Program: Write a program to implement Linear search or Sequential search algorithm.

Linear search or sequential search is a method for finding a particular value in a list, that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

Linear search is the simplest search algorithm. For a list with n items, the best case is when the value is equal to the first element of the list, in which case only one comparison is needed. The worst case is when the value is not in the list (or occurs only once at the end of the list), in which case n comparisons are needed.

The worst case performance scenario for a linear search is that it has to loop through the entire collection, either because the item is the last one, or because the item is not found. In other words, if you have N items in your collection, the worst case scenario to find an item is N iterations. In Big O Notation it is O(N). The speed of search grows linearly with the number of items within your collection.

Linear searches don't require the collection to be sorted.

```
package com.java2novice.algos;
    public class MyLinearSearch {
    public static int linerSearch(int[] arr, int key){
       int size = arr.length;
       for(int i=0;i<size;i++){
          if(arr[i] == key){
             return i;
          }
       }
       return -1;
   }

   public static void main(String a[]){
       int[] arr1= {23,45,21,55,234,1,34,90};
       int searchKey = 34;
       System.out.println("Key "+searchKey+" found at index: "+linerSearch(arr1, searchKey));
       int[] arr2= {123,445,421,595,2134,41,304,190};
       searchKey = 421;
       System.out.println("Key "+searchKey+" found at index: "+linerSearch(arr2, searchKey));
   }
}
```

Program: Implement Binary search in java using divide and conquer technique.

A binary search or half-interval search algorithm finds the position of a specified value (the input "key") within a sorted array. In each step, the algorithm compares the input key value with the key value of the middle element of the array. If the keys match, then a matching element has been found so its index, or position, is returned. Otherwise, if the sought key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the input key is greater, on the sub-array to the right. If the remaining array to be searched is reduced to zero, then the key cannot be found in the array and a special "Not found" indication is returned.

Every iteration eliminates half of the remaining possibilities. This makes binary searches very efficient - even for large collections.

Binary search requires a sorted collection. Also, binary searching can only be applied to a collection that allows random access (indexing).

Worst case performance: O(log n)

Best case performance: O(1)

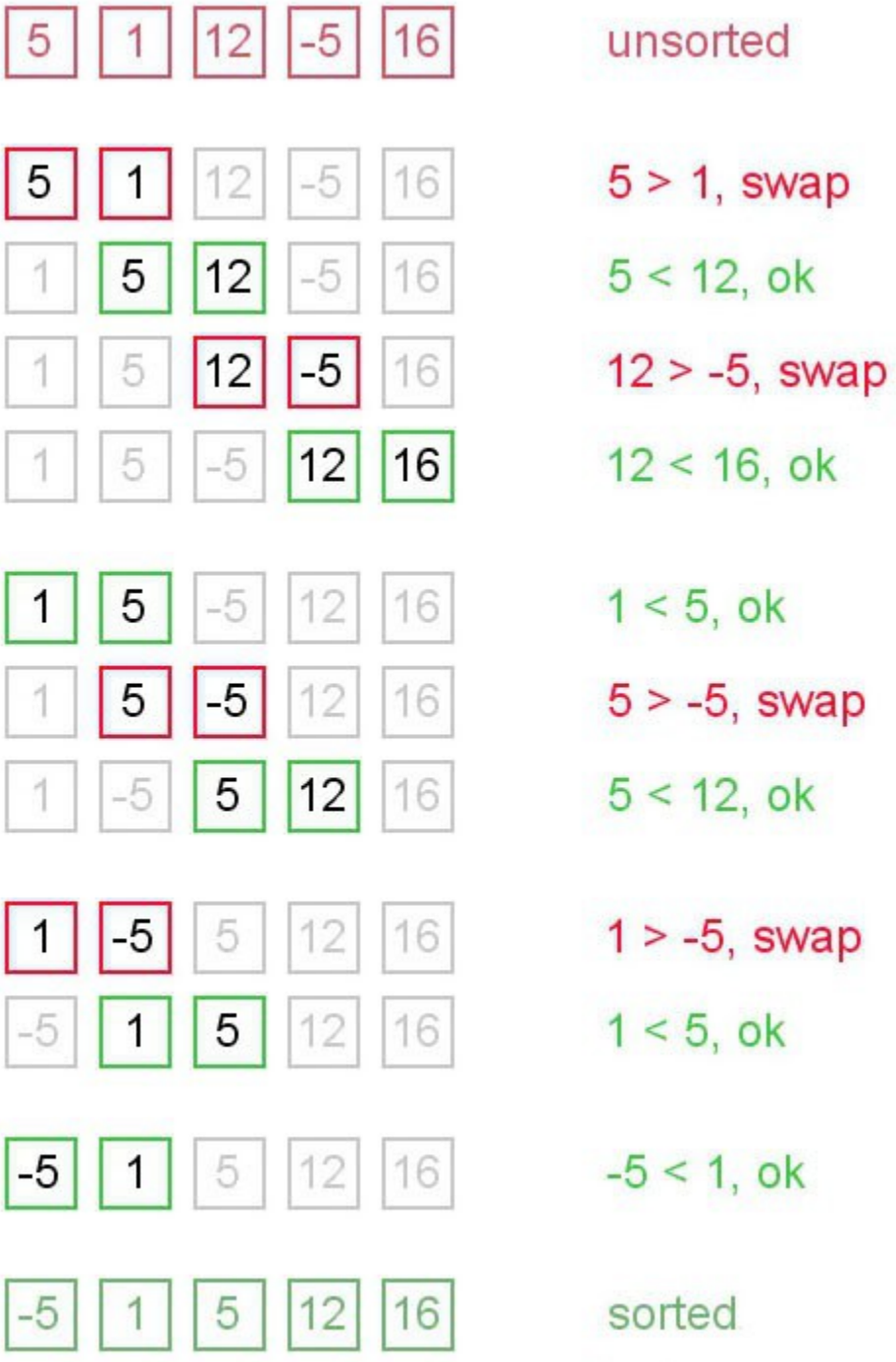
```
package com.java2novice.algos;

public class MyBinarySearch {

   public int binarySearch(int[] inputArr, int key) {

      int start = 0;
      int end = inputArr.length - 1;
      while (start <= end) {
         int mid = (start + end) / 2;
         if (key == inputArr[mid]) {
            return mid;
         }
         if (key < inputArr[mid]) {
            end = mid - 1;
         } else {
            start = mid + 1;
         }
      }
      return -1;
   }

   public static void main(String[] args) {

      MyBinarySearch mbs = new MyBinarySearch();
      int[] arr = {2, 4, 6, 8, 10, 12, 14, 16};
      System.out.println("Key 14's position: "+mbs.binarySearch(arr, 14));
      int[] arr1 = {6,34,78,123,432,900};
      System.out.println("Key 432's position: "+mbs.binarySearch(arr1, 432));
   }
}
```

Program: Implement bubble sort in java.

Bubble sort, also referred to as sinking sort, is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort. Although the algorithm is simple, most of the other sorting algorithms are more efficient for large lists.

Bubble sort has worst-case and average complexity both O(n2), where n is the number of items being sorted. There exist many sorting algorithms with substantially better worst-case or average complexity of O(n log n). Even other O(n2) sorting algorithms, such as insertion sort, tend to have better performance than bubble sort. Therefore, bubble sort is not a practical sorting algorithm when n is large.Performance of bubble sort over an

already-sorted list (best-case) is O(n).



