Datenstrukturen und Algorithmen

Zusammenfassung von M.Sondermann
SS 2002

Vorwort

Dies ist eine Zusammenfassung der Vorlesung 'Datenstrukturen und Algorithmen' des Sommersemesters 2002 bei Prof. Dr. Kobbelt. Es erhebt keinerlei Anspruch auf Korrektheit und Vollständigkeit und stellt **keine** offizielle Veröffentlichung des Lehrstuhls dar. Der meiste Inhalt stammt aus den von Prof. Dr. Kobbelt veröffentlichten Vorlesungsfolien. Bei Fehlern, Verbesserungsvorschlägen oder sonstigen Anregungen wird um eine Email an die unten angegebene Adresse gebeten.

Matthias Sondermann, matthias.sondermann@post.rwth-aachen.de

Letzte Änderung : 21.August 2002

Inhaltsverzeichnis

1 Datenstrukturen

1.1 Definition

- statische Beziehungen
- (explizite) funktionale Zusammenhänge

1.2 abstrakte Datentypen

- spezifizieren Form und Funktionalität der zu verarbeitenden Daten
- Top-Down-Software-Entwurf

1.3 abstrakte Datentypen-Typen

- **Aufzählungstypen**
  - bool, enum
- **skalare Typen**
  - 'Zahlen' mit 1-dimensionalen Wertebereich
  - char, int, float, ...
  - bei Implementierung meist endlicher Wertebereich
- **zusammengesetzte Typen**
  - endliche/feste Kombination von skalaren Typen
  - z.B. k-dim Vektoren, Adresseneintrag
- **lineare Struktur**
  - beliebige Sequenz von Basisobjekten (skalare/zusammengesetzte Typen) mit variabler Länge
  - z.B. Arrays, Listen (einfach/doppelt verkettet), Queue (FIFO), Stack (LIFO)
  - unterscheiden sich in der Zugriffsfunktionalität
- **Baumstruktur**
  - hierarchische Struktur
  - Vater-Sohn-Beziehung
  - keine Zyklen, eindeutige Vorfahren
- **Graphen**
  - beliebige topologische Struktur
  - Nachbarschaftsbeziehungen
  - Spezialfälle: planare, gerichtete, zyklenfreie Graphen

1.4 konkrete Datentypen

- spezifizieren Form und Funktionalität der zu verarbeitenden Daten, aber keine Axiome

1.4.1 Listen

- L=$\{x_1, ..., x_n\}$
- Zugriff per Index (random access) - Nachteile: garbage collection, statische Obergrenze
- per Marker (sequential access) - Nachteile: lineare Suche - Vorteil: beliebiges Löschen/Erweitern
- einfach verkettet : Delete() : $marker.next \ = \ marker.next.next;$
- doppelt verkettet : Delete(): $marker.prev.next \ = \ marker.next;$
  $marker.next.prev \ = \ marker.prev;$
  $marker \ = \ marker.prev;$
- einfach verkettet : Insert(Y) : $Y.next \ = \ marker.next \ ; \ marker.next = Y;$
- doppelt verkettet : Insert(Y): $Y.prev \ = \ marker;$
  $Y.next \ = \ marker.next;$
  $Y.prev.next \ = \ Y; \ \ Y.next.prev \ = \ Y;$

1.4.2 Queue / Warteschlange

- Prinzip : first in - first out (FIFO)
- da nur am Anfang eingefügt und am Ende gelöscht wird, ist keine garbage collection notwendig und die maximale Länge wir alles bekannt vorausgesetzt $\Rightarrow$ **Array-Implementierung**
- jeder Stack hat die Form : $Enq(x_n, Enq(x_{n-1}, ..., Enq(x_1, Create())))$
- Implementierung mit $Datentyp\ S[Länge]$ und $int\ front,\ back$
- Create() : $front\ =\ back\ =\ 0;$
- Empty() : $return\ (front == back);$
- Get() : $if\ (front! = back)\ return\ S[front]\ else\ Q.Empty.Error;$
- Deq() : $if\ (front! = back)\ front = (front + 1)\%Länge\ ;\ elseQ.Empty.Error;$
- Enq(x) : $S[back] = x;\ back = (back + 1)\%Länge\ ;\ if\ (front == back)\ Q.Full.Error;$

1.4.3 Stack / Keller

- Prinzip : last in - first out (LIFO)
- Liste mit eingeschränkter Funktionalität
- einfügen nur am Anfang
- auslesen / entfernen nur am Anfang
- jeder Stack hat die Form : $Push(x_1, Push(x_2, ..., Push(x_n, Create())))$
- da nur am Anfang eingefügt und am Ende gelöscht wird, ist keine garbage collection notwendig und die maximale Länge wir ales bekannt vorausgesetzt $\Rightarrow$ **Array-Implementierung**
- Implementierung mit $Datentyp\ S[Grösse]$ und $int\ top$
- Create() : $top = -1;$
- Empty() : $return(top == -1);$
- Top() : $if(top! = -1)\ return(S[top])\ ;\ else\ Stack.Empty.Error;$
- Push(x) : $top + +\ ;\ if(top < Grösse)\ S[top] = x\ ;\ else\ Stack.Full.Error;$
- Pop() : $if(top! = -1)\ top - -\ ;\ else\ Stack.Empty.Error;$
- Merke : rekursive Algorithmen lassen sich meißt auch mit einem Stack iterativ formulieren
- LIFO entspricht der Abarbeitungsreihenfolge der geschachtelten Prozeduren
- Beispiel : systematsche Suche in einem Labyrinth (bei Sackgassen zurückgehen)

