INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal - 500 043, Hyderabad, Telangana

COURSE CONTENT

DATA STRUCTURES LABORATORY								
III Semester: AE / ME / CE / ECE / EEE / CSE / CSE (AI & ML) / CSE (DS) / CSE (CS) / IT								
Course Code	**Category**	**Hours / Week**			**Credits**	**Maximum Marks**		
ACSD11	Core	L	T	P	C	CIA	SEE	Total
		-	-	2	1	40	60	100
Contact Classes: Nil	Tutorial Classes: Nil	Practical Classes: 45			Total Classes: 45			
Prerequisite: Essentials of Problem Solving								

I. COURSE OVERVIEW:

The course covers some of the general-purpose data structures and algorithms, and software development. Topics covered include managing complexity, analysis, static data structures, dynamic data structures and hashing mechanisms. The main objective of the course is to teach the students how to select and design data structures andalgorithms that are appropriate for problems that they might encounter in real life. This course reaches to studentby power point presentations, lecture notes, and lab which involve the problem solving in mathematical and engineering areas.

II. COURSES OBJECTIVES:

The students will try to learn

I. To provide students with skills needed to understand and analyze performance trade-offs of different algorithms / implementations and asymptotic analysis of their running time and memory usage.

II. To provide knowledge of basic abstract data types (ADT) and associated algorithms: stacks, queues, lists, tree, graphs, hashing and sorting, selection and searching.

III. The fundamentals of how to store, retrieve, and process data efficiently.

IV. To provide practice by specifying and implementing these data structures and algorithms in Python.

V. Understand essential for future programming and software engineering courses.

III. COURSE OUTCOMES:

At the end of the course students should be able to:

CO 1	Interpret the complexity of algorithm using the asymptotic notations.
CO 2	Select appropriate searching and sorting technique for a given problem.
CO 3	Construct programs on performing operations on linear and nonlinear data structures for organization ofa data.
CO 4	Make use of linear data structures and nonlinear data structures solving real time applications.
CO 5	Describe hashing techniques and collision resolution methods for efficiently accessing data with respectto performance.
CO 6	Compare various types of data structures; in terms of implementation, operations and performance.

DATA STRUCTURES LABORATORY COURSE CONTENT

IV. COURSE CONTENT:

EXERCISES FOR DATA STRUCTURES LABORATORY

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

1. Getting Started Exercises

1.1 Sum of last digits of two given numbers

Rohit wants to add the last digits of two given numbers. For example, If the given numbers are 267 and 154, the output should be 11.
Below is the explanation -
Last digit of the 267 is 7
Last digit of the 154 is 4
Sum of 7 and 4 = 11
Write a program to help Rohit achieve this for any given two numbers.
The prototype of the method should be -
int addLastDigits(int input1, int input2);
where input1 and input2 denote the two numbers whose last digits are to be added.
Note: The sign of the input numbers should be ignored.
if the input numbers are 267 and 154, the sum of last two digits should be 11
if the input numbers are 267 and -154, the sum of last two digits should be 11
if the input numbers are -267 and 154, the sum of last two digits should be 11
if the input numbers are -267 and -154, the sum of last two digits should be 11
Input: 267 154
Output: 11
Input: 267 -154
Output: 11
Input: -267 154
Output: 11
Input: -267 -154
Output: 11

```
import java.util.Scanner;

class AddLastDigitsFunction
{
      int addLastDigits(int n1, int n2)
      {
              # Write code here
      }

      public static void main(String args[])
      {
            AddLastDigitsFunction obj = new AddLastDigitsFunction();
            # Write code here
            System.out.println(obj.addLastDigits(n1,n2));
```

```
        }

}
```

1.2 Is N an exact multiple of M?

Write a function that accepts two parameters and finds whether the first parameter is an exact multiple of the second parameter. If the first parameter is an exact multiple of the second parameter, the function should return 2 else it should return 1.
If either of the parameters are zero, the function should return 3.
Assumption: Within the scope of this question, assume that - the first parameter can be positive, negative or zero the second parameter will always be >=0
Input: num1 = 10, num2 = 5
Output: 2
Input: num1 = -10, num2 = 5
Output: 2
Input: num1 = 0, num2 = 5
Output: 3
Input: num1 = 10, num2 = 3
Output: 1

```
public class MultipleChecker
{
    public static int checkMultiple(int num1, int num2)
    {
        # Write code here
    }

    public static void main(String[] args)
    {
        # Write code here
    }
}
```

1.3 Combine Strings

Given 2 strings, a and b, return a new string of the form short+long+short, with the shorter string on the outside and the longer string in the inside. The strings will not be the same length, but they may be empty (length 0).
If input is "hi" and "hello", then output will be "hihellohi"
Input: Enter the first string: “hi”
Enter the second string: “hello”
Output: “hihellohi”
Input: Enter the first string: “iare”
Enter the second string: “college”
Output: “iarecollegeiare”

```
public class StringCombiner
{
    public static void main(String[] args)
    {
        # Write code here
    }

    public static String combineStrings(String a, String b)
    {
        # Write code here
    }
}
```

1.4 Even or Odd

Write a function that accepts 6 input parameters. The first 5 input parameters are of type int. The sixth input parameter is of type string. If the sixth parameter contains the value "even", the function is supposed to return the count of how many of the first five input parameters are even. If the sixth parameter contains the value "odd", the function is supposed to return the count of how many of the first five input parameters are odd.

Example:

If the five input parameters are 12, 17, 19, 14, and 115, and the sixth parameter is "odd", the function must return 3, because there are three odd numbers 17, 19 and 115.

If the five input parameters are 12, 17, 19, 14, and 115, and the sixth parameter is "even", the function must return 2, because there are two even numbers 12 and 14.

Note that zero is considered an even number.

Input: num1 = 12;
num2 = 17;
num3 = 19;
num4 = 14;
num5 = 115;
type = "odd"

Output: 3

Input: num1 = 12;
num2 = 17;
num3 = 19;
num4 = 14;
num5 = 115;
type = "even"

Output: 2

```
public class NumberCounter
{
    public static int countNumbers(int num1, int num2, int num3, int num4, int num5,
String type)
   {
        # Write code here
   }

    public static void main(String[] args)
    {
        # Write code here
    }
}
```

1.5 Second last digit of a given number

Write a function that returns the second last digit of the given number. Second last digit is being referred to the digit in the tens place in the given number.
Example: if the given number is 197, the second last digit is 9.
Note 1: The second last digit should be returned as a positive number. i.e. if the given number is -197, the second last digit is 9.
Note 2: If the given number is a single digit number, then the second last digit does not exist. In such cases, the function should return -1. i.e. if the given number is 5, the second last digit should be returned as -1.
Input: 197
Output: 9
Input: 5
Output: -1
Input: -197
Output: 9

```
public class SecondLastDigit
{
    public static int getSecondLastDigit(int number)
    {
        # write code here
    }
    public static void main(String[] args)
    {
        # write code here
    }
}
```

1.6 Alternate String Combiner

Given two strings, a and b, print a new string which is made of the following combination-first character of a, the first character of b, second character of a, second character of b and so on.
Any characters left, will go to the end of the result.
Hello,World
HWeolrllod
Input: “Hello,World”
Output: “HWeolrllod”
Input: “Iare,College”
Output: “ICaorlelege”

```
public class AlternateStringCombiner
{
    public static void main(String[] args)
    {
        # write code here
    }
    public static String combineStrings(String a, String b)
    {
        # write code here
    }
}
```

1.7 Padovan Sequence

The Padovan sequence is a sequence of numbers named after Richard Padovan, who attributed its discovery to Dutch architect Hans van der Laan. The sequence was described by Ian Stewart in his Scientific American column Mathematical Recreations in June 1996. The Padovan sequence is defined by the following recurrence relation:

P(n) = P(n-2) + P(n-3)

with the initial conditions P(0) = P(1) = P(2) = 1.

In this sequence, each term is the sum of the two preceding terms, similar to the Fibonacci sequence. However, the Padovan sequence has different initial conditions and exhibits different growth patterns.

The first few terms of the Padovan sequence are: 1, 1, 1, 2, 2, 3, 4, 5, 7, 9, 12, ...

Input: num = 10

Output: Padovan Sequence up to 10:

1 1 1 2 2 3 4 5 7 9 12

Input: num = 20

Output: Padovan Sequence up to 20:

1 1 1 2 2 3 4 5 7 9 12 16 21 28 37 49 65 86 114 151 200

```
public class PadovanSequence
{
    public static int padovan(int n)
    {
        # write code here
    }
    public static void main(String[] args)
    {
        # write code here
    }
}
```

1.8 Leaders in an array

Given an array arr of n positive integers, your task is to find all the leaders in the array. An element of the array is considered a leader if it is greater than all the elements on its right side or if it is equal to the maximum element on its right side. The rightmost element is always a leader.

Input: n = 6, arr[] = (16, 17, 4, 3, 5, 2}

Output: 17 5 2

Input: n = 5, arr[] = {10, 4, 2, 4, 1}

Output: 10 4 4 1

Input: n = 4, arr[] = {5, 10, 20, 40}

Output: 40

Input: n = 4, arr[] = {30, 10, 10, 5}

Output: 30 10 10 5

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

```
public class ArrayLeaders
{
    public static List<Integer> findArrayLeaders(int[] arr)
    {
        # write code here
    }
    public static void main(String[] args)
    {
        # write code here
    }
}
```

1.9 Find the Value of a Number Raised to its Reverse

Given a number N and its reverse R. The task is to find the number obtained when the number is raised to the power of its own reverse

Input : N = 2, R = 2

Output: 4

Explanation: Number 2 raised to the power of its reverse 2 gives 4 which gives 4 as a result after performing modulo 10^9+7

Input: N = 57, R = 75

Output: 262042770

Explanation: 57^{75} modulo 10^9+7 gives us the result as 262042770

```
public class NumberPower
{
    public static long powerOfReverse(int N, int R)
    {
        # write code here
    }
    public static void main(String[] args)
    {
        # write code here
    }
}
```

1.10 Mean of Array using Recursion

Find the mean of the elements of the array.

Mean = (Sum of elements of the Array) / (Total no of elements in Array)

Input: 1 2 3 4 5

Output: 3.0

Input: 1 2 3

Output: 2.0

To find the mean using recursion assume that the problem is already solved for N-1 i.e. you have to find for n

Sum of first N-1 elements = (Mean of N-1 elements) * (N-1)

Mean of N elements = (Sum of first N-1 elements + N-th elements) / (N)

```
public class ArrayMean
{
    public static double findArrayMean(int[] arr)
    {
        # write code here
    }
    public static void main(String[] args)
    {
        # write code here
    }
}
```

Try:

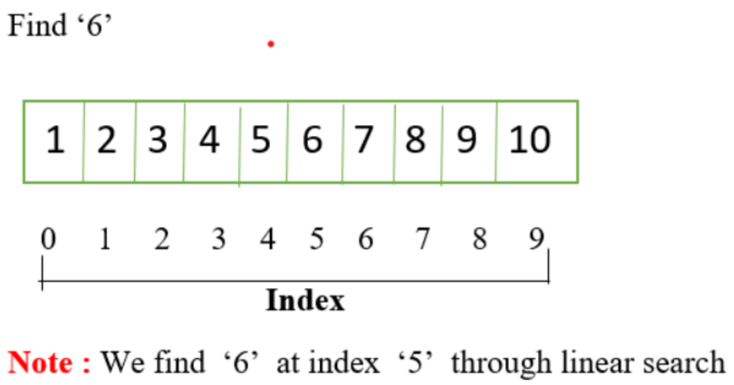
1. **Kth Smallest Element:** Given an array arr[] and an integer k where k is smaller than the size of the array, the task is to find the kth smallest element in the given array. It is given that all array elements are distinct.
Note: l and r denotes the starting and ending index of the array.
Input: n = 6, arr[] = {7, 10, 4, 3, 20, 15}, k = 3, l = 0, r = 5
Output: 7
Explanation: 3rd smallest element in the given array is 7.
Input: n = 5, arr[] = {7, 10, 4, 20, 15}, k = 4, l=0 r=4
Output: 15

Explanation: 4th smallest element in the given array is 15.
Your task is to complete the function **kthSmallest()** which takes the array arr[], integers l and r denoting the starting and ending index of the array and an integer k as input and returns the kth smallest element.
2. **Count pairs with given sum:** Given an array of N integers, and an integer K, find the number of pairs of elements in the array whose sum is equal to K. Your task is to complete the function **getPairsCount()** which takes arr[], n and k as input parameters and returns the number of pairs that have sum K.
Input: N = 4, K = 6, arr[] = {1, 5, 7, 1}
Output: 2
Explanation: arr[0] + arr[1] = 1 + 5 = 6 and arr[1] + arr[3] = 5 + 1 = 6.
Input: N = 4, K = 2, arr[] = {1, 1, 1, 1}
Output: 6
Explanation: Each 1 will produce sum 2 with any 1.

2. Searching

2.1 Linear / Sequential Search

Linear search is defined as the searching algorithm where the list or data set is traversed from one end to find the desired value. Given an array arr[] of n elements, write a recursive function to search a given element x in arr[].



Linear search procedure:

1. Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
2. If x matches with an element, return the index.
3. If x doesn't match with any of the elements, return -1.

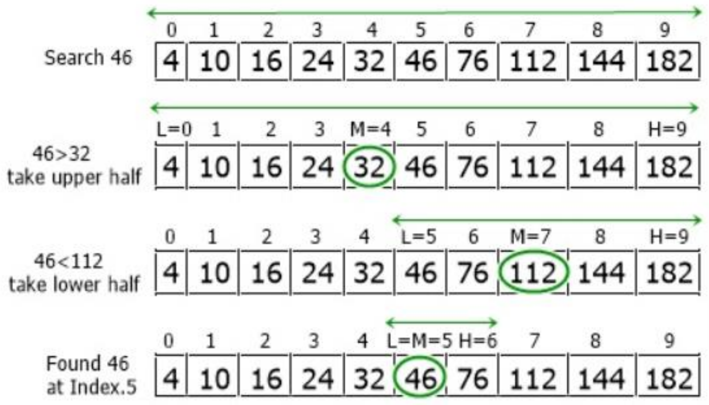
Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}
x = 110;
Output: 6
Element x is present at index 6

Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}
x = 175;
Output: -1
Element x is not present in arr[].

```
public class RecursiveLinearSearch
{
    public static int recursiveLinearSearch(int[] arr, int key, int index)
    {
        # write code here
    }

    public static void main(String[] args)
    {
        # write code here
    }
}
```

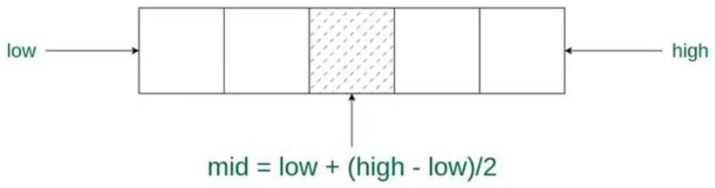
2.2 Binary Search

Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).



Conditions for Binary Search algorithm:

1. The data structure must be sorted.
2. Access to any element of the data structure takes constant time.



Binary Search Procedure:

1. Divide the search space into two halves by finding the middle index "mid".
2. Compare the middle element of the search space with the key.
3. If the key is found at middle element, the process is terminated.
4. If the key is not found at middle element, choose which half will be used as the next search space.
    a. If the key is smaller than the middle element, then the left side is used for next search.
    b. If the key is larger than the middle element, then the right side is used for next search.
5. This process is continued until the key is found or the total search space is exhausted.

