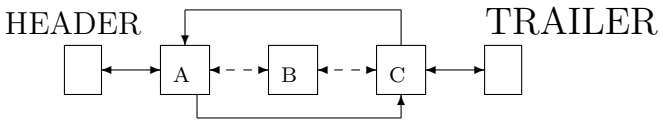
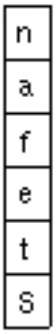


Abbildung 4.6: Löschen in doppeltverketteter Liste

- A.next auf C , C.prev auf A setzen
- B ist dereferenziert und wird vom Garbage Collector eingesammelt

4.5 Stack



Stack (oder Kellerspeicher) speichert Objekte und gibt sie nach dem LIFO Prinzip wieder aus, das heisst das letzte Element wird als erstes zurückgegeben.
'Stefan' ergibt ausgelesen 'nafetS'

Abbildung 4.7: Stack

4.5.1 ArrayStack

Implementierung des Stacks als ArrayStack (Array mit N Feldern von denen t belegt sind)
Nachteile:
Feldgrösse ist vordefiniert, da ein Array zur Realisierung verwendet wird.

4.5.2 Stack als Liste

Bei der Implementierung des Stack als Liste fällt die `StackFullException` weg, da die Größe der Liste beliebig geändert werden kann.
Notwendige Operation
- Anfügen eines neuen Elements am Ende
- Rückgabe und Entfernen des letzten Elements

Aufwand:
Einfügen an head: O(1)
Einfügen an tail: O(1)
Löschen an head: O(1)
Löschen an tail: O(n), da Vorgänger gefunden werden muss

Algorithm 3 ArrayStack

```
Algorithmus size()
return t + 1 // Anzahl belegte Felder des ArrayStack
Algorithmus isEmpty()
return (t < 0)
Algorithmus top()// gibt Element zurueck
if isEmpty()    /* Abfrage ob Stack leer */ then
   throw StackEmptyException
   return S[t] // oberstes Feldelement wird zurueckgegeben
end if
Algorithmus push(o)
if size() = N then
   throw StackFullException
   t ← t + 1
   S[t] ← o
end if
Algorithmus pop() // gibt Element zurueck & loescht es
if isEmpty()    /* Abfrage ob Stack leer*/ then
   throw StackEmptyException
end if
e ← S[t]    // e wird oberstes Element
S[t] ← null    //oberstes Feld wird genullt
t ← t − 1    // Groesse des Feldes wird dekrementiert
return e
```

4.6 Vektor

- lineare Sequenz von Elementen
- Element kann über Rang (Position) angesprochen werden
- Realisierung erfolgt idealerweise über ein Feld

Aufwand:

size()	O(1)
isEmpty()	O(1)
elementAtRank()	O(1)
replaceAtRank	O(1)
insertAtRank	O(n) durch Umsortierung
removeAtRank	O(n) durch Umsortierung

Nachteil: Feldgröße ist begrenzt
Lösung: Wenn Feld voll ist, Feld erweitern.
1. Möglichkeit:

- erzeuge neues Feld der Größe 2N
- kopiere alten Feldinhalt in neues
- setze Referenzierung des alten Feldes auf neues Feld
- altes Feld ist dereferenziert und wird eingesammelt
- Aufwand Verlängerung O(k) allerdings ist die Häufigkeit der Verlängerungen relevant, daher Durchschnittsbetrachtung

2. Möglichkeit:
- erzeuge neues Feld mit jeweils doppelter Länge

4.7 Liste

- Abstraktion der verketteten Liste
- Lineare Sequenz
- Elemente werden über Position angesprochen

4.7.1 ADT Position

- immer relativ
- hat Nachfolger / Vorgänger (ausser Head/Tail)
- ändert sich nicht wenn neue Elemente eingefügt werden

wirft InvalidPositionException bei ungültiger Position
- p=null
- p von Liste gelöscht
- p zeigt auf andere Liste

4.8 ADT Sequenz

- Kombination aus Liste und Vektor
- allgemeinste Form eines linearen Datentyps
- erbt Interface von Liste und Vektor und zusätzlich Brückenmethoden:
atRank(r): gibt Position des Elements an Rang r zurück
rankOf(p): gibt Rang eines Elements an der Position p zurück

4.8.1 Sequenz als doppelt verkettete Liste

- Methoden können aus NodeList übernommen werden, zusätzlich noch Methoden aus Vektor und Brückenmethoden implementieren.
- hoher Aufwand durch Linkhopping in Liste bei Rang/Positionsoperationen
- Ausnahme bei Operationen auf erstem/letzten Element

4.8.2 Sequenz als Feld

Ableitung aus Vektor und zusätzlich zu implementierenden Operationen.
Nachteil: Einfügen/Verschieben kostet O(n), Positionsvariablen nicht änderbar.

4.8.3 Sequenz als Mischimplementierung

- Feld enthält Indizes auf Positionselemente
- Positionselemente zeigen auf Listenelemente

$$
\begin{array}{ccc}
\textit{Rang} & \textit{Position} & \textit{Elemente} \\
r_1 & \longrightarrow p1 & \longrightarrow E_1 \\
r_2 & \longrightarrow p2 & \longrightarrow E_2 \\
r_3 & \longrightarrow p_3 & \longrightarrow E_3 \\
r_4 & \longrightarrow p_4 & \longrightarrow E_4 \\
 & \ldots & \\
r_n & \longrightarrow p_n & \longrightarrow E_n
\end{array}
$$

4.9 ADT Iterator

Abstrahiert Konzept der Elementverfolgung (Linkhopping bzw. wandern durch Feldindex) unabhängig von der Implementierung.
- Sequenz von Elementen
- aktuelle Position
- Methode zum geordneten Betrachten der Elemente

4.10 ADT Container

Abstraktion aller ADT ausser Iterator mit:
- Anfragemethoden
- Zugangsmethoden
- Aktualisierungsmethoden
- wird über Referenzierung der Superklasse zum Inspectable Container (read-only)

5 Bäume

Menge von Knoten mit Eltern-Kind Beziehungen und folgenden Eigenschaften:

- Es gibt eine einzige Wurzel
- jeder Knoten hat einen Elternknoten
- jeder Knoten kann beliebig viele Kinder haben
- Knoten ohne Kind heisst Blatt oder externer Knoten
- Knoten mit Kind sind intern
- Knoten mit gleichem Elternknoten heissen Geschwister
- Unterbaum: alle Kindknoten und direkten und indirekten Kinder eines Knoten
- Knoten = Position, enthält Element

Darstellung geklammert:

```
A(B(CDEF)G(H))
```

```
A(
  B(
    C
    D
    E
    F
   )
  G(
    H
   )
 )
```

5.1 binärer Baum

- jeder Knoten maximal 2 Kinder
- jeder Knoten 0 oder 2 Kinder → regulärer binärer Baum

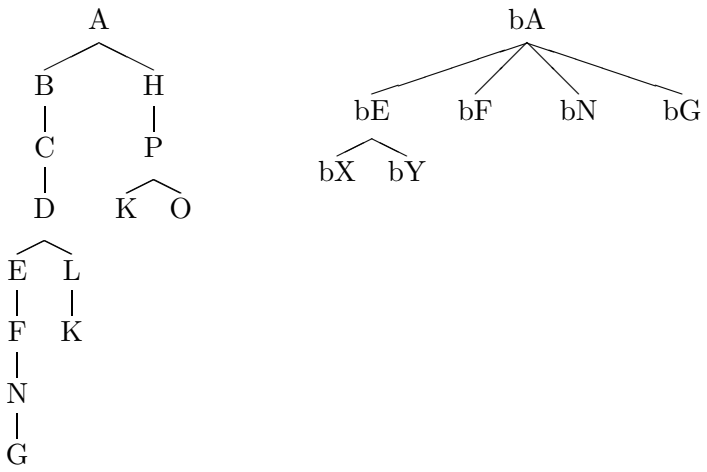
5.2 Methoden auf Baum

element()	gibt Element an Position zurück
root()	liefert Wurzelposition
parent(v)	gibt Vater zurück, ausser v=root
children(v)	Iterator mit Anzahl der Kinder
isInternal(v)	true wenn v intern
isExternal(v)	true wenn v Blatt
isRoot(v)	true wenn v root
size()	Zahl der Knoten
elements()	Iterator mit allen enthaltenen Elementen
positions()	Iterator mit Position aller Elemente
swapElements(v,w)	Elemente in v und w tauschen
replaceElements(v,e)	Element in v durch e ersetzen, v zurückgeben

