SORTING

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING 16: Binary Trees, Binary Search, Heaps, and Priority Queues

Alexandra Papoutsaki
she/her/hers

Today, we will start working toward the last sorting algorithm we will encounter in this course: heapsort. To do so though, we will need to learn about a new data structure, the tree. We will specifically focus on binary trees and a subcategory of them which are called heaps.

Lecture 16: Binary Trees, Binary Search, Heaps, and Priority Queues

- Binary Trees
- Tree traversals
- Binary Search
- Binary Heaps
- Priority Queues

All the data structures that we have seen so far (arrays, arraylists, queues, and stacks) have been linear: one element follows another and there is a logical start and end to them.

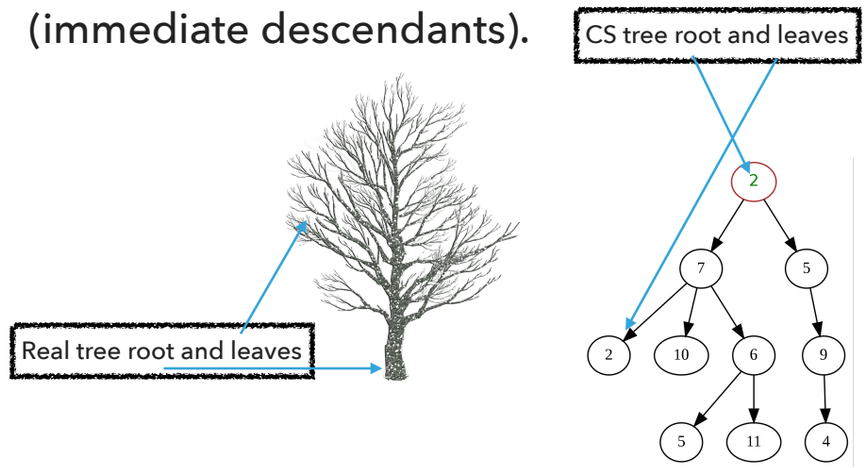
Trees in Computer Science

- 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.).

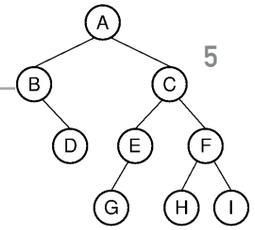
- Trees in Computer Science are abstract data types that store elements hierarchically. There are tons of examples out there that the linear data structures we have encountered so far wouldn't be appropriate to model them.
- For example, you might have encountered organization charts which in essence are trees.
- Companies have at the top the CEO who's followed by CFO, CMO, COO, CTO, etc and each one of these manages a different set of people.
- The same applies to universities, where the board of trustees sits at the top, followed by the president, who's followed by VPs, etc.
- If you visit a website, you can model its sitemap as a tree, with the homepage at the top followed by the about, products, contact etc. page. Each one of these pages links to another page.
- Finally, when you navigate your computer file system you might have noticed that it has a hierarchical structure. You as the user are at the top and below you there are directories such as Documents and Downloads. Each one of these can have more directories.

Trees in Computer Science

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



- Now, I said that trees in computer science store elements hierarchically. Formally, by hierarchically, we mean that each element in a tree has a **single** parent (immediate ancestor) and zero or more children (immediate descendants). In general, an element can be followed by many elements and that number can vary.
- You can imagine trees in CS like real-life trees that grow upside down, with the root being at the top, and the leaves at the bottom.
- Given the single parent relationship, we can see why a family tree would practically violate this property. Or why if our website has two pages pointing to the same page we can't use a tree to model its sitemap. For these cases, rather than using trees we will use another hierarchical data structure, the graph.

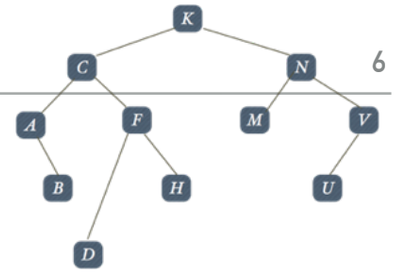


Definition of a tree

- 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.

- Let's examine the definition of a tree. A tree is a set of nodes (visually represented as circles) that store elements organized based on a parent-child hierarchical relationship.
- If a tree is non-empty, it has a unique node called the root of the tree that has no parent. In our example, the root would be the node A.
- Every other node, will have a unique parent node and it will be its child. For example, here, E's parent is C and F has two children H and I.

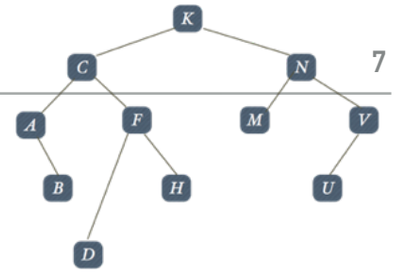
Tree Terminology



- 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.
- A set of trees is called a forest.

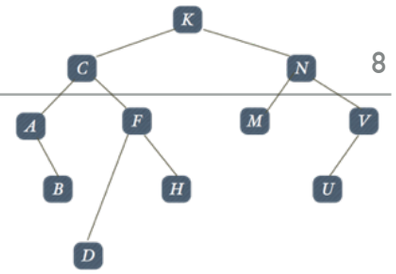
- With this, we need to establish some tree terminology.
- A parent and a child node (for example K and C) are connected by an edge (a line in the visual representation).
- The parent is directly above the child (for example K is the parent of C and N nodes).
- Two nodes that have the same parent are called siblings (for example A and F are siblings because they share the same parent, C).
- Following family tree terminology, we have ancestors and descendants that we can specifically call parents, grandparents, and so on and children, grandchildren, etc.
- If we take a node plus all its descendants then we get a subtree. For example, a subtree rooted at node C would consist of C, A, B, F, D, and H.
- If a node does not have a descendant, it is called a leaf or an external node. All other nodes are called internal.
- If you have more than one tree, you have a forest!

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.

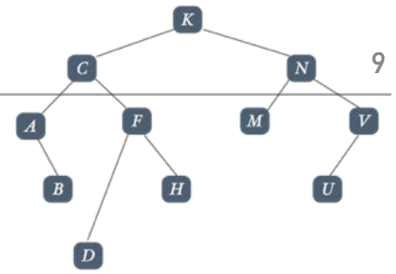
- A simple path is a sequence of distinct connected nodes, for example K-N-V-U.
- The path length is the number of edges in the path, for example the path K-C-A has length 2.
- The height of a node is the length of the longest path from that node to a leaf. e.g., For N there are two possible paths to leaves, one has length 1 (N-M) and the other has length 2 (N-V-U). Therefore, the height of N is 2.
- Similarly, the height of the entire tree is the length of the longest path from the root to any leaf. Here the height would be 3.
- The degree of a node is the number of its children.
- If we examine all nodes in a tree, the maximum degree of any of its nodes is the degree of a tree or its arity. Which brings us officially to binary trees.
- Binary trees are trees with arity of 2. That means that any node in a binary tree will have 0, 1, or at most 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 node defined recursively:
  - If leaf, height is 0.
  - Else, height is max height of child + 1.

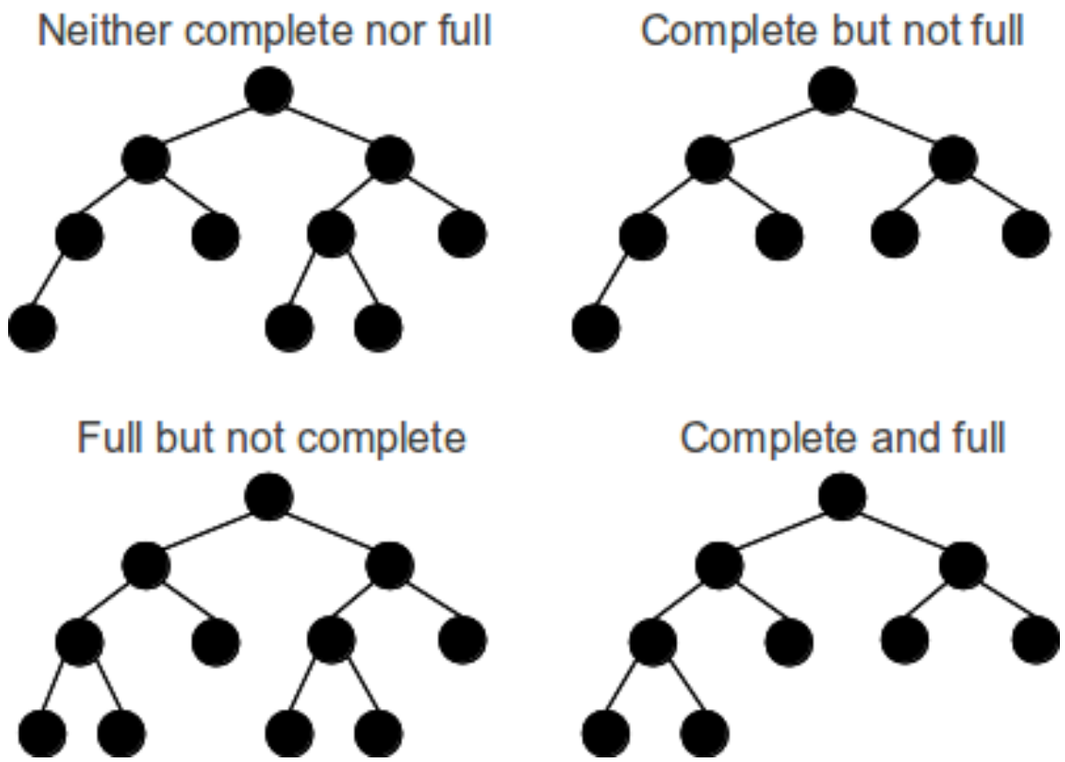
- The level or depth of a node is the number of ancestors excluding itself or the length of the path from the root to that node. We can see this definition recursively:
- The depth/level of the root is 0. Any other node's level is equal to the level of the parent + 1
- We saw before, that the height of a node is the length of the longest path from the node to the tree. Similarly to the recursive level definition, the height of a node is 0 if that node is a leaf, otherwise, it is the maximum height of all of its children + 1.



