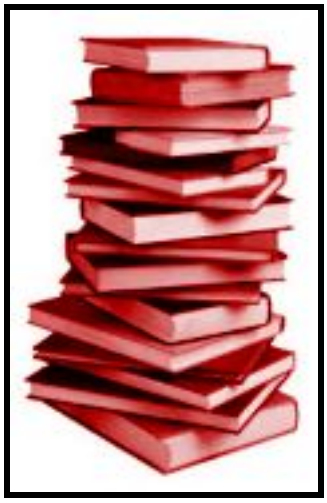
Stacks

Introduction

- Stacks are simple data structures that allow us to store and retrieve data sequentially.
- A stack is a linear data structure like arrays and linked lists.
- It is an abstract data type**(ADT)**.
- In a stack, the order in which the data arrives is essential. It follows the LIFO order of data insertion/abstraction. LIFO stands for **Last In First Out.**
- Consider the example of a pile of books:

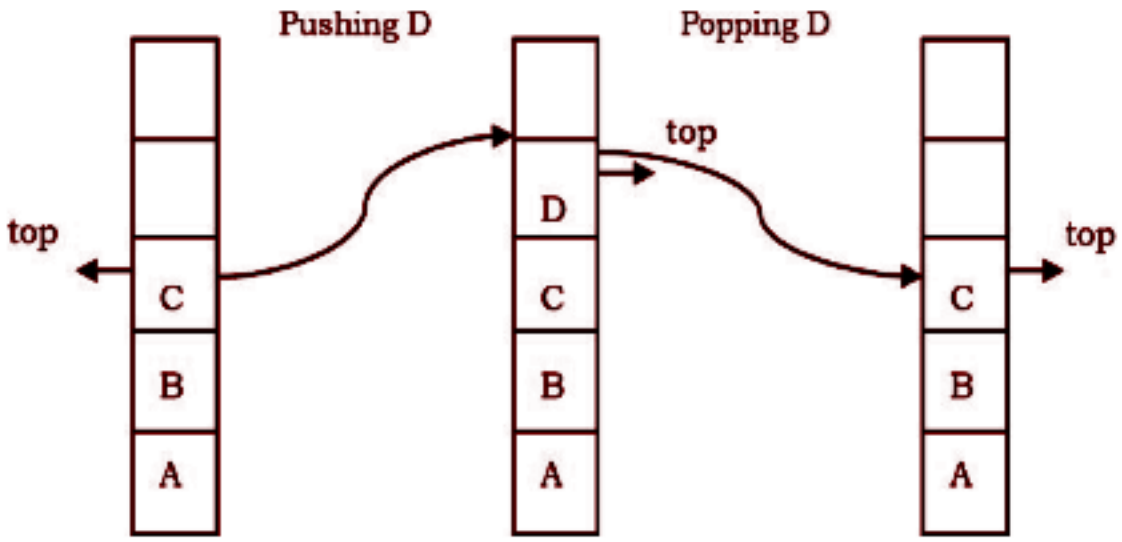


Here, unless the book at the topmost position is removed from the pile, we can't have access to the second book from the top and similarly, for the books below the second one. When we apply the same technique over the data in our program then, this pile-type structure is said to be a stack.

Like deletion, we can only insert the book at the top of the pile rather than at any other position. This means that the object/data that made its entry at the last would be one to come out first, hence known as **LIFO.**

Operations on the stack:

- In a stack, insertion and deletion are done at one end, called **top**.
- **Insertion**: This is known as a **push** operation.
- **Deletion**: This is known as a **pop** operation.



Main stack operations

• `push (int data)`: Insert data onto the stack.

• `int pop()`: Removes and returns the last inserted element from the stack.

Auxiliary stack operations

• `int top()`: Returns the last inserted element without removing it.

• `int size()`: Returns the number of elements stored in the stack.

• `boolean isEmpty()`: Indicates whether any elements are stored in the stack or not.

Performance

Let n be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

Space Complexity (for n push operations)	O(*n*)
Time Complexity of Push()	O(1)
Time Complexity of Pop()	O(1)
Time Complexity of Size()	O(1)
Time Complexity of IsEmptyStack()	O(1)
Time Complexity of IsFullStackf)	O(1)
Time Complexity of DeleteStackQ	O(1)

Exceptions

- Attempting the execution of an operation may sometimes cause an error condition, called an exception.
- Exceptions are said to be “thrown” by an operation that cannot be executed.
- Attempting the execution of pop() on an empty stack throws an exception called **Stack Underflow**.
- Trying to push an element in a full-stack throws an exception called **Stack Overflow**.