1.4.4 Bäume

- hierarchische Datenstruktur mit : Zusammenfassung von Gruppen, eindeutige Schachtelung
- typische Anwendungen :
  Such-/Entscheidungsprobleme, strukturierte Aufzählungen, Komplexitätsreduktion
- Ein Baum B=(V,E) besteht aus einer Menge von Knoten V und Kanten E mit
  $\exists$ keine gerichteten oder ungerichteten Zyklen
  jeder normale Knoten hat genau einen Vorgänger
  $\exists$ genau ein Wurzelknoten, der keinen Vorgänger besitzt
  $\forall$ Knoten $\exists$ ein eindeutiger Pfad, der ihn mit dem Wurzelknoten verbindet
- jeder Knoten ist Wurzelknoten eines zugehörigen Subbaumes
- Knoten ohne Nachfolger heissen Blätter, alle anderen heissen innere Knoten
- $w_1, ..., w_i$ Nachfolger des Knotens v:
  height(subtree(w_i))$\geq$height(subtree(v))-2 $\Rightarrow$ der Baum ist balanciert
- Menge von Knoten V=($v_1, ..., v_n$) , v beliebiger Datentyp
- Menge von Kanten E=($(a_1, b_1), ..., (a_m, b_m)$)
- falls (a,b)$\in$ E, dann ist v_a Vorgänger von v_b und v_b Nachfolger von v_a
- eine Folge von Kanten $(i_1, i_2), (i_2, i_3), ..., (i_{k-1}, i_k)$ heißt Pfad von v_{i1} nach v_{ik}

- Arrayimplementierung : N(k)=Anzahl der Knoten der Tiefe k
  N(k)=2N(k-1) $\Rightarrow$ N(k)=2^k
  Speicherung der Knoten der Tiefe k in A[$2^k ... 2^{k+1} - 1$]
  jeder Knoten A[i] findet seine Nachfolger in A[2i] und A[2i+1]
- Für n Knoten ist die maximale Höhe des Binärbaumes $H_{max}(n) = n - 1$
  die minimale Höhe $H_{min}(n) = \lceil log(n+1)/log(2) \rceil - 1$
  die Tiefe+1 eines Knotens beschreibt die Anzahl der Zugriffe, um ihn zu finden
- Traversierung :
  Präfix : output(value(t)), traverse(left(t)), traverse(right(t))
  Infix : traverse(left(t)), output(value(t)), traverse(right(t))
  Postfix : traverse(left(t)), traverse(right(t)), output(value(t))
- Suchen :
  $BoolSearch(ElementX, BinTree\ T)$
  $if\ (X == Value(T))\ return\ true;$
  $else\ if\ (Leaf(T))\ return\ false;$
  $else\ if\ (X < Value(T))\ Search(X, Left(T));$
  $else\ Search(X, Right(T));$
- Einfügen :
  $Insert(X, Create())\ =\ Node(Create(), X, Create());$
  $Insert(X, Node(L, Y, R))\ =\ if(X \leq Y) Insert(X, L);\ if(X > Y) Insert(X, R);$
- Löschen :
  $Remove(X, Node(L, Y, R))\ =$
  $if(L \neq Create())\ Node(Remove(Max(L), L), Max(L), R);$
  $else\ if(R \neq Create())\ Node(L, Min(R), Remove(Min(R), R));$
  $else\ Create();$

1.4.5 Prioritätsschlangen / Heap

- Warteschlange (Queue) mit Vordrängeln
- Einfügen am Ende der Schlange, jedes Element hat eine Priorität
- Deq() liefert immer das Element mit der höchsten Priorität
- Element mit maximaler Priorität ist im Wurzelknoten gespeichert
- Sortierung der ELemente nur entlang der Pfade im Binärbaum

1.4.6 Graphen

- Darstellung allgemeiner Beziehungen zwischen Objekten/Elementen
- **Zusammenhängend :** Für jedes Paar v_i, v_j von Knoten existiert ein Pfad von Kanten von v_i nach v_j oder (und) umgekehrt
- **Vollständig :** Zwischen zwei Knoten existiert eine Kante - m=n·(n-1)/2
- **Isomorph :** Graphen (V_1, E_1) und (V_2, E_2) durch Perm. der Knoten ineinander überführbar
- **Planar :** Zerlegung der Ebene in disjunkte Zellen - Kompatibilitätsbedingung zwischen Kanten. Es ist möglich die Knoten so zu platzieren, dass sich die Kanten nicht kreuzen

1.5 verschieden Baumimplementierungen

1.5.1 optimaler Suchbaum

- Zugriffswahrscheinlichkeit auf die Elemente a priori bekannt,
  versuche die Elemente mit häufigem Zugriff in der Nähe der Wurzel zu speichern
- optimale Suchbäume müssen nicht unbedingt minimale Höhe besitzen
- Bestimmung des optimalen Suchbaumes → dynamische Programmierung
- Speichere Zwischenerbegnisse in den 2d-Arrays:
  C[i,j] : Kosten des opt. Suchbaumes für die Knoten $N_i ... N_j$ und $l_{i-1} ... l_j$
  W[i,j] : Zugriffswahrscheinlichkeit für den entsprechenden Teilbaum
  R[i,j] : Wurzelknoten des optimalen Suchbaumes
- Algorithmus
  $OptimalTree(P[1...n], Q[0...n], n)$
  $for(i = 1; i \leq n + 1; i + +)$
  $C[i, i-1] = W[i, i-1] = Q[i-1];$
  $for(l = 1; l \leq n; i + +)$
  $j = i + l - 1;$
  $W[i, j] = W[i, j-1] + P[j] + Q[j];$
  $C[i, j] = min\{C[i, k-1] + C[k+1, j]\} + W[i, j];$
  $R[i, j] = argmin\{C[i, k-1] + C[k+1, j]\};$
- Aufwandsabschätzung O(n^3) → lohnt nur bei vielen Anfragen

1.5.2 balancierte Bäume

- Balance verhindert den worst case O(n) bei 'normalen' Suchbäumen
- Gewichtsbalance: Für jeden Knoten N unterscheidet sich die Anzahl der Knoten im linken und rechten Teilbaum um maximal eins.
- Höhenbalance: Für jeden Knoten N unterscheidet sich die Höhe des linken und rechten Teilbaumes um maximal eins.

