Algorithms You Should Know

Week 7

Searching Algorithms

Linear Search

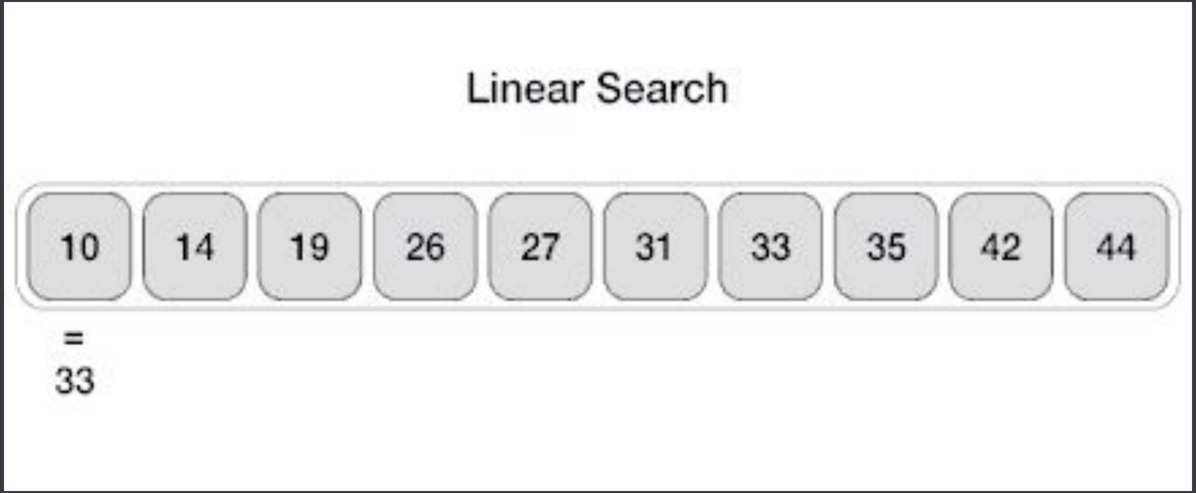
Description: Check each element sequentially until your target element is found in the list.

Worst complexity: O(n)

Best complexity: O(1)

Space complexity: O(1)

Best Use Cases: If you know an element is towards the end / beginning of the list, you can run linear search on that end.



Linear Search - Java Code

```
public static int search(int arr[], int x) {
    int n = arr.length;
    for (int i = 0; i < n; i++){
        if (arr[i] == x)
            return i;
    }
    return -1;
}
```

Binary Search

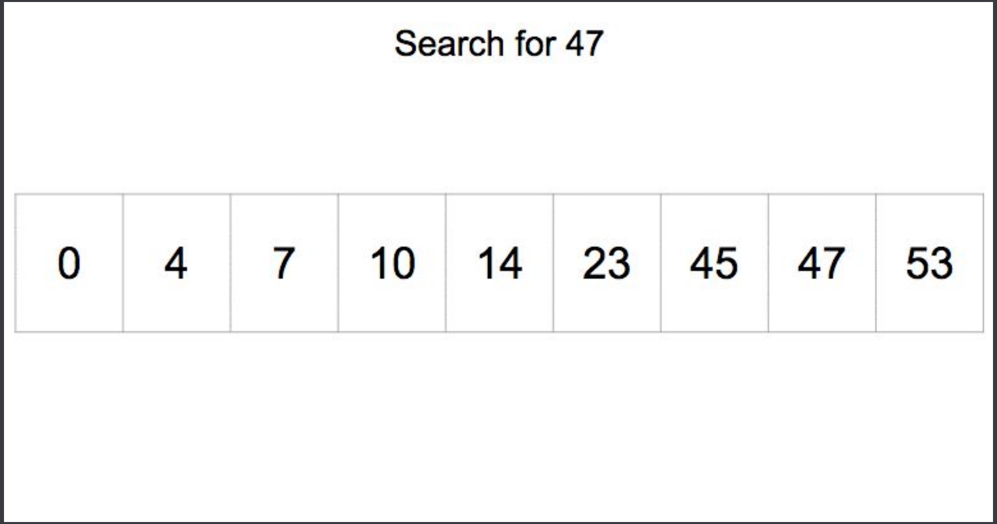
Description: Compares the target value to the middle element of the section you are checking in a sorted list. Splits list into half if target does not equal middle element, continues on left split if element is less than middle element, right split otherwise.

Worst complexity: O(log n)

Best complexity: O(1)

Space complexity: O(1)

Best Use Cases: Finding a target element in a sorted array.



Binary Search - Java Code

```
class BinarySearch {
    // Returns index of x if it is present in arr[l, …,r], else return -1
    int binarySearch(int arr[], int l, int r, int x){
        if (r >= l) {
            int mid = l + (r - l) / 2;
            // If the element is present at the middle itself
            if (arr[mid] == x)
                return mid;
            // If element is smaller than mid then it must be in left
subarray
            if (arr[mid] > x)
                return binarySearch(arr, l, mid - 1, x);
            // Else the element can only be present in right subarray
            return binarySearch(arr, mid + 1, r, x);
        }
        // We reach here when element is not present in array
        return -1;
    }
```