Input: arr = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
Output: target = 23
Element 23 is present at index 5

```
public class RecursiveBinarySearch
{
    public static int recursiveBinarySearch(int[] arr, int key, int left, int right)
    {
         # write code here
    }
    public static void main(String[] args)
    {
         # write code here
    }
}
```

2.3 Uniform Binary Search

Uniform Binary Search is an optimization of Binary Search algorithm when many searches are made on same array or many arrays of same size. In normal binary search, we do arithmetic operations to find the mid points. Here we precompute mid points and fills them in lookup table. The array look-up generally works faster than arithmetic done (addition and shift) to find the mid-point.

Input: array = {1, 3, 5, 6, 7, 8, 9}, v=3
Output: Position of 3 in array = 2

Input: array = {1, 3, 5, 6, 7, 8, 9}, v=7
Output: Position of 7 in array = 5

The algorithm is very similar to Binary Search algorithm, the only difference is a lookup table is created for an array and the lookup table is used to modify the index of the pointer in the array which makes the search faster. Instead of maintaining lower and upper bound the algorithm maintains an index and the index is modified using the lookup table.

```
public class RecursiveUniformBinarySearch
{
    public static int recursiveUniformBinarySearch(int[] arr, int key, int[] lookupTable, int left, int right)
    {
        # write code here
    }
    public static void main(String[] args)
    {
        # write code here
    }
}
```

2.4 Interpolation Search

Interpolation search works better than Binary Search for a Sorted and Uniformly Distributed array. Binary search goes to the middle element to check irrespective of search-key. On the other hand, Interpolation search may go to different locations according to search-key. If the value of the search-key is close to the last element, Interpolation Search is likely to start search toward the end side. Interpolation search is more efficient than binary search when the elements in the list are uniformly distributed, while binary search is more efficient when the elements in the list are not uniformly distributed.

Interpolation search can take longer to implement than binary search, as it requires the use of additional calculations to estimate the position of the target element.

Input: arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
Output: target = 5

```
public class InterpolationSearch
{
    public static int interpolationSearch(int[] arr, int key)
    {
        # write code here
    }

    public static void main(String[] args)
    {
        # write code here
    }
}
```

2.5 Fibonacci Search

Given a sorted array arr[] of size n and an element x to be searched in it. Return index of x if it is present in array else return -1.

Input: arr[] = {2, 3, 4, 10, 40}, x = 10
Output: 3
Element x is present at index 3.

Input: arr[] = {2, 3, 4, 10, 40}, x = 11
Output: -1
Element x is not present.

Fibonacci Search is a comparison-based technique that uses Fibonacci numbers to search an element in a sorted array.
Fibonacci Numbers are recursively defined as F(n) = F(n-1) + F(n-2), F(0) = 0, F(1) = 1. First few Fibonacci Numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Fibonacci Search Procedure:
Let the searched element be x. The idea is to first find the smallest Fibonacci number that is greater than or equal to the length of the given array. Let the found Fibonacci number be fib (m'th Fibonacci number). We use (m-2)'th Fibonacci number as the index (If it is a valid index). Let (m-2)'th Fibonacci Number be i, we compare arr[i] with x, if x is same, we return i. Else if x is greater, we recur for subarray after i, else we recur for subarray before i.

Let arr[0..n-1] be the input array and the element to be searched be x.

1. Find the smallest Fibonacci number greater than or equal to n. Let this number be fibM [m'th Fibonacci number]. Let the two Fibonacci numbers preceding it be fibMm1 [(m-1)'th Fibonacci Number] and fibMm2 [(m-2)'th Fibonacci Number].
2. While the array has elements to be inspected:
    i. Compare x with the last element of the range covered by fibMm2
    ii. If x matches, return index
    iii. Else If x is less than the element, move the three Fibonacci variables two Fibonacci down, indicating elimination of approximately rear two-third of the remaining array.

    iv. Else x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Together these indicate the elimination of approximately front one-third of the remaining array.

3. Since there might be a single element remaining for comparison, check if fibMm1 is 1. If Yes, compare x with that remaining element. If match, return index.

```
public class FibonacciSearch
{
    public static int fibonacciSearch(int[] arr, int key)
    {
        # write code here
    }

    public static void main(String[] args)
    {
        # write code here
    }
}
```

3. Sorting

3.1 Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

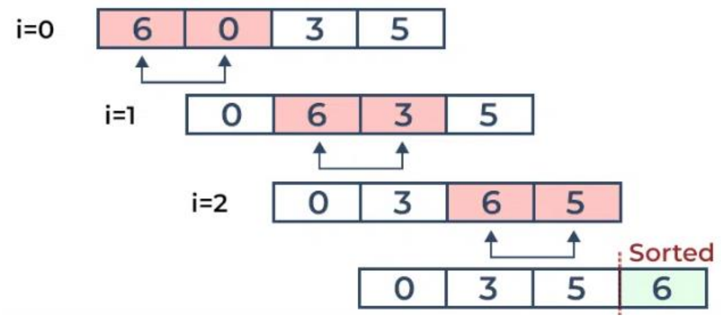
Bubble Sort Procedure:

1. Traverse from left and compare adjacent elements and the higher one is placed at right side.
2. In this way, the largest element is moved to the rightmost end at first.
3. This process is then continued to find the second largest and place it and so on until the data is sorted.

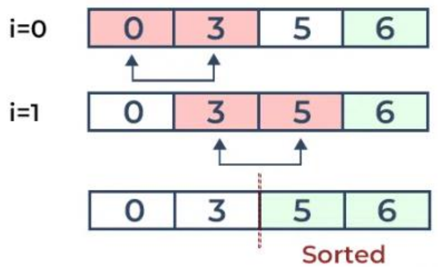
Input: arr = [6, 3, 0, 5]

Output:

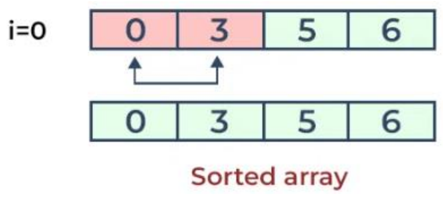
First Pass:



Second Pass:



Third Pass:



```
import java.util.Scanner;

class BubbleSortExample
{
    public static void main(String[] args)
    {
          # write code here

    }

    public static void bubbleSort(int[] arr)
    {
          # write code here

    }
}
```

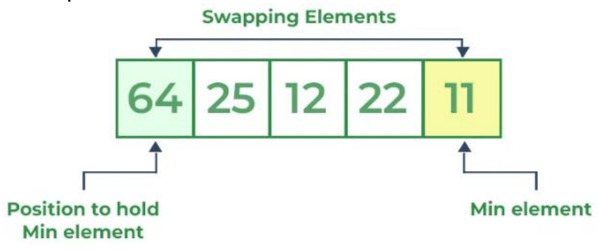
3.2 Selection Sort

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list. The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

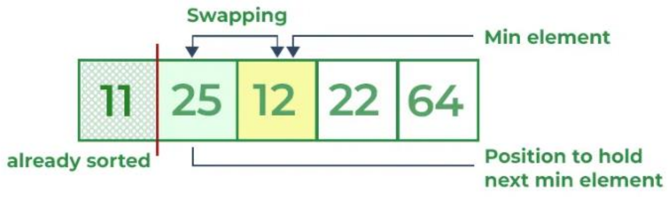
Input: arr = [64, 25, 12, 22, 11]

Output: arr = [11, 12, 22, 25, 64]

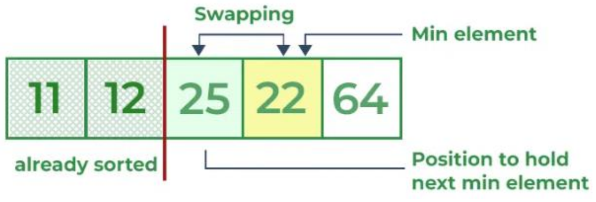
First Pass: For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where 64 is stored presently, after traversing whole array it is clear that 11 is the lowest value. Thus, replace 64 with 11. After one iteration 11, which happens to be the least value in the array, tends to appear in the first position of the sorted list.



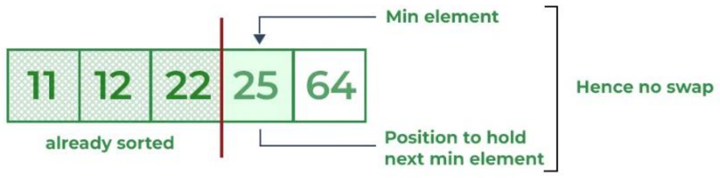
Second Pass: For the second position, where 25 is present, again traverse the rest of the array in a sequential manner. After traversing, we found that 12 is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.



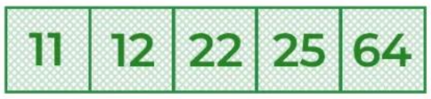
Third Pass: Now, for third place, where 25 is present again traverse the rest of the array and find the third least value present in the array. While traversing, 22 came out to be the third least value and it should appear at the third place in the array, thus swap 22 with element present at third position.



Fourth Pass: Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array. As 25 is the 4th lowest value hence, it will place at the fourth position.



Fifth Pass: At last the largest value present in the array automatically get placed at the last position in the array. The resulted array is the sorted array.



Sorted array

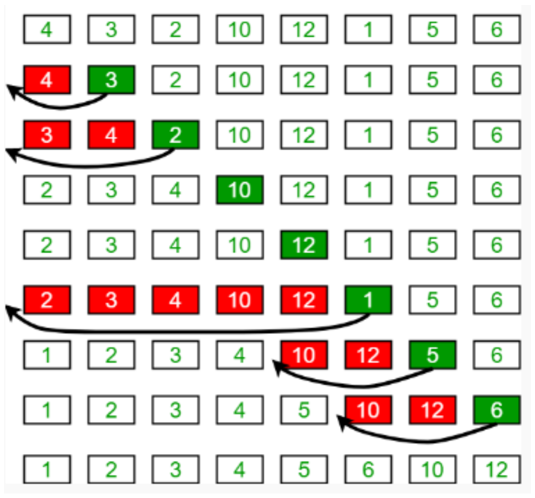
```
import java.util.Scanner;

class SelectionSortExample
{
    public static void main(String[] args)
    {
        # write code here
    }

    public static void selectionSort(int[] arr)
    {
        # write code here
    }
}
```

3.3 Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Insertion Sort Procedure:

1. To sort an array of size N in ascending order iterate over the array and compare the current element (key) to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before.
2. Move the greater elements one position up to make space for the swapped element.



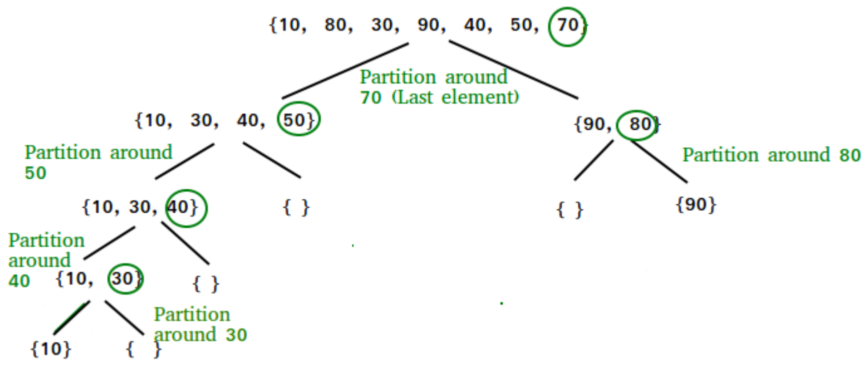
Input: arr = [4, 3, 2, 10, 12, 1, 5, 6]
Output: arr = [1, 2, 3, 4, 5, 6, 10, 12]

```
import java.util.Scanner;

class InsertionSortExample
{
    public static void main(String[] args)
    {
      # write code here
    }

    public static void insertionSort(int[] arr)
    {
        # write code here
    }
}
```

4. Divide and Conquer

4.1 Quick Sort

QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array. The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot. Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.



The quick sort method can be summarized in three steps:
1. **Pick:** Select a pivot element.
2. **Divide:** Split the problem set, move smaller parts to the left of the pivot and larger items to the right.
3. **Repeat and combine:** Repeat the steps and combine the arrays that have previously been sorted.

Algorithm for Quick Sort Function:

```
//start –> Starting index,  end --> Ending index
Quicksort(array, start, end)
{
        if (start < end)
        {
                pIndex = Partition(A, start, end)
                Quicksort(A,start,pIndex-1)
                Quicksort(A,pIndex+1, end)
        }
}
```

Algorithm for Partition Function:

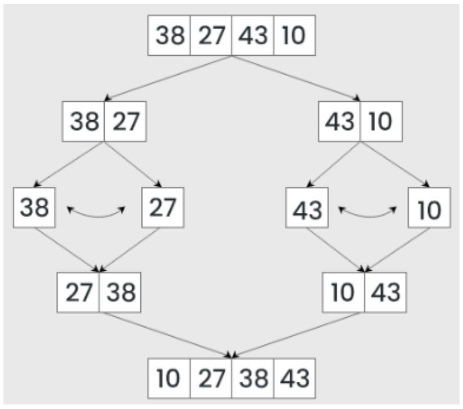
```
partition (array, start, end)
{
        // Setting rightmost Index as pivot
        pivot = arr[end];
        i = (start - 1)  // Index of smaller element and indicates the
            // right position of pivot found so far
        for (j = start; j <= end- 1; j++)
        {
                // If current element is smaller than the pivot
                if (arr[j] < pivot)
                {
                        i++;    // increment index of smaller element
                        swap arr[i] and arr[j]
                }
        }
        swap arr[i + 1] and arr[end])
        return (i + 1)
}
```

Input: arr = [10, 80, 30, 90, 40, 50, 70]
Output: arr = [10, 30, 40, 50, 70, 80, 90]

```
import java.util.Scanner;

class QuickSortExample
{
    public static void main(String[] args)
    {
        # write code here
    }

    public static void quickSort(int[] arr, int low, int high)
    {
        # write code here
    }

    public static int partition(int[] arr, int low, int high)
    {
        # write code here
    }
}
```

4.2 Merge Sort

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array. In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.



Input: arr = [12, 11, 13, 5, 6, 7]
Output: arr = [5, 6, 7, 11, 12, 13]

```
import java.util.Scanner;

class MergeSortExample
{
    public static void main(String[] args)
    {
        # write code here
    }

    public static void mergeSort(int[] arr, int low, int high)
    {
        # write code here
    }

    public static void merge(int[] arr, int low, int mid, int high)
    {
        # write code here
    }
}
```

4.3 Heap Sort

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

Heap Sort Procedure:

First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of heap is greater than 1.

- Build a heap from the given input array.
- Repeat the following steps until the heap contains only one element:

    - Swap the root element of the heap (which is the largest element) with the last element of the heap.
    - Remove the last element of the heap (which is now in the correct position).
    - Heapify the remaining elements of the heap.
- The sorted array is obtained by reversing the order of the elements in the input array.

Input: arr = [12, 11, 13, 5, 6, 7]
Output: Sorted array is 5 6 7 11 12 13

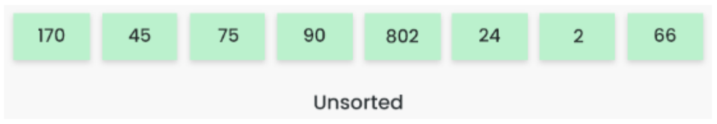
```
import java.util.Scanner;

class HeapSortExample
{
    public static void main(String[] args)
    {
                # write code here
    }

    public static void heapSort(int[] arr)
    {
                # write code here
    }

    public static void heapify(int[] arr, int n, int i)
    {
        # write code here
    }
}
```

4.4 Radix Sort

Radix Sort is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys. Rather than comparing elements directly, Radix Sort distributes the elements into buckets based on each digit's value. By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, Radix Sort achieves the final sorted order.