1.5.3 AVL-Bäume

- wie normale binäre Suchbäume + Speicherung des Ungleichgewichts in jedem Knoten
- nach Insert() und Delete() Wiederherstellung der Balance durch **Rotation**
- Übergewicht 'außen': einfache Rotation
- Übergewicht 'innen': doppelte Rotation

1.5.4 Rot-Schwarz-Bäume

- Jeder Knoten ist rot oder schwarz
- Es folgen **nie** zwei rote Knoten direkt aufeinander
- Alle Pfade von der Wurzel zu den Blättern haben dieselbe Anzahl schwarzer Knoten
- Wurzel und Blätter sind immer schwarz
- Verhältnis längster zu kürzester Pfad = 2:1
- Insert()
  jeder eingefügte Knoten ist zunächst rot
  bei Konsistenzverletzung: beachte Farbe des Onkels
  Onkel rot: Umfärben
  Onkel ist schwarz: einfache Rotation, umfärben oder doppelte Rotation

- Delete()
  Fall 1: Bruder rot:
  umfärben, rotieren → Bruder schwarz
  Fall 2: Bruder schwarz:
  - beide Neffen schwarz → umfärben
  - innerer rot, äußerer schwarz → rotieren, umfärben ⇒
  - innerer schwarz, äußerer rot → rotation, umfärben

1.5.5 (a,b)-Bäume

- Idee: Zusammenfassung von größeren Mengen von Knoten → passen besser in HD-Blöcke
- **Hysterese**: nicht in jedem Schritt muß rebalanciert werden
- N-äre Suchbäume:
  jeder Knoten hat bis zu n-1 verschiedene Schlüssel und bis zu n Nachfolger
  K.n=Füllungsgrad, K.x[i]=Schlüssel, i=1,...,K.n
  K.s[j]=Nachfolger, j=0,...,K.n
  Boolean K.leaf=t/f, ob K ein Blatt ist
- (a,b)-Bäume sind balancierte n-äre Suchbäume
  alle Blätter haben dieselbe Tiefe
  alle Knoten außer die Wurzel haben min. t-1 Schlüssel mit $t \geq 2$
  alle inneren Knoten außer die Wurzel haben min. t Nachfolger
  am besten: t=n/2
  die meißten Daten stehen in den Blättern
  Rebalancierung durch Rotation
- Insert()
  anstatt wie bei Binärbäumen, wo immer ein neuer Blattknoten ergänzt wird,
  wird bei B-Bäumen solange kein neuer Knoten hinzugefügt, bis der Blattknoten voll ist.
  One-Pass: teile die Knoten entlang des Suchpfades schon auf dem 'Hinweg', falls diese voll sind

2 Algorithmen

2.1 Definition

- schrittweise Modifikation von Daten zur Lösung eines Problems
- dynamische Prozesse, (implizite) funktionale Zusammenhänge

2.2 Kriterien

- Determinismus
- Input ($\sharp \geq 0$)
- Output ($\sharp \geq 1$)
- Terminierung ($\sharp$ steps $< \infty$)

2.3 Analyse

- (partielle) Korrektheit
- Vollständigkeit
- Komplexität
- Robustheit (bei inkorrekten Eingaben 'garbage in - garbage out')

2.4 Begriffe

- **Effizienz** = gute Ausnutzung von Resourcen
- **Resourcen** = Rechenzeit, Speicherplatz
- **Aufwand/Komplexität** = tatsächlicher Verbrauch von Resourcen
  qualitative Aussage interessant : wie verändert sich der Aufwand wenn anderer Input
- **Performanzfaktoren :** Prozessor, Hauptspeicher, Caches, Compiler, Betriebssystem
- **abstrakte Analyse :** zählt die wesentlichen Operationen auf einem idealen Computer

2.5 Schranken

- O(f)= $\{g \mid \exists c > 0, \exists n > 0, \forall x \geq n \ : \ g(x) \leq cf(x)\}$
- Ω(f)= $\{g \mid \exists c > 0, \exists n > 0, \forall x \geq n \ : \ g(x) \geq cf(x)\}$
- Θ(f)= O(f)$\cap\Omega(f) = \{g \mid \exists c > 0, \exists n > 0, \forall x \geq n \ : \ f(x)/c \leq g(x) \leq cf(x)\}$

2.6 Komplexitätsklassen

- O(1) = Elementaroperation
- O(log(n)) = binäre Suche
- O(n) = lineare Suche
- O(n·log(n)) = Sortieren
- O(n^2) = Sortieren
- O(n^3) = Invertieren von Matrizen
- O(2^n) = Labyrinthsuche (vollständig)
- O(n!) = Zahl von Permutationen

2.7 Standardabschätzungen

2.7.1 Reihensummen

- geschachtelte Schleifen
  - for(i=0; i<n; i++) O(1) = O(n)
  - for(i=0; i<n; i++) O(i) = O(n^2)
  - for(i=0; i<n; i++)
    for(j=0; j<n; j++) O(1) = O(n^2)
  - for(i=0; i<n; i++)
    for(j=0; j<i; j++) O(1) = O(n^2)
- generell : der Aufwand T_k(n) einer k-fach geschachtelten Schleife besitzt als obere Schranke ein Polynom vom Grad k : $T_k(n) \; = \; O(\sum\limits_{i=0}^{k} a_i n^i) \; = \; O(n^k)$
- Integralabschätzungen
  Summanden monton wachsend/fallend
  Interpretation als Approximation eines bestimmten Integrals
  $\int\limits_0^n f(x)dx \; \approx \; \sum\limits_{i=0}^{n-1} f(i+h) \qquad h \in [0,1)$
- oder mit vollständiger Induktion

