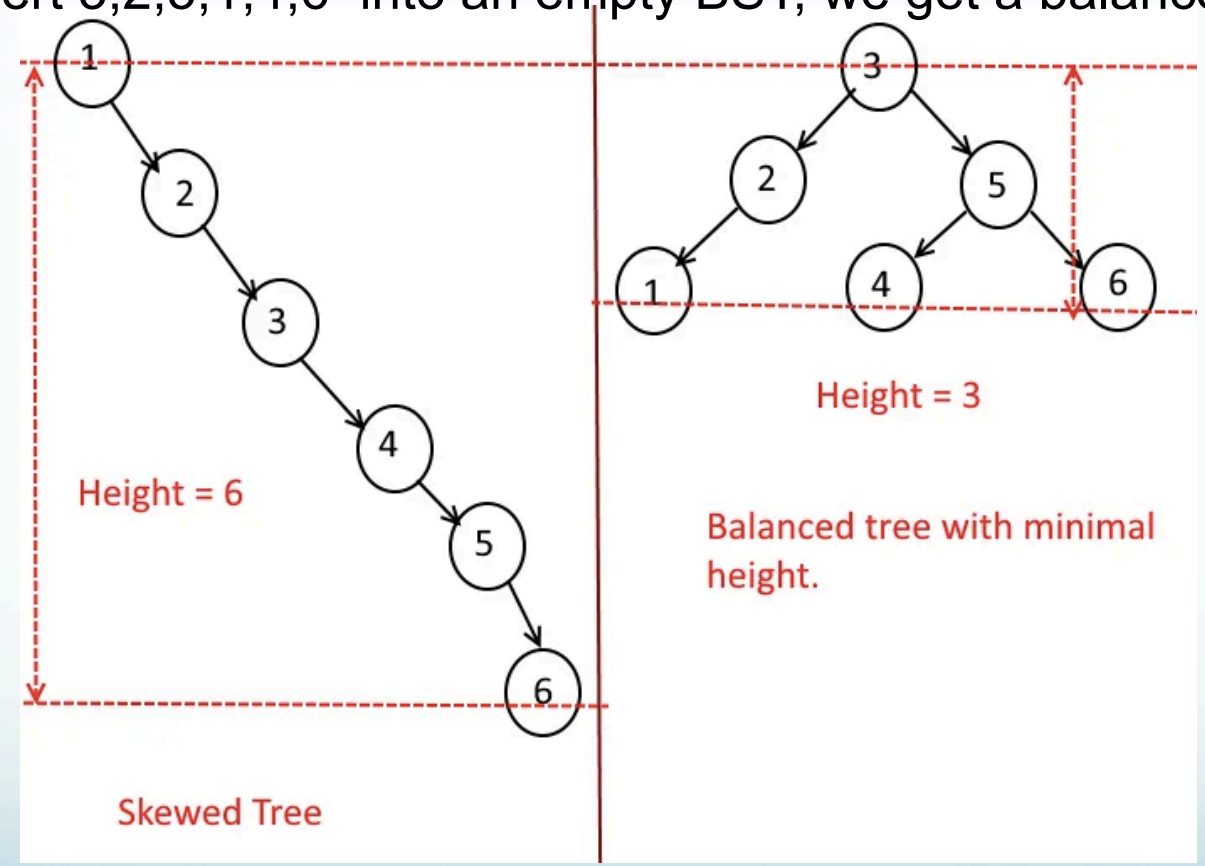
COMP20007 Workshop Week 9

	Preparation: - *have draft papers and pen ready* - open `ws9.pptx` from `github.com/anhvir/c207 and/or` - open `wokshop9.pdf` (from `LMS`), and - download lab files from `LMS`
1	BST: problems, Rotation (T1), Balance factor (T2)
2	AVL Tree: Concepts, Insertion (T3)
3	2-3 Tree: Concepts, Insertion (T4)
4	Binary Heap: Operations (T5, T6)
5	Quickselect (T7)
LAB	Lab: Playing with Quicksort code

BST: skewed, unbalanced, balanced

The efficiency of searching in a BST depends on how balance the tree is.

