Algorithm To Insert An Element in a Simple Queue

Procedure QINSERT (F, R, Q, N, Y)

In this procedure, the F and R pointers to the front and rear elements of a queue, a queue Q consisting of N elements, and an element Y, this procedure inserts Y at rear of the queue. Prior to first invocation of the procedure, F and R have been set to zero.

Step 1: [Overflow ?]

If $R \geq N$

then Write('Overflow')

Return

Step 2: [Increment Rear Pointer]

$R \leftarrow R + 1$

Step 3: [Insert Element]

$Q[R] \leftarrow Y$

Step 4: [Is front pointer properly set]

If $F = 0$

then $F \leftarrow 1$

Return

Algorithm To Delete An Element in a Simple Queue

Function QDELETE (Q, F, R)

Given F and R, the pointers to the front and rear elements of a queue, respectively, and the queue Q to which they correspond, this function deletes and returns the last element of the queue. Y is a temporary variable.

Step 1: [Underflow ?]

If $F = 0$

then Write('Underflow')

Return(0) (Here 0 denotes that the Queue is empty)

Step 2: [Delete the element]

$Y \leftarrow Q[F]$

Step 3: [Queue Empty?]

If $F = R$

then $F \leftarrow R \leftarrow 0$

else $F \leftarrow F + 1$

Step 4: [Return Element]

Return(Y)

SIMPLE QUEUE

```
import java.util.*;
class Queue {
    int F, R, N;
    int Q[];
    Queue(int N) {
        Q = new int[N];
        this.N = N;
        F = -1;
        R = -1;
    }
    public void enQ(int Y) {
        if (R >= N - 1) {
            System.out.println("Queue Overflow");
        } else {
            R++;
            Q[R] = Y;
             // If element inserted is the first one, reset F
            if (F == -1)
                F = 0;
        }
    }
    public int deQ() {
        if (F == -1 && R == -1) {
            System.out.println("Queue Underflow");
            return 0;
        } else {
            int Y = Q[F];
            if (F == R) // If only one element left in queue, reset F and R
                F = R = -1;
            else
                F++;
            return Y;
        }
    }
    public void disp() {
        if (F == -1 && R == -1) {
            System.out.println("Queue Empty");
        } else {
            System.out.print("Queue: ");
```

```java
            for (int i = F; i <= R; i++) {
                System.out.print(Q[i] + "|");
            }
            System.out.println();
        }
    }
    public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter size of Queue: ");
    int N = sc.nextInt();
    Queue q = new Queue(N);
    int choice;
    do {
        System.out.println("\nMenu:");
        System.out.println("1. Enqueue");
        System.out.println("2. Dequeue");
        System.out.println("3. Display");
        System.out.println("4. Exit");
        System.out.print("Enter your choice: ");
        choice = sc.nextInt();
        switch (choice) {
            case 1:
                System.out.print("Enter element to enqueue: ");
                int val = sc.nextInt();
                q.enQ(val);
                break;
            case 2:
                q.deQ();
                break;
            case 3:
                q.disp();
                break;
            case 4:
                System.out.println("Exiting...");
                break;
            default:
                System.out.println("Invalid choice! Please enter 1-4.");
        }
    } while (choice != 4);
   }
}
```

Algorithm To Insert An Element In A Circular Queue

Procedure CQINSERT (F, R, Q, N, Y)

Given pointers to the front and rear of a circular queue, F and R, a vector Q consisting of N elements, and an element Y, this procedure inserts Y at the rear of the queue, Initially, F and R are set to Zero.

Step 1: [Check for the Overflow Condition]
If (F=R+1) OR (F=1 AND R=N)
then Write(Queue is Overflow)
Return

Step 2: [Is front Pointer Properly Set?]
If F = 0
then F ←1

Step 3: [Reset Rear Pointer]
If R = N
then R ← 1
else R ← R +1

Step 4: [Insert Element]
Q[R] ← Y

Return

Algorithm To Delete An Element In A Circular Queue

Procedure CQDELETE (F, R, Q, N)

Step 1: [Underflow ?]
If F = 0
then Write('Underflow')
Return(0) (Here 0 denotes that the Queue is empty)

Step 2: [Delete the element]
Y← Q[F]

Step 3: [Queue Empty?]
If F = R
then F ← R ← 0
Return(Y)

Step 4: [Increment Front Pointer]
If F = N
then F ← 1
else F ← F + 1
Return(Y)