2.7.2 Rekursionsgleichungen

- **Elimination**
  F(n)=n+F(n-1)=n+(n-1)+F(n-2)= ... =n(n-1)/2+F(0)=O(n^2)
  Abschätzung nach oben, typisch T(n)=aT(n/2)+b,
  einfach, wenn n=2^k, sonst finde m=2^k mit m/2<n≤m, dann gilt T(n)≤T(m)
- **Master-Theorem**
  $$\text{T(n)} = \begin{cases} c & n = 1 \\ aT(n/b) + cn & n > 1 \end{cases}$$
  $$\text{T(n)} = \begin{cases} O(n) & a < b \\ O(nlogn & a = b \\ O(n^{log_b(a)}) & a > b \end{cases}$$
  sei n=b^k: T(n) = aT(n/b) + cn
  = a^2T(n/b^2) + a c n/b + cn
  = a^3T(n/b^3)+a^2 c n/b^2 a c n/b + cn
  = ... = cn $\sum\limits_{i=0}^{k} (a/b)^i$
  ⇒ a< b : T(n)≤ cn$\frac{1}{1-a/b}$ = O(n)
  a=b : T(n)= cn(k+1)=O(nk)=O(nlogn)
  a>b : T(n)= cn$\frac{(a/b)^{k+1}-1}{a/b \; -1}$ = O($a^{log_b n}$) = O($n^{log_b a}$)
- direkter Ansatz

2.8 Entwurfsparadigmen

2.8.1 Top-down

- bei der Implementierung eines Teilschrittes werden die Sub-Teilschritte als Blackboxes verwendet
- betrachte die Teilschritte in der Reihenfolge der geringsten Querbezüge
- vemeide Seiteneffekte

2.8.2 Bottom-up

- löse Teilproblem und kombiniere diese zu einer Gesamtlösung
- möglich bei unvollständiger Spezifikation
- bessere Test-Suites parallel zur Entwicklung (Entwurf & Implementierung)
- Code reuse

2.8.3 Divide & conquer

- gegeben : Problem der Grösse n
  Divide : zerlege das Problem rekursiv in k Teilprobleme der Grösse n/k
  Conquer : löse kleine Probleme direkt
  kombiniere die Teillösungen zur Gesamtlösung
- meistens k=2, rekursive Implementierung

2.8.4 dynamische Programmierung

- reduziere gegebenes Problem auf kleinere Teilprobleme
  löse Teilprobleme
  (memoization) speichere Teillösungen in Tabelle
  kombiniere Teillösungen zur Gesamtlösung
- durch Speicherung der Zwischenergebnisse vermeidet man doppelte Berechnung

2.8.5 Unterschied D&C - dyn. Programmieren

- top-down vs. bottom-up
- rekursive vs. iterative Implementierung

2.9 Sortieren

2.9.1 Allgemeines

- Def. : Bringe eine gegebene Folge von Datenobjekten in eine wohledefinierte Reihenfolge
  Finde eine Permutation $\Pi = \{\pi(i)\}$, so dass $\forall$ i=1,...,n-1 : $R_{\pi(i)}.key \leq R_{\pi(j).key}$
- Folge $\neq$ Menge (doppelte Objekte)
- Reihenfolge abhängig von Ordnungsrelation, Sortierschlüssel muss vorhanden sein
- direktes Suchen : Umkopieren der Objekte
  indirektes Suchen : Erzeugung einer Permutationstabelle
- Unterscheidung zwischen einfachen, höheren und speziellen Algorithmen
- Stabilität : Reihenfolge von Objekten mit gleichem Schlüssel bleibt erhalten

2.9.2 Selection-Sort

- Suche kleinstes Element und kopiere es an die erste Stelle, danach sortiere die restl. (n-1) Elemente
- Vorteil: Jedes Objekt wird nur dreimal kopiert (lohnt sich bei großen Objekten)
- SelectionSort(A,n)

```
i=0;
while (i<n)
   min=i; j=i+1;
   while (j<n)
      if (A[j]<A[min]) min=j; j++;
   swap(A[i],A[min]); i++;
```

- Aufwandsabschätzung : best=worst=average case : O(n^2)

2.9.3 Bubble-Sort

- Nutze die Vergleiche in Selection-Sort, um die Liste vorzusortieren
- Innere Schleife läuft in umgekehrter Reihenfolge
- Swap-Operationen in der inneren Schleife
- BubbleSort(A,n)

```
i=0;
while (i<n)
   j=n;
   while (j>i)
      if (A[j]<A[i]) Swap(A[j],A[i]);
      j- -;
   i++;
```

- Aufwandsabschätzung :
  best case 0, worst case n(n-1)/2 $\Rightarrow$ O(n^2), average case n(n-1)/4 $\Rightarrow$ O(n^2)
- Vorteil: pro Durchlauf mehr Ordnung
- Nachteil: viele Kopieroperationen, Vorsortierung wird nicht genutzt

2.9.4 Insertion-Sort

- Vorbild: manuelles Sortieren
- Nehme das nächste Element und füge es in der schon sortierten Teilfolge richtig ein
- InsertonSort(A,n)

```
i=1;
while (i<n)
   h=A[i]; j=i;
   while (j>0 ∧ A[j-1]>h)
      A[j]=A[j-1]; j- -;
   A[j]=h; i++;
```

- Aufwandsabschätzung:
  best case n-1, worst case n(n-1)/2 $\Rightarrow$ O(n^2), average case n(n-1)/4 $\Rightarrow$ O(n^2)
  Kopieren um Faktor 3 besser, da einfaches Kopieren anstatt Swap

2.9.5 Zusammenfassung 'einfache Sortieralgorithmen'

- **Vergleichen**
  - Selection $n^2/2$ $n^2/2$ $n^2/2$
  - Bubble $n^2/2$ $n^2/2$ $n^2/2$
  - Insertion n $n^2/4$ $n^2/2$
- **Kopieren**
  - Selection $3(n-1)$ $3(n-1)$ $3(n-1)$
  - Bubble 0 $3n^2/4$ $3n^2/2$
  - Insertion $2(n-1)$ $n^2/4$ $n^2/2$

