danish

Handin 3

Thomas Vinther & Jens Kristian Refsgaard Nielsen

21-09-18

1 Analysis of dairy heaps

A d-ary heap is like a binary heap, but (with one possible exception) non-leaf nodes have d children instead of 2 children. In the following we assume that A=A[1],...,A[n] is an array of length n>0.

1.1 Subtask a.)

How would you represent a d-ary heap in an array
We represent the d-ary heap as an array by maintaining the heap property $A[Parent(i)] \geq A[i]$ but we change the Parent function as follows
Parent(i)

Time	Line nr	Code
1	1	if i = 1
1	2	return 0
1	3	if i < d+2
1	4	return 1
1	5	m = 2
n/d	6	while i > md+1
1	7	m++
1	8	return m

This amounts to the function

$$\text{Parent}(i) = \begin{cases} 0 & \text{if } i = 1 \\ 1 & \text{if } 1 < i < d+2 \\ m & \text{if } i = (m-1)d+2, \dots, md+1 \end{cases} \tag{1.1}$$

Correctness: It is clear that the algorithm works for $i < d+2$. Now if $i > d+2$ we set $m = 2$ and as long as $i > md+1$ we count up m, the first time the while loop doesnt activate we have $i < md+1$ but we saw that $i > kd+1$ for all $k \in [2, m)$, especially for $k = m-1$ so we get the inequality $(m-1)d+2 \leq i \leq md+1$ which is what we wanted. $\square$
With this version of the parent function we get an array where $A[1]$ is the root and
$A[2], \dots, A[d+1]$ are its children. Now the d elements $A[d+2], \dots, A[2d+1]$ become the children of $A[2]$ and so on.
This algorithm has worst case runtime $O(n/d)$, we will later see the importance of keeping the n/d notation.

1.2 Subtask b.)

What is the height of a d-ary jeap of n elements in terms of n and d
The zeroth level of a d-ary tree has 1 element, this element has d children, and each of these in turn have d children, so in total we end up having d^h nodes at the h'th level. So in total for a full d-ary tree we get

$$\begin{aligned} n = \sum_{i=0}^{h} d^i = \frac{d^{h+1}-1}{d-1} \implies \\ n(d-1)+1 = d^{h+1} \\ \log_d(n(d-1)+1) = \log_d(d^{h+1}) = h+1 \\ \log_d(n(d-1)+1)-1 = h \end{aligned}$$

Now in the worst case we work with an incomplete tree with only 1 leaf at the h'th level

$$\begin{aligned} n = 1 + \sum_{i=0}^{h-1} d^i = 1 + \frac{d^h-1}{d-1} \implies \\ (n-1)(d-1)+1 = d^h \\ \log_d((n-1)(d-1)+1) = \log_d(d^h) = h \\ \log_d((n-1)(d-1)+1) = h \end{aligned}$$

We have found an upper and lower bound for our true height h

$$O(\log_d(n)) \ni \log_d((n-1)(d-1)+1) \geq h \geq \log_d(n(d-1)+1)-1 \in \Omega(\log_d(n)) \tag{1.2}$$

This shows that $h = \Theta(\log_d(n))$

1.3 Subtask c.)

Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyse its running time in terms of d and n.
As HEAP-EXTRACT-MAX does not in itself have anything to do with the -aryity of the tree no modification is necessary. However HEAP-EXTRACT-MAX calls MAX-HEAPIFY(A,1), which we indeed need to modify to MAX-HEAPIFY'(A,m):

Time	Line nr	Code
1	1	largest = m
d	2	kids = [(m-1)d+2,(m-1)d+3,. . . ,dm+1]
d	3	for k in kids
2	4	if k $\leq$A.heapsize and A[k] > A[largest]
1	5	largest = k
1	6	if largest $\neq$ m
3	7	exchange A[m] with A[largest]
a	8	MAX-HEAPIFY'(A,largest)

Correctness: In the usual version of MAX-HEAPIFY(A,m) we only need to compare to the left and right children of a given node, however in the dairy case the Parent function is a bit more complicated, to counteract this added complexity we introduce the kids array, which clearly is the index of the children to the m'th element of our starting Arraylist. We use a modified version of the find best algorithm, thanks Kurt, to find the index of the largest child, or the element we started with. If the largest of the kids is larger than the input(parent) we switch them around and check if the former parent should move further down the tree with another call of the MAX-HEAPIFY' algorithm. If however none of the kids are larger than their parent we simply do nothing, and since we call HEAP-EXTRACT-MAX on an already built max-heap the remaining elements all satisfy the max-heap condition, and in line 3-5 we checked that the new Parent also satisfies the max-heap condition with all of its children. $\square$
Next consider the runtime, line 1 through 7 takes $1+d+d(2(1))+1(3) = O(d)$ so the call in line 8 is also $a = O(d)$. In the worst case we have to MAX-HEAPIFY' once for each layer of the tree, and we've seen that this was $\log_d(n)$ so we get a total runtime of $O(d\log_d(n)) = O(\log_d(n))$.

1.4 Subtask d.) & e.)

We wish to modify MAX-HEAP-INSERT to work on our d-ary trees.
The base kit for MAX-HEAP-INSERT(A,key) will work without modification. However the HEAP-INCREASE-KEY(A,i,key) will need modification as follows

Time	Line nr	Code
1	1	if key < A[i]
1	2	error
1	3	A[i] = key
$\log_d(n)$+n/d	4	while i > 1 and A[Parent(i)]<A[i]
3+n/d	5	exchange A[i] with A[Parent(i)]
1+n/d	6	i = Parent(i)

The n/d is from the Parent function. In total we get

$$\begin{aligned}\sum_{j=0}^{\log_d(n)} \frac{n}{d^j} = n \sum_{j=0}^{\log_d(n)} \frac{1}{d^j} = \sum_{j=0}^{\log_d(n)} \left(\frac{1}{d}\right)^j \\ = n\left(\frac{(\frac{1}{d})^{\log_d(n)} - 1}{\frac{1}{d} - 1}\right) \\ = n\left(\frac{(\frac{1}{n}) - 1}{\frac{1}{d} - 1}\right) \\ = \left(\frac{(1 - n}{\frac{1-d}{d}}\right) \\ = (1-n)\frac{d}{1-d} \\ = \frac{d(n-1)}{d-1}\end{aligned}$$

Here the $\frac{n}{d^j}$ summands represent the Parent function being called j times on n, because the parent funciton pretty much just represents division by d. And we call it $\log_d(n)$ times. In total the new version runs in linear time.

Correctness: The correctness of this algorithm depends entirely on the correctness of the Parent function. Since we start out with a legal max-heap, we can safely assume that if A[Parent(i)]<A[i] we can exchange A[Parent(i)] with A[i] and maintain the heap structure, because A[i]>A[Parent(i)]≥A[k] for k∈kids(i)=[(i-1)d+2,...,id+1] as wanted. If however A[Parent(i)]≥A[i] we do nothing, which in this case preserves the heap structure □