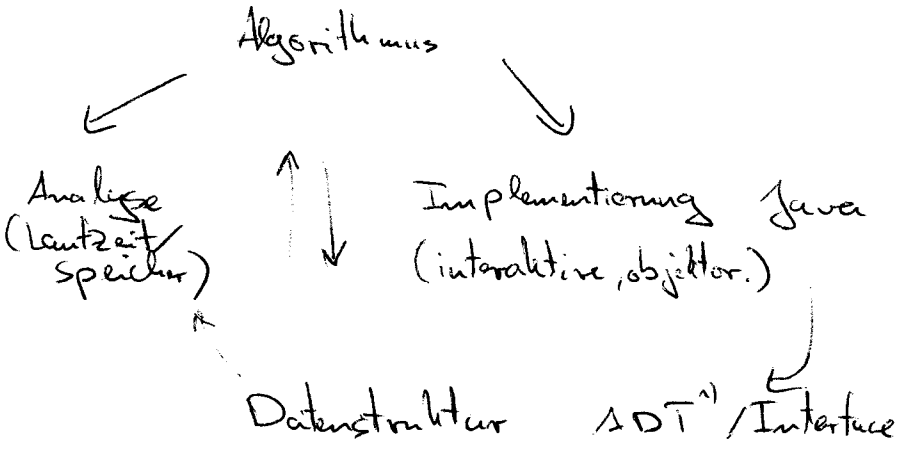
14.04.04

Inf B



- O-Notation
- Imperative Programmierung
- Java als Bsp einer OO-Sprache
- Datenstrukturen u. ADT
  - Kellerspeicher (stack)
  - Listen (verkettete u. doppelt verk. L.)
  - Warteschlange (Queue)
- Binäre Bäume
- Hashverfahren
- Graphen u. Graphalgos

*) Abstrakter Datentyp

Mo 14.11

Mo 14-16

Die 14-16

Fr 10-12

Do 14-16

Mo 10-12

Fr 10-12

Mi 10-12

Mo 10-12

16.04.04

1. Asymptotisches Verhalten (Wachstum) und die O-Notation

Zur Abschätzung der asymptotischen Laufzeit von Algos

Ein alg. Problem besteht aus einer Menge von Eingaben (Instanzen), einer Menge von Ausgaben und einer eindeutig spezifizierten Regel, die jeder Eingabe eine Ausgabe zuordnet

→ Ein Algo. ist deterministisches Verfahren, das für jede Eingabe in endlich vielen Elementarschritten die spezifizierte Ausgabe berechnet

RAM-Modell

Die Anzahl der Elementarschritte für eine spezielle Eingabe nennt man die Laufzeit (für die spezielle Eingabe)

Quicksort

Jede Eingabe hat eine Größe (Länge) n

Laufzeit des Algo.: $T(n)$ = maximale Anzahl von Elementarschritten bei Eingaben der Größe n (worst case analysis)

Bsp.:

1) Eingaben: Listen von Zahlen
Ausgaben: sortierte Listen von Zahlen
Problem: sortieren
Elementarschritte: Vergleiche

2) Eingaben: Boolesche Formeln in KNF
Ausgaben: 0, 1
Problem: Erfüllbarkeit
Elementarschritte: Boolesche Operationen

3) Eingaben: $\mathbb{N}$
Ausgaben: 0, 1
Problem: Ist $q \in \mathbb{N}$ prim
(n = Anzahl der Bits in der Binärdarstellung der Zahl q)
Elementarschritte: arithmetische Operationen

4) Eingaben: Punktmengen in der Ebene
(n = Anzahl der Punkte)
Ausgaben: reelle Zahlen
Problem: minimaler Abstand zwischen zwei Punkten aus der Menge
Elementarschritte: arithmetische Operationen

Algorithmen

1 a) Quicksort $T(n) = \frac{n(n-1)}{2}$
b) Mergesort $T(n) = c_1 \cdot n \cdot \log_2(n)$

2 a) Alle Belegungen testen (brute force) $T(n) = 2^n$
b) Resolutionskalkül $T(n) = a^n = 2^{n/c}$

3 a) alle Zahlen von 1 bis $\sqrt{q}$ als Teiler testen $T(n) = 2^{n/2}$
b) Sieb des Eratosthenes $T(n) = c \cdot \frac{2^{n/2}}{n}$

4 a) alle Entfernungen ausprobieren $3n(n-1)$
b) ? (Graphentheorie?)

Welche Funktionen spielen eine Rolle?

- 2^n, $2^{n/2}$ $\Rightarrow a^n$
- $\log_2(n)$, $\ln(n)$ $\Rightarrow \log_b(n)$
- Polynome
- n^a für $a \in \mathbb{Q}$
- $n!$

- Summen, Produkte, u.a. der obigen Funktionen

Welcher Algo ist besser?

1. Grundsatz: Konstante Faktoren sind zweitrangig

Def.: Seien $f, g : \mathbb{N} \to \mathbb{R}^+$

$g(n)$ ist asymptotische obere Schranke von $f(n)$ mit einer Konst. $c > 0$ und ein $n_0 \in \mathbb{N}$ existiert, so dass $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$

Schreibweise $f(n) \in O(g(n))$

oder $f(n) = O(g(n))$

Def. $O(g(n)) = \{ f(n) \mid \exists c > 0 \; \exists n_0 \; \forall n \geq n_0 \; f(n) \leq c \cdot g(n) \}$

$\Omega(g(n)) = \{ f(n) \mid \ldots \; c \cdot g(n) \leq f(n) \}$

$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

$o(g(n)) = \{ f(n) \mid \forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \; f(n) \leq c \cdot g(n) \}$

$\omega(g(n))$

Weiterführung O-Notation

Bei $O, \omega \rightsquigarrow$ große Konstanten

Bei $\Omega, o \rightsquigarrow$ kleine Konstanten

Satz: $f(n) \in O(g(n)) \Leftrightarrow \frac{f(n)}{g(n)}$ ist beschränkt

$\Leftrightarrow g(n) \in \Omega(f(n))$

$f(n) \in o(g(n)) \Leftrightarrow \frac{f(n)}{g(n)} \xrightarrow{n \to \infty} 0$

$\Leftrightarrow g(n) \in \omega(f(n))$

Grundregeln und Werkzeuge

1) Polynome: Bei P. entscheidet der Grad über die Stärke des Wachstums

z.B. ist $0{,}01\, n^4$ ist asymp. obere schranke

für $10^6 n^3 + 10^9 = f(n)$

$\frac{f(n)}{g(n)} \to 0$

$f(n)$ $g(n)$ Polynome von gleichem Grad

$\Rightarrow f(n) = \Theta(g(n))$

2) Exp. Wachstum ist stärker als polynom. Wachstum

a^n wächst stärker als n^b für $a > 1$

$a^n = e^{\ln(a) \cdot n}$

$$\lim_{n \to \infty} \frac{n^b}{e^{n \cdot c}} = \lim_{n \to \infty} \frac{b \cdot n^{b-1}}{c \cdot e^n} = \dots = \frac{b!}{c^b \cdot e^{n \cdot c}}$$

Bernoulli

$= 0$

3) Logarithmusfunktion

$k = \log_a(n) \Leftrightarrow n = a^k$

Basiswechsel $\log_b(n) = \frac{\log_a(n)}{\log_a(b)}$

$\Rightarrow$ Für alle $a, b > 1$ ist $\log_a(n) \in \Theta(\log_b(n))$, da

$$\log_a(n) = \log_b(n) \cdot \underbrace{\log_a(b)}_{\text{const.}}$$

4) Polynome und Wurzelfunktionen wachsen stärker als Logarithmusfunktionen

5) Ist $f(n) \in O(g(n))$ und $\lim_{n\to\infty} f(n) = \lim g(n) = \infty$

dann gilt auch $\log(f(n)) \in O(\log(g(n)))$

Achtung: Diese Regel kann nicht auf Exponentialfunktionen übertragen werden

Bsp.: $f(n) = 2n \quad g(n) = n \Rightarrow f(n) \in O(g(n))$

$h(n) = 2^{f(n)} \quad i(n) = 2^{g(n)}$

$\frac{h(n)}{i(n)} = \frac{2^{2n}}{2^n} = 2^n$ nicht beschränkt!

