**/\*max equilibrium array o(n^2)\*/**

import java.io.\*;

class GFG {

static int findMaxSum(int []arr, int n)

{

int res = Integer.MIN\_VALUE;

for (int i = 0; i < n; i++)

{

int prefix\_sum = arr[i];

for (int j = 0; j < i; j++)

prefix\_sum += arr[j];

int suffix\_sum = arr[i];

for (int j = n - 1; j > i; j--)

suffix\_sum += arr[j];

if (prefix\_sum == suffix\_sum)

res = Math.max(res, prefix\_sum);

}

return res;

}

// Driver Code

public static void main (String[] args)

{

int arr[] = {-2, 5, 3, 1, 2, 6, -4, 2 };

int n = arr.length;

System.out.println(findMaxSum(arr, n));

}

}

**/\*max equilibrium array o(n)\*/**

import java.lang.Math.\*;

import java.util.stream.\*;

class GFG {

// Function to find maximum equilibrium

// sum.

static int findMaxSum(int arr[], int n)

{

int sum = IntStream.of(arr).sum();

int prefix\_sum = 0,

res = Integer.MIN\_VALUE;

for (int i = 0; i < n; i++)

{

prefix\_sum += arr[i];

if (prefix\_sum == sum)

res = Math.max(res, prefix\_sum);

sum -= arr[i];

}

return res;

}

// Driver Code

public static void main(String[] args)

{

int arr[] = { -2, 5, 3, 1,

2, 6, -4, 2 };

int n = arr.length;

System.out.print(findMaxSum(arr, n));

}

}

**/\*leaders with O(n^2)\*/**

int arr[] = new int[]{16, 17, 4, 3, 5, 2};

int size = arr.length;

for (int i = 0; i < size; i++)

{

int j;

for (j = i + 1; j < size; j++)

{

if (arr[i] <=arr[j])

break;

}

if (j == size) // the loop didn't break

System.out.print(arr[i] + " ");

}

**/\*leaders with o(n)\*/**

int arr[] = new int[]{16, 17, 4, 3, 5, 2};

int n = arr.length;

int max\_from\_right=0;

for (int i = size-1; i >= 0; i--)

{

if (max\_from\_right < arr[i])

{

max\_from\_right = arr[i];

System.out.print(max\_from\_right + " ");

}

}

**/\*majority element o(n^2)\*/**

import java.util.Scanner;

public class majorityele {

public static void main(String[] args) {

Scanner s = new Scanner(System.in);

int arr[]={2,3,3,4,4,4,4,4,4,4,4,4,4,4,4,3,3,3,3,3};

int n= arr.length;

int index=0,max=0,count;

for(int i =0;i<n;i++){

count=0;

for(int j=0;j<n;j++){

if(arr[i]==arr[j]){

count++;

}

}

if(count>max){

max=count;

index=i;

}

}

if(max>n/2)

System.out.println("Majority element "+arr[index]);

else

System.out.println("No majority element");

}

}

**/\*majority element o(n)\*/**

import java.io.\*;

import java.util.HashMap;

public class majorityele1 {

public static void main(String[] args) {

HashMap<Integer,Integer> map=new HashMap<>();

int ct=0;

int arr[]={2,4,3,4,4};

int n=arr.length;

for(int i=0;i<n;i++)

map.put(arr[i], 0);

int max=0,index=0;

for(int i=0;i<n;i++){

ct=map.get(arr[i])+1;

map.put(arr[i],ct);

if(ct>max){

max=ct;

index=i;

}

}

if(max>n/2)

System.out.println("Majority element "+arr[index]);

else

System.out.println("No majority element");

}

}

**Majority element Boyer Moore**

import java.io.\*;

class majelt2

{

// Function to find majority element

public static int findMajority(int[] nums)

{

int count = 0, candidate = -1;

// Finding majority candidate

for (int index = 0; index < nums.length; index++) {

if (count == 0) {

candidate = nums[index];

count = 1;

}

else {

if (nums[index] == candidate)

count++;

else

count--;

}

}

count = 0;

for (int index = 0; index < nums.length; index++) {

if (nums[index] == candidate)

count++;

}

if (count > (nums.length / 2))

return candidate;

return -1;

}

public static void main(String[] args)

{

int arr[] = { 1, 1, 1, 1, 2, 3, 4 };

int majority = findMajority(arr);

System.out.println(" The majority element is : "

+ majority);

}

}

**Selection Sort**

import java.io.\*;

public class selectionsort

{

void sort(int arr[])

{

int n = arr.length;

for (int i = 0; i < n-1; i++)

{

int min\_idx = i;

for (int j = i+1; j < n; j++)

if (arr[j] < arr[min\_idx])

min\_idx = j;

int temp = arr[min\_idx];

arr[min\_idx] = arr[i];

arr[i] = temp;

}

}

void printArray(int arr[])

{

int n = arr.length;

for (int i=0; i<n; ++i)

System.out.print(arr[i]+" ");

System.out.println();

}

public static void main(String args[])

{

selectionsort ob = new selectionsort();

int arr[] = {64,25,12,22,11};

ob.sort(arr);

System.out.println("Sorted array");

ob.printArray(arr);

}

}

**Quick Sort**

public class quicksort {

static void Pivotrecursion(int[] arr,int low,int high){

if(low<high){

int pivotp=getpivot(arr,low,high);

Pivotrecursion(arr, pivotp+1, high);

Pivotrecursion(arr, low, pivotp-1);

}

}

static int getpivot(int []arr,int low, int high){

int pivtelt=arr[high];

int pivotp=low;

for(int i=low;i<=high;i++){

if(arr[i]<pivtelt){

int temp=arr[i];

arr[i]=arr[pivotp];

arr[pivotp]=temp;

pivotp++;

}

}

int tmp=arr[pivotp];

arr[pivotp]=arr[high];

arr[high]=tmp;

return pivotp;

}

public static void main(String[] args) {

int[] arr= new int[]{20,81,43,98,82,28,66};

Pivotrecursion(arr,0,arr.length-1);

for(int i=0;i<arr.length;i++)

System.out.print(arr[i]+" ");

}

}

**Sorted Unique Permutation**

**Method 1 : O(N\*N!)**

// Java program to print all the permutation

// of the given String.

//include <algorithm>

//include <String>

import java.util.\*;

class GFG{

// Count of total permutations

static int total = 0;

static void permute(int i, String s)

{

// Base case

if (i == (s.length() - 1))

{

System.out.print(s + "\n");

total++;

return;

}

char prev = '\*';

// Loop from j = 1 to length of String

for(int j = i; j < s.length(); j++)

{

char []temp = s.toCharArray();

if (j > i && temp[i] == temp[j])

continue;

if (prev != '\*' && prev == s.charAt(j))

{

continue;

}

// Swap the elements

temp = swap(temp,i,j);

prev = s.charAt(j);

// Recursion call

permute(i + 1, String.valueOf(temp));

}

}

static char[] swap(char []arr, int i, int j)

{

char temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

return arr;

}

static String sortString(String inputString)

{

// Convert input string to char array

char tempArray[] = inputString.toCharArray();

// Sort tempArray

Arrays.sort(tempArray);

// Return new sorted string

return new String(tempArray);

}

// Driver code

public static void main(String[] args)

{

String s = "abca";

// Sort

s = sortString(s);

// Function call

permute(0, s);

System.out.print("Total distinct permutations = " +

total +"\n");

}

}

**Method 2 : O(N!)**

import java.util.ArrayList;

import java.util.List;

import java.util.Map;

import java.util.TreeMap;

public class StringPermutation {

public List<String> permute(char input[]) {

Map<Character, Integer> countMap = new TreeMap<>();

for (char ch : input) {

countMap.compute(ch, (key, val) -> {

if (val == null) {

return 1;

} else {

return val + 1;

}

});

}

char str[] = new char[countMap.size()];

int count[] = new int[countMap.size()];

int index = 0;

for (Map.Entry<Character, Integer> entry : countMap.entrySet()) {

str[index] = entry.getKey();

count[index] = entry.getValue();

index++;

}

List<String> resultList = new ArrayList<>();

char result[] = new char[input.length];

permuteUtil(str, count, result, 0, resultList);

return resultList;

}

public void permuteUtil(char str[], int count[], char result[], int level, List<String> resultList) {

if (level == result.length) {

resultList.add(new String(result));

return;

}

for(int i = 0; i < str.length; i++) {

if(count[i] == 0) {

continue;

}

result[level] = str[i];

count[i]--;

permuteUtil(str, count, result, level + 1, resultList);

count[i]++;

}

}

private void printArray(char input[]) {

for(char ch : input) {

System.out.print(ch);

}

System.out.println();

}

public static void main(String args[]) {

StringPermutation sp = new StringPermutation();

sp.permute("AABC".toCharArray()).forEach(s -> System.out.println(s));

}

}

**Manuevering**

class manuevering {

static int numberOfPaths(int m, int n){

if (m == 1 || n == 1)

return 1;

return numberOfPaths(m - 1, n)+ numberOfPaths(m, n - 1);

}

public static void main(String args[])

{

System.out.println(numberOfPaths(3, 3));

}

}

**Combinations**

import java.io.\*;

static void combinationUtil(int arr[], int n, int r, int index, int data[], int i)

{

if (index == r)

{

for (int j=0; j<r; j++)

System.out.print(data[j]+" ");

System.out.println("");

return;

}

if (i >= n)

return;

data[index] = arr[i];

combinationUtil(arr, n, r, index+1, data, i+1);

combinationUtil(arr, n, r, index, data, i+1);

}

public static void main (String[] args) {

int arr[] = {1, 2, 3};

int r = 2;

int n = arr.length;

int data[]=new int[r];

combinationUtil(arr, n, r, 0, data, 0);

