BridgeLabz

Employability Delivered

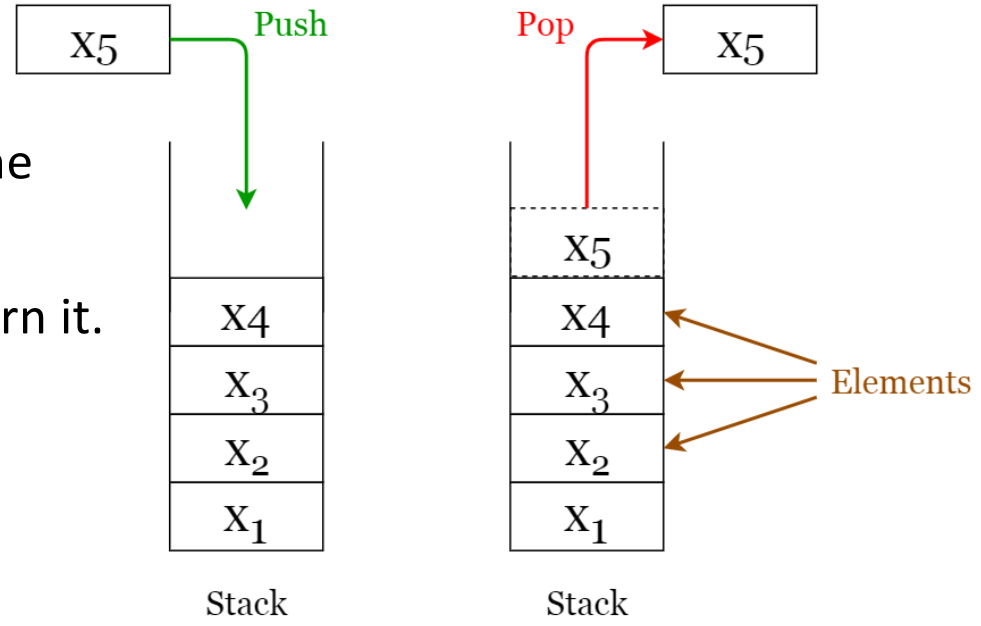
Java Core Concepts – Data Structures using Generics