CIRCULAR QUEUE

```
import java.util.*;
class CircularQueue {
    int F, R, N;
    int[] Q;
    CircularQueue(int N) {
        this.N = N;
        Q = new int[N];
        F = R = -1;
    }
    void enqueue(int Y) {
        // only (R+1)%N == F is also enough to check overflow
        if ((F == 0 && R == N - 1) || ((R + 1) == F)) {
            System.out.println("Queue Overflow");
            return;
        }
        if (F == -1) {
            F = R = 0;
        } else {
            R = (R + 1) % N;
        }
        Q[R] = Y;
    }
    int dequeue() {
        if (F == -1) {
            System.out.println("Queue Underflow");
            return -1;
        }
        int Y = Q[F];
        if (F == R) {
            F = R = -1;
        } else {
            F = (F + 1) % N;
        }
        return Y;
    }
    void display() {
        int i=0;
        if (F == -1) {
            System.out.println("Queue is empty");
            return;
```

```java
        }
        System.out.print("Queue elements: ");
        for (i = F; i != R; i = (i + 1) % N) {
            System.out.print(Q[i] + " ");
        }
       System.out.print(Q[i] + " ");
    }
}
class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        CircularQueue cq = new CircularQueue(5);
        int choice;
        do {
            System.out.println("\nMenu:");
            System.out.println("1. Enqueue");
            System.out.println("2. Dequeue");
            System.out.println("3. Display");
            System.out.println("4. Exit");
            System.out.print("Enter your choice: ");
            choice = sc.nextInt();
            switch (choice) {
                case 1:
                    System.out.print("Enter element to enqueue (Y): ");
                    int Y = sc.nextInt();
                    cq.enqueue(Y);
                    break;
                case 2:
                    cq.dequeue();
                    break;
                case 3:
                    cq.display();
                    break;
                case 4:
                    System.out.println("Exiting...");
                    break;
                default:
                    System.out.println("Invalid choice! Enter 1-4.");
            }
        } while (choice != 4);
    }
}
```

CIRCULAR DEQUEUE [ONLY PROGRAMS]

CIRCULAR DOUBLE ENDED QUEUE

```
class CircularQueue {
    int Q[], F, R, N;
    CircularQueue(int size) {
        N = size;
        Q = new int[N];
        F = R = -1;
    }
    // Insert at rear
    void InsertRear(int Y) {
        if ((R + 1) % N == F) {
            System.out.println("Queue Overflow");
            return;
        }
        if (F == -1) { // Queue empty
            F = R = 0;
        } else {
            R = (R + 1) % N;
        }
        Q[R] = Y;
    }
    // Insert at front
    void InsertFront(int Y) {
        if ((R + 1) % N == F) {
            System.out.println("Queue Overflow");
            return;
        }
        if (F == -1) { // Queue empty
            F = R = 0;
        } else {
            F = (F - 1 + N) % N;
        }
        Q[F] = Y;
    }
    // Delete from front and return value
    int DeleteFront() {
        if (F == -1) {
            System.out.println("Queue Underflow");
            return -1;
```

```
        }
        int Y = Q[F];
        if (F == R) { // Only one element
            F = R = -1;
        } else {
            F = (F + 1) % N;
        }
        return Y;
    }
    // Delete from rear and return value
    int DeleteRear() {
        if (F == -1) {
            System.out.println("Queue Underflow");
            return -1;
        }
        int Y = Q[R];
        if (F == R) { // Only one element
            F = R = -1;
        } else {
            R = (R - 1 + N) % N;
        }
        return Y;
    }
      // Display the queue contents using for loop
      void display() {
        if (F == -1) {
        System.out.println("Queue is Empty");
        return;
       }
       System.out.print("Queue elements: ");
       for (int i = F; i != R; i = (i + 1) % N) {
        System.out.print(Q[i] + " ");
       }
       System.out.println(Q[R]); // Print the last element (R)
      }
}
class Main {
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        CdeQueue cq = new CdeQueue(5);
        int choice = 0;
```

```java
        do {
            System.out.println("\nMenu:");
            System.out.println("1: Insert Front");
            System.out.println("2: Insert R");
            System.out.println("3: Delete F");
            System.out.println("4: Delete R");
            System.out.println("5: Display");
            System.out.println("6: Exit");
            System.out.print("Enter your choice: ");
            choice = sc.nextInt();
            switch (choice) {
                case 1:
                    System.out.print("Enter number to insert at F: ");
                    int numF = sc.nextInt();
                    cq.insertFront(numF);
                    break;
                case 2:
                    System.out.print("Enter number to insert at R: ");
                    int numR = sc.nextInt();
                    cq.insertR(numR);
                    break;
                case 3:
                    cq.deleteF();
                    break;
                case 4:
                    cq.deleteR();
                    break;
                case 5:
                    cq.display();
                    break;
                case 6:
                    System.out.println("Exiting...");
                    break;
                default:
                    System.out.println(" Please enter a number from 1 to 6.");
            }
        } while (choice != 6);
    }
}
```