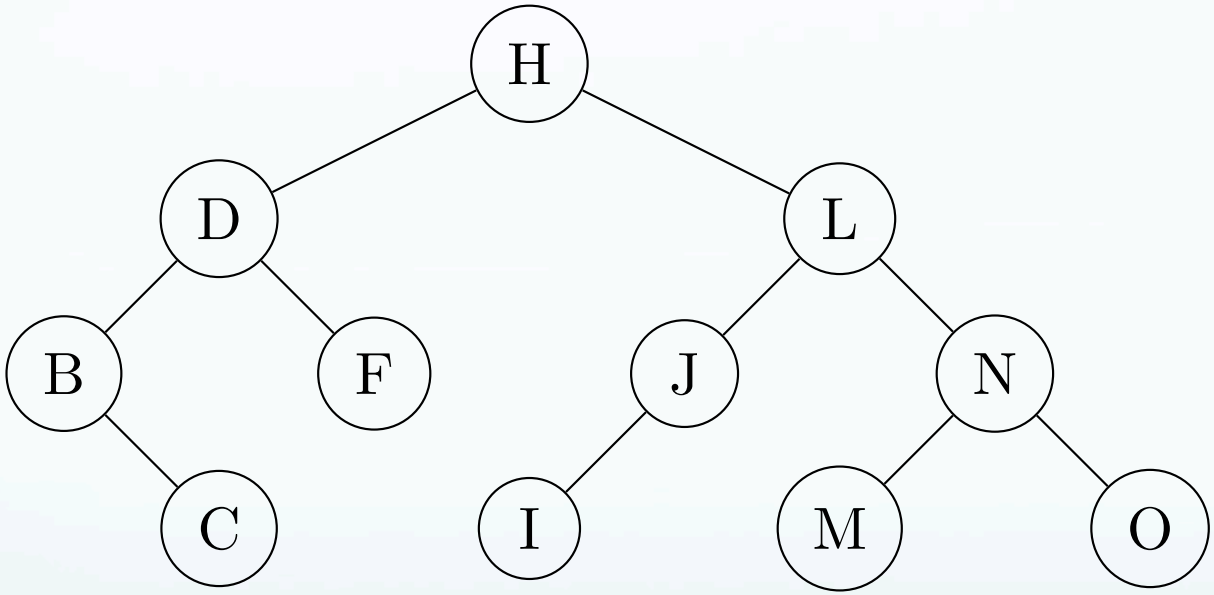
- If we insert 1,2,3,4,5,6 into an empty BST, we get a skewed tree.
- If we insert 3,2,5,1,4,6 into an empty BST, we get a balanced tree.



We want to have balanced search trees, no matter what's the data input order

T2: Balance factor

A node's 'balance factor' is defined as the height of its right subtree minus the height of its left subtree. Calculate the balance factor of each node in the following binary search tree.

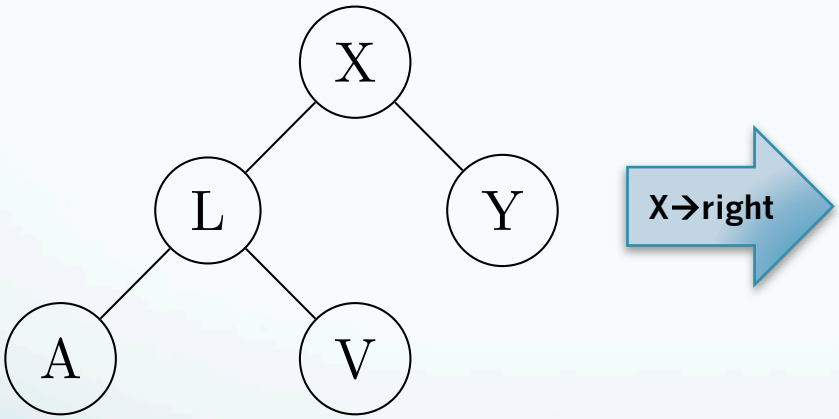


Balanced tree = when the balance factor of each node is 0, -1, or +1.

T1: Rotation

In the following binary trees, rotate the ‘X’ node to the right (that is, rotate it and its left child). Do these rotations improve the overall balance of the tree?

