Topics to cover:

1. Stack
2. Queue
3. LinkedList

1. Stack

1.1 What is a Stack?

A **stack** is a simple yet powerful data structure that is a collection of elements that follows the **Last-In-First-Out (LIFO) principle**, meaning that the last element added to the stack is the first element that gets removed from it.

In Java, the stack data structure is implemented as a class, which provides various methods to manage its elements.

Think of a stack of plates at a restaurant. When you add a new plate, you put it on top of the stack. When you take a plate away, you take it from the top of the stack.

In computer science, we use stacks to store information that we might need to use later on. For example, when you use the “undo” function on a computer program, the program is able to take you back to your previous action by “popping” the most recent action off of the stack.

Another way to think about a stack is like a pile of books. When you add a new book, you put it on top of the pile. When you want to read a book, you take the book from the top of the pile.

push() method

It adds an element to the top of the stack. When an element is pushed onto the stack, it becomes the new top element, and all other elements are pushed down one position.

pop() method

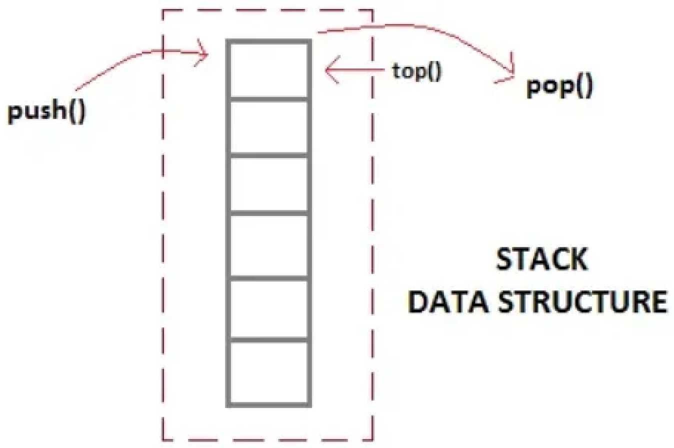
It removes the last element that was added to the stack. When an element is popped from the stack, it is removed from the top of the stack and the size of the stack is decreased by one.

peek() method

It returns the last element that was added to the stack without removing it. It allows the caller to peek at the top element without modifying the stack.

isEmpty() method

It returns whether the stack is empty or not. A stack is considered empty if it has no elements, i.e., its size is zero. This method is often used to check whether there are any elements left in the stack before calling the pop() method, to avoid errors.



1.2 Creating a Stack in Java

There are various ways to implementing a stack that includes using an array, linked list, array list and collection framework which is the easiest of all. Let’s take a closer look at the stack data structure in Java and understand its functions.

A stack can be of any type, such as Integer, String, Character, or Float. The four primary operations in a stack are push, pop, top/peek, and empty.

While on the get go a stack and array may not look much differnet. But the biggest point which comes to my mind is that while an array size is fixed

In this implementation, we use an array to store the elements of the stack. We use two variables — top to keep track of the topmost element of the stack, and max_size to keep track of the maximum size of the stack.

The biggest difference between an array and a stack is that an array is a static data structure, while a stack is a dynamic data structure. Arrays have a fixed size, which means that the number of elements they can hold is predetermined at the time of their creation. On the other hand, stacks can grow or shrink in size as elements are added or removed.