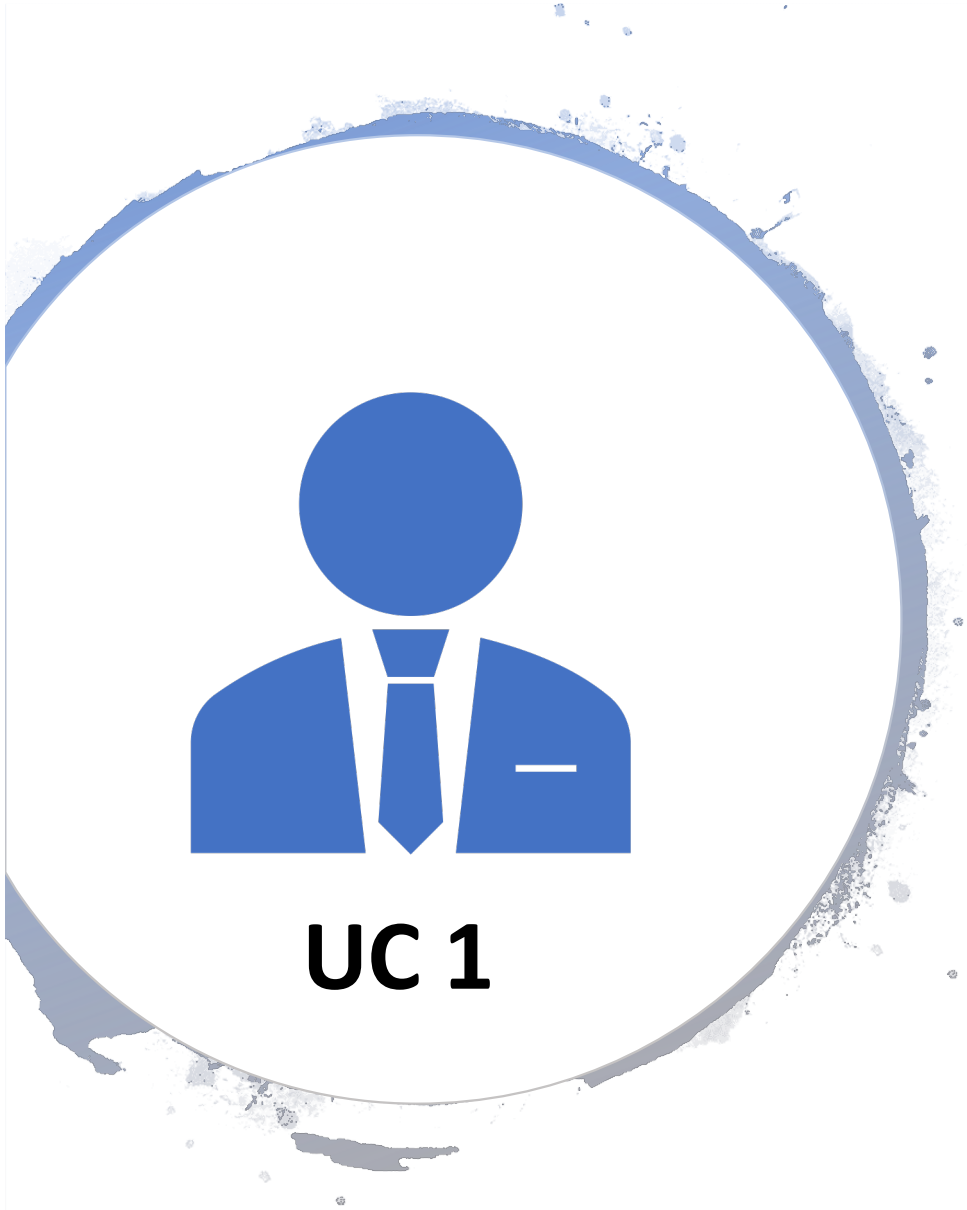
Section 2: Stacks and Queues

Stack

BridgeLabz
Employability Delivered

- **LIFO –** A Stack is Last in First Out Data Structure
- **Stack Operations –**
  - **Push:** Insert an element on to the top of the stack.
  - **Pop:** Delete the topmost element and return it.
  - **Peek:** Return the top element of the stack without deleting it.
  - **isEmpty:** Check if the stack is empty.
  - **isFull:** Check if the stack is full.





Ability to create a Stack of 56->30->70

- Use LinkedList to do the Stack Operations
- Here push will internally call add method on LinkedList.
- So 70 will be added first then 30 and then 56 to make 56 on top of the Stack

Ability to peak and pop from the Stack till it is empty

56->30->70

- Use LinkedList to do the Stack Operations

The Stack Abstract Data Type

The stack abstract data type is defined by the following structure and operations. A stack is structured, as described above, as an ordered collection of items where items are added to and removed from the end called the "top." Stacks are ordered LIFO. The stack operations are given below.

- `Stack()` creates a new stack that is empty. It needs no parameters and returns an empty stack.
- `push(item)` adds a new item to the top of the stack. It needs the item and returns nothing.
- `pop()` removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.
- `peek()` returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.
- `isEmpty()` tests to see whether the stack is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items on the stack. It needs no parameters and returns an integer.

