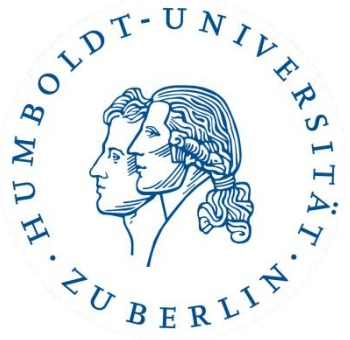
Übung Algorithmen und Datenstrukturen



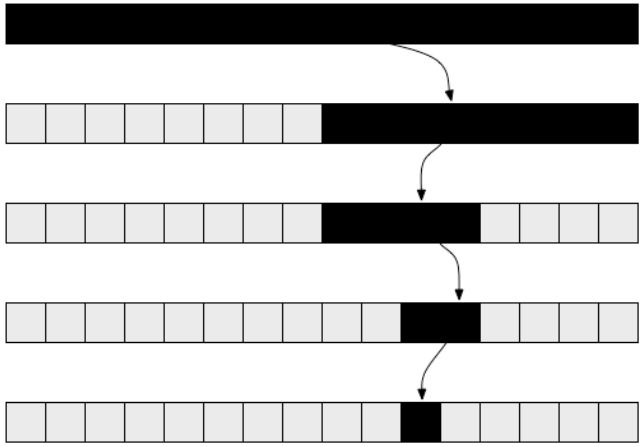
Sommersemester 2016

Kim Völlinger

(in Anlehnung an die Folien von Marc Bux)

Binäre Suche – Schreibtischtest

```
1.  A: sorted_int_array;
2.  c: int;
3.  l := 1;
4.  r := |A|;
5.  while l≤r do
6.    m := (l+r) div 2;
7.    if c<A[m] then
8.      r := m-1;
9.    else if c>A[m] then
10.     l := m+1;
11.   else
12.     return true;
13. end while,
14. return false;
```



Source: railspikes.com

Schreibtischtest für die binäre Suche:

$$A = [5, 12, 15, 17, 22, 29, 45, 47, 60, 61, 68, 74, 77]$$

gesuchter Wert $c = 69$

Tabelle: Werte von `l`, `r` und `m` nach jedem Aufruf von Zeile 6

Interpolationssuche

Interpolationssuche

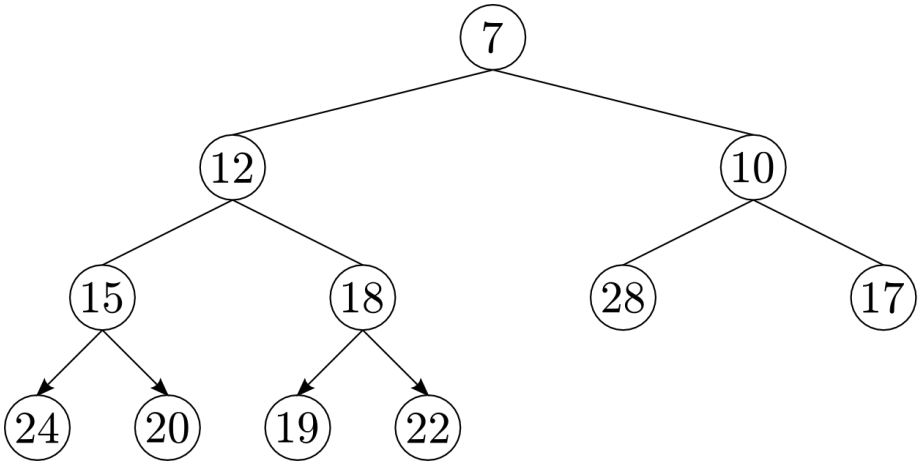
Input: sortiertes Array A der Länge n, zu suchende Zahl c
Output: **true** falls c in A, sonst **false**

```
1:  l := 1; r := n;
2:  while c ≥ A[l] and c ≤ A[r] do
3:     diff := c − A[l];
4:     range := A[r] − A[l];
5:     if range = 0 then
6:        rank := l;
7:     else
8:        rank := l + ⌊(r − l) * diff / range⌋;
9:     end if
10:    if c > A[rank] then
11:       l := rank + 1;
12:    else
13:       if c < A[rank] then
14:          r := rank − 1;
15:       else
16:          return true;
17:       end if
18:    end if
19: end while
20: return false;
```

Was ist die Idee des Algorithmus?

