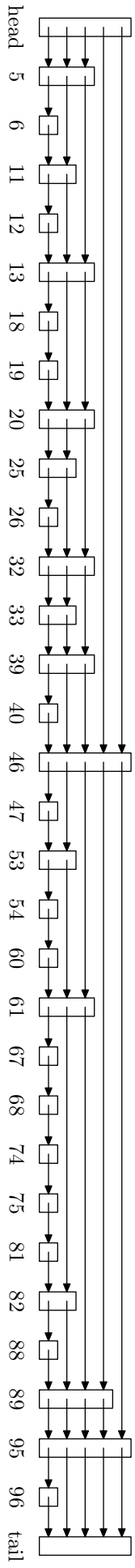
RWTH Aachen **SoSe 2007**
Lehrgebiet Theoretische Informatik Blatt 7
Kneis—Mölle—Richter—Rossmanith—Thiemann 21.5.2007

Übungen zur Vorlesung Datenstrukturen und Algorithmen

T16

Gegeben sei die rechts abgebildete Skip-List. Welche Knoten werden bei einer Suche nach 54 beziehungsweise 80 berührt?

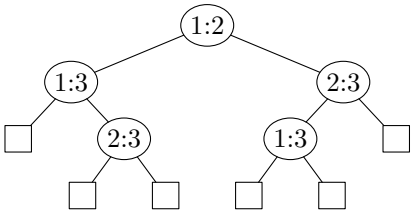


Lösungsvorschlag:

Die Suche nach 54 läuft über *head*, 46, 53 und schließlich 54. Die erfolglose Suche nach 80 läuft über *head*, 46, 61, 67, 68, 74 und 75. Der nächste Knoten enthält den Schlüssel 81, so daß die Suche abgebrochen werden kann.

T17

Gibt es einen vergleichsbasierten Sortieralgorithmus, der folgenden Vergleichsbaum besitzt?



Lösungsvorschlag:

Nein, denn die Eingaben $1, 2, 3$ und $1, 3, 2$ sind in diesem Vergleichsbaum nicht zu unterscheiden (sie sind demselben Blatt zugeordnet).

T18

Analysieren Sie die Anzahl der Vergleiche (von Elementen) im Erwartungswert für den Algorithmus Mergesort, wenn die Eingabe aus $n = 2^k$ verschiedenen Schlüsseln besteht und jede ihrer Permutationen gleich wahrscheinlich ist.

```
procedure Mergesort(l,r):
  if l ≥ r then return fi;
  m ← ⌈(r+l)/2⌉;
  Mergesort(l,m−1);
  Mergesort(m,r);
  i ← l; j ← m; k ← l;
  while k ≤ r do
    if a[i] ≤ a[j] and i<m or j>r
    then b[k] ← a[i]; k ← k+1; i ← i+1
    else b[k] ← a[j]; k ← k+1; j ← j+1 fi
```

od;
for i=l ,..., r **do** a[k] ← b[k] **od**

Lösungsvorschlag:
Wir verwenden die Rekursionsgleichung aus der Vorlesung, betrachten aber den Term $\Theta(n)$ für die Vergleiche von Elementen beim Mischen genauer. Derartige Vergleiche passieren nur am Anfang des Rumpfes der *while*-Schleife.
Man sieht leicht, daß k die Werte von l bis r durchläuft und pro Vergleich um eins steigt. Die Anzahl der Vergleiche beträgt somit $r-l+1$, was aber gerade die Länge n des jeweils betrachteten Teilarrays ist. Also erhalten wir für $n \geq 2$ die Gleichung

$$T(n) = n + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) = n + 2T(n/2),$$

weil $n = 2^k$ eine Zweierpotenz ist. Andererseits ist $T(1) = 0$. Wiederholtes Einsetzen liefert

$$T(n) = n + 2\frac{n}{2} + 4\frac{n}{4} + \ldots + 2^{k-1}\frac{n}{2^{k-1}} = kn = n \log n.$$

H14 (5 Punkte)
Sei $\mathcal{F}$ die Menge aller Funktionen $U \to \{1, \ldots, m\}$.
Beweisen oder widerlegen Sie: $\mathcal{F}$ ist eine universelle Familie von Hashfunktionen.
Falls Ihre Antwort *ja* ist, sollte man diese Familie dann praktisch verwenden? Falls die Anwort *nein* ist, kann man es einfach reparieren?