Radix Sort Procedure:
The key idea behind Radix Sort is to exploit the concept of place value.
1. It assumes that sorting numbers digit by digit will eventually result in a fully sorted list.
2. Radix Sort can be performed using different variations, such as Least Significant Digit (LSD) Radix Sort or Most Significant Digit (MSD) Radix Sort.

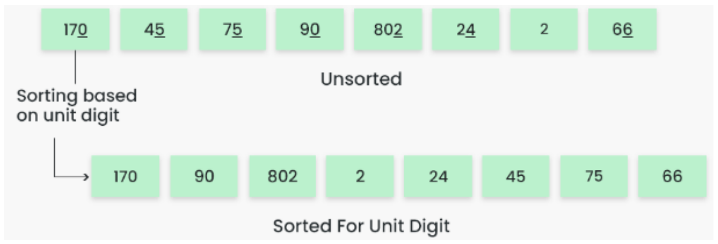
To perform radix sort on the array [170, 45, 75, 90, 802, 24, 2, 66], we follow these steps:



Step 1: Find the largest element in the array, which is 802. It has three digits, so we will iterate three times, once for each significant place.

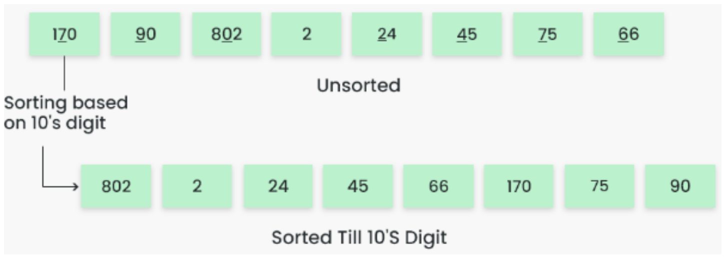
Step 2: Sort the elements based on the unit place digits (X=0). We use a stable sorting technique, such as counting sort, to sort the digits at each significant place.

Sorting based on the unit place:
Perform counting sort on the array based on the unit place digits.
The sorted array based on the unit place is [170, 90, 802, 2, 24, 45, 75, 66]



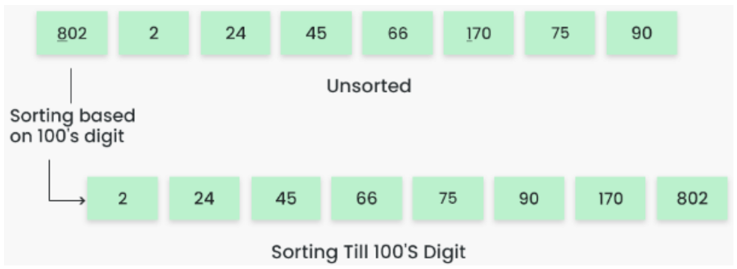
Step 3: Sort the elements based on the tens place digits.

Sorting based on the tens place:
Perform counting sort on the array based on the tens place digits.
The sorted array based on the tens place is [802, 2, 24, 45, 66, 170, 75, 90]



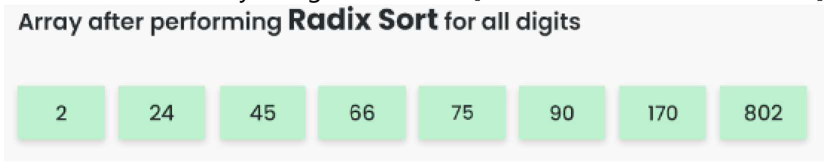
Step 4: Sort the elements based on the hundreds place digits.

Sorting based on the hundreds place:
Perform counting sort on the array based on the hundreds place digits.
The sorted array based on the hundreds place is [2, 24, 45, 66, 75, 90, 170, 802]



Step 5: The array is now sorted in ascending order.

The final sorted array using radix sort is [2, 24, 45, 66, 75, 90, 170, 802]



```
import java.util.Arrays;

class RadixSortExample
{
    public static void main(String[] args)
    {
        # write code here
    }

    public static void radixSort(int[] arr)
    {
        # write code here
    }

    public static int getMax(int[] arr)
    {
        # write code here
    }

    public static void countSort(int[] arr, int exp)
    {
        # write code here
    }
}
```

4.5 Shell Sort

Shell sort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of ShellSort is to allow the exchange of far items. In Shell sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h'th element are sorted.

Shell Sort Procedure:

1. Initialize the value of gap size h
2. Divide the list into smaller sub-part. Each must have equal intervals to h
3. Sort these sub-lists using insertion sort
4. Repeat this step 1 until the list is sorted.
5. Print a sorted list.

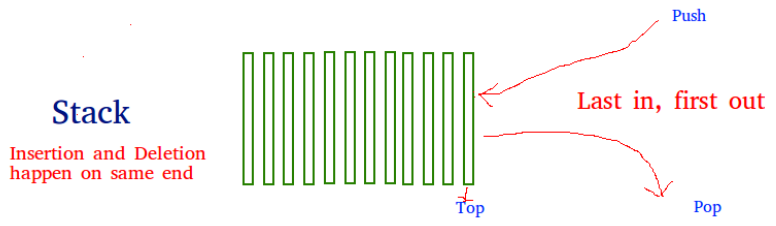
```
Procedure Shell_Sort(Array, N)
  While Gap < Length(Array) /3 :
             Gap = ( Interval * 3 ) + 1
  End While Loop
  While Gap > 0 :
     For (Outer = Gap; Outer < Length(Array); Outer++):
         Insertion_Value = Array[Outer]
              Inner = Outer;
         While Inner > Gap-1 And Array[Inner – Gap] >= Insertion_Value:
              Array[Inner] = Array[Inner – Gap]
              Inner = Inner – Gap
          End While Loop
             Array[Inner] = Insertion_Value
     End For Loop
     Gap = (Gap -1) /3;
  End While Loop
End Shell_Sort
```

```
import java.util.Scanner;

class ShellSortExample
{
    public static void main(String[] args)
    {
        # write code here
    }

    public static void shellSort(int[] arr)
    {
        # write code here
    }
}
```

5. Stack

5.1 Implementation of Stack

A stack is a linear data structure that stores items in a Last-In/First-Out (LIFO) or First-In/Last-Out (FILO) manner. In stack, a new element is added at one end and an element is removed from that end only. The insert and delete operations are often called push and pop.



The functions associated with stack are:

- **empty()** – Returns whether the stack is empty
- **size()** – Returns the size of the stack
- **top() / peek()** – Returns a reference to the topmost element of the stack
- **push(a)** – Inserts the element 'a' at the top of the stack
- **pop()** – Deletes the topmost element of the stack

```
class Stack
{
    private int maxSize;
    private int top;
    private int[] stackArray;

    public Stack(int size)
    {
        # write code here
    }

    public void push(int value)
    {
        # write code here
    }

    public int pop()
    {
        # write code here
    }

    public int peek()
    {
        # write code here
    }

    public boolean isEmpty()
    {
       # write code here
    }
    public boolean isFull()
    {
        # write code here
    }
}

class StackExample
{
    public static void main(String[] args)
    {
        Stack stack = new Stack(5);
        stack.push(10);
        stack.push(20);
        stack.push(30);
        stack.pop();
        stack.peek();
        stack.push(40);
        stack.push(50);
        stack.push(60);
    }
}
```

5.2 Balanced Parenthesis Checking

Given an expression string, write a java program to find whether a given string has balanced parentheses or not.
Input: "{(a+b)*(c-d)}"
Output: true
Input: "{(a+b)*[c-d)}"
Output: false
One approach to check balanced parentheses is to use stack. Each time, when an open parentheses is encountered push it in the stack, and when closed parenthesis is encountered, match it with the top of stack and pop it. If stack is empty at the end, return true otherwise, false

```
import java.util.Stack;

class BalancedParenthesisChecker
{
    public static boolean isBalanced(String expression)
    {
        # write code here
    }

    public static void main(String[] args)
    {
        String expression1 = "{(a+b)*(c-d)}";
        String expression2 = "{(a+b)*[c-d)}";

        # write code here
    }
}
```

5.3 Evaluation of Postfix Expression

Given a postfix expression, the task is to evaluate the postfix expression. Postfix expression: The expression of the form "a b operator" (ab+) i.e., when a pair of operands is followed by an operator.
Input: str = "2 3 1 * + 9 -"
Output: -4
Explanation: If the expression is converted into an infix expression, it will be 2 + (3 * 1) – 9 = 5 – 9 = -4.
Input: str = "100 200 + 2 / 5 * 7 +"
Output: 757
Procedure for evaluation postfix expression using stack:

- Create a stack to store operands (or values).
- Scan the given expression from left to right and do the following for every scanned element.
  - If the element is a number, push it into the stack.
  - If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack.
- When the expression is ended, the number in the stack is the final answer.

```
import java.util.Stack;

class PostfixEvaluator
{
    public static int evaluatePostfix(String expression)
    {
        # write code here
    }

    public static int performOperation(char operator, int operand1, int operand2)
    {
        # write code here
    }

    public static void main(String[] args)
    {
        # write code here
    }
}
```

5.4 Infix to Postfix Expression Conversion

For a given Infix expression, convert it into Postfix form.
Infix expression: The expression of the form "a operator b" (a + b) i.e., when an operator is in-between every pair of operands.
Postfix expression: The expression of the form "a b operator" (ab+) i.e., When every pair of operands is followed by an operator.

Infix to postfix expression conversion procedure:

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, put it in the postfix expression.
3. Otherwise, do the following

- If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack [or the stack is empty or the stack contains a '('], then push it in the stack. ['^' operator is right associative and other operators like '+','–','*' and '/' are left-associative].
    - Check especially for a condition when the operator at the top of the stack and the scanned operator both are '^'. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack.
    - In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.
- Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.
    - After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

4. If the scanned character is a '(', push it to the stack.
5. If the scanned character is a ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-5 until the infix expression is scanned.
7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
8. Finally, print the postfix expression.

Input: A + B * C + D
Output: A B C * + D +

Input: ((A + B) – C * (D / E)) + F
Output: A B + C D E / * - F +

```
import java.util.Stack;

class Conversion
      {

        # Write Code Here

      }
```

5.5 Reverse a Stack

The stack is a linear data structure which works on the LIFO concept. LIFO stands for last in first out. In the stack, the insertion and deletion are possible at one end the end is called the top of the stack. Define two recursive functions BottomInsertion() and Reverse() to reverse a stack using Python. Define some basic function of the stack like push(), pop(), show(), empty(), for basic operation like respectively append an item in stack, remove an item in stack, display the stack, check the given stack is empty or not.

BottomInsertion(): this method append element at the bottom of the stack and BottomInsertion accept two values as an argument first is stack and the second is elements, this is a recursive method.

Reverse(): the method is reverse elements of the stack, this method accept stack as an argument Reverse() is also a Recursive() function. Reverse() is invoked BottomInsertion() method for completing the reverse operation on the stack.

Input: Elements = [1, 2, 3, 4, 5]
Output: Original Stack

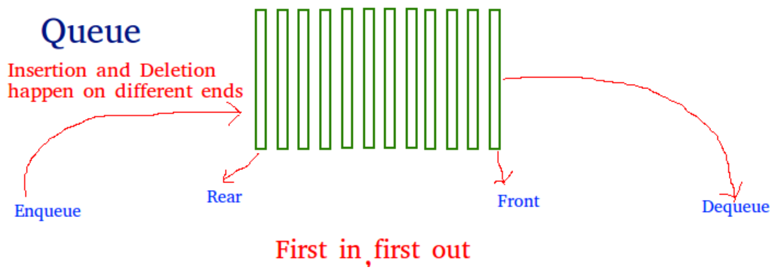
```
5
4
3
2
1
Stack after Reversing
1
2
3
4
5
```

```
import java.util.Stack;
class StackClass {
        # Write Code Here
    }
```

6. Queue

6.1 Linear Queue

Linear queue is a linear data structure that stores items in First in First out (FIFO) manner. With a queue the least recently added item is removed first. A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.



```
import java.util.Scanner;

public class LinearQueue
       {
       # Write Code Here
       }
    public static boolean isEmpty() {
        return front == rear;
    }
    public static boolean isFull() {
        return rear == MAX;
    }
    public static void enqueue(int item)
       {
       # Write Code Here
       }

    public static void dequeue()
       {
       # Write Code Here
       }
public static void display()
       {
        # Write Code Here
       }
    public static void main(String[] args)
       {
        # Write Code Here
       }
}
```

6.2 Stack using Queues

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (push, top, pop, and empty).

- void push(int x) Pushes element x to the top of the stack.
- int pop() Removes the element on the top of the stack and returns it.
- int top() Returns the element on the top of the stack.
- boolean empty() Returns true if the stack is empty, false otherwise.

Input:
["MyStack", "push", "push", "top", "pop", "empty"]
[[], [1], [2], [], [], []]
Output:
[null, null, null, 2, 2, false]

```
import java.util.LinkedList;
import java.util.Queue;

class MyStack
          {
            # Write Code Here
          }
    public void push(int x)
      {
        # Write Code Here
      }
    }

    public int pop()
      {
        return queue.remove();
      }

    public int top() {
        return queue.peek();
    }

    public boolean empty() {
        return queue.isEmpty();
    }
    public static void main(String[] args)
      {
        # Write Code Here
      }
}
```

6.3 Queue using Stacks

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).

- void push(int x) Pushes element x to the back of the queue.
- int pop() Removes the element from the front of the queue and returns it.
- int peek() Returns the element at the front of the queue.
- boolean empty() Returns true if the queue is empty, false otherwise.

Input:

["MyQueue", "push", "push", "peek", "pop", "empty"]
[[], [1], [2], [], [], []]
Output:
[null, null, null, 1, 1, false]

```
import java.util.Stack;

class MyQueue {

    private Stack<Integer> stack1;
    private Stack<Integer> stack2;
    public MyQueue() {
        stack1 = new Stack<>();
        stack2 = new Stack<>();
    }
    public void push(int x) {
        stack1.push(x);
    }
    public int pop()
      {
        # Write Code Here
      }
    public int peek()
      {
      # Write Code Here
      }
    public boolean empty() {
        return stack1.isEmpty() && stack2.isEmpty();
    }
    public static void main(String[] args)
      {
        # Write Code Here
      }
}
```

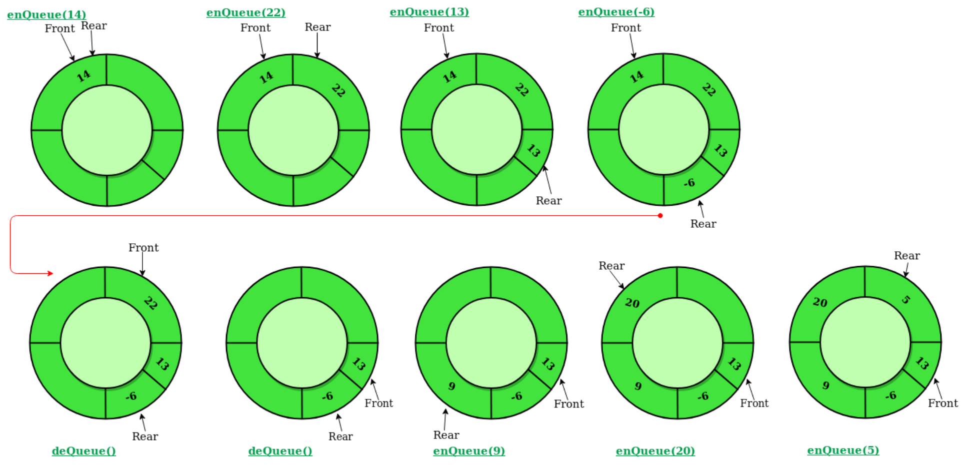
6.4 Circular Queue

A Circular Queue is an extended version of a normal queue where the last element of the queue is connected to the first element of the queue forming a circle. The operations are performed based on FIFO (First In First Out) principle. It is also called 'Ring Buffer'.