Binary Search Tree

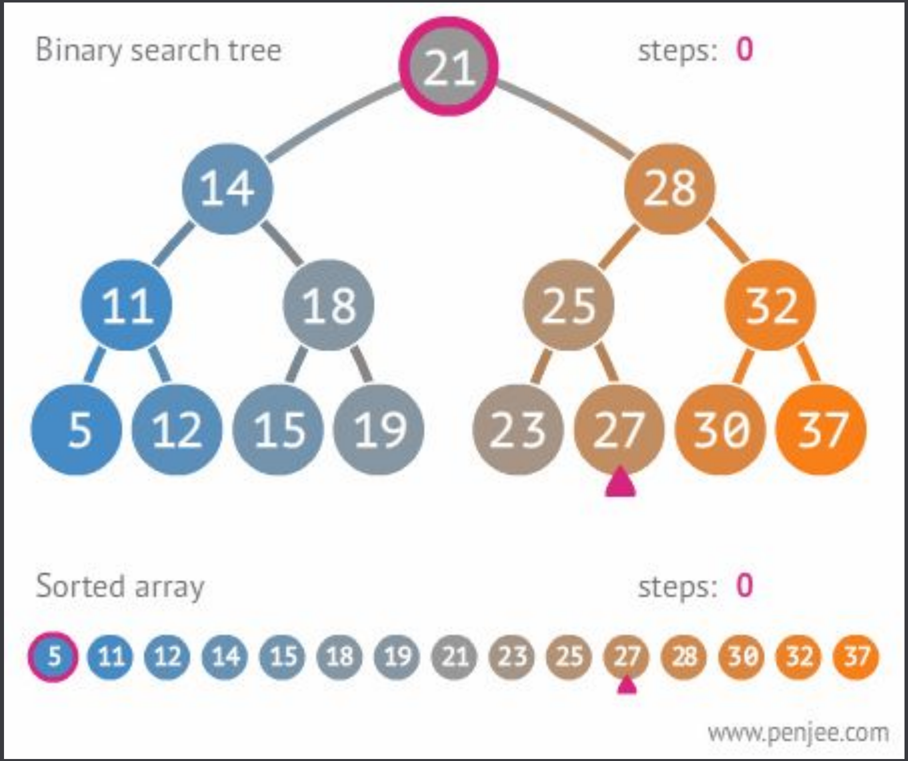
Description: A tree data structure whose internal nodes each store a key greater than all the keys in the node's left subtree and less than those in its right subtree.

Worst complexity: O(n)

Best complexity: O(log n)

Space Complexity: O(n)

Best Use Case: When you want to compare lower/greater items and store them effectively (especially when you know the stream of items is not in strictly ascending/descending order, but a mix of the two).



Binary Search Tree - Java Code

```
public Node search(Node root, int key){
    // Base Cases: root is null or key is present at root
    if (root==null || root.key==key)
        return root;
     // Key is greater than root's key
    if (root.key < key)
       return search(root.right, key);
     // Key is smaller than root's key
    return search(root.left, key);
}
```

AVL Tree

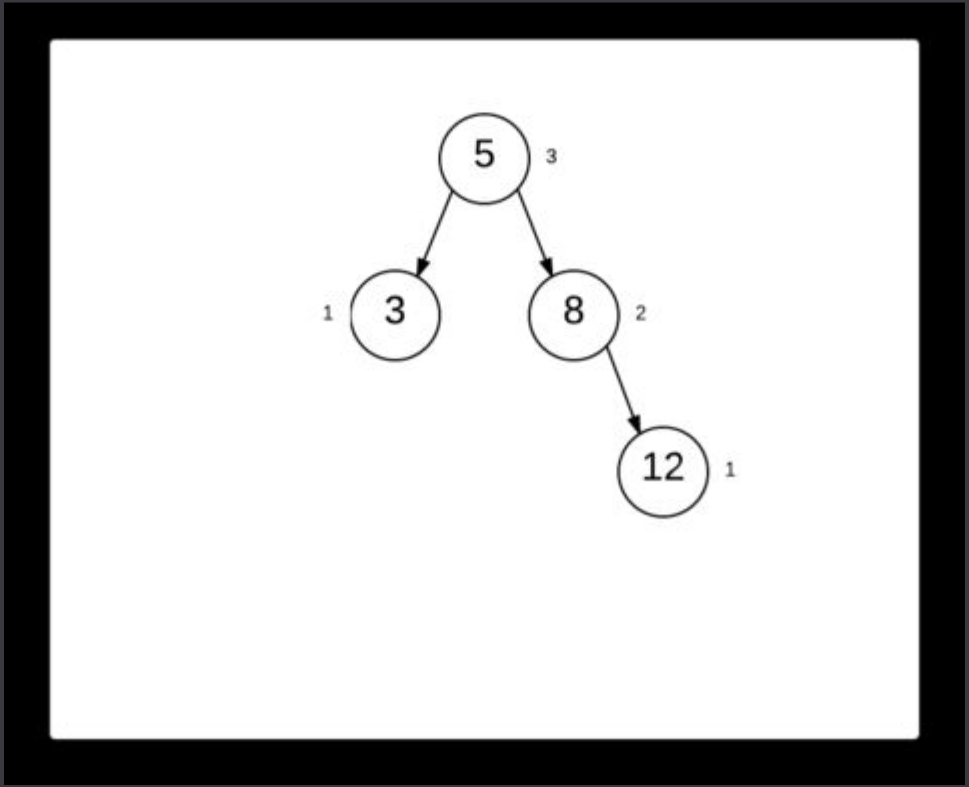
Description: A self-balancing binary search tree where the heights of two child subtrees of any node differ by at most one. If at any time they differ by more than one, they rotate left/right to restore this property.

Worst complexity: O(log n)

Best complexity: O(log n)

Space Complexity: O(n)

Best Use Case: When you are sorting a stream of elements and want to have a guaranteed runtime of O(log n).