Stack API

MyStack Code Snippet

BridgeLabz
Employability Delivered

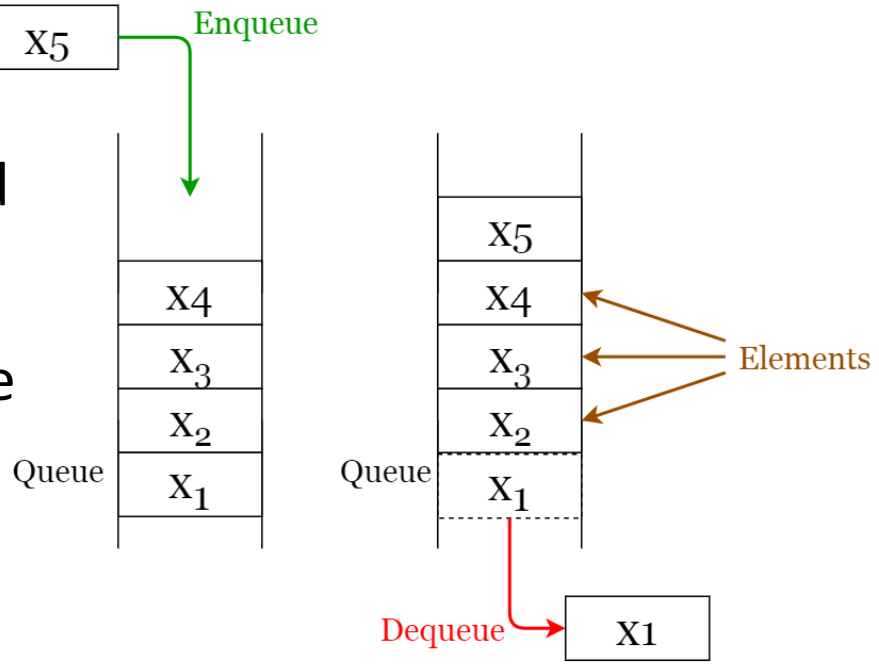
```java
public class MyStackTest {
    @Test
    public void given3NumbersWhenAddedToStackShouldHaveLastAddedNode() {
        MyStack<Integer> myStack = new MyStack<>();
        MyNode<Integer> myFirstNode = new MyNode<>( key: 70);
        MyNode<Integer> mySecondNode = new MyNode<>( key: 30);
        MyNode<Integer> myThirdNode = new MyNode<>( key: 56);
        myStack.push(myFirstNode);
        myStack.push(mySecondNode);
        myStack.push(myThirdNode);
        INode<Integer> myNode = myStack.peak();
        myStack.printStack();
        Assert.assertEquals(myThirdNode,myNode);
    }

    @Test
    public void given3NumbersInStackWhenPopedShouldMatchWithLastAddedNode() {
        MyStack<Integer> myStack = new MyStack<>();
        MyNode<Integer> myFirstNode = new MyNode<>( key: 70);
        MyNode<Integer> mySecondNode = new MyNode<>( key: 30);
        MyNode<Integer> myThirdNode = new MyNode<>( key: 56);
        myStack.push(myFirstNode);
        myStack.push(mySecondNode);
        myStack.push(myThirdNode);
        INode<Integer> popedNode = myStack.pop();
        myStack.printStack();
        Assert.assertEquals(myThirdNode,popedNode);
    }
}
```