Array	Linked List
Memory Allocation Static (fixed size at compile time)	**Memory Allocation Dynamic** (memory allocated for each node at runtime)
- Can be dynamically allocated (new/malloc) but size fixed after allocation	- Size can grow/shrink dynamically as needed
Insertions Efficient at the end (O(1)) if space available	**Insertions Efficient at front** (O(1)) if pointer to node available
- **Inefficient in middle or front** due to shifting (O(n))	- **Inefficient in middle** (O(n)) if pointer not available
Deletions Efficient at the end (O(1))	**Deletions Efficient at front** (O(1)) if pointer to node available
- **Inefficient in middle or front** due to shifting (O(n))	- **Inefficient in middle** (O(n)) if pointer not available
Direct access using index (O(1))	**No direct access** (must traverse nodes, O(n))
Linear search (O(n)) or **Binary search** (O(log n)) if sorted	**Linear search** (O(n))
Shifting Elements Required for **insertions** and **deletions** in the middle/beginning (O(n))	**Shifting Elements** No shifting, only pointer manipulation

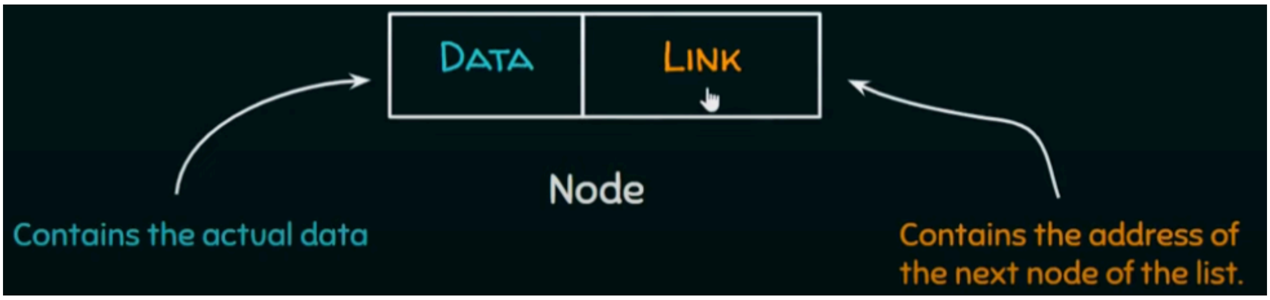
Types of Linked List

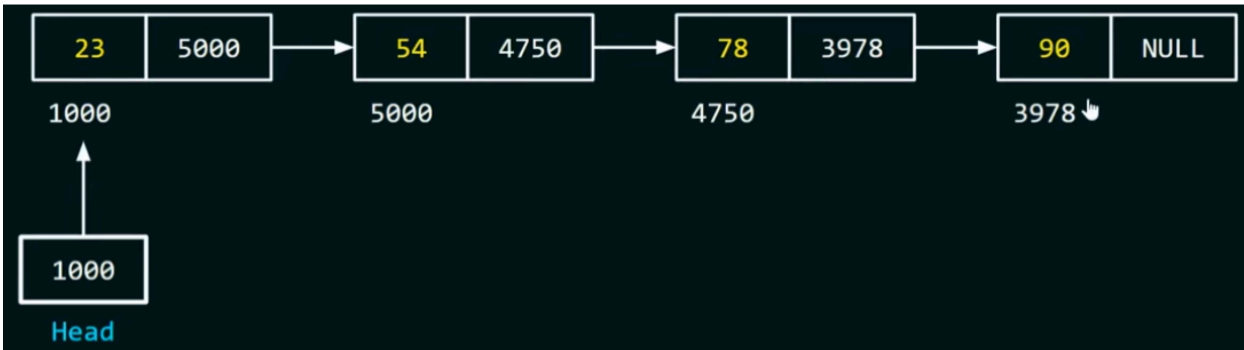
- **Single Linked List:** Navigation is forward only.
- **Doubly Linked List:** Forward and backward navigation is possible.
- **Circular Linked List:** Last element is linked to the first element.

Single Linked List

A single linked list is a list made up of nodes that consists of two pa

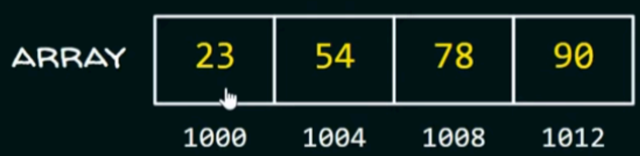
- Data





Nodes are scattered here and there in the memory but they are still connected with each other.

