DSA using Java - Quick Guide

DSA using Java - Overview

What is a Data Structure?

Data Structure is a way to organized data in such a way that it can be used efficiently. Following terms are foundation terms of a data structure.

- **Interface** – Each data strucure has an interface. Interface represents the set of operations that a datastructure supports.An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.
- **Implementation** – Implementation provides the internal representation of a data structure. Implementation also provides the defination of the alogrithms used in the opreations of the data structure.

Characteristics of a Data Structure

- **Correctness** – Data Structure implementation should implement its interface correctly.
- **Time Complexity** – Running time or execution time of operations of data structure must be as small as possible.
- **Space Complexity** – Memory usage of a data structure operation should be as little as possible.

Need for Data Structure

As applications are getting complex and data rich, there are three common problems applications face now-a-days.

- **Data Search** – Consider an inventory of 1 million(10^6) items of a sto... If application is to search an item. It has to search item in 1 million(10^6)

every time slowing down the search. As data grows, search will become slower.

- **Processor speed** − Processor speed although being very high, falls limited if data grows to billon records.
- **Multiple requests** − As thousands of users can search data simultaneously on a web server,even very fast server fails while searching the data.

To solve above problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be search and required data can be searched almost instantly.

Execution Time Cases

There are three cases which are usual used to compare various data structure's execution time in relative manner.

- **Worst Case** − This is the scenario where a particular data structure operation takes maximum time it can take. If a operation's worst case time is $f(n)$ then this operation will not take time more than $f(n)$ time where $f(n)$ represents function of n.
- **Average Case** − This is the scenario depicting the average execution time of an operation of a data structure. If a operation takes $f(n)$ time in execution then m operations will take $mf(n)$ time.
- **Best Case** − This is the scenario depicting the least possible execution time of an operation of a data structure. If a operation takes $f(n)$ time in execution then actual operation may take time as random number which would be maximum as $f(n)$.

DSA using Java - Environment Setup

Local Environment Setup

If you are still willing to setup your environment for Java programming language, then this section guides you on how to download and set up Java on your machine. Please follow the following steps to set up the environment.

Java SE is freely available from the link Download Java. So you download a version based on your operating system.

Follow the instructions to download java and run the **.exe** to install Java on your machine. Once you installed Java on your machine, you would need to set environment variables to point to correct installation directories:

Setting up the path for windows 2000/XP

Assuming you have installed Java in c:\Program Files\java\jdk directory:

- Right-click on 'My Computer' and select 'Properties'.
- Click on the 'Environment variables' button under the 'Advanced' tab.
- Now alter the 'Path' variable so that it also contains the path to the Java executable. Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

Setting up the path for windows 95/98/ME

Assuming you have installed Java in c:\Program Files\java\jdk directory –

- Edit the 'C:\autoexec.bat' file and add the following line at the end: 'SET PATH=%PATH%;C:\Program Files\java\jdk\bin'

Setting up the path for Linux, UNIX, Solaris, FreeBSD:

Environment variable PATH should be set to point to where the java binaries have been installed. Refer to your shell documentation if you have trouble doing this.

Example, if you use bash as your shell, then you would add the following line to the end of your '.bashrc: export PATH=/path/to/java:$PATH'

Popular Java Editors

To write your java programs you will need a text editor. There are even more sophisticated IDE available in the market. But for now, you can consider one of the following:

- **Notepad –** On Windows machine you can use any simple text editor like Notepad (Recommended for this tutorial), TextPad.

- **Netbeans –**is a Java IDE that is open source and free which can be downloaded from https://www.netbeans.org/index.html.
- **Eclipse –** is also a java IDE developed by the eclipse open source community and can be downloaded from https://www.eclipse.org/.

What is Next ?

Next chapter will teach you how to write and run your first java program and some of the important basic syntaxes in java needed for developing applications.

DSA using Java - Algorithms

Algorithm concept

Algorithm is a step by step procedure, which defines a set of instructions to be executed in certain order to get the desired output. In term of data structures, following are the categories of algorithms.

- **Search** – Algorithms to search an item in a datastrucure.
- **Sort** – Algorithms to sort items in certain order
- **Insert** – Algorithm to insert item in a datastructure
- **Update** – Algorithm to update an existing item in a data structure
- **Delete** – Algorithm to delete an existing item from a data structure

Algorithm analysis

Algorithm analysis deals with the execution time or running time of various operations of a data structure. Running time of an operation can be defined as no. of computer instructions executed per operation. As exact running time of any operation varies from one computer to another computer, we usually analyze the running time of any operation as some function of n, where n is the no. of items processed in that operation in a datastructure.

Asymptotic analysis

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, running time of one operation is

computed as f(n) and of another operation as g(n^2). Which means first operation running time will increase linearly with the increase in n and running time of second operation will increase exponentially when n increases. Similarly the running time of both operations will be nearly same if n is significantly small.

Asymptotic Notations

Following are commonly used asymptotic notations used in calculating running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

The O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or longest amount of time an algorithm can possibly take to complete. For example, for a function f(n)

$O(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq c.f(n) \text{ for all } n > n_0. \}$

Big Oh notation is used to simplify functions. For example, we can replace a specific functional equation 7nlogn + n - 1 with O(f(nlogn)). Consider the scenario as follows:

$$7nlogn + n - 1 \leq 7nlogn + n$$

$$7nlogn + n - 1 \leq 7nlogn + nlogn$$

$$\text{for } n \geq 2 \text{ so that } logn \geq 1$$

$$7nlogn + n - 1 \leq 8nlogn$$

It demonstrates that f(n) = 7nlogn + n - 1 is within the range of outout of O(nlogn) using constants c = 8 and $n_0 = 2$.

Omega Notation, Ω

The Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or best amount of time an algorithm can possibly take to complete.

For example, for a function f(n)

$\Omega(f(n)) \geq \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq c.f(n) \text{ for all } n > n_0. \}$

Theta Notation, θ

The θ(n) is the formal way to express both the lower bound and upper bound of an algorithm's running time. It is represented as following.

$\theta(f(n)) = \{ g(n) \text{ if and only if } g(n) = O(f(n)) \text{ and } g(n) = \Omega(f(n)) \text{ for all } n > n_0. \}$

DSA using Java - Data Structures

Data Structure is a way to organized data in such a way that it can be used efficiently. Following terms are basic terms of a data structure.

Data Definition

Data Definition defines a particular data with following characteristics.

- Atomic − Defition should define a single concept
- Traceable − Definition should be be able to be mapped to some data element.
- Accurate − Definition should be unambiguous.
- Clear and Concise − Definition should be understandable.

Data Object

Data Object represents an object having a data.

Data Type

Data type is way to classify various types of data such as integer, string etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. Data type of two types −

- Built-in Data Type
- Derived Data Type

Built-in Data Type

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provides following built-in data types.

- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

Derived Data Type

Those data types which are implementation independent as they can be implemented in one or other way are known as derived data types. These data types are normally built by combination of primary or built-in data types and associated operations on them. For example –

- List
- Array
- Stack
- Queue

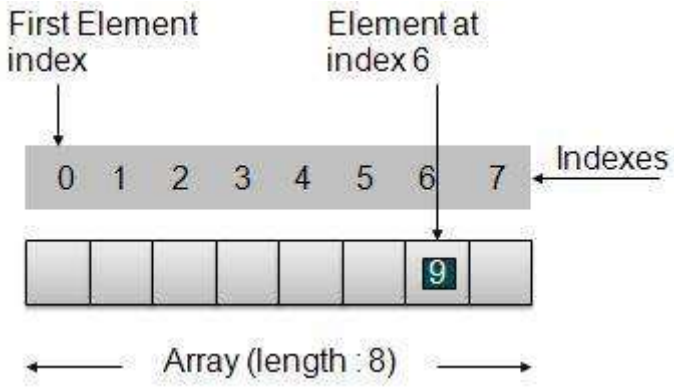
DSA using Java - Arrays

Array Basics

Array is a container which can hold fix number of items and these items should be of same type. Most of the datastructure make use of array to implement their algorithms. Following are important terms to understand the concepts of Array

- **Element** – Each item stored in an array is called an element.
- **Index** – Each location of an element in an array has a numerical index which is used to identify the element.

Array Representation



As per above shown illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 8 which means it can store 8 elements.
- Each element can be accessed via its index. For example, we can fetch element at index 6 as 9.

Basic Operations

Following are the basic operations supported by an array.

- **Insertion** – add an element at given index.
- **Deletion** – delete an element at given index.
- **Search** – search an element using given index or by value.
- **Update** – update an element at given index.

In java, when an array is initialized with size, then it assigns defaults values to its elements in following order.

Data Type	Default Value
byte	0
short	0
int	0
long	0L

float	0.0f
double	0.0d
char	'\u0000'
boolean	false
Object	null

Demo

```
package com.tutorialspoint.array;

public class ArrayDemo {
   public static void main(String[] args){

      // Declare an array
      int intArray[];

      // Initialize an array of 8 int
      // set aside memory of 8 int
      intArray = new int[8];

      System.out.println("Array before adding data.");

      // Display elements of an array.
      display(intArray);

      // Operation : Insertion
      // Add elements in the array
      for(int i = 0; i< intArray.length; i++)
      {
         // place value of i at index i.
         System.out.println("Adding "+i+" at index "+i);
         intArray[i] = i;
      }
      System.out.println();

      System.out.println("Array after adding data.");
      display(intArray);
```

```
      // Operation : Insertion
      // Element at any location can be updated directly
      int index = 5;
      intArray[index] = 10;

      System.out.println("Array after updating element at index " + index);
      display(intArray);

      // Operation : Search using index
      // Search an element using index.
      System.out.println("Data at index " + index + ": "+ intArray[index]);

      // Operation : Search using value
      // Search an element using value.
      int value = 4;
      for(int i = 0; i< intArray.length; i++)
      {
         if(intArray[i] == value ){
            System.out.println(value + " Found at index "+i);
            break;
         }
      }
      System.out.println("Data at index " + index + ": "+ intArray[index]);
   }

   private static void display(int[] intArray){
      System.out.print("Array : [");
      for(int i = 0; i< intArray.length; i++)
      {
         // display value of element at index i.
         System.out.print(" "+intArray[i]);
      }
      System.out.println(" ]");
      System.out.println();
   }
}
```

If we compile and run the above program then it would produce following result –

```
Array before adding data.
Array : [ 0 0 0 0 0 0 0 0 ]
```

```
Adding 0 at index 0
Adding 1 at index 1
Adding 2 at index 2
Adding 3 at index 3
Adding 4 at index 4
Adding 5 at index 5
Adding 6 at index 6
Adding 7 at index 7

Array after adding data.
Array : [ 0 1 2 3 4 5 6 7 ]

Array after updating element at index 5
Array : [ 0 1 2 3 4 10 6 7 ]

Data at index 5: 10
4 Found at index: 4
```

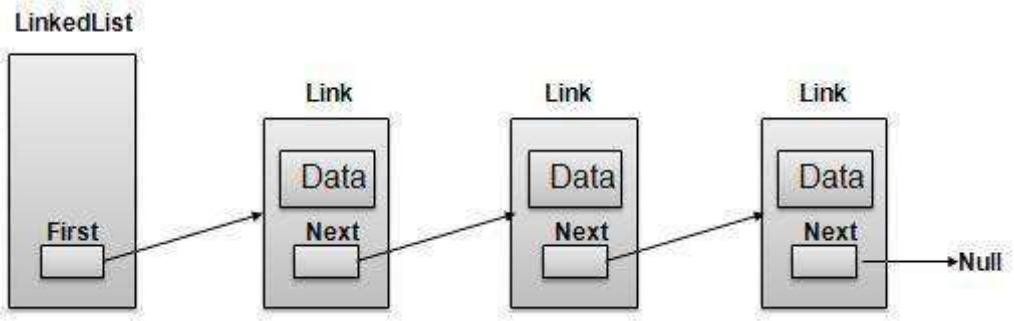
DSA using Java - Linked List

Linked List Basics

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list the second most used data structure after array. Following are important terms to understand the concepts of Linked List.

- **Link** − Each Link of a linked list can store a data called an element.
- **Next** − Each Link of a linked list contain a link to next link called Next.
- **LinkedList** − A LinkedList contains the connection link to the first Link called First.

Linked List Representation



As per above shown illustration, following are the important points to be considered.

- LinkedList contains an link element called first.
- Each Link carries a data field(s) and a Link Field called next.
- Each Link is linked with its next link using its next link.
- Last Link carries a Link as null to mark the end of the list.

Types of Linked List

Following are the various flavours of linked list.

- **Simple Linked List** – Item Navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward way.
- **Circular Linked List** – Last item contains link of the first element as next and and first element has link to last element as prev.