}

}

**Josephus trap**

import java.io.\*;

class josephustrap {

static int josephus(int n, int k)

{

if (n == 1)

return 1;

else

return (josephus(n - 1, k) + k - 1) % n + 1;

}

public static void main(String[] args)

{

int n = 5;

int k = 2;

System.out.println("The chosen place is " + josephus(n, k));

}

}

**Rate in maze**

public class RatMazeSolving{

static int sol[][], cont=0;

static boolean MazeSolve(int maze[][],int x,int y){

if(x==maze.length-1 && y==maze[0].length-1){

sol[x][y]=1;

return true;

}

if(ispassible(maze,x,y)){

sol[x][y]=1;

if(MazeSolve(maze,x,y+1))

return true;

if(MazeSolve(maze,x+1,y))

return true;

sol[x][y]=0;

}

return false;

}

static boolean ispassible(int maze[][], int x, int y){

cont++;

if(x>=0 && y>=0 && x<maze.length && y<maze[0].length && maze[x][y]==1)

return true;

return false;

}

public static void main(String[] args) {

int maze[][]={ {1, 1, 1, 1, 0},

{0, 0, 0, 1, 1},

{1, 1, 1, 1, 1},

{1, 0, 0, 0, 1},

{1, 1, 1, 1, 1}};

sol= new int[maze.length][maze[0].length];

if(MazeSolve(maze, 0,0))

for (int i=0;i<sol.length;i++){

for (int j=0;j<sol[0].length;j++)

System.out.print(" "+sol[i][j]+" ");

System.out.println();

}

else

System.out.println("Solution is not possible");

}

}

**N Queens**

public class NQueens {

static int N = 4;

static boolean isSafe(int board[][], int row, int col){

int i, j;

for (j = col; j >=0; j--)

if (board[row][j] == 1)

return false;

for (i=row,j=col; i>=0 && j>=0; i--,j--)

if (board[i][j] == 1)

return false;

for (i = row, j = col; j >= 0 && i < N;i++,j--)

if (board[i][j] == 1)

return false;

return true;

}

static boolean solveNQUtil(int board[][], int col){

if (col >= N)

return true;

for (int i = 0; i < N; i++) {

if (isSafe(board, i, col)) {

board[i][col] = 1;

if (solveNQUtil(board, col+1))

return true;

}

board[i][col] = 0;

}

return false;

}

public static void main(String args[]) {

int board[][] = new int[N][N];

if (solveNQUtil(board, 0)){

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++)

System.out.print(" " + board[i][j]+ " ");

System.out.println();

}

}

else

System.out.print("Solution does not exist");

}

}

**Warnsdorff’s Algorithm**

**public class KnightTour {**

**public static void main(String[] args) {**

**int chess\_board\_size = 8;**

**KnightTour knightTour = new KnightTour(chess\_board\_size);**

**knightTour.solveKnightTourProblem();**

**}**

**int BOARD\_SIZE;**

**int[][] visited;**

**int[] xMoves = { 2, 1, -1, -2, -2, -1, 1, 2 };**

**int[] yMoves = { 1, 2, 2, 1, -1, -2, -2, -1 };**

**public KnightTour(int chessBoardSize) {**

**this.BOARD\_SIZE = chessBoardSize;**

**this.visited = new int[BOARD\_SIZE][BOARD\_SIZE];**

**this.initializeBoard();**

**}**

**private void initializeBoard() {**

**for (int i = 0; i < BOARD\_SIZE; i++)**

**for (int j = 0; j < BOARD\_SIZE; j++)**

**this.visited[i][j] = Integer.MIN\_VALUE;**

**}**

**public void printSolution() {**

**for (int i = 0; i < BOARD\_SIZE; i++) {**

**for (int j = 0; j < BOARD\_SIZE; j++) {**

**System.out.print(visited[i][j] + "\t");**

**}**

**System.out.println();**

**}**

**}**

**public void solveKnightTourProblem() {**

**visited[0][0] = 0;**

**// start knight tour from top left corner square (0, 0)**

**if( solveProblem(1, 0, 0)) {**

**printSolution();**

**} else {**

**System.out.println("No feasible solution found...");**

**}**

**}**

**public boolean solveProblem(int moveCount, int x, int y) {**

**// Base Case : We were able to move to each square exactly once**

**if (moveCount == BOARD\_SIZE \* BOARD\_SIZE) {**

**return true;**

**}**

**for (int i = 0; i < xMoves.length; ++i) {**

**int nextX = x + xMoves[i];**

**int nextY = y + yMoves[i];**

**// check if new position is a valid and not visited yet**

**if ( isValidMove(nextX, nextY) && visited[nextX][nextY] == Integer.MIN\_VALUE) {**

**visited[nextX][nextY] = moveCount;**

**if ( solveProblem(moveCount + 1, nextX, nextY) ) {**

**return true;**

**}**

**// BACKTRACK !!!**

**visited[nextX][nextY] = Integer.MIN\_VALUE;**

**}**

**}**

**return false;**

**}**

**public boolean isValidMove(int x, int y) {**

**if (x < 0 || x >= BOARD\_SIZE || y < 0 || y >= BOARD\_SIZE) {**

**return false;**

**} else {**

**return true;**

**}**

**}**

**}**

**Hamiltonian Cycle**

class HamiltonianCycle

{

final int V = 5;

int path[];

boolean isSafe(int v, int graph[][], int path[], int pos)

{

if (graph[path[pos - 1]][v] == 0)

return false;

for (int i = 0; i < pos; i++)

if (path[i] == v)

return false;

return true;

}

boolean hamCycleUtil(int graph[][], int path[], int pos)

{

if (pos == V)

{

if (graph[path[pos - 1]][path[0]] == 1)

return true;

else

return false;

}

for (int v = 1; v < V; v++)

{

if (isSafe(v, graph, path, pos))

{

path[pos] = v;

if (hamCycleUtil(graph, path, pos + 1) == true)

return true;

path[pos] = -1;

}

}

return false;

}

int hamCycle(int graph[][])

{

path = new int[V];

for (int i = 0; i < V; i++)

path[i] = -1;

path[0] = 0;

if (hamCycleUtil(graph, path, 1) == false)

{

System.out.println("\nSolution does not exist");

return 0;

}

printSolution(path);

return 1;

}

void printSolution(int path[])

{

System.out.println("Solution Exists: Following" +

" is one Hamiltonian Cycle");

for (int i = 0; i < V; i++)

System.out.print(" " + path[i] + " ");

System.out.println(" " + path[0] + " ");

}

// driver program to test above function

public static void main(String args[])

{

HamiltonianCycle hamiltonian =

new HamiltonianCycle();

int graph1[][] = {{0, 1, 0, 1, 0},

{1, 0, 1, 1, 1},

{0, 1, 0, 0, 1},

{1, 1, 0, 0, 1},

{0, 1, 1, 1, 0},

};

hamiltonian.hamCycle(graph1);

int graph2[][] = {{0, 1, 0, 1, 0},

{1, 0, 1, 1, 1},

{0, 1, 0, 0, 1},

{1, 1, 0, 0, 0},

{0, 1, 1, 0, 0},

};

hamiltonian.hamCycle(graph2);

}

}

**Kruskal’s algorithm**

Kruskal’s algorithm is the concept that is introduced in the graph theory of discrete mathematics. It is used to discover the shortest path between two points in a connected weighted graph. This [algorithm](https://www.simplilearn.com/data-structures-and-algorithms-article) converts a given graph into the forest, considering each node as a separate tree. These trees can only link to each other if the edge connecting them has a low value and doesn’t generate a cycle in MST structure. In this tutorial, you will learn more about Kruskal Algorithm in detail.

## Introduction to Kruskal Algorithm

As mentioned earlier, the Kruskal algorithm is used to generate a minimum [spanning tree](https://www.simplilearn.com/tutorials/data-structure-tutorial/spanning-tree-in-data-structure) for a given graph. But, what exactly is a minimum spanning tree? A minimum spanning tree is a subset of a graph with the same number of vertices as the graph and edges equal to the number of vertices -1. It also has a minimal cost for the sum of all edge weights in a spanning tree.

Kruskal’s algorithm sorts all the edges in increasing order of their edge weights and keeps adding nodes to the tree only if the chosen edge does not form any cycle. Also, it picks the edge with a minimum cost at first and the edge with a maximum cost at last. Hence, you can say that the Kruskal algorithm makes a locally optimal choice, intending to find the global optimal solution. That is why it is called a [Greedy Algorithm.](https://www.simplilearn.com/tutorials/data-structure-tutorial/greedy-algorithm)

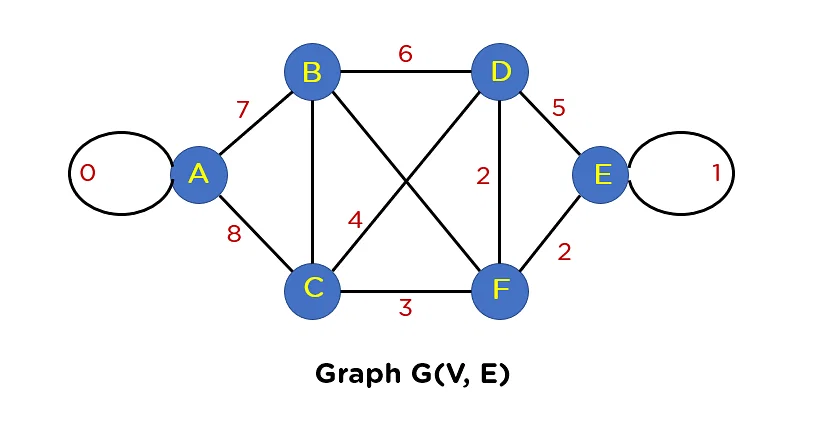
## Creating Minimum Spanning Tree Using Kruskal Algorithm

You will first look into the steps involved in Kruskal’s Algorithm to generate a minimum spanning tree:

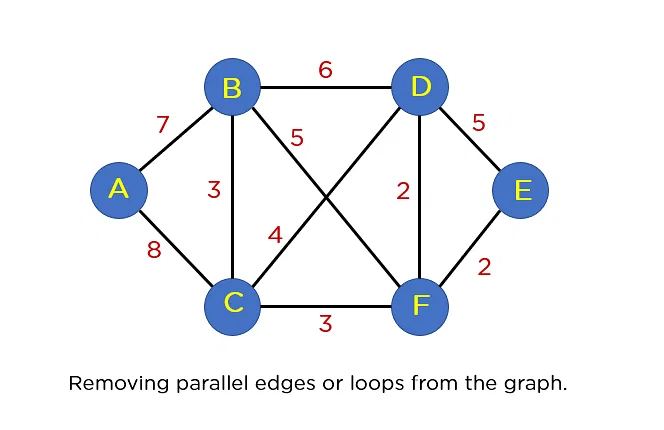
* Step 1: Sort all edges in increasing order of their edge weights.
* Step 2: Pick the smallest edge.
* Step 3: Check if the new edge creates a cycle or loop in a spanning tree.
* Step 4: If it doesn’t form the cycle, then include that edge in MST. Otherwise, discard it.
* Step 5: Repeat from step 2 until it includes |V| - 1 edges in MST.