2.9.6 Quick-Sort

- Idee: Divide and Conquer
- wähle ein Element aus (Pivot) und teile Folge in zwei Sub-Folgen auf mit $R_L \leq$ R und $R_R \geq$ R, sortiere R_L und R_R durch rekursiven Aufruf und setze Teilfolgen zusammen: $R_L \oplus R \oplus R_R$
- QuickSort (A,l,r)
  if (l<r)
  m=Partition(A,l,r); QuickSort(A,l,m-1); QuickSort(A,m+1,r);
- Partition(A,l,r)
  i=l-1; j=r;
  while (i<j)
  i++; while(A[i]<A[j]) i++;
  j–; while(A[j]>A[i]) j–;
  swap([i],A[j]);
  swap(A[i],A[j]); swap(A[i],A[r])
- Aufruf mit QuickSort(A,1,n)
- Auwandsabschätzung : best case: O(nlog(n)), worst case: O(n^2), average case: O(nlog(n))
- Quick-Sort ist schnell, wenn die beiden Teilfolgen nach Partition() ca. die gleiche Grösse haben

2.9.7 Merge-Sort

- Idee: teile die Folge ohne Vorsortierung in zwei gleiche Teile, sortiere die Teilfolgen, kombiniere Teilergebnisse
- Merge-Sort (rekursiv):
  Merge-Sort(A,l,r)
  if (l<r)
  m=$\lfloor$(l+r)/2$\rfloor$;
  MergeSort(A,l,m); MergeSort(A,m+1,r); Merge(A,l,m,r);
- Merge(A,l,m,r)
  new B[r-l+1]; i=0; j=l; k=m+1;
  while (j$\leq$m) $\wedge$ (k$\leq$r)
  if (A[j]$\leq$ A[k]) B[i++]=A[j++];
  else B[i++]=A[k++];
  while (j$\leq$m) B[i++]=A[j++];
  while (k$\leq$r) B[i++]=A[k++];
  for (i=l; i$\leq$r; i++) A[i]=B[i-l];
- Aufwandsabschätzung : best=worse=average case: O(nlog(n))

- Merge-Sort (iterativ)

```
MergeSort(A,n)
 new B[n]; m=1; k=n;
 while (m<n)
    B=A; k=⌈k/2⌉;
    for (i=0; i<2km; i=2m)
       p=0; q=o;
       for (j=0; j<Min(2m,n-i); j++)
          if (p<m) ∧ (B[i+p]<B[i+m+p]) ∧ (q≥Min(m,n-i-m))
             A[i+j]=B[i+p]; p++;
          else A[i+j]=B[i+m+q]; q++;
    m=2m;
```

2.9.8 Heap-Sort

- Idee : mit einem Heap kann das kleinste Element schnell gefunden werden
  Aufwand nlog(n) zum Wiederherstellen der Heap-Bedingung
- konvertiere ein beliebiges Array in einen Heap (A[⌊ i/2 ⌋] ≥A[i])
  entferne sukzessive das Top-Element und stelle die Heap-Eigenschaft wieder her
- A[1,...,n] ein Heap, nach Entfernen wird A[n] nach A[1] kopiert und versickert,
  danach das selbe mit A[1,...n-1]
- ConvertToHeap(A,n)

```
 for (i=⌊n/2⌋; i≥1; i- -) Sink(A,n,i);
 (bei i=p hat das Array A ab Index p die Heap-Eigenschaft)
```

- Sink(A,n,i)

```
 while(i≤ ⌊n/2⌋)
    j=2i;
    if (j<n) ∧ (A[j+1]>A[j]) j++;
    if (A[j]>A[i]) Swap(A[j],A[i]); i=j;
    else i=n;
```

- HeapSort(A,n)

```
 ConvertToHeap(A,n)
 for (i=n; i≥1; i–)
    Swap(A[1],A[i]); Sink(A,i-1,1);
```

- Aufwandsabschätzung : best case: O(n), worst und average case: O(nlog(n))

2.9.9 Zusammenfassung 'höhere Sortieralgorithmen'

Algorithmus	average	worst
Quick-Sort	O(nlog(n))	O(n^2)
Merge-Sort	O(nlog(n))	O(nlog(n))
Heap-Sort	O(nlog(n))	O(nlog(n))

2.9.10 optimaler Sortieralgorithmus

- finden durch Entscheidungsbaum $\Rightarrow$
  1. Zahl der möglichen Permutationen n! $\Rightarrow$ jeder E-Baum hat min. n! Blätter
  2. max. Zahl von Blättern in einem Binärbaum der Höhe h ist 2^h

  $\Rightarrow 2^h \geq$n!
- Ω(nlog(n)) ist eine untere Schranke für die Komplexität des allgemeinen Sortierproblems
- Fazit: jeder nur auf Vergleichen basierende Sortieralg. hat einen Mindestaufwand von nlog(n)
- Merge-Sort und Heap-Sort sind optimale Sortieralgorithmen

2.9.11 Stabilität

- Ein Sortieralgorithmus heißt stabil, wenn sich die relative Reihenfolge von gleichen Elementen während des Sortierens nicht ändert

2.9.12 Counting-Sort

- Annahme $R_1, R_2, ..., R_n \in \{1, ..., k\}$
- Idee : bestimme zu jedem R_i die Zahl der Elemente $\leq R_i$ und sortiere R_i an die entspr. Stelle
- CountingSort(A,B)
  for (i=1; i≤k; i++) C[i]=0;
  for (i=1; i≤n; i++) ++C[A[i]];
  for (i=2; i≤k; i++) C[i]+=C[i-1];
  for (i=n; i≥1; i- -) B[C[A[i]]]=A[i]; C[A[i]]- -;
- Aufwandsabschätzung : O(n+k)
- Counting-Sort ist nur sinnvoll, wenn k=O(n) und damit T(n)=O(n)
- es werden keine Vergleiche verwendet und Counting-Sort ist stabil

2.9.13 Radix-Sort

- Annahme: $R_1, R_2, ..., R_n \in \{0, ..., k^d - 1\}$
- d-stellige Zahlen zur Basis k, Wörter der Länge d aus einem Alphabet der Grösse k
- Radix-Sort(A)
  for (i=0; i<d; i++) 'sortiere A stabil nach k^i-wertiger Stelle'
- Aufwandsabschätzung : T(n)=Ω(nlog(n))

2.9.14 Bucket-Sort

- Annahme $R_1, R_2, ..., R_n$ gleichverteilt aus [0,1)
- Idee :
  1. unterteile [0,1) in n Buckets [0,1/n],[1/n,2/n],...,[(n-1)/n,1]
  2. füge die R_i in die Buckets ein (erwartet ist ein Element pro Bucket)
  3. sortiere die Buckets
  4. hänge die Buckets aneinander

