

### Abstrakte Datentypen:

$zB + : Z \times Z \rightarrow Z$

$+(0,b)=b$

$+(a+1,b) = +(a,b)+1$

$+(a-1,b) = +(a,b)-1$

### Listen:

**Get()** löscht Insert\* solange, bis er vor Insert steht

**Next(), Insert(), Delete()** schiebt sich solange durch die Insert\*, bis es vor Insert steht

Bei einfach und doppeltverketteten Listen an das del / insert denken, Anfang mit Anchor/Sentinel

### Warteschlangen:

FIFO, hinten auslesen / löschen, vorne einfügen oder umgekehrt, Enq, Deq, Get

### Stack:

LIFO, auslesen, entfernen, einfügen am Anfang oder umgekehrt, Pop, Push, Top

### Baum:

**Tiefe** eines Knotens: Länge des Pfades zur Wurzel

**Höhe** eines Baumes: Tiefe des tiefsten Knotens (zB Baum hat nur Wurzel und 1 Sohn: Höhe=1)

**Grad** eines Knotens: Anzahl der Söhne

**Balanciert:** Für jeden inneren Knoten: Seine Höhe darf max 2 größer sein, als die Höhe der Söhne

**Vollständig:** Für jeden inneren Knoten: Seine Höhe -1 ist gleich der Höhe aller seiner Söhne ggf dürfen Blätter fehlen

**Darstellung** mit Graph, Klammerung, Mengen, Inhaltsverzeichnis

**Baum im Array:** Knoten A[i] hat Nachfolger in A[2i] und A[2i+1]

**Tiefensuche:** bis zu den Blättern runtergehen und diese nacheinander ausgeben

**Breitensuche:** erst Elemente der Höhe 1, dann Höhe 2,... ausgeben

### Prioritätsschlangen:

Enq\*( ) fügt Element bzgl seiner Priorität ein

### Heap:

Jeder Vaterknoten hat höhere Priorität als jeder Knoten seiner Teilbäume, Lösung als Array

### Graphen:

**(stark) zusammenhängend:** Es ex von  $v_i$  zu  $v_j$  ein Pfad oder (und) umgekehrt

**vollständig:** Zwischen je 2 Knoten ex eine Kante,  $|Kanten| = n(n-1)/2$

**isomorph:** 2 Graphen sind ineinander überführbar

**planar:** klar

$O(f) : g(n) \leq c \cdot f(n)$

$g \in O(f)$  sagt, dass f eine obere Schranke für g ist

$\Omega(f) : g(n) \geq c \cdot f(x)$

$g \in \Omega(f)$  sagt, dass f eine untere Schranke für g ist

$\Theta(f) : f(n)/c \leq g(n) \leq c \cdot f(n)$

$g \in \Theta(f)$  sagt, dass g genau so wächst, wie f

$\lim_{n \rightarrow \infty} g(n) / f(n)$

g wächst langsamer:  $g \in O(f) \setminus \Theta(f)$

g und f wachsen gleich:  $g \in \Theta(f)$

g wächst schneller:  $g \notin O(f)$

$$\bullet T(n) = \begin{cases} c & \dots n = 1 \\ axT(n/b)+c \cdot n & \dots n > 1 \end{cases}$$

$$\bullet T(n) = \begin{cases} O(n) & \dots a < b \\ O(n \times \log n) & \dots a = b \\ O(n^{\log_b a}) & \dots a > b \end{cases}$$

$$\bullet T(n) = \begin{cases} c & \dots n = 1 \\ axT(n/b)+O(n^p) & \dots n > 1 \end{cases}$$

$$\bullet T(n) = \begin{cases} O(n^p) & \dots a < b^p \\ O(n^p \times \log n) & \dots a = b^p \\ O(n^{\log_b a}) & \dots a > b^p \end{cases}$$

### Master Theorem: 2.1, F82/83

Bestimmung der Laufzeitklasse für rekursive Funktionen →

### Top-Down:

Problem in Sub-Probleme, diese in Sub-Sub-Probleme...