Trie

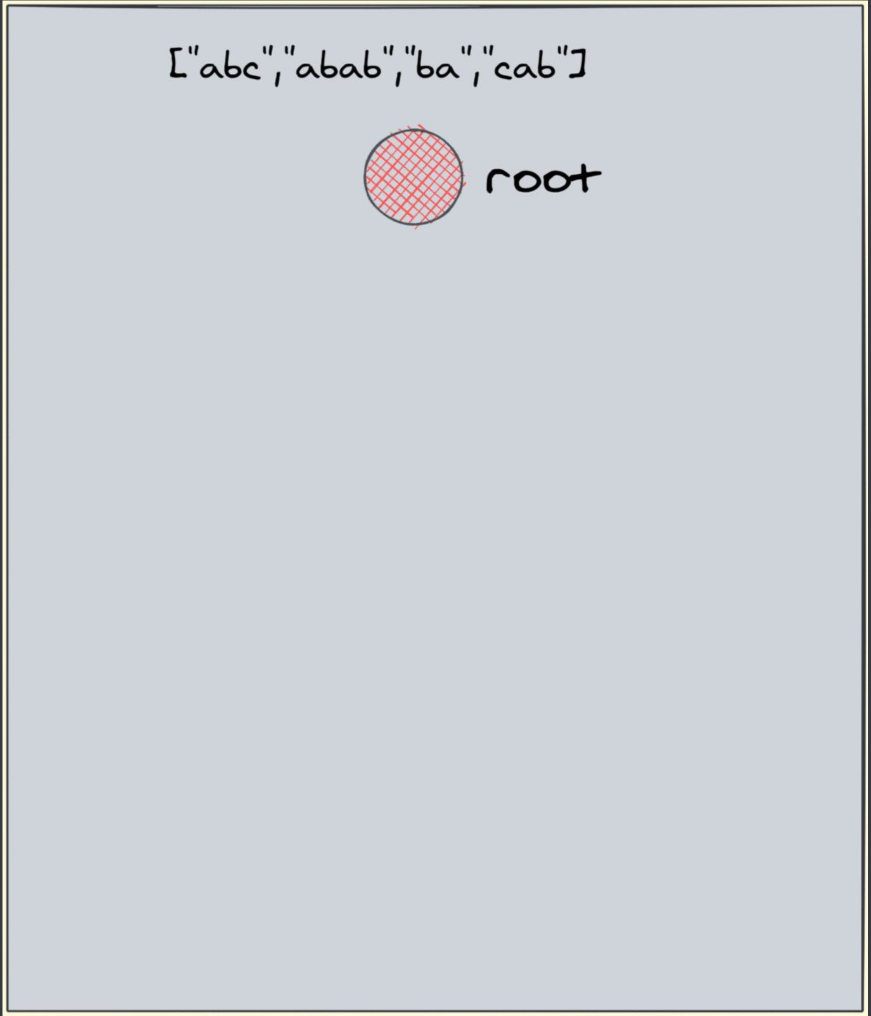
Description: A tree data structure used for locating specific keys from within a set. The nodes are not defined by the entire key, but by individual characters of the input string being stored.

Worst Complexity: O(k) where k is the length of a key

Best Complexity: O(k) where k is the length of a key

Space Complexity: O(k * n) where k is the length of a key, and n is the number of keys

Best Use Case: Used for storing and parsing through strings of text, either character by character (in a word), or word by word (in a sentence).



Sorting Algorithms

Selection Sort

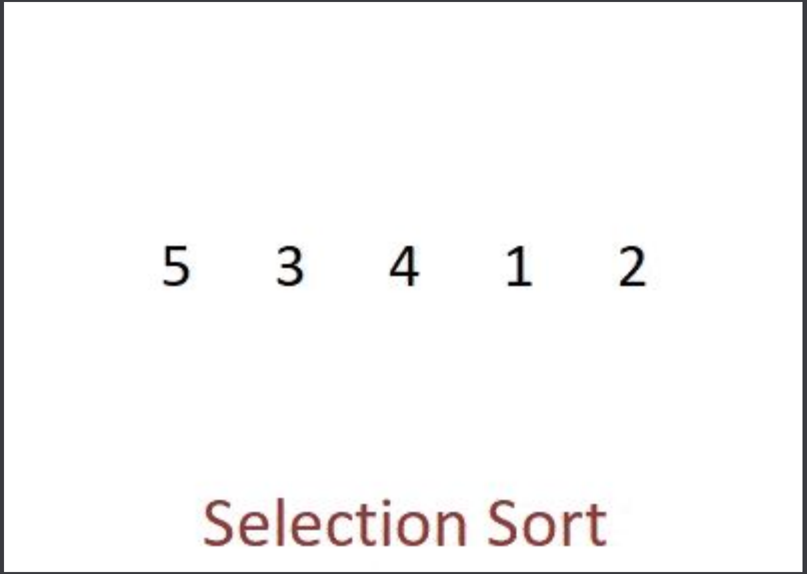
Description: Sorts an array by repeatedly finding the minimum element from an unsorted part and putting it at the beginning of the list.

Worst complexity: O(n^2)

Best complexity: O(n^2)

Space complexity: O(1)

Best Use Case: When you want to use a simple and in-place sorting algorithm. Also when we have an almost-sorted array with only a few minimum elements that are out of place towards the beginning of our list.