- BucketSort(A,R) Eingabe: A[0,...,n-1],Ausgabe: R[0,...,n-1]

```
for (i=0; i<n; i++)
    B[i]=[ ];                   alle Buckets leer
for (i=0; i<n; i++)
    B[⌊nA[i]⌋].push(A[i]);      Buckets gefüllt
for (i=0; i<n; i++)
    sort(B[i]);                 Buckets sortiert
R=[ ];
for (i=0; i<n; i++)
    for (j=0; j<B.size(); j++)
        R.push(B[i][j]);        Buckets zusammengehängt
```

- Aufwandsabschätzung : O(n) wird erwartet (Achtung letzte for-Schleife ist auch O(n))

2.9.15 Zusammenfassung 'spezielle Sortieralgorithmen'

- Counting-Sort O(n+k)
- Radix-Sort O(d(n+k))
- Bucket-Sort O(n) erwartet

2.10 Suchen

2.10.1 k-Selektion

- Selektionsproblem:
  gegeben: n (verschiedene) Zahlen A[1...n], eine Zahl k mit 1≤k≤n
  gesucht: k-kleinstes Element von A, d.h. dasjenige x∈A mit $|\{i \ : \ A[i] < x\}| = k-1$
- k=1 suche Minimum O(n)
  k=2 suche zweitkleinstes Element O(n)
  ... k-kleinstes Element O(kn)
  allgemein: k=O(n)⇒O(kn)=O(n^2)
- Selektion durch Sortieren

```
SortSelect(A,k)
    sort(A);
    return A[k];
```

  Aber: Aufwand O(nlog(n))
- Selektion durch Partitionierung

```
PartitionSelect (A,l,r,k)
if (l==r)
    return A[p]
m=Partition(A,l,r);
i=m-l+1;
if (k==i)
    return A[m];
else if (k<i)
    return PartitionSelect(A,l,m-1,k);
else return PartitionSelect(A,m+1,r,k-i);
```

- Aufwandsabschätzung : average case O(n), worst case O(n^2) (vergl. Quick-Sort)

2.10.2 Median der Mediane

- Select (A,k)
  n=|A|
  if (n<50) return SortSelect(A,k); magische Konstante
  M={$SortSelect(A[5i+1...5i+5])$: i=0,...,$\lfloor n/5 \rfloor$}; $\lfloor n/5 \rfloor$ Gruppen mit 5 El. / Medianbest.
  m=Select(M,$\lfloor |M|/2 \rfloor$); Median der Mediane
  $S_< = \{A[i] \ : \ A[i] < m\}$;
  $S_= = \{A[i] \ : \ A[i] = m]\}$;
  $S_> = \{A[i \ : \ A[i] < m]\}$;
  if ($|S_<| > k$) return Select($S_<$,k); k-te EL. liegt in $S_<$
  if ($|S_>| + |S_|$ =<k) return m; k-te EL. liegt in $S_=\{m\}$
  return Select($S_>, k - |S_<| - |S_=$); k-te El. ist das (k-$|S_<| + |S_=|$)-te El.
- Aufwandsabschätzung O(1),O(n),T(n/5),O(n)<T(3n/4) $\Rightarrow$ O(n)

2.10.3 String Matching (allgemein)

- gegeben: Text T[1...n] und Muster P[1...m] (pattern)
  gesucht: erster/alle Vorkommen von P in T

2.10.4 naives String Matching

- NaiveStringMatch(T,P)
  for (s=0; s≤n-m; s++)
    i=1;
    while (i≤m and P[i]==T[s+i]) i++;
    if (i==m+1) print 'Muster gefunden an Stelle' s+1;
- Aufwandsabschätzung : T(n,m)=O((n-m+1)m)=O(nm)

2.10.5 Rabin-Karp Algorithmus

- Idee: Fasse Wörter aus Σ als Zahlen aus $\mathbb{N}$ auf
  String 145=1⊕4⊕5 (Concenation) , Zahl 145=100+40+5 (Addition)
- Zwei Worte aus Σ sind gleich, wenn die entsprechenden Zahlen gleich sind
  $\Rightarrow$ Vergleich von Zahlen besser
- Berechnung der Zahlen mit dem Horner-Schema:
  p=0;
  for (i=1; i≤m; i++)
    p=p·10+P[i];
  Aufwand: O(m)
- Aufwand der Berechnung aller Zahlen (p und alle t): O((n-m+1)m)
  oder besser: berechen t_0 und berechne dann t_{s+1} aus t_s durch
  $t_{s+1} = 10(T[s+1] \cdot 10^{m-1}) + T[s+m+1]$

- RabinKarpPre(T,P)
  h=10^{m-1} mod q ; p=0; t=0; Vorberechnung O(m)
  for (i=1; i≤m; i++)
    p+=10·p+P[i] mod q ; 2x Horner-Schema O(m)
    t+=10·t+T[i] mod q ;
  for (s=0; s≤n-m; s++) O(n)
    if (p=t)
      if(P[1...m]=T[s+1...s+m])
        print 'Muster gefunden an Stelle' s+1;
    if (s<n-m)
      t=10·(t-h·T[s+1])+T[s+m+1] mod q ; Muster nach rechts verschieben O(1)

vs O(m)
- Aufwandsabschätzung : worst case O((n-m+1)m),
  average case O(n) wenn q≥m und erwartete Matches in O(1)

2.10.6 Knuth-Morris-Pratt Algorithmus

- gegeben: Text[1...n], Muster[1...m]
- beim naiven Ansatz und bei Rabin-Karp werden Symb. aus T[] mehrfach vergl. (bis zu m-mal)
- allgemein: Suche das größte j, für das gilt: P[1 ... j] = P[i-j+1 ... i]
- Vorberechnung (Prefixfunction) des Suchmusters in Q[] mit Q[i]=j ↔ P[1 ... j] = P[i-j+1 ... i]
  und Q[0]=0 und $0 \leq Q[i] \leq i$
- Aufwandsabschätzung: O(n)

