Vorlesung
Datenstrukturen und Algorithmen
von Prof. Seidl
SS 2003

geTeXt von
Michael Neuendorf
(michael.neuendorf@post.rwth-aachen.de)

Vielen Dank an Anne, Meea und Sascha,
die mir gelegentlich mit ihren Mitschriften
geholfen haben.

Da ich die Vorlesung mit meinem Laptop direkt in TeX setze, fehlen mir leider die Grafiken, die in der Vorlesung ergänzt wurden. Solltet Ihr diese Grafiken haben, so schickt sie mir doch bitte zu (entweder gescannt oder besser noch als xfig-file).

18. Juli 2003

Ich habe diese Mitschrift begonnen, weil Professor Seidl zu Beginn der Vorlesung ankündigte, daß es kein Skript geben werde und ich deutliche Probleme beim handschriftlichen Mitschreiben hatte.
Leider wird es auch zu einigen Fehlern in dieser Mitschrift gekommen sein, bedingt durch Probleme, Prof. Seidls Handschrift zu lesen oder die richtige Taste auf der Tastatur zu treffen.
Diese Mitschrift soll Euch eine Lernhilfe bei der Klausurvorbereitung sein, die den Inhalt in strukturierter Form wiedergibt. Dabei besteht weder ein Anspruch auf Vollständigkeit noch auf Korrektheit des wiedergegeben Stoffes.
Solltet Ihr irgendwelche Fehler finden, so sendet sie mir doch bitte in einer kurzen eMail an michael.neuendorf@post.rwth-aachen.de zu, in die Ihr bitte nicht die Seitennummer, auf der sich der Fehler befindet, sondern die entsprechende Absatzüberschrift zusammen mit dem Fehler, der Korrektur und ggf. einer kurzen Begründung schreibt.

Michael Neuendorf

Inhaltsverzeichnis

1 Grundlagen

1.1 Algorithmen und Komplexität

1.1.1 Notation von Algorithmen

- Algorithmen in JAVA
  - Grobstruktur des Algorithmus
  - keine lauffähigen Programme
  - Syntax an JAVA angelehnt
  - oft ohne Variablen- und Klassendeklarationen
- Implementation
  - Klassenhierarchie ?
  - Variablenverwendung ?
  - Feinheiten fehlen noch ?
  - Strukturierung der Daten → sinnvolle Teilstrukturen, Klassen, Methoden

1.1.2 Effizienz von Algorithmen

- Kategorien
  - Kommunikationsaufwand
  - Rechenzeit (Anzahl der Einzelschritte)
  - Speicherplatzbedarf
  - Zugriffe auf die sekundären Speicher (Platten, Bänder,...)
- Komplexität
  - Abhängig von Eingabedaten: Größe / Anzahl der Eingabesätze
  - Üblicherweise asymptotische Betrachtung
  - z.B. abstrakte Rechenzeit $T(n)$ (n ist Größe der Eingabe):
    * bei Sortierverfahren: Anzahl der zu sortierenden Werte $a_1, a_2, \ldots, a_n$
    * bei Suche: obere Grenze des Bereichs
  - Weitere Abhängigkeiten
    * Hardware
    * Betriebssystem

Beispiel: Sequentielle Suche

- Sequentiell:
  - Zahlenreihe
  - durchlaufen der Zahlenreihe
  - vergleichen
  - gefunden, Ausgabe der Position
  - nicht gefunden, Ausgabe der Fehlermeldung
- Laufzeitanalyse
  1. erfolgreiche Suche (n Vergleiche max, $\frac{n}{2}$ im Durchschnitt)
  2. erfolgelose Suche (n Vergleiche)

Beispiel: Binäres Suchen

- Suche:
  - Zahlen müssen sortiert sein
  - Halbieren der Anzahl
  - Anfang: Mitte
  - je nachdem: nach rechts / links weitersuchen
  - ggf. rekursives Suchen nach dem gleichen Verfahren
  - falls neue Hälfte leer, Fehlermeldung
- Laufzeitanalyse: Suche benötigt max $[\log_2(n)]$ Vergleiche: Höhe des Entscheidungsbaumes

Vergleich der Suchverfahren

- $n = 1000$
  1. sequentiell: 1000 Vergleiche
  2. binär: 10 Vergleiche
- $n = 1.000.000$
  1. sequentiell: 1.000.000 Vergleiche
  2. binär: 20 Vergleiche

Komplexitätsklassen

1	konstant
$\log n$	logarithmisch
n	linear
$n \log n$	(überlinear)
n^2	quadratisch
n^3	kubisch
n^k	polynomiell vom Grad k
2^n	exponentiell
$n!$	Fakultät
n^n	

Random-Access-Machine

- Axiomatisches Rechnermodell als Vergleichsmaßstab
- Speicherzellen mit ganzzahligen Werten (externe Speicher nicht betrachtet)
- Ein-, Ausgabeband, von links nach rechts les- und schreibbar
- Recheneinheit mit Akkumulator, Programmzähler
- Programm: abstrakte Assemblersprache (Tabelle)
- Kostenmaße:
  - Einheitskostenmaß: pro Anweisung
  - logarithmisches Kostenmaß: Anhängigkeit Operandenmaß
  - Befehlskosten: wenig standardisiert, Speicherzugriffsabhängig (DIV teurer als ADD)
- Unterprogramme (z.B. Methoden) mittels Compiler in RAM:
  - Aufruf: Aktivierungsblock reserviert für Parameter, Variablen

1.1.3 O-Notation für asymptotische Aussagen

- "Groß O" $O(f) = \{g : N \to R^+ | \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : g(n) \leq c * f(n)\}$
- Omega $\Omega(f) = \{g : N \to R^+ | \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : g(n) \geq c * f(n)\}$
- Theta $\Theta(f) = \{g : N \to R^+ | \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : \frac{1}{c} f(n) \leq g(n) \leq c * f(n)\}$
- "Klein o" $o(f) = \{g : N \to R^+ | \exists c > 0 \exists n > 0 \forall n \geq n_0 : 0 \leq g(n) < c * f(n)\}$
- "Klein omega" $\omega(f) = \{g : N \to R^+ | \exists c > 0 \exists n > 0 \forall n \geq n_0 : 0 \leq c * f(n) < g(n)\}$

Schreib- und Sprechweisen

- Sprechweisen
  - $g \in O(f)$: "f ist obere Schranke von g", "g wächst höchstens so schnell wie f"
  - $g \in \Omega(f)$: "f ist untere Schranke von g", "g wächst mindestens so schnell wie f"
  - $g \in \Theta(f)$: "f ist Wachstumsrate von g", "g wächst wie f"
- Schreibweisen: statt $g \in O(f)$ oft: $g(n) \in O(f(n))$, $\Theta(\log n)$

Rechenregeln

- Definiere Addition, Multiplikation, Maximumbildung bildweise: Bsp.: $f + g)(n) = f(n) + g(n)$
- Addition $f + g \in O(\max\{f, g\})$
- Multiplikation $a \in O(f); b \in O(g) \Rightarrow ab \in O(fg)$
- Bsp.: Linearität $g(n) = \alpha f(n) + \beta; \alpha, \beta \in R^+; f \in \Omega(1);$ dann: $g \in O(f)$
- Falls $\lim\limits_{n\to\infty} \frac{g(n)}{f(n)}$ existiert, ist $g \in O(f)$
- Es gilt: $\Theta(f) = \Omega(f) \cap O(f)$

Regeln für die Laufzeitanalyse

- Elementare Anweisungen
  - in $O(1)$
  - wichtig: welche Anweisungen in gewählten Rechnermodell elementar ? – In höheren Programmiersprachen scheinbar elementare Anweisungen möglicherweise komplexe Anweisungsfolgen
  - Beispiel: Menge M, Object D: isElement (M,O);
- Folgen von Anweisungen
  - Sei "A; B" eine Folge von Anweisungen A und B, dann $T_{A;B} = T_A + T_B \in O(\max\{T_A; T_B\})$
  - mehrere Hintereinanderausführungen: analog, maßgeblich ist der höchste Aufwand
- Schleifen
  - Summe über die einzelnen Durchläufe

- Falls Laufzeit des Schleifenkörpers unabhängig von jeweiligem Durchlauf ist: $T = d * T_s$ mit d: Anzahl Durchläufe, T_s: Laufzeit Schleifenkörper
- Allgemein: $d \in O(g)$ und $T_s \in O(f) \Rightarrow T \in O(f * g)$

- Bedingte Anweisungen
  - if (Bedingung) then A; else B;
  - falls die Bedingung in konstanter Zeit ausgewertet werden kann: $T \in O(T_A + T_B) + O(1) = O(\max\{T_A; T_B\})$

- Methodenaufrufe
  - Nicht-rekursiv: jede Methode kann einzeln analysiert werden, Komplexitäten jeweils weiter einsetzen; (zusätzlich konstanten Aufwand)
  - rekursiv: Rekursionsgleichung (später)

Beispiel: Fibonacci-Zahlen

- Fibonacci 1202 (Italien)
- berühmte Kaninchenaufgabe:
  - Start: 1 Paar Kaninchen
  - jedes Paar wirft nach 2 Monaten ein neues Paar
  - dann monatlich jeweils ein weiteres Paar
  - Wie viele Kaninchenpaare gibt es nach einem Jahr, wenn keines der Kaninchen vorher stirbt ? 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233
- Anzahl im n-ten Monat, rekursive Formel: $f(n) = \left\{ \begin{array}{ll} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n > 1 \end{array} \right\}$

Naiver, rekursiver Algorithmus: Definition eins zu eins in ein Programm

```
int Fibonacci(n) {
  if (n <= 1) {
    return n;
  } else {
    return Fibonacci(n-1) + Fibonacci(n-2);
  }
}
```

Laufzeitanalyse Fibonacci

- Komplexität: $T(n) \in O(2^n), T(n) \in \Omega(2^{n/2})$
- Beweis:
  - $T(n)$ positiv: $T(n) > 0 \forall n > 0$
  - $T(n)$ für $n > 2$ streng monoton wachsend: $T(n) < T(n+1)$
  - Abschätzung nach oben: für alle $n > 3$ gilt: $T(n) = T(n-1) + T(n-2) < 2 * T(n-1) < 2 * T(n-2) \ldots 2^{n-1} * T(1) = \frac{1}{2}2^n$, d.h. $T(n) \in O(2^n)$
  - Abschätzung nach unten: für alle $n > 3$ gilt: $T(n) = T(n-1) + T(n-2) > 2 * T(n-2) > 4 * T(n-4) > \ldots > \left\{ \begin{array}{ll} 2^{\frac{n}{2}-1} * T(2) & n \text{ gerade} \\ 2^{\frac{n-1}{2}} * T(1) & n \text{ ungerade} \end{array} \right\}$, d.h. $T(n) \in \Omega(2^{\frac{n}{2}})$.

Iterativer Algorithmus

- keine redundanten Berechnungen
- Programm

```
int Fibonacci (n) {
  int result = 1, previous = 0;
  while (n > 0) {
    int prev = previous;
    previous = result;
    result = result + prev;
    --n;
    return result;
  }
}
```

- Laufzeitanalyse: $d = n, T_S \in O(1) \Rightarrow T(n) \in O(n)$, d.h. linear! enorme Verbesserung

Rekursion versus Iteration

- Vergleich
  - Rekursive Formulierung oft eleganter
  - Iterative Lösung oft effizienter aber komplizierter
- Rekursive Lösungen
  - Rekursion mächtiges allgemeines Programmierungsprinzip

- Jede rekursive Lösung auch iterativ (d.h. mit Schleife) lösbar
- Rekursion ist nicht immer ineffizient: endständige Rekursion

Fibonacci-Zahlen, endständig rekursiv

- Idee:
  - Rekursion ohne Nachklappen, d.h. alte Werte auf dem Stapel werden nicht mehr aufgegriffen
  - führe Ergebnis rekursiv mit
  - n zählt Schritte bis zum Ende
- Programm:

```
int Fibonacci (n) {
  return Fibbonacci (n, 1, 0);
}

int Fibonacci1 (n, result, previous) {
  if (n == 1)
    {
      return result;
    } else {
      return Fibonnaci1 (n-1, result + previous, result);
}
```

- Laufzeit: $O(n)$, d.h. linear

1.2 Datentypen

- Definition: Menge von Werten und Operationen auf diesen Werten
- Elementare (atomare) Datentypen: (JAVA)
  - int: ganze Zahlen 32bit (auch byte (8), short (16), long (64))
  - boolean: true oder false
  - char: Zeichen
  - float: Fließkommazahlen 32bit (double (64))
- Zusammengesetzte Typen:
- Record: Datensatz (in JAVA: class), ggf inhomogen
- Set: Menge (in JAVA: in Bibliotheken vordefiniert)
- Array: Reihung gleichartiger Daten

1.2.1 Objektverweise als Zeiger (Pointer)

- in Java nicht explizit, sondern implizit vorgegeben
- Referenz auf ein (anderes) Objekt
- Besteht aus Speicheradresse des referenzierten Objektes
- Für dynamische Datenstrukturen: Speicher er bei Bedarf belegt
- In einigen Programmiersprachen explizite Speicherfreigabe
- Java arbeitet mit "Garbage-Collection": Falls keine Referenz mehr auf ein Objekt existiert, wird der Speicher freigegeben (zur Wiederverwendung)

1.2.2 Zusammengesetzte Typen: Arrays

- Array: Reihung (Feld) fester Länge von Daten gleichen Typs
  - $a[i]$ bedeutet Zugriff auf das Element Nummer i in a
  - erlaubt effizienten Zugriff auf Elemente: $O(1)$ für $a[i]$
  - wichtig: Array-Grenzen beachten
- Referenz-Typ: int [] a = new int [n]
- Vorsicht: Array a beginnt in Java bei 0 und geht bis a.length-1

Beispiel Arrays

- Sieb des Eratosthenes
  - such nach Primzahlen, kleine n
  - in Array, da Elemente effizient zugreifbar sind
  - Idee: a[i] = true falls i Primzahl, a[i] = false, falls i keine Primzahl
- Ablauf:
  - initialisiere Array-Werte bis n mit true
  - setze Vielfache sukzessive auf false
  - bis Wurzel der Obergrenze
  - Arrayeinträge "true" für Primzahlen bleiben übrig

1.2.3 Mehrdimensionale Arrays

- Zweidimensionale Arrays (=Matrizen) und deren Speicherung

a[0]	a[0][0]	a[0][1]	a[0][2]
a[1]	a[1][0]	a[1][1]	a[1][2]
a[2]	a[2][0]	a[2][1]	a[2][2]
a[3]	a[3][0]	a[3][1]	a[3][2]

- Deklaration
  int [][] a = new int [3][4] // ohne Initialisierung
  int [][] m = 1,2,3,4,5,6; // Initialisierung mit Konstanten

- Höhere Dimensionen: int [][][] g = new int [2][2][2] // 3d

Beispiel: Matrixmultiplikation

```
static float [][] product (float[][] left, float[][] right) {
  float [][] prod = new float [left.length][right[0].length];
    for (int i=0; i < left.length; ++i) {
      for (int j=0; j < right[0].length; ++j) {
        float prod_ij=0;
        for (int k=0; k < right[0].length; ++k) {
          prod_ij = prod_ij + left [i][k] * right [k][j];
        }
        prod [i][j] = prod_ij;
      }
    }
  return prod;
}
```

1.2.4 Benutzerdefinierte Datentypen: Klassen

- Zusammenfassung verschiedener Attribute zu einem Objekt

```
class Time {
  int h, m, s;
}
Time t;
```

```
class Date {
  int day;
  string month;
  int year;
}
Date d;
```

- Beispiel: Rückgabe mehrerer Funktionsergebnisse auf einmal
  - Realisiert als Rückgabe eines einzigen komplexen Ergebnisobjektes

```
static Time convert (int sec) {
  Time t = new Time();
  t.h = sec / 3600;
  t.m = (sec % 3600) / 60;
  t.s = sec % 60;
  return t;
}
```

1.2.5 Klassen vs. Arrays

Klassen:

- bestehen i.a. aus verschiedenartigen Elementen: class c [string s; int i]
- jedes Element hat einen eigenen Namen: c.s; c.i
- Anzahl der Elemente wird statisch bei der Deklaration der Klasse festgelegt

Arrays:

- bestehen immer aus mehreren gleichartigen Elementen: int []
- Elemente haben keinen eigenen Namen, sondern werden über Indices angesprochen: a[i]
- Anzahl der Elemente wird dynamisch (??) bei der Erzeugung des Arrays festgelegt: new int[n]

1.3 Abstrakte Datentypen (=ADT)

- Datenstruktur definiert durch auf ihr zugelassene Methoden
- Spezielle Implementierung nicht betrachtet
- Definition über:
  - Menge von Objekten ("Typen", "Sorten")
  - Methoden auf diesen Objekten → Syntax des Datentyps ("Signatur")
  - Axiome → Semantik des Datentyps
- Top-down Software-Entwurf

- Spezifikation:
  - zuerst "was" festlegen, noch nicht "wie"
  - klarere Darstellung von Programmkonzepten
- Abstraktion in Java:
  - abstract class
  - interface

Beispiel ADT: Boolean

- Wertebereich: { true, false }
- Methoden:

not	boolean → boolean
and	boolean × boolean → boolean
or	boolean × boolean → boolean

- Axiome:

not true	= false;
not false	= true;
x and true	= x;
x and false	= false;
x or true	= true;
x or false	= x;
Beachte:	true ≠ false

- Es gilt: alle Terme, die wir bilden über not, and or, true, false, können auf die Form true oder false gebracht werden.

Beweis:	Induktion über den Termaufbau (=strukturell)	
Induktionsanfang:	true, false	
Induktionsannahme:	Die Terme s, t seien als true oder false darstellbar.	
	not t	not true = false; not false = true;
Induktionsschritt:	t and s	t and true = t; t and false = false;
	t or s	t or true = true; t or false = t;

  q.e.d. ⇒ mit Hilfe der Axiome bewiesen!

1.3.1 Beispiel: Algebraische Spezifikation Stack (Stapel, Keller)

- Sorten (Datentypen)
  - Stack (hier zu definieren)
  - Element ("Elementtyp")
- Operationen:
  - stackinit: "nichts" → stack

- isStackEmpty: Stack → boolean
- push: Element × stack → stack
- pop: stack → Element × stack

• Axiome: Elementtyp x; stack s;

  - pop (push (x, s)) = (x, s);
  - push (pop (s)) = s; // für isStackEmpty (s) = false;
  - isStackEmpty (stackinit) = true;
  - isStackEmpty (push (x, s)) = false;

• Undefinierte Operationen, z.B. Pop (stackinit), erfordern Fehlerbehandlung; formal z.B.