Another difference is that accessing elements in an array is done using an index, which is a constant-time operation, while accessing elements in a stack requires popping elements off the stack, which is a linear-time operation. Therefore, stacks are generally slower than arrays when it comes to accessing elements.

```
public class Stack {

  int n;

  int arr[]; // Array to store the elements of the stack

  int top; // Index of the top element of the stack

  Stack(int n) { // Constructor to create an empty stack with a specified size(n)

    this.n = n;

    this.arr = new int[n];
    this.top = -1; // The stack is initially empty, so top is set to -1

}



public boolean isEmpty() {

    return (top == -1);

}



public boolean isFull() {

    return (n - 1 == top);

}



// Pushes an element onto the top of the stack

public void push(int pushedElement) {

    if (!isFull()) {

        top++;

        arr[top] = pushedElement;
        System.out.println("Pushed element:" + pushedElement);

    } else {

        System.out.println("Stack is full !"); // Overflow

    }

}



// Removes and returns the element at the top of the stack

public int pop() {

    if (!isEmpty()) {

        int poppedElement = top;

        top--;

        System.out.println("Popped element :" + arr[poppedElement]);

        return arr[poppedElement];



    } else {

        System.out.println("Stack is empty!");
        return -1;
    }
}


// Returns the element at the top of the stack without removing it
public int peek() {
    if (!this.isEmpty())
        return arr[top];
    else {
        System.out.println("Stack is Empty");
        return -1;
    }
}


public static void main(String[] args) {
    Stack st = new Stack(5); // Create a new stack with a size of 5
```

```
        st.pop();

        st.push(1);

        st.push(2);

        st.push(3);



        st.pop();

        st.pop();

        st.pop();

    }

}
```

Output:

```
Stack is empty!

Pushed element:1

Pushed element:2

Pushed element:3

Popped element :3
Popped element :2

Popped element :1
```

2. Queue

A **queue** is a linear data structure that consists of a collection of items that follow a **first-in-first-out** sequence. This implies that the first item to be inserted will be the first to be removed. You can also say that items are removed in the order they were inserted.

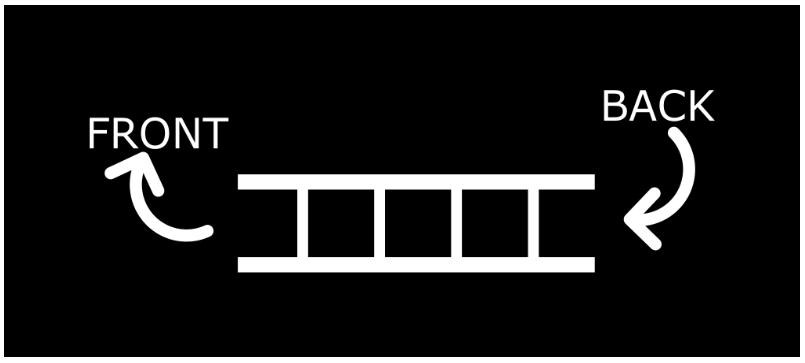
Using a real world example, we can compare a queue data structure to a queue of individuals standing in line for a service. Once one individual is attended to, they leave the queue for the next person to be attended to. They are helped in the order in which they came.

Structure of a Queue

A queue is mainly made up of two parts: the front/head and the rear/tail/back. For the purpose of clarity and consistency, we will stick to using front and back.

The **back** is where the items are inserted and the **front** is the part of the queue where items are removed/deleted.

Here is a diagram to help you understand better:



The image shows an array with various cells. The items are inserted through the **back** and removed through the **front**. There are terms used for the insertion and deletion of items in a queue which we will cover in the next section.

Note that you can reverse the structure of your queue – you can have the front on the right and the back on the left side. Whichever structure you go with, always remember that insertion of items happens through the back and deletion through the front.

Common Operations of a Queue

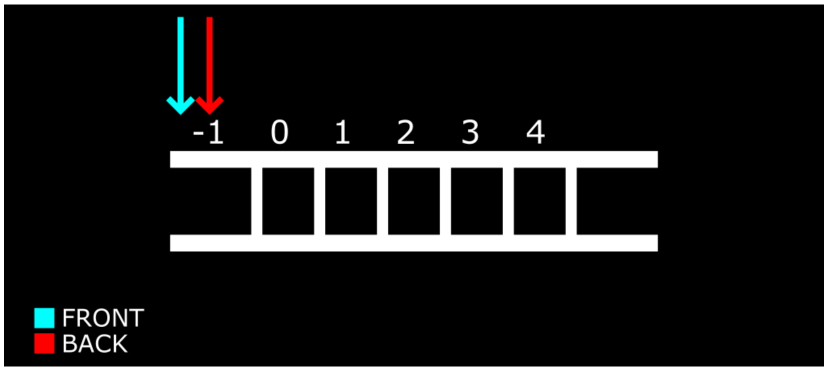
The following operations are commonly used in a queue:

- **Enqueue**: Adds an item from the back of the queue.
- **Dequeue**: Removes an item from the front of the queue.
- **Front**/**Peek**: Returns the value of the item in front of the queue without dequeuing (removing) the item.
- **IsEmpty**: Checks if the queue is empty.
- **IsFull**: Checks if the queue is full.
- **Display**: Prints all the items in the queue.

Before we see how to implement this with code, you need to understand how the **enqueue** and **dequeue** operations work and how they affect the front and back positions.

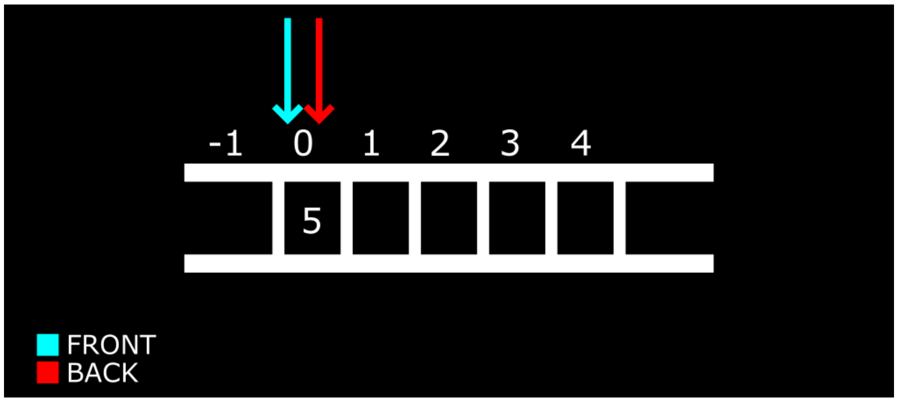
The indices of arrays in most programming languages start from 0. While implementing our code, we are going to set the index of the front and back values of our array to -1. This will enable us move the front and back position properly as values are added.

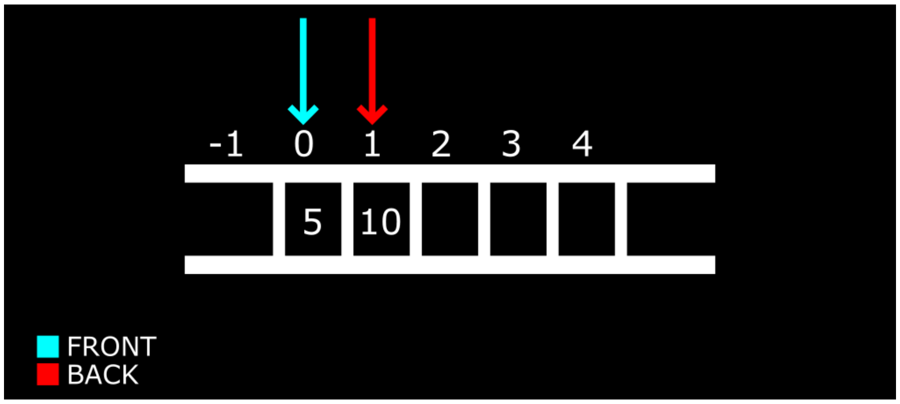
Consider the image below:



The arrows show the position of the front and back of our array. When both positions are at -1, it means the array is empty.

Let us add some items into our array and see what happens.

