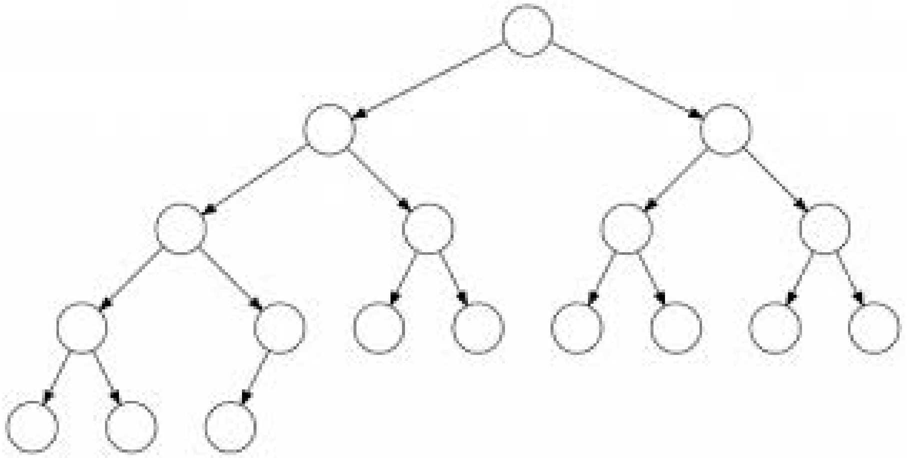
Data structures

HEAPS

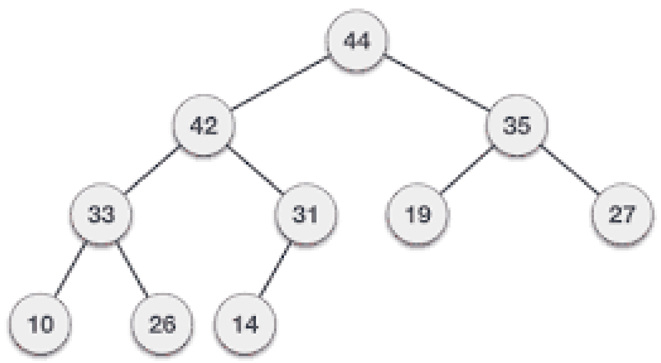
Heaps

- A heap is an (almost) complete binary tree.
  - Such a tree is composed of nodes.
  - Each nodes has at most two children.
  - A childless node is called a leaf.
  - All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.
  - In a complete tree, all leaves are on the same level



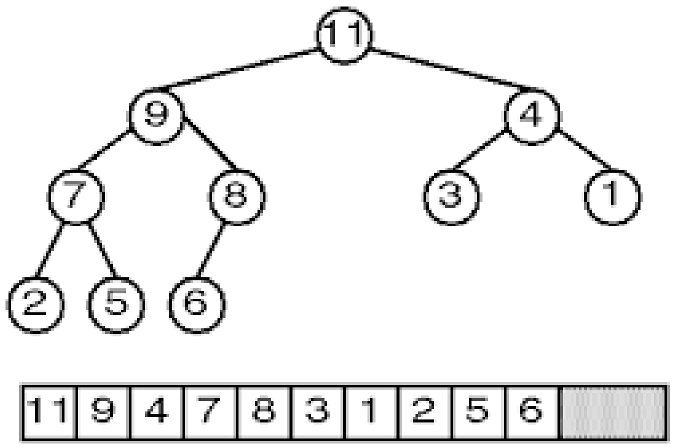
Heaps

- In addition to the requirement of the almost complete binary tree, nodes in a heap satisfy the **heap property.**
- The element at each node is always bigger (or equal) to all of it’s descendants.
- In this case we have we call the structure a **max heap.**
- Analogously we could define a **min heap**, by requiring that the element at each node will be smaller or equal to all of it’s descendants.