2.10.7 Boyer-Moore-Algorithmus

- anders als beim KMP wird hier das Wissen über die Ungleichheit von Symbolen genutzt
- Vorwissen durch Skip-Array mit z.B. S[7,6,1,5,3,0,7,7,7,...] für P=1314225 mit S[i] und P[i] $\in \mathbb{N}_0$
- Anzahl der Stellen, die übersprungen werden können ist $max(1\ ,\ S[x]-(P.length - richtige\ Matchings))$
- Aufwandsabschätzung: worst case O((n-m+1)m+|Σ|) bei konstanter Folge
  große Skips, wenn langer Such-String und/oder $|\Sigma_P| << |\Sigma_T|$

2.10.8 Lazy-Initialization

- Idee: Implementierung der Menge als boolsches Array → einfügen, zugreifen und löschen in O(1)
- Probleme: Speicherbedarf und vor allem vollständige Initialisierung des Arrays,
  da sonst Zufallswerte enthalten sein können
- → Lazy Initialization → keine komplette Initialisierung des Arrays, aber trotzdem die Gewissheit haben, dass der Wert an der gefragten Stelle richtig ist
- → großer Speicherbedarf, da zusätzlich zum boolschen Array (B) noch zwei Integer-Arrays (P und Q) existieren
  Dabei zeigt P[k]=i an, dass B[k]=t/f im i-ten Schritt initialisiert worden ist und
  Q[k]=i zeigt an, dass P[k]=i gültig ist (Bestätigung)
- So wird nahezu garantiert, dass B[k] den gewünschten und keinen zufälligen Wert enthlt
- Algorithmus:
  $Insert(k)$
  $if(p \leq P[k] \leq top\ und\ Q[P[k]] = k)$
  $B[k] = true;$
  $else$
  $top++;\ P[k] = top;\ Q[top] = k;\ B[k] = true;$

2.10.9 Hashing

- Grund für die Idee: große Wertebereiche sind meißt sehr dünn besetzt (z.B. Matr.-Nummern)
  ⇒ Abbildung des Wertebereichs W auf eine kleinere Indexmenge I mit Hilfe einer
  Abbildungsfunmktion, der sogenannten Hash-Funktion, F: W→ I , Schlüssel F(w)=i
- Schwierigkeiten bei der Wahl der Funktion, da Schlüssel möglichst gleichverteilt in I liegen sollen
  Kollision bei F(w)=F(w'), obwohl w≠w'
  gute Hash-Funktion: surjektiv, gute Streuung
- perfekte Hash-Funktion existiert immer, aber schwer zu finden (Aufwand O(nN)) mit $|W| = N$ und $|S| = n$
- Primzahlen zum Mudulo-rechnen sehr gut geeignet
  Problem: Arraygröße ≠ Primzahl :
  F(w)=(w·s+t) mod q mit q große Primzahl und
  G(w)=w mod q' mit q' Arraygröße
- **offenes Hashing**
  jeder Arrayeintrag ist Anker-Element einer Liste und alle Objekte mit gleichem Hashwert
  werden an die entsprechende Liste angehängt
  offen = dynamisch mehr Speicherplatz und geschlossen = kein Kollisionsproblem
- **geschlossenes Hashing**
  falls Kollision ⇒ Berechnung eines alternativen Schlüssels
  offen = Verwendung anderer Adressen und geschlossen = kein neuer Speicherplatz
- lineares Sondieren
  F(w,i)=(F(w)+i) mod m
  führt meißt zu Clusterbildung → schlecht wenn viele Objekte gleichen Hashwert haben
- quadratisches Sondieren
  F(w,i)=(F(w)+i^2) mod m
  bessere ('chaotischere') Streuung, am besten m Primzahl
- doppeltes Hashing
  F(w,i)=(F_1(w)+F_2(w)·i^2) mod m
  geringe Wahrscheinlichkeit, daß zwei Objekte auch bei 2 Funktionen gleich abbilden
- Problem: Löschen
  bei offenem Hashing kein Problem → siehe Listen
  bei geschlossenem Hashing muß der gelöschte Arrayeintrag als gelöscht markiert werden,
  damit beim Suchen eines anderen Objektes über diese Stelle hinaus weitergesucht wird

2.11 Graphen-Algorithmen

2.11.1 Breitensuche

- von einem Startknoten s ausgehend, werden nacheinander alle von s erreichbaren Knoten besucht
- durch die Breitensuche entsteht implizit oder explizit ein Breitensuch**baum**
- die Tiefe eines Knotens k im Breitensuchbaum ist gleich der Länge des
  kürzesten Pfades von s zu k
- Implementierung durch **Queue**

- Algorithmus:
  $BreadthFirst(G)$
  $\forall\ v\ \in\ V\ do$
  $depth[v] = \infty$
  $\forall\ v\ \in\ V\ do$
  $if(depth[v] == \infty)$
  $VisitBreadthFirst(G, v)$

  $VisitBreadthFirst(G, s)$
  $depth[s] = 0$
  $Q = \oslash\ and\ enqueue(Q, s)$
  $while(!empty(Q))$
  $u = dequeue(Q)$
  $\forall\ v\ \in\ Adj[u]\ do$
  $if(depth[v] = \infty)$
  $depth[v] = depth[u] + 1$
  $enqueue(Q, v)$
- mit der Breitensuche kann man auch die Zahl der Zusammenhangskomponenten berechnen, folglich auch, ob ein Graph zusammenhängend ist
- Aufwandsabschätzung O(V+E)

2.11.2 Tiefensuche

- von einem Startknoten s ausgehend, werden nacheinander alle von s erreichbaren Knoten besucht
- vom zuletzt besuchten Knoten werden zunchst die folgenden Knoten besucht, dann Backtracking
- in jedem Knoten werden sowohl Eintritts- als auch Austrittszeitpunkte gespeichert
- Algorithmus:
  $DepthFirst(G)$
  $\forall\ v\ \in\ V\ do$
  $pre[v] = 0$
  $post[v] = 0$
  $precounter = 0$
  $postcounter = 0$
  $\forall\ v\ \in\ V do$
  $if(pre[v] = 0) DepthFirstVisit(v)$