  - durch Vorbedingung ausschließen (pop mit not isStackEmpty (s))
  - Fehlerelement einführen: ”error’ (pop (stackinit) = error)

1.3.2 Potentielle Probleme bei ADTs

- Potentielle Probleme:
  - Anzahl der Axiome kann sehr groß sein
  - Spezifikation ist nicht immer leicht zu verstehen
  - Prüfung, ob vollständig und widerspruchsfrei, ist nicht immer einfach
- Hier nicht weiter betrachtet; bei uns:
  - relativ kleine, kompakte Programme
  - wir betrachten konkrete Implementierungen, die durch die Abstraktion gerade verborgen werden

1.3.3 Dynamische Datenstrukturen

- Motivation
  - Länge eines Arrays ist nach der Erzeugung festgelegt
  - hilfreich wären unbeschränkt große Datenstrukturen
  - Lösungsidee: Verkettung einzelner Objekte zu größeren Datenstrukturen
- Beispiele:
  - Liste
  - Baum
  - allgemeiner Graph

- Charakterisierung:
  - Knoten zur Laufzeit (dynamisch) erzeugt und verkettet
  - Strukturen können dynamisch wachsen oder / und schrumpfen
  - Größe einer Struktur nur durch verfügbaren Speicherplatz beschränkt; muß nicht im Vorhinein bestimmt werden
- z.B. Listen, Stacks, Schlangen, Bäume
  - "Containertypen", in der Regel für die Speicherung von Elementen eines bestimmten Typs
- Listen:
  - Definition: Sequenz (Folge) von Elementen
  - Operationen: insert, delete, read
  - Verkettete Liste: (mit Objektverweisen)
    * Definition: Menge von Elementen; jedes Element ist Teil eines Knotens (node), der neben dem Inhalt (hay) einen Verweis auf einen weiteren Knoten enthält
    * leichtes Umsortieren
    * schlechterer beliebiger Zugriff als beim Array ("random access")

Implementierung: Verkettete Liste

```
class List {
  Object first;
  List remainder;
}
```

- Deklaration und Initialisierung: List Head = null; Head = new List ();
- Alternativen:
  - Anfang: zusätzlicher Leerknoten
  - zwei Knoten (Anker): head, z
- Bei Implementierung der Operationen genaue Darstellung beachten! (List head, end;)

1.3.4 Verkettete Liste

- Löschen: Knoten nach x (=x.remainder):
  - x.remainder=x.remainder.remainder;

- Knoten ohne Referenz wird in Java per garbage collection entfernt (in anderen Sprachen muß selbst aufgeräumt werden)

- Einfügen: Knoten t als Nachfolger von Knoten x:
  - t.remainder=x.remainder; x.remainder=t;
- Achtung: Am Anfang und Ende der Liste aufpassen!

Implementierung von Listen = Alternative Implementierungen ohne Objektverweise

- Sequentielles Array
- Cursor-Darstellung Idee: Zweites Array für Indizes der Nachfolgelemente

```
class ListC {
  int first, last, lastOccupied;
  Object [] Key;
  int [] next;

  ListC (capacity) {
    key = new Object [capacity + 2];
    next = new int [capacity + 2];
    first = 0; last = 1; lastOccupied = 1;
    next[first] = last; next[last] = last;
  }

  DeleteNext (t) {
    next[t] = next[next[t]];
  }

  InsertAfter (v,t) {
    lastOccupied = lastOccupied + 1;
    key[lastOccupied] = v;
    next[lastOccupied] = next[t];
    next[t] = lastOccupied;
  }
}
```

- Vorsicht:
  - kein Test, ob Array bereits voll
  - freigegebene Positionen werden nicht wiederverwendet!

Vor- und Nachteile der Implementierungen

- Sequentielle Speicherung in einem Array
  - vorgegebene maximale Größe
  - Aufwand für "Insert" und "Delete": O(n) bei n Einträgen
  - nur für "statische" Daten gut geeignet, d.h. wenig Einfügungen und Entfernungen
- Cursor-Darstellung
  - vorgegebene maximale Größe
  - Aufwand für Einfügen und Entfernen: O(1)
  - Spezielle Freispeicherverwaltung erforderlich (bei vielen Entfernungen)
- Pointer-Implementierung (d.h. mit Objektverweisen)
  - maximale Anzahl der Elemente ist nur durch Speichergröße begrenzt
  - Freispeicherverwaltung wird vom System übernommen
  - Fehlererkennung schwierig

1.3.5 Doppelt verkettete Liste

- Flexiblerer Zugriff (vorwärts, rückwärts)
- Bei Änderungen: (Einfügen, Entfernen) müssen jeweils beide Verkettungen aktualisiert werden
-
  ```
  class List () {
    Object key;
    List prev, next;
    List head;
  }
  ```
- Einfügen: Knoten t nach x:

  ```
  t.next = x.next;
  t.prev = x;
  t.prev.next = t;
  t.next.prev = t;
  ```
- Löschen: Knoten x

  ```
  x.prev.next = x.next;
  x.next.prev = x.prev;
  ```

1.3.6 Stacks

- "Stapel" von Elementen ("Kellerspeicher")
- Wie Liste: Sequentielle Ordnung, aber nur Zugriff auf 1. Element
  - push: neues Element oben einfügen
  - pop: oberstes Element auslesen
- LIFO-Struktur (Last In First Out)
- ```
  Class Node () {
    Object key;
    Node next;
  }
  ```

Implementierung mit Pointern

```
class Stack {
  Node head;

  Stack () {
    head = null;
  }

  void Push (Object t) {
    Node x = new Node ();
    x.key = t;
    x.next = head;
    head = x;
  }

  Object Pop () {
    if (isEmpty ()) ... /* Fehlerbehandlung */
    Node x = head;
    head = x.next;
    return x.key;
  }

  boolean isEmpty () {
    return head == null;
  }
}
```
```

Implementierung mit Arrays

```
Class StackA {
  int top;
  Object [] stackA;

  StackA (int capacity) {
    top = 0;
    stackA = new Object (capacity);
  }

  void Push (v) {
    if (isFull ()) ... /* Fehlerbehandlung */
    stackA [top] = v;
    top = top + 1;
  }

  Object Pop () {
    if (isEmpty ()) ... /* Fehlerbehandlung */
    top = top - 1;
    return stackA [top];
  }

  boolean isEmpty () {
    return top == 0;
  }

  boolean isFull () {
    return top >= stackA.length;
  }
}
```

1.3.7 Queues (Warteschlangen / Schlangen)

- Wie Liste: sequentielle Ordnung, aber:
  - Einfügen: neues Elemente am Ende anhängen (put)
  - Auslesen: vorderstes Element zurückgeben (get)
  - FIFO: First In First Out
- Implementierung als zyklisches Array:
  - Spart Speicherplatz
  - beschränkte Länge
  - sehr flexibel, für dynamische Datenmengen geeignet

Implementierung mit Array

```
Class Queue {
  Object [] queue;
  int head, tail;

  Queue (int capacity) {
    queue = new Object [capacity];
    head = 0;
    tail = 0;
  }

  Put (Object v) {
    if (ifFull ()) ... /* Fehlerbehandlung */
    queue [tail] ) v;
    tail = tail + 1;
    if (tail >= queue.length {
      tail = 0;
    }
  }

  Object Get () {
    if (isEmpty ()) ... /* Fehlerbehandlung */
    t = queue [head];
    head = (head + 1) % queue.length; /* das gibt einen integerüberlauf */
    return t;
  }

  boolean isEmpty () {
    return (head == tail);
  }

  boolean isFull () {
    return (head == (tail + 1) % queue.length);
  }

}
```

1.3.8 Bäume

- Menge von Knoten
- Relation, die Hierarchie definiert
  - jeder Knoten (außer der Wurzel) hat einen Elternknoten (unmittelbar voran-

gehender Knoten)
  - Kante drückt eine Beziehung zwischen zwei direkt aufeinanderfolgenden Knoten aus

- Grad eines Knotens: Zahl der unmittelbaren Nachfolger
- Blatt: Knoten ohne Nachfolger
- Geschwister: Direkte Nachfolger desselben Elternknotens
- Pfad: Folge von Knoten, die jeweils direkt Nachfolger voneinander sind
- Pfadlänge: Anzahl der Knoten im Pfad
- Höhe eines Knotens: Länge des Pfades ohne Wurzel
- Höhe eines Baums: Maximale Pfadlänge im Baum
- Grad eines Baums: Maximaler Grad eines Knotens
- Alternative Definition Pfadlänge: Anzahl der Kanten
- Binärbaum: Ein Baum mit Grad 2

Beispiel Baum: Darstellung als Graph

Alternative Baumdarstellungen

1.3.9 Baum

- Baumdefinition induktiv (=rekursiv):
  - ein einzelner Knoten ist ein Baum (zugleich Wurzel)
  - sei n ein Knoten, $T_1, T_2, \ldots, T_k$ Bäume mit Wurzelknoten $n_1, n_2, \ldots, n_k$, dann ist $n, T_1, \ldots, T_k$ ein Baum
- Maximaler Baum:
  - Baum mit maximaler Knotenzahl $N(h, d)$: d.h. gegeben sei Höhe h und Grad d; gesucht; Wieviele Knoten kann der Baum maximal haben ?
    1. Ebene: 1 Knoten (Wurzel)
    2. Ebene: d Knoten
    3. Ebene: d^2 Knoten

    $N(h,d) = 1 + d + d^2 + \ldots + d^{h-1} = \sum\limits_{i=0}^{h-1} d^i = \frac{d^h-1}{d-1}$ (geometrische Reihe)

Minimale, maximale Baumhöhe

- T Baum vom Grad $d \geq 2, n$ Knoten:
  - Maximale Höhe: $n-1$
  - Minimale Höhe: $\lceil \log_d((n-(d-1)+1) \rceil \approx log_d(n)$
- Beweis:
  - Maximale Höhe: n Knoten untereinander
  - Minimale Höhe: vgl. Formel maximaler Baum bei gegebener Höhe: maximale Anzahl Knoten: $N(h-1,d) < n \leq N(h,d) \Leftrightarrow N(h-1,d) < n \leq N(h,d) \Leftrightarrow \frac{d^{h-1}-1}{d1} < n \leq \frac{d^h-1}{d-1} \Leftrightarrow d^{h-1} < n(d-1) \leq d^h - 1 \Leftrightarrow d^{h-1} < n(d-1)+1 \leq d^h \Leftrightarrow h-1 < \log_d(n(d-1)+1) \leq h$ wegen $n(d-1)+1 \leq nd \forall n > 1$ folgt: $h = \lceil \log_d(n(d-1)+1) \rceil \leq \lceil \log_d(nd) \rceil = \lceil \log_d(n) \rceil + 1$
    Für Binärbäume (d=2): $h \leq \lceil \log_2 n \rceil$

n	h
1	1
3	2
7	3
15	4
1023	10
1Mio	20
1Mia	30

Implementierung von Binärbäumen

```
class Tree {
  Object key;
  Tree left, right;
}
```

- Verankerung:
  - Baum ist "Tree"-Objekt:

    ```
    Tree root;
    ```

  - leere Teilbäume durch null markieren

Array-Einbettung

- Am besten für vollständige Bäume:
  - alle Ebenen bis auf letzte voll

- letzte Ebene von links nach rechts gefüllt

- Als Array:
  - Kindknoten zu Knoten i an Positionen $2i$ und $2i+1$
  - Wurzel steht an Position 1
  - geht auf für andere Verzweigungsgrade
  - vollständiger Baum $\Rightarrow$ lückenloses Array

Baumdurchläufe

- Tiefendurchlauf (depth first):
  Durchlaufe zu jedem Knoten rekursiv die Teilbäume (von links nach rechts)
  - Preorder / Präfix:
    notiere erst einen Knoten, dann seine Teilbäume
  - Postorder / Postfix:
    notiere erst Teilbäume eines Knotens, dann den Knoten selbst
  - Inorder / Infix (nur für Binärbäume):
    notiere erst linken Teilbaum, dann den Knoten, dann den rechten Teilbaum
- Breitendurchlauf (breadth first):
  durchlaufe Knoten ebenenweise (von links nach rechts)
  - Preorder: 38, 18, 11, 23, 32, 42, 39, 10, 45
  - Postorder: 11, 23, 18, 32, 39, 10, 45, 42, 37
  - Breitendurchlauf: 37, 18, 32, 42, 11, 23, 39, 10, 45

Rekursiver Durchlauf für Binärbäume

hier: Tiefendurchläufe

```
Preorder (node) {
  if (node != null {
    eval (node);
    Preorder (node.left);
    Preorder (node.right);
  }
}

Postorder (node) {
  if (node != null) {
    Postorder (node.left);
    Postorder (node.right);
    eval (node);
```

```
  }
}

Inorder (node) {
  if (node != null) {
    Inorder (node.left);
    eval (node);
    Inorder (node.right);
  }
}
```

Nicht-rekursiver Durchlauf für Binärbäume

```
PreorderStack (node) {
  Stack stack = new Stack ();
  stack.push (node);
  while (!stack.isEmpty()) {
    Node current = stack.pop ();
    if (current != null) {
      eval (node);
      stack.push (current.right);
      stack.push (current.left);
    }
  }
}

BreadthQueue (node) {
  Queue queue = new Queue;
  queue.put (node);
  while (!queue.isEmpty()) {
    Node current = queue.get ();
    if (current != null) {
      eval (current);
      queue.put (current.left);
      queue.put (current.right);
    }
  }
}
```

1.4 Entwurf von Algorithmen

- Bewährte Vorgehensweisen, um Algorithmen zu entwerfen
- Beispiele:

- Divide-and-Conquer
  - Dynamische Programmierung
  - Branch-and-Bound
  - Plane-Sweep

- Divide-and-Conquer (divide et impera)
  - Ansatz: Zerlege Problem in Teilprobleme
  - Man erhält typischerweise rekursive Algorithmen
  - Divide: Zerlege das Problem, bis hin zu elementaren Problemen
  - Conquer: Löse elementare Probleme
  - Merge: Setze die Teillösungen zu Gesamtlösung zusammen

1.4.1 Beispiel: MergeSort

- Problem: Sortiere Folge von Zahlen (Strings, ...)
- Idee:
  - Divide: Zerlege die Folge in zwei gleich große Teilfolgen, rekursiv bis Teilfolgen der Länge 1
  - Conquer: Trivial, Folgen der Länge 1 sind bereits sortiert
  - Merge: Verschmelze sortierte Teilfolgen sukzessive zu sortierter Gesamtfolge

MergeSort Implementierung

```
MergeSort (Array a, left, right) {
  if (left < right) {
    middle = (left + right) / 2;     /* divide  */
    MergeSort (a, left, middle);     /* conquer */
    MergeSort (a, middle+1, right);
    Merge (a, left, middle, right);  /* merge   */
  }
}
```

- Schnittstelle für die Benutzung

```
MergeSort (Array a) {
  MergeSort (a, 0, a.length-1);
}
```

```
Merge (Array a, left, middle, right) {
  int [] b = new int [right - left + 1];
  int i = 0,
    j = left,
    k = middle + 1;
  while (j <= middle and k <= right) {
    if (a[j] <= a[k]) {
      b[i] = a[j]; j++;
    } else {
      b[i] = a[k]; k++;
    }
  }
  i++;

  /* j > middle or k > right */
  while (j <= middle) {
    b[i++] = a[j++];
  }
  while (k <= right) {
    b[i++] = a[k++];
  }
  for (i=left; i <= right; i++) {
    a[i] = b[i-left];
  }
}
```

1.4.2 Dynamische Programmierung

- Optimierungsaufgabe: Optimiere eine Zielfunktion unter Beachtung von Nebenbedingungen
- Grundprinzip:
  - Finde optimale Lösungen für "kleine" Elementarprobleme
  - Konstruiere sukzessive "größere" Lösungen bis zur Gesamtlösung
- Bottom-up-Strategie (divide-and-conquer: Top-down)
- Schritte:
  - Teilprobleme bearbeiten
  - Teilergebnisse in Tabellen eintragen
  - Zusammensetzen der Gesamtlösung
- Anwendungsbedingungen:

- Optimale Lösung enthält optimale Teillösungen
- überlappende Teillösungen

Beispiel: Matrix-Kettenprodukte

- Problem: gegeben 2 reellwertige Matrizen, berechne ihr Produkt
  $A = B * C, \quad B \in R^{l \times m}, C \in R^{m \times n}$
  $a_{i,k} = \sum_{j=1}^{m} b_{i,j} c_{j,k}$
- Zur Berechnung lmn Multiplikationen und Additionen erforderlich
- Bei mehreren Matrizen:
  - $M_1 * M_2 \cdots M_N$
  - Wert nich abhängig von der Art der Klammerung (wg. Assoziativgesetz)
  - Rechenaufwand sehr wohl von der Klammerung abhängig
    $M_1(M_2(M_3M_4)) = (M_1(M_2M_3))M_4$
    $10 \times 20 \quad 20 \times 50 \quad 50 \times 1 \quad 1 \times 100$
    125.000 Operationen vs. 2.200 Operationen

Matrixprodukt: Optimierungsproblem

- Gegeben: Folge natürlicher Zahlen, die die Dimensionen von N Matrizen beschreibt:
  Z.B. $< 10, 20, 50, 1, 100 >$
- Gesucht: optimale Klammerung, die die Anzahl der skalaren Multiplikationen minimiert
- Formale Beschreibung:
  - C_{ij}: Minimale Kosten der Multiplikation von Teilprodukten $M_i \cdots M_j$
  - r_i: Dimensionen der Matrizen
  - Zerlegungsbreite:
    $\underbrace{(M_i \cdots M_K)}_{r_{i-1} \times r_k = C_{i,k}} \underbrace{(M_{k+1} \cdots M_j)}_{r_k \times r_j = C_{k+1,j}}, \quad (i \leq k < N)$
    ergibt Rekursionsgleichung:
    $$C_{i,j} = \begin{cases} 0 & j = i \\ \min\limits_{i \leq k < j} C_{i,k} + C_{k+1,j} + r_{i-1} r_k r_j & j > i \end{cases}$$

Matrixprodukt: Berechnung des Optimums

- Naive Lösung: Rekursives Programm aus Formel
- Komplexität: Exponentiell: $O(n^n)$

- Beobachtung:
  - Mehrfachberechnungen, z.B. $C_{1,1}$ auch im Zweig $C_{1,2}, C_{1,4}$
- Idee: Bottom-Up- statt Top-Down-Berechnung

Dynamische Programmierung: Matrixprodukt

$$C_{i,j} = \left\{ \begin{array}{ll} 0 & j = i \\ \min\limits_{i \leq k < j} C_{i,k} + C_{k+1,j} + r_{i-1} r_k r_j & j > i \end{array} \right\}$$

Auswertung: Matrixprodukt

- Ansatz:
  - Zwischenlösungen in Tabelle speichern
- Reihenfolge der Auswertung der $C_{i,j}$:
  - beginne mit einzelnen Matrizen M_{ii}
  - dann betrachte die Produkte zweier Matrizen
  - dann 3
- Algorithmus:
  - Tabelle für C_{ij} wird sukzessive von der Diagonale aus gefüllt
  - Endergebnis steht dann in $C_{1,n}$
  - Protokolliere Splitstellen ("k") in Matrix S

Implementierung Matrixprodukt

```
MatrixProduct (Array r /* in */, Matrix C /* out */, Matrix S /* out */) {
  /* Diagonale mit Nullen füllen: */
  for (i=1; i<=n; ++i) c[i][i]=0;

  /* andere Felder sukzessive von der Diagonale aus füllen */
  for (j=0; j<=n; ++j) {     /* erste bis letzte Spalte */
    for (i=j-1; i>=1; --i) { /* Zeilen von der Diagonalen aus nach oben */
      minval = MAXINT;
      minpos = i;
      for (k=i; k<j; ++k) {
        tmpval = C[i][k] + C[k+1][j] + r[i-1] * r[k] * r[j];
        if (tmpval < minval) {
          minval = tmpval;
          minpos = k;
        }
```