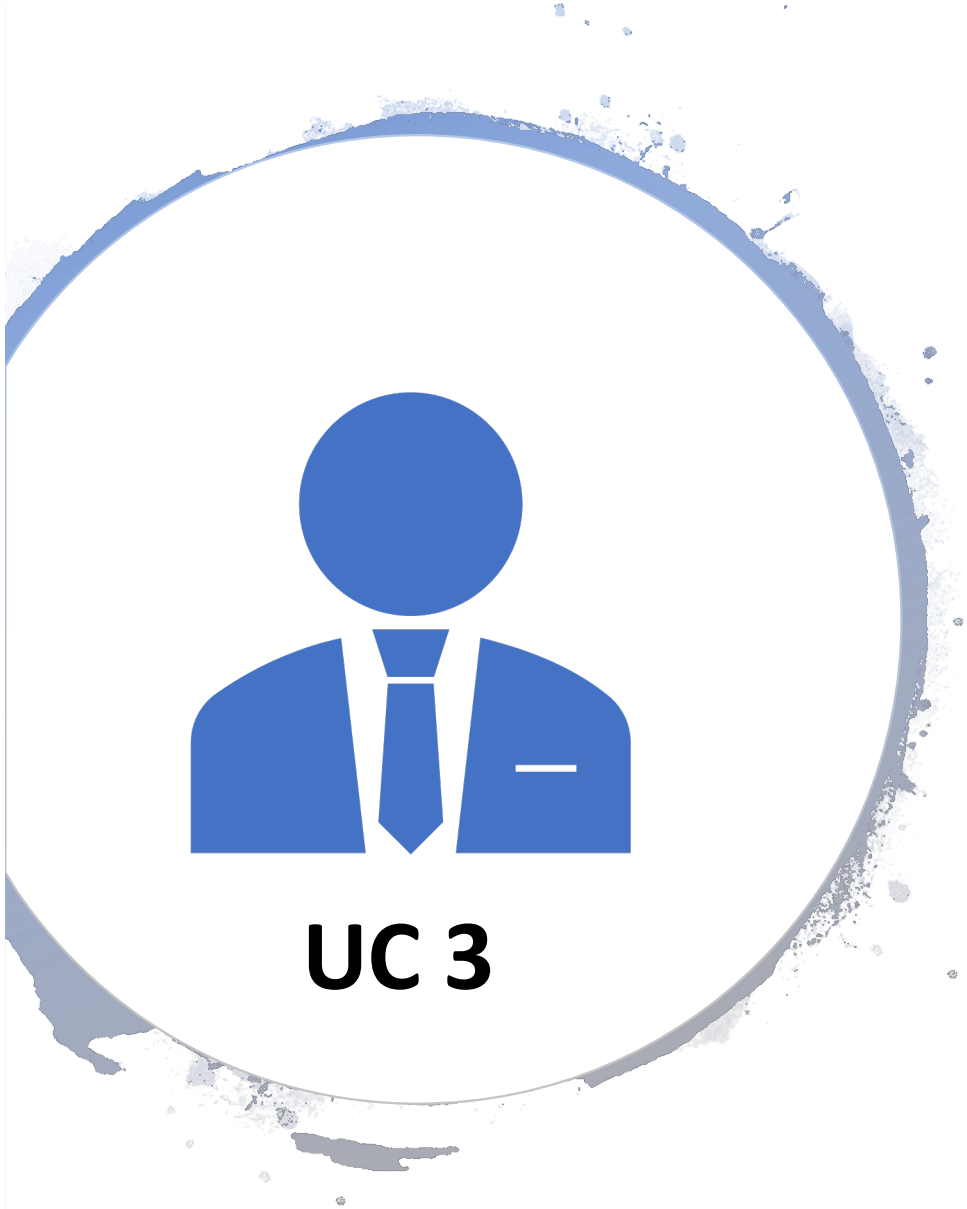
```java
public class MyStack<K> {
    private final MyLinkedList<K> myLinkedList;

    public MyStack() { myLinkedList = new MyLinkedList<>(); }

    public void push(INode<K> element) {
        myLinkedList.add(element);
    }

    public INode<K> peak() { return myLinkedList.head; }

    public INode<K> pop() { return myLinkedList.pop(); }

    public void printStack() { myLinkedList.printMyNodes(); }
}
```

