

# Datenstrukturen und effiziente Algorithmen

## Blatt 11

Markus Vieth, David Klopp, Christian Stricker

21. Januar 2016



## Aufgabe 1

### Python-Code

```
1  #!/usr/bin/env python
2  # coding: utf8

4  str = '' # Text der Seite hier einfügen
5  count = 0
6  for c in str:
7      o = ord(c)
8      if (o >= 97 and o <= 122) or (o >= 65 and o <= 90):
9          count += 1
10 print count
```

Die Anzahl der Buchstaben beträgt 2252.

## Aufgabe 2

### Pseudo-Code

```
1  for ( ; n>0; n--) {
2      insert(F, getMin(F)+1);
3      insert(F, getMin(F)-1);
4      insert(F, getMin(F)-2);
5      deleteMin(F);
6      x = min(F).childs[0]; // immer das größte Kind des Minimums
7      decreaseKey(x, getMin(F)-2);
8      deleteMin(F);
9  }
```

### Beschreibung

Füge drei Knoten in den Fibonacci-Heap ein. Der erste Knoten ist hierbei um 1 größer, der zweite um 1 kleiner und der dritte um 2 kleiner als das aktuelle Minimum. Lösche nun den Minimumknoten, aktualisiere ihn und setze den Wert des Größten Kindes des neuen Minimumknotens auf das aktuelle Minimum -2. Es ergibt sich somit ein neues Minimum, das in die Wurzelliste aufgenommen wird. Dieses wird nun abschließend gelöscht. In jedem Schritt wird so also genau ein Knoten in den Heap eingefügt. Es ergibt sich somit eine vertikale Liste.

## Aufgabe 3

a)

### Hilfsmethode

```
1  int left(int i) {
2      return 2i+1;
3  }

5  int right(int i) {
6      return 2i+2;
7  }

9  int parent(int i) {
10     return floor((i-1)/2);
11 }

13 // Heap-Eigenschaft herstellen
14 void heapify(Array b, int i) {
15     // kleinstes Element des Asts finden
16     int min = i
17     if (left(i) < b.size && b[left(i)].key < b[min].key)
18         min = left(i)
19     if (right(i) < b.size && b[right(i)].key < b[min].key)
20         min = right(i)

22     // wenn ein kleines Element existiert dann stelle die Heap-Eigenschaft her
23     if (min != i) {
24         b.swap(i, min)
25         // verfahren kaskadenartig weiter
26         heapify(b, min)
27     }
28 }
```

### decreaseKey

```
1  bool decrease(Array b, int i, newKey) {
2      // neuer Schlüssel muss kleiner sein als alter
3      if (newKey > b[i].key)
4          return false;

6      // Schlüssel setzten
7      b[i].key = newKey;

9      // Heap-Eigenschaft herstellen
10     while (i > 0 && b[i].key < b[parent(i)].key) {
11         b.swap(i, parent(i));
12         i = parent(i);
13     }
14     return true;
15 }
```

**deleteMin**

```

1  int deleteMin(Array b) {
2      if (i < b.size)
3          return -1; // Index out of bounds
4      min = b[0];
5      b.swap(i, -1); // sei -1 der Index des letzten Elements
6      b.size -= 1 // verringere die Array Größe um 1

8      // Heap-Eigenschaft herstellen
9      if ((b.size > 0))
10         heapify(b, i)
11     return min;
12 }

```

**insert**

```

1  void insert(Array b, newKey) {
2      int i = b.size
3      b.size += 1;
4      // setze b auf den höchst möglichen Wert
5      b.key = MAX_INT
6      // Update den Wert und die Heap-Eigenschaft mit decrease
7      decrease(b, i, newKey)
8  }

```

**b)**

Die Laufzeit für die decreaseKey Operation beträgt  $O(\text{Baumtiefe}) = O(\log(n))$ . Unter der Annahme, dass das Resizen des Arrays in konstanter Zeit realisiert werden kann, beträgt somit die Laufzeit für insert ebenfalls  $O(\log(n))$ . Ausschlaggebend für die deleteMin Operation ist die heapify Methode, da sich der Rest der Implementierung in konstanter Zeit realisieren lässt. Die Laufzeit für heapify und somit auch für deleteMin beträgt ebenfalls  $O(\text{Baumtiefe}) = O(\log(n))$ .