Using the steps mentioned above, you will generate a minimum spanning tree structure. So, now have a look at an example to understand this process better.

The graph G(V, E) given below contains 6 vertices and 12 edges. And you will create a minimum spanning tree T(V’, E’) for G(V, E) such that the number of vertices in T will be 6 and edges will be 5 (6-1).



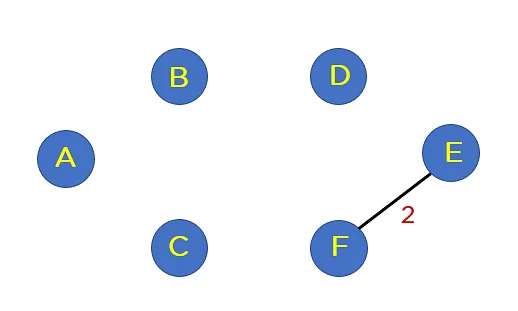
If you observe this graph, you’ll find two looping edges connecting the same node to itself again. And you know that the tree structure can never include a loop or parallel edge. Hence, primarily you will need to remove these edges from the graph structure.



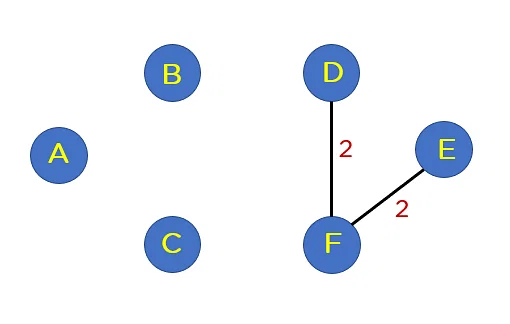
The next step that you will proceed with is arranging all edges in a sorted list by their edge weights.

| **The Edges of the Graph** | **Edge Weight** |  |
| --- | --- | --- |
| **Source Vertex** | **Destination Vertex** |  |
| E | F | 2 |
| F | D | 2 |
| B | C | 3 |
| C | F | 3 |
| C | D | 4 |
| B | F | 5 |
| B | D | 6 |
| A | B | 7 |
| A | C | 8 |

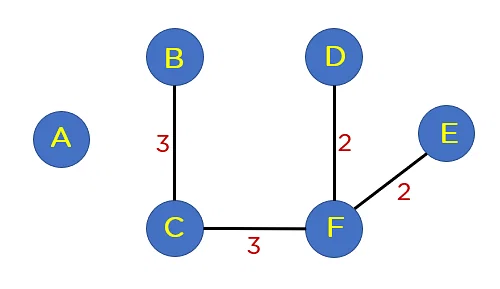
After this step, you will include edges in the MST such that the included edge would not form a cycle in your tree structure. The first edge that you will pick is edge EF, as it has a minimum edge weight that is 2.

****

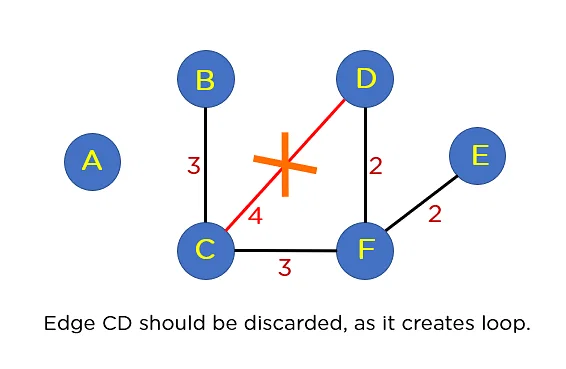
Add edge FD to the spanning tree.



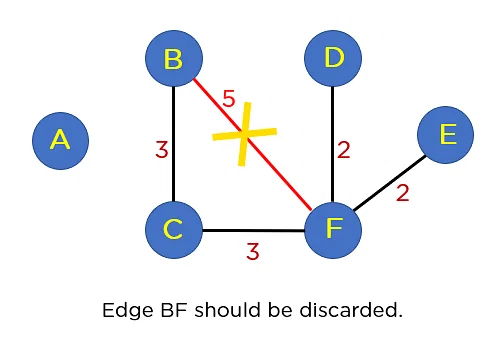
Add edge BC and edge CF to the spanning tree as it does not generate any loop.



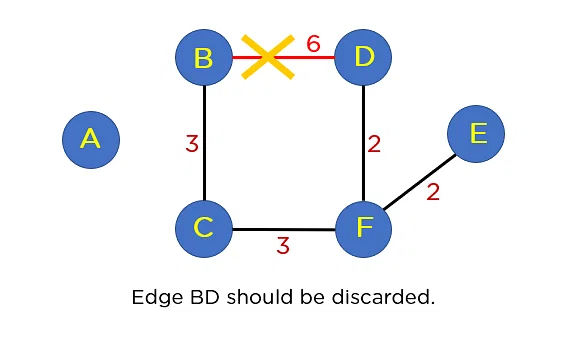
Next up is edge CD. This edge generates the loop in Your tree structure. Thus, you will discard this edge.



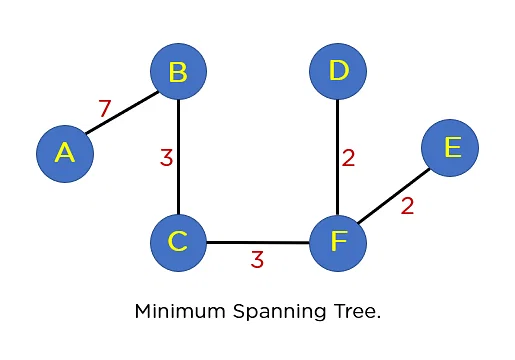
Following edge CD, you have edge BF. This edge also creates the loop; hence you will discard it.



Next up is edge BD. This edge also formulates a loop, so you will discard it as well.



Next on your sorted list is edge AB. This edge does not generate any cycle, so you need not include it in the MST structure. By including this node, it will include 5 edges in the MST, so you don’t have to traverse any further in the sorted list. The final structure of your MST is represented in the image below:



The summation of all the edge weights in MST T(V’, E’) is equal to 17, which is the least possible edge weight for any possible spanning tree structure for this particular graph. Moving ahead, you will learn about implementing Kruskal algorithms using the Union Find Algorithm.

**Activity Selection Problem**

class ActivitySelection {

public static void printMaxActivities(int s[], int f[],int n){

int i, j;

System.out.println("Following activities are selected");

i = 0;

System.out.print(i + " ");

for (j = 1; j < n; j++) {

if (s[j] >= f[i]) {

System.out.print(j + " ");

i = j;

}

}

}

public static void main(String[] args)

{

int s[] = { 1, 3, 0, 5, 8, 5 };

int f[] = { 2, 4, 6, 7, 9, 9 };

int n = s.length;

printMaxActivities(s, f, n);

}

}

**Graph Coloring**

**M coloring decision making problem**

import java.util.\*;

public class graphcoloring{

static ArrayList < ArrayList < Integer >> adj = new ArrayList < > ();

public static void addEdge(int a, int b){

adj.get(a).add(b);

adj.get(b).add(a);

}

static boolean graphColoring(ArrayList < ArrayList < Integer >> adj, int[]color, int m, int v, int V)

{

if (v == V)

return true;

for (int c=1; c <=m; c++)

{

if (isValid(adj, color, c, v))

{

color[v] = c;

if (graphColoring(adj, color, m, v+1, V))

return true;

color[v] = 0;

}

}

return false;

}

static boolean isValid(ArrayList < ArrayList < Integer >> adj, int[]color, int c, int v)

{

for (int u: adj.get(v))

{

if (color[u] == c)

return false;

}

return true;

}

public static void main(String[] args) {

int V = 10;

for (int i = 0; i < V; i++) {

adj.add(new ArrayList < > ());

}

addEdge(0, 1);

addEdge(0, 4);

addEdge(0, 5);

addEdge(1, 2);

addEdge(1, 6);

addEdge(2, 3);

addEdge(2, 8);

addEdge(3, 4);

addEdge(3, 9);

addEdge( 4, 7);

addEdge(5, 8);

addEdge( 5, 9);

addEdge(6, 7);

addEdge(6, 9);

addEdge(7, 8);

System.out.println(adj);

int[] color = new int [V];

for(int i =0;i<V;i++ )

color[i]=0;

int m = 1;

System.out.println( "Is coloring possible using 1 color: " + graphColoring(adj, color, m, 0, V));

m = 2;

