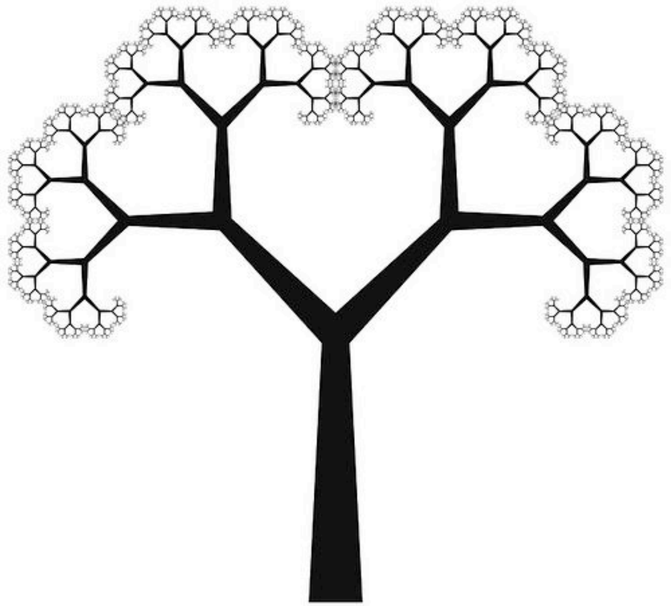
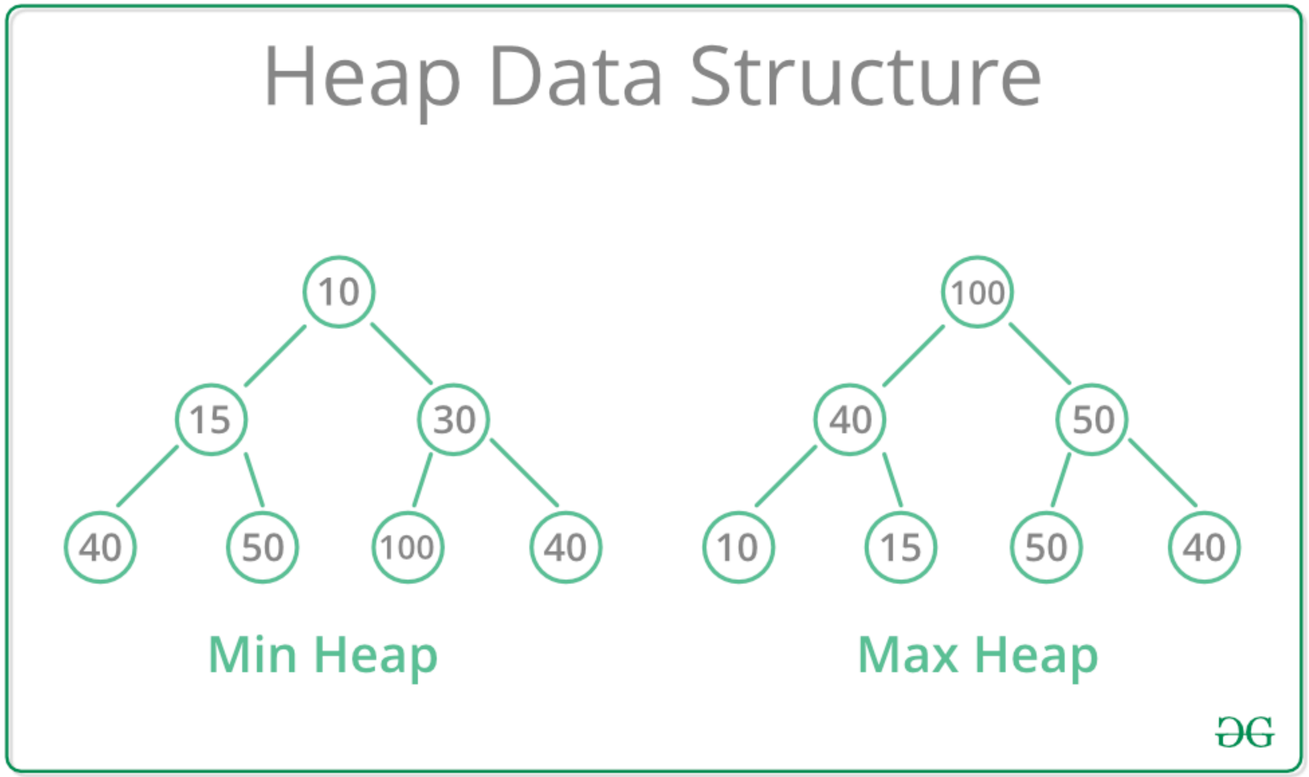
CS62 Class 13: Binary Trees & Heaps

Trees



Binary tree: ≤ 2 children per node



Heap: ordered binary tree

Agenda

- Binary Trees
- Tree Traversals
- Binary Search
- Binary Heaps

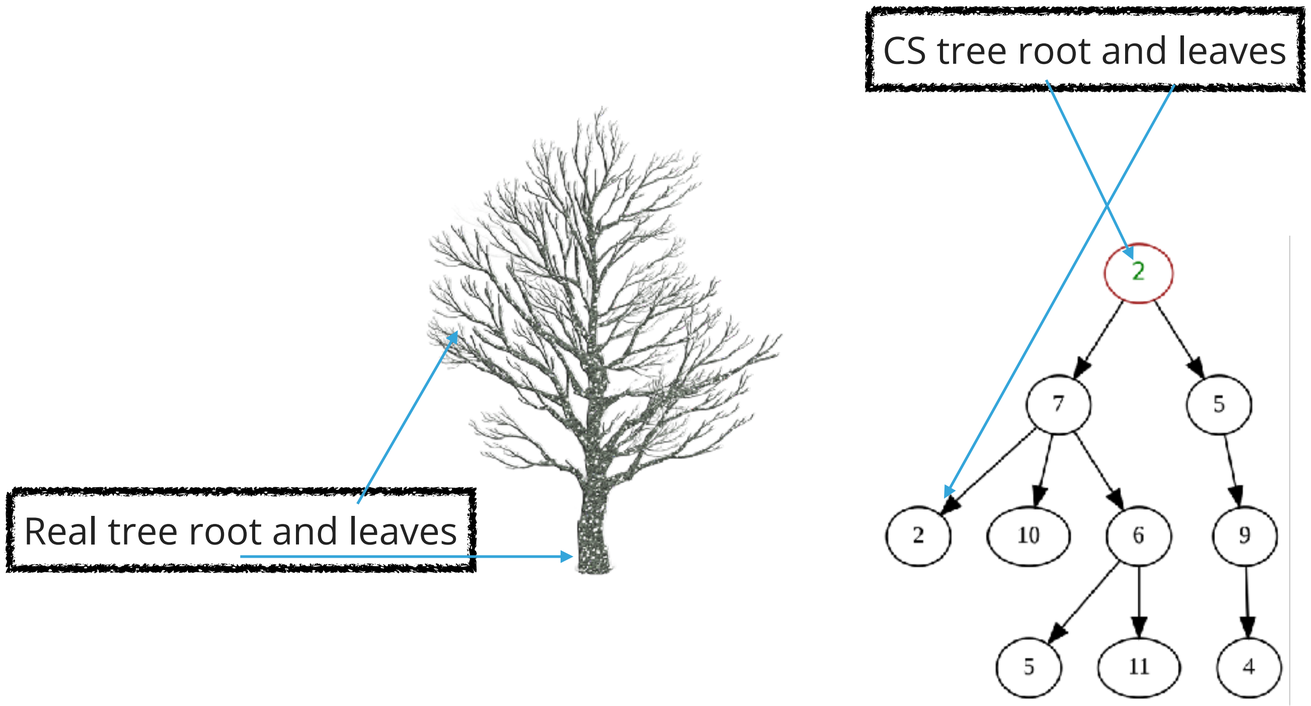
Binary trees

Trees in Computer Science

- Abstract data types that store elements hierarchically rather than linearly.
- Examples of hierarchical structures:
  - Organization charts for
    - Companies (CEO at the top followed by CFO, CMO, COO, CTO, etc).
    - Universities (Board of Trustees at the top, followed by President, then by VPs, etc).
  - Sitemaps (home page links to About, Products, etc. They link to other pages).
  - Computer file systems (user at top followed by Documents, Downloads, Music, etc. Each folder can hold more folders.).

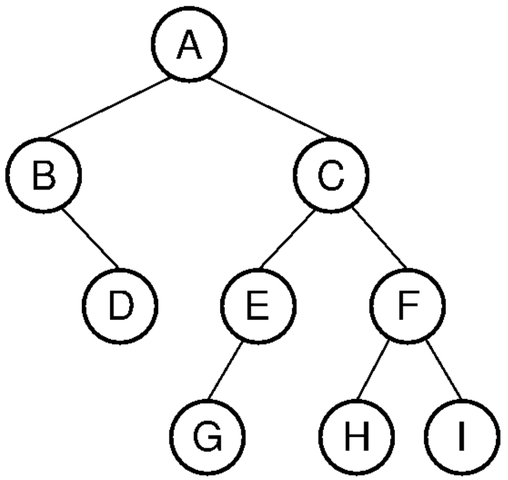
Trees in Computer Science

- Hierarchical: Each element in a tree has a **single** parent (immediate ancestor) and zero or more children (immediate descendants).



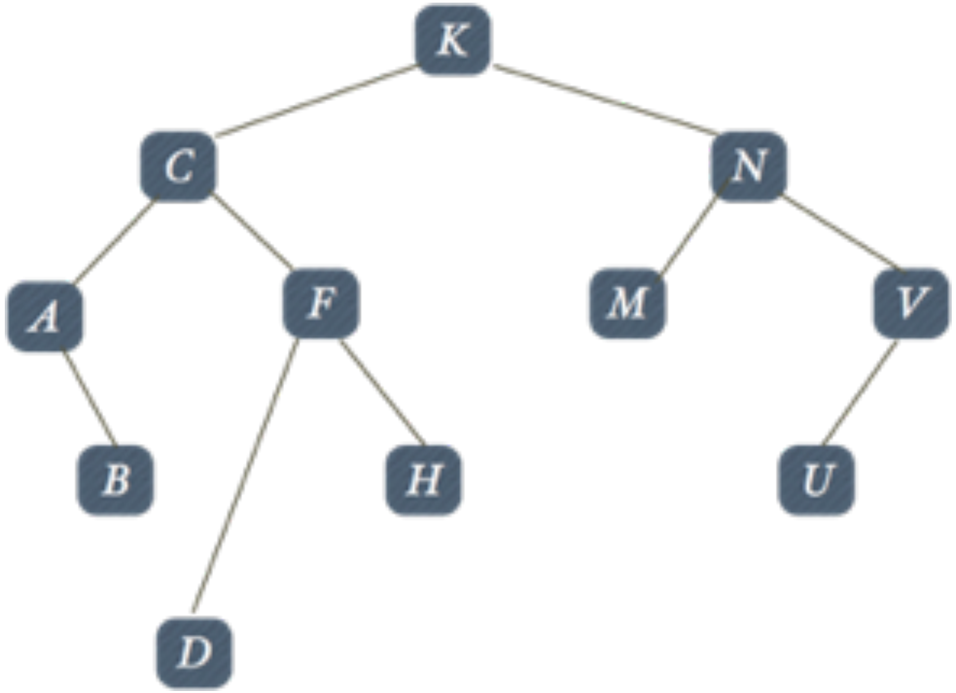
Definition of a tree

- A tree T is a set of nodes that store elements based on a parent-child relationship:
  - If T is non-empty, it has a node called the root of T, that has no parent.
    - Here, the root is A.
  - Each node v, other than the root, has a unique parent node u. Every node with parent u is a child of u.
    - Here, E's parent is C and F has two children, H and I.



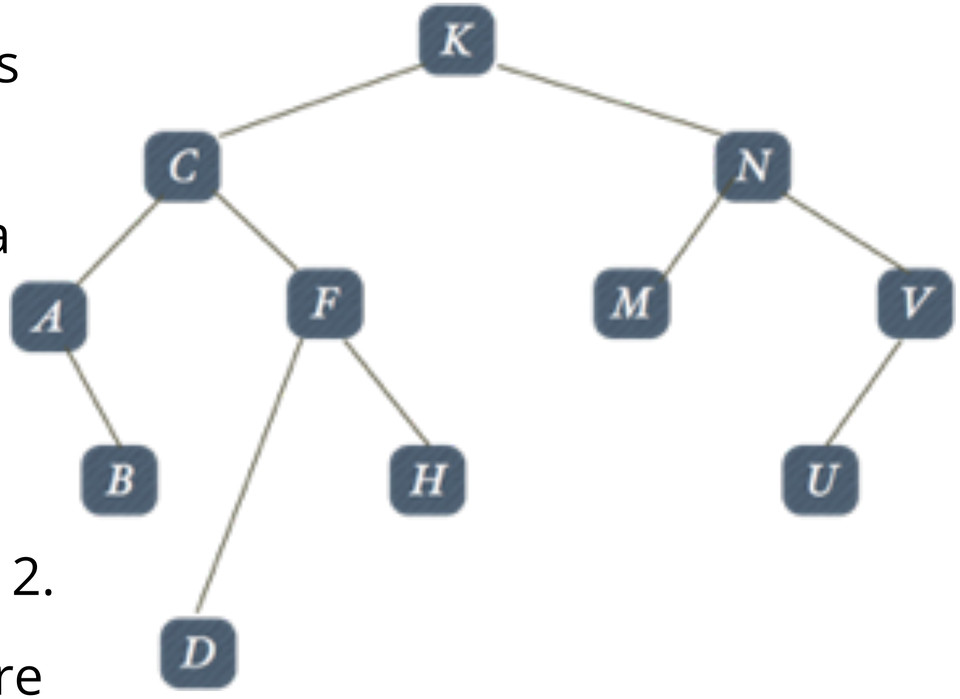
Tree Terminology

- Edge: a pair of nodes s.t. one is the parent of the other, e.g., (K,C).
- Parent node is directly above child node, e.g., K is parent of C and N.
- Sibling nodes have same parent, e.g., A and F.
- K is ancestor of B.
- B is descendant of K.
- Node plus all descendants gives subtree.
- Nodes without descendants are called leaves or external. The rest are called internal.



