d-ary Heap Analysis

Or Haifler

January 2024

1 Representing a d-ary Heap in an array

The representation of a d-ary Heap in an array is fairly similar to the representation of a binary(2-ary) Heap in an array. When using an array to represent a binary heap, we're inserting the root, letting it be the element corresponding to the index 0, and then inserting each node, in a traversal from left to right(e.g in the second level, if the nodes are 1,2,3, we'll insert them by this order). We can do the exact same thing in the case of a general d-ary Heap, and the only question left to ask is how can we access the childs indices of a node, and how can we access the parent index of a particular node.

Parents and Children indices

At first, we're going to calculate the child's indices of a particular node by its index. I'm assuming that we're using a 0-indexed array, if not, only a minor change is required. Suppose we got a d-ary heap, and its array is A, and additionally we got a node with the index i. If we'll want to access the childs of this node, the indices will be $\{i \cdot d + j, 1 \leq j \leq d\}$, one can check that it is indeed the case pretty easily. Now, the trickier part is how one could calculate the parent's index from his childs indicies, in general, the method that is being used in the case of $d = 2$ will not work with $d > 3$. We can see that if we have an index i, his children are $i \cdot d + 1, i \cdot d + 2 ... i \cdot d + d$, thus, if we denote the index of the j'th child of the i'th node by $S_{i,j}$, we got $\frac{S_{i,j}}{d} = \frac{i \cdot d + j}{d} = i + \frac{j}{d}$, so $i < \frac{S_{i,j}}{d} \leq i + 1$, and if we'll take the floor of this value, we'll see that every child except the last one is being mapped to i, but the last one is being mapped to $i + 1$. Fortunately, we could fix that by subtracting one from the numerator, and we'll get $\frac{S_{i,j}-1}{d} = \frac{i \cdot d + j - 1}{d} < \frac{i \cdot d + d}{d} = i + 1$, and we're in the safe zone. Finally, we got our mappings from children to parent and vice versa.

$$\text{Parent}(i, d) = \lfloor \frac{i-1}{d} \rfloor$$
$$\text{Child}(i, j, d) = i \cdot d + j$$

Height of a d-ary Heap

Suppose that we got a d-ary Heap H with n nodes, what's its height? Firstly, we know that if its height is k, then every level, except maybe the k'th one is full. Each level H_l, for $l < k$, contains exactly d^l nodes, so we got a lower bound $\sum_{l=0}^{k-1} d^l$, but more than that, we know that the k'th level contains at least one node, so we got $1 + \sum_{l=0}^{k-1} d^l \leq n$. For the upper bound, we know that the last level contains at most d^k nodes, so we got our final bounds $1 + \sum_{l=0}^{k-1} d^l \leq n \leq \sum_{l=0}^{k} d^l$, with geometric series sum formula, we get $1 + \frac{d^k - 1}{d-1} \leq n \leq \frac{d^{k+1}-1}{d-1}$, because n is a natural number, we could write this equivalently as

$$\frac{d^k - 1}{d-1} < n \leq \frac{d^{k+1} - 1}{d-1} \implies d^k < (d-1)n + 1 \leq d^{k+1} \implies k < \log_d((d-1)n+1) \leq k+1$$
$$k + 1 = \lceil \log_d((d-1)n+1) \rceil \implies k = \lceil \log_d((d-1)n+1) \rceil - 1$$

2 Max Heapifying a d-ary Heap

The pseudo code of the general d-ary Heap Max-Heapify procedure is similar to the binary Heap one, we can Max-Heapify a d-ary Heap, at index i, with the next procedure:

Algorithm 1 Max-Heapify(A,i)

```
1: if i ≥ Heap-Size(A) then
2:     return
3: end if
4: max ← i
5: for l ← i · d + 1 to i · d + d do
6:     if A[l] > A[max] then
7:         max ← l
8:     end if
9: end for
10: if i ≠ max then
11:     A[i] ⟷ A[max]
12:     Max-Heapify(A,max)
13: end if
```

One can see that this procedure is exactly the same as the binary Heap Max-Heapify procedure, except here we're checking each child, and searching for the maximum child(In the case of a binary Heap, we're just checking the left and right child, so we don't need a for loop). To show that Max-Heapify works correctly, one can use the exact same loop invariant that's being used in the case of a binary Heap.

3 Building a d-ary Heap

As in the Max-Heapify procedure, Build-Max-Heap for a d-ary Heap stays pretty much the same, except that in the case of a general d-ary Heap, the leafs start at some index, and we need to find it. After finding the leafs starting index, we could write a similar procedure to the binary case. For finding the leafs starting index, we could use some simple logic. We know that a node is leaf if and only if it doesn't have any children, so suppose that we're looking at the i'th node in the array, and our Heap size is H_{size}, then the node's first children are placed in the $i \cdot d + 1$ position, so we know that a node with index i is a leaf if and only if $i \cdot d + 1 \geq H_{\text{size}}$, or equivalently $i \geq \frac{H_{\text{size}}-1}{d}$, and thus for $i = \lfloor \frac{H_{\text{size}}-1}{d} \rfloor$, i will be the first leaf in our Heap.