6) Vergleich von Exponentialausdrücken
- entweder gemeinsame Basis oder gemeinsamen Exponenten herstellen

7) Behandlung von $n!$

1) $2^{n-1} \leq \lceil n! \leq n^n \rceil$

$\Rightarrow n! \in \Omega(2^n) \wedge n! \in O(n^n)$

Herleitung mit Stirling-Formel

$n! \rightsquigarrow \log_2(n!) \in \Theta(n \cdot \log_2(n))$

Informatik B

Erg. zu O-Notation

8) Die Relation "asymptotische Schranke zu sein" ist transitiv

d.h. aus $f(n) \in O(g(n))$ und $g(n) \in O(h(n))$ folgt $f(n) \in O(h(n))$

9) Es gibt Paare von unvergleichbaren Funktionen, d.h. $f(n) \notin O(g(n))$ und $g(n) \in O(f(n))$

z.B.: $f(n) = 1 + 2^n + (-2)^n$

$g(n) = 1 + 2^{(n+1)} + (-2)^{(n+1)}$

Objektorientierte Programmierung und Java

1. Funktionale und anweisungsorientierte Programmierung im Vergleich

- Funktional (applikativ)
  - Programmablauf ist Auswertung eines Termersetzungsmechanismus
  - Keine direkte Möglichkeit der Speicherverwaltung
  - Keine Variablen für Speicherinhalte
  - Rekursion in der Form, dass eine Funktion sich selbst aufrufen kann, ist zentraler und wichtiger Bestandteil

- Anweisungsorientiert (imperativ)
  - Programm ist eine Folge von Anweisungen, die i.d.R. sequentiell abgearbeitet werden (Ausnahme: bed. od. unbed. Sprünge und Schleifen)

- Man kann Eingaben, Zwischenergebnisse und das Endergebnisse und das Endergebnis speichern. Dazu werden Variablen eingeführt als Symbol für einen best. Speicherinhalt

- Ähnlichkeit zur Maschinensprache mit erweiterten Mitteln z.B. Anweisung zur Ausführung eines Unterprogramms

- Rekursion in dem Sinne, das Funktionen sich selbst aufrufen sind nicht notwendig, der in allen modernen Sprachen vorhandene Konzept der Rekursion wird durch Schleifen realisiert

- Typen: primitiv
  abgeleitete bzw zusammengesetzte Datentypen (können Vordefiniert sein oder selbst definiert werden

- Jede Variable hat einen Typ (Deklaration)

- Elementarbausteine eines Programms sind Anweisungen:
  - Variablendeklaration (oft verbunden mit Initial.)
  - Zuweisungen
  - Auswertung von Ausdrücken (verbunden mit Zuweisung des Ergebnisses)
  - Blockanweisung
  - Funktionsaufruf
  - Bedingte Anweisungen
  - Schleifenanweisungen
  - Sprunganweisungen (break, continue)
  - leere Anweisung
  - weitere sprachspezifische Anweisungen

Java und C: for- und while-Schleifen gleich mächtig

Einstieg in Java

Geschichte:

- Produkt von SUN, ursprünglich geplant als Software für interaktives TV
- Eng verbunden mit der Entwicklung des WWW
- 94 erste experimentelle Applets im Web
- 95 Verbindung mit Netscape
  → explosionsartiger Erfolg
- 99 JDK 1.2

Was ist Java ?

Java ist eine volle Sprache, nicht nur Tool für Internet-Anwendungen

- objektorientiert
- architekturneutral (plattformunabhängig)
- interpretiert
- Syntax an C angelehnt

Wie arbeitet man mit Java?

1) Erstellung eine Quellcode - Datei
   Name.java

2) Compilieren der Dateien mit javac, eine
   neue Datei Name.class wird erzeugt

3) Zur Ausführung wird Name.class
   interpretiert
   Aufruf: java Name

Bei vielen Sprachen erzeugt Compiler ein ausführbares Programm. Diese sind dann nur auf einer Plattform lauffähig

Der Java - Compiler erzeugt ein Programm in Bytecode, das maschinenah ist

Die compilierten Programme werden durch einen virtuellen Prozessor interpretiert.
JVM nimmt die Anpassung an die Plattform vor

Datentypen in Java

- Primitive Datentypen

boolean		
char	Unicode-Zeichen	
byte	ganze Zahlen	2 byte
short		4 byte
int		8 byte
long		
float	Gleitkommazahlen	4 byte
double		8 byte

Operationen: && (UND) || (oder)
! (Negation) für boolean
% (Modulo) für integer

&, |, ^ Bitweise log. Verknüpfung von Integern

<, <=, >, >=, == Vgl.-op

++, -- Zum hoch/runterzählen

Es gibt Wrapper-Klassen

Typumwandlungen (Type Casting)

- automatische, z.B. byte → long

- Explizite Umwandlung durch (typ)

```
float x;
int i = (int) x;
```

Werttypen und Referenztypen

Unterscheidung von Datentypen danach, ob für Variablen dieses Typs bei der Deklaration ein Speicherplatz für ihren Inhalt angelegt wird oder erst bei der Definition

Referenztypen arbeiten über Zeiger

Java: alle primitiven Typen sind Werttypen, alle anderen sind Referenztypen (Felder, Klassen)

Bei prim. Typen erfolgt automatische Initialisierung (0) bei der Deklaration

Referenztypen:
Variablen eines solchen Typs verweisen auf ein Objekt, welches nicht durch die Deklaration sondern erst durch die Definition (= erste Zuweisung) (Bei Feldern reicht die größenangabe) angelegt wird.

Felder

```
int[] a1 ;  // enthält nil
a1 = new int[12] ; // Dimensionierung auf 12
```

auch möglich:

```
int[] a1 = {2,3,4,5}
a1.length = Länge des Feldes
```

Achtung: Bei Referenztypen wird bei Zuweisung immer nur die Referenz kopiert

Anlegen einer Kopie eines Feldes kann mit clone angefordert werden

Zum Vergleichen der Inhalte von Objekten equals() verwenden.

b. equals(a);

! gilt auch für strings !

Objektorientierter Zugang:

- Aufhebung der Trennung zw. Daten und
- Objekt ist Repräsentation eines "Dinges", das gekennzeichnet ist durch:
  - eine Identität (Eigsch. zur Unterscheidung v. and. O)
  - Attribute (Eigenschaften)
  - Methoden

Bsp. für eine Klasse:

```
Public class Point {
    public double x, y;   //Koordinaten-Attrib.
    public Point (double a, double b) { //Konstr.
            x = a;
            y = b;
    }

    public double distFromOrigin {
            return Math.sqrt(x*x + y*y);
    }
```

Abstrakte Klassen

- abstrakte Methode: nur deklariert, aber nicht definiert (implementiert).
  Modifikator abstract verwenden und Deklaration mit Semikolon abschließen
- Klassen mit abstrakten Methoden müssen selbst als abstract deklariert werden

- Jede Unterklasse einer abstrakten Klasse muss alle abstrakten Methoden der Oberklasse implementieren oder sie ist selbst abstrakt
- Abstrakte methoden können nicht static, private oder final sein

Beispiel einer abstrakten Klasse für Figuren

```
public abstract class Shape {
    public abstract double area();
    Public abstract double circumf();

}
```

Interface ist eine abstrakte Klasse, in der alle Methoden abstrakt sind.

Syntax: Schlüsselwort interface an Stelle von class

Alle Methoden sind implizit public und abstract

darf nur Attribute besitzen, die static und final sind

das Interface ist nicht instantiierbar (d.h. kein Konstruktor)

wenn eine Klasse ein Interface implementiert, verwendet man das Schlüsselwort implements

=> Einfachvererbung kann "aufgeweicht" werden: Eine Klasse kann von ihrer Superklasse erben und zusätzlich ein oder mehrere Interfaces implementieren

Equals

equals ist Methode von Object
in Klasse Object vergleicht equals nur die Referenzen, d.h. gleiches Ergebnis wie bei ==.
Im Ggs. zu == kann (und sollte man) equals überschreiben.
In der Klasse String wird die Gleichheit des Inhalts geprüft.
Bei Feldern ist dies nicht der Fall
Ab Java 1.2 kann die Methode java.util.Arrays.equals dazu verwendet werden.