5.3 Aufwand

Aufwand im schlechtesten Fall ist die Anzahl der Abstiege (Ebenen) im Baum.
Aufwand ist $O(n) = n \quad (n = Ebenen)$
Besser ist ein ausgeglichener Baum (balancierter Baum / B-Tree) mit wenigeren Ebenen

unausgeglichener Baum *balancierter Baum*



Aufwand für unbalancierten Baum im Worst Case: $O(n) = h$
Aufwand für balancierten Baum im Best Case: $O(n) = log_2 h$

5.4 Tiefe des Baumes

Tiefe = Anzahl der Vorfahren

Algorithm 4 depth()

```
Algorithmus depth(T,v)
if T.isRoot(v) then
  return 0
else
  return 1+depth(T,T.parent(v))
end if
```

5.5 Höhe des Baumes

Höhe eines Baumes = Höhe der Wurzel = maximale Tiefe aller Knoten
über Iterator Aufwand

Algorithm 5 height()

```
Algorithmus height(T)
h=0 // Höhe null
for each v in T.positions() do // durch alle Knoten rasen do
  if T.isExternal(v) then
    h=max(h.depth(T,v)) // Maximum der Tiefen suchen
  end if
end for
return h
```

Alle Knoten finden:	n	
Alle Blätter finden:	c	Worst Case: $n-1$
Tiefe für ein Blatt:	$1+d_c$	Worst Case: n
Gesamtaufwand:	$n+sum(d_c+1)$	
Worst Case:	$n+n(n-1)=O(n^2)$	
Gesamtaufwand:	$n+sum(d_c+1)$	

rekursiv

Durchläuft Baum von der Wurzel her **Aufwand:**
Jeder Knoten wird einmal als Elter einmal als Kind betrachtet = $0(n)$

5.6 Traversierung

- Betrachtung aller Knoten in vorgegebener Reihenfolge

Algorithm 6 heightRek()

```
Algorithmus heightRek(T,v)
if T.isExternal(v) then
  return 0
else
  h=0
  for each w in T.children(v) do
    h = max(h,heightRek(T,w))
  end for
end if
return 1+h
```

5.6.1 Preorder Traversierung

Vorfahren jedes Knoten werden zuerst betrachtet.

Algorithm 7 preorder()

```
Algorithmus preorder(T,v)
visit(v) // beliebige Operation auf Objekt des Knoten
for each child w of v do
  preorder(T,w)
end for
```

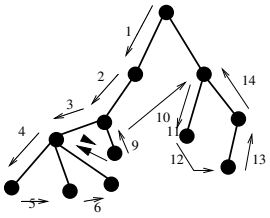


Abbildung 5.8: Prätraversierung

5.6.2 Postorder Traversierung

Kinder jedes Knoten werden zuerst betrachtet

5.6.3 Inorder Traversierung

spezielle Traversierung für binäre Bäume
linker Knoten → Wurzel → rechter Knoten

Algorithm 8 postorder()

```
Algorithmus postorder(T,v)
for each child w of v do do
  preorder(T,w)
  visit(v) // beliebige Operation auf Objekt des Knoten
end for
```

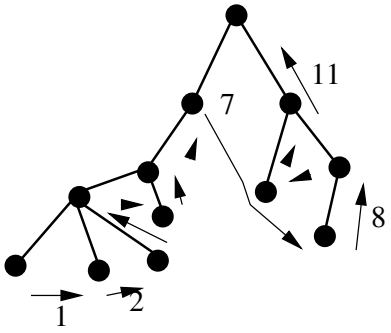


Abbildung 5.9: Posttraversierung

Algorithm 9 inorder()

```
Algorithmus inorder(T,v)
if v.isInternal() then
  inorder(T,T.leftChild())
end if
visit(v)
if if v.isInternal()  then
  inorder(T,T.rightChild())
end if
```

5.7 Binäre Suchbäume

Strukturierung der Daten in einem binären Baum

- Blätter sind leer
- rechter Knoten ist größer als Vater
- linker Knoten ist kleiner als Vater

Suche in binärem Suchbaum

- Start: Wurzelknoten
- Wenn: Knoten == Blatt $\rightarrow$ Schlüssel nicht vorhanden
- Wenn: Knoten == Schlüssel $\rightarrow$ Stop
- Wenn: Schlüssel $>$ Knoten $\rightarrow$ weiter mit rechtem Kind
- Wenn: Schlüssel $<$ Knoten $\rightarrow$ weiter mit linkem Kind

Geordnete Ausgabe aller Elemente über Inorder Traversierung.

Zeichnen eines binären Baumes:

- Inorder Traversierung
- für jeden Knoten eine X und Y Koordinate berechnen
- x == Anzahl der vorher besuchten Knoten
- y == Tiefe im Baum

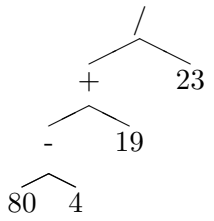
5.8 Eulertour

- Vereinigung von Inorder, Preorder und Postorder Traversierung.
- Jeder Knoten wird einmal von links, von unten und von rechts besucht, dabei können unterschiedliche Operationen (oder auch keine) ausgeführt werden.
- Blätter werden unmittelbar nacheinander dreimal besucht.

Algorithm 10 EulerTour(Tree T, node V)

```
visitFromLeft(v)
if T.isInternal(v) then
  EulerTour(T,T.leftChild(v))
end if
visitFromBelow(v)
if T.isInternal(v) then
  EulerTour(T,T.rightChild(v))
end if
visitFromRight(v)
```

Beispiel: arithmetischer Ausdruck:



$((80-4)+19)/23$

5.9 Datenstruktur für Bäume

- Referenz auf Elter
- Referenzen auf Kinder
- Inhalt als Objekt oder Referenz auf Objekt
- Operationen auf InspectableTree/InspectableBinaryTree mit konstantem Aufwand

5.9.1 Vektoren für binäre Bäume

- Knoten einer Ebene: 2^h, Alle Knoten: $2^{h+1}-1$
- Knoten erhalten eindeutige Adresse $p(v)$ in Vektor
  - Wurzel = $p(root) = 1$
  - linkes Kind: $p(u) = 2 * p(v)$ (gerade)
  - rechtes Kind: $p(u) = 2 * p(v) + 1$ (ungerade)
- Positionen = Ränge in ein als erweiterbarem Feld implementierten Vektor
- jede *mögliche* Position hat einen Rang
- Position Interface ist eine Wrapper-Class für den Rang
- Tiefe des Baumes := $log_2\ lastRank$
- root() := positionAtRank(0)

5.9.2 Speicheraufwand

- nicht alle Elemente müssen belegt sein
- Best Case: Baum ist balanciert
- Worst Case: jedes rechte Kind ist Blatt
  - jede Ebene hat zwei Knoten
  - Ränge im Vektor verdoppeln sich mit jeder Ebene
  - Speicherbedarf für n Knoten = $2^{\frac{n}{2}}$

5.9.3 binärer Baum als Zeigerstruktur

- Position relativ über Elter/Kind definiert
- jeder Knoten steht in Relation zu drei andern Knoten
- Knoten: Zeigerstruktur mit 3 Positionen und einem Objekt

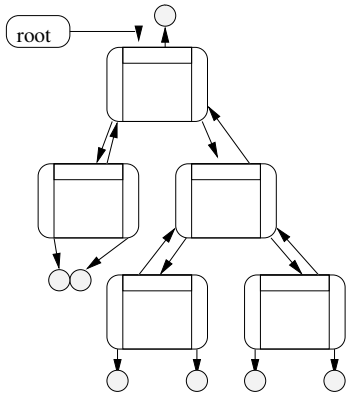


Abbildung 5.10: binärer Baum als Zeigerstruktur

Implementierung:
- Zeiger
- `root`-Knoten
- Anzahl der Felder

5.9.4 beliebiger Baum als Zeigerstruktur

Problem: Anzahl der Knoten vorher unbekannt → Kinderzeiger zeigt auf Container der Kindknoten enthält.

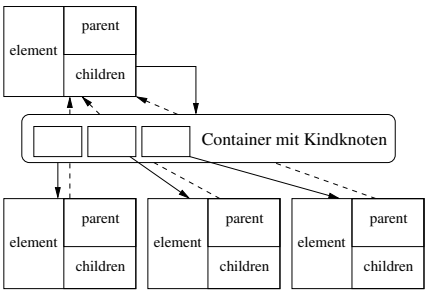


Abbildung 5.11: Baum als Zeigerstruktur

5.10 Binären Baum kopieren

5.11 Transformation in binären Baum

Transformation eines Baumes G mit g_i Knoten in einen binären Baum B mit b_j Knoten.

