

# Wiederholung

---

- Beschreibung von Algorithmen durch *Pseudocode*.
- Korrektheit von Algorithmen durch *Invarianten*.
- Laufzeitverhalten beschreiben durch *O-Notation*.
  - Wollen uns auf *asymptotische Laufzeit* konzentrieren.
  - Laufzeiten immer mit Hilfe von  $O$ -,  $\Omega$ -,  $\Theta$ -Notation angeben.

# Regeln für Kalküle

---

*Satz 2.10:* Sei  $f : \mathbb{N} \rightarrow \mathbb{R}^+$  mit  $f(n) \geq 1$  für alle  $n$ . Weiter sei  $k, l \geq 0$  mit  $k \geq l$ . Dann gilt

1.  $f(n)^l = O(f(n)^k)$ .
2.  $f(n)^k = \Omega(f(n)^l)$ .

*Satz 2.11:* Seien  $\varepsilon, k > 0$  beliebig. Dann gilt

1.  $\log(n)^k = O(n^\varepsilon)$ .
2.  $n^\varepsilon = \Omega(\log(n)^k)$ .

# 3. Inkrementelle Algorithmen

---

*Definition 3.1:* Bei einem inkrementellen Algorithmus wird sukzessive die Teillösung für die ersten  $i$  Objekte aus der bereits bekannten Teillösung für die ersten  $i-1$  Objekte berechnet,  $i=1, \dots, n$ .

*Beispiel Min-Search:*

- Objekte sind Einträge des Eingabearrays.
- Teilproblem bestehend aus ersten  $i$  Objekten bedeutet Minimum im Teilarray  $A[1..i]$  bestimmen.

# Sortieren und inkrementelle Algorithmen

---

Eingabe bei Sortieren : Folge von  $n$  Zahlen  $(a_1, a_2, \dots, a_n)$ .

Ausgabe bei Sortieren : Umordnung  $(b_1, b_2, \dots, b_n)$  der Eingabefolge, so dass  $b_1 \leq b_2 \leq \dots \leq b_n$ .

Sortieralgorithmus : Verfahren, das zu jeder Folge  $(a_1, a_2, \dots, a_n)$  sortierte Umordnung  $(b_1, b_2, \dots, b_n)$  berechnet.

Eingabe : (31,41,59,26,51,48)

Ausgabe : (26,31,41,48,51,59)

*Idee:* Sukzessive wird eine Sortierung der Teilarrays  $A[1..i]$ ,  $1 \leq i \leq \text{length}(A)$  berechnet.

# Insertion Sort

InsertionSort(Array A)

1. **for**  $j \leftarrow 2$  **to**  $\text{length}(A)$  **do**
2.      $\text{key} \leftarrow A[j]$
3.      $i \leftarrow j-1$
4.     **while**  $i > 0$  and  $A[i] > \text{key}$  **do**
5.          $A[i+1] \leftarrow A[i]$
6.          $i \leftarrow i-1$
7.      $A[i+1] \leftarrow \text{key}$

- Eingabegröße  $n$
- $(\text{length}(A)=n)$
- verschiebe alle
- $A[1, \dots, j-1]$ , die größer als
- $\text{key}$  sind eine Stelle
- nach rechts
- Speichere  $\text{key}$  in „Lücke“

# Invariante bei Insertion-Sort

---

**Invariante:** Vor Durchlauf der for-Schleife (Zeilen 1-8) für Index  $j$  gilt, dass  $A[1..j-1]$  die  $j-1$  Eingabezahlen in sortierter Reihenfolge enthält.

**Initialisierung:**  $j=2$  und  $A[1]$  ist sortiert, da es nur eine Zahl enthält.

**Erhaltung:** while-Schleife (Zeilen 5-7) zusammen mit Zeile 8 sortiert  $A[j]$  korrekt ein.

**Terminierung:** Vor Durchlauf mit  $j=length(A)+1$  ist  $A[1..length(A)]$  sortiert.

# Insertion-Sort – Analyse (2)

---

*Satz 3.2.* Insertion-Sort besitzt Laufzeit  $\Theta(n^2)$ .

Zum Beweis wird gezeigt:

1. Es gibt ein  $c_2$ , so dass die Laufzeit von Insertion - Sort bei allen Eingaben der Größe  $n$  immer höchstens  $c_2 n^2$  ist.
2. Es gibt ein  $c_1$ , so dass für alle  $n$  eine Eingabe  $I_n$  der Größe  $n$  existiert bei der Insertion - Sort mindestens Laufzeit  $c_1 n^2$  besitzt.

# 4. Divide & Conquer – Merge-Sort

---

*Definition 4.1:* Divide&Conquer (Teile&Erobere) ist eine auf Rekursion beruhende Algorithmentechnik.

Eine Divide&Conquer-Algorithmus löst ein Problem in 3 Schritten:

- **Teile** ein Problem in mehrere Unterprobleme.
- **Erobere** jedes einzelne Unterproblem durch rekursive Lösung. Ausnahme sind kleine Unterprobleme, diese werden direkt gelöst.
- **Kombiniere** die Lösungen der Teilprobleme zu einer Gesamtlösung.

# Merge-Sort

---

Merge-Sort ist eine mögliche Umsetzung des Divide&Conquer-Prinzips auf das Sortierproblem.

Merge - Sort( $A, p, r$ )

```
1 if  $p < r$ 
2   then  $q \leftarrow \lfloor (p + r) / 2 \rfloor$ 
3       Merge - Sort( $A, p, q$ )
4       Merge - Sort( $A, q + 1, r$ )
5       Merge( $A, p, q, r$ )
```

- Merge ist Algorithmus zum Mischen zweier sortierter Teilfolgen
- Aufruf zu Beginn mit Merge-Sort( $A, 1, length[A]$ ).

