CSE 12 Exam 2

Run Time Analysis

- **Big O**
  - Upper bound of a function
  - $g(n)$ is $O(f(n))$ if $f(n) \leq c \cdot g(n)$ for $n \geq n_0$
  - Example:

$$g(n) \text{ is } O(f(n)) \text{ given } g(n) = n^2 \text{ and } f(n) = n$$
$$n_0 = 0 \text{ and } c = 1$$

- **Big Omega**
  - Lower bound of a function
  - $g(n)$ is $\Omega(f(n))$ if $c \cdot g(n) \leq f(n)$ for $n \geq n_0$
  - Example:

$$g(n) \text{ is } \Omega(f(n)) \text{ given } g(n) = \log(n) \text{ and } f(n) = n$$
$$n_0 = 0 \text{ and } c = 1$$

- **Big Theta**
  - "Tight" bound of a function
  - A function that, dependent on its coefficient, could be both lower and upper bound
  - $g(n)$ is $\Theta(f(n))$ if $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for $n \geq n_0$
  - Example:

$$g(n) \text{ is } \Theta(f(n)) \text{ given } g(n) = n \text{ and } f(n) = log(n)$$
$$n_0 = 1 \text{ and } c_1 = 0$$
$$n_0 = 0 \text{ and } c_2 = 5$$

- **Common Runtime Analysis Questions**
  - **Array List**
    - Insertion and deletion at beginning - $[O(n), \Omega(n), \Theta(n)]$
    - Insertion and deletion at middle - $[O(n), \Omega(n), \Theta(n)]$
    - Insertion and deletion at end - $[O(1), \Omega(1), \Theta(1)]$
    - Find at beginning - $[O(1), \Omega(1), \Theta(1)]$
    - Find at middle - $[O(n), \Omega(n), \Theta(n)]$

    - Find at end - $[O(n), \Omega(n), \Theta(n)]$
- **Linked List (Single)**
    - Insertion and deletion at beginning - $[O(1), \Omega(1), \Theta(1)]$
    - Insertion and deletion at middle - $[O(n), \Omega(n), \Theta(n)]$
    - Insertion and deletion at end - $[O(n), \Omega(n), \Theta(n)]$
    - Find at beginning - $[O(1), \Omega(1), \Theta(1)]$
    - Find at middle - $[O(n), \Omega(n), \Theta(n)]$
    - Find at end - $[O(n), \Omega(n), \Theta(n)]$
    - Find (General) - $[O(n), \Omega(1)]$
    - Insertion and deletion (General) - $[O(n), \Omega(1)]$
- **Linked List (Double)**
    - Insertion and deletion at beginning - $[O(1), \Omega(1), \Theta(1)]$
    - Insertion and deletion at middle - $[O(n), \Omega(n), \Theta(n)]$
    - Insertion and deletion at end - $[O(1), \Omega(1), \Theta(1)]$
    - Find at beginning - $[O(1), \Omega(1), \Theta(1)]$
    - Find at middle - $[O(n), \Omega(n), \Theta(n)]$
    - Find at end - $[O(n), \Omega(n), \Theta(n)]$
    - Find (General) - $[O(n), \Omega(1)]$
    - Insertion and deletion (General) - $[O(n), \Omega(1)]$

Hashing

- **Hashing Introduction**
    - Hash function allows us to pass in an object and obtain a numerical value representing that object.
    - We can use this hash value as an index in an array to access objects by a simple numerical value (ideation for HashMap). We encounter an issue because of a theoretically infinite number of hash values.
    - Rather than using the hash value for the index, we can use the hash value modulus the fixed length of our array. We encounter another issue when two object's hash values, modulus the fixed length of the array, equals the same value (both map to the same index).
    - Two objects mapping to the same index is called a collision
    - Hash-based structures are generically understood to be $O(1)$ based on the understanding that the hash function runs in constant time.
- **Collision Resolution**
    - **Load Factor**
        - Load Factor (typically set at 0.75):

$$\frac{\text{Maximum allowed number of elements}}{\text{Capacity of Hash Table}}$$

- Current Load:

$$\frac{\text{Current number of elements}}{\text{Capacity of Hash Table}}$$

- If current load exceeds the load factor, then we must resize the array to accommodate more elements and we must rehash every object in the initial array according to the new array size.

