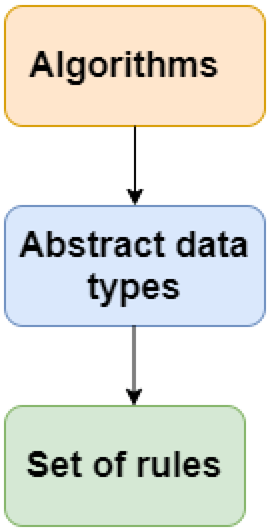
Data Structures in java

Data Structure is a way to store and organize data so that it can be used efficiently

The data structure name indicates itself that organizing the data in memory.

To structure the data in memory, 'n' number of algorithms were proposed, and all these algorithms are known as Abstract data types. These abstract data types are the set of rules.



Types of Data Structures

There are two types of data structures:

- Primitive data structure
- Non-primitive data structure

Primitive Data structure

The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

Non-Primitive Data structure

The non-primitive data structure is divided into two types:

- Linear data structure
- Non-linear data structure

Linear data structure

The arrangement of data in a sequential manner is known as a linear data structure. The data structures used for this purpose are **Arrays, Linked list, Stacks, and Queues**. In these data structures, one element is connected to only one another element in a linear form.

Non-linear data structure

When one element is connected to the 'n' number of elements known as a non-linear data structure. The best example is **trees and graphs**. In this case, the elements are arranged in a random manner.

Data structures can also be classified as:

- **Static data structure:** It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.
- **Dynamic data structure:** It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

Major Operations

The major or the common operations that can be performed on the data structures are:

- **Searching:** We can search for any element in a data structure.
- **Sorting:** We can sort the elements of a data structure either in an ascending or descending order.
- **Insertion:** We can also insert the new element in a data structure.

- **Updation:** We can also update the element, i.e., we can replace the element with another element.
- **Deletion:** We can also perform the delete operation to remove the element from the data structure.

Which Data Structure?

A data structure is a way of organizing the data so that it can be used efficiently. Here, we have used the word efficiently, which in terms of both the space and time. For example, a stack is an ADT (Abstract data type) which uses either arrays or linked list data structure for the implementation. Therefore, we conclude that we require some data structure to implement a particular ADT.

An ADT tells **what** is to be done and data structure tells **how** it is to be done. In other words, we can say that ADT gives us the blueprint while data structure provides the implementation part. Now the question arises: how can one get to know which data structure to be used for a particular ADT?.

As the different data structures can be implemented in a particular ADT, but the different implementations are compared for time and space. For example, the Stack ADT can be implemented by both Arrays and linked list. Suppose the array is providing time efficiency while the linked list is providing space efficiency, so the one which is the best suited for the current user's requirements will be selected.

Advantages of Data structures

The following are the advantages of a data structure:

- **Efficiency:** If the choice of a data structure for implementing a particular ADT is proper, it makes the program very efficient in terms of time and space.
- **Reusability:** The data structure provides reusability means that multiple client programs can use the data structure.
- **Abstraction:** The data structure specified by an ADT also provides the level of abstraction. The client cannot see the internal working of the data structure, so it does not have to worry about the implementation part. The client can only see the interface.

Sorting

Sorting is a class of algorithms that are tasked with rearranging the positions of elements of an array such that all of its elements are either in ascending or descending order.

A good sorting algorithm also needs to ensure that elements having the same value don't change their locations in the sorted array. Sorting is necessary for getting a concrete understanding of data structures and algorithms.

5 Popular Sorting Algorithms in Java

Java is a flexible language in itself and supports the operation of a variety of sorting algorithms. Most of these algorithms are extremely flexible themselves and can be implemented with both **a recursive** as well as an **iterative** approach.