# Laufzeit von D&C-Algorithmen

---

$T(n)$  := Gesamtlaufzeit bei Eingabegröße  $n$ .

$a$  := Anzahl der Teilprobleme durch Teilung.

$n/b$  := Größe der Teilprobleme.

$D(n)$  := Zeit für Teilungsschritt.

$C(n)$  := Zeit für Kombinationsschritt.

Für  $n \leq u$  wird Algorithmus mit Laufzeit  $\leq c$  benutzt.

Dann gilt:

$$T(n) \leq \begin{cases} c & \text{falls } n \leq u \\ aT(n/b) + D(n) + C(n) & \text{sonst} \end{cases}$$

# Laufzeit von Merge-Sort (1)

---

- $u = 1, a = 2, b \approx 2.$
- $D(n) = \Theta(1), C(n) = \Theta(n)$  (Lemma 4.3).
- Sei  $c$  so gewählt, dass eine Zahl in Zeit  $c$  sortiert werden kann und  $D(n) + C(n) \leq cn$  gilt.

*Lemma 4.4:* Für die Laufzeit  $T(n)$  von Merge-Sort gilt:

$$T(n) \leq \begin{cases} c & \text{falls } n \leq 1 \\ 2T(n/2) + cn & \text{sonst} \end{cases}$$

# Average-Case Laufzeit

---

## Average-case Laufzeit:

- Betrachten alle Permutationen der  $n$  Eingabezahlen.
- Berechnen für jede Permutation Laufzeit des Algorithmus bei dieser Permutation.
- Average-case Laufzeit ist dann der Durchschnitt über all diese Laufzeiten.
- Average-case Laufzeit ist die erwartete Laufzeit einer zufällig und gleichverteilt gewählten Permutation aus der Menge aller Permutationen der  $n$  Eingabezahlen.

# Quicksort - Idee

---

**Eingabe:** Ein zu sortierendes Teilarray  $A[p\dots r]$ .

**Teilungsschritt:** Berechne einen Index  $q, p \leq q \leq r$  und vertausche die Reihenfolge der Elemente in  $A[p\dots r]$ , so dass die Elemente in  $A[p\dots q-1]$  nicht größer und die Elemente in  $A[q+1\dots r]$  nicht kleiner sind als  $A[q]$ .

**Eroberungsschritt:** Sortiere rekursiv die beiden Teilarrays  $A[p\dots q-1]$  und  $A[q+1\dots r]$ .

**Kombinationsschritt:** Entfällt, da nach Eroberungsschritt das Array  $A[p\dots r]$  bereits sortiert ist.

# Quicksort – Analyse

---

*Satz 5.2:* Es gibt ein  $c > 0$ , so dass für alle  $n$  und alle Eingaben der Größe  $n$  Quicksort mindestens Laufzeit  $cn \log(n)$  besitzt.

*Satz 5.3:* Quicksort besitzt worst-case Laufzeit  $\Theta(n^2)$ .

*Satz 5.4:* Quicksort besitzt *average-case* Laufzeit  $O(n \log(n))$ .

**Average-case Laufzeit:** Betrachten alle Permutationen der  $n$  Eingabezahlen. Berechnen für jede Permutation Laufzeit von Quicksort bei dieser Permutation. Average-case Laufzeit ist dann der Durchschnitt über all diese Laufzeiten.

# Heaps

---

*Definition 8.1:* Ein Heap über einem Array  $A$  ist das Array  $A$  zusammen mit einem Parameter  $heap-size[A]$  und drei Funktionen

Parent, Left, Right :  $\{1, \dots, heap-size[A]\} \rightarrow \{1, \dots, length[A]\}$  .

Dabei gilt:

1.  $1 \leq heap - size[A] \leq length[A]$
2.  $Parent(i) = \lfloor i / 2 \rfloor$  für alle  $1 \leq i \leq heap - size[A]$
3.  $Left(i) = 2i$  für alle  $1 \leq i \leq heap - size[A]$
4.  $Right(i) = 2i + 1$  für alle  $1 \leq i \leq heap - size[A]$ .

Arrayelemente  $A[1], \dots, A[heap-size[A]]$  heissen Heapelemente.

# Heaps (2)

---

- Die Funktionen Parent, Left, Right versehen ein Array mit einer binären Baumstruktur.
- Nehmen dabei an, dass der Baum fast vollständig ist, d.h. der Baum ist bis auf die letzte Ebene auf jeder Ebene vollständig besetzt. Die letzte Ebene ist von links nach rechts besetzt.
- Parent liefert Elternknoten, Left und Right liefern linkes und rechtes Kind eines Knotens.
- Liegt dabei ein Funktionswert außerhalb des Intervalls  $[1, \dots, \text{heap} - \text{size}[A]]$ , so bedeutet dies, dass der Knoten kein Eltern oder kein linkes bzw. rechtes Kind besitzt.
- Es gibt nur einen Knoten ohne Eltern, dies ist die Wurzel  $A[1]$ .

# Heaps

---

*Definition 8.2:* Ein Heap heißt

1. max-Heap, wenn für alle  $1 \leq i \leq \text{heap} - \text{size}[A]$  gilt

$$A[\text{Parent}(i)] \geq A[i].$$

2. min-Heap, wenn für alle  $1 \leq i \leq \text{heap} - \text{size}[A]$  gilt

$$A[\text{Parent}(i)] \leq A[i].$$

Die Eigenschaften in 1. und 2. werden max-Heap- bzw. min-Heap-Eigenschaft genannt.

# Algorithmen auf Heaps

---

- **Max-Heapify** wird benutzt, um die Max-Heap Eigenschaft aufrecht zu erhalten.
- **Build-Max-Heap** konstruiert aus einem unstrukturierten Array einen max-Heap.
- **Heapsort** sortiert mit Hilfe von Heaps.