Clone

```
int[▯] k = (int[]) i.clone();
```

Abstrakte Datentypen (ADT)

ADT dient zur Beschreibung der äußeren Funktionalität und der von außen sichtbaren Eigenschaften von bestimmten Objekten und abstrahiert von der konkreten Art der Implementierung der Funktionalität.

Datenstruktur : ADT mit Implementierung

Java: ADT wird durch ein Interface beschrieben

Datenstruktur ist eine Klasse, die dieses Interface implementiert

Objekte aus der Realität oder mathematische Strukturen

↓ abstrakte Beschr.

ADT

↑

mit der Entwicklung d. Inform. entstandene Standard-Datenstrukturen (stack, queue, verkettete

– und doppelt verkettete Listen, ...)

Mathematische Strukturen als ADT

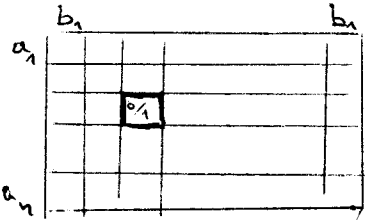
Mengen:
- Größe (endliche Menge ⇒ natürliche Zahl)
- Test auf Zugehörigkeit
- Elemente einfügen und streichen
- Aufzählung der Menge (in beliebiger Reihenfolge)

geordnete Mengen:
- Einfügen mit Positionsparameter oder mit vorgegebenem Ordnungskriterium

Folgen:
- Einfügen und Streichen mit Position

Relationen: Graphen und Matrizen

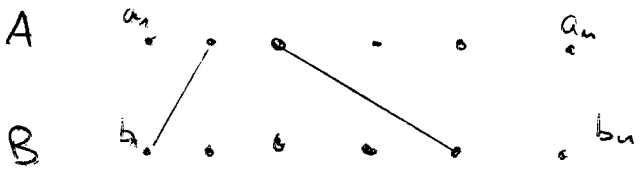
Eine Relation zw. zwei Mengen A und B ist eine Teilmenge R von $A \times B$



- 0 falls $(a,b) \notin R$
- 1 falls $(a,b) \in R$

bei beliebigen Einträgen → Matrix

Darstellung einer Relation als Graph



Kante von a nach b, genau dann wenn $(a,b) \in R$

als ADT:

- Größe von A und B
- Abfrage $(a,b) \in R$
- Funktionen zur Aktualisierung (a,b) in R setzen und löschen

Stapel (Stacks)

Ein Stack (Stapel, Kellerspeicher) ist eine Datenstruktur zur Verwaltung einer geordneten Menge von Objekten, bei der man immer ein Element anfügen kann oder das zuletzt eingefügte Element löschen kann

Beispiele:

1) Kartenstapel bei einigen Varianten von Patience
2) Back-Button bei Internet-Browsern
3) Rekursive Programme bzw Ausführung von Methoden, die andere Methoden aufrufen

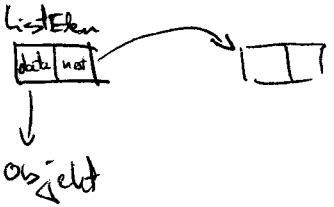
Stack

Implementierung mit verketteten Listen

Definieren Klasse ListElem
bestehend aus zwei Referenzen

```
private ListElem next;
Private Object data;
```

plus get und set-Methoden



ListStack besteht nur aus einem
Listenelement topElem

```
public void push (Objekt o) {
    ListElem newtop = new ListElem ();
    newtop.setData (o);
    newtop.setNext (topElem);
    topElem = newtop;
}
```

```
public Object pop() {
    Object o = topElem.getData();
    topElem = topElem.getNext();
    return(o)
}
```

Vorteil:

- Alle Operationen in konstanter und kleiner Zeit
- i.A. kein Stack Overflow möglich
- dynamische Speicheranpassung

Nachteil:

- Zusätzlicher Speicherverbrauch für next

Implementierung mit dynamischen Arrays

Anpassung der Feldgröße an den aktuellen Bedarf durch 3 Regeln

- Beginn mit Array der größe N
- Wenn ein Objekt gepusht werden soll, aber der Stack schon voll ist wird die Größe verdoppelt (alle Werte kopieren)
- wenn im Array nur noch 1/4 der Plätze belegt sind, wird die Größe auf 50% verkleinert (Einträge kopieren)

Vorteil:

- kein Überlauf möglich
- dynamische Speicheranpassung
- durchschnittliche Kosten pro Operation konstant

Nachteil:

- Operationen können beliebig teuer werden

Amortisierte Analyse der Laufzeit von push- und pop- Operationen:

- Angenommen, das Feld wurde gerade auf Größe M verändert (Kosten $M/2$)
- Nächste Größenveränderung frühestens nach $M/2$ push-operationen oder nach $M/4$ pop-operationen
- Kosten von $M/2$ können auf mind. $M/4$ operationen verteilt werden $\rightarrow$ 2 pro Operation

---

Bemerkungen

1) Die Art, dynamische Felder anzulegen, ist als Datenstruktur Vector bekannt
   java.util.Vector
   (auch in der StandardTemplateLibrary von C++)

2) Stacks können für beliebige Objekte verwendet werden (Personen, Punkten, ...), aber nicht für primitive Datentypen

- Ausweg: Wrapperklassen benutzen (z.B. Integer oder Double)

Queues

Eine Warteschlange ist eine Datenstruktur zur Verwaltung von Folgen von Objekten, bei der immer Objekte an das Ende angelegt werden können und zu jedem Zeitpunkt das Objekt entfernt werden kann, das als erstes eingefügt wurde.

Stack	Queue
last in first out	first in first out
LIFO	FIFO

Queue - Anwendungen:

- Druckerschlange
- Tastatureingabe

Queue als ADT

Hauptmethoden:

```
public void enqueue (Object o);
public Object dequeue ();
```

Zusatzmethoden:

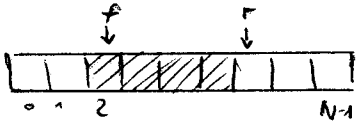
```
public int size();
public boolean isEmpty();
public Object front(); // Objekt nur ansehen
```

Implementierung von Queue

1. Array - Implementierung

a) Array fester Größe

N Default - Größe bzw. als Größenparameter übergeben



f: erste belegte Zelle

r: erste freie Zelle hinter dem belegten Teil

- Initialisierung

  f = ~~0~~ Ø, r = Ø

- Einfügen von Objekt o.

```
Q[r] = o;
r++;
```

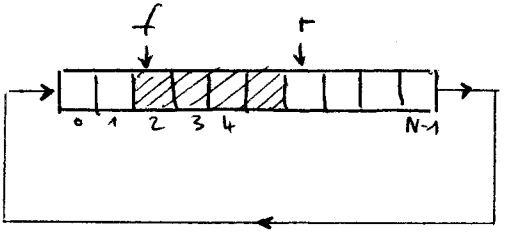
- Löschen

```
Object o = Q[f++]
return o;
```

Problem:

- Speicherprobleme auch bei kleinem Queue-Inhalt, da freigegebener Speicher wird nicht wiederverwendet

b) Zyklische Arrays



Beim Einfügen r und beim Löschen f um 1 modulo N erhöhen.