Selection Sort - Java Code

```
void sort(int arr[]){
    int n = arr.length;
      // One by one move boundary of unsorted subarray
    for (int i = 0; i < n-1; i++){
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
        // Swap the found minimum element with the first element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Insertion Sort

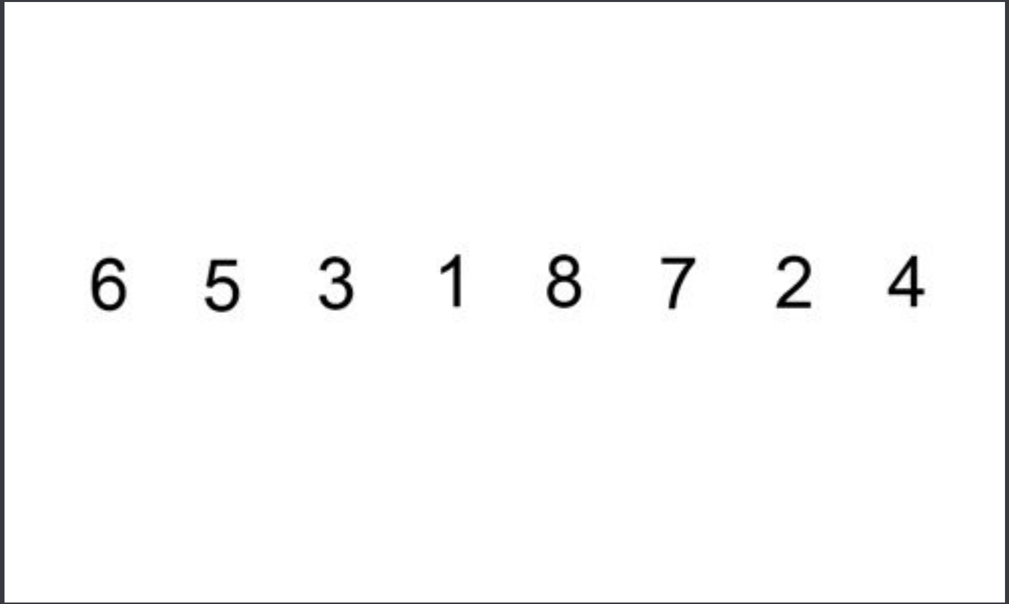
Description: The list is split into a sorted and an unsorted part where elements from the unsorted part are picked and placed at the correct position in the sorted part.

Worst complexity: O(n^2)

Best complexity: O(n)

Space complexity: O(1)

Best Use Case: When you know you only have a few minimum elements to sort in the array.



Insertion Sort - Java Code

```
void sort(int arr[]){
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Merge Sort

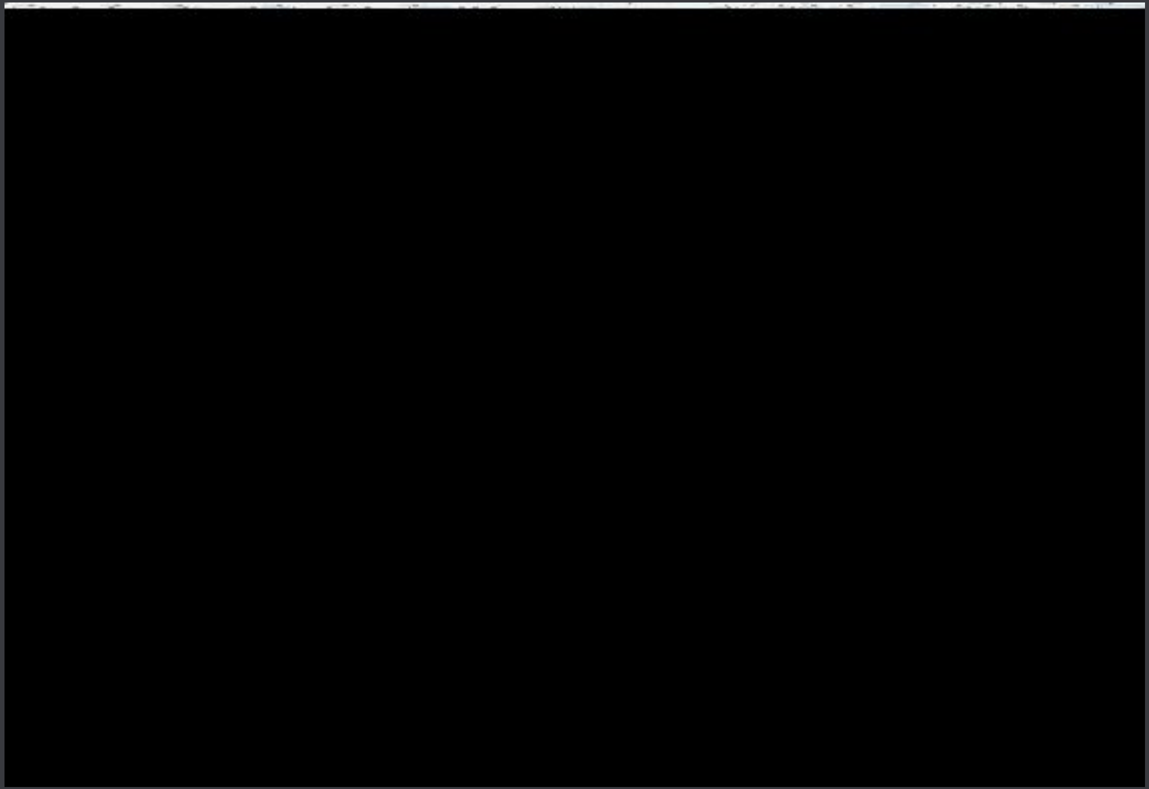
Description: A Divide and Conquer algorithm that divides the list into two halves, calls itself for the two halves, and then merges the two sorted halves.

Worst complexity: O(n*log(n))

Best complexity: O(n*log(n))

Space complexity: O(n)

Best Use Case: When we want a guaranteed O(n*log n) runtime, which is faster than Insertion and Selection sort in the worst case.