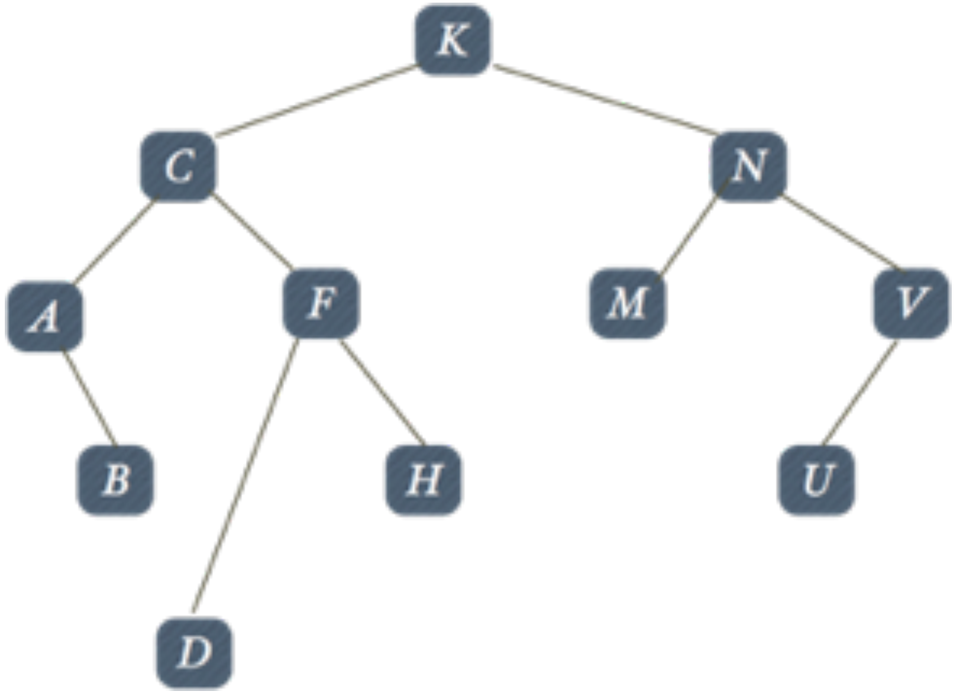
More Terminology

- Simple path: a series of distinct nodes s.t. there are edges between successive nodes, e.g., K-N-V-U.
- Path length: number of edges in path, e.g., path K-C-A has length 2.
- Height of node: length of longest path from the node to a leaf, e.g., N's height is 2 (for path N-V-U).
- Height of tree: length of longest path from the root to a leaf. Here 3.
- Degree of node: number of its children, e.g., F's degree is 2.
- Degree of tree (arity): max degree of any of its nodes. Here is 2.
- **Binary tree**: a tree with arity of 2, that is any node will have 0-2 children.



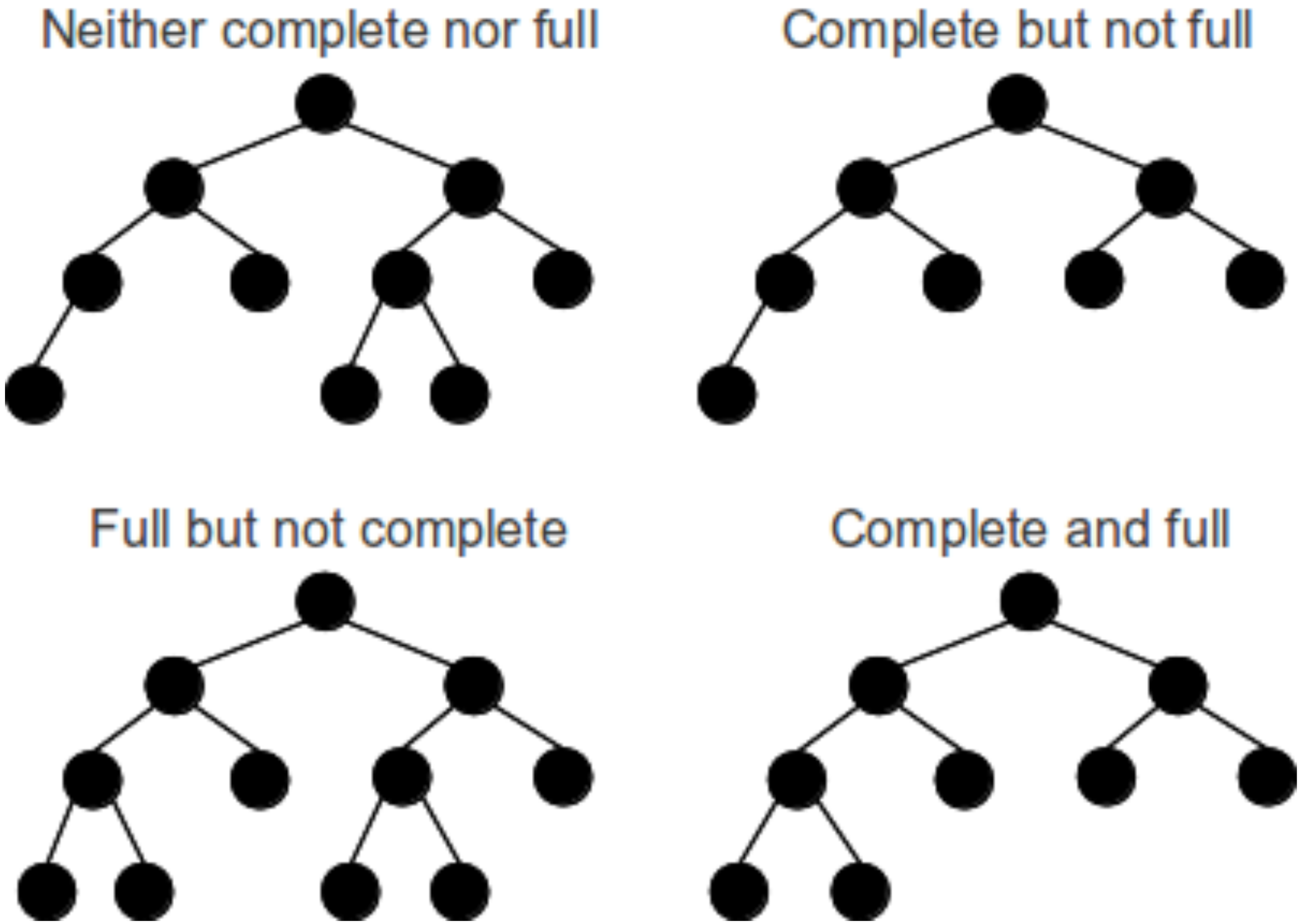
Even More Terminology

- Level/depth of node defined recursively:
  - Root is at level 0.
  - Level of any other node is equal to level of parent + 1.
  - It is also known as the length of path from root or number of ancestors excluding itself.
- Height of a node defined recursively:
  - If leaf, height is 0.
  - Else, height is max height of child + 1.
- The height of a binary tree is equal to the level of its deepest leaf node.



Full and complete

- Full (or proper): a binary tree whose every node has 0 or 2 children.
- Complete: a binary tree with minimal height. Any holes in tree would appear at last level to the right, i.e., all nodes of last level are as left as possible.



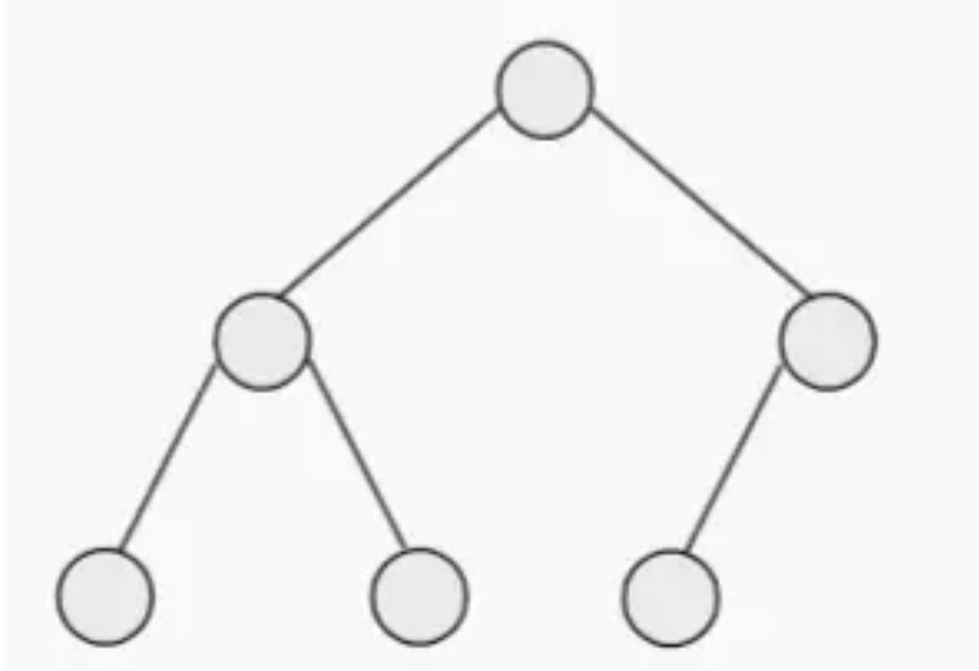
http://code.cloudkaksha.org/binary-tree/types-binary-tree

Practice Time: This tree is

- A: Full
- B: Complete
- C: Full and Complete
- D: Neither Full nor Complete

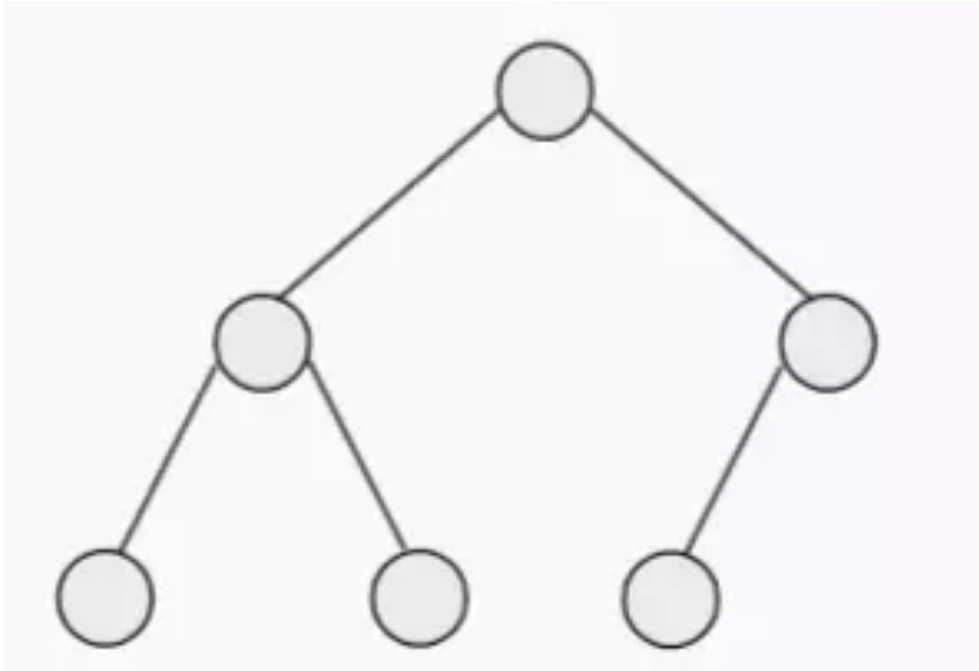
Its height is

- 1
- 2
- 3



Answer

- A: Full
- **B: Complete**
- C: Full and Complete
- D: Neither Full nor Complete

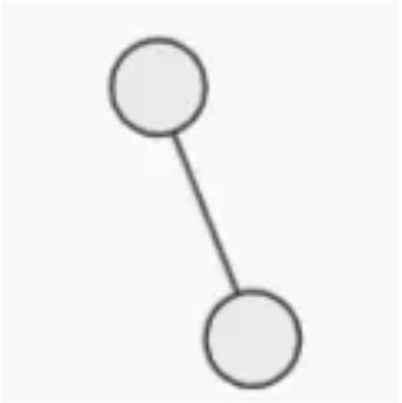


Its height is

- 1
- **2**
- 3

Practice Time: This tree is

- A: Full
- B: Complete
- C: Full and Complete
- D: Neither Full nor Complete

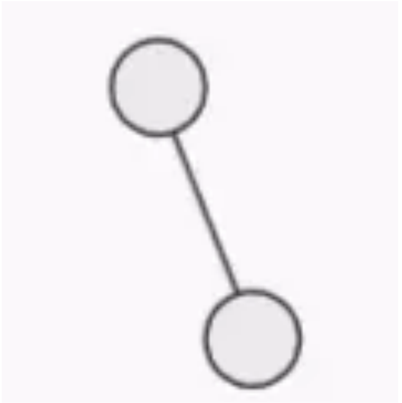


The degree of the tree is

- 0
- 1
- 2

Answer

- A: Full
- B: Complete
- C: Full and Complete
- **D: Neither Full nor Complete**

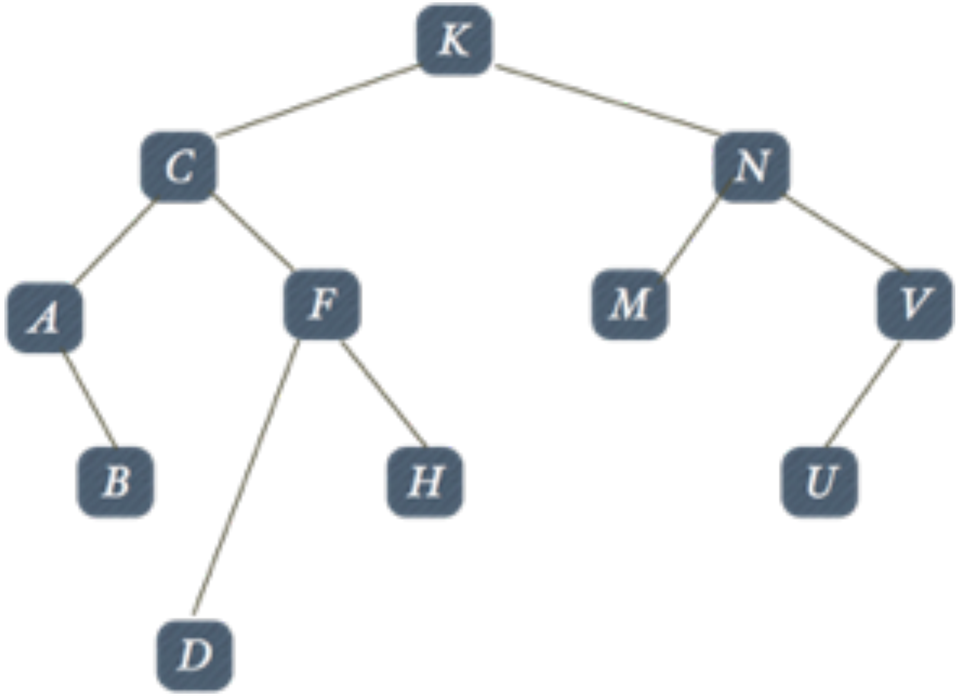


The degree of the tree is

- 0
- **1**
- 2

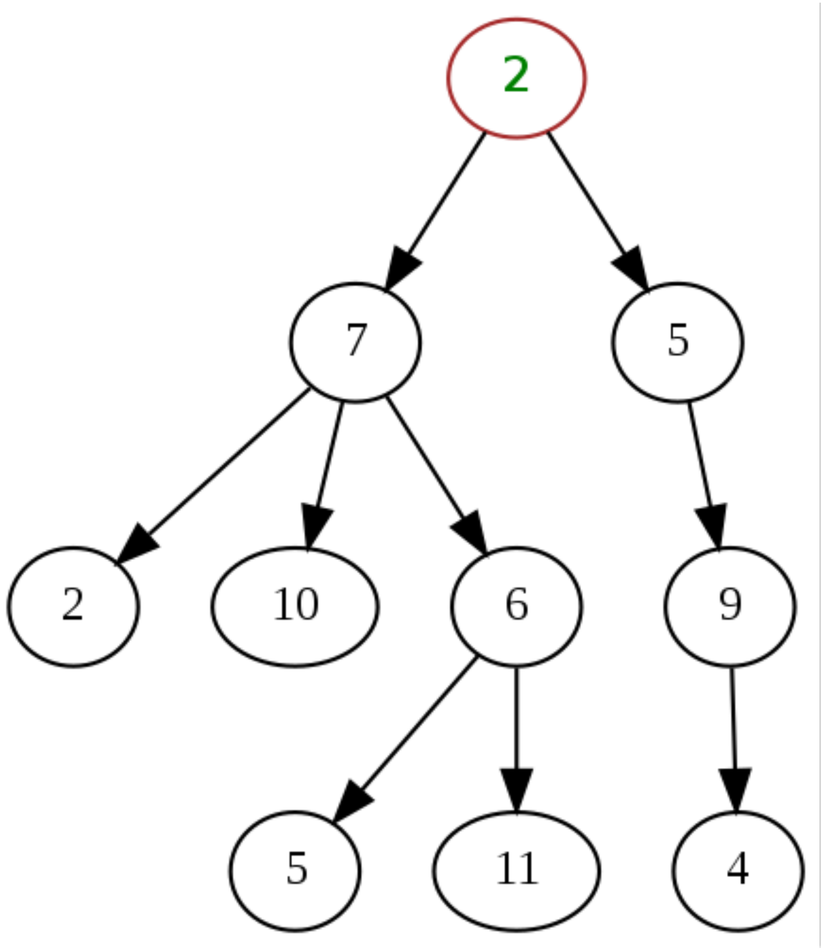
Counting in binary trees

- Lemma: if T is a binary tree, then at level k, T has $\leq 2^k$ nodes.
  - E.g., at level 2, at most 4 nodes (A, F, M, V)
- Theorem: If T has height h, then # of nodes n in T satisfy: $h + 1 \leq n \leq 2^{h+1} - 1$.
- Equivalently, if T has n nodes, then $log(n + 1) - 1 \leq h \leq n - 1$.
  - Worst case: When $h = n - 1$ or $O(n)$, the tree looks like a left or right-leaning "stick".
  - Best case: When a tree is as compact as possible (e.g., complete) it has $O(\log n)$ height.