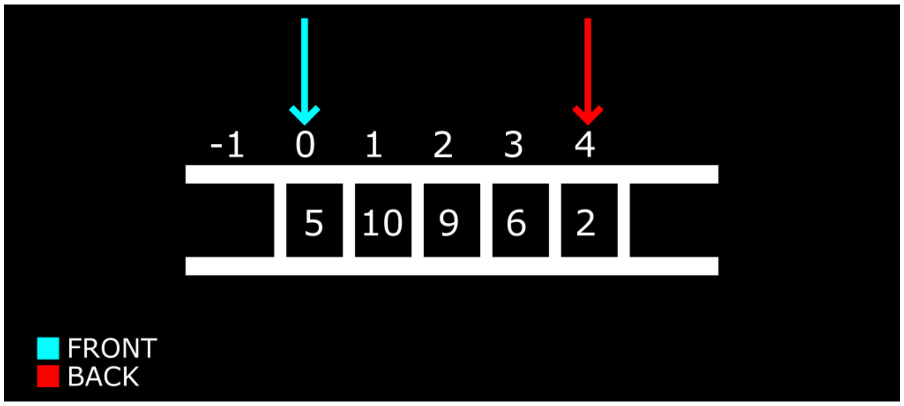




A second item has been added but only the back moved. This will continue as we enqueue more items. The front and back moved together in the last example so that the front could assume the position of the first item.

Since that was the first and only item then, the front and back sat at that position. But now that we have enqueued more items, the back will keep following the last item.

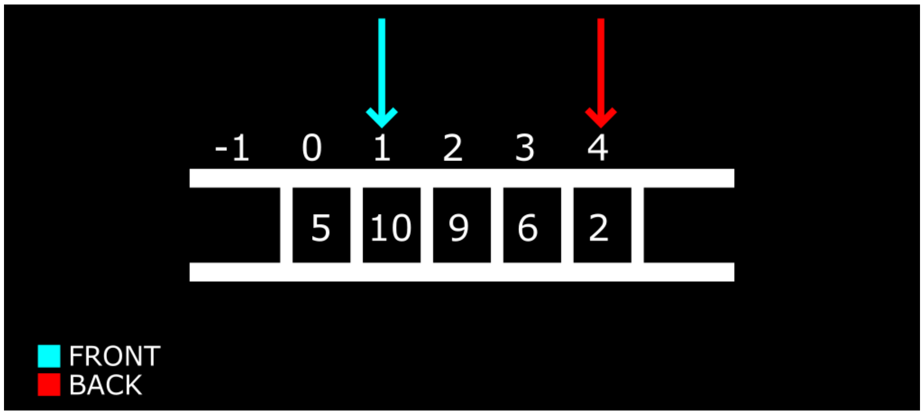
We will go on and fill up the array so we can see what happens when we dequeue.

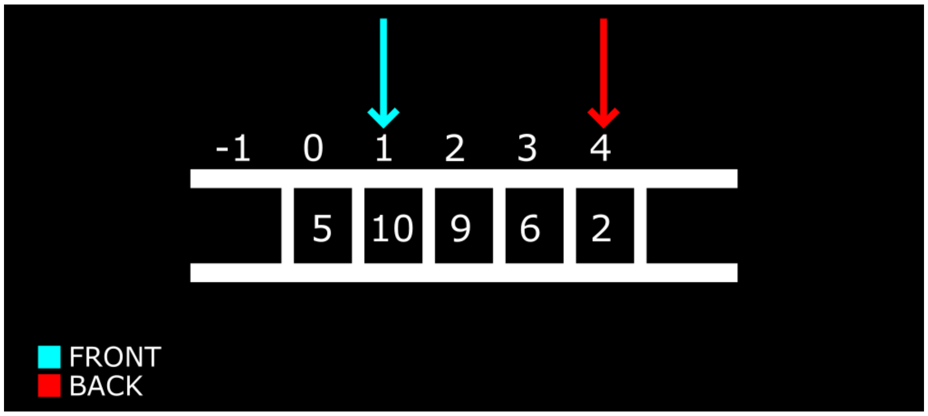


So the back arrow followed the items in the order they were added all the way to the last. Now let's delete (dequeue) some items.