Heaps – Implementation

- One way to implement a heap is to represent it internally as an array.
- The size of the array must be bigger than the number of elements in the heap.
- All elements of the heap are stored on the ‘left’ side of the array.
- The root is the first element of the array.
- For a node at location *i,* it’s left child will be stored at location *2i,* it’s right child will be stored at location *2i + 1.*
- Question: where can we find the parent of the node at location *i?*



Heaps – Implementation

- Another possible implementation is similar to a linked list, and consists of a Node class.
- Except having a field to store the data, each node will have two pointers for it's two children.
- A node with both children set to null is by definition a leaf.
- It may also be beneficial to have a third pointer, pointing to the parent of the node.

Heapify up and down

- Two important operations on a heap are:
  - **Percolate up** – Move a node up the tree, as long as needed, until it reaches the correct level.
  - **Max Heapify**– Move a node down the tree, as long as needed, until it reaches the correct level.

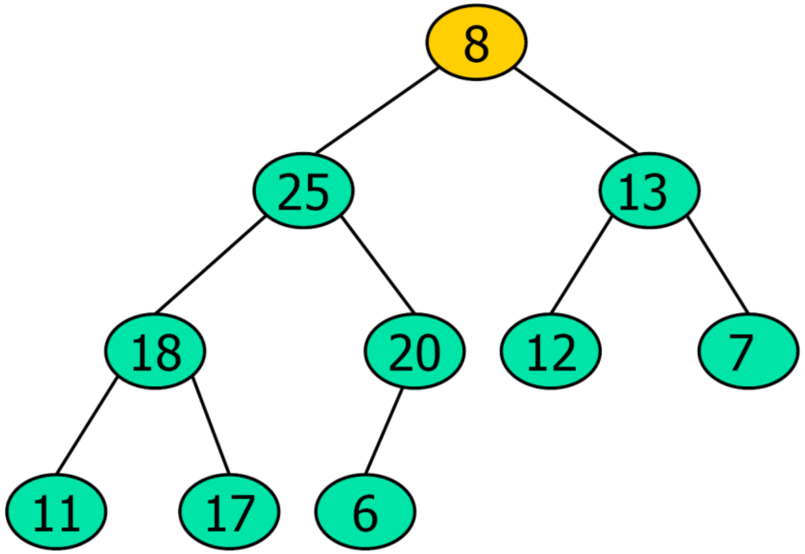
```
Max Heapify(i):
        while(A[i] is smaller than one of its children)
                j <- index of maximal child
                switch A[i] and A[j]
                i <- j
```

```
PercolateUp(i):
        while(A[i] is bigger than its parent)
                j <- index of parent
                switch A[i] and A[j]
                i <- j
```

  - Both methods visit each level of the tree at most once.

Max Heapify

How will the tree look after percolating down the root?



Common operations

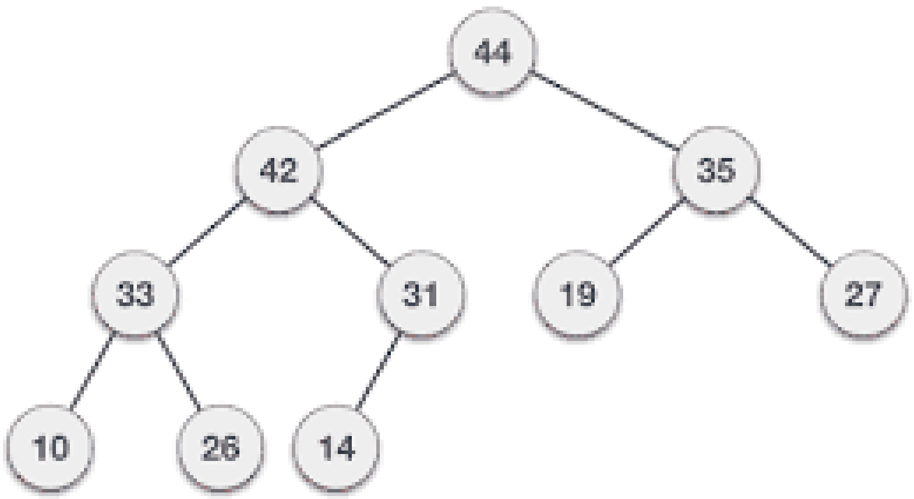
- FindMax:
  - Return the value of the root.
  - Runs in time O(1).
- ExtractMax:
  - Put the last element in place of the root.
  - Percolate down the root.
- Insert(x):
  - Put x as the last element.
  - Percolate up the last element.