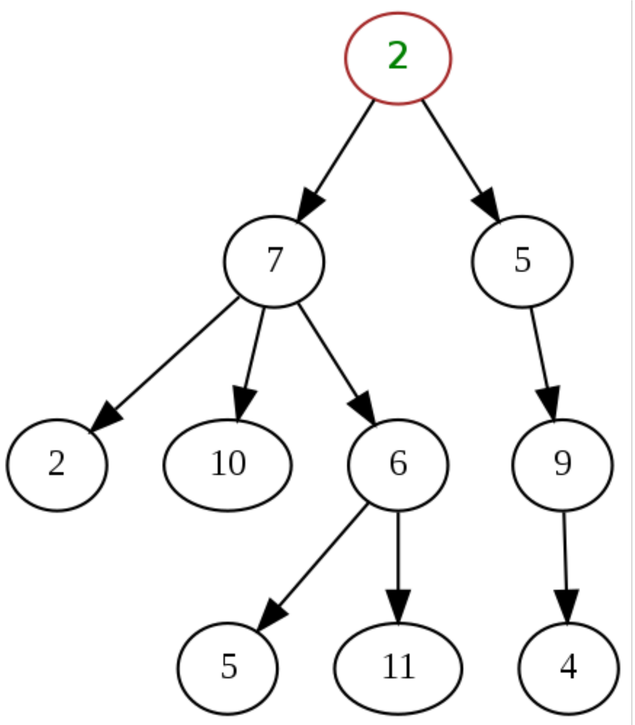
Worksheet time!

- Follow the instructions in the worksheet about the following tree:



Worksheet answers

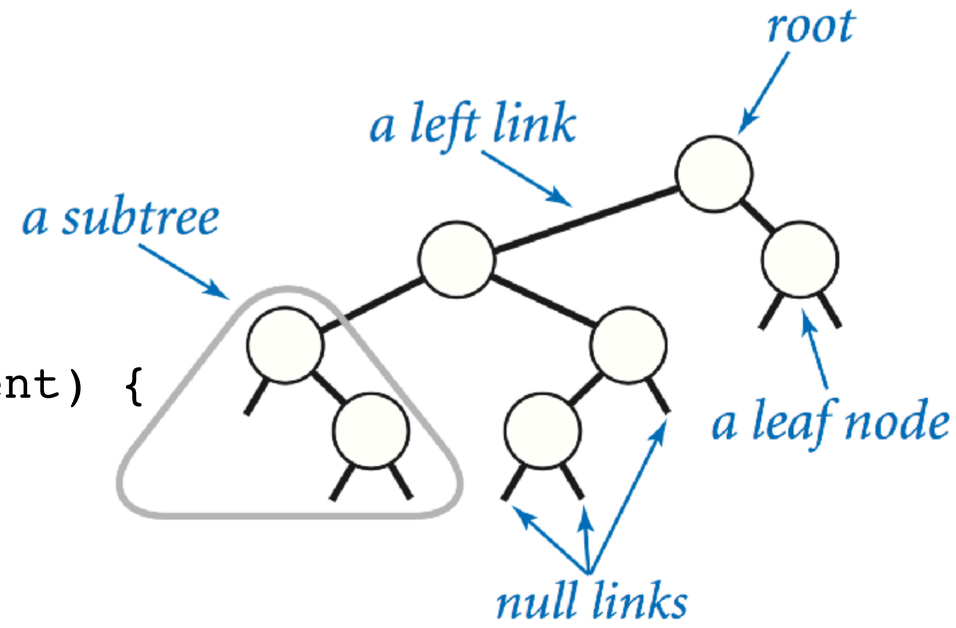
- Root: 2
- Leaves: 2 (in black), 10, 5, 11, 4
- Internal nodes: 2 (red), 7, 5, 6, 9
- Siblings of 10: 2, 6
- Parent of 6: 7
- Children of 2 (in red): 7, 5
- Ancestors of 10: 7 and 2 (in red)
- Descendants of 7: 2, 10, 6, 5, 11
- Length of path 2 to 4: 3
- Height of 7: 2
- Height of tree: 3
- Degree of 7: 3
- Arity/Degree of tree: 3
- Level/depth of 11: 3



Tree traversal

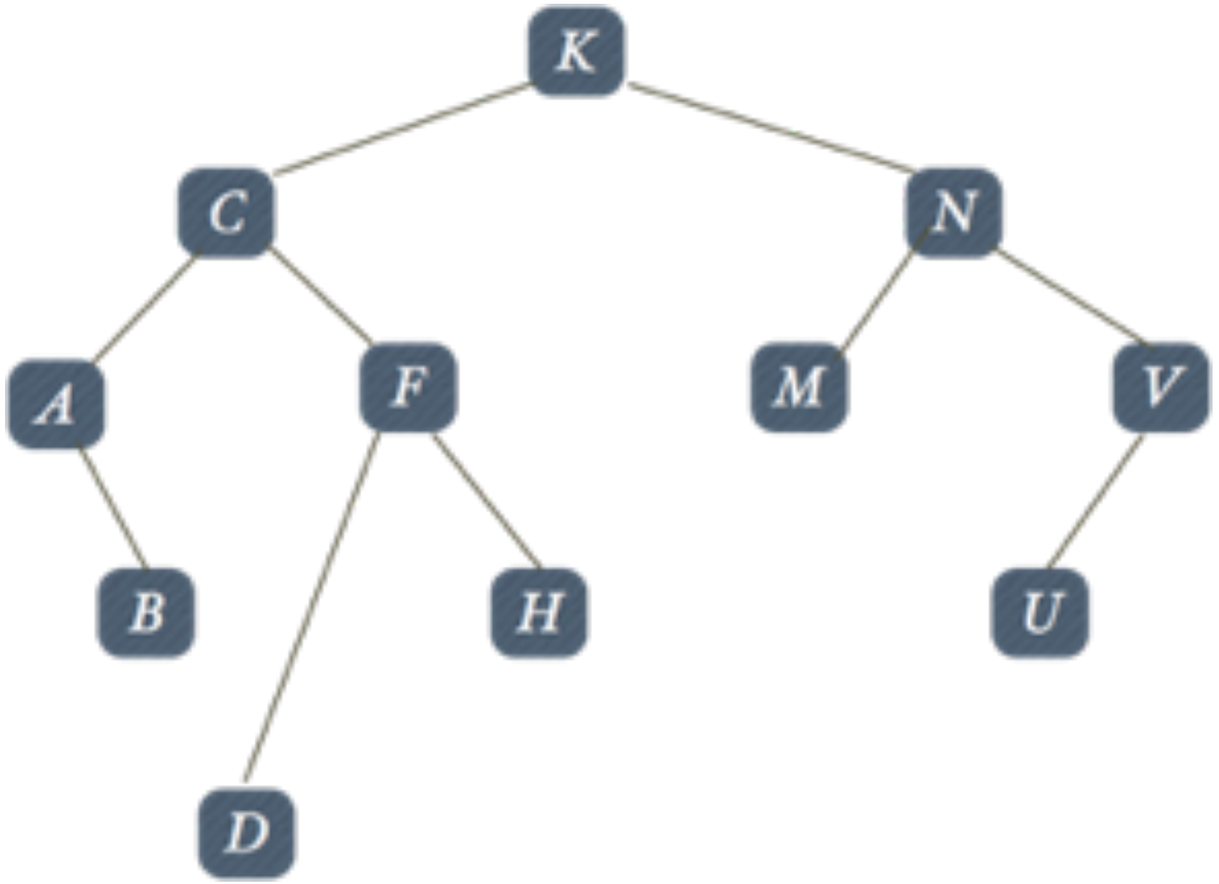
Basic idea behind a simple Binary Tree implementation

```
public class BinaryTree<E> {
   private Node root;

   private class Node {
      private E element;

      private Node left;
      private Node right;

      public Node(Node left, Node right, E element) {
         this.left = left;
         this.right = right;
         this.element = item;
      }
    }
}
```



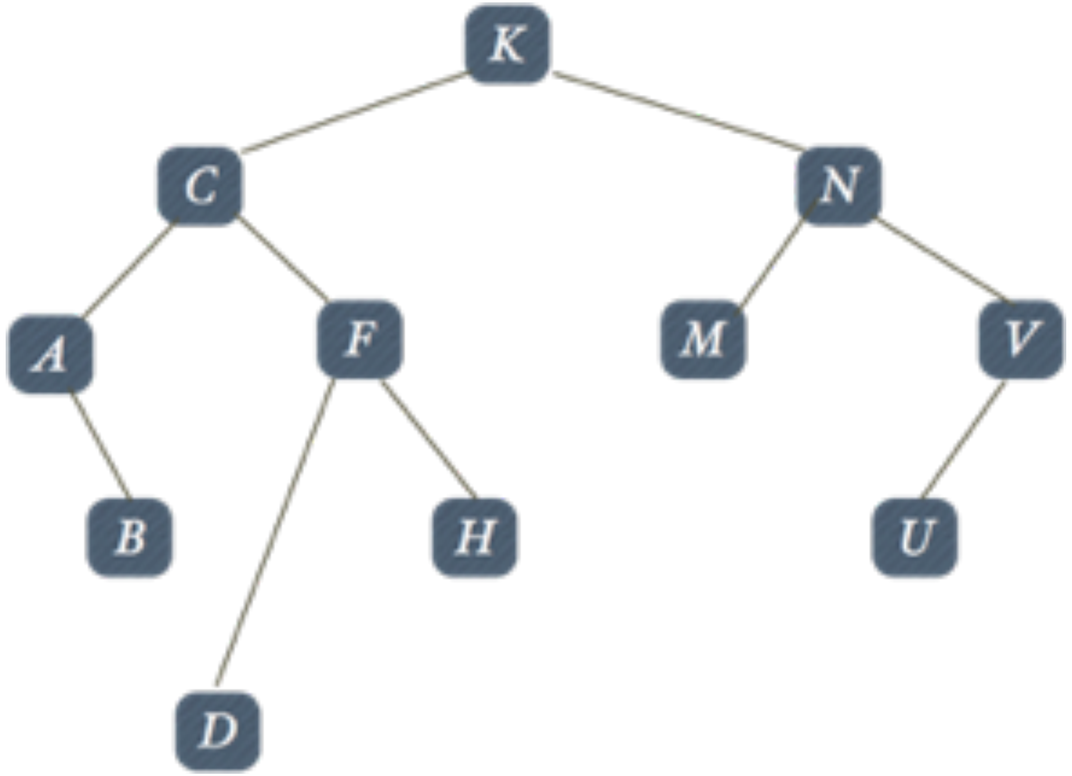
Pre-order traversal

- Preorder(Tree)
  - Mark root as visited
  - Preorder(Left Subtree)
  - Preorder(Right Subtree)
- K C A B F D H N M V U



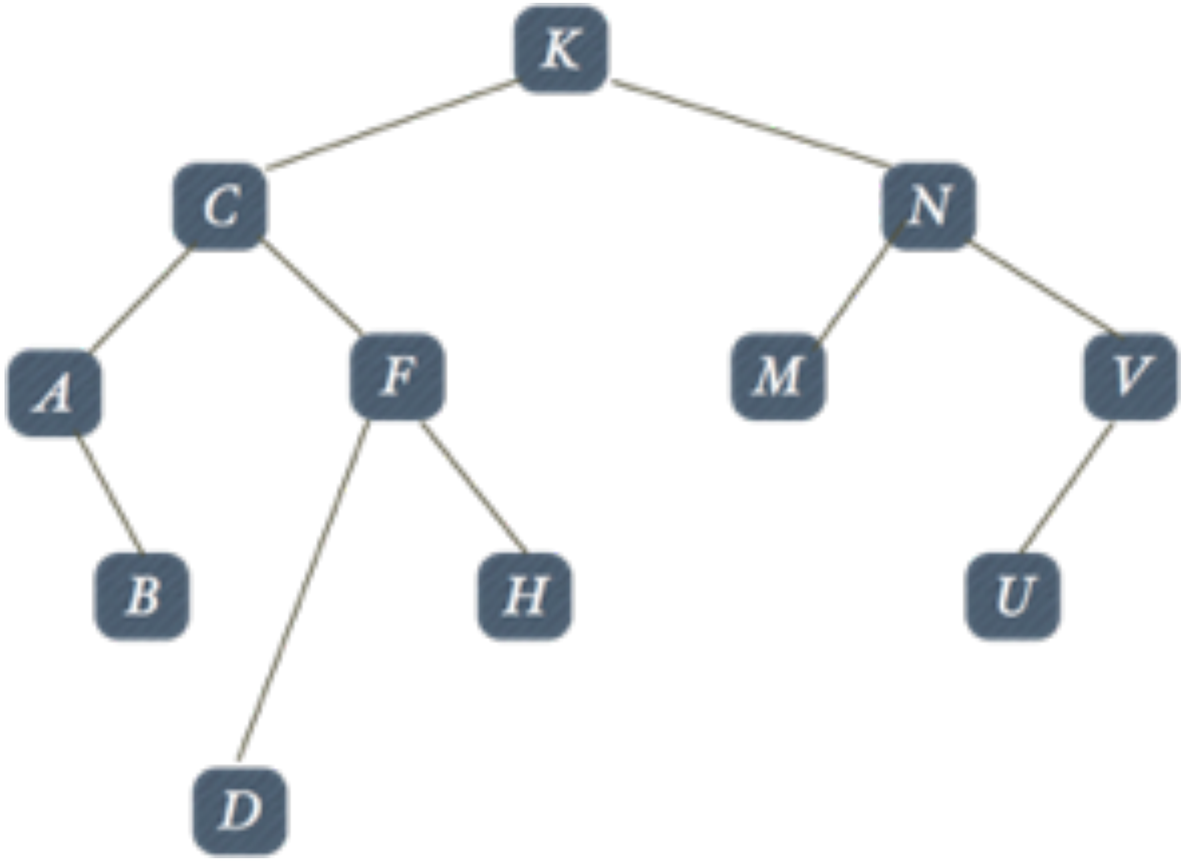
In-order traversal

- Inorder(Tree)
  - Inorder(Left Subtree)
  - Mark root as visited
  - Inorder(Right Subtree)
- A B C D F H K M N U V