We want to store a list of numbers: 23, 54, 78, 90



In an array, elements are stored in consecutive memory locations.

// Program for singly linked list

```
class SLL {
    class Node {
        int data;
        Node next;
        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }
    private Node head;
    // 1. Insert at the Beginning (Iterative)
    void insertBeg(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        newNode.next = head;
        head = newNode;
    }
    // 2. Insert at the End (Iterative)
    void insertEnd(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node temp = head;
        while (temp.next != null) {
            temp = temp.next;
        }
        temp.next = newNode;
    }
    // 3. Delete from the Beginning (Iterative)
    void delBeg() {
        if (head == null) {
            System.out.println("List is empty");
            Return; }
        System.out.println("Deleted from Beginning: " + head.data);
        head = head.next;
```

```java
    }
    // 4. Delete from the End (Iterative)
    void delEnd() {
        if (head == null) {
            System.out.println("List is empty! Cannot delete from end.");
            return;
        }
        if (head.next == null) { // Only one node
            System.out.println("Deleted from End: " + head.data);
            head = null;
            return;
        }
        Node temp = head;
        while (temp.next.next != null) {
            temp = temp.next;
        }
        System.out.println("Deleted from End: " + temp.next.data);
        temp.next = null;
    }
    // 5. Display the Linked List (Iterative)
    void displayIterative() {
        if (head == null) {
            System.out.println("List is empty!");
            return;
        }
        Node temp = head;
        System.out.print("Linked List: ");
        while (temp != null) {
            System.out.print(temp.data + " -> ");
            temp = temp.next;
        }
        System.out.println("NULL");
    }

    public static void main(String[] args) {
        SLL sll = new SLL();
        sll.insertBeg(10);
        sll.insertBeg(20);
        sll.insertEnd(30);
        sll.insertEnd(40);
        sll.displayIterative();
    }
}
```

```
// Program for stack using singly linked list
class StackUsingLinkedList {
    static class Node {
        int data;
        Node next;
        Node(int data) {
            this.data = data;
            next = null;
        }
    }
    Node top; // top of the stack
    // Insert First
    void push(int val) {
        Node newNode = new Node(val);
        newNode.next = top;
        top = newNode;
    }
    // Delete First
    int pop() {
        if (top == null) {
            System.out.println("Stack Underflow");
            return -1;
        }
        int popped = top.data;
        top = top.next;
        return popped;
    }
    // Display stack from top to bottom
    void display() {
        if (top == null) {
            System.out.println("Stack is Empty");
            return;
        }
        Node temp = top;
      while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
```

```
        System.out.println();
    }
    // Main for demo
    public static void main(String[] args) {
        StackUsingLinkedList stack = new StackUsingLinkedList();
        stack.push(10);
        stack.push(20);
        stack.push(30);
        stack.display(); // 30 20 10
        System.out.println("Popped: " + stack.pop()); // 30
        stack.display(); // 20 10
    }
}
```

// Program for queue using singly linked list

```
class QueueUsingLinkedList
{
    static class Node {
        int data;
        Node next;
        Node(int data) {
            this.data = data;
            next = null;
        }
    }
    Node front, rear;
    void enqueue(int val) {
        Node newNode = new Node(val);
        if (rear == null) {
            front = rear = newNode;
        } else {
            rear.next = newNode;
            rear = newNode;
        }
    }
```

```java
    int dequeue() {
        if (front == null) {
            System.out.println("Queue Underflow");
            return -1;
        }
        int val = front.data;
        front = front.next;
        if (front == null) rear = null; // Queue empty
        return val;
    }
    void display() {
        if (front == null) {
            System.out.println("Queue is Empty");
            return;
        }
        Node temp = front;
        System.out.print("Queue: ");
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
    public static void main(String[] args) {
        QueueUsingLinkedList queue = new QueueUsingLinkedList();
        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);
        queue.display(); // 10 20 30
        System.out.println("Dequeued: " + queue.dequeue()); // 10
        queue.display(); // 20 30
    }
}
```