Common operations

- BuildHeap (arr):
  - For i<-$n/2$ to 1:
    - percolate down the element at location i

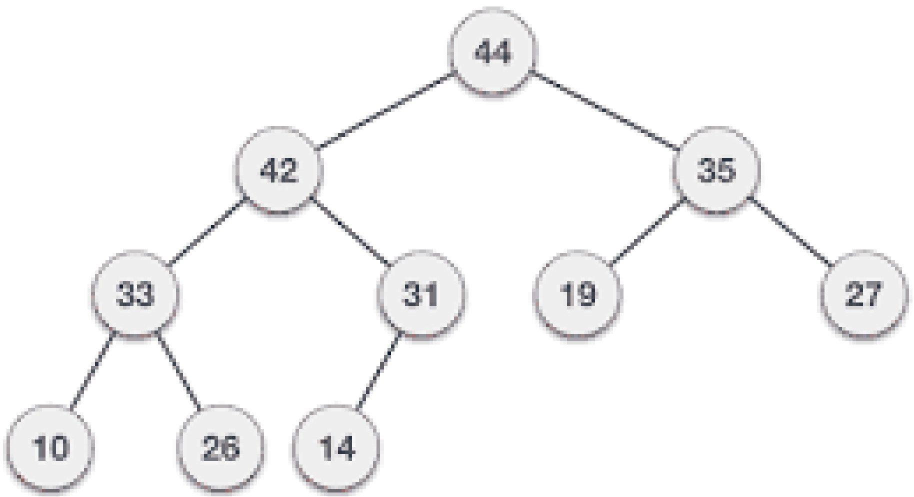
Add the element

- What will the tree look like after adding 43?



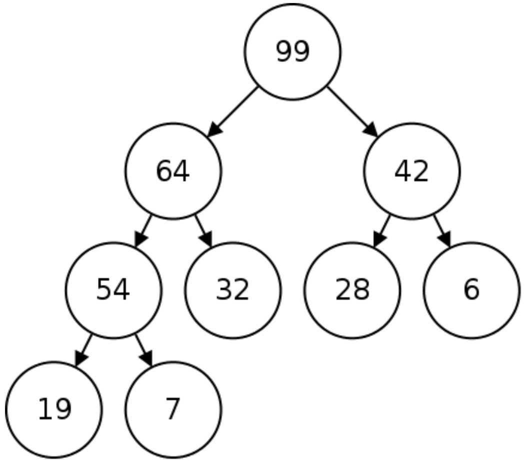
Extract maximum

- After extracting the maximum, twice?



Question

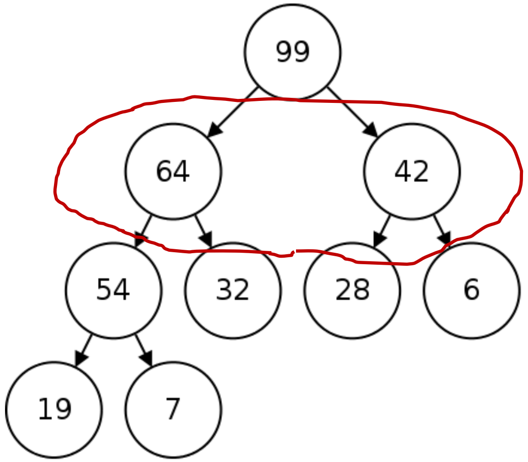
- Finding the value of the maximal element in the heap in trivial.
- How can we find the value of the second largest element?