```
      }
      C[i][j] = minval;
      S[i][j] = minpos;
    }
  }
  /* C[1][n] enthält die gesuchten Kosten für das gesamte Matrixprodukt */
}
```

1.4.3 Rekursionsgleichungen: Sukzessiv einsetzen

- Zur Laufzeitanalyse (insbes. rekursiver) Algorithmen
- Sukzessives Einsetzen: Sei n eine Zweierpotenz
  - Bsp:
    $T(n) = T(n-1) + n \quad n > 1, T(1) = 1$
    $= T(n-2) + (n-1) + n = \ldots = T(1) + 2 + \ldots + (n-2) + (n-1) + n = \frac{n*(n+1)}{2}$
  - Bsp: $T(n) = T(\frac{n}{2}) + 2, \quad n > 1, T(1) = 0$
    $= T(\frac{n}{4}) + \frac{n}{2} + n = T(\frac{n}{8}) + \frac{n}{4} + \frac{n}{2} + n = \ldots = T(1) + \ldots + \frac{n}{8} + \frac{n}{4} + \frac{n}{2} + n = 2*(n-1)$

1.4.4 Rekursionsgleichungen: Masterproblem

- Für Rekursionsgleichungen der Form:
  $T(n) = \left\{ \begin{array}{ll} c & n = 1 \\ a * T(\frac{n}{b}) + c * n & n > 1 \end{array} \right\}$
  - für $a \geq 1, b > 1, c \geq 0$
  - statt $T(\frac{b}{n})$ auch $T(\lceil \frac{n}{b} \rceil)$ oder $T(\lfloor \frac{n}{b} \rfloor)$
- Dann gilt:
  $T(n) = \left\{ \begin{array}{ll} O(n) & a < b \\ O(n \log n) & a = b \\ O(n^{\log_b a}) & a > b \end{array} \right\}$ z.B. MergeSort: $a = 2, b = 2$

Master Theorem

- Beweis: sei $n = b^k \Leftrightarrow k = \log_b n$
- $T(n) = a*T(\frac{n}{b}) + c*n = a^2*T(\frac{n}{b^2}) + a*c*\frac{n}{b} + c*n = a^3*T(\frac{n}{b^3}) + a^2*c*\frac{n}{b^2} + ac\frac{n}{b} + cn =$
  $\ldots = c * n * \sum_{i=0}^{k} \left(\frac{a}{b}\right)^i =$ geometrische Reihe
- Fallunterscheidung:
  $a < b \quad T(n) \leq c * n * \frac{1}{1-\frac{a}{b}} = O(n)$
  $a = b \quad T(n) = c * n * (k+1) = O(n * k) = O(n \log n)$
  $a > b \quad T(n) = c * n * \frac{\left(\frac{a}{b}\right)^{k+1} - 1}{\frac{a}{b} - 1} = c * n * O(\left(\frac{a}{b}\right)^n) = c * n * O(\left(\frac{a}{b}\right)^{\log_b n}) = c * n * O(\frac{a^{\log_b n}}{n} = O(a^{\log_b n}) =$

Master Theorem erweitert

- Rekursion:
  $T(n) = \left\{ \begin{array}{ll} 1 & n = 1 \\ a * T(\frac{n}{b}) + d(n) & n > 1 \end{array} \right\}$ für positive Funktion $d(n) \in O(n^\gamma), \gamma > 0$

- Dann: $T(n) = \left\{ \begin{array}{ll} O(n^\gamma) & a < b^\gamma \\ O(n^\gamma \log_b n) & a = b^\gamma \\ O(n^{\log_b a}) & a > b^\gamma \end{array} \right\}$

- Weitergehende Formen in der Literatur (vgl. Cormen et al. 73ff.)

Rekursionsgleichungen: Rekursionsbäume

Idee: Formel intuitiv aus Baum entwickeln (und dann beweist)
$T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + O(n)$. Das Mastertheorem ist hier nicht anwendbar.
$\Sigma : O(n \log n)$ – kann mit sukzessivem Einsetzen verifiziert werden.

1.4.5 Rekursionsungleichungen

- Rekursionsgleichungen:
  - oft schwierig direkt zu lösen
- Benutzte Ungleichungen:
  - Abschätzungen der Rekursion nach oben oder unten, um obere bzw. untere Schranken zu erhalten
- Auch möglich für Θ:
  - falls "enge" Abschätzungen nach oben und unten (getrennt voneinander) gefunden werden, dann mittels $\Theta(n) = O(n) kopf_u \Omega(n)$ gezeigt.
  - Bsp: $f(n) \in O(n^2), f(n) \in \Omega(n^2) \Rightarrow f(n) = \Theta(n^2)$

Konstruktive Induktion

- Idee: Bestimme noch unbekannte Konstanten im Verlauf des Induktionsbeweises
- Bsp: Fibonacci-Zahlen: $f(n) = f(n-1) + f(n-2), \quad n > 1$
- Behauptung: Für $n \geq n_0$ gilt: $f(n) \geq a * c^n$ mit Konstanten $a, c > 0$, die noch zu bestimmen sind.
- Induktionsschritt: $f(n) = f(n-1) + f(n-2) \geq a * c^{n-1} + a * c^{n-2}$
- Konstruktiver Teil: Bestimme ein $c > 0$ mit $a * c^{n-1} + a * c^{n-2} \geq ac^n$. Durch Umformung: $c+1 \geq c^2 \Rightarrow c^2 - c - 1 \leq 0$. Ergibt Lösung: $c_{1,2} = \frac{1}{2} \pm \sqrt{\frac{1}{4} + 1} = \frac{1}{2} \pm \frac{\sqrt{5}}{2}$, also $c \leq \frac{1+\sqrt{5}}{2} \approx 1,61803\ldots$. Wähle a, n_0 geeignet.

- Induktionsanfang:

Substitution

- Substitution: Schwierige Ausdrücke auf Bekanntes zurückführen
- Bsp: $T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \log_2 n$
  - Substituiere $m = \log_2 n$
  - Ergibt $T(2^m) = 2T(\lfloor 2^{\frac{m}{2}}\rfloor) + m$
  - Setze $S(m) = T(2^m)$
  - Ergibt: $S(m) = 2 * S(\frac{m}{2}) + m$
  - mit einfacher Lösung (Master Theorem) $S(m) = O(m * \log_2 m)$
  - Rücksubstituieren: $T(n) = T(2^m) = S(m) = O(m * \log_2 m) = O(\log_2 n \log_2 \log_2 n)$

1.4.6 Rekursionsbeweise

- Beliebte Fehlerquelle:
  - Beispiel: Zu zeigen $T(n) \in O(n)$
    $T(n) \leq 2 * (c * \frac{n}{2}) + n \leq cn + n \leq O(n)$ FALSCH!
  - Fehler liegt darin, daß nicht $T(n) \in O(n)$ bewiesen wurde, denn das würde voraussetzen, daß $T(n) \leq cn$ und nicht nur $T(n) \leq cn + n = (c+1)n$
  - Richtige Lösung: s. Übungblatt 4

2 Sortieren

2.1 Voraussetzungen

- Für nachfolgende Betrachtungen sei stets eine Folge $a[1], \dots, a[n]$ von Datensätzen gegeben.
- Jeder Datensatz $a[i]$ besitzt eines Schlüsselkomponente $a[i].key \quad (i = 1, \dots, n)$
- Datensätze können außer der Schlüsselkomponente weitere Attribute enthalten (z.B. Name, Adresse, PLZ, etc.)
- Sortierung erfolgt ausschließlich nach der Schlüsselkomponente key
- Für Sortierung muß auf der Menge aller Schlüssel eine totale Ordnung definiert sein.

2.1.1 Partielle Ordnung

Es sei M eine nicht-leere Menge und $'' \leq'' \supseteq M \times M$ eine binäre Relation auf M. Das Paar $(M, \leq)$ heißt eine partielle Ordnung auf M genau dann, wenn $\leq$ die folgenden Eigenschaften erfüllt:

1. Reflexivität: $\forall x \in M : x \leq x$
2. Transitivität: $\forall x, y, z \in M : x \leq y \wedge y \leq z \Rightarrow x \leq z$
3. Antisymmetrie: $\forall x, y \in M : x \leq y \wedge y \leq x \Rightarrow y = x$

2.1.2 Strikter Anteil einer Ordnungsrelation

Für eine partielle Ordnung $\leq$ auf einer Menge M definieren wir die Relation $<$ durch:

$x < y := x \leq y \wedge x \neq y$

Die Relation $<$ heißt auch der strikte Anteil von $\leq$.

2.1.3 Totale Ordnung

Es sei M eine nicht-leere Menge und $\leq \supseteq M \times M$ eine binäre Ordnung über M. $\leq$ heißt eine totale Ordnung auf M genau dann, wenn gilt: $(M, \leq)$ ist eine partielle Ordnung und Trichotomie:

$\forall x, y \in M : x < y \vee x = y \vee y < x$

Beispiel

Lexikographische Ordnung $<_{lex}$: Sei $<$ die totale Ordnung auf den Buchstaben:

$$A < B < C < \ldots < Y < Z$$

Für zwei endliche Wörter $u = u_1 u_2 \ldots u_m$ und $v = v_1 v_2 \ldots v_n$ mit $u_i, v_j \in \{A, B, \ldots Z\}$ gilt $u <_{lex} \Leftrightarrow$ u leer oder (u, v nicht-leer und ($u_1 < v_1$ oder $u_1 = v_1$ und $u_2 \ldots u_m <_{lex} v_2 \ldots v_n$)).

2.2 Sortierproblem

Gegeben sei eine Folge $a[1], \ldots, a[n]$ von Datensätzen mit einer Schlüsselkomponente $a[i].key \quad (i = 1, \ldots, n)$ und eine totale Ordnung $<$ auf der Menge aller Schlüssel.
Das Sortierproblem besteht nun darin, eine Permutation π der ursprünglichen Folge zu bestimmen, so daß gilt:

$$a[\pi_1].key \leq a[\pi_2].key \leq \ldots \leq a[\pi_{n-1}].key \leq a[\pi_n].key$$

2.2.1 Beispiel

Liste	Schlüsselelement	Ordnung
Telefonbuch	(Nachname, Vorname)	lexikographische Ordnung
Klausurergebnisse	Punktezahl	$<$ auf den Zahlen
Lexikon	Stichwort	lexikographische Ordnung
Studentenverzeichnis	Matrikelnr.	$<$ auf N
Entfernungstabelle	Distanz	$<$ auf R
Fahrplan	Abfahrtszeit	"früher als"

2.2.2 Unterscheidungskriterien

Sortieralgorithmen können nach verschiedenen Kriterien klassifiziert werden:

- Berechnungsaufwand (Komplexitätsklasse): $O(n^2), O(n \log n), O(n)$
- Worst-Case vs. Average-Case
  - Ausnutzung von Vorsortierungen
- Speicherbedarf
  - In-place
  - Kopieren
- Stabilität
  - Reihenfolge von Datensätzen mit gleichem Schlüssel bleibt erhalten (oder auch nicht)

2.2.3 Weitere Aufgaben

Viele Aufgaben sind mit dem Sortieren verwandt und können auf das Sortierproblem zurückgeführt werden:

- Bestimmung des Median (Median: Element an der mittleren Position einer sortierten Folge)
- Bestimmung der k kleinsten bzw. der k größten Elemente (z.B. Internetsuchmaschinen)

2.2.4 Typen von Sortieralgorithmen

Einfache:

- SelectionSort
- BubbleSort
- InsertionSort

Höhere:

- QuickSort
- MergeSort
- HeapSort

Spezielle:

- BucketSort

2.3 Elementare Sortierverfahren

2.3.1 Swap

Im Folgenden wird häufiger der Aufruf "swap (a, i, j)" verwendet, wobei a ein Array ist und i, j integer-Werte.
Der Aufruf ersetzt folgende drei Zuweisungen:

- $h = a[i];$
- $a[i] = a[j];$
- $a[j] = h;$

2.3.2 SelectionSort

1. Suche kleinstes Element
2. Vertausche es mit dem Datensatz an der ersten Stelle
3. Wende den Algorithmus auf die restlichen $n - 1$ Elemente an

Beispiel

```
3 7 8 6 4 2
2 7 8 6 4 3
2 3 8 6 4 7
2 3 4 6 8 7
2 3 4 6 8 7
2 3 4 6 7 8
```

SelectionSort - Implementierung

```
static void selection (int [] a) {
  for (int i=0; i <= a.length-2; i++) {
    /* Minimumsuche */
    int min = i;
    for (int j=i+1; i <= a.length-1; i++) {
      if (a[j] < a[min]) { min = j; }
    }
    /* Vertauschen */
    swap (a, i, min);
  }
}
```

SelectionSort - Komplexitätsanalyse

- Zum Sortieren der gesamten Folge $a[1], \ldots, a[n]$ werden $n-1$ Durchläufe zur Minimumsuche benötigt
- Pro Schleifendurchlauf i gibt es eine Vertauschung, die sich aus je einer Ausführung von swap und $(n-i)$ Vergleichen zusammensetzt. ($(n-i)$ ist die Anzahl der noch nicht sortierten Elemente)
- Insgesamt ergeben sich: $(n-1)$ swaps $= O(n)$ und $(n-1)+(n-2)+\ldots+2+1 = \frac{n(n-1)}{2}$ Vergleiche $= O(n^2)$
- Anzahl der Vertauschungen wächst nur linear mit der Anzahl der Datensätze $\Rightarrow$ besonders für Sortieraufgaben mit sehr großen Datensätzen geeignet

2.3.3 InsertionSort

InsertionSort - Prinzip

- Vorbild: Manuelles Sortieren
- ($n-1$ Schritte für $i = 2, \ldots, n$

- im i-ten Schritt:
  Füge den Datensatz $a[i]$ an der korrekten Position der bereits sortierten Teilfolge $a[1] \dots a[i-1]$ ein.

InsertionSort - Beispiel

```
3 7 8 6 4 2
3 7 8 6 4 2
3 7 8 6 4 2
3 6 7 8 4 2
3 4 6 7 8 2
2 3 4 6 7 8
```

InsertionSort - Implementierung

```
static void insertionSort (int [] a) {
  for (int i=1; i < a.length; i++) {
    int v = a[i];
    int j = i;
    while ((j >= 0) && (a[j-1] > v)) {
      a[j] = a[j-1];
      --j;
    }
    a[j] = v;
  }
}
```

- Variante: Um den Test der Laufvariable j auf die linke Arraygrenze zu vermeiden, kann man ein Sentinel-Element verwenden.
  Sentinel: Wörter, Anfangs- oder Endmarkierung (z.B. $-\infty$)
- Aufruf mit:
  - Array um 1 verlängern, der Inhalt liegt dann in $a[1], \dots, a[n]$
  - $a[0] = -\infty$

InsertionSort - Komplexitätsanalyse

Vergleiche Das Einfügen des Elements $a[i]$ in die bereits sortierte Anfangsfolge $a[1], \dots, a[i-1]$ erfordert mindestens einen Vergleich und höchstens $i-1$ Vergleicher. Im Mittel sind das $\frac{i}{2}$ Vergleiche, denn bei Gleichverteilung der Schlüssel ist die Hälfte der bereits eingefügten Elemente größer als das Element $a[i]$.

- Best Case
  Bei vollständig vorsortierten Folgen: $n-1$ Vergleiche

- Worst Case
  Bei umgekehrt sortierten Folgen gilt für die Anzahl der Vergleiche $\sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \in O(n^2)$
- Average Case
  Die Anzahl der Vergleiche ist etwa $\frac{n^2}{4} \in O(n^2)$.

Bewegungen

- Best Case: Bei vollständig vorsortierten Folgen: $2(n-1)$ Bewegungen (mind. 2 Bewegungen für den i-ten Schritt erforderlich. Durch zusätzliche if-Abfragen auch weniger Bewegungen möglich, dafür aber mehr Vergleiche.
- Worst Case: Bei umgekehrt sortierten Folgen: $\frac{n^2}{2}$ Bewegungen.
- Average Case: Ungefähr $\frac{n^2}{4}$ Bewegungen.

2.3.4 BubbleSort

Prinzip

- $n-1$ Schritte, $i = n, n-1, \ldots, 2$
- im i-ten Schritt:
  $n-i$ Schritte, $j = 1, \ldots, i$, ordne Paar $a[j-1], a[j]$

BubbleSort - Beispiel

```
3 7 8 6 4 2
3 7 6 4 2 8
3 6 4 2 7 8
3 4 2 6 7 8
3 2 4 6 7 8
2 3 4 6 7 8
```

BubbleSort - Implementierung

```
static void bubble (int [] a) {
  for (int i=a.length-1; i>=1; i--) {
    for (int j=1; j<=i; j++) {
      if (a[j-1] > a[j]) {
        swap (a, j-1, j);
      }
    }
  }
}
```

BubbleSort - Komplexitätanalyse

Vergleiche Anzahl der Vergleiche ist unabhängig vom Grad der Vorsortierung der Folge. Daher sind worst case, best case und average case gleich. Es werden immer alle Elemente der noch nicht sortierten Teilfolge miteinander verglichen. Im i-ten Schleifendurchlauf: $n - i + 1$ Elemente, dafür $n - i$ Vergleiche. Insgesamt:

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

Bewegungen / Swaps Aus der Analyse der Bewegungen für den gesamten Durchlauf ergeben sich:

- Best Case: 0 Vertauschungen
- Worst Case: $\sim \frac{n^2}{2}$ Vertauschungen
- Average Case: $\sim \frac{n^2}{4}$ Vertauschungen

2.3.5 Indirektes Sortieren - Motivation

Für Programmiersprachen, in denen Arrays über zusammengesetzten Typen als unmittelbare Reihung (ohne Indirektion) gespeichert werden:
Ziel: Reduziere die Anzahl der Vertauschungen für große Datensätze.

Indirektes Sortieren - Prinzip

1. Verwende ein Index-Array $p[1 \ldots n]$, das mit $p[i] = i, \quad i = 1, \ldots, n$, initialisiert wird.
2. Für Vergleiche erfolgt der Zugriff auf einen Datensatz mit $a[p[i]]$.
3. Vertauschen der Indizes $p[i]$ statt der Array-Elemente $a[p[i]]$.
4. Optional werden nach dem Sortieren die Datensätze selbst umsortiert. Aufwand: $O(n)$

Sollen die Datensätze im Anschluß an die indirekte Sortierung selbst umsortiert werden, so gibt es hierzu zwei mögliche Varianten:

1. Permutation mit zusätzlichem Array b:
   $for(i = 1 \ldots n); b[i] = a[p[i]]$
2. Permutation ohne zusätzliches Array (in situ, in place), lohnt nur bei großen records:
   Ziel: $p[i] = i, \quad (i = 1, \ldots, n)$

- falls $p[i] = i$: nichts zu tun
- sonst: zyklische Vertauschungen
  a) kopiere record $t = a[i]$
  b) iteriere $t = a[2], a[2] = a[11], a[11] = a[1], a[1] = t$

Indirektes Sortieren - Beispiel

```
Vor dem Sortieren:
  i    1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
  a[i] A  S  O  R  T  I  N  G  E  X  A  M  P  L  E
  p[i] 1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
Nach dem indirekten Sortieren:
  i    1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
  a[i] A  S  O  R  T  I  N  G  E  X  A  M  P  L  E
  p[i] 1  11 9  15 8  6  14 12 7  3  13 4  2  5  10
Durch Permutation mittels p[i] ergibt sich das sortierte Array:
  i    1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
  a[i] A  A  E  E  G  I  L  M  N  O  P  R  S  T  X
  p[i] 1  ...
```

2.3.6 BucketSort

Voraussetzung

Schlüssel sind darstellbar als ganzzahlige Worte im Bereich $0, \ldots, M-1$, sodaß sie als Array-Index verwendet werden können. $a[i] \in \{0, \ldots, M-1\} \forall i = 1, \ldots, n$

Arbeitsweise

1. Erstelle ein Histogramm, d.h. zähle für jeden Schlüsselwert, wie häufig er vorkommt.
2. Berechne aus dem Histogramm die Position für jeden Record.
3. Bewege die Datensätze an ihre erechnete Position. Läuft man dabei rückwärts, ist BucketSort stabil.

BucketSort - Beispiel

