Data Structures and Algorithms I

Homework Assignment 4 - Halden

Wachmann Elias

8. November 2021

1 Aufgabenstellung

A d-ary heap is like a binary heap, but nonleaf nodes have d children instead of 2 children.

(A) How would you represent a d-ary heap in a array?

(B) What is the height of a d-ary heap of n elements in terms of n and d? Justify your answer.

(C) Give an efficient implementation of `HEAPIFY` in a d-ary max-heap. Analyze its running time in terms of d and n.

2 A - Darstellung in einem Array

Gleich wie bei einer binären Halde wird der Maximale wert an den Anfang des Arrays (im folgendem `A`), also `A[0]` gespeichert. In einer d-äre Halde hat nun jeder Knoten (außer leafs) d Kinder. Diese d Kinder werden nun an die nächsten d Stellen im eindimensionalen Array gespeichert. Damit sind die Kinder der Wurzel `A[0]` im Array an den Stellen `A[1]` bis `A[d]`. Die Nächste Ebene im Baum hat $d \cdot d$ Kinder, welche im Array von `A[d+1]` bis `A[d`^{2}`+d]`. Abermals die nächste von `A[d`^{2}`+d+1]` bis `A[d`^{3}`+d`^{2}`+d]` Allgemein liegt die n-te Ebene (Wurzel ist 0. Ebene) von `A[`$\sum_{i=0}^{n}$`d`^{i}`]` bis `A[(`$\sum_{i=0}^{n+1}$`d`^{i}`)-1]` im 0-indiziertem Array.

3 B - Höhe einer *d*-ären Halde

Wie schon in Abschnitt 2 ersichtlich ist, wächst eine d-ären Halde um h^d - Kinder wobei h die Ebene/Höhe angibt. Die Anzahl n der in einer Halde gespeicherten Elemente liegt nun sicherlich wie folgt:

$$1 + \sum_{i=0}^{h-1} d^i \leq n \leq \sum_{i=0}^{h} d^i$$
$$1 + \frac{d^h - 1}{d-1} \leq n \leq \frac{d^{h+1} - 1}{d-1}$$

Die linke Seite $\frac{d^h+d-2}{d-1}$ wird wie folgt abgeschätzt:

$$\frac{d^h + d - 2}{d-1} > \frac{d^h + d - 2}{d-1} - 1 = \frac{d^h - 1}{d-1}$$

Nimmt man nun davon $\log_d$ und formt beide Seiten jeweils auf h um erhält man:

$$\log_d(dn - n + 1) - 1 \leq h < \log_d(dn - n + 1)$$
$$\implies \lceil \log_d(dn - n + 1) - 1 \rceil = h$$

Daraus folgt nun schließlich, dass die von der Ordnung

$$h = \Omega(\log_d(dn)) = \Omega(\log_d(n))$$

ist. d ist für eine d-ären Halde konstant und kann in der $\mathcal{O}$-Notation deshalb weggelassen werden.

4 C - Implementation von `HEAPIFY`

Es folgt eine Implementation von `HEAPIFY`.

Algorithm 1 `HEAPIFY`

```
1:  function HEAPIFY(A,i,d)
2:      // Input: A array to heapify
3:      // i index which should be heapified
4:      // d order of d-ary heap
5:      n ← length of A
6:      index ← i
7:      for count ← 1 to d+1 do                          ▷ including d+1
8:          kid ← d*i+count
9:          if kid < n and A[kid]] > A[index] then
10:             index ← k
11:     if i does not equal index then
12:         switch A[index] and A[i]
13:         HEAPIFY(A,index,d)
```

4.1 Laufzeitanalyse von `HEAPIFY`

Die Laufzeit von `HEAPIFY` ergibt sich nun wie folgt: Die For-Schleife im Algorithmus wird beim Aufruf d mal ausgeführt, somit $\mathcal{O}(d)$. Innerhalb der For-Schleife erfolgen die Zuweisungen und vergleiche in $\mathcal{O}(1)$ Zeit. Auch die Zuweisungen vor und der Vergleich sowie der „switch“ nach der Schleife erfolgen in konstanter Zeit. Der rekursive Funktionsaufruf wird maximal so oft aufgerufen, wie die Halde tief ist, also $\mathcal{O}(log_d(n))$. Dadurch ergibt sich insgesamt:

$$T(d,n) = \mathcal{O}(d * log_d(n))$$

5 Bonus - Laufzeit Plot

Der Algorithmus wurde wie folgt in Python implementiert:

```
def heapify(A, i, d):
    n = len(A)
    index = i
    for j in range(1, d+1):
        k = d*i+j
        if k < n and A[k] > A[index]:
            index = k
    if i != index:
        temp = A[index]
        A[index] = A[i]
        A[i] = temp
        heapify(A, index, d)
```