System.out.println("Is coloring possible using 2 colors: " + graphColoring(adj, color, m, 0, V));

m = 3;

System.out.println( "Is coloring possible using 3 colors: " + graphColoring(adj, color, m, 0, V));

}

{

}

}

**M Color permutation program**

import java.util.\*;

public class graphcoloring{

static ArrayList < ArrayList < Integer >> adj = new ArrayList < > ();

public static void addEdge(int a, int b){

adj.get(a).add(b);

adj.get(b).add(a);

}

static void graphColoring(ArrayList < ArrayList < Integer >> adj, int[]color, int m, int v, int V)

{

if (v == V){

for (int i=0; i < V; i++)

System.out.print(color[i] +" ");

System.out.println();

return;

}

for (int c=1; c <=m; c++)

{

if (isValid(adj, color, c, v))

{

color[v] = c;

graphColoring(adj, color, m, v+1, V);

color[v] = 0;

}

}

}

static boolean isValid(ArrayList < ArrayList < Integer >> adj, int[]color, int c, int v)

{

for (int u: adj.get(v))

{

if (color[u] == c)

return false;

}

return true;

}

public static void main(String[] args) {

int V = 10;

for (int i = 0; i < V; i++) {

adj.add(new ArrayList < > ());

}

addEdge(0, 1);

addEdge(0, 4);

addEdge(0, 5);

addEdge(1, 2);

addEdge(1, 6);

addEdge(2, 3);

addEdge(2, 8);

addEdge(3, 4);

addEdge(3, 9);

addEdge( 4, 7);

addEdge(5, 8);

addEdge( 5, 9);

addEdge(6, 7);

addEdge(6, 9);

addEdge(7, 8);

int[] color = new int [V];

for(int i =0;i<V;i++ )

color[i]=0;

int m = 3;

System.out.println("Valid permutations using 2 color: ");

graphColoring(adj, color, m, 0, V);

}

{

}

}

**M Coloring Optimization program**

**import java.util.ArrayList;**

**import java.util.Arrays;**

**public class graphcoloring {**

**public static void graphColoring(ArrayList<Integer>[] graph, int V) {**

**int[] color = new int[V];**

**Arrays.fill(color, -1);**

**color[0] = 1;**

**// colors already assigned to neighbors**

**boolean[] alreadyAssigned = new boolean[V];**

**for (int u = 1; u < V; u++) {**

**Arrays.fill(alreadyAssigned, false);**

**for (int i = 0; i < graph[u].size(); i++) {**

**int av = graph[u].get(i);**

**if (color[av] != -1)**

**alreadyAssigned[color[av]] = true;**

**}**

**// Find the first available color**

**int c;**

**for (c = 1; c <= V; c++)**

**if (!alreadyAssigned[c])**

**break;**

**color[u] = c; // Assign the found color**

**}**

**for (int u = 0; u < V; u++)**

**System.out.println("Vertex " + u + " ---> Color " + color[u]);**

**System.out.println("Number of colors required: " + Arrays.stream(color).max().getAsInt());**

**}**

**public static void addEdge(ArrayList<Integer>[] graph, int u, int v) {**

**graph[u].add(v);**

**graph[v].add(u);**

**}**

**public static void main(String[] args) {**

**int V = 10;**

**ArrayList<Integer>[] graph = new ArrayList[V];**

**for (int i = 0; i < V; i++)**

**graph[i] = new ArrayList<Integer>();**

**addEdge(graph, 0, 1);**

**addEdge(graph, 0, 4);**

**addEdge(graph, 0, 5);**

**addEdge(graph, 1, 2);**

**addEdge(graph, 1, 6);**

**addEdge(graph, 2, 3);**

**addEdge(graph, 2, 8);**

**addEdge(graph, 3, 4);**

**addEdge(graph, 3, 9);**

**addEdge(graph, 4, 7);**

**addEdge(graph, 5, 8);**

**addEdge(graph, 5, 9);**

**addEdge(graph, 6, 7);**

**addEdge(graph, 6, 9);**

**addEdge(graph, 7, 8);**

**graphColoring(graph, V);**

**System.out.println();**

**}**

**}**

**Huffman Coding**

import java.util.Comparator;

import java.util.PriorityQueue;

import java.util.Scanner;

class Huffman {

public static void printCode(HuffmanNode root, String s)

{

if (root.left == null && root.right == null

&& Character.isLetter(root.c)) {

System.out.println(root.c + ":" + s);

return;

}

printCode(root.left, s + "0");

printCode(root.right, s + "1");

}

public static void main(String[] args)

{

Scanner s = new Scanner(System.in);

int n = 6;

char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f' };

int[] charfreq = { 5, 9, 12, 13, 16, 45 };

PriorityQueue<HuffmanNode> q

= new PriorityQueue<HuffmanNode>(

n, new MyComparator());

for (int i = 0; i < n; i++) {

HuffmanNode hn = new HuffmanNode();

hn.c = charArray[i];

hn.data = charfreq[i];

hn.left = null;

hn.right = null;

q.add(hn);

}

HuffmanNode root = null;

while (q.size() > 1) {

HuffmanNode x = q.peek();

q.poll();

HuffmanNode y = q.peek();

q.poll();

HuffmanNode f = new HuffmanNode();

f.data = x.data + y.data;

f.c = '-';

f.left = x;

f.right = y;

root = f;

q.add(f);

}

printCode(root, "");

}

}

class HuffmanNode {

int data;

char c;

HuffmanNode left;

HuffmanNode right;

}

class MyComparator implements Comparator<HuffmanNode> {

public int compare(HuffmanNode x, HuffmanNode y)

{

return x.data - y.data;

}

}

**Detect Loop in linked list**

import java.util.\*;

class dloop {

// Link list node

static class Node {

int data;

Node next;

int flag;

};

static Node push(Node head\_ref, int new\_data)

{

Node new\_node = new Node();

new\_node.data = new\_data;

new\_node.flag = 0;

new\_node.next = head\_ref;

head\_ref = new\_node;

return head\_ref;

}

static boolean detectLoop(Node h)

{

while (h != null) {

if (h.flag == 1)

return true;

h.flag = 1;

h = h.next;

}

return false;

}

public static void main(String[] args)

{

Node head = null;

head = push(head, 20);

head = push(head, 4);

head = push(head, 15);

head = push(head, 10);

head.next.next.next.next = head;

if (detectLoop(head))

System.out.print("Loop Found");

else

System.out.print("No Loop");

}

}

**Bitonic DLL**

class bitonicdll

{

// structure of node of the doubly linked list

static class Node

{

int data;

Node next;

Node prev;

};

// function to sort a biotonic doubly linked list

static Node sort(Node head)

{

// If number of elements are less than or

// equal to 1 then return.

if (head == null || head.next == null)

{

return head;

}

// Pointer to first element of doubly

// linked list.

Node front = head;

// Pointer to last element of doubly

// linked list.

Node last = head;

// Dummy node to which resultant

// sorted list is added.

Node res = new Node();

// Node after which next element

// of sorted list is added.

Node resEnd = res;

// Node to store next element to

// which pointer is moved after

// element pointed by that pointer

// is added to result list.

Node next;

// Find last element of input list.

while (last.next != null)

{

last = last.next;

}

// Compare first and last element

// until both pointers are not equal.

while (front != last)

{

if (last.data <= front.data)

{

resEnd.next = last;

next = last.prev;

last.prev.next = null;

last.prev = resEnd;

last = next;

resEnd = resEnd.next;

}

// If front element is smaller, then

// add it to result list and change

// front pointer to its next element.

else

{

resEnd.next = front;

next = front.next;

front.next = null;

front.prev = resEnd;

front = next;

resEnd = resEnd.next;

}

}

// Add the single element left to the

// result list.

resEnd.next = front;

front.prev = resEnd;

// The head of required sorted list is

// next to dummy node res.

return res.next;

}

// Function to insert a node at the beginning

// of the Doubly Linked List

static Node push(Node head\_ref, int new\_data)

{

// allocate node

Node new\_node = new Node();

// put in the data

new\_node.data = new\_data;

// since we are adding at the beginning,

// prev is always null

new\_node.prev = null;

// link the old list off the new node

new\_node.next = (head\_ref);

// change prev of head node to new node

if ((head\_ref) != null)

(head\_ref).prev = new\_node;

// move the head to point to the new node

(head\_ref) = new\_node;

return head\_ref;

}

// Function to print nodes in a given doubly

// linked list

static void printList(Node head)

{

// if list is empty

if (head == null)

System.out.print( "Doubly Linked list empty");

while (head != null)

{

System.out.print( head.data + " ");

head = head.next;

}

}

// Driver code

public static void main(String args[])

{

Node head = null;

// Create the doubly linked list:

// 2<.5<.7<.12<.10<.6<.4<.1

head = push(head, 1);

head = push(head, 4);

head = push(head, 6);

head = push(head, 10);

head = push(head, 12);

head = push(head, 7);

head = push(head, 5);

head = push(head, 2);

System.out.print("Original Doubly linked list:\n");

printList(head);

// sort the biotonic DLL

head = sort(head);

System.out.print("\nDoubly linked list after sorting:\n");

printList(head);

}

}

**Segregate even and odd elements in the list**

public class Segregate

{

Node head;

class Node

{

int data;

Node next;

Node(int d)

{

data = d;

next = null;

}

}

public void segregateEvenOdd()

{

Node evenStart = null;

Node evenEnd = null;

Node oddStart = null;

Node oddEnd = null;

Node currentNode = head;

while(currentNode != null) {

int element = currentNode.data;

if(element % 2 == 0) {

if(evenStart == null) {

evenStart = currentNode;

evenEnd = evenStart;

} else {

evenEnd.next = currentNode;

evenEnd = evenEnd.next;