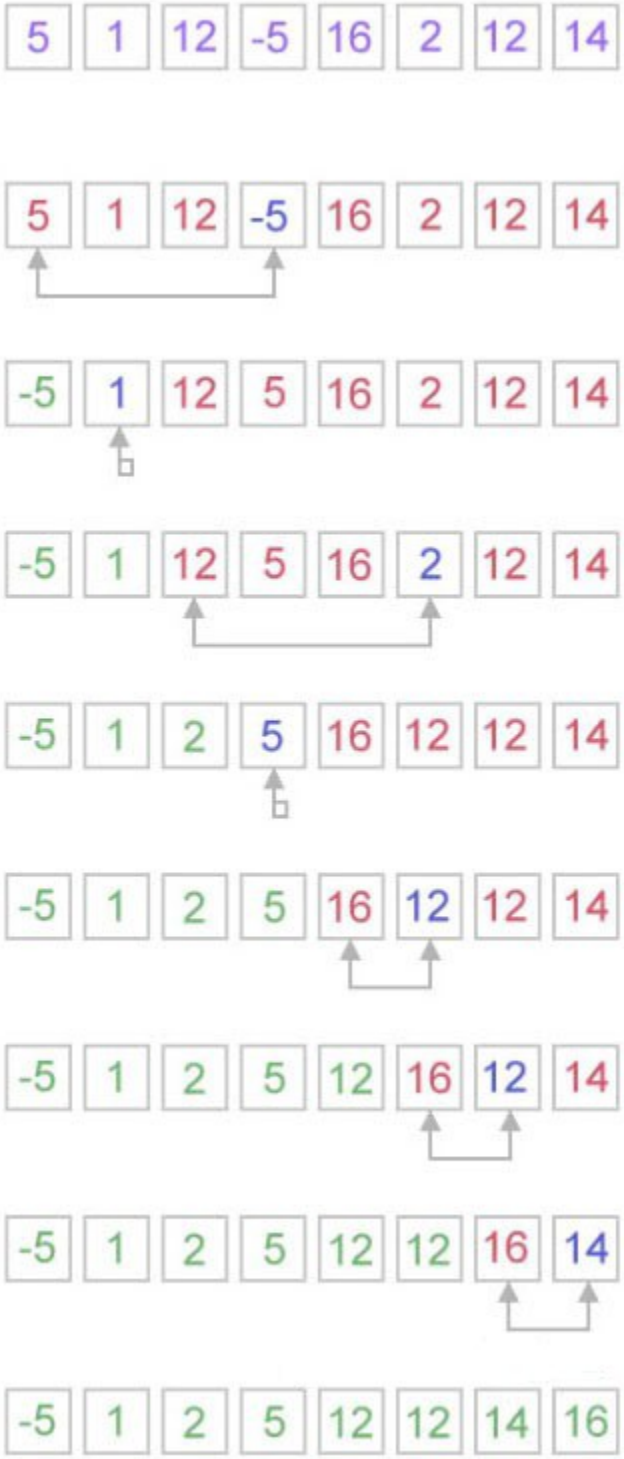

```
package com.java2novice.algos;

public class MyBubbleSort {

    // logic to sort the elements
    public static void bubble_srt(int array[]) {
        int n = array.length;
        int k;
        for (int m = n; m >= 0; m--) {
            for (int i = 0; i < n - 1; i++) {
                k = i + 1;
```

```
            if (array[i] > array[k]) {
                swapNumbers(i, k, array);
            }
        }
        printNumbers(array);
    }
}

private static void swapNumbers(int i, int j, int[] array) {

    int temp;
    temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}

private static void printNumbers(int[] input) {

    for (int i = 0; i < input.length; i++) {
        System.out.print(input[i] + ", ");
    }
    System.out.println("\n");
}

public static void main(String[] args) {
    int[] input = { 4, 2, 9, 6, 23, 12, 34, 0, 1 };
    bubble_srt(input);

}
}
```

Program: Implement selection sort in java.

The selection sort is a combination of searching and sorting. During each pass, the unsorted element with the smallest (or largest) value is moved to its proper position in the array. The number of times the sort passes through the array is one less than the number of items in the array. In the selection sort, the inner loop finds the next smallest (or largest) value and the outer loop places that value into its proper location.

Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array. Selecting the lowest element requires scanning all n elements (this takesn − 1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining n − 1 elements and so on, for (n − 1) + (n − 2) + ... + 2 + 1 = n(n − 1) / 2 ∈ Θ(n2) comparisons. Each of these scans requires one swap for n − 1 elements.




```
package com.java2novice.algos;

public class MySelectionSort {

    public static int[] doSelectionSort(int[] arr){

        for (int i = 0; i < arr.length - 1; i++)
        {
            int index = i;
            for (int j = i + 1; j < arr.length; j++)
                if (arr[j] < arr[index])
```

```
                index = j;

            int smallerNumber = arr[index];
            arr[index] = arr[i];
            arr[i] = smallerNumber;
        }
        return arr;
    }

    public static void main(String a[]){

        int[] arr1 = {10,34,2,56,7,67,88,42};
        int[] arr2 = doSelectionSort(arr1);
        for(int i:arr2){
            System.out.print(i);
            System.out.print(", ");
        }
    }
}
```

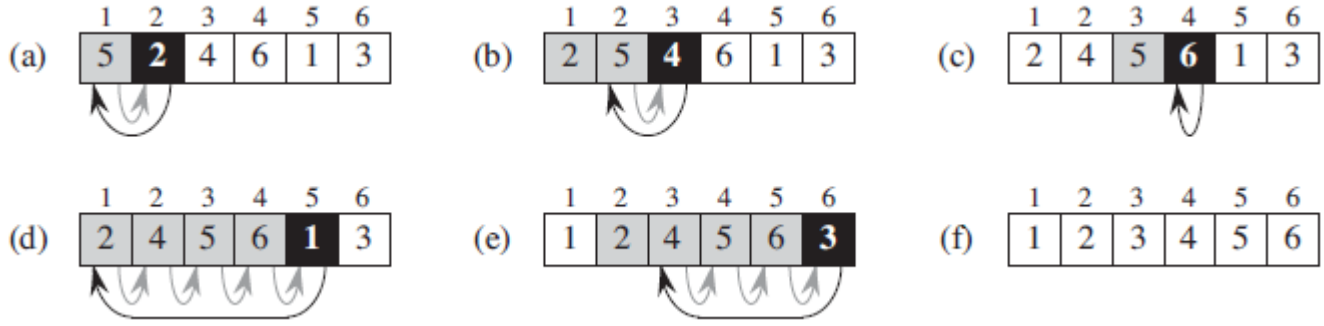
Program: Implement insertion sort in java.

Insertion sort is a simple sorting algorithm, it builds the final sorted array one item at a time. It is much less efficient on large lists than other sort algorithms.

Advantages of Insertion Sort:

1) It is very simple.
2) It is very efficient for small data sets.
3) It is stable; i.e., it does not change the relative order of elements with equal keys.
4) In-place; i.e., only requires a constant amount O(1) of additional memory space.

Insertion sort iterates through the list by consuming one input element at each repetition, and growing a sorted output list. On a repetition, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.