Lösungsvorschlag:
Die Anzahl der Funktionen $U \to \{1, \ldots, m\}$ ist natürlich $m^{|U|}$. Für beliebige x, y mit $x \neq y$ gibt es hingegen nur $m^{|U|-1}$ viele Funktionen $h\colon U \to \{1, \ldots, m\}$ mit $h(x) = h(y)$, weil der Funktionswert von y durch $h(x)$ determiniert ist. Folglich gilt

$$\frac{\{\, h \in \mathcal{F} \mid h(x) = h(y) \,\}}{|\mathcal{F}|} = \frac{m^{|U|-1}}{m^{|U|}} = \frac{1}{m},$$

so daß $\mathcal{F}$ universell ist.
Allerdings spricht gegen eine praktische Verwendung, daß die meisten Funktionen aus $\mathcal{F}$ nur mit extrem hohen Speicheraufwand dargestellt werden können: Um die Bilder von $|U|$ verschiedenen Objekten speichern zu können, benötigt man bei einer zufällig gewählten Funktion aus $\mathcal{F}$ etwa $|U| \log m$ viele Bits. So würde eine zufällig gewählte Hashfunktion aus $\mathcal{F}$, die Zehn-Byte-Nachnamen auf eine Million Hashwerte abbildet, etwa $2^{80} \cdot 20$ Bits benötigen. Da 2^{33} Bits ein Gigabyte bilden, entspricht dies

$$2^{47} \cdot 20 = 2,814,749,767,106,560$$

Gigabyte bzw. 2.5 Yottabyte.

H15 (10+10 Punkte)

Implementieren Sie eine Klasse *Bitset* für durch Bitmaps repräsentierte Mengen, die Funktionen für die Berechnung aller Teilmengen enthält:

Bitset(int i)	Konstruktor mit Menge
void start()	initialisiere den Teilmengeniterator
int subset()	liefere die aktuelle Teilmenge
boolean more()	teste, ob eine weitere Iteration möglich ist
void step()	gehe zur nächsten Teilmenge über

Insbesondere soll Ihre Klasse mit dem unten dargestellten Programm korrekt funktionieren, dessen Ausgabe rechts vom Quelltext aufgelistet ist. Selbstverständlich soll die Korrektheit nicht nur für 51, sondern für alle *int*-Werte erzielt werden.

```
class Bitset {
  ...
}

public class H15 {
  public static void main(String args[]) {
    Bitset s = new Bitset(51);
    for(s.start(); s.more(); s.step()) {
      for(int i=31; i ≥ 0; i--)
        System.out.print((s.subset()>>i)&1);
      System.out.println();
    }
  }
}
```

```
00000000000000000000000000000000
00000000000000000000000000000001
00000000000000000000000000000010
00000000000000000000000000000011
00000000000000000000000000010000
00000000000000000000000000010001
00000000000000000000000000010010
00000000000000000000000000010011
00000000000000000000000000100000
00000000000000000000000000100001
00000000000000000000000000100010
00000000000000000000000000100011
00000000000000000000000000110000
00000000000000000000000000110001
00000000000000000000000000110010
00000000000000000000000000110011
```

Zur Erläuterung: Die 51, binär 110011, repräsentiert eine vierelementige Menge, die das erste, zweite, fünfte und sechste Element des Universums enthält. Natürlich hat diese Menge genau $2^4 = 16$ Teilmengen, deren binäre Repräsentationen gerade die Ausgabe des Programmes bilden.

Zur Analyse wollen wir annehmen, daß ein *int* nicht zwangsweise auf 32 Bit beschränkt ist, sondern die variable Länge w hat. Implementieren Sie Ihre Klasse so, daß die folgenden amortisierten Laufzeiten eingehalten werden:

Bitset(int i)	$O(w)$
void start()	$O(w)$
int subset()	$O(1)$
boolean more()	$O(1)$
void step()	$O(1)$

Beweisen Sie diese Schranken!

