1 Invariante

- Vorm ersten Durchlauf erfüllt sein (Initialisierung, Induktionsanfang)
- Muss bei jedem Schleifendurchlauf erhalten bleiben (Erhaltng, Induktionsschritt)
- Invar nach Beendigung der Schleife zeigt Korrektheit (Terminierung)

1.1 Rekursionsgleichungen

- ineinander einsetzen
- Summe erkennen und zusammengefasst aufschreiben

1.2 Mastertheorem

1.2.1 Additives Mastertheorem

- a, b, c *positiv* $n = b^k$
- $T(n) \leq \begin{cases} c & n = 1 \\ a \cdot T(\frac{n}{b}) + c & n > 1 \end{cases}$

$$\begin{array}{lll} T(n) \leq c \cdot \frac{a}{a-1} n^{log_b(a)} - \frac{c}{a-1} & = O(n^{log_b(a)}) & falls\ a > 1 \\ T(n) \leq c \cdot \frac{a}{a-1} n - \frac{c}{a-1} & = O(n) & falls\ a = b > 1 \\ T(n) \leq c \cdot log_b(n) + c & = O(log(n)) & falls\ a = 1 \end{array}$$

1.2.2 allgemeines Mastertheorem

- $a, b, d, q \geq 1$
- $T(n) \leq \begin{cases} d & n \leq q \\ a \cdot T(\frac{n}{b}) + f(n) & n > q \end{cases}$

$$\begin{array}{ll} f(n) = O(n^{log_b(a)-\epsilon})\ mit\ \epsilon > 0 & T(n) = O(n^{log_b(a)}) \\ f(n) = \Theta(n^{log_b(a)}) & T(n) = O(n^{log_b(a)} log(n)) \\ f(n) = \Omega(n^{log_b(a)+\epsilon})\ mit\ \epsilon > 0, & T(n) = \Theta(f(n)) \\ a \cdot f(\frac{n}{b}) \leq \delta \cdot f(n), \delta < 1, n \to \infty & \end{array}$$

2 O-Notation

2.1 Definition

- $f = O(g) \Leftrightarrow \exists c > 0 \exists n_0 \in N\ \forall n \geq n_0 : f(n) \leq cg(n)$
  (f wächst asymptotisch höchstens so schnell wie g)
- $f = \Omega(g) \Leftrightarrow g = O(f)$
  (f wachst asymptotisch mindestens so schnell wie g)

- f $= \Theta(g) \Leftrightarrow f = O(g)\ und\ g = O(f)$
  (f und g wachsen asymptotisch gleich schnell)
- $f = o(g) \Leftrightarrow \forall c > 0\ \exists n_0 \in N \forall n \geq n_0 : f(n) < cg(n)$
  $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
  (f wächst asymptotisch langsamer als g)
- $f = \omega(g) \Leftrightarrow g = o(f)$
  (f wächst asymptotisch langsamer als g)

2.2 Eigenschaften

- $f = o(g) \Rightarrow f = O(g)$
- $f = o(g)\ und\ h = o(g) \Rightarrow f + h = o(g)(auch O, \Omega, \omega, \Theta)$
- $f = o(g)\ und\ h = o(g) \Rightarrow f \cdot h = o(g^2)(auch O, \Omega, \omega, \Theta)$

2.3 Reihenfolge

$c < log(n) < n^{\frac{1}{k}} < n < nlog(n) < n^2 < n^k < 2^n$

3 Sortieralgorithmen

3.1 Bubblesort

- Jeden Durchlauf wird das größte Element auf die n-ite Stelle getauscht. Jeder Vergleich ggf ein Swap.
- inkrementelle, inplace, stabil
- $BC : \Theta(n)\ AV : \Theta(n^2)\ WC : \Theta(n^2)$

3.2 Insertionsort

- Key wird in sortiertes Array eingeordnet. Key wird gemerkt, falls kleiner wird das gößere Element auf Pos von Key kopiert aber Key wird erst kopiert, wenn die richtige Stelle gefunden worden ist.
- inkrementelle, inplace, stabil
- $BC : \Theta(n)\ AV : \Theta(n^2)\ WC : \Theta(n^2)$

3.3 Mergesort

- Teile Array bis auf ein Element Array und sortiere beim rekursiven zusammenfügen. Geteilt wird p bis q, q+1 bis r.
  Merge vergleicht erste Pos von den Arrays und fügt immer das kleinere ins Zielarray ein.
- D&C,
- $BC : \Theta(nlog(n))$ $AV : \Theta(nlog(n))$ $WC : \Theta(nlog(n))$

3.4 Quicksort(A,p,r)

- Teile Array nach Pivotelement und füge Pivot dann an Grenze ein. Dann partition bis q-1 und
- D&C,
- $BC : \Theta(nlog(n))$ $AV : \Theta(nlog(n))$ $WC : \Theta(n^2)$

4 Heap

4.1 Definition

- $Heap := A[i] \geq A[2i] und A[i] \geq A[2i+1]$
- jeder Knoten hat mindestens so großen Wert wie seine Kinder
- Wird als binärer Baum dargestellt.