**c)**

Füge alle Elemente unsortiert in ein Array ein. Laufzeit:  $O(1)$ .

Wende nun Radix-Sort auf das Array an, um es zu sortieren. Laufzeit:  $O(n)$ . (Siehe hierzu VL 11)

Die Heap-Eigenschaft ist nun für alle Knoten erfüllt.

**Zusatz 1****a)**

**Behauptung**  $(E, M) = (\{a, b, c, d\}, \{\emptyset, \{a\}, \{b\}, \{c\}\{d\}\})$  ist ein Matroid.

**Beweis**

**zu zeigen: Nicht-Leerheit**

$$\emptyset \in M = \{\emptyset, \{a\}, \{b\}, \{c\}\{d\}\}$$

**zu zeigen: Vererbung** Die Teilmengen einer einelementigen Menge sind die Menge selbst und die leere Menge. Somit ist offensichtlich, dass auch diese Bedingung erfüllt ist.

**zu zeigen: Austausch** Sei  $X$  eine der einelementigen Mengen aus  $M$ . Das einzige Element mit geringerer Kardinalität ist die leere Menge.

$$\Rightarrow \exists x \in X \setminus \emptyset = X \mid \{x\} \cup \emptyset = X \in M$$

$\Rightarrow (E, M)$  ist ein Matroid.

**b)**

**Nich-Leerheit**  $M$  muss die leere Menge enthalten  $\Rightarrow M' := \{\emptyset, \{a, b\}, \{c, d\}\} \subseteq M$ .

**Vererbung** Sei  $\mathcal{P}(X)$  die Potenzmenge der Menge  $X$

$$M' \cup \mathcal{P}(\emptyset) \cup \mathcal{P}(\{a, b\}) \cup \mathcal{P}(\{c, d\}) = \{\emptyset, \{a, b\}, \{c, d\}, \{a\}, \{b\}, \{c\}, \{d\}\} =: M'' \subseteq M$$

**Austausch** Wie in Teilaufgabe a) erklärt, muss die leere Menge nicht weiter betrachtet werden, da mit dieser lediglich die einelementigen Mengen erzeugt werden können.

$$A = \{v\}, B = \{v, w\} \text{ mit } v, w \in E \quad B \setminus A = \{w\}, \{w\} \cup A = \{v, w\}$$

$$A = \{a\}, B = \{c, d\} \quad B \setminus A = \{c, d\} \Rightarrow \{c\} \cup \{a\} = \{a, c\} \in M \vee \{d\} \cup \{a\} = \{a, d\} \in M$$

$$A = \{b\}, B = \{c, d\} \quad B \setminus A = \{c, d\} \Rightarrow \{c\} \cup \{b\} = \{b, c\} \in M \vee \{d\} \cup \{b\} = \{b, d\} \in M$$

$$A = \{c\}, B = \{a, b\} \quad B \setminus A = \{a, b\} \Rightarrow \{a\} \cup \{c\} = \{a, c\} \in M \vee \{b\} \cup \{c\} = \{b, c\} \in M$$

$$A = \{d\}, B = \{a, b\} \quad B \setminus A = \{a, b\} \Rightarrow \{a\} \cup \{d\} = \{a, d\} \in M \vee \{b\} \cup \{d\} = \{b, d\} \in M$$

$$\Rightarrow M = \{\emptyset, \{a, b\}, \{c, d\}, \{a\}, \{b\}, \{c\}, \{d\}, \{a, c\}, \{b, d\}\} \vee M = \{\emptyset, \{a, b\}, \{c, d\}, \{a\}, \{b\}, \{c\}, \{d\}, \{a, d\}, \{b, c\}\}$$

$\Rightarrow (E, M)$  ist der kleinste Matroid, welcher den Anforderungen entspricht.

q.e.d.

