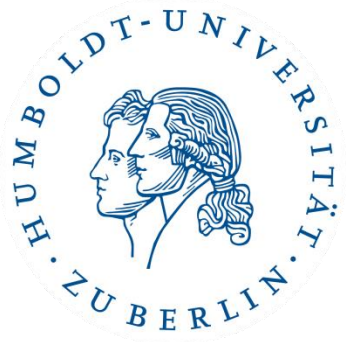
Übung Algorithmen und Datenstrukturen



Sommersemester 2016

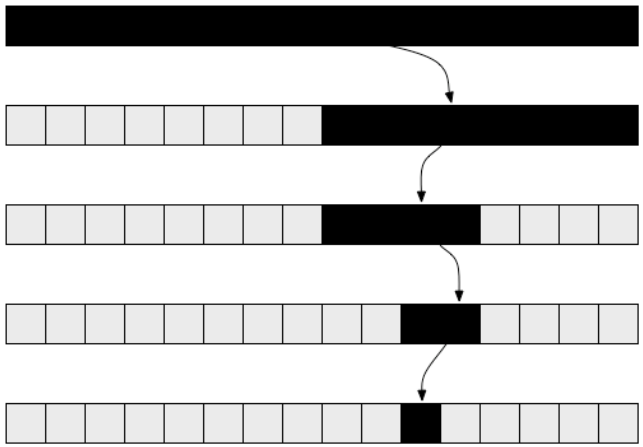
Marc Bux, Humboldt-Universität zu Berlin

Agenda

- Suche in sortierten Arrays
- Heaps
- Amortisierte Analyse

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;
```

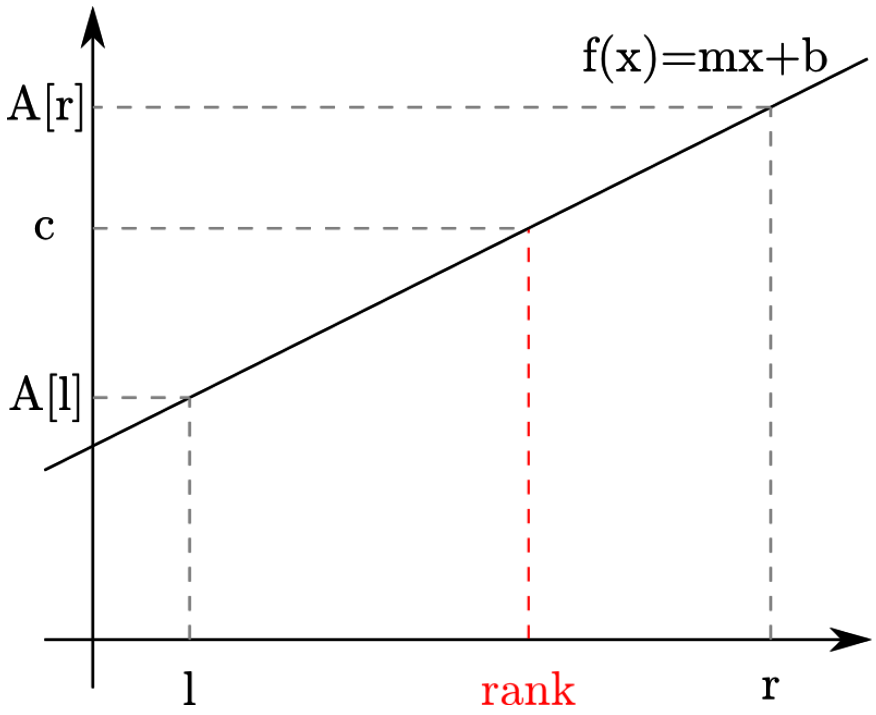


Source: railspikes.com

Führen Sie einen Schreibtischtest für die binäre Suche durch, bei dem das folgende Array A nach dem Wert $c = 69$ durchsucht wird. Geben Sie dazu an, mit welchen Werten die Variablen `l`, `r` und `m` nach jedem Aufruf von Zeile 6 belegt sind.

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

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;
```