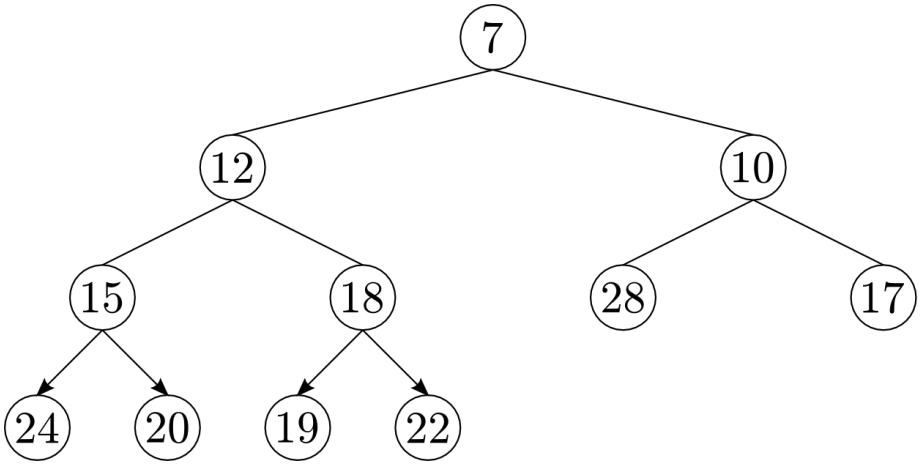
Heaps

- Ein Heap ist eine auf Bäumen basierende Datenstruktur zum Speichern von Elementen, über deren Schlüssel eine totale Ordnung definiert ist



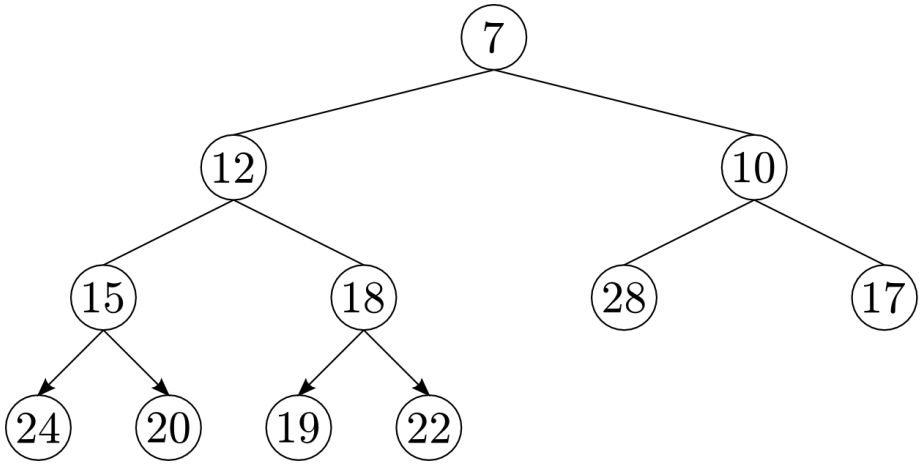
Heaps

- Ein Heap ist eine auf Bäumen basierende Datenstruktur zum Speichern von Elementen, über deren Schlüssel eine totale Ordnung definiert ist
  - Form Constraint: Der Baum ist fast vollständig
    - Alle Ebenen außer der untersten müssen vollständig gefüllt sein
    - In der letzten Ebene werden Elemente von links nach rechts aufgefüllt