```
A B B A C A D A B B A D D A

Histogramm:
  6 4 1 3
        x   * Histogramm
        x   x neue, sortierte Reihenfolge
        x
```

```
      x
    x
    x
    x
    x
6 *
5 *
4 * *
3 * *   *
2 * *   *
1 * * * *    ==> A A A A A A B B B B C D D D
  A B C D
```

BucketSort - Komplexitätsanalyse

Für die Zeitkomplexität gilt:

$$T(n) = O(n + M) = O(\max\{n, M\})$$

n: Berechnung des Histogramms, M: Berechnung der neuen Positionen, falls $M < n$: $O(n)$

2.4 Höhere Sortierverfahren

2.4.1 MergeSort

MergeSort - Komplexitätsanalyse: Komplexitätsklasse

Erster Ansatz Bestimmung der Laufzeit mit Hilfe des Master-Theorems, dazu: Rekursionsformel
Sei $T(n)$ die Anzahl der Operationen (im wesentlichen Vergleiche). Dann gilt:

$$T(n) = \left\{ \begin{array}{ll} c_1 & n = 1 \\ 2 * T(\frac{n}{2}) + c_2 n & n > 1 \end{array} \right\}$$

für entsprechende Konstaten $c_1, c_2 > 0$. Dabei charakterisiert
c_1 den Aufwand zur Lösung des trivialen Problems (n=1)
nc_2 den Aufwand für das Verschmelzen zweier Listen
Mit Hilfe des Master-Theorems erhalten wir:

$$a = 2, b = 2 \Rightarrow \frac{2}{2} = 1 \Rightarrow T(n) = O(n \log n)$$

Genauer Sei $T(n)$ die Anzahl der Vergleiche. Die Folge wird in zwei Teifolgen aufgeteilt: Eine Teilfolge mit $\lceil \frac{n}{2} \rceil$ Elementen und eine Teilfolge mit $\lfloor \frac{n}{2} \rfloor$ Elementen. Zur

Verschmelzung werden $\lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor - 1 = n - 1$ Vergleiche benötigt.
Daher ergibt sich folgende Rekursionsgleichung:

$$T(n) = \left\{ \begin{array}{ll} 0 & n = 1 \\ n - 1 + T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) & n > 1 \end{array} \right\}$$

Behauptung:

$$T(n) = n\lceil \log_2(n) \rceil - 2^{\lceil \log_2 n \rceil} + 1$$

Beweis: Die Rekursion wird verifiziert mittels vollständiger Induktion:
Induktionsanfang: n=1

$$T(1) = 0, \quad T(1) = 1 * 0 - 2^0 + 1 = 0$$

Induktionsannahme: Behauptung gelte für alle natürlichen Zahlen m mit $1 \leq m < n$
Induktionsschritt:

$$\lceil \log_2 \lfloor \frac{n}{2} \rfloor \rceil = \left\{ \begin{array}{ll} \lceil \log_2 \frac{n}{2} \rceil = \lceil \log_2 n \rceil - 1 & n \text{gerade} \\ \lceil \log_2 \frac{n-1}{2} \rceil = \left\{ \begin{array}{ll} \lceil \log_2 n \rceil - 1 & \text{n ungerade, } n \neq 2^k + 1 \\ \lceil \log_2 n \rceil - 2 & n = 2^k + 1 \end{array} \right\} & \text{für geeignetes k} \end{array} \right\}$$

$$\lceil \log_2 \lceil \frac{n}{2} \rceil \rceil = \left\{ \begin{array}{ll} \lceil \log_2 \frac{n}{2} \rceil = \lceil \log_2 n \rceil - 1 & \text{n gerade} \\ \lceil \log_2 \frac{n+1}{2} \rceil = \lceil \log_2 n \rceil - 1 & \text{n ungerade}, n \neq 2^k \end{array} \right\}$$

Fall 1: $n \neq 2^k + 1$

$$\begin{array}{rcl} T(n) & = & n - 1 + T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) \\ & = & n - 1 + \lceil \frac{n}{2} \rceil * \lceil \log_2 \lceil \frac{n}{2} \rceil \rceil - 2^{\lceil \log_2 \lceil \frac{n}{2} \rceil \rceil + 1} + \lfloor \frac{n}{2} \rfloor * \lceil \log_2 \lfloor \frac{n}{2} \rfloor \rceil - 2^{\lceil \log_2 \lfloor \frac{n}{2} \rfloor \rceil} + 1 \\ & = & n * \lceil \log_2(n) \rceil + 2^{\lceil \log_2 n \rceil} + 1 \end{array}$$

Fall 2: $n = 2^k + 1$

$$\begin{array}{rcl} T(n) & = & n + \lceil \frac{n}{2} \rceil * (\lceil \log_2 n \rceil - 1) - 2^{\lceil \log_2 n \rceil - 1} + \lfloor \frac{n}{2} \rfloor * (\lceil \log_2 n \rceil - 2) - 2^{\lceil \log_2 n \rceil - 2} + 1 \\ & = & n * \lceil \log_2 n \rceil 2^{\lceil \log_2 n \rceil} + 1 \end{array}$$

2.4.2 QuickSort

QuickSort - Prinzip

Gegeben sei eine Folge F von Schlüsselelementen.

Divide: Zerlege F bzgl. eines partitionierenden Elementes (Pivot-Element) $p \in F$ in zwei Teilfolgen F_1 und F_2 mit:

$$\left\{ \begin{array}{ll} x_1 \leq p & \forall x_1 \in F_1 \\ p \leq x_1 & \forall x_2 \in F \end{array} \right\}$$

Conquer: Sortiere F_1 und F_2 durch rekursiven Aufruf zu F_1^S und F_2^S, rückwärts trivial, falls F_1, F_2 eindeutig (oder leer)

Merge: Trivial: Setze Teilfolgen zusammen $F_1^S + p + F_2^S$

QuickSort - Erklärung der Idee

Ziel Zerlegung (Partitionierung) des Teilarrays (Divide) $a[l \ldots r]$ bzgl. eines Pivotelementes $a[k]$ in zwei Teilarrays $a[l \ldots k-1]$ und $a[k+1 \ldots r]$ (ggf. durch Vertauschen von Elementen), so daß gilt:

$$\begin{array}{ll} \forall i \in \{l, \ldots, k-1\} & a[i] \leq a[k] \\ \forall i \in \{k+1, \ldots, r\} & a[k] \leq a[i] \end{array}$$

Methode Laufe mit zwei Indizes gegenläufig von l bzw. r zur Mitte und tausche ggf. Schlüssel zwischen den Teilarrays aus. Die Indizes treffen sich an einer Position k, an der dann auch das Pivot-Element zu liegen kommt.

Rekursion Bearbeite linkes und rechtes Teilarray rekursiv.

QuickSort - Beispiel

```
5 2 4 7 8 1 3 6   Pivot: 6
5 2 4 3 8 1 7 6
5 2 4 3 1 8 7 6   Pivot holen
5 2 4 3 1 6 7 8   Pivot: 1; Pivot holen
1 2 4 3 5 6 7 8   Pivot: 5; Pivot holen
1 2 4 3 5 6 7 8   Pivot: 3
1 2 3 4 5 6 7 8   Pivot: 8
```

QuickSort - Implementierung

```
static void quickSort (int [] a, int l, int r) {
  int m;
  if (l < r) {
    m = partition (a, l, r);  /* divide  */
    quickSort (a, l, m-1);    /* conquer */
    quickSort (a, m+1, r);
  }                           /* merge   */
}

Aufruf mit: quickSort (a, 1, n);

static int partition (int [] a, int l, int r) {
  int i = l-1;
  int j = r;
  while (i < j) {
    ++i; while (a[i] < a[r]) { ++i; }
    --j; while (a[j] >= a[r]) { --j; }
    swap (a, i, j);
```

```
  }
  swap (a, i, j); /* technischer Grund */
  swap (a, i, r); /* Pivot holen */
  return i;
}
```

- Achtung: Verwende Sentinel (Wärter) ($i\infty$ am linken Rand), um korrekte Terminierung der inneren Schleife über j zu garantieren.
- Aufruf mit:

```
a[0] := -unendlich
quickSort (a, 1, n);
```

- ```
  while (a[i] < a[r]) i++;  /* bricht spätestens für i=r ab   */
  while (a[j] > a[r]) j--;  /* bricht spätestens für j=l-1 ab */
  ```

QuickSort - Komplexitätsanalyse

Best Case: $T(n) = 2 * T(\frac{n}{2}) + O(n) \Rightarrow O(n \log n)$
: Die Folge zerfällt immer in zwei gleich große Teilfolgen.

Worst Case: $T(n) = T(n-1) + O(n) \Rightarrow O(n^2)$
: Die Folge ist bereits auf- oder absteigend sortiert (da wir Pivot immer am Ende wählen).

Average Case: $T(n) = n \log n$
: Bei der Analyse fließen folgende Annahmen ein:

  - Gleichverteilung für den Index des Pivot-Elemetes → Verwendung des arithmetischen Mittels für die Rekursionsaufrufe
  - lineare Zeit für die restlichen Operationen → führt zu folgendem Ansatz: $T(n) \approx O(n) + \frac{1}{n}\left(\sum_{i=0}^{n-1} T(i) + T(n-1-i)\right) = O(n) + \frac{2}{n}\sum_{i=0}^{n-1} T(i)$

2.4.3 HeapSort

Heap

Ein Heap ist ein links-vollständiger Binärbaum, in dem der Schlüssel jedes inneren Knotens größer ist als die Schlüssel seiner Kinder.
  ```

Heap - Arrayeinbettung Bettet man einen Heap (mit Breitendurchlauf) in ein Array $a[1 \dots n]$ ein, so gilt:

$$a[\lfloor \frac{1}{2} \rfloor] \geq a[i] \quad \forall i = 2, \dots, n$$

Insbesondere gilt damit für einen Heap im Array $a[1 \dots n]$:

- Ein Array $a[1 \dots n]$ erfüllt die Heap-Eigenschaft, falls a die obige Bedingung erfüllt.
- $a[1] = \max\{a[i] : i = 1, \dots, n\}$

Folgerungen Teilweise (partielle) Heap-Eigenschaft: Ein Array $a[1 \dots n]$ ist ein partieller Heap ab Position l, falls gilt:

$$a[\lfloor \frac{i}{2} \rfloor] \geq a[i] \quad \forall i = 2l, \dots, n$$

Außerdem gilt folgendes Korollar:
Jeder Array $a[1 \dots n]$ ist ein Heap ab Position $l = \lfloor \frac{n}{2} \rfloor + 1$.

HeapSort - Prinzip

Phase 1: Aufbau eines Heaps
Wandele das Array $a[1 \dots n]$ in einen (vollständigen) Heap um.

Phase 2: Sortierung des Heap
for i=1 to n-1 do

1. Tausche a[1] (=Maximum der Einträge im Heap) mit a[n-i+1] (letzte Position im Heap)
2. Stelle für das Restarray $a[1 \dots n - i]$ die Heap-Eigenschaft wieder her.

Beispiel: Heapaufbau

```
Ausgangssituation:
  1  2  3  4  5  6  7  8
  10 14 7  17 3  21 11 18

Vertausche 17 und 18:
  1  2  3  4  5  6  7  8
  10 14 7  18 3  21 11 17

Vertausche 7 und 21:
  1  2  3  4  5  6  7  8
  10 14 21 18 3  7  11 17
```

```
Vertausche 14 und 18, 17 und 14:
  1  2  3  4  5  6  7  8
  10 18 21 17 3  7  11 14

Vertausche 10 und 21, 11 und 10:  "versickern" (engl. "sink")
  1  2  3  4  5  6  7  8
  21 18 10 17 3  7  10 14
```

Beispiel - Sortierphase

```
Ausgangssituation nun (| = Sortiergrenze):
  21 18 11 17 3  7  10 14 |

Vertausche 21 und 14, versickere 14:
  18 17 11 14 3  7  10 | 21

Vertausche 18 und 10, versickere 10:
  17 14 11 10 3  7  | 18 21

Vertausche 17 und 7, versickere 7:
  14 10 11 7  3  | 17 18 21

Vertausche 14 und 3, versickere 3:
  11 10 3  7  | 14 17 18 21

Vertausche 11 und 7, versickere 7:
  10 7  3  | 11 14 17 18 21

Vertausche 10 und 3, versickere 3:
  7  3  | 10 11 14 17 18 21

Vertausche 7 und 3:
  3  | 7  10 11 14 17 18 21
```

HeapSort - Implementierung

-
  ```
  static void convertToHeap (int [] a, int n) {
    /* ab (n div 2) liegt Heap-Eigenschaft schon vor */
    for (int i=n/2; i>=1; --i) {
      sink (a, n, i);
    }
  }
  ```

- Wenn die Schleife bis i gelaufen ist, hat das Array a ab Index i die Heap-Eigenschaft

```
static void sink (int [] a, int n, int i) {
  /* versickere das Element an Position i im Heap a bis max. Position n */
  while (i <= n/2) {
    int j = 2 * i;
    if (j < n && a[j+1] > a[j]) ++j;
    if (a[j] > a[i]) {
      swap (a, j, i);
      i = j;
    } else {
      /* a[i] >= a[j] */
      i = n;
    }
  }
}

static void heapSort (int [] a, int n) {
  /* Phase 1: Heap Aufbau  */
  convertToHeap (a, n);

  /* Phase 2: Sortierphase */
  for (int i=n; i>=1; --i) {
    swap (a, 1, i);
    sink (a, i-1, 1);
  }
}
```

HeapSort - Komplexitätsanalyse

Wir betrachten die Anzahl der Vergleiche, um ein Array der Größe $n = 2^k - 1, \quad k \in N$ zu sortieren.
Zur Veranschaulichung betrachten wir die Analyse exemplarisch für ein Array der Größe $n^5 - 1 = 31$ (d.h. $k = 5$).

i=0	1 Knoten
i=1	2 Knoten
i=2	4 Knoten
i=3	8 Knoten
i=4	16 Knoten

Heap-Aufbau für Array mit $n = 2^k - 1$ Knoten:

- Auf der Ebene i $(i = 0, \ldots, k-1)$ gibt es 2^i Knoten.
- Neues Element auf Ebene $i = k-2, k-3, \ldots, 0$ versickern lassen.
- Dieses Element kann maximal bis auf Ebene $k - 1$ sinken.

- Pro Ebene werden dabei maximal 2 Vergleiche benötigt.

1. Vergleich	`if a[j] <= a[i]` (Vergleich des abzusenkenden Elementes mit dem größeren der beiden Nachfolger)
2. Vergleich	`if a[j+1] <= a[j]` (Vergleich der beiden Nachfolgeknoten untereinander)

- Gesamt: Obere Schranke für die Anzahl der Vergleiche:
  $\sum\limits_{i=0}^{k-2} 2 * (k-1-i) * 2^i = 2^{k+1} - 2(k+1) = O(n)$

Sortierphase: Nach dem Aufbau des Heaps muß noch die endgültige Ordnung auf dem Array hergestellt werden. Dazu wird ein Knoten der Tiefe i auf die Wurzel gesenkt. Dieser Knoten kann mit ßinkmaximal um i Ebenen sinken. Pro Ebene sind hierzu maximal zwei Vergleiche nötig. Damit gilt die obere Schranke:

$$\sum_{i=0}^{k-1} 2i * 2^i = 2(k-2) * 2^k + 4 = O(n \log n)$$

Zusammen: Sei $n = 2^k - 1$, dann gilt für die Anzahl $T(n)$ der Vergleiche:

$$T(n) \leq s^{k+1} - 2(k+1) + 2(k-2)2^k + 4 = 2k(2^k - 1) - 2(2^k - 1) = 2n \log_2(n+1) - 2n = O(n \log n)$$

Für $n \neq 2^k - 1$ erhält man ein ähnliches Ergebnis. Die Rechnung gestaltet sich jedoch umständlicher.

Resultat: HeapSort sortiert jede Folge $a[1 \dots n]$ mit höchstens $2n \log_2(n+1) - 2n \in O(n \log n)$ Vergleichen.

2.5 Untere und obere Schranken für das Sortierproblem

- bisher: Komplexität eines Algorithmus
- jetzt: Komplexität eines Problems (einer Aufgabenstellung)
- Ziel: $T_A(n)$:= Anzahl der Schlüsselvergleiche, um eine n-elementige Folge von Elementen mit Algorithmus zu sortieren. $T_{min}(n)$:= Anzahl der Vergleiche für den effizientesten Algorithmus.

Komplexität des Sortierproblems

- Suche nach einer unteren Schranke:
  Gibt es ein $T_0(n)$, so daß gilt:
  $T_0(n) \leq T_A(n) \forall$ Algorithmen A
  Das bedeutet: Jeder denkbare Algorithmus benötigt mindestens $T_0(n)$ viele Vergleiche.

- Sucher nach einer oberen Schranke:
  Wähle einen Sortieralgorithmus A mit Komplexität $T_A(n)$. Das bedeutet: $T_A(n)$ Vergleiche reichen aus, um jedes Sortierproblem der Größe n zu lösen.

Untere und obere Schranke zusammen

$$T_0(n) \leq T_{min}(n) \leq T_A(n)$$

- Wunsch: $T_0(n)$ und $T_A(n)$ sollen möglichst eng zusammen liegen.
- Konkret: Wir betrachten für das Sortierproblem nur Vergleichsoperationen, d.h. Beschränkung auf Algorithmen, die ihr Wissen über die Anordnung der Eingabefolge nur durch Vergleiche erhalten.

Obere Schranke Wähle möglichst effizienten Algorithmus, z.B. MergeSort mit $O(n \log n)$ Vergleichen:

$$T_A(n) = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1 \leq n\lceil \log_2 n \rceil - n + 1$$

Warum nicht BucketSort mit $O(n + M)$ Vergleichen ? BucketSort erfordert zusätzliche Bedingungen an die Schlüsselmenge, es ist somit kein allgemeines Sortierverfahren (z.B. nicht für Strings geeignet). BucketSort durchbricht auch die von uns ermittelte untere Schranke (s.u.).

Untere Schranke Gegeben sei eine n-elementige Folge.