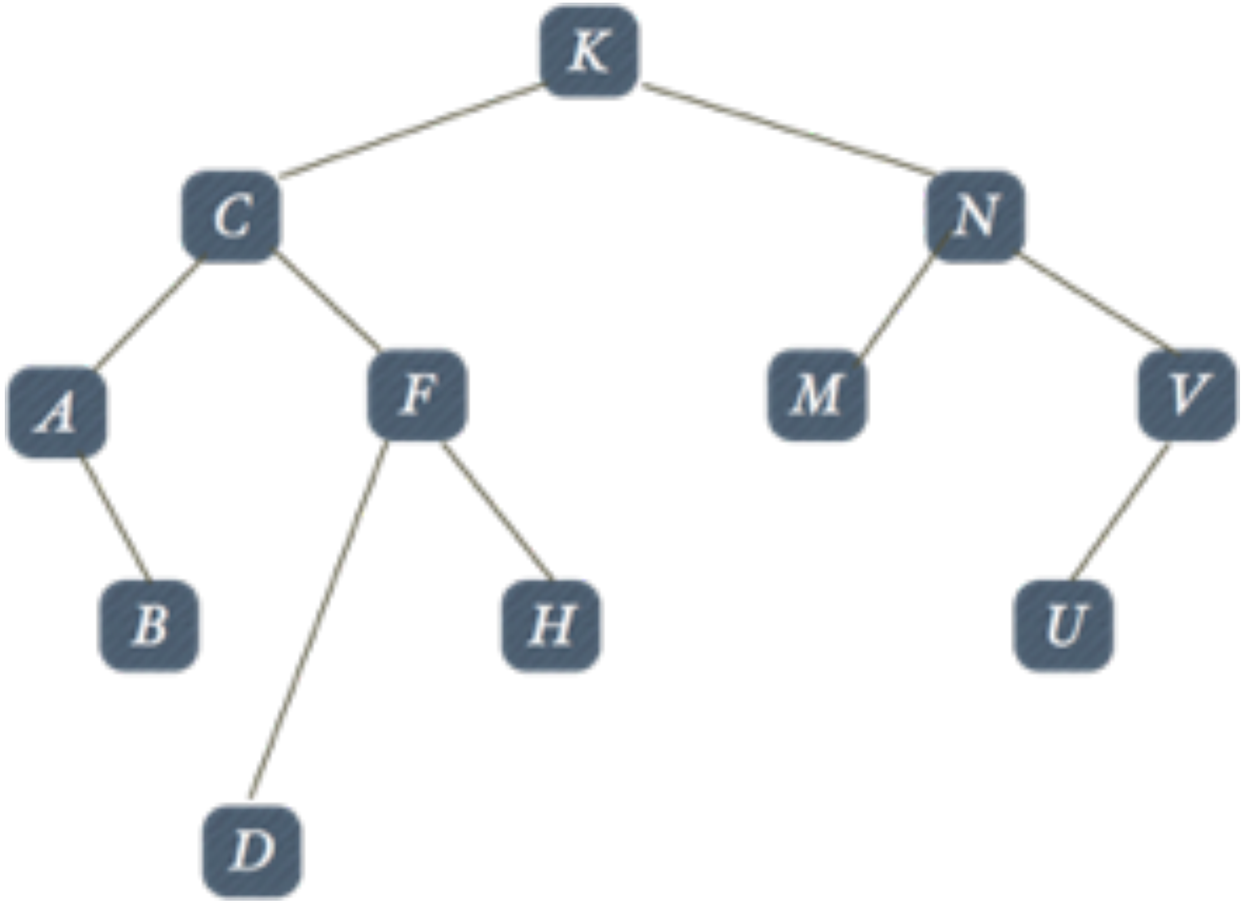
Post-order traversal

- Postorder(Tree)
  - Postorder(Left Subtree)
  - Postorder(Right Subtree)
  - Mark root as visited
- B A D H F C M U V N K



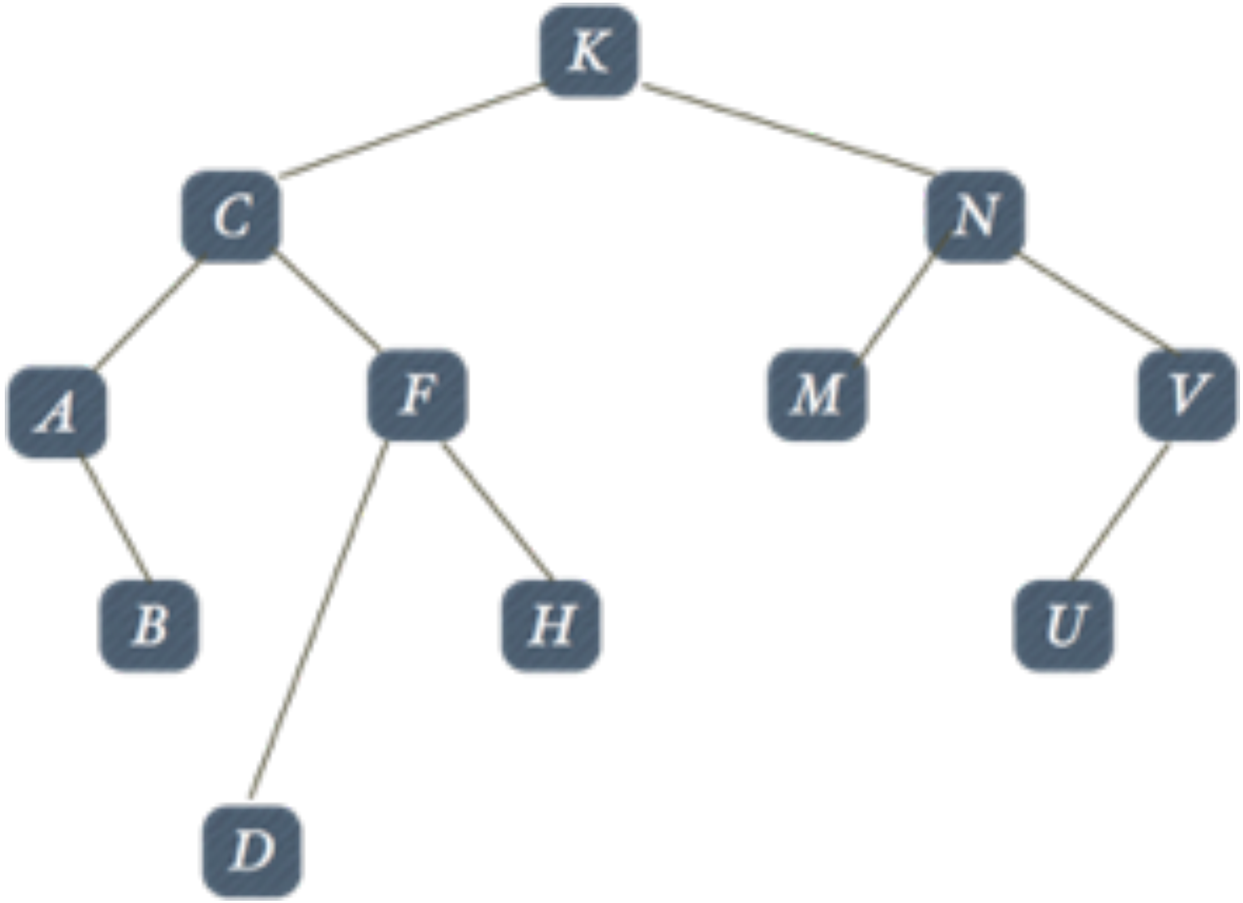
Level-order traversal

- From left to right, mark nodes of level i as visited before nodes in level $i + 1$. Start at level 0.
- K C N A F M V B D H U



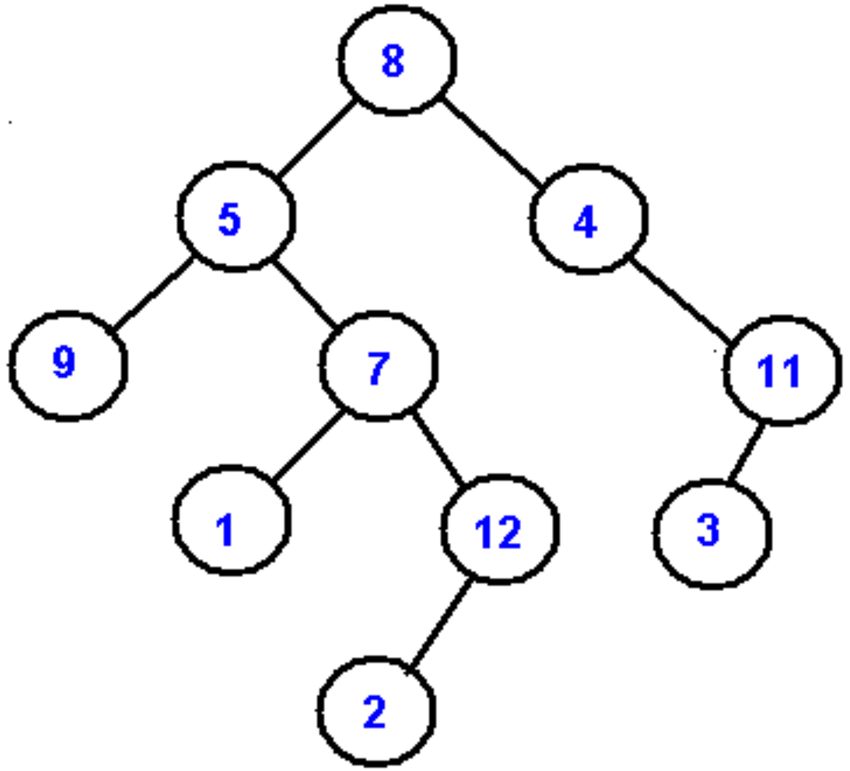
Tree traversal summary

- Pre: Root, left, right
  - K C A B F D H N M V U
- In: Left, root, right
  - A B C D F H K M N U V
- Post: Left, right, root
  - B A D H F C M U V N K
- Level: Go down levels, L->R
  - K C N A F M V B D H U



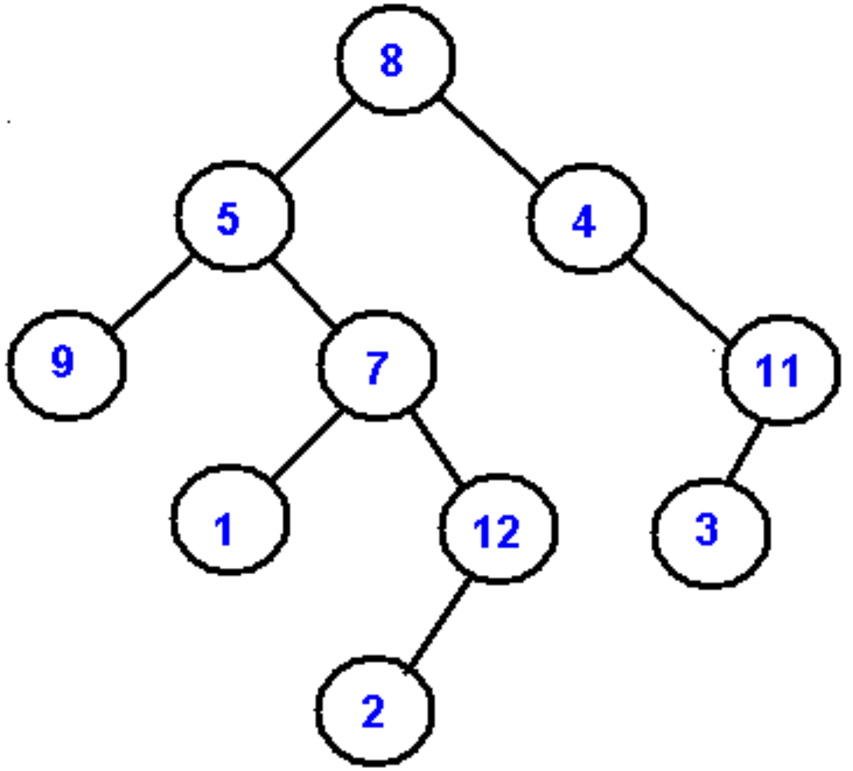
Worksheet time!

- List the nodes in pre-order, in-order, post-order, and level order:



Worksheet answers

- List the nodes in pre-order, in-order, post-order, and level order:



- Pre-order: 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
- In-order: 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
- Post-order: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
- Level-order: 8, 5, 4, 9, 7, 11, 1, 12, 3, 2

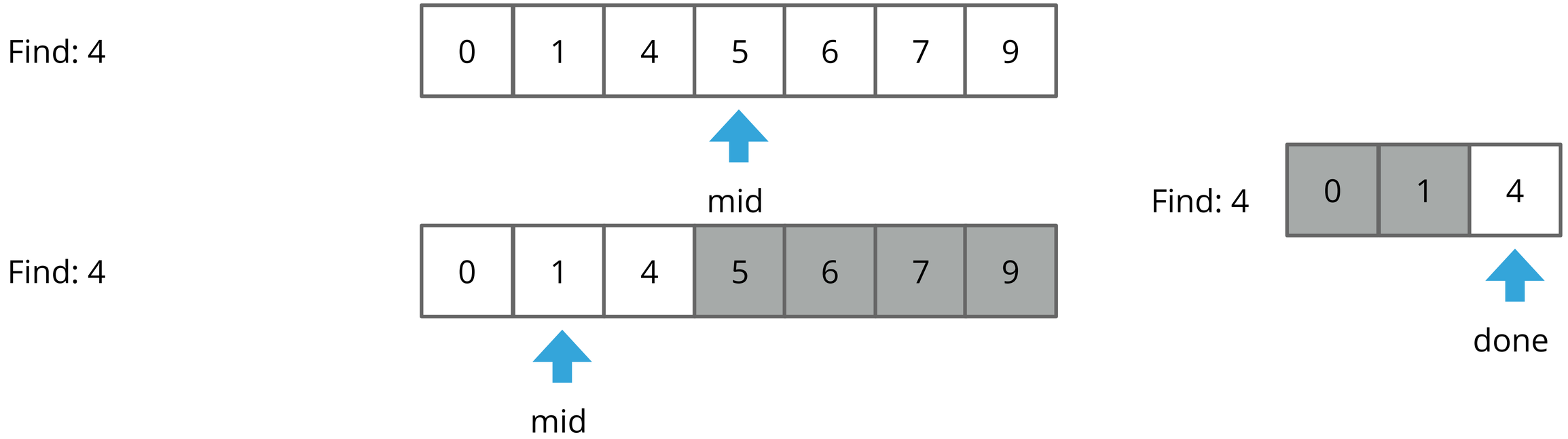
Binary Search

Binary search

- Goal: Given a *sorted* array and an item, find index of the item in the array. E.g., find 3 in [1, 3, 5, 10, 200]. Return -1 if not found.
- Guaranteed O(n) worst case. (Why?)
- We can do better! (How? Hint: Think of the fact that the array is *ordered)*
- Basic mechanism: Compare item against middle entry.
  - If too small, repeat in left half.
  - If too large, repeat in right half.
  - If equal, you are done.