The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., Θ(n)). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array. The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(n2)). The

average case is also quadratic, which makes insertion sort impractical for sorting large arrays. However, insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than quicksort; indeed, good quicksort implementations use insertion sort for arrays smaller than a certain threshold, also when arising as subproblems; the exact threshold must be determined experimentally and depends on the machine, but is commonly around ten.

```
public class MyInsertionSort {
   public static void main(String a[]){
      int[] arr1 = {10,34,2,56,7,67,88,42};
      int[] arr2 = doInsertionSort(arr1);
      for(int i:arr2){
         System.out.print(i);
         System.out.print(", ");
      }
   }

   public static int[] doInsertionSort(int[] input){

      int temp;
      for (int i = 1; i < input.length; i++) {
         for(int j = i ; j > 0 ; j--){
            if(input[j] < input[j-1]){
               temp = input[j];
               input[j] = input[j-1];
               input[j-1] = temp;
            }
         }
      }
      return input;
   }
}
```

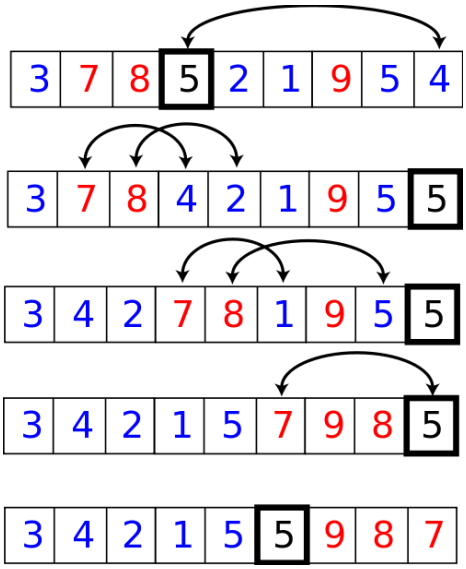
Program: Implement quick sort in java.

Quicksort or partition-exchange sort, is a fast sorting algorithm, which is using divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.

Steps to implement Quick sort:

1) Choose an element, called pivot, from the list. Generally pivot can be the middle index element.
2) Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3) Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

The complexity of quick sort in the average case is $\Theta(n \log(n))$ and in the worst case is $\Theta(n2)$.




```
package com.java2novice.sorting;

public class MyQuickSort {

    private int array[];
    private int length;

    public void sort(int[] inputArr) {

        if (inputArr == null || inputArr.length == 0) {
            return;
        }
        this.array = inputArr;
        length = inputArr.length;
        quickSort(0, length - 1);
    }

    private void quickSort(int lowerIndex, int higherIndex) {

        int i = lowerIndex;
        int j = higherIndex;
        // calculate pivot number, I am taking pivot as middle index number
        int pivot = array[lowerIndex+(higherIndex-lowerIndex)/2];
        // Divide into two arrays
        while (i <= j) {
            /**
             * In each iteration, we will identify a number from left side which
             * is greater then the pivot value, and also we will identify a number
             * from right side which is less then the pivot value. Once the search
             * is done, then we exchange both numbers.
             */
            while (array[i] < pivot) {
                i++;
            }
```

```
            while (array[j] > pivot) {
                j--;
            }
            if (i <= j) {
                exchangeNumbers(i, j);
                //move index to next position on both sides
                i++;
                j--;
            }
        }
        // call quickSort() method recursively
        if (lowerIndex < j)
            quickSort(lowerIndex, j);
        if (i < higherIndex)
            quickSort(i, higherIndex);
    }
    private void exchangeNumbers(int i, int j) {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
    public static void main(String a[]){

        MyQuickSort sorter = new MyQuickSort();
        int[] input = {24,2,45,20,56,75,2,56,99,53,12};
        sorter.sort(input);
        for(int i:input){
            System.out.print(i);
            System.out.print(" ");
        }
    }
}
```

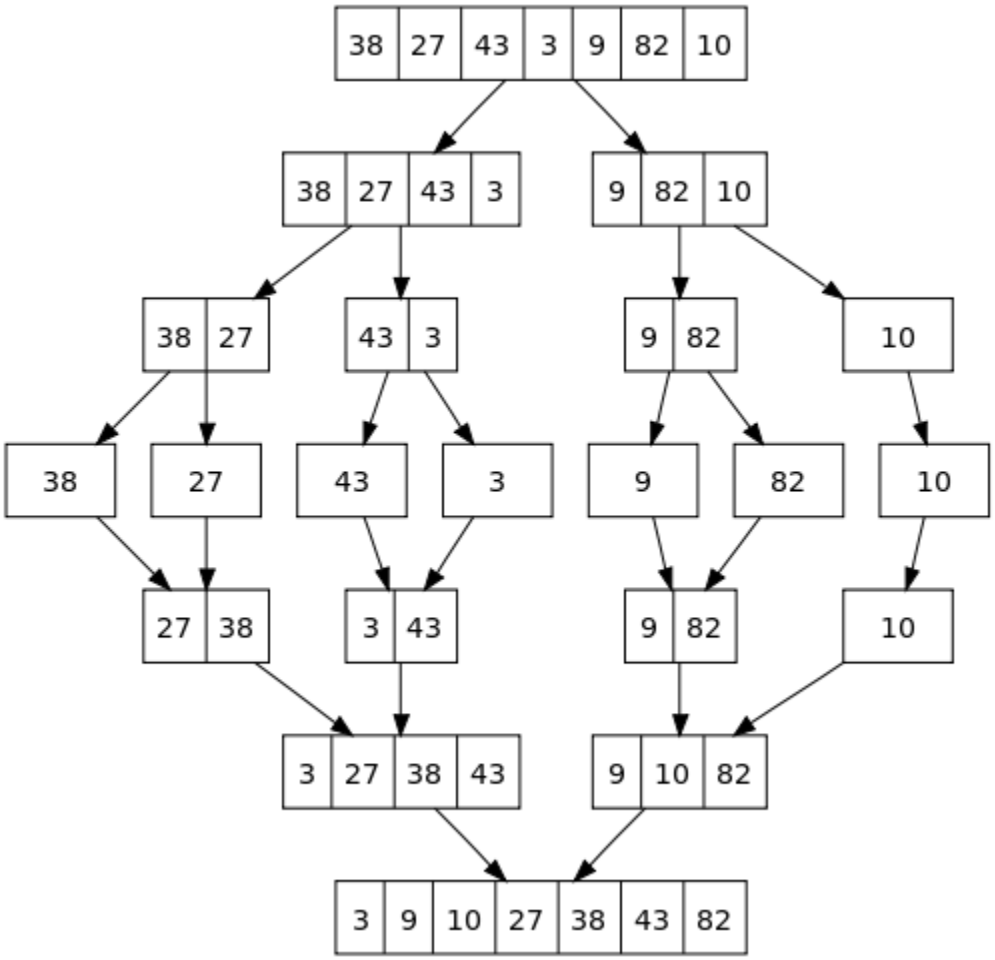
Program: Implement merge sort in java.

Merge sort is a divide and conquer algorithm.

Steps to implement Merge Sort:

1) Divide the unsorted array into n partitions, each partition contains 1 element. Here the one element is considered as sorted.
2) Repeatedly merge partitioned units to produce new sublists until there is only 1 sublist remaining. This will be the sorted list at the end.