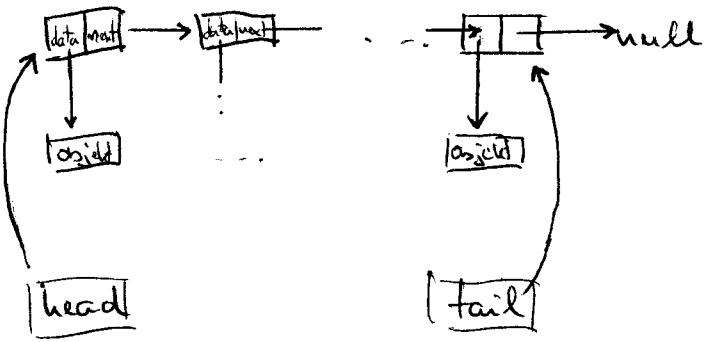
Speicherfehler bei r = f beim Einfügen

(beim Löschen ist für r = f die Queue leer → keine weitere Löschoperation erlaubt)

c) dynamisch-zyklische Arrays

2. Implementierung mit verketteten Listen

Klasse ListElem mit Referenzen next und data



Queue hält Listenelemente head und tail

Einfügen erfolgt auf der rechten Seite (tail)
(könnte auch auf der Seite von head realisiert werden)

Löschen kann nur auf der Seite von head erfolgen, dies ist für die Implementierung jedoch aus.

```
enqueue (Object o) {
  ListElem le = new ListElem();
  le.setData(o);
  le.setNext(null);
```

→

```
    tail.setNext(le);
    tail = le;
}

dequeue() {
    Object o = head.getData();
    head = head.getNext();
    return o;
}
```

Konstante Kosten pro Operation

Double-Ended Queues (DEQs)

auch Deque genannt; ist eine geordnete Datenstruktur, bei der man am Anfang und am Ende einfügen und löschen kann.

Methoden:

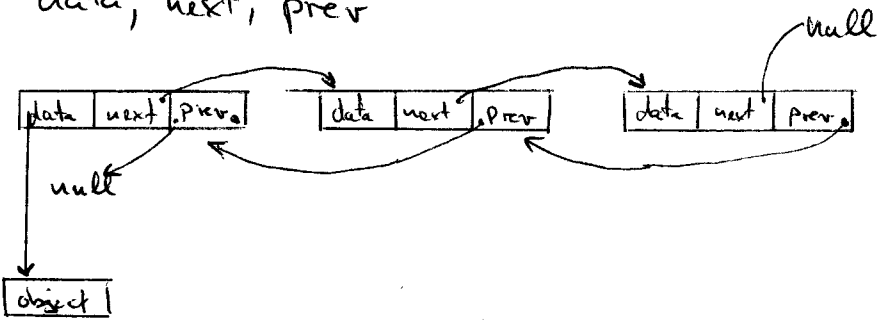
void insertFirst(Object o)

void insertLast(Object o)

```
Object removeFirst();
Object removeLast();
Object first(); // nur ansehen, nicht löschen
object last();
```

Implementierung durch doppelt verketteten Listen
(alternativ: zyklische Arrays)

Klasse DLListElem mit Referenzen
data, next, prev



Implementierung analog zu Queues

Weiterer Vorteil: Einfügen in der Liste auch an beliebiger Stelle in konstanter Zeit möglich:

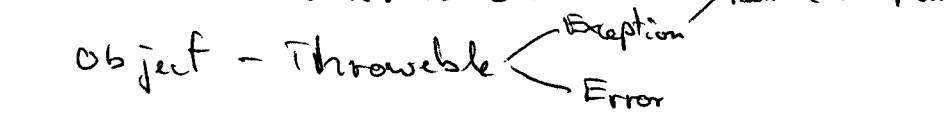
insertAfter(DLListElem le, Object o)

delete(DLListElem, le)

Fehlerbehandlung

Eine Exception ist ein Signal, das auf das Auftreten einer Ausnahmesituation verweist.
Jede Exception ist auch eine Klasse

In der Klassenhierarchie:



Exceptions können geprüft (checked) oder ungeprüft (unchecked) sein, alle, die nicht Unterklasse von RuntimeException sind, sind checked

Methoden können Exceptions "werfen", das muss deklariert werden.
Wenn eine Methode m_1 eine Methode m_2 verwendet und m_2 nach Deklaration eine bestimmte Exception wirft, dann muss auch m_1 entsprechend deklariert werden

Geworfene Exceptions können "aufgefangen" werden. Werden sie in der aufrufenden Methode nicht abgefangen, so werden sie weiter nach außen gereicht:

try - catch - finally

1. Ausnahmeklasse definieren (oder vordefinierte verwenden)

```
public class MyException extends Exception {
    public MyException () {
        super();
    }
}
```

Andere Konstruktoren möglich mit Fehlerursachen als Parameter

2. Methode m1 in der diese Exception auftreten kann

```
m1 (...) throws MyException {...}
```

in der Definition:

Wenn ~~fuelle~~ Exception auftritt, dann

```
throw new MyException();
```

3. Methode m2 verwendet m1 und soll Ausnahme auffangen

```
m2(...) throws MyException {
    try {
     ...
     m1(...);
     ...
    }
    catch (MyException e) {
        // Fehlerbehandlung
    }
    finally {
     ...
    }
```

Bäume

Motivation - Der Sequence - ADT

- Verallgemeinerung von Stack, Queue und Deque
- Verwaltung von linear geordneten Daten mit Zugriffen an beliebiger Stelle

  Beschreibung einer Stelle durch Position (Referenz auf das Objekt) oder Rang (Stelle in der nummerierten Reihenfolge)

Implementierung mit

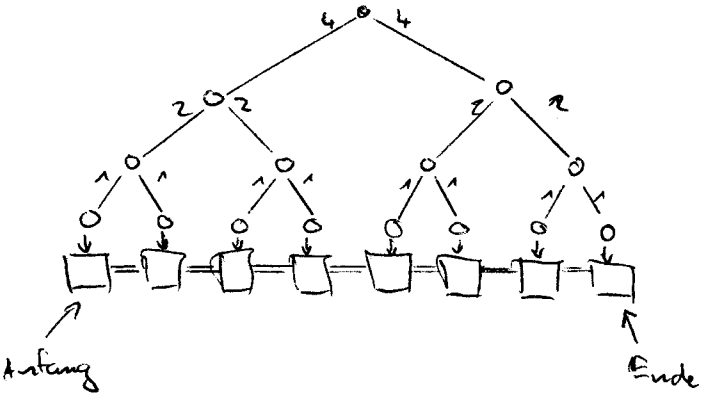
a) dynamischen Arrays

b) doppelt verketteten Listen

operation	Laufzeit (Array)	Laufzeit (Liste)
lesezugriff auf Pos	$\Theta(1)$	$\Theta(1)$
" auf Rang	$\Theta(1)$	$\Theta(n)$
Einfügen mit Pos	$\Theta(n)$	$\Theta(1)$
" mit Rang	$\Theta(n)$	$\Theta(n)$
Löschen mit Pos	$\Theta(n)$	$\Theta(1)$
" mit Rang	$\Theta(n)$	$\Theta(n)$

Idee für Verbesserung:

Lege über doppelt verkettete Liste einen binären Baum, so dass jedes Blatt Referenz auf ein Listenelement enthält.



Jeder Knoten enthält Informationen darüber, wieviele Blätter der linke und der rechte Unterbaum besitzen

Ein Durchlauf von der Wurzel zum entsprechenden Blatt reicht, um Position Rang k zu finden

Laufzeit: Tiefe des Blatts, $\Theta(\log(n))$ für balancierte Bäume

Problem: Baum muss bei Einfügen bzw. Löschen aktualisiert werden

Ungünstige Folge von Einfügeoperationen kann zu stark unbalanciertem Baum führen

Tiefe von Blättern kann $\omega(\log n)$ werden

$\Rightarrow$ Bäume

Definition: Ein (gewurzelter) Baum (rooted tree) besteht aus einer Menge T von Knoten, die wie folgt strukturiert sind:

1) Es gibt genau einen hervorgehobenen Knoten $r \in T$, die Wurzel des Baums
2) Jeder andere Knoten hat genau einen Vaterknoten
3) Die Wurzel ist Vorfahre jedes Knotens

Der Begriff "$v \in T$ ist Vorfahre von $u \in T$" ist rekursiv definiert durch:

i) $v = u$ oder

ii) $v =$ ist Vorfahre des Vaters von u

Weitere Begriffe:

u ist Kind von v $\Leftrightarrow$ v ist Vater von u

u ist Nachfolger von v $\Leftrightarrow$ v ist Vorfahre von u

u, v sind Geschwister (Nachbarn) $\Leftrightarrow$ u und v haben den gleichen Vater

Knoten ohne Kinder heißen Blätter

Knoten mit mind. einem Kind heißen innere Knoten

Definition: Ein Baum ist ein geordneter Baum, wenn für jeden Knoten die Menge seiner Kinder geordnet ist. (erstes Kind, zweites Kind, ...)