- **Linear Probing**
  - If a collision occurs while trying to insert an element, we simply insert the element at the next available index
  - Example where m is the size of the backing array:

$$\text{Hash function is } h(x) = x\%m$$

$$\text{Initial array (arr)} = [\_, \_, \_, \_, \_]$$

We'll insert the value 1 into the Hash Table

$$\text{arr} = [1, \_, \_, \_, \_]$$

No collisions occur. We'll insert the value 6 into the Hash Table ($6\%5 = 1\%5 = 1$). Because we can't insert at index 1, we move on to the next available index (2).

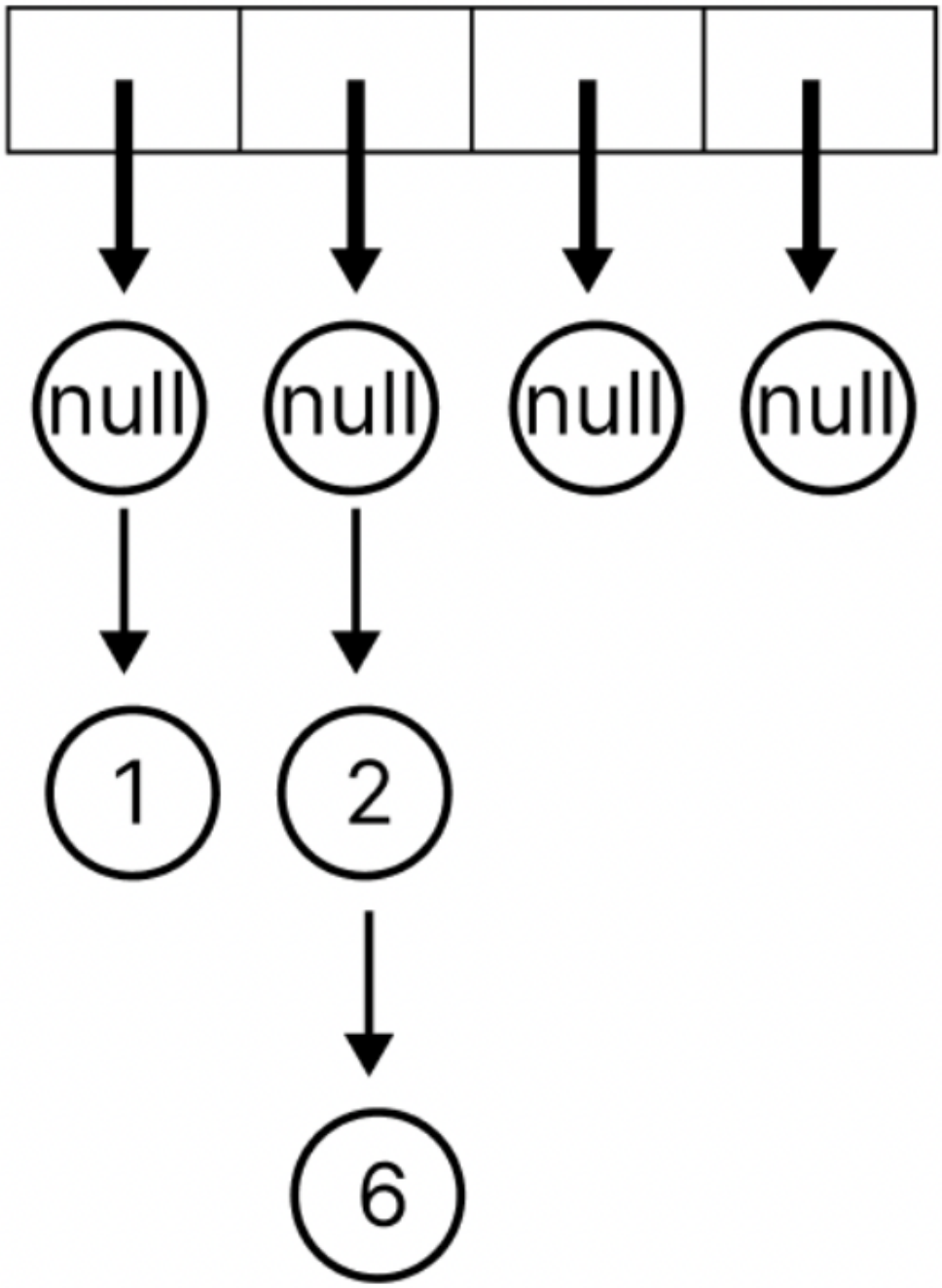
$$\text{arr} = [\_, 1, 6, \_, \_]$$

We'll try inserting the value 11 into this array ($11\%5 = 1$)

$$\text{arr} = [\_, 1, 6, 11, \_]$$

  - Linear probing uses circular arrays (the next available index of the last index of the array ($\text{arr.length} - 1$) is index 0).

- **Separate Chaining**
  - Separate chaining uses an array of Lists to deal with collisions, simply add the element to the list at the index.
  - Separate Chaining Visualization:



- **Hash Map Function Terminology**
  - Insertion - $\text{put}(\text{key}, \text{value})$
  - Search - $\text{get}(\text{key})$
  - Remove - $\text{remove}(\text{key})$