Merge sort is a fast, stable sorting routine with guaranteed O(n*log(n)) efficiency. When sorting arrays, merge sort requires additional scratch space proportional to the size of the input array. Merge sort is relatively simple to code and offers performance typically only slightly below that of quicksort.




```
package com.java2novice.sorting;

public class MyMergeSort {

    private int[] array;
    private int[] tempMergArr;
    private int length;

    public static void main(String a[]){

        int[] inputArr = {45,23,11,89,77,98,4,28,65,43};
        MyMergeSort mms = new MyMergeSort();
        mms.sort(inputArr);
        for(int i:inputArr){
            System.out.print(i);
            System.out.print(" ");
        }
    }

    public void sort(int inputArr[]) {
        this.array = inputArr;
        this.length = inputArr.length;
        this.tempMergArr = new int[length];
        doMergeSort(0, length - 1);
```

```
    }

    private void doMergeSort(int lowerIndex, int higherIndex) {

        if (lowerIndex < higherIndex) {
            int middle = lowerIndex + (higherIndex - lowerIndex) / 2;
            // Below step sorts the left side of the array
            doMergeSort(lowerIndex, middle);
            // Below step sorts the right side of the array
            doMergeSort(middle + 1, higherIndex);
            // Now merge both sides
            mergeParts(lowerIndex, middle, higherIndex);
        }
    }

    private void mergeParts(int lowerIndex, int middle, int higherIndex) {

        for (int i = lowerIndex; i <= higherIndex; i++) {
            tempMergArr[i] = array[i];
        }
        int i = lowerIndex;
        int j = middle + 1;
        int k = lowerIndex;
        while (i <= middle && j <= higherIndex) {
            if (tempMergArr[i] <= tempMergArr[j]) {
                array[k] = tempMergArr[i];
                i++;
            } else {
                array[k] = tempMergArr[j];
                j++;
            }
            k++;
        }
        while (i <= middle) {
            array[k] = tempMergArr[i];
            k++;
            i++;
        }
    }
}
```

Linked List Data Structure

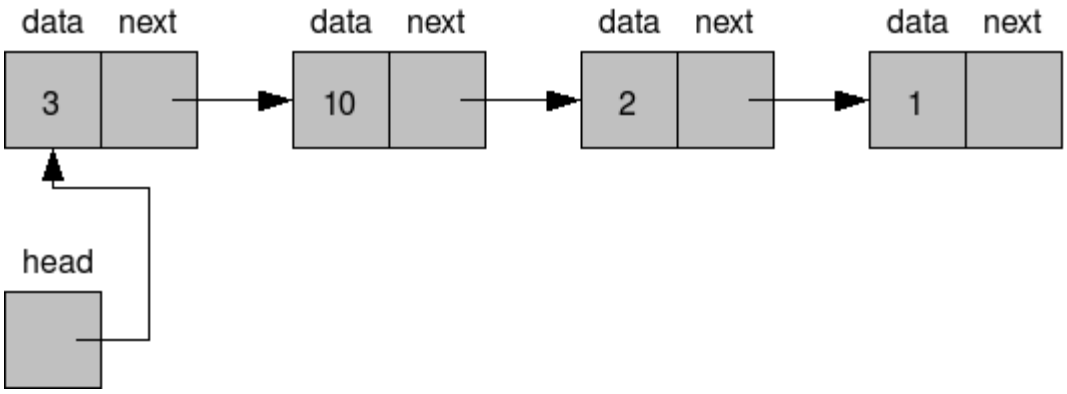
A linked list is a data structure consisting of a group of nodes which together represent a sequence. Each node is composed of a data and a link or reference to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence.The last node is linked to a terminator used to signify the end of the list.

Linked lists are the simplest and most common data structures. They can be used to implement several other abstract data types, including lists, stacks, queues, associative arrays, and S-expressions, etc.

The benefits of a linked list over a conventional array is that the linked list elements can easily be inserted or

removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk. Linked lists allow insertion and removal of nodes at any point in the list.

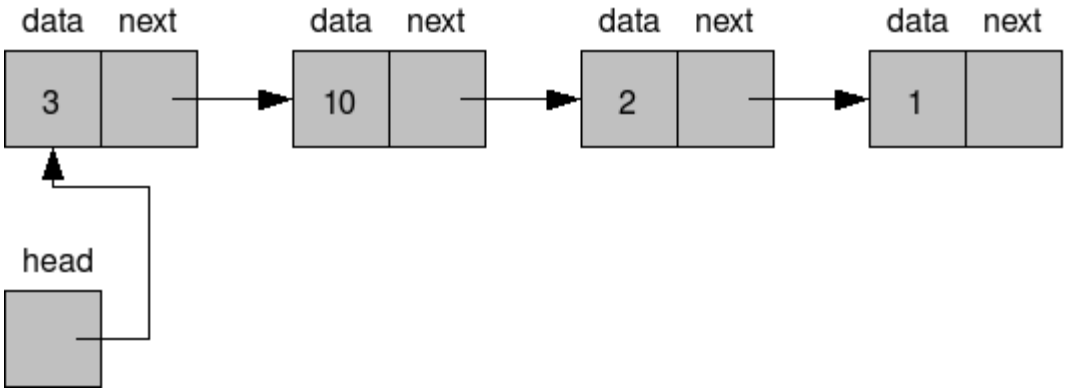
On the other hand, simple linked lists do not allow random access to the data, or by using indexing. Thus, many basic operations like obtaining the last node of the list, or finding a node with required data, or locating the place where a new node should be inserted, may require scanning most of the list elements.



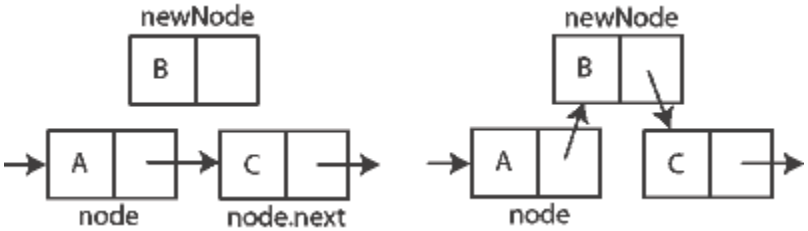
Singly linked list implementation

Singly Linked Lists are a type of data structure. It is a type of list. In a singly linked list each node in the list stores the contents of the node and a pointer or reference to the next node in the list. It does not store any pointer or reference to the previous node. It is called a singly linked list because each node only has a single link to another node. To store a single linked list, you only need to store a reference or pointer to the first node in that list. The last node has a pointer to nothingness to indicate that it is the last node.

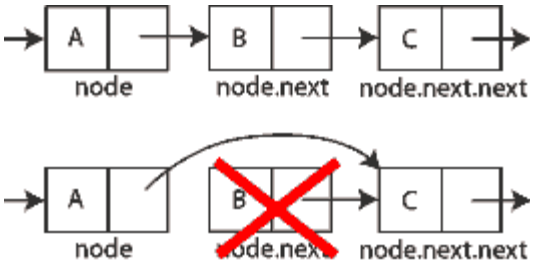
Here is the pictorial view of singly linked list:



Here is the pictorial view of inserting an element in the middle of a singly linked list:



Here is the pictorial view of deleting an element in the middle of a singly linked list:



Below shows the java implementation of singly linked list:

```
package com.java2novice.ds.linkedlist;

public class SinglyLinkedListImpl<T> {

    private Node<T> head;
    private Node<T> tail;

    public void add(T element){

        Node<T> nd = new Node<T>();
        nd.setValue(element);
        System.out.println("Adding: "+element);
        /**
         * check if the list is empty
         */
        if(head == null){
            //since there is only one element, both head and
            //tail points to the same object.
            head = nd;
            tail = nd;
        } else {
            //set current tail next link to new node
            tail.setNextRef(nd);
            //set tail as newly created node
            tail = nd;
        }
    }

    public void addAfter(T element, T after){

        Node<T> tmp = head;
        Node<T> refNode = null;
        System.out.println("Traversing to all nodes..");
        /**
         * Traverse till given element
         */
        while(true){
            if(tmp == null){
                break;
            }
            if(tmp.compareTo(after) == 0){
                //found the target node, add after this node
```

```
            refNode = tmp;
            break;
        }
        tmp = tmp.getNextRef();
    }
    if(refNode != null){
        //add element after the target node
        Node<T> nd = new Node<T>();
        nd.setValue(element);
        nd.setNextRef(tmp.getNextRef());
        if(tmp == tail){
            tail = nd;
        }
        tmp.setNextRef(nd);

    } else {
        System.out.println("Unable to find the given element...");
    }
}

public void deleteFront(){

    if(head == null){
        System.out.println("Underflow...");
    }
    Node<T> tmp = head;
    head = tmp.getNextRef();
    if(head == null){
        tail = null;
    }
    System.out.println("Deleted: "+tmp.getValue());
}

public void deleteAfter(T after){

    Node<T> tmp = head;
    Node<T> refNode = null;
    System.out.println("Traversing to all nodes..");
    /**
     * Traverse till given element
     */
    while(true){
        if(tmp == null){
            break;
        }
        if(tmp.compareTo(after) == 0){
            //found the target node, add after this node
            refNode = tmp;
            break;
        }
        tmp = tmp.getNextRef();
```

```
        }
        if(refNode != null){
            tmp = refNode.getNextRef();
            refNode.setNextRef(tmp.getNextRef());
            if(refNode.getNextRef() == null){
                tail = refNode;
            }
            System.out.println("Deleted: "+tmp.getValue());
        } else {
            System.out.println("Unable to find the given element...");
        }
    }

    public void traverse(){

        Node<T> tmp = head;
        while(true){
            if(tmp == null){
                break;
            }
            System.out.println(tmp.getValue());
            tmp = tmp.getNextRef();
        }
    }

    public static void main(String a[]){
        SinglyLinkedListImpl<Integer> sl = new SinglyLinkedListImpl<Integer>();
        sl.add(3);
        sl.add(32);
        sl.add(54);
        sl.add(89);
        sl.addAfter(76, 54);
        sl.deleteFront();
        sl.deleteAfter(76);
        sl.traverse();

    }
}

class Node<T> implements Comparable<T> {

    private T value;
    private Node<T> nextRef;

    public T getValue() {
        return value;
    }
    public void setValue(T value) {
        this.value = value;
    }
    public Node<T> getNextRef() {
```

```
        return nextRef;
    }
    public void setNextRef(Node<T> ref) {
        this.nextRef = ref;
    }
    @Override
    public int compareTo(T arg) {
        if(arg == this.value){
            return 0;
        } else {
            return 1;
        }
    }
}
```

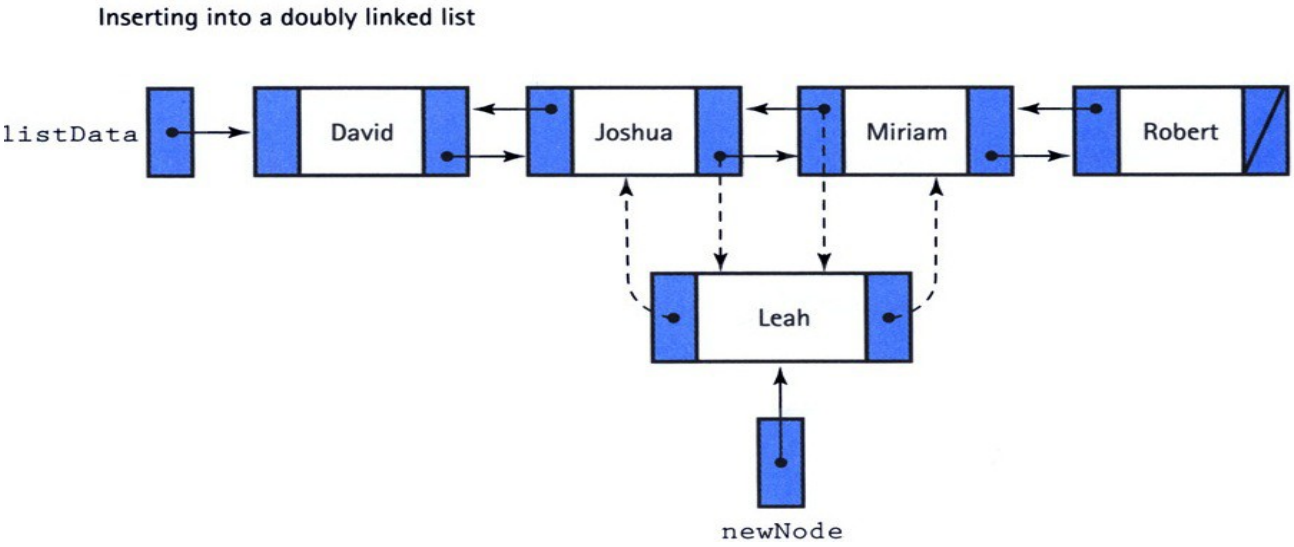
Doubly linked list implementation

A doubly-linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes. The beginning and ending nodes previous and next links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list. If there is only one sentinel node, then the list is circularly linked via the sentinel node. It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.

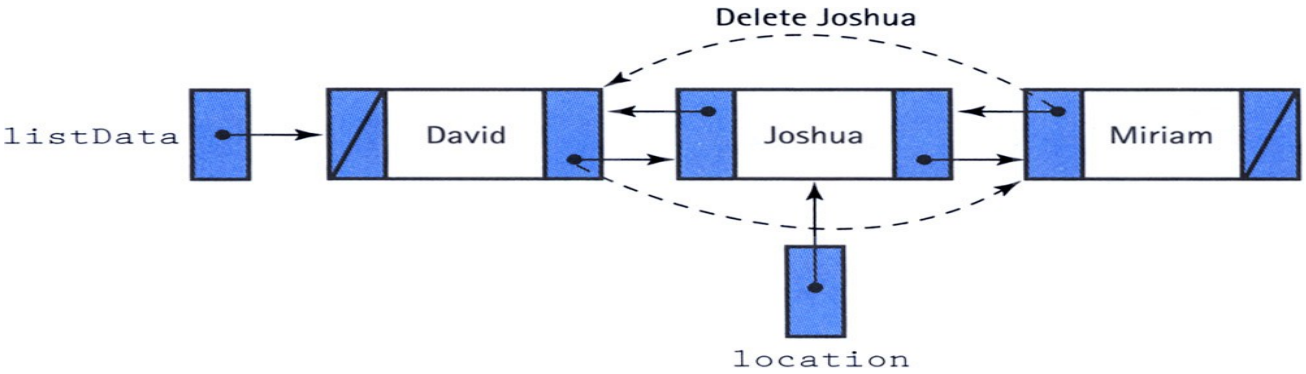
Here is the pictorial view of doubly linked list:

The two node links allow traversal of the list in either direction. While adding or removing a node in a doubly-linked list requires changing more links than the same operations on a singly linked list, the operations are simpler and potentially more efficient, because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified.