Algorithm 11 public class BTNode

```
public class BTNode implements Position {
    private Object element; // Element des Knotens
    private BTNode parent, left, right; // Zeiger im Baum
    public BTNode() { } // Konstruktor für leeren Knoten
    public BTNode(Object o, BTNode myParent, BTNode leftChild,
                  BTNode rightChild) {
        setElement(o); // Konstruktor mit aktuellen Parametern
        setParent(myParent);
        setLeft(leftChild);
        setRight(rightChild);
      }
   public Object element() { return element; }
        // Implementierung der Schnittstelle
   public void setElement(Object o) { element=o;}
        // zusätzliche Methoden
   public BTNode getParent() { return parent; }
        // usw. für setParent, getLeft, setLeft, getRight, setRight}
```

Algorithm 12 clone(T1,T2,v1,v2)

```
Algorithmus clone(T1,T2,v1,v2)
T2.replaceElement(v2,v1.element())
if T1.isInternal(v1) then
  t2.expandExternal(v2)
  clone(T1,T2,T1.righChild(v1),T2.leftChild(v2))
  clone(T1,T2,T1.righChild(v1),T2.rightChild(v2))
end if
```

- Geschwister in $G \rightarrow$ rechte Kinder in B
- Kinder in $G \rightarrow$ linke Kinder in B
- jeder g_i in $G \leftrightarrow b_j$ in B
- Regularisierung mit $NULL$-Knoten

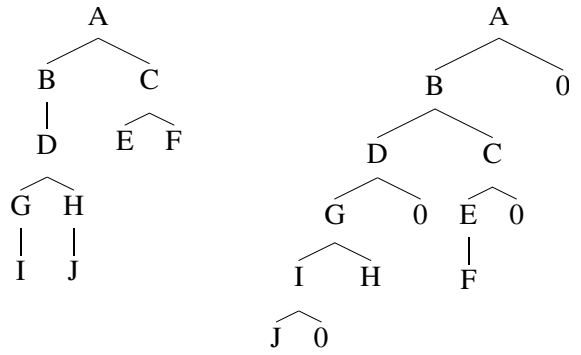


Abbildung 5.12: Transformation in binären Baum

5.12 binärer Baum als Vektor

Position der Wurzel 1, linkes Kind von m ist $2m$, rechtes Kind ist $2m + 1$, Vektorlänge ist $2 \sup h + 1$

5.13 AVL-Baum

AVL-Bäume sind durch Einhaltung des AVL-Kriteriums immer ausgeglichen.
AVL-Kriterium: Der Höhenunterschied des linken und rechten Teilbaumes eines Knoten ist maximal 1
Dauraus ergibt sich eine Höhe von $h = log\ n$
Einfügen und Löschen können vom binären Baum übernommen werden, Löschen und Einfügen kann Restrukturierung durch (Doppel-)Rotation erfordern.
Beispiel:

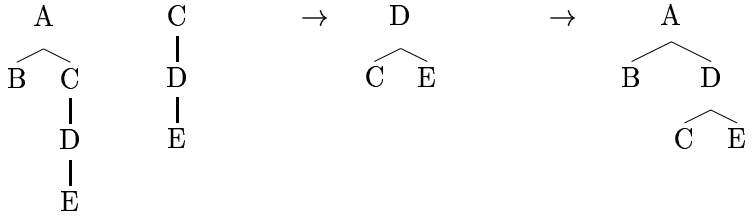


Abbildung 5.13: Restrukturierung eines Nicht-AVL Baumes

Der Höhenunterschied zwischen E und B beträgt 2, somit ist das AVL-Kriterium verletzt. Die Wurzel des entarteten Teilbaumes ist C, also müssen die Knoten C-D-E einfach linksrotiert werden, danach ist der Baum wieder AVL.
Beim Löschen im AVL-Baum wird der entsprechende Knoten gesucht und entfernt, danach wird von dieser Stelle aus bis zur Wurzel hoch ausgeglichen.

5.14 (2-4)-Baum

Jeder Knoten hat 1, 2 oder 3 Schlüssel und 2, 3 oder 4 Kinder. Externe Knoten haben gleiche Tiefe

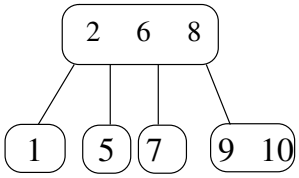


Abbildung 5.14: Beispiel (2-4)-Baum

Einfügen:
Wenn 4 Schlüssel in einem Knoten sind wird der mittlere (also der 2. oder 3.) Schlüssel hochgezogen und neuer Vater des alten Knoten. Dies wird gegebenenfalls zur Wurzel hoch fortgesetzt, d.h. der Baum wächst zur Wurzel hin.
Beispiel:

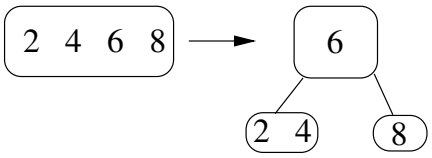


Abbildung 5.15: Split eines (2-4)-Knotens

Löschen:
- nur externe Knoten werden geändert
- gesuchter Schlüssel nicht extern → mit rechtestem Kind tauschen
- Restrukturieren

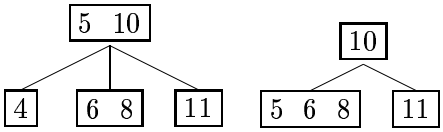


Abbildung 5.16: Schlüssellöschen im (2-4)-Baum

5.15 Rot-Schwarz Baum

Binäre Darstellung eines (2-4)-Baums mit zusätzlichen Farbinformationen [1] (rot/schwarz) in jedem Knoten.

- Wurzel → schwarz
- Blätter → schwarz
- Kinder eines Rotknotens → schwarz
- schwarz → Master
- rot → slave
- 2-Knoten → schwarz
- 3-Knoten → schwarz mit einem roten Kind
- 4-Knoten → schwarz mit zwei roten Kindern
- Schwarztiefe → für alle Blätter gleich

Höhe ist $O(log\ n)$ und maximal doppelt so hoch wie die des entsprechenden (2-4)-Baumes. Einfügen erfolgt als Blatt mit anschließender Restaurierung der Farbeigen-

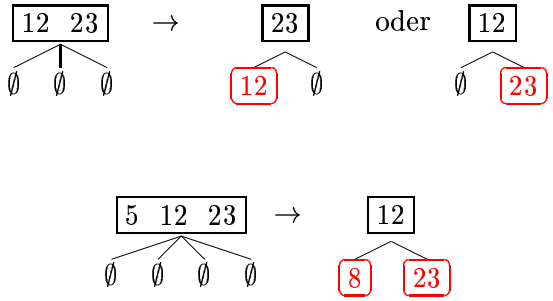


Abbildung 5.17: Schwarz-Rot Präsentation eines (2-4)-Baumes

schaften.
Wenn einzufügender Knoten nicht die Wurzel ist als rotes Blatt einfügen (Wurzel-, Tiefen- und Blättereigenschaft bleiben erhalten), es kann aber Doppelrot auftreten.
Restrukturierung
Onkel ist schwarz, entspricht falschem Einfügen in (2-4)-Baum, siehe Abbildung 5.18.

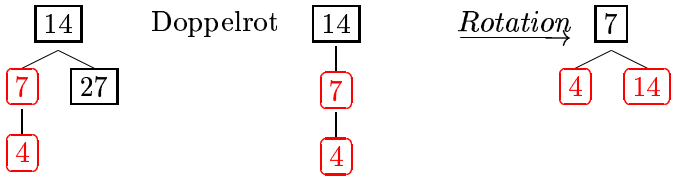


Abbildung 5.18: Rotation im Rot-Schwarz-Baum

Der mittlere Schlüssel eines 4er Knotens ist schwarz, Rotation durchführen

Onkel ist rot, entspricht Schlüsselüberlauf im (2-4)-Baum
Großvater des eingefügten Knoten ist `root`, kann daher nicht umgefärbt werden. Vater und Onkel werden schwarz umgefärbt, eingefügter Knoten bleibt rot, siehe Abildung 5.19.

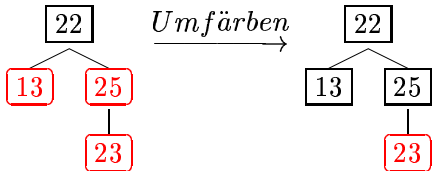


Abbildung 5.19: Umfärben 1 im Rot-Schwarz-Baum

Großvater ist nicht `root`, eingefügter Knoten und Großvater sind/werden rot, Vater und Onkel werden schwarz, siehe Abildung 5.20. Umfärbung wird zur Wurzel hin propagiert.