Merge Sort - Java Code

```
// Main function that sorts arr[l..r] using
 // merge()
 void sort(int arr[], int l, int r)
 {
     if (l < r) {
         // Find the middle point
         int m =l+ (r-l)/2;

         // Sort first and second halves
         sort(arr, l, m);
         sort(arr, m + 1, r);

         // Merge the sorted halves
         merge(arr, l, m, r);
     }
 }
```

Quick Sort

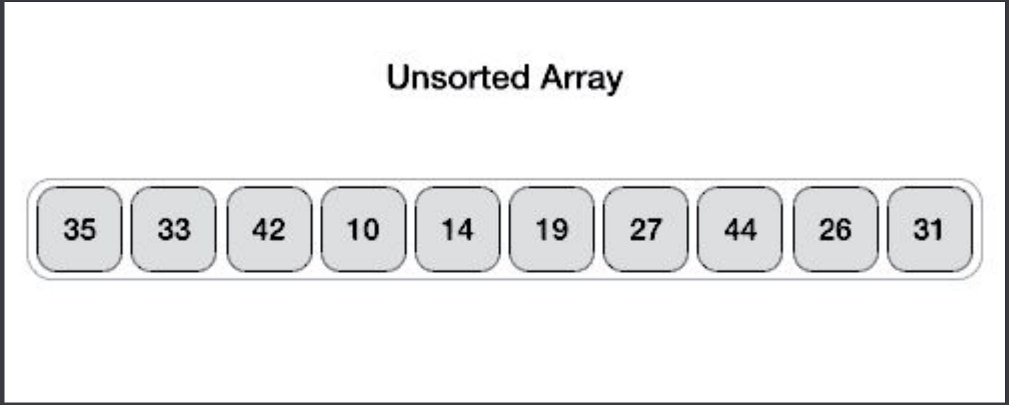
Description: A Divide and Conquer algorithm that picks an element as pivot and partitions the list around the picked pivot.

Worst complexity: O(n^2)

Best complexity: O(n*log(n))

Space complexity: O(n) (Average/Best case is O(log n))

Best Use Case: When we have a good pivot in the middle of our list and have larger unsorted lists to sort.



Quick Sort - Java Code

```
/* The main function that implements QuickSort
          arr[] --> Array to be sorted,
          low --> Starting index,
          high --> Ending index
 */
static void quickSort(int[] arr, int low, int high){
    if (low < high){
        // pi is partitioning index, arr[p] is now at right place
        int pi = partition(arr, low, high);

        // Separately sort elements before partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Heap Sort

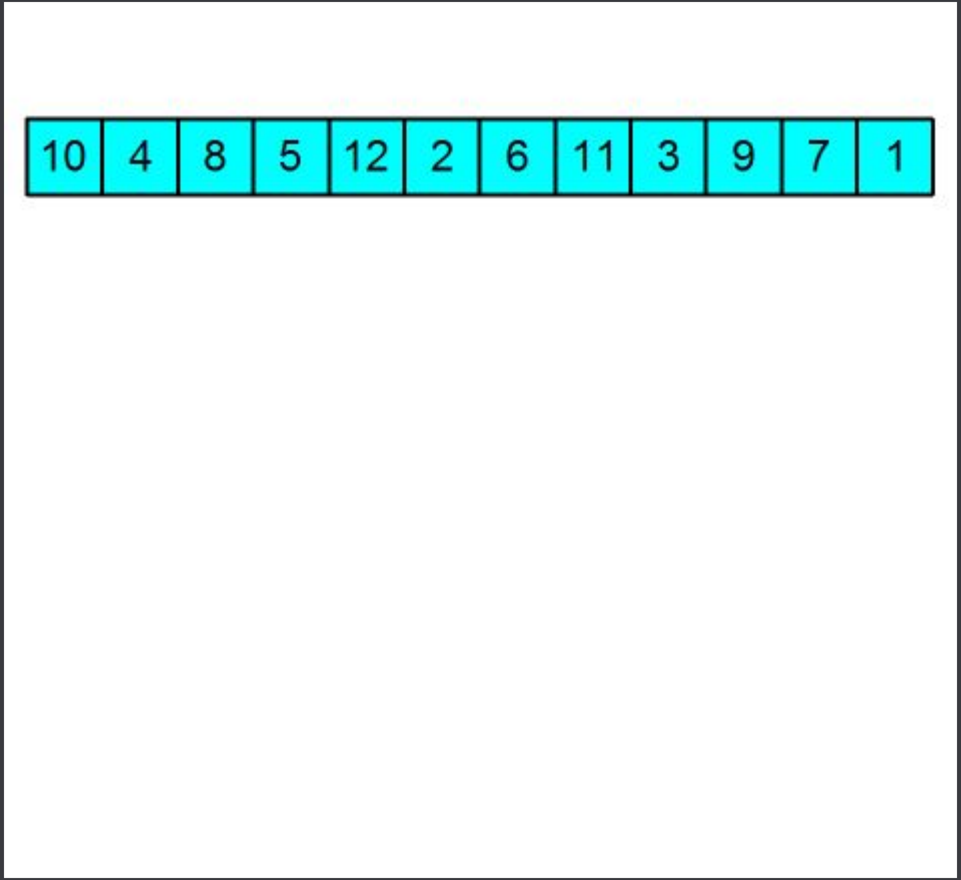
Description: A sort that is similar to Selection Sort where we first find the minimum element and place the minimum element at the beginning, but with a Binary Heap data structure. We repeat the same process for the remaining elements.

Worst complexity: O(n*log(n))

Best complexity: O(n*log(n))

Space complexity: O(1)

Best Use Case: When we have a large number of elements to sort, don't want to create any additional data structures, and want a guaranteed runtime of O(n*log n).



Heap Sort - Java Code