Der ADT TreeNode

```
TreeNode parent()
Object element()        // Daten, die v. d. Knoten gehalten wird
TreeNode[] children()
boolean isRoot()
boolean isLeaf()
setElem(Object o)       // Typ void oder Object
addChild(TreeNode v)    // Variante 1, v hat keine Kinder
addSubtree(TreeNode v)  // Variante 2, v kann Kinder haben
```

Def.: Ist T ein Baum, so nennt man $T' \subseteq T$ einen Unterbaum, wenn T' ein Baum ist (d.h. eine Wurzel hat) und für jedes $v \in T'$ gehören alle Kinder von v zu T'

Es gibt eine bijektive Abb. Knoten $\rightarrow$ Unterbäume

Tiefe und Höhe

Tiefe eines Knotens = Abstand zu Wurzel

Höhe eines Knotens = Abstand zum weitesten Nachfahren

Höhe (T) = Tiefe (T) = Höhe der Wurzel

Binäre Bäume

Def.: Ein binärer Baum ist ein geordneter Baum, bei dem jeder innere Knoten genau zwei Kinder hat (links u. rechts)

Allgemein. Ein b.n. Baum ist ein Baum, in dem jeder Knoten höchstens 2 Kinder hat (links oder/und rechts).
Wenn jeder innere Knoten zwei Kinder hat, dann echter (voller) binärer Baum

Satz: Für jeden binären Baum gilt: Anzahl der Blätter = Anzahl der inneren Knoten +1

Achtung: gilt nicht für allg. Def.

Beweis mit vollst. Ind. bez. Tiefe d des Baums

Knoten eines Baums kann man in Level einteilen nach Höhe der Knoten

Level 0 : Wurzel

Level 1 : Kinder der Wurzel

Level 2 : Kinder der Kinder ...

Für bin. Bäume:

Anzahl der Knoten in Level $(i+1) \leq 2 \cdot$ (Anzahl d in Level i)

Satz: Sei T ein bin. Baum mit n Knoten und Höhe h dann gilt:

1) $h+1 \leq$ Anzahl der Blätter $\leq 2^h$

2) $h \leq$ Anzahl der inneren Knoten $\leq 2^h - 1$

3) $2h+1 \leq n \leq 2^{h+1} - 1$

4) $\log_2(n+1) - 1 \leq h \leq \frac{n-1}{2}$

Zusätzliche Methoden für bin. Bäume

BinTreenode leftChild(); } nur für innere Knoten
BinTreenode rightChild(); }

expandExternal(); this muss ein Blatt sein, wird zu einem inneren Knoten durch Anlegen (Anhängen) von zwei Blättern

removeAboveExternal() this muss Blatt sein, löscht this und den Vater und der Bruder von this rückt an die Stelle des Vaters

Baumdurchläufe für bin Bäume

1) Preorder
2) Postorder
3) Inorder : Besuche erst linken Teilbaum, dann Wurzel, dann rechten Teilbaum

Traversierungen

Preorder

Postorder

Inorder – nur für bin. Bäume

Preorder – einsetzbar für Methoden bei denen die Bewertung der Kinder von der Bewertung der Eltern abhängt

Postorder – falls Bewertung des Elternknotens von der Bewertung der Kinder abhängt

---

Tiefe → Preorder

Höhe → Postorder

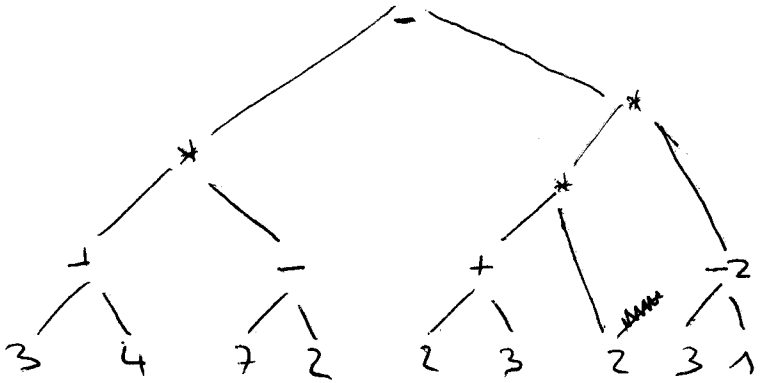
Size → Postorder
(Größe des Teilbaums)

Postorder für Auswertung von arithmetischen Ausdrücken

Operationen → innere Knoten

Eingabewerte → Blätter

$$((3+4)\cdot(7-2)) - (((2+3)\cdot 2)\cdot(3-1))$$



Anwendungen von Inorder

```
inorder (v) {
    if ( ! v.isLeaf()) inorder (v.leftChild());
    visit(v);
    if ( !v.isLeaf()) inorder (v.rightChild() );
```

→ binäre Suchbäume

Ein binärer Baum, der in seinen Knoten Zahlen (oder vergleichbare Objekte) speichert, wird b.n. Suchbaum genannt, wenn für jeden Knoten v das folgende gilt:

Zahlen im linken Teilbaum ≤ Zahl in v ≤ Zahlen in rechten Teilbaum

Inorder - Durchlauf gibt dann sortierte Folge aus

Eine "Mischung" der 3 Methoden ergibt die Euler-Tour eines binären Baums
"Linke-Hand-Methode"

Entscheidungsbäume

- dienen zur Klassifizierung von Objekten
- Objekte liegen in den Blättern des Baums
- in den inneren Knoten können Fragen über das Objekt gestellt werden, die mit ja/nein beantwortet werden (binäre Entscheidungsb.)
  größere Anzahl von Antworten → allgemeineres Modell)

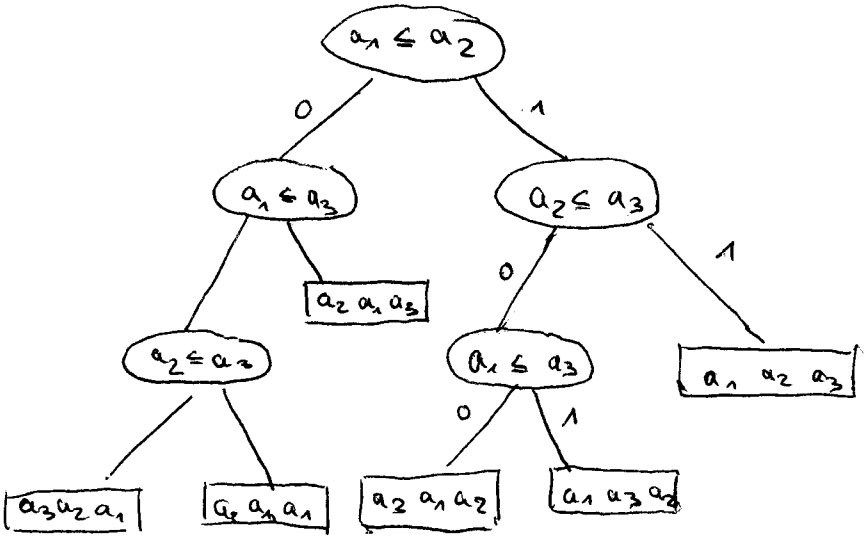
Entscheidungsbäume simulieren Algorithmen

Jeder innere Knoten ist eine if-Anweisung

Tiefe des Baums ist untere Schranke für die Laufzeit

besonders geeignet für vergleichsbasierte Algorithmen wie sortieren

Entscheidungsbaum für das Sortieren von Folgen der Länge 3 : $a_1\ a_2\ a_3$



- Jeder vergleichsbasierte Sortieralgorithmus kann durch einen binären Entscheidungsbaum simuliert werden