But wait there's more!

- 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 right, i.e., all nodes of last level are as left as possible.

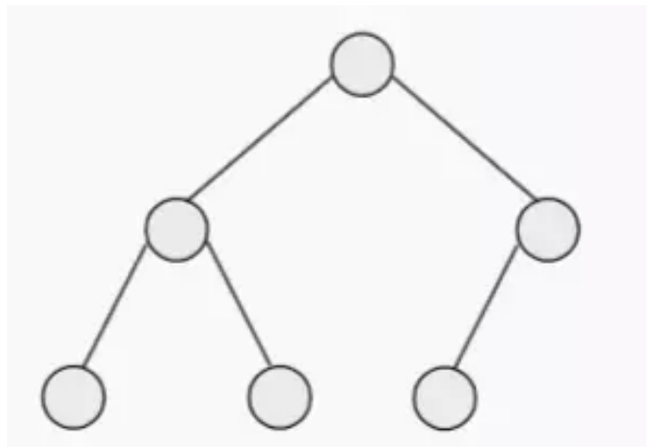
- And to conclude our definitions, a full or proper binary tree is a binary tree whose every node has 0 or 2 children exactly.
- A complete tree is a binary tree of minimal height where any hole in the tree appear at the last level and as much to the right as possible.
- I know that these definitions can be hard to process so let's see them in action.



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

- Remember, a full (or proper) binary tree has nodes with 0 or 2 children exactly while a complete tree has minimal height and all nodes in the last level are as left as possible.
- Let's take the top-left tree. It is neither complete nor full. It is not full as it has a node that has a single child. It is not complete because although its height is minimal, at the last level, not all the nodes are as left as possible.
- In contrast, the top-right tree is complete. It is of minimal height and the last level has a single node that is as left as possible. But it is not full because the parent of that node has 1 child only.
- Applying the same logic, you can see why the bottom left tree is full but not complete and the bottom right tree is both complete and full.

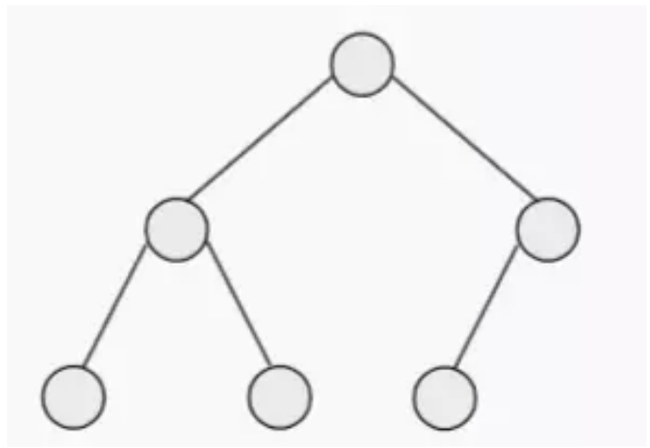
Practice Time: This tree is



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

Now, for some practice time, look at this tree and vote whether it is full, complete, full and complete, neither full nor complete.

Answer

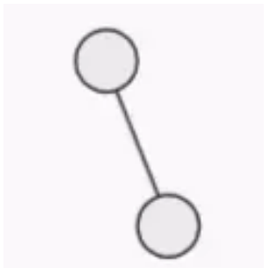


- A: Full
- **B: Complete**
- C: Full and Complete
- D: Neither Full nor Complete

The correct answer is B. The tree is complete but it is not full as there is a node with a single child.

Practice Time: This tree is

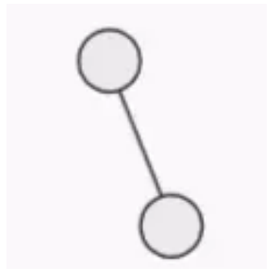
- A: Full
- B: Complete
- C: Full and Complete
- D: Neither Full nor Complete



Let’s apply the same question to this tree. What is the correct option?

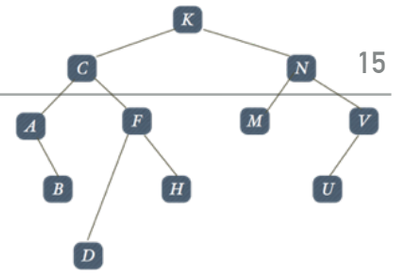
Answer

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



The correct answer is that it is neither full nor complete. It is not full because the root has a single child. It is not complete because the leaf should have been the left not the right child of the root.

Counting in binary trees

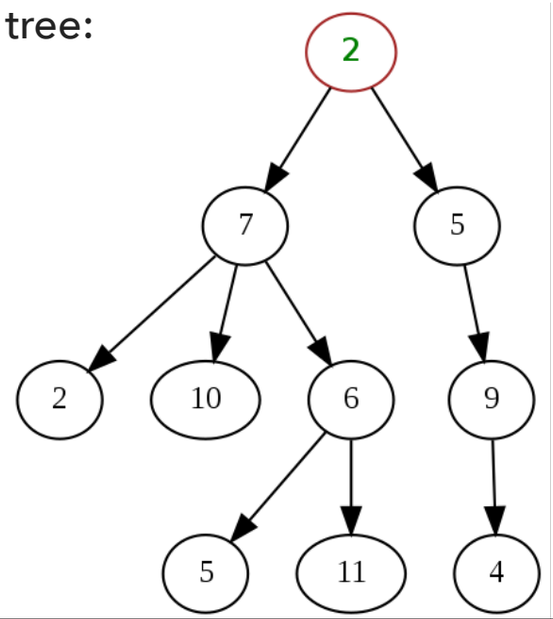


- Lemma: if T is a binary tree, then at level k, T has $\leq 2^k$ nodes.
  - E.g., at level 2, at most 4 nodes (A, F, M, V)
- Theorem: If T has height h, then # of nodes n in T satisfy: $h + 1 \leq n \leq 2^{h+1} - 1$.
- Equivalently, if T has n nodes, then $log(n + 1) - 1 \leq h \leq n - 1$.
  - Worst 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.

- Now I want us to see some important properties of binary trees. We won't be doing the proofs but you can do them if you have missed induction from CS54 :)
- The first property comes from a lemma that says that for a specific level k there are at most 2^k nodes. Remember, the level of root is 0. At level 0 we have at most 2^0=1 nodes (K). At level 1 we have at most 2^1 nodes (C and N). At level 2 we have at most 2^2 nodes (A,F, M,V), and so on.
- The next property of binary trees comes from a theorem that works on the height of trees. Remember the height of a tree is the length of the longest path from the root to a leaf. Rather than focusing on the number of nodes n that a tree of height h can have, I want us to focus on its equivalent property:
- If you are given n nodes to place in a tree, the tree with the maximum height you will get is n-1 which would look like a "stick". The minimum will look like a complete tree (although there is freedom in where nodes can be placed in the last level) and that will be O(logn) height.
- You can now see why if we are given a "stick" the number of nodes will be equal to h+1 and if we are given a minimal height tree like a complete tree it will be 2^(h+1)-1.

Practice Time - Problem 1 Worksheet #16

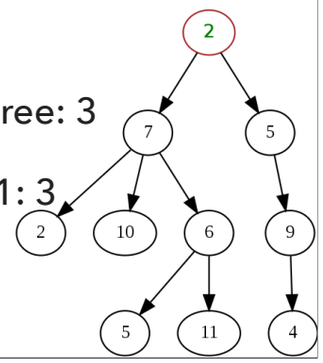
- Follow the instructions in the worksheet about the following tree:



- Let's put everything in practice by answering some questions about the provided tree

ANSWER 1 - Worksheet #16

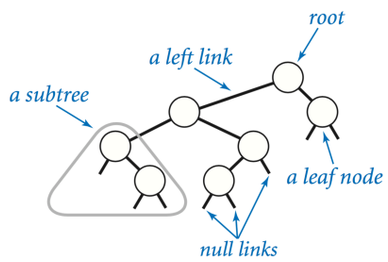
- Root: 2
- Leaves: 2 (in black), 10, 5, 11, 4
- Internal nodes: 7, 5, 6, 9
- Siblings of 10: 2, 6
- Parent of 6: 7
- Children of 2 (in red): 7, 5
- Ancestors of 10: 7 and 2 (in red)
- Descendants of 7: 2, 10, 6, 5, 11
- Length of path 2-5-9-4: 3
- Height of 7: 2
- Height of tree: 3
- Degree of 7: 3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



- And here are the answers

Basic idea behind a simple implementation

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

    /**
     * A node subclass which contains various recursive methods
     *
     * @param <E>  The type of the contents of nodes
     */
    private class Node {
        private E element;

        private Node left;
        private Node right;