Binary search example

- Goal: Given a *sorted* array and an item, find index of the item in the array.
- Basic mechanism: Compare item against middle entry.
  - If too small, repeat in left half.
  - If too large, repeat in right half.
  - If equal, you are done.



Binary search implementation - iterative

- First binary search published in 1946 but first bug-free in 1962.
- Bug in Java's `Arrays.binarySearch()` discovered in 2006 https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html (the calculation of the mid point) (by our friend Joshua Bloch, creator of ArrayLists!)

```
public static <Item extends Comparable<Item>> int binarySearch(Item[] a, Item item) {
    int lo = 0, hi = a.length-1;
    while (lo <= hi) {
         int mid = lo + (hi - lo) / 2;
         if (item.compareTo(a[mid])<0)
              hi = mid - 1;
         else if (item.compareTo(a[mid])>0)
              lo = mid + 1;
         else return mid; }
    return -1;
}
```

- Uses at most $1 + \log n$ compares to search in a sorted array of size n, that is it is $O(\log n)$.

Binary search implementation - recursive

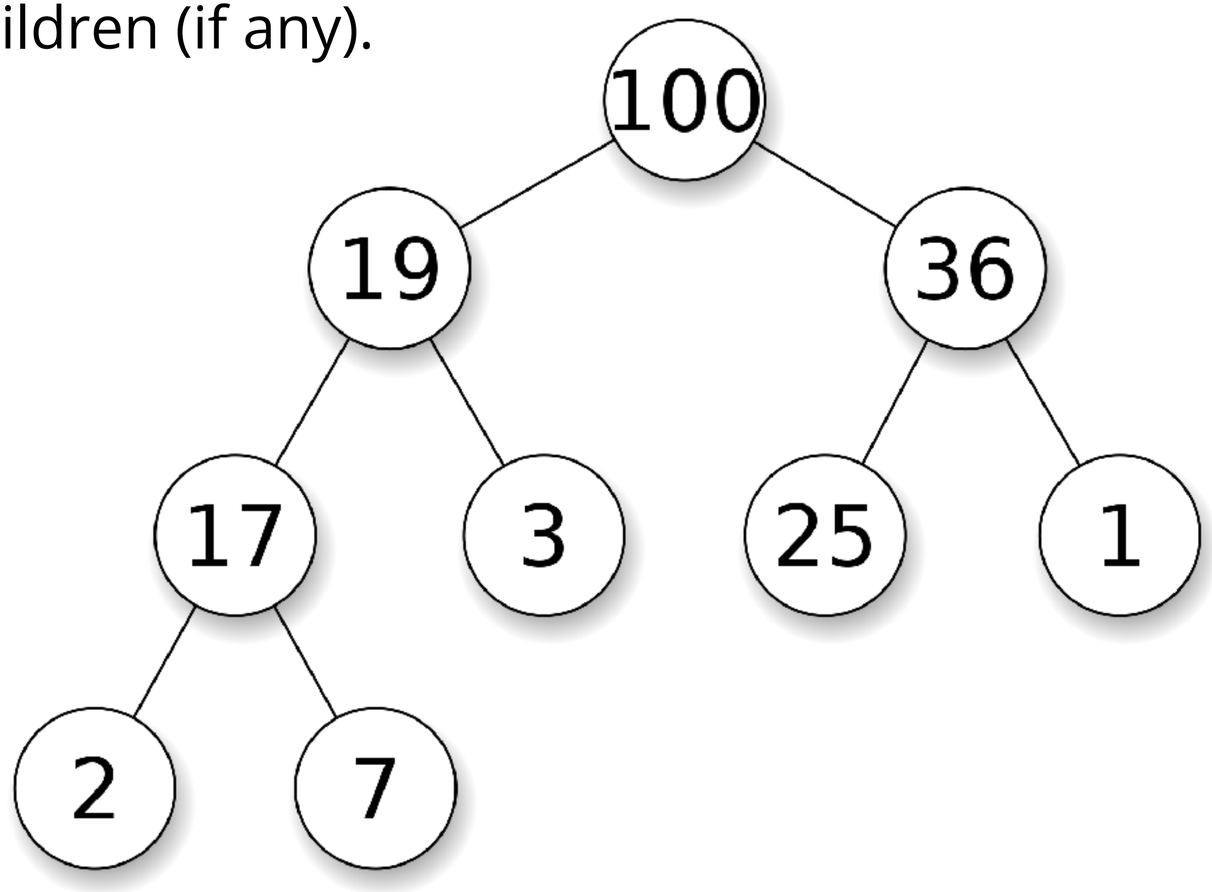
```
private static <Item extends Comparable<Item>> int binarySearch(Item[] a, int lo,
int hi, Item item) {
    if (lo < hi) {
         int mid = lo + (hi - lo) / 2;
         if (item.compareTo(a[mid])<0)
              return binarySearch(a, lo, mid - 1, item);
         else if (item.compareTo(a[mid])>0)
              return binarySearch(a, mid+1, hi, item);
         else return mid; }
    return -1;
}
public static <Item extends Comparable<Item>> int binarySearch(Item[] a, Item item)
{
    return binarySearch(a, 0, a.length-1, item);
}
```

- Uses at most $1 + \log n$ compares to search in a sorted array of size n, that is it is $O(\log n)$.

Binary Heap

Heap-ordered binary trees

- The largest key in a heap-ordered binary tree is found at the root!
- A binary tree is heap-ordered if the key in each node is larger than or equal to the keys in that node's two children (if any).



Heap-ordered binary trees

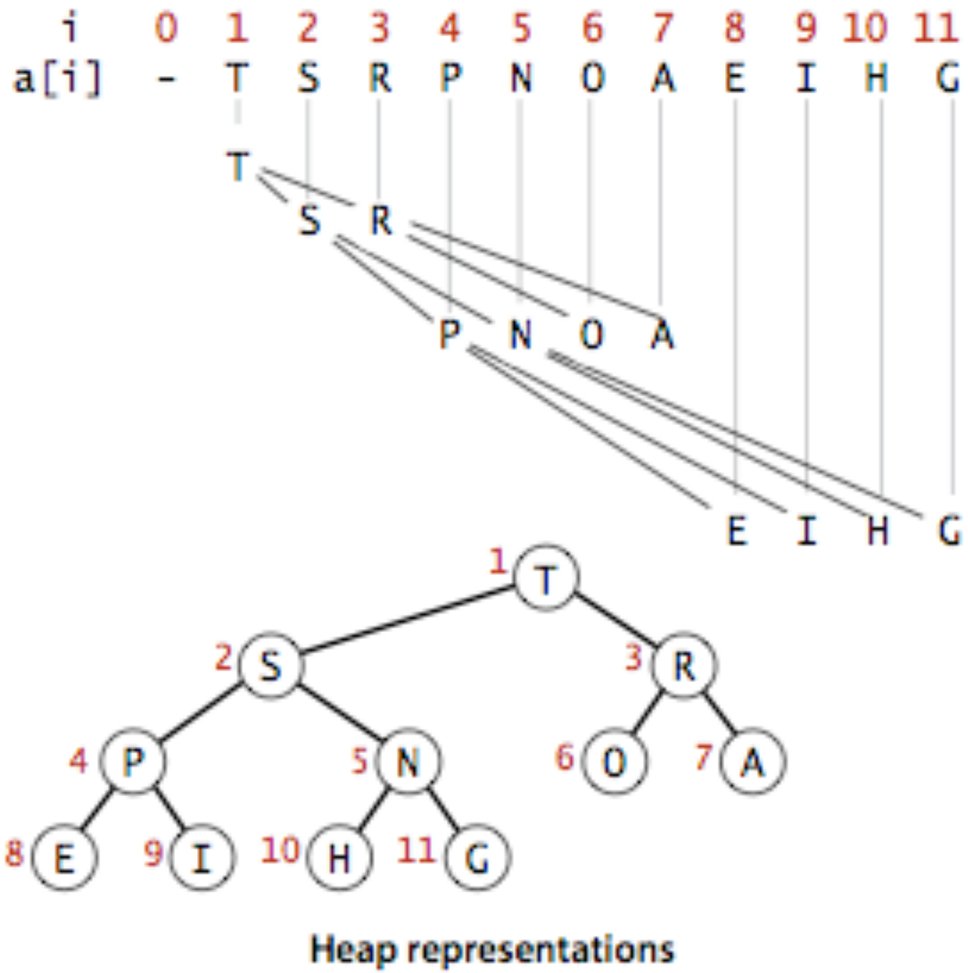
- A binary tree is heap-ordered if the key in each node is larger than or equal to the keys in that node's two children (if any).
  - Specifically, this is called a *max heap* (since the maximum is the root)
- Equivalently, the key in each node of a heap-ordered binary tree is smaller than or equal to the key in that node's parent (if any).
- **No assumption of which child is smaller.**
- Moving up from any node, we get a non-decreasing sequence of keys.
- Moving down from any node we get a non-increasing sequence of keys.

Binary heap representation

- We could use a linked representation like we use for binary trees (this, this.left, this.right), but we would need three links for every node (one for parent, one for left subtree, one for right subtree).
- If our binary tree is *complete* (minimal height), we can use instead an array.
  - Compact arrays vs explicit links means memory savings!

Array representation of heaps

- Nothing is placed at index 0 (for arithmetic convenience).
- Root is placed at index 1.
- Rest of nodes are placed **in level** order.
- No unnecessary indices and no wasted space because it’s a complete heap.



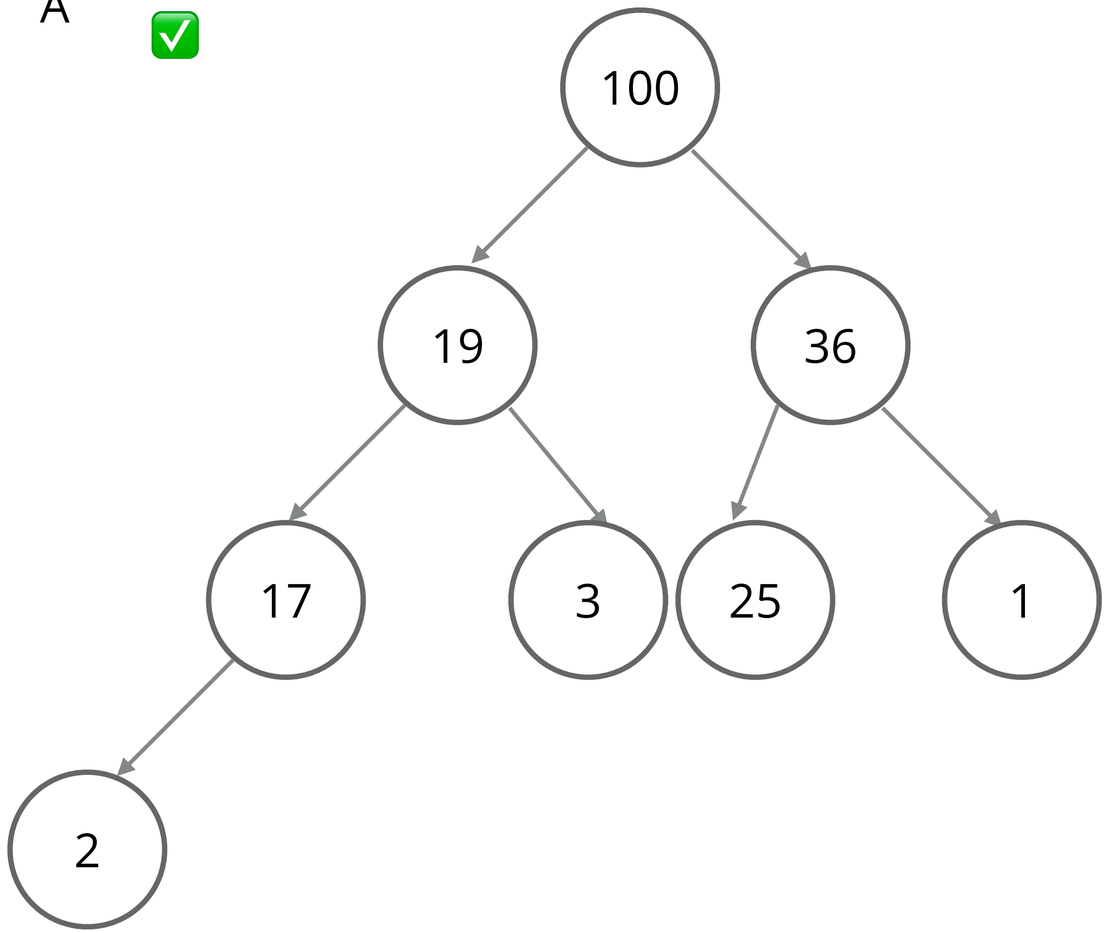
Heap representations

Binary heaps

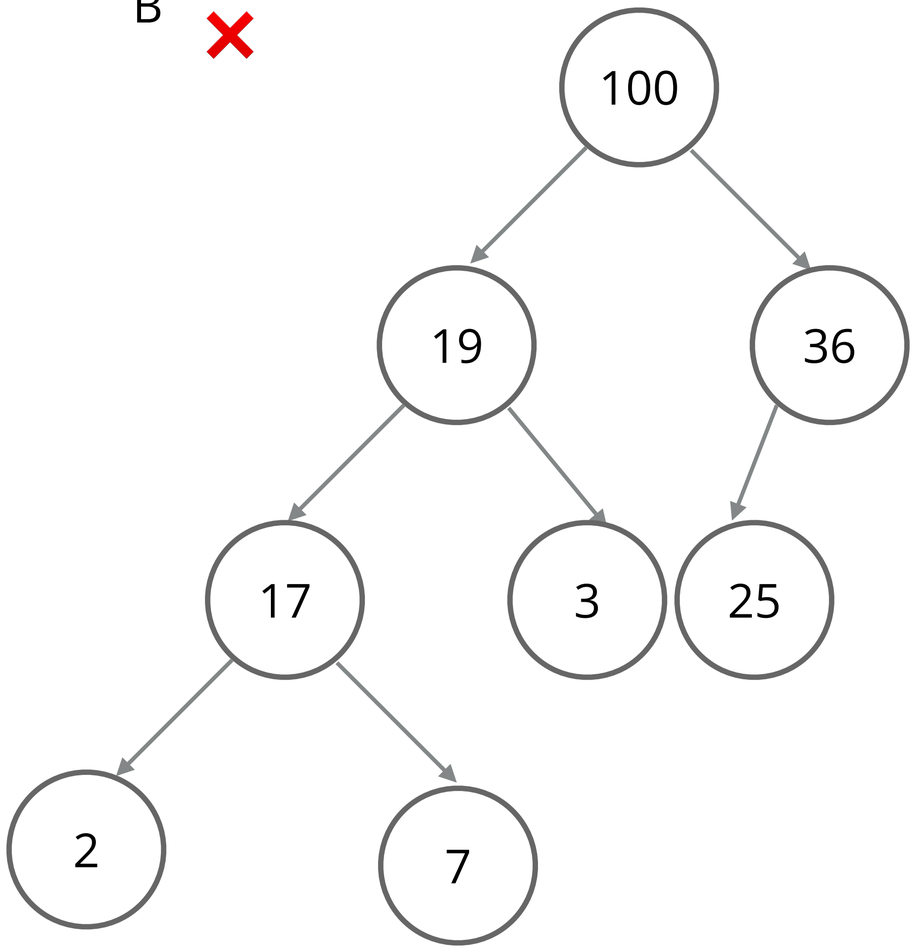
- Binary heap: the array representation of a complete heap-ordered binary tree.
  - Items are stored in an array such that each key is guaranteed to be larger (or equal to) than the keys at two other specific positions (children).
- Max-heap but there are min-heaps (root is smallest), too.