Remember the **first-come-first-out** sequence? When we execute the dequeue operation, it will first remove 5 from the queue. If we execute it again then it will move to the next number which is 10 and continue in that order for as long as we call it.

Here, the first dequeue operation:





Now the front arrow has moved to index 1. This implies that the item at index 0 has been removed. By removed, we do not mean from the array but from the queue – only items from the front position to the back position are part of the queue.

In the same order, if we keep removing items, it will get to a point where the front arrow meets the back arrow at the end of the queue. If we dequeue again at that point, the front arrow will move past the back arrow and then the queue will be considered empty because there is nothing there to delete. When this happens, we will reset their index to -1 (their initial starting point).

Program:

```
public class Queue {
  int SIZE = 5;
  int items[] = new int[SIZE];
  int front, rear;
```

```java
Queue() {
  front = -1;
  rear = -1;
}

// check if the queue is full
boolean isFull() {
  if (front == 0 && rear == SIZE - 1) {
    return true;
  }
  return false;
}

// check if the queue is empty
boolean isEmpty() {
  if (front == -1)
    return true;
  else
    return false;
}

// insert elements to the queue
void enQueue(int element) {

  // if queue is full
  if (isFull()) {
    System.out.println("Queue is full");
  }
  else {
    if (front == -1) {
      // mark front denote first element of queue
      front = 0;
    }

    rear++;
    // insert element at the rear
    items[rear] = element;
    System.out.println("Insert " + element);
  }
}

// delete element from the queue
int deQueue() {
  int element;

  // if queue is empty
  if (isEmpty()) {
    System.out.println("Queue is empty");
    return (-1);
  }
  else {
    // remove element from the front of queue
    element = items[front];

    // if the queue has only one element
    if (front >= rear) {
      front = -1;
      rear = -1;
    }
    else {
      // mark next element as the front
      front++;
    }
    System.out.println( element + " Deleted");
    return (element);
  }
}

// display element of the queue
void display() {
  int i;
  if (isEmpty()) {
    System.out.println("Empty Queue");
  }
  else {
    // display the front of the queue
    System.out.println("\nFront index-> " + front);

    // display element of the queue
    System.out.println("Items -> ");
    for (i = front; i <= rear; i++)
      System.out.print(items[i] + "  ");

    // display the rear of the queue
    System.out.println("\nRear index-> " + rear);
  }
}
```

```
  public static void main(String[] args) {

    // create an object of Queue class
    Queue q = new Queue();

    // try to delete element from the queue
    // currently queue is empty
    // so deletion is not possible
    q.deQueue();

    // insert elements to the queue
    for(int i = 1; i < 6; i ++) {
      q.enQueue(i);
    }

    // 6th element can't be added to queue because queue is full
    q.enQueue(6);

    q.display();

    // deQueue removes element entered first i.e. 1
    q.deQueue();

    // Now we have just 4 elements
    q.display();

  }
}
```