        /**
         * Node constructor with subtrees
         *
         * @param left   the left node child
         * @param right      the right node child
         * @param E      the element contained in the node
         */
        public Node(Node left, Node right, E element) {
            this.left = left;
            this.right = right;
            this.element = item;
        }
```



Having seen all these properties, we can now think of a basic Java implementation of the abstract data type of binary trees. Here's a template of how this would look. Our `BinaryTree` class will work with generics (`E`). It has a private inner class `Node` that holds an element of type `E` and two pointers to the left and right children. These pointers will be set to `null` if there is no equivalent child. Finally, the entire binary tree has a pointer to a single `Node`, its `root`.

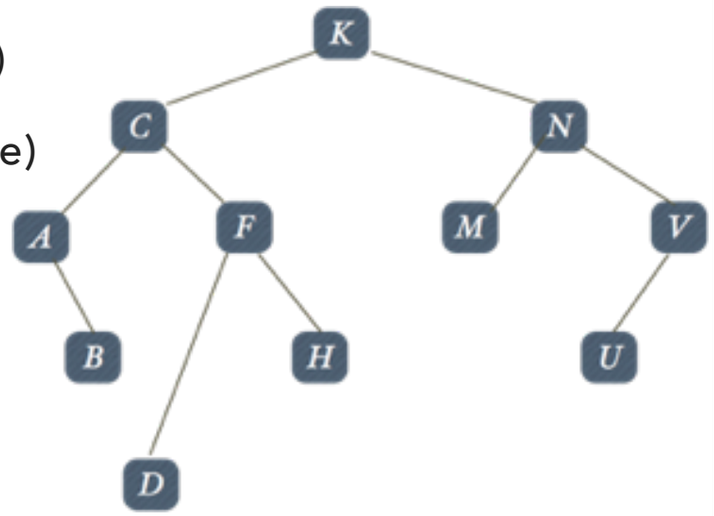
Lecture 16: Binary Trees, Binary Search, Heaps, and Priority Queues

- Binary Trees
- Tree traversals
- Binary Search
- Binary Heaps
- Priority Queues

- With all the linear data structures that we've seen in the past, we've said that it was clear what was their beginning and end and it was straightforward how you could navigate them to enumerate all of their items.
- This is not so clear in binary trees and there are many different approaches that depend on the order that we mark the root as visited and then call the algorithm recursively on the left and right subtrees. These are pre-order, in-order, post-order, and level order. The prefix on pre-,in-, and post- indicates the order in which we will mark the root of the (sub) tree as visited: for pre, before the left and right subtree, for in, between the left and right subtree, and for post, after left and right subtree. Let's see these in more detail.

Pre-order traversal

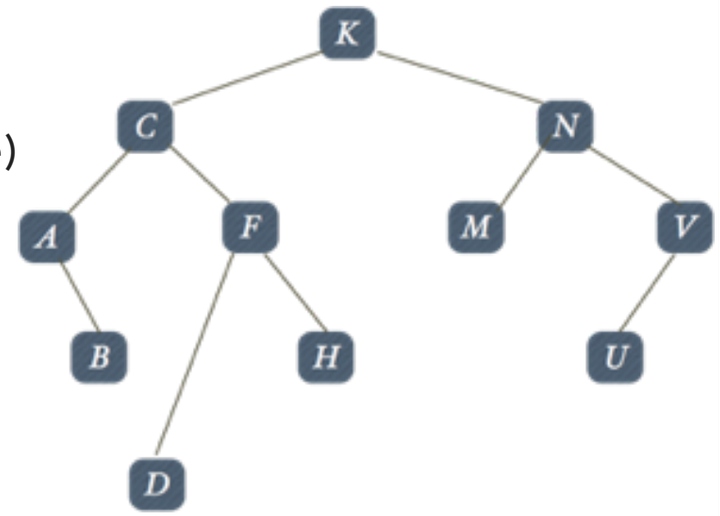
- Preorder(Tree)
  - Mark root as visited
  - Preorder(Left Subtree)
  - Preorder(Right Subtree)
- K C A B F D H N M V U



The first traversal is called pre-order and we call it recursively on the entire tree. If we look at the pseudocode, it starts by marking the root as visited, then calling recursively the method on the left subtree, when done with the left subtree, it calls itself recursively on the right subtree. Let’s walk over the example on the whiteboard.

In-order traversal

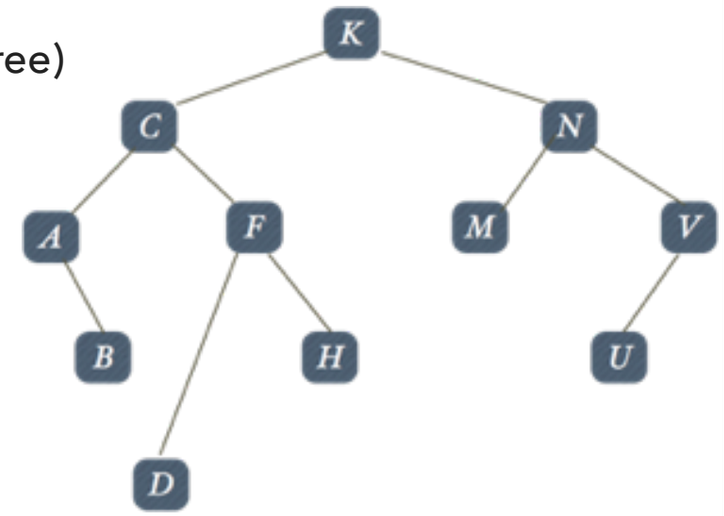
- Inorder(Tree)
  - Inorder(Left Subtree)
  - Mark root as visited
  - Inorder(Right Subtree)
- A B C D F H K M N U V



- The second traversal is called in-order and we call it recursively on the entire tree. It starts by calling recursively itself on the left subtree, when done it marks the root as visited, and the it calls itself recursively on the right subtree.
- Let’s go over this example on the whiteboard.
- You might have noticed an interesting thing: we visited our nodes in alphabetical order. The reason is that this binary tree is a special type of tree called a binary search tree that we will encounter in a couple of lectures. For every node, the left child is smaller than it and the right child is larger than it. In-order traversals in binary search trees visit the nodes in sorted order.

Post-order traversal

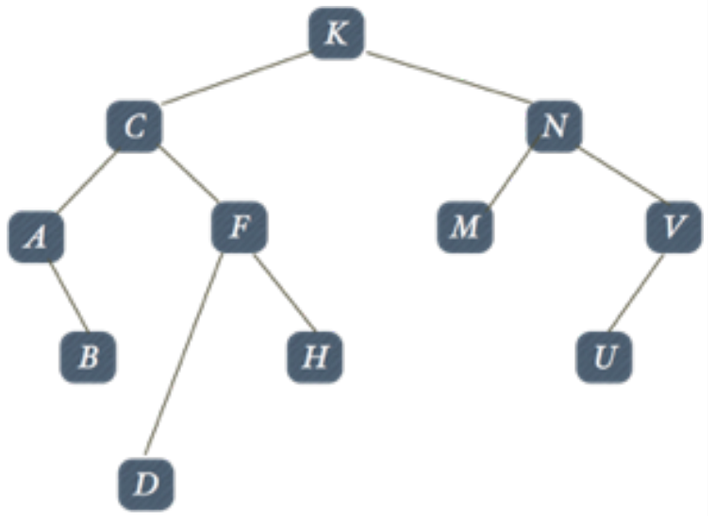
- Postorder(Tree)
  - Postorder(Left Subtree)
  - Postorder(Right Subtree)
  - Mark root as visited
- B A D H F C M U V N K



The third traversal is called post-order and we call it recursively on the entire tree. It starts by calling recursively itself on the left subtree, when done, it calls itself recursively on the right subtree, and finally it marks the root of that subtree as visited.

Level-order traversal

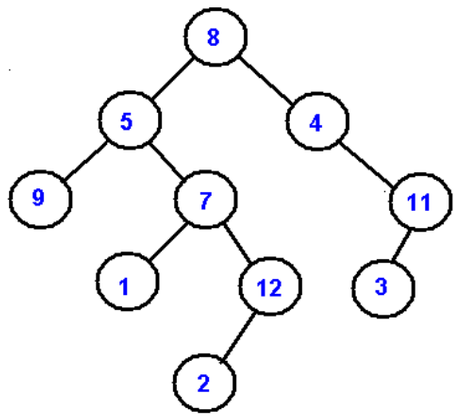
- From left to right, mark nodes of level i as visited before nodes in level $i + 1$. Start at level 0.
- K C N A F M V B D H U



The last and easiest traversal is called level-order. We mark the nodes as visited from left to right starting at level 0 and advancing one level at a time.

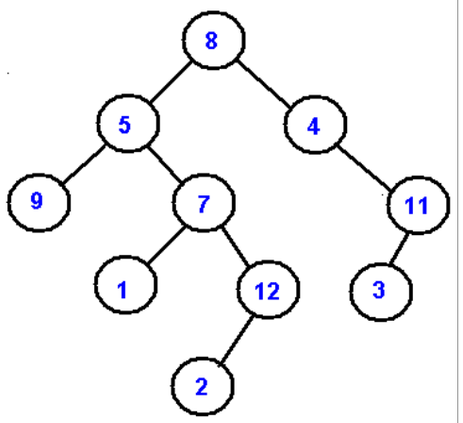
Practice Time - Problem 2 Worksheet #16

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



Now I want you to try to apply the four traversal algorithms we saw on the following tree.

ANSWER Problem 2 Worksheet #16



- 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

And this is the answer for all traversal algorithms. Let me know if you reached a different result.

Lecture 16: Binary Trees, Binary Search, Heaps, and Priority Queues

- Binary Trees
- Tree traversals
- Binary Search
- Binary Heaps
- Priority Queues

Let's take a detour for a moment to talk about binary search a concept you might have encountered before and which will be handy for the next assignment.

Binary search

- Goal: Given a sorted array and a key, find index of the key in the array.
- Basic mechanism: Compare key 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

- First binary search published in 1946.
- First bug-free one 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

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

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

