Heaps, Priority Queues and Graphs

Jan 29

1. Priority Queues:

1. Background:

A queue is an ADT that stores its items in First-In, First-Out (FIFO) order. A priority queue is an extension of a queue with the following properties:

1. Every item has a priority associated with it.
2. An element with a higher priority is removed before an element with a lower priority.
3. If two elements have the same priority, they are served based on their order.

2. Operations:

1. insert(p, j): Insert job j with priority p.
2. max(): read the job with the highest priority.
3. extract-max(): Read and remove the job with the highest priority.
4. increase-priority(j, p'): Increase the priority of job j to p'

2. Heaps:

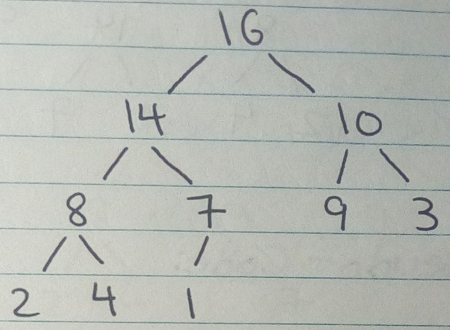
1. Background:

A heap is one way to store a priority queue.

A heap is:
1. A binary tree
2. Every level i has 2^i nodes, except for the bottom level
3. The bottom fills in left to right
4. At each node, its priority is greater than or equal to both of its children's priorities.

2. Insert:

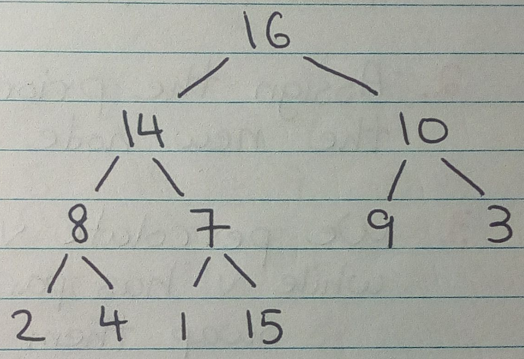
Suppose we have this heap.



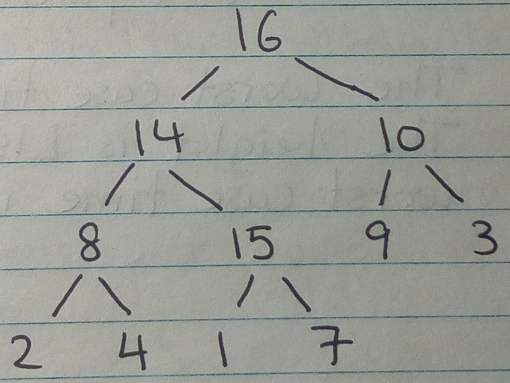
Note: Only priorities shown

If we insert a key with priority 15, then this happens:

1.

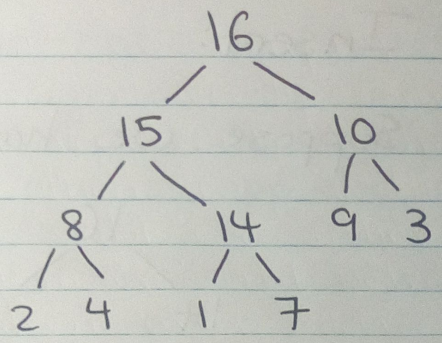


2.



Hilroy

3.



Pseudo-Code:

1. Create a new leaf at the bottom level, leftmost open location. This maintains the requirement that the tree is nearly-complete.