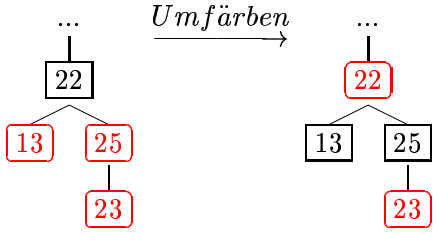


Abbildung 5.20: Umfärben 2 im Rot-Schwarz-Baum

[1]Der Druckbarkeit wegen seien hier eckige Boxen schwarze und abgerundete Boxen rote Knoten

6 Graphen

6.1 Darstellung

Repräsentiert mehrdimensionale Zusammenhänge, besteht aus zwei endlichen Mengen von Kanten E (Edges) und Knoten V (Vertices). Wird definiert über alle Knoten und Kanten, z.B.:

$$G_\alpha \ = \ \{V_\alpha, E_\alpha\} \ mit \ V_\alpha = \{1,2,3\}, \ E_\alpha = \{(1,2),(1,3),(3,2)\}$$

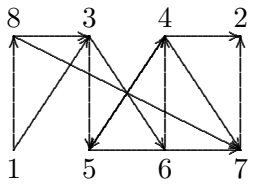
Ein Pfad ist eine Folge von Kanten wobei jede Kante zwei Knoten verbindet, der Graph ist zusammenhängend wenn für jedes Knotenpaar ein Pfad existiert.
Gerichtete Graphen sind streng zusammenhängend falls für beliebige Knoten u und v gilt das u von v erreichbar ist.
u ist von v erreichbar falls eon Pfad von u nach v existiert, Erreichbarkeit ist im gerichteten Graphen nicht kommutativ.
Ein (gerichteter) Zyklus ist ein (gerichteter) Pfad dessen Anfangs und Endknoten gleich ist.
Eine Transitive Hülle G^* ist ein Graph der alle Knoten und Kanten von G und zusätzlich Kanten zwischen Knoten enthält die per einem Pfad verbunden sind. Die Transitive Hülle zeigt Abhängigkeiten in der Prozessfolge und der Eingrad eines Knoten gibt die Anzahl der zu erfüllenden Vorbedingungen an.
Die Topologische Sortierung entfernt Zyklen aus der Prozessfolge, hält dabei aber die Prozessreihenfolge ein.

6.2 Interface

numVertices() numEdges()	Gibt die Anzahl der Knoten bzw. Kanten zurück
vertices() edges()	Gibt je einen Iterator mit Knoten bzw. Kanten zurück
aVertex()	Liefert einen beliebigen Knoten zurück
degree(v)	Grad des Knotens v
adjacentVertices(v)	Liefert Iterator mit zu v benachbarten Knoten
incidentEdges(v)	Iterator mit inzidenten Kanten
endVertices(e)	Iterator mit den beiden Knoten an e
opposite(v,e)	an e zu v gegenüberliegende Knoten
areAdjacent(v,w)	*true* wenn v und w durch Kante verbunden
insertEdge(v,w,o)	neue Kante zwischen v und w einfügen, Objekt o in der Kante speichern
insertVertex(o)	isolierten Knoten generieren und Objekt o inKnoten speichern
removeVertex(v)	Knoten v und alle von ihm ausgehenden Kanten entfernen
removeEdge(e)	Kante entfernen
directedEdges() undirectedEdges()	Iteratoren mit den gerichteten/ungerichteten Kanten
destination(e) origin(e)	Knoten in dem die Kante e endet bzw. beginnt
isDirected(e)	true wenn e gerichtet
inDegree(v) outDegree(v)	Eingrad und Ausgrad
inIncidentEdges(v) outIncidentEdges(v)	Ein- und ausgehende Kanten von Knoten v
inAdjacentVertices(v) outAdjacentVertices(v)	Iteratoren mit Knoten, deren Kanten nach v zeigen bzw. auf die Kanten von v zeigen
insertDirectedEdge(v,w,o)	Einfügen und Rückgabe einer gerichteten Kante von v nach w. Das Objekt o wird in Kante eingetragen
makeUndirected(e)	macht aus der gerichteten Kante e eine ungerichtete Kante
reverseDirection(e)	Umkehrung der Richtung von e
setDirectionFrom(e,v)	macht aus der zu v inzidenten Kante e eine gerichtete Kante, die von v weg zeigt
setDirectionTo(e,v)	macht aus der zu v inzidenten Kante e eine gerichtete Kante, die zu v hin zeigt

6.3 Datenstruktur

Graph



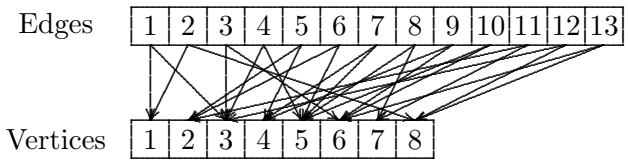
Adjazenzmatrix

	1	2	3	4	5	6	7	8
1	0	0	1	0	0	0	0	1
2	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0
4	0	1	0	0	0	0	1	0
5	0	0	0	1	0	0	1	0
6	0	0	0	1	1	0	0	0
7	0	1	0	0	0	0	0	0
8	0	0	1	0	0	1	0	0

Kantenliste

Kante besteht aus:

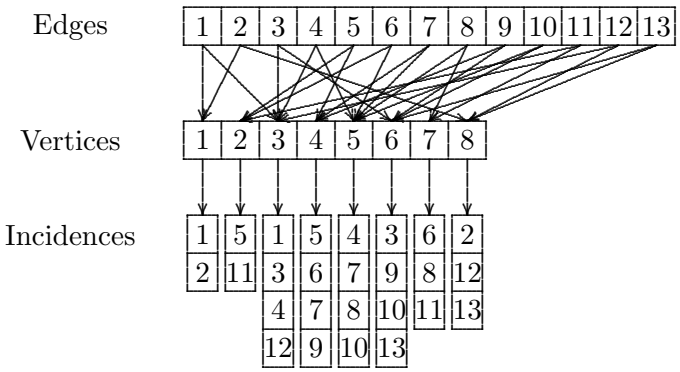
- Objekt
- Boolean ob gerichtet oder ungerichtet
- Referenz zu beiden inzidenten Knoten
- Positionsreferenz im Kantencontainer



Adjazenzliste

Kante besteht aus:

- Objekt
- Boolean ob gerichtet oder ungerichtet
- Referenz zu beiden inzidenten Knoten
- Positionsreferenz im Kantencontainer
- Speicherung der inzidenten Kanten
- gerichteter Graph enthält ein- und ausgehende Kanten getrennt



6.4 Traversierung von Graphen

Es gibt zwei Arten der Traversierung, die Tiefensuche (depth-first search, DFS) und die Breitensuche (breadth-first search, BFS).
Tiefensuche:

- Startpunkt → beliebiger Knoten
- suche vom aktuellen Knoten aus
  - beliebigen Nachbarknoten
  - falls bereits traversiert → anderer Knoten
  - falls noch nicht traversiert → neuer Knoten = aktueller Knoten
- wenn alle Nachbarknoten traversiert wurden → zurück zum Vorgänger

DFS besucht jeden Knoten eines zusammenhängenden Graphen, d.h. sind nach der Traversierung noch Knoten übrig ist der Graph nicht zusammenhängend.
Der Aufwand für n Knoten und m Kanten ist $O(m+n)$.
Entdeckungskanten formen einen tiefen Spanning Tree, also eine Baumrepräsentation eines Graphen, es existieren somit darin auch keine Zyklen.
Breitensuche:

- Startpunkt → beliebiger Knoten
- suche vom aktuellen Knoten aus
  - beliebigen Nachbarknoten
  - kehre zurück zum Startknoten
  - traversiere beliebigen nächsten Nachbar
  - wenn alle Nachbarn traversiert
  - wähle einen beliebigen traversierten Nachbar als neuen Knoten

Erzeugt einen breiten Spanning Tree, der nach Ebenen geordnet ist. BFS prüft ob der Graph zusammenhängend ist, findet alle Subgraphen, alle Zyklen, kürzesten Weg zwischen zwei Knoten, sofern vorhanden.
Aufwand für BFS ist $O(m+n)$

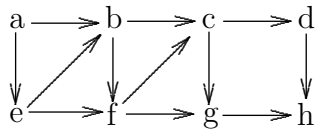


Abbildung 6.21: Beispielgraph

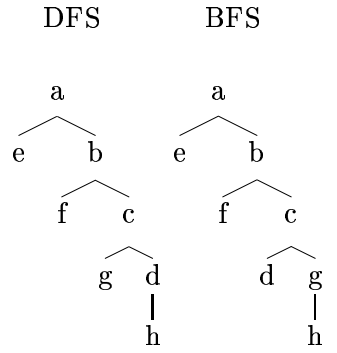


Abbildung 6.22: Beispielgraph

7 Lexikon

Bildet Paare aus Schlüssel und Elementen, Schlüssel und Element sind Objekte.
Simuliert ein Lexikon über Einfügen, Löschen und Suchen von Elementen via zugehörigem Schlüssel.
Ungeordnetes Lexikon:
Einträge sind ungeordnet
schnelles Einfügen
Schlüssel müssen vergleichbar sein
Geordnetes Lexikon:
Einträge sind geordnet nach Schlüsseln
schnelle Suche
Vergleichsmöglichkeit notwendig (Comparator)
Existieren mehrere gleiche Schlüssel wird irgendeiner zurückgegeben.
`NO_SUCH_KEY` ist spezielles Element das anstatt einer Exception zurückgegeben wird wenn ein Schlüssel nicht existiert.
Datenstrukturen für ungeordnetes Lexikon:
Liste, Vektor oder Sequenz