Lecture 16: Binary Trees, Binary Search, Heaps, and Priority Queues

- Binary Trees
- Tree traversals
- Binary Search
- Binary Heaps
- Priority Queues

OK let's now see binary heaps, a special group of binary trees that are heap-ordered. Let's see what that means.

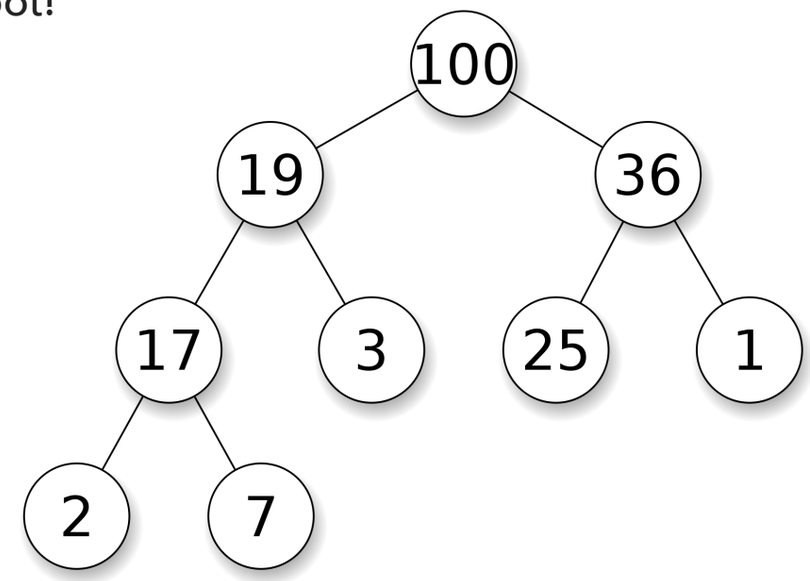
Heap-ordered binary trees

- 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).
- 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.

- A binary tree is heap-ordered if each node is larger than or equal to its two children (if any). Equivalently, each node is smaller than its parent. Notice that we don't have any assumption about which child is smaller, just that they are smaller than the parent.
- If we were to pick a node and start moving up the tree, given the heap-ordered property, we would get a non-decreasing sequence of keys.
- Equivalently, if we were to pick a node and start moving down the tree, given the heap-ordered property, we would get a non-increasing sequence of keys.

Heap-ordered binary trees

- The largest key in a heap-ordered binary tree is found at the root!



Based on these properties, it’s not hard to imagine that the largest key in a heap-ordered binary tree can be found at the root.

Binary heap representation

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

- Given the heap-ordered property, we can start wondering how could we code such a tree in Java. We could use the basic linked representation but we would need three links for every node (one for the parent, and one for its of the children) in order to go up and down the tree.
- Instead, we can use the following idea. If we use complete binary trees, that we know are as compact as possible in terms of height and have all their nodes on the last level as left as possible, then we can represent our heap-ordered binary tree as an array.
- We know that arrays are more memory-efficient than linked representations which means that we save memory.

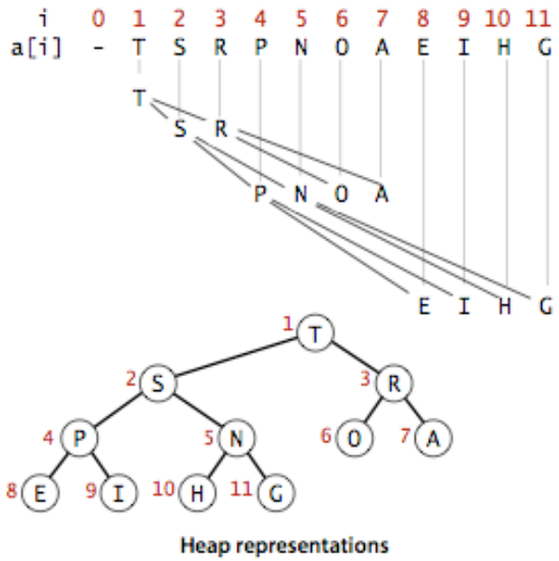
Binary heaps

- 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, too.

- If we put all these pieces together: binary trees that have the heap-ordered property, that are complete, and are represented in memory as arrays, we get what is called a binary heap.
- The 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 that will correspond to its children.
- Specifically, this definition corresponds to a max-heap since the maximum element is at the root. If we were to reverse the definition of heap order and have a node be smaller than its two children (if any), the root would become the smallest element and the binary tree would be known as a min-heap.

Array representation of heaps

- Nothing is placed at index 0.
- Root is placed at index 1.
- Rest of nodes are placed in level order.
- No unnecessary indices and no wasted space because it's complete.



Heap representations

- Let's look into the array representation of a binary heap. We will ignore index 0 for arithmetic convenience. Our root will placed at index 1. The rest of nodes will be placed based in the order that are returned from a level-order traversal. Because the tree is complete, there won't be any wasted space which makes binary heaps very memory efficient.
- Take a look at the indices of the above example. Can you guess what is the relationship between the index of a node and its two children? Given the index of a node where would its parent be located?

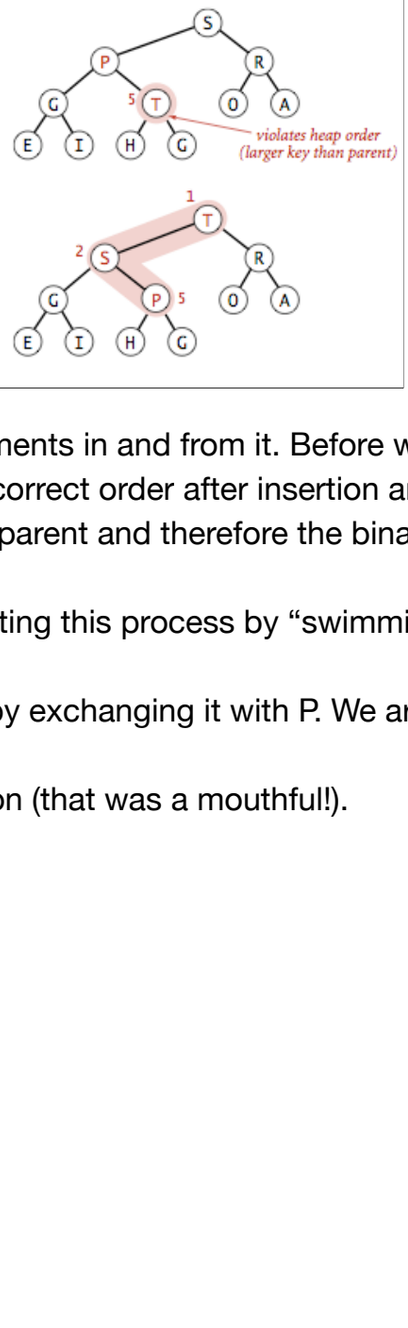
Reuniting immediate family members.

- 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.

If you are given a node at index k, it is not hard to figure out where its parent and children are located. The parent will be located at floor(k/2), the left child at 2k, and the right child 2k+1. We can use this simple arithmetic to go up and down the heap.

Swim/promote/percolate up/bottom up reheapify

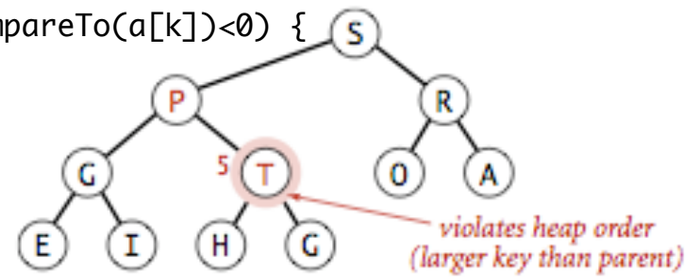
- Scenario: a key becomes larger than its parent therefore it violates the heap-ordered property.
- To eliminate the violation:
  - Exchange key in child with key in parent.
  - Repeat until heap order restored.

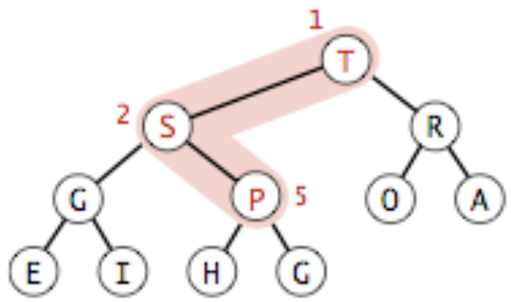


- For every data structure we have seen, we always ask how do we add and how do we remove elements in and from it. Before we answer this question for binary heaps, I want us to learn two fundamental moves that will make sure that our heap will stay in the correct order after insertion and deletion.
- Let's assume that we have the following scenario. For some reason, a key becomes larger than its parent and therefore the binary heap violates the heap-ordered property.
- What we will do to eliminate this violation is exchange the child with the parent. We will keep repeating this process by "swimming up" the key until the heap order is restored.
- For example, T violates the heap order as it is larger than P. To fix this violation, we will swim it up by exchanging it with P. We are still not done. We need to swim it up one more position till it finds its final place at the root.
- This move is known under different names: swim/promote/percolate up or bottom up reheapification (that was a mouthful!).

Swim/promote/percolate up