Sortieren: Auswahl einer Permutation dieser Folge (in aufsteigender Reihenfolge)

Es gibt $n!$ Permutationen, aus denen die "richtige" auszuwählen ist. Betrachte dazu den Entscheidungsbaum = Darstellung für den Ablauf eines nur auf Vergleichen basierenden Sortieralgorithmus:

- innere Knoten: Vergleiche
- Blätter: Permutationen

Entscheidungsbaum - Beispiel

Der folgende binäre Entscheidungsbaum sortiert das Array $a[1 \ldots 3]$. Da $3! = 6$, muß der Entscheidungsbaum 6 Blätter besitzen. Wegen $\log_2 6 \approx 2.58$ existiert im Baum mindestens ein Pfad der Länge 3:

```
              a[1] < a[2]
             /           \
            ja            nein
           /                \
    a[2] < a[3]              a[1] < a[3]
```

```
     /         \              /        \
    ja        nein         ja        nein
   123        a[1]<a[3]   213        a[2]<a[3]
             /         \            /          \
           ja          nein       ja           nein
          132          312       231           321
```

Ein binärer Entscheidungsbaum für das Sortierproblem besitzt genau $n!$ viele Blätter. Damit ergibt sich als untere Schranke für das Sortierproblem:

$$\log_2 n! \leq \lceil \log_2 n \rceil \leq T_0(n)$$

Mit der Ungleichung $n \log_2 - n \log_2 e \leq \log_2 n!$ (Beweis s.u.) erhalten wir das Gesamtergebnis:

$$n \log_2 n - n \log_2 e \leq T_{min}(n) \leq n \lceil \log_2 n \rceil - n + 1$$

Obere Schranke aus MergeSort, untere Schranke aus folgendem Beweis:

Einfache Schranken für $n!$

- Obere Schranke:

$$n! = \prod_{i=1}^{n} i \leq \prod_{i=1}^{n} n = n^n$$

- Untere Schranke:

$$n! = \prod_{i=1}^{n} i \geq \prod_{i=\lceil \frac{n}{2} \rceil}^{n} i \geq \prod_{i=\lceil \frac{n}{2} \rceil}^{n} \lceil \frac{n}{2} \rceil \geq \left(\frac{n}{2}\right)^{\frac{n}{2}} = \sqrt{\left(\frac{n}{2}\right)^{n}}$$

- Zusammen:

$$\left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n! \leq n^n$$

Integralmethode Engere Schranken für $n!$ können mittels der Integralmethode berechnet werden: Approximiere das Flächenintegral monotoner und konkaver Funktionen von oben und unten durch Trapezsummen.
Ansatz für die Fakultätsfunktion:

$$\log_2 n! = \log_2 \prod_{i=1}^{n} = \sum_{i=1}^{n} \log_2 i$$

Untere Schranke Die Logarithmusfunktion $x \to f(x) := \log_2(x)$ ist monoton und konvex (oder doch konkav ?). Daher:

$$\int\limits_{i-\frac{1}{2}}^{i+\frac{1}{2}} \log_2(x)dx \leq \log_2(i) * 1$$

[1]

Obere Schranke Summation von $i = 1, \ldots, n$ ergibt:

$$\int\limits_{\frac{1}{2}}^{n+\frac{1}{2}} \log_2(x)dx = \sum_{i=1}^{n} \int\limits_{i-\frac{1}{2}}^{i+\frac{1}{2}} \log_2(x)dx \leq \sum_{i=1}^{n} \log_2(i) = \log_2 n!$$

Mit $\int\limits_a^b \log_2(x)dx = [x \log_2 x - x \log_2 e]_a^b$ folgt für die untere Schranke für $\log_2(n!)$:

$$\log_2(n!) \geq (n+\frac{1}{2}) * \log_2(n+\frac{1}{2}) - (n+\frac{1}{2}) \log_2 e - \frac{1}{2}\log_2(\frac{1}{2}) + \frac{1}{2}\log_2 e \geq n * \log_2 n - n * \log_2 e$$

Dies schließt den Beweis für die untere Schranke $n \log_2 n - n \log_2 e$ des Sortierproblems ab.

2.6 Sortieren

$O(n^2)$	InsertionSort, SelectionSort, BubbleSort
$O(n \log n)$	MergeSort, QuickSort ($O(n^2)$), HeapSort
$O(n + M)$	BucketSort (nicht nur Vergleiche!)
$O(n \log n)$	untere Schranke für Sortieren mit Vergleichen

[1] Der redet von Trapezsummen, rechnet aber mit Rechtecksummen.

3 Suchen in Mengen

3.1 Problemstellung

- Gegeben:
  - Eine Menge von Elementen (Records)
  - Eindeutig indentifiziert durch einen Schlüssel
  - In der Regel werden Duplikate ausgeschlossen
  - Bsp: Eine Menge von Personen (Ausweisnummer, Name, Adresse); Ausweisnummer ist eindeutig
- Aufgabe:
  - Finde zu einem gegebenen Schlüsselwert das entsprechende Element
- Notation:

Universum $U :=$	Menge aller möglichen Schlüssel (z.B. N)
Menge S	$S \subseteq U$

3.2 Einführung

- Beispiele für Universen:
  - Symboltabelle Compiler, z.B. nur 6 Zeichen (Buchstaben und Zahlen): $|U| = (26 + 10)^6$
  - Konten einer Bank: z.B. 6-stellige Kontonummer: $|U| = 10^6$
- Typische Aufgaben:
  Finde zu einem gegeben Schlüssel das entsprechende Element in der Menge und führe eine Operation aus (z.B. Ausgabe). Im Prinzip werden alle Mengen-Operationen betrachtet (Datenbanken): z.B. Suchen, Einfügen, Entfernen

3.3 Operationen

- Wörterbuch Operationen
  - Insert (x, S) - Fügt x der Menge S hinzu: $S := S \cup \{x\}$

– Search (x, S) - Sucht x in der Menge S: $\{y|y \in S \wedge y = x\}$
– Delete (x, S) - Entfernt x aus der Menge S: $S := S \setminus \{x\}$

- Weitere Operationen
  – Order (S) - Produziere eine sortierte Liste / Ausgabe
  – Union (A, S) - Vereinigung $S := S \cup A$
  – Intersection (A, S) - Durchschnitt $S := S \cap A$
  – Difference (A, S) - Differenz $S := S \setminus A$

3.3.1 Operation Suche

- Spezifikation
  Suchalgorithmus nutzt Vergleichsoperation, um gesuchtes Element zu finden. Menge $S \subseteq U$ (U: Universum) mit n Elementen $|S| = n$, Element $S[i]$ mit $1 \leq i \leq n$ (bzw. $0 \leq i \neq n - 1$), Suche: a bzw. i mit $S[i] = a$

- Verschiedene Varianten
  – Sequentielle (oder lineare) Suche
  – Binäre Suche
  – Interpolationssuche

  Für die binäre Suche und die Interpolationssuche ist eine Ordnung auf den Suchschlüsseln nötig.

Erfolglose Suche

- Spezifikation
  Was gibt die Funktion zurück, wenn die Suche erfolglos war ?
  1. `int seek (int a, int [] S) -> -1` wenn erfolglos
  2. `int seek (int a, int [] S)` throws exception

  Welche Alternative sinnvoll ist, hängt vom Kontext ab.

- Beobachtungen zur Effizienz
  I.a. dauert die Suche nach einem Element, welches nicht in S enthalten ist, länger als die Suche nach enthaltenen Elementen. Das liegt daran, daß bei erfolgreicher Suche oft frühzeitig abgebrochen werden kann, wenn das Element gefunden wurde.

Lineare Suche

- Lineare Suche durchläuft S sequentiell
- S kann ungeordnet sein, daher ist keine Sortierung / Ordnung der Elemente nötig

```
public static int seek (int a, int S[],) throws Exception {
  for (int i=0; i< S.length; ++i) {
    if (S[i] == a) return i;
  } /* Suche war erfolglos */
  throw new Exception ("seek failed" + a);
}
```

Binäre Suche

- Voraussetzung: S ist sortiert
- Binäre Suche halbiert den Suchbereich sukzessive

```
public static int seek (int a, int [] S) throws Exception {
  int low = 0; high = S.length-1;
  while (low <= high) {
    int mid = (high + low) / 2;
    if (a == S[mid]) {
      return mid;
    } else {
      if (a < S[mid]) {
        high = mid - 1;
      } else {
        low = mid + 1;
      }
    }
  }
  throw new Exception ("seek failed: " + a);
}
```

Interpolationssuche

- S ist sortiert
- Annahme: Gleichverteilung der Elemente

- Idee: Schätze gesuchte Position durch lineare Interpolation

```
public static int seek (int a, int [] S) throws Exception {
  int low = 0; int high = S.length-1;
  while (low < high) {
    int i = low + (a - S[low]) * (high - low) / (S[high] - S[low]);
    if (a == S[i]) {
      return i;
    } else {
      if (a < S[i]) {
        high = i-1;
      } else {
        low = i+1;
      }
    }
  }
  if (a == S[low]) return low;
  throw new Exception ("failed");
}
```

Komplexität: Suche in Mengen

- Komplexität: Anzahl der Schlüsselvergleiche
- Lineare Suche
  Anzahl der Vergleiche bei linearer Suche in n Elementen:

Im Mittel:	$\frac{n}{2}$ Vergleiche, falls erfolgreich und falls Suchschlüssel ähnlich verteilt sind wie die Schlüssel in der Menge S
erfolglose Suche:	n Vergleiche (falls Array unsortiert), $\frac{n}{2}$ Vergleiche bei Ausnutzung einer Sortierung (falls Sortierung vorhanden)

  Komplexität: $O(n)$
- Binäre Suche: Sukzessives Halbieren
  Voraussetzung: Liste ist sortiert. Bei jedem Schleifendurchlauf halbiert sich der zu durchsuchende Bereich.
  Rekursionsgleichung: $T(n) = T(\frac{n}{2}) + 1, \quad T(1) = 1$
  Komplexität: $O(\log n)$, auch im Worst Case
- Interpolationssuche
  Voraussetzung: Liste ist sortiert.
  Komplexität im Durchschnitt: $O(\log \log n)$ (ohne Beweis)
  Komplexität im Worst Case: $O(n)$

- Vergleich: z.B. n=1000

Suchmethode	n = 1000	n = 1.000.000
Sequentielle Suche	500	500.000
Binäre Suche	10	20
Interpolationssuche	2	4

  Überlegung: Unsortiertes Array sortieren, um dann mit der binären Suche schneller suchen zu können: $O(n \log n) + O(n)$, also lohnt sich das vorherige Sortieren für eine einmalige Suche nicht.

3.3.2 Bitvektor-Darstellung von Mengen

- Geeignet für kleine Universen U
  $N = |U|$ vorgegebene maximale Anzahl von Elemente
  $S \leq U = \{0, 1, \ldots, N-1\}$
  Suche hier nur als Test "ist enthalten"

- Darstellung als Bitvektor
  Verwende Schlüssel i als Index im Bitvektor (= Array von Bits)
  `boolean isElement (Bit [], i) { return Bit[i] == 1; }`
  Bitvektor:
  `Bit[i] = 0` falls $i \notin S$
  `Bit[i] = 1` falls $i \in S$

Bitvektor-Darstellung Komplexität

- Operationen
  - Insert, Delete – $O(1)$: setze / lösche entsprechendes Bit
  - Search – $O(1)$: teste (=lese) entsprechendes Bit
  - Initialize – $O(N)$: setze alle Bits des Arrays auf 0
- Speicherbedarf
  - Anzahl Bits: $O(N)$ maximale Anzahl Elemente
- Problem bei Bitvektor
  - Initialisierung benötigt $O(N)$ Operationen
  - Verbesserung: "spezielle Array-Implementierung"
    Ziel: Initialisierung in $O(1)$

Spezielle Array-Implementierung

- Verbesserte Bitvektor-Darstellung für kleine Universen $U = \{0 \dots 1\}$
  Benötigte Vektoren:
  Bit $0 \dots N-1$
  Prt $0 \dots N-1$
  Stck $0 \dots N-1$
  Idee: $i \in S \Leftrightarrow Bit[i] = 1 \rightarrow$ "könnte enthalten sein" (keine Initialisierung) und $0 \leq Ptr[i] \leq top \rightarrow Ptr[i]$ zeigt auf Element im Stack und $Stck[Prt[i]] = i \rightarrow$ Stack bestätigt, daß $Bit[i]$ initialisiert ist.
  Init: $top = -1 \rightarrow$ kein Element auf dem Stack

- Initialisierung: $top = -1$;

- Anmerkung: $top + 1$ ist die Anzahl der jemals eingefügten Elemente

- Suchen:

```
seek (int i) {
  if (0 <= Ptr[i] && Ptr[i] <= top && Stck[Ptr[i]] == i) {
    return Bit[i]
  } else {
    throw new Exception ("seek failed" + i);
  }
}
```

- Einfügen:

```
insert (int i) {
  Bit[i] = true;
  if (! (0 <= Prt[i] && Ptr[i] <= top && Stck[Ptr[i]] == i)) {
    top = top + 1;
    Ptr[i] = top;
    Stck[top] = i;
    /* Stck und Ptr beinhalten alle bisher betrachteten Elemente */
  }
}
```

- Löschen:

```
delete (int i) {
  Bit[i] = false;
}
```

3.3.3 Zusammenfassung

n: Anzahl der Elemente in der Suchmenge, $n = |S|$, N: Kardinalität des Universums, $N = |U|$

Methode	Search	Platz	Vorteil	Nachteil
Lineare Suche	n	n	keine Initialisierung	hohe Suchkosten
Binäre Suche	$\lceil \log_2 n \rceil$, $O(\log n)$	n	worst case auch in $\lceil \log_2 n \rceil$	sortiertes Array erforderlich
Interpolationssuche	$O(\log \log n$	n	schnelle Suche	sortiertes Array erforderlich, worst case: $O(n)$
Bit-Vektor	1	N	schnellstmögliche Suche	nur für kleine Universen möglich, Initialisierung: $O(n)$
Spezielle Array-Implementierung	$O(1)$	N	schnellstmögliche Suche, Initialisierung: $O(1)$	nur für kleine Universen möglich

3.4 Hashing

Ziel: Zeitkomplexität Suche: $O(1)$ (wie bei Bitvektoren, aber auch für große Universen)

Ausgangspunkt: Bei der Bitvektordarstellung wird der Schlüsselwert direkt als Index in einem Array verwendet.

Grundidee: Oft hat man ein sehr großes Universum (z.B. mit Strings), aber nur eine kleine Objektmenge (z.B. Straßennamen einer Stadt), für die ein kleines Array ausreichen würde.

Idee: Bilde verschiedene Schlüssel auf dieselben Indexwerte ab; dabei sind Kollisionen möglich.

Grundbegriffe: Übersicht

U	Universum aller Schlüssel
$S \subseteq U$	Menge der zu speichernden Schlüssel $n = \|S\|$
T	Hash-Tabelle der Größe m (Array)

S wird in T gespeichert, daher muß $n < m$ gelten.

Hash-Funktion h: Def.: $h : U \to \{0, \ldots, m-1\}$ "Schlüsseltransformation", "Streuspeicherung", $h(x)$ ist der Hash-Wert von x.

Anwendung: Hashing wird angewendet, wenn

- $|U|$ sehr groß ist

- $|S| << |U|$: Anzahl der zu speichernden Elemente ist viel kleiner als die Größe des Universums

3.4.1 Anwendung von Hashing

- Beispiel: Symboltabelle Compiler

  - Universum U: Alle möglichen Bezeichner (Zeichenketten)
  - hier: Einschränkung auf Länge 20 (nur Buchstaben und Ziffern) $\rightarrow |U| = (26 + 10)^2 0 \approx 1,3 * 10^3 1$ (dreizehn Quitilliarden)
  - Somit: keine umkehrbare Speicherfunktion realistisch
  - es werden nur m viele Symbole ($m << 10^3 1$) in einem Programm verwendet

- Beispiel: Studenten

  - Universum U: Alle möglichen Matrikelnummern (6-stellig, d.h. $|U| = 10^6$)
  - nur n Studenten besuchen eine Vorlesung (z.B. n=500); Schlüsselmenge $S = \{k_1, \ldots, k_n\} \leq U$
  - Verwendung einer Hash-Tabelle T mit z.B. $m = |T| = 800$

3.4.2 Hashing-Prinzip

- Grafische Darstellung – Beispiel: Studenten
- Gesucht: Hash-Funktion, welche die Matrikelnummern möglichst gleichmäßig auf die 800 Einträge der Hash-Tabelle abbildet.

3.4.3 Hash-Funktion

- Abbildung auf Hash-Tabelle
  Hash-Tabelle T hat m Plätze (Slots, Buckets)
  - in der Regel $m << |U|$, daher Kollisionen möglich
    Speichern von $|S >= N$ Elementen ($n < m$)
  - Belegungsfaktor $\alpha = \frac{n}{m}$
- Anforderung an Hash-Funktion
  h soll die gesamte Tabelle abdecken: $h : U \rightarrow \{0 \ldots m - 1\}$ surjektiv
  h(x) soll effizient berechenbar sein
  h soll die Schlüssel möglichst gleichmäßig über die Hash-Tabelle verteilen, um Kollisionen vorzubeugen
  - Speicherung: $T[h(x)] = x$
    Kollision: $h(x) = h(y)$ für $x \neq y$
  - dann kann x (bzw. y) evtl. nicht in $T[h(x)]$ gespeichert werden

Divisions-Methode

- Sei 'm' die Größe der Hash-Tabelle, dann ist eine Hashfunktion nach der Divisions-Methode $h(x) = xmodm$
- Bewährt hat sich folgende Wahl für m:
  - m ist Primzahl
  - m teilt nicht $2^i \pm j$, wobei i, j kleine natürliche Zahlen sind.
    $rightarrow$ diese Wahl gewährleistet eine surjektive, gleichmäßige Verteilung
- Vorteil: leicht berechenbar
- Nachteil: Clustering: aufeinanderfolgende Schlüssel werden auf aufeinanderfolgende Plätze abgebildet

Mittel-Quadrat-Methode

- Ziel: Verhindern von Clustering
  - nahe beieinander liegende Schlüssel streuen
    h(x) = mittlerer Block von Ziffern aus x^2
- Nachteil: rechenaufwendig
- Beispiel: Hash-Tabelle ist $m = 101$ Einträge groß

x	x mod 101	x^2	$h(x)$
127	26	16.$\underline{12}$9	12
128	27	16.$\underline{38}$4	38
129	28	16.$\underline{64}$1	64

Beispiel: Hash-Funktion

- sehr kleines Beispiel: Symboltabelle Compiler
  - Zeichenkette $c = c_1 \dots c_k$ mit $c_i = \{A, B, \dots, Z\}$
  - Codierung als Zahl: $N('A') = 1, \quad N('B') = 2 \dots$
  - mögliche Hash-Funktion: (Beispiel für die Divisionsmethode):
    $h(c) = (\sum\limits_{i=1}^{k} N(c_i)) \text{ mod } m$
  - Hash-Tabelle der Größe 17
  - Verwende nur die ersten drei Zeichen der Bezeichner
    $h(c) = (N(c_1) + N(c_2) + N(c_3)) \text{ mod } 17$

- Beispiel: Einsortieren der Monatsnamen in Symboltabelle

0	November
1	April, Dezember (Orkan)
2	Maerz
3	
4	
5	
6	Mai, September (Blitz und Donnerschlag)
7	
8	Januar
9	Juli
10	
11	Juni
12	August, Oktober (krass Gewitter)
13	Februar
14	
15	
16	

  3 Kollisionen
  November: $(14 + 15 + 22) \bmod 17 = 51 mod 17 = 0$

Perfekte Hash-Funktion

- Eine Hash-Funktion ist perfekt: wenn für $h : U \to \{0, \ldots, m-1\}$ mit $S = \{k_1, \ldots, k_n\} \subseteq U$ gilt: $h(k_i) = h(k_j) \Leftrightarrow i = j$ (Injektivität), d.h. für die Menge S treten keine Kollisionen auf.

- Eine Hash-Funktion ist minimal, wenn $m = n$ ist, also nur genau so viele Plätze wie zu speichernde Elemente benötigt werden.

- I.a. können perfekte Hash-Funktionen nur ermittelt werden, wenn alle einzufügenden Elemente und deren Anzahl (also S) im Voraus bekannt sind. → "statisches Problem"

3.4.4 Kollisions-Behandlung

- Zwei allgemeine Strategien um Kollisionen zu behandeln:
  - Geschlossenes Hashing
    * Speicherung innerhalb der Hash-Tabelle
    * verschiedene Verfahren, um einen alternativen Speicherplatz zu finden ("sondieren")
    * heißt auch offene Adressierung, da sich die Adresse nicht nur aus dem Hash-Wert ergibt.
  - Offenes Hashing

* Speicherung der kollidierenden Elemente außerhalb der Hash-Tabelle, z.B. in verketteter Liste
* heißt auch geschlossene Adressierung, da der Wert der Hash-Funktion als Adresser erhalten bleibt.