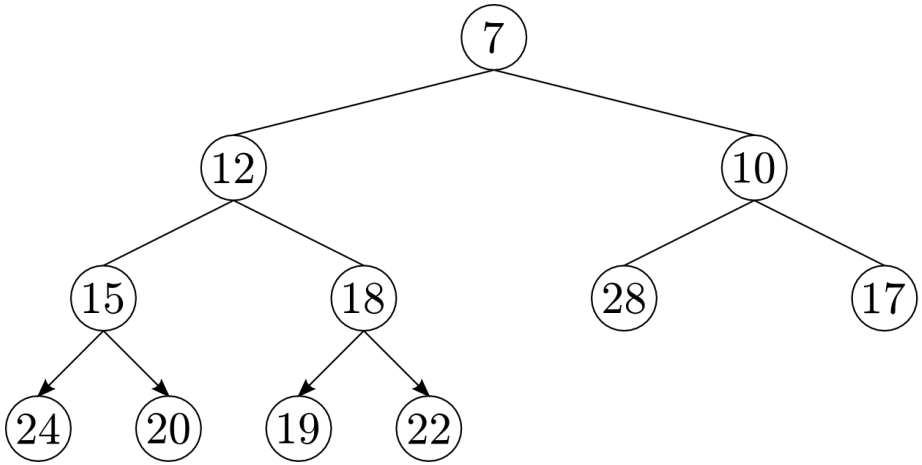
- $\frac{r-l}{A[r]-A[l]} = \frac{\text{rank}-l}{c-A[l]}$
- $\text{rank} = l + \frac{(r-l)(c-A[l])}{A[r]-A[l]} = l + \frac{(r-l)\cdot \text{diff}}{\text{range}}$

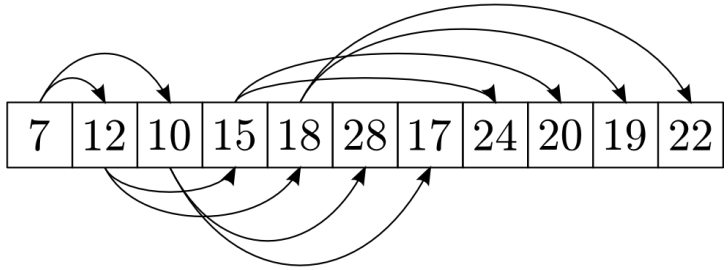
Agenda

- Suche in sortierten Arrays
- Heaps
- Amortisierte Analyse

Heaps

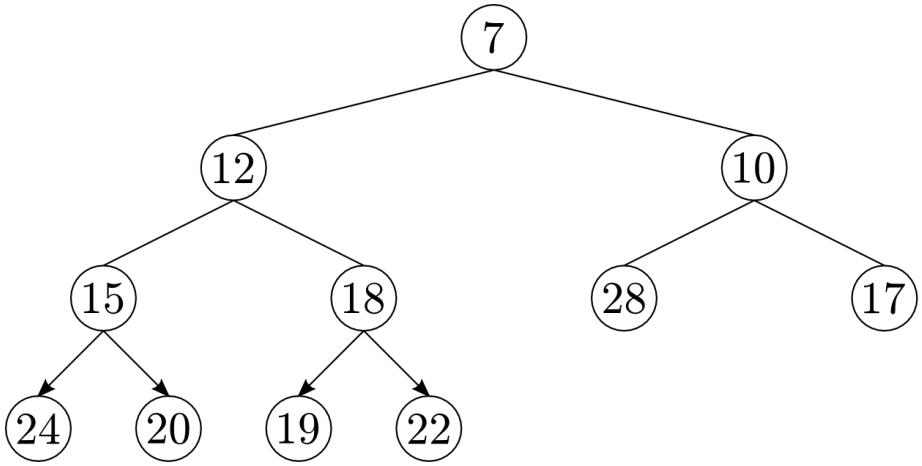
- Ein Heap ist eine auf Bäumen basierende Datenstruktur zum Speichern von Elementen, über deren Schlüssel eine totale Ordnung definiert ist
  - Form Constraint: Der Baum ist fast vollständig
    - Alle Ebenen außer der untersten müssen vollständig gefüllt sein
    - In der letzten Ebene werden Elemente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