Question

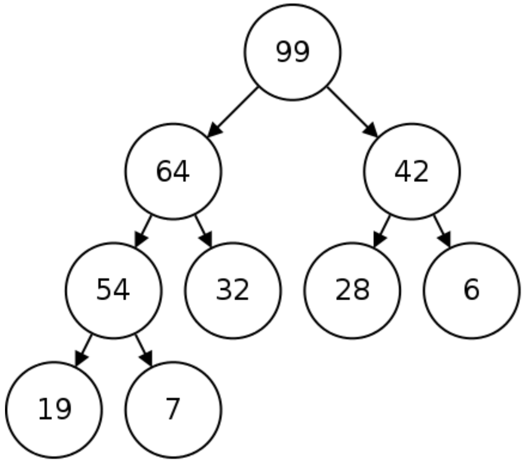
- Finding the value of the maximal element in the heap in trivial.
- How can we find the value of the second largest element?

- **Answer:**
- It must be a child of the root, one of the nodes in the second level.
- Just compare the two possibilities.
- This results in O(1) comparisons.



Question

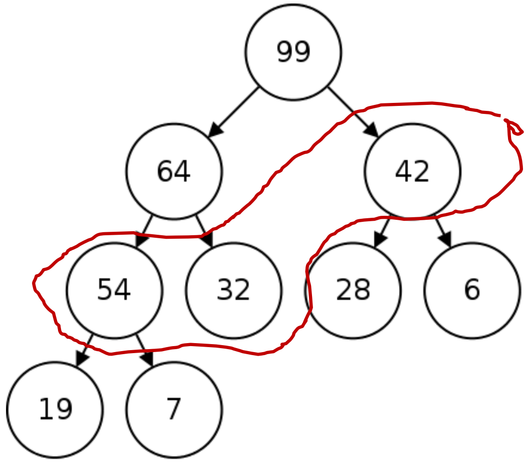
- How about the third largest?
- This time it could either be in the second level, but also on the third level.



Question

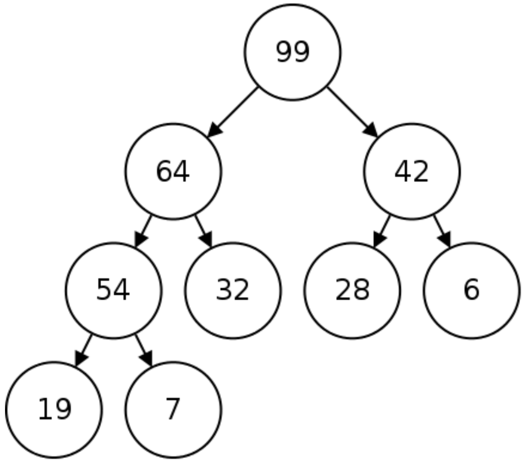
- How about the third largest?
- This time it could either be in the second level, but also on the third level.

- **Answer:**
- The third largest element is either a child of the largest element (not counting the second largest element) or a child of the second largest element.
- Just compare the three possibilities.
- This results in O(1) comparisons.



Question

- We now wish to find the value of the k largest element, where k could be any positive integer.



