Final Review

1. Linked Lists

- Adding elements: You can add elements to different types of linked lists using appropriate insertion methods.
- Deleting elements: Elements can be removed by adjusting the pointers in the list.

Linked Lists Using Array Lists:
Pros:

- uses indices
- easy to code

Cons:

- Has overhead to use
- Consumes extra memory

Linked Lists Using Nodes:
Pros:

- Memory efficient
- Efficient insertions and deletions
- Minimal overhead

Cons:

- More difficult to code
- No indices

We will know if a circularly linked list is done when the tail points to the head.

```
head.next = tmp;
tmp.next = head;
```

ArrayLists vs. LinkedList

Pros of ArrayList:

- An array, and contiguous
- Inherits from List class
- Better for storing and accessing

Cons of ArrayList:

- With every deletion, the array needs to be rebuilt
- Slow
- Default capacity of 10

Pros of LinkedList:

- Uses doubly linked list
- Fast
- Can act as a list and a queue
- Better for data manipulation

Cons of LinkedList:

- Not contiguous

Example:

```
// Singly Linked List
LinkedList<Integer> singlyLinkedList = new LinkedList<>();
singlyLinkedList.add(10); // Adding element
singlyLinkedList.removeFirst(); // Deleting element

// Doubly Linked List
LinkedList<Integer> doublyLinkedList = new LinkedList<>();
doublyLinkedList.add(20); // Adding element
doublyLinkedList.removeLast(); // Deleting element
```

2. Stacks and Queues

- Stacks: LIFO (Last-In-First-Out) structure, useful for managing function calls, undo operations, etc.
- Queues: FIFO (First-In-First-Out) structure, used in tasks like scheduling, breadth-first search, etc.

Example:

```
// Stack
Stack<Integer> stack = new Stack<>();
stack.push(30); // Pushing element onto the stack
stack.pop(); // Popping element from the stack

// Queue
Queue<Integer> queue = new LinkedList<>();
queue.offer(40); // Adding element to the queue
queue.poll(); // Removing element from the queue
```

3. Trees

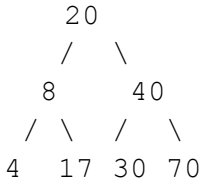
- Basic tree terminology: Root, node, edge, leaf, parent, child.
- Types of trees: Binary tree, binary search tree, balanced tree, etc.

Trees:
A collection of vertices & edges where no cycle is produced.

Binary Tree:
It's a tree where each vertex has 0, 1, or 2 leaf nodes.

Tree vocab:
Full - All nodes have 0,1, or 2 children
Complete - All levels filled but last level is filled from the left
Perfect - full and complete
Depth is the longest branch, count edges without root

Binary Search Trees
L.data < p.data < r.data (left, parent, right)



Example:

```
// Binary Tree
class TreeNode
   {
     int val;
     TreeNode left;
     TreeNode right;
     TreeNode(int val) {
         this.val = val;
     }
}

// Binary Search Tree
class BSTNode
   {
     int val;
     BSTNode left;
     BSTNode right;
```

```
        BSTNode(int val)
        {
            this.val = val;
        }
    }
```

4. Tree Creation and Traversal

- Creating trees using classes and nodes.
- Tree traversal: Pre-order, in-order, post-order.

Pre Order T - Visit, Left, Right

Post Order T - Left, Right, Visit

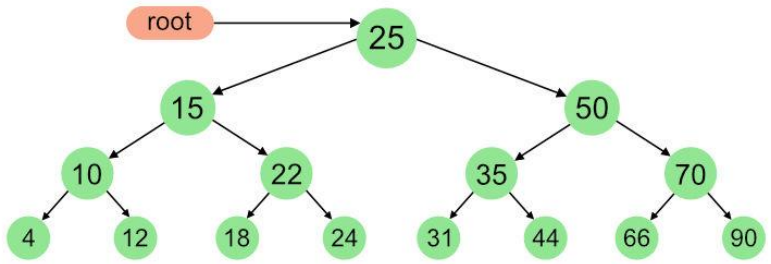
In Order T - Left, Visit, Right
QOTD: Given a tree of T, M, G, C, B, K, H, W, U, V, X give the preorder, postorder, and inorder of the tree.