}

} else {

if(oddStart == null) {

oddStart = currentNode;

oddEnd = oddStart;

} else {

oddEnd.next = currentNode;

oddEnd = oddEnd.next;

}

}

currentNode = currentNode.next;

}

if(oddStart == null || evenStart == null) {

return;

}

evenEnd.next = oddStart;

oddEnd.next = null;

head=evenStart;

}

void push(int new\_data)

{

Node new\_node = new Node(new\_data);

new\_node.next = head;

head = new\_node;

}

void printList()

{

Node temp = head;

while(temp != null)

{

System.out.print(temp.data+" ");

temp = temp.next;

}

System.out.println();

}

public static void main(String args[])

{

Seggregate llist = new Seggregate();

llist.push(3);

llist.push(2);

llist.push(1);

llist.push(4);

llist.push(5);

System.out.println("Original Linked List");

llist.printList();

llist.segregateEvenOdd();

System.out.println("Modified Linked List");

llist.printList();

}

}

**Merge sort**

class mergesortdll {

static class Node{

int data;

Node prev;

Node next;

public Node(int data) {

this.data = data;

}

}

static Node head,tail = null;

public static Node addNode(int item) {

Node newNode = new Node(item);

if(head == null) {

head = tail = newNode;

head.prev = null;

tail.next = null;

}

else {

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

tail.next = null;

}

return head;

}

static void print(Node node) {

Node temp = node;

System.out.println("Forward Traversal using next pointer");

while (node != null) {

System.out.print(node.data + " ");

temp = node;

node = node.next;

}

System.out.println("\nBackward Traversal using prev pointer");

while (temp != null) {

System.out.print(temp.data + " ");

temp = temp.prev;

}

}

// Split a doubly linked list (DLL) into 2 DLLs of

// half sizes

static Node split(Node head) {

Node fast = head, slow = head;

while (fast.next != null && fast.next.next != null) {

fast = fast.next.next;

slow = slow.next;

}

Node temp = slow.next;

slow.next.prev=null;

slow.next = null;

return temp;

}

static Node mergeSort(Node node) {

if (node == null || node.next == null) {

return node;

}

Node second = split(node);

// Recur for left and right halves

node = mergeSort(node);

second = mergeSort(second);

// Merge the two sorted halves

return merge(node, second);

}

// Function to merge two linked lists

static Node merge(Node first, Node second) {

// If first linked list is empty

if (first == null) {

return second;

}

// If second linked list is empty

if (second == null) {

return first;

}

if (first.data < second.data) {

first.next = merge(first.next, second);

first.next.prev = first;

first.prev = null;

return first;

} else {

second.next = merge(first, second.next);

second.next.prev = second;

second.prev = null;

return second;

}

}

public static void main(String[] args) {

head=addNode(10);

head=addNode(30);

head=addNode(3);

head=addNode(4);

head=addNode(20);

Node node = null;

node = mergeSort(head);

System.out.println("Linked list after sorting :");

print(node);

}

}

**Min stack**

**import java.util.\*;**

**class MyStack {**

**Stack<Integer> s;**

**Integer minEle;**

**// Constructor**

**MyStack() {**

**s = new Stack<Integer>();**

**}**

**// Prints minimum element of MyStack**

**void getMin()**

**{**

**// Get the minimum number in the entire stack**

**if (s.isEmpty())**

**System.out.println("Stack is empty");**

**// variable minEle stores the minimum element**

**// in the stack.**

**else**

**System.out.println("Minimum Element in the "**

**+ " stack is: " + minEle);**

**}**

**// prints top element of MyStack**

**void peek()**

**{**

**if (s.isEmpty()) {**

**System.out.println("Stack is empty ");**

**return;**

**}**

**Integer t = s.peek(); // Top element.**

**System.out.print("Top Most Element is: ");**

**// If t < minEle means minEle stores**

**// value of t.**

**if (t < minEle)**

**System.out.println(minEle);**

**else**

**System.out.println(t);**

**}**

**// Removes the top element from MyStack**

**void pop()**

**{**

**if (s.isEmpty()) {**

**System.out.println("Stack is empty");**

**return;**

**}**

**System.out.print("Top Most Element Removed: ");**

**Integer t = s.pop();**

**// Minimum will change as the minimum element**

**// of the stack is being removed.**

**if (t < minEle) {**

**System.out.println(minEle);**

**minEle = 2 \* minEle - t;**

**}**

**else**

**System.out.println(t);**

**}**

**// Insert new number into MyStack**

**void push(Integer x)**

**{**

**if (s.isEmpty()) {**

**minEle = x;**

**s.push(x);**

**System.out.println("Number Inserted: " + x);**

**return;**

**}**

**// If new number is less than original minEle**

**if (x < minEle) {**

**s.push(2 \* x - minEle);**

**minEle = x;**

**}**

**else**

**s.push(x);**

**System.out.println("Number Inserted: " + x);**

**}**

**};**

**// Driver Code**

**public class minstack {**

**public static void main(String[] args)**

**{**

**MyStack s = new MyStack();**

**// Function calls**

**s.push(3);**

**s.push(5);**

**s.getMin();**

**s.push(2);**

**s.push(1);**

**s.getMin();**

**s.pop();**

**s.getMin();**

**s.pop();**

**s.peek();**

**}**

**}**

**Celebrity program**

**import java.util.\*;**

**public class celebrity {**

**public static int celebritySolution(int[][] mat) {**

**Stack<Integer> stk = new Stack<>();**

**for(int i=0;i<mat.length;i++) {**

**stk.push(i);**

**}**

**while(stk.size() > 1) {**

**int col = stk.pop();**

**int row = stk.pop();**

**if(mat[row][col] == 1) {**

**stk.push(col); //col may or may not be a celeb**

**} else {**

**stk.push(row); //row may or may not be a celeb**

**}**

**}**

**int x = stk.pop();**

**for(int j=0;j<mat.length;j++) {**

**if(j == x)**

**continue;**

**if(mat[x][j] == 1) {**

**return -1;**

**}**

**}**

**for(int i=0;i<mat.length;i++) {**

**if(i == x) continue;**

**if(mat[i][x] == 0) {**

**return -1;**

**}**

**}**

**return x;**

**}**

**public static void main(String[] args) {**

**int[][] matrix = {{0,1,1,0},**

**{0,0,0,0},**

**{0,1,0,0},**

**{1,1,0,0}};**

**int res = celebritySolution(matrix);**

**if(res == -1) {**

**System.out.println("There is no celebrity in the party");**

**} else {**

**System.out.println(res + " is the celebrity in the party");**

**}**

**}**

**}**

**Iterative tower of hanoi**

class iterativetowerofhanoi{

class Stack

{

int capacity;

int top;

int array[];

}

Stack createStack(int capacity)

{

Stack stack = new Stack();

stack.capacity = capacity;

stack.top = -1;

stack.array = new int[capacity];

return stack;

}

boolean isFull(Stack stack)

{

return (stack.top == stack.capacity - 1);

}

boolean isEmpty(Stack stack)

{

return (stack.top == -1);

}

void push(Stack stack, int item)

{

if (isFull(stack))

return;

stack.array[++stack.top] = item;

}

int pop(Stack stack)

{

if (isEmpty(stack))

return Integer.MIN\_VALUE;

int ret;

ret=stack.array[stack.top];

stack.array[stack.top--]=0;

return ret;

}

void moveDisksBetweenTwoPoles(Stack A, Stack B,char s, char d)

{

int pole1TopDisk = pop(A);

int pole2TopDisk = pop(B);

if (pole1TopDisk == Integer.MIN\_VALUE)

{

push(A, pole2TopDisk);

moveDisk(d, s, pole2TopDisk);

}

else if (pole2TopDisk == Integer.MIN\_VALUE)

{

push(B, pole1TopDisk);

moveDisk(s, d, pole1TopDisk);

}

else if (pole1TopDisk > pole2TopDisk)

{

push(A, pole1TopDisk);

push(A, pole2TopDisk);

moveDisk(d, s, pole2TopDisk);

}

else

{

push(B, pole2TopDisk);

push(B, pole1TopDisk);

moveDisk(s, d, pole1TopDisk);

}

}

void moveDisk(char fromPeg, char toPeg, int disk)

{

System.out.println("Move the disk " + disk +" from " + fromPeg +" to " + toPeg);

}

void tohIterative(int num\_of\_disks, Stack src, Stack aux, Stack dest)

{

int i, total\_num\_of\_moves;

char s = 'S', d = 'D', a = 'A';

if (num\_of\_disks % 2 == 0)

{

char temp = d;

d = a;

a = temp;

}

total\_num\_of\_moves = (int)(Math.pow(2, num\_of\_disks) - 1);

for(i = num\_of\_disks; i >= 1; i--)

push(src, i);

for(i = 1; i <= total\_num\_of\_moves; i++)

{

if (i % 3 == 1)

moveDisksBetweenTwoPoles(src, dest, s, d);

else if (i % 3 == 2)

moveDisksBetweenTwoPoles(src, aux, s, a);

else if (i % 3 == 0)

moveDisksBetweenTwoPoles(aux, dest, a, d);

}

}

public static void main(String[] args)

{

int num\_of\_disks = 3;

iterativetowerofhanoi ob = new iterativetowerofhanoi();

Stack src, dest, aux;

src = ob.createStack(num\_of\_disks);

dest = ob.createStack(num\_of\_disks);

aux = ob.createStack(num\_of\_disks);

ob.tohIterative(num\_of\_disks, src, aux, dest);

}

}

**Stock Span**

import java.util.\*;

public class stockSpanUsingStack {

static void calculateSpan(int price[], int n, int S[])

{

Stack<Integer> st = new Stack<>();

st.push(0);

S[0] = 1;

for (int i = 1; i < n; i++) {

while (!st.isEmpty() && price[st.peek()] <= price[i])

st.pop();