7.1 Hashtabellen

Dient der Beschleunigung des Suchens.
Hashtabelle besteht aus:
Bucket-Array: enthält Schlüssel-Element Paar gleichmäßig verteilt
Hashfunktion: bildet Schlüssel auf Bucket ab

7.1.1 Hashfunktion

Zweistufige Funktion h die jeden Schlüssel k auf eine Ganzzahl zwischen 0 und $n-1$ abbildet.
Erste Stufe: Hash-Code bildet Schlüssel auf Integer ab
Zweite Stufe: Hash-Code wird auf Bereich 0 bis $n-1$ abgebildet
Aufwand zur Berechnung sollte gering und leicht nachvollziehbar sein.
Kollisionen (mehrere gleiche Hashcodes für unterschiedliche Elemente) solten vermieden werden, ohne Kollisionen ist der Aufwand für Zugriff $O(1)$.
Kompressionsverfahren reduzieren den Definitionsbereich des Hashcodes aud den der Hashtabelle.
Divisionsverfahren:

$$h(k) = k \ mod \ n$$

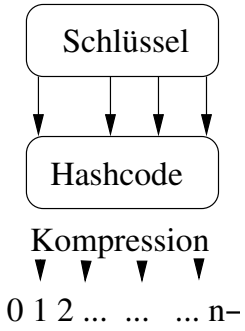


Abbildung 7.23: Prinzip der Hashfunktion

n sollte Prim sein um wiederkehrende Muster unterschiedliche abzubilden.
Besipiel

$$k = (105, 110, 115, \ ... \ 205, 210, 215)$$

$$n = 100 \ \rightarrow \ h(k) = (5, 10, 15, \ ... \ 5, 10, 15)$$

$$n = 101 \ \rightarrow \ h(k) = (4, 9, 14, \ ... \ 3, 8, 13)$$

MAD
Multiply, add, divide

$h(k) = |ak+b| \ mod \, n \ mit \ n = Tabellengrösse \ (Prim); \ a, b = Zufallszahlen \ mit \ a \ mod \, n \neq 0$

7.2 Beispielcodes

Speicherort:
Java Object-Klasse hat Methode `hashCode()` die einen 32-Bit Integer (Speicherort) zurückliefert.
Summation:
Objekte werden durch Integer dargestellt und aufsummiert. Beispielsweise werden Buchstaben durch ihren Platz im Alphabet repräsentiert und in Strings die Summe der Buchstaben gebildet. Nachteil sind gleiche Hashcodes für Strings mit gleichen Buchstaben aber unterschiedlicher Ordnung: $h(asdf) = h(fdsa) = h(dasf)$
Polynom:
Polynomielle Wichtung der Ordnung, z. B.

$$h(x_0, x_1 ... x_{k1}) \ = \ x_0 a^{k-1} + x_1 a^{k-2} + x_2 a^{k-3} + x_{k-2} a + x_{k-1} \ mit \ a > 1$$

$h(asdf) \neq h(fdsa) \neq h(dasf)$
Berechnung erfolgt mittels Hornerschema.
Shift:
logisches Shift
Siehe Algorithmus `Shift()`

Algorithm 13 Shift()

```
static int Shift(String S)
{
    int h=0;
    for(int i=0;i<s.length();i++)
    {
        h=(h<<5 | h>>27);
        h+=(int)s.charAt(i);
    }
}
```

7.3 Kollisionsverarbeitung

Kollisionen entstehen, wenn zwei Elemente mit Schlüsseln k_1 und k_2 auf den gleichen Hashcode abgebildet werden, also: $h(k_1) = h(k_2)$
Problem:
Mehrere Lexikoneinträge auf selbem Tabellenplatz, d. h. Suche nach Einträgen wird erschwert.
Der Ladefaktor ist das Verhältnis $\frac{n}{N}$ zwischen Anzahl der Lexikoneinträge und Anzahl der Plätze in der Hashtabelle.
Eine Hashfunktion ist gut wenn die Anzahl von Einträgen jedes Tabellenplatzes nahe am Ladefaktor L liegt. Für die ideale Hashfunktion ist der Aufwand für die Such- und Löschoperationen $O(L)$ und $O(1)$, falls $L < 1$ ist.
Seperate Chaining
Jeder Tabelleneintrag verweist auf ein eigenes Lexikon (Liste, Vektor oder Sequenz).
Ein Lexikon unter dem Tabelleneintrag i enthält nur diejenigen Items (k, e) mit Schlüsseln k, so dass $h(k) = i$ ist.
Open Addressing
Nie mehr als ein Element pro Behälter
Benötigt Algorithmus um freien Platz zu finden falls Ursprungsplatz schon belegt ist
Lineares Testen
Einfügen:
Von ursprünglichem Einfügeplatz den nächsten suchen der leer ist
Suchen:
Von ursprünglichem Einfügeplatz das Element suchen oder abbrechen wenn ein leerer Platz gefunden wurde
Löschen:
Zustand vor Einfügen wiederherstellen oder Element deaktivieren, also beim Suchen überspringen und beim Einfügen überschreiben
Lineares Testen führt zu Häufungen, wenn viele Schlüssel auf gleiche Tabelleneinträge abgebildet werden, erhöht so den Suchaufwand
zu hoher Ladefakter ($L \geq 1$) erhöht Aufwand

Quadratisches Testen

$$i \ = \ i + j^2 \ mod \ N, \ \ für \ j = 0, 1, 2, ...$$

verhindert Häufungen, freie Plätze werden möglicherweise übersprungen besonders, wenn N keine Primzahl ist

Double Hashing

Anwenden einer zweiten Hash-Funktion $h'()$
wenn $i = h(k)$ befüllt ist teste Behälter an $i + f(j) \ mod \ N \ mit \ f(j) = j * h'(k) \ h'() \neq 0$
gute Hash-Funktion, Beispiel: $h(k) = q - (k \ mod \ q), \ q < N, \ q, N \ sind \ Primzahlen$

Re-Hashing

Anzahl der Einträge wächst mit jedem neuen Element, wird der Ladefaktor zu groß, erstelle größere Hashtabelle, bilde neue Hashfunktion und sortiere alte Tabelle in Neue um

7.4 Geordnete Lexika

Aufwändiges Einfügen, schnelles Suchen
erfordert Ordnungsrelation
Implementierung über Hash-Tabelle ist wenig effizient

Look-Up Table

Nach Schlüsseln geordneter Vektor von Elementen
Rang des Elements ermöglicht Zugriff
Nächstes / Voriges Element kann mit $O(1)$ gefunden werden
Speicherplatz ist n bei n Elementen

Einfügen

Suche Schlüssel $k1 \ mit \ k1 > k: \ O(n)$
Füge Element vor $k1$ ein
Aufwand $O(n)$

Löschen Suche ersten Schlüssel k mit $O(n)$
Lösche Items mit Schlüssel k mit $O(n)$
Aufwand $O(n)$

7.5 Sprunglisten

zweidimensionale Datenstruktur mit Einfüge-/Suchaufwand von $O(\log \ n)$

- Folge von geordneten Listen $S_0, S_1, ..., S_h$
- zwei Zusatzelementen $-\infty, \ +\infty$
- Liste S_0 enthält alle Elemente des Lexikons plus $-\infty, \ +\infty$
- Für $i = 1, ..., h-1$ enthält jede Liste S_i eine zufällig gewählte echte Teilmenge der Elemente von Liste $Si-1$

- Liste S_h enthält nur $-\infty,\ +\infty$

Höhe 4	$-\infty$									$+\infty$
Höhe 3	$-\infty$								19	$+\infty$
Höhe 2	$-\infty$		4						19	$+\infty$
Höhe 1	$-\infty$	1	4		9		15		19	$+\infty$
Höhe 0	$-\infty$	1	4	7	9	11	15	18	19	$+\infty$

Erzeugung der Liste:

- Liste S_i wird aus S_{i-1} erzeugt
- für jedes Element eine Pseudozufallszahl erzeugen
- Element wird mit ca. 50%-iger Wahrscheinlichkeit nach S_i übernommen
- $S_i - 1$ enthält ca. 50% der Elemente aus S_i

Löschen
Element suchen und aus der Liste und allen darüberliegenden Liste austragen.
Realisierung:
Folge von doppelt verketteten Listen, die an den Towers zwischen den Ebenen zusätzlich verkettet sind.
Aufwand für `before()`, `after()`, `above()`, `below()` ist $O(1)$

8 Sortieren und Suchen

8.1 Stabilitiät

Ein Sortierverfahren ist stabil wenn nach dem Sortieren vorsortierter Daten die Vorsortierung erhalten bleibt.
(Bsp: Lagerbestände sind nach Verfallsdatum sortiert und sollen nun nach Preis umsortiert werden. Ein stabiles Sortierverfahren führt dazu das Bestände mit gleichem Preis auch weiterhin nach Verfallsdatum sortiert bleiben.)