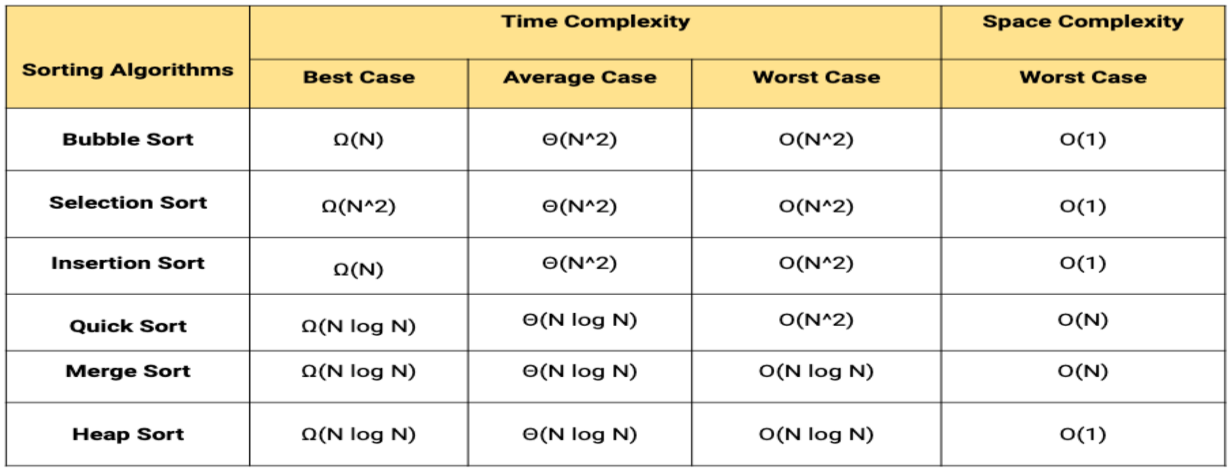
Here are 5 most popular sorting algorithms in java:

1. Merge Sort
2. Heap Sort
3. Insertion Sort
4. Selection Sort
5. Bubble Sort

Time Complexity

Now, learn about the time complexity of each sorting algorithm in java. Merge sort is a divide and conquer algorithm - hence it offers a more optimized approach for sorting than the others.

The time complexity of mergeSort() function is O(nlogn) while the time complexity of merge() function is O(n) - making the average complexity of the algorithm as O(nlogn). Heap sort, like merge sort, is an optimized sorting algorithm (even though it is not a part of the divide and conquer paradigm). The time complexity of heapify() is O(nlogn) while the time complexity of the heapSort() function is O(n) – making the average complexity of the algorithm as O(nlogn). Selection sort, bubble sort, and insertion sort all have the best case time complexity is O(n) and the worst-case time complexity is O(n2).



Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(1)

Java Sorting Algorithms Cheat Sheet

Here is a cheat sheet for all sorting algorithms in Java:

Algorithm	Approach	Best Time Complexity
Merge Sort	Split the array into smaller subarrays till pairs of elements are achieved, and then combine them in such a way that they are in order.	O(n log (n))
Heap Sort	Build a max (or min) heap and extract the first element of the heap (or root), and then send it to the end of the heap. Decrement the size of the heap and repeat till the heap has only one node.	O(n log (n))
Insertion Sort	In every run, compare it with the predecessor. If the current element is not in the correct location, keep shifting the predecessor subarray till the correct index for the element is found.	O (n)
Selection Sort	Find the minimum element in each run of the array and swap it with the element at the current index is compared.	O(n^2)
Bubble Sort	Keep swapping elements that are not in their right location till the array is sorted.	O(n)

Bubble Sort

In Bubble sort, Each element of the array is compared with its adjacent element. The algorithm processes the list in passes. A list with n elements requires n-1 passes for sorting. Consider an array A of n elements whose elements are to be sorted by using Bubble sort. The algorithm processes like following.

1. In Pass 1, A[0] is compared with A[1], A[1] is compared with A[2], A[2] is compared with A[3] and so on. At the end of pass 1, the largest element of the list is placed at the highest index of the list.
2. In Pass 2, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of Pass 2 the second largest element of the list is placed at the second highest index of the list.
3. In pass n-1, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of this pass. The smallest element of the list is placed at the first index of the list.

Algorithm :

- **Step 1**: Repeat Step 2 For i = 0 to N-1
- **Step 2**: Repeat For J = i + 1 to N - I
- **Step 3**: IF A[J] > A[i]
  SWAP A[J] and A[i]
  [END OF INNER LOOP]
  [END OF OUTER LOOP
- **Step 4**: EXIT