PRACTICE FUNCTIONS

```
void insertBeforeX(int data, int value) {
    Node newNode = new Node(data);
    // Case 1: List is empty
    if (head == null) {
        System.out.println("List is empty. Cannot insert before " + value);
        return;
    }
    // Case 2: Insert before head
    if (head.data == value) {
        newNode.next = head;
        head = newNode;
        return;
    }
    // Case 3: Traverse to find node before the target value
    Node temp = head;
    //until next node exist AND you cant find the target value
    while (temp.next != null && temp.next.data != value) {
        temp = temp.next;
    }
   // you reached last node and couldn't find  target value
    if (temp.next == null) {
        System.out.println("Value " + value + " not found in the list.");
        return;
    }
    // Insert new node before the matched node
    newNode.next = temp.next;
    temp.next = newNode;
}
```

```
void insertAfterX(int data, int value) {
    Node newNode = new Node(data);
    if (head == null) {
        System.out.println("List is empty. Cannot insert after " + value);
        return;
    }
    Node temp = head;
    while (temp != null && temp.data != value) {
        temp = temp.next;
    }
    if (temp == null) {
        System.out.println("Value " + value + " not found.");
        Return; }
    newNode.next = temp.next;
```

```
    temp.next = newNode;
}
```

```
void deleteNodeY(int value) {
    if (head == null) return; // Empty list
    // Case: delete head node
    if (head.data == value) {
        Node temp = head;
        head = head.next;
        temp.next = null;
        return;
    } //Traverse Now
    Node prev = null, curr = head;
    while (curr != null && curr.data != value) {
        prev = curr;
        curr = curr.next;
    }
    // If node is found, delete it
    if (curr != null) {
        prev.next = curr.next;
        curr.next = null;
    } else {
        System.out.println("Value " + value + " not found in the
list.");
    }
}
```

```
 // Check if all elements are unique
    public boolean areAllElementsUnique() {
        for (Node current = head; current != null; current = current.next) {
            for (Node checker = current.next; checker != null; checker = checker.next) {
                if (current.data == checker.data) {
                    return false; // duplicate found
                }
            }
        }
        return true; // all unique
    }
```

```java
// Insert into sorted position (ascending order)
    public void insertInOrder(int value) {
        Node newNode = new Node(value);

        // Case 1: Empty list or new node should be first
        if (head == null || value < head.data) {
            newNode.next = head;
            head = newNode;
            return;
        }
        // Use temp for traversal
        Node temp = head;
        while (temp.next != null && temp.next.data < value) {
            temp = temp.next;
        }

        // Insert after temp
        newNode.next = temp.next;
        temp.next = newNode;
    }
```

//Write a JAVA function that copies one linked list to another linked list.

```java
public static Node copyList(Node head1) {
    if (head1 == null) return null;

    Node head2 = new Node(head1.data); // first node copied
    Node temp1 = head1.next;           // pointer in original
list
    Node temp2 = head2;                // pointer in new list

    while (temp1 != null) {
        temp2.next = new Node(temp1.data); // copy node
        temp2 = temp2.next;
        temp1 = temp1.next;
    }
```

```
    return head2;
}
```

▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬

```
// Write a java function named "displayfromend()"
 void displayFromEnd(Node temp) {
        if (temp == null) return;
        displayFromEnd(temp.next);
        System.out.print(temp.data + " ");
    }
    // Wrapper function for recursion
    void displayFromEnd() {
        displayFromEnd(head);
        System.out.println();
    }
```

▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬

```
//Write a program to search an element in a linked list.
boolean searchRecursive(Node temp, int key) {
    if (temp == null)
        return false;
    if (temp.data == key)
        return true;
    return searchRecursive(temp.next, key);
}
```

▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬

```
// DELETE A NODE WITH VALUE Y
void deleteNodeY(int value) {
    if (head == null) return;
    if (head.data == value) {
        head = head.next;
        return;
    }
    Node prev = null, curr = head;
    while (curr != null && curr.data != value) {
        prev = curr;
        curr = curr.next;
    }
    if (curr != null) {
```

```
        prev.next = curr.next;
    }
```

```
// Find Middle element in LL
 int findMiddle(Node head) {
     Node fast = head;
     Node slow = head;

     while (fast != null && fast.next != null) {

         fast = fast.next.next; //2 jump
         slow = slow.next; // 1 jump
     }
     return slow.data; // Middle node's data
 }
```

```
void deleteAfterX(int value) {
    if (head == null || head.next == null ) {
        System.out.println("List is empty OR has only 1 node.");
        return;
    }
    Node temp = head;
    while (temp != null && temp.data != value) {
        temp = temp.next;
    }
    if (temp == null || temp.next == null) {
        System.out.println("No node exists after " + value + " to delete.");
        return;
    }
    Node toDelete= temp.next; // node to be deleted
    temp.next = toDelete.next;
    toDelete.next = null;
}
```