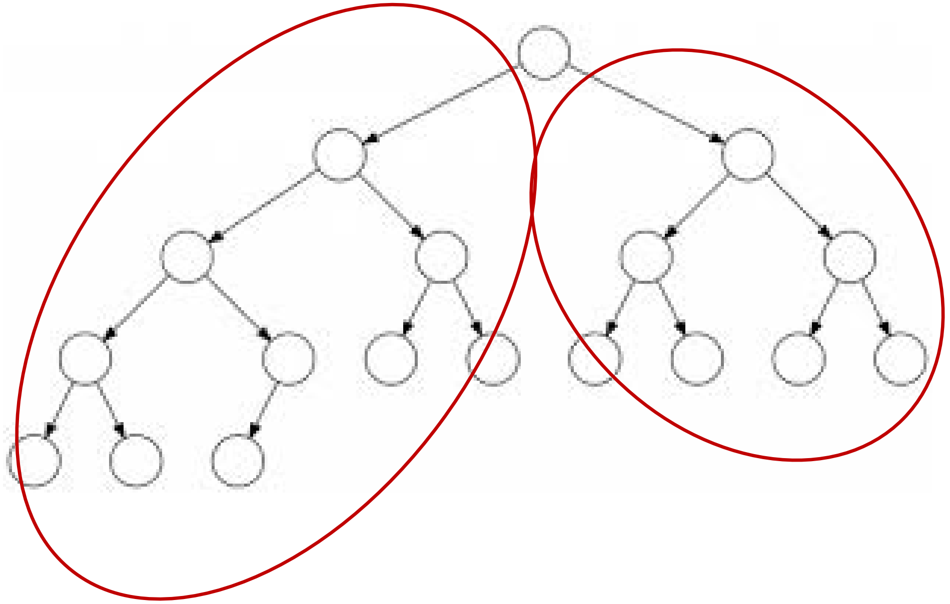
Question

- We now wish to find the value of the k largest element, where k could be any positive integer.

- **Answer:**
- Extract the maximal element k times.
- But this will destroy the heap in the process.
- Thus the k largest elements must be stored and inserted back into the heap after finding the required value.
- What is the runtime?

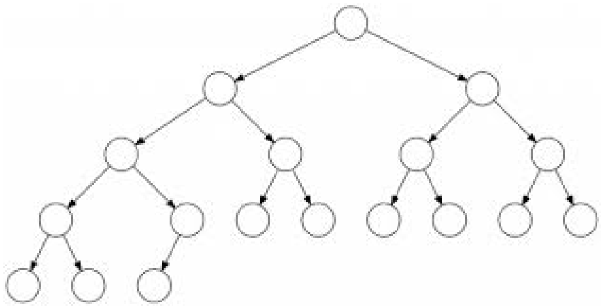
Question

- Prove: in every nearly complete binary tree with n vertices, exactly $\left\lceil \frac{n}{2} \right\rceil$ of the vertices are leaves.



Question

- Prove: in every nearly complete binary tree with n vertices, at least $\left\lceil \frac{n}{2} \right\rceil$ of the vertices are leaves.
- We prove by induction on h, the height of the tree.
- **Base:** For a tree of height 0, the only node is the root which is also a leaf.
- **Assumption:** Assume that for a nearly complete binary tree of height smaller or equal to h the claim is true and show for a nearly complete binary tree of height $h+1$.
- **Step:** Let T be a nearly complete binary tree with n vertices of height $h+1$ and consider T_L and T_R its left and right subtrees respectively and, denote by n_L and n_R the number of vertices in the trees. We note that each of the trees is a nearly complete binary tree and that each has height either h or $h-1$. Thus, by the induction hypothesis, the number of leaves in each tree is at least $\left\lceil \frac{n_L}{2} \right\rceil$ and $\left\lceil \frac{n_R}{2} \right\rceil$. One may show the claim now, since $n = n_L + n_R + 1$, along with the observation that either T_L or T_R are complete binary trees and hence have an odd number of vertices.



Question

- A queue is a data structure similar to a stack with a reverse order. That is, the first element into the queue is the first element to exit it.
- Suggest a way to implement a queue using two stacks.
- You may only use the basic stack operations.
- Try to keep runtime to a minimum.

Question

- We will hold two stacks S_1 and S_2.
- Items are always pushed to S_1, moved to S_2 and then popped from there.
- Notice that pushing and popping elements reverses their order.

Question

- enqueue(x):
  - S_1.push(x)

- dequeue():
  - if (S_2.isEmpty()):
    - while(! S_1.isEmpty()):
      - S_2.push(S_1.pop())
  - S_2.pop()