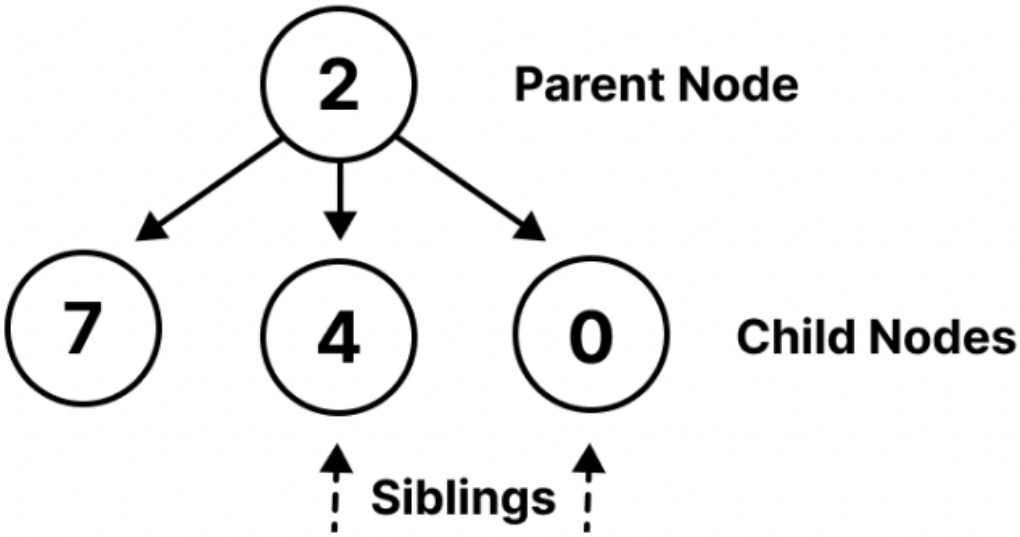
Stacks and Queues

- **Dequeues**

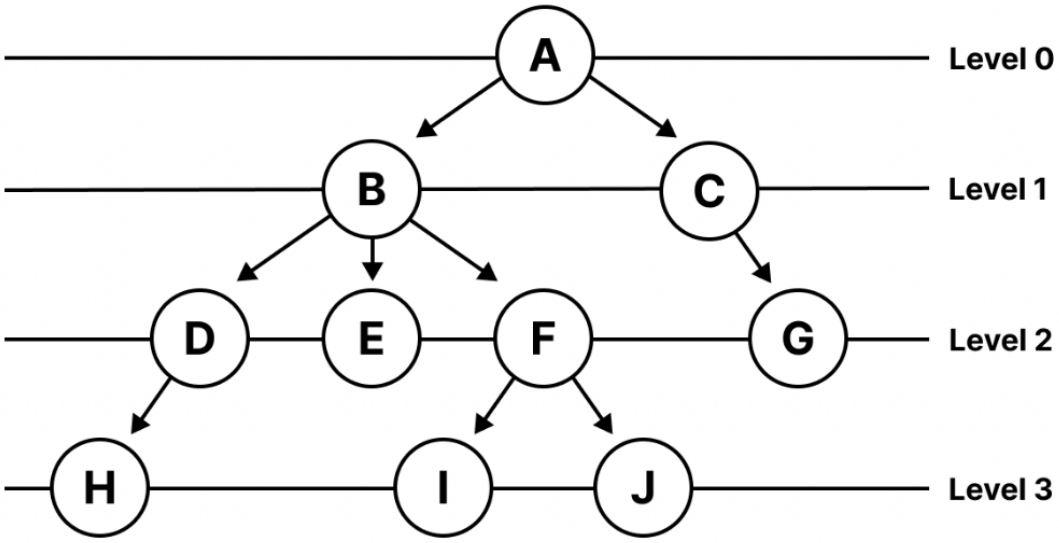
    - Double ended queue (insertion, deletion, peek at both ends of queue)
    - Implemented using Doubly Linked List
    - Insertion at front - $\text{addFirst}(\text{element})$
    - Insertion at back - $\text{addLast}(\text{element})$
    - Peek at front - $\text{peekFirst}()$
    - Peek at back - $\text{peekLast}()$
    - Remove at front - $\text{removeFirst}()$
    - Remove at back - $\text{removeLast}()$
- **Stacks**
    - Last In, First Out (LIFO)