Complexity

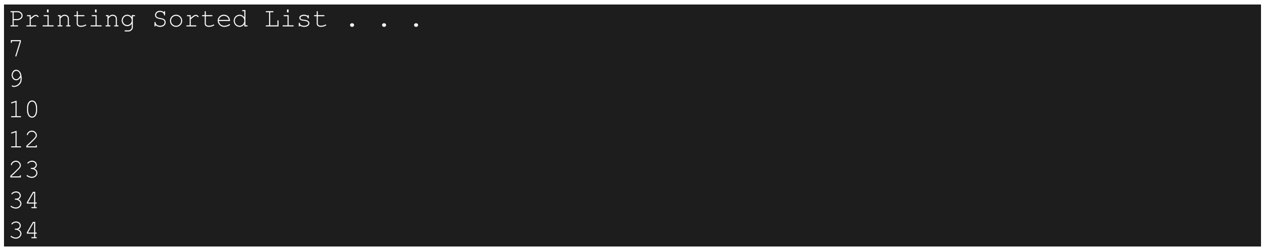
Scenario	**Complexity**
Space	O(1)
Worst case running time	$O(n^2)$

Average case running time	O(n)
Best case running time	O(n^2)

Java Program

```java
public class BubbleSort {
    public static void main(String[] args) {
    int[] a = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    for(int i=0;i<10;i++)
    {
        for (int j=0;j<10;j++)
        {
            if(a[i]<a[j])
            {
                int temp = a[i];
                a[i]=a[j];
                a[j] = temp;
            }
        }
    }
    System.out.println("Printing Sorted List ...");
    for(int i=0;i<10;i++)
    {
        System.out.println(a[i]);
    }
} }
```

Output:



```
Printing Sorted List . . .
7
9
10
12
23
34
34
```

```
44
78
101
```

Searching

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the arrangement of the list.

- Linear Search
- Binary Search

Linear Search

Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.

Linear search is mostly used to search an unordered list in which the items are not sorted. The algorithm of linear search is given as follows.

Algorithm

- LINEAR_SEARCH(A, N, VAL)
- **Step 1:** [INITIALIZE] SET POS = -1
- **Step 2:** [INITIALIZE] SET I = 1
- **Step 3:** Repeat Step 4 while I<=N
- **Step 4:** IF A[I] = VAL
  SET POS = I
  PRINT POS
  Go to Step 6
  [END OF IF]
  SET I = I + 1
  [END OF LOOP]

- **Step 5:** IF POS = -1
  PRINT " VALUE IS NOT PRESENTIN THE ARRAY "
  [END OF IF]
- **Step 6:** EXIT

Complexity of algorithm

Complexity	Best Case	Average Case	Worst Case
Time	O(1)	O(n)	O(n)
Space			O(1)

```
import java.util.Scanner;

public class Leniear_Search {
public static void main(String[] args) {
   int[] arr = {10, 23, 15, 8, 4, 3, 25, 30, 34, 2, 19};
   int item,flag=0;
   Scanner sc = new Scanner(System.in);
   System.out.println("Enter Item ?");
   item = sc.nextInt();
   for(int i = 0; i<10; i++)
   {
      if(arr[i]==item)
      {
         flag = i+1;
         break;
      }
      else
         flag = 0;
   }
   if(flag != 0)
   {
      System.out.println("Item found at location" + flag);
   }
```

```
        else
            System.out.println("Item not found");

    }
}
```

Output:

```
Enter Item ?
23
Item found at location 2
Enter Item ?
22
Item not found
```

Binary Search

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Binary search algorithm is given below.

BINARY_SEARCH(A, lower_bound, upper_bound, VAL)

- **Step 1:** [INITIALIZE] SET BEG = lower_bound
  END = upper_bound, POS = - 1
- **Step 2:** Repeat Steps 3 and 4 while BEG <=END
- **Step 3:** SET MID = (BEG + END)/2
- **Step 4:** IF A[MID] = VAL
  SET POS = MID
  PRINT POS
  Go to Step 6
  ELSE IF A[MID] > VAL

```
        SET END = MID - 1
        ELSE
        SET BEG = MID + 1
        [END OF IF]
        [END OF LOOP]
```

- **Step 5:** IF POS = -1
  PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
  [END OF IF]
- **Step 6:** EXIT

Complexity

SN	Performance	Complexity
1	Worst case	O(log n)
2	Best case	O(1)
3	Average Case	O(log n)
4	Worst case space complexity	O(1)

Example

Let us consider an array arr = {1, 5, 7, 8, 13, 19, 20, 23, 29}. Find the location of the item 23 in the array.