Practice: Which are the following are valid binary heaps?

A ✅

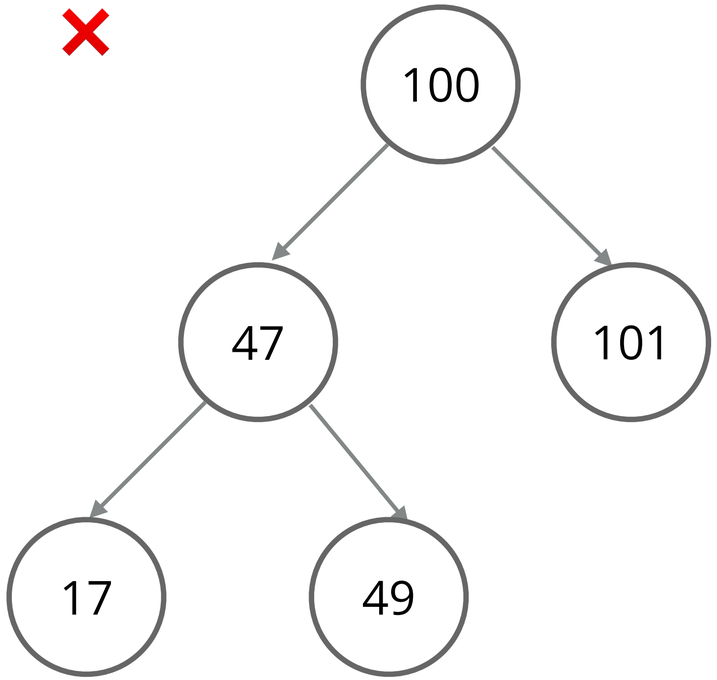


B ❌

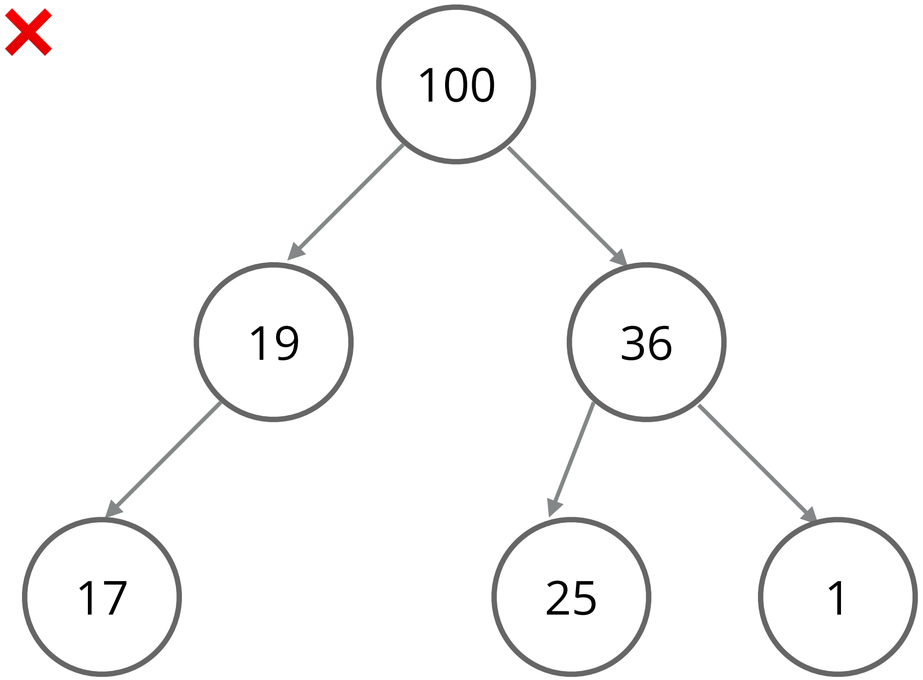


Practice: Which are the following are valid binary heaps?

C ✗

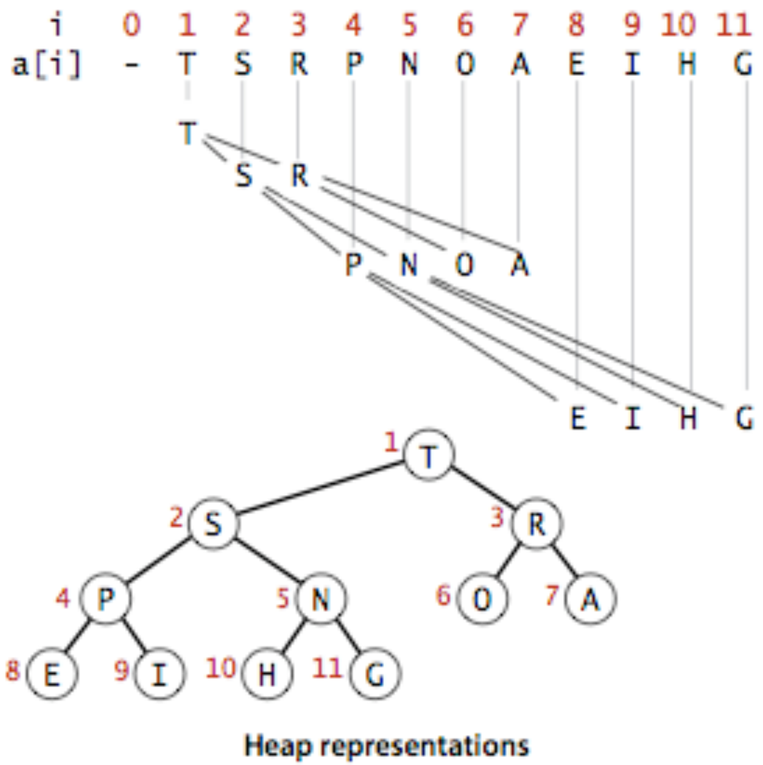


D ✗



Reuniting immediate family members

- For every node at index k, its parent is at index $\lfloor k/2 \rfloor$.
- Its two children are at indices $2k$ and $2k + 1$.
- We can travel up and down the heap by using this simple arithmetic on array indices.



Heap representations

Example: P is at 4, so its parent (S) is at 2

A is at 7, so its parent (R) is at 3 (round down)

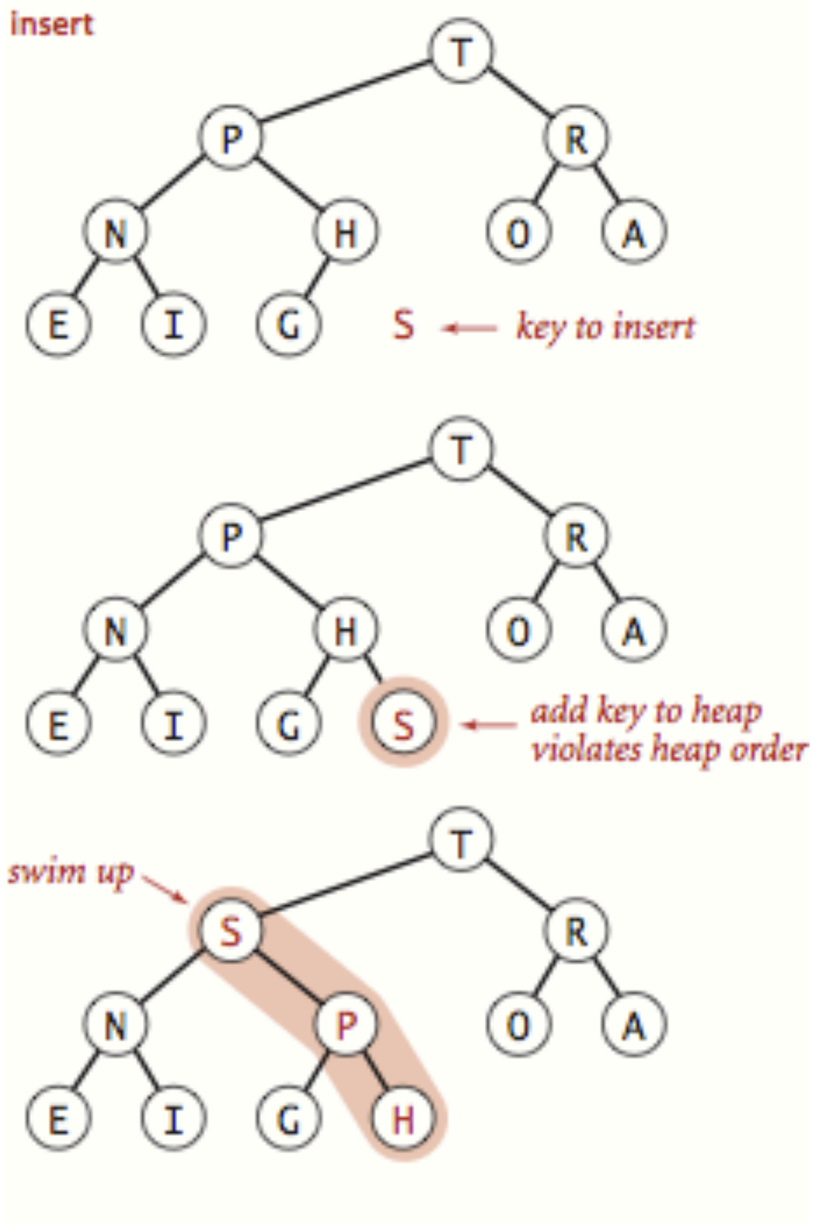
Example: R is at 3, so its children are at 6 and 7 (O & A).

Binary heap: insertion

- Insert: Add node at end in bottom level, then swim it up.
- Cost: At most $\log n + 1$ compares.

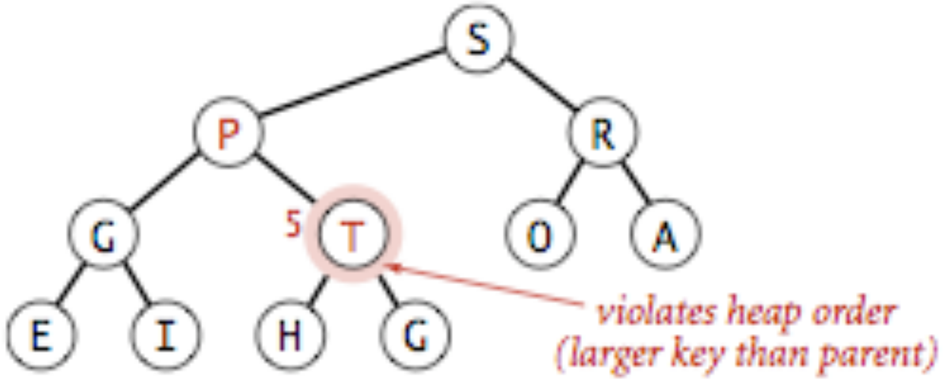
```
public void insert(E x) {
   a[++n] = x;
   swim(n);
}
```

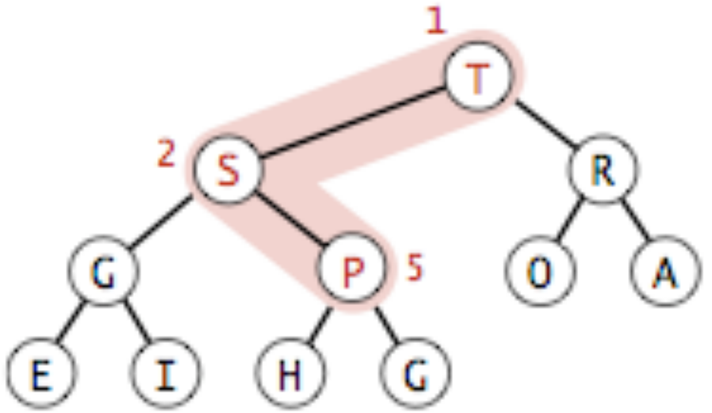
n is current size of array



Swimming: Maintaining heap order

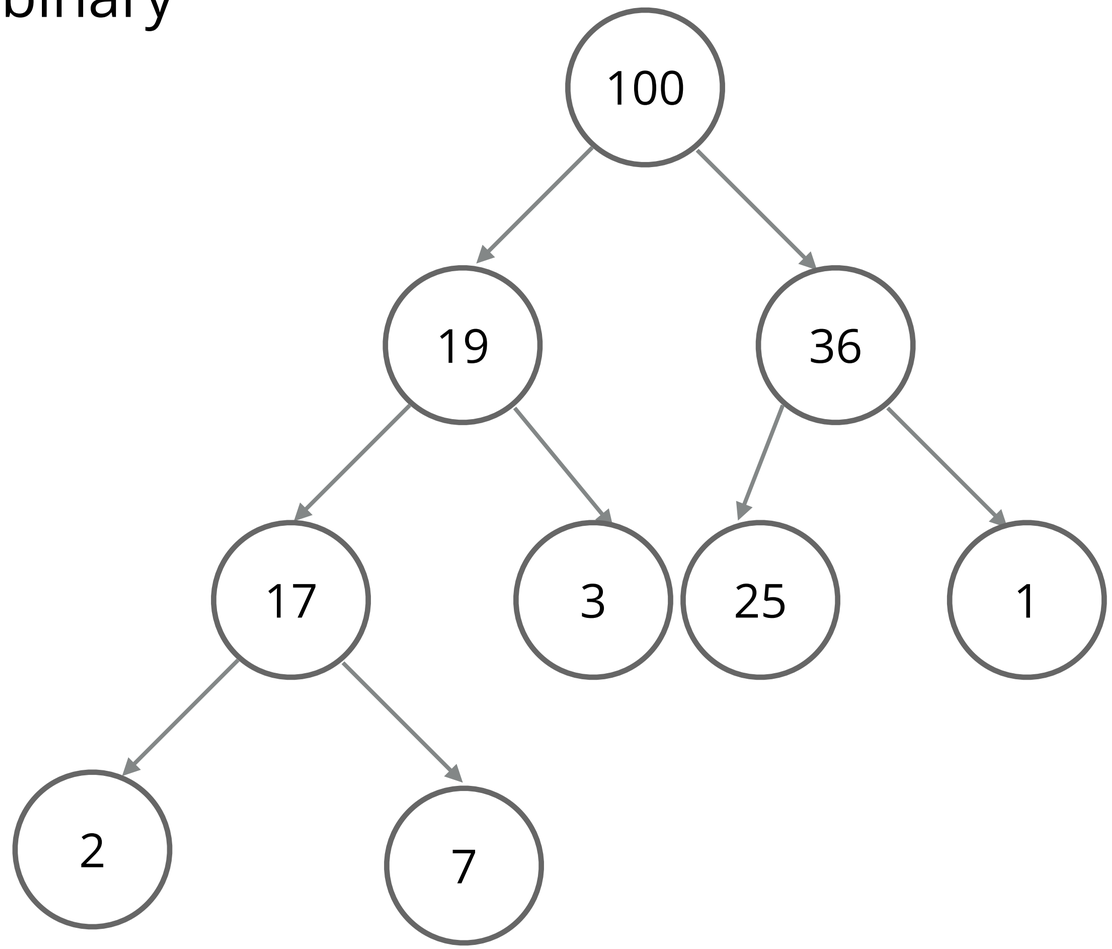
- Scenario: a key becomes larger than its parent therefore it violates the heap-ordered property.
- To eliminate the violation:
  - Exchange key in child with key in parent.
  - Repeat until heap order restored.
- This is called swimming, percolating, promoting up, or bottom up reheapify