Operations on Circular Queue:

- **Front:** Get the front item from the queue.
- **Rear:** Get the last item from the queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at the rear position.
    - Check whether the queue is full – [i.e., the rear end is in just before the front end in a circular manner].
    - If it is full then display Queue is full.

    - If the queue is not full then, insert an element at the end of the queue.
- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from the front position.
    - Check whether the queue is Empty.
    - If it is empty then display Queue is empty.
        - If the queue is not empty, then get the last element and remove it from the queue.



Implement Circular Queue using Array:

1. Initialize an array queue of size **n**, where n is the maximum number of elements that the queue can hold.
2. Initialize two variables front and rear to -1.
3. **Enqueue:** To enqueue an element **x** into the queue, do the following:
    - Increment rear by 1.
        - If **rear** is equal to n, set **rear** to 0.
    - If **front** is -1, set **front** to 0.
    - Set queue[rear] to x.
4. **Dequeue:** To dequeue an element from the queue, do the following:
    - Check if the queue is empty by checking if **front** is -1.
        - If it is, return an error message indicating that the queue is empty.
    - Set **x** to queue [front].
    - If **front** is equal to **rear**, set **front** and **rear** to -1.
    - Otherwise, increment **front** by 1 and if **front** is equal to n, set **front** to 0.
    - Return x.

```
class CircularQueue {

    private int size;
    private int front, rear;
    private int[] queue;

    public CircularQueue(int size) {
        this.size = size;
        this.queue = new int[size];
        this.front = this.rear = -1;
    }


    public void enqueue(int data)
      {
      # Write Code Here
      }
    public int dequeue()
      {
        # Write Code Here
      }
    public void display()
      {
      # Write Code Here
      }
    public static void main(String[] args)
      {
      # Write Code Here
      }
}
```

6.5 Deque (Doubly Ended Queue)

In a Deque (Doubly Ended Queue), one can perform insert (append) and delete (pop) operations from both the ends of the container. There are two types of Deque:

1. **Input Restricted Deque:** Input is limited at one end while deletion is permitted at both ends.
2. **Output Restricted Deque:** Output is limited at one end but insertion is permitted at both ends.

Operations on Deque:

1. **append():** This function is used to insert the value in its argument to the right end of the deque.
2. **appendleft():** This function is used to insert the value in its argument to the left end of the deque.
3. **pop():** This function is used to delete an argument from the right end of the deque.
4. **popleft():** This function is used to delete an argument from the left end of the deque.
5. **index(ele, beg, end):** This function returns the first index of the value mentioned in arguments, starting searching from beg till end index.
6. **insert(i, a):** This function inserts the value mentioned in arguments(a) at index(i) specified in arguments.
7. **remove():** This function removes the first occurrence of the value mentioned in arguments.
8. **count():** This function counts the number of occurrences of value mentioned in arguments.
9. **len(dequeue):** Return the current size of the dequeue.
10. **Deque[0]:** We can access the front element of the deque using indexing with de[0].
11. **Deque[-1]:** We can access the back element of the deque using indexing with de[-1].
12. **extend(iterable):** This function is used to add multiple values at the right end of the deque. The argument passed is iterable.
13. **extendleft(iterable):** This function is used to add multiple values at the left end of the deque. The argument passed is iterable. Order is reversed as a result of left appends.
14. **reverse():** This function is used to reverse the order of deque elements.

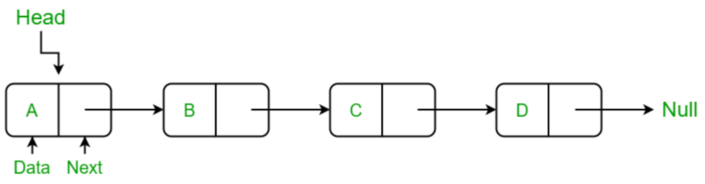
15. **rotate():** This function rotates the deque by the number specified in arguments. If the number specified is negative, rotation occurs to the left. Else rotation is to right.

```
import java.util.ArrayDeque;
import java.util.Deque;

public class DequeOperations
{
    # Write Code Here
}
```

7. Linked List

7.1 Singly Linked List

A singly linked list is a linear data structure in which the elements are not stored in contiguous memory locations and each element is connected only to its next element using a pointer.



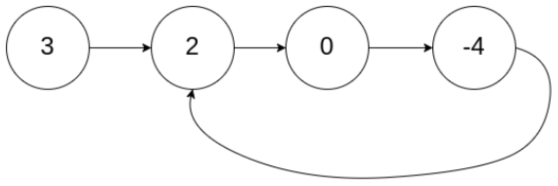
Creating a linked list involves the following operations:

1. Creating a Node class:
2. Insertion at beginning:
3. Insertion at end
4. Insertion at middle
5. Update the node
6. Deletion at beginning
7. Deletion at end
8. Deletion at middle
9. Remove last node
10. Linked list traversal
11. Get length

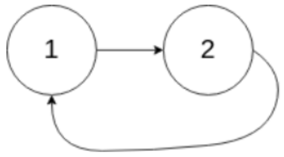
```
class Node {
    String data;
    Node next;

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

class LinkedList
      {
      # Write Code Here
      }

    public void insertAtEnd(String data)
      {
        # Write Code Here
      }

    public void updateNode(String val, int index)
      {
      # Write Code Here
      }

    public void remove_first_node() {
        # Write Code Here
    }
    public void remove_last_node()
      {
       # Write Code Here
      }

    public void remove_at_index(int index)

       {
      # Write Code Here
       }

    public void remove_node(String data)
      {
        # Write Code Here
       }

    public int sizeOfLL()
      {
     # Write Code Here
      }

    public void printLL()
      {
        # Write Code Here
      }

    public static void main(String[] args)
      {
        # Write Code Here
      }
}
```

7.2 Linked List Cycle

Given head, the head of a linked list, determine if the linked list has a cycle in it. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to.
Note that pos is not passed as a parameter.
Return true if there is a cycle in the linked list. Otherwise, return false.



Input: head = [3, 2, 0, -4], pos = 1
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).



Input: head = [1, 2], pos = 0

Output: true

Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.

Input: head = [1], pos = -1

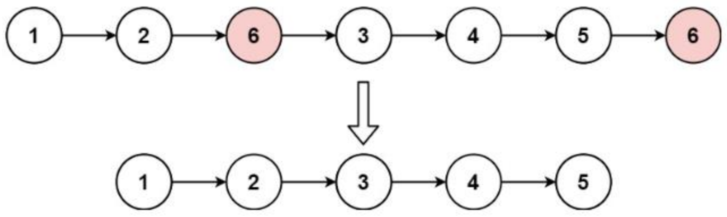
Output: false

Explanation: There is no cycle in the linked list.

```
class ListNode
      {
    # Write Code Here
      }

public class Solution
      {
       # Write Code Here
      }
}
```

7.3 Remove Linked List Elements

Given the head of a linked list and an integer val, remove all the nodes of the linked list that has Node.val == val, and return the new head.



Input: head = [1, 2, 6, 3, 4, 5, 6], val = 6

Output: [1, 2, 3, 4, 5]

Input: head = [ ], val = 1

Output: [ ]

Input: head = [7, 7, 7, 7], val = 7

Output: [ ]

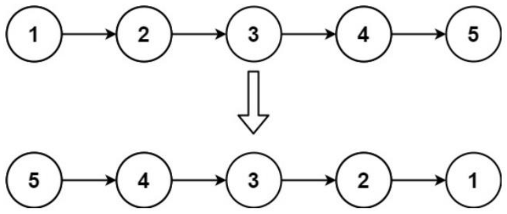
```
class ListNode {
    # Write Code Here
    }
```

```
}
public class Solution {
    public boolean hasCycle(ListNode head)
       {
        # Write Code Here
       }

    public static void main(String[] args)
       {
        # Write Code Here
       }
}
```

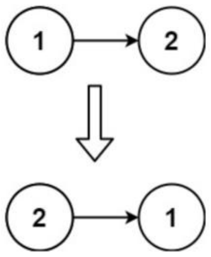
7.4 Reverse Linked List

Given the head of a singly linked list, reverse the list, and return the reversed list.

Input: head = [1, 2, 3, 4, 5]
Output: [5, 4, 3, 2, 1]



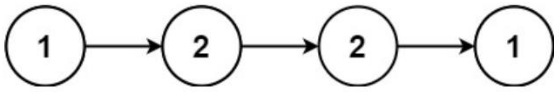
Input: head = [1, 2]
Output: [2, 1]



```
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
}

public class Solution {
    public ListNode reverseList(ListNode head)
       {
        # Write Code Here
       }
    public static void main(String[] args)
      {
        # Write Code Here
      }
```

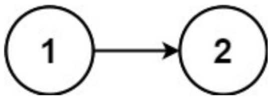
7.5 Palindrome Linked List

Given the head of a singly linked list, return true if it is a palindrome or false otherwise.



Input: head = [1, 2, 2, 1]

Output: true



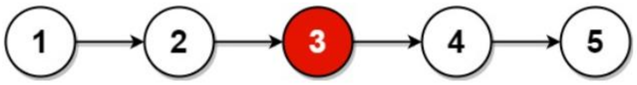
Input: head = [1, 2]

Output: false

```
class ListNode
      {
      # Write Code Here
      }
public class Solution {
    public boolean isPalindrome(ListNode head)
      {
        # Write Code Here
      }

    public static void main(String[] args)
      {
      # Write Code Here
      }
}
```

7.6 Middle of the Linked List

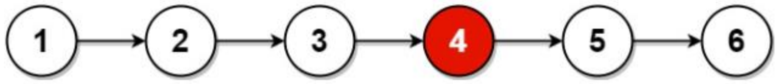
Given the head of a singly linked list, return the middle node of the linked list. If there are two middle nodes, return the second middle node.



Input: head = [1, 2, 3, 4, 5]

Output: [3, 4, 5]

Explanation: The middle node of the list is node 3.



Input: head = [1, 2, 3, 4, 5, 6]

Output: [4, 5, 6]

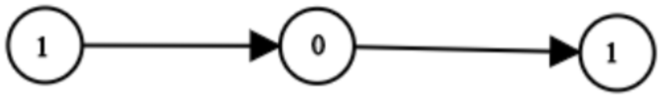
Explanation: Since the list has two middle nodes with values 3 and 4, we return the second one.

```
class ListNode
      {
      # Write Code Here
      }
}
public class Solution
    {
      # Write Code Here
    }

    public static void main(String[] args) {
       # Write Code Here
    }
}
```

7.7 Convert Binary Number in a Linked List to Integer

Given head which is a reference node to a singly-linked list. The value of each node in the linked list is either 0 or 1. The linked list holds the binary representation of a number.

Return the decimal value of the number in the linked list. The most significant bit is at the head of the linked list.



Input: head = [1, 0, 1]

Output: 5

Explanation: (101) in base 2 = (5) in base 10

Input: head = [0]

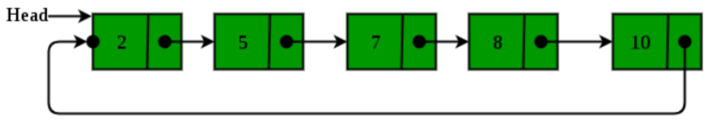
Output: 0

```
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
}

public class Solution
        {
          # Write Code Here
       }

    public static void main(String[] args)
      {
      # Write Code Here
      }
}
```

8. Circular Single Linked List and Doubly Linked List

8.1 Circular Linked List

The circular linked list is a linked list where all nodes are connected to form a circle. In a circular linked list, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.



Operations on the circular linked list:

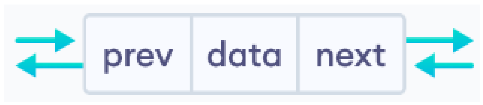
1. Insertion at the beginning
2. Insertion at the end
3. Insertion in between the nodes
4. Deletion at the beginning
5. Deletion at the end
6. Deletion in between the nodes
7. Traversal

```
import java.util.ArrayList;
public class Main{
        static class Node{
                int data;
                Node next;
                Node(int data){
                        this.data = data;
                        this.next = null;
                }
        }
        static class CircularLinkedList
        {
        # Write Code Here
        }
        Node addAfter(int data, int item)
        {
        # Write Code Here
        }
void deleteNode(Node last, int key)
        {
        # Write Code Here
        }
                        System.
```

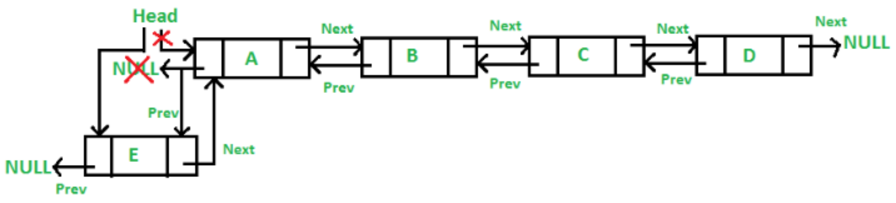
8.2 Doubly Linked List

The A doubly linked list is a type of linked list in which each node consists of 3 components:

1. *prev - address of the previous node
2. data - data item
3. *next - address of next node.



Double Linked List Node



Operations on the Double Linked List:

1. Insertion at the beginning
2. Insertion at the end
3. Insertion in between the nodes
4. Deletion at the beginning
5. Deletion at the end
6. Deletion in between the nodes
7. Traversal

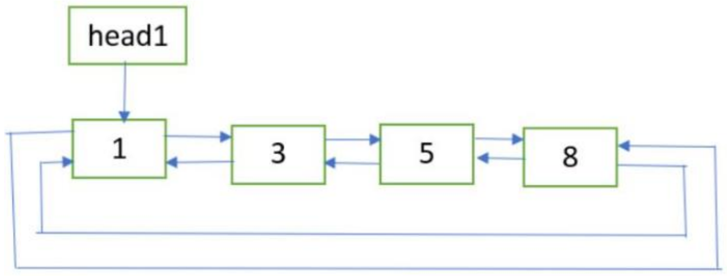
```
import java.util.Scanner;

class Node {
    int data;
    Node next;
    Node prev;

    Node(int data) {
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}

class DLinkedList {
    Node head;
    int ctr;

    DLinkedList() {
        this.head = null;
        this.ctr = 0;
    }

    void insertBeg(int data)
      {
      # Write Code Here
      }
    void insertEnd(int data)
      {
      # Write Code Here
```

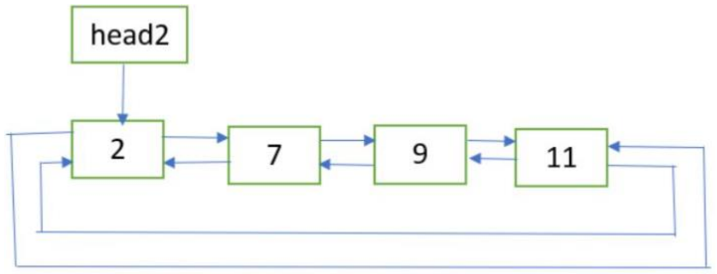
```
        }

    void deleteBeg()
      {
      # Write Code Here
      }
    void deleteEnd()
      {
        # Write Code Here
      }

    void insertPos(int pos, int data)
      {
      # Write Code Here
      }

    void deletePos(int pos)
      {
        # Write Code Here
      }

    void traverseF()
      {
        # Write Code Here
      }

    void traverseR()
      {
       # Write Code Here
      }

public class Main {
    public static void main(String[] args)
      {
       # Write Code Here
      }
}
```

8.3 Sorted Merge of Two Sorted Doubly Circular Linked Lists

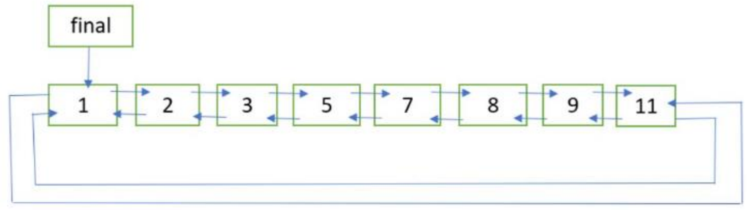
Given two sorted Doubly circular Linked List containing n1 and n2 nodes respectively. The problem is to merge the two lists such that resultant list is also in sorted order.