Program 2:

```
public class Queue {
  int SIZE = 5;
  int items[] = new int[SIZE];
  int front, rear;

  Queue() {
    front = -1;
    rear = -1;
  }
```

```
// check if the queue is full
boolean isFull() {
  if (front == 0 && rear == SIZE - 1) {
    return true;
  }
  return false;
}

// check if the queue is empty
boolean isEmpty() {
  if (front == -1)
    return true;
  else
    return false;
}

// insert elements to the queue
void enQueue(int element) {

  // if queue is full
  if (isFull()) {
    System.out.println("Queue is full");
  }
  else {
    if (front == -1) {
      // mark front denote first element of queue
      front = 0;
    }

    rear++;
    // insert element at the rear
    items[rear] = element;
    System.out.println("Insert " + element);
  }
}

// delete element from the queue
int deQueue() {
  int element;

  // if queue is empty
  if (isEmpty()) {
    System.out.println("Queue is empty");
```

```
      return (-1);
    }
    else {
      // remove element from the front of queue
      element = items[front];

      // if the queue has only one element
      if (front >= rear) {
        front = -1;
        rear = -1;
      }
      else {
        // mark next element as the front
        front++;
      }
      System.out.println( element + " Deleted");
      return (element);
    }
  }

  // display element of the queue
  void display() {
    int i;
    if (isEmpty()) {
      System.out.println("Empty Queue");
    }
    else {
      // display the front of the queue
      System.out.println("\nFront index-> " + front);

      // display element of the queue
      System.out.println("Items -> ");
      for (i = front; i <= rear; i++)
        System.out.print(items[i] + "  ");

      // display the rear of the queue
      System.out.println("\nRear index-> " + rear);
    }
  }

  public static void main(String[] args) {

    // create an object of Queue class
    Queue q = new Queue();
    // try to delete element from the queue
    // currently queue is empty
    // so deletion is not possible
    q.deQueue();

    // insert elements to the queue
    for(int i = 1; i < 6; i ++) {
      q.enQueue(i);
    }

    // 6th element can't be added to queue because queue is full
    q.enQueue(6);

    q.display();

    // deQueue removes element entered first i.e. 1
    q.deQueue();

    // Now we have just 4 elements
    q.display();

  }
}
```

Output

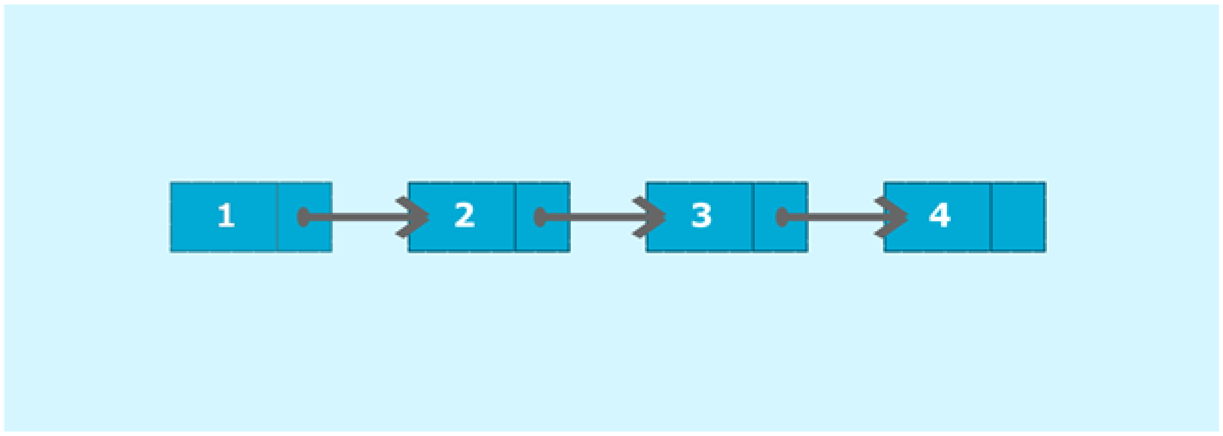
```
Queue is empty
Insert 1
Insert 2
Insert 3
Insert 4
Insert 5
Queue is full

Front index-> 0
Items ->
1  2  3  4  5
Rear index-> 4
1 Deleted
```

```
Front index-> 1
Items ->
2  3  4  5
Rear index-> 4
```

3. LinkedList

A linked list is a linear data structure in which each element of the list contains a pointer which points to the next element in the list. Each element in the linked list is called a node. Each node has two components: data and a pointer next which points to the next node in the list. The first node of the list is called a head, and the last node of the list is called a tail. The last node of the list contains a pointer to the null. Each node in the list can be accessed linearly by traversing through the list from head to tail.



Consider the above example; node 1 is the head of the list and node 4 is the tail of the list. Each node is connected in such a way that node 1 is pointing to node 2 which in turn pointing to node 3. Node 3 is again pointing to node 4. Node 4 is pointing to null as it is the last node of the list.