Mit Heaps können *Prioritätswarteschlangen* realisiert werden.

# Heapsort

---

Eingabe: Array  $A$

Ausgabe: Zahlen in  $A$  in aufsteigender Reihenfolge sortiert.

Heapsort( $A$ )

1. Build - Max - Heap( $A$ )
2. **for**  $i \leftarrow \text{length}[A]$  **downto** 2
3.     **do**  $A[1] \leftrightarrow A[i]$
4.          $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5.         Max - Heapify( $A, 1$ )

# Laufzeit von Heapsort

---

*Satz 8.9:* Heapsort besitzt Laufzeit  $O(n \log(n))$ .

Beweisskizze:

1. Aufruf von Build-Max-Heap:  $O(n)$ .
2. for-Schleife in Zeilen 2-5  $(n-1)$ -mal durchlaufen.
3. Pro Durchlauf Laufzeit  $O(\log(n))$  (Max-Heapify).

# Laufzeit von Sortieralgorithmen

---

		Laufzeit	
		worst-case	average-case
Algorithmus	Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$
	Merge-Sort	$\Theta(n \log(n))$	$\Theta(n \log(n))$
	Quick-sort	$\Theta(n^2)$	$\Theta(n \log(n))$
	Heap-sort	$\Theta(n \log(n))$	-

# Vergleichssortierer

---

**Definition 8.1:** Ein Vergleichssortierer ist ein Algorithmus, der zu jeder beliebigen Eingabefolge  $(a_1, a_2, \dots, a_n)$  von Zahlen eine Permutation  $\pi$  berechnet, so dass  $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$ . Dabei benutzt ein Vergleichssortierer außer den durch den Pseudocode definierten Kontrolloperationen nur die Vergleichsoperationen  $=, \neq, \leq, \geq, <, >$ .

## Bemerkungen:

1. Nehmen an, dass Eingabezahlen immer paarweise verschieden sind. Benötigen daher  $=$  nicht.
2. Können uns auf den Vergleich  $\leq$  einschränken. Andere Vergleichen sind hierzu äquivalent.

# Untere Schranke für Vergleichssortierer

---

*Lemma 8.3:* Für Eingaben der Größe  $n$  hat ein Entscheidungsbaum für einen Vergleichssortierer mindestens  $n!$  Blätter.

*Lemma 8.4:*  $\log(n!) = \Theta(n \log(n))$ .

*Satz 8.5:* Die von einem Vergleichssortierer bei Eingabegröße  $n$  benötigte Anzahl von Vergleichen ist  $\Omega(n \log(n))$ .

*Korollar 8.6:* Die von Merge-Sort und Heapsort benötigte Laufzeit von  $\Theta(n \log(n))$  ist asymptotisch optimal.

# Sortieren durch Abzählen (1)

---

**Annahme:** Es gibt ein dem Algorithmus Counting-Sort bekannter Parameter  $k$ , so dass für die Eingabefolge  $(a_1, \dots, a_n)$  gilt:

$$0 \leq a_i \leq k \text{ für alle } 1 \leq i \leq n.$$

**Algorithmusidee:**

1. Für alle  $i, 0 \leq i \leq k$  bestimme Anzahl  $C_i$  der  $a_j$  mit  $a_j \leq i$ .
2. Kopiere  $a_j$  mit  $a_j = i$  in Felder  $B[C_{i-1} + 1], \dots, B[C_i]$  eines Arrays  $B$  mit  $length[B]=n$ . Dabei gilt  $C_{-1} = 0$ .

Es gilt:  $C_i - C_{i-1}$  ist die Anzahl der  $a_j$  mit  $a_j = i$ .

# Analyse von Counting-Sort (2)

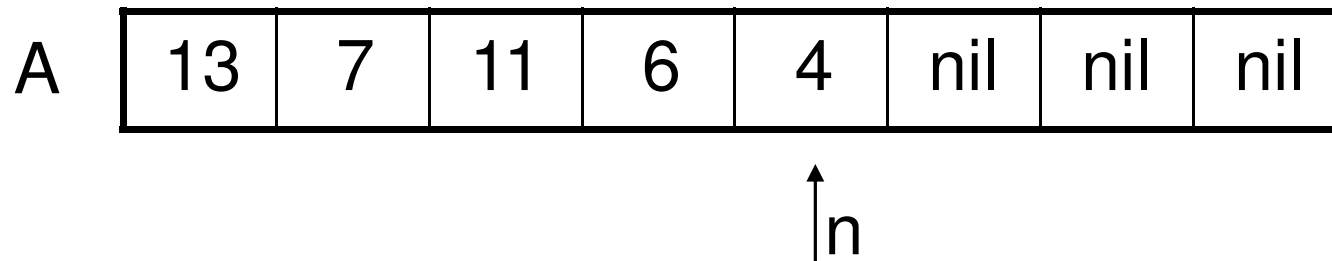
---

*Satz 10.1:* Counting-Sort besitzt Laufzeit  $\mathbf{O}(n+k)$ .

*Korollar 10.2:* Gilt  $k = \mathbf{O}(n)$ , so besitzt Counting-Sort Laufzeit  $\mathbf{O}(n)$ .