### Bottom-Up:

Teilprobleme lösen und zu einer Gesamtlösung kombinieren

**Divide & Conquer:**

Problem rekursiv in  $k$  Teilprobleme der Größe  $n/k$  zerlegen, lösen und zusammenfügen

**Dynamisches Programmieren:**

Reduziere Problem auf kleinere Teilprobleme, diese lösen und in Tabelle speichern. Dann Teillösungen zu Gesamtlösung kombinieren

**Memoization:**

Dyn.Prog. mit speichern der Zwischenergebnisse in Tabelle, lohnt sich wenn Neuberechnung teurer ist

**Sortieren:**

Direkt: Umkopieren, Indirekt: Permutationstabelle

**Stabilität:**

Reihenfolge von Objekten mit gleichem Schlüssel bleibt erhalten

**Selection-Sort:**

Kleinstes Element wird gesucht und mit dem ersten der unsortierten Teilfolge vertauscht  $O(n^2)$

**Bubble-Sort:**

Jedes Element wird mit seinem Nachfolger verglichen und evtl vertauscht.  $O(n^2/2)$

**Insertion-Sort:**

$A[i]$  wird mit der schon sortierten Teilfolge verglichen und richtig einsortiert  $O(n) O(n^2/4) O(n^2/2)$

**Quick-Sort:**

Pivot-Element, links alle kleiner, rechts alle größer, dann neues Pivot, bis  $P\_P\_...$ , nicht stabil  $O(n \log n)$ , worst:  $O(n^2)$

**Merge-Sort:**

Zerteilung in Hälften und Vergleich auf der kleinstmöglichen (sortierten) Ebene  $O(n \log n)$  aber stabil

**Heap-Sort:**

Elemente in einen Heap, erstes Element raus & speichern, dann heap wiederherstellen... Alle Elemente gleich:  $O(n)$  sonst  $O(n \log n)$

**Optimaler Suchalgorithmus:**  $O(n \log n)$

**Counting-Sort:**

Vorkommen zählen und in  $C$  speichern, dann  $C$  aufsummieren und  $B[C[A[i]]]=A[i]$ ,  $C[A[i]]--$ ,  $i--$   $O(k+n)$

**Radix-Sort:**

$b$ =Basis,  $d$ =Stellen,  $b^d$  Zahlen können sortiert werden, stellenweise vrnI mit Counting-Sort, dh  $d$  mal sortieren,  $O(d(k+n))$

**Bucket-Sort:**

Verteilung der  $n$  El. auf  $n$  Buckets, sortieren innerhalb der  $B$ 's, dann aneinanderhängen,  $O(n)$  erwartet

**k-Selektion:**

erst sortieren  $O(n \log n)$ , dann  $k$ -tes Element  $((n+1)/2)$  auf- und abger. =  $o+u$  Median)

**Selektion durch Partitionierung:**

Quicksort bis Pivot in der Mitte, wenn  $i < k$ , dann rechte Seite mit  $k-i$ ,  $i > k$  dann links mit  $k$  ( $k$ -tes gesucht)

**Median der Mediane:**

$\geq 3\lfloor n/10 \rfloor$  Elemente kleiner  $m$ ,  $\leq n - 3\lfloor n/10 \rfloor$  Elemente größer  $m$ . Aufteilen in  $S_{\leq}$ ,  $S_{=}$ ,  $S_{\geq}$ , wenn  $k = |S_{\leq}| + |S_{=}|$ , dann return  $m$ , sonst mit der passenden Menge eine Rekursion

**Rabin-Karp:**

String-Suche mit Horner und mit modulo

**Knuth-Morris-Pratt:**

Wird ein Teilstring verworfen, so fange nicht zwingend direkt beim nächsten an, sondern erst da, wo es passt. Abhängig vom Suchstring (Widerholungen im Suchstring)  $O(n)$

Prefix-Funktion: Gibt an, um wieviele Stellen der Suchstring verschoben werden muss  $O(m) \rightarrow O(n+m)$

**Boyer-Moore:**

SkipListe: m-letztes Vorkommen von a, testen auf Gleichheit von re nach li, wenn falsch, skip nach rechts um  $\max[1, \text{skip}() - (m-j)]$   $O(n \times m)$

