Datenstrukturen und effiziente Algorithmen
Blatt 11

Markus Vieth, David Klopp, Christian Stricker

19. Januar 2016

Aufgabe 1

Python-Code

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

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

Die Anzahl der Buchstaben beträgt 2252.

Aufgabe 2

Pseudo-Code

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

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 setzte 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.

Markus Vieth, David Klopp, Christian Stricker

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     // verfahre 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    // setzte 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 $O(Baumtiefe) = O(\log(n))$. 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 $O(Baumtiefe) = O(\log(n))$.

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.