Basic Operations

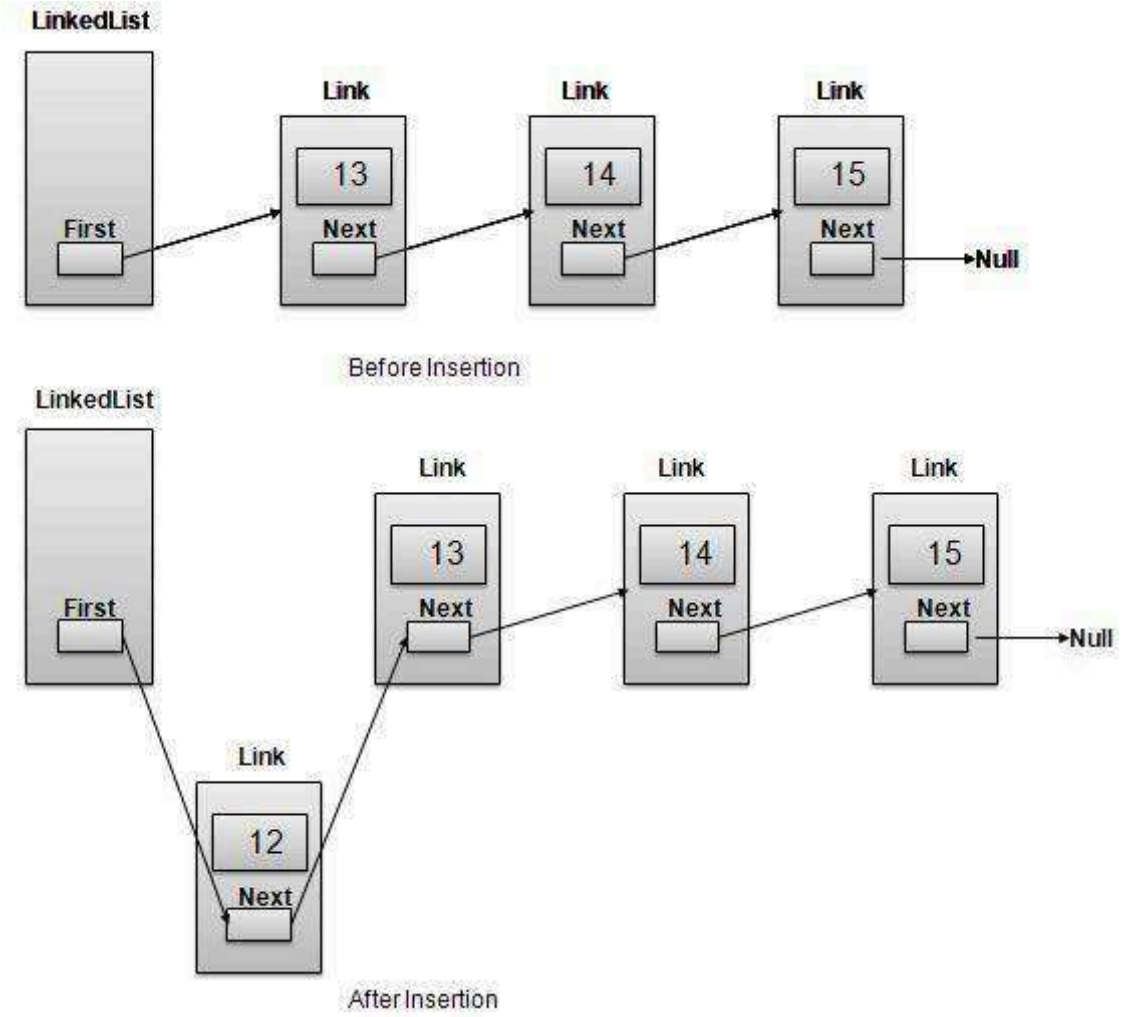
Following are the basic operations supported by a list.

- **Insertion** – add an element at the beginning of the list.
- **Deletion** – delete an element at the beginning of the list.
- **Display** – displaying complete list.
- **Search** – search an element using given key.
- **Delete** – delete an element using given key.

Insertion Operation

Insertion is a three step process:

- Create a new Link with provided data.
- Point New Link to old First Link.
- Point First Link to this New Link.

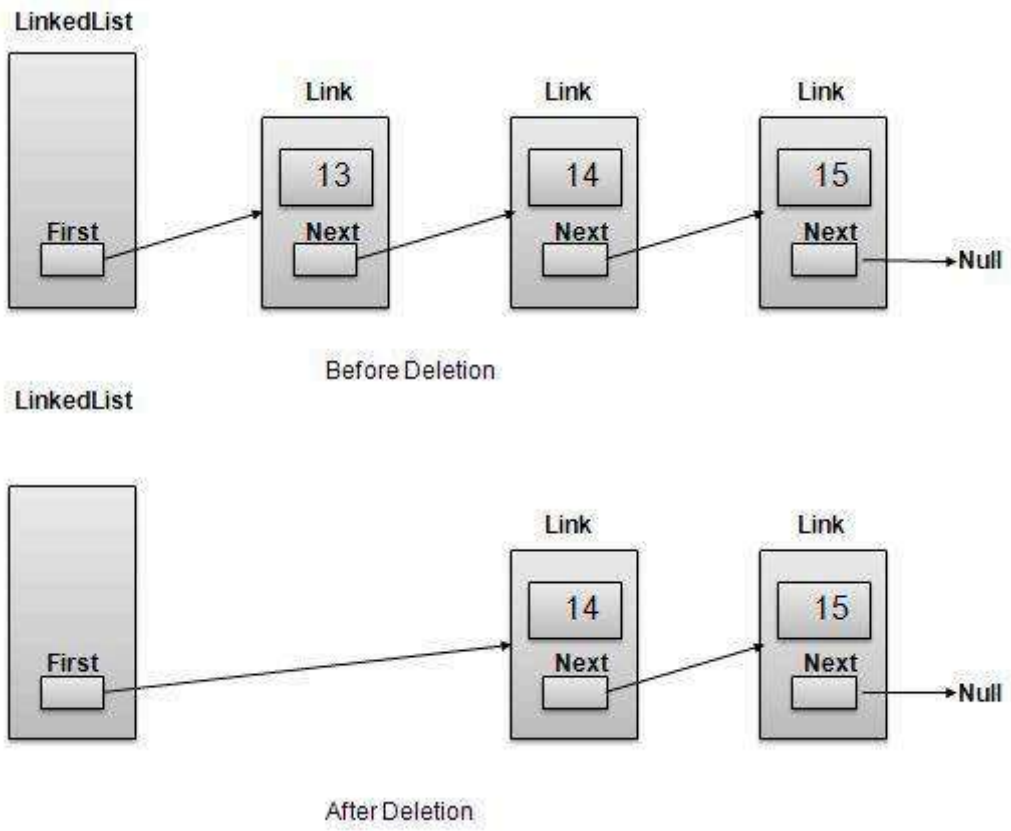



```
//insert link at the first location
public void insertFirst(int key, int data){
   //create a link
   Link link = new Link(key,data);
   //point it to old first node
   link.next = first;
   //point first to new first node
   first = link;
}
```

Deletion Operation

Deletion is a two step process:

- Get the Link pointed by First Link as Temp Link.
- Point First Link to Temp Link's Next Link.



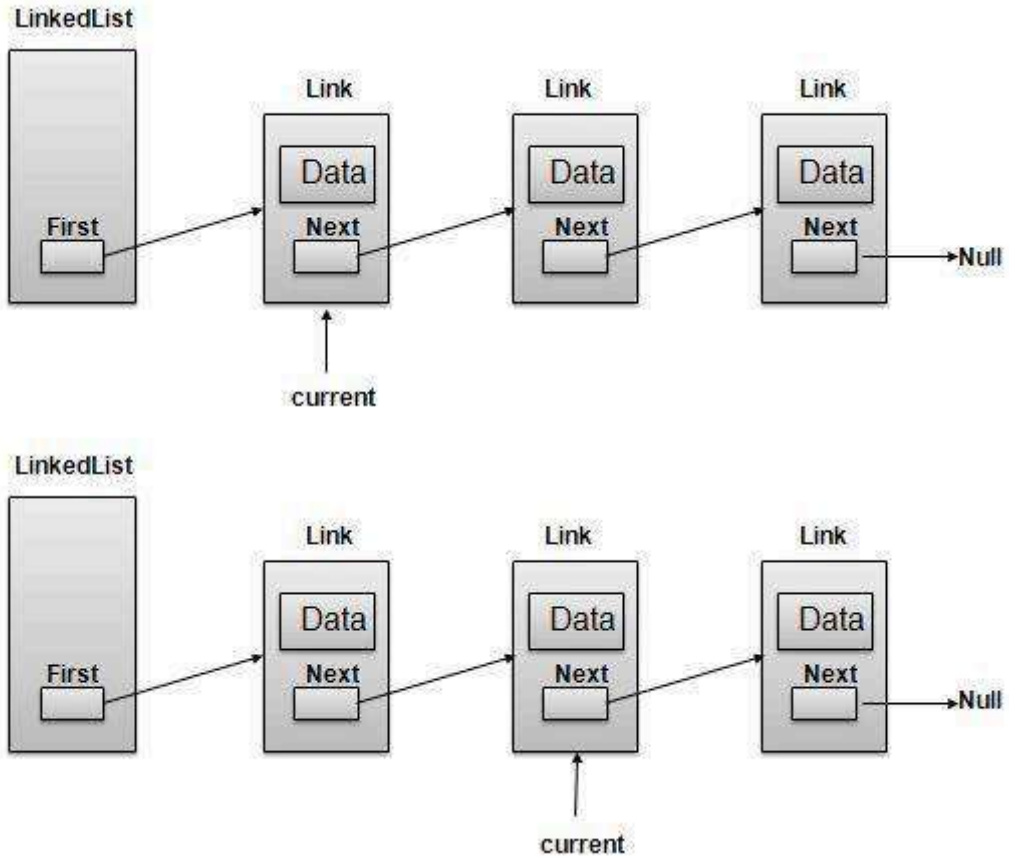

```
//delete first item
public Link deleteFirst(){
   //save reference to first link
   Link tempLink = first;
   //mark next to first link as first
   first = first.next;
   //return the deleted link
   return tempLink;
}
```

Navigation Operation

Navigation is a recursive step process and is basis of many operations like search, delete etc.:

- Get the Link pointed by First Link as Current Link.
- Check if Current Link is not null and display it.

- Point Current Link to Next Link of Current Link and move to above step.



Note

```
//display the list
public void display(){
   //start from the beginning
   Link current = first;
   //navigate till the end of the list
   System.out.print("[ ");
   while(current != null){
      //print data
      current.display();
      //move to next item
      current = current.next;
      System.out.print(" ");
   }
   System.out.print(" ]");
}
```

Advanced Operations

Following are the advanced operations specified for a list.

- **Sort** – sorting a list based on a particular order.
- **Reverse** – reversing a linked list.
- **Concatenate** – concatenate two lists.

Sort Operation

We've used bubble sort to sort a list.

```
public void sort(){

   int i, j, k, tempKey, tempData ;
   Link current,next;
   int size = length();
   k = size ;
   for ( i = 0 ; i < size - 1 ; i++, k-- ) {
      current = first ;
      next = first.next ;
      for ( j = 1 ; j < k ; j++ ) {
         if ( current.data > next.data ) {
            tempData = current.data ;
            current.data = next.data;
            next.data = tempData ;

            tempKey = current.key;
            current.key = next.key;
            next.key = tempKey;
         }
         current = current.next;
         next = next.next;
      }
   }
}
```

Reverse Operation

Following code demonstrate reversing a single linked list.

```
public LinkedList reverse() {
   LinkedList reversedlist = new LinkedList();
   Link nextLink = null;
   reversedlist.insertFirst(first.key, first.data);

   Link currentLink = first;
   // Until no more data in list,
   // insert current link before first and move ahead.
   while(currentLink.next != null){
      nextLink = currentLink.next;
      // Insert at start of new list.
      reversedlist.insertFirst(nextLink.key, nextLink.data);
      //advance to next node
      currentLink = currentLink.next;
   }
   return reversedlist;
}
```

Concatenate Operation

Following code demonstrate reversing a single linked list.

```
public void concatenate(LinkedList list){
   if(first == null){
      first = list.first;
   }
   if(list.first == null){
      return;
   }
   Link temp = first;
   while(temp.next !=null) {
      temp = temp.next;
   }
   temp.next = list.first;
}
```

Demo

Link.java

```
package com.tutorialspoint.list;

public class Link {
   public int key;
   public int data;
   public Link next;

   public Link(int key, int data){
      this.key = key;
      this.data = data;
   }

   public void display(){
      System.out.print("{"+key+","+data+"}");
   }
}
```

LinkedList.java

```
package com.tutorialspoint.list;

public class LinkedList {
   //this link always point to first Link
   //in the Linked List
   private Link first;

   // create an empty linked list
   public LinkedList(){
      first = null;
   }

   //insert link at the first location
   public void insertFirst(int key, int data){
      //create a link
      Link link = new Link(key,data);
      //point it to old first node
      link.next = first;
      //point first to new first node
      first = link;
   }
```

```
   //delete first item
   public Link deleteFirst(){
      //save reference to first link
      Link tempLink = first;
      //mark next to first link as first
      first = first.next;
      //return the deleted link
      return tempLink;
   }

   //display the list
   public void display(){
      //start from the beginning
      Link current = first;
      //navigate till the end of the list
      System.out.print("[ ");
      while(current != null){
         //print data
         current.display();
         //move to next item
         current = current.next;
         System.out.print(" ");
      }
      System.out.print(" ]");
   }

   //find a link with given key
   public Link find(int key){
      //start from the first link
      Link current = first;

      //if list is empty
      if(first == null){
         return null;
      }
      //navigate through list
      while(current.key != key){
         //if it is last node
         if(current.next == null){
            return null;
         }else{
            //go to next link
```

```
            current = current.next;
         }
      }
      //if data found, return the current Link
      return current;
   }

   //delete a link with given key
   public Link delete(int key){
      //start from the first link
      Link current = first;
      Link previous = null;
      //if list is empty
      if(first == null){
         return null;
      }

      //navigate through list
      while(current.key != key){
         //if it is last node
         if(current.next == null){
            return null;
         }else{
            //store reference to current link
            previous = current;
            //move to next link
            current = current.next;
         }
      }

      //found a match, update the link
      if(current == first) {
         //change first to point to next link
         first = first.next;
      }else {
         //bypass the current link
         previous.next = current.next;
      }
      return current;
   }
```

```
   //is list empty
   public boolean isEmpty(){
      return first == null;
   }

   public int length(){
      int length = 0;
      for(Link current = first; current!=null;
         current = current.next){
         length++;
      }
      return length;
   }

   public void sort(){
      int i, j, k, tempKey, tempData ;
      Link current,next;
      int size = length();
      k = size ;
      for ( i = 0 ; i < size - 1 ; i++, k-- ) {
         current = first ;
         next = first.next ;
         for ( j = 1 ; j < k ; j++ ) {
            if ( current.data > next.data ) {
               tempData = current.data ;
               current.data = next.data;
               next.data = tempData ;

                   tempKey = current.key;
                   current.key = next.key;
                   next.key = tempKey;
            }
            current = current.next;
          next = next.next;
         }
      }
   }

   public LinkedList reverse() {
      LinkedList reversedlist = new LinkedList();
      Link nextLink = null;
      reversedlist.insertFirst(first.key, first.data);
```

```
      Link currentLink = first;
      // Until no more data in list,
      // insert current link before first and move ahead.
      while(currentLink.next != null){
         nextLink = currentLink.next;
         // Insert at start of new list.
         reversedlist.insertFirst(nextLink.key, nextLink.data);
         //advance to next node
         currentLink = currentLink.next;
      }
      return reversedlist;
   }

   public void concatenate(LinkedList list){
      if(first == null){
         first = list.first;
      }
      if(list.first == null){
         return;
      }
      Link temp = first;

      while(temp.next !=null) {
         temp = temp.next;
      }
      temp.next = list.first;
   }
}
```