## Unsere erste Datenstruktur:

- Feld  $A[1, \dots, \text{max}]$
- Integer  $n$ ,  $1 \leq n \leq \text{max}$
- $n$  bezeichnet Anzahl Elemente in Datenstruktur



## Datenstruktur Feld:

- Platzbedarf  $\Theta(\max)$
- Laufzeit Suche:  $\Theta(n)$
- Laufzeit Einfügen/Löschen:  $\Theta(1)$

## Vorteile:

- Schnelles Einfügen und Löschen

## Nachteile:

- Speicherbedarf abhängig von max (nicht vorhersagbar)
- Hohe Laufzeit für Suche

# Stacks (Stapel) und Queues (Schlangen)

---

## *Definition 11.4:*

1. Stacks (Stapel) sind eine Datenstruktur, die die dynamische Menge LIFO (last-in-first-out) implementiert.  
In LIFOs sollen beliebige Objekte eingefügt und das zuletzt eingefügte Objekt entfernt werden können.
2. Queues (Schlangen) sind eine Datenstruktur, die die dynamische Menge FIFO (first-in-first-out) implementiert.  
In FIFOs sollen beliebige Objekte eingefügt und das am längsten in der Menge befindliche Objekt entfernt werden können.

# Stacks

---

- Einfügen eines Objekts wird bei Stacks **Push** genannt.
- Entfernen des zuletzt eingefügten Objekts wird **Pop** genannt.
- Zusätzliche Hilfsoperation ist Stack-Empty, die überprüft, ob ein Stack leer ist.
- Stack mit maximal  $n$  Objekten wird realisiert durch ein Array  $S[1\dots n]$  mit einem zusätzlichen Feld  $top[S]$ , das den Index des zuletzt eingefügten Objekts speichert.

# Queues

---

- Einfügen eines Objekts wird **Enqueue** genannt.
- Entfernen des am längsten in der Queue befindlichen Objekts wird **Dequeue** genannt.
- Queue mit maximal  $n-1$  Objekten wird realisiert durch ein Array  $Q[1\dots n]$  mit zusätzlichen Feldern  $head[Q]$ ,  $tail[Q]$ .
- $head[Q]$ : Position des am längsten in Queue befindlichen Objekts
- $tail[Q]$ : erste freie Position.
- Alle Indexberechnungen modulo  $n (+1)$ , betrachten Array kreisförmig. Auf Position  $n$  folgt wieder Position 1.

# Doppelt verkettete Listen (1)

---

- **Verkettete Listen** bestehen aus einer Menge linear angeordneter Objekte.
- Anordnung realisiert durch Verweise.
- Unterstützen alle dynamischen Mengen mit den Operationen **Insert, Delete, Search, Search-Minimum**, usw. Unterstützung ist nicht unbedingt effizient.
- Objekte in **doppelt verketteter Liste  $L$**  besitzen mindestens drei Felder: ***key, next, prev***. Außerdem Felder für Satellitendaten möglich.
- Zugriff auf Liste  $L$  durch Verweis/Zeiger ***head[L]***.

# Doppelt verkettete Listen - Beispiel

---

## Datenstruktur Liste:

- Platzbedarf  $\Theta(n)$
- Laufzeit Suche:  $\Theta(n)$
- Laufzeit Einfügen/Löschen:  $\Theta(1)$

## Vorteile:

- Schnelles Einfügen/Löschen
- $O(n)$  Speicherbedarf

## Nachteile:

- Hohe Laufzeit für Suche

# Binäre Bäume

---

- Neben Felder für Schlüssel und Satellitendaten  
Felder  $p$ ,  $left$ ,  $right$ .
- $x$  Knoten:  $p[x]$  Verweis auf Elternknoten,  $left[x]$  Verweis auf linkes Kind,  $right[x]$  Verweis auf rechtes Kind.
- Falls  $p[x]=NIL$ , dann ist  $x$  Wurzel des Baums.
- $left[x] / right[x]=NIL$ : kein linkes/rechtes Kind.
- Zugriff auf Baum  $T$  durch Verweis  $root[T]$  auf Wurzelknoten.

# Binäre Suchbäume

---

## Binäre Suchbäume

- Verwende Binärbaum
- Speichere Schlüssel „geordnet“

## Binäre Suchbaumeigenschaft:

- Sei  $x$  Knoten im binären Suchbaum
- Ist  $y$  Knoten im **linken Unterbaum** von  $x$ , dann gilt  $\text{key}[y] \leq \text{key}[x]$
- Ist  $y$  Knoten im **rechten Unterbaum** von  $x$ , dann gilt  $\text{key}[y] \geq \text{key}[x]$

# Binäre Suchbäume

---

## Binäre Suchbäume:

- Ausgabe aller Elemente in  $O(n)$
- Suche, Minimum, Maximum, Nachfolger in  $O(h)$
- Einfügen, Löschen in  $O(h)$

## Frage:

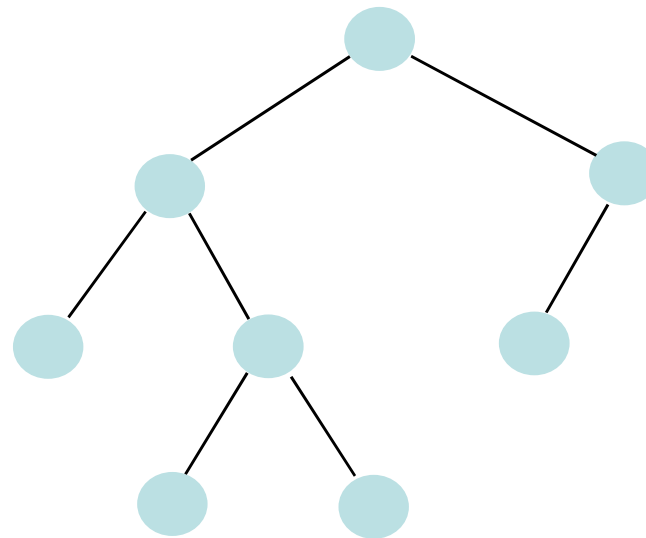
- Wie kann man eine „kleine“ Höhe unter Einfügen und Löschen garantieren?

# Balancierte binäre Suchbäume

---

## AVL-Bäume

- Ein Baum heißt AVL-Baum, wenn für jeden Knoten gilt: Die Höhe seines linken und rechten Teilbaums unterscheidet sich höchstens um 1.



