Nearly Complete Binary Trees and Heaps

DEFINITIONS:

i) The ***depth*** of a node p in a binary tree is the length (number of edges) of the path from the root to p.

ii) The ***height*** (or ***depth***) of a binary tree is the maximum depth of any node, or -1 if the tree is empty.

Any binary tree can have at most 2^d nodes at depth d. (Easy proof by induction)

DEFINITION: A ***complete binary tree*** of height h is a binary tree which contains exactly 2^d nodes at depth d, $0 \le d \le h$.

- In this tree, every node at depth less than h has two children. The nodes at depth h are the leaves.

- The relationship between n (the number of nodes) and h (the height) is given by

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

  and

$$h = \lg(n+1) - 1.$$

- Complete binary trees are perfectly balanced and have the maximum possible number of nodes, given their height

- However, they exist only when n is one less than a power of 2.

DEFINITION: A ***nearly complete binary tree*** of height h is a binary tree of height h in which

a) There are 2^d nodes at depth d for $d = 1, 2, \ldots, h-1$,

b) The nodes at depth h are as far left as possible.

- Condition (b) can be stated more rigorously, like this:

  If a node p at depth $h-1$ has a left child, then every node at depth $h-1$ to the left of p has 2 children. If a node at depth $h-1$ has a right child, then it also has a left child.

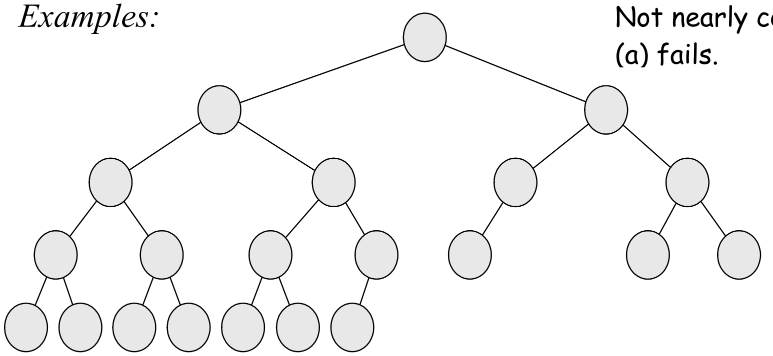
- The relationship between the height and number of nodes in a nearly complete binary tree is given by

$$2^h \le n \le 2^{h+1} - 1, \text{ or } \quad h = \lfloor \lg(n) \rfloor.$$

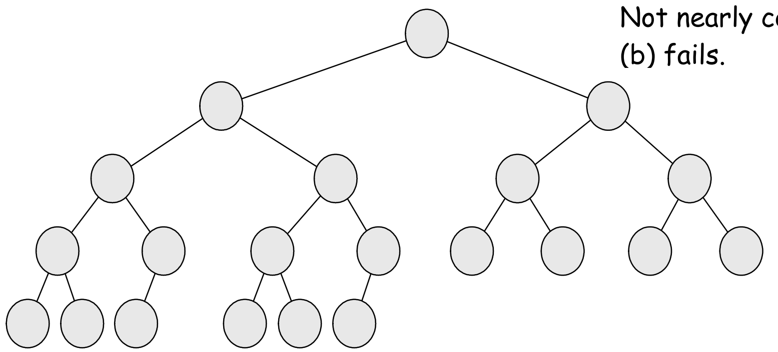
(This depends only on condition (a) in the definition.)

Examples:

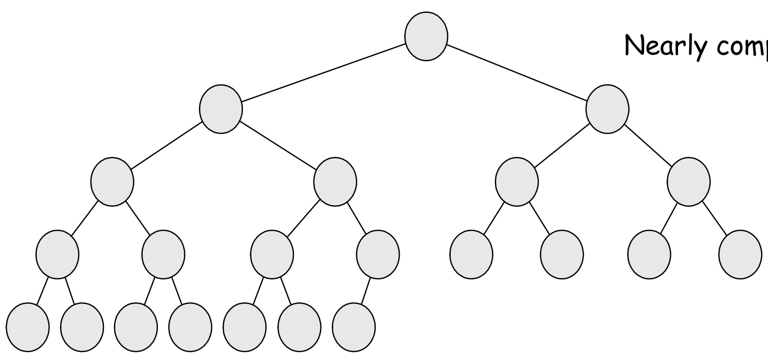
Not nearly complete:
(a) fails.



