Dynamic Stack using Linked List

A **dynamic stack** is a type of stack data structure that **can grow or shrink in size during runtime**, depending on how much data it holds. This contrasts with a **static stack**, where the size is fixed at the time of creation.

Key Characteristics of a Dynamic Stack:

1. **Dynamic Memory Allocation**:
   - Memory for the stack is allocated **dynamically** (usually using pointers and heap memory).
   - It grows when more elements are pushed and may shrink when elements are popped or when no longer needed.
2. **No Fixed Size**:
   - Unlike arrays used in static stacks, a dynamic stack doesn't have a pre-defined maximum size.
   - Theoretically, it can grow as long as memory is available.
3. **Implemented Using**:
   - **Linked list** (most common way)
   - **Dynamic arrays** (e.g., vectors in C++, ArrayList in Java, or Python lists)

Advantages of Dynamic Stack:

- **Efficient memory usage** – uses only as much memory as needed.
- **No overflow** unless system memory is exhausted.
- Useful in **recursive algorithms**, **expression evaluation**, and **backtracking** problems.

Disadvantages:

- Slightly **slower** than static stacks due to dynamic memory management.
- May involve **more complex code** and potential issues like memory leaks if not handled properly.

Program code on Dynamic Stack using Linked List using Java

```
// Node class to represent each element in the stack
class Node
{
   int data;
   Node next;
   Node(int data)
   {
      this.data = data;
      this.next = null;
   }
}

// Stack class
class DynamicStack
{
   private Node top;

   // Constructor
   public DynamicStack()
   {
      top = null;
   }

   // Push operation
   public void push(int value)
   {
      Node newNode = new Node(value);
      newNode.next = top;
      top = newNode;
      System.out.println(value + " pushed to stack.");
   }

   // Pop operation
   public int pop()
   {
      if (isEmpty()) {
         System.out.println("Stack Underflow - Cannot pop.");
         return -1;
      }
      int popped = top.data;
      top = top.next;
      return popped;
   }
```

```
    // Peek operation
    public int peek()
    {
        if (isEmpty()) {
            System.out.println("Stack is empty - Cannot peek.");
            return -1;
        }
        return top.data;
    }

    // Check if stack is empty
    public boolean isEmpty()
    {
        return top == null;
    }

    // Display the stack
    public void display()
    {
        if (isEmpty()) {
            System.out.println("Stack is empty.");
            return;
        }
        Node temp = top;
        System.out.print("Stack elements: ");
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
}

// Demo class to test the dynamic stack
public class DynamicStackExample
{
    public static void main(String[] args)
    {
        DynamicStack stack = new DynamicStack();
        stack.push(10);
        stack.push(20);
        stack.push(30);
        stack.display();
        System.out.println("Top element is: " + stack.peek());
        System.out.println("Popped element: " + stack.pop());
        System.out.println("Popped element: " + stack.pop());
        stack.display();
    }
}
```

Dynamic Queue using Linked List

A **dynamic queue** is a type of queue data structure whose size **can grow or shrink at runtime**, based on the number of elements in it. This contrasts with a **static queue**, which has a fixed maximum size (usually implemented with an array).

Key Features of a Dynamic Queue:

1. **Dynamic Memory Allocation**:
    - Memory is allocated as needed, often using a **linked list** or a **dynamic array** (ArrayList in Java, list in Python).
2. **No Predefined Size Limit**:
    - The queue can keep growing as long as there's available memory.
3. **Efficient for Variable Workloads**:
    - Ideal for applications where the number of items isn't known in advance or keeps changing (e.g. print queues, job scheduling, message handling).

Advantages of Dynamic Queue:

- No overflow unless system memory is full.
- Efficient use of memory.
- Ideal for applications with unpredictable data flow.

Disadvantages:

- Slightly more complex than static queues.
- Care is needed to manage memory and references properly.

Program code on Dynamic Queue using Linked List using Java

```
// Node class representing each element in the queue
class Node
{
   int data;
   Node next;
   Node(int data)
   {
      this.data = data;
      this.next = null;
   }
}

// Dynamic Queue class using Linked List
class DynamicQueue
{
   private Node front, rear;
   // Constructor
   public DynamicQueue()
   {
      front = rear = null;
   }

   // Enqueue operation
   public void enqueue(int value)
   {
      Node newNode = new Node(value);
      if (rear == null) { // empty queue
         front = rear = newNode;
      } else {
         rear.next = newNode;
         rear = newNode;
      }
      System.out.println(value + " enqueued to queue.");
   }

   // Dequeue operation
   public int dequeue() {
      if (isEmpty()) {
         System.out.println("Queue Underflow - Cannot dequeue.");
         return -1;
      }
      int value = front.data;
      front = front.next;
      if (front == null) // queue became empty
         rear = null;
      return value;
   }
```

```java
    // Peek operation
    public int peek() {
        if (isEmpty()) {
            System.out.println("Queue is empty.");
            return -1;
        }
        return front.data;
    }

    // Check if queue is empty
    public boolean isEmpty()
   {
        return front == null;
    }

    // Display queue elements
    public void display()
    {
        if (isEmpty()) {
            System.out.println("Queue is empty.");
            return;
        }
        Node temp = front;
        System.out.print("Queue elements: ");
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
}

// Demo class to test the dynamic queue
public class DynamicQueueExample
{
    public static void main(String[] args)
    {
        DynamicQueue queue = new DynamicQueue();
        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);
        queue.display();
        System.out.println("Front element is: " + queue.peek());
        System.out.println("Dequeued: " + queue.dequeue());
        System.out.println("Dequeued: " + queue.dequeue());
        queue.display();
    }
}
```