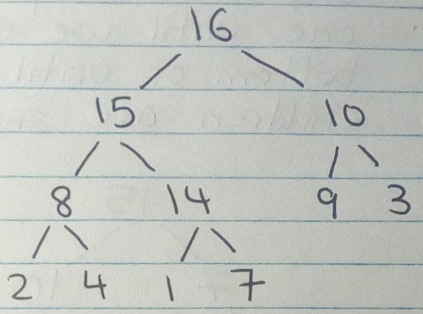
2. Assign the priority and job in the new node. Let v = the new node

3. We percolate v up.
   while v has parent with smaller priority:
      Swap them
      v = v.parent

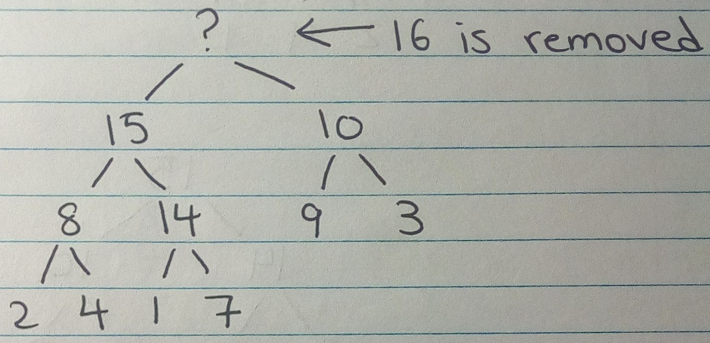
The worst case time is Θ(height). The height is $\lfloor \lg n \rfloor + 1$, so the worst case time is $\Theta(\lg n)$.

3. Extract - Max:

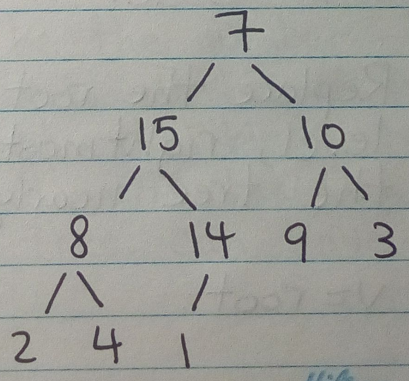
Suppose we have this heap.



If we extract-max, then this happens.

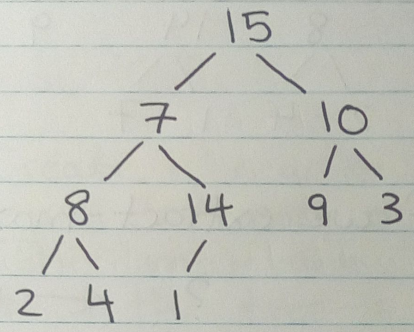
1. 

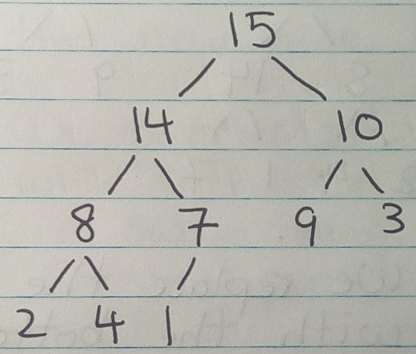
2. We replace the deleted node with the bottom level, rightmost element.



Hilroy

3. Then, we heapify.
We continuously compare the new root with both of its children and then swap with the larger one until we either reach the bottom or until both of its children are smaller than it.





Pseudo-Code:

1. Replace the root with the bottom level, rightmost item. This keeps the tree nearly-complete.

2. v = root

3. while v has larger child:
   swap with the largest child
   v = that child node

The worst case time is Θ(height).
The height is $\lfloor \lg n \rfloor + 1$, so the worst case time is $\Theta(\lg n)$.

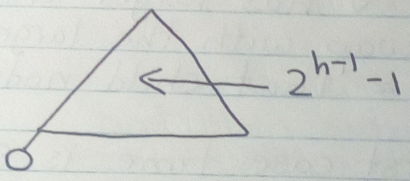
4. Binary Tree Size:

A binary tree of height h has at most $2^h - 1$ nodes.

h	n
1	1
2	3
3	7
4	15
5	31
i	$2^i - 1$

5. Heap: Height

Let n be the number of nodes a binary tree of height h can have. At most, the binary tree can have $2^h - 1$ nodes. At the very least, the binary tree can have 2^{h-1} nodes.