**Menge A in Menge B enthalten?**

Nur kleiner Wertebereich: Menge B durchlaufen,  $C[B[i]] = \text{true}$ , dann A durchlaufen, wenn  $C[A[i]] = \text{true}$  dann  $i++$ , sonst return false  $O(n+m)$

**Lazy-Initialisation:**

$B[]$  T/F,  $P[]$  fürs top-Element,  $Q[]$  eingefügter Wert,  $P[i] < \text{top} \ \&\& \ Q[P[i]] = i$

**Hash-Funktion:**

Große Ausgangsmenge, kleine Zielmenge, möglichst wenige Kollisionen

**Offenes Hashing:**

Tabelle mit Anchor-Elementen von Listen, dynamisch, keine zusätzlichen Speicherplätze

**Geschlossenes Hashing:**

Bei Kollision zusätzlicher Index, kein neuer Speicherplatz, es werden neue Adressen verwendet

**Kollisionsauflösung:**

**lineares sondieren:**  $F(w,i) = (F(w)+i) \bmod m \leftarrow$  nicht so gut

**quadratisches sondieren:**  $F(w,i) = (F(w)+i^2) \bmod m \leftarrow$  mit m als Primzahl besseres Streuverhalten

**doppeltes sondieren:**  $F(w,i) = (F1(w) + F2(w) * i) \bmod m \leftarrow$  fast ideales Streuverhalten

**Binäre Suchbäume:**

**Search:** Baum durchlaufen, wenn x kleiner nach links, wenn x größer nach rechts, sonst x ausgeben

**Min/Max:** Baum solange nach links/rechts durchlaufen, bis das nächste Element NIL ist

**Successor:** kleinstes rechtes Element, falls nicht existent, erster Vorgänger, der größer als sein Sohn ist

**Insert:** laufe Baum runter, dann Pointer auf Vater und auf eigene Söhne, Pointer von Vater auf sich

**Delete:** Blatt: löschen, 1 Sohn: Sohn an seine Stelle, 2 Söhne: Successor suchen und damit ersetzen  
Alle  $O(\log n)$ , worst  $O(n)$

**Optimale Suchbäume:**

Knoten mit der höchsten Zugriffswahrscheinlichkeit liegen oben im Baum

**Ballancierter Baum:**

Höhe und Anzahl der Knoten im rechten & linken TB jedes Knotens unterscheiden sich um max 1

**AVL-Bäume:**

Normale binäre Suchbäume, die die Balance durch Rotation nach insert, del wiederherstellen. Für jeden Knoten gilt: Höhe des linken und rechten Teilbaums unterscheiden sich max um 1. Balance-Angaben

## RS-Bäume:

### INSERT:

1. Onkel ist rot: Vater, Opa und Onkel umfärben, weiter mit Opa
2. Onkel schwarz, x ist rechter Sohn: x auf Vater, um neues x nach links rotieren
3. Onkel schwarz, x ist linker Sohn: Vater schwärzen, Opa röten, nach rechts um Opa rotieren

### DELETE:

-normaler Baum:

1. Knoten ist ganz unten → einfach löschen
2. Knoten hat 1 Sohn → Knoten löschen, Sohn an dessen Stelle
3. Knoten hat 2 Söhne → Nachfolger von Knoten an dessen Stelle (Nachfolger löschen mit 1/2)

-RS-Baum:

Knoten rot: nichts passiert

Knoten schwarz: schwarze Marke liegen lassen.

→ Marke schwarz und neuer Knoten rot: verschmelzen zu schwarzem Knoten

→ Marke schwarz und neuer Knoten schwarz:

1. Bruder rot: nach links rotieren, Bruder neue Wurzel, ex-Bruder und Vater umfärben
2. Bruder und dessen Söhne schwarz: Bruder rot, Marke zum Vater
3. Bruder schwarz, dessen linker Sohn rot, rechter schwarz: linken Sohn nach rechts rotieren und ihn und seinen Ex-Vater umfärben
4. Bruder ist schwarz, dessen rechter Sohn rot: Vaters Farbe auf Bruder, Marke auf Vaters Farbe, roten Sohn des Bruders schwärzen und dann Rotation des kompletten TB nach rechts (Bruder neue Wurzel)