  $DepthFirstVisit(u)$
  $precounter + +$
  $pre[u] = precounter$
  $\forall\ v\ \in\ Adj[u]\ do$
  $if(pre[v] = 0)\ DepthFirstVisit(v)$
  $postcounter + +$
  $post[u] = postcounter$
- Aufwandsabschätzung O(V+E)

2.11.3 topologisches Sortieren

- Eine vollständige Ordnung heißt eine top. Sortierung des gerichteten, zyklenfreien Graphen G=(V,E), falls gilt: $(u, v) \in E\ \Rightarrow\ u \leq v$
- Implementierung durch modifizierte Tiefensuche
- Aufwandsabschätzung O(V+E)

2.11.4 minimal spannender Baum

- Gegeben sei ein zusammenhängender, gewichteter Graph G=(V,E) mit Gewichtsfunktion w: E$\rightarrow \mathbb{R}$. Gesucht ist ein Spannbaum T$\subseteq$E, der w(T)= $\sum_{e \in T} w(e)$ minimiert
- Idee: Greedy-Strategie
  füge nach und nach Kanten zu einer anfangs leeren Kantenmenge A hinzu unter Beachtung von:
  Invariante: A ist Teilmenge eines MST (minimal spannenden Baumes)

2.11.5 Prims-Algorithmus

- Algorithmus bestimmt zu einem Graphen einen minimal spannenden Baum
- Algorithmus:
  (1) Starte mit beliebigem Knoten und füge alle von ihm ausgehenden Kanten hinzu
  (2) Gehe über die leichteste Kante zum nächsten Knoten und fahre bei (1) fort mit:
  es dürfen keine Zyklen entstehen und lösche 'alte' Kanten heraus, wenn es zu diesen Knoten hin eine leichtere gibt

$PrimAlg(G, w, r)$
$\quad \forall\ u\ in\ V\ do$
$\qquad key[u] = \infty\ and\ p[u] = null$
$\quad key[r] = 0$
$\quad Q = V$
$\quad while(Q \neq \emptyset)$
$\qquad u = extract - minimum(Q)$
$\qquad \forall\ v\ \in\ Adj[u]$
$\qquad\quad if(v\ \in\ Q\ and\ w(u, v) < key[v])$
$\qquad\qquad p[v] = uandkey[v] = w(u, v)$

- Implementierung durch binären Heap
- Aufwandsabschätzung: O(E$\cdot$logV)
  Aufwand mit Fibonacci-Heap: O(E+V$\cdot$logV)

2.11.6 Kruskals-Algorithmus

- Idee: In jedem Schritt wird die Kante mit dem kleinsten Gewicht gesucht, die zwei Bäume des Waldes verbindet. Diese Kante wird zur aktuellen Kantenmenge hinzugefügt.
- Algorithmus
  $KruskAlg(G, w)$
  $A = \oslash$
  $\forall\ v\ \in\ V$
  $makeSet(v)$
  $sort\ E\ into\ nondecreasing\ order$
  $\forall\ (u, v)\ \in\ E$
  $if(findSet(u) \neq findset(v))$
  $A = A \cup \{(u, v)\}$
  $union(u, v)$
- Aufwandsabschätzung O(E·logV)

2.11.7 kürzeste Pfade

- das kürzeste Gewicht von Knoten u zu Knoten v ist definiert als
  δ(u,v)=min$\{w(p)\ :\ p\ ist\ ein\ Pfad\ von\ u\ nach\ v\}$
  wobei min$\oslash$=∞

2.11.8 Dijkstras Algorithmus

- (1) sei S eine leere Menge
  (2) wähle einen Knoten v$\in$V\S mit min. Oberschranke v.d.
  (3) füge v in S ein und relaxiere alle von v ausgehenden Kanten
  (4) gehe zu Schritt (2), falls V\S$\neq \oslash$
- Aufwandsabschätzung O(E·logV)

2.11.9 Floyd-Warshall

- Idee: dynamisches Programmieren
  Rekursion für Gewichte:
  Sei d_{ij}^k das Gewicht eines kürzesten Pfades von i nach j in P^k
  Insbesondere ist also d_{ij}^n das Gewicht eines kürzesten Pfades von i nach j in G
- Die Idee des Algorithmus ist es, im ersten Durchlauf nur die Pfade einzubeziehen, die aus einer Kante bestehen. Im nächsten Schritt dann diejenigen Pfade, die die Länge zwei besitzen und durch Knoten 1 gehen, usw.
- Aufwandsabschätzung O(V^3)

2.12 Zusammenfassung

Algorithmus	**Aufwand**
Boyer-Moore	O((n-m+1)m+|Σ|)
BreadthFirst-Search	O(V+E)
Bubble-Sort	O(n^2)
Bucket-Sort	O(n) erwartet
Counting-Sort	O(n+k) → O(n) falls k=O(n)
DepthFirst-Search	O(V+E)
Dijkstras Algortihmus	O(E·log(V))
Floyd-Warshall Algorithmus	O(V^3)
Heap-Sort	best: O(n) und average/worst: O(n·log(n))
Insertion-Sort	O(n^2)
Knuth-Morris-Pratt	O(n)
Kruskals-Algorithmus	O(E·log(V))
Median der Mediane	O(n)
Merge-Sort	O(n·log(n))
naives String-Matching	(n·m)
Partition-Select	average O(n), worst O(n^2), siehe Quick-Sort
Prims-Algorithmus	O(E·log(V)) bzw. O(E+V·log(V))
Quick-Sort	average: O(n·log(n)) und worst: O(n^2)
Rabin-Karp	worst O((n-m+1)m), average: O(n) falls q≥m
Radix-Sort	O(d(n+k)), average: O(n·log(n))
Selection-Sort	O(n^2)
Sort-Select	O(n·log(n))
topologisches Sortieren	O(V+E)