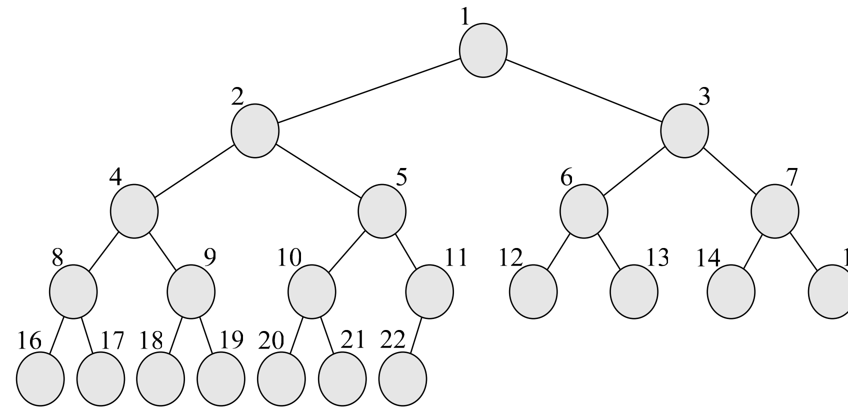
Not nearly complete:
(b) fails.



Nearly complete.



Say we label the nodes of a nearly complete binary tree by 1, 2, 3, ..., n in order of increasing depth, and left-to-right at a given depth.



Then, equating each node with its label,

i) $left(k) = 2k$, if $2k \le n$,

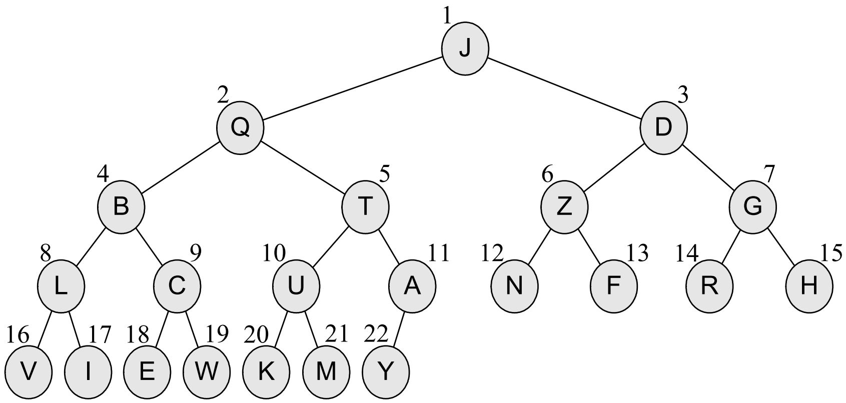
ii) $right(k) = 2k+1$, if $2k+1 \le n$,

iii) $parent(k) = \lfloor k/2 \rfloor$ if $k > 1$.

iv) k has one or more children if $2k \le n$. It has two children if any only if $2k+1 \le n$.

v) k is the left child of its parent if and only if k is even.

Suppose each node in the tree contains an element from some set. Denote the element in node p as *element*(p).



We don't really need the tree structure (nodes with pointers to the two children, and possibly the parent).

We can represent the tree implicitly by an array.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	J	Q	D	B	T	Z	G	L	C	U	A	N	F	R	H	V	I	E	W	K	M	Y

The array contains all the information in the tree.