## B-Bäume:

Innere Knoten haben eine variable Anzahl von Nachfolgern (zw  $t$  und  $2t$ ). Jeder innere Knoten hat  $n$  Schlüssel und  $n+1$  Verweise auf Kinder. Reihenfolge beachten. `isLeaf()` gibbet auch noch.

$$h \leq \log_t((n+1)/2) \quad n \geq (2 * t^h) - 1$$

**Insert:** auf dem Hinweg werden alle Knoten geteilt, die zu groß sind, mittleres Element kommt zum Vater.

**Delete:** mit steal, merge, rotate

## Anwendung B-Bäume auf RS-Bäume:

nur RSR oder RS bzw SR, keine SS zusammen

## Union-Find:

Wald, Bäume, Knoten mit Pointer auf Vater, Find gibt root, Union fügt einen Baum zu einem anderen hinzu (achte auf Höhe der Bäume)

**Path Compression:** Find wird mit  $x$  ausgeführt,  $x$  setzt seinen Pointer auf Wurzel statt auf den Vater

**Einfacher Pfad:** alle Knoten verschieden

**Hamiltonkreis:** Jeden Knoten genau einmal

**Eulerkreis:** Jede Kante genau einmal

**Zusammenhängender Graph:** Jedes Knotenpaar durch Pfad verbunden

Graph mit  $k$  ZH-Komponenten enthält Kreis, wenn  $E > V - k$

## Adjazenz-Liste:

Knoten  $i$  ist mit Knoten  $a, b, c$  verbunden, Knoten  $j$  ist ... Achtung bei gerichteten Graphen!  $O(V+E)$

## Adjazenz-Matrix:

$A(i,j)=1$  wenn  $i$  mit  $j$  verbunden, sonst 0,  $O(V^2)$

## Breiten-Suche:

Startknoten bekommt 0, Nachbarknoten bekommen 1, davon Nachbarknoten bekommen 2, wenn nicht schon besucht...  $O(V+E)$

## Tiefensuche:

Erst einen Pfad durchgehen, dabei  $x/?$  als Eingangszähler speichern, dann zurück und  $x/y$  als Ausgangszähler, dann rekursiv weiter mit dem letzten Knoten, der noch Nachbarn hat.  $O(V+E)$

**Topologisches Sortieren:**

Tiefensuche in einem Graphen, nachher sortieren anhand der 2ten Zahl (rückwärts)  
Andere Möglichkeit: Suche Knoten ohne Vorgänger, aufschreiben, streichen, nächster...

**Generischer Algorithmus:**

[FEHLT HIER NOCH]

**Prims Algorithmus:**

Startpunkt, jedem Nachbarknoten eines besuchten Knotens wird der Wert der minimalen Kante zw  
besuchten und diesem Nachbarknoten zugeordnet  
mit binärem Heap:  $O(E \times \log V)$   
mit Fibonacci-Heap:  $O(E + V \times \log V)$

**Kruskals Algorithmus:**

Es wird jeweils die minimalste Kante genommen, die noch übrig ist, und die 2 Bäume verbindet  
 $O(E \times \log V)$

**Kürzeste Pfade:**

Teilpfade von kürzesten Pfaden sind wieder kürzeste Pfade, jeder Knoten: Zeiger auf Vorgänger und  
Speicher für den aktuell kürzesten Weg zu ihm

**Dijkstra:**

Startknoten, gib jedem Nachbarknoten den Wert  $d[\text{besuchter Knoten}] + \text{Wegkosten}$ , wenn ein unbesuchter  
Knoten günstiger zu erreichen ist, setze  $d[\text{unbesuchter Knoten}]$  runter  $O(E \times \log V)$

**Floyd-Warshall:**

2 Matrizen, D und P, init mit  $\infty$ . In D stehen alle Wege, in P alle direkten Vorgänger auf diesen Wegen. Alle  $\infty$   
abarbeiten durch Wege verlängern, außer in P auf der Hauptdiagonalen  $= (V^3)$