Here is the pictorial view of inserting an element in the middle of a doubly linked list:



Here is the pictorial view of deleting an element in the middle of a doubly linked list:



Below shows the java implementation of doubly linked list:

```
package com.java2novice.ds.linkedlist;
import java.util.NoSuchElementException;
public class DoublyLinkedListImpl<E> {
    private Node head;
    private Node tail;
    private int size;

    public DoublyLinkedListImpl() {
        size = 0;
    }
    /**
     * this class keeps track of each element information
     * @author java2novice
     *
     */
    private class Node {
        E element;
        Node next;
        Node prev;

        public Node(E element, Node next, Node prev) {
            this.element = element;
            this.next = next;
            this.prev = prev;
        }
    }
    /**
     * returns the size of the linked list
     * @return
     */
    public int size() { return size; }

    /**
     * return whether the list is empty or not
     * @return
     */
    public boolean isEmpty() { return size == 0; }
```

```
/**
 * adds element at the starting of the linked list
 * @param element
 */
public void addFirst(E element) {
    Node tmp = new Node(element, head, null);
    if(head != null ) {head.prev = tmp;}
    head = tmp;
    if(tail == null) { tail = tmp;}
    size++;
    System.out.println("adding: "+element);
}

/**
 * adds element at the end of the linked list
 * @param element
 */
public void addLast(E element) {

    Node tmp = new Node(element, null, tail);
    if(tail != null) {tail.next = tmp;}
    tail = tmp;
    if(head == null) { head = tmp;}
    size++;
    System.out.println("adding: "+element);
}

/**
 * this method walks forward through the linked list
 */
public void iterateForward(){

    System.out.println("iterating forward..");
    Node tmp = head;
    while(tmp != null){
        System.out.println(tmp.element);
        tmp = tmp.next;
    }
}

/**
 * this method walks backward through the linked list
 */
public void iterateBackward(){

    System.out.println("iterating backword..");
    Node tmp = tail;
    while(tmp != null){
        System.out.println(tmp.element);
        tmp = tmp.prev;
```

```
        }
    }

    /**
     * this method removes element from the start of the linked list
     * @return
     */
    public E removeFirst() {
        if (size == 0) throw new NoSuchElementException();
        Node tmp = head;
        head = head.next;
        head.prev = null;
        size--;
        System.out.println("deleted: "+tmp.element);
        return tmp.element;
    }

    /**
     * this method removes element from the end of the linked list
     * @return
     */
    public E removeLast() {
        if (size == 0) throw new NoSuchElementException();
        Node tmp = tail;
        tail = tail.prev;
        tail.next = null;
        size--;
        System.out.println("deleted: "+tmp.element);
        return tmp.element;
    }

    public static void main(String a[]){

        DoublyLinkedListImpl<Integer> dll = new DoublyLinkedListImpl<Integer>();
        dll.addFirst(10);
        dll.addFirst(34);
        dll.addLast(56);
        dll.addLast(364);
        dll.iterateForward();
        dll.removeFirst();
        dll.removeLast();
        dll.iterateBackward();
    }
}
```

Stacks

A Stack is an abstract data type or collection where in Push,the addition of data elements to the collection, and Pop, the removal of data elements from the collection, are the major operations performed on the collection. The Push and Pop operations are performed only at one end of the Stack which is referred to as the 'top of the stack'.

In other words,a Stack can be simply defined as Last In First Out (LIFO) data structure,i.e.,the last element

added at the top of the stack(In) should be the first element to be removed(Out) from the stack.

Stack introduction & implementation

A Stack is an abstract data type or collection where in Push,the addition of data elements to the collection, and Pop, the removal of data elements from the collection, are the major operations performed on the collection. The Push and Pop operations are performed only at one end of the Stack which is referred to as the 'top of the stack'.

In other words,a Stack can be simply defined as Last In First Out (LIFO) data structure,i.e.,the last element added at the top of the stack(In) should be the first element to be removed(Out) from the stack.

Stack Operations:

Push: A new entity can be added to the top of the collection.
Pop: An entity will be removed from the top of the collection.
Peek or Top: Returns the top of the entity with out removing it.

Overflow State: A stack may be implemented to have a bounded capacity. If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state.

Underflow State: The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items or results in an empty stack, but, if the stack is empty, it goes into underflow state, which means no items are present in stack to be removed.

A stack is a restricted data structure, because only a small number of operations are performed on it. The nature of the pop and push operations also means that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition. Therefore, the lower elements are those that have been on the stack the longest.

Efficiency of Stacks

In the stack, the elements can be push or pop one at a time in constant O(1) time. That is, the time is not dependent on how many items are in the stack and is therefore very quick. No comparisons or moves are necessary.

```
package com.java2novice.ds.stack;

public class MyStackImpl {

    private int stackSize;
    private int[] stackArr;
    private int top;

    /**
     * constructor to create stack with size
     * @param size
     */
    public MyStackImpl(int size) {
        this.stackSize = size;
        this.stackArr = new int[stackSize];
        this.top = -1;
    }

    /**
     * This method adds new entry to the top
```

```
 * of the stack
 * @param entry
 * @throws Exception
 */
public void push(int entry) throws Exception {
   if(this.isStackFull()){
      throw new Exception("Stack is already full. Can not add element.");
   }
   System.out.println("Adding: "+entry);
   this.stackArr[++top] = entry;
}

/**
 * This method removes an entry from the
 * top of the stack.
 * @return
 * @throws Exception
 */
public int pop() throws Exception {
   if(this.isStackEmpty()){
      throw new Exception("Stack is empty. Can not remove element.");
   }
   int entry = this.stackArr[top--];
   System.out.println("Removed entry: "+entry);
   return entry;
}

/**
 * This method returns top of the stack
 * without removing it.
 * @return
 */
public int peek() {
   return stackArr[top];
}

/**
 * This method returns true if the stack is
 * empty
 * @return
 */
public boolean isStackEmpty() {
   return (top == -1);
}

/**
 * This method returns true if the stack is full
 * @return
 */
public boolean isStackFull() {
   return (top == stackSize - 1);
```

```
    }

    public static void main(String[] args) {
        MyStackImpl stack = new MyStackImpl(5);
        try {
            stack.push(4);
            stack.push(8);
            stack.push(3);
            stack.push(89);
            stack.pop();
            stack.push(34);
            stack.push(45);
            stack.push(78);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        try {
            stack.pop();
            stack.pop();
            stack.pop();
            stack.pop();
            stack.pop();
            stack.pop();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Java Dynamic Stack Implementation

In this example, the stack will never comes to stack overflow case, beacuse its capacity will keep increases as it reaches to max capacity, so it is very dynamic in capacity.

```
package com.java2novice.ds.stack;

public class MyDynamicStack {

    private int stackSize;
    private int[] stackArr;
    private int top;

    /**
     * constructor to create stack with size
     * @param size
     */
    public MyDynamicStack(int size) {
        this.stackSize = size;
        this.stackArr = new int[stackSize];
        this.top = -1;
    }
```

```
/**
 * This method adds new entry to the top
 * of the stack
 * @param entry
 * @throws Exception
 */
public void push(int entry){
   if(this.isStackFull()){
      System.out.println(("Stack is full. Increasing the capacity."));
      this.increaseStackCapacity();
   }
   System.out.println("Adding: "+entry);
   this.stackArr[++top] = entry;
}

/**
 * This method removes an entry from the
 * top of the stack.
 * @return
 * @throws Exception
 */
public int pop() throws Exception {
   if(this.isStackEmpty()){
      throw new Exception("Stack is empty. Can not remove element.");
   }
   int entry = this.stackArr[top--];
   System.out.println("Removed entry: "+entry);
   return entry;
}

/**
 * This method returns top of the stack
 * without removing it.
 * @return
 */
public long peek() {
   return stackArr[top];
}

private void increaseStackCapacity(){

   int[] newStack = new int[this.stackSize*2];
   for(int i=0;i<stackSize;i++){
      newStack[i] = this.stackArr[i];
   }
   this.stackArr = newStack;
   this.stackSize = this.stackSize*2;
}

/**
 * This method returns true if the stack is
```

```
     * empty
     * @return
     */
    public boolean isStackEmpty() {
        return (top == -1);
    }

    /**
     * This method returns true if the stack is full
     * @return
     */
    public boolean isStackFull() {
        return (top == stackSize - 1);
    }

    public static void main(String[] args) {
        MyDynamicStack stack = new MyDynamicStack(2);
        for(int i=1;i<10;i++){
            stack.push(i);
        }
        for(int i=1;i<4;i++){
            try {
                stack.pop();
            } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```

Stack implementation using generics bound type.