```
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;
   }
}
```



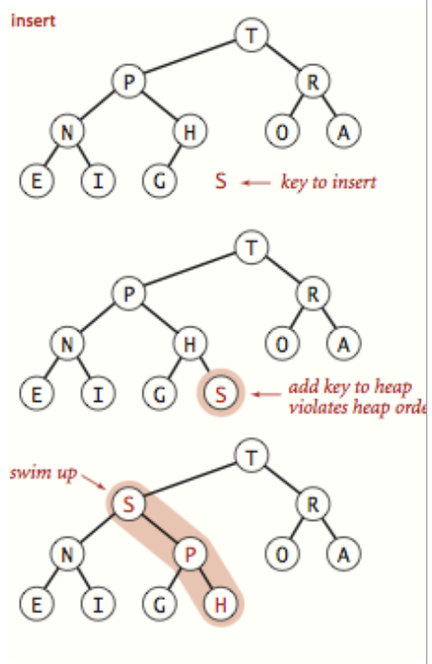


The Java code for swimming up a node that is located at the index k is quite straightforward and is based on the arithmetic that we saw to find the parent. Basically, we will keep exchanging the node that exists at position k with its parent that exists at position k/2 until we either hit the root (k==1) or we find that the node is smaller than its parent.

Binary heap: insertion

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

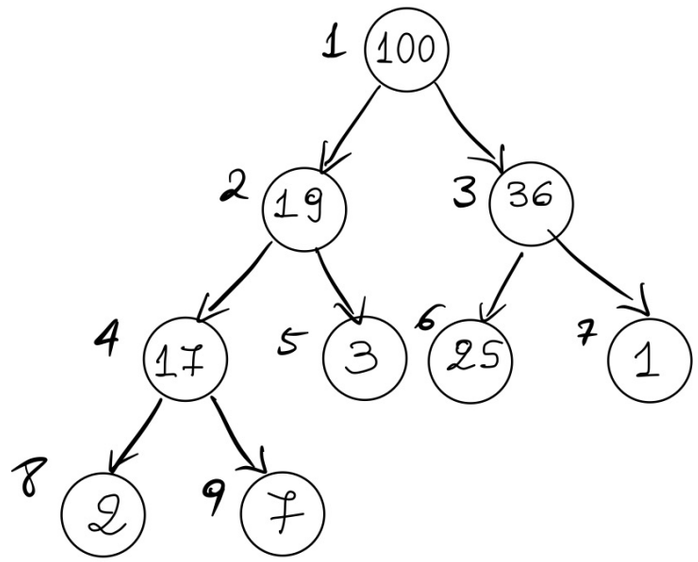
```
public void insert(E x) {
   a[++n] = x;
   swim(n);
}
```



- Now that we mastered how to swim a node up, we are ready to actually start inserting elements into a heap. The recipe is very simple and it relies on the two properties that we need to preserve: 1) our binary tree has to remain complete and 2) it has to be heap-ordered.
- To remain complete, we need to add the new node at the last level, as left as possible.
- To remain heap-ordered, we need to swim it up until it finds its right place.
- For example, if we are given the key S to enter in this heap, it has to become H's child. But now it violates the heap order so we have to swim it up by exchanging it first with H, and then with P.
- The code for insertion is super easy. If pq is the array that we're holding the elements and n is the number of elements that we've added so far, we just increase the number of elements n first, we go to that index and assign the provided key x and then swim it up! TADA!
- The number of comparisons we will perform is at most logn+1, where logn is the length of the longest path (height of tree!) since our tree is complete. Therefore, the cost of insertion in binary heaps is O(logn).

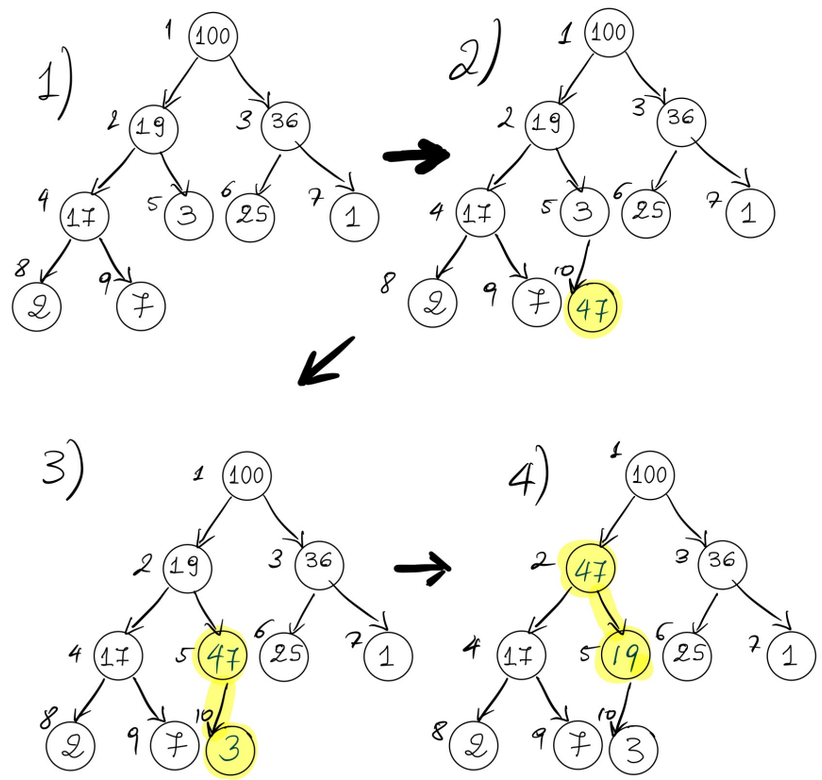
Practice Time - Problem 3 Worksheet #16

- Insert 47 in this binary heap.



Now I want us to practice with inserting the key 47 (!) in this binary heap. Remember, your goal will be to keep this tree complete and heap-ordered.

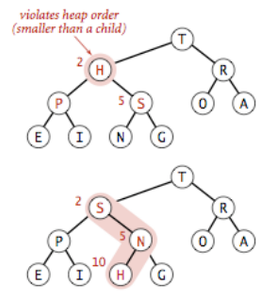
ANSWER 3



- To remain complete, 47 has to become the left child of 3, that is to be inserted at 9+1=10 index. But now the heap-order is violated so we need to swim it up by first exchanging it with 47 and then with 21.
- If we were to see the swimming of 47 in more detail according to the Java code, we would find its parent which is located at 10/2=5 index. Since 47 is larger than 3, we would exchange it. Now k=5 and its parent is located at 5/2=2. At index 2, we have 21, which is smaller than 47 so we exchange them. Now k=2 and its parent is 2/2=1 which stores the key 100. 100<47 so we don’t need to exchange it and 47 found its final place at index 2.

Sink/demote/top down heapify

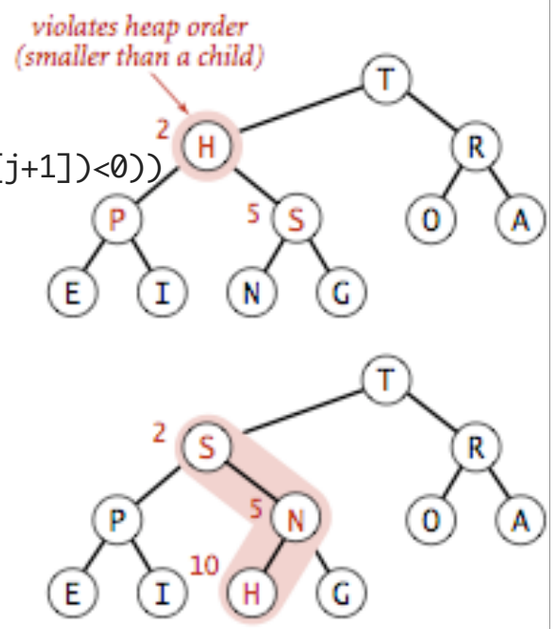
- Scenario: a key becomes smaller than one (or both) of its children's keys.
- To eliminate the violation:
  - Exchange key in parent with key in **larger** child.
  - Repeat until heap order is restored.



- Now that we've seen how to insert a key to a binary heap we can start thinking of how to perform the remove operation. For this, we will need another building block, a method that will allow us to sink/demote a node that is smaller than one or both of its children.
- In this scenario, we will exchange the parent with the larger child. This is critical so that we keep the heap order! We will keep repeating this step until the heap order is restored.
- For example here, H violates the heap-order as it is smaller both than P and S. We will exchange it with the largest child, that is S. It still violates the heap order so we will exchange it with N since N is larger than G.

Sink/demote/top down heapify

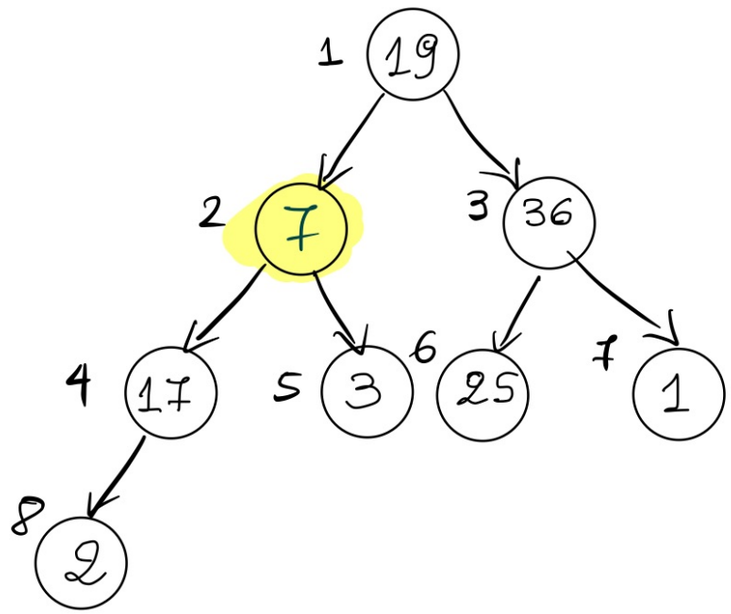
```
private void sink(int k) {
    while (2*k <= n) {
        int j = 2*k;
        if (j < n && a[j].compareTo(a[j+1])<0))
            j++;
        if (a[k].compareTo(a[j])>=0))
            break;
        E temp = a[k];
        a[k] = a[j];
        a[j] = temp;
        k = j;
    }
}
```



- The code for sinking/demotion/top-down heapify (seriously what a name?!) of a node at index k works as follows:
- We will find the left child of the node that is located at j=2*k. If j is smaller than the total number of nodes and smaller than the right child of k, we will examine the right child.
- If the key at index k is smaller than the larger of the two children, we will exchange it with the larger child. We repeat this step, by moving down the heap until we reach a node with both children being smaller, or we reach the bottom of the tree.

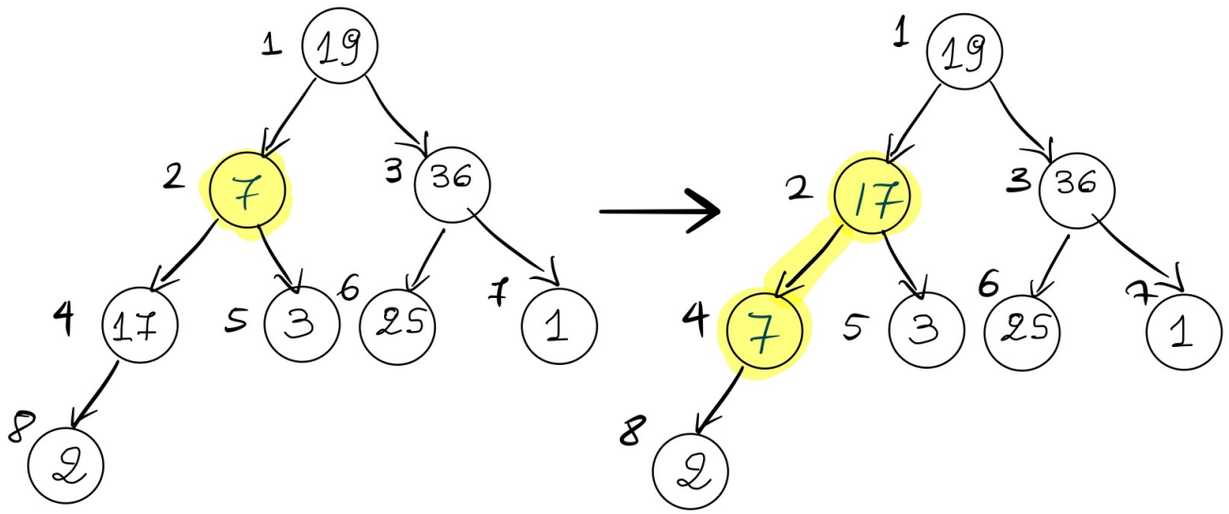
Practice Time - Problem 4 Worksheet #16

- Sink 7 to its appropriate place in this binary heap.



Let’s practice what we just learned by sinking 7 which is located at index 2 to its appropriate place.

ANSWER 4



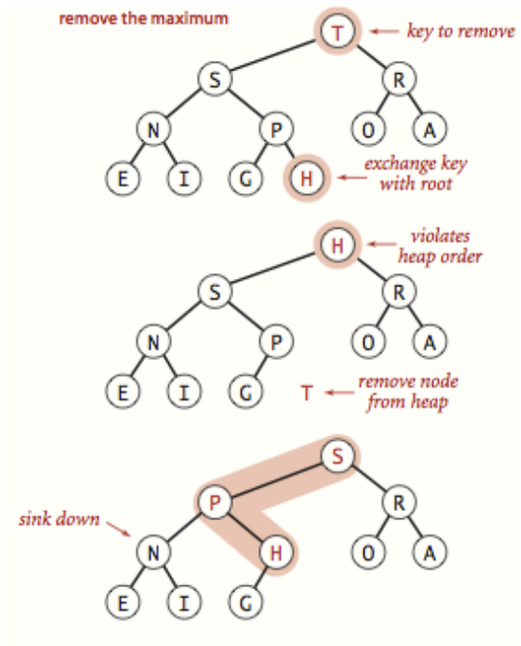
It’s quite easy to see that 7 is smaller than 17 and that we need to exchange it. If we were to follow the code more closely, that would look like: To sink 7 which is located at index 2, we need to calculate where its children would be. The left is at 2*2 = 4 index and the right is at 2*2+1=5 index. Since 3<7, we will ignore the right child and compare the key at index 2 (7) with the key at index 4 (17). 7<17, therefore we need to exchange them. k now becomes 4 and since 7 is larger than its only child (2), we stop the process of sinking it.