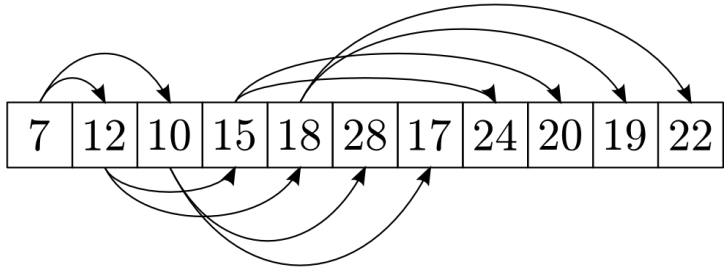




Aufgaben zu Min-Heaps

1. Geben Sie alle möglichen Min-Heaps zur Speicherung der Zahlen 1, 2, 3, 4 und 5 an.
2. Es sei der folgende Min-Heap als heapgeordnetes Array gegeben:





Wie sehen Heap und Array nach Anwendung der folgenden Operationen (in der gegebenen Reihenfolge) aus?
`deleteMin()`, `deleteMin()`, `add(14)`, `add(8)`

Agenda

- Suche in sortierten Arrays
- Heaps
- Amortisierte Analyse

Binärzähler

- Geg.: k-Bit Binärzähler
- Array von k Bits $b_0, b_1, \ldots, b_{k-1}$
  - Entspricht Binärzahl $b_{k-1}\ldots b_1 b_0$ bzw. Dezimalzahl $\sum_i b_i 2^i$
- Operation: Zahl inkrementieren (um 1 erhöhen)
- Kosten: Anzahl der Bitänderungen (jedes Bit kostet 1)

n	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1

1 Bitwechsel

n	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
23	0	0	0	1	0	1	1	1
24	0	0	0	1	1	0	0	0

4 Bitwechsel

Binärzähler – Kostenabschätzung

- Gesucht: Kosten für n Inkrement-Operationen, wenn Zähler bei 0 beginnt
- Best Case: Eine Bitänderung
- Worst Case: Alle k Bits werden verändert
- Ergibt Kosten: $n \cdot k = \mathrm{O}(nk)$
- Problem
  - sehr pessimistische Abschätzung, da Worst Case eher selten

n	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
127	0	1	1	1	1	1	1	1
128	1	0	0	0	0	0	0	0

k Bitwechsel

Binärzähler – Kostenabschätzung (2)

n	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	#BW
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	**2**
3	0	0	0	0	0	0	1	1	**1**
4	0	0	0	0	0	1	0	0	**3**
5	0	0	0	0	0	1	0	1	**1**
6	0	0	0	0	0	1	1	0	**2**
7	0	0	0	0	0	1	1	1	**1**
8	0	0	0	0	1	0	0	0	**4**
9	0	0	0	0	1	0	0	1	**1**
10	0	0	0	0	1	0	1	0	**2**

⇒ Häufig geringe Kosten

Gesamt

n	#BW	kn
1	1	8
2	3	16
3	4	24
4	7	32
5	8	40
6	10	48
7	11	56
8	15	64
9	16	72
10	18	80

