Algorithmen und Datenstrukturen, Übung 5

Marouane Soussi, Lars Happel, Mustafa Miresh

Mai 2022

Aufgabe 1

siehe .java Datei

Aufgabe 2

a)

1. A= [3, x, y, 1, 1, 1, 2], A.heapsize = 7
   $x = \{1, 2, 3\}, y = \{2, 3\}$. Da beide Kinder von 3 sind, außerdem müssen auch beide größer als ihren Kindern.

2. A = [40, 39, 37, x, 36, 38, y], Hier ist die Heap Eigenschaft nicht erfüllt da $38 > 37$. Obwohl 38 das Leftkind von 37 ist. Somit können wir nicht sagen, was x und y sein könnten. Hätte man 37 mit 38 getauscht, wäre dann :
   $x = \{0, .., 39\}, y = \{0, .., 38\}$

3. A = [91, 79, 20, x, 70, y], A.heapsize = 5
   $x = \{0, .., 79\}, y = \{0, .., \infty\}$. Da y Kein Heapelement ist.

4. A = [x, y, 10, 1, 7, 6, 10, 1], A.heapsize = 7
   $y = \{7, .., x\}, x = \{10, .., \infty\}$. Wir setzen, dass x ab 10 beginnen darf, dann hängt y von x ab.

b) Find(A, i, x)

Input: Ein Array A das einen Max-Heap darstellt, ein Index i und eine Zahl x
Output: True falls A die Zahl x in einem Unterheap von i enthält, False falls nicht.

```
Find(A, i, x)
if (A[i] = x) return true
if (A[i] < x) return false
left := i*2
```

```
right := i*2+1
if (left <= A.size && Find(A, left, x) = true): return true
else if (right <= A.size): return (Find(A, right, x)
else return false
```

Aufgabe 3

a)

Sei A=[100, 99, 80, 95, 98, 70, 79, 91, 92, 93, 94, 69, 68, 70, 68, 90]
Sei z.B. $i = 7$, also $A[i] = 79$. Dann ist nach dem Tausch von A[i] mit A[A.heapsize] A[i] = 90. Allerdings ist der Vaterknoten davon (A[3] = 80) und somit $A[i] > A[3]$ also die Heapeigenschaft an einer Stelle verletzt, die durch Max-Heapify nicht korrigiert wird.
In diesem Fall funktioniert Heap-Delete nicht wie erwartet, da nach Austausch von A[i] und A[A.heapsize] der am Index i verwurzelte Heap zwar ein Max-Heap ist, jedoch die Heap-Eigenschaft dadurch verletzt wird, dass A[i] nun größer ist als sein Vaterknoten. Diese Verletzung wird durch Max-Heapify nicht behoben, da es nur Kinderknoten von i berührt.

b)

Input: Ein Heap und ein Index der gelöscht werden soll
Output: Ein Max Heap ohne das zu löschende Element

```
Heap-Delete(A, i)
  Increase-Key(A, i, infinity)
  A[1] = A[A.heapsize]
  A.heapsize =- 1
  Max-Heapify(A,1)
```

Dieser Algorithmus bedient sich der bereits aus den Folien bekannten Methoden Increase-Key und Max-Heapify, von denen wir bereits wissen, dass sie korrekt funktionieren. Zuerst wird das zu löschende Element durch Increase-Key in einen Sentinel umgewandelt, welcher beim Wiederherstellen der Heapeigenschaft bis zur Wurzel des Gesamtheap befördert wird. Es kann also kein Element darüber geben, welches die Heapeigenschaft verletzt.
Anschließend wird die Wurzel entfernt entsprechend dem Vorgehen bei "Extract-Max". Die beim löschen der Wurzel verletzte Heapeigenschaft kann dann vollständig wiederhergestellt werden, da das Wiederherstellen an der Wurzel des Gesamtheaps beginnt. Der Algorithmus hat $O(log(n))$ Laufzeit, da die Operationen Increase-Key und Max-Heapify je $O(log(n))$ Laufzeit haben.

Aufgabe 4

Input: Eine Linkedlist
Output: True wenn die Linkedlist einen Zyklus enthält, False sonst

```
Findcircles(L)
i := L.head
j := L.head
while (true):
  for (k = 0 to 3):
    if (i.next = NIL): return false
    else: i := i.next
    if (i = j) return true
  j := j.next
```

Idee: Einen "schnellen" Index (i) und einen "langsamen" Index (j) verwenden um einen Zyklus zu entdecken. Angenommen die Liste enthält keinen Zyklus, dann wird i.next irgendwann NIL und der Algorithmus terminiert mit False. Angenommen die Liste enthält einen Zyklus, dann wird die Bedingung, dass i.next = NIL ist nie erreicht. Jedoch werden beide Zeiger, i und j, irgendwann in den Loop hineinlaufen. Da i sich schneller bewegt als j, wird der Index i den Index j im Loop irgendwann "überrunden" und an diesem Zeitpunkt wird (i = j) true und der Algorithmus terminiert mit true.