LinkedListDemo.java

```
package com.tutorialspoint.list;

public class LinkedListDemo {
   public static void main(String args[]){
      LinkedList list = new LinkedList();

      list.insertFirst(1, 10);
      list.insertFirst(2, 20);
      list.insertFirst(3, 30);
      list.insertFirst(4, 1);
```

```
      list.insertFirst(5, 40);
      list.insertFirst(6, 56);

      System.out.print("\nOriginal List: ");
      list.display();
      System.out.println("");
      while(!list.isEmpty()){
         Link temp = list.deleteFirst();
         System.out.print("Deleted value:");
         temp.display();
         System.out.println("");
      }
      System.out.print("List after deleting all items: ");
      list.display();
      System.out.println("");
      list.insertFirst(1, 10);
      list.insertFirst(2, 20);
      list.insertFirst(3, 30);
      list.insertFirst(4, 1);
      list.insertFirst(5, 40);
      list.insertFirst(6, 56);

      System.out.print("Restored List: ");
      list.display();
      System.out.println("");

      Link foundLink = list.find(4);
      if(foundLink != null){
        System.out.print("Element found: ");
         foundLink.display();
         System.out.println("");
      }else{
         System.out.println("Element not found.");
      }

      list.delete(4);
      System.out.print("List after deleting an item: ");
      list.display();
      System.out.println("");
      foundLink = list.find(4);
      if(foundLink != null){
         System.out.print("Element found: ");
```

```
            foundLink.display();
            System.out.println("");
         }else{
            System.out.print("Element not found. {4,1}");
         }
         System.out.println("");
         list.sort();
         System.out.print("List after sorting the data: ");
         list.display();
         System.out.println("");
         System.out.print("Reverse of the list: ");
         LinkedList list1 = list.reverse();
         list1.display();
         System.out.println("");

         LinkedList list2 = new LinkedList();

         list2.insertFirst(9, 50);
         list2.insertFirst(8, 40);
         list2.insertFirst(7, 20);

         list.concatenate(list2);
         System.out.print("List after concatenation: ");
         list.display();
         System.out.println("");
      }
   }
```

If we compile and run the above program then it would produce following result:

```
Original List: [ {6,56} {5,40} {4,1} {3,30} {2,20} {1,10} ]
Deleted value:{6,56}
Deleted value:{5,40}
Deleted value:{4,1}
Deleted value:{3,30}
Deleted value:{2,20}
Deleted value:{1,10}
List after deleting all items: [ ]
Restored List: [ {6,56} {5,40} {4,1} {3,30} {2,20} {1,10} ]
Element found: {4,1}
List after deleting an item: [ {6,56} {5,40} {3,30} {2,20} {1,10} ]
Element not found. {4,1}
```

```
List after sorting the data: [ {1,10} {2,20} {3,30} {5,40} {6,56} ]
Reverse of the list: [ {6,56} {5,40} {3,30} {2,20} {1,10} ]
List after concatenation: [ {1,10} {2,20} {3,30} {5,40} {6,56} {7,20} {8,40} {9,50}
```

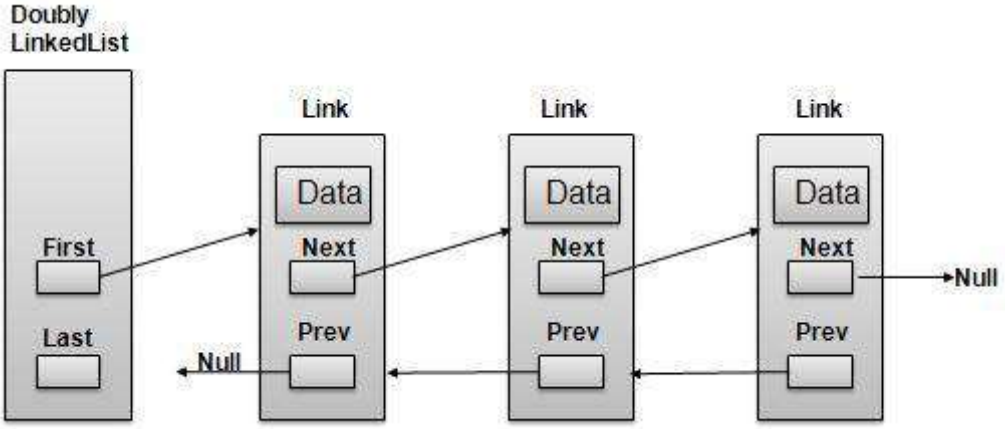
DSA using Java - Doubly Linked List

Doubly Linked List Basics

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways either forward and backward easily as compared to Single Linked List. Following are important terms to understand the concepts of doubly Linked List

- **Link** − Each Link of a linked list can store a data called an element.
- **Next** − Each Link of a linked list contain a link to next link called Next.
- **Prev** − Each Link of a linked list contain a link to previous link called Prev.
- **LinkedList** − A LinkedList contains the connection link to the first Link called First and to the last link called Last.

Doubly Linked List Representation



As per above shown illustration, following are the important points to be considered.

- Doubly LinkedList contains an link element called first and last.
- Each Link carries a data field(s) and a Link Field called next.
- Each Link is linked with its next link using its next link.
- Each Link is linked with its previous link using its prev link.
- Last Link carries a Link as null to mark the end of the list.

Basic Operations

Following are the basic operations supported by an list.

- **Insertion** – add an element at the beginning of the list.
- **Deletion** – delete an element at the beginning of the list.
- **Insert Last** – add an element in the end of the list.
- **Delete Last** – delete an element from the end of the list.
- **Insert After** – add an element after an item of the list.
- **Delete** – delete an element from the list using key.
- **Display forward** – displaying complete list in forward manner.
- **Display backward** – displaying complete list in backward manner.

Insertion Operation

Following code demonstrate insertion operation at beginning in a doubly linked list.

```
//insert link at the first location
public void insertFirst(int key, int data){
   //create a link
   Link link = new Link(key,data);

   if(isEmpty()){
      //make it the last link
      last = link;
   }else {
      //update first prev link
      first.prev = link;
   }
```

```
   //point it to old first link
   link.next = first;
   //point first to new first link
   first = link;
}
```

Deletion Operation

Following code demonstrate deletion operation at beginning in a doubly linked list.

```
//delete link at the first location
public Link deleteFirst(){
   //save reference to first link
   Link tempLink = first;
   //if only one link
   if(first.next == null){
      last = null;
   }else {
      first.next.prev = null;
   }
   first = first.next;
   //return the deleted link
   return tempLink;
}
```

Insertion at End Operation

Following code demonstrate insertion operation at last position in a doubly linked list.

```
//insert link at the last location
public void insertLast(int key, int data){
   //create a link
   Link link = new Link(key,data);

   if(isEmpty()){
      //make it the last link
      last = link;
   }else {
      //make link a new last link
```

```
      last.next = link;
      //mark old last node as prev of new link
      link.prev = last;
   }

   //point last to new last node
   last = link;
}
```

Demo

Link.java

```
package com.tutorialspoint.list;

public class Link {
   public int key;
   public int data;
   public Link next;
   public Link prev;

   public Link(int key, int data){
      this.key = key;
      this.data = data;
   }

   public void display(){
      System.out.print("{"+key+","+data+"}");
   }
}
```

DoublyLinkedList.java

```
package com.tutorialspoint.list;

public class DoublyLinkedList {

   //this link always point to first Link
   private Link first;
   //this link always point to last Link
```

```
   private Link last;

   // create an empty linked list
   public DoublyLinkedList(){
      first = null;
      last = null;
   }

   //is list empty
   public boolean isEmpty(){
      return first == null;
   }

   //insert link at the first location
   public void insertFirst(int key, int data){
      //create a link
      Link link = new Link(key,data);

      if(isEmpty()){
         //make it the last link
         last = link;
      }else {
         //update first prev link
         first.prev = link;
      }

      //point it to old first link
      link.next = first;
      //point first to new first link
      first = link;
   }

   //insert link at the last location
   public void insertLast(int key, int data){
      //create a link
      Link link = new Link(key,data);

      if(isEmpty()){
         //make it the last link
         last = link;
      }else {
         //make link a new last link
```

```
         last.next = link;
         //mark old last node as prev of new link
         link.prev = last;
      }

      //point last to new last node
      last = link;
   }

   //delete link at the first location
   public Link deleteFirst(){
      //save reference to first link
      Link tempLink = first;
      //if only one link
      if(first.next == null){
         last = null;
      }else {
         first.next.prev = null;
      }
      first = first.next;
      //return the deleted link
      return tempLink;
   }

   //delete link at the last location
   public Link deleteLast(){
      //save reference to last link
      Link tempLink = last;
      //if only one link
      if(first.next == null){
         first = null;
      }else {
         last.prev.next = null;
      }
      last = last.prev;
      //return the deleted link
      return tempLink;
   }

   //display the list in from first to last
   public void displayForward(){
      //start from the beginning
```

```
      Link current = first;
      //navigate till the end of the list
      System.out.print("[ ");
      while(current != null){
         //print data
         current.display();
         //move to next item
         current = current.next;
         System.out.print(" ");
      }
      System.out.print(" ]");
   }

   //display the list from last to first
   public void displayBackward(){
      //start from the last
      Link current = last;
      //navigate till the start of the list
      System.out.print("[ ");
      while(current != null){
         //print data
         current.display();
         //move to next item
         current = current.prev;
         System.out.print(" ");
      }
      System.out.print(" ]");
   }

   //delete a link with given key
   public Link delete(int key){
      //start from the first link
      Link current = first;
      //if list is empty
      if(first == null){
         return null;
      }

      //navigate through list
      while(current.key != key){
      //if it is last node
      if(current.next == null){
```

```
            return null;
         }else{
            //move to next link
            current = current.next;
         }
      }

      //found a match, update the link
      if(current == first) {
         //change first to point to next link
            first = current.next;
         }else {
            //bypass the current link
            current.prev.next = current.next;
         }

         if(current == last){
            //change last to point to prev link
            last = current.prev;
         }else {
            current.next.prev = current.prev;
         }
         return current;
      }

   public boolean insertAfter(int key, int newKey, int data){
      //start from the first link
      Link current = first;
      //if list is empty
      if(first == null){
         return false;
      }

      //navigate through list
      while(current.key != key){
         //if it is last node
         if(current.next == null){
            return false;
         }else{
            //move to next link
            current = current.next;
         }
```

```
      }

      Link newLink = new Link(newKey,data);
      if(current==last) {
         newLink.next = null;
         last = newLink;
      }
      else {
         newLink.next = current.next;
         current.next.prev = newLink;
      }
      newLink.prev = current;
      current.next = newLink;
      return true;
   }
}
```

DoublyLinkedListDemo.java

```
package com.tutorialspoint.list;

public class DoublyLinkedListDemo {
    public static void main(String args[]){
        DoublyLinkedList list = new DoublyLinkedList();

        list.insertFirst(1, 10);
        list.insertFirst(2, 20);
        list.insertFirst(3, 30);

        list.insertLast(4, 1);
        list.insertLast(5, 40);
        list.insertLast(6, 56);

        System.out.print("\nList (First to Last): ");
        list.displayForward();
        System.out.println("");
        System.out.print("\nList (Last to first): ");
        list.displayBackward();

        System.out.print("\nList , after deleting first record: ");
        list.deleteFirst();
```

```
        list.displayForward();

        System.out.print("\nList , after deleting last record: ");
        list.deleteLast();
        list.displayForward();

        System.out.print("\nList , insert after key(4) : ");
        list.insertAfter(4,7, 13);
        list.displayForward();

        System.out.print("\nList  , after delete key(4) : ");
        list.delete(4);
        list.displayForward();

    }
}
```

If we compile and run the above program then it would produce following result –

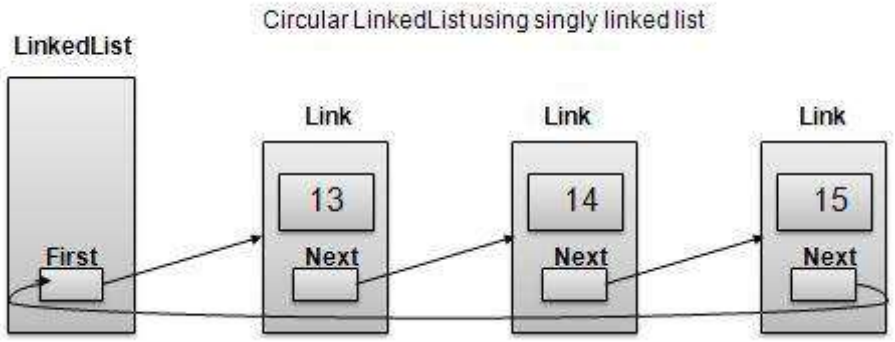
```
List (First to Last): [ {3,30} {2,20} {1,10} {4,1} {5,40} {6,56}  ]

List (Last to first): [ {6,56} {5,40} {4,1} {1,10} {2,20} {3,30}  ]
List (First to Last) after deleting first record: [ {2,20} {1,10} {4,1} {5,40} {6,56}  ]
List  (First to Last) after deleting last record: [ {2,20} {1,10} {4,1} {5,40}  ]
List  (First to Last) insert after key(4) : [ {2,20} {1,10} {4,1} {7,13} {5,40}  ]
List  (First to Last) after delete key(4) : [ {2,20} {1,10} {7,13} {5,40}  ]
```

DSA using Java - Circular Linked List

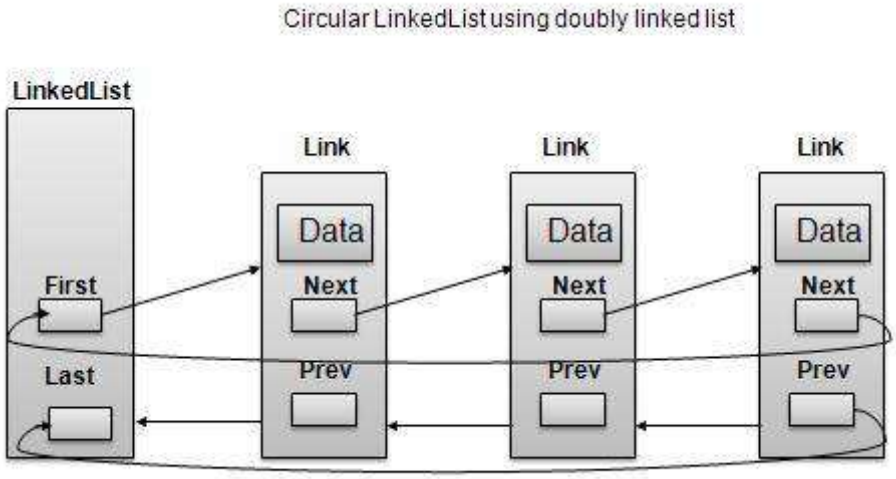
Circular Linked List Basics

Circular Linked List is a variation of Linked list in which first element points to last element and last element points to first element. Both Singly Linked List and Doubly Linked List can be made into as circular linked list

Singly Linked List as Circular



Doubly Linked List as Circular



As per above shown illustrations, following are the important points to be considered.