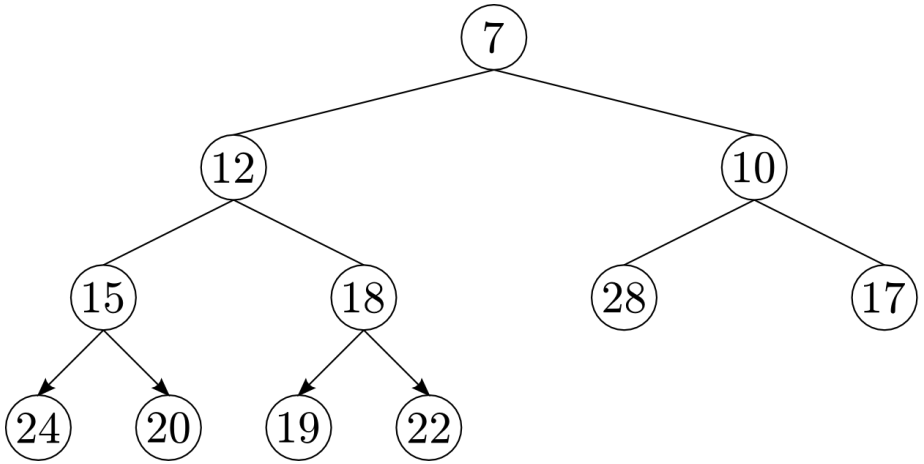
Heaps

- Ein Heap ist eine auf Bäumen basierende Datenstruktur zum Speichern von Elementen, über deren Schlüssel eine totale Ordnung definiert ist
  - Form Constraint: Der Baum ist fast vollständig
    - Alle Ebenen außer der untersten müssen vollständig gefüllt sein
    - In der letzten Ebene werden Elemente von links nach rechts aufgefüllt
  - Heap Constraint: Bei einem Min-Heap (Max-Heap) sind die Schlüssel jedes Knotens kleiner (größer) als die Schlüssel seiner Kinder



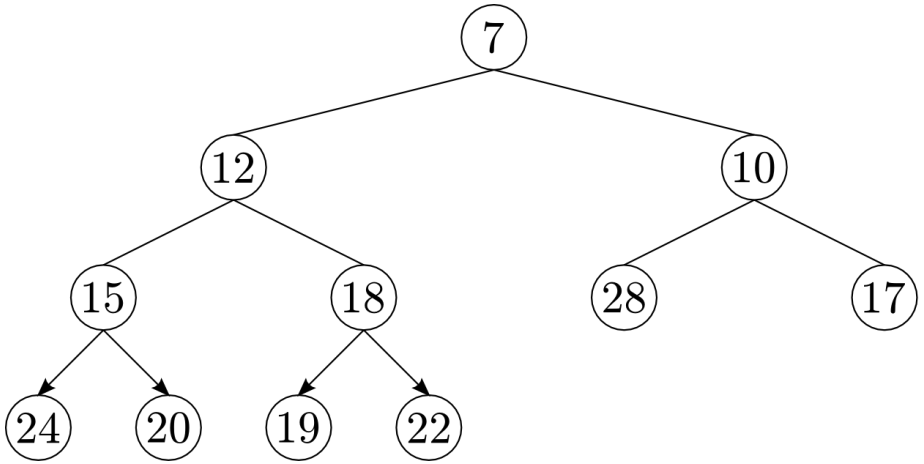
Heaps

- Ein Heap ist eine auf Bäumen basierende Datenstruktur zum Speichern von Elementen, über deren Schlüssel eine totale Ordnung definiert ist
  - Form Constraint: Der Baum ist fast vollständig
    - Alle Ebenen außer der untersten müssen vollständig gefüllt sein
    - In der letzten Ebene werden Elemente von links nach rechts aufgefüllt
  - Heap Constraint: Bei einem Min-Heap (Max-Heap) sind die Schlüssel jedes Knotens kleiner (größer) als die Schlüssel seiner Kinder
- Heaps lassen sich als heapgeordnete Arrays repräsentieren