```
public void sort(int arr[]){
    int n = arr.length;
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

     // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

Graph Algorithms

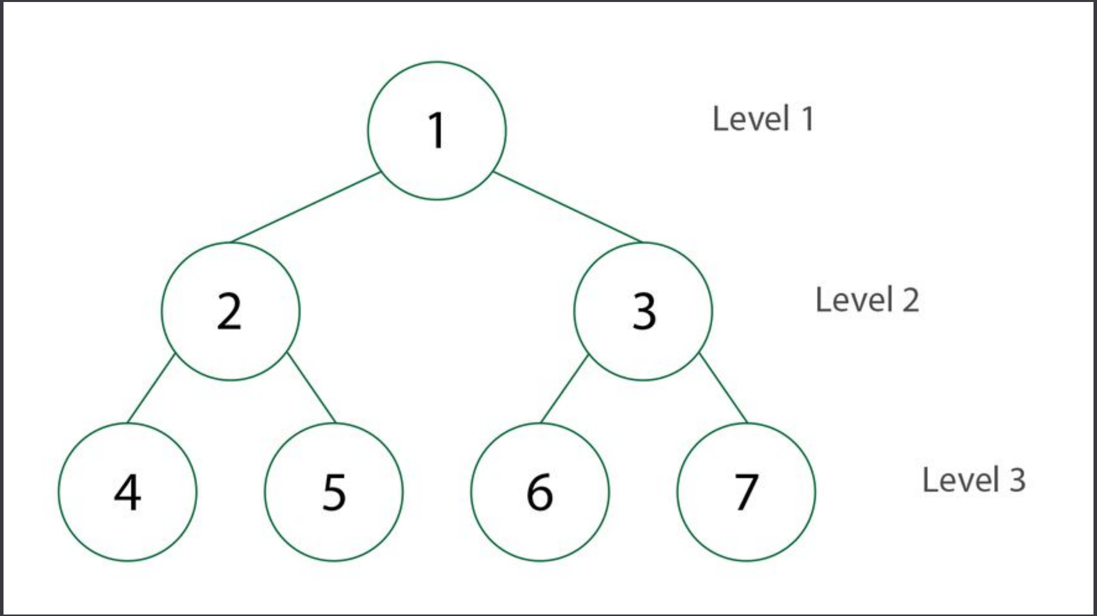
Breadth-First Search

Description: Uses a boolean visited set to go through nodes in a graph level by level from a starting node (i.e. we visit all the neighbors of the starting node before traversing to the next node).

Time Complexity: O(V + E)

Space complexity: O(V)

Best Use Case: When you want to search through the neighbors of a certain node in a graph first before traversing through the whole depth of a path.



BFS - Java Code

```
void BFS(int n){
     //initialize boolean array for holding the data
    boolean nodes[] = new boolean[V];
    int a = 0;
    nodes[n]=true;
    queue.add(n); //root node is added to the top of the queue
    while (queue.size() != 0) {
        n = queue.poll();//remove the top element of the queue
        System.out.print(n+" ");//print the top element of the queue
     //iterate through the linked list and push all neighbors into queue
        for (int i = 0; i < adj[n].size(); i++){
            a = adj[n].get(i);
         //only insert nodes into queue if they have not been explored already
            if (!nodes[a]){
                nodes[a] = true;
                queue.add(a);
            }
        }
    }
}
```