Binary heap: return (and delete) the maximum

- Delete max: Exchange root with node at end. Return it and delete it. Sink the new root down.
- Cost: At most $2 \log n$ compares.

- Rather than seeing how to remove any element, with binary heaps we will focus on how to remove and return the maximum element that we know is the root. The idea is simple and again it is based on the fact that our tree needs to remain complete and heap-ordered.
- We will first save the root in a temporary variable so that we can eventually return it. We will exchange the root with the last node, followed by reducing the number of elements n by 1. Then we will sink the root until it finds its appropriate place. Now we can set the n+1 item to null to ensure that no one considers it part of the binary heap. And finally, we can return the old stored maximum.
- The delete max operation requires two compares for each node on the path (except at the bottom): one to find the child with the larger key, the other to decide whether that child needs to be promoted, therefore the cost of deletion will be at most 2logn or O(logn).

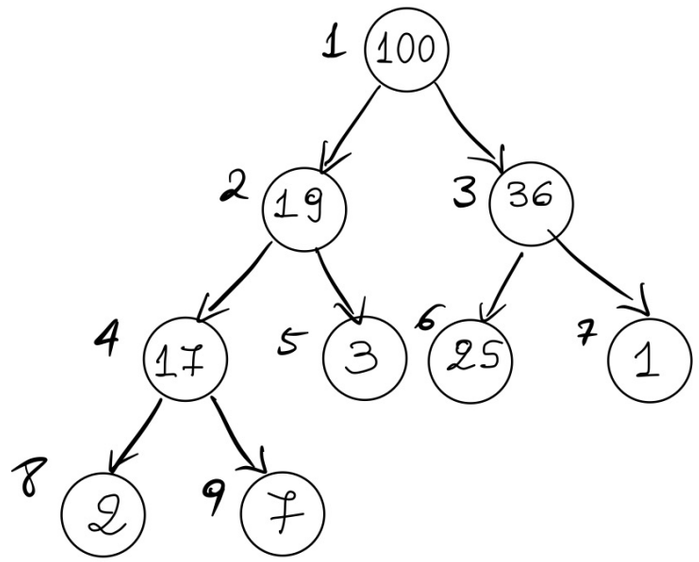
Binary heap: delete and return maximum



Here's an example of how deleting the maximum element would look like. The maximum element is located at the root. We exchange it with the last element (exchange T and H). We sink H to its appropriate position and then nullify the last node and return T.

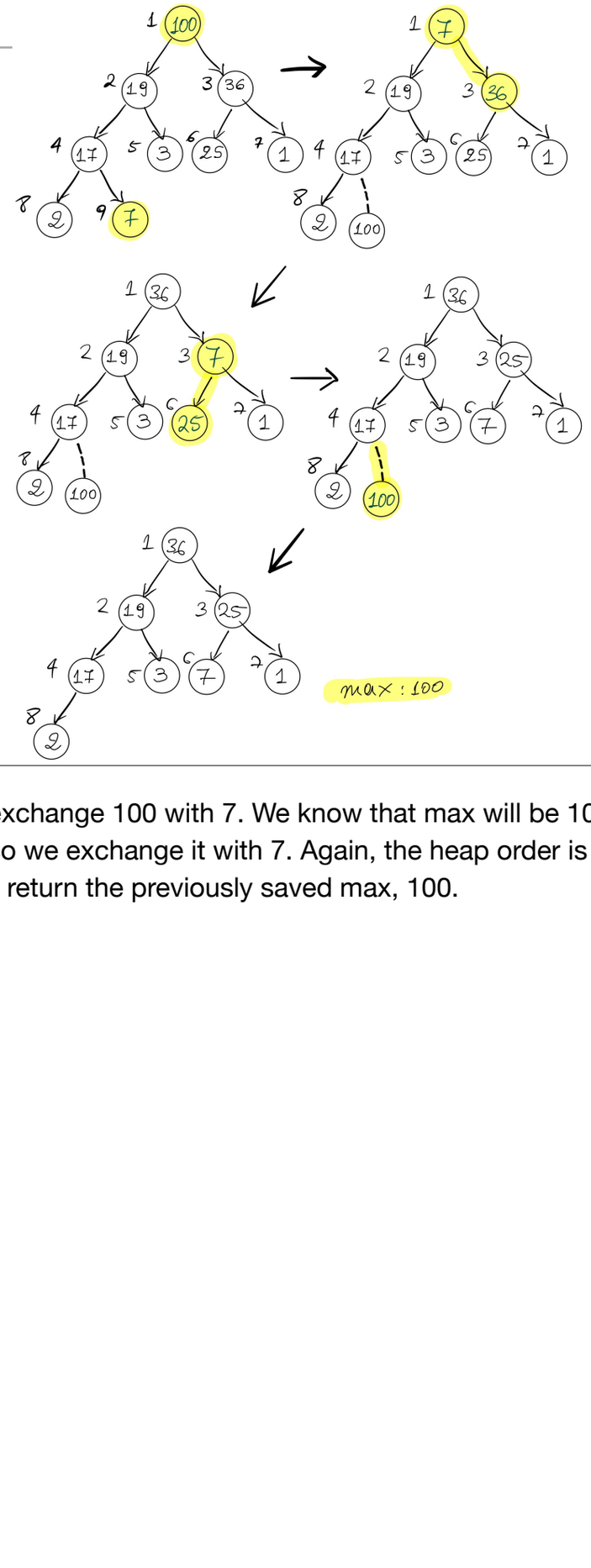
Practice Time - Problem 5 Worksheet #16

- Delete max (and return it!)



Now I want us to practice with deleting the max (and returning it) of this binary heap.

ANSWER 5



We start by exchanging the root with the maximum index, 9, that is we exchange 100 with 7. We know that max will be 100 and that our elements are now going to be 8. We ask whether we need to sink 7 and we do so. The larger child is 36 so we exchange it with 7. Again, the heap order is violated and we exchange 7 with 25. We have now restored the heap-order and can set the node at index 9 to null and return the previously saved max, 100.

Things to remember about running time complexity of heaps

- Insertion is $O(\log n)$.
- Delete max is $O(\log n)$.
- Space efficiency is $O(n)$.

So what we've seen so far is that if we have a heap with n elements we can insert a new element or delete the max in O(logn). Space efficiency is also great as we only need O(n) of it without wasting space for links and without any gaps in the array as all levels are filled due to the heap being a complete tree.

Algorithms ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

2.4 BINARY HEAP DEMO

This video demonstrates all the steps that we saw together.

Lecture 16: Binary Trees, Binary Search, Heaps, and Priority Queues

- Binary Trees
- Tree traversals
- Binary Search
- Binary Heaps
- Priority Queues

Many applications require that we process items having their keys in order, but not necessarily in full sorted order and not necessarily all at once. Often, we collect a set of items, then process the one with the largest key, then perhaps collect more items, then process the one with the current largest key, and so forth.

Priority Queue ADT

- Two operations:
  - Delete the maximum
  - Insert
- Applications: load balancing and interruption handling in OS, Huffman codes for compression, A* search for AI, Dijkstra's and Prim's algorithm for graph search, etc.
- How can we implement a priority queue efficiently?

- For example, you can think of how an ER would try to treat patients not simply on a first-come/first-serve basis, but by taking into account what case is the most critical and move that at the top of the queue.
- Today, we will see a data type that will allow us to model such environments. That data type is a priority queue. There are two main operations that a priority queue should support.
- We should be able to delete (and return) the maximum element (or the element with the highest priority), and we should be able to insert a new element in it.
- Priority queues have wide applications in many classic Computer Science problems and especially in Systems and AI.
- Given all the data structures we have seen so far, let's start thinking of how we can implement a priority queue efficiently.

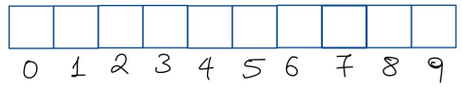
Option 1: Unordered array

- The *lazy* approach where we defer doing work (deleting the maximum) until necessary.
- Insert is $O(1)$ and assumes we have the space in the array.
- Delete maximum is $O(n)$ (have to traverse the entire array to find the maximum element and exchange it with the last element).

- The simplest priority-queue implementation is based on an unsorted array. This approach can be considered lazy as we defer doing the hard work (which is deleting the maximum element) until it is necessary.
- Insertion be achieved in O(1) running time as we will just insert the key at the end of the array.
- Since our elements are stored based on the order they arrived, if we are asked to delete the maximum, we have to traverse the entire array and examine each element to find which one is the true maximum. If the array holds n elements, this linear search to find the global maximum will be O(n).