⇒ Abschätzung zu pessimistisch

Amortisierte Analyse

- Amortisation: Anfängliche Aufwendungen für spätere Erträge
- 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
- 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
- Guthabenmethode (Bankkontomethode): Für jede Operation wird etwas Guthaben auf einem Konto eingezahlt / abgehoben
  - amortisierte Kosten ergeben sich aus den realen Kosten und dem Guthabenzuwachs
- Potentialmethode: Ermitteln einer Potentialfunktion, die jedem Zustand der Datenstruktur einen Wert zuweist und hinreichend Potential für spätere teure Operationen sammelt
  - amortisierte Kosten ergeben sich aus den realen Kosten und dem Potentialzuwachs
- Grundidee von Guthaben- und Potentialmethode: Anfängliche (eigentlich günstige) Operation teurer bewerten um spätere (teure) Operationen auszugleichen

Aggregatmethode – k-Bit Binärzähler

- Beobachtung (vgl. Vorlesung, Folie 7):
  - Niedrigstes Bit ändert sich bei jeder Inkrementation
  - Zweitniedrigstes Bit ändert sich bei jeder zweiten Inkrementation
  - Drittniedrigstes Bit ändert sich bei jeder vierten Inkrementation
  - k-niedrigstes Bit ändert sich bei jeder 2^{k-1}ten Inkrementation
- Für n Inkrement-Operationen ergeben sich Gesamtkosten

$$T(n) = n + \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor + \cdots + \left\lfloor \frac{n}{2^{k-1}} \right\rfloor \leq n \sum_{i=0}^{k-1} \frac{1}{2^i}$$

$$\leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n \in O(n)$$

- Als amortisierte Kosten pro Operation ergeben sich

$$\frac{T(n)}{n} \leq 2 \in O(1)$$

Potentialmethode

- Potentialfunktion $\Phi: D_i \rightarrow \mathbb{R}$ ordnet jedem Zustand der Datenstruktur ein Potential zu
- Seien c_i die realen Kosten der i-ten Operation
- Amortisierte Kosten $d_i = c_i + \Phi(\mathrm{D_i}) - \Phi(D_{i-1})$
  - ergibt sich aus den realen Kosten und dem Potentialzuwachs
- Amortisierte Gesamtkosten:

$$\sum_{i=1}^{n} d_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

- Ziel: Finde eine Potentialfunktion Φ, so dass
  1. $\Phi(D)$ von einer Eigenschaft von D abhängt,
  2. $\Phi(D_i) \geq \Phi(D_0)$,
  3. d_i lässt sich für alle i berechnen.
- Dann sind die amortisierten Gesamtkosten obere Schranke für die tatsächlichen Gesamtkosten
- Herangehensweise: Ermitteln einer Potentialfunktion, die hinreichend (aber nicht unnötig viel) Potential für spätere teure Operationen sammelt

Potentialmethode – Binärzähler

- $\Phi(\mathrm{D_i})$: Anzahl der Einsen im Zähler
- In der i-ten Operation springen t_i Bits auf 0 und 0-1 Bits auf 1
  - Es folgt: $c_i \leq t_i + 1$
- Falls $\Phi(D_i) = 0$, so ist $\Phi(D_{i-1}) = k$
- Falls $\Phi(D_i) > 0$, so ist $\Phi(D_i) \leq \Phi(D_{i-1}) - t_i + 1$
  - In beiden Fällen gilt $\Phi(D_i) \leq \Phi(D_{i-1}) - t_i + 1$
- Es folgt:

$$d_i = c_i + \Phi(\mathrm{D_i}) - \Phi(D_{i-1})$$
$$d_i \leq t_i + 1 + \Phi(D_{i-1}) - t_i + 1 - \Phi(D_{i-1}) \leq 2$$

- Es gilt:
  1. $\Phi(D)$ hängt von einer Eigenschaft von D ab,
  2. $\Phi(D_i) \geq \Phi(D_0)$,
  3. $d_i \leq 2$.
- $\sum_{i=1}^{n} d_i \leq 2n \in \mathrm{O}(n)$ ist obere Schranke für die Komplexität

Ausblick: Nächste Woche

- Besprechung der Lösungen des vierten Übungsblatts
- Fragen?