(a)

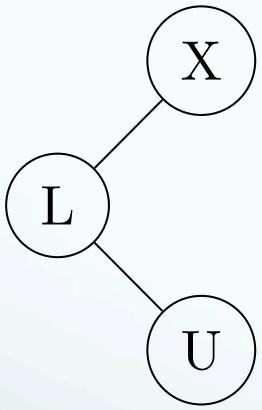


Note: 2 types of rotations: *Right Rotation* (a node and its left child), and *Left Rotation* (a node and its right child)

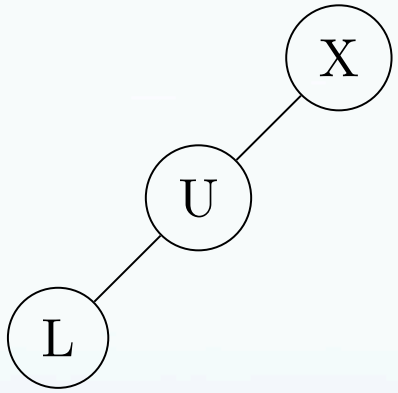
T1: Rotation

In the following binary trees, rotate the 'X' node to the right (that is, rotate it and its left child). Do these rotations improve the overall balance of the tree?

(b)



(c)



Note: 2 types of rotations: *Right Rotation* (a node and its left child), and *Left Rotation* (a node and its right child)

AVL Tree

- What? – just a balanced BST
- Why?

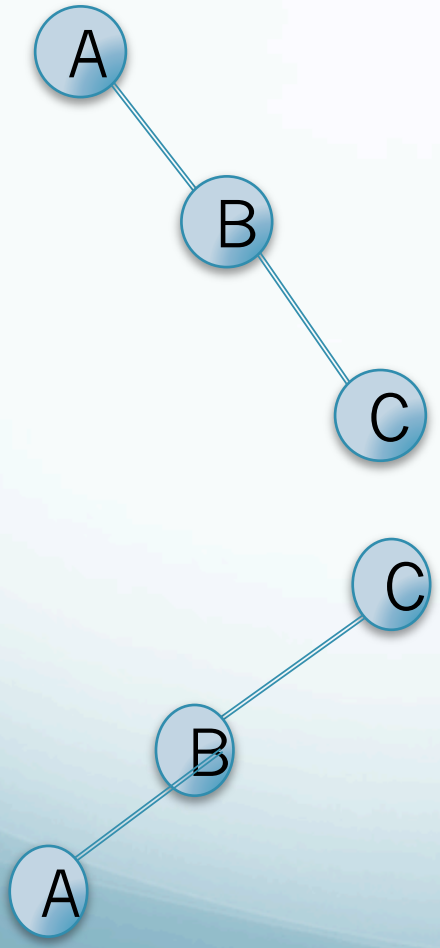
Using Rotations to rebalance AVL

- *At the start:* an empty BST is an AVL
- *Problem:* When inserting a new node to an AVL, the latter might become unbalanced
- *Approach*: use Rotations to rebalance. It's important first to determine:

  Which node to be rotated? Rotated left or right?

Two Basic Rotations: 1) Single Rotation

Applied when an AVL (subtree) has a stick form:



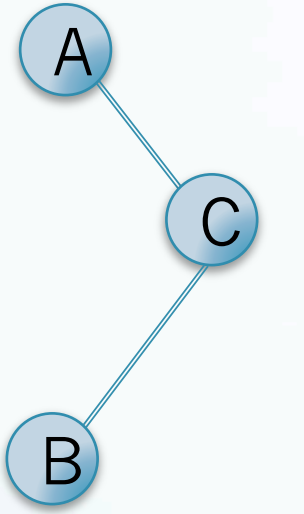
Questions we should ask ourselves:

- Which node (or tree) is unbalanced?
- Which rotation can be done?

→ Rotate the root and hence balance the stick

Two Basic Rotations: 1) Double Rotation

Applied when an AVL (subtree) has a non-stick form:



Rotation1:

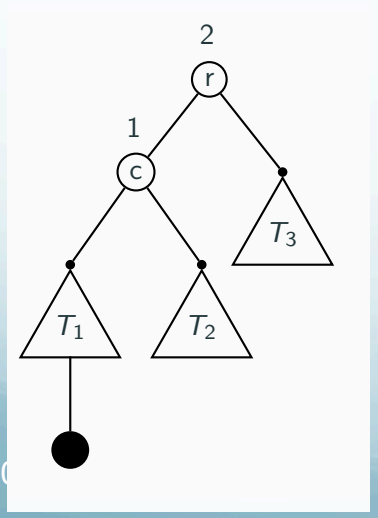
- Rotate the child of the unbalanced root and turn the tree to a stick