- In the tree, if p is the node containing T (node 5), then *parent*(p) contains Q, *left*(p) contains U, and *right*(p) contains A. (We examine the link fields in the node.)
- In the array representation, we compute $\lfloor 5/2 \rfloor = 2$, $2 \cdot 5 = 10$, and $2 \cdot 5+1 == 11$, and we find *parent*(a[5]] = a[2] = Q, *left*(a[5]] = a[10] = U, and *right*(a[5]) = a[11] = A.

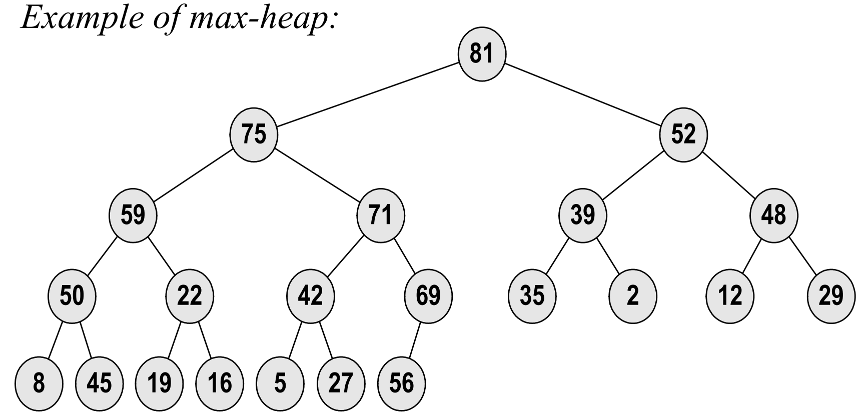
It is useful to think in terms of the tree, but all computation is actually performed with the array.

DEFINITION: A ***max-heap*** (or simply a ***heap***) is a nearly complete binary tree in which each node contains an element from a set S with a strict weak ordering, such that:

> For each node p except the root, $element(parent(p)) \gtrsim element(p)$. } Heap condition at node p

A ***min-heap*** is defined similarly except the heap condition is $element(parent(p)) \lesssim element(p)$.

Example of max-heap:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
81	75	52	59	71	39	48	50	22	42	69	35	2	12	29	8	45	19	16	5	27	56

Note in a max-heap:

i) The largest element is in the root.

ii) The second largest element is in one of the children of the root, but the third largest element need not be in the other child.

With a heap, we can perform at least these operations efficiently (time at worst Θ(lg(*n*)).

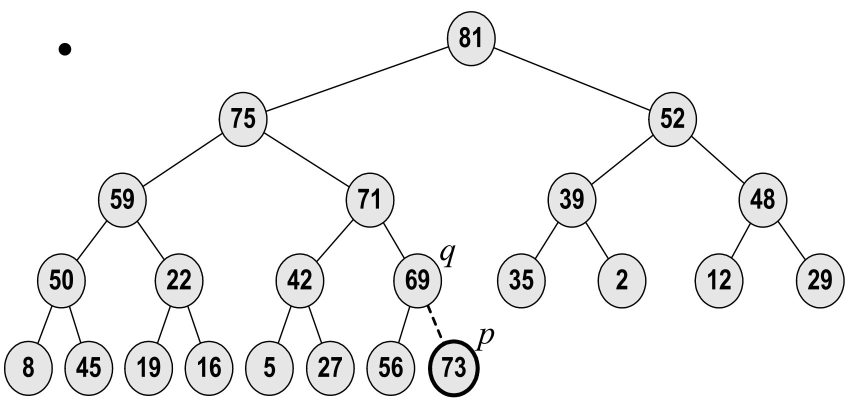
1) Insert a new element.

2) Find the largest element.

3) Remove the largest element.