Algorithm 2 Build-Heap(A,n)

```
1: Heap-Size(A) ← n
2: for i ← ⌊(H_size − 1)/d⌋ to 0 do
3:     Max-Heapify(A,i)
4: end for
```

4 Extracting The Maximum Value from a d-ary Heap

This one is pretty simple, from the Heap property, the maximum value will always be stored in the root node, and thus we could just extract it, give it the value of the last node, and then subtract the Heap size by one and call Max-Heapify on the root node.

Algorithm 3 Extract-Max(A)

1: max $\leftarrow A[0]$
2: $A[0] \leftarrow A[\text{Heap-Size}(A) - 1]$
3: Heap-Size(A) = Heap-Size$(A) - 1$
4: Max-Heapify(A,0)
5: **return** max

5 Increasing a Key in a d-ary Heap

This one is also similar to the binary case, we'll change the node's value to the maximum between its current value and the key value, and then "bubble up" until the current node value isn't greater then its parent's value

Algorithm 4 Increase-Key(A,i,key)

1: $[i] \leftarrow \max(A[i], key)$
2: **while** $i > 0$ and $A[i] > A[\text{Parent}(i)]$ **do**
3: $\quad A[i] \leftrightarrow A[\text{Parent}(i)]$
4: $\quad i \leftarrow \text{Parent}(i)$
5: **end while**

6 Inserting a New Value Into a d-ary Heap

The insertion procedure is going to be based on the correctness of the Increase-Key procedure, we're going to add one entry to our Heap, put minus infinity there as a value, and then use Increase-Key.

Algorithm 5 Insert(A,key)

1: Heap-Size(A) = Heap-Size(A) + 1
2: $A[\text{Heap-Size(A)} - 1] = -\infty$
3: Increase-Key(A,Heap-Size(A) - 1, key)

7 Complexity Analysis

Now we're going to calculate the Time Complexity of each procedure, by the same order that they're written in this file. All of them are pretty much alike the analysis of the binary case. I'm going to denote $n =$ Heap-Size(A)

Max-Heapify

In the Max-Heapify procedure, lines 1-4 takes only constant time, and the for loop + recursion call is the stuff that "dominates" the time complexity. We can see that the for loop takes exactly d checks, and then Max-Heapify is being called on the sub-Heap. We got the recurrence relation $T(n) = T(\lfloor \frac{n}{d} \rfloor) + \Theta(d)$, suppose that $n = d^r$, then we got

$$T(n) = T(\frac{n}{d}) + \Theta(d) = T(\frac{n}{d^2}) + \Theta(d) + \Theta(d) = ... = \sum_{j=1}^{r} \Theta(d) = \Theta(d \log_d(n))$$

Build-Heap

Firstly, we can get a nice upper bound for the Build-Heap procedure by observing that each call to $Max - Heapify$ takes $O(d \log_d(n)$ time, and there are $O(n)$ calls, so we got $O(dn \log_d(n)$. But like in the case of a binary heap, we could observe that most of the node's levels are low, the height of the Heap is $\lceil \log_d((d-1)n+1) \rceil - 1$, and there is at most $\left\lceil \frac{n}{d^{h+1}} \right\rceil$ nodes of length h. We know that the time complexity of a node with height h is $O(dh)$, and so

$$\sum_{h=0}^{\lceil \log_d(n) \rceil} \left\lceil \frac{n}{d^{h+1}} \right\rceil O(dh) = O\Big(dn \sum_{h=0}^{\lceil \log_d(n) \rceil} \frac{h}{d^h}\Big)$$

We know that $\sum_{h=0}^{\infty} \frac{h}{d^h} = \frac{\frac{1}{d}}{(1-\frac{1}{d})^2} = \frac{d}{(d-1)^2} < C$, so, finally, we got

$$O\Big(dn \sum_{h=0}^{\lceil \log_d(n) \rceil} \frac{h}{d^h}\Big) = O\Big(dn \sum_{h=0}^{\infty} \frac{h}{d^h}\Big) = O(dn)$$

8 Extract-Max, Increase-Key, Insert

This one is easy, all of our complexities have already been calculated. For the Extract-Max procedure, we know that the only "dominant" call here is to Max-Heapify, and we've already seen that the cost of Max-Heapify is $d \log_d(n)$, and so the time complexity of Extract-Max is the same. For the Increase-Key procedure, each call to the while loop executes only a constant time operations, and the while loops can be called at most $\log_d(n)$ times, so this is the procedures time complexity. The Insert procedure executes only two constant time operations, and then a call to Increase-Key, thus the time complexity is the same as Increase-Key.