What happens in case we want to create a generic stack which should allow us to create a stack for any kind of object. For instance, I want to create a stack which should accept only string objects. Similarly I want to create a stack which accepts only integers. How can I fulfill this requirement? We can implement this using generics bound type. Using this you can add group of objects from the same family as well. Incase if your stack wants to allow the objects extended by a super class, then you need to replace Object with your super class at line no:3 , in the below example.

```
package com.java2novice.ds.stack;

public class MyGenericsStack<T extends Object> {

    private int stackSize;
    private T[] stackArr;
    private int top;

    /**
     * constructor to create stack with size
     * @param size
     */
    @SuppressWarnings("unchecked")
```

```
public MyGenericsStack(int size) {
   this.stackSize = size;
   this.stackArr = (T[]) new Object[stackSize];
   this.top = -1;
}

/**
 * This method adds new entry to the top
 * of the stack
 * @param entry
 * @throws Exception
 */
public void push(T entry){
   if(this.isStackFull()){
      System.out.println(("Stack is full. Increasing the capacity."));
      this.increaseStackCapacity();
   }
   System.out.println("Adding: "+entry);
   this.stackArr[++top] = entry;
}

/**
 * This method removes an entry from the
 * top of the stack.
 * @return
 * @throws Exception
 */
public T pop() throws Exception {
   if(this.isStackEmpty()){
      throw new Exception("Stack is empty. Can not remove element.");
   }
   T entry = this.stackArr[top--];
   System.out.println("Removed entry: "+entry);
   return entry;
}

/**
 * This method returns top of the stack
 * without removing it.
 * @return
 */
public T peek() {
   return stackArr[top];
}

private void increaseStackCapacity(){

   @SuppressWarnings("unchecked")
   T[] newStack = (T[]) new Object[this.stackSize*2];
   for(int i=0;i<stackSize;i++){
      newStack[i] = this.stackArr[i];
```

```
        }
        this.stackArr = newStack;
        this.stackSize = this.stackSize*2;
    }

    /**
     * This method returns true if the stack is
     * empty
     * @return
     */
    public boolean isStackEmpty() {
        return (top == -1);
    }

    /**
     * This method returns true if the stack is full
     * @return
     */
    public boolean isStackFull() {
        return (top == stackSize - 1);
    }

    public static void main(String a[]){
        MyGenericsStack<String> stringStack = new MyGenericsStack<String>(2);
        stringStack.push("java2novice");
        MyGenericsStack<Integer> integerStack = new MyGenericsStack<Integer>(2);
        integerStack.push(23);
    }
}
```

Queue Data Structure

A queue is a kind of abstract data type or collection in which the entities in the collection are kept in order and the only operations on the collection are the addition of entities to the rear terminal position, called as enqueue, and removal of entities from the front terminal position, called as dequeue. The queue is called as First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed. Often a peek or front operation is also entered, returning the value of the front element without dequeuing it. A queue is an example of a linear data structure, or more abstractly a sequential collection.

Queue introduction & array based implementation

A queue is a kind of abstract data type or collection in which the entities in the collection are kept in order and the only operations on the collection are the addition of entities to the rear terminal position, called as enqueue, and removal of entities from the front terminal position, called as dequeue. The queue is called as First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed. Often a peek or front operation is also entered, returning the value of the front element without dequeuing it. A queue is an example of a linear data

structure, or more abstractly a sequential collection.

Queues provide services in computer science, transport, and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer.

Queue Operations:

enqueue: Adds an item onto the end of the queue.
front: Returns the item at the front of the queue.
dequeue: Removes the item from the front of the queue.

Overflow State: A queue may be implemented to have a bounded capacity. If the queue is full and does not contain enough space to accept an entity to be pushed, the queue is then considered to be in an overflow state.

Underflow State: The dequeue operation removes an item from the top of the queue. A dequeue operation either reveals previously concealed items or results in an empty queue, but, if the queue is empty, it goes into underflow state, which means no items are present in queue to be removed.

Efficiency of Queue

The time needed to add or delete an item is constant and independent of the number of items in the queue. So both addition and deletion can be O(1) operation.

```
package com.java2novice.ds.queue;

public class QueueImpl {

   private int capacity;
   int queueArr[];
   int front = 0;
   int rear = -1;
   int currentSize = 0;

   public QueueImpl(int queueSize){
      this.capacity = queueSize;
      queueArr = new int[this.capacity];
   }

   /**
    * this method adds element at the end of the queue.
    * @param item
    */
   public void enqueue(int item) {
      if (isQueueFull()) {
         System.out.println("Overflow ! Unable to add element: "+item);
      } else {
         rear++;
         if(rear == capacity-1){
            rear = 0;
         }
         queueArr[rear] = item;
         currentSize++;
         System.out.println("Element " + item+ " is pushed to Queue !");
      }
   }
```

```
/**
 * this method removes an element from the top of the queue
 */
public void dequeue() {
    if (isQueueEmpty()) {
        System.out.println("Underflow ! Unable to remove element from Queue");
    } else {
        front++;
        if(front == capacity-1){
            System.out.println("Pop operation done ! removed: "+queueArr[front-1]);
            front = 0;
        } else {
            System.out.println("Pop operation done ! removed: "+queueArr[front-1]);
        }
        currentSize--;
    }
}

/**
 * This method checks whether the queue is full or not
 * @return boolean
 */
public boolean isQueueFull(){
    boolean status = false;
    if (currentSize == capacity){
        status = true;
    }
    return status;
}

/**
 * This method checks whether the queue is empty or not
 * @return
 */
public boolean isQueueEmpty(){
    boolean status = false;
    if (currentSize == 0){
        status = true;
    }
    return status;
}

public static void main(String a[]){

    QueueImpl queue = new QueueImpl(4);
    queue.enqueue(4);
    queue.dequeue();
    queue.enqueue(56);
    queue.enqueue(2);
    queue.enqueue(67);
```

```
        queue.dequeue();
        queue.dequeue();
        queue.enqueue(24);
        queue.dequeue();
        queue.enqueue(98);
        queue.enqueue(45);
        queue.enqueue(23);
        queue.enqueue(435);
    }
}
```

Double-ended queue (Decue) Implementation.

A double-ended queue (dequeue or deque) is an abstract data type that generalizes a queue, for which elements can be added to or removed from either the front or rear. Deque differs from the queue abstract data type or First-In-First-Out List (FIFO), where elements can only be added to one end and removed from the other. This general data class has some possible sub-types:

1) An input-restricted deque is one where deletion can be made from both ends, but insertion can be made at one end only.
2) An output-restricted deque is one where insertion can be made at both ends, but deletion can be made from one end only.

