# **Day 14 - 4 July 2025**

**Document Name:**Day 14 - hmuvvala@ - Hari Gopal Muvvala

### Task 1: Custom Node

* Created basic Node class with data and next
* Linked nodes manually → helped me visualize a simple linked list
* Useful to understand how .next connects elements

### Task 2: Insert at Beginning

* New node’s .next points to current head
* Then update head to the new node
* Prepends a node — like stack push from front

### Task 3: Insert at End

* Traverse to last node, then set .next = newNode
* Used while (current.next != null) logic
* Helps with appending in linked list

### Task 4: Insert at Specific Position

* Traverse till (pos-1), then change links
* Use temporary prev and next pointers for correct linking
* Used when inserting between nodes

### Task 5: Stack Push/Pop

* Stack is LIFO → Last In First Out
* .push() adds, .pop() removes last
* Useful for backtracking, undo operations

### Task 6: Stack Search

* .search(element) gives 1-based index from top
* Returns -1 if not found
* Rarely used, but good to know it exists

### Task 7: Stack Peek

* .peek() returns top element without removing it
* Used to check what’s next to come

### Task 8: Pop All and Check Empty

* Use loop to .pop() until .isEmpty() == true
* Confirms complete cleanup of stack
* Always check .isEmpty() before popping

### Task 9: Methods of Stack Class

#### Stack-specific methods (from java.util.Stack):

1. push(E item) – Adds an element to the top of the stack.
2. pop() – Removes and returns the top element.
3. peek() – Returns the top element without removing it.
4. search(Object o) – Returns the 1-based position of the element from the top; returns -1 if not found.
5. empty() – Returns true if the stack is empty.

### Task 10: Custom Queue

* Implemented own queue using linked list
* Tracked front, rear, and size
* enqueue at rear, dequeue from front
* Logic for isEmpty() → size == 0
* Logic for isFull() → size == maxSize

### Task 10A: Convert Stack & Deque to List using Streams

* stack.stream().collect(Collectors.toList())
* Java 8 streams used for clean transformation
* Can apply filters, maps before collecting

### Recursion

#### What is Recursion?

* A method calling itself to solve smaller instances of a problem.
* Always has:
  + Base Case → stopping condition
  + Recursive Case → breaks the problem into sub-problems

#### Why use Recursion?

* Simplifies code for problems with repetitive substructure
* Fits naturally for:
  + Factorial
  + Fibonacci
  + Tree/graph traversal
  + Backtracking (e.g., N-Queens, permutations)

#### Types of Recursion:

* Direct Recursion: Method calls itself directly
* Indirect Recursion: Method A calls B, B calls A
* Tail Recursion: Recursive call is the last thing in the function

#### Common Pitfalls:

* No base case → leads to infinite recursion
* Too deep recursion → can cause StackOverflowError
* Slower than iteration if not optimized (due to call stack usage)

#### Good To Remember:

* Use recursion when the problem is naturally recursive
* For performance, prefer iteration unless recursion simplifies logic