Algorithm

Create a class Node which has two attributes: data and next. Next is a pointer to the next node.

- Create another class which has two attributes: head and tail.

- addNode() will add a new node to the list:
    - Create a new node.
    - It first checks whether the head is equal to null which means the list is empty.
    - If the list is empty, both head and tail will point to the newly added node.
    - If the list is not empty, the new node will be added to the end of the list such that tail's next will point to the newly added node. This new node will become the new tail of the list.

a. display() will display the nodes present in the list:

- Define a node current which initially points to the head of the list.
- Traverse through the list till current points to null.
- Display each node by making current to point to node next to it in each iteration.

Program 1: Write a program to create your own linked list, add nodes and display the nodes.

```
public class LinkedList {
   //Represent a node of the singly linked list
   class Node{
      int data;
      Node next;

      public Node(int data) {
         this.data = data;
         this.next = null;
      }
   }

   //Represent the head and tail of the singly linked list
```

```
public Node head = null;
public Node tail = null;

//addNode() will add a new node to the list
public void addNode(int data) {
    //Create a new node
    Node newNode = new Node(data);

    //Checks if the list is empty
    if(head == null) {
        //If list is empty, both head and tail will point to new node
        head = newNode;
        tail = newNode;
    }
    else {
        //newNode will be added after tail such that tail's next will point to newNode
        tail.next = newNode;
        //newNode will become new tail of the list
        tail = newNode;
    }
}

//display() will display all the nodes present in the list
public void display() {
    //Node current will point to head
    Node current = head;

    if(head == null) {
        System.out.println("List is empty");
        return;
    }
    System.out.println("Nodes of linked list: ");
    while(current != null) {
        //Prints each node by incrementing pointer
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}

public static void main(String[] args) {

    LinkedList ll = new LinkedList();
```

```
        //Add nodes to the list
        ll.addNode(1);
        ll.addNode(2);
        ll.addNode(3);
        ll.addNode(4);

        //Displays the nodes present in the list
        ll.display();
    }
}
```