In 1st step :

1. BEG = 0
2. END = 8ron
3. MID = 4
4. a[mid] = a[4] = 13 < 23, therefore

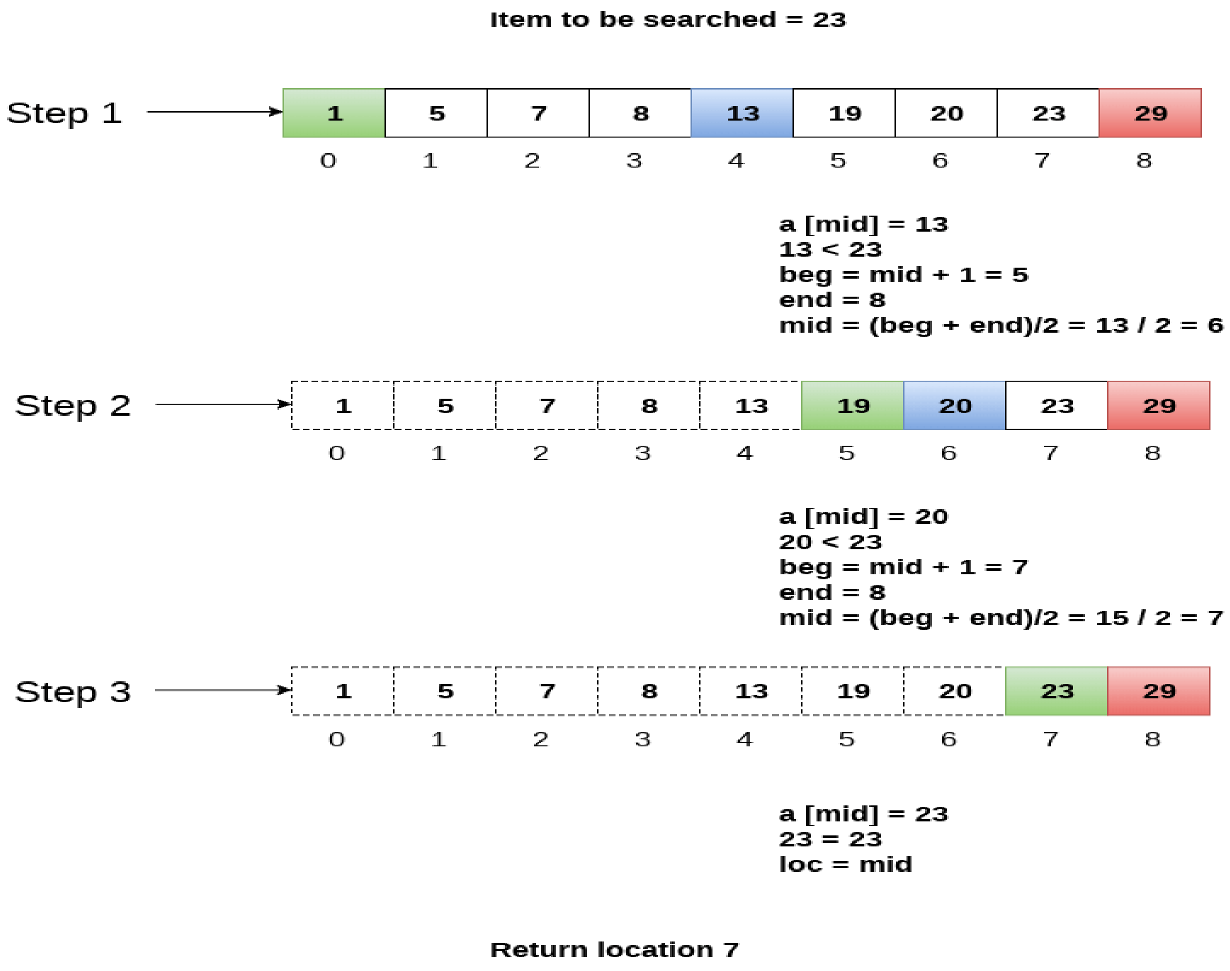
in Second step:

1. Beg = mid +1 = 5
2. End = 8
3. mid = 13/2 = 6
4. a[mid] = a[6] = 20 < 23, therefore;

in third step:

1. beg = mid + 1 = 7
2. End = 8
3. mid = 15/2 = 7
4. a[mid] = a[7]
5. a[7] = 23 = item;
6. therefore, set location = mid;
7. The location of the item will be 7.