Therefore:

$2^{h-1} \leq n \leq 2^h - 1$

$2^{h-1} \leq n < 2^h$

$h-1 \leq \lg(n) < h$

$h \leq \lg(n)+1 < h+1$

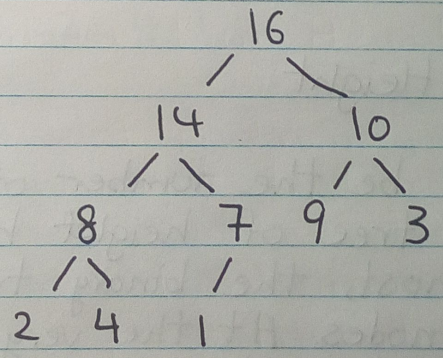
$h = \lfloor \lg n \rfloor + 1$

6. Implementing A Heap:

We can use an array to implement a heap.

- The start index is 1.
- Left child of node i is 2i.
- Right child of node i is 2i+1
- Parent of node i is $\lfloor i/2 \rfloor$

E.g.



	16	14	10	8	7	9	3	2	4	1
Index:	0	1	2	3	4	5	6	7	8	9	10

To insert an item into a heap with an array, we append the new item into the end of the array, update the heap size if necessary and percolate the new node up.

To extract max, we read and replace the item at index 1 with the item at the very end (last index) of the heap, decrement the heap size if necessary and heapify starting at index 1.

If we know where a node is in a heap stored in array A, we can increase the priority value in $\Theta(\log n)$ time. We set the new p value and then percolate the node up. Consider the pseudo code.

```
increasePriority(A, index, P):
    if p <= A[index]:
        return
    else:
        A[index] == P
        while A[⌊index/2⌋] < A[index]:
            swap them
            index = ⌊index/2⌋
```

The logic is that if P is less than or equal to the current priority, we do nothing. If p is greater than the current priority, we set the priority to p and we percolate up. We compare the priority of the element to the priority of its parent, swap if the parent's priority is smaller than the node's priority, and continue until you reach the root or until you get to a node who has a higher priority.

7. Building Heaps:

Given an array A of elements with priorities, whose only empty slots are at the far right, how can we turn A into a heap?

1. Sort A from highest priority element to lowest. $\Theta(n \log n)$
2. Create a new array B that represents a heap and go through every element of A and insert it into B. $\Theta(n \log n)$

3. Use the fact that each subtree of a heap is a heap. We start with the smallest subtrees, turning them into heaps using heapify. Then, we work up the tree.

```
heapify(A, i, size):
    max = i
    if (2i <= size and A[2i] > A[i]):
        max = 2i
    if (2i+1 <= size and A[2i+1] > A[max]):
        max = 2i+1
    if (max != i):
        swap A[i] and A[max]
        heapify(A, max, size)
```

We call heapify starting at the first node that is the root of a tree of height at least 2. We can get this node by taking the last child's parent, so $\left\lfloor \frac{\text{heapsize}}{2} \right\rfloor$.

A node at height h takes h-1 steps to fix.

At height h, there are at most $\frac{n}{2^h}$ nodes.

$$\sum_{h=2}^{\lfloor \lg n \rfloor + 1} (\text{num trees of height } h) \times (h-1)$$

$$= \sum_{h=2}^{\lfloor \lg n \rfloor + 1} \frac{n}{2^h} \times (h-1)$$

$$\leq n \times \sum_{h=2}^{\infty} \frac{h-1}{2^h}$$

$$= n \times \text{constant}$$

Therefore, building a heap takes $\Theta(n)$ time.

8. Heaps and Sorting:

Given an array, how can we use a heap to efficiently sort the array?

Soln:
We can convert the array into a heap ($\Theta(n)$ time) and repeatedly using extract-max, updating and balancing the tree and decrementing the size.

In total, it takes $\Theta(n \lg n)$ time.

Example:

Say we have this array [5, 4, 9, 7] and we want to sort it in increasing order.

We can first turn the array into a heap.

[5, 4, 9, 7] → [9, 7, 5, 4]

Then we continuously do extract-max until the array is sorted.

1. [9, 7, 5, 4]
2. Extract-max, update heap

   [7, 4, 5, 9]
3. Extract-max, update heap

   [5, 4, 7, 9]
4. Extract-max, update heap

   [4, 5, 7, 9]

   Done

9. Max VS Min

So far, we've been working with max priority queues and max heaps. There are min priority queues and min heaps, too.