Basics

Linear Data Structures

Arrays

- It's a collection of elements of a particular data type
- Elements are adjacent to each other.
- Each partition has 2 neighbours, except 1st and last one.
- Size of the box is fixed, and cannot be modified.
- The positions are indexed, and we can find an element by it's index.
- Implementation: code
- Resize an array: There is no way to directly resize an array in Java. To do it, create another bigger array, and then copy all elements from the current array to the new array.

Sentinel Search

> Sentinel search is a variation of the linear search algorithm used to improve its efficiency by reducing the number of comparisons. It involves placing a sentinel value (a special value) at the end of the data structure to avoid having to check for out-of-bounds conditions during the search.

- **Steps**:
  - Before starting the search, add a sentinel value to the end of the list or array. This sentinel value should be greater than any other value in the list, so it will guarantee that the search will eventually find it if the item is not present.
  - Initialize the search index variable i to 0.
  - Set the last element of the array to the search key.
  - While the search key is not equal to the current element of the array (i.e., arr[i]), increment the search index i.
  - If i is less than the size of the array or arr[i] is equal to the search key, return the value of i (i.e., the index of the search key in the array).
  - Otherwise, the search key is not present in the array, so return -1 (or any other appropriate value to indicate that the key is not found).
- The key benefit of the Sentinel Linear Search algorithm is that it eliminates the need for a separate check for the end of the array, which can improve the average case performance of the algorithm.
- However, it does not improve the worst-case performance, which is still O(n) (where n is the size of the array), as we may need to scan the entire array to find the sentinel value.

Infix, Prefix & Postfix expressions

- The conversions are done using `stack` data structure.

Infix to Postfix:

- **Rules**:
    - Scan the infix expression from left to right.
    - If the scanned character is an operand, put it in the postfix expression.
    - Otherwise, do the following: If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack [or the stack is empty or the stack contains a '('], then push it in the stack. ['^' operator is right associative and other operators like '+','–','*' and '/' are left-associative].
        - Check especially for a condition when the operator at the top of the stack and the scanned operator both are '^'. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack.
        - In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.
    - Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.
        - After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
    - If the scanned character is a '(', push it to the stack.
    - If the scanned character is a ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
    - Repeat steps 2-5 until the infix expression is scanned.
    - Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
    - Finally, print the postfix expression.

- Example: `a*b+c/d` (Infix) to Postfix:

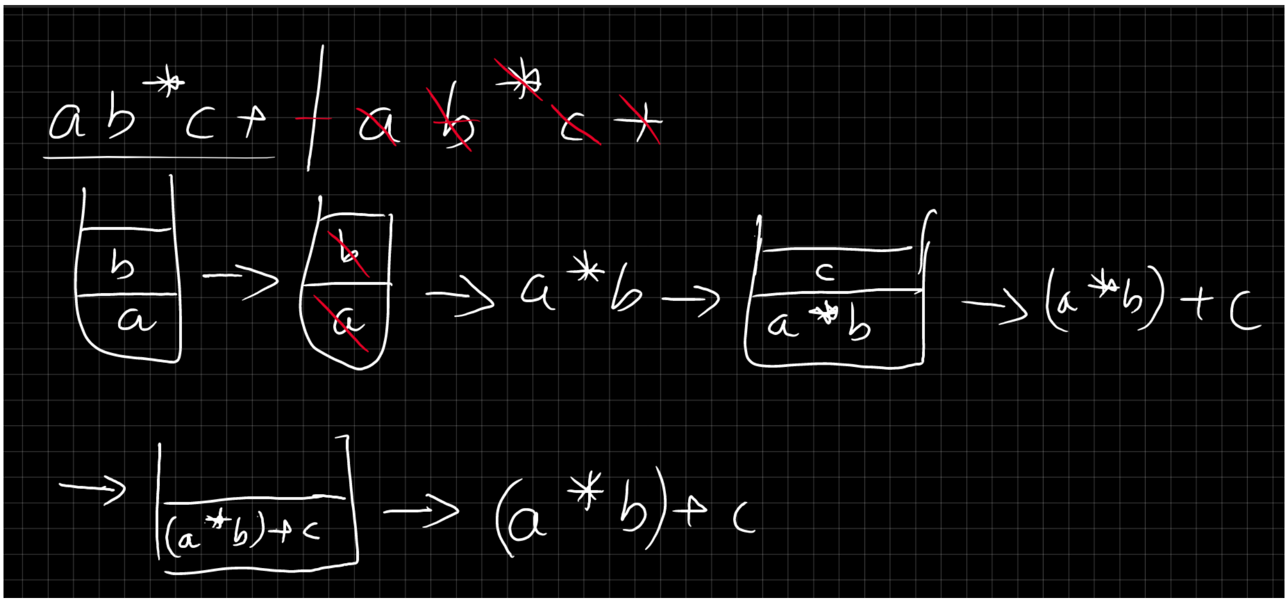
Symbols	Stack	Postfix Expression
a		a
*	*	a
b	*	ab
+	+	ab*
c	+	ab*c
/	+/	ab*c
d	+/	ab*cd
	+	ab*cd/
		ab*cd/+