S[i] = (i - st.peek());

st.push(i);

}

}

static void printArray(int arr[])

{

System.out.print(Arrays.toString(arr));

}

public static void main(String[] args)

{

int price[] = { 10, 4, 5, 90, 120, 80 };

int n = price.length;

int S[] = new int[n];

calculateSpan(price, n, S);

printArray(S);

}

}

**Priority queue using DLL**

import java.util.\*;

class PqUsingDLL {

static Node front, rear;

static class Node {

char info;

int priority;

Node prev, next;

}

static void display() {

Node current = front;

if(isEmpty()) {

System.out.println("List is empty");

return;

}

System.out.println("Nodes of doubly linked list: ");

while(current != null) {

System.out.print("In " + current.info + " pr " + current.priority+" \n");

current = current.next;

}

}

static void push(char n, int p)

{

Node news = new Node();

news.info = n;

news.priority = p;

if (isEmpty()) {

front = news;

rear = news;

news.next = null;

}

else {

if (p <= (front).priority) {

news.next = front;

front.prev = news;

front = news;

}

else if (p > (rear).priority) {

news.next = null;

(rear).next = news;

news.prev = rear;

rear = news;

}

else {

Node start = (front).next;

while (start.priority < p)

start = start.next;

(start.prev).next = news;

news.prev = start.prev;

news.next = start;

start.prev = news;

}

}

}

static char peek() {

return front.info;

}

static boolean isEmpty() {

return (front == null);

}

static char pop()

{

Node temp = front;

char res = temp.info;

(front) = (front).next;

if (isEmpty())

rear = null;

return res;

}

public static void main(String args[])

{

push('b', 3);

push( 'c', 4);

push( 'e', 6);

push( 'f', 7);

push( 'a', 2);

push( 'd', 5);

display();

System.out.println("peek1 "+peek());

System.out.println("poped elt "+pop());

System.out.println("peek2 "+peek());

}

}

**Sort the Queue without using extra space**

import java.util.LinkedList;

import java.util.Queue;

public class QueuSortWithoutExtraSpace

{

// Queue elements after sortIndex are

// already sorted. This function returns

// index of minimum element from front to

// sortIndex

public static int minIndex(Queue<Integer> list,int sortIndex)

{

int min\_index = -1;

int min\_value = Integer.MAX\_VALUE;

int s = list.size();

for (int i = 0; i < s; i++)

{

int current = list.peek();

// This is dequeue() in Java STL

list.poll();

// we add the condition i <= sortIndex

// because we don't want to traverse

// on the sorted part of the queue,

// which is the right part.

if (current <= min\_value && i <= sortIndex)

{

min\_index = i;

min\_value = current;

}

list.add(current);

}

return min\_index;

}

// Moves given minimum element

// to rear of queue

public static void insertMinToRear(Queue<Integer> list,int min\_index)

{

int min\_value = 0;

int s = list.size();

for (int i = 0; i < s; i++)

{

int current = list.peek();

list.poll();

if (i != min\_index)

list.add(current);

else

min\_value = current;

}

list.add(min\_value);

}

public static void sortQueue(Queue<Integer> list)

{

for(int i = 1; i <= list.size(); i++)

{

int min\_index = minIndex(list,list.size() - i);

insertMinToRear(list, min\_index);

}

}

//Driver function

public static void main (String[] args)

{

Queue<Integer> list = new LinkedList<Integer>();

list.add(30);

list.add(11);

list.add(4);

list.add(15);

//Sort Queue

sortQueue(list);

//print sorted Queue

while(list.isEmpty()== false)

{

System.out.print(list.peek() + " ");

list.poll();

}

}

}

**Stack permutation**

import java.util.Queue;

import java.util.LinkedList;

import java.util.Stack;

class demo{

static boolean checkStackPermutation(int ip[], int op[],int n)

{

Queue<Integer> input = new LinkedList<>();

for (int i = 0; i < n; i++)

{

input.add(ip[i]);

}

Queue<Integer> output = new LinkedList<>();

for (int i = 0; i < n; i++)

{

output.add(op[i]);

}

Stack<Integer> st = new Stack<>();

while (!input.isEmpty())

{

st.push(input.poll());

while(!st.isEmpty() && st.peek().equals(output.peek())){

st.pop();

output.poll();

}

}

return (input.isEmpty() && st.isEmpty());

}

public static void main(String[] args)

{

int input[] = { 1, 2, 3 };

int output[] = { 2, 1, 3 };

int n = 3;

if (checkStackPermutation(input, output, n))

System.out.println("Yes");

else

System.out.println("Not Possible");

}

}

**Correcting the Binary Search tree**

class Node {

int data;

Node left, right;

Node(int d) {

data = d;

left = right = null;

}

}

class BSTswap

{

Node first, middle, last, prev;

void correctBSTUtil( Node root)

{

if( root != null )

{

correctBSTUtil( root.left);

if (prev != null && root.data < prev.data)

{

if (first == null)

{

first = prev;

middle = root;

}

else

last = root;

}

prev = root;

correctBSTUtil( root.right);

}

}

void correctBST( Node root )

{

first = middle = last = prev = null;

correctBSTUtil( root );

// Fix (or correct) the tree

if( first != null && last != null )

{

int temp = first.data;

first.data = last.data;

last.data = temp;

}

else if( first != null && middle !=

null )

{

int temp = first.data;

first.data = middle.data;

middle.data = temp;

}

}

void printInorder(Node node)

{

if (node == null)

return;

printInorder(node.left);

System.out.print(" " + node.data);

printInorder(node.right);

}

public static void main (String[] args)

{

Node root = new Node(6);

root.left = new Node(10);

root.right = new Node(2);

root.left.left = new Node(1);

root.left.right = new Node(3);

root.right.right = new Node(12);

root.right.left = new Node(7);

System.out.println("Inorder Traversal"+ " of the original tree");

BSTswap tree = new BSTswap();

tree.printInorder(root);

tree.correctBST(root);

System.out.println("\nInorder Traversal"+ " of the fixed tree");

tree.printInorder(root);

}

}

**Vertical order traversal**

import java.util.\*;

class TreeNode {

int data;

TreeNode left, right;

TreeNode(int data) {

this.data = data;

left = null;

right = null;

}

}

class Tuple {

TreeNode node;

int row;

int col;

public Tuple(TreeNode \_node, int \_row, int \_col) {

node = \_node;

row = \_row;

col = \_col;

}

}

public class verticalorder {

public static List < List < Integer >> findVertical(TreeNode root) {

TreeMap < Integer, TreeMap < Integer, PriorityQueue < Integer >>> map = new TreeMap < > ();

Queue < Tuple > q = new LinkedList < Tuple > ();

q.offer(new Tuple(root, 0, 0));

while (!q.isEmpty()) {

Tuple tuple = q.poll();

TreeNode node = tuple.node;

int x = tuple.row;

int y = tuple.col;

if (!map.containsKey(x)) {

map.put(x, new TreeMap < > ());

}

if (!map.get(x).containsKey(y)) {

map.get(x).put(y, new PriorityQueue < > ());

}

map.get(x).get(y).offer(node.data);

if (node.left != null) {

q.offer(new Tuple(node.left, x - 1, y + 1));

}

if (node.right != null) {

q.offer(new Tuple(node.right, x + 1, y + 1));

}

}

System.out.println(map.values());

List < List < Integer >> list = new ArrayList < > ();

for (TreeMap < Integer, PriorityQueue < Integer >> ys: map.values()) {

list.add(new ArrayList < > ());

for (PriorityQueue < Integer > nodes: ys.values()) {

while (!nodes.isEmpty()) {

list.get(list.size() - 1).add(nodes.poll());

}

}

}

return list;

}

public static void main(String args[]) {

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

root.left.left = new TreeNode(4);

root.left.right = new TreeNode(10);

root.left.left.right = new TreeNode(5);

root.left.left.right.right = new TreeNode(6);

root.right = new TreeNode(3);

root.right.left = new TreeNode(9);

root.right.right = new TreeNode(10);

List < List < Integer >> list = new ArrayList < > ();

list = findVertical(root);

System.out.println("The Vertical Traversal is : ");

for (List < Integer > it: list) {

for (int nodeVal: it) {

System.out.print(nodeVal + " ");

}

System.out.println();

}

}

}

**Left view**

import java.util.\*;

class TreeNode {

int data;

TreeNode left, right;

TreeNode(int data) {

this.data = data;

left = null;

right = null;

}

}

public class leftview {

static void leftView(TreeNode root,List < Integer > list,int level )

{

if(root==null)

return;

if(level==list.size())

list.add(root.data);

if(root.left!=null)

leftView(root.left, list, level+1);

if(root.right!=null)

leftView(root.right, list, level+1);

}

public static void main(String args[]) {

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

root.left.left = new TreeNode(4);

root.left.right = new TreeNode(10);

root.left.left.right = new TreeNode(5);

root.left.left.right.right = new TreeNode(6);

root.right = new TreeNode(3);

root.right.left = new TreeNode(9);

root.right.right = new TreeNode(10);

List < Integer > list = new ArrayList < > ();

leftView(root,list,0);

System.out.println("The left view is : ");

for (int nodeVal: list) {

System.out.print(nodeVal + " ");

}

System.out.println();

}

}

**Right View**

import java.util.\*;

class TUF{

static int knapsack(int[] wt,int[] val, int n, int W){

int dp[][] = new int[n][W+1];

//Base Condition

for(int i=wt[0]; i<=W; i++){

dp[0][i] = val[0];

}

for(int ind =1; ind<n; ind++){

for(int cap=0; cap<=W; cap++){

int notTaken = 0 + dp[ind-1][cap];

int taken = Integer.MIN\_VALUE;

if(wt[ind] <= cap)

taken = val[ind] + dp[ind-1][cap - wt[ind]];

dp[ind][cap] = Math.max(notTaken, taken);