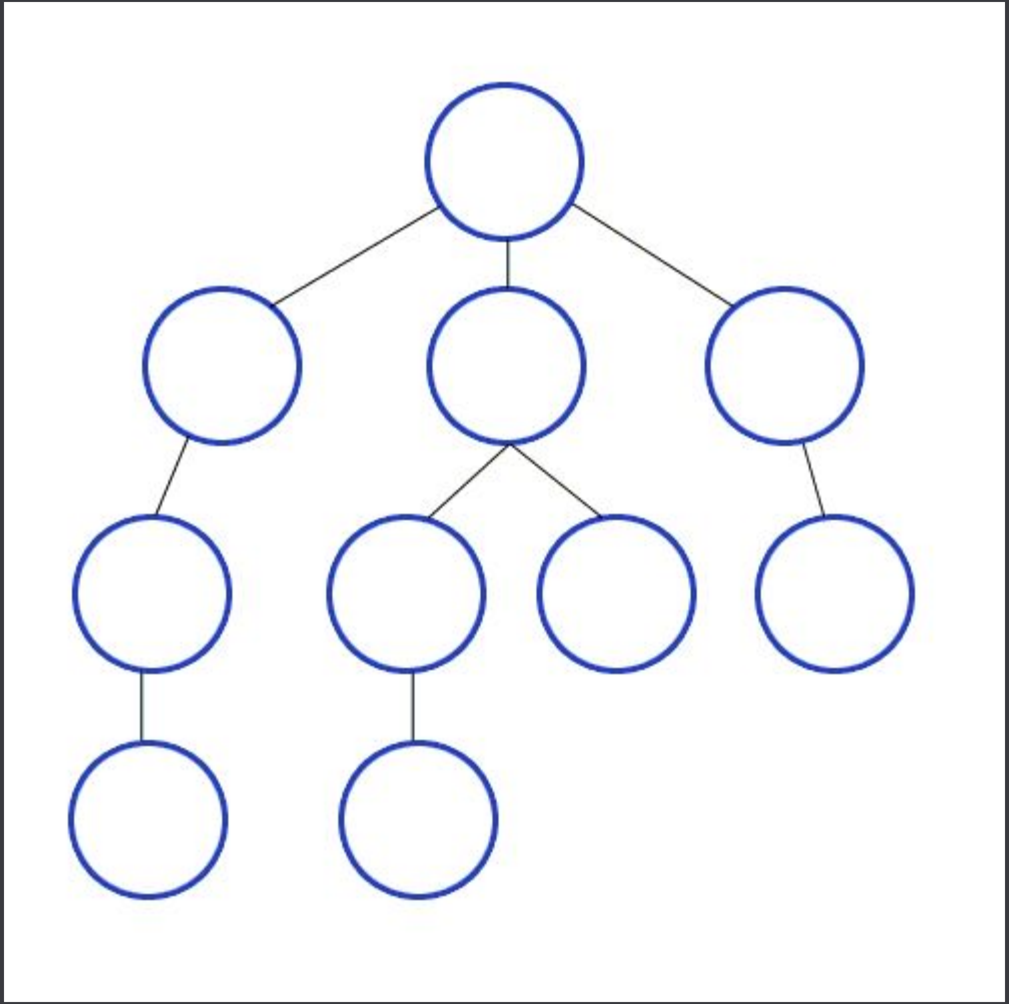
Depth-First Search

Description: Uses a boolean visited set to go through nodes in a graph. From the starting node, we explore as far as possible along each branch before backtracking.

Time Complexity: O(V + E)

Space complexity: O(V)

Best Use Case: When you have solutions located much deeper and further away from your starting node in the graph.



DFS - Java Code

```
public void dfs(int start) {
    Stack<Integer> stack = new Stack<Integer>();
    boolean[] isVisited = new boolean[adjVertices.size()];
    stack.push(start);
    while (!stack.isEmpty()) {
        int current = stack.pop();
        if(!isVisited[current]){
            isVisited[current] = true;
            visit(current);
            for (int dest : adjVertices.get(current)) {
                if (!isVisited[dest])
                    stack.push(dest);
            }
    }
}
```

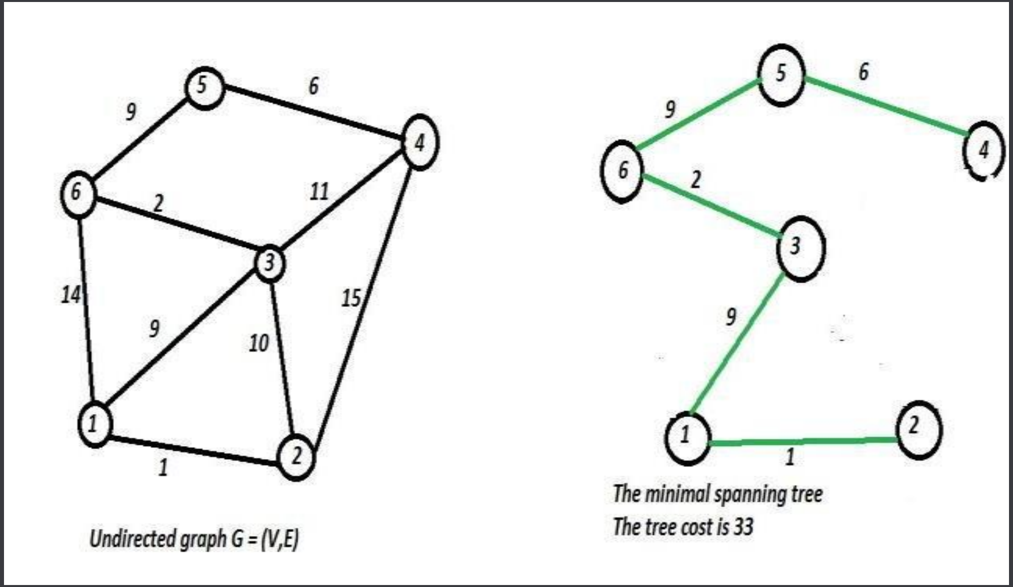
Minimum Spanning Tree

Description: A subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

Prim's Algorithm Time Complexity: O((V+E)log V)

Kruskal's Algorithm Time Complexity: O(E log V)

Best Use Case: When you want to find the lowest costing path that connects all the points in a graph together (i.e. a flights problem with a starting location and you are trying to find the lowest cost to fly to all other locations in the graph).



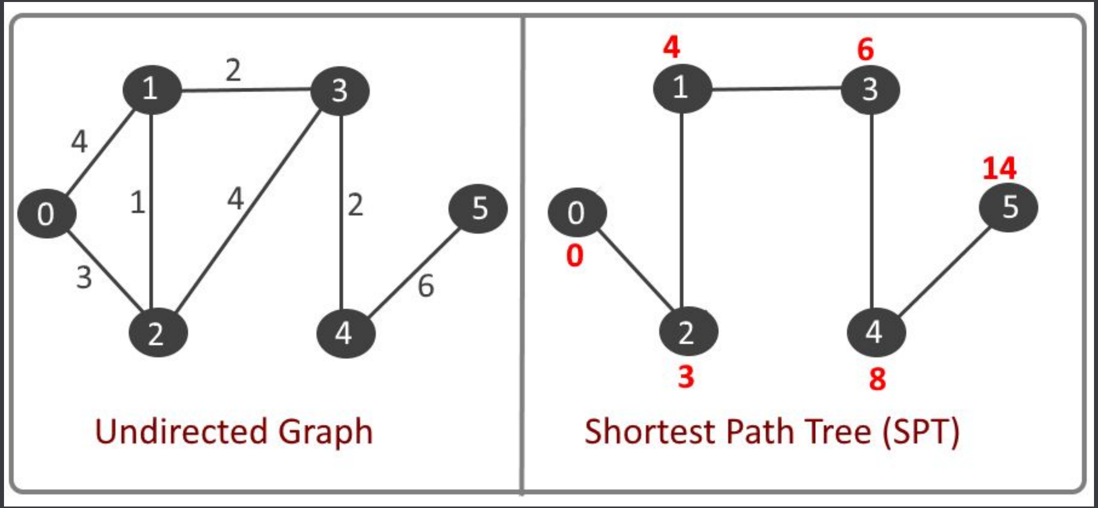
Shortest Paths Tree

Description: Finds the lowest costing path from a specific starting node, to a certain destination node in the graph.