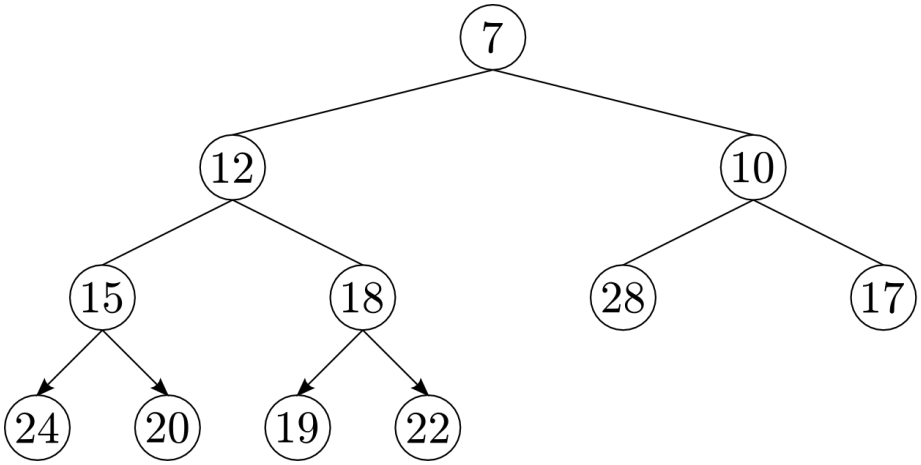
Heapgeordnetes Array

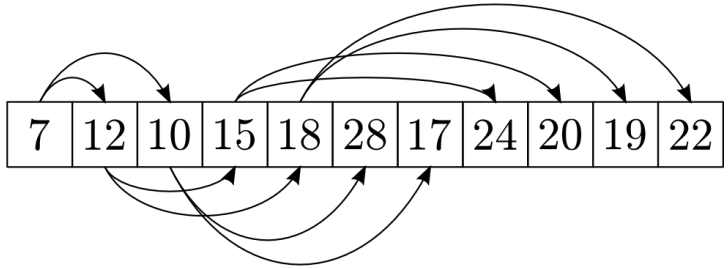
Stell den Heap als heapgeordnetes Array dar.



Heapgeordnetes Array

Stell den Heap als heapgeordnetes Array dar.



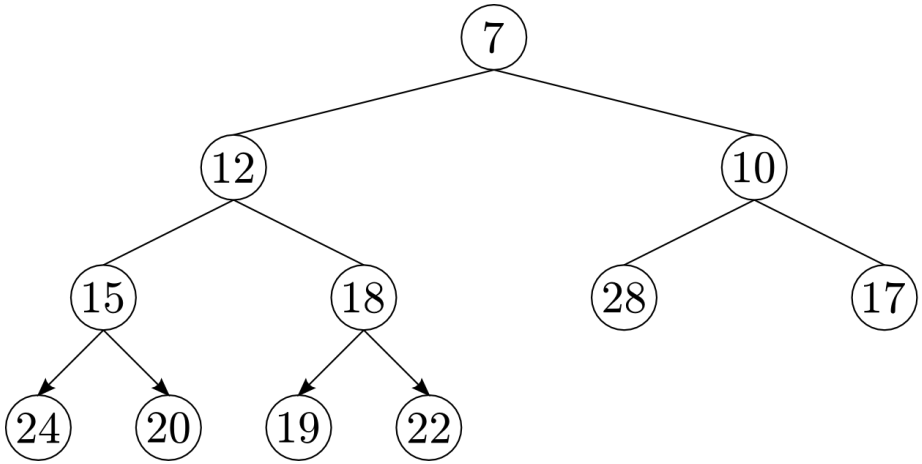


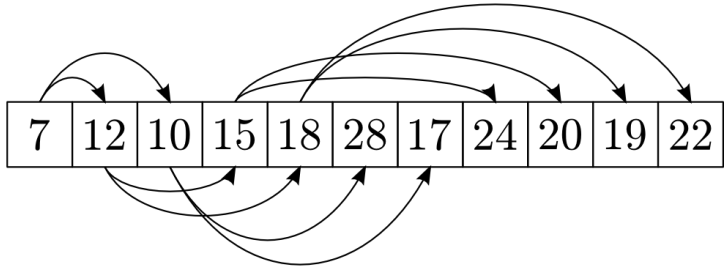
Mögliche Min-Heaps

Gib alle möglichen Min-Heaps zur Speicherung der Zahlen 1, 2, 3, 4 und 5 an.