- Example: `(a*b)+c/d` (Infix) to Postfix:

Symbols	Stack	Postfix Expression
(	(	
a	(	a
*	(*	a
b	(*	ab
)		ab*
+	+	ab*
c	+	ab*c
/	+/	ab*c
d	+/	ab*cd
	+	ab*cd/
		ab*cd/+

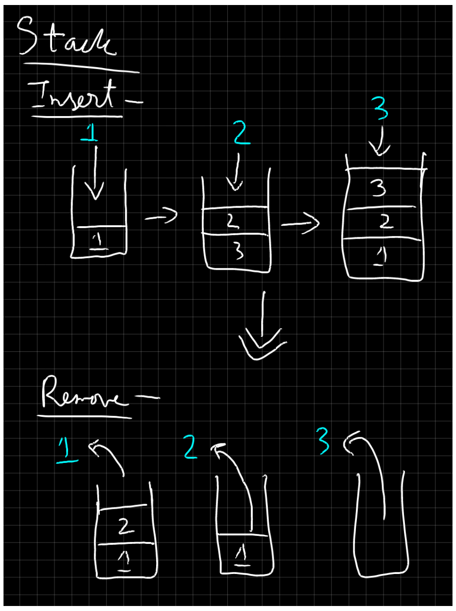
Postfix to Infix:

- Algorithm:
  - While there are input symbols left, read the next symbol from the input.
  - If the symbol is an operand, push it onto the stack.
  - If the symbol is an operator.
    - Pop the top 2 values from the stack (**write the values from bottom to top**).
    - Put the operator, with the values as arguments and form a string.
    - Push the resulted string back to stack.
  - If there is only one value in the stack, that value in the stack is the desired infix string.
- Example: `ab*c+` -> `(a*b)+c`:



Stack // Code

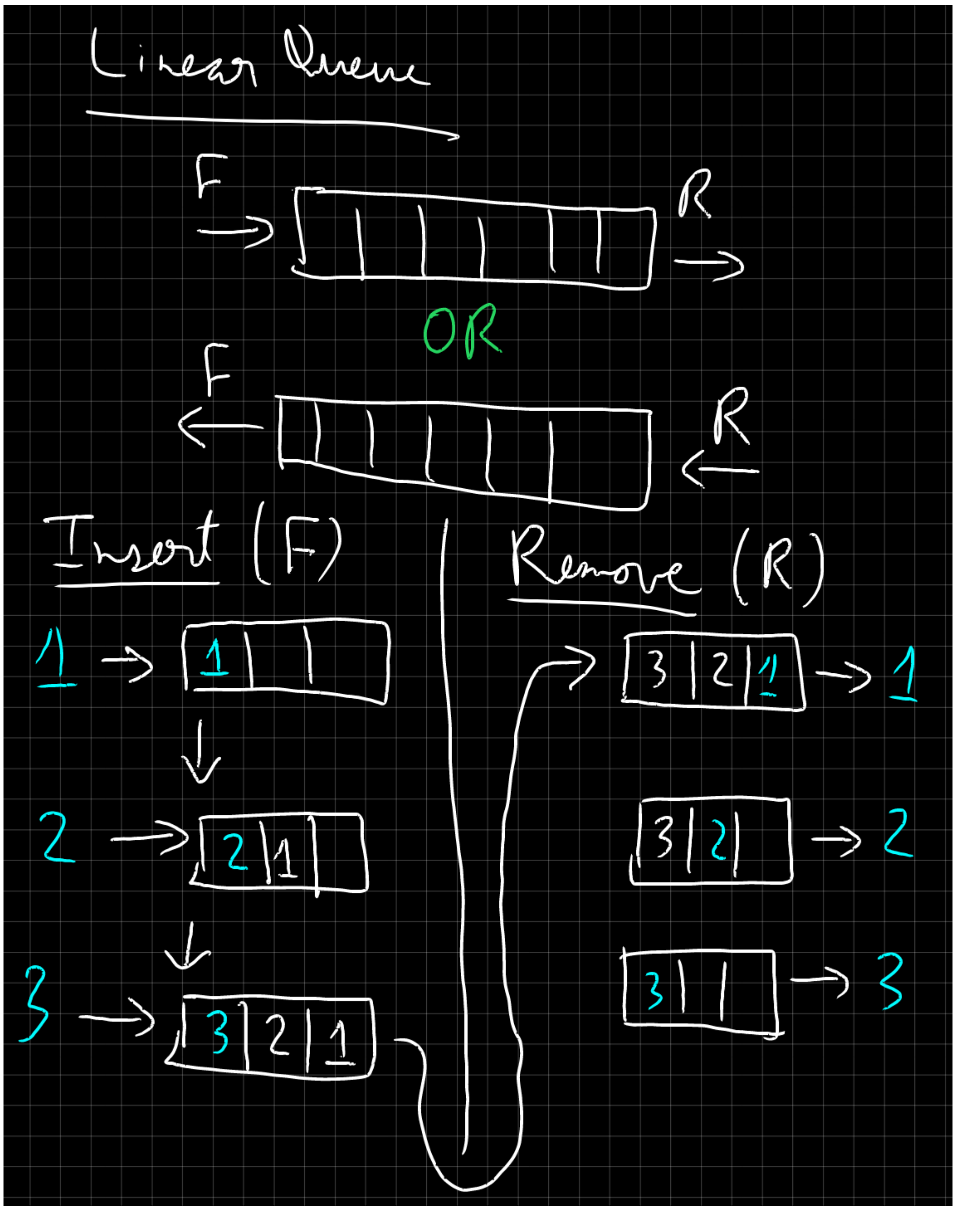
- A stack is a linear data structure that follows the **Last-In-First-Out (LIFO)** principle.
- It is an ordered collection of elements where the insertion and deletion of elements can only be performed at one end, called the top of the stack.
- The element that is inserted last is the first one to be removed.
- The operations performed on a stack are:
  - **Push()**: Adds an element to the top of the stack.
  - **Pop()**: Removes and returns the top element from the stack.
  - **Peek/Top()**: Returns the top element without removing it.
  - **isEmpty()**: Checks if the stack is empty.
  - **Size()**: Returns the number of elements in the stack.
- Stack can be implemented using arrays or linked lists.
- It is used in various applications such as expression evaluation, function call management, backtracking, and more.
- The time complexity of stack operations is O(1).
- How a stack works:



Queue // Code

- A queue is a linear data structure that follows the **First-In-First-Out (FIFO)** principle.
- It is an ordered collection of elements where the insertion of elements is done from the front, and the removal of elements is done from the rear. Subsequently, elements can be inserted from the rear and removed from the front too.
- The element that is inserted first is the first one to be removed.
- The operations performed on a queue are:
  - **Enqueue()**: Adds an element to the rear of the queue.
  - **Dequeue()**: Removes and returns the element from the front of the queue.
  - **Peek()/Front()**: Returns the element at the front without removing it.
  - **isEmpty()**: Checks if the queue is empty.
  - **Size**: Returns the number of elements in the queue.
- Queue can be implemented using arrays or linked lists.
- It is used in various applications such as job scheduling, breadth-first search, and more. The time complexity of queue operations is O(1).