- Last Link'next points to first link of the list in both cases of singly as well as doubly linked list.
- First Link's prev points to the last of the list in case of doubly linked list.

Basic Operations

Following are the important operations supported by a circular list.

- **insert** – insert an element in the start of the list.
- **delete** – insert an element from the start of the list.
- **display** – display the list.

length Operation

Following code demonstrate insertion operation at in a circular linked list based on single linked list.

```
//insert link at the first location
public void insertFirst(int key, int data){
   //create a link
   Link link = new Link(key,data);
   if (isEmpty()) {
      first  = link;
      first.next = first;
   }
   else{
      //point it to old first node
      link.next = first;
      //point first to new first node
      first = link;
   }
}
```

Deletion Operation

Following code demonstrate deletion operation at in a circular linked list based on single linked list.

```
//delete link at the first location
public Link deleteFirst(){
   //save reference to first link
   Link tempLink = first;
   //if only one link
   if(first.next == null){
      last = null;
   }else {
      first.next.prev = null;
   }
   first = first.next;
   //return the deleted link
   return tempLink;
}
```

Display List Operation

Following code demonstrate display list operation in a circular linked list.

```
public void display(){
   //start from the beginning
   Link current = first;
   //navigate till the end of the list
   System.out.print("[ ");
   if(first != null){
      while(current.next != current){
         //print data
         current.display();
         //move to next item
         current = current.next;
         System.out.print(" ");
      }
   }
   System.out.print(" ]");
}
```

Demo

Link.java

```
package com.tutorialspoint.list;

public class CircularLinkedList {
   //this link always point to first Link
   private Link first;

   // create an empty linked list
   public CircularLinkedList(){
      first = null;
   }

   public boolean isEmpty(){
      return first == null;
   }
```

```
   public int length(){
      int length = 0;

      //if list is empty
      if(first == null){
         return 0;
      }

      Link current = first.next;

      while(current != first){
         length++;
         current = current.next;
      }
      return length;
   }

   //insert link at the first location
   public void insertFirst(int key, int data){
   //create a link
   Link link = new Link(key,data);
      if (isEmpty()) {
         first  = link;
         first.next = first;
      }
      else{
         //point it to old first node
         link.next = first;
         //point first to new first node
         first = link;
      }
   }

   //delete first item
   public Link deleteFirst(){
      //save reference to first link
      Link tempLink = first;
      if(first.next == first){
         first = null;
         return tempLink;
      }
```

```
      //mark next to first link as first
      first = first.next;
      //return the deleted link
      return tempLink;
   }

   public void display(){

      //start from the beginning
      Link current = first;
      //navigate till the end of the list
      System.out.print("[ ");
      if(first != null){
         while(current.next != current){
            //print data
            current.display();
            //move to next item
            current = current.next;
            System.out.print(" ");
         }
      }
      System.out.print(" ]");
   }
}
```

DoublyLinkedListDemo.java

```
package com.tutorialspoint.list;

public class CircularLinkedListDemo {
   public static void main(String args[]){
      CircularLinkedList list = new CircularLinkedList();

      list.insertFirst(1, 10);
      list.insertFirst(2, 20);
      list.insertFirst(3, 30);
      list.insertFirst(4, 1);
      list.insertFirst(5, 40);
      list.insertFirst(6, 56);

      System.out.print("\nOriginal List: ");
      list.display();
```

```
      System.out.println("");
      while(!list.isEmpty()){
         Link temp = list.deleteFirst();
         System.out.print("Deleted value:");
         temp.display();
         System.out.println("");
      }
      System.out.print("List after deleting all items: ");
      list.display();
      System.out.println("");
   }
}
```

If we compile and run the above program then it would produce following result –

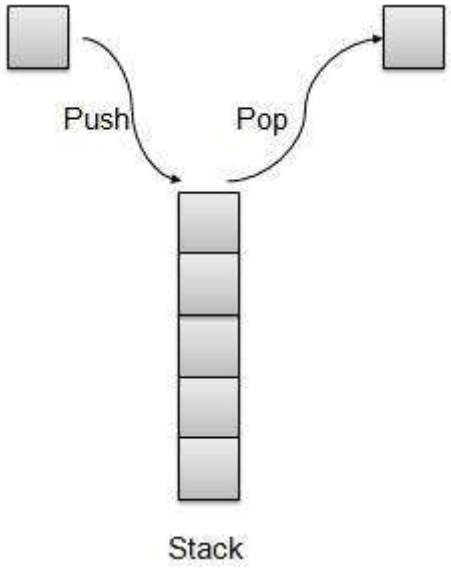
```
Original List: [ {6,56} {5,40} {4,1} {3,30} {2,20} ]
Deleted value:{6,56}
Deleted value:{5,40}
Deleted value:{4,1}
Deleted value:{3,30}
Deleted value:{2,20}
Deleted value:{1,10}
List after deleting all items: [ ]
```

DSA using Java - Stack

Overview

Stack is kind of data structure which allows operations on data only at one end. It allows access to the last inserted data only. Stack is also called LIFO (Last In First Out) data structure and Push and Pop operations are related in such a way that only last item pushed (added to stack) can be popped (removed from the stack).

Stack Representation



We're going to implement Stack using array in this article.

Basic Operations

Following are two primary operations of a stack which are following.

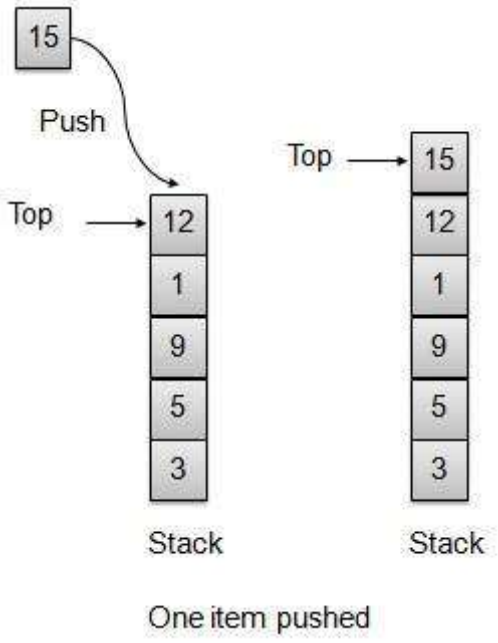
- **Push** – push an element at the top of the stack.
- **Pop** – pop an element from the top of the stack.

There is few more operations supported by stack which are following.

- **Peek** – get the top element of the stack.
- **isFull** – check if stack is full.
- **isEmpty** – check if stack is empty.

Push Operation

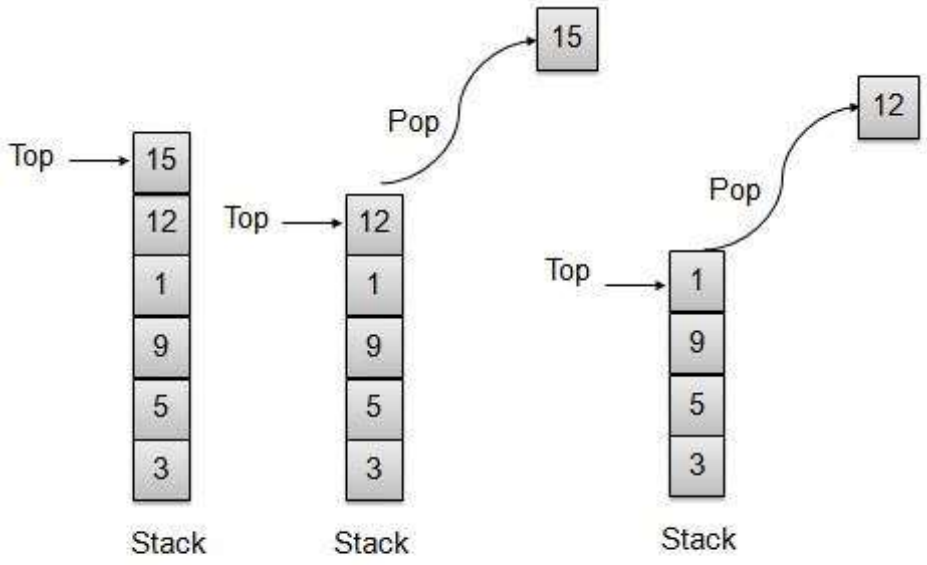
Whenever an element is pushed into stack, stack stores that element at the top of the storage and increments the top index for later use. If storage is full then an error message is usually shown.




```
// push item on the top of the stack
public void push(int data) {

   if(!isFull()){
      // increment top by 1 and insert data
      intArray[++top] = data;
   }else{
      System.out.println("Cannot add data. Stack is full.");
   }
}
```

Pop Operation

Whenever an element is to be popped from stack, stack retrives the element from the top of the storage and decrements the top index for later use.



Two Items popped

```
// pop item from the top of the stack
public int pop() {
   // retrieve data and decrement the top by 1
   return intArray[top--];
}
```

Stack Implementation

Stack.java

```
package com.tutorialspoint.datastructure;

public class Stack {
   private int size;           // size of the stack
   private int[] intArray;     // stack storage
   private int top;            // top of the stack

   // Constructor
   public Stack(int size){
      this.size = size;
      intArray = new int[size];   //initialize array
      top = -1;                   //stack is initially empty
   }

   // Operation : Push
   // push item on the top of the stack
```

```
   public void push(int data) {

      if(!isFull()){
         // increment top by 1 and insert data
         intArray[++top] = data;
      }else{
         System.out.println("Cannot add data. Stack is full.");
      }
   }

   // Operation : Pop
   // pop item from the top of the stack
   public int pop() {
      //retrieve data and decrement the top by 1
      return intArray[top--];
   }

   // Operation : Peek
   // view the data at top of the stack
   public int peek() {
      //retrieve data from the top
      return intArray[top];
   }

   // Operation : isFull
   // return true if stack is full
   public boolean isFull(){
      return (top == size-1);
   }

   // Operation : isEmpty
   // return true if stack is empty
   public boolean isEmpty(){
      return (top == -1);
   }
}
```

Demo Program

StackDemo.java

```
package com.tutorialspoint.datastructure;

public class StackDemo {
   public static void main (String[] args){

      // make a new stack
      Stack stack = new Stack(10);

      // push items on to the stack
      stack.push(3);
      stack.push(5);
      stack.push(9);
      stack.push(1);
      stack.push(12);
      stack.push(15);

      System.out.println("Element at top of the stack: " + stack.peek());
      System.out.println("Elements: ");

      // print stack data
      while(!stack.isEmpty()){
         int data = stack.pop();
         System.out.println(data);
      }

      System.out.println("Stack full: " + stack.isFull());
      System.out.println("Stack empty: " + stack.isEmpty());
   }
}
```

If we compile and run the above program then it would produce following result –

```
Element at top of the stack: 15
Elements:
15
12
1
9
5
3
```

```
Stack full: false
Stack empty: true
```

DSA using Java - Parsing Expressions

Ordinary airthmetic expressions like 2*(3*4) are easier for human mind to parse but for an algorithm it would be pretty difficult to parse such an expression. To ease this difficulty, an airthmetic expression can be parsed by an algorithm using a two step approach.

- Transform the provided arithmetic expression to postfix notation.
- Evaluate the postfix notation.
-

Infix Notation

Normal airthmetic expression follows Infix Notation in which operator is in between the operands. For example A+B here A is first operand, B is second operand and + is the operator acting on the two operands.

Postfix Notation

Postfix notation varies from normal arithmetic expression or infix notation in a way that the operator follows the operands. For example, consider the following examples

Sr.No	Infix Notation	Postfix Notation
1	A+B	AB+
2	(A+B)*C	AB+C*
3	A*(B+C)	ABC+*
4	A/B+C/D	AB/CD/+
5	(A+B)*(C+D)	AB+CD+*
6	((A+B)*C)-D	AB+C*D-

Infix to PostFix Conversion

Before looking into the way to translate Infix to postfix notation, we need to consider following basics of infix expression evaluation.

- Evaluation of the infix expression starts from left to right.
- Keep precedence in mind, for example * has higher precedence over +. For example
  - 2+3*4 = 2+12.
  - 2+3*4 = 14.
- Override precedence using brackets, For example
  - (2+3)*4 = 5*4.
  - (2+3)*4= 20.

Now let us transform a simple infix expression A+B*C into a postfix expression manually.

Step	Character read	Infix Expressed parsed so far	Postfix expression developed so far	Remarks
1	A	A	A	
2	+	A+	A	
3	B	A+B	AB	
4	*	A+B*	AB	+ can not be copied as * has higher precedence.
5	C	A+B*C	ABC	
6		A+B*C	ABC*	copy * as two operands are there B and C
7		A+B*C	ABC*+	copy + as two operands are there BC

				and A

Now let us transform the above infix expression A+B*C into a postfix expression using stack.