Pre Order: T -> M -> G -> C -> B -> K -> H -> W -> U -> V -> X
Post Order: B -> C -> H -> K -> G -> M -> V -> U -> X -> W -> T
In Order: B -> C -> G -> H -> K -> M -> T -> U -> V -> W -> X

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Recap:
Full: A binary tree with each node having 0 or 2 children
Complete: All levels filled, but level (n-1) is filled “completely” from the left
Perfect: All levels filled to max capacity.

```
   Example:
   // Creating a binary tree
   TreeNode root = new TreeNode(50);
   root.left = new TreeNode(30);
   root.right = new TreeNode(70);

// Pre-order traversal
   Public void preOrder(TreeNode node)
     {
       if (node != null)
      {
           System.out.print(node.val + " ");
           preOrder(node.left);
           preOrder(node.right);
       }
   }

// Post-order traversal
private void postOrderTraversalRecursive(TNode node)
     {
          if (node != null) {
               postOrderTraversalRecursive(node.left);
               postOrderTraversalRecursive(node.right);
               System.out.print(node.data + ", ");
          }
     }

// In-order traversal
private void inOrderTraversalRecursive(TNode node)
     {
          if (node != null) {
               inOrderTraversalRecursive(node.left);
               System.out.print(node.data + ", ");
               inOrderTraversalRecursive(node.right);
          }
     }
```

5. Recursion

- Recursion involves solving a problem by breaking it into smaller subproblems.
- Recursive functions call themselves with modified inputs.

Example:

```
int factorial(int n)
{
    if (n == 0)
    {
        return 1;
    }
    return n * factorial(n - 1);
}
```

6. Sorting Algorithms

- Bubble sort: Repeatedly swap adjacent elements if they're in the wrong order.
- Insertion sort: Build a sorted portion of the array by inserting elements in their correct positions.
- Selection sort: Find the minimum element and swap it with the current element.
- Merge sort:

7. Algorithm Efficiencies

Big O-Types:

- O(1) Constant Time Complexity
- O(log n) Logarithmic Time Complexity
- O(n) Linear Time Complexity
- O(n log n) Linearithmic Time Complexity
- O(n^2) Quadratic Time Complexity
- O(n^k) Polynomial Time Complexity
- O(2^n) Exponential Time Complexity
- O(n!) Factorial Time Complexity

How they work, their uniqueness, etc.:

- O(1) - Constant Time Complexity:
  An algorithm is said to have constant time complexity if its execution time or memory usage remains constant, regardless of the input size. It's the most efficient complexity since the algorithm's performance doesn't change as the input grows.
- O(log n) - Logarithmic Time Complexity:
  Algorithms with logarithmic complexity have runtimes that increase logarithmically as the input size grows. This means that as the input size increases, the increase in execution time becomes slower. Binary search algorithms are a classic example of algorithms with logarithmic time complexity.
- O(n) - Linear Time Complexity:
  Algorithms with linear time complexity have runtimes that grow linearly with the size of the input. As the input size increases by a certain factor, the execution time also increases by a proportional factor. Simple array traversal is an example of linear time complexity.
- O(n log n) - Linearithmic Time Complexity:
  Algorithms with linearithmic complexity have runtimes that are a product of linear and logarithmic growth. This complexity often appears in efficient sorting algorithms like Merge Sort and Heap Sort.
- O(n^2) - Quadratic Time Complexity:
  Quadratic time complexity means that the runtime grows quadratically with the input size. Nested loops that iterate over the input are common causes of quadratic complexity. Examples include selection sort and insertion sort.
- O(n^k) - Polynomial Time Complexity:
  Polynomial time complexities, where k is a constant greater than 1, indicate that the runtime grows as a polynomial function of the input size. This category includes cubic time (O(n^3)), quartic time (O(n^4)), and so on.
- O(2^n) - Exponential Time Complexity:
  Algorithms with exponential complexity have runtimes that double with each additional element in the input. These algorithms quickly become impractical for larger inputs due to their exponential growth.
- O(n!) - Factorial Time Complexity:
  Algorithms with factorial complexity have runtimes that are proportional to the factorial of the input size. These algorithms are extremely inefficient and are generally not usable for any practical inputs.