    - Last element inserted will be the first item to be removed (popped)
- **Queues**
    - First In, First Out (FIFO)
    - First element inserted will be the first item to be removed (popped)
- **Stack/Queue Function Terminology**
    - Insertion - $\text{push}(\text{element})$
    - Remove - $\text{pop}()$
    - Peek - $\text{peek}()$
    - Check if empty - $\text{isEmpty}()$
    - Check if full - $\text{isFull}()$

Trees

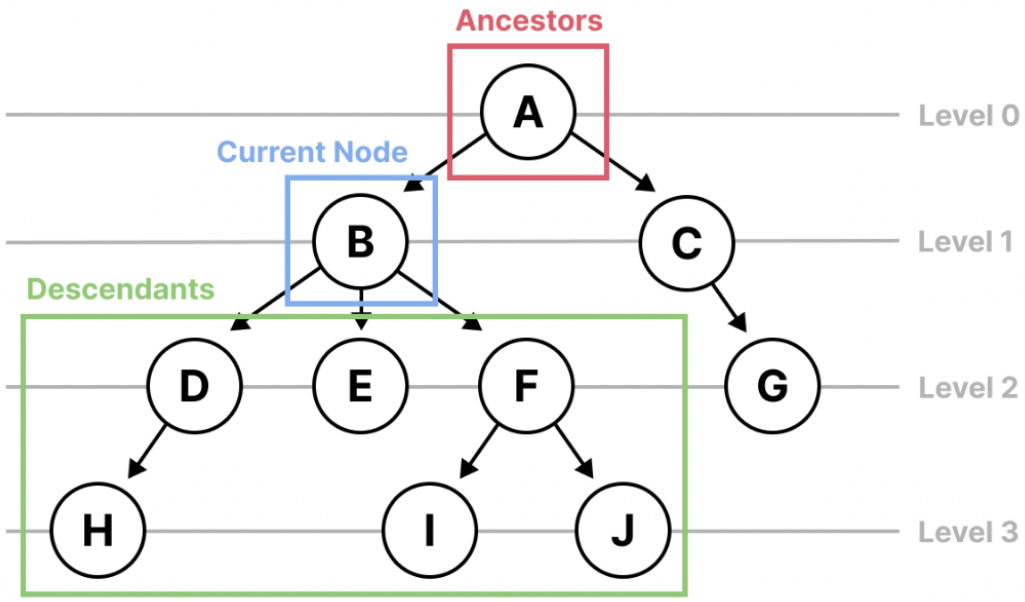
- **Trees (General)**
    - A system of nodes with parents, children, and siblings
    - An example of a tree:



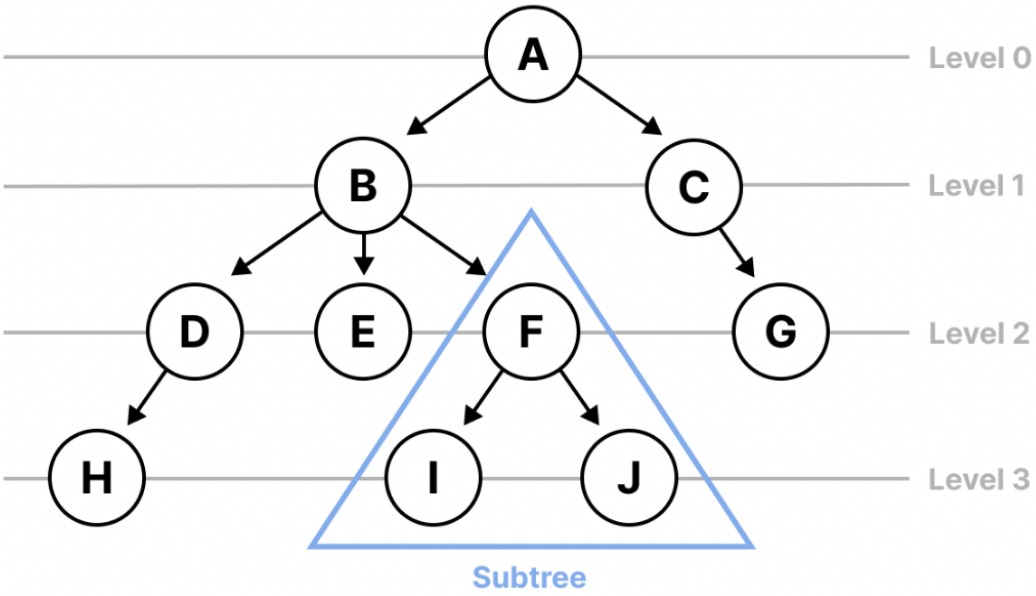
- *Roots* are the top node from which all other nodes branch off
- *Branches* are nodes that have a parent and have children
- *Leaves* are nodes without children
- *Level* of a node is which "generation" the node is in
  - Levels visualization:



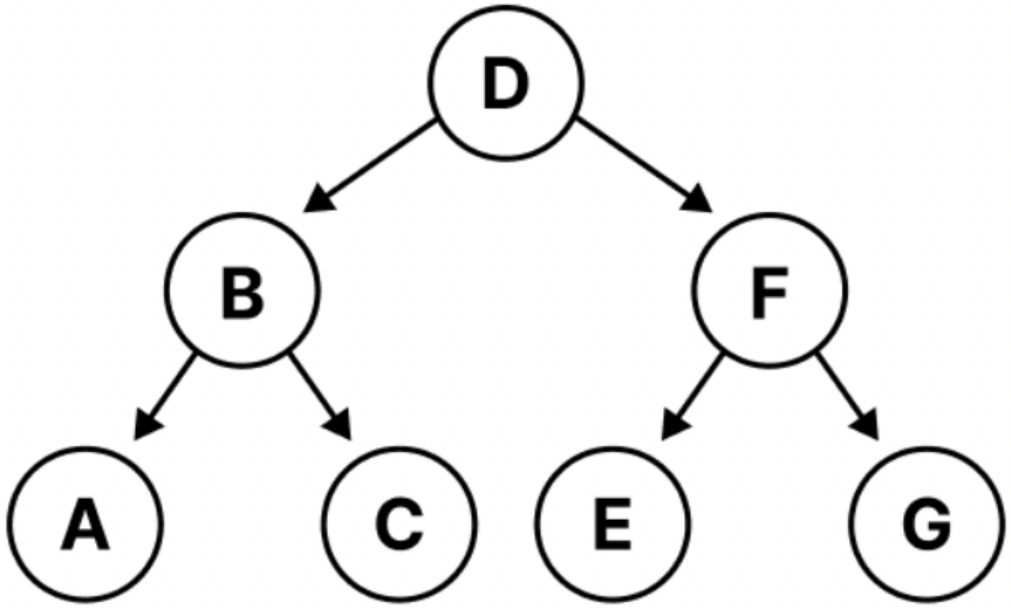
- *Ancestors* are any nodes that appear in the path up from the current node to the root, including the root
- *Descendants* are any nodes that appear in any path down beginning at the node, not including the node itself
  - Ancestor/Descendant Visualization:



- *Subtree* is the collection of the node and all of its children
  - Subtree Visualization:



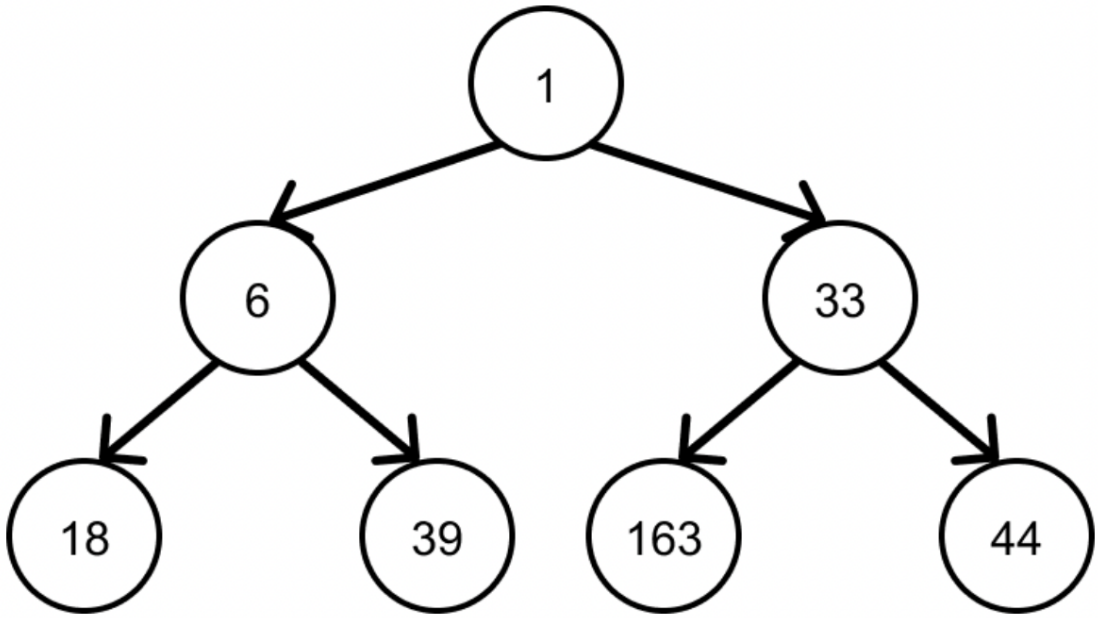
  - Both singly and doubly linked lists are trees
- **Binary Search Trees (BST)**
  - Binary search trees have maximum two children for each node, where every node in the left subtree must be less than or equal to the node and every node in the right subtree must be greater than the node
  - Sample binary search tree:



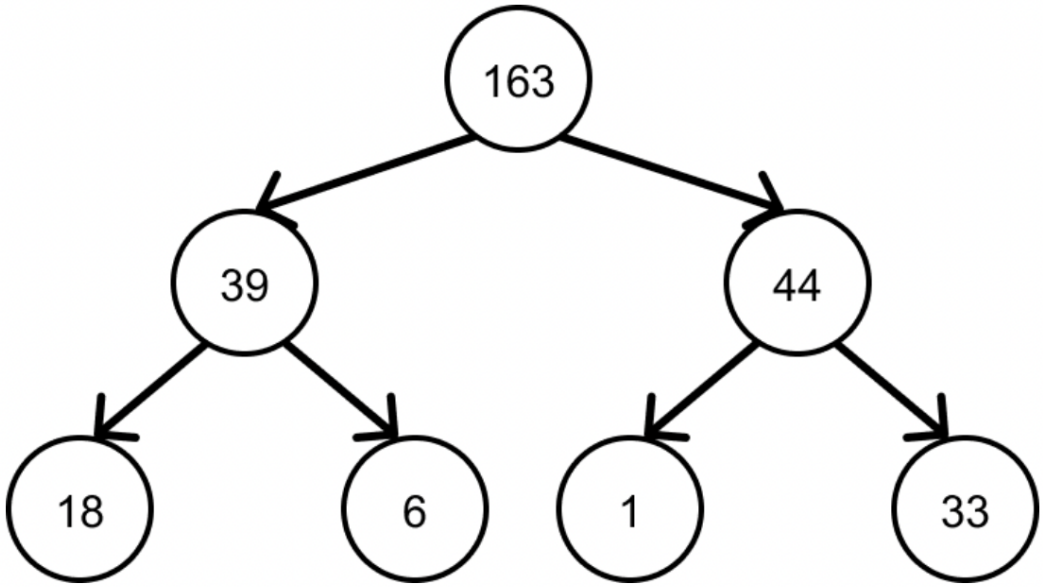
- **Runtime Analysis**
  - Insertion - $[O(n), \Omega(1)]$
  - Search/Deletion - $[O(n), \Omega(\log(n))]$
  - Traversal - $[O(n), \Omega(n), \Theta(n)]$
- **BST Traversals**
  - Preorder Traversal - Center, Left, Right
  - Inorder Traversal - Left, Center, Right
  - Postorder Traversal - Left, Right, Center
  - Level Order Traversal - Level by level, left to right

Heaps

- **Heaps Introduction**
  - Heaps have binary tree structure (maximum two children per node)
  - Heaps are filled in by level. Inserted nodes are put at the next available location in the tree, creating complete binary tree
  - *Completeness Property*
    - Any "gaps" in a given level of nodes must be filled, unless it is the last level (in which it may or not be filled)
    - The last level may have missing nodes, but only on the rightmost side
    - Any extra nodes on the last level must be filled from left to right
  - *Min-Heap Property* - A parent's data must always be less than or equal to its children's data
    - Min-Heap Visualization:

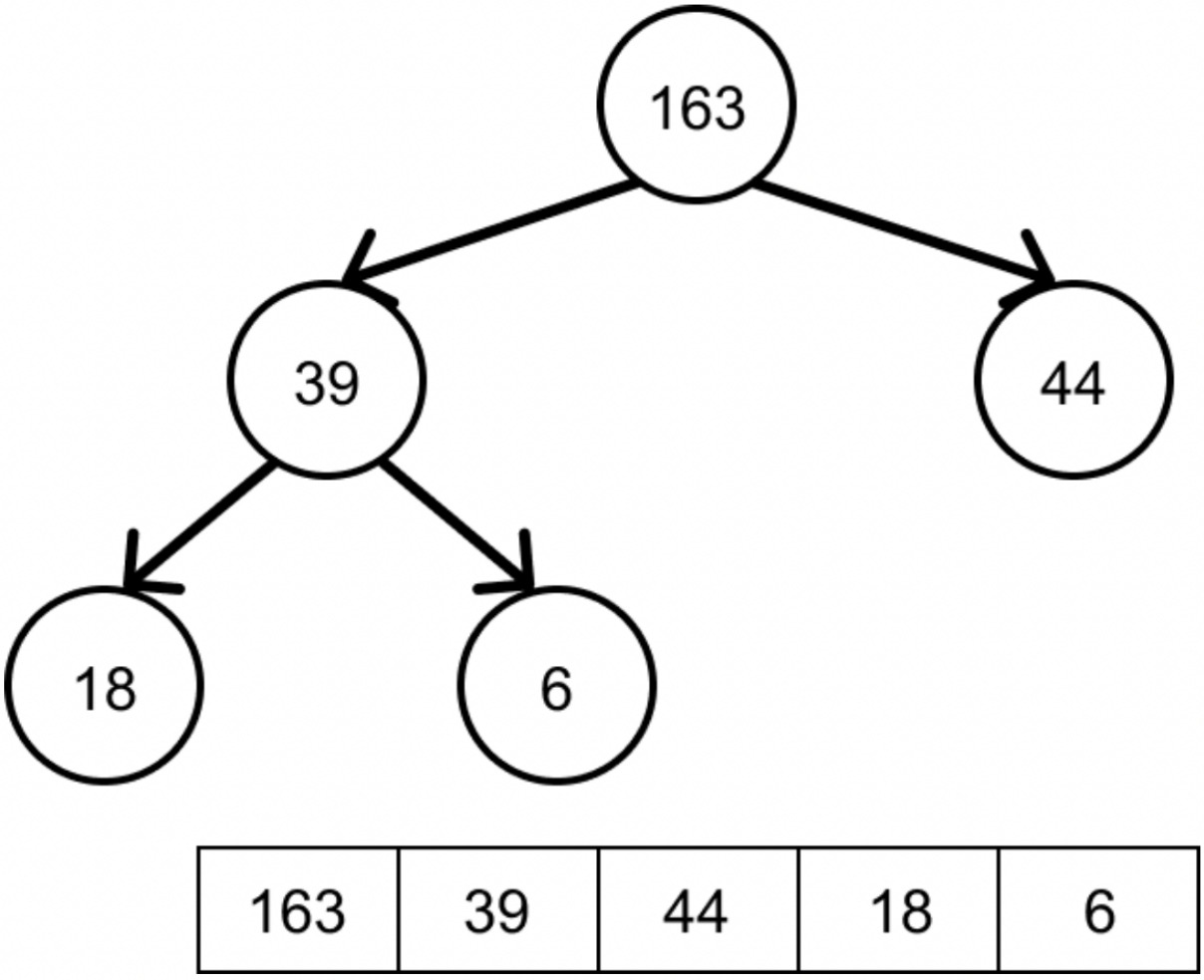


- *Max-Heap Property* - A parent's data must always be greater than or equal to its children's data
  - Max-Heap Visualization:



- **Heap Insertion**
  - Inserting value at next available space may work, but it may also violate min-heap or max-heap properties, so we need to use the Bubble Up function to move the inserted value up in the heap
  - *Bubble Up* - If inserted node value is (based on comparator) greater than its parent, then swap the parent and the inserted node, then recursively call on the new position
- **Heap Deletion**
  - Swap top node with last node, then call Bubble Down function to move new root node into its place

    - *Bubble Down* - If new root node is (based on comparator) less than its children, swap with child. Automatically pick the child node with higher value. If children have equal values, pick the left node. Then, recursively call on new location of root node.
- Heap Array Implementation Visualization:



Sorting

- **Bubble Sort**
    - While swaps are still occurring or until we have iterated through the list $n-1$ times, we iterate through the list and swap any two neighbor elements that are out of order.
    - Runtime Analysis - $[O(n^2), \Omega(n)]$
- **Selection Sort**
    - Create a sorted section (initial length of 0) and an unsorted section (initial length of n). The smallest element is selected from the unsorted section of the list and is swapped with the last element of the sorted section. Repeat until the sorted section encompasses the entire array.
    - Runtime Analysis - $[O(n^2), \Omega(n^2), \Theta(n^2)]$
- **Insertion Sort**

    - Create a sorted section (initial length of 1) and an unsorted section (initial length of n - 1). The first element is then inserted into its correct position in the sorted section. Repeat until the sorted section encompasses the entire array.
    - Runtime Analysis - $[O(n^2), \Omega(n)]$
- **Merge Sort**
    - Split the array into two parts and recursively call on both lists, with an array of length 1 as the base case (return the array in this case). Return the merger of the two resultant recursive calls.
    - Runtime Analysis - $[O(n \log(n)), \Omega(n \log(n)), \Theta(\log(n))]$
- **QuickSort**
    - Runtime Analysis - $[O(n^2), \Omega(n \log(n))]$