Step	Character read	Infix Expressed parsed so far	Postfix expression developed so far	Stack Contents	Remarks
1	A	A	A		
2	+	A+	A	+	push + operator in a stack.
3	B	A+B	AB	+	
4	*	A+B*	AB	+*	Precedence of operator * is higher than +. push * operator in the stack. Otherwise, + would pop up.
5	C	A+B*C	ABC	+*	
6		A+B*C	ABC*	+	No more operand, pop the * operator.
7		A+B*C	ABC*+		Pop the + operator.

Now let us see another example, by transforming infix expression A*(B+C) into a postfix expression using stack.

Step	Character read	Infix Expressed	Postfix expression	Stack Contents	Remarks

		parsed so far	developed so far		
1	A	A	A		
2	*	A*	A	*	push * operator in a stack.
3	(	A*(	A	*(	push (in the stack.
4	B	A*(B	AB	*(	
5	+	A*(B+	AB	*(+	push + in the stack.
6	C	A*(B+C	ABC	*(+	
7	)	A*(B+C)	ABC+	*(	Pop the + operator.
8		A*(B+C)	ABC+	*	Pop the (operator.
9		A*(B+C)	ABC+*		Pop the rest of the operator(s).

Demo program

Now we'll demonstrate the use of stack to convert infix expression to postfix expression and then evaluate the postfix expression.

Stack.java

```
package com.tutorialspoint.expression;

public class Stack {

   private int size;
   private int[] intArray;
   private int top;
```

```
   //Constructor
   public Stack(int size){
      this.size = size;
      intArray = new int[size];
      top = -1;
   }

   //push item on the top of the stack
   public void push(int data) {
      if(!isFull()){
         //increment top by 1 and insert data
         intArray[++top] = data;
      }else{
         System.out.println("Cannot add data. Stack is full.");
      }
   }

   //pop item from the top of the stack
   public int pop() {
      //retrieve data and decrement the top by 1
      return intArray[top--];
   }

   //view the data at top of the stack
   public int peek() {
      //retrieve data from the top
      return intArray[top];
   }

   //return true if stack is full
   public boolean isFull(){
      return (top == size-1);
   }

   //return true if stack is empty
   public boolean isEmpty(){
      return (top == -1);
   }
}
```

InfixToPostFix.java

```
package com.tutorialspoint.expression;

public class InfixToPostfix {
   private Stack stack;
   private String input = "";
   private String output = "";

   public InfixToPostfix(String input){
      this.input = input;
      stack = new Stack(input.length());
   }

   public String translate(){
      for(int i=0;i<input.length();i++){
         char ch = input.charAt(i);
            switch(ch){
               case '+':
               case '-':
                  gotOperator(ch, 1);
                  break;
               case '*':
               case '/':
                  gotOperator(ch, 2);
                  break;
               case '(':
                  stack.push(ch);
                  break;
               case ')':
                  gotParenthesis(ch);
                  break;
               default:
                  output = output+ch;
                  break;
            }
      }

      while(!stack.isEmpty()){
         output = output + (char)stack.pop();
      }

      return output;
   }
```

```
      //got operator from input
      public void gotOperator(char operator, int precedence){
      while(!stack.isEmpty()){
         char prevOperator = (char)stack.pop();
         if(prevOperator == '('){
            stack.push(prevOperator);
            break;
         }else{
            int precedence1;
            if(prevOperator == '+' || prevOperator == '-'){
               precedence1 = 1;
            }else{
               precedence1 = 2;
            }

            if(precedence1 < precedence){
               stack.push(Character.getNumericValue(prevOperator));
               break;
            }else{
               output = output + prevOperator;
            }
         }
      }
      stack.push(operator);
   }

   //got operator from input
   public void gotParenthesis(char parenthesis){
      while(!stack.isEmpty()){
         char ch = (char)stack.pop();
         if(ch == '('){
            break;
         }else{
            output = output + ch;
         }
      }
   }
}
```

PostFixParser.java

```
package com.tutorialspoint.expression;

public class PostFixParser {
private Stack stack;
private String input;

public PostFixParser(String postfixExpression){
   input = postfixExpression;
   stack = new Stack(input.length());
}

   public int evaluate(){
      char ch;
      int firstOperand;
      int secondOperand;
      int tempResult;

      for(int i=0;i<input.length();i++){
         ch = input.charAt(i);

         if(ch >= '0' && ch <= '9'){
            stack.push(Character.getNumericValue(ch));
         }else{
            firstOperand = stack.pop();
            secondOperand = stack.pop();
            switch(ch){
               case '+':
                  tempResult = firstOperand + secondOperand;
                  break;
               case '-':
                  tempResult = firstOperand - secondOperand;
                  break;
               case '*':
                  tempResult = firstOperand * secondOperand;
                  break;
               case '/':
                  tempResult = firstOperand / secondOperand;
                  break;
               default:
                  tempResult = 0;
            }
            stack.push(tempResult);
```

```
            }
         }
         return stack.pop();
      }
   }
```

PostFixDemo.java

```
package com.tutorialspoint.expression;

public class PostFixDemo {
   public static void main(String args[]){
      String input = "1*(2+3)";
      InfixToPostfix translator = new InfixToPostfix(input);
      String output = translator.translate();
      System.out.println("Infix expression is: " + input);
      System.out.println("Postfix expression is: " + output);

      PostFixParser parser = new PostFixParser(output);
      System.out.println("Result is: " + parser.evaluate());
   }
}
```

If we compile and run the above program then it would produce following result –

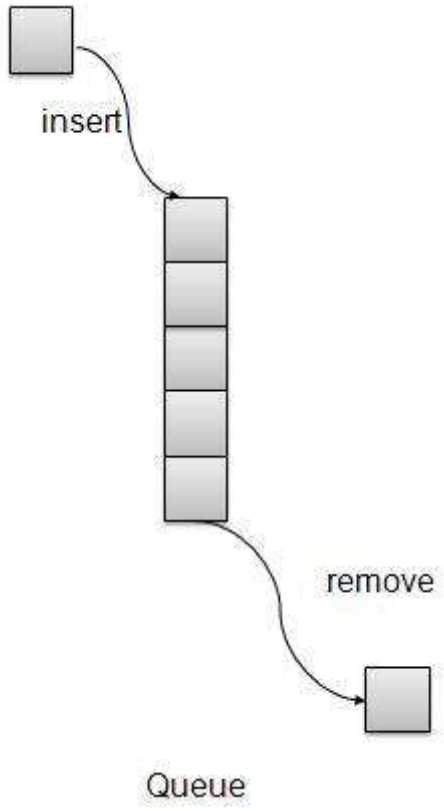
```
Infix expression is: 1*(2+3)
Postfix expression is: 123+*
Result is: 5
```

DSA using Java - Queue

Overview

Queue is kind of data structure similar to stack with primary difference that the first item inserted is the first item to be removed (FIFO - First In First Out) where stack is based on LIFO, Last In First Out principal.

Queue Representation



Basic Operations

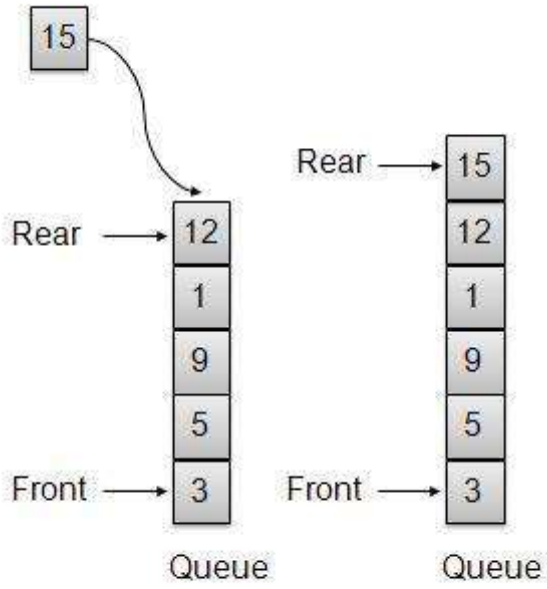
- **insert / enqueue** – add an item to the rear of the queue.
- **remove / dequeue** – remove an item from the front of the queue.

We're going to implement Queue using array in this article. There is few more operations supported by queue which are following.

- **Peek** – get the element at front of the queue.
- **isFull** – check if queue is full.
- **isEmpty** – check if queue is empty.

Insert / Enqueue Operation

Whenever an element is inserted into queue, queue increments the rear index for later use and stores that element at the rear end of the storage. If rear end reaches to the last index and it is wrapped to the bottom location. Such an arrangement is called wrap around and such queue is circular queue. This method is also termed as enqueue operation.

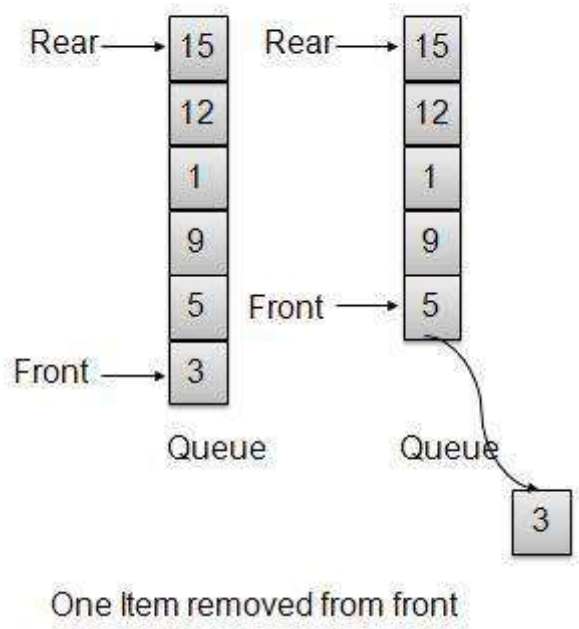


One item inserted at rear end

```
public void insert(int data){
   if(!isFull()){
      if(rear == MAX-1){
         rear = -1;
      }

      intArray[++rear] = data;
      itemCount++;
   }
}
```

Remove / Dequeue Operation

Whenever an element is to be removed from queue, queue get the element using front index and increments the front index. As a wrap around arrangement, if front index is more than array's max index, it is set to 0.



One Item removed from front

```
public int remove(){
   int data = intArray[front++];
   if(front == MAX){
      front = 0;
   }
   itemCount--;
   return data;
}
```

Queue Implementation

Queue.java

```
package com.tutorialspoint.datastructure;

public class Queue {

   private final int MAX;
   private int[] intArray;
   private int front;
   private int rear;
   private int itemCount;

   public Queue(int size){
```

```
      MAX = size;
      intArray = new int[MAX];
      front = 0;
      rear = -1;
      itemCount = 0;
   }

   public void insert(int data){
      if(!isFull()){
         if(rear == MAX-1){
            rear = -1;
         }

         intArray[++rear] = data;
         itemCount++;
      }
   }

   public int remove(){
      int data = intArray[front++];
      if(front == MAX){
         front = 0;
      }
      itemCount--;
      return data;
   }

   public int peek(){
      return intArray[front];
   }

   public boolean isEmpty(){
      return itemCount == 0;
   }

   public boolean isFull(){
      return itemCount == MAX;
   }

   public int size(){
      return itemCount;
```

```
   }
}
```

Demo Program

QueueDemo.java

```
package com.tutorialspoint.datastructure;

public class QueueDemo {
   public static void main(String[] args){
      Queue queue = new Queue(6);

      //insert 5 items
      queue.insert(3);
      queue.insert(5);
      queue.insert(9);
      queue.insert(1);
      queue.insert(12);

      // front : 0
      // rear  : 4
      // ------------------
      // index : 0 1 2 3 4
      // ------------------
      // queue : 3 5 9 1 12

      queue.insert(15);

      // front : 0
      // rear  : 5
      // ----------------------
      // index : 0 1 2 3 4  5
      // ----------------------
      // queue : 3 5 9 1 12 15

      if(queue.isFull()){
         System.out.println("Queue is full!");
      }
```

```
      //remove one item
      int num = queue.remove();
      System.out.println("Element removed: "+num);
      // front : 1
      // rear  : 5
      // -------------------
      // index : 1 2 3 4  5
      // -------------------
      // queue : 5 9 1 12 15

      //insert more items
      queue.insert(16);

      // front : 1
      // rear  : -1
      // ----------------------
      // index : 0  1 2 3 4  5
      // ----------------------
      // queue : 16 5 9 1 12 15

      //As queue is full, elements will not be inserted.
      queue.insert(17);
      queue.insert(18);

      // ----------------------
      // index : 0  1 2 3 4  5
      // ----------------------
      // queue : 16 5 9 1 12 15
      System.out.println("Element at front: "+queue.peek());

      System.out.println("----------------------");
      System.out.println("index : 5 4 3 2  1  0");
      System.out.println("----------------------");
      System.out.print("Queue:  ");
      while(!queue.isEmpty()){
         int n = queue.remove();
         System.out.print(n +" ");
      }
   }
}
```

If we compile and run the above program then it would produce following result –

```
Queue is full!
Element removed: 3
Element at front: 5
----------------------
index : 5 4 3 2  1  0
----------------------
Queue:  5 9 1 12 15 16
```

DSA using Java - Priority Queue

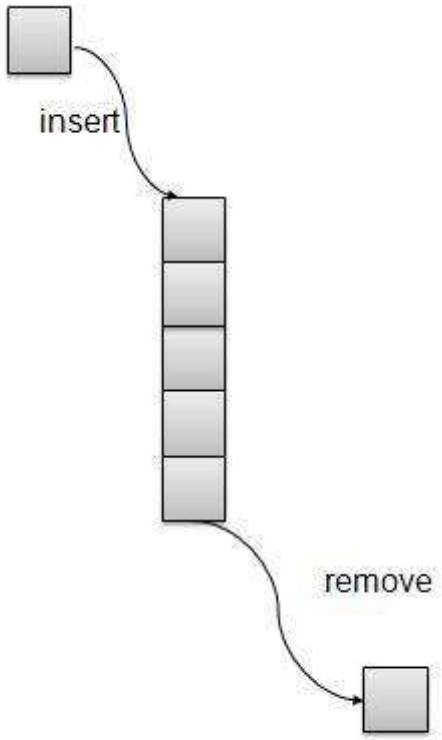
Overview

Priority Queue is more specilized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Basic Operations

- **insert / enqueue** – add an item to the rear of the queue.
- **remove / dequeue** – remove an item from the front of the queue.