8. Heaps

- Heaps are specialized trees used for priority queues and heap sort.

```
left.data < root.data < right.data
       x
      /   \
     <x   >x
```

Heap -> A complete tree in which every left-parent right-parent trio satisfies
the following:
root > left && root > right

Heap Sort
Max Heap
Heapify
Re Heapify
Remove + set aside the top(root)
Replace top with right most element in last row (thus preserving the heap structure)
Heapify

```
private void heapifyUp(int index)
     {
          int parentIndex = (index - 1) / 2;
          while (index > 0 && maxpq.get(index) >
maxpq.get(parentIndex))
          {
               Collections.swap(maxpq, index, parentIndex);
               index = parentIndex;
               parentIndex = (index - 1) / 2;
          }
     }
```

9. Graphs

- Graphs consist of vertices and edges.
- Graph representation: Adjacency matrix and adjacency list.

A graph is a collection of a set of vertices & edges.

Example: G = {V, E}
V = {V1, v2, v3, v4, v5}
E = {e1, e2, e3}

If an edge along v3 and v1 is a weight of 7. It will take 7 to reach between the nodes.

What is a Minimal Cost Spanning Tree (MCST)?

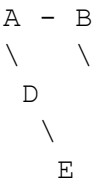
Tree that connects nodes from a graph with least amount of weight (weight of edges).

What is the process for generating MCST using Prim's Algorithm?

(Prim starts with vertices), pick the node to begin with, then starting
from said node and continue adding nodes while not forming a cycle.

..... Kruskal's algorithm?

(Kruskal starts with edges; lowest edge), build from there and do not form a cycle.



to	A	B	C	D	E	G
from A	0	1	1	1	0	0
B	1	0	1	0	0	1
C	1	1	0	0	0	1
D	1	0	0	0	1	0
E	0	0	0	0	0	0
G	0	1	1	0	0	0

Cost Adjacency Matrix (CAM)

Minimum Cost Spanning Tree (MCST)

1 -> 2 = 28
2 -> 7 = 14
2 -> 3 = 16
3 -> 4 = 12
4 -> 7 = 18
4 -> 5 = 22
5 -> 7 = 24
5 -> 6 = 26
6 -> 1 = 10
14 + 16 + 12 + 26 + 10 = 100

Robert Prim & Joseph Kruskal came up with two separate algorithms that work.

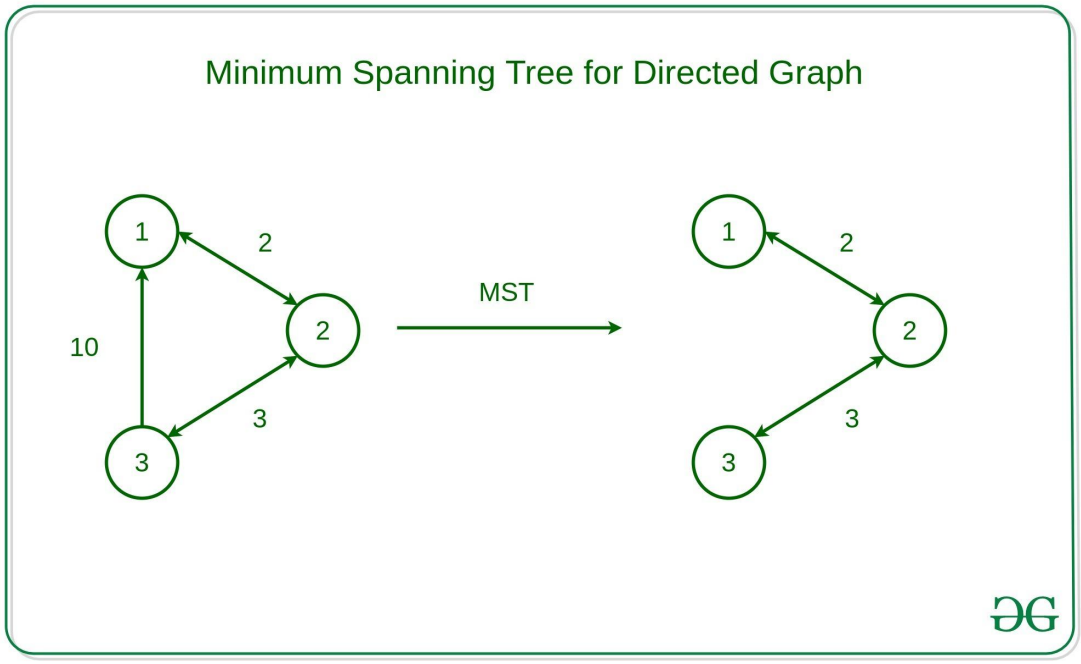
Prim - Start at the same node. Add lowest edges as long as no cycles are produced. Finished when all nodes are connected.