Offenes Hashing

- Speicherung außerhalb der Tabelle
  In jedem Behälter (Bucket) der Hash-Tabelle wird eine Liste von Elementen gespeichert.

- Beispiel: Einsortieren der Monatsnamen

$$\begin{array}{lllllll}
0 & \cdot & \rightarrow & & \cdot & & \\
1 & \cdot & \rightarrow & \text{April} & \cdot & \rightarrow & \text{Dezember} \quad \cdot \\
2 & \cdot & \rightarrow & \text{Maerz} & \cdot & & \\
3 & \cdot & & & & & \\
4 & \cdot & & & & & \\
5 & \cdot & & & & & \\
6 & \cdot & \rightarrow & \text{Mai} & \cdot & \rightarrow & \text{September} \quad \cdot \\
7 & \cdot & & & & & \\
8 & \cdot & \rightarrow & \text{Januar} & \cdot & &
\end{array}$$

- Kostenabschätzung für Operationen Insert / Search / Delete
  Alle Operationen bestehen aus zwei Schritten:

  1. Hash-Funktion anwenden, um Adresse des Behälters zu berechnen.
  2. Durchlaufen der Liste, um entsprechenden Eintrag zu finden.

- Zeitkomplexität in Abhängigkeit vom Belegungsfaktor:
  Belegungsfaktor: $\alpha = \frac{n}{m}$

  1. Adresse berechnen: $O(1)$
  2. zusätzlich: Liste durchlaufen:
     average case $O(1 + \frac{\alpha}{2})$
     worst case $O(n)$ $O(\alpha)$

- Platzkomplexität:
  $O(n + m)$, m Buckets, n Listenelemente

- Verhalten:
  $\alpha << 1$ $(n << m)$: Wie BitVektor
  $\alpha \approx 1$ $n \approx m$: Übergangsbereich
  $\alpha >> 1$ $(n >> m)$: Wie verkettete Liste (sequentielle Suche), viele (lange) Listen

Geschlossenes Hashing

- Neuberechnung des Speicherplatzes (Behälter / Bucket)
  bei Kollision wird ein alternativer Speicherplatz berechnet (geht nur für $n < m$).
- Sondieren / Rehashing
  - Die Sondierungsfunktion listet Schritt für Schritt alle möglichen Speicherplätze auf.
  - Sondierungsschritt als Parameter der Hash-Funktion: $h(x, j)$ ist Speicherplatz für $x \in U$ im Sondierungsschritt j.
  - Bei der Suche muß ebenfalls sondiert werden und ggf. müssen mehrere Behälter durchsucht werden.

Lineares Sondieren

- (Re)Hashing durch Lineares Sondieren:
  $h(x, j) = (h(x) + j) \bmod m$
- Beispiel: Symboltabelle Monatsnamen

0	November
1	April
2	Maerz
3	Dezember
4	
5	
6	Mai
7	September
8	Januar
9	Juli
10	
11	Juni
12	August
13	Februar
14	Oktober
15	
16	

- Nachteile des Linearen Sondierens
  Cluster-Bildung: Einträge aus verschiedenen Sondierungsfunktionen können in denselben Cluster fallen. Die Vermischung von Clustern kann beim offenen Hashing nicht auftreten.

Quadratisches Sondieren

- Ziel: BEsseres Streuverhalten als bei linearem Sondieren
  $h(x, j) = (h(x) + j^2) \bmod m$
- Qualität
  Wie viele der noch freien Behälter werden durch $h(x, j)$ aufgezählt ? (Ziel: alle)
  hier: Wenn m eine Primzahl ist, dann ist $j^2 \bmod m$ für $j = 0 \ldots \lfloor \frac{m}{2} \rfloor$ immer verschieden, d.h. es kann freie Zellen geben, die beim Sondieren nicht gefunden werden.
- Verbesserung gegenüber linearem Sondieren:
  - Primäre Kollisionen treten wie beim linearen Sondieren auf.
  - jedoch keine linearen Ketten (keine primären Cluster)
  - verstreute Cluster können auch hier auftreten (sekundäre Cluster) → Schlüssel mit gleichen Hash-Werten werden auf dieselbe Art und Weise auf alternative Speicherplätze abgebildet.

Doppel-Hashing

- Ziel: Cluster verhindern
  - Sondieren im Abhängigkeit vom Schlüssel
  - Verwendung von zwei unabhängigen Hash-Funktionen $h(x)$ und $h'(x)$
- Die zwei Hash-Funktionen sind unabhängig, wenn gilt:
  $P(h(x) = h(y)) = \frac{1}{m}$ Kollisionswahrscheinlichkeit für h
  $P(h'(x) = h'(y)))\frac{1}{m}$ Kollisionswahrscheinlichkeit für h'
  $P(h(x) = h(y) \wedge h'(x) = h'(y)) = \frac{1}{m^2}$ (Unabhängigkeit von h und h'
- Definition Hash-Funktion mit sondieren:
  $h(x, i) = (h(x) + h'(x) * i^2) \bmod m$
- Eigenschaften
  - mit Abstand die beste der vorgestellten Kollisionsstrategien
  - fast ideales Verhalten aufgrund der Unabhängigkeit

3.4.5 Operationen beim Hashing

- Einfügen
  - Offenes Hashing: Behälter suchen, Element in Liste einfügen
  - Geschlossenes Hashing: Freien Behälter suchen, ggf. durch Sondieren
- Löschen
  - Offenes Hashing: Behälter suchen, Element aus der Liste entfernen

- Geschlossenes Hashing:
  * Behälter suchen, ggf. sondieren
  * Element entfernen und
  * Zelle als gelöscht markieren
    nötig, da evtl. bereits hinter dem gelöschten Element andere Elemente durch Sondieren eingfügt wurden; dann: Beim Suchen über freien Behälter hinweg sondieren
  * Gelöschte Elemente (=freigewordene Behälter) dürfen wieder überschrieben werden

3.4.6 Aufwandsabschätzung

- Kosten für Sondieren (Kollisionen)
  - Wieviele Kollisionen treten auf ?
  - Wie oft muß man sondieren ?
  - Annahme: Hashfunktion sei ideal → Schlüsselwerte werden gleichverteilt, alle Behälter sind gleich wahrscheinlich
- Werte für die Aufwandsabschätzung

m	Größe der Hashtabelle
n	Anzahl der Einträge
i	Anzahl der Kollisionen
$\alpha = \frac{n}{m}$	Belegungsfaktor der Hashtabelle
$q(i, n, m)$	Wahrscheinlichkeit, daß mind. i Sondierungsschritte nötig sind

Komplexität des geschlossenen Hashings

Kollisionswahrscheinlichkeit:

$$i = 1: \quad q(1, n, m) = \frac{n}{m}$$

Nach einer Kollision kommt für das Sondieren ein belegter Platz weniger in Frage:

$$i = 2: \quad q(2, n, m) = \frac{n}{m}\frac{n-1}{m-1}$$

Somit gilt für i: Wir haben bereits $(i-1)$ Kollisionen, die Sondierungsfunktion test noch $m-(i-1)$ Behälter, von denen $n-(i-1)$ belegt sind:

$$q(i, n, m) = q(i-1, n, m) \cdot \frac{n-(i-1)}{m-(i-1)} = \frac{n}{m}\frac{n-1}{m-1} \cdots \frac{n-(i-1)}{m-(i-1)} = \prod_{j=0}^{i-1} \frac{n-j}{m-j}$$

Komplexität: Einfügen

Kosten Einfügen: Anzahl der Sondierungsschritte

- Summe der Wahrscheinlichkeiten, daß $1., 2., \ldots$ Platz belegt ist
- Für jede Kollision ein Sondierungsschritt:

$$C_{insert}(n,m) = \sum_{i=0}^{n} q(i,n,m) = \frac{m+1}{m+1-n} \approx \frac{1}{1-\alpha}, \quad (\alpha = \frac{n}{m} \text{ Belegungsfaktor})$$

  Beweis mittels vollständiger Induktion über n und m

Komplexität: Suche

- Erfolglose Suche
  - Suche bis sicher ist, daß das Element nicht in der Hashtabelle enthalten ist, d.h. bis eine freie Stelle gefunden wurde (beachte gelöschte Einträge!)

    $$C^-_{search}(n,m) \approx C_{insert}(n,m) \approx \frac{1}{1-\alpha}$$

- Erfolgreiche Suche
  - Beobachtung: Bei jedem Sondierungsschritt könnte das gesuchte Element gefunden werden.
  - Testen in Sondierungsschritt i, ob $T[h(x,i)] = x$ (dieser Test war auch bei Einfügen entscheidend)

    $$C^+_{search} = \frac{1}{n} \sum_{j=0}^{n-1} C_{insert}(j,m) = \ldots \approx \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

Komplexität - Übersicht

Belegung α	$C^-_{search}(n,m) = C_{Insert}(n,m) \approx \frac{1}{1-\alpha}$	$C^+_{search}(n,m) = C_{Delete}(n,m) \approx \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$
0,5	≈ 2	$\approx 1,38$
0,7	$\approx 3,3$	$\approx 1,72$
0,9	≈ 10	$\approx 2,55$
0,95	≈ 20	$\approx 3,15$

[1]

[1] An dieser Stelle widerspricht der Vordruck dem Vorlesungsinhalt: Laut Vorlesung ist $C^+_{search}(n,m) = C_{insert}(n,m)$, der Vordruck, aus dem wiederum der Tabellenkopf übernommen wurde, widerspricht dem jedoch. Meiner Meinung nach macht jedoch der Tabellenkopf mehr Sinn, da die Delete-Operation wesentlich weniger zeitaufwendig ist, was der Tabelle entspricht.

3.4.7 Zusammenfassung: Hashing

- Anwendung
  - statische Dictionaries (Telefonbuch, PLZ-Verzeichnis)
  - IP-Adresse zu MAC-Adresse (in Hauptspeicher)
  - bei Datenhash → Hash-Join, Sekundärspeicher
- Vorteil
  - in Average-Case sehr effizient (oft $O(1)$, aber sehr abhängig von der Parametrisierung)
- Nachteil
  - Skalierung: Größe der Hash-Tabelle muß vorher bekannt sein. Abhilfe: Symbolhashing, lineares Hashing
  - keine Bereichsabfragen, keine Ähnlichkeitsanfragen → Lösung: Suchbäume

3.4.8 Kollisionen beim Hashing

- Verteilungsverhalten von Hash-Funktionen
  - Untersuchung mit Hilfe von Wahrscheinlichkeitsrechnung
  - S: Ereignisraum
  - E: Ereignis $E \in S$
  - P: Wahrscheinlichkeitsdichtefunktion
- Beispiel: Gleichverteilung
  - Einfacher Münzwurd: $S = \{Kopf, Zahl\}$
  - Wahrscheinlichkeit für Kopf: $P(Kopf) = \frac{1}{2}$
  - n-fache Münzwürfe: $S = \{Kopf, Zahl\}^n$
  - Wahrscheinlichkeit für n-mal Kopf: $P(''n - malKopf'') = \frac{1}{2^n}$
- Analogie zum Geburtstagsproblem(-paradoxon)
  "Wie groß ist die Wahrscheinlichkeit, daß mindestens 2 Personen am gleichen Tag Geburtstag haben ?" $(m = 3 * 5, \quad n = Personen)$
- Eintragen des Geburtstages in die Hash-Tabelle

$p(i, m) =$	Wahrscheinlichkeit, daß für den i-ten Eintrag eine Kollision auftritt
$p(1, m) =$	0, da noch kein Behälter belegt ist
$p(2, m) =$	$\frac{1}{3*5} = \frac{1}{m}$, da ein Behälter belegt ist
$p(i, m) =$	$\frac{i-1}{m}$, da $i - 1$ Behälter belegt sind

Die Wahrscheinlichkeit für eine einzige Kollision bei m Einträgen in eine Hash-Tabelle mit m Behältern ist gleich dem Produkt der Einzelwahrscheinlichkeiten:

$$P(nocoll|n,m) = \prod_{i=1}^{n}(1 - p(i,m)) = \prod_{i=0}^{n-1}(1 - \frac{i}{m})$$

Die Wahrscheinlichkeit, daß es mindestens zu einer Kollision kommt, ist dann:

$$P(coll|n,m) = 1 - P(nocoll|n,m)$$

- Kollisionen bei Geburtstagstabelle (m = 365 Behälter)
  Schon bei einer Belegung von $\frac{23}{365} = 6\%$ kommt es mit einer Wahrscheinlichkeit von 50% zu einer Kollision. Daher sind Strategien zur Kollisionsbehandlung wichtig. Fragen:
  - Wann ist eine Hashfunktion gut ?
  - Wie groß muß eine Hashtabelle in Abhängigkeit von der Anzahl der Elemente sein ?

Anzahl der Personen n	$P(coll\|n,m)$
10	$0,11695$
20	$0,41144$
22	$0,47570$
23	$0,50730$
24	$0,53838$
30	$0,70632$
40	$0,89123$
50	$0,97037$

- Frage: Wie muß m in Abhängigkeit von n wachsen, damit $P(coll|n,m)$ konstant bleibt ?

$$P(nocoll|n,m) = \prod_{i=0}^{n-1}(1 - \frac{i}{m})$$

- Durch Anwendung der Logarithmus-Rechenregel kann ein Produkt in eine Summe umgewandelt werden:

$$P(nocoll|n,m) = \exp[\sum_{i=0}^{n-1}\ln(1 - \frac{i}{m})]$$

Logarithmus: $\ln(1-\varepsilon) \approx -\varepsilon$ (für kleine $\varepsilon > 0$), da $n << m$ gilt: $\ln(1-\varepsilon) \approx -\frac{i}{m}$

- Auflösen der Gleichung

$$P(nocoll|n,m) \approx \exp(-\sum_{i=0}^{n-1} \frac{i}{m}) = \exp(-\frac{n(n-1)}{2m}) \approx \exp(-\frac{n^2}{2m})$$

- Ergebnis: Die Kollisionswahrscheinlichkeit bleibt konstant, wenn m (Größe der Hash-Tabelle) quadratisch mit n (Anzahl der Einträge) wächst.

3.5 Suchbäume

Bisher betrachtete Algorithmen für Suche in Mengen

- Sortierte Arrays
  - Nur sinnvoll für statische Mengen, da Einfügen und Entfernen $O(n)$ Zeit benötigt
  - Zeitbedarf für Suche: $O(\log n)$ (binäre Suche)
  - Bereichsanfragen: "alle Einträge zwischen 127 und 135"
- Hashing
  - stark abhängig von gewählter Hash-Funktion
  - Kollisionsbehandlung nötig
  - Anzahl der Objekte muß vorher bekannt sein (im Groben)
  - keine Bereichs- oder Ähnlichkeitsanfragen. "Einträge für Maier o.ä."

3.5.1 Suchbäume

- Suchbäume
  - beliebig dynamisch erweiterbar
  - Operationen Einfügen, Entfernen, Suchen sind in $O(\log n)$ realisierbar
  - effiziente Lösungen für die Speicherung im Sekundärspeicher
- Wir betrachten im weiteren folgende Arten von Bäumen
  - Binäre Suchbäume
  - Balancierte Bäume (binäre / nicht-binäre)
    - AVL-Bäume, B-Bäume, R-Bäume
  - optimale binäre Suchbäume (für statische Daten)

3.5.2 Binäre Suchbäume

Ausgangspunkt: Binäre Suche

- Start in der Mitte → Wurzel
- Aufteilung in (jeweils ohne Mitte)
  - linken Teil
  - rechten Teil
- Rekursiv weiter
  - linker Teilbaum aus linker Hälfte
  - rechter Teilbaum aus rechter Hälfte

Definition: Binärer Suchbaum

- Definition
  Ein binärer Suchbaum für eine Menge von n Schlüsseln $S = \{x_1, x_2, \ldots, x_n\}$ besteht aus einer Menge von beschrifteten Knoten $v = \{v_1, v_2, \ldots, v_n\}$ mit der Beschriftungsfunktion value: $v \to S$. value ist ordnungserhaltend, d.h. für einen Knoten v_i im linken Teilbaum und v_j im rechten Teilbaum des Knotens v_k gilt: $value(v_i) \leq value(v_k) \leq value(v_j)$
- Begriffe (Zeichnung)
- Datenstruktur

```
class Node {
  private int value;       /* mit getValue / setValue */
  private Node leftChild;  /* set / get LeftChild     */
  private Node rightChild; /* set / get RightChild    */
}

class Tree {
  private Node root;       /* mit get / set Root      */
  public void insert (int value);
  public void delete (int value);
  public Node search (int value);
}
```

Binärer Suchbaum mit Bereichsblättern

- Bereichsblätter-Darstellung
  - leere Teilbäume werden hier als Bereichsblätter dargestellt, die Intervalle zwischen den gespeicherten Schlüsseln beinhalten.
  - Ein Baum mit n Knoten hat $(n + 1)$ Bereichsblätter
  - erfolglose Suche endet immer in einem Bereichsblatt
  - Bereichsblätter werden in der Regel nicht gespeichert
- Beispiel (Grafik)

Suche in Binärem Suchbaum

- Programm - Suche in Binärem Baum

```
/* Iterative Version */
public Node search (int value) {
  Node v = root;
  while (v != null && v.getValue() != value) {
    if (value < v.getValue()) {
      v = v.getLeftChild ();
    } else {
      v = v.getRightChild ();
    }
  }
  /* v == null || v.getValue == value */
  return v;
}
```

- Die Methode search endet
  - in einem inneren Knoten, wenn value gefunden wurde
  - in einem leeren Teilbaum (Bereichsblatt), wenn value nicht gefunden wurde

Operationen beim Binären Suchbaum

- Operation t.insert (value) für Tree T
  - Suche value in T, Ergbnis sei Bereichsblatt (x_i, x_{i+1})
  - Ersetze Bereichsblatt durch: (Grafik)
- Operation t.delete (value) für Tree T
  - Suche zu löschenden Knoten v, unterscheide 3 Fälle:

1. v hat nur leere Teilbäume → v kann unmittelbar gelöscht werden
2. v hat genau einen nicht-leeren Teilbaum v_s → ersetze v durch v_s
3. v hat zwei nicht-leere Teilbäume:
   * Suche im linken Teilbaum von v den rechtesten (größten) Unterknoten w
   * ersetze v durch w
   * lösche w aus der ursprünglichen Stelle

Beispiel zum Einfügen Operationen:

1. Suche Einfügestelle → endet in einem Bereichsblatt
2. Bereichsblatt wird ersetzt durch neuen Knoten 12

Beispiel zum Löschen Operationen: (s. Grafik)

Beispiel: Entferne 23

- suche im linken Teilbaum den am weitesten rechts liegenden Knoten w: 21
- erste 23 durch 21
- entferne 21

Suchbäume für lexikographische Schlüssel

- Beispiel: Deutsche Monatsnamen (Einfügungen in kalendarischer Reihenfolge)
- Binärer Suchbaum: Inorder-Durchlauf: April, August, Dezember, Februar, Januar, Juli, Juni, Maerz, Mai, November, Oktober, September (alphabetische Sortierung)
- Beochbachtung: Der Baum ist nicht balanciert.