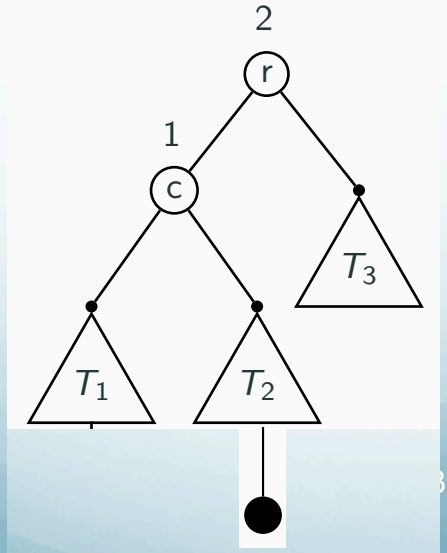
Rotation2:

- Just a single rotation for a stick.

Using Rotations to rebalance AVL

- Problem: When inserting to AVL, it might become unbalanced
- Approach: Rotations (Rotate WHAT?, and HOW?)
- Rotate WHAT?
  - The *lowest* subtree X which is unbalanced
- HOW
  - Consider *the first 3 nodes* X→A→B in the path from root X to the just-inserted node
  - Apply a single rotations if that path is a stick, double rotation otherwise





T3: AVL Tree Insertion

Insert the following letters into an initially-empty AVL Tree (*easy example*).

A V L T R E X M P

2-3 Trees

- *What?* It's a search tree, but not a binary! Each node might have 1 or 2 data, and hence 2 or 3 children.

- How to insert:
  - start from root, go down and insert to a leaf
  - if the new leaf has <=2 data, it's ok
  - if the new leaf has 3 data: promote the middle data to the parent

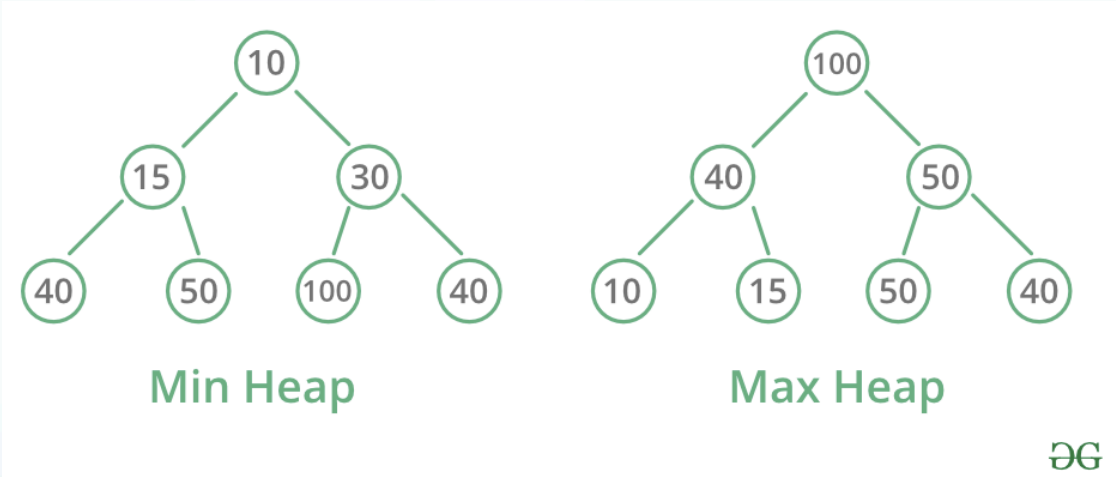
T5: 2-3 Tree Insertion

Insert the following letters into an initially-empty 2-3 Tree.