## Dynamische AVL-Bäume

- Operationen Suche, Einfügen, Löschen, Min/Max, Vorgänger/Nachfolger,... wie in der letzten Vorlesung
- Laufzeit  $O(h)$  für diese Operationen
- Nur Einfügen/Löschen verändern Struktur des Baums

### Idee:

- Wir brauchen Prozedur, um AVL-Eigenschaft nach Einfügen/Löschen wiederherzustellen.

# Balancierte binäre Suchbäume

---

## Kurze Zusammenfassung:

- Wir können aus einem beinahe-AVL-Baum mit Hilfe von maximal 2 Rotationen einen AVL-Baum machen
- Dabei erhöht sich die Höhe des Baums nicht

## Einfügen:

- Wir fügen ein wie früher
- Dann laufen wir den Pfad zur Wurzel zurück
- An jedem Knoten balancieren wir, falls der Unterbaum ein beinahe-AVL-Baum ist

## Satz 12.3

Mit Hilfe von AVL-Bäumen kann man Suche, Einfügen, Löschen, Minimum und Maximum in einer Menge von  $n$  Zahlen in  $\Theta(\log n)$  Laufzeit durchführen.

# 13. Hashing

---

- Hashing einfache Methode um Wörterbücher zu implementieren, d.h. Hashing unterstützt die Operationen Search, Insert, Delete.
- Worst-case Zeit für Search:  $\Theta(n)$ .
- In der Praxis jedoch sehr gut.
- Unter gewissen Annahmen, erwartete Suchzeit  $\mathbf{O}(1)$ .
- Hashing Verallgemeinerung von direkter Adressierung durch Arrays.

# Hashing – Idee (1)

---

- Nehmen an, dass die Menge  $K$  der zu speichernden Schlüssel immer kleiner als das Universum  $U$  ist.
- Hashing hat dann Speicherbedarf  $\Theta(|K|)$ . Zeit für Insert wie bei direkter Adressierung  $O(1)$ .
- Zeit für Search und Delete ebenfalls  $O(1)$ , aber nur im Durchschnitt bei geeigneten Annahmen.
- Legen Array  $T[0, \dots, m-1]$  an, wobei  $m < |U|$ . Nennen  $T$  **Hashtabelle**.
- Benutzen **Hashfunktion**  $h:U \rightarrow \{0, \dots, m-1\}$ .
- Verweis auf Objekt mit Schlüssel  $k$  in  $T[h(k)]$ .

# Hashing – Idee (2)

---

- Da  $m < |U|$ , gibt es Objekte, deren Schlüssel auf denselben Wert gehasht werden, d.h., es gibt Schlüssel  $k_1, k_2$  mit  $k_1 \neq k_2$  und  $h(k_1) = h(k_2)$ . Dieses wird **Kollision** genannt.
- Verwaltung von Kollision erfolgt durch **Verkettung**.
- Speichern Objekte, deren Schlüssel auf den Hashwert  $h$  abgebildet werden, in einer doppelt verketteten Liste  $L_h$ . Dann verweist  $T[h]$  auf den Beginn der Liste, speichert also  $head[L_h]$ .
- Insert, Delete, Search jetzt mit Listenoperationen.

# Offene Adressierung (1)

---

- Hashing mit Kollisionsvermeidung weist Objekt mit gegebenen Schlüssel feste Position in Hashtabelle zu.
- Bei Hashing durch offene Adressierung wird Objekt mit Schlüssel keine feste Position zugewiesen.
- Position abhängig von Schlüssel und bereits belegten Positionen in Hashtabelle.
- Für neues Objekt wird erste freie Position gesucht. Dazu wird Hashtabelle nach freier Position durchsucht.
- Reihenfolge der Suche hängt vom Schlüssel des einzufügenden Objekts ab.

# Offene Adressierung (2)

---

- Keine Listen zur Kollisionsvermeidung. Wenn Anzahl eingefügter Objekte ist  $m$ , dann sind keine weiteren Einfügungen mehr möglich.
- Listen zur Kollisionsvermeidung möglich, aber Ziel von offener Adressierung ist es, Verfolgen von Verweisen zu vermeiden.
- Da keine Listen benötigt werden, kann die Hashtabelle vergrößert werden.
- Suchen von Objekten in der Regel schneller, da keine Listen linear durchsucht werden müssen.

# Offene Adressierung (3)

---

- Laufzeit für Einfügen nur noch im Durchschnitt  $\Theta(1)$ .
- Entfernen von Objekten schwierig, deshalb Anwendung von offener Adressierung oft nur, wenn Entfernen nicht benötigt wird.

# Hashfunktionen bei offener Adressierung

---

- Hashfunktion legt für jeden Schlüssel fest, in welcher Reihenfolge für Objekte mit diesem Schlüssel nach freier Position in Hashtabelle gesucht wird.

- Hashfunktion  $h$  von der Form

$$h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}.$$

$m$ :=Größe der Hashtabelle.

- Verlangen, dass für alle Schlüssel  $k$  die Folge  $(h(k,0), h(k,1), \dots, h(k, m-1))$  eine Permutation der Folge  $(0, 1, \dots, m-1)$  ist.
- $(h(k,0), h(k,1), \dots, h(k, m-1))$  heisst **Testfolge** bei Schlüssel  $k$ .

# Breitensuche – Überblick (1)

---

- Die *Breitensuche* ist ein Algorithmus, der die Grundlage vieler Graphenalgorithmien bildet.
- Ziel der Breitensuche ist es, bei einem Graphen  $G=(V,E)$  und einer *Quelle*  $s \in V$  alle Knoten  $v \in V$  zu finden, die von  $s$  aus *erreichbar* sind. Dabei ist ein Knoten  $v$  von  $s$  aus erreichbar, wenn es in  $G$  einen Pfad von  $s$  nach  $v$  gibt.
- Die Breitensuche berechnet auch für alle Knoten  $v$  den *Abstand*  $\delta(s,v)$  von  $s$  zu  $v$ . Dabei ist der Abstand von  $s$  zu  $v$  die minimale Anzahl von Kanten auf einem Pfad von  $s$  nach  $v$ .