Implementing stack- Simple Array Implementation

This implementation of stack ADT uses an array. In the array, we add elements from left to right and use a variable to keep track of the index of the **top** element.

Consider the given implementation in Java for more understanding:

```
class StackUsingArray{
      int[] data;                    // Dynamic array created serving as stack
      int nextIndex                  // To keep the track of current top index
      int capacity;                // To keep the track of total size of stack

      public StackUsingArray(int totalSize) {    //Constructor
            data = new int[totalSize];
            nextIndex = 0;
            capacity = totalSize;
      }

      // return the number of elements present in my stack
      public int size() {
            return nextIndex;
      }

      public boolean isEmpty() {
            /*
            if(nextIndex == 0) {
                  return true;
            }
            else {
                  return false;
            }
            */

            return nextIndex == 0;    //Above program written in short-hand
      }

      // insert element
      public void push(int element) {
            if(nextIndex == capacity) {
                  System.out.println("Stack full");
                  return;
            }
            data[nextIndex] = element;
            nextIndex++;                              //Size incremented
      }

      // delete element
      public int pop() {
         //Before deletion check for empty to prevent underflow
            if(isEmpty()) {
                  System.out.println("Stack is empty");
                  return Integer.MIN_VALUE;
            }
```

```
            nextIndex--;                //Conditioned satisfied so deleted
            return data[nextIndex];
      }

      //to return the top element of the stack
      public int top() {
            if(isEmpty()) {                  // checked for empty stack
                  System.out.println("Stack is empty");
                  return Integer.MIN_VALUE;
            }
            return data[nextIndex - 1];
      }
}
```

Limitations of Simple Array Implementation

In programming languages like C++, Java, etc, the maximum size of an array must first be defined i.e. it is fixed and it cannot be changed.

Dynamic Stack

There is one limitation to the above approach, which is the size of the stack is fixed. In order to overcome this limitation, whenever the size of the stack reaches its limit we will simply double its size. To get the better understanding of this approach, look at the code below...

```
class StackUsingArray{
      int[] data;                  // Dynamic array created serving as stack
      int nextIndex               // To keep the track of current top index
      int capacity;             // To keep the track of total size of stack

      public StackUsingArray() {    //Constructor
            data = new int[4];
            nextIndex = 0;
            capacity = 4;
      }

      // return the number of elements present in my stack
      public int size() {
            return nextIndex;
```

```
	}

	public boolean isEmpty() {
		/*
		if(nextIndex == 0) {
			return true;
		}
		else {
			return false;
		}
		*/

		return nextIndex == 0;	//Above program written in short-hand
	}

	// insert element
	public void push(int element) {
		if(nextIndex == capacity) {
			int newData[] = new int[2 * capacity]; //Capacity doubled
			for(int i = 0; i < capacity; i++) {
				newData[i] = data[i];		//Elements copied
			}
			capacity *= 2;
			data = newData;
		}
		data[nextIndex] = element;
		nextIndex++;				//Size incremented
	}

	// delete element
	public int pop() {
	   //Before deletion check for empty to prevent underflow
		if(isEmpty()) {
			System.out.println("Stack is empty");
			return Integer.MIN_VALUE;
		}
		nextIndex--;			//Conditioned satisfied so deleted
		return data[nextIndex];
	}

	//to return the top element of the stack
	public int top() {
		if(isEmpty()) {			// checked for empty stack
			System.out.println("Stack is empty");
			return Integer.MIN_VALUE;
		}
		return data[nextIndex - 1];
```

```
	}
}
```