8.2 Bubble Sort

In einer Schleife wird durch die Elemente gegangen, wenn 2 Nachbarelement unterschiedlich (im Sortierschlüssel) sind, werden die Positionen der Elemente ausgetauscht. Die Äussere Schleife beendet das Verfahren wenn keine Elemente mehr getauscht wurden.
Da nur Elemente mit unterschiedlichen Schlüsseln getauscht werden, bleibt die Vorsortierung erhalten.
Der Aufwand ist quadratisch, da zwei ineinander verschachtelte Operationen verwendet werden (Lauf durch alle Elemente und Vergleich mit Tausch)

$$O(n) = n^2$$

8.3 Selection Sort

Erstellt eine neue Liste in dem immer das kleinste Element aus der Vaterliste an die neue Liste angefügt wird, dabei kann das Verfahren instabil sein. Stabilität hängt ab vom Suchverfahren und Einfügen in die neue Liste
Aufwand: n Suchschritte (abhängig von Listenlänge, daher abnehmend), n Umsortierschritte:

$$O(n) = n * n = n^2$$

8.4 Binary Search

Suche in vorsortiertem Feld nach Divide & Conquer Prinzip. Feld wird geteilt und untere Grenze mit zu suchendem Element verglichen. Wenn die Grenze kleiner als das Element ist, wird im Teil mit den größeren elementen nach gleicher methode weitergesucht und geteilt. abbruch wenn sich obere und untere liste treffen, aufwand ist damit

$$o(n) = logn$$

8.5 Merge Sort

Sortierverfahren nach Divide-and-Conquer-Prinzip.
Eingabe: zu sortierende Sequenz

Divideschritt: Zerlegung der Eingabesequenz in zwei Teilsequenzen
Lösung: Trivial bei einem oder keinem Element
Conquer Zusammenführung unter Erhaltung der Sortierreihenfolge

9 Mustererkennung

9.1 Brute Force

Bei der Brute Force Methode werden die jeweiligen Buchstaben aus dem Text und dem Pattern nacheinander verglichen. Bei einem Fehler wird das Pattern um einen Buchstaben weiter nach rechts verschoben und wieder von vorne buchstabenweise mit dem Text verglichen.
Aufwand im schlechtesten fall ist $O(nm)$

9.2 Boyer Moore

Zuerst wird eine Last-Occurence-Table erstellt, in der die letzte Position (Position beginnt bei 0) der einzelnen Buchstaben des Pattern notiert werden. Buchstaben die im Alphabet, aber nicht im Pattern vorkommen werden mit -1 aufgenommen.
Das Pattern wird nun von hinten beginnend mit dem Text verglichen, tritt ein Fehler auf, wird das Pattern um $Patternlänge - Last\ Occurence\ des\ Fehlbuchstabens$ verschoben. Tritt der Fehler an einem Buchstaben auf, der nicht im Pattern vorkommt (also den Wert -1 hat) wird das Patttern hinter den Bad Character verschoben.
Aufwand ist im worst case $O(n * m + s)$ mit s = Anzahl der Buchstaben im Alphabet.

Beispiel:
Text: ABABABBABADBABABBABAAA
Pattern: BABBA
Last-Occurence:

1	2	3
A	B	D
4	3	-1

Verschieben

1)	A	B	A	B	A	B	B	A	$BADBABABBABAAA$
	B	A	B	B	A				
2)	A	B	A	B	A	B	B	A	$BADBABABBABAAA$
			B	A	B	B	A		
3)	A	B	A	B	A	B	B	A	$BADBABABBABAAA$
				B	A	B	B	A	

1. Schritt: Fehler in B, verschiebe um $5 - 3 = 2$
2. Schritt: Fehler in A, verschiebe um $5 - 4 = 1$
3. Schritt: keine Fehler, Treffer, gib Anfangsposition des Pattern zurück

9.3 Knuth Morris Pratt

Bestimme zuerst den Rand des Pattern von einem bis alle Buchstaben des Pattern.
Der Rand sind Übereinstimmungen im Präfix und Suffix eines Pattern.

$\underline{B}ABBA \quad Rand = 0$

$\underline{BA}BBA \quad Rand = 0$

$\underline{\overline{B}A\overline{B}}BA \quad Rand = 1$

$\underline{\overline{B}AB\overline{B}}A \quad Rand = 1$

$\underline{\overline{B}AB\overline{BA}} \quad Rand = 2$

Das Pattern wird von vorne mit dem Text verglichen, bei Fehlern wird um $1 + Rand[Fehlbuchstabe]$ verschoben.

A	B	A	B	A	B	B	A	B	A	D	B	A	B	A	B	B	A	B	A	A	A
B	A	B	B	A																	
A	B	A	B	A	B	B	A	B	A	D	B	A	B	A	B	B	A	B	A	A	A
	B	A	B	B	A																
A	B	A	B	A	B	B	A	B	A	D	B	A	B	A	B	B	A	B	A	A	A
			B	A	B	B	A														

1. Schritt: Fehler im nullten Buchstabe → Verschiebe um 1 + Rand[0] = 1
2. Schritt: Fehler im dritten Buchstabe → Verschiebe um 1 + Rand[3] = 2
3. Schritt: kein Fehler, Treffer

Aufwand ist $O(m + n)$

10 Exceptions

Exceptions sind unerwartete Ereignisse während der Programmausführung.
Exception Handling
ist die Reaktion mit der das Programm kontrolliert weiterarbeitet oder beendet wird.
Runtime Exeption Handling
Ausnahmen werden in der Laufzeit des Programms abgefangen, d. h. der Compiler muss von den Exceptions Kenntnis haben und die Ausnahmebehandlungen sowie Tests dazu müssen spezifiziert werden.
Exceptions werden ohne 'catch' an die aufrufende Methode weitergreicht.
zu werfende (throw) Exceptions werden in der Methodendeklaration spezifiziert.

10.1 Die Klasse Throwable

```
Throwable
/       \
Error    Exception
         |
         Runtime Exception
```

Eigene Exceptions, die nicht aus RuntimeException abgeleitet sind, müssen in der throws-Anweisung deklariert werden.

11 Entwurfsmuster

Es gibt mehrere Entwurfsmuster um Probleme algorithmisch zu lösen:

11.1 Adaption

Das Problem wird anhand der Adaption anderer Lösungen modelliert
- hilft semantische Fehler zu vermeiden
- gegenwärtige Implementierung erbt von anderen Implementierungen
- andere Lösungen werden spezialisiert auf das Problem übertragen

11.2 Divide and Conquer

Das Gesamtproblem wird durch kleinere Teilllösungen gelöst

- Problem in Teilprobleme zerlegen
  - Teilprobleme lösen
  - Teillösungen zusammensetzen um Anfangsproblem zu lösen
  - ermöglicht polynomiale Lösung für exponentielle Anzahl an Lösungsmöglichkeiten
- Teilprobleme so oft aufsplitten bis diese explizit lösbar sind
- Teilprobleme sollten von der selben Art wie andere Teilprobleme sein

11.3 Brute Force

- es werden alle möglichen Lösungen durchprobiert und die beste ausgewählt
- Voraussetzung: Alternativen sind endlich abzählbar

11.4 Greedy

- es wird der momentan beste Algorithmus angewendet
- vereinfacht Entscheidungsprinzipien, dabei aber nicht zwangsläufig die Lösung

11.5 Dynamic Programming

- Teilllösungen werden als Netzwerk zusammengefasst
- Qualität der Gesamtlösung wird bei Veränderungen iterativ akualisiert

12 Entwurfstechniken

Dienen dem kontrollierten Umgang mit:

- Problemstellung
- Problemabstraktion
- Lösungsstrategien
- Verifizierung
- Validierung
- Evaluierung
- Kommunikation
- Dokumentation