# Breitensuche - Überblick (2)

---

- Die Breitensuche bestimmt alle Knoten mit Abstand  $< k$  vor den Knoten mit Abstand  $k$ . Daher der Name Breitensuche.
- Graphensuche funktioniert in gleicher Weise bei gerichteten und ungerichteten Graphen.
- Nehmen an, dass Eingabegraph in Adjazenzlisten-Darstellung gegeben ist.
- Sagen, dass ein Knoten *entdeckt* wird, wenn er das erste Mal bei der Breitensuche angetroffen wird.

# Tiefensuche

---

- Suche zunächst „tiefer“ im Graph
- Neue Knoten werden immer vom zuletzt gefundenen Knoten entdeckt
- Sind alle adjazenten Knoten des zuletzt gefundenen Knoten  $v$  bereits entdeckt, springe zurück zum Knoten, von dem aus  $v$  entdeckt wurde
- Wenn irgendwelche unentdeckten Knoten übrigbleiben, starte Tiefensuche von einem dieser Knoten

# Tiefensuche - Laufzeitanalyse

---

**Satz 14.11:** Bei Eingabe von Graph  $G=(V,E)$  besitzen DFS und BFS Laufzeit  $O(|V| + |E|)$ .

**Analyse:** Nutzen aus dass Gesamtgröße aller Adjazenzlisten  $O(|E|)$

# Kürzeste Wege

---

## Algorithmus von Dijkstra:

1. Es sei  $S$  die Menge der entdeckten Knoten
2. Invariante: Merke optimale Lösung für  $S$ :  
Für alle  $v \in S$  sei  $d[v] = \delta(s, v)$  die Länge des kürzesten Weges von  $s$  nach  $v$
3. Zu Beginn:  $S = \{s\}$  und  $d[s] = 0$
4. **while**  $V \neq S$  **do**
5. Wähle Knoten  $v \in V \setminus S$  mit mindestens einer Kante aus  $S$  und für den  $d'[v] = \min_{(u,v) \in E} d[u] + w(u, v)$  so klein wie möglich ist
6. Füge  $v$  zu  $S$  hinzu und setze  $d[v] \leftarrow d'[v]$

## Wie kann man Dijkstras Algorithmus effizient implementieren?

- Halte  $d'[v]$  Werte für alle  $v \in V-S$  aufrecht
- Speichere alle Knoten aus  $V-S$  in Prioritätenschlange ab mit Schlüssel  $d'[v]$
- Für jeden Knoten  $v$  speichere Zeiger auf sein Vorkommen im Heap
- $d'[v]$  Werte vergrößern sich nie
- Benutze Decrease-Key, wenn sich Wert  $d'[v]$  verringert

## Dijkstras Algorithmus

- Kürzeste Wege von einem Knoten und mit positiven Kantengewichten
- Laufzeit  $O((|V|+|E|) \log |V|)$
- Bessere Laufzeit von  $O(|V| \log |V| + |E|)$  möglich mit besseren Prioritätenschlängen

## Fragen:

- Was passiert bei negativen Kantengewichten?
- Wie kann man das kürzeste Wege Problem für alle Paare von Knoten effizient lösen?

# Berechnung minimaler Spannäume

---

**Ziel:** Gegeben ein gewichteter ungerichteter Graph  $(G,w)$ ,  $G=(V,E)$ . Wollen effizient einen minimalen Spannbaum von  $(G,w)$  finde.

**Vorgehensweise:** Erweitern sukzessive eine Kantenmenge  $A \subseteq E$  zu einem minimalen Spannbaum

1. Zu Beginn  $A = \{ \}$ .
2. Ersetzen in jedem Schritt  $A$  durch  $A \cup \{(u, v)\}$ , wobei  $(u, v)$  eine  $A$ -sichere Kante ist.
3. Solange bis  $|A| = |V| - 1$

**Definition 15.3:**  $(u, v)$  heißt  $A$ -sicher, wenn mit  $A$  auch  $A \cup \{(u, v)\}$  zu einem minimalen Spannbaum erweitert werden kann.

# Algorithmus von Prim - Idee

---

- Zu jedem Zeitpunkt des Algorithmus besteht der Graph  $G_A = (V, A)$  aus einem Baum  $T_A$  und einer Menge von isolierten Knoten  $I_A$ .
- Eine Kante minimalen Gewichts, die einen Knoten in  $I_A$  mit  $T_A$  verbindet, wird zu  $A$  hinzugefügt.
- Die Knoten in  $I_A$  sind in einem Min-Heap organisiert. Dabei ist der Schlüssel  $\text{key}[v]$  eines Knoten  $v \in I_A$  gegeben durch das minimale Gewicht einer Kante, die  $v$  mit  $T_A$  verbindet.

# Algorithmus von Kruskal - Datenstruktur

---

- Kruskals Algorithmus benötigt Datenstruktur mit deren Hilfe
  1. Für jede Kante  $(u,v) \in E$  effizient entschieden werden kann, ob  $u$  und  $v$  in derselben Zusammenhangskomponente von  $G_A$  liegen.
  2. Zusammenhangskomponenten effizient verschmolzen werden können.
- Solch eine Datenstruktur ist *Union-Find*-Datenstruktur für *disjunkte dynamische Mengen*.

# Union-Find mit verketteten Listen

---

Realisierung einer Datenstruktur für disjunkte dynamische Menge kann mit verketteten Listen erfolgen:

1. Für jede Teilmenge  $S$  gibt es eine verkettete Liste  $S$  der Objekte in  $S$ .
2. Repräsentant ist dabei das erste Objekt der Liste.
3. Zusätzlich  $\text{head}[S]$  und  $\text{tail}[S]$  Verweise auf erstes bzw. letztes Element der Liste. Feld  $\text{size}[S]$  speichert Größe der Liste.
4. Für jedes Element  $x$  der Liste Verweis  $\text{rep}[x]$  auf Repräsentanten der Liste.

# Algorithmus von Kruskal - Pseudocode

---

Kruskal – MST( $G, w$ )

1  $A \leftarrow \{ \}$

2 **for** alle  $v \in V$

3     **do** Make - Set( $v$ )

4     Sortiere die Kanten von  $G$  nach aufsteigendem Gewicht.

5 **for** alle Kante  $(u, v)$  von  $G$  in der Reihenfolge aufsteigenden Gewichts

6     **do if** Find - Set( $u$ )  $\neq$  Find - Set( $v$ )

7         **then**  $A \leftarrow A \cup \{(u, v)\}$

8         Union( $u, v$ )

9 **return**  $A$

# Laufzeitanalyse von Kruskals Algorithmus

---

*Satz 15.9:* Werden verkettete Listen als Union-Find-Datenstruktur benutzt, so ist die Laufzeit von Kruskals Algorithmus  $O(|V|\log(|V|) + |E|\log(|E|))$ .

# Matrix Multiplikation

## Teile und Herrsche:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix}$$

## Trick:

$$P_1 = A \cdot (F - H)$$

$$P_2 = (A + B) \cdot H$$

$$P_3 = (C + D) \cdot E$$

$$P_4 = D \cdot (G - E)$$

$$P_5 = (A + D) \cdot (E + H)$$

$$P_6 = (B - D) \cdot (G + H)$$

$$P_7 = (A - C) \cdot (E + F)$$

$$AE + BG = P_5 + P_4 - P_2 + P_6$$

$$AF + BH = P_1 + P_2$$

$$CE + DG = P_3 + P_4$$

$$CF + DH = P_5 + P_1 - P_3 - P_7$$

7 Multiplikationen!!!

# Divide & Conquer

---

## Teile & Herrsche:

- Problem in Teilprobleme aufteilen
- Teilprobleme rekursiv lösen
- Lösung aus Teillösungen zusammensetzen

## Probleme:

- Wie setzt man zusammen?  
[erfordert algorithmisches Geschick und Übung]
- Laufzeitanalyse (Auflösen der Rekursion)  
[ist normalerweise nach Standardschema; erfordert ebenfalls Übung]

# 18. Gierige Algorithmen

---

- Gierige Algorithmen sind eine Algorithmenmethode, um so genannte **Optimierungsprobleme** zu lösen.
- Bei einem Optimierungsproblem gibt es zu jeder *Probleminstanz* viele mögliche oder **zulässige Lösungen**. Lösungen haben **Werte** gegeben durch eine **Zielfunktion**. Gesucht ist dann eine möglichst gute zulässige Lösung. Also eine Lösung mit möglichst kleinem oder möglichst großem Wert (**Minimierungs- bzw. Maximierungsproblem**).

# Gieriges 1-Prozessor-Scheduling (1)

---

- Gegeben sind  $n$  Jobs  $j_1, \dots, j_n$  mit **Dauer**  $t_1, \dots, t_n$ .
- Jeder der Jobs muss auf einem einzigen Prozessor abgearbeitet werden. Der Prozessor kann zu jedem Zeitpunkt nur einen Job bearbeiten. Ein einmal begonnener Job darf nicht abgebrochen werden.
- Gesucht ist eine Reihenfolge, in der die Jobs abgearbeitet werden, so dass die **durchschnittliche Bearbeitungszeit** der Jobs möglichst gering ist.

# Gieriges 1-Prozessor-Scheduling (3)

---

*Lemma 18.1:* Bei Permutation  $\pi$  ist die durchschnittliche Bearbeitungszeit gegeben durch

$$\frac{1}{n} \sum_{i=1}^n (n-i+1) t_{\pi(i)}$$

*Lemma 18.2:* Eine Permutation  $\pi$  führt genau dann zu einer minimalen durchschnittlichen Bearbeitungszeit, wenn die Folge  $(t_{\pi(1)}, \dots, t_{\pi(n)})$  aufsteigend sortiert ist.

# Gieriges Mehr-Prozessor-Scheduling

---

- Gegeben sind  $n$  Jobs  $j_1, \dots, j_n$  mit **Dauer**  $t_1, \dots, t_n$  und  $m$  identische Prozessoren.
- Jeder der Jobs muss auf einem einzigen Prozessor abgearbeitet werden. Jeder Prozessor kann zu jedem Zeitpunkt nur einen Job bearbeiten. Ein einmal begonnener Job darf nicht abgebrochen werden.
- Gesucht ist eine Aufteilung der Jobs auf die Prozessoren und für jeden Prozessor eine Reihenfolge der ihm zugewiesenen Jobs, so dass durchschnittliche Bearbeitungszeit der Jobs möglichst gering ist.
- Bearbeitungszeit eines Jobs wie vorher definiert.

# Gieriges Mehr-Prozessor-Scheduling (2)

---

*Lemma 18.3:* Die Permutation  $\pi$  sei so gewählt, dass die Folge  $(t_{\pi(1)}, \dots, t_{\pi(n)})$  aufsteigend sortiert ist. Weiter werde dann Job  $j_{\pi(i)}$  auf dem Prozessor mit Nummer  $i \bmod m$  ausgeführt. Das so konstruiert Scheduling minimiert dann die durchschnittliche Bearbeitungszeit.

Hierbei identifizieren wir 0 und  $m$ , d.h. ist  $i \bmod m = 0$ , so wird Job  $j_{\pi(i)}$  auf dem Prozessor mit Nummer  $m$  gestartet.

## Formale Problemformulierung:

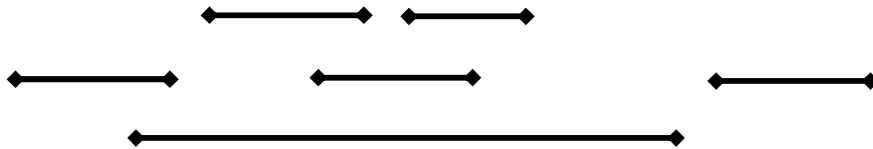
- Problem: Interval Scheduling
- Eingabe: Felder  $s$  und  $f$ , die die Intervalle  $(s[i], f[i])$  beschreiben
- Ausgabe: Indizes der ausgewählten Intervalle

## Wichtige Annahme:

- Eingabe nach Intervallendpunkten sortiert, d.h.
- $f[1] \leq f[2] \leq \dots \leq f[n]$

## Generelle Überlegung:

- Wähle erste Anfrage  $i_1$  geschickt
- Ist  $i_1$  akzeptiert, weise alle Anfragen zurück, die nicht kompatibel sind
- Wähle nächste Anfrage  $i_2$  und weise alle Anfragen zurück, die nicht mit  $i_2$  kompatibel sind
- Mache weiter, bis keine Anfragen mehr übrig sind



# Gierige Algorithmen – Datenkompression

---

## Grundlegendes Problem:

- Eingabe: Text in Alphabet  $\Sigma$
- Gesucht: Eine binäre Kodierung von  $\Sigma$ , so dass die Länge des Textes in dieser Kodierung minimiert wird

## Beispiel:

- $\Sigma = \{0, 1, 2, \dots, 9\}$
- Text = 00125590004356789 (17 Zeichen)

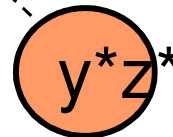
0	1	2	3	4	5	6	7	8	9
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

# Gierige Algorithmen – Datenkompression

---

## Idee des Algorithmus:

- Die beiden Symbole  $y^*$  und  $z^*$  mit den niedrigsten Frequenzen sind Bruderknoten
- Fasse  $y^*$  und  $z^*$  zu einem neuen Symbol zusammen
- Löse das Problem für die übrigen  $n-1$  Symbole (z.B. rekursiv)



## **Gierige Algorithmen:**

- Berechne Lösung schrittweise
- In jedem Schritt mache lokal optimale Wahl

## **Anwendbar:**

- Wenn optimale Lösung eines Problems optimale Lösung von Teilproblemen enthält

## **Algorithmen:**

- Scheduling Probleme
- Optimale Präfix-Kodierung (Huffman Codes)

# 18. Dynamisches Programmieren

---

- Dynamische Programmierung wie gierige Algorithmen eine Algorithmenmethode, um *Optimierungsprobleme* zu lösen.
- Wie Divide&Conquer berechnet Dynamische Programmierung Lösung eines Problems aus Lösungen zu Teilproblemen.
- Lösungen zu Teilproblemen werden *nicht* rekursiv gelöst.
- Lösungen zu Teilproblemen werden iterativ beginnend mit den Lösungen der kleinsten Teilprobleme berechnet (*bottom-up*).

# Längste gemeinsame Teilfolge (2)

---

- Z heisst längste gemeinsame Teilfolge von X und Y, wenn Z gemeinsame Teilfolge von X und Y ist und es keine andere gemeinsame Teilfolge von X und Y gibt, die größere Länge als Z besitzt.
- *Beispiel:*  $Z=(B,C,A,C)$  ist nicht längste gemeinsame Teilfolge von  $X=(A,B,A,C,A,B,C)$  und  $Y=(B,A,C,C,A,B,B,C)$ . Denn  $(B,A,C,A,C)$  ist eine längere gemeinsame Teilfolge von X und Y.
- Beim Problem **Längste-Gemeinsame-Teilfolge (LCS)** sind als Eingabe zwei Teilfolgen  $X = (x_1, \dots, x_m)$  und  $Y = (y_1, \dots, y_n)$  gegeben. Gesucht ist dann eine längste gemeinsame Teilfolge von X und Y.

# Rekursion für Länge von LCS

---

*Lemma 18.3:* Sei  $c[i, j]$  die Länge einer längsten gemeinsamen Teilfolge des  $i$ -ten Präfix  $X_i$  von  $X$  und des  $j$ -ten Präfix  $Y_j$  von  $Y$ . Dann gilt:

$$c[i, j] = \begin{cases} 0, & \text{falls } i = 0 \text{ oder } j = 0 \\ c[i-1, j-1] + 1, & \text{falls } i, j > 0 \text{ und } x_i = y_j . \\ \max\{c[i-1, j], c[i, j-1]\}, & \text{falls } i, j > 0 \text{ und } x_i \neq y_j \end{cases}$$

*Beobachtung:* Rekursive Berechnung der  $c[i, j]$  würde zu Berechnung immer wieder derselben Werte führen. Dieses ist ineffizient. Berechnen daher die Werte  $c[i, j]$  iterativ, nämlich zeilenweise.

# Dynamisches Programmieren - Struktur

---

1. Bestimme rekursive Struktur einer optimalen Lösung.
2. Entwerfe rekursive Methode zur Bestimmung des Werts einer optimalen Lösung.
3. Transformiere rekursiv Methode in eine iterative (bottom-up) Methode zur Bestimmung des Werts einer optimalen Lösung.
4. Bestimmen aus dem Wert einer optimalen Lösung und in 3. ebenfalls berechneten Zusatzinformationen eine optimale Lösung.