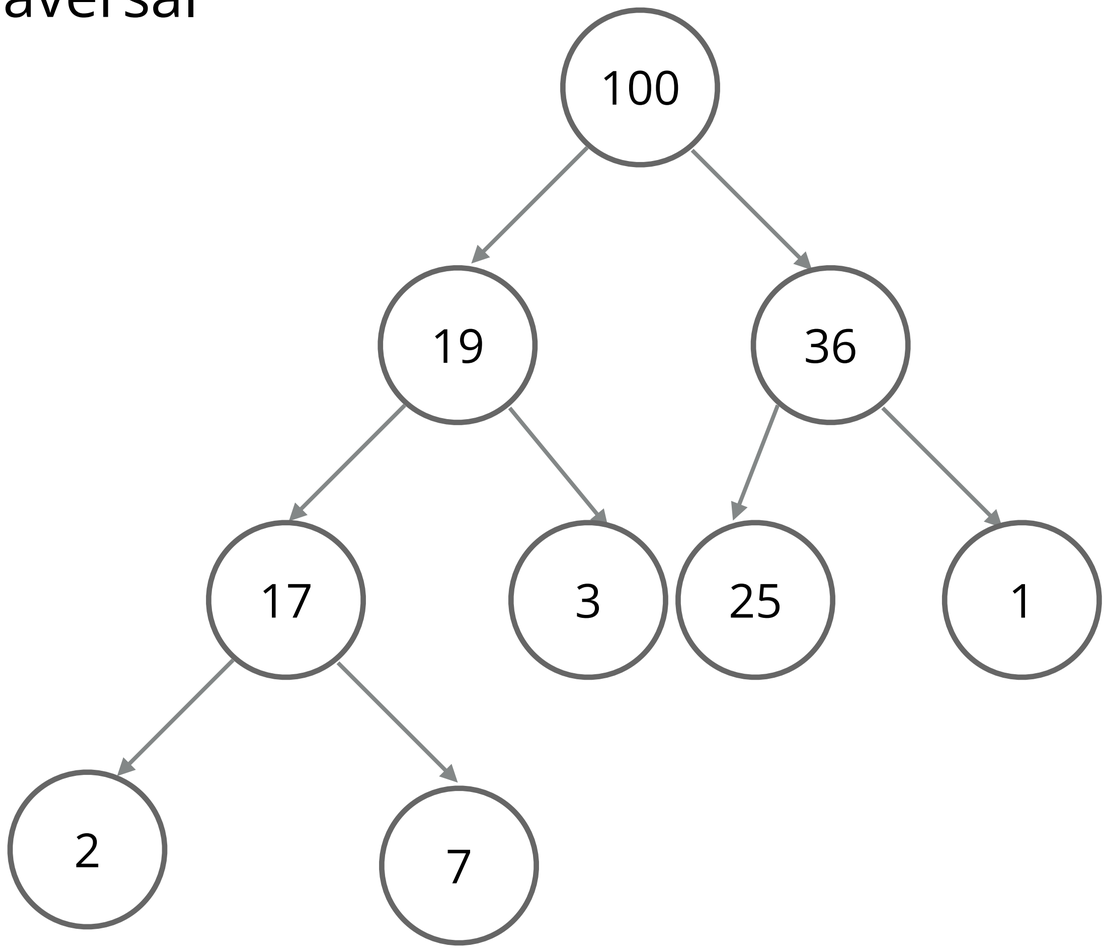
Worksheet time!

- Write the array representation of this binary heap.
- Insert 47 in this binary heap.



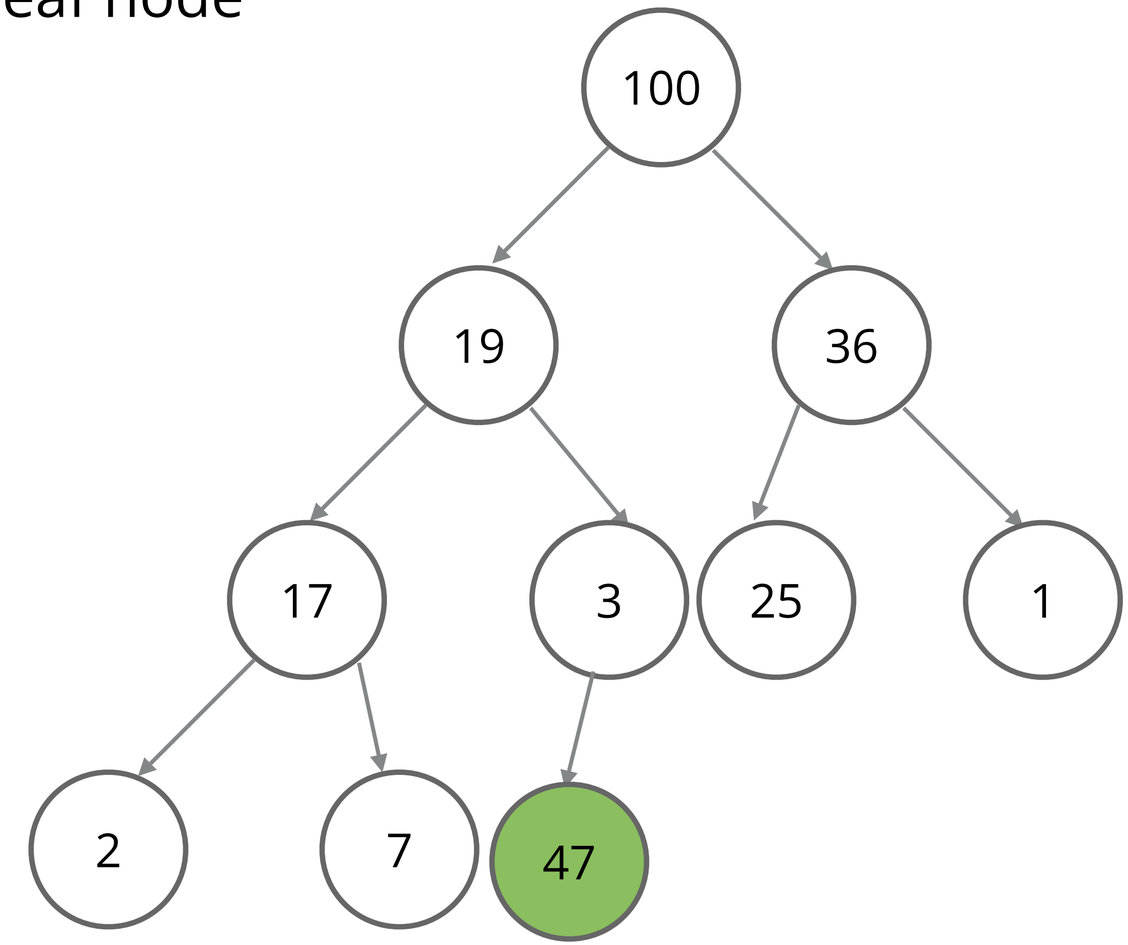
Worksheet answers

- Array representation is just in-order traversal with root first
- [-, 100, 19, 36, 17, 3, 25, 1, 2, 7]
  1 2 3 4 5 6 7 8 9



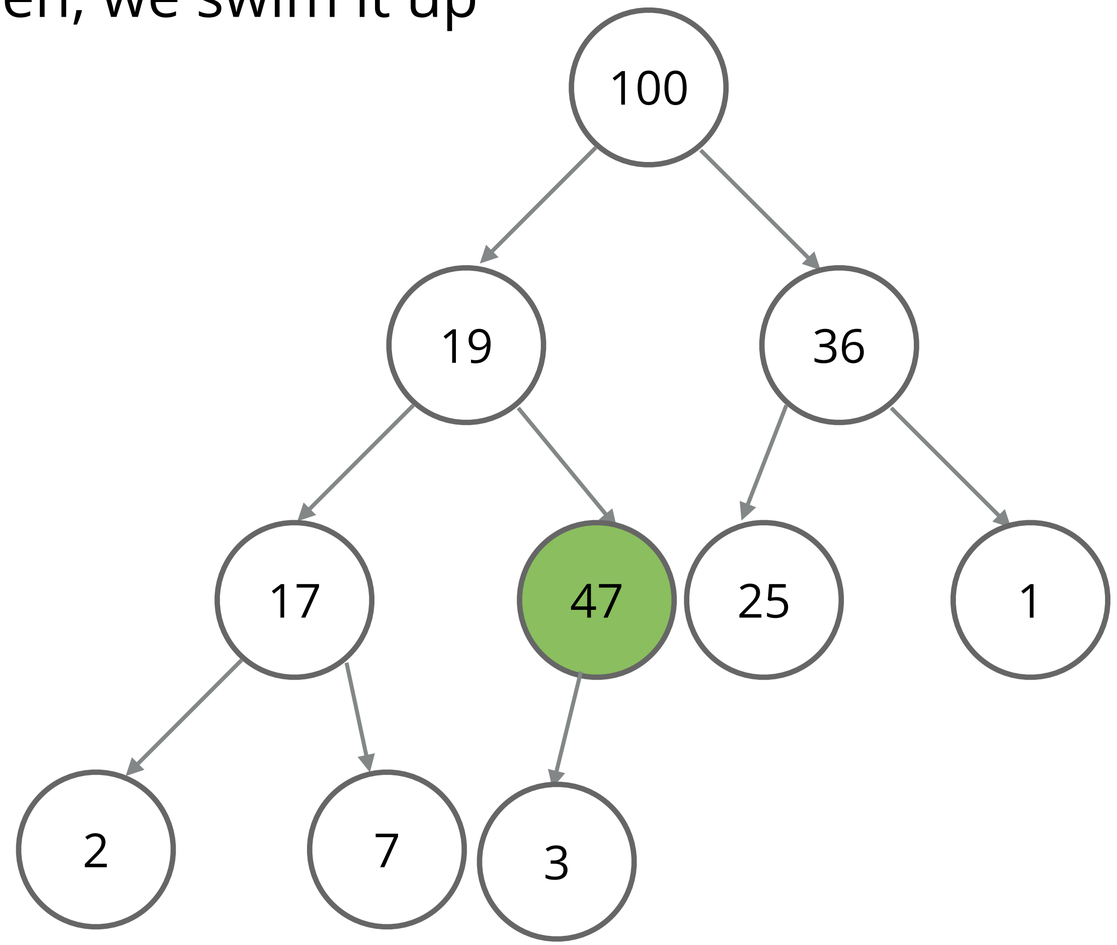
Worksheet answers

- First, 47 needs to go to the left-most leaf node (left child of 3)

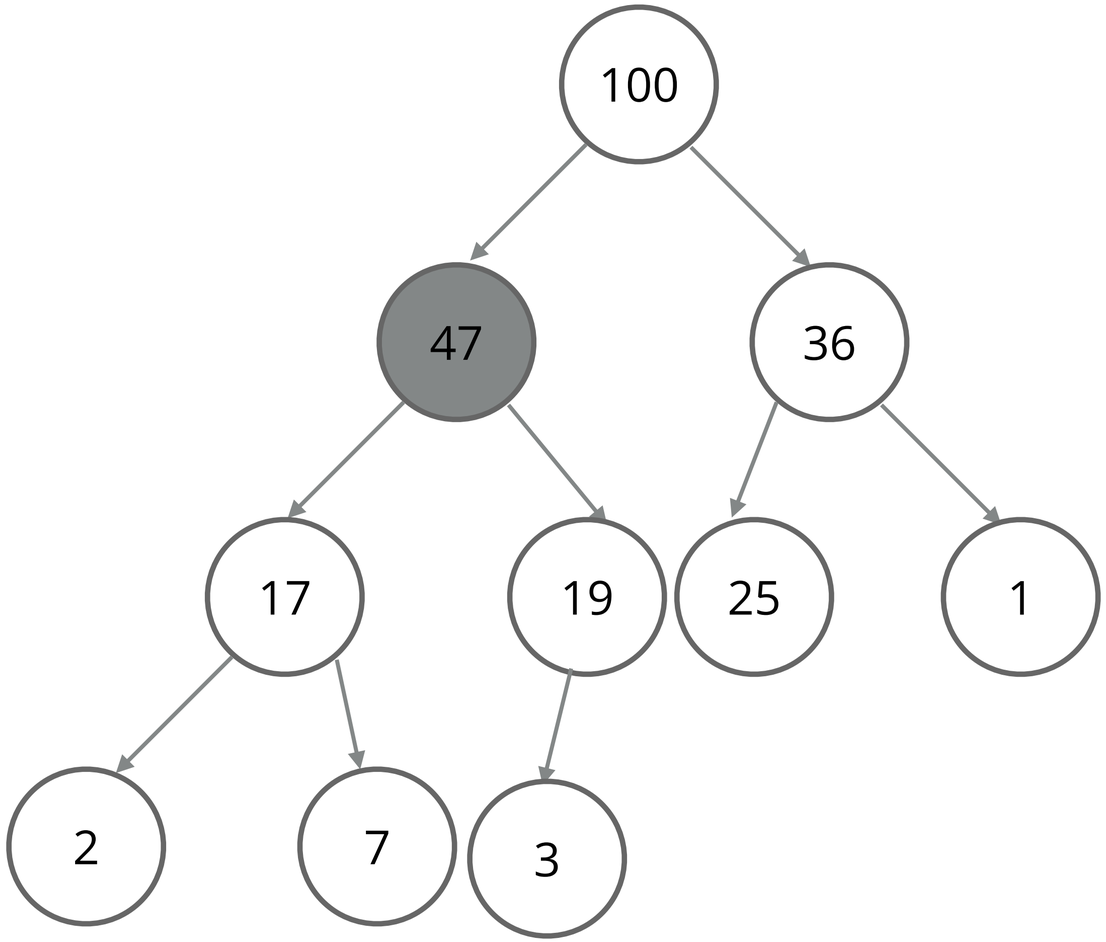


Worksheet answers

- Then, we swim it up



Final heap!