Löschen und Eingügen

Es sei der folgende Min-Heap als heapgeordnetes Array gegeben:





Wie sieht der Heap nach Anwendung der folgenden Operationen (in der gegebenen Reihenfolge) aus?

`deleteMin()`, `deleteMin()`, `add(14)`, `add(8)`

Amortisierte Analyse

- Gegeben: Datenstruktur D, Folge Q von Operationen

Amortisierte Analyse

- Gegeben: Datenstruktur D, Folge Q von Operationen
- Voraussetzungen:
  - die Kosten aufeinanderfolgender Operation schwanken stark
  - teure Operationen benötigen viele vorangehende günstigere

Amortisierte Analyse

- Gegeben: Datenstruktur D, Folge Q von Operationen
- Voraussetzungen:
  - die Kosten aufeinanderfolgender Operation schwanken stark
  - teure Operationen benötigen viele vorangehende günstigere
- Ziel: Bessere obere Schranke für Zeit- oder Speicher-komplexität der Datenstruktur im Worst Case ermitteln

Amortisierte Analyse

- Gegeben: Datenstruktur D, Folge Q von Operationen
- Voraussetzungen:
  - die Kosten aufeinanderfolgender Operation schwanken stark
  - teure Operationen benötigen viele vorangehende günstigere
- Ziel: Bessere obere Schranke für Zeit- oder Speicher-komplexität der Datenstruktur im Worst Case ermitteln
- Ermitteln durchschnittliche Kosten für jede Operation im Worst Case

Amortisierte Analyse

- Gegeben: Datenstruktur D, Folge Q von Operationen
- Voraussetzungen:
  - die Kosten aufeinanderfolgender Operation schwanken stark
  - teure Operationen benötigen viele vorangehende günstigere
- Ziel: Bessere obere Schranke für Zeit- oder Speicher-komplexität der Datenstruktur im Worst Case ermitteln
- Ermitteln durchschnittliche Kosten für jede Operation im Worst Case
- Grundidee: Betrachtung konstruierter, amortisierter Kosten für Operationen, die im Gegensatz zu den realen Kosten
  - weniger stark schwanken und
  - für mehrere Operationen aufsummiert eine obere Schranke für die aufsummierten realen Kosten liefern

Amortisierte Analyse – Verfahren

- Aggregatmethode: Ermitteln einer oberen Schranke (oder eines exakten Terms) für die aufsummierten Gesamtkosten $T(n)$ von n Operationen
  - amortisierte Kosten ergeben sich als $\frac{T(n)}{n}$ und sind für alle Operationen identisch

Amortisierte Analyse – Verfahren

- Aggregatmethode: Ermitteln einer oberen Schranke (oder eines exakten Terms) für die aufsummierten Gesamtkosten $T(n)$ von n Operationen
  - amortisierte Kosten ergeben sich als $\frac{T(n)}{n}$ und sind für alle Operationen identisch
- Guthabenmethode

Amortisierte Analyse – Verfahren

- Aggregatmethode: Ermitteln einer oberen Schranke (oder eines exakten Terms) für die aufsummierten Gesamtkosten $T(n)$ von n Operationen
  - amortisierte Kosten ergeben sich als $\frac{T(n)}{n}$ und sind für alle Operationen identisch
- Guthabenmethode
- Potentialmethode

Amortisierte Analyse – Verfahren

- Aggregatmethode: Ermitteln einer oberen Schranke (oder eines exakten Terms) für die aufsummierten Gesamtkosten $T(n)$ von n Operationen
  - amortisierte Kosten ergeben sich als $\frac{T(n)}{n}$ und sind für alle Operationen identisch
- Guthabenmethode
- Potentialmethode
- Grundidee von Guthaben- und Potentialmethode: Anfängliche (eigentlich günstige) Operation teurer bewerten um spätere (teure) Operationen auszugleichen

Beispiel Binärzähler

Gegeben:

Bit-Array A[0..k-1] und Operation Increment

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3        A[i] = 0
4        i = i + 1
5  if i < A.length
6        A[i] = 1
```

Kosten für increment

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Kosten:
Anzahl der Bitänderungen
(jedes Bit kostet 1)