Stack using templates for Generic Data type Stack

While implementing the dynamic stack, we kept ourselves limited to the data of type integer only, but what if we want a generic stack(something that works for every other data type as well). For this we will be using templates. Refer the code below(based on the similar approach as used while creating dynamic stack):

```
class StackUsingArray <T>{
	T[] data;                    // Dynamic array created serving as stack
	int nextIndex                  // To keep the track of current top index
	int capacity;                // To keep the track of total size of stack

	public StackUsingArray() {    //Constructor
		data = new T[4];
		nextIndex = 0;
		capacity = 4;
	}

	// return the number of elements present in my stack
	public int size() {
		return nextIndex;
	}

	public boolean isEmpty() {
		/*
		if(nextIndex == 0) {
			return true;
		}
		else {
			return false;
		}
		*/

		return nextIndex == 0;    //Above program written in short-hand
	}

	// insert element
```

```
	public void push(T element) {
		if(nextIndex == capacity) {
			T newData[] = new T[2 * capacity]; //Capacity doubled
			for(int i = 0; i < capacity; i++) {
				newData[i] = data[i];		//Elements copied
			}
			capacity *= 2;
			data = newData;
		}
		data[nextIndex] = element;
		nextIndex++;				//Size incremented
	}

	// delete element
	public T pop() {
	   //Before deletion check for empty to prevent underflow
		if(isEmpty()) {
			System.out.println("Stack is empty");
			return Integer.MIN_VALUE;
		}
		nextIndex--;		//Conditioned satisfied so deleted
		return data[nextIndex];
	}

	//to return the top element of the stack
	public T top() {
		if(isEmpty()) {			// checked for empty stack
			System.out.println("Stack is empty");
			return Integer.MIN_VALUE;
		}
		return data[nextIndex - 1];
	}
}
```

You can see that every function whose return type was int initially now returns T type (i.e., template-type).

Generally, the template approach of stack is preferred as it can be used for any data type irrespective of it being int, char, float, etc.

Stack using Generic Linked Lists

Till now we have learned how to implement a stack using arrays, but as discussed earlier, we can also create a stack with the help of linked lists. All the five functions that stacks can perform could be made using linked lists:

```
class Node<T> {                  //Node class for Linked list
      T data;
      Node<T> next;

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


class Stack {
      Node<T> head;
      Node<T> tail;
      int size;         // number of elements present in stack

      public Stack() {         // Constructor to initialize the head and
                               //tail to NULL and size to zero
      }

      public int getSize() {   // traverse the LL and return its length

      }

      public boolean isEmpty() {// check if the head pointer is NULL or not

      }

      public void push(T element) { // insert the newNode at the end
                                    // update the tail node
      }

      public T pop() {  // remove the tail node and then update the tail
                        // pointer to the previous position
      }

      public T top() { //return the value at the tail node.
```

```
        }
}
```

Inbuilt Stack in Java

Java provides the in-built stack in it's **library** which can be used instead of creating/writing a stack class each time. To use this stack, we need to use the import following file:

```
import java.util.Stacks;
```

To declare a stack use the following syntax:

```
Stack <datatype_that_will_be_stored> Name_of_stack = new Stack<>();
```

There are various functions available in this module:

- **st.push(value_to_be_inserted)** : To insert a value in the stack
- **st.top()** : Returns the value at the top of the stack
- **st.pop()** : Deletes the value at the top from the stack.
- **st.size()** : Returns the total number of elements in the stack.
- **st.isEmpty()** : Returns a boolean value (True for empty stack and vice versa).

Problem Statement- Balanced Parenthesis

For a given string expression containing only round brackets or parentheses, check if they are balanced or not. Brackets are said to be balanced if the bracket which opens last, closes first. You need to return a boolean value indicating whether the expression is balanced or not.

Approach:

- We will use stacks.
- Each time, when an open parenthesis is encountered, push it in the stack, and when closed parenthesis is encountered, match it with the top of the stack and pop it.
- If the stack is empty at the end, return Balanced otherwise, Unbalanced.

Java Code:

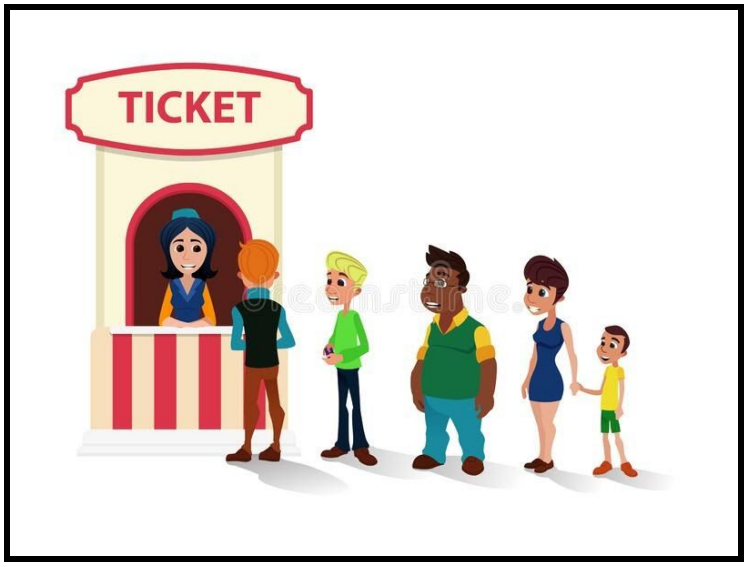
```
public String checkBalanced(inputStr){//Function to check parentheses
    Stack<Character> s = new Stack<>(); //The stack
    for(char i : inputStr.toCharArray){
        if (i=='[' || i=='{' || i=='('
            s.push(i);
        else if (i==']' || i=='}' || i==')')
            if (s.size()>0 && s.top()==i)
                s.pop();
            else
                return "Unbalanced";
    }
    if (s.size() == 0)
        return "Balanced";
    else
        return "Unbalanced";
}
```

Queues

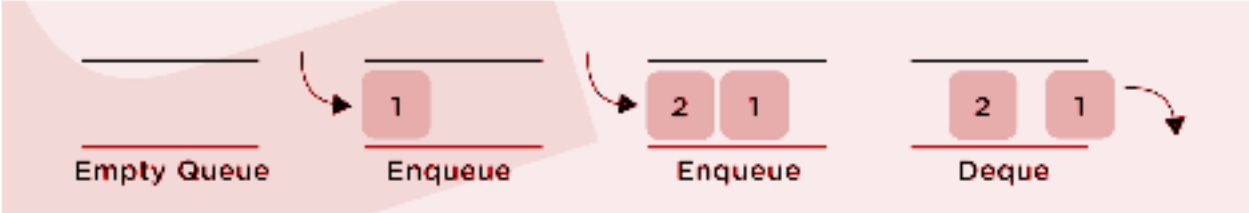
Introduction

- Like stack, the queue is also an abstract data type.
- As the name suggests, in queue elements are inserted at one end while deletion takes place at the other end.
- Queues are open at both ends, unlike stacks that are open at only one end(the top).

Let us consider a queue at a movie ticket counter:



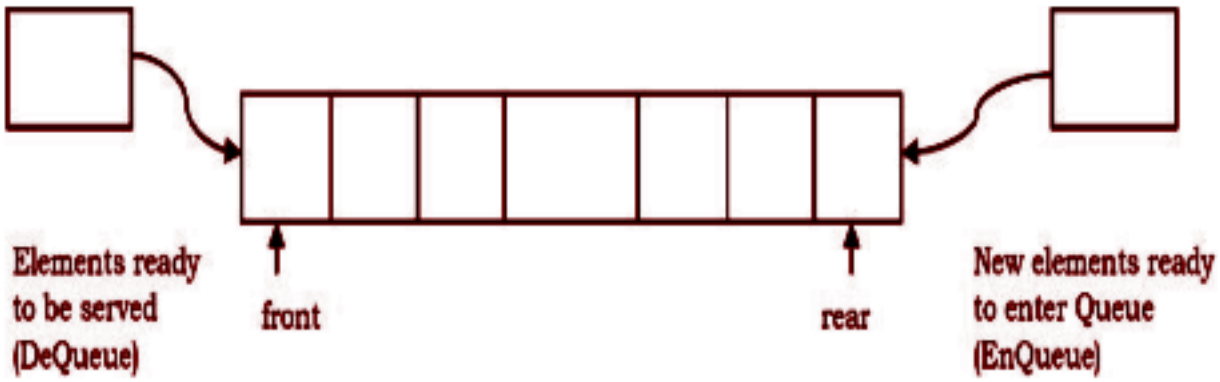
- Here, the person who comes first in the queue is served first with the ticket while the new seekers of tickets are added back in the line.
- This order is known as **First In First Out (FIFO)**.
- In programming terminology, the operation to add an item to the queue is called "enqueue", whereas removing an item from the queue is known as "dequeue".