Komplexitätsanalyse: Binärer Suchbaum

- Analyse der Laufzeit: Operationen Insert und Delete bestehen aus
  - Suche der Position im Baum
  - lokale Änderung im Baum: $O(1)$
- Analyse der Suchzeit:
  - die Anzahl der Vergleiche entspricht der Höhe des Baumes, da immer genau ein Pfad betroffen ist.
  - Sei $h(t)$ die Höhe des Suchbaumes t, dann ist die Komplexität für search: $O(h(t))$

- Abhängigkeit von der Anzahl der Knoten ?
    - ”Wie hoch ist ein binärer Suchbaum mit n Knoten ?”
    - ”Wieviele Knoten enthält ein binärer Suchbaum der Höhe h maximal oder auch minimal ?”
    - Best-Case: maximale Anzahl Knoten für gegebene Höhe h (Grafik) Betrachte Fall: alle Knoten haben zwei Nachfolger (bis auf die Blätter): $n = 2^h - 1$
    - Worst-Case:
        - alle Knoten bis auf ein Blatt haben nur einen Nachfolger
        - Baum degeneriert zu einer linearen Liste
        - Baum der Höhe h hat nur noch $n = k$ Knoten
    - Anzahl der Vergleiche in Abhängigkeit von n (Anzahl Knoten): Die Anzahl der Vergleiche entspricht der Höhe h.
        - Best Case: voll gefüllter Baum hat $n \leq 2^{k-1}$ Knoten, $h \geq \log_2(n+1)$ bzw. $h = \lceil \log_2(n+1) \rceil$, Komplexität: $O(\log n)$, vgl. Binäre Suche
        - Worst Case: lineare Liste als Baum hat $n = k$ Knoten, Komplexität: $O(n)$, vgl. lineare Suche

- Problemanalyse:
    - Der binäre Suchbaum hat im optimalen Fall eine gute Komplexität für die Operationen Insert, Delete, Search: $O(\log n)$
    - Durch Insert, Delete kann ein Binärbaum zu einer linearen List entarten.

- Ziel:
  Die Operationen Insert und Delete sollen so verändert werden, daß ein Baum immer balanciert [2] bleibt.

Balancierte Bäume

- Ziel:
    - Verhindern der Worst-Case-Komplexität $O(n)$
    - entartete Bäume verhindern durch Ausbalancieren

- Balancieren (zwei Arten)
    - Gleichgewichtsbalancierung $BB(a)$
      ”Bounded Balance” mit Grenze a: Die Anzahl der Blätter in den Unterbäumen wird ausbalanciert. Dabei beschreibt a den maximalen Unterschied zwischen den Teilbäumen.
    - Höhenbalancierung:
      Die Höhe der beiden Teilbäume wird ausbalanciert. (z.B. Höhe ± 1)

[2] Anzahl der Knoten im linken und rechten Teilbaum ungefähr gleich / Höhe der beiden Teilbäume etwa gleich groß

3.5.3 AVL-Baum

- Balancierter Baum:
  Historisch erste Variante eines balancierten Baumes (von Adelson, Velski und Landis)

- Definition:
  Ein AVL-Baum ist ein binärer Suchbaum mit folgender Strukturbedingung:
  Für alle Knoten gilt: Die Höhen der beiden Teilbäume unterscheiden sich höchstens um eins.

- Operationen:
  Damit diese AVL-Bedingung auch nach Insert oder Delete noch gilt, muß der Baum ggf. rebalanciert werden.

- Beispiele:

- Untersuchung der Komplexität:
  - Operation Search hängt weiterhin von der Höhe ab
  - Frage: Wie hoch kann ein AVL-Baum für eine gegebene Knotenanzahl n maximal werden, bzw. aus wie vielen Knoten muß ein AVL-Baum der Höhe h mindestens bestehen ?

Höhe eines AVL-Baumes

- Anzahl der Knoten in Abhängigkeit von der Höhe
  Gesucht: minimale Knotenzahl, d.h. wir betrachten minmal gefüllte Bäume. Dabei sei $N(h)$ die minimale Anzahl Knoten für AVL-Baum der Höhe h.
  $h = 1$ $N(h) = 1$ (nur Wurzel)
  $h = 2$ $N(h) = 2$ (nur ein Zweig gefüllt)
  $h = 3$ $N(h) = 4$

- Anzahl Knoten in Abhängigkeit von der Höhe h
  Für beliebigen minimal gefüllten AVL-Baum der Höhe $h \geq 3$ gilt:
  1. Die Wurzel besitzt zwei Teilbäume
  2. Ein Teilbaum hat die Höhe $h - 1$.
  3. Der andere Teilbaum hat die Höhe $h - 2$

  $$N(h) = N(h-1) + N(h-2) + 1$$

  Ähnlich zu den Fibonacci-Zahlen.

- Ein minimal gefüllter AVL-Baum heißt auch manchmal Fibonacci-Baum. Baum:

$$N(h) = \left\{ \begin{array}{ll} 1 & h = 1 \\ 2 & h = 2 \\ N(h-1) + N(h-2) + 1 & h > 2 \end{array} \right\}$$

- AVL-Baum (Fibonacci-Baum)
  Vergleich Fibonacci-Reihe $f(h)$ mit AVL-Baum-Höhe $N(h)$

h	=	1	2	3	4	5	6	7	8	9	10
f(h)	=	1	1	2	3	5	8	13	21	34	55
N(h)	=	1	2	4	7	12	20	33	54	88	...

$$N(h) = f(h+2) - 1$$

(Beweis mittels vollständiger Induktion)

- Wie hoch ist ein Baum, der aus "n" Knoten besteht ?
  Verwendung der Fibonacci-Formel:

$$f(h) = \frac{\Phi_0(h) - \Phi_1(h)}{\sqrt{5}}$$

mit $\Phi_0 = \frac{1+\sqrt{5}}{2}$, $\Phi_1 = \frac{1-\sqrt{5}}{2}$.

- Knotenanzahl in Abhängigkeit von der Höhe

$$N(h) = F(h+2) - 1 = \frac{1}{\sqrt{5}}(\Phi_0^{k+2} - \Phi_1^{k+2}) - 1 \geq \Phi_0^h$$

- Also gilt für einen AVL-Baum mit "n" Knoten für die Höhe "h":
  $N(h) \leq n$ und mit $N(h) = F(h+2) - 1$ ergibt sich:

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

$$n \geq \Phi_0^h - 1$$

$$n + 1 \leq \Phi_0^h$$

$$\log_{\Phi_0}(n+1) \geq h$$

$$h \leq \log_{\Phi_0}(n+1)$$

- Höhe in Abhängigkeit von der Knoten-Anzahl

$$h \leq \log_{\Phi_0}(n+1)$$

$$h \leq \frac{\ln 2}{\ln \Phi_0} \log_2(n+1)$$

$$h < 1.4404 \log_2(n+1)$$

  Zum Vergleich: Für den maximal gefüllten binären Suchbaum gilt: $h \leq \log_2(n+1)$

- Ergebnis:
  - Ein AVL-Baum ist maximal 44% Höher als ein maximal ausgeglichener binärer Suchbaum.
  - Komplexität für Search: $O(\log n)$ auch im worst-case.
- Operationen:
  Frage: Wie müssen die Operationen Insert, Delete verändert werden, damit die Balance eines AVL-Baumes gewährleistet bleibt.

Operationen auf AVL-Bäumen

- Suchen: genauso wie bei binärem Suchbaum
- Einfügen / Löschen
  1. Wie bei binärem Suchbaum (Suche Einfügestelle bzw. zu entfernenden Knoten und führe Einfügen bzw. Entfernen aus)
  2. Feststellen, ob die AVL-Eigenschaft an der Einfüge- bzw. Entfernungsstelle noch gilt.
  3. Rebalancierung durch "Rotation".
- Untersuchung
  1. Wie kann festgestellt werden, ob ein Baum nicht mehr ausbalanciert ist ?
  2. Wie kann ein Baum rebalanciert werden ?

Balance bei AVL-Bäumen

- Vorgehensweise
  Bei jedem Knoten wird die Höhendifferenz (Balance b) der beiden Teilbäume abgespeichert: b = Höhe (rechter Teilbaum) - Höhe (linker Teilbaum)

- Beispiel
  Es wird gespeichert, welcher Teilbaum größer ist:
  -1 linker Teilbaum größer als rechter
  0 Teilbäume gleich groß
  1 rechter Teilbaum größer als linker

- Einfügen

  - zuerst einfügen wie bei Binärbaum (s.o.)
  - Beim Einfügen kann sich nur die Balance b von Knoten verändern, die auf dem Suchpfad liegen, dabei kann AVL-Kriterium verletzt werden.
  - Gib nach dem Einfügen eines neuen Knotens den Suchpfad wieder zurück und aktualisiere die Balance.

Einfügen bei AVL-Bäumen

- Ablauf

  - Nach Einfügen: [3] Kritischen Knoten bestimmen (nächstgelegene Vorgänger zum neuen Knoten mit $b = \pm 2$): Dieser Knoten ist Ausgangspunkt der Reorganisation.
  - Der Pfad vom kritischen zum neuen Knoten legt Rotationstyp fest.
    * Einfachrotation
      · LL-Rotation: Beispiel: Einfügung war im linken Teilbaum des linken Teilbaums; Baum ist nach LL-Rotation wieder balanciert.
      · RR-Rotation: Beispiel: Rotiere rechten Teilbaum
    * Doppelrotation
      · LR-Rotation: Eine einfache Rotation reicht hier nicht aus, da der "problematische" Teilbaum "innen" liegt. → Betrachte Teilbaum B2 näher. Bemerkung: Es spielt keine Rolle, ob der neue Knoten im Teilbaum B2a oder in B2b eingefügt wurde.

      · RL-Rotation: analog (symmetrisch zur LR-Rotation)

- Komplexität

  - Einfügen
    * Die Rotationen stellen das AVL-Kriterium im unbalancierten Unterbaum wieder her und sie bewahren die Sortierreihenfolge (d.h. Suchbaumeigenschaft).

[3] Einfügen wie in einem normalen Binärbaum

    * Wenn ein Baum rebalanciert wird, ist der entsprechende Unterbaum danach immer genauso hoch wie vor dem Einfügen. $\Rightarrow$ Der restliche Baum bleibt konstant und muß nicht überprüft werden. $\Rightarrow$ Beim Einfügen eines Knotens benötigt man höchstens eine Rotation zur Rebalancierung.
  - Aufwand:
    * Einfügen: Suche + Einfügen + Rotation: $O(h) + O(1) + O(1) = O(h) = O(\log n)$

Löschen bei AVL-Bäumen

- Vorgehensweise
  - Zuerst normales Löschen wie beim Binärbaum.
  - Nur für Knoten auf dem Lösch-Pfad kann das AVL-Kriterium verletzt werden. (vgl. Einfügen)
- Ablauf
  - Nach dem eigentlichen Löschen muß der kritische Knoten bestimmt werden: Nächster Vorgänger zum tatsächlich entfernten Knoten mit $b = \pm 2$.
  - Dieser kritische Knoten ist Ausgangspunkt der Reorganisation (=Rotation)
  - Rotationstyp wird bestimmt, als ob im gegenüberliegenden Unterbaum ein Knoten eingefügt worden wäre.
- Kritisches Beispiel Beobachtung: Der Teilbaum ist nach der Rotation hier nicht ausbalanciert. Dies liegt daran, daß der Teilbaum rotiert wird, der selbst ausbalanciert war.

- Komplexität
  Beim Löschen eines Knotens
  - wird das AVL-Kriterium (lokal) wiederhergestellt, die Sortierreihenfolge bleibt erhalten.
  - kann es vorkommen, daß der rebalancierte Unterbaum nicht dieselbe Höhe wie vor dem Löschen besitzt. $\rightarrow$ auf dem weiteren Pfad zur Wurzel kann es zu weiteren Rotationen kommen. $rightarrow$ Beim Löschen werden maximal h Rotationen benötigt.
- Aufwand
  Entfernen (Suchen + Löschen) + Rebalancieren $= O(h)+O(h) = O(h) = O(\log n)$

3.5.4 Zusammenfassung: Binäre Bäume

- Vergleich mit Hashing
  - Hashing hat im optimalen Fall eine konstante Komplexität $O(1)$, im schlechtesten Fall $O(n)$
  - Damit Hashing gut funktioniert, müssen viele Parameter bekannt sein:
    * Wertebereich, Anzahl der Daten
    * Größe der Tabelle, d.h. Speicherplatz
    * gute Hashfunktion
  - Bäume haben in allen Operationen (Suchen, Einfügen, Entfernen) logarithmische Komplexität, $O(\log n)$
  - Bei Bäumen ist kein Wissen über die Daten nötig

3.6 Verwendung von Sekundärspeicher

- Motivation
  - Falls Daten persistent (=dauerhaft) gespeicher werden müssen.
  - Falls Datenmenge zu groß für den Hauptspeicher ist.
- Sekundärspeicher / Festplatte Festplatte besteht aus übereinanderliegenden rotierenden Platten mit magnetischen / optischen Oberflächen, die in Spuren und Sektoren eingeteilt sind sowie zugehörigen Schreib- / Leseköpfen an beweglichen Armen.
- Zugriffszeit der Festplatten
  1. Suchzeit: Armpositionierung (Translation)
  2. Latenzzeit: Rotation bis Blockanfang
  3. Transferzeit: Übertragung der Daten
- Blockgrößen
  - Größere Transfereinheiten sind günstiger
  - Gebräuchlich sind Seiten der Größe 4 kilobyte oder 8 kilobyte
- Problem
  - Seitenzugriffe sind teurer als Vergleichsoperationen (in Suchbäumen)
  - möglichst viele ähnliche Schlüssel auf einer Seite (=ein oder mehrere Block(s)) speichern. $\rightarrow$ Lokalitätsprinzip

3.6.1 Mehrwegbäume

- Definition: Knoten haben maximal $m \geq 2$ Nachfolger.
  Knoten $K = (b, P_1, k_1, P_2, k_2, P_3, \ldots, P_{b-1}, P_b)$ eines m-Wege-Suchbaumes B besteht aus:
  - Grad $b =$ Grad $k \leq m$
  - Schlüssel $k_i \quad (1 \leq i \leq b-1)$
  - Zeiger P_i auf Nachfolgeknoten
- Definition der Ordnung (Suchbaumeigenschaft):
  - Die Schlüssel in einem Knoten $K = (k_1, k_2, \ldots, k_b)$ sind geordnet: $k_i \leq k_{i+1}$ für $i = 1, \ldots, b-2$
  - Für einen Knoten $K' = (k'_1, k'_2, \ldots, k'_b)$, der zwischen den Schlüsseln k_{p-1} und k_p liegt, gilt:

    $$k_{p-1} \leq k'_i \leq k_p \qquad \text{für } i = 1, \ldots, b'-1$$

    (Sei $k_0 = -\infty$ und $k_b = +\infty$)

B-Baum

- Definition: Ein B-Baum der Ordnung k ist ein m-Wege-Suchbaum mit $m = 2k+1$.
  Für einen nicht leeren B-Baum gilt:
  1. Jeder Knoten enthält höchstens $2k$ Schlüssel.
  2. Jeder Knoten (außer der Wurzel) enthält mindestens k Schlüssel.
  3. Die Wurzel enthält mindestens einen Schlüssel.
  4. Ein innerer Knoten mit b Schlüsseln hat genau $b+1$ Unterbäume.
  5. Alle Blätter befinden sich auf demselben Level.
- Bedeutung für das "B":
  - Balanciert
  - Bayer (Prof. Rudolf Bayer, Ph.D.)
  - Boeing (Seattle, WA)
  - Barbara
  - Banyan (australischer Baum)

Beispiele für B-Bäume mit $k = 1$ (maximal $m = 3$ Nachfolger)

Operationen bei B-Bäumen Grundoperationen bei B-Bäumen

- Suchen eines Daten-Elementes
  - analog zu binären Suchbäumen: rekursives Absteigen, geleitet durch die Schlüssel
- Einfügen und Löschen
  - Suche entsprechende Position im Baum
  - Einfügen des neuen Elementes in den Knoten → Knoten kann überlaufen → Reorganisieren, z.B Split (Spalten) von Knoten
  - Löschen des Elements aus dem Knoten → Knoten kann unterfüllt sein, d.h. es sind zu wenige Elemente enthalten → Reorganisieren: Zusammenfassen von Knoten

Einfügen in B-Bäume Algorithmus

- Durchlaufe Baum und suche das Blatt B, in welches der neue Schlüssel k gehört
- Füge k sortiert dem Blatt hinzu
  Wenn das Blatt $B = (k_1, \ldots, k_m)$ überläuft: Split (Spaltung)
  1. Erzeuge ein neues Blatt B'
  2. Verteile die Schlüssel auf altes und neues Blatt: $B = (k_1, \ldots, k_{\frac{m}{2}})$, $\quad B' = (k_{\frac{m}{2}+2}, \ldots, k_m)$
  3. Füge den Schlüssel $k_{\frac{m}{2}+1}$ dem Vorgängerknoten hinzu.

  Vorgänger kann auch überlaufen: rekursiv, ggf. bis zur Wurzel

Löschen in B-Bäumen Algorithmus

- Suchen den Knoten K, der den zu löschenden Schlüssel k enthält
- Falls K ein innerer Knoten ist
  - Suchen den größten Schlüssel k' im Teilbaum links von Schlüssel k
  - Ersetze k im Knoten K durch k'
  - Lösche k' aus dem Blatt B
- K ist ein Blatt, dann lösche den Schlüssel aus dem Blatt
  - Es ist möglich, daß k nun weniger als $\frac{m}{2}$ Schlüssel beinhaltet
  - Reorganisation unter Einbeziehung der Nachbarknoten. Bemerkung: Die Wurzel hat keine Nachbarknoten und darf weniger als $\frac{m}{2}$ Schlüssel beinhalten.

Algorithmus Unterlauf bei einem Knoten / Blatt (ohne Wurzel)

- Betrachten der Nachbarn des Knoten K
  - Der Knoten K hat einen Nachbarn N mit mehr als $\frac{m}{2}$ Schlüssel beinhaltet; dann: Ausgleich von K durch Schlüssel aus dem Nachbarknoten N
  - Der Knoten K hat einen Nachbarn N, der genau $\frac{m}{2}$ Elemente hat; dann: Verschmelzung von K und N inklusive dem zugehörigen Schlüssel s im Vorgängerknoten zu einem neuen Knoten. $\rightarrow$ entferne zugehörigen Schlüssel aus dem Vorgänger $\rightarrow$ setze den Verweis im Vorgänger auf den neuen Knoten
  - Sonderfall: K ist die Wurzel
    Die Wurzel wird erst dann gelöscht, wenn sie keine Schlüssel mehr beinhaltet; dann ist der Baum leer.

Ausgleich Aus dem Knoten $K = (k_1 \dots k_a)$ soll der Schlüssel k_i entfernt werden. Ausgleich für B-Baum der Größe $m = 2k + 1$

- Sei $N = (k'_1 \dots k'_n)$ ein Nachbarknoten mit mehr als k Schlüsseln $(n > k)$
- O.B.d.A. sei N rechts von K und p der Trennschlüssel im Vorgänger: Verteile die Schlüssel $k_1, \dots, k_a, p, k'_1, \dots, k'_n$ auf die Knoten K und N gleichmäßig, ersetze den Schlüssel p im Vorgänger durch den mittleren Schlüssel.