Lösungsvorschlag:

Zuerst der Quelltext:

```
class Bitset {
  final static int W=32;
  private int s;
  private int[] b;
  private int[] counter;
  private int popcount;
  private int subset;
  public Bitset(int i) {
    s=i;
    b=new int[W+1];
  }
  public void start() {
    counter=new int[W+1];
    popcount=0;
    subset=0;
    for(int i=0; i<W; i++)
      if(((s>>i)&1) == 1) b[popcount++] = 1<<i;
  }
  public int subset() { return subset; }
  public void step() {
    int i=0;
    while(counter[i] == 1) {
      subset ^= b[i];
      counter[i++]=0;
    }
    subset ^= b[i];
    counter[i]=1;
  }
  public boolean more() {
    return counter[popcount] == 0;
  }
}

public class H15 {
  public static void main(String args[]) {
    Bitset s = new Bitset(51);
    for(s.start(); s.more(); s.step()) {
      for(int i=Bitset.W-1; i >= 0; i--) System.out.print((s.subset()>>i)&1);
      System.out.println();
    }
  }
}
```

(a) Wieso genügt die Klasse *Bitset* unseren Anforderungen?

Im folgenden verwenden wir Mengen und ihre Bitmap-Repräsentationen synonym. Die Grundidee besteht darin, für jedes in der gegebenen Menge S vorkommende Element eine Grundmenge bereitzustellen, die genau das betreffliche Element enthält. Aus diesen Grundmengen lassen sich dann bequem die jeweiligen Teilmengen herstellen.

Für unser Beispiel 110011 wären dies die vier Grundmengen

$$b[0] = 000001, b[1] = 000010, b[2] = 010000 \text{ und } b[3] = 100000.$$

Nimmt man das logische Oder über allen 2^4 Kombinationen dieser Grundmengen, so erhält man natürlich die 2^4 Teilmengen von S.

Die jeweiligen Kombinationen werden durch den Zähler *counter* repräsentiert. Dabei bedeutet $counter[i] = 1$, daß die Grundmenge $b[i]$ in der aktuellen Kombination vorkommen soll. Wenn wir den Zähler inkrementieren, müssen wir für jedes geflippte Bit die entsprechende Grundmenge aus der aktuellen Teilmenge heraus- bzw. wieder hereinnehmen. Dies läßt sich leicht mit einem exklusiven Oder erreichen.

Zur Erläuterung diene der Fall, daß der Zähler von 0011, der im obigen Beispiel die Teilmenge $b[0] + b[1] = 000011$ repräsentiert, auf 0100 inkrementiert wird, was der Teilmenge $b[2] = 010000$ entspräche. Geflippt wurden die letzten drei Stellen des Zählers, und $000011 \oplus b[0] \oplus b[1] \oplus b[2]$ liefert gerade den Wert 010000.

(b) Wieso gelten die gewünschten Laufzeitschranken?

Schwierig ist lediglich die Analyse der Methode *step*(). Im schlimmsten Fall wird jede Stelle des Zählers *counter* geflippt, was zu einer Laufzeit von $\Theta(w)$ führt. Hier muß also eine amortisierte Analyse durchgeführt werden. Diese ist deshalb erfolgversprechend, weil in den meisten Fällen nur sehr wenige Bits geflippt werden müssen (z.B. lediglich eins in der Hälfte der Fälle).

Es bezeichne $|counter|$ die Anzahl der Einsen im Zähler und $counter\gg$ die Anzahl der *trailing ones*, also die Länge der Einserkette am Ende des Bitstrings *counter*.

Die Potentialfunktion Φ definieren wir wie in der Fragestunde als $c|counter|$ für eine geeignete Konstance c. Die tatsächlichen Kosten eines Aufrufs von *step()* sind $c(counter\gg+1)$, wenn c richtig gewählt wurde. Somit ergeben sich die amortisierten Kosten von

$$c(counter\gg + 1) + c|counter'| - c|counter|,$$

wobei *counter'* den nächsten Zählerstand bezeichnet. Andererseits gilt

$$|counter'| = |counter| + 1 - counter\gg.$$

Durch Einsetzen erhalten wir für die amortisierte Kosten

$$c(counter\gg + 1) + c(|counter| + 1 - counter\gg) - c|counter| = c = O(1).$$