A L G O R I T H M

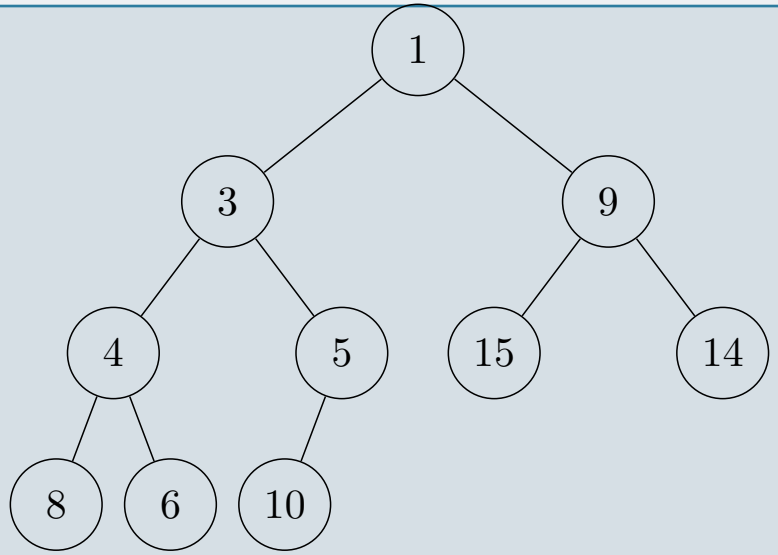
Binary Heap

- Binary heap is a priority queue
- For simple keys, we can have min-heap or max-heap



Binary Heap is implemented as an array!

Visualisation



Implementation

Array indices and content:

```
 0  1   2  3   4  5   6  7   8  9   10
|_|1  |3 |9  |4 |5  |15|14|8 |6  |10
   -- ----- ---------- --------
```

Heap `h` is a pair `{H[], n}`

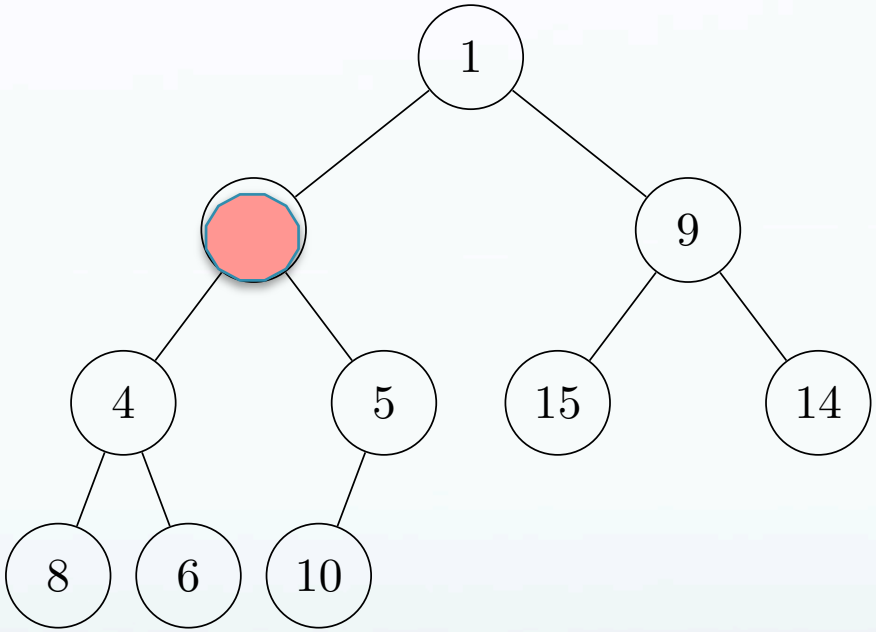
- `H[]` is an array
- `n:` number of elements in `H[]`
- `H[1..n]` is used, `H[0]` not employed

Heap `h` is implemented as an array `H[1..n]` (`H[0]` is not used)

- node of level 0 is in `H[1]`
- level `k` has 2^k nodes, and they are in $H[2^k..2^{k+1}-1]$ and so:
- parent of `H[i]` is `H[i/2]` iif `i>1`
- left child of `H[i]` is `H[2*i]` iif `2*i<=n`
- right child of `H[i]` is `H[2*i+1]` iif `2*i+1<=n`