}

}

return dp[n-1][W];

}

public static void main(String args[]) {

int wt[] = {1,2,4,5};

int val[] = {5,4,8,6};

int W=5;

int n = wt.length;

System.out.println("The Maximum value of items, thief can steal is "

+knapsack(wt,val,n,W));

}

}

**Top view**

import java.util.\*;

class TreeNode {

int data;

TreeNode left= null, right= null;

TreeNode(int data) {

this.data = data;

}

}

class Pair{

TreeNode node;

int hd;

Pair(TreeNode node, int hd){

this.node=node;

this.hd=hd;

}

}

public class topview {

static ArrayList<Integer> topView(TreeNode root){

ArrayList<Integer> ans = new ArrayList<>();

if(root == null) return ans;

Map<Integer, Integer> map = new TreeMap<>();

Queue<Pair> q = new LinkedList<Pair>();

q.add(new Pair(root, 0));

while(!q.isEmpty()) {

Pair it = q.remove();

int hd = it.hd;

TreeNode temp = it.node;

if(!map.containsKey(hd)) map.put(hd, temp.data);

if(temp.left != null) {

q.add(new Pair(temp.left, hd - 1));

}

if(temp.right != null)

q.add(new Pair(temp.right, hd + 1));

}

for (Map.Entry<Integer,Integer> entry : map.entrySet())

ans.add(entry.getValue());

return ans;

}

public static void main(String args[]) {

TreeNode root = new TreeNode(8);

root.left = new TreeNode(1);

root.left.right = new TreeNode(2);

root.left.right.right = new TreeNode(3);

root.left.right.right.right = new TreeNode(4);

root.left.right.right.right.right = new TreeNode(5);

root.right = new TreeNode(10);

root.right.left = new TreeNode(9);

root.right.right = new TreeNode(11);

List < Integer > list = new ArrayList < > ();

list = topView(root);

System.out.println("The top view is : ");

for (int nodeVal: list)

System.out.print(nodeVal + " ");

System.out.println();

}

}

**Bottom view**

import java.util.\*;

class TreeNode {

int data;

TreeNode left, right;

TreeNode(int data) {

this.data = data;

left = null;

right = null;

}

}

class Pair{

TreeNode node;

int hd;

Pair(TreeNode node, int hd){

this.node=node;

this.hd=hd;

}

}

public class bottomview {

static ArrayList<Integer> BotView(TreeNode root)

{

ArrayList<Integer> ans = new ArrayList<>();

if(root == null) return ans;

Map<Integer, Integer> map = new TreeMap<>();

Queue<Pair> q = new LinkedList<Pair>();

q.add(new Pair(root, 0));

while(!q.isEmpty()) {

Pair it = q.remove();

int hd = it.hd;

TreeNode temp = it.node;

map.put(hd, temp.data);

if(temp.left != null) {

q.add(new Pair(temp.left, hd - 1));

}

if(temp.right != null) {

q.add(new Pair(temp.right, hd + 1));

}

}

for (Map.Entry<Integer,Integer> entry : map.entrySet()) {

ans.add(entry.getValue());

}

return ans;

}

public static void main(String args[]) {

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

root.left.left = new TreeNode(4);

root.left.right = new TreeNode(10);

root.left.left.right = new TreeNode(5);

root.left.left.right.right = new TreeNode(6);

root.right = new TreeNode(3);

root.right.left = new TreeNode(9);

root.right.right = new TreeNode(10);

List < Integer > list = new ArrayList < > ();

list = BotView(root);

System.out.println("The bottom view is : ");

for (int nodeVal: list) {

System.out.print(nodeVal + " ");

}

System.out.println();

}

}

**Boundary traversal**

import java.util.\*;

class Node {

int data;

Node left, right;

Node(int data) {

this.data = data;

left = null;

right = null;

}

}

public class boundarytraversal{

static Boolean isLeaf(Node root) {

return (root.left == null) && (root.right == null);

}

static void addLeftBoundary(Node root, ArrayList < Integer > res) {

Node cur = root.left;

while (cur != null) {

if (isLeaf(cur) == false) res.add(cur.data);

if (cur.left != null) cur = cur.left;

else cur = cur.right;

}

}

static void addRightBoundary(Node root, ArrayList < Integer > res) {

Node cur = root.right;

ArrayList < Integer > tmp = new ArrayList < Integer > ();

while (cur != null) {

if (isLeaf(cur) == false) tmp.add(cur.data);

if (cur.right != null) cur = cur.right;

else cur = cur.left;

}

int i;

for (i = tmp.size() - 1; i >= 0; --i) {

res.add(tmp.get(i));

}

}

static void addLeaves(Node root, ArrayList < Integer > res) {

if (isLeaf(root)) {

res.add(root.data);

return;

}

if (root.left != null) addLeaves(root.left, res);

if (root.right != null) addLeaves(root.right, res);

}

static ArrayList < Integer > printBoundary(Node node) {

ArrayList < Integer > ans = new ArrayList < Integer > ();

if (isLeaf(node) == false) ans.add(node.data);

addLeftBoundary(node, ans);

addLeaves(node, ans);

addRightBoundary(node, ans);

return ans;

}

public static void main(String args[]) {

Node root = new Node(1);

root.left = new Node(2);

root.left.left = new Node(3);

root.left.left.right = new Node(4);

root.left.left.right.left = new Node(5);

root.left.left.right.right = new Node(6);

root.right = new Node(7);

root.right.right = new Node(8);

root.right.right.left = new Node(9);

root.right.right.left.left = new Node(10);

root.right.right.left.right = new Node(11);

ArrayList < Integer > boundaryTraversal;

boundaryTraversal = printBoundary(root);

System.out.println("The Boundary Traversal is : ");

for (int i = 0; i < boundaryTraversal.size(); i++) {

System.out.print(boundaryTraversal.get(i) + " ");

}

}

}

**BFS of Graph**

import java.util.\*;

class BFS {

public static void bfsOfGraph(int I) {

Queue < Integer > q = new LinkedList <> ();

q.add(I);

vis[I] = true;

while (!q.isEmpty()) {

Integer node = q.poll();

bfs.add(node);

for (Integer it: adj.get(node)) {

if (vis[it] == false) {

vis[it] = true;

q.add(it);

}

}

}

}

static void printAns(ArrayList < Integer > ans) {

for (int i = 0; i < ans.size(); i++) {

System.out.print(ans.get(i) + " ");

}

}

static ArrayList < ArrayList < Integer >> adj = new ArrayList < > ();

static boolean vis[] = new boolean[9];

static ArrayList < Integer > bfs = new ArrayList < > ();

static void addEdge(int a, int b){

adj.get(a).add(b);

adj.get(b).add(a);

}

public static void main(String args[]) {

for (int i = 0; i < 9; i++) {

adj.add(new ArrayList < > ());

}

addEdge(1,2);

addEdge(2,3);

addEdge(2,7);

addEdge(3,5);

addEdge(7,5);

addEdge(4,6);

addEdge(0,5);

addEdge(0, 8);

for(int i=0;i<8;i++){

if(!vis[i])

bfsOfGraph(i);

}

printAns(bfs);

}

}

**DFS of graph**

**import java.util.\*;**

**class DFS {**

**public static void dfsOfGraph(int I) {**

**Stack < Integer > q = new Stack<>();**

**q.add(I);**

**vis[I] = true;**

**while (!q.isEmpty()) {**

**Integer node = q.pop();**

**dfs.add(node);**

**for(Integer it : adj.get(node)){**

**if(!vis[it]){**

**vis[it]=true;**

**q.push(it);**

**}**

**}**

**}**

**}**

**static void printAns(ArrayList < Integer > ans) {**

**for (int i = 0; i < ans.size(); i++) {**

**System.out.print(ans.get(i) + " ");**

**}**

**}**

**static ArrayList < ArrayList < Integer >> adj = new ArrayList < > ();**

**static boolean vis[] = new boolean[9];**

**static ArrayList < Integer > dfs = new ArrayList < > ();**

**static void addEdge(int a, int b){**

**adj.get(a).add(b);**

**adj.get(b).add(a);**

**}**

**public static void main(String args[]) {**

**for (int i = 0; i < 9; i++) {**

**adj.add(new ArrayList < > ());**

**}**

**addEdge(1,2);**

**addEdge(2,3);**

**addEdge(2,7);**

**addEdge(3,5);**

**addEdge(7,5);**

**addEdge(4,6);**

**addEdge(0,5);**

**addEdge(0, 8);**

**System.out.println(adj);**

**for(int i=0;i<8;i++){**

**if(!vis[i])**

**dfsOfGraph(i);**

**}**

**printAns(dfs);**

**}**

**}**

**Dial’s algo**

import java.util.\*;

public class Dial {

static final int INF = Integer.MAX\_VALUE;

private int V;

private ArrayList<ArrayList<Tuple> > adj;

public Dial(int v) // Constructor

{

this.V = v;

this.adj = new ArrayList<ArrayList<Tuple> >();

for (int i = 0; i < v; i++)

this.adj.add(new ArrayList<Tuple>());

}

public void AddEdge(int u, int v, int w)

{

adj.get(u).add(new Tuple(v, w));

adj.get(v).add(new Tuple(u, w));

}

public void shortestPath(int src, int W)

{

int[] dist = new int[V];

Arrays.fill(dist, INF);

ArrayList<Integer>[] B = new ArrayList[W \* V + 1];

for (int i = 0; i < W \* V + 1; i++)

B[i] = new ArrayList<Integer>();

B[0].add(src);

dist[src] = 0;

int idx = 0;

while (true) {

while (B[idx].size() == 0 && idx < W \* V)

idx++;

if (idx == W \* V)

break;

int u = B[idx].get(0);