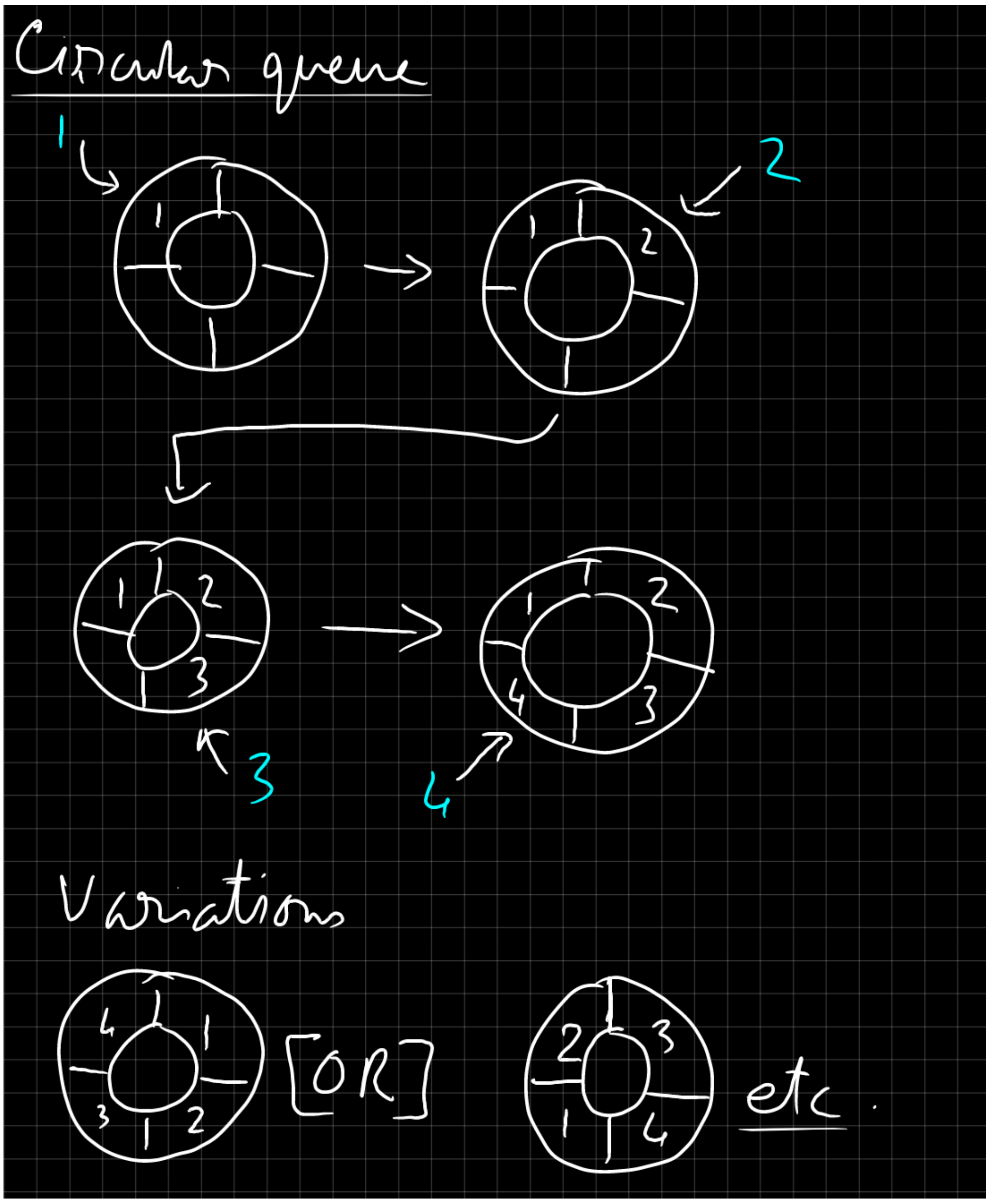
- How a Linear Queue works:



Circular Queue // Code

- A circular queue is a variation of the queue data structure where the last element points back to the first element, forming a circular structure.
- In a circular queue, the front and rear pointers wrap around to the beginning of the queue when they reach the end, allowing for efficient space utilization.
- Differences from a regular queue:
  - The rear pointer can move to the beginning of the queue when it reaches the end, creating a circular structure.
  - The front and rear pointers can be equal, indicating that the queue is full.
  - The queue can be implemented using an array or a linked list.
- Circular queues are commonly used in scenarios where elements are continuously added and removed, such as in scheduling algorithms or buffer management.
- The time complexity of enqueue and dequeue operations in a circular queue is $O(1)$.
- The first end can be inserted anywhere, but all subsequent elements are inserted in order after it.
- Consequently, front doesn't have to be index 0. So, when checking for isFull(), we need to check if *front is **just after** *rear, instead of checking if the index of *rear is length of queue - 1.

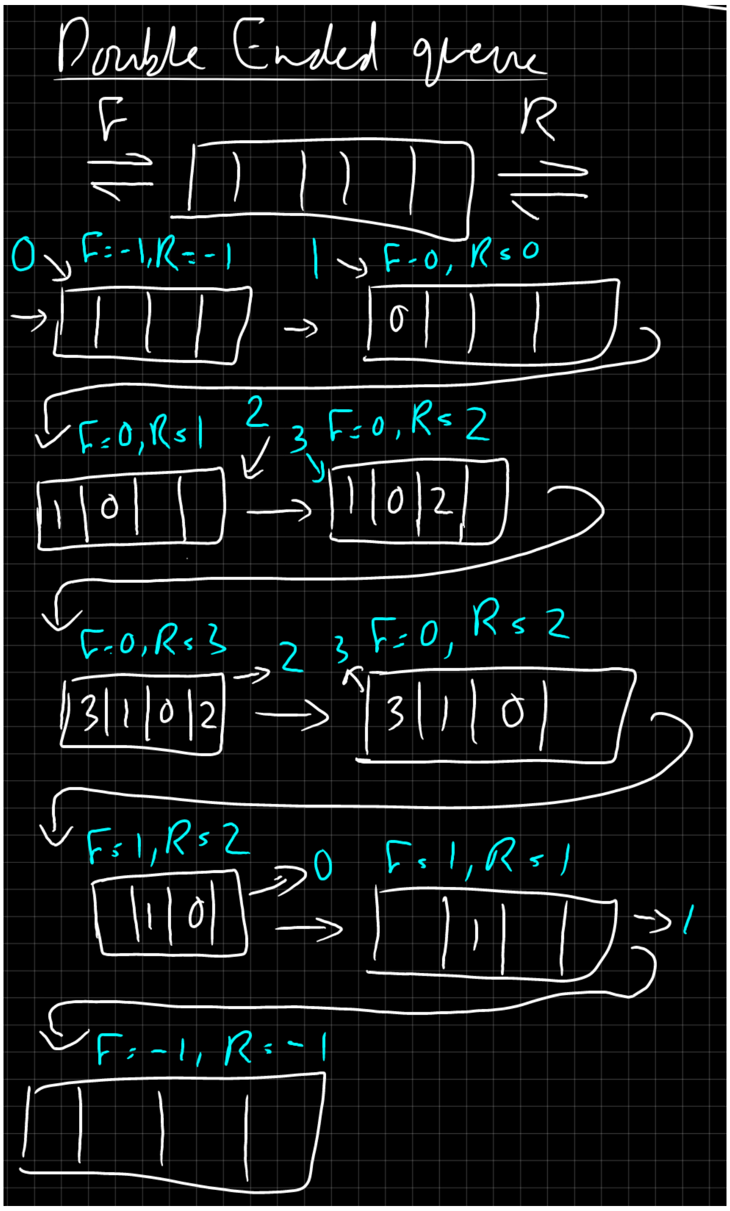
- How a Circular Queue works:



Double-Ended Queue

- A double-ended queue, also known as a deque, is a linear data structure that allows insertion and deletion of elements from both ends.
- In a double-ended queue, elements can be added or removed from either the front or the rear end, providing flexibility in implementing various algorithms.
- Operations supported by a double-ended queue:
  - **InsertFront()**: Adds an element to the front of the deque.
  - **InsertRear()**: Adds an element to the rear of the deque.
  - **DeleteFront()**: Removes and returns the element from the front of the deque.
  - **DeleteRear()**: Removes and returns the element from the rear of the deque.
  - **GetFront()**: Returns the element at the front of the deque without removing it.
  - **GetRear()**: Returns the element at the rear of the deque without removing it.
  - **isEmpty()**: Checks if the deque is empty.
  - **Size()**: Returns the number of elements in the deque.
- Double-ended queues can be implemented using arrays or linked lists.

- They are used in scenarios where efficient insertion and deletion operations are required at both ends, such as in implementing deque-based algorithms or data structures like priority queues.
- The time complexity of deque operations depends on the implementation, but commonly, it is O(1) for most operations.
- How a Double-Ended Queue works:



Linked List

- A linked list is a linear data structure consisting of a sequence of elements, called nodes, where each node contains a data element and a reference (or pointer) to the next node in the sequence.
- Unlike arrays, which have a fixed size and store elements in contiguous memory locations, linked lists dynamically allocate memory for each node, allowing for efficient insertion, deletion, and traversal of elements.
- Usage: Linked lists are often used when the size of the data structure is not known in advance or when frequent insertions and deletions are required, as they offer flexibility and efficient memory utilization. However, they may have slower access times for random access compared to arrays, as elements are not stored in contiguous memory locations.

Singly Linked List

- In this type of linked list, each node contains data and a pointer to the next node in the sequence. The last node points to `NULL` to indicate the end of the list.

Doubly linked list

- In a doubly linked list, each node contains data, a pointer to the next node, and a pointer to the previous node in the sequence. This allows for traversal in both forward and backward directions.

Circular linked list

- In a circular linked list, the last node points back to the first node, forming a circular structure. This can simplify certain operations like traversal or insertion at the end of the list.

Graphs

- Components of a graph:
  - Node / Vertex
  - Edge / Ordered pair: Connection between 2 nodes.
- Terms:
  - Adjacent node: Nodes connected to the current node.
  - Degree of a node:
    - Undirected Graph: Number of nodes we can reach from current node.
    - Directed Graph:
      - Indegree: Number of nodes through which we can reach current node.
      - Outdegree: Number of nodes we can reach from current node.
  - Path: A Path of length `n` from node `u` to `v` is a sequence of `n+1` nodes.
  - Isolated node: If the degree of a node is `0`.
- Types:
  - Directed: Direction of the edge is defined.
    - Directed Acryllic Graph: A directed graph with no cycle.
    - Tree: A DAG, where the additional condition is that a child can only have 1 parent.
  - Undirected: Direction of the edge is not defined.
    - Connected: There exists an edge between every pair of nodes. No node is unreachable.
    - Complete: Every node is connected to every other node. A particular node can be reached from all other nodes.
    - Bi-connected: A connected graph with no articulation points.
      - Articulation point: If a node gets removed, the graph gets disconnected.
        Graphs 0
  - Weighted: Edges have some costs associated with them.
  - Unweighted: Edges don't have costs associated with them.
  - Bipartite: Given 2 colours, a graph is a bipartite graph if no 2 adjacent nodes have the same colour.
    - Linear graphs are always bipartite by nature.
    - Graphs with even cycle length are always bipartite.
    - Graphs with odd cycle length are never bipartite.
      Graphs 1
  - Sparse: A graph with relatively few edges compared to the number of vertices.
  - Dense: A graph with relatively few vertices compared to the number of edges.

  - Finite: A graph with a finite number of vertices and edges.
  - Infinite: A graph with an infinite number of vertices and edges.