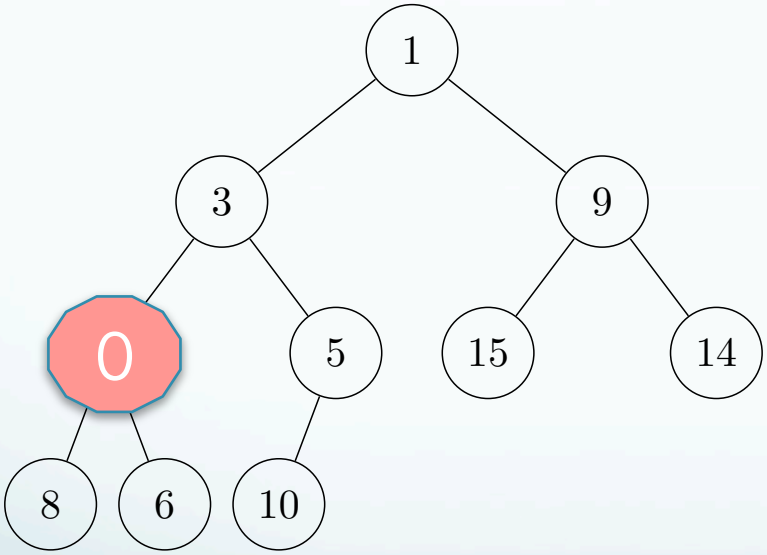
Heap Repair: `SiftUp` and `SiftDown`

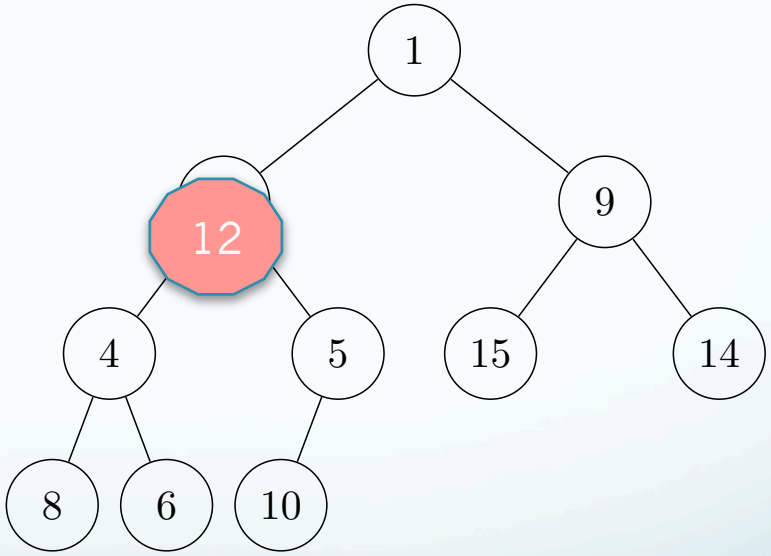
- **When?** When there is one, and only one, element that might violate the heap property.



Heap Repair: `SiftUp` and `SiftDown`

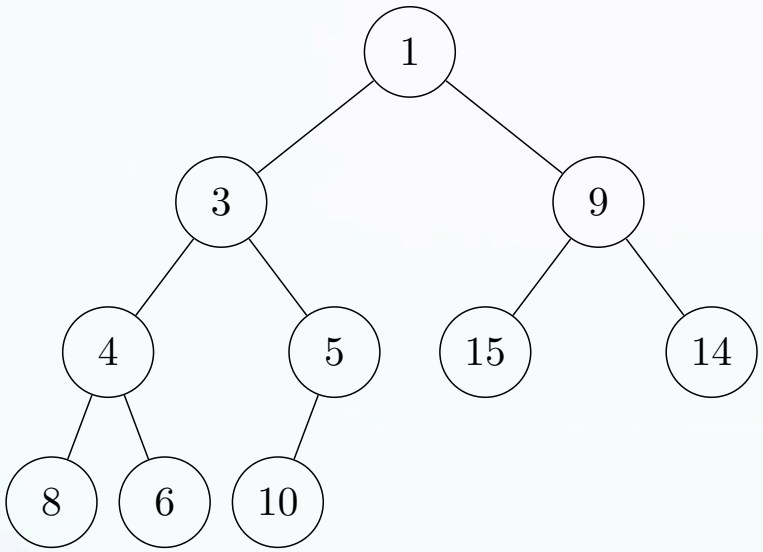
- When: there one, and only one, element that might violate the heap property.





- Complexity=?

A Priority Queue: Binary Min Heap

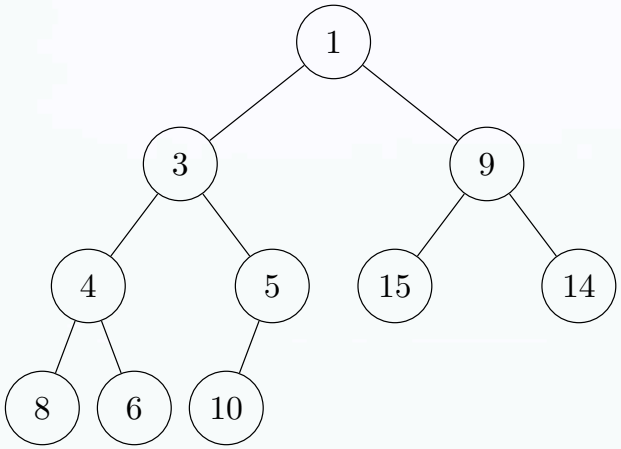
Example	Conditions
	*Array is:* [ ]