```
public class UnorderedArrayMaxPQ<Key extends Comparable<Key>> {
    private Key[] pq;      // elements
    private int n;         // number of elements

    // set inititial size of heap to hold size elements
    public UnorderedArrayMaxPQ(int capacity) {
        pq = (Key[]) new Comparable[capacity];
        n = 0;
    }

    public boolean isEmpty()   { return n == 0; }
    public int size()          { return n;      }
    public void insert(Key x)  { pq[n++] = x;   }

    public Key delMax() {
        int max = 0;
        for (int i = 1; i < n; i++){
            if (pq[max].compareTo(pq[i]) < 0) {
                max = i;
            }
        }
        Key temp = pq[max];
        pq[max] = pq[n-1];
        pq[n-1] = temp;

        return pq[--n];
    }
}
```

- If we were to implement this lazy approach in Java, we would have an array of generic comparable `Key`'s and a counter of elements `n`. The insertion is very simple: we just add the given element at the end of the array and increase the counter `n` by 1. In total, this is accomplished in O(1) running time.
- The deletion of the maximum has a for-loop that has to go through all `n` elements to find the `max` element. Once it finds it, it exchanges it with the last element, decreases the counter `n` by 1, and returns it. This is accomplished in O(n) time.



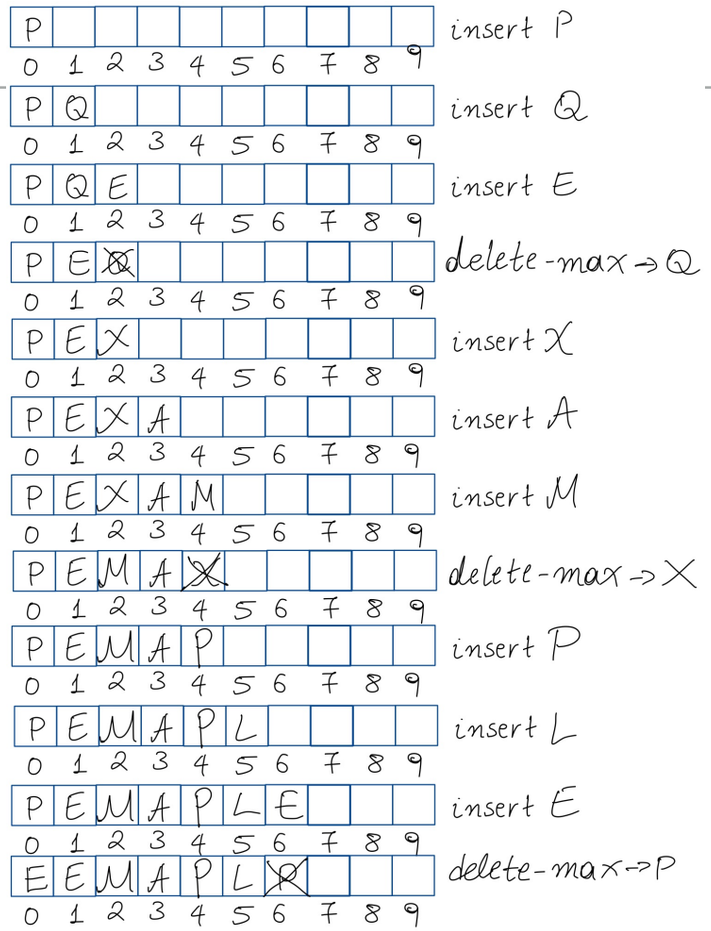
Practice Time

- Given an empty array of capacity 10, perform the following operations in a priority queue based on an unordered array (lazy approach):

1. Insert P
2. Insert Q
3. Insert E
4. Delete max
5. Insert X
6. Insert A
7. Insert M
8. Delete max
9. Insert P
10. Insert L
11. Insert E
12. Delete max

Let's practice implementing priority queues with this lazy approach. Assume you are given an array of capacity 10 and you have to perform the 12 operations above. Remember, you insert at the end of the array and when you are asked to delete the max, you iterate through the entire array to find the maximum element and then exchange it with the last element.

Answer



You should end up with these intermediate steps. Notice that for every delete-max call, we exchange the maximum element with the last element and then delete it.

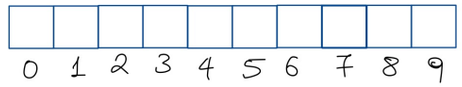
Option 2: Ordered array

- The *eager* approach where we do the work (keeping the array sorted) up front to make later operations efficient.
- Insert is $O(n)$ (we have to find the index to insert and shift elements to perform insertion).
- Delete maximum is $O(1)$ (just take the last element which will be the maximum).

- A second option is the eager approach where we keep the array sorted. In contrast to the lazy approach where we defer the expensive work (deletion of max) until it is needed, here, we try to do the expensive work up front to keep later operations efficient. The expensive work here will be to keep the array sorted.
- Every time a new element arrives, we will have to find the index to insert it and shift the rest of the elements to the right to keep the array sorted. This is done in O(n) time.
- Now, when asked to delete the maximum, we know that the last element is the one we're looking for and we can return it in O(1) time.
- Notice that compared to the lazy approach, the costs for insertion and deletion of the maximum element have been reversed.

```
public class OrderedArrayMaxPQ<Key extends Comparable<Key>> {
    private Key[] pq;          // elements
    private int n;             // number of elements

    // set inititial size of heap to hold size elements
    public OrderedArrayMaxPQ(int capacity) {
        pq = (Key[]) (new Comparable[capacity]);
        n = 0;
    }


    public boolean isEmpty() { return n == 0;  }
    public int size()        { return n;       }
    public Key delMax()      { return pq[--n]; }