- Laufzeit im schlechtesten Fall $\in \Omega$ (Tiefe des Baums)
- Für Folgen der Länge n muss der Baum $n!$ Blätter haben (denn das ist die Anzahl der möglichen Ergebnisse)
- Anzahl der Blätter $\leq 2^{\text{Tiefe d. Baums}}$

$$n! \leq 2^{d(T)} \Rightarrow \log_2(n!) \leq d(T)$$

$$\log(n!) = \Theta(n \cdot \log_2(n))$$

$\Rightarrow$ Laufzeit von vergleichsbasierten Sortieralgorithmen $\Omega(n \log n)$

Heap-Sort (und and. Sortalgos)

Eingabe: $A = (a_1, \ldots, a_n)$ Folge von Zahlen
Ausgabe: B = sortierte Folge

Insertion-Sort

- Listen-Methode insert(x), einfügen von x in eine sortierte Liste an der richtigen Position
- Sortieren: $B = \emptyset$, sortiere $a_1 \ldots a_n$ nacheinander in B mit Methode insert
- Laufzeit

  insert(x) $O(l)$, l = Listenlänge

  Gesamtlaufzeit $\sum_{i=1}^{n-1} c \cdot i = c \cdot \frac{n(n-1)}{2} \in O(n^2)$

Quicksort

rekursiv

- Teile Liste A in a_1, A_l, A_r wobei a beliebig
- Sortiere rekursiv A_l und A_r (B_l, B_r) $\Theta(n)$
- $B = B_l + a_1 + B_r$ $O(1)$

Rekursionstiefe $\in n$

$\rightsquigarrow T(n) \in O(n^2)$

$T_{ges}(n) \in \Omega(n^2)$ sortierte Liste

Merge - Sort

- Teile A in $A_l = (a_1, \ldots a_{\lfloor \frac{n}{2} \rfloor})$

  und $A_r = (a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots a_n)$ $O(n)$

- Sortiere rekursiv A_l und A_r ?
- merge: Bilde aus B_l und B_r sortierte Liste B $O(n)$

Behauptung $T(n) = O(n \cdot \log n)$

Für $n = 2^k$ Rekursionstiefe k

$$T(n) = c \cdot n + 2 \cdot T\left(\frac{n}{2}\right)$$

↑ für Teilen u. Mergen

Zeigen: $T(n) \leq (c+1)\, n \cdot \log_2(n)$

mit vollst. Ind.

$n = 1$ $\quad T(1) = 0$ ✓

Induktions $\leq n \Rightarrow n$

$$T(n) \leq c \cdot n + 2\,T\left(\frac{n}{2}\right) \leq c \cdot n + 2 \cdot (c+1) \cdot \frac{n}{2} \cdot \log_2\left(\frac{n}{2}\right)$$

$$\leq (c+1) \cdot n + (c+1)\, n \left(\log_2 n - \log_2 2\right)$$

$$= (c+1)\, n \log_2 n$$

für n keine Zweierpotenz $\quad T(n) \leq T(\underbrace{2^{\lceil \log_2 n \rceil}}_{m})$

$$\leq 2(c+1) \cdot n \cdot \log_2(n) + 1 \in O(n \log n)$$

- Counting - Sort (speziell)

Voraussetzung: Die zu sortierenden Zahlen sind aus dem Bereich $\{1, 2, \ldots k\}$

1) Array C der Länge k mit Nullen init.
2) für $i = 1$ bis n : $C[a_i]$ ++ i
   danach: $C[j]$ Anzahl der j in A
3) für $j = 1$ bis k : Füge $C[j]$ mal die Zahl j in die Ausgabefolge ein

alternativ

3' a) für $j = 2$ bis k, $C[j] = C[j] + C[j-1]$;

danach $C[j]$ = Anzahl der Elemente $\leq j$ in A

3' b) für $j = n$ bis 1:

$B[C[A[j]]] = A[j]$

$C[A[j]] = C[A[j]] - 1$

Laufzeit: $\Theta(n+k)$

für $k = O(n)$ $\rightsquigarrow$ lineare Laufzeit

Radix-Sort

Idee: Zahlenfolge in Dezimaldarstellung

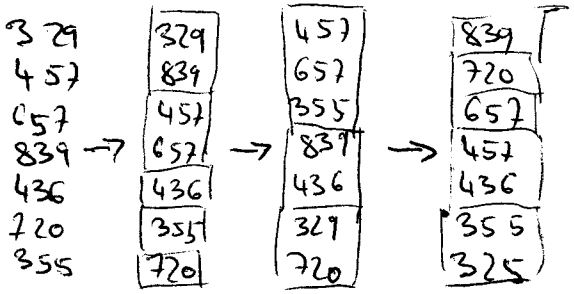
Verteile Elemente in Stapel $0, \ldots, 9$ nach letzter Stelle

und lege Stapel übereinander (9 oben)

Wiederhole mit vorletzter Stelle

" mit vorvorletzter Stelle

Beispiel



$O(d \cdot n)$ vgl. $O(d \cdot (n+k))$

↑ max Stellenzahl

↑ Anzahl der Ziffern

Heap-Sort

Ein Heap ist ein bin. Baum in dessen inneren Knoten Zahlen gehalten werden, so dass die folgenden 2 Eigenschaften erfüllt sind

1) Heap-ordnungseigenschaft: Für jeden inneren Knoten ist seine Zahl ≥ den Zahlen der Kinder

2) Vollständigkeitseigenschaft: Ist h die Höhe des Baums so ist das vorletzte Level $h-1$ vollständig gefüllt und

im Level 4 stehen die Knoten linksbündig

Halden (Heaps)

- Halden-Ordnungs-Eigenschaft
- Vollständigkeitseigenschaft

Heap-Sort

1) Halde aufbauen mit allen Elementen der Eingabefolge

2) Selection-Sort durch:
   Element der Wurzel herausnehmen und Heap-Eigenschaften wiederherstellen

→ Einfüge und Löschoperationen implementieren
dazu braucht man neben root noch lastElem,
der letzte innere Knoten

Einfügen: Einfügeposition wird als erstes Blatt auf der Euler-Tour, die von lastElem nach oben startet, gefunden

- Element einfügen, zwei Blätter anhängen und lastElem aktualisieren

- Ordnungseigenschaft herstellen durch schrittweises Vertauschen nach oben (Up-Heap-Bubbling)

Löschen des Elements aus der Wurzel

- Element aus Wurzel herausnehmen und dafür Element von lastElem einsetzen
- lastElem aktualisieren (Umkehrung von Einfügen)
- Ordnungseigenschaft herstellen durch schrittweises vertauschen nach unten (Down-Heap-Bubbling)
  Vertauschen gegen kleineres Element unter aktueller Position, wenn beide Kinder kleiner, Tausch gegen das kleinste von beiden

Analyse:

Zeit für eine Einfügeoperation ist

$O(d)$ d - Tiefe des Heaps

Zeit für eine Löschoperation

$O(d)$ d - Tiefe des Heaps

Laufzeit für Sortieren $O(n \log n)$, denn $\log_2 n$ ist obere Schranke für Tiefe der Halden

Satz: Die Höhe (Tiefe) eines Heaps, der n Elemente speichert ist $h = \lceil \log_2 (n+1) \rceil$

Beweis: aus der Vollständigkeit folgt

$$1 + 2 + 4 + \ldots + 2^{h-2} + 1 \leq n$$

mind. ein innerer Knoten im vorletzten Level

$$n \leq 1 + 2 + 4 + \ldots 2^{h-2} + 2^{h-1}$$

vorletztes Level nur mit inneren Knoten

$$2^{h-1} \leq n \leq 2^h - 1$$

$$2^{h-1} + 1 \leq n+1 \leq 2^h \qquad | \log_2$$

$$\log(2^{h-1} + 1) \leq \log(n+1) \leq h$$

$$h-1 < \log_2(n+1) \leq h$$

$$\Rightarrow \lceil \log_2(n+1) \rceil = h$$

Heap-Aufbau kann schneller gemacht werden durch Bottom-Up Heap-Konstruktion

Sei $n = 2^h - 1$

Idee: 2^{h-1} Elemente werden in Heaps der Höhe 1 eingefügt

Paarung bilden

Jedes Paar wird durch ein Element aus dem Rest zu einem Heap zusammengefasst

→ jetzt 2^{h-2} Heaps der Höhe 2

rekursiv weiter: Paarung bilden

Kosten für Bottom-Up Konstruktion (nur Tauschop)

$$= \sum_{i=1}^{h-1} i \cdot 2^{h-(i+1)} = \underbrace{2^h}_{h=1} \cdot \underbrace{\sum_{i=1}^{h-1} i \cdot 2^{-(i+1)}}_{\text{Konstant ?}} *$$

$$\underbrace{\sum_{i=1}^{h-1} \sum_{j=i}^{h-1} 2^{-(j+1)}}_{2^{-i} - 2^{-h}} \rightsquigarrow 1$$

Beobachtung: Jede Zahl im Level $h-2, h-3, \ldots 1$ bekommt $2 €$

Pro Tauschritt muss $1 €$ bezahlt werden

Induktiv: Am Ende bleibt 1 Euro übrig

Array-Implementierung von Heaps

Idee: Level nacheinander von 0 bis k eintragen, jedes Level von links nach rechts

Die Kinder von $A[i]$ sind $A[2i+1]$ und $A[2i+2]$

Vater von $A[j]$ ist $A\left[\frac{j-2}{2}\right]$ wenn j gerade und $A\left[\frac{j-1}{2}\right]$ wenn j ungerade

warum?