Priority Queue Representation



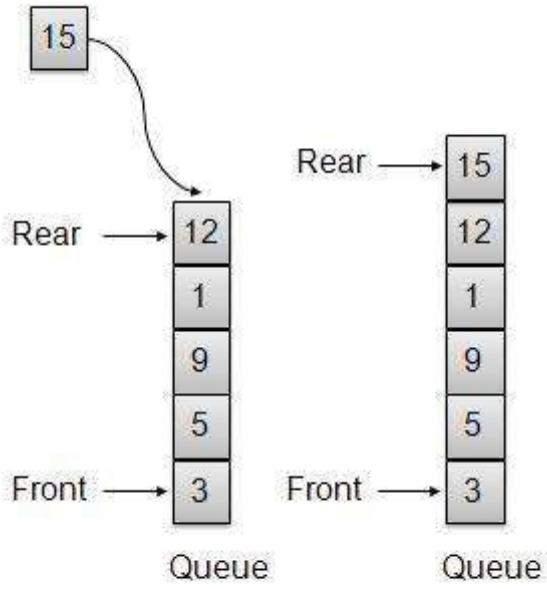
Queue

We're going to implement Queue using array in this article. There is few more operations supported by queue which are following.

- **Peek** − get the element at front of the queue.
- **isFull** − check if queue is full.
- **isEmpty** − check if queue is empty.

Insert / Enqueue Operation

Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.

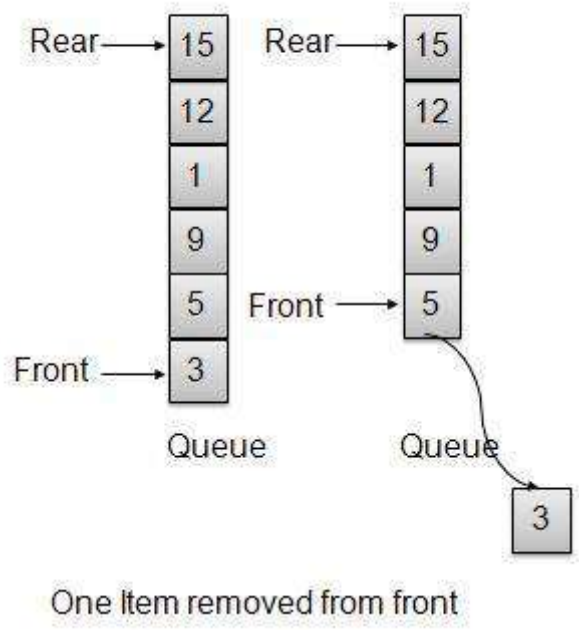


One item inserted at rear end

```
public void insert(int data){
   int i =0;

   if(!isFull()){
      // if queue is empty, insert the data
      if(itemCount == 0){
         intArray[itemCount++] = data;
      }else{
         // start from the right end of the queue
         for(i = itemCount - 1; i >= 0; i-- ){
            // if data is larger, shift existing item to right end
            if(data > intArray[i]){
               intArray[i+1] = intArray[i];
            }else{
               break;
            }
         }
         // insert the data
         intArray[i+1] = data;
         itemCount++;
      }
   }
}
```

Remove / Dequeue Operation

Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one.



One Item removed from front

```
public int remove(){
    return intArray[--itemCount];
}
```

Priority Queue Implementation

PriorityQueue.java

```
package com.tutorialspoint.datastructure;

public class PriorityQueue {
   private final int MAX;
   private int[] intArray;
   private int itemCount;

   public PriorityQueue(int size){
      MAX = size;
      intArray = new int[MAX];
      itemCount = 0;
   }

   public void insert(int data){
```

```
      int i =0;

      if(!isFull()){
         // if queue is empty, insert the data
         if(itemCount == 0){
            intArray[itemCount++] = data;
         }else{
            // start from the right end of the queue
            for(i = itemCount - 1; i >= 0; i-- ){
               // if data is larger, shift existing item to right end
               if(data > intArray[i]){
                  intArray[i+1] = intArray[i];
               }else{
                  break;
               }
            }
            // insert the data
            intArray[i+1] = data;
            itemCount++;
         }
      }
   }

   public int remove(){
      return intArray[--itemCount];
   }

   public int peek(){
      return intArray[itemCount - 1];
   }

   public boolean isEmpty(){
      return itemCount == 0;
   }

   public boolean isFull(){
      return itemCount == MAX;
   }

   public int size(){
      return itemCount;
```

```
   }
}
```

Demo Program

PriorityQueueDemo.java

```
package com.tutorialspoint.datastructure;

public class PriorityQueueDemo {
   public static void main(String[] args){
      PriorityQueue queue = new PriorityQueue(6);

      //insert 5 items
      queue.insert(3);
      queue.insert(5);
      queue.insert(9);
      queue.insert(1);
      queue.insert(12);

      // ------------------
      // index : 0  1 2 3 4
      // ------------------
      // queue : 12 9 5 3 1

      queue.insert(15);

      // ---------------------
      // index : 0  1 2 3 4  5
      // ---------------------
      // queue : 15 12 9 5 3 1

      if(queue.isFull()){
         System.out.println("Queue is full!");
      }


      //remove one item
      int num = queue.remove();
      System.out.println("Element removed: "+num);
      // ---------------------
```

```
      // index : 0  1  2 3 4
      // ---------------------
      // queue : 15 12 9 5 3

      //insert more items
      queue.insert(16);

      // ----------------------
      // index :  0  1 2 3 4  5
      // ----------------------
      // queue : 16 15 12 9 5 3

      //As queue is full, elements will not be inserted.
      queue.insert(17);
      queue.insert(18);

      // ----------------------
      // index : 0   1  2 3 4 5
      // ----------------------
      // queue : 16 15 12 9 5 3
      System.out.println("Element at front: "+queue.peek());
      System.out.println("-----------------------");
      System.out.println("index : 5 4 3 2  1  0");
      System.out.println("-----------------------");
      System.out.print("Queue:  ");
      while(!queue.isEmpty()){
         int n = queue.remove();
         System.out.print(n +" ");
      }
   }
}
```

If we compile and run the above program then it would produce following result –

```
Queue is full!
Element removed: 1
Element at front: 3
----------------------
index : 5 4 3 2  1  0
----------------------
Queue:  3 5 9 12 15 16
```

DSA using Java - Tree

Overview

Tree represents nodes connected by edges. We'll going to discuss binary tree or binary search tree specifically.

Binary Tree is a special datastructure used for data storage purposes. A binary tree has a special condition that each node can have two children at maximum. A binary tree have benefits of both an ordered array and a linked list as search is as quick as in sorted array and insertion or deletion operation are as fast as in linked list.

Terms

Following are important terms with respect to tree.

- **Path** − Path refers to sequence of nodes along the edges of a tree.
- **Root** − Node at the top of the tree is called root. There is only one root per tree and one path from root node to any node.
- **Parent** − Any node except root node has one edge upward to a node called parent.
- **Child** − Node below a given node connected by its edge downward is called its child node.
- **Leaf** − Node which does not have any child node is called leaf node.
- **Subtree** − Subtree represents descendents of a node.
- **Visiting** − Visiting refers to checking value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If root node is at level 0, then its next child node is at level 1, its grandchild is at level 2 and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Binary Search tree exibits a special behaviour. A node's left child must have value less than its parent's value and node's right child must have value greater than it's parent value.

Binary Search Tree Representation

We're going to implement tree using node object and connecting them through references.

Basic Operations

Following are basic primary operations of a tree which are following.

- **Search** – search an element in a tree.
- **Insert** – insert an element in a tree.
- **Preorder Traversal** – traverse a tree in a preorder manner.
- **Inorder Traversal** – traverse a tree in an inorder manner.
- **Postorder Traversal** – traverse a tree in a postorder manner.

Node

Define a node having some data, references to its left and right child nodes.

```
public class Node {
   public int data;
   public Node leftChild;
   public Node rightChild;

   public Node(){}

   public void display(){
      System.out.print("("+data+ ")");
   }
}
```

Search Operation

Whenever an element is to be search. Start search from root node then if data is less than key value, search element in left subtree otherwise search element in right subtree. Follow the same algorithm for each node.

```
public Node search(int data){
   Node current = root;
   System.out.print("Visiting elements: ");
   while(current.data != data){
      if(current != null)
         System.out.print(current.data + " ");
         //go to left tree
         if(current.data > data){
            current = current.leftChild;
         }//else go to right tree
         else{
            current = current.rightChild;
         }
         //not found
         if(current == null){
            return null;
         }
      }
   return current;
}
```

Insert Operation

Whenever an element is to be inserted. First locate its proper location. Start search from root node then if data is less than key value, search empty location in left subtree and insert the data. Otherwise search empty location in right subtree and insert the data.

```
public void insert(int data){
   Node tempNode = new Node();
   tempNode.data = data;

   //if tree is empty
   if(root == null){
      root = tempNode;
   }else{
      Node current = root;
      Node parent = null;

      while(true){
```

```
         parent = current;
         //go to left of the tree
         if(data < parent.data){
            current = current.leftChild;
            //insert to the left
            if(current == null){
               parent.leftChild = tempNode;
               return;
            }
         }//go to right of the tree
         else{
            current = current.rightChild;
            //insert to the right
            if(current == null){
               parent.rightChild = tempNode;
               return;
            }
         }
      }
   }
}
```

Preorder Traversal

It is a simple three step process.

- visit root node
- traverse left subtree
- traverse right subtree

```
private void preOrder(Node root){
   if(root!=null){
      System.out.print(root.data + " ");
      preOrder(root.leftChild);
      preOrder(root.rightChild);
   }
}
```

Inorder Traversal

It is a simple three step process.

- traverse left subtree
- visit root node
- traverse right subtree

```
private void inOrder(Node root){
   if(root!=null){
      inOrder(root.leftChild);
      System.out.print(root.data + " ");
      inOrder(root.rightChild);
   }
}
```

Postorder Traversal

It is a simple three step process.

- traverse left subtree
- traverse right subtree
- visit root node

```
private void postOrder(Node root){
   if(root!=null){
      postOrder(root.leftChild);
      postOrder(root.rightChild);
      System.out.print(root.data + " ");
   }
}
```

Tree Implementation

Node.java

```
package com.tutorialspoint.datastructure;

public class Node {
   public int data;
   public Node leftChild;
   public Node rightChild;

   public Node(){}

   public void display(){
      System.out.print("("+data+ ")");
   }
}
```

Tree.java

```
package com.tutorialspoint.datastructure;

public class Tree {
   private Node root;

   public Tree(){
      root = null;
   }

   public Node search(int data){
      Node current = root;
      System.out.print("Visiting elements: ");
      while(current.data != data){
         if(current != null)
            System.out.print(current.data + " ");
            //go to left tree
            if(current.data > data){
               current = current.leftChild;
            }//else go to right tree
            else{
               current = current.rightChild;
            }
            //not found
            if(current == null){
               return null;
```

```
            }
         }
      return current;
   }

   public void insert(int data){
      Node tempNode = new Node();
      tempNode.data = data;

      //if tree is empty
      if(root == null){
         root = tempNode;
     }else{
         Node current = root;
         Node parent = null;

         while(true){
            parent = current;
            //go to left of the tree
            if(data < parent.data){
               current = current.leftChild;
               //insert to the left
               if(current == null){
                  parent.leftChild = tempNode;
                  return;
               }
            }//go to right of the tree
            else{
               current = current.rightChild;
               //insert to the right
               if(current == null){
                  parent.rightChild = tempNode;
                  return;
               }
            }
         }
      }
   }

   public void traverse(int traversalType){
      switch(traversalType){
         case 1:
```

```
            System.out.print("\nPreorder traversal: ");
            preOrder(root);
            break;
         case 2:
            System.out.print("\nInorder traversal: ");
            inOrder(root);
            break;
         case 3:
            System.out.print("\nPostorder traversal: ");
            postOrder(root);
            break;
         }
   }

   private void preOrder(Node root){
      if(root!=null){
         System.out.print(root.data + " ");
         preOrder(root.leftChild);
         preOrder(root.rightChild);
      }
   }

   private void inOrder(Node root){
      if(root!=null){
         inOrder(root.leftChild);
         System.out.print(root.data + " ");
         inOrder(root.rightChild);
      }
   }

   private void postOrder(Node root){
      if(root!=null){
         postOrder(root.leftChild);
         postOrder(root.rightChild);
         System.out.print(root.data + " ");
      }
   }
}
```

Demo Program

TreeDemo.java

```
package com.tutorialspoint.datastructure;

public class TreeDemo {
   public static void main(String[] args){
      Tree tree = new Tree();

      /*                    11              //Level 0
      */
      tree.insert(11);
      /*                    11              //Level 0
      *                     |
      *                     |---20          //Level 1
      */
      tree.insert(20);
      /*                    11              //Level 0
      *                     |
      *                 3---|---20          //Level 1
      */
      tree.insert(3);
      /*                    11              //Level 0
      *                     |
      *                 3---|---20          //Level 1
      *                         |
      *                         |--42       //Level 2
      */
      tree.insert(42);
      /*                    11              //Level 0
      *                     |
      *                 3---|---20          //Level 1
      *                         |
      *                         |--42       //Level 2
      *                             |
      *                             |--54   //Level 3
      */
      tree.insert(54);
      /*                    11              //Level 0
      *                     |
      *                 3---|---20          //Level 1
      *                         |
      *                      16--|--42       //Level 2
```

```
        *                                |
        *                                |--54   //Level 3
        */
        tree.insert(16);
        /*                      11              //Level 0
        *                       |
        *                  3---|---20           //Level 1
        *                           |
        *                        16--|--42       //Level 2
        *                                |
        *                            32--|--54   //Level 3
        */
        tree.insert(32);
        /*                      11              //Level 0
        *                       |
        *                  3---|---20           //Level 1
        *                  |        |
        *                  |--9 16--|--42       //Level 2
        *                                |
        *                            32--|--54   //Level 3
        */
        tree.insert(9);
        /*                      11              //Level 0
        *                       |
        *                  3---|---20           //Level 1
        *                  |        |
        *                  |--9 16--|--42       //Level 2
        *                     |        |
        *                  4--|     32--|--54   //Level 3
        */
        tree.insert(4);
        /*                      11              //Level 0
        *                       |
        *                  3---|---20           //Level 1
        *                  |        |
        *                  |--9 16--|--42       //Level 2
        *                     |        |
        *                  4--|--10 32--|--54   //Level 3
        */
        tree.insert(10);
        Node node = tree.search(32);
        if(node!=null){
```

```
            System.out.print("Element found.");
            node.display();
            System.out.println();
         }else{
            System.out.println("Element not found.");
         }

         Node node1 = tree.search(2);
         if(node1!=null){
            System.out.println("Element found.");
            node1.display();
            System.out.println();
         }else{
            System.out.println("Element not found.");
         }

         //pre-order traversal
         //root, left ,right
         tree.traverse(1);
         //in-order traversal
         //left, root ,right
         tree.traverse(2);
         //post order traversal
         //left, right, root
         tree.traverse(3);
      }
   }
```

If we compile and run the above program then it would produce following result –

```
Visiting elements: 11 20 42 Element found.(32)
Visiting elements: 11 3 Element not found.

Preorder traversal: 11 3 9 4 10 20 16 42 32 54
Inorder traversal: 3 4 9 10 11 16 20 32 42 54
Postorder traversal: 4 10 9 3 16 32 54 42 20 11
```

DSA using Java - Hash Table

Overview

HashTable is a datastructure in which insertion and search operations are very fast irrespective of size of the hashtable. It is nearly a constant or O(1). Hash Table uses array as a storage medium and uses hash technique to generate index where an element is to be inserted or to be located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hashtable of size 20, and following items are to be stored. Item are in (key,value) format.

- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Linear Probing

As we can see, it may happen that the hashing technique used create already used index of the array. In such case, we can search the next empty location in the array by looking into the next cell until we found an empty cell. This technique is called linear probing.

Sr.No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Basic Operations

Following are basic primary operations of a hashtable which are following.

- **Search** – search an element in a hashtable.
- **Insert** – insert an element in a hashtable.
- **delete** – delete an element from a hashtable.

DataItem

Define a data item having some data, and key based on which search is to be conducted in hashtable.

```
public class DataItem {
   private int key;
   private int data;

   public DataItem(int key, int data){
      this.key = key;
      this.data = data;
   }

   public int getKey(){
      return key;
   }

   public int getData(){
      return data;
   }
}
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
public int hashCode(int key){
   return key % size;
}
```

Search Operation

Whenever an element is to be searched. Compute the hash code of the key passed and locate the element using that hashcode as index in the array. Use linear probing to get element ahead if element not found at computed hash code.

```
public DataItem search(int key){
   //get the hash
```

```
   int hashIndex = hashCode(key);
   //move in array until an empty
   while(hashArray[hashIndex] !=null){
      if(hashArray[hashIndex].getKey() == key)
         return hashArray[hashIndex];
      //go to next cell
      ++hashIndex;
      //wrap around the table
      hashIndex %= size;
   }
   return null;
}
```

Insert Operation

Whenever an element is to be inserted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing for empty location if an element is found at computed hash code.

```
public void insert(DataItem item){
   int key = item.getKey();

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty or deleted cell
   while(hashArray[hashIndex] !=null
      && hashArray[hashIndex].getKey() != -1){
      //go to next cell
      ++hashIndex;
      //wrap around the table
      hashIndex %= size;
   }

   hashArray[hashIndex] = item;
}
```

Delete Operation

Whenever an element is to be deleted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing to get element ahead if an element is not found at computed hash code. When found, store a dummy item there to keep performance of hashtable intact.

```
public DataItem delete(DataItem item){
   int key = item.getKey();

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] !=null){
      if(hashArray[hashIndex].getKey() == key){
         DataItem temp = hashArray[hashIndex];
         //assign a dummy item at deleted position
         hashArray[hashIndex] = dummyItem;
         return temp;
      }
      //go to next cell
      ++hashIndex;
      //wrap around the table
      hashIndex %= size;
   }
   return null;
}
```

HashTable Implementation

DataItem.java

```
package com.tutorialspoint.datastructure;

public class DataItem {
   private int key;
   private int data;

   public DataItem(int key, int data){
      this.key = key;
      this.data = data;
   }
```

```
   public int getKey(){
      return key;
   }

   public int getData(){
      return data;
   }
}
```

HashTable.java

```
package com.tutorialspoint.datastructure;

public class HashTable {

   private DataItem[] hashArray;
   private int size;
   private DataItem dummyItem;

   public HashTable(int size){
      this.size = size;
      hashArray = new DataItem[size];
      dummyItem = new DataItem(-1,-1);
   }

   public void display(){
      for(int i=0; i<size; i++) {
         if(hashArray[i] != null)
            System.out.print(" ("
               +hashArray[i].getKey()+","
               +hashArray[i].getData() + ") ");
         else
            System.out.print(" ~~ ");
      }
      System.out.println("");
   }

   public int hashCode(int key){
      return key % size;
   }
```

```
   public DataItem search(int key){
      //get the hash
      int hashIndex = hashCode(key);
      //move in array until an empty
      while(hashArray[hashIndex] !=null){
         if(hashArray[hashIndex].getKey() == key)
            return hashArray[hashIndex];
         //go to next cell
         ++hashIndex;
         //wrap around the table
         hashIndex %= size;
      }
      return null;
   }

   public void insert(DataItem item){
      int key = item.getKey();

      //get the hash
      int hashIndex = hashCode(key);

      //move in array until an empty or deleted cell
      while(hashArray[hashIndex] !=null
         && hashArray[hashIndex].getKey() != -1){
         //go to next cell
         ++hashIndex;
         //wrap around the table
         hashIndex %= size;
      }

      hashArray[hashIndex] = item;
   }

   public DataItem delete(DataItem item){
      int key = item.getKey();

      //get the hash
      int hashIndex = hashCode(key);

      //move in array until an empty
      while(hashArray[hashIndex] !=null){
         if(hashArray[hashIndex].getKey() == key){
```

```
            DataItem temp = hashArray[hashIndex];
            //assign a dummy item at deleted position
            hashArray[hashIndex] = dummyItem;
            return temp;
         }
         //go to next cell
         ++hashIndex;
         //wrap around the table
         hashIndex %= size;
      }
      return null;
   }
}
```

Demo Program

HashTableDemo.java

```
package com.tutorialspoint.datastructure;

public class HashTableDemo {
   public static void main(String[] args){
      HashTable hashTable = new HashTable(20);

      hashTable.insert(new DataItem(1, 20));
      hashTable.insert(new DataItem(2, 70));
      hashTable.insert(new DataItem(42, 80));
      hashTable.insert(new DataItem(4, 25));
      hashTable.insert(new DataItem(12, 44));
      hashTable.insert(new DataItem(14, 32));
      hashTable.insert(new DataItem(17, 11));
      hashTable.insert(new DataItem(13, 78));
      hashTable.insert(new DataItem(37, 97));

      hashTable.display();

      DataItem item = hashTable.search(37);

      if(item != null){
         System.out.println("Element found: "+ item.getData());
      }else{
```

```
         System.out.println("Element not found");
      }

      hashTable.delete(item);

      item = hashTable.search(37);

      if(item != null){
         System.out.println("Element found: "+ item.getData());
      }else{
         System.out.println("Element not found");
      }
   }
}
```

If we compile and run the above program then it would produce following result –

```
~~ (1,20) (2,70) (42,80) (4,25) ~~ ~~ ~~ ~~ ~~ ~~ ~~ (12,44) (13,78) (1
Element found: 97
Element not found
```

DSA using Java - Heap

Overview

Heap represents a special tree based data structure used to represent priority queue or for heap sort. We'll going to discuss binary heap tree specifically.

Binary heap tree can be classified as a binary tree with two constraints –

- **Completeness** – Binary heap tree is a complete binary tree except the last level which may not have all elements but elements from left to right should be filled in.
- **Heapness** – All parent nodes should be greater or smaller to their children. If parent node is to be greater than its child then it is called Max heap otherwise it is called Min heap. Max heap is used for heap sort and Min heap is used for priority queue. We're considering Min Heap and will use array implementation for the same.

Basic Operations

Following are basic primary operations of a Min heap which are following.

- **Insert** – insert an element in a heap.
- **Get Minimum** – get minimum element from the heap.
- **Remove Minimum** – remove the minimum element from the heap

Insert Operation

- Whenever an element is to be inserted. Insert element at the end of the array. Increase the size of heap by 1.
- Heap up the element while heap property is broken. Compare element with parent's value and swap them if required.

```
public void insert(int value) {
   size++;
   intArray[size - 1] = value;
   heapUp(size - 1);
}

private void heapUp(int nodeIndex){
   int parentIndex, tmp;
   if (nodeIndex != 0) {
      parentIndex = getParentIndex(nodeIndex);
      if (intArray[parentIndex] > intArray[nodeIndex]) {
         tmp = intArray[parentIndex];
         intArray[parentIndex] = intArray[nodeIndex];
         intArray[nodeIndex] = tmp;
         heapUp(parentIndex);
      }
   }
}
```

Get Minimum

Get the first element of the array implementing the heap being root.

```
public int getMinimum(){
   return intArray[0];
}
```

Remove Minimum

- Whenever an element is to be removed. Get the last element of the array and reduce size of heap by 1.
- Heap down the element while heap property is broken. Compare element with children's value and swap them if required.

```
public void removeMin() {
   intArray[0] = intArray[size - 1];
   size--;
   if (size > 0)
      heapDown(0);
}

private void heapDown(int nodeIndex){
   int leftChildIndex, rightChildIndex, minIndex, tmp;
   leftChildIndex = getLeftChildIndex(nodeIndex);
   rightChildIndex = getRightChildIndex(nodeIndex);
   if (rightChildIndex >= size) {
      if (leftChildIndex >= size)
         return;
      else
         minIndex = leftChildIndex;
   } else {
      if (intArray[leftChildIndex] <= intArray[rightChildIndex])
         minIndex = leftChildIndex;
      else
         minIndex = rightChildIndex;
   }
   if (intArray[nodeIndex] > intArray[minIndex]) {
      tmp = intArray[minIndex];
      intArray[minIndex] = intArray[nodeIndex];
      intArray[nodeIndex] = tmp;
      heapDown(minIndex);
```

```
   }
}
```

Heap Implementation

Heap.java

```
package com.tutorialspoint.datastructure;

public class Heap {
   private int[] intArray;
   private int size;

   public Heap(int size){
      intArray = new int[size];
   }

   public boolean isEmpty(){
      return size == 0;
   }

   public int getMinimum(){
      return intArray[0];
   }

   public int getLeftChildIndex(int nodeIndex){
      return 2*nodeIndex +1;
   }

   public int getRightChildIndex(int nodeIndex){
      return 2*nodeIndex +2;
   }

   public int getParentIndex(int nodeIndex){
      return (nodeIndex -1)/2;
   }

   public boolean isFull(){
      return size == intArray.length;
   }
```

```
   public void insert(int value) {
      size++;
      intArray[size - 1] = value;
      heapUp(size - 1);
   }

   public void removeMin() {
      intArray[0] = intArray[size - 1];
      size--;
      if (size > 0)
         heapDown(0);
   }

   /**
   * Heap up the new element,until heap property is broken.
   * Steps:
   * 1. Compare node's value with parent's value.
   * 2. Swap them, If they are in wrong order.
   * */
   private void heapUp(int nodeIndex){
      int parentIndex, tmp;
      if (nodeIndex != 0) {
         parentIndex = getParentIndex(nodeIndex);
         if (intArray[parentIndex] > intArray[nodeIndex]) {
            tmp = intArray[parentIndex];
            intArray[parentIndex] = intArray[nodeIndex];
            intArray[nodeIndex] = tmp;
            heapUp(parentIndex);
         }
      }
   }

   /**
   * Heap down the root element being least in value,until heap property is b
   * Steps:
   * 1.If current node has no children, done.
   * 2.If current node has one children and heap property is broken,
   * 3.Swap the current node and child node and heap down.
   * 4.If current node has one children and heap property is broken, find sma
   * 5.Swap the current node and child node and heap down.
   * */
   private void heapDown(int nodeIndex){
```

```
      int leftChildIndex, rightChildIndex, minIndex, tmp;
      leftChildIndex = getLeftChildIndex(nodeIndex);
      rightChildIndex = getRightChildIndex(nodeIndex);
      if (rightChildIndex >= size) {
         if (leftChildIndex >= size)
            return;
         else
            minIndex = leftChildIndex;
      } else {
         if (intArray[leftChildIndex] <= intArray[rightChildIndex])
            minIndex = leftChildIndex;
         else
            minIndex = rightChildIndex;
      }
      if (intArray[nodeIndex] > intArray[minIndex]) {
         tmp = intArray[minIndex];
         intArray[minIndex] = intArray[nodeIndex];
         intArray[nodeIndex] = tmp;
         heapDown(minIndex);
      }
   }
}
```

Demo Program

HeapDemo.java

```
package com.tutorialspoint.datastructure;

public class HeapDemo {
   public static void main(String[] args){
      Heap heap = new Heap(10);
       /*                    5               //Level 0
        *
        */
      heap.insert(5);
       /*                    1               //Level 0
        *                     |
        *                5---|               //Level 1
        */
      heap.insert(1);
```

```
        /*                    1                //Level 0
         *                    |
         *                5---|---3            //Level 1
         */
        heap.insert(3);
        /*                    1                //Level 0
         *                    |
         *                5---|---3            //Level 1
         *                |
         *             8--|                    //Level 2
         */
        heap.insert(8);
        /*                    1                //Level 0
         *                    |
         *                5---|---3            //Level 1
         *                |
         *             8--|--9                 //Level 2
         */
        heap.insert(9);
        /*                    1                //Level 0
         *                    |
         *                5---|---3            //Level 1
         *                |       |
         *             8--|--9 6--|            //Level 2
         */
        heap.insert(6);
        /*                    1                //Level 0
         *                    |
         *                5---|---2            //Level 1
         *                |       |
         *             8--|--9 6--|--3         //Level 2
         */
        heap.insert(2);

        System.out.println(heap.getMinimum());

        heap.removeMin();
        /*                    2                //Level 0
         *                    |
         *                5---|---3            //Level 1
         *                |       |
         *             8--|--9 6--|            //Level 2
```

```
         */
        System.out.println(heap.getMinimum());
    }
}
```

If we compile and run the above program then it would produce following result –

```
1
2
```

DSA using Java - Graph

Overview

Graph is a datastructure to model the mathematical graphs. It consists of a set of connected pairs called edges of vertices. We can represent a graph using an array of vertices and a two dimentional array of edges.