Input: List 1 and List 2





Output: Merged List



Procedure for Merging Doubly Linked List:

1. If head1 == NULL, return head2.
2. If head2 == NULL, return head1.
3. Let **last1** and **last2** be the last nodes of the two lists respectively. They can be obtained with the help of the previous links of the first nodes.
4. Get pointer to the node which will be the last node of the final list. If last1.data < last2.data, then **last_node** = last2, Else **last_node** = last1.
5. Update last1.next = last2.next = NULL.
6. Now merge the two lists as two sorted doubly linked list are being merged. Refer **merge** procedure of this post. Let the first node of the final list be **finalHead**.
7. Update finalHead.prev = last_node and last_node.next = finalHead.
8. Return **finalHead**.

```
class Node {
    int data;
    Node next, prev;

    Node(int data) {
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}

public class SortedMergeDoublyCircularLinkedList
      {
      # Write Code Here
      }

    static Node mergeUtil(Node head1, Node head2)
      {
      # Write Code Here
```

```
       }

    static void printList(Node head)
       {
       # Write Code Here
       }

    public static void main(String[] args)
       {
         # Write Code Here
       }
}
```

8.4 Delete all occurrences of a given key in a Doubly Linked List

Given a doubly linked list and a key x. The problem is to delete all occurrences of the given key x from the doubly linked list.

Input: 2 <-> 2 <-> 10 <-> 8 <-> 4 <-> 2 <-> 5 <-> 2
x = 2
Output: 10 <-> 8 <-> 4 <-> 5

Algorithm:
delAllOccurOfGivenKey (head_ref, x)

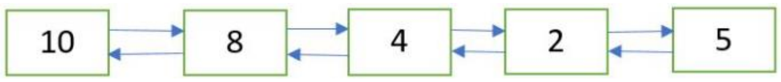
```
   if head_ref == NULL
      return
   Initialize current = head_ref
   Declare next
   while current != NULL
      if current->data == x
         next = current->next
         deleteNode(head_ref, current)
         current = next
      else
         current = current->next
```

```
class Node {
    int data;
    Node next, prev;

    Node(int data) {
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}

public class DeleteOccurrenceInDoublyLinkedList
       {
        # Write Code Here
       }
    static Node deleteAllOccurOfX(Node head, int x)
       {
       # Write Code Here
       }

    static void printList(Node head)
       {
        # Write Code Here
       }

    public static void main(String[] args)
       {
       # Write Code Here
       }
}
```

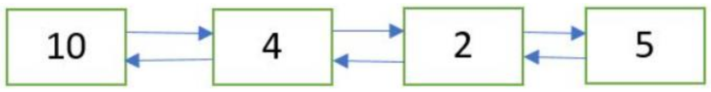
8.5 Delete a Doubly Linked List Node at a Given Position

Given a doubly linked list and a position n. The task is to delete the node at the given position n from the beginning.

Input: Initial doubly linked list



Output: Doubly Linked List after deletion of node at position n = 2



Procedure:

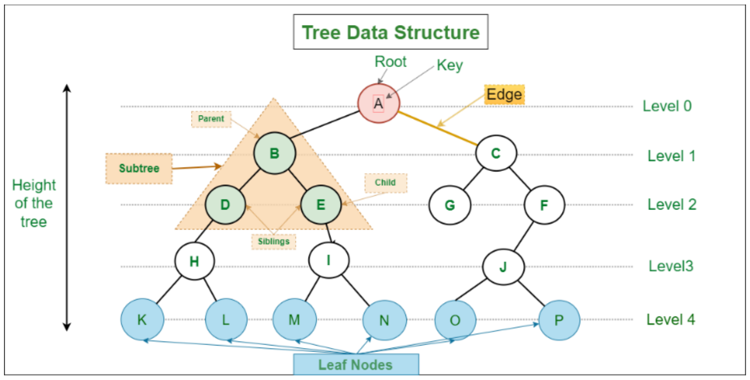
1. Get the pointer to the node at position n by traversing the doubly linked list up to the nth node from the beginning.
2. Delete the node using the pointer obtained in Step 1.

```
class Node {
    int data;
    Node next, prev;

    Node(int data) {
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}

public class DeleteNodeAtGivenPosition
     {
       # Write Code Here
     }
    static Node deleteNode(Node head, Node del)
     {
        # Write Code Here
     }

    static Node deleteNodeAtGivenPos(Node head, int n)
     {
        # Write Code Here
     }
    static void printList(Node head)
     {
      # Write Code Here
     }
}
```

9. Trees

9.1 Tree Creation and Basic Tree Terminologies

A tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.



Basic Terminologies in Tree:

1. **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. {B} is the parent node of {D, E}.
2. **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {D, E} are the child nodes of {B}.
3. **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
4. **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {K, L, M, N, O, P} are the leaf nodes of the tree.
5. **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {A, B} are the ancestor nodes of the node {E}
6. **Descendant:** Any successor node on the path from the leaf node to that node. {E, I} are the descendants of the node {B}.
7. **Sibling:** Children of the same parent node are called siblings. {D, E} are called siblings.
8. **Level of a node:** The count of edges on the path from the root node to that node. The root node has level 0.
9. **Internal node:** A node with at least one child is called Internal Node.
10. **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
11. **Subtree:** Any node of the tree along with its descendant.

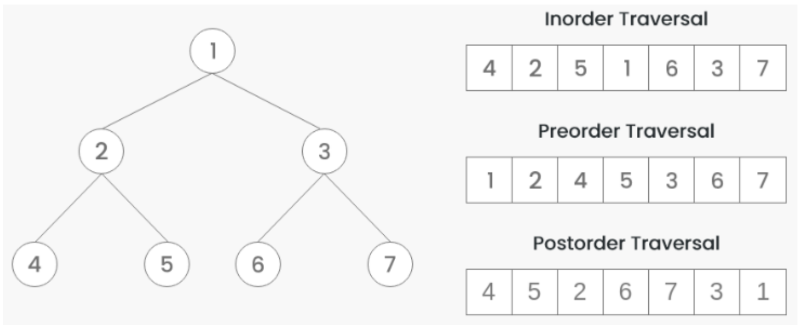
```
import java.util.ArrayList;
import java.util.List;

public class TreeBasicTerminologies
       {

        # Write Code Here
       }
static void printChildren(int root, List<List<Integer>> adj)
```

```
        {
          # Write Code Here
        }
    static void printLeafNodes(int root, List<List<Integer>> adj)
        {
          # Write Code Here
        }
    static void printDegrees(int root, List<List<Integer>> adj)
        {
          # Write Code Here
        }
    public static void main(String[] args)
        {
          # Write Code Here
        }
}
```

9.2 Binary Tree Traversal Techniques

A binary tree data structure can be traversed in following ways:

1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal
4. Level Order Traversal



Algorithm Inorder (tree)

1. Traverse the left subtree, i.e., call Inorder(left->subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right->subtree)

Algorithm Preorder (tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left->subtree)
3. Traverse the right subtree, i.e., call Preorder(right->subtree)

Algorithm Postorder (tree)

1. Traverse the left subtree, i.e., call Postorder(left->subtree)
2. Traverse the right subtree, i.e., call Postorder(right->subtree)

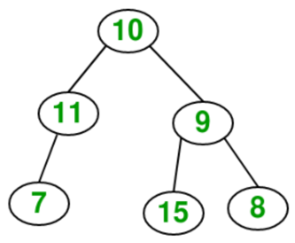
3. Visit the root.

```
import java.util.Scanner;
class Node
        {
          # Write Code Here
        }
class BT {
         Node root;
         BT() {
          this.root = null;
    }
    void insert(int data)
        {
        # Write Code Here
        }
    Node insertRec(Node root, int data)
        {
          # Write Code Here
        }
    void postorder(Node root)
        {
          # Write Code Here
        }
    void preorder(Node root)
        {
          # Write Code Here
       }
    }
    void inorder(Node root)
        {
        # Write Code Here
        }
}

public class BinaryTreeTraversal {
    public static void main(String[] args)
             {
        # Write Code Here
            }
        }
    }
}
```

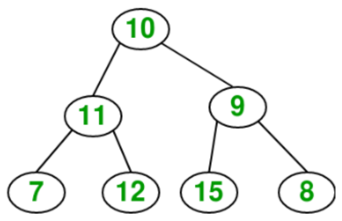
9.3 Insertion in a Binary Tree in Level Order

Given a binary tree and a key, insert the key into the binary tree at the first position available in level order.

Input: Consider the tree given below



Output:



After inserting 12

The idea is to do an iterative level order traversal of the given tree using queue. If we find a node whose left child is empty, we make a new key as the left child of the node. Else if we find a node whose right child is empty, we make the new key as the right child. We keep traversing the tree until we find a node whose either left or right child is empty.

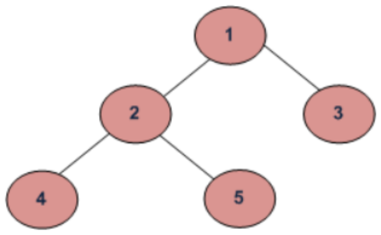
```
class Node
{
# Write Code Here
}

public class BinaryTreeInsertion
{
# Write Code Here
       }
    static Node insert(Node root, int key)
{
# Write Code Here
       }

    public static void main(String[] args)
{
        # Write Code Here
       }
}
```

9.4 Finding the Maximum Height or Depth of a Binary Tree

Given a binary tree, the task is to find the height of the tree. The height of the tree is the number of edges in the tree from the root to the deepest node.

Note: The height of an empty tree is 0.

Input: Consider the tree below



Recursively calculate the height of the left and the right subtrees of a node and assign height to the node as max of the heights of two children plus 1.

maxDepth('1') = max(maxDepth('2'), maxDepth('3')) + 1 = 2 + 1
because recursively
maxDepth('2') = max (maxDepth('4'), maxDepth('5')) + 1 = 1 + 1 and (as height of both '4' and '5' are 1)
maxDepth('3') = 1

Procedure:

- Recursively do a Depth-first search.
- If the tree is empty then return 0
- Otherwise, do the following
  - Get the max depth of the left subtree recursively i.e. call maxDepth(tree->left-subtree)
  - Get the max depth of the right subtree recursively i.e. call maxDepth(tree->right-subtree)
  - Get the max of max depths of left and right subtrees and add 1 to it for the current node.

$$max_depth = max(maxdeptofleftsubtree, maxdepthofrightsubtree) + 1$$

- Return max_depth.

```
class Node
{
    int data;
    Node left, right;

    Node(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}

public class MaximumDepthOfTree
      {
        # Write Code Here
      }

    public static void main(String[] args)
      {
        # Write Code Here
      }
```

9.5 Deletion in a Binary Tree

Given a binary tree, delete a node from it by making sure that the tree shrinks from the bottom (i.e. the deleted node is replaced by the bottom-most and rightmost node).

Input: Delete 10 in below tree

```
      10
    /   \
  20     30
```

Output:

```
      30
    /
 20
```

Input: Delete 20 in below tree

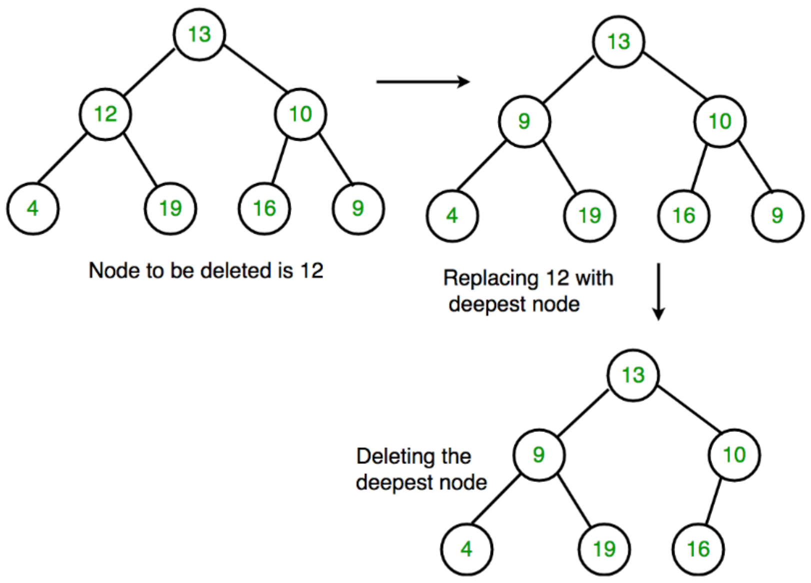
```
      10
    /   \
 20     30
           \
            40
```

Output:

```
      10
   /     \
40        30
```

Algorithm:

1. Starting at the root, find the deepest and rightmost node in the binary tree and the node which we want to delete.
2. Replace the deepest rightmost node's data with the node to be deleted.
3. Then delete the deepest rightmost node.



```
class Node {
    int data;
    Node left, right;

    Node(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}

public class BinaryTreeDeletion
{
        # Write Code Here
      }
    static void deleteDeepest(Node root, Node dNode)
{
        # Write Code Here
}
    static Node deletion(Node root, int key)
{
        # Write Code Here
}


    public static void main(String[] args)
{
        # Write Code Here
}
}
```

10. Binary Search Tree (BST)

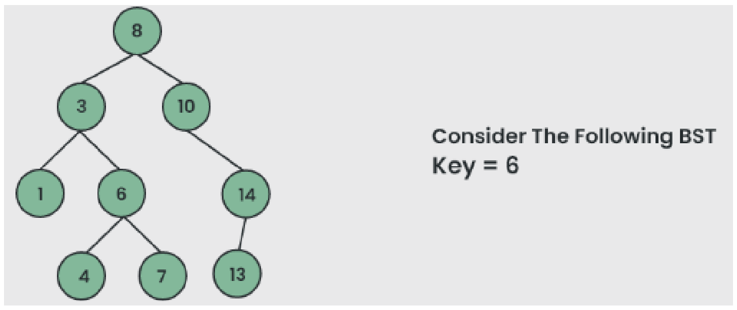
10.1 Searching in Binary Search Tree

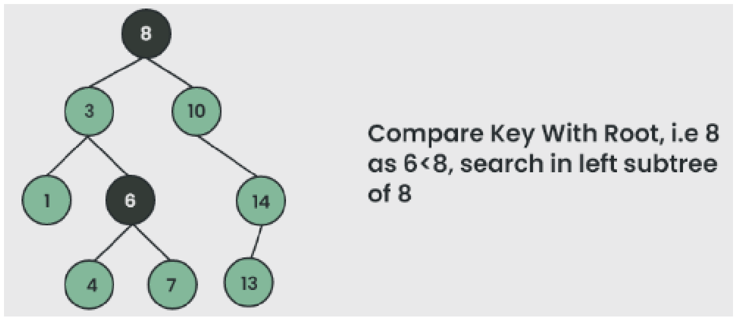
Given a BST, the task is to delete a node in this BST. For searching a value in BST, consider it as a sorted array. Perform search operation in BST using Binary Search Algorithm.