## Zusatz 2

a)

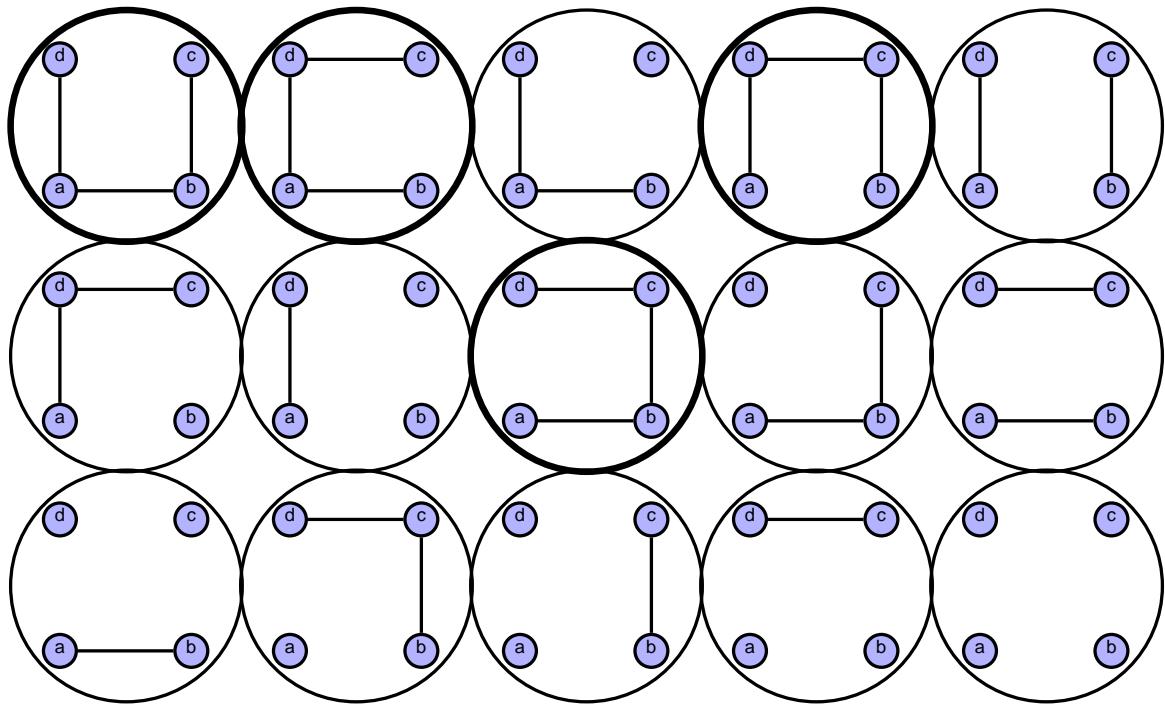


Abbildung 1: graphischer Matroid

Die Teilmengen größter Kardinalität sind  $M_1 := \{(a, b), (b, c), (c, d)\}$ ,  $M_2 := \{(a, b), (b, c), (d, a)\}$ ,  $M_3 := \{(a, b), (c, d), (d, a)\}$ ,  $M_4 := \{(b, c), (c, d), (d, a)\}$ . Es handelt sich dabei um die möglichen Spannbäume des Graphen  $G$ .

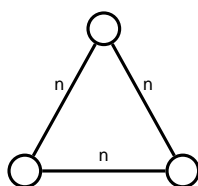
b)

Es handelt sich um den Algorithmus von Kruskal. Der Algorithmus gibt  $B = \{(a, b), (b, c), (c, d)\}$  aus. Die Abfrage verhindert das Bilden von Zyklen.

c)

Das Sortieren der Kanten hat im Allgemeinen (also ohne Radix-Sort) eine Laufzeit in  $\mathcal{O}(|E| \log |E|)$ . Die Schleife läuft über die Menge aller Kanten, also  $|E|$  mal. Innerhalb der Schleife gibt es die Abfrage mit der Laufzeit  $f(|E|)$  und die Einfüge-Operation, welche mit passender Datenstruktur in konstanter Zeit ausgeführt werden kann. Somit ergibt sich eine Laufzeit in  $\mathcal{O}(|E| \log |E| + |E|f(|E|))$ .

d)



Nein, der Algorithmus ist nicht eindeutig. Man denke sich einen dreieckigen Graphen mit den Kanten  $E := \{(a, b), (b, c), (c, a)\}$  welche ein einheitliches Kantengewicht aufweisen. Das Ergebnis ist nun vom Sortiervorgang abhängig. So liefern allein stabile und instabile Algorithmen verschiedene Ergebnisse. Mögliche Lösungen wären  $L_1 := \{(a, b), (b, c)\}$ ,  $L_2 := \{(a, b), (c, a)\}$ ,  $L_3 := \{(b, c), (c, a)\}$

Abbildung 2: Sei  $n$  eine beliebigen Länge