4.2 Eigenschafen

- Baumtiefe $\lfloor log(n) \rfloor$
- $A[\lfloor \frac{n}{2} \rfloor + 1]$ bis $A[n]$ sind Kinder
- Maximum ist die Wurzel A[1]

4.3 Heapify

- Tausche mit größtem Kind
- wird auf getauschten Knoten erneut aufgerufen bis er richtit steht

4.4 Build-Heap

- Bei uns meist MaxHeap
- Heapify auf $A[\lfloor \frac{n}{2} \rfloor]$ *downto* $A[1]$
- $BC : \Theta(n)$ $AV : \Theta(n)$ $WC : \Theta(n)$
- #Vertauschungen $\leq \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} (log(n) - log(i))$
- #Vergleiche $\leq$ #Vertauschungen

4.5 Heap-Sort

- Build-Heap
- Sort-Heap
  $VertauscheA[1]undA[i]$
  Danach Heapify(A[i ... i-1])
- $BC : \Theta(nlog(n))$ $AV : \Theta(nlog(n))$ $WC : \Theta(nlog(n))$

4.6 Bucket-Sort

- Hänge a_j an Liste L[a_j] an. Gebe alle Listen aus
- stabil
- Zahlen aus {1,...,m} und O(n+m) und braucht O(n+m) Platz

5 Dynamische Arrays

- Falls Array A nicht mehr ausreicht (n¿w). Verdoppel Array
- Falls $\frac{1}{4} and n > 0$ Halbiere Array

6 Stack

- Empty(S), Pop(S), Push(S), Top(S)(Pos)

7 Queue

- Enqueue,Dequeue, Head(Q), Tail(Q)(erste freie Pos)
- falls Array vonn starte bei 1, falls frei

8 Doppelt verkettete Listen

- Head(L), Insert(L,x)(hängt vorne dran), Remove(L,x)
- key(x), next(x), prev(x)
- falls next(x) = nil, x letztes Element

9 Skiplisten

- verschiedene Niveaus
- perfekte hat $\lceil log(n) \rceil$ Niveaus (ermöglicht binäre Suche)
- left(v), right(v), down(v), up(v), Search(L,x), Insert(L,v) (mit RandomHight)
- Niveau 0, wo jeder Knoten ist (2^0)
- Niveau k behinaltet jeden 2^k-ten Knoten

10 Binäre Suchbäume

10.1 Binäre Suchbäume

- lc[x], rc[x], p[x],root[x]
- Inorder-Tree-Walk(x) gebe Knoten sortiert, nach abgeben, aus.
  Gehe am Ende von ganz rechts zur Wurzel zurück. $\Theta(n)$
- Baumsuche(x,k) meist mit x = root[T]
- Min/Max linkestes/rechtestes Element
- Nachfolger(x) linkestes Element im rechten Teilbaum. Wenn nicht verfügbar im Baum aufsteigen
- Delete(x), x hat 2 Kinder, Nachfolger von x wird verschoben, ggf wiederholen

10.2 Balancierte Suchbäume

- Höhe höchstens $2log(n+1) - 2$
- Rotation ändern nur Höhe
- Rechtsrota(T,x),Linksrota(T,x)
- Balance(t):
  Falls lc[t] und rc[lc[t]] größer dann Linksrota(lc[t]), sonst Rechtsrota(t)
  Falls rc[t] und lc[rc[t]] größer dann Rechtsrota(rc[t]), sonst Linksrota(t)

11 Hashing

11.1 Geschlossene Adressierung

- Kollisionsuaflösung durch Listen
- Suchen/Löschen AV: $\Theta(1+\alpha)$, Falls m= $\Theta(n)$.$\Theta(1)$

11.2 Offene Adressierung

- (nächste) freie Stelle in der Hashtabelle
- Lineares Hashing: $h(k,i) = (h'(k) + i) mod\ m$
- Quadratisches Hashing: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) mod\ m$
- Delete problematisch, deshalb offene Adressierung bei Anwendungen ohne Delete nutzen
- falls zu viele deleted, dann neu hashen in größerer Tabelle(amortisierte Laufzeit)