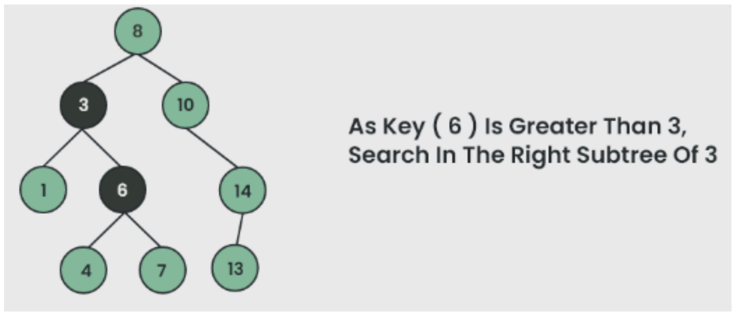
Algorithm to search for a key in a given Binary Search Tree:

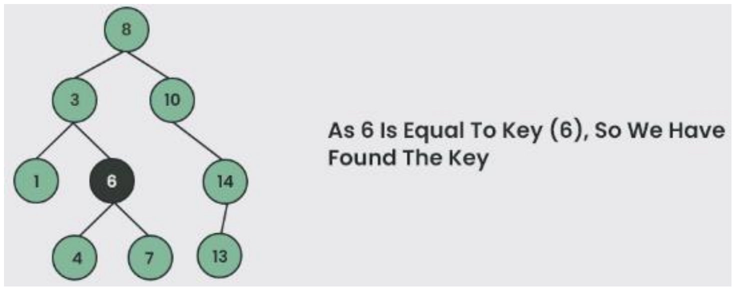
Let's say we want to search for the number **X,** We start at the root. Then:

- We compare the value to be searched with the value of the root.
  - If it's equal we are done with the search if it's smaller we know that we need to go to the left subtree because in a binary search tree all the elements in the left subtree are smaller and all the elements in the right subtree are larger.
- Repeat the above step till no more traversal is possible
- If at any iteration, key is found, return True. Else False.






8
3
10
1
6
14
4
7
13
As Key (6) Is Greater Than 3,
Search In The Right Subtree Of 3

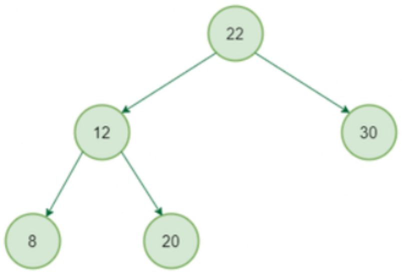


```
// Node class to represent each node of the BST
class Node {
    int key;
    Node left, right;

    public Node(int item) {
        key = item;
        left = right = null;
    }
}
class BST
{
# Write Code Here
}
    Node search(int key) {
      return searchRec(root, key);
}
    Node searchRec(Node root, int key)
      {
      # Write Code Here
      }
    public static void main(String[] args)
{
       # Write Code Here
      }
}
```

10.2 Find the node with Minimum Value in a BST

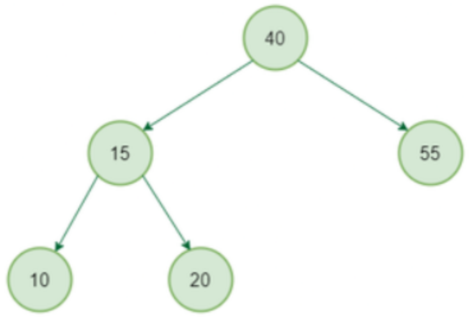
Write a function to find the node with minimum value in a Binary Search Tree.

Input: Consider the tree given below



Output: 8

Input: Consider the tree given below



Output: 10

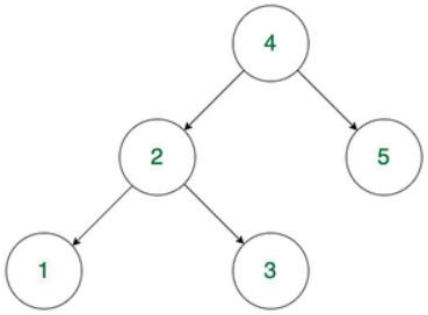
```
import java.util.ArrayList;
import java.util.List;

class Node {
    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = right = null;
    }
}

class BinarySearchTree
      {
        # Write Code Here
      }

    public static void main(String[] args)
      {
      # Write Code Here
      }
}
```

10.3 Check if a Binary Tree is BST or not

A binary search tree (BST) is a node-based binary tree data structure that has the following properties.

1. The left subtree of a node contains only nodes with keys less than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. Both the left and right subtrees must also be binary search trees.
4. Each node (item in the tree) has a distinct key.

Input: Consider the tree given below



Output: Check if max value in left subtree is smaller than the node and min value in right subtree greater than the node, then print it “Is BST” otherwise “Not a BST”

Procedure:
1. If the current node is null then return true
2. If the value of the left child of the node is greater than or equal to the current node then return false
3. If the value of the right child of the node is less than or equal to the current node then return false
4. If the left subtree or the right subtree is not a BST then return false
5. Else return true

```
class Node {
    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = right = null;
    }
}

class BinaryTree
      {
      # Write Code Here
      }

    boolean isBST(Node node) {
        return isBSTUtil(node, Integer.MIN_VALUE, Integer.MAX_VALUE);
    }

    boolean isBSTUtil(Node node, int min, int max)
      {
      # Write Code Here
      }

    public static void main(String[] args)
      {
      # Write Code Here
      }
}
```

10.4 Second Largest Element in BST

Given a Binary search tree (BST), find the second largest element.

Input: Root of below BST

```
         10
        /
      5
```

Output: 5

Input: Root of below BST

```
        10

       /  \

     5     20

             \

              30
```

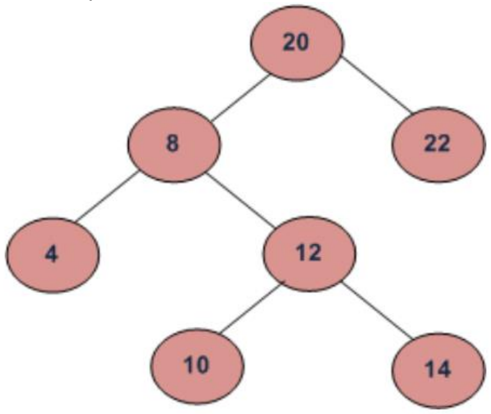
Output: 20

Procedure: The second largest element is second last element in inorder traversal and second element in reverse inorder traversal. We traverse given Binary Search Tree in reverse inorder and keep track of counts of nodes visited. Once the count becomes 2, we print the node.

```
class Node {
    int key;
    Node left, right;

    public Node(int item) {
        key = item;
        left = right = null;
    }
}

class BinarySearchTree
      {
      # Write Code Here
      }

        secondLargestUtil(node.right);
        count++;

        // If count is equal to 2 then this is the second largest
        if (count == 2) {
            System.out.println("The second largest element is " + node.key);
            return;
        }
        secondLargestUtil(node.left);
    }

    // Function to find the second largest element
    void secondLargest(Node node) {
        count = 0;
        secondLargestUtil(node);
    }

    // Driver code
    public static void main(String[] args)
      {
      # Write Code Here
      }
}
```

Try:

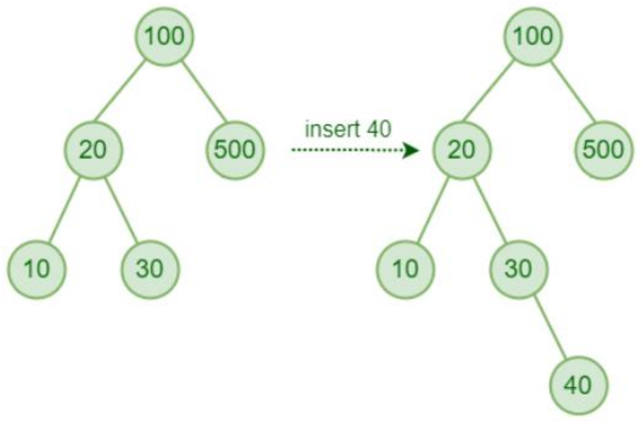
1. **Kth largest element in BST when modification to BST is not allowed:** Given a Binary Search Tree (BST) and a positive integer k, find the k'th largest element in the Binary Search Tree. For a given BST, if k = 3, then output should be 14, and if k = 5, then output should be 10.



10.5 Insertion in Binary Search Tree (BST)

Given a Binary search tree (BST), the task is to insert a new node in this BST.

Input: Consider a BST and insert the element 40 into it.



Procedure for inserting a value in a BST:

A new key is always inserted at the leaf by maintaining the property of the binary search tree. We start searching for a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node. The below steps are followed while we try to insert a node into a binary search tree:

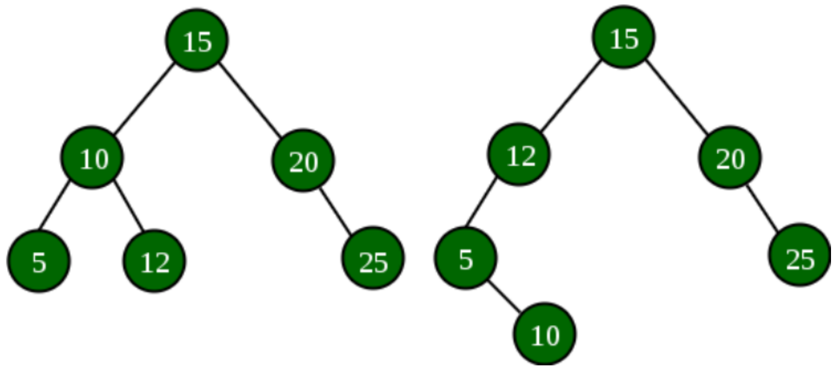
- Check the value to be inserted (say X) with the value of the current node (say val) we are in:
  - If X is less than val move to the left subtree.
  - Otherwise, move to the right subtree.
- Once the leaf node is reached, insert X to its right or left based on the relation between X and the leaf node's value.

```
// A utility class that represents an individual node in a BST
class Node {
    int val;
    Node left, right;

    public Node(int item) {
        val = item;
        left = right = null;
    }
}

class BinarySearchTree
      {

      # Write Code Here

      }

    void inorder()
      {
        inorderRec(root);
      }

    void inorderRec(Node root)
      {
        # Write Code Here
        }
    }

    // Driver code
    public static void main(String[] args)
      {
        # Write Code Here
      }
}
```

Try:

1. **Check if two BSTs contain same set of elements:** Given two Binary Search Trees consisting of unique positive elements, we have to check whether the two BSTs contain the same set of elements or not.

Input: Consider two BSTs which contains same set of elements {5, 10, 12, 15, 20, 25}, but the structure of the two given BSTs can be different.



11. AVL Tree

11.1 Insertion in an AVL Tree

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

- Left Rotation
- Right Rotation

T1, T2 and T3 are subtrees of the tree, rooted with y (on the left side) or x (on the right side)

```
    y                               x
   / \     Right Rotation          /  \
  x   T3   - - - - - - - >        T1   y
 / \       < - - - - - - -            / \
T1  T2     Left Rotation            T2  T3
```

Keys in both of the above trees follow the following order
keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.

Procedure for inserting a node into an AVL tree

Let the newly inserted node be w

- Perform standard BST insert for w.
- Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that need to be handled as x, y and z can be arranged in 4 ways.
- Following are the possible 4 arrangements:
  - y is the left child of z and x is the left child of y (Left Left Case)
  - y is the left child of z and x is the right child of y (Left Right Case)
  - y is the right child of z and x is the right child of y (Right Right Case)
  - y is the right child of z and x is the left child of y (Right Left Case)

```
class TreeNode {
    int val, height;
    TreeNode left, right;

    TreeNode(int d) {
        val = d;
        height = 1;
    }
}
class AVL_Tree {

    # write the code
    }

    TreeNode leftRotate(TreeNode x)
      {
      # write the code
      }
    public static void main(String[] args) {
        # write the code
      }
```

11.2 Deletion in an AVL Tree

Given an AVL tree, make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

1. Left Rotation
2. Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)

```
    y                               x
   / \     Right Rotation          /  \
  x   T3   - - - - - - - >        T1   y
 / \       < - - - - - - -            / \
T1  T2     Left Rotation            T2  T3
```

Keys in both of the above trees follow the following order
keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.

Procedure to delete a node from AVL tree:

Let w be the node to be deleted

1. Perform standard BST delete for w.
2. Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.
3. Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
    i. y is left child of z and x is left child of y (Left Left Case)
    ii. y is left child of z and x is right child of y (Left Right Case)
    iii. y is right child of z and x is right child of y (Right Right Case)

iv. y is right child of z and x is left child of y (Right Left Case)

```
class TreeNode
     {
     int val, height;
     TreeNode left, right;

     TreeNode(int d) {
      val = d;
      height = 1;
   }
}

class AVL_Tree {

   TreeNode leftRotate(TreeNode z)
     {
     # Write code here
     }

   TreeNode rightRotate(TreeNode z)
     {
     # Write code here
     }

     TreeNode insert(TreeNode node, int key)
     {
     # Write code here
      }
```

11.3 Count Greater Nodes in AVL Tree

Given an AVL tree, calculate number of elements which are greater than given value in AVL tree.

Input: x = 5

Root of below AVL tree

```
       9
      / \
     1   10
    / \   \
   0   5   11
  /   / \
-1   2   6
```

Output: 4

Explanation: There are 4 values which are greater than 5 in AVL tree which are 6, 9, 10 and 11.

```
class TreeNode {
    int key, height, desc;
    TreeNode left, right;

    TreeNode(int d) {
        key = d;
        height = 1;
        desc = 0;
    }
}

class AVL_Tree
       {
       # Write code here
       }

    TreeNode insert(TreeNode node, int key)
       {
       # Write code here
       }
    TreeNode minValueNode(TreeNode node)
       {
       # Write code here
       }

    TreeNode deleteNode(TreeNode root, int key)
       {
        # Write code here
       }
void preOrder(TreeNode node)
       {
       # Write code here
       }

public class Main
       {
       # Write code here
       }
}
```

11.4 Minimum Number of Nodes in an AVL Tree with given Height

Given the height of an AVL tree 'h', the task is to find the minimum number of nodes the tree can have.

Input: H = 0
Output: N = 1
Only '1' node is possible if the height
of the tree is '0' which is the root node.

Input: H = 3
Output: N = 7

Recursive approach:
In an AVL tree, we have to maintain the height balance property, i.e. difference in the height of the left and the right subtrees cannot be other than -1, 0 or 1 for each node.
We will try to create a recurrence relation to find minimum number of nodes for a given height, n(h).