$A[i]$ in Level k von 2^k-1 bis $2^{k+1}-2$

Wieviele Kinder stehen zwischen i und dem ersten (linken) Kind von i:

= Nachfolger von i in Level k +
2. Vorgänger von i in Level k

$$= (2^{k+1}-2) - i + 2 \cdot (i - (2^k - 1))$$

$$= -i + 2i = i$$

$\Rightarrow$ erstes Kind bei $i + i + 1 = 2i + 1$

Prioritätswarteschlangen (Priority-Queue)

- verwaltet Objekte mit einem Schlüssel (oft Zahlen)
- Schlüssel sind von einem Datentyp mit totalem Vergleichsoperator $\leq$, der 4 Eigenschaften haben muss:
  1) total, d.h. für a, b gilt $a \leq b \vee b \leq a$
  2) reflexiv
  3) transitiv
  4) antisymmetrisch $a \leq b \wedge b \leq a \Rightarrow a = b$
- man kann zu jedem Zeitpunkt ein Objekt mit Schlüssel einfügen
- man hat zu jedem Zeitpunkt Zugriff zu Objekt mit kleinstem Schlüssel und kann es auch löschen

Wörterbücher (Dictionarys)

Objekte mit Schlüssel speichern

Operationen: Finden, Einfügen, Löschen

für Lexika, Duden, Telefonbuch, ...

Methoden

```
int size()
Boolean isEmpty()
object findElement(key k)
    ein Objekt mit Schl. k oder
                NO_SUCH_KEY
Object[] findAllElements(key k)
void insertItem(key k, Object e)
Object remove(key k)      // streiche ein Obj. mit Sch. k
Object[] removeAll(key k)
key closestKeyBefore(key k)
key closestKeyAfter(key k)
```

Binärsuche in geordneten Feldern

Schreibe key(i) für den Schlüssel des i-ten Items

finde Schlüssel k im Bereich low und high

if key(high) = k return high

while (high > low)

mid = $\lfloor \frac{high + low}{2} \rfloor$

if key(mid) = k return k

else if (key(mid)) < k low = mid

else high = mid

return NO_SUCH_KEY

zeigen mit Induktion, dass (high - low + 1) wird in jedem Schleifendurchlauf mind. halbiert

$\Rightarrow$ Rekursionstiefe $\lfloor \log_2 n \rfloor + 1$

geeignet, wenn nur gesucht werden soll

Binäre Suchbäume als Wörterbücher

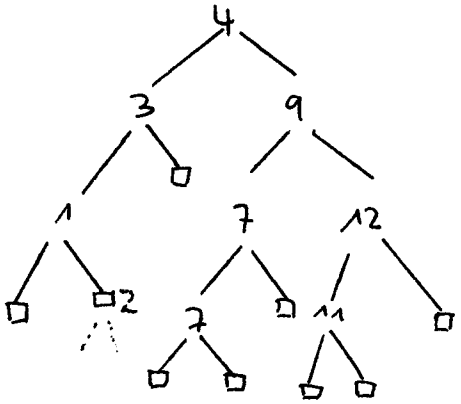
- items nur in inneren Knoten
- u, v, w innere Knoten, u im linken und w im rechten Teilbaum von v dann
  $key(u) \leq key(v) \leq key(w)$

Suche nach einem Schlüssel k, Rückgabe von Knoten mit Schlüssel bzw. eines Blatts für NO_SUCH_KEY

```
TreeSearch (k, v)
    if (v Blatt) return v
    else if k = key(v) return v
        else if k < key(v) TreeSearch (k, leftChild (v))
            else TreeSearch (k, rightChild (v))
```

Laufzeit $O(h)$ h Höhe von v

Einfügen / Löschen



2 einfügen

4 einfügen: wenn 4 schon vorhanden, mit Inorder zum nächsten Blatt gehen

9 löschen: mit inOrder von der 9 ins nächste Blatt → 1 hoch → vertauschen → 9 löschen (d.h. eins der Blätter von 9 mit Vater löschen)

Einfügen von (k, e)

1) TreeSearch $(k, root) \rightarrow v$

2) v Blatt $\rightarrow (k, e)$ eintragen und zwei Blätter an-

v innerer Knoten → Inorder-Traversierung für rightChild(v) beginnt bei Blatt w

(k, e) in w eintragen und Blätter anhängen

Löschen von Eintrag mit Schl. k

1) TreeSearch (k, root) → v

2) v Blatt → return NO_SUCH_KEY

v innerer Knoten : Inorder für rightChild (v)

→ Blatt w

u = Vater (w)

Einträge von v und u austauschen

w und u löschen, Bruder von w übernimmt Stellung von u

Laufzeit für Suchen, Einfügen und Löschen ist O(h) h = Höhe des Baums

---

AVL - Bäume

Def.: Ein bin. Baum T hat die Höhe - Balance - Eigenschaft, wenn für jeden inneren Knoten v die Höhen seiner Kinder um höchstens 1 differieren

Def.: Ein binärer Suchbaum mit Höhen-Balance Eigenschaft wird AVL-Baum genannt

Achtung: Tiefen von Blättern in einem AVL-Baum können sehr große Differenzen aufweisen

Satz: Die Höhe eines AVL-Baums, der n Einträge speichert ist $O(\log_2(n))$

Bew.: $n(h) :=$ min Anzahl von inneren Knoten im AVL-Baum der Höhe h

$h=1 \rightsquigarrow n(h) = 1$

$h=2 \rightsquigarrow n(h) = 2$

$n(h) \geq n(h-1) + n(h-2) + 1$

$> 2 \cdot n(h-2)$

$\geq 2^{\lfloor \log_2 n \rfloor}$

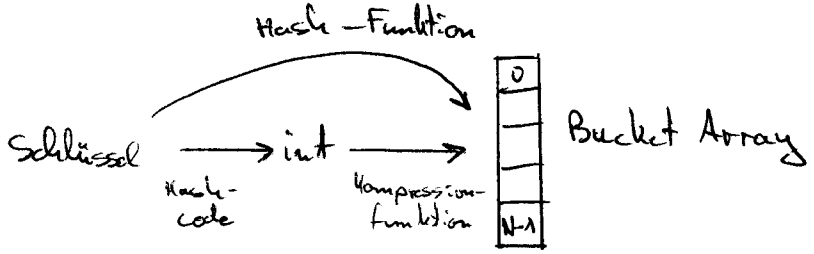
$\rightsquigarrow h \leq 2 \log_2(n)$

Suchen in $O(\log_2 n)$ Zeit

Einfügen und Löschen auch in $O(\log_2 n)$ Zeit, aber Höhen-Balance-Eigenschaft kann verloren gehen.