Queue

BridgeLabz
Employability Delivered

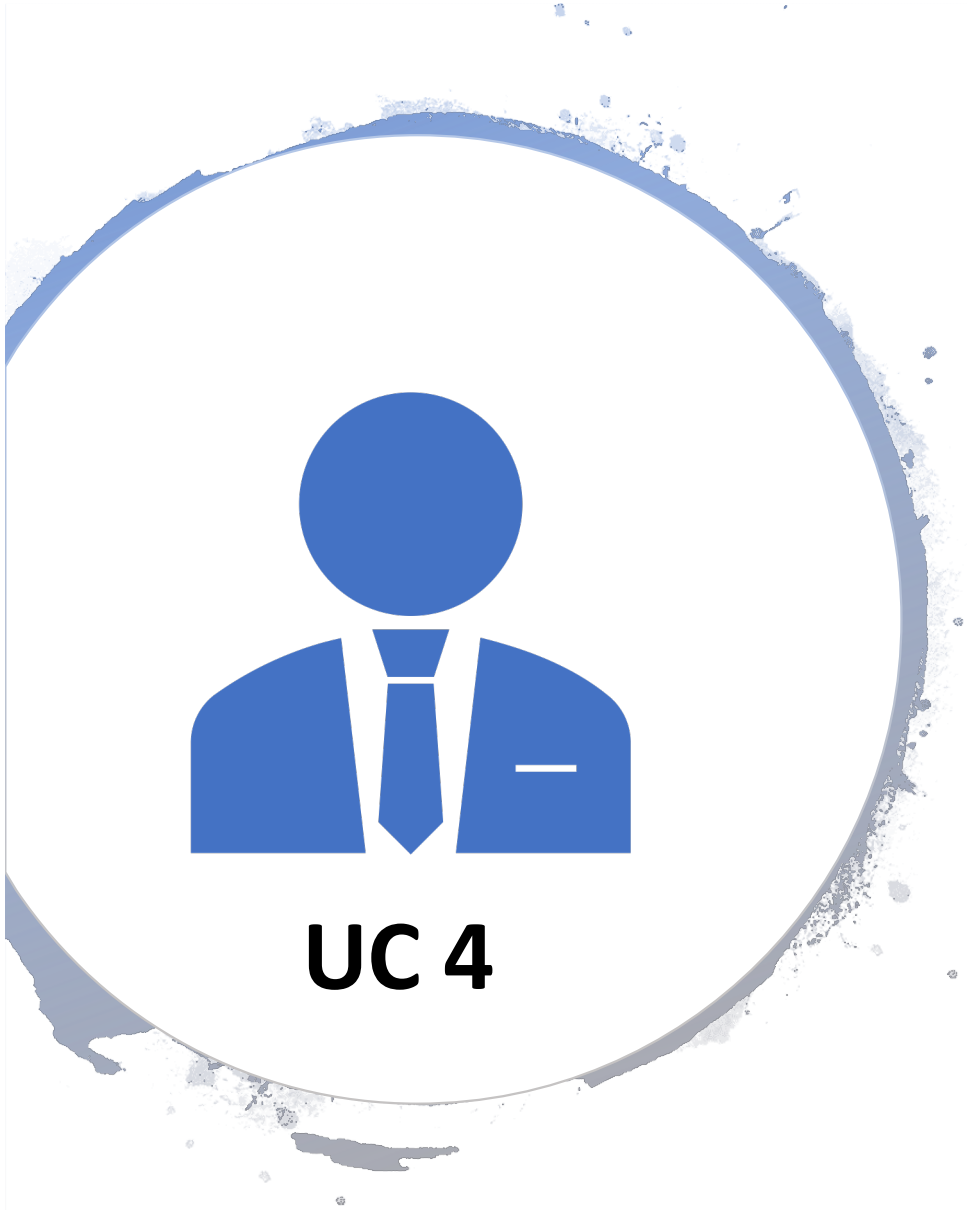
- **FIFO –** A Queue is First in First Out Data Structure
- **Queue Operations –**
  - **Enqueue –** Insert an element to the end of the queue.
  - **Dequeue –** Delete the element from the beginning of the queue.





Ability to create a Queue of 56->30->70

- Use LinkedList to do the Queue Operations
- Here enqueu will internally call append method on LinkedList.
- So 56 will be added first then 30 and then 70 to make 56 on top of the Stack



Ability to **dequeue** from the beginning

- Use LinkedList to do the Queue Operations

The Queue Abstract Data Type

The queue abstract data type is defined by the following structure and operations. A queue is structured, as described above, as an ordered collection of items which are added at one end, called the "rear," and removed from the other end, called the "front." Queues maintain a FIFO ordering property. The queue operations are given below.

- `Queue()` creates a new queue that is empty. It needs no parameters and returns an empty queue.
- `enqueue(item)` adds a new item to the rear of the queue. It needs the item and returns nothing.
- `dequeue()` removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
- `isEmpty()` tests to see whether the queue is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items in the queue. It needs no parameters and returns an integer.

As an example, if we assume that `q` is a queue that has been created and is currently empty, then Table 1 shows the results of a sequence of queue operations. The queue contents are shown such that the front is on the right. 4 was the first item enqueued so it is the first item returned by dequeue.

Queue API

BridgeLabz

Employability Delivered

Thank You