Phasenkonzept

- Pflichtenheft
  - Marktanalyse
  - Präzisierung des Entwicklungsziels
- Entwurf
  - Grobstruktur des Systems
  - Zerlegung in Komponenten
- Implementierung
  - Umsetzung in Programm
- Komponentest
  - Testen der einzelnen Komponenten
  - Testen des Gesamtsystems
- Abnahme
  - Übergabe
  - Testen unter Realbedingungen
- Vorschläge für nächste Version

12.1 Zerlegungskonzept

- Schrittweise Verfeinerung
  - Begin mit Gesamtproblem
  - Bestimmen der Aufgaben
  - Grobbeschreibung der einzelnen Prozesse
  - Detaillierung der Schritte
- Prozeduralisierung
- Modularisierung
- Objekte und Klassen
- Generizität

12.2 Überprüfung von Algorithmen

Die Überprüfung der Algorithmen kann unterschiedliche Ergebnisse liefern und teilweise ist auch nur der Einsatz bestimmter Überprüfungsmechanismen möglich. Die Überprüfung erfolt dabei immer bezogen auf die Spezifikation. Die Ziele dabei sind:

- Syntaktische Korrektheit
  - korrekte Grammatik ('Compilercheck')
- Semantische Korrektheit
  - intendierte Bedeutung wird vom Programm erfüllt
    - Partielle Korrektheit:
      - erzeugt korrektes Ergebnis *wenn* terminiert
    - Totale Korrektheit:
      - erzeugt korrektes Ergebnis *und* terminiert

`Verifikation`[2]
Formaler Beweis der Korrektheit bezüglich einer formalen Spezifikation.
`Validierung`
Plausibilisierung der Korrektheit einer formal oder informell spezifiezierten Software (zum Beispiel durch systematisches Testen).

12.2.1 Validierung

- Validierung durch Testen: Es wird nachgewiesen, dass das Programm für die angegebenen Eingaben korrekt arbeitet.
- Validierung ist dann eine Verifikation, falls das Programm für alle spezifizierten Eingaben getestet wurde.
- Falls die Anzahl der möglichen Eingaben sehr groß oder unendlich ist, ist eine Verifikation durch Validierung unpraktikabel oder unmöglich.
- Die Wahrscheinlichkeit, dass ein validiertes Programm korrekt ist, steigt mit der Anzahl der Testfälle, falls sie repräsentativ für den Definitionsbereich sind.

12.2.2 Verifikation

- Korrektheit durch Verifikation: ein Algorithmus $ALG(a)$ liefert für alle Eingaben a des Definitionsbereichs D das spezifizierte Ergebnis (falls er terminiert).
- Für imperative Algorithmen:
- Erwartetes Ergebnis wird durch formale Parameter beschrieben (z.B. $r * r = afuerr = SQRT(a)$)
- Zustandsänderungen werden auf den formalen Parameter angewandt.
- Es wird geprüft, ob das Ergebnis erreicht wird.

[2] Hierzu sei ein Zitat von `Donald E. Knuth` angemerkt:
Beware of bugs in the above code; I have only proved it correct, not tried it.

12.2.3 Spezifikation

Die Spezifikation ist das eindeutige Festlegen des Verhaltens der Software sowie der Eingabe, Ausgabe, deren Abhängigkeit voneinander und den benutzten Variablen.

12.2.4 Backus-Naur-Form

Standardisierte Form der Syntaxbeschreibung aus Variablen $< v >$, Konstanten k und Ersetzungsregeln :=
Beispiele:

```
<Buchstabe>      ::= a|b|c|...|z|A|B|...|Z
<Zeichenkette>   ::= <Buchstabe>|<Ziffer>|<Unterstrich>|
       <Buchstabe><Zeichenkette>|<Ziffer><Zeichenkette>|
                              <Unterstrich><Zeichenkette>
<Variablenname>  ::= <Buchstabe>|<Buchstabe><Zeichenkette>
```

12.3 Visualisierung von Algorithmen

Die Visualisierung dient der besseren Verständlichkeit von Algorithmen. Es existieren verschiedene Standards, unter anderem Struktogramme oder Programmablaufpläne.

Diagram — Beispiel

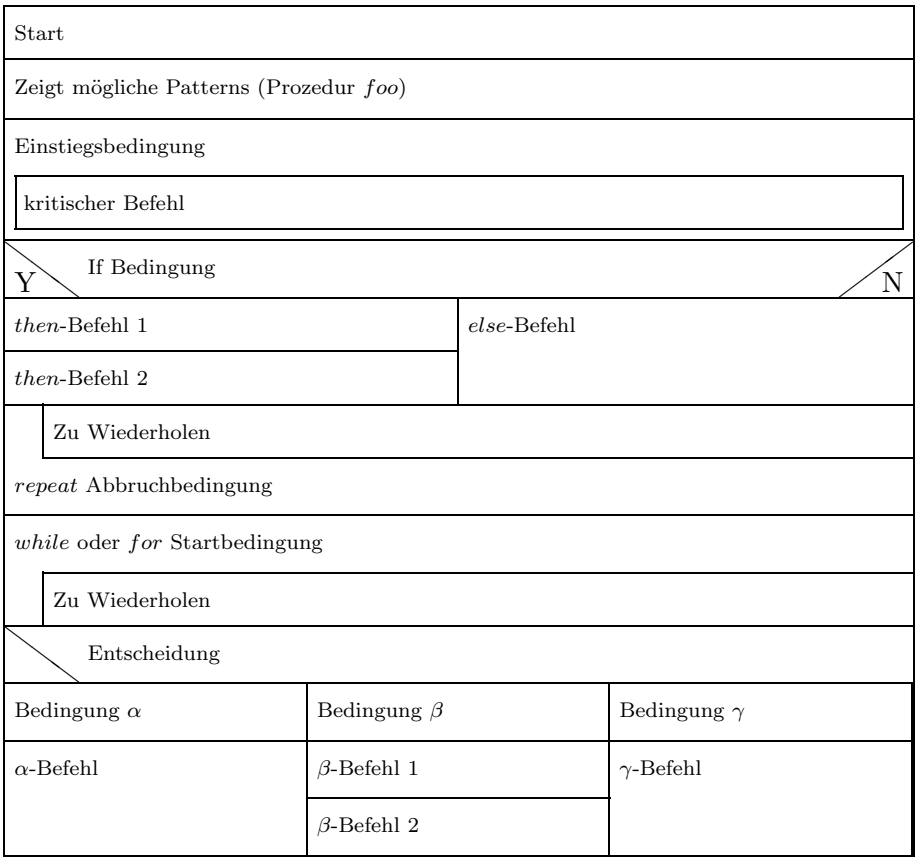


Abbildung 12.24: Struktogramm / Nassi Schneidermann Diagram

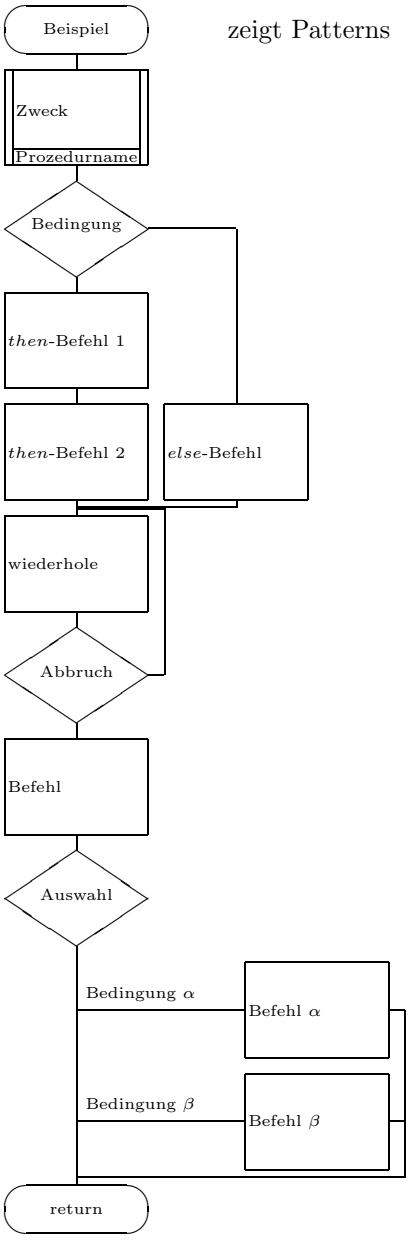


Abbildung 12.25: Programmablaufplan

12.4 Pseudocode

Algorithm 14 pseudocode-pattern

```
Algorithmus voodoo(t,v)
Input: something, black cock
Output: solved problem

if foo = bar then
  do something
  variable ← 23
end if
while n < 42 do
  for n > 23 do
    voodoo()
  end for
end while
```

12.5 Prioritätsschlange

Ordnet die Bearbeitungsreihenfolge nach Nettigkeit. Priorität wird dabei aus einem Schlüssel berechnet, der alle Eigenschaften des Objekts, die die Priorisierung bestimmen, berücksichtigt.