B[idx].remove(0);

for (Tuple i : adj.get(u)) {

int v = i.v;

int weight = i.w;

int du = dist[u];

int dv = dist[v];

if (dv > du + weight) {

dist[v] = du + weight;

dv = dist[v];

B[dv].add(0, v);

}

}

}

System.out.println("Vertex Distance from Source");

for (int i = 0; i < V; ++i)

System.out.println(i + "\t\t" + dist[i]);

}

static class Tuple {

int v, w;

Tuple(int v, int w)

{

this.v = v;

this.w = w;

}

}

public static void main(String[] args)

{

int V = 9;

Dial g = new Dial(V);

// making above shown graph

g.AddEdge(0, 1, 4);

g.AddEdge(0, 7, 8);

g.AddEdge(1, 2, 8);

g.AddEdge(1, 7, 11);

g.AddEdge(2, 3, 7);

g.AddEdge(2, 8, 2);

g.AddEdge(2, 5, 4);

g.AddEdge(3, 4, 9);

g.AddEdge(3, 5, 14);

g.AddEdge(4, 5, 10);

g.AddEdge(5, 6, 2);

g.AddEdge(6, 7, 1);

g.AddEdge(6, 8, 6);

g.AddEdge(7, 8, 7);

g.shortestPath(0, 14);

}

}

**Bellman ford’s algo**

import java.util.\*;

class Solution {

static int[] bellman\_ford(int V,

ArrayList<ArrayList<Integer>> edges, int S) {

int[] dist = new int[V];

for (int i = 0; i < V; i++) dist[i] = (int)(1e8);

dist[S] = 0;

// V x E

for (int i = 0; i < V - 1; i++) {

for (ArrayList<Integer> it : edges) {

int u = it.get(0);

int v = it.get(1);

int wt = it.get(2);

if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {

dist[v] = dist[u] + wt;

}

}

}

for (ArrayList<Integer> it : edges) {

int u = it.get(0);

int v = it.get(1);

int wt = it.get(2);

if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {

int temp[] = new int[1];

temp[0] = -1;

return temp;

}

}

return dist;

}

}

public class Bellmanford {

public static void main(String[] args) {

int V = 6;

int S = 0;

ArrayList<ArrayList<Integer>> edges = new ArrayList<>() {

{

add(new ArrayList<Integer>(Arrays.asList(3, 2, 6)));

add(new ArrayList<Integer>(Arrays.asList(5, 3, 1)));

add(new ArrayList<Integer>(Arrays.asList(0, 1, 5)));

add(new ArrayList<Integer>(Arrays.asList(1, 5, -3)));

add(new ArrayList<Integer>(Arrays.asList(1, 2, -2)));

add(new ArrayList<Integer>(Arrays.asList(3, 4, -2)));

add(new ArrayList<Integer>(Arrays.asList(2, 4, 3)));

}

};

int[] dist = Solution.bellman\_ford(V, edges, S);

for (int i = 0; i < V; i++) {

System.out.print(dist[i] + " ");

}

System.out.println("");

}

}

**Heap sort**

// Java program for implementation of Heap Sort

public class heapsort {

public static void sort(int arr[])

{

int N = arr.length;

// Build heap (rearrange array)

for (int i = N / 2 - 1; i >= 0; i--)

heapify(arr, N, i);

// One by one extract an element from heap

for (int i = N - 1; i > 0; i--) {

// Move current root to end

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

// call max heapify on the reduced heap

heapify(arr, i, 0);

}

}

// To heapify a subtree rooted with node i which is

// an index in arr[]. n is size of heap

static void heapify(int arr[], int N, int i)

{

int largest = i; // Initialize largest as root

int l = 2 \* i + 1; // left = 2\*i + 1

int r = 2 \* i + 2; // right = 2\*i + 2

// If left child is larger than root

if (l < N && arr[l] > arr[largest])

largest = l;

// If right child is larger than largest so far

if (r < N && arr[r] > arr[largest])

largest = r;

// If largest is not root

if (largest != i) {

int swap = arr[i];

arr[i] = arr[largest];

arr[largest] = swap;

// Recursively heapify the affected sub-tree

heapify(arr, N, largest);

}

}

/\* A utility function to print array of size n \*/

static void printArray(int arr[])

{

int N = arr.length;

for (int i = 0; i < N; ++i)

System.out.print(arr[i] + " ");

System.out.println();

}

// Driver's code

public static void main(String args[])

{

int arr[] = { 12, 11, 13, 5, 6, 7 };

int N = arr.length;

// Function call

sort(arr);

System.out.println("Sorted array is");

printArray(arr);

}

}

**Binomial heap**

he main application of [Binary Heap](https://www.geeksforgeeks.org/binary-heap/) is as implement a priority queue. Binomial Heap is an extension of [Binary Heap](http://geeksquiz.com/binary-heap/) that provides faster union or merge operation with other operations provided by Binary Heap.

*A Binomial Heap is a collection of Binomial Trees*   
**What is a Binomial Tree?**

A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order k-1 and making one the leftmost child of the other.

A Binomial Tree of order k the has following properties.

* It has exactly 2k nodes.
* It has depth as k.
* There are exactly kaiCi nodes at depth i for i = 0, 1, . . . , k.
* The root has degree k and children of the root are themselves Binomial Trees with order k-1, k-2,.. 0 from left to right.

k = 0 (Single Node)

o

k = 1 (2 nodes)

[We take two k = 0 order Binomial Trees, and

make one as a child of other]

o

/

o

k = 2 (4 nodes)

[We take two k = 1 order Binomial Trees, and

make one as a child of other]

o

/ \

o o

/

o

k = 3 (8 nodes)

[We take two k = 2 order Binomial Trees, and

make one as a child of other]

o

/ | \

o o o

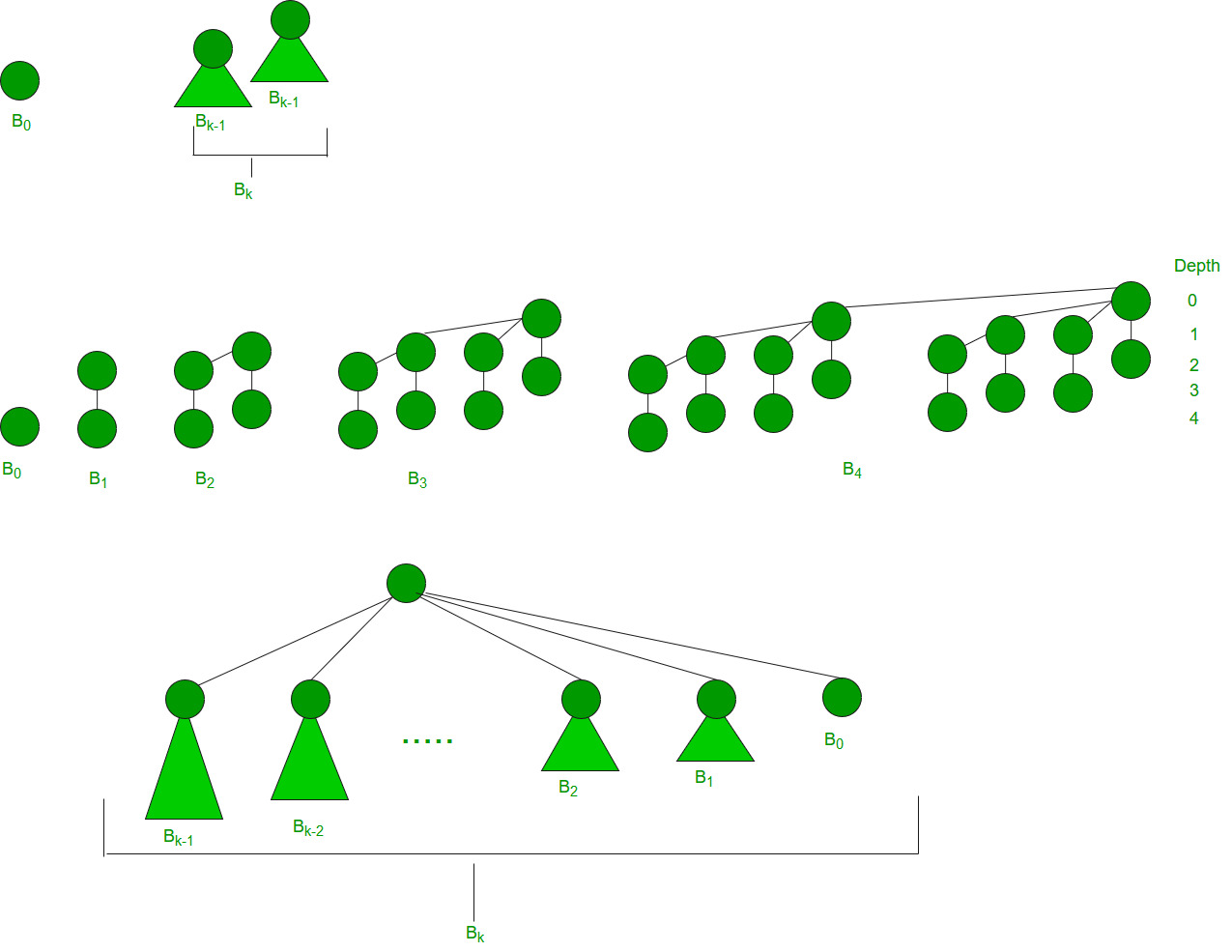
/ \ |

o o o

/

o

The following diagram is referred to from the 2nd Edition of the [CLRS book](http://www.flipkart.com/introduction-algorithms-english-3rd/p/itmdwxyrafdburzg?pid=9788120340077&affid=sandeepgfg).



**Binomial Heap:**

A Binomial Heap is a set of Binomial Trees where each Binomial Tree follows the Min Heap property. