# 12. AuD Tafelübung T-C3

Simon Ruderich

26. Januar 2011

# ADTs

## ADTs

- `void` (= nichts) kann nie als Rückgabewert vorkommen!
- das „Objekt“ muss immer selbst mit übergeben werden

## Falsch

$$\text{push} : \text{int} \longrightarrow \text{AuDMenge}$$

(Natürlich wäre dies richtig für einen möglichen `create`-Konstruktor.)

## Richtig

$$\text{push} : \text{AuDMenge} \times \text{int} \longrightarrow \text{AuDMenge}$$

# ADTs

## AuDMenge: doppelte Elemente

A1: $\text{countElements}(\text{create}) = 0$

A2: $\text{countElements}(\text{push}(A, x)) = 1 + \text{countElements}(A)$

$\text{countElements}(\text{push}(\text{push}(\text{push}(\text{create}(), 5), 10), 5)) = ?$

# ADTs

## AuDMenge: doppelte Elemente

A1: $\text{countElements}(\text{create}) = 0$

A2: $\text{countElements}(\text{push}(A, x)) = 1 + \text{countElements}(A)$

$$\text{countElements}(\text{push}(\text{push}(\text{push}(\text{create}(), 5), 10), 5)) = 3$$

# ADTs

## AuDMenge: doppelte Elemente

$$\text{A1: countElements(create)} = 0$$

$$\text{A2: countElements(push}(A, x)) = 1 + \text{countElements}(A)$$

$$\text{countElements(push(push(push(create}(), 5), 10), 5)) = 3$$

## Richtig

$$\text{A1: countElements(create)} = 0$$

$$\text{A2: countElements(push}(A, x))$$

$$= \text{countElements}(A) + \begin{cases} 1 & \text{falls contains}(A, x) = \text{false} \\ 0 & \text{sonst} \end{cases}$$

# Hashtable

## Hashtable

- speichert Elemente mit eindeutigem Schlüssel
- Einfügen und Suchen (meist) in $\mathcal{O}(1)$
- Löschen schwieriger zu implementieren
- wird in einem Array gespeichert

## Hashfunktion

- bildet Schlüssel auf Indizes im Array ab
- reduziert nötigen Speicherplatz
- sollte Elemente möglichst gleichmäßig verteilen
- kann Kollisionen erzeugen

# Kollisionsauflösung

## Kollision

- gleicher Hashwert für unterschiedliche Schlüssel
- neuer Wert muss irgendwie gespeichert werden

## Auflösung durch verkettete Listen

- jede Stelle im Array hat noch einen `next`-Pointer
- bei einer Kollision wird dort das Element eingehängt
- bei mehrfacher Kollision wird die Liste ggf. sehr lang
  $\rightarrow$ Einfügen/Suchen in $\mathcal{O}(n)$
- Löschen leicht möglich

# Kollisionsauflösung

## Kollision

- gleicher Hashwert für unterschiedliche Schlüssel
- neuer Wert muss irgendwie gespeichert werden

## Auflösung durch Sondierung

- ist ein Feld besetzt wird nach fester Regel weitergesucht
- z. B. ein Feld weiter; mehrere Felder weiter, nicht linear, etc.
- das erste gefundene freie Feld wird verwendet
- Löschen ist problematisch

# Kollisionen

## (Primär)Kollision

- Stelle mit „normal“ eingefügtem Element schon belegt (gleicher Hashwert)
- tritt bei verketteten Listen und Sondierung auf

## Sekundärkollision

- Stelle mit Element aus Sondierung besetzt (meist anderer Hashwert)
- tritt nur bei Sondierung auf
- Risiko kann durch gute Sondierfunktion reduziert werden

# Hashfunktion

## Hashfunktion

- muss die Elemente gut verteilen (sonst Kollisionen)
- optimal nur ein Element pro Feld (unwahrscheinlich)
- gute Hashfunktion: $h(x) = x \mod m$,
  $m$ Länge des Arrays und Primzahl
- schlechte Hashfunktion: viele Elemente mit gleichem Hash
  $\Leftrightarrow$ viele Kollisionen
  $\Leftrightarrow$ viele Felder leer (bei verketteter Liste)

# Heaps

## Heaps

- balancierter Binärbaum (Vater hat maximal zwei Kinder)
- Kinder immer kleiner/gleich als der Vater (Max-Heap)
- Kinder immer größer/gleich als der Vater (Min-Heap)
- größtes (Max-Heap)/kleinstes (Min-Heap) Element in $\mathcal{O}(1)$
- Einfügen/Entfernen in $\mathcal{O}(\log n)$
- meist in Array gespeichert, da einfacher Zugriff

# Heaps

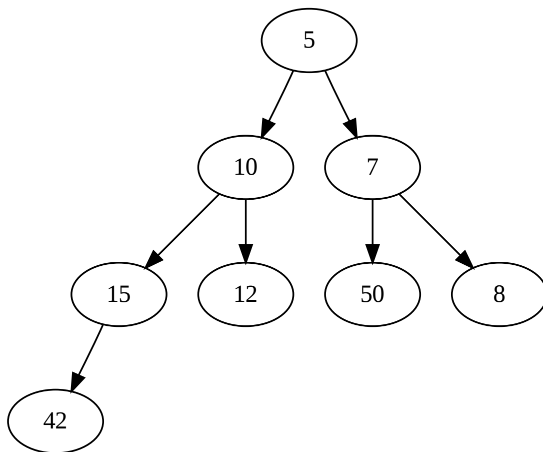
## Heaps

- balancierter Binärbaum (Vater hat maximal zwei Kinder)
- Kinder immer kleiner/gleich als der Vater (Max-Heap)
- Kinder immer größer/gleich als der Vater (Min-Heap)
- größtes (Max-Heap)/kleinstes (Min-Heap) Element in $\mathcal{O}(1)$
- Einfügen/Entfernen in $\mathcal{O}(\log n)$
- meist in Array gespeichert, da einfacher Zugriff

## Zugriff

- Wurzel: $A[0]$
- linker Nachfolger von Knoten $i$: $A[2 * i + 1]$
- rechter Nachfolger von Knoten $i$: $A[2 * i + 2]$
- Vorgänger des Knoten $k$: $A[(k - 1)/2]$

# Beispiel (Min-Heap)



# Einfügen/Entfernen

## Einfügen

- neues Element am Ende des Heaps eintragen
- mit Vater tauschen falls kleiner (Min-Heap)/größer (Max-Heap)
- solange wiederholen bis Heap-Eigenschaft hergestellt ist

## Entfernen

- erstes Element entfernen
- letztes Element als erstes eintragen
- mit dem kleinsten (Min-Heap)/größten (Max-Heap) Kind tauschen (falls nötig) („versickern“)
- solange wiederholen bis Heap-Eigenschaft hergestellt ist

# Heap-Eigenschaft herstellen

## in $\mathcal{O}(n \log n)$

- leeren Heap erstellen
- jedes Element einzeln einfügen und Heap-Eigenschaft herstellen (versickern)

## in $\mathcal{O}(n)$

- hintere Hälfte der Halde sind schon Halden (die Kinder liegen außerhalb der Halde)
- betrachte Knoten $i = n/2 - 1$ als eigenen Heap, Heap-Eigenschaft durch versickern herstellen
- Knoten $i - 1$ betrachten und auf diesem „Heap" wieder mit versickern die Heap-Eigenschaft herstellen
- bis zum Anfang des Heaps wiederholen

# Heapsort

## Heapsort

- zuerst Aufbau eines Heaps ($\mathcal{O}(n)$ bzw. $\mathcal{O}(n \log n)$)
- dann jeweils das erste Element entnehmen ($\mathcal{O}(1)$) (am Ende des Arrays speichern)
- Heap-Struktur wieder herstellen ($\mathcal{O}(\log n)$)
- fortsetzen bis der Heap leer ist
- Aufwand: $\mathcal{O}(n \log n)$
- in-place
- keine stabile Sortierung

# Quicksort Aufgabe

## Pivotelement

- Pivotelement am Anfang
- $i$ auf das erste, $j$ auf das letzte Element setzen
- solange $i \neq j$
  - $j$ verkleinern bis $A[j]$ kleiner als Pivotelement oder $i = j$
  - $i$ vergrößern bis $A[i]$ größer als Pivotelement oder $i = j$
  - $A[i]$ und $A[j]$ vertauschen
- Pivotelement ($A[0]$) mit $i$ ($= j$) vertauschen
- rekursiv auf linkes und rechtes Teilarray anwenden

# Dynamische Programmierung

Aufgabe

# Fragen

Fragen?
Evaluation