You can use Deque as a stack by making insertion and deletion at the same side. Also you can use Deque as queue by making insetting elements at one end and removing elements at other end.

The common way of deque implementations are by using dynamic array or doubly linked list. Here this example shows the basic implementation of deque using a list, which is basically a dynamic array.

The complexity of Deque operations is O(1), when we not consider overhead of allocation/deallocation of dynamic array size.

```
package com.java2novice.ds.queue;

import java.util.ArrayList;
import java.util.List;

public class DoubleEndedQueueImpl {

    private List<Integer> deque = new ArrayList<Integer>();
```

```
public void insertFront(int item){
   //add element at the beginning of the queue
   System.out.println("adding at front: "+item);
   deque.add(0,item);
   System.out.println(deque);
}

public void insertRear(int item){
   //add element at the end of the queue
   System.out.println("adding at rear: "+item);
   deque.add(item);
   System.out.println(deque);
}

public void removeFront(){
   if(deque.isEmpty()){
      System.out.println("Deque underflow!! unable to remove.");
      return;
   }
   //remove an item from the beginning of the queue
   int rem = deque.remove(0);
   System.out.println("removed from front: "+rem);
   System.out.println(deque);
}

public void removeRear(){
   if(deque.isEmpty()){
      System.out.println("Deque underflow!! unable to remove.");
      return;
   }
   //remove an item from the beginning of the queue
   int rem = deque.remove(deque.size()-1);
   System.out.println("removed from front: "+rem);
   System.out.println(deque);
}

public int peakFront(){
   //gets the element from the front without removing it
   int item = deque.get(0);
   System.out.println("Element at first: "+item);
   return item;
}

public int peakRear(){
   //gets the element from the rear without removing it
   int item = deque.get(deque.size()-1);
   System.out.println("Element at rear: "+item);
   return item;
}
```

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

        DoubleEndedQueueImpl deq = new DoubleEndedQueueImpl();
        deq.insertFront(34);
        deq.insertRear(45);
        deq.removeFront();
        deq.removeFront();
        deq.removeFront();
        deq.insertFront(21);
        deq.insertFront(98);
        deq.insertRear(5);
        deq.insertFront(43);
        deq.removeRear();
    }
}
```

Double-ended queue (Decue) implementation using Doubly linked list.

A double-ended queue (dequeue or deque) is an abstract data type that generalizes a queue, for which elements can be added to or removed from either the front or rear. Deque differs from the queue abstract data type or First-In-First-Out List (FIFO), where elements can only be added to one end and removed from the other. We have given more details on Deque in the previous example

In this page you will see Deque implementation by using double linked list.

```
package com.java2novice.ds.queue;


public class DequeDblLinkedListImpl<T> {

    private Node<T> front;
    private Node<T> rear;

    public void insertFront(T item){
        //add element at the beginning of the queue
        System.out.println("adding at front: "+item);
        Node<T> nd = new Node<T>();
        nd.setValue(item);
        nd.setNext(front);
        if(front != null) front.setPrev(nd);
        if(front == null) rear = nd;
        front = nd;
    }

    public void insertRear(T item){
        //add element at the end of the queue
        System.out.println("adding at rear: "+item);
        Node<T> nd = new Node<T>();
        nd.setValue(item);
        nd.setPrev(rear);
```

```
    if(rear != null) rear.setNext(nd);
    if(rear == null) front = nd;

    rear = nd;
}

public void removeFront(){
    if(front == null){
        System.out.println("Deque underflow!! unable to remove.");
        return;
    }
    //remove an item from the beginning of the queue
    Node<T> tmpFront = front.getNext();
    if(tmpFront != null) tmpFront.setPrev(null);
    if(tmpFront == null) rear = null;
    System.out.println("removed from front: "+front.getValue());
    front = tmpFront;
}

public void removeRear(){
    if(rear == null){
        System.out.println("Deque underflow!! unable to remove.");
        return;
    }
    //remove an item from the beginning of the queue
    Node<T> tmpRear = rear.getPrev();
    if(tmpRear != null) tmpRear.setNext(null);
    if(tmpRear == null) front = null;
    System.out.println("removed from rear: "+rear.getValue());
    rear = tmpRear;
}

public static void main(String a[]){
    DequeDblLinkedListImpl<Integer> deque = new DequeDblLinkedListImpl<Integer>();
    deque.insertFront(34);
    deque.insertFront(67);
    deque.insertFront(29);
    deque.insertFront(765);
    deque.removeFront();
    deque.removeFront();
    deque.removeFront();
    deque.insertRear(43);
    deque.insertRear(83);
    deque.insertRear(84);
    deque.insertRear(546);
    deque.insertRear(356);
    deque.removeRear();
    deque.removeRear();
    deque.removeRear();
    deque.removeRear();
    deque.removeFront();
```

```
        deque.removeFront();
        deque.removeFront();
    }
}

class Node<T>{

    private Node<T> prev;
    private Node<T> next;
    private T value;

    public Node<T> getPrev() {
        return prev;
    }
    public void setPrev(Node<T> prev) {
        this.prev = prev;
    }
    public Node<T> getNext() {
        return next;
    }
    public void setNext(Node<T> next) {
        this.next = next;
    }
    public T getValue() {
        return value;
    }
    public void setValue(T value) {
        this.value = value;
    }
}
```

Priority Queue introduction and Java implementation

A priority queue is an abstract data type, it is like a regular queue or stack data structure, but where additionally each element has a priority associated with it. In a priority queue, an element with high priority is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.

There are a variety of simple ways to implement a priority queue. For instance, one can keep all the elements in an unsorted list. Whenever the highest-priority element is requested, search through all elements for the one with the highest priority. In big O notation: O(1) insertion time, O(n) pull time due to search.

Below example shows basic implementation of priority queue. The condition here is, the element to be inserted should implement Comparable interface, we have taken Integer object, which is already implementing this interface.

```
package com.java2novice.ds.queue;
public class PriorityQueueImpl {
    @SuppressWarnings("rawtypes")
    private Comparable[] pQueue;
    private int index;
```

```
    public PriorityQueueImpl(int capacity){
        pQueue = new Comparable[capacity];
    }

    public void insert(Comparable item ){
        if(index == pQueue.length){
            System.out.println("The priority queue is full!! can not insert.");
            return;
        }
        pQueue[index] = item;
        index++;
        System.out.println("Adding element: "+item);
    }

    @SuppressWarnings("unchecked")
    public Comparable remove(){
        if(index == 0){
            System.out.println("The priority queue is empty!! can not remove.");
            return null;
        }
        int maxIndex = 0;
        // find the index of the item with the highest priority
        for (int i=1; i<index; i++) {
            if (pQueue[i].compareTo (pQueue[maxIndex]) > 0) {
                maxIndex = i;
            }
        }
        Comparable result = pQueue[maxIndex];
        System.out.println("removing: "+result);
        // move the last item into the empty slot
        index--;
        pQueue[maxIndex] = pQueue[index];
        return result;
    }

    public static void main(String a[]){
        PriorityQueueImpl pqi = new PriorityQueueImpl(5);
        pqi.insert(34);
        pqi.insert(23);
        pqi.insert(5);
        pqi.insert(87);
        pqi.insert(32);
        pqi.remove();
        pqi.remove();
        pqi.remove();
        pqi.remove();
        pqi.remove();
    }
}
```