Example:



Source code:

```
import java.util.*;
public class BinarySearch {
public static void main(String[] args) {
    int[] arr = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};
    int item, location = -1;
    System.out.println("Enter the item which you want to search");
    Scanner sc = new Scanner(System.in);
    item = sc.nextInt();
    location = binarySearch(arr,0,9,item);
    if(location != -1)
    System.out.println("the location of the item is "+location);
    else
        System.out.println("Item not found");
    }
public static int binarySearch(int[] a, int beg, int end, int item)
{
    int mid;
    if(end >= beg)
    {
        mid = (beg + end)/2;
        if(a[mid] == item)
        {
            return mid+1;
        }
        else if(a[mid] < item)
        {
            return binarySearch(a,mid+1,end,item);
        }
        else
        {
            return binarySearch(a,beg,mid-1,item);
        }

```

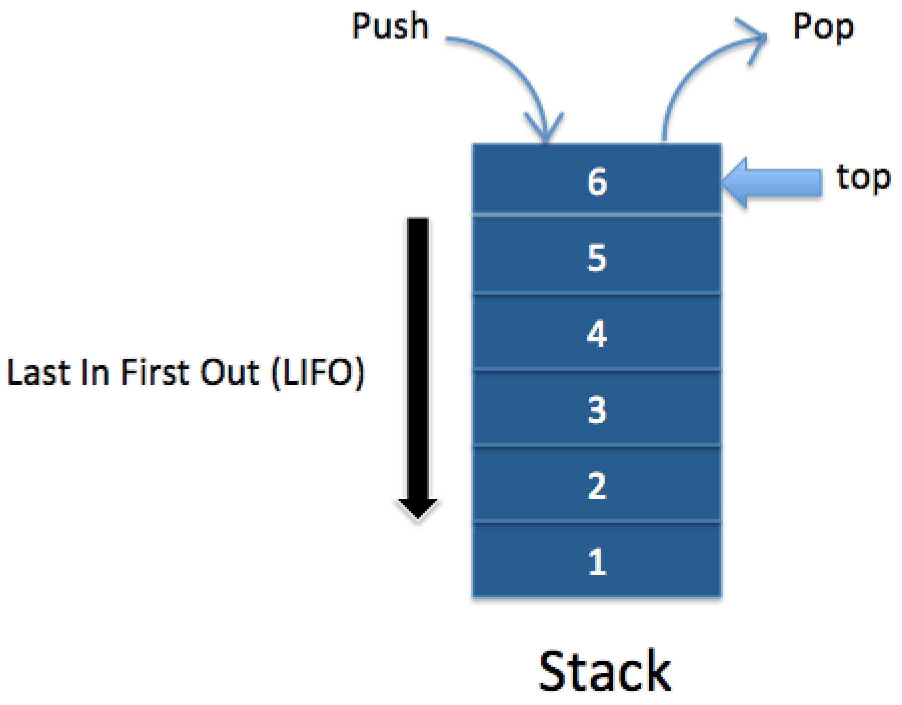
```
            }
            return -1;
        }
    }
```

Output:

```
Enter the item which you want to search
45
the location of the item is 5
```

Stack

Stack is abstract data type which depicts Last in first out (LIFO) behavior.



Stack implementation using Array

```
/**
 *
 */
public class MyStack {
    int size;
    int arr[];
    int top;

    MyStack(int size) {
        this.size = size;
        this.arr = new int[size];
        this.top = -1;
    }
```

```
public void push(int element) {
    if (!isFull()) {
        top++;
        arr[top] = element;
        System.out.println("Pushed element:" + element);
    } else {
        System.out.println("Stack is full !");
    }
}

public int pop() {
    if (!isEmpty()) {
        int topElement = top;
s        top--;
        System.out.println("Popped element :" + arr[topElement]);
        return arr[topElement];

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

public int peek() {
    if(!this.isEmpty())
        return arr[top];
    else
    {
        System.out.println("Stack is Empty");
        return -1;
    }
}

public boolean isEmpty() {
    return (top == -1);
}

public boolean isFull() {
    return (size - 1 == top);
}

public static void main(String[] args) {
    MyStack myStack = new MyStack(5);
    myStack.pop();
    System.out.println("=================");
    myStack.push(100);
    myStack.push(90);
    myStack.push(10);
    myStack.push(50);
    System.out.println("=================");
    myStack.pop();
    myStack.pop();
```