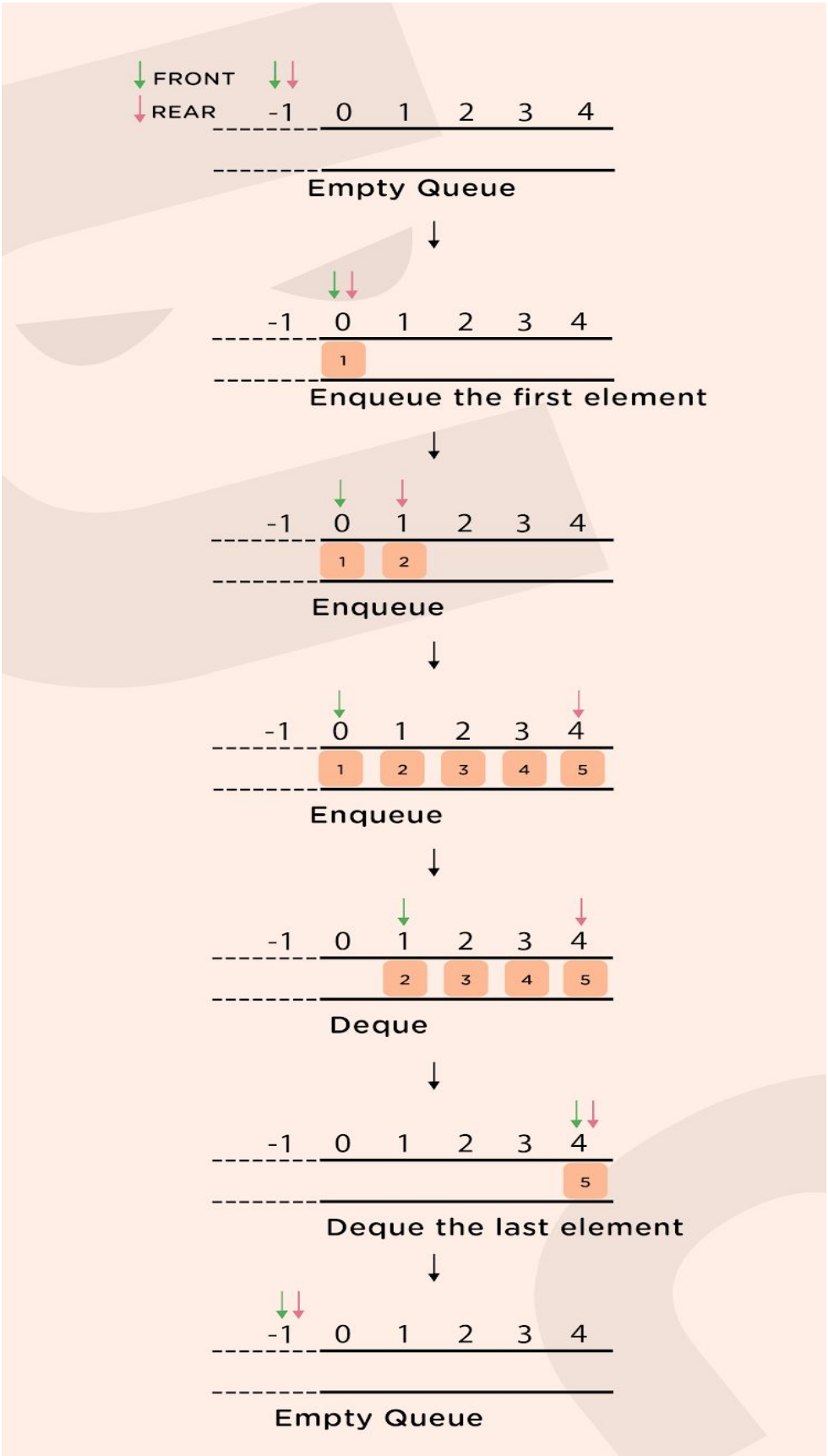


Working of A Queue

Queue operations work as follows:

1. Two pointers called **FRONT** and **REAR** are used to keep track of the first and last elements in the queue.
2. When initializing the queue, we set the value of FRONT and REAR to -1.
3. On **enqueuing** an element, we increase the value of the REAR index and place the new element in the position pointed to by REAR.
4. On **dequeuing** an element, we return the value pointed to by FRONT and increase the FRONT index.
5. Before enqueuing, we check if the queue is already full.
6. Before dequeuing, we check if the queue is already empty.
7. When enqueuing the first element, we set the value of FRONT to 0.
8. When dequeuing the last element, we reset the values of FRONT and REAR to -1.