Wiederherstellung durch "Rotation"

Inf 7.7.04



$h(k) = k \bmod N$ Divisionsmethode

$h(k) = (ak + b) \bmod N$

Kollisionsbehandlung

1) verkettete Listen für jedes Fach

2) Sondierungsstrategien, um an Stelle des schon besetzten Fachs ein freies Fach zu finden

a) Lineare Sondieren

Wenn $h(k)$ besetzt, sondiere $h(k)+1$, $h(k)+2$...

bis freies Fach gefunden

Problem: Klusterbildung

besser: $h(k)+c$, $h(k)+2c$, ...

b) quadratisches Sondieren

ist $h(k)$ besetzt, sondiere $h(k)+1^2$, $h(k)+2^2$, ...

Graphen und ihre Darstellung

Def.: Ein Graph (genauer: ein einfacher ungerichteter Graph) besteht aus einer Menge V von Knoten (Ecken, Vertex) und einer Menge E von Kanten (Edge), wobei jede Kante ein ungeordnetes Paar von Knoten ist

ungeordnetes Paar $\{u, v\} = \{v, u\} \rightarrow$ ung. Kante

Variante: E besteht aus geordneten Paaren
$\rightarrow$ gerichteter Graph

$\binom{V}{2}$ bezeichnet die Menge aller ungeordneten Paare aus V

$G = (V, E)$ gegeben

v ist adjazent zu $u \Leftrightarrow \{u, v\} \in E$

v ist inzident zu Kante $e \Leftrightarrow v \in e$

adjazente Knoten nennt man auch benachbart

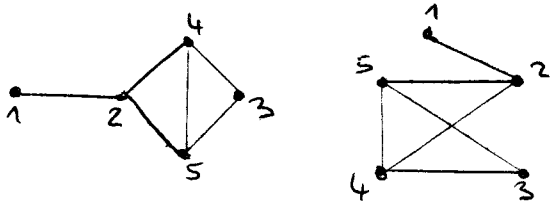
Nachbarschaft von v $N(v) = \{u \in V \mid \{v,u\} \in E\}$

Grad von v ist Anzahl der Nachbarn

$$deg(v) = |N(v)|$$

Darstellung von Graphen

1) Darstellung durch Zeichnung



2) Adjazenzmatrix

pro Knoten eine Zeile und eine Spalte

$$a_{u,v} = \begin{cases} 1 & \text{falls } \{u,v\} \in E \\ 0 & \text{sonst} \end{cases}$$

	1	2	3	4	5
1	0	1	0	0	0
2	1	0	0	1	1
3	0	0	0	1	1
4	0	1	1	0	1
5	0	1	1	1	0

3) Adjazenzlisten

Für jeden Knoten Liste der Nachbarn

1: 2; 2: 1,4,5; 3: 4,5 , ...

(2) , (1,4,5), (4,5), (2,3,5), (2,3,4)

$G=(V,E)$
$|V|=n$
$|E|=m$

	Adj. - Matrix	Adj - Liste
Speicher	$\Theta(n^2)$	$\Theta(n+m)$ $\Theta(m)$* * wenn man leere Listen weglässt
Anfrage $\{u,v\} \in E$?	$\Theta(1)$	$\Theta(n)$ $\Theta(\log n)$* *bei aufwend. Datenstruktur
Updates	$\Theta(1)$	$\Theta(n)$ $\Theta(\log n)$*
Nachbarsch. ausgeben	$\Theta(n)$	$\Theta(\deg v)$

Probleme und Anwendungen

1) 4-Farben-Problem

Kann man jede Landkarte mit 4 Farben einfärben, dass benachbarte Länder verschiedene Farben haben

Länder ~> Knoten

adjazent, falls gemeinsames Grenzstück

2) Euler - Kreise

Gibt es einen Weg, d.h. Kreis, der jede Kante genau einmal enthält

Nur, wenn alle Knoten geraden Grad haben

3) Hamilton - Kreis:

Gibt es einen Kreis, der jeden Knoten genau einmal besucht?

NP - vollst.

4) TSP

Billigste Rundreise gesucht

NP - vollst.

$G = (V, E)$ ungerichteter Graph

Kanten sind gewichtet, Adjazenzmatrix mit positiven Werten belegt

Problem: Route finden / Entscheidungsproblem: Gibt es eine Rundreise?

Deduktion von Hamilton-Kreis aus TSP-Entscheidungsp.

$G=(V,E)$ hat G Hamilton-Kreis ?

$G \to G' = (V, \binom{V}{2})$ $\quad |V|=n$

↑ alle Kanten

+ Knoten $w(e) = \begin{cases} 1 & e \in G \\ n+1 & e \notin G \end{cases}$

TSP für G' mit $k=n$

Antwort ja → Hamiltonkreis existiert

Antwort nein → kein Hamiltonkreis

Satz: Für jeden Graphen $G=(V,E)$ gilt

$\sum_{v \in V} \deg(v) = 2|E|$, d.h. $\sum_{v \in V} \deg v$ ist immer gerade

Doppeltes Abzählen der Knoten-Kante-Inzidenzen

1) Ausgehend von den ~~Knoten~~ Kanten → $2|E|$

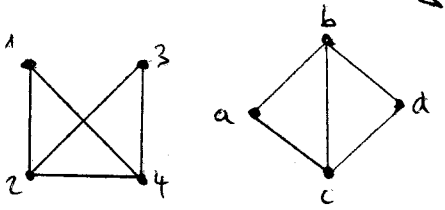
2) Ausgehend von Knoten → $\sum_{v \in V} \deg v$

Folgerung Die Anzahl der Knoten mit ungeradem Grad ist gerade.

Def.: Seien $G = (V, E)$ und $G' = (V', E')$ zwei Graphen.
Eine Abbildung $\varphi : V \to V'$ ist ein Graph-Homomorphismus, mit der Eigenschaft, dass für alle $\{u, v\} \in E$ auch $\{\varphi(u), \varphi(v)\} \in E'$
Ist φ auch umkehrbar und φ^{-1} auch ein Graph-Homomorphismus, dann ist φ ein Graph-Isomorphismus

Anschaulich:

G und G' sind isomorph, wenn sie sich nur durch die Benennung der Knoten unterscheiden



$1 \to a$
$2 \to b$
$3 \to d$
$4 \to c$

Standardbeispiele, Isomorphieklassen

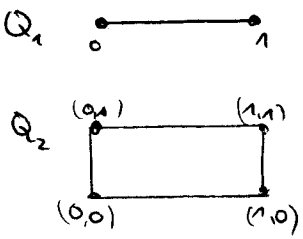
1) vollständige Graphen K_n mit n Knoten und allen Kanten

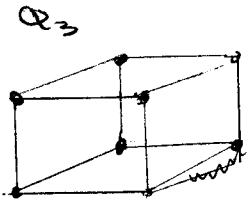
2) Kreise C_n

3) n-dimensionale Würfelgraphen Q_n

$V = \{0,1\}^n$ (n-Tupel)

Kante zwischen zwei Tupel, wenn sie sich an genau einer Stelle unterscheiden





Q_n hat 2^n Knoten und $2^{n-1} \cdot n$ Kanten

4) Vollständige bipartite Graphen $K_{n,m}$

$V = A \dot\cup B \quad |A| = n \quad |B| = m$

alle Knoten zwischen A und B

$K_{1,3}$ $K_{3,3}$

nicht planar

$K_{2,n}$ ist immer planar

Def.: $G = (V, E)$ ein Graph; $G' = (V', E')$ ist Untergraph von G, wenn $V' \subseteq V$ und $E' \subseteq E$

G' ist induzierter Untergraph von G, wenn $V' \subseteq V$ und $E' = E \cap \binom{V'}{2}$, d.h. adjazente Knoten sind wieder adjazent (wenn sie beide im Untergraphen sind

Def.: G ist ein bipartiter Graph, wenn er Untergraph eines vollst. bipartiten Graphen ist

Def.: Das Komplement von $G=(V,E)$ ist der Graph $\bar{G} = (V, \binom{V}{2} \setminus E)$

Def.: Ein Weg der Länge k in $G=(V,E)$ ist eine Folge von paarweise verschiedenen Knoten $v_0, v_1, \dots v_k$ mit $\{v_i, v_{i+1}\} \in E$.
Es ist ein Kreis der Länge $k+1$, wenn auch $\{v_k, v_0\} \subset E$
Abstand ist Länge des kürzesten Wegs