Dijkstra Time complexity: O(E+V*log V)

Bellman-Ford Time complexity: O(V*E)

Best Use Case: When you want to find the lowest costing path between two nodes, use Dijkstra if you have non-negative edge weights and Bellman-Ford if you do, since Dijkstra is faster.



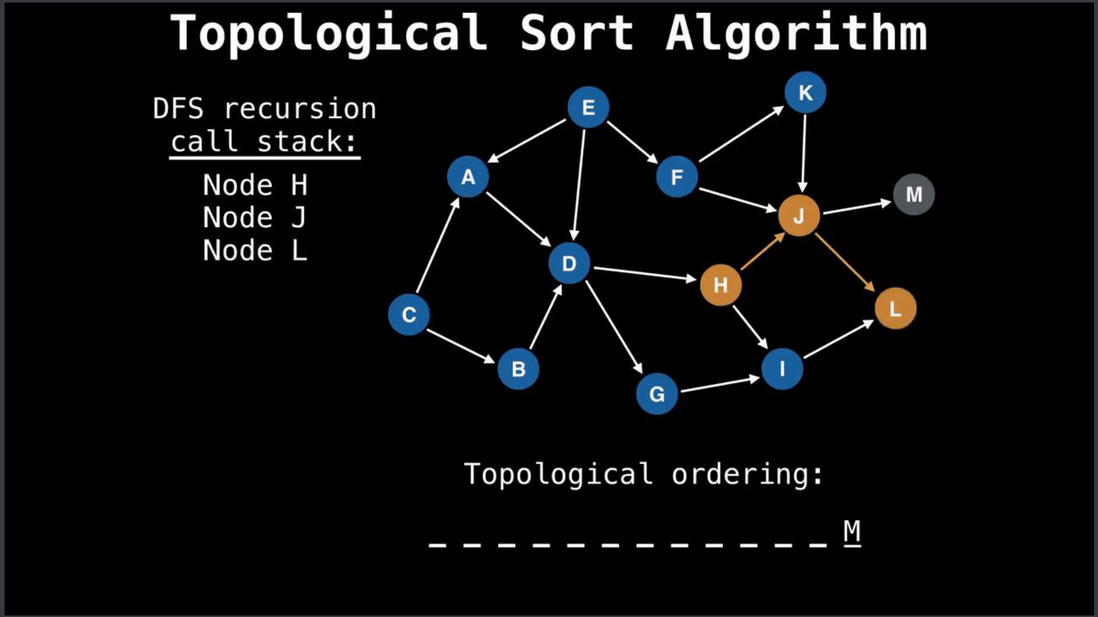
Topological Sort

Description: Sorts a Directed Acyclic Graph (DAG) in a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering.

Time complexity: O(V+E)

Space complexity: O(V)

Best Use Case: When you (1) want to find a cycle in a graph, or (2) are working on a problem where you cannot access certain nodes before you visit others first (i.e. dependency problems).


Topological Sort Algorithm
DFS recursion call stack:
Node H
Node J
Node L
A
B
C
D
E
F
G
H
I
J
K
L
M
Topological ordering:
_ _ _ _ _ _ _ _ _ _ _ _ M

Topo Sort - Java Code

```
void topologicalSort(){
    Stack<Integer> stack = new Stack<Integer>();

     // Mark all the vertices as not visited
    boolean visited[] = new boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;
    // Call the recursive helper function to store
    // Topological Sort starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, stack);
    // Print contents of stack
    while (stack.empty() == false)
        System.out.print(stack.pop() + " ");
}
```

Weekly Problems 🤗 - Searching Algorithms

https://leetcode.com/problems/search-insert-position

https://leetcode.com/problems/first-bad-version

https://leetcode.com/problems/two-sum-ii-input-array-is-sorted

https://leetcode.com/problems/find-a-peak-element-ii/

https://leetcode.com/problems/koko-eating-bananas/

https://leetcode.com/problems/find-minimum-in-rotated-sorted-array

https://leetcode.com/problems/find-peak-element

Weekly Problems 🤗 - Sorting Algorithms

https://leetcode.com/problems/sorting-the-sentence/

https://leetcode.com/problems/find-target-indices-after-sorting-array/

https://leetcode.com/problems/squares-of-a-sorted-array

https://leetcode.com/problems/minimum-absolute-difference

https://leetcode.com/problems/minimum-difference-between-largest-and-smallest-valu
e-in-three-moves

https://leetcode.com/problems/sort-colors

Weekly Problems 🤗 - Graph Algorithms

https://leetcode.com/problems/course-schedule/

https://leetcode.com/problems/find-eventual-safe-states/

https://leetcode.com/problems/find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance/

https://leetcode.com/problems/min-cost-to-connect-all-points/

https://leetcode.com/problems/network-delay-time/