11.3 Kuckuckshashing

- 2 Hashfunktionen mit eine Tabelle
- Insert: Falls belegt, neuen Wert einfügen und alten mit anderer Funktion neu hashen
- max $dlog(n)$ Hashversuche

12 Graphentheorie

12.1 Adjazenzmatrix/liste

- Zeile: von, Spalte: nach
- Zeile: von, Einträge: erreichende Knoten (einfach verkettete liste)

12.2 SSSP

12.2.1 BFS

- berechnet Abstand von allen Knoten zu s in einem ungewichteten Graphen
- d[u] ist der Abstand am Anfang ∞
- π[u] ist der Vorgänger am Anfang NIL
- Color[u] = weiß, grau,schwarz am Anfang weiß

- Queue zum Speichern der grauen Knoten
- BFS entdeckt alle Knoten $v \in V$, die von s aus erreichbar sind
- entdeckt die Zusammenhangskomponenten
- $O(|V| + |E|)$

12.2.2 DFS

- Neue Konten vom zuletzt gefundenen Knoten entdeckt. Fals alle adj[v] entdeckt gehe zu π[v]
- löst nicht SSSP, bildet Spannbaum
- d[u] ist der Abstand am Anfang ∞
- π[u] ist der Vorgänger am Anfang NIL
- Color[u] = weiß, grau,schwarz am Anfang weiß
- Stack zum Speichern der grauen Knoten
- Baumkante: rot, Rückkante: grün (auf grauen), Sonstigekante: blau (auf schwarzen)
- G enthält einen Kreis $\Leftrightarrow$ DFS(G) erzeugt mindestens eine Rückwärtskante
- DAG $\Leftrightarrow$ es existiert eine topologische Sortierung (alle Kanten einzeichnen)
- TopoligischesSorrtieren(G) legt eine Liste an und fügt die schwarzen Knoten vorne an

12.3 Dijkstra

- kein negativer Kreis aber gewichtet
- setze alle Knoten auf ∞ und update alle adj(u) mit dem Gewicht, dalls geringer als aktueller Wert
- Laufzeit mit Heaps:$O((|V| + |E|)log(|V|))$

13 Dynamische Programmierung

- Berechne die Funktionswerte iterativ und bottom-up und nutze Teilergebnisse

13.1 Bellman-Ford

- ohne negative Zyklen erhält man kürzesten Pfad
- Laufzeit $O(|V|^2 + |V||E|)$ und Speicherbedarf $O(|V|^2)$

13.2 APSP

13.2.1 Floyd-Warshall(W,n)

- Keine negative Zyklen
- Initialisiere Adjazenzmatrix mit 1 Nachbarknoten
- halte k mal Spalte K und Zeit K fest und guck, ob geringere Summen entstehen
- Laufzeit $O(|V|^3)$ mit Platz $O(|V|^2)$

14 Greedy

14.1 Allgemein

- das zum Zeitpunkte beste auswählen, was man machen kann
- EDF für Scheduling Probleme

14.2 Kruskal

- gewichteter, ungerichteter, zusammenhängender Graph errechnet daraus einen MST
- Nimm Kante mit geringstem Gewicht, die zwei Bäume im aktuellen aufspannenden Wald verbindet und füge diese zu A hinzu
- rote Kante: Spannbaum, grüne Kante: ungebraucht

14.3 Union-Find - SIEHE üBUNG

- Jede Menge ist eine lineare Liste, die durch das erste Element repräsentiert wird
- doppelt verkettete Liste mit einem weiteren Zeiger auf den Kopf
- hänge kürzere Liste an längere Liste

14.4 Prim

- Lässt einen Wald wachsen. Immer nur von bekannten Knoten aus auswählen
- Laufzeit $O(|E|log(|V|)$

14.5

-

15 Rechentricks

- Arithmetische Reihe $\sum_{i=1}^{n} k = \frac{n(n+1)}{2}$
- logarithmus!!! bsp laufzeit aus probe 1
- Stirling'sche Formel $n! \approx (\frac{n}{e})^n \cdot \sqrt{2\pi n}$
  $\Rightarrow log(n!) \approx nlog(n) - nlog(e)$
- $log((\frac{P}{Q})^k) = k \cdot log(\frac{P}{Q}) = k \cdot (log(P) - log(Q))$
- $log_a(P) = \frac{log_b(P)}{log_b(a)}$
- Union Find