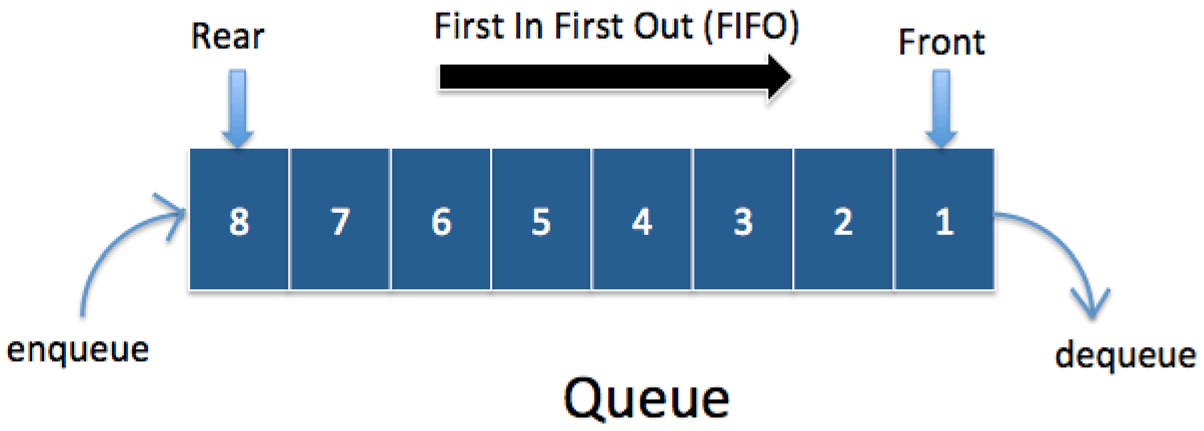
```
        myStack.pop();
        System.out.println("=================");
    }
}
```

Output:

```
Stack is empty !
=================
Pushed element:100
Pushed element:90
Pushed element:10
Pushed element:50
=================
Popped element :50
Popped element :10
Popped element :90
=================
```

Queue

Queue is abstract data type which depicts First in first out (FIFO) behavior.



Queue implementation using array

```
package org.arpit.java2blog.datastructures;

public class MyQueue {

    private int capacity;
    int queueArr[];
    int front;
    int rear;
    int currentSize = 0;

    public MyQueue(int size) {
        this.capacity = size;
        front = 0;
        rear = -1;
        queueArr = new int[this.capacity];
    }

    /**
     * Method for adding element in the queue
     *
     * @param data
     */
    public void enqueue(int data) {
        if (isFull()) {
```

```
        System.out.println("Queue is full!! Can not add more elements");
    } else {
        rear++;
        if (rear == capacity - 1) {
            rear = 0;
        }
        queueArr[rear] = data;
        currentSize++;
        System.out.println(data + " added to the queue");
    }
}

/**
 * This method removes an element from the front of the queue
 */
public void dequeue() {
    if (isEmpty()) {
        System.out.println("Queue is empty!! Can not dequeue element");
    } else {
        front++;
        if (front == capacity - 1) {
            System.out.println(queueArr[front - 1] + " removed from the queue");
            front = 0;
        } else {
            System.out.println(queueArr[front - 1] + " removed from the queue");
        }

        currentSize--;
    }
}

/**
 * Method for checking if Queue is full
 *
 * @return
 */
public boolean isFull() {
    if (currentSize == capacity) {
        return true;
    }
    return false;
}

/**
 * Method for checking if Queue is empty
 *
 * @return
 */
public boolean isEmpty() {

    if (currentSize == 0) {
        return true;
    }

```

```
        return false;
    }

    public static void main(String a[]) {

        MyQueue myQueue = new MyQueue(6);
        myQueue.enqueue(1);
        myQueue.dequeue();
        myQueue.enqueue(30);
        myQueue.enqueue(44);
        myQueue.enqueue(32);
        myQueue.dequeue();
        myQueue.enqueue(98);
        myQueue.dequeue();
        myQueue.enqueue(70);
        myQueue.enqueue(22);
        myQueue.dequeue();
        myQueue.enqueue(67);
        myQueue.enqueue(23);
    }
}
```

Output:

```
1 added to the queue
1 removed from the queue
30 added to the queue
```

```
44 added to the queue
32 added to the queue
30 removed from the queue
98 added to the queue
44 removed from the queue
70 added to the queue
22 added to the queue
32 removed from the queue
67 added to the queue
23 added to the queue
```