Output:

```
1 2 3 4
```

Program 2: To insert, delete at any position including size of LinkedList

```
class LinkedList{
    Node head;

    class Node{
        int data;
        Node next;

        Node(int data){
            this.data=data;
            this.next=null;
        }
    }

    void insertNodeAtFirst(int data){
        Node newNode = new Node(data);
        if(head==null){
            System.out.println("Adding node: "+newNode.data);
            head = newNode;
        }
        else{
            System.out.println("Adding node: "+newNode.data);
            newNode.next = head;
            head = newNode;
        }

    }

    void insertNodeAtLast(int data){
        Node newNode = new Node(data);
        if(head==null){
            System.out.println("Adding node: "+newNode.data);
            head = newNode;
        }
        else{
            System.out.println("Adding node: "+newNode.data);
            Node currNode = head.next;
            Node node2ndLast = head;
            while(currNode!=null){
                currNode = currNode.next;
                node2ndLast = node2ndLast.next;
            }
            node2ndLast.next = newNode;
        }

    }

    void insertNodeAtPosition(int data, int pos){
        Node newNode = new Node(data);
        if(pos<0){
            System.out.println("Invalid position to insert ");
        }
        else{
            if(pos==0){
                insertNodeAtFirst(data);
            }
            else if(pos==getSize()){
                insertNodeAtLast(data);
            }
            else if (pos>getSize()){
                System.out.println("List has "+getSize()+" nodes, position should not exceed
"+getSize());
```

```
        }
        else{
            Node currNode = head;
            int i = 1;
            while(i<pos){
                i++;
                currNode=currNode.next;
            }
            //System.out.println("currNode: "+currNode.data);
            //currNode.next=currNode.next.next;
            newNode.next=currNode.next;
            currNode.next=newNode;
        }
    }
}

void deleteAtFirst(){
    if(head==null){
       System.out.println("The list is empty, can't delete anything");
    }
    else{
        if(head.next==null){
            System.out.println("Deleting the only node "+head.data);
            head = null;
        }
        else{
            head = head.next;
        }
    }
}

void deleteAtLast(){
    if(head==null){
       System.out.println("The list is empty, can't delete anything");
    }
    else{
        if(head.next==null){
            System.out.println("Deleting the only node "+head.data);
            head = null;
        }
        else{
            Node currNode = head.next.next;
            Node node2ndLast = head;
            while(currNode!=null){
                currNode = currNode.next;
                node2ndLast = node2ndLast.next;
            }
            System.out.println("Deleting the node "+node2ndLast.next.data);
            node2ndLast.next=null;
        }
    }
}

void deleteAtPosition(int pos){
    if(pos<0){
        System.out.println("Invalid position to delete ");
    }
    else{
        if(pos==0){
            deleteAtFirst();
        }
        else if(pos==getSize()-1){
            deleteAtLast();
        }
        else if (pos>=getSize()){
            System.out.println("List has "+getSize()+" nodes, position should not exceed
"+getSize());
        }
        else{
            Node currNode = head;
            int i = 1;
            while(i<pos){
                i++;
                currNode=currNode.next;
            }
            //System.out.println("currNode: "+currNode.data);
            currNode.next=currNode.next.next;
        }
    }
}

void showElements(){
    if(head==null){
        System.out.println("The list is empty.");
    }
    else{
        Node currNode = head;
        while(currNode!=null){
            System.out.print(currNode.data+" -> ");
            currNode = currNode.next;
        }
        System.out.print(currNode+"\n");
    }
}

int getSize(){
    Node currNode = head;
    int size=0;
    while(currNode!=null){
        size++;
        currNode = currNode.next;
    }
    return size;
}
}


public class Main{
     public static void main(String []args){
    System.out.println("We are going to create our linked list...");

    LinkedList myList = new LinkedList();

    myList.insertNodeAtPosition(10, 2);

    myList.deleteAtFirst();
    myList.insertNodeAtFirst(5);
    myList.showElements();

    myList.deleteAtFirst();
    myList.showElements();

    myList.insertNodeAtFirst(4);
    myList.showElements();
    myList.insertNodeAtFirst(3);
    myList.showElements();
    myList.insertNodeAtFirst(2);
    myList.showElements();
    myList.insertNodeAtFirst(1);
    myList.showElements();

    myList.insertNodeAtPosition(10, 12);
    myList.insertNodeAtLast(6);
    myList.showElements();

    myList.insertNodeAtLast(7);
    myList.showElements();

    myList.insertNodeAtLast(8);
    myList.showElements();

    myList.insertNodeAtLast(9);
    myList.showElements();

    myList.insertNodeAtPosition(10, 9);
    myList.showElements();

    System.out.println("Total number of nodes in the list is "+myList.getSize());

    myList.deleteAtFirst();
    myList.showElements();

    myList.deleteAtFirst();
    myList.showElements();

    myList.deleteAtLast();
    myList.showElements();

    System.out.println("Total number of nodes in the list is "+myList.getSize());

    myList.deleteAtPosition(3);
    myList.showElements();

    myList.deleteAtPosition(1);
    myList.showElements();

    System.out.println("Total number of nodes in the list is "+myList.getSize());

    myList.insertNodeAtPosition(100,1);
    myList.showElements();
    System.out.println("Total number of nodes in the list is "+myList.getSize());

    myList.insertNodeAtPosition(200,1);
    myList.showElements();
    System.out.println("Total number of nodes in the list is "+myList.getSize());
```

```
        myList.insertNodeAtPosition(200,5);
        myList.showElements();
        System.out.println("Total number of nodes in the list is "+myList.getSize());

        myList.insertNodeAtPosition(1000,6);
        myList.showElements();
        System.out.println("Total number of nodes in the list is "+myList.getSize());
    }
}
```