Verschmelzen Aus dem Knoten $K = (k_1 \dots k_a)$ soll der Schlüssel k_i entfernt werden. Verschmelzen für B-Baum der Größe $m = 2k + 1$

- Sei $N = (k'_1 \dots k'_k)$ ein Nachbarknoten mit genau k Schlüsseln
- O.B.d.A. sei N rechts von K und p der Trennschlüssel der Vorgänger. $\rightarrow$ Verschmelze Knoten N und K zu einem Knoten $K' = (k_1 \dots k_{a-1}, p, k'_1 \dots k'_a)$
  - löschen den Knoten N sowie den Verweis auf N
  - entferne p aus Vorgänger

Abschätzung der minimalen Höhe Komplexität von Suche ist abhängig von der Höhe (wie bei Binären Bäumen): N_{max} sei maximale Anzahl von Schlüsseln in einem B-Baum der Höhe h

$h = 1 \quad N_{max} = m - 1 \quad (= 2k)$

$h = 2 \quad N_{max} = m(m - 1) + (m - 1)$

Für beliebiges h gilt:

$$\begin{aligned} N_{max} &= m - 1 + m(m-1) + \ldots + m^{h-1}(m-1) \\ &= (m-1)\sum_{i=0}^{h-1} m^i \\ &= (m-1)\frac{m^h - 1}{m-1} \\ &= m^h - 1 \end{aligned}$$

Die minimale Höhe für n Schlüssel ist also: $h \geq \log_m(n + 1)$

Abschätzung der maximalen Höhe Betrachtung der minimalen Anzahl von Schlüsseln N_{min} für einen B-Baum der Höhe h:
$h = 1 \quad N_{min} = 1$
$h = 2 \quad N_{min} = 1 + 2k$
$h = 3 \quad N_{min} = 1 + 2k + 2(k+1)k$
Für beliebiges h gilt:

$$\begin{aligned} N_{min} &= 1 + 2k + 2(k+1)k + \ldots + 2(k+1)^{k-2}k \\ &= 1 + 2k \sum_{i=0}^{h-2} (k+1)^i \\ &= 1 + 2k \frac{(k+1)^{h-1} - 1}{(k+1) - 1} \\ &= 2(k+1)^{h-1} - 1 \end{aligned}$$

Höhenabschätzung bei B-Bäumen Somit ergibt sich für die Höhe h:

$$h \leq \log_{k+1}\left(\frac{n+1}{2}\right) + 1$$

Ergebnis für h ist somit und $m = 2k + 1$:

$$\log_{2k+1} n \leq h \log_{k+1}\left(\frac{n+1}{2}\right) + 1$$

Betrachtung der Schranken:
Die Höhe eines B-Baumes ist logarithmisch zur Basis der minmalen bzw. maximalen Anzahl der Nachfolger eines Knotens beschränkt.
Verallgemeinerung: (a, b)-Bäume: Suchbäume mit dem mindestens a und maximal b Schlüsseln in den Knoten.

Einsatz von B-Bäumen

- Datenbanken
  - Verwendung zur Suche / Sortierung von Datensätzen anhand eines (oder mehrerer) Attribute (s)
  - Dazu: Zusätzlich Speicherung von Datensätzen (oder Verweisen darauf) auf den Seiten des B-Baumes
  - d_1 : Datensatz zu k_i bzw. Verweis auf extern gespeicherten Datensatz
  - Beobachtung:
    - Tritt ein Schlüsselwert mehrfach auf, so wird dieser Wert mehrfach gespeichert
    - Die Datensätze (bzw. Verweise) benötigen zusätzlichen Platz

B*-Bäume

- Definition
  - Ein B*-Baum ist im wesentlichen ein B-Baum mit zwei Knotentypen
    - innere Knoten: haben nur Wegweiserfunktion, d.h. keine (Verweise auf) Datensätze
    - Blätter: enthalten Schlüssel mit (Verweisen auf) Datensätze
    - Blätter sind verkettet, um effizientes Durchlaufen größerer Bereiche zu ermöglichen
- Schematisches Bild: Verkettete Blätter mit Schlüsseln und Verweisen auf Datensätze
- Speicherung: Vergleich B*-Baum und B-Baum
  - In dem inneren Knoten des B*-Baums werden nur Schlüssel gespeichert keine Daten / Verweise
    - es haben mehr Schlüssel Platz
    - höherer Verzweigungsgrad
    - niedrigere Bäume
  - Schlüssel haben nur Wegweiserfunktion und können deswegen oft verkürzt werden; Beispiel:
    - höherer Verzweigungsgrad
    - geringere Baumhöhe
- Bereichsanfrage
  - Einsatzgebiet Datenbanken Neben exakten Suchanfragen treten oft Bereichs- bzw. Intervallanfrage

```
SQL: SELECT * FROM Calendar
     WHERE remindDate BETWEEN '16.06.2003' AND '24.06.2003'
```

  - Beispiel Bereichsanfrage
    Verkettung der Blätter ermöglicht effiziente Bereichsanfragen
- Zusammengesetzte Schlüssel
  - Einsatzgebiet
    Bäume können auf zusammengesetzte Schlüssel erweitert werden.
    - Beispiel
      - (Nachname, Vorname)
      - Punktdaten (x, y)
    - lexikographische Ordnung: Sortiere zuerst gemäß erster Komponente, bei gleichen Werten gemäß zweiter Komponente
  - Anwendung

Räumliche Daten Beispiel Punktdaten in 2D

- 2D-Punktdaten können in einem (B-)Baum über (x, y) indiziert werden, natürlich auch über (y, x)
- Mögliche Anfragen sind:
    1. Punktanfragen
    2. Regionenanfrage (Bereichsanfrage)
    3. Nächster Nachbar
       Anwendung, z.B. GIS (Geographic Information System)
- Beispiele
    - Beispiel für Bereichsanfrage
      Annahme: Index (x, y), Anfrage $10 \leq x \leq 150$ und $2 \leq 2y \leq 50$. Selektion nach x schränkt Suchbereich nicht gut ein: Hier wäre Index (y, x) besser.
    - Beispiel für "Nächster Nachbar" (NN)
      Annahme: Index (x, y). Suche nach NN bzgl. x liefert lange nicht den NN bzgl. $(x, y) \rightarrow$ gibt es bessere Speicherungsstrukturen als B-Bäume (x, y) ?

R-Bäume

- Struktur eines R-Baumes (Guttman, 1984)
    - Unterscheidung von zwei Knotentypen
        * Blätter enthalten Punktdaten (oder auch Rechtecke)
        * innere Knoten enthalten Verweise auf Nachfolgeknoten sowie deren MBRs (als Wegweiser) (MBR: Minimum Bounding Rectangle)
- Aufbau
    - balanciert wie B-Baum
    - Regionen dürfen nicht überlappen
    - ggf. Suche in mehreren Teilbäumen nötig

Zusammenfassung: Mehrwegbäume

- B-Bäume
    - Speicherung auf Festplatten
    - Verkettung (B*-Bäume) für Bereichsanfragen
    - B*-Bäume sind wichtigste Variante in der Praxis
- R-Bäume

- mehrdimensionale Daten (Bsp. GIS)
- nächste-Nachbar-Suche, Bereichsanfragen
- Varianten: z.B. S-Baum: MBS (Minimum Bounding Spheres)

- Hochdimensionale Daten
  - mehrdimensionale Indexstrukturen haben Probleme mit sehr hochdimensionalen Daten. Bsp.: Ähnlichkeitssuche in Multimediadatenbanken

3.6.2 Bitmap-Index

- Einsatzgebiet: Wenn der Wertebereich eines Attributs klein ist, Bsp.
  - männlich / weiblich = binär
  - Berufscode: Hilfskraft, Programmierer, Manager

  Hier eignen sich Bäume schlecht, da immer große Bereiche gelesen werden.
  - Bäume sind gut bei Selektivitäten ≤ 10
  - hier wäre sequentielle Suche oft besser

- Idee
  Jeder Attributwert wird durch (sehr) wenige Bits dargestellt → Bitmap (vgl. Bitvektor)

- Beispiel
  Liste von Angestellen

empNo	M	W
1005	1	0
1464	0	1
2009	0	1
3670	1	0

empNo	H	M	P
1005	1	0	0
1464	0	0	1
2009	0	1	0
3670	0	0	1

- Vergleich mit B-Baum Bitmap-Index benötigt nur wenige Zugriffe, um alle männlichen oder weiblichen Angestellen zu ermitteln.
  - weniger Seitenzugriffe, da weniger Bits pro Datensatz als im B-Baum

3.7 Zusammenfassung

- Hashing
  extrem schneller Zugriff für spezielle Anwendungen → Laufzeit sehr abhängig von guter Parametrisierung

- Binärer Baum (AVL-Baum)
  → allgemeines, effizientes Verfahren für Hauptspeicher

  - benötigt Ordnung
  - Bereichsanfragen möglich
  - effiziente Updates (Einfügen, Löschen, Ändern)

- B-Baum, B*-Baum, R-Baum, etc.
  - effiziente Speicherungsstrukturen für Sekundärspeicher (Festplatten, opt. Platten)

4 Graphen

Viele reale Fragestellungen lassen sich durch Graphen darstellen:

- Beziehungen zwischen Personen
  - Person a kennt Person B
  - A spielt gegen B
- Verbindungen zwischen Punkten (GIS)
  - Straßennetz
  - Eisenbahnnetz
- Telefonnetz
- elektronische Schaltkreise

Bezogen auf einen Graphen ergeben sich spezielle Aufgaben und Fragen:

- Existiert eine Verbindung von A nach B ?
- Existiert eine zweite Verbindung, falls die erste blockiert ist ?
- Wie lautet die kürzeste Verbindung von A nach B ?
- Minimaler Spannbaum
- Optimale Rundreise (Traveling Salesman Problem)

Begriffe:

- Knoten ("Objekte"): Vertex
- Kanten: Edge
- gerichtet / ungerichtet (Richtung): directed / undirected
- gewichtet / ungewichtet: labeled / unlabeled

4.1 Darstellungen

4.1.1 Gerichteter Graph

Ein gerichteter Graph (engl. digraph = "directed graph") ist en Paar $G = (V, E)$:

V	endliche, nichtleere Menge (Knoten, Vertices)
$E \subseteq V \times V$	Menge von Kanten (edges)

Bemerkungen:

$\lvert V\rvert$	Knotenanzahl
$\lvert E\rvert \leq \lvert V\rvert^2$	Kantenzahl

in der Regel werden Knoten numeriert: $i = 0, 1, 2 \ldots, |V| - 1$

Beispiel

$$V = \{0, 1, 2, 3, 4\}$$

$$E = \{(0, 1), (1, 0), (0, 3), (1, 3), (3, 4), (2, 4), (4, 2), (2, 2)\} \subseteq V \times V$$

4.1.2 Definitionen

- Grad eines Knotens := Anzahl der ein- und ausgehenden Kanten
- Ein Pfad ist eine Folge von Knoten $v_0, \ldots, v_{n-1}$ mit $(v_i, v_{i+1}) \in E$, alsoe eine Folge zusammenhängender Kanten
- Länge eines Pfades := Anzahl der Kanten auf dem Pfad (alternativ: Anzahl Knoten)
- Ein Pfad heißt einfach, wenn alle Knoten auf dem Pfad paarweise verschieden sind.
- Ein Zyklus eines Graphen $G = (V, E)$ ist ein Graph $G' = (V', E')$ mit $V' \subseteq V$ und $E' \subseteq E$.

Markierungen

Man kann Markierungen (=Beschriftungen, Gewichtungen, Kosten) für Kanten und Knoten einführen:
Beispiel: Kostenfunktion für Kanten

- Notation: $c[v, w]$ oder $cost(v, w)$
- Bedeutung:
  - Entfernung zwischen v und w
  - Reisezeit (Stunden)
  - Reisekosten (Euro)

4.1.3 Ungerichteter Graph

Ein ungerichteter Graph ist ein gerichteter Graph, in dem die Relation E symmetrisch ist:

$$(V, W) \in E \Leftrightarrow (W, V) \in E$$

Graphische Darstellung:

Bemerkung: Die eingeführten Begriffe (Grad, Pfad, ...) sind analog zu denen für gerichtete Graphen. Änderungen: Z.B. ein Zyklus muß mindestens drei Knoten haben.

4.1.4 Graph-Darstellungen

D.h. die Repräsentation im Speicher, zur Verwendung von Algorithmen:

- je nach Zielsetzung:
  - knotenorientiert (ausgehend von V)
  - kantenorientiert (ausgehend von E)
- knotenorientierte Darstellungen sind gebräuchlicher
  Bsp.: Adjazenzmatrix, Adjazenzlisten

Graph-Interface

```
interface Graph {
  Vertex [] getVertices ();
  void addVertex (Vertex v);
  void deleteVertex (Vertex v);
  void addEdge (Vertex, Vertex);
  void deleteEdge (Vertex, Vertex);
  boolean containsEdge (Vertex, Vertex);
  Liste: successors (Vertex); /* Liste wie gehabt */
}

interface Vertex {
  int getIndex ();
  void setIndex (int);
}
```

Adjazenzmatrix

Die Adjazenzmatrix A ist eine boolesche Matrix mit:

$$A_{ij} = \begin{cases} true & \text{falls } (v_i, v_j) \in E \\ false & \text{sonst} \end{cases}$$

Eine solche Matrix läßt sich als zweidimensionales Array darstellen $A_{ij} = A[i, j]$

Beispiel Für den Beispiel-Graph G_1 ergibt sich folgende Adjazenzmatrix mit der Konvention true=1, false=0:

- Vorteile
  - Entscheidung, ob $(i, j) \in E$ in $O(1)$ Zeit
  - Kantenbeschriftung möglich: Statt boolescher Werte werden Zusatzinformationen (ggf. Verweise) als Matrixeinträge gespeichert.
- Nachteile
  - Platzbedarf stets $O(|V|^2)$, ineffizient, falls $|E| << |V|^2$
  - Initialisierung benötigt $O(|V|^2)$ Zeit.

Implementierung

```
public class AdjMatrix implements Graph {
  private Vertex [] vertices;
  private boolean [][] edges;
  private AdjMatrix (Liste v) {
    int l = r.length ();
    vertices = new Vertex[l];
    for (int i=0; i<l; ++i) {
      v.first ().setIndex (i);
      vertices[i] = v.first ();
      v.delete (v.first ());
    }
    edges = new boolean[l][l];
    for (int i=0; i<l; ++i)
      for (int j=0; j<l; ++j) {
        edges [i][j] = false;
      }
  }
  public Vertex [] getVertices () {
    return vertices;
  }
  public void addVertex (Vertex v) {
    /* Einfügen einer zusätzlichen Spalte und Zeile */
  }
  public void deleteVertex (Vertex v) {
    /* Löschen der entsprechenden Zeile und Spalte  */
  }
  public void addEdge (Vertex u, Vertex v) {
    edges [u.getIndex()][v.getIndex()] = true;
  }
```

```
  public void deleteEdge (Vertex u, Vertex v) {
    edges [u.getIndex()][v.getIndex()] = false;
  }
  public boolean containsEdge (Vertex u, Vertex v) {
    return edges [u.getIndex()][v.getIndex()];
  }
  public Liste successors (Vertex v) {
    int h = v.getIndex ();
    Liste result = new Liste ();
    for (int i=0; i<vertices.length; ++i) {
      if (edges [h][i]) result.insert (vertices[i]);
    }
    return result;
  }
}
```

Adjazenzliste

Für jeden Knoten wird eine Liste der ausgehenden Nachbarknoten verwaltet. [1]

Beispiel Für G_1 ergibt sich folgende Adjazenzliste:

- Vorteile
  - geringer Platzbedarf: $O(|V| + |E|)$
  - Initialisierung in $O(|V| + |E|)$ Zeit
  - Kantenbeschriftung als Zusatzinformation ebi Listenelementen speichern
- Nachteile
  - Entscheidung, ob $(i, j) \in E$ benötigt $O(|E| + |V|)$ Zeit im average case, $\min\{O(|E|), O(|V|)\}$ Zeit im worst case.

Implementierung

```
private class AdjListe implements Graph {
  private Vertex [] vertices;
  private Liste [] edges;
  public AdjListe (Liste v) {
    int l = v.length ();
    /* Knoteninitialisierung wie oben */
    /* Kanteninitialisierung : */
    edges = new Liste [l];
```

[1]Vorgänger: "transponierte" Ajanzenzliste

```
    for (int i=0; i<l; ++i) {
      edges [i] = new Liste ();
    }
  }
  public Vertex [] getVertices () {
    return vertices;
  }
  public void addVertex (Vertex v) {
    /*
      hier: zusätzlichen Knoten eintragen, leere Nachfolgerliste
      dazu anlegen
    */
  }
  public void deleteVertex (Vertex v) {
    /* hier: Knoten aus allen Listen entfernen */
  }
  public void addEdge (Vertex u, Vertex v) {
    if (!containsEdge (u,v))
      edges [u.getIndex()].insert (v);
  }
  public void deleteEdge (Vertex u, Vertex v) {
    edges [u.getIndex()].delete (v);
  }
  public boolean containsEdge (Vertex u, Vertex v) {
    return edges [u.getIndex()].search(v) != null;
  }
  public Liste successors (Vertex v) {
    return edges [v.getIndex()];
  }
}
```

4.1.5 Mischform

- Verwende zwei eindimensionale Arrays "from" und "to" mit Referenzen auf Kantenobjekte.
- Es gibt einen Referenzenpfad zu einem Objekt von from[i] nach to[j] genau dann wenn der Graph G eine Kanten vom Knoten i zu Knoten j enthält.
- Vorteile
  - geringer Platzbedarf $O(|V| + |E|)$ wie Adjazenzlisten
  - Initialisierung in $O(|V| + |E|)$ Zeit wie Adjazenzlisten
  - Vorgängerliste leicht erhältlich, ähnlich wie Adjazenzmatrix

- Nachteile
  - Entscheidung, ob eine Kante $(i, j) \in E$ benötigt $O(|E| + |V|)$ Zeit im average case (wie Adjazenzlisten)

4.1.6 Kantenorientierte Darstellung

- Methode: Array (oder verkettete Liste) von Kanten
- Prinzip: Adressierung jeder Kante über einen Index
- Speicherung für jede Kante:
  - Vergängerknoten
  - Nachfolgerknoten
  - ggf. Beschriftung (Markierung, Kosten)

4.2 Dichte eine Graphen

Man unterscheidet Typen von Graphen:

- vollständiger (complete) Graph:
  $|E| = |V|^2$ (gerichtet) bzw. $|E| = \frac{|V|(|V|-1)}{2}$ (ungerichtet)
- dichter Graph (dense): $|E| \approx |V|^2$
- dünner Graph (sparse): $|E| << |V|^2$

Die Dichte eines Graphen ist ein wichtiges Kriterium bei der Auswahl einer geeigneten internen Repräsentation.

4.2.1 Expansion

DIe Expansion $X_G(v)$ eines Graphen G in einem Knoten v ist ein Baum, der wie folgt definiert ist:

- Falls v keine Nachfolger hat, ist $X_G(v)$ nur der Knoten v.
- Falls $v_1, \ldots, v_k$ die Nachfolger von v sind, ist $X_G(v)$ der Baum mit der Wurzel v und den Teilbäumen $X_G(v_1), \ldots, X_G(v_k)$

Anmerkungen

- Die Knoten des Graphen können mehrfach im Baum vorkommen.
- Ein Baum ist unendlich, falls der Graph Zyklen hat.
- Der Baum $X_G(v)$ stellt die Menge aller Pfade dar, die von v ausgehen.

Beispiel Beispielgraph G_1 in seiner Expansion $X_G(v)$ mit $v = 0$.

4.3 Graphdurchlauf

4.4 Kürzeste Wege

4.5 Minimal Spanning Tree

4.6 Exkurs: Union-Find-Strukturen