- For height = 0, we can only have a single node in an AVL tree, i.e. n(0) = 1
- For height = 1, we can have a minimum of two nodes in an AVL tree, i.e. n(1) = 2

- Now for any height ‘h’, root will have two subtrees (left and right). Out of which one has to be of height h-1 and other of h-2. [root node excluded]
- So, n(h) = 1 + n(h-1) + n(h-2) is the required recurrence relation for h>=2 [1 is added for the root node]

```
public class AVLTreeMinimumNodes {

    public static int AVLnodes(int height)
      {
      # Write code here
      }

    public static void main(String[] args) {
        int H = 3;
        System.out.println(AVLnodes(H)); // Output: 4
    }
}
```

12. Graph Traversal

12.1 Breadth First Search

The **Breadth First Search (BFS)** algorithm is used to search a graph data structure for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.
For a given graph G, print BFS traversal from a given source vertex.

```
import java.util.*;

public class Graph {
    private Map<Integer, List<Integer>> graph;

    public Graph()
      {
        graph = new HashMap<>();
    }

    public void addEdge(int u, int v) {
        if (!graph.containsKey(u)) {
            graph.put(u, new ArrayList<>());
        }
        graph.get(u).add(v);
    }


          public void BFS(int s)
      {
      # Write code here
      }

    public static void main(String[] args)
      {
      # Write code here
      }
}
```

Output: Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1

12.2 Depth First Search

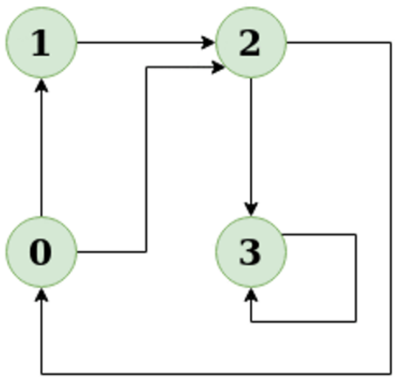
Depth First Traversal (or DFS) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.

For a given graph G, print DFS traversal from a given source vertex.

Input: n = 4, e = 6
0 -> 1, 0 -> 2, 1 -> 2, 2 -> 0, 2 -> 3, 3 -> 3

Output: DFS from vertex 1: 1 2 0 3

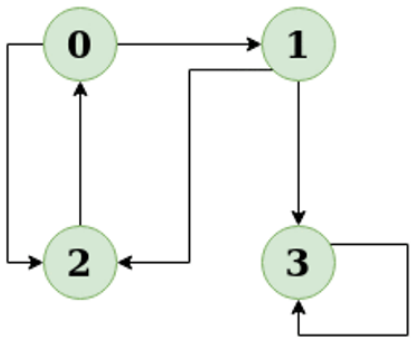
Explanation:
DFS Diagram:



Input: n = 4, e = 6
2 -> 0, 0 -> 2, 1 -> 2, 0 -> 1, 3 -> 3, 1 -> 3

Output: DFS from vertex 2: 2 0 1 3

Explanation:
DFS Diagram:



```
import java.util.*;
class Graph {
    private Map<Integer, List<Integer>> graph;

    public Graph() {
        // Initialize the graph as a HashMap of ArrayLists
        graph = new HashMap<>();
    }
    public void addEdge(int u, int v)
      {
      # Write code here
     }

    public void DFS(int v) {

        DFSUtil(v, visited);
    }
    public static void main(String[] args)
      {
      # Write code here
      }
```
}

12.3 Best First Search (Informed Search)

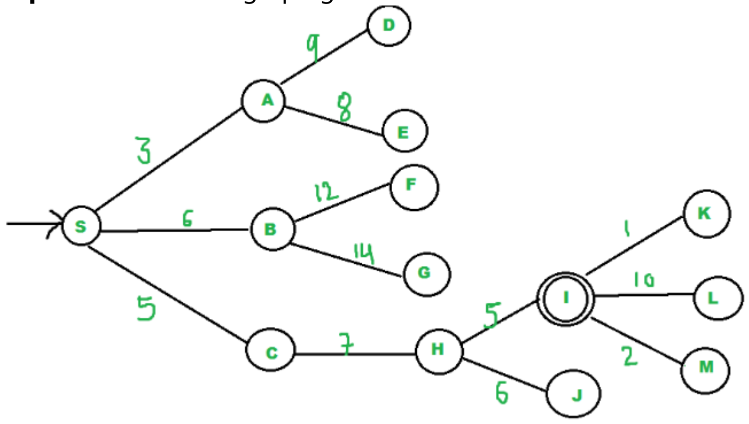
The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.

Implementation of Best First Search:
We use a priority queue or heap to store the costs of nodes that have the lowest evaluation function value. So the implementation is a variation of BFS, we just need to change Queue to PriorityQueue.

Algorithm:
Best-First-Search(Graph g, Node start)

```
  1) Create an empty PriorityQueue
     PriorityQueue pq;
  2) Insert "start" in pq.
     pq.insert(start)
  3) Until PriorityQueue is empty
       u = PriorityQueue.DeleteMin
       If u is the goal
          Exit
       Else
          Foreach neighbor v of u
             If v "Unvisited"
                 Mark v "Visited"
                 pq.insert(v)
          Mark u "Examined"
End procedure
```

Input: Consider the graph given below.



- We start from source "S" and search for goal "I" using given costs and Best First search.

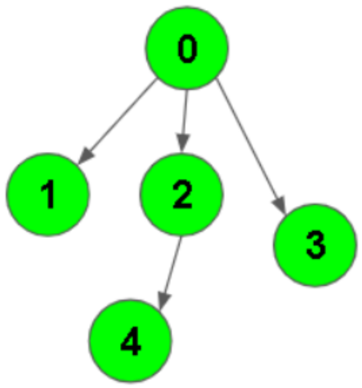
- pq initially contains S
    - We remove S from pq and process unvisited neighbors of S to pq.
    - pq now contains {A, C, B} (C is put before B because C has lesser cost)
- We remove A from pq and process unvisited neighbors of A to pq.
    - pq now contains {C, B, E, D}
- We remove C from pq and process unvisited neighbors of C to pq.
    - pq now contains {B, H, E, D}
- We remove B from pq and process unvisited neighbors of B to pq.
    - pq now contains {H, E, D, F, G}
- We remove H from pq.
- Since our goal "I" is a neighbor of H, we return.

```
import java.util.*;

public class BestFirstSearch {
    static int v = 14;
    static List<List<Pair<Integer, Integer>>> graph = new ArrayList<>();
    static void addedge(int x, int y, int cost) {
        graph.get(x).add(new Pair<>(y, cost));
        graph.get(y).add(new Pair<>(x, cost));
    }

    static void best_first_search(int actual_Src, int target, int n)
      {
      # Write code here
      }

    public static void main(String[] args)
      {
      # Write code here
      }
}
```

12.4 Breadth First Traversal of a Graph

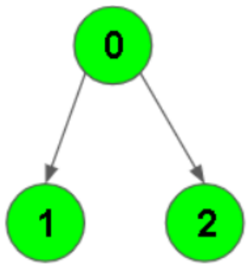
Given a directed graph. The task is to do Breadth First Traversal of this graph starting from 0.
One can move from node u to node v only if there's an edge from u to v. Find the BFS traversal of the graph starting from the 0th vertex, from left to right according to the input graph. Also, you should only take nodes directly or indirectly connected from Node 0 in consideration.

Input: Consider the graph given below where V = 5, E = 4, edges = {(0,1), (0,2), (0,3), (2,4)}



Output: 0 1 2 3 4
Explanation:
0 is connected to 1, 2, and 3.
2 is connected to 4.
So starting from 0, it will go to 1 then 2 then 3. After this 2 to 4, thus BFS will be 0 1 2 3 4.

Input: Consider the graph given below where V = 3, E = 2, edges = {(0, 1), (0, 2)}



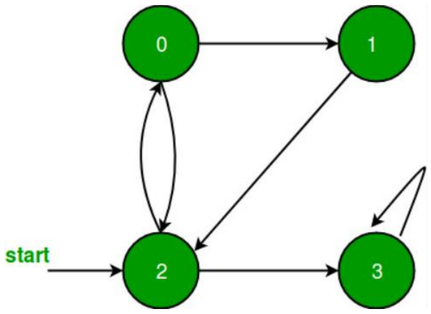
Output: 0 1 2
Explanation:
0 is connected to 1, 2. So starting from 0, it will go to 1 then 2, thus BFS will be 0 1 2.
Your task is to complete the function **bfsOfGraph()** which takes the integer V denoting the number of vertices and adjacency list as input parameters and returns a list containing the BFS traversal of the graph starting from the 0th vertex from left to right.

```
import java.util.*;

class Graph {
    private int V;
    private LinkedList<Integer>[] adj;

    Graph(int v) {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i)
            adj[i] = new LinkedList();
    }

    void addEdge(int v, int w) {
        adj[v].add(w);
    }

    void BFS(int s)
      {
      # Write Code Here
        }
 }

    public static void main(String args[]) {
       # Write Code Here
    }
}
```

12.5 Depth First Search (DFS) for Disconnected Graph

Given a Disconnected Graph, the task is to implement DFS or Depth First Search Algorithm for this Disconnected Graph.

Input: Consider the graph given below.



Output: 0 1 2 3

Procedure for DFS on Disconnected Graph:
Iterate over all the vertices of the graph and for any unvisited vertex, run a DFS from that vertex.

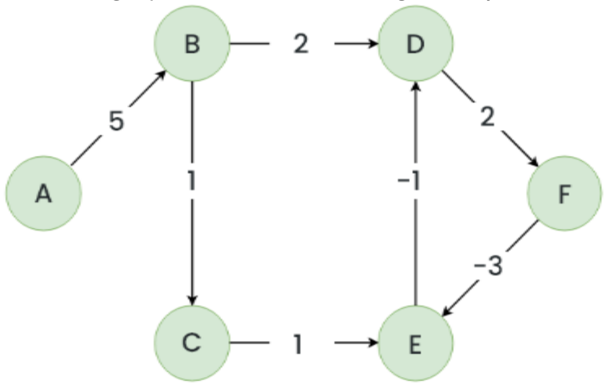
```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

// Class representing a directed graph using adjacency list representation
class Graph
      {
       # Write Code Here
      }

    public Graph()
      {
        graph = new HashMap<>();
      }

    public void addEdge(int u, int v)
```

```
        {
        # Write Code Here
        }
    private void DFSUtil(int v, boolean[] visited)
        {
        # Write Code Here
        }
    }

    public void DFS() {
        boolean[] visited = new boolean[graph.size()];
        # Write Code Here
        }

    public static void main(String[] args)
        {
        # Write Code Here
        }
}
```

Try:

1. **Detect a negative cycle in a Graph (Bellman Ford):** A Bellman-Ford algorithm is also guaranteed to find the shortest path in a graph, similar to Dijkstra's algorithm. Although Bellman-Ford is slower than Dijkstra's algorithm, it is capable of handling graphs with negative edge weights, which makes it more versatile. The shortest path cannot be found if there exists a negative cycle in the graph. If we continue to go around the negative cycle an infinite number of times, then the cost of the path will continue to decrease (even though the length of the path is increasing).

Consider a graph G and detect a negative cycle in the graph using Bellman Ford algorithm.



13. Minimum Spanning Tree (MST)

13.1 Kruskal's Algorithm

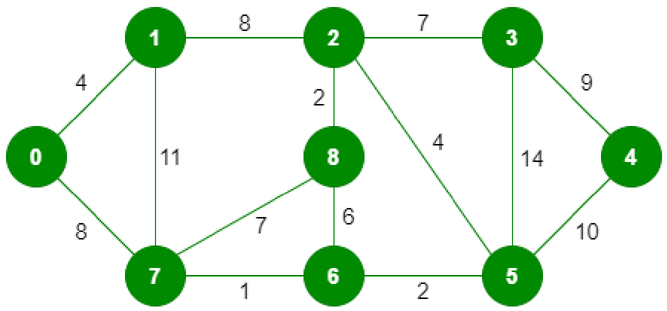
In Kruskal's algorithm, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first and the maximum weighted edge at last.

MST using Kruskal's algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far.

Input: For the given graph G find the minimum cost spanning tree.



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

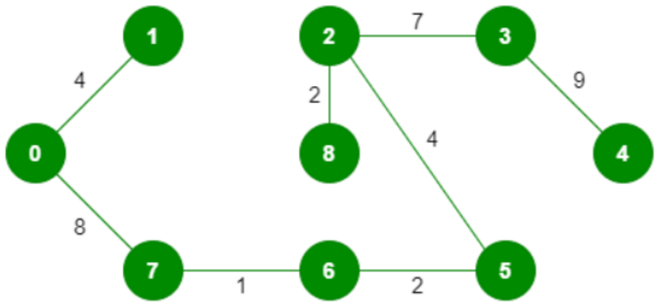
After sorting:

Weight	Source	Destination
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7

14	3	5

Now pick all edges one by one from the sorted list of edges.

Output:



```
// Kruskal's algorithm to find minimum Spanning Tree of a given connected,
import java.util.*;

public class Graph {

    class Edge implements Comparable<Edge> {
        int src, dest, weight;

        // Comparator function used for sorting edges
        public int compareTo(Edge compareEdge) {
            return this.weight - compareEdge.weight;
        }
    }

    private int V; // Number of vertices
    private List<Edge> edges; // List of edges

    public Graph(int vertices) {
        this.V = vertices;
        this.edges = new ArrayList<>();
    }

    public void addEdge(int u, int v, int w)
      {
      # Write Code Here
      }
    private void union(int[] parent, int[] rank, int x, int y)
      {
      # Write Code Here
      }

    public void KruskalMST()
      {
      # Write Code Here
      }
```

```
 public static void main(String[] args) {
        Graph g = new Graph(4);
        g.addEdge(0, 1, 10);
        g.addEdge(0, 2, 6);
        g.addEdge(0, 3, 5);
        g.addEdge(1, 3, 15);
        g.addEdge(2, 3, 4);

        // Function call
        g.KruskalMST();
    }
}
```

Output: Following are the edges in the constructed MST

```
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19
```

13.2 Prim’s Algorithm

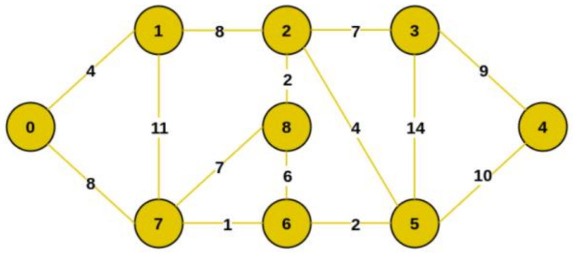
The Prim’s algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

Prim’s Algorithm:

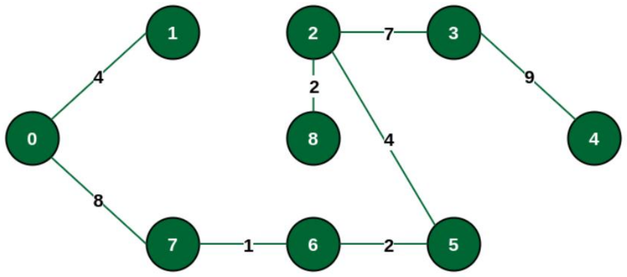
The working of Prim’s algorithm can be described by using the following steps:

1. Determine an arbitrary vertex as the starting vertex of the MST.
2. Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
3. Find edges connecting any tree vertex with the fringe vertices.
4. Find the minimum among these edges.
5. Add the chosen edge to the MST if it does not form any cycle.
6. Return the MST and exit

Input: For the given graph G find the minimum cost spanning tree.