12.5.1 Kompositionsobjekte

Objekte bestehen aus mindestens zwei Entitäten, Schlüssel und Element. Zugriff auf einzelne Elemente wird von Kompisitionselementen zugelassen.

12.5.2 Comparator

Zum Vergleichen der Schlüssel muss eine entsprechende Totalordnung definiert werden.

Eigenschaften der Totalordnung:

- Transitiv $k_1 \circ k_2 \wedge k_2 \ \circ \ k_3 \ \leftrightarrow k_1 \circ k_3 \forall \ k_1, \ k_2, \ k_3 \in \ K$
- Reflexiv $k \circ k \ \forall \ k \ \in \ K$
- Antisymmetrisch $k_1 \circ k_2 \wedge k_2 \circ \ k_1 \leftrightarrow k_1 = k_2 \ \forall \ k_1, \ k_2 \ \in \ K$

Jede Abfrage muss nach dieser Totalordnung sortieren. Dazu gibt es mehere Möglichkeiten:

- konkrete Realisierung
- Methode der Schlüssel

- separate Methode `(Comparator)`

Der `Comparator` ist flexibler, da sich die Umstände ändern können, ausserdem ist die Methode leichter zu pflegen.

Sortieren einer Sequenz

über Insertion Sort oder Bubble Sort innerhalb der Queue oder mit Selection Sort in eine zweite Queue.

12.6 Heap

Datenstruktur ist ein binärer Baum, speichert die Objekte in internen Knoten. Externe Knoten sind `null`, Aufwand für Einfügen/Entfernen ist $O(logN)$.
Aufbau des Heaps:
Schlüssel in den Knoten werden anhand einer Ordnung in den Baum eingefügt, kleinster Schlüssel ist `root`, Schlüssel in Kinderknoten sind größer oder gleich dem Kinderknoten.
Ein Heap ist volständig, ein binärer Baum ist volständig wenn die Ebenen 0 bis h-1 die maximale Anzahl der Knoten haben und in Ebene h-1 alle internen Knoten links der externen Knoten d.h. die Höhe ist $h = log(n + 1)$

12.7 Heap als Vektor

- Index der Wurzel ist 1
- letzter Knoten ist n
- erster Knoten ist $n + 1$
- auch wenn Queue leer ist

13 Anmerkungen

Wie bereits dem Titel zu entnehmen ist, basiert dieses Werk auf der genannten Vorlesung, die von Prof. Dr.-Ing. Klaus Tönnies gehalten wird, sowie in weiteren grossen Teilen auf dem Buch 'Algorithmen & Datenstrukturen: Eine Einführung mit Java' von Gunter Saake und Kai-Uwe Sattler, sowie weiterhin auf Sedgewicks 'Algorithms in C'[1]
Diese Mitschrift ist eigentlich nur für meinen privaten Gebrauch gedacht, da ich zu den Typen gehöre die beim Lernen den Stoff noch einmal durcharbeiten und mit eigenen Formulierung niederschreiben.
Sollte sie irgendjemand für irgend etwas anderes nützlich finden, ist das natürlich auch nicht schlecht, ich freue mich über Spenden in Höhe von 23,5 € ;-)
Ansonsten gilt für dieses Dokument die BSD Lizenz

Ausserdem war dies eine weitere nützlich Übung um das geniale Textsatzsystem

LaTeX

noch besser kennenzulernen.
(2003 - 5 years of happy LaTeXing and I really do love it!)
Bei Schreibfehlern sind diff-patches willkommen.
Probleme mit Treepaket und PDF umgehen:

```
eigene Datei mit     \pagestyle{empty}
kompilieren:         latex tree2.tex
in PS wandeln:       dvips -E -f tree2.dvi > tree2.ps
in EPS wandeln:      ps2epsi tree2.ps
EPS in PDF wandeln:  epstopdf tree2.epsi
```

Editor:VIm
Graphiken :xfig
OS: NetBSD

Magdeburg, 10. Juli 2003
stefan@net-tex.de

[1] If Java had true garbage collection, most programs would delete themselves upon execution.
`Robert Sewell`

Index

(2-4)-Baum, 32
Überprüfungung von Algorithmen, 52

Abstiege, 23
Abstrakt und Konkret, 12
Abstrakte Datentypen und Java, 12
Adaption, 50
Adjazenzliste, 37
Algorithmen, Überprüfung, 52
Algorithmus, 9
Algortihmen, 9
Analysetypen, 11
ArrayStack, 17
Attribute, 12
Aufwand, 23
Aufwandsabschätzung, 11
Average Case, 11
AVL-Baum, 31
AVL-Kriterium, 31

Bäume, 22
Backus-Naur-Form, 53
Bad Character, 47
Best Case, 11
BFS, 38
Binäre Suchbäume, 27
binärer Baum, 22
Binary Search, 45
BNF, 53
Boyer Moore, 47
breadth-first search, 38
Breitensuche, 38
Brute Force, 50
Brute Force Mustererkennung, 47
Bubble Sort, 45
Bucket-Array, 40

Comparator, 56
Container, ADT, 21

Datenstruktur für Bäume, 28
depth-first search, 38
DFS, 38
Dictionary, 40
Divide and Conquer, 50
Double Hashing, 43
Dynamic Programming, 50

Einfügen von Elementen in doppelverkettete Liste , 16
Eltern-Kind Beziehungen, 22
Endrekursion, 9
Entdeckungskanten, 38
Entitäten, 56
Entwurfsmuster, 50
Entwurfstechniken, 51
Eulertour, 27
Exception Handling, 49
Exceptions, 49

Geordnete Lexika, 43
Geschwister, 22
Greedy, 50

Höhe des Baumes, 24
Hash-Code, 40
Hashfunktion, 40
Hashtabelle, 40
Heap, 57
Heap als Vektor, 57

Implementierung, 13
Inorder Traversierung, 25
Inspectable Container, 21
Interfaces, 13
InvalidPositionException, 19
Iterator, ADT, 20

Kantenliste, 37
Klasse, 12
KMP, 48
Knuth Morris Pratt, 48
Kollisionsverarbeitung, 42
Kompositionsobjekt, 56

Konstruktor, 13
Konstruktorverkettung, 13
Konzept, Zerlegung, 51
kopieren, Bin. Baum, 29

Löschen von Elementen in doppelverketteter Liste , 16
Ladefaktor, 42
Last-Occurence-Table, 47
Lexikon, 40
Lineares Testen, 42
Linkhopping, 20
Liste, 19
Liste, Aufwand:, 15
Liste, doppelt verkettete , 16
Liste, Operationen, 14
Liste, verkettete , 13
Look-Up Table, 43

Methoden, 12
Methoden auf Baum, 23
Mustererkennung, 47

Nachrichten, 12
NetBSD, 58
new, 13

Objekte, 12
Open Addressing, 42
Operationen, Listenelement:, 14

Paradigmen, 9
Patternrecognition, 47
Phasenkonzept, 51
Position, ADT, 19
Postorder Traversierung, 25
Präfix, 48
Preorder Traversierung, 25
Prioritätsschlange, 56
Programmablauf, 53
public class Node, 15

Quadratisches Testen, 43

Rand, 48
Re-Hashing, 43
Rechenzeit:, 11
Rot-Schwarz Baum, 33
Runtime Exeption Handling, 49

Selection Sort, 45
Seperate Chaining, 42
Sequenz als dvL, 20
Sequenz als Feld, 20
Sequenz als Mischimplementierung, 20
Sequenz, ADT, 20
Sequenz, lineare, 19
Signatur, 12
Sorten, 12
Sortieren, 45, 57
Spanning Tree, 38
Speicher:, 11
Speicheraufwand, 28
Spezifikation, 53
Sprunglisten, 43
Stabilität, 45
Stack, 17
Stack als Liste, 17
Struktogram, 53
Suchen, 45
Suffix, 48

Throwable, Klasse, 49
Tiefe des Baumes, 24
Tiefensuche, 38
Topologische Sortierung, 35
Totalordnung, 56
Transformation, binären Baum, 29
Transitive Hülle, 35
Traversierung, 24
Traversierung, Graphen, 38

Validierung, 52
Vektor, 19, 31
Vektoren für binäre Bäume, 28
Vererbung, 12
Verifikation , 52
VIm, 58
Visualisierung von Algorithmen, 53
Vorlesung, 7

Worst Case, 11

xfig, 58

Zeigerstruktur, beliebiger Baum, 29
Zeigerstruktur, binärer Baum, 28
Zyklen, 38