Important terms

- **Vertex** – Each node of the graph is represented as a vertex. In example given below, labeled circle represents vertices. So A to G are vertices. We can represent them using an array as shown in image below. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** – Edge represents a path between two vertices or a line between two vertices. In example given below, lines from A to B, B to C and so on represents edges. We can use a two dimentional array to represent array as shown in image below. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In example given below, B is adjacent to A, C is adjacent to B and so on.
- **Path** – Path represents a sequence of edges betweeen two vertices. In example given below, ABCD represents a path from A to D.

Basic Operations

Following are basic primary operations of a Graph which are following.

- **Add Vertex** – add a vertex to a graph.
- **Add Edge** – add an edge between two vertices of a graph.
- **Display Vertex** – display a vertex of a graph.

Add Vertex Operation

```
//add vertex to the array of vertex
public void addVertex(char label){
   lstVertices[vertexCount++] = new Vertex(label);
}
```

Add Edge Operation

```
//add edge to edge array
public void addEdge(int start,int end){
   adjMatrix[start][end] = 1;
   adjMatrix[end][start] = 1;
}
```

Display Edge Operation

```
//display the vertex
public void displayVertex(int vertexIndex){
   System.out.print(lstVertices[vertexIndex].label+" ");
}
```

Traversal Algorithms

Following are important traversal algorithms on a Graph.

- **Depth First Search** – traverses a graph in depthwards motion.
- **Breadth First Search** – traverses a graph in breadthwards motion.

Depth First Search Algorithm

Depth First Search algorithm(DFS) traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.

As in example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex found, pop up a vertex from stack. (It will pop up all the vertices from the stack which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until stack is empty.

```
public void depthFirstSearch(){
   //mark first node as visited
   lstVertices[0].visited = true;
   //display the vertex
   displayVertex(0);
   //push vertex index in stack
   stack.push(0);

   while(!stack.isEmpty()){
      //get the unvisited vertex of vertex which is at top of the stack
      int unvisitedVertex = getAdjUnvisitedVertex(stack.peek());
      //no adjacent vertex found
      if(unvisitedVertex == -1){
         stack.pop();
      }else{
         lstVertices[unvisitedVertex].visited = true;
         displayVertex(unvisitedVertex);
         stack.push(unvisitedVertex);
      }
   }

   //stack is empty, search is complete, reset the visited flag
   for(int i=0;i<vertexCount;i++){
      lstVertices[i].visited = false;
   }
}
```

Breadth First Search Algorithm

Breadth First Search algorithm(BFS) traverses a graph in a breadthwards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.

As in example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex found, remove the first vertex from queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until queue is empty.

```
public void breadthFirstSearch(){
   //mark first node as visited
   lstVertices[0].visited = true;
   //display the vertex
   displayVertex(0);
   //insert vertex index in queue
   queue.insert(0);

   int unvisitedVertex;
   while(!queue.isEmpty()){
      //get the unvisited vertex of vertex which is at front of the queue
      int tempVertex = queue.remove();
      //no adjacent vertex found
      while((unvisitedVertex=getAdjUnvisitedVertex(tempVertex)) != -1){
         lstVertices[unvisitedVertex].visited = true;
         displayVertex(unvisitedVertex);
         queue.insert(unvisitedVertex);
      }
   }

   //queue is empty, search is complete, reset the visited flag
   for(int i=0;i<vertexCount;i++){
      lstVertices[i].visited = false;
   }
}
```

Graph Implementation

Stack.java

```
package com.tutorialspoint.datastructure;

public class Stack {
   private int size;           // size of the stack
   private int[] intArray;     // stack storage
   private int top;            // top of the stack

   // Constructor
   public Stack(int size){
      this.size = size;
      intArray = new int[size];   //initialize array
      top = -1;                   //stack is initially empty
   }

   // Operation : Push
   // push item on the top of the stack
   public void push(int data) {

      if(!isFull()){
         // increment top by 1 and insert data
         intArray[++top] = data;
      }else{
         System.out.println("Cannot add data. Stack is full.");
      }
   }

   // Operation : Pop
   // pop item from the top of the stack
   public int pop() {
      //retrieve data and decrement the top by 1
      return intArray[top--];
   }

   // Operation : Peek
   // view the data at top of the stack
   public int peek() {
      //retrieve data from the top
```

```
      return intArray[top];
   }

   // Operation : isFull
   // return true if stack is full
   public boolean isFull(){
      return (top == size-1);
   }

   // Operation : isEmpty
   // return true if stack is empty
   public boolean isEmpty(){
      return (top == -1);
   }
}
```

Queue.java

```
package com.tutorialspoint.datastructure;

public class Queue {

   private final int MAX;
   private int[] intArray;
   private int front;
   private int rear;
   private int itemCount;

   public Queue(int size){
      MAX = size;
      intArray = new int[MAX];
      front = 0;
      rear = -1;
      itemCount = 0;
   }

   public void insert(int data){
      if(!isFull()){
         if(rear == MAX-1){
            rear = -1;
         }
```

```
         intArray[++rear] = data;
         itemCount++;
      }
   }

   public int remove(){
      int data = intArray[front++];
      if(front == MAX){
         front = 0;
      }
      itemCount--;
      return data;
   }

   public int peek(){
      return intArray[front];
   }

   public boolean isEmpty(){
      return itemCount == 0;
   }

   public boolean isFull(){
      return itemCount == MAX;
   }

   public int size(){
      return itemCount;
   }
}
```

Vertex.java

```
package com.tutorialspoint.datastructure;

public class Vertex {
   public char label;
   public boolean visited;

   public Vertex(char label){
      this.label = label;
      visited = false;
```

```
   }
}
```

Graph.java

```
package com.tutorialspoint.datastructure;

public class Graph {
   private final int MAX = 20;
   //array of vertices
   private Vertex lstVertices[];
   //adjacency matrix
   private int adjMatrix[][];
   //vertex count
   private int vertexCount;

   private Stack stack;
   private Queue queue;

   public Graph(){
      lstVertices = new Vertex[MAX];
      adjMatrix = new int[MAX][MAX];
      vertexCount = 0;
      stack = new Stack(MAX);
      queue = new Queue(MAX);
      for(int j=0; j<MAX; j++) // set adjacency
         for(int k=0; k<MAX; k++) // matrix to 0
            adjMatrix[j][k] = 0;
   }

   //add vertex to the vertex list
   public void addVertex(char label){
      lstVertices[vertexCount++] = new Vertex(label);
   }

   //add edge to edge array
   public void addEdge(int start,int end){
      adjMatrix[start][end] = 1;
      adjMatrix[end][start] = 1;
   }

   //display the vertex
```

```
   public void displayVertex(int vertexIndex){
      System.out.print(lstVertices[vertexIndex].label+" ");
   }

   //get the adjacent unvisited vertex
   public int getAdjUnvisitedVertex(int vertexIndex){
      for(int i=0; i<vertexCount; i++)
         if(adjMatrix[vertexIndex][i]==1 && lstVertices[i].visited==false)
            return i;
      return -1;
   }

   public void depthFirstSearch(){
      //mark first node as visited
      lstVertices[0].visited = true;
      //display the vertex
      displayVertex(0);
      //push vertex index in stack
      stack.push(0);

      while(!stack.isEmpty()){
         //get the unvisited vertex of vertex which is at top of the stack
         int unvisitedVertex = getAdjUnvisitedVertex(stack.peek());
         //no adjacent vertex found
         if(unvisitedVertex == -1){
            stack.pop();
         }else{
            lstVertices[unvisitedVertex].visited = true;
            displayVertex(unvisitedVertex);
            stack.push(unvisitedVertex);
         }
      }

      //stack is empty, search is complete, reset the visited flag
      for(int i=0;i<vertexCount;i++){
         lstVertices[i].visited = false;
      }
   }

   public void breadthFirstSearch(){
      //mark first node as visited
      lstVertices[0].visited = true;
```

```
      //display the vertex
      displayVertex(0);
      //insert vertex index in queue
      queue.insert(0);
      int unvisitedVertex;
      while(!queue.isEmpty()){
         //get the unvisited vertex of vertex which is at front of the queue
         int tempVertex = queue.remove();
         //no adjacent vertex found
         while((unvisitedVertex=getAdjUnvisitedVertex(tempVertex)) != -1){
            lstVertices[unvisitedVertex].visited = true;
            displayVertex(unvisitedVertex);
            queue.insert(unvisitedVertex);
         }
      }

      //queue is empty, search is complete, reset the visited flag
      for(int i=0;i<vertexCount;i++){
         lstVertices[i].visited = false;
      }
   }
}
```

Demo Program

GraphDemo.java

```
package com.tutorialspoint.datastructure;

public class GraphDemo {
   public static void main(String args[]){
      Graph graph = new Graph();

      graph.addVertex('A');   //0
      graph.addVertex('B');   //1
      graph.addVertex('C');   //2
      graph.addVertex('D');   //3
      graph.addVertex('E');   //4
      graph.addVertex('F');   //5
      graph.addVertex('G');   //6
```

```
      /*       1  2  3
       * 0  |--B--C--D
       * A--|
       * |
       * |     4
       * |-----E
       * |     5  6
       * |  |--F--G
       * |--|
       */
      graph.addEdge(0, 1);   //AB
      graph.addEdge(1, 2);   //BC
      graph.addEdge(2, 3);   //CD
      graph.addEdge(0, 4);   //AC
      graph.addEdge(0, 5);   //AF
      graph.addEdge(5, 6);   //FG
      System.out.print("Depth First Search: ");
      //A B C D E F G
      graph.depthFirstSearch();
      System.out.println("");
      System.out.print("Breadth First Search: ");
      //A B E F C G D
      graph.breadthFirstSearch();
   }
}
```

If we compile and run the above program then it would produce following result –

```
Depth First Search: A B C D E F G
Breadth First Search: A B E F C G D
```

DSA using Java - Search techniques

Search refers to locating a desired element of specified properties in a collection of items. We are going to start our discussion using following commonly used and simple search algorithms.

Sr.No	Technique & Description
1	**Linear Search**

	Linear search searches all items and its worst execution time is n where n is the number of items.
2	**Binary Search** Binary search requires items to be in sorted order but its worst execution time is constant and is much faster than linear search.
3	**Interpolation Search** Interpolation search requires items to be in sorted order but its worst execution time is O(n) where n is the number of items and it is much faster than linear search.

DSA using Java - Sorting techniques

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are numerical or lexicographical order.

Importance of sorting lies in the fact that data searching can be optimized to a very high level if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Some of the examples of sorting in real life scenarios are following.

- **Telephone Directory** – Telephone directory keeps telephone no. of people sorted on their names. So that names can be searched.
- **Dictionary** – Dictionary keeps words in alphabetical order so that searching of any work becomes easy.

Types of Sorting

Following is the list of popular sorting algorithms and their comparison.

Sr.No	Technique & Description
1	**Bubble Sort** Bubble sort is simple to understand and implement algorithm but is very poor in performance.
2	**Selection Sort** Selection sort as name specifies use the technique to select the required item and prepare sorted array accordingly.

3	**Insertion Sort** Insertion sort is a variation of selection sort.
4	**Shell Sort** Shell sort is an efficient version of insertion sort.
5	**Quick Sort** Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
6	**Sorting Objects** Java objects can be sorted easily using java.util.Arrays.sort()

DSA using Java - Recursion

Overview

Recursion refers to a technique in a programming language where a function calls itself. The function which calls itself is called a recursive method.

Characteristics

A recursive function must posses the following two characteristics

- Base Case(s)
- Set of rules which leads to base case after reducing the cases.

Recursive Factorial

Factorial is one of the classical example of recursion. Factorial is a non-negative number satisfying following conditions.

- 0! = 1
- 1! = 1
- n! = n * n-1!

Factorial is represented by "!". Here Rule 1 and Rule 2 are base cases and Rule 3 are factorial rules.

As an example, 3! = 3 x 2 x 1 = 6

```
private int factorial(int n){
   //base case
   if(n == 0){
      return 1;
   }else{
      return n * factorial(n-1);
   }
}
```

Recursive Fibonacci Series

Fibonacci Series is another classical example of recursion. Fibonacci series a series of integers satisfying following conditions.

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$

Fibonacci is represented by "F". Here Rule 1 and Rule 2 are base cases and Rule 3 are fibonnacci rules.

As an example, F_5 = 0 1 1 2 3

Demo Program

RecursionDemo.java

```
package com.tutorialspoint.algorithm;

public class RecursionDemo {
   public static void main(String[] args){
      RecursionDemo recursionDemo = new RecursionDemo();
      int n = 5;
      System.out.println("Factorial of " + n + ": "
         + recursionDemo.factorial(n));
      System.out.print("Fibbonacci of " + n + ": ");
      for(int i=0;i<n;i++){
```

```
            System.out.print(recursionDemo.fibbonacci(i) +" ");
         }
      }

      private int factorial(int n){
         //base case
         if(n == 0){
            return 1;
         }else{
            return n * factorial(n-1);
         }
      }

      private int fibbonacci(int n){
         if(n ==0){
            return 0;
         }
         else if(n==1){
            return 1;
         }
         else {
            return (fibbonacci(n-1) + fibbonacci(n-2));
         }
      }
   }
```

If we compile and run the above program then it would produce following result –

```
Factorial of 5: 120
Fibbonacci of 5: 0 1 1 2 3
```