1) *Insert a new element* (say insert 73, in the heap above)

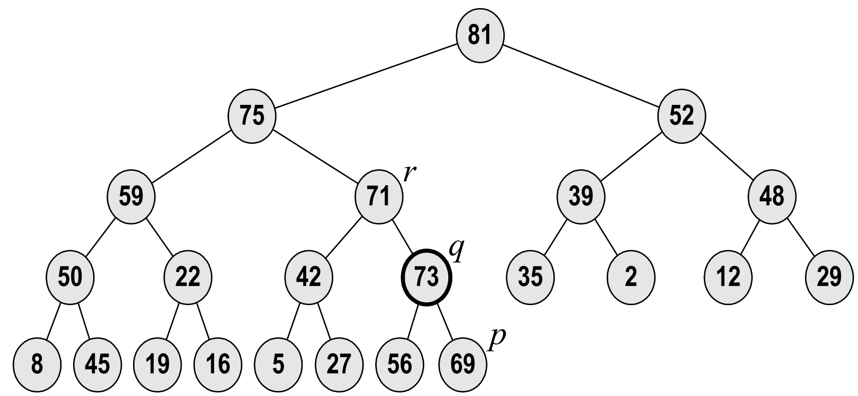
- There is only one place where we can insert a new node, and still have a nearly complete binary tree.
-



In general, if the old size of the heap is *n*, the new node becomes of child of node $\lfloor (n-1)/2 \rfloor$ — a right child is *n* is even, and a left child if it is odd.

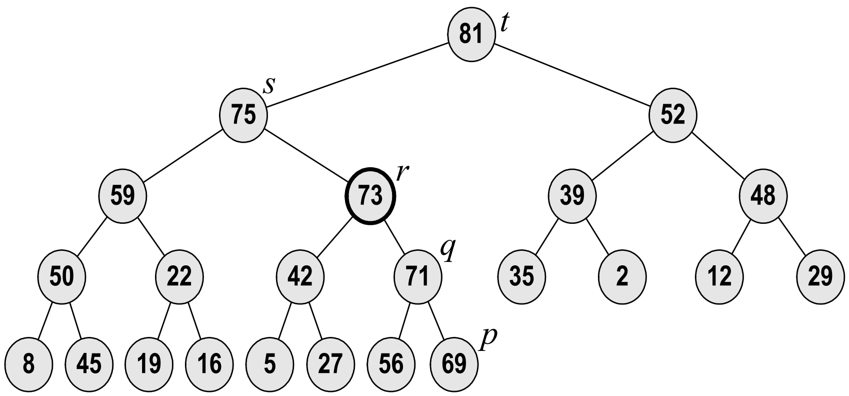
- The only place the heap property can possibly fail is at the new node (node *p*).

- We compare the element in node *p* (73) with the element in node *parent*(*p*) = *q* (69), and find that the heap property does fail at node *p*.
  - We correct the problem at *p* by exchanging the elements in nodes *p* (73) and *q* (69).



- Now the only place the heap property can possibly fail is at node *q*.

- We compare the element in node q (73) with the element in node $parent(q) = r$ (71), and find that the heap property does fail at node q.
  - We correct the problem by exchanging the elements in nodes q and r.



- Now the only place the heap property can possibly fail is at the parent of r (node s).
- We compare the element in node r (73) with the element in node $parent(r) = s$ (75), and find that the heap property actually holds at node r.
  - We are done.
- In the worst case, we would have compared the new element with the elements in nodes q, r, s, and t.

- In general, the worst-case number of comparisons to insert a new element is the depth of the new node.
  - This is the height of a heap with n+1 elements, or $\lfloor \lg(n+1) \rfloor$.
  - Thus: $C_{\max}(n) = \lfloor \lg(n+1) \rfloor \approx \lg(n)$,
    $T_{\max}(n) = \Theta(\lg(n))$.
- With the array representation, the algorithm to insert a new element is:

```
// Insert a new element x into a heap of size n
//  represented in an array A of size at least n+1.
max-heap-insert( A, n, x)
    n = n + 1;
    A[n] = x;
    while ( n > 1 and A[n] > A[⌊n/2⌋] )
        swap( A[n], A[⌊n/2⌋] );
        n = ⌊n/2⌋;
```

2) ***Find the largest element***

- The largest element is in the root.
- Simply return the element in the root (constant time)