Heaps

Eklavya Sharma

Contents

1 Almost-complete trees

1.1 Definition

A d-ary almost-complete tree is a rooted tree such that:

- Every vertex has at most d children.
- The tree is completely filled on all levels, except possibly the lowest, which is filled from the left up to a point.

Unless d is explicitly specific, it is taken to be 2 (binary tree).

The size of an almost-complete tree is the number of vertices in it.

1.2 Indexing

Every vertex in an almost-complete tree of size n can be indexed by a unique integer from 0 to $n-1$ by using a left-to-right level-ordered indexing.

Index 0 is the root. The vertex at index i has its parent at index $\left\lfloor \frac{i-1}{d} \right\rfloor = \left\lceil \frac{i}{d} \right\rceil - 1$. This means that the vertex at i has $di + j$ as a child for all $1 \leq j \leq d$, except if $di + j \geq n$, in which case i does not have a j^{th} child.

1.3 Height

Let there be a d-ary almost-complete tree of size n and height h. Let x be the number of nodes in the last level. Therefore, $1 \leq x \leq d^h$.

$$n = 1 + d + d^2 + \ldots + d^{h-1} + x = \frac{d^h - 1}{d - 1} + x$$

Theorem 1. $h = \lfloor \log_d(n(d-1)) \rfloor$

2 Heaps

2.1 Definition

Let T be a rooted tree where every vertex has a value associated with it. A vertex $v \in T$ is said to follow the max-heap property iff its value is greater than or equal to the value of all its children.

A max-heap is a rooted tree where every vertex follows the max-heap property.

Similarly, in a min-heap, the value of a vertex is always less than or equal to the value of all of its children.

Heaps typically use almost-complete trees. This makes it possible to specify the vertex values as an array A such that $A[i]$ is the value of the vertex with index i.

2.2 Max-heapify

The procedure `max_heapify`(A, i, n) takes as input an array A of size n representing a d-ary almost-complete tree and an integer $0 \leq i < n$ where the subtrees rooted at children of vertex i are max-heaps. `max_heapify` makes changes to A to make the subtree rooted at i a max-heap.

If i is a leaf node, `max_heapify` does nothing. Otherwise it finds out the largest child of i. Let it be j. If the value of j is greater than the value of i, it swaps the values of i and j and then calls `max_heapify`(A, j, n).

The time taken by `max_heapify` is proportional to the number of comparisons performed.

Theorem 2. `max_heapify` *performs at most* $dh(A, i, n)$ *comparisons, where* $h(A, i, n)$ *is the height of vertex* i *(i.e. the length of the longest path from* i *to a leaf).*

2.3 Building a max heap

The procedure `build_max_heap`(A, i, n) takes an array A of size n and modifies it so that the subtree rooted at i is a heap. It does this by calling `build_max_heap`(A, j, n) for every child j and then calling `max_heapify`(A, i, n).

Therefore, time taken by `build_max_heap` is proportional to the number of comparisons and the number of comparisons is upper bounded by dH, where H is the sum of heights of all vertices.

Let the heap have size n, height h and x nodes in the last layer.

There are $h+1$ layers in the heap. If the last layer is removed, there would be d^i vertices at depth i and they would all have height $h-1-i$. Adding back the last layer will increase by 1 the height of all ancestors of the vertices in the last layer. Therefore,

$$H = \underbrace{\left(\sum_{i=0}^{h-1}(h-1-i)d^i\right)}_{H_1} + \underbrace{\left(\sum_{i=1}^{h}\left\lceil\frac{x}{2^i}\right\rceil\right)}_{H_2}$$

$$(d-1)H_1 = \frac{d^h-1}{d-1} - h = n - x - h$$

$$H_2 = \sum_{i=1}^{h}\left\lceil\frac{x}{d^i}\right\rceil \geq \sum_{i=1}^{h}\frac{x}{d^i} = \frac{x(1-d^{-h})}{d-1} \geq \frac{x-1}{d-1}$$

$$H_2 = \sum_{i=1}^{h}\left\lceil\frac{x}{d^i}\right\rceil = \sum_{i=1}^{h}\left(1 + \left\lfloor\frac{x-1}{d^i}\right\rfloor\right)$$
$$\leq h + \sum_{i=1}^{h}\frac{x-1}{d^i} = h + \frac{(x-1)(1-d^{-h})}{d-1} \leq h + \frac{x-1}{d-1}$$

$$H \in \frac{(n-1) + h[-1, d-2]}{d-1}$$

Therefore, number of comparisons to build d-ary max-heap

$$\leq dH \leq \left(1 + \frac{1}{d-1}\right)(n-1) + \left(d - 1 - \frac{1}{d-1}\right)\left\lfloor \frac{\lg n + \lg(d-1)}{\lg d} \right\rfloor$$

Number of comparisons to build a binary max-heap $\leq 2(n-1)$.

2.4 Heapsort

Algorithm 1 `heapsort`(A, n)

```
build_max_heap(A, 0, n).
for i from n to 2 do
    Swap A[0] and A[i − 1].
    max_heapify(A, 0, i − 1).
end for
```

Number of comparisons

$$\leq dH + \sum_{i=1}^{n-1} d\,\lfloor \log_d(i(d-1)) \rfloor \leq dH + \frac{d}{\lg d}\left(\lg((n-1)!) + (n-1)\lg(d-1)\right)$$

For large n, this is minimized at $d = 3$.

With a binary heap, total number of comparisons for heapsort

$$\leq 2(n - 1 + \lfloor \lg((n-1)!) \rfloor) \leq 2n \lg n - 2(\lg e - 1)n - \lg n + \lg \pi - \frac{5}{6}$$

2.5 Push

To push a value into a max-heap array A, append it to the end of the array and then keep swapping with parent while it's greater than parent.

Time taken is proportional to number of comparisons. Number of comparisons is upper bounded by height of new tree $= \lfloor \lg_d((n+1)(d-1)) \rfloor$.

2.6 Pop

To pop a value from a max-heap of size n, swap $A[0]$ and $A[n-1]$ and call `max_heapify`$(A, 0, n-1)$.

Time taken is proportional to number of comparisons. Number of comparisons is upper bounded by $d\,\lfloor \lg_d((n-1)(d-1)) \rfloor$.