Output: The final structure of the MST is as follows and the weight of the edges of the MST is (4 + 8 + 1 + 2 + 4 + 2 + 7 + 9) = 37.



```
import java.util.Arrays;

public class Graph {

    private int V; // Number of vertices
    private int[][] graph; // Adjacency matrix representation of graph

    public Graph(int vertices) {
        this.V = vertices;
        this.graph = new int[V][V];
    }
    public void printMST(int[] parent) {
        System.out.println("Edge \tWeight");
        for (int i = 1; i < V; i++) {
            System.out.println(parent[i] + " - " + i + "\t" + graph[i][parent[i]]);
        }
    }
    private int minKey(int[] key, boolean[] mstSet)
      {
              # Write Code Here
      }
    public void primMST()
      {
              # Write Code Here

      }
    public static void main(String[] args)
      {
              # Write Code Here
      }
}
```

Output:

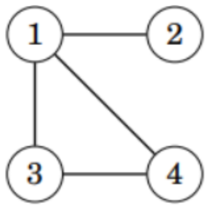
```
Edge    Weight
0 - 1     2
1 - 2     3
0 - 3     6
1 - 4     5
```

13.3 Total Number of Spanning Trees in a Graph

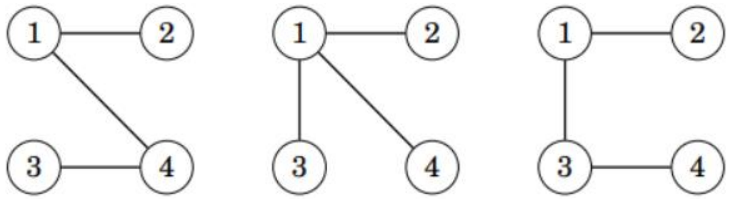
If a graph is a complete graph with n vertices, then total number of spanning trees is $n^{(n-2)}$ where n is the number of nodes in the graph. In complete graph, the task is equal to counting different labeled trees with n nodes for which have Cayley's formula.

Laplacian matrix:
A Laplacian matrix L, where L[i, i] is the degree of node i and L[i, j] = −1 if there is an edge between nodes i and j, and otherwise L[i, j] = 0.

Kirchhoff’s theorem provides a way to calculate the number of spanning trees for a given graph as a determinant of a special matrix. Consider the following graph,



All possible spanning trees are as follows:



In order to calculate the number of spanning trees, construct a Laplacian matrix L, where L[i, i] is the degree of node i and L[i, j] = −1 if there is an edge between nodes i and j, and otherwise L[i, j] = 0.
for the above graph, The Laplacian matrix will look like this

$$L = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}$$

The number of spanning trees equals the determinant of a matrix.
The Determinant of a matrix that can be obtained when we remove any row and any column from L.
For example, if we remove the first row and column, the result will be,

$$\det(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}) = 3.$$

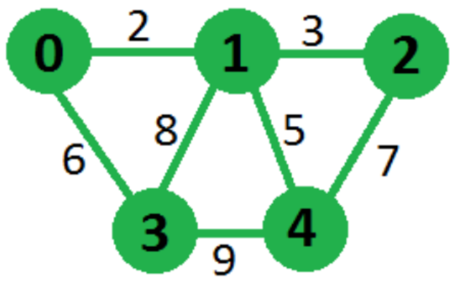
The determinant is always the same, regardless of which row and column we remove from L.

```
import java.util.Arrays;

public class NumberOfSpanningTrees {

    static final int MAX = 100;
    static final int MOD = 1000000007;
    void multiply(long[][] A, long[][] B, long[][] C, int size) {
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                C[i][j] = 0;
                for (int k = 0; k < size; k++) {
                    C[i][j] = (C[i][j] + A[i][k] * B[k][j]) % MOD;
                }
            }
        }
    }
 void power(long[][] A, int N, long[][] result, int size)
      {
      # Write Code Here
      }

 long numOfSpanningTree(int[][] graph, int V)
      {
      # Write Code Here
      }

    public static void main(String[] args) {
        int V = 4; // Number of vertices in graph
        int E = 5; // Number of edges in graph
        int[][] graph = { { 0, 1, 1, 1 }, { 1, 0, 1, 1 }, { 1, 1, 0, 1 }, { 1, 1, 1,
0 } };

        NumberOfSpanningTrees obj = new NumberOfSpanningTrees();
        System.out.println(obj.numOfSpanningTree(graph, V));
    }
}
```

13.4 Minimum Product Spanning Tree

A minimum product spanning tree for a weighted, connected, and undirected graph is a spanning tree with a weight product less than or equal to the weight product of every other spanning tree. The weight product of a spanning tree is the product of weights corresponding to each edge of the spanning tree. All weights of the given graph will be positive for simplicity.

Input:



Output: Minimum Product that we can obtain is 180 for above graph by choosing edges 0-1, 1-2, 0-3 and 1-4

This problem can be solved using standard minimum spanning tree algorithms like Kruskal and prim's algorithm, but we need to modify our graph to use these algorithms. Minimum spanning tree algorithms

tries to minimize the total sum of weights, here we need to minimize the total product of weights. We can use the property of logarithms to overcome this problem.
log(w1* w2 * w3 * .... * wN) = log(w1) + log(w2) + log(w3) ..... + log(wN)
We can replace each weight of the graph by its log value, then we apply any minimum spanning tree algorithm which will try to minimize the sum of log(wi) which in turn minimizes the weight product.

```
import java.util.Arrays;

public class MinimumProductMST {
    // Number of vertices in the graph
    static final int V = 5;

    // A utility function to find the vertex with minimum key value, from the set of
    // vertices not yet included in MST
    int minKey(int key[], boolean mstSet[]) {
        int min = Integer.MAX_VALUE, min_index = -1;

        for (int v = 0; v < V; v++) {
            if (mstSet[v] == false && key[v] < min) {
                min = key[v];
                min_index = v;
            }
        }

        return min_index;
    }

    void printMST(int parent[], int n, int graph[][])
      {
      # Write Code Here
      }
    void primMST(int inputGraph[][], int logGraph[][])
      {
      # Write Code Here
      }
    void minimumProductMST(int graph[][])
      {
      # Write Code Here
      }
    public static void main(String[] args)
      {
      # Write Code Here
    }
}
```

13.5 Reverse Delete Algorithm for Minimum Spanning Tree

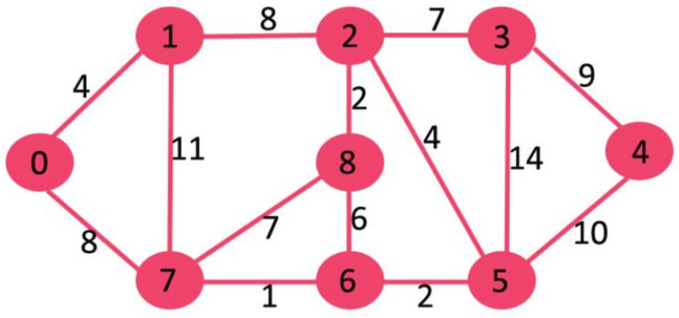
In Reverse Delete algorithm, we sort all edges in decreasing order of their weights. After sorting, we one by one pick edges in decreasing order. We include current picked edge if excluding current edge causes disconnection in current graph. The main idea is delete edge if its deletion does not lead to disconnection of graph.

Algorithm:

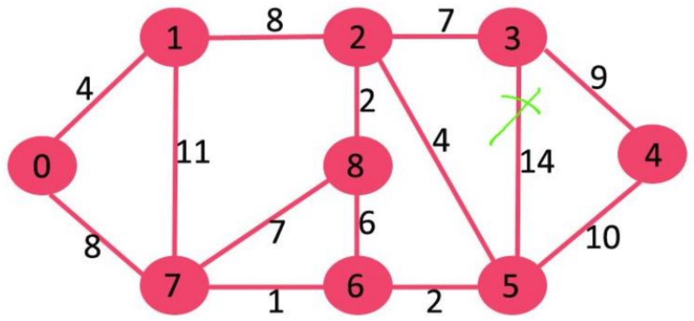
1. Sort all edges of graph in non-increasing order of edge weights.

2. Initialize MST as original graph and remove extra edges using step 3.
3. Pick highest weight edge from remaining edges and check if deleting the edge disconnects the graph or not.
   If disconnects, then we don't delete the edge.
   Else we delete the edge and continue.

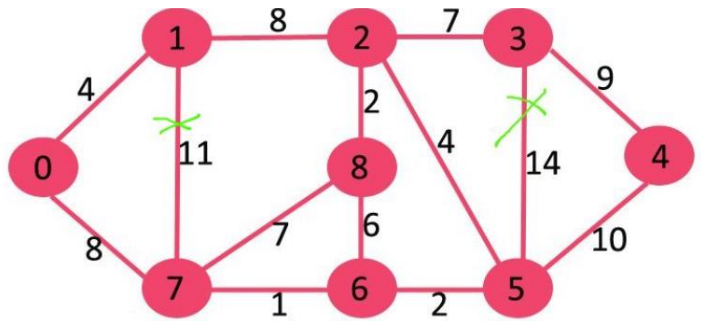
Input: Consider the graph below



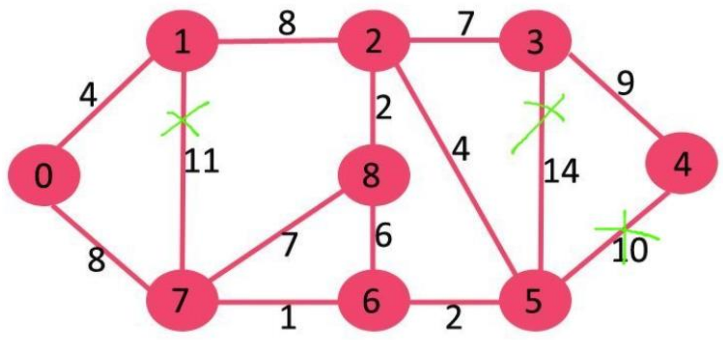
If we delete highest weight edge of weight 14, graph doesn't become disconnected, so we remove it.



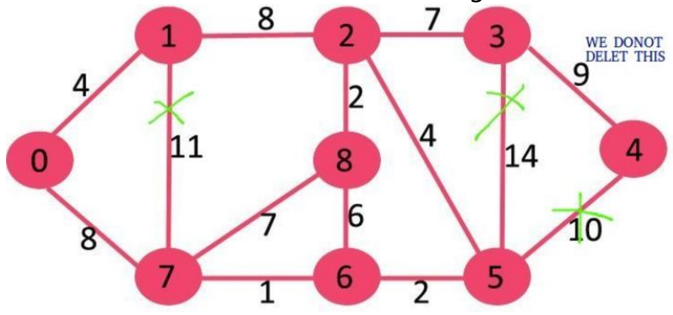
Next we delete 11 as deleting it doesn't disconnect the graph.



Next we delete 10 as deleting it doesn't disconnect the graph.



Next is 9. We cannot delete 9 as deleting it causes disconnection.



We continue this way and following edges remain in final MST.

Edges in MST

(3, 4)
(0, 7)
(2, 3)
(2, 5)
(0, 1)
(5, 6)
(2, 8)
(6, 7)

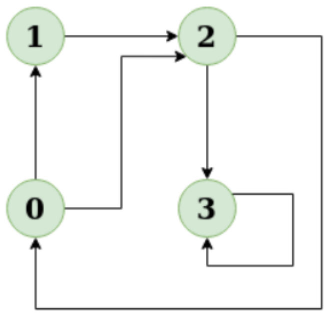
```
import java.util.ArrayList;
import java.util.Collections;

// Edge class to represent edges in the graph
class Edge {
    int src, dest, weight;

    Edge(int src, int dest, int weight) {
        this.src = src;
        this.dest = dest;
        this.weight = weight;
    }
}
class Graph
      {
      # Write Code Here
```

```
       }

    // Function to add an edge to the graph
    void addEdge(int u, int v, int w) {
       # Write Code Here
       }
    void dfs(int v, boolean[] visited)
       {
        # Write Code Here
       }

    // Function to check if the graph is connected
    boolean connected()
       {
       # Write Code Here
       }
void reverseDeleteMST()
       {
        # Write Code Here
       }
}

public class ReverseDeleteMST {
    public static void main(String[] args)
       {
       # Write Code Here
       }
}
```

Try:

1. **Detect Cycle in a Directed Graph:** Given the root of a Directed graph, The task is to check whether the graph contains a cycle or not.

Input: N = 4, E = 6



Output: Yes
Explanation: The diagram clearly shows a cycle 0 -> 2 -> 0

14. Final Notes

The only way to learn programming is program, program and program on challenging problems. The problems in this tutorial are certainly NOT challenging. There are tens of thousands of challenging problems available – used in training for various programming contests (such as International Collegiate Programming Contest (ICPC), International Olympiad in Informatics (IOI)). Check out these sites:

- The ACM - ICPC International collegiate programming contest (https://icpc.global/)
- The Topcoder Open (TCO) annual programming and design contest (https://www.topcoder.com/)
- Universidad de Valladolid's online judge (https://uva.onlinejudge.org/).
- Peking University's online judge (http://poj.org/).
- USA Computing Olympiad (USACO) Training Program @ http://train.usaco.org/usacogate.
- Google's coding competitions (https://codingcompetitions.withgoogle.com/codejam, https://codingcompetitions.withgoogle.com/hashcode)
- The ICFP programming contest (https://www.icfpconference.org/)
- BME International 24-hours programming contest (https://www.challenge24.org/)
- The International Obfuscated C Code Contest (https://www0.us.ioccc.org/main.html)
- Internet Problem Solving Contest (https://ipsc.ksp.sk/)
- Microsoft Imagine Cup (https://imaginecup.microsoft.com/en-us)
- Hewlett Packard Enterprise (HPE) Codewars (https://hpecodewars.org/)
- OpenChallenge (https://www.openchallenge.org/)

Coding Contests Scores

Students must solve problems and attain scores in the following coding contests:

	Name of the contest	**Minimum number of problems to solve**	**Required score**
•	CodeChef	20	200
•	Leetcode	20	200
•	GeeksforGeeks	20	200
•	SPOJ	5	50
•	InterviewBit	10	1000
•	Hackerrank	25	250
•	Codeforces	10	100
•	BuildIT	50	500
		Total score need to obtain	2500

Student must have any one of the following certification:

1. HackerRank - Problem Solving Skills Certification (Basic and Intermediate)
2. GeeksforGeeks – Data Structures and Algorithms Certification
3. CodeChef - Learn Data Structures and Algorithms Certification
4. Interviewbit – DSA pro / Python pro
5. Edx – Data Structures and Algorithms
5. NPTEL – Programming, Data Structures and Algorithms
6. NPTEL – Introduction to Data Structures and Algorithms
7. NPTEL – Data Structures and Algorithms
8. NPTEL – Programming and Data Structure

V. TEXT BOOKS:

1. Rance D. Necaise, “Data Structures and Algorithms using Python”, Wiley Student Edition.
2. Benjamin Baka, David Julian, “Python Data Structures and Algorithms”, Packt Publishers, 2017.

VI. REFERENCE BOOKS:

1. S. Lipschutz, “Data Structures”, Tata McGraw Hill Education, 1st Edition, 2008.
2. D. Samanta, “Classic Data Structures”, PHI Learning, 2nd Edition, 2004.

VII. ELECTRONICS RESOURCES:

1. https://www.tutorialspoint.com/data_structures_algorithms/algorithms_basics.htm
2. https://www.codechef.com/certification/data-structures-and-algorithms/prepare
3. https://www.cs.auckland.ac.nz/software/AlgAnim/dsToC.html
4. https://online-learning.harvard.edu/course/data-structures-and-algorithms

VIII. MATERIALS ONLINE

1. Syllabus
2. Lab manual