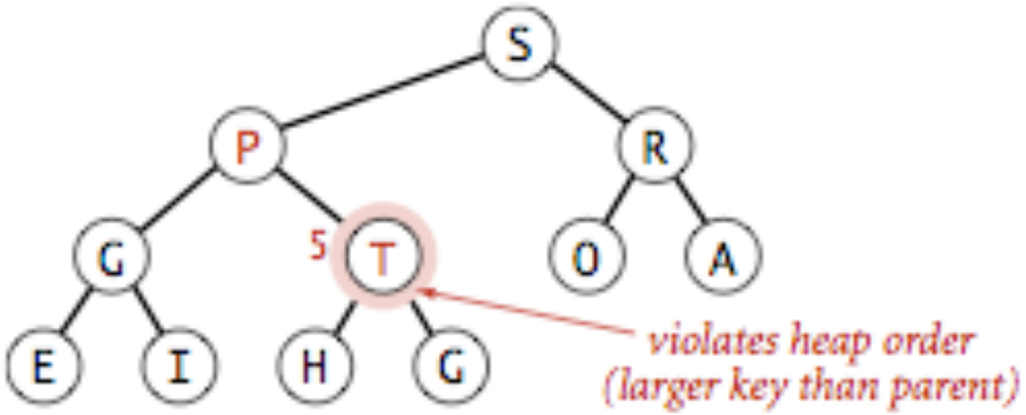


Swim/promote/percolate up: code

```
private void swim(int k) {
   while (k > 1 && a[k/2].compareTo(a[k])<0) {
      E temp = a[k];
      a[k] = a[k/2];
      a[k/2] = temp;
      k = k/2;
   }
}
```

exchange with parent

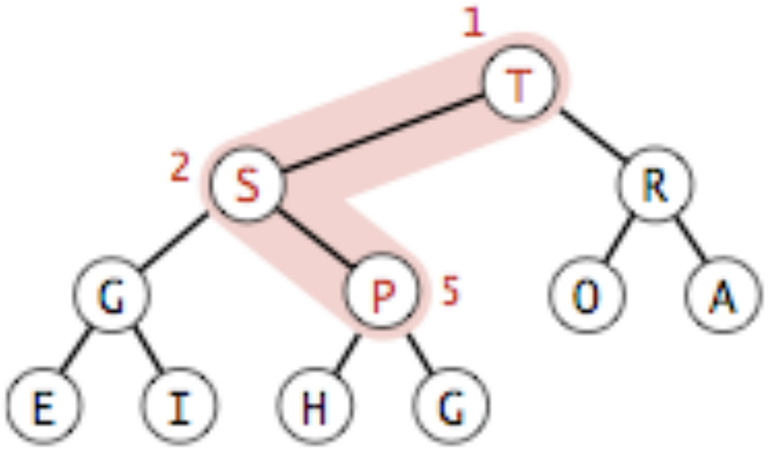
change index to be parent's



End conditions:

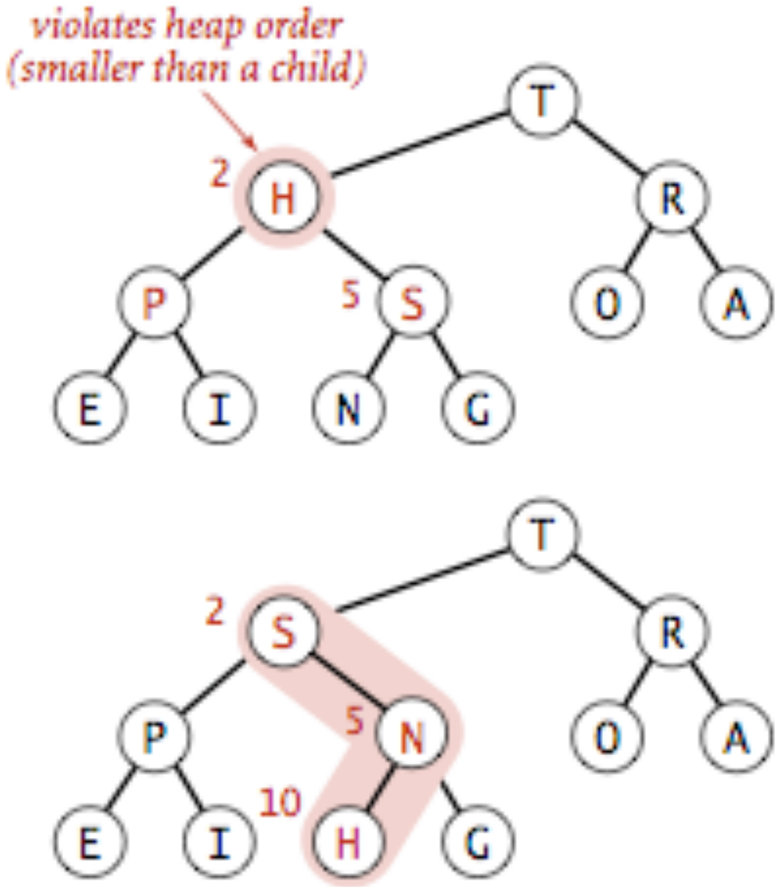
k == 1: it's already at the root

a[k/2] > a[k]: node is smaller than parent



Sink/demote/top down heapify

- Scenario: a key becomes smaller than one (or both) of its children's keys.
- To eliminate the violation:
  - Exchange key in parent with key in **larger** child.
  - Repeat until heap order is restored.



Sink/demote/top down heapify code

```
private void sink(int k) {
    while (2*k <= n) {
        int j = 2*k;
        if (j < n && a[j].compareTo(a[j+1])<0))
            j++;
        if (a[k].compareTo(a[j])>=0))
            break;
        E temp = a[k];
        a[k] = a[j];
        a[j] = temp;
        k = j;
    }
}
```

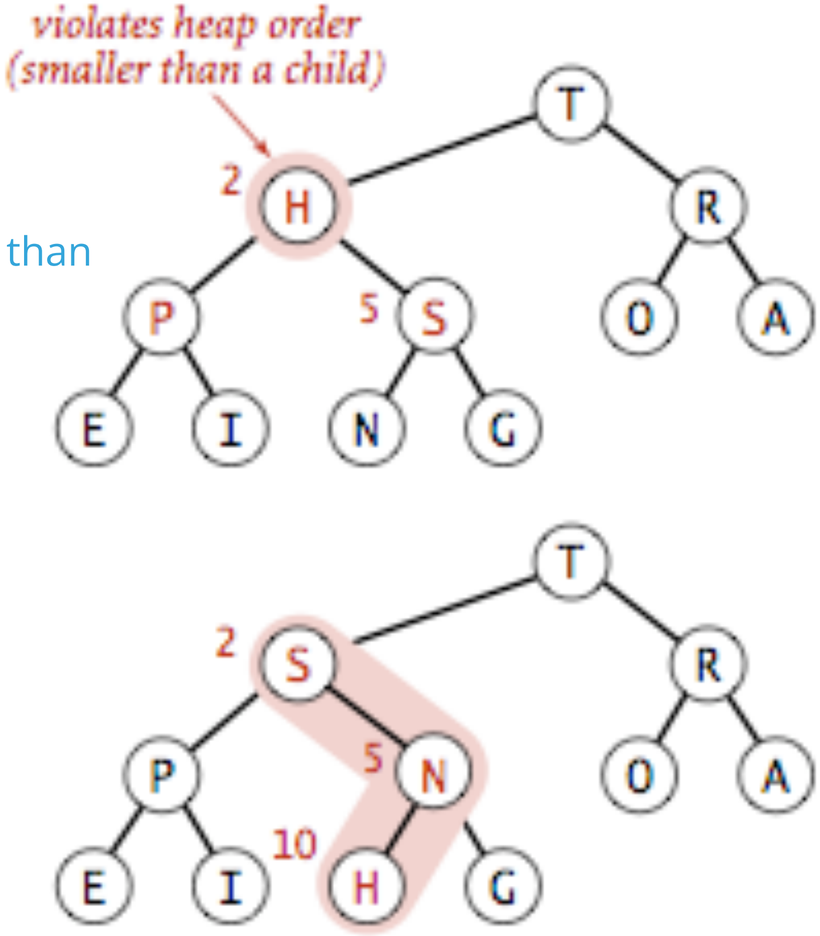
while the left child exists

j is left child

look at j+1 (right child) instead

stop swapping when it's bigger than or equal to child

swap node with correct child



Lecture 13 wrap-up

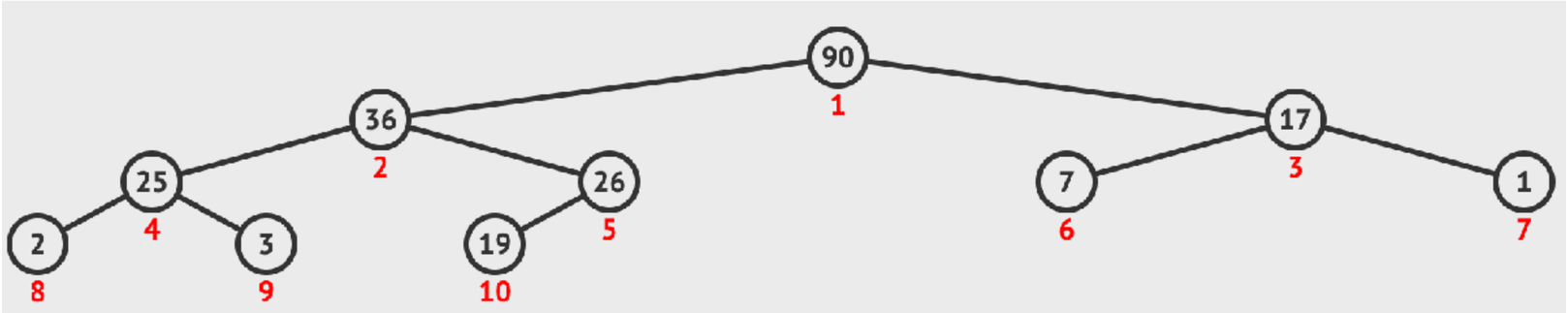
- HW6: On Disk sort due 11:59pm tonight
- Lab tonight on shell scripting - lots of people are absent, so come to the first section if you can!
- HW7: Autocomplete released
- I know this class seems like a lot with all the assignments and stuff - you get a break and there's no HW the week of your checkpoint 2 (Nov 3)
- Special OH for quiz makeups (and other 62 things) Fri 3-4pm

Resources

- Reading from textbook: Chapter 2.4 (Pages 308-327)
- Heap visualizations: Insert and ExtractMax: https://visualgo.net/en/heap
- Online textbook website - https://algs4.cs.princeton.edu/24pq/
- Practice problems behind this slide

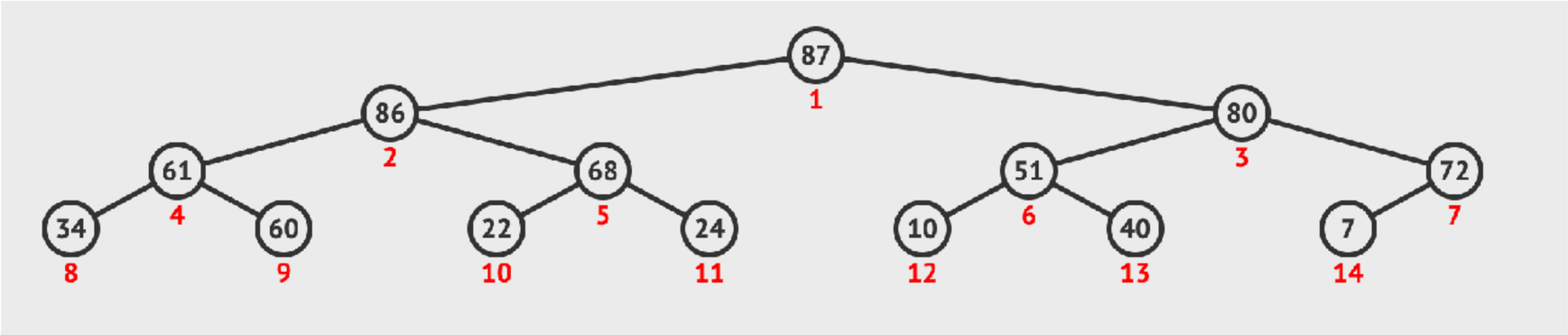
Practice Problem 1

- Given the tree below, list the nodes in order of visit in a:
  - pre-order traversal
  - in-order traversal
  - post-order traversal
  - level-order traversal



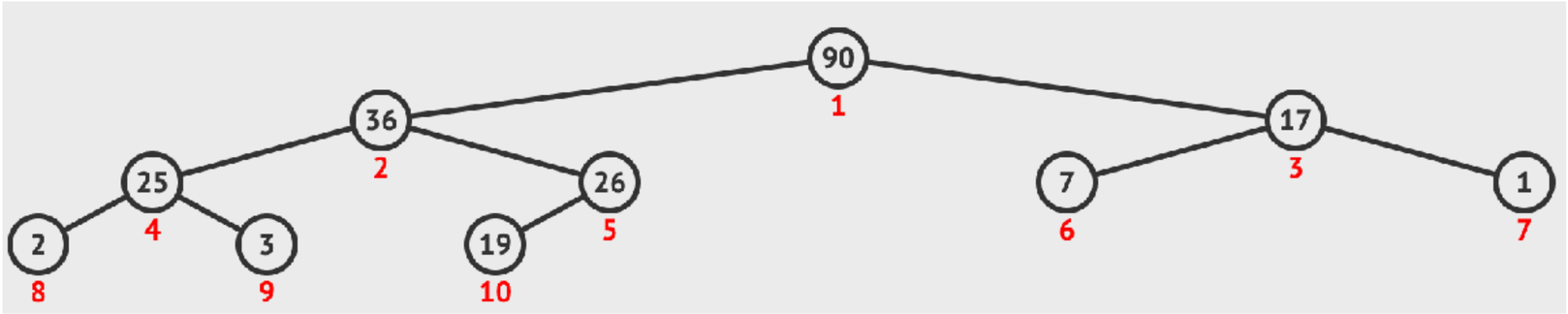
Practice Problem 2

- Given the binary heap below, delete and return the max.



ANSWER 1

- pre-order: 90, 36, 25, 2, 3, 26, 19, 17, 7, 1
- in-order: 2, 25, 3, 36, 19, 26, 90, 7, 17, 1
- post-order: 2, 3, 25, 19, 26, 36, 7, 1, 17, 90
- level-order: 90, 36, 17, 25, 26, 7, 1, 2, 3, 19



ANSWER 2

- Given the binary heap below, delete and return the max.