FRONT
REAR
-1 0 1 2 3 4
Empty Queue
1
Enqueue the first element
1 2
Enqueue
1 2 3 4 5
Enqueue
2 3 4 5
Deque
5
Deque the last element
Empty Queue

Applications of queue

- CPU Scheduling, Disk Scheduling.
- When data is transferred asynchronously between two processesQueue is used for synchronization. eg: IO Buffers, pipes, file IO, etc.
- Handling of interrupts in real-time systems.
- Call Center phone systems use Queues to hold people in order of their calling.

Implementation of A Queue Using Array

Queue contains majorly these five functions that we will be implementing:

- **enqueue()**: Insertion of element
- **dequeue()**: Deletion of element
- **front()**: returns the element present in the front position
- **getSize()**: returns the total number of elements present at current stage
- **isEmpty()**: returns boolean value, TRUE for empty and FALSE for non-empty.

Now, let's implement these functions in our program.

NOTE: We will be using templates in the implementation, so that it can be generalised.

```
class QueueUsingArray <T> {
      T data;                                  // to store data
      int nextIndex;                          // to store next index
      int firstIndex;                        // to store the first index
      int size;                             // to store the size
      int capacity;                  // to store the capacity it can hold

      public QueueUsingArray(int s) {   // Constructor to initialize values
            data = new T[s];
            nextIndex = 0;
            firstIndex = -1;
```

```
        size = 0;
        capacity = s;
    }

    public int getSize() {         // Returns number of elements present
        return size;
    }

    public boolean isEmpty() {     // To check if queue is empty or not
        return size == 0;
    }

    public void enqueue(T element) {        // Function for insertion
        if(size == capacity) { // To check if the queue is already full
            System.out.println("Queue Full!");
            return;
        }
        data[nextIndex] = element;     // Otherwise added a new element
        nextIndex = (nextIndex + 1) % capacity ;    // in cyclic way
        if(firstIndex == -1) {           // Suppose if queue was empty
            firstIndex = 0;
        }
        size++;                    // Finally, incremented the size
    }



    public T front() {       //  To return the element at front position
        if(isEmpty()) {    // To check if the queue was initially empty
            System.out.println("Queue is Empty!");
            return 0;
        }
        return data[firstIndex];     // otherwise returned the element
    }

    public T dequeue() {               // Function for deletion
        if(isEmpty()) {       // To check if the queue was empty
            System.out.println("Queue is Empty!");
            return 0;
        }
        T ans = data[firstIndex];
        firstIndex = (firstIndex + 1) % capacity;
        size--;           // Decrementing the size by 1
        if(size == 0) {  // If queue becomes empty after deletion, then
            firstIndex = -1;    // resetting the original parameters
            nextIndex = 0;
```

```
            }
            return ans;
      }
}
```

Dynamic queue

In the dynamic queue. we will be preventing the condition where the queue becomes full and we were not able to insert any further elements in that.

As we all know that when the queue is full it means the internal array that we are using in the form of queue has become full, we can resolve this problem by creating a new array of double the size of previous one and copy pasting the elements of previous array to the new one. Now this new array which has the double size will be considered as our queue. We will do this in insert function when we check for queue full (size==capacity), when this happens we will discard the previous array and create a new array of double size, copy pasting all the elements so that we don't lose the data. Let's now check the implementation of the same.