    public void insert(Key key) {
        int i = n-1;
        while (i >= 0 && key.compareTo(pq[i]) < 0) {
            pq[i+1] = pq[i];
            i--;
        }
        pq[i+1] = key;
        n++;
    }
}
```

- Similarly, the implementation of the eager approach reflects that deleting the maximum consists of only returning the last element and increasing the counter n by 1. This is accomplished in O(1) time.
- The insertion starts us at the last element, keeps shifting elements to the right until it finds where to place the given key so that the array remains sorted. Finally it increases the counter n by 1. This is accomplished in O(n) time.



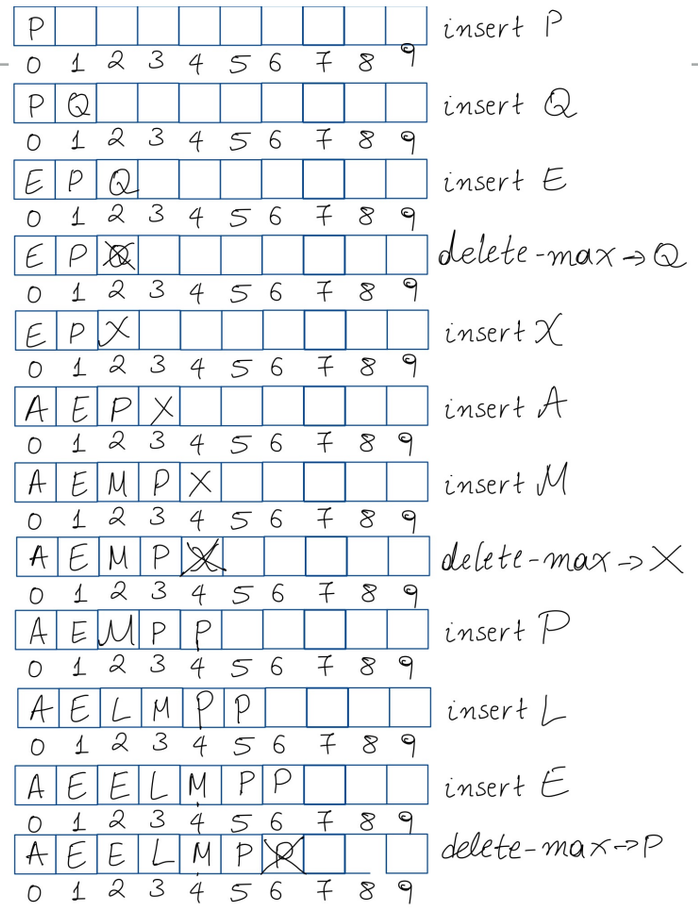
Practice Time

- Given an empty array of capacity 10, perform the following operations in a priority queue based on an ordered array (eager approach):

1. Insert P
2. Insert Q
3. Insert E
4. Delete max
5. Insert X
6. Insert A
7. Insert M
8. Delete max
9. Insert P
10. Insert L
11. Insert E
12. Delete max

Let's practice implementing priority queues with this second, eager approach. Assume you are given an array of capacity 10 and you have to perform the same 12 operations above. Remember, you insert by starting at the end, shifting elements to the right and stop when you find the appropriate position to enter the given key. When you are asked to delete the max, you know that it is at the last position so that you can just return it.

Answer



You should end up with these intermediate steps.

Option 3: Binary heap

- Will allow us to both insert and delete max in $O(\log n)$ running time.
- There is no way to implement a priority queue in such a way that insert *and* delete max can be achieved in $O(1)$ running time.
- Priority queues are synonyms to binary heaps.

- Perhaps you've already thought about it, but the third and much better option for implementing a priority queue is to use a binary heap, the array representation of complete heap-ordered binary trees that we encountered during the last class.
- Using a binary heap we can achieve both insertion and deletion of the maximum element in O(logn) time! We just have to follow the usual way of inserting and deleting the max element in binary heaps.
- There is no better way to implement a binary heap so that both insertion and deletion of max can be achieved in O(1) time, that is why often priority queues are used as synonyms to binary heaps.

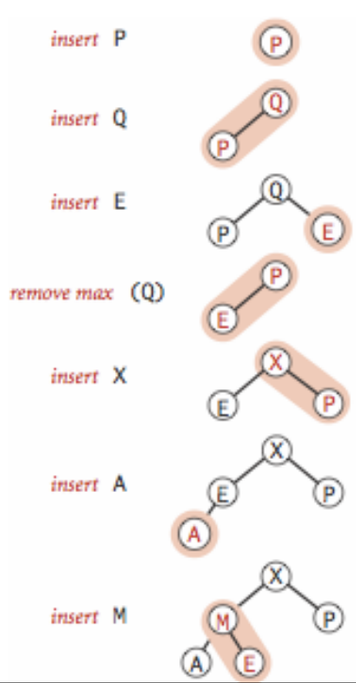
Practice Time

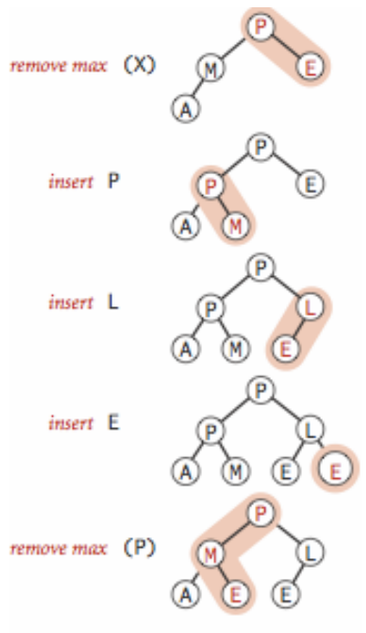
- Given an empty binary heap that represents a priority queue, perform the following operations:

1. Insert P
2. Insert Q
3. Insert E
4. Delete max
5. Insert X
6. Insert A
7. Insert M
8. Delete max
9. Insert P
10. Insert L
11. Insert E
12. Delete max

Finally, let's practice with performing the same 12 operations in a priority queue implemented as a binary heap. Remember that for every step should always end up with a complete heap-ordered array.

Answer





You should end up with these intermediate steps.

Lecture 16: Binary Trees, Binary Search, Heaps, and Priority Queues

- Binary Trees
- Tree traversals
- Binary Search
- Binary Heaps
- Priority Queues

And this concludes our lecture on binary trees, binary search, and heaps/priority queues. We introduced trees to model hierarchical rather than linear relationships, we focused on binary trees, trees that have at most two children. Since there is no obvious beginning and end we learned how to traverse binary trees based on pre-,in-,post-,level-order. And followed with binary heaps which are the array representation of complete heap-ordered binary trees and learned how to insert a new element and delete the maximum element from them. We concluded that priority queues make most sense to be implemented as heaps.

Readings:

- Recommended Textbook:
  - Chapter 2.4 (Pages 308-327)
- Website:
  - Heaps: https://algs4.cs.princeton.edu/24pq/
- Visualization:
  - Insert and ExtractMax: https://visualgo.net/en/heap

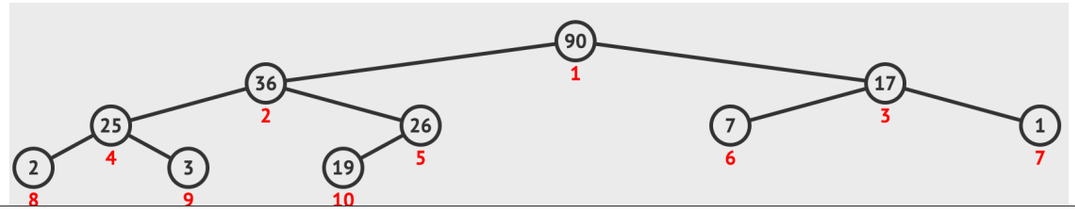
Worksheet

- Lecture 16 worksheet

I highly recommend you supplement the lecture with the following textbook readings, a visualization tool that you can use to practice insertions and deletions of the maximum element, and advice on how to practice.

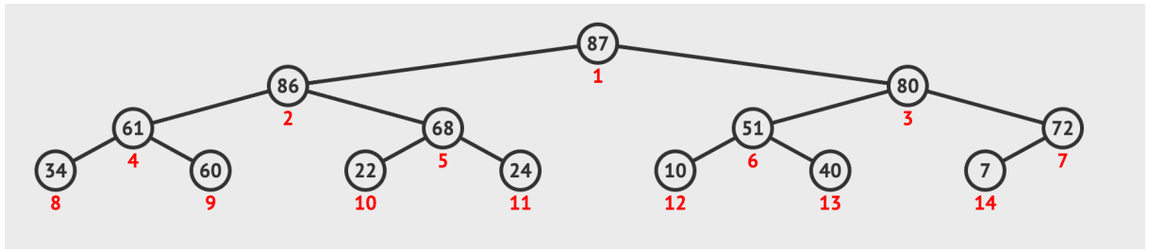
Practice Problem 1

- 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.

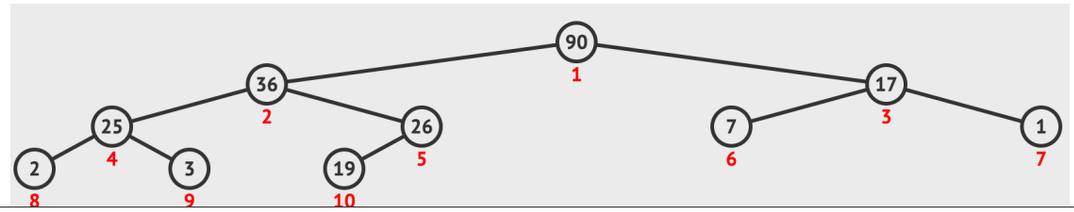


Practice Problem 3

- Suppose that the sequence 16, 18, 9, 15, *, 18, *, *, 9, *, 20, *, 25, *, *, *, 17, 21, 5, *, *, *, 21, *, 5 (where a number means insert and an asterisk means delete the maximum) is applied to an initially empty priority queue. Give the sequence of numbers returned by the delete maximum operations.

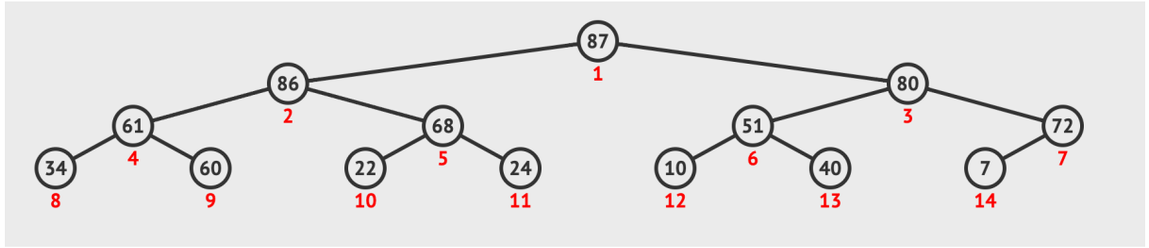
ANSWER 1

- 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

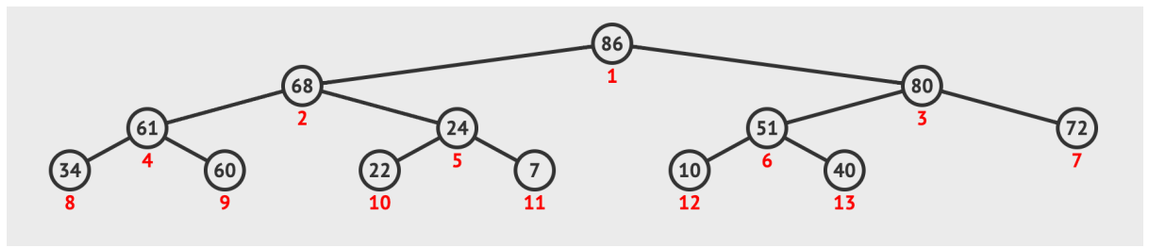


ANSWER 2

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



-



ANSWER 3

- Suppose that the sequence 16, 18, 9, 15, *, 18, *, *, 9, *, 20, *, 25, *, *, *, 17, 21, 5, *, *, *, 21, *, 5 (where a number means insert and an asterisk means delete the maximum) is applied to an initially empty priority queue. Give the sequence of numbers returned by the delete maximum operations.
- 18, 18, 16, 15, 20, 25, 9, 9, 21, 17, 5, 21