Kruskal - Start at Tree T node made up of all vertices and no edges.
Add the longest cost edge to T, continue adding lowest cost edges.
If a cycle is produced, add the next lowest cost.
Stop when n - 1 edges are added to n vertices.
height vs depth:
height - node count; n + 1
depth - edge count; n

Using Kruskal: 10 + 12 + 14 + 16 + 22 + 26 = 100
Using Prim: 14 + 16 + 12 + 22 + 26 + 10 = 100

10. Directed Graphs (Digraphs)

- Examine a Cost Adjacency Matrix to determine if it's a directed graph.



11. Deque (Double-ended Queue)

- Deque is a queue where elements can be inserted and removed from both ends.

```
import java.util.ArrayList;

// DOUBLY ENDED QUEUE
public class Deque<T>
{
	private ArrayList<T> dq;

	public Deque()
	{
		dq = new ArrayList<T>();
	}

	// push/pop/top/size/isEmpty/print-display

	public void pushFront(T item)
	{
		dq.add(0, item);
	}

	public void pushBack(T item)
	{
		dq.add(item);
	}

	public boolean popFront()
	{
		if(!dq.isEmpty())
		{
			dq.remove(0);
			return true;
		}
		return false;

	}

	public boolean popBack()
	{
```

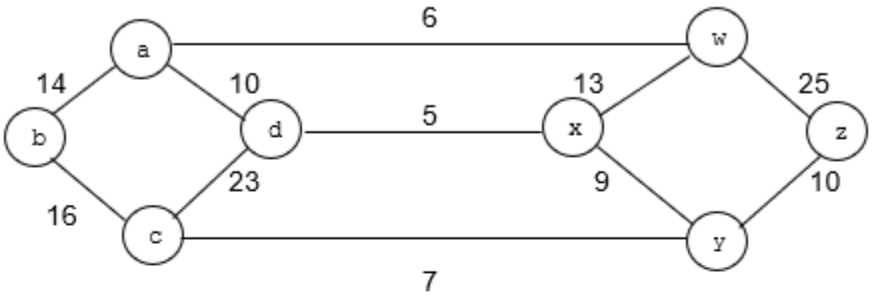
```
        if(!dq.isEmpty())
        {
            dq.remove(dq.size() - 1);
            return true;
        }
        return false;
    }

    public T topFront()
    {
        if(!dq.isEmpty())
        {
            return dq.get(0);
        }
        return null;
    }

    public T topBack()
    {
        if(!dq.isEmpty())
        {
            return dq.get(dq.size() - 1);
        }
        return null;
    }

    public T peekFront()
    {
        return topFront();
    }

    public T peekBack()
    {
        return topBack();
    }

    public int size()
    {
        return dq.size();
    }

    public boolean isEmpty()
    {
        return dq.isEmpty();
```

```
	}

	public void print()
	{
		System.out.print(toString());
	}

	@Override
	public String toString()
	{
		return "Front -> " + dq + " <- Back";
	}
}
```

12. Dijkstra's Algorithm

10. Prim's, Kruskal's, and Dijkstra's Algorithm

Given the following graph (the same graph from the previous problem, including the edge weights)



Use Prim's algorithm to generate a MCST. Start at vertex x.
x – d = 5
x – y = 9
y – c = 7
y – z = 10
d – a = 10
a – w = 6
c – b = 16
= 63
Use Kruskal's algorithm to generate a MCST.
d – x = 5
a – w = 6
c – y = 7
x – y = 9
y – z = 10
a – d = 10
a – b = 16
= 63

Use Dijkstra's Algorithm to find the shortest path from node *a* to the other nodes in the graph. A table is provided below (the first cell has been filled in):

V	a	b	c	d	w	x	y	z
a	0 a	14a	Inf.	10a	6a	Inf.	Inf.	Inf.
w		14a	Inf.	10a	6a	18w	Inf.	31w
d		14a	Inf.	10a		18w	Inf.	31w
b		14a	33a			18w	Inf.	31w
x			30b			18w	27x	31w
y			30b				27x	31w
c			30b					31w
								31w