Implementation is pretty similar to the static approach discussed above. A few minor changes are there which could be followed with the help of comments in the code below.

```
class QueueUsingArray <T> {
      T data;                                  // to store data
      int nextIndex;                          // to store next index
      int firstIndex;                        // to store the first index
      int size;                             // to store the size
      int capacity;                  // to store the capacity it can hold

      public QueueUsingArray() {   // Constructor to initialize values
            data = new T[4];
            nextIndex = 0;
            firstIndex = -1;
            size = 0;
            capacity = 4;
```

```
    }

    public int getSize() {          // Returns number of elements present
        return size;
    }

    public boolean isEmpty() {    // To check if queue is empty or not
        return size == 0;
    }

    public void enqueue(T element) {        // Function for insertion
        if(size == capacity) { // To check if the queue is already full
            T[] newData = new T[2 * capacity];// we simply doubled
                                              // the capacity
            int j = 0;
            for(int i=firstIndex; i<capacity; i++) {// Now copied the
                                                    //Elements to new one
                newData[j] = data[i];
                j++;
            }
            for(int i=0; i<firstIndex; i++) {//Overcoming the initial
                                        //  cyclic insertion by copying
                                        //  the elements linearly
                newData[j] = data[i];
                j++;
            }
            data = newData;
            firstIndex = 0;
            nextIndex = capacity;
            capacity *= 2;          // Updated here as well
        }
        data[nextIndex] = element;     // Otherwise added a new element
        nextIndex = (nextIndex + 1) % capacity ;    // in cyclic way
        if(firstIndex == -1) {           // Suppose if queue was empty
            firstIndex = 0;
        }
        size++;                    // Finally, incremented the size
    }



    public T front() {        //  To return the element at front position
        if(isEmpty()) {    // To check if the queue was initially empty
            System.out.println("Queue is Empty!");
            return 0;
        }
```

```
            return data[firstIndex];     // otherwise returned the element
      }

      public T dequeue() {                // Function for deletion
            if(isEmpty()) {       // To check if the queue was empty
                  System.out.println("Queue is Empty!");
                  return 0;
            }
            T ans = data[firstIndex];
            firstIndex = (firstIndex + 1) % capacity;
            size--;           // Decrementing the size by 1
            if(size == 0) {  // If queue becomes empty after deletion, then
                  firstIndex = -1;    // resetting the original parameters
                  nextIndex = 0;
            }
            return ans;
      }
}
```

Queues using Generic LL

Given below is an implementation of Queue using Linked List. This is similar to the way we wrote the LL Implementation for a Stack:

```
class Node <T> {       // Node class for linked list, no change needed
      T data;
      Node<T> next;
      Node(T data) {
            this -> data = data;
            next = NULL;
      }
}

class Queue <T> {
      Node<T> head;                  // for storing front of queue
      Node<T> tail;                 // for storing tail of queue
      int size;                     // number of elements in queue

      public Queue() {     // Constructor to initialise head, tail to NULL
                           // and size to 0
      }

      public int getSize() {      // just return the size of linked list
```

```
    }

    public boolean isEmpty() {      // just check if head is NULL or not

    }

    public void enqueue(T element) {   // Simply insert the new node
                                       //at the tail of LL

    }

    public T front() { // Returns the head pointer of LL.
                       // Be careful for the case when size is 0
    }

    public T dequeue() {  // moves the head pointer one position ahead
                          // and deletes the head pointer.
    }                     // Also decrease the size by 1
}
```

In-built Queue in Java

Java provides the in-built queue in it's **library** which can be used instead of creating/writing a queue class each time. To use this queue, we need to use the import following file:

```
import java.util.Queues;
import java.util.LinkedList;
```

Key functions of this in-built queue:

- **.push(element_value)** : Used to insert the element in the queue
- **.pop()** : Used to delete the element from the queue
- **.front()** : Returns the element at front of the queue

- **.size()** : Returns the total number of elements present in the queue
- **.isEmpty()** : Returns TRUE if the queue is empty and vice versa

Let us now consider an example to implement queue using inbuilt library:

Problem Statement: Implement the following parts using queue:

1. Declare a queue of integers and insert the following elements in the same order as mentioned: 10, 20, 30, 40, 50, 60.
2. Now tell the element that is present at the front position of the queue
3. Now delete an element from the front side of the queue and again tell the element present at the front position of the queue.
4. Print the size of the queue and also tell if the queue is empty or not.
5. Now, print all the elements that are present in the queue.

```
import java.util.Queues;
import java.util.LinkedList;

Class QueueTesting{
    public static void main(String[] args) {
         Queue<Integer> q = new LinkedList<>();
          q.push(10);      // part 1
          q.push(20);
          q.push(30);
          q.push(40);
          q.push(50);
          q.push(60);
          System.out.println(q.front());     // Part 2
          q.pop();                           // Part 3
          System.out.println(q.front());     // Part 3
          System.out.println(q.size());      // Part 4
          System.out.println(q.isEmpty());  // prints 1 for TRUE and 0 for
                                            // FALSE(Part 4)

          while(!q.isEmpty()) {  // prints all the elements until the queue
                                 // is empty (Part 5)
                System.out.println(q.front());
                q.pop();
```

```
            }
        }
}
```

We get the following output:

```
10
20
5
0
20
30
40
50
60
```