T1 a) Show how this heap would be stored in an array as discussed in lectures (root is at index `1`; node at index `i` has children at indices `2i` and `2i+1`)

Your answer: _

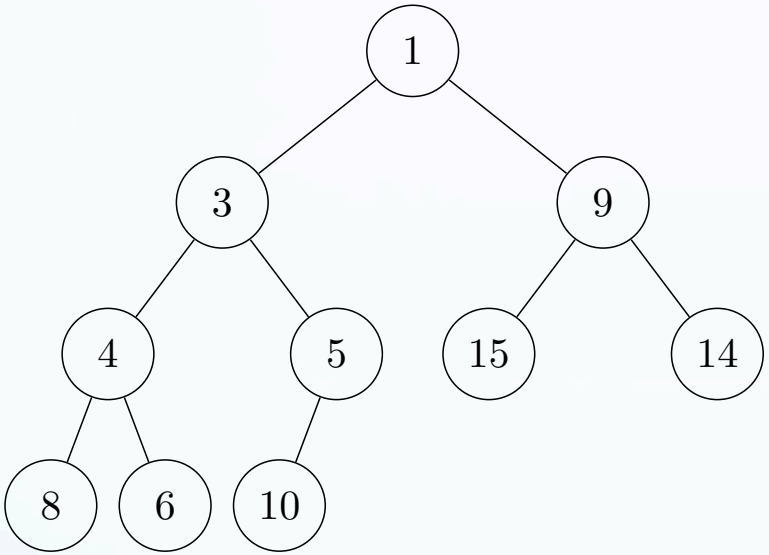
T5 b)

- Run the `RemoveRootFromHeap` algorithm from lectures on this heap by hand



T5 c)

- Run the `InsertIntoHeap` algorithm and insert the value 2 into the heap



Heap: Big-O Complexity

- Heap operations:
- `SiftUp`:
- `SiftDown`:
- `InsertIntoHeap`:
- `Heapify`: turn un-ordered array `H[1..n]` into a heap
- `RemoveRootFromHeap`:
- `heapsort`:

T6 [opt]: k-smallest using min-heap

- The *k-th smallest* problem:
  - Given an array `A[]` of `n` elements, and an integer `k`
  - Find the `k`-th smallest value (suppose that `k` is zero-origin, that is, `k` can be any of 0, 1, 2, ..., n-1)
- Using min-heap for the k-th smallest:
  - How
  - What the complexity?

Algorithm	Complexity
`function HeapkthSmallest(A[0..n-1],k)`	

T7

a) Design an algorithm Quickselect based on Quicksort which uses the `Partition` algorithm to find the `k`-th smallest element in an array `A`.

b) Show how you can run your algorithm to find the `k`-th smallest element where `k = 4` and `A = [9,3,2,15,10,29,7]`.

c) What is the best-case time-complexity of your algorithm? What type of input will give this time-complexity?

d) What is the worst-case time-complexity of your algorithm? What type of input will give this time-complexity?

e) What is the expected-case (i.e., average) time-complexity of your algorithm?

f) When would we use this algorithm instead of the heap based algorithm from Question T6?

g)

partitionning & qselect

```
   partition( A[lo..hi]):
     ...
     return m
```

```
11 qselect( A[lo..hi], k):
12   m= partition(A[lo..hi])
13   if (k==m) return A[m]
14   if (k<m)
15     qselect(A[lo..m-1], k)
16   else
17     qselect(A[m+1..hi], k)
```

```
21 ksmallest(A[0..n-1], k):
22   if (k>=0 && k<n)
23     return qselect(A[0..n-1], k)
```

T2

a) Design an algorithm based on Quicksort which uses the Partition algorithm to find the `k`-th smallest element in an array `A`.

b) Show how you can run your algorithm to find the `k`-th smallest element where `k = 4` and `A = 9,3,2,15,10,29,7`.

c) What is the best-case time-complexity of your algorithm? What type of input will give this time-complexity?

d) What is the worst-case time-complexity of your algorithm? What type of input will give this time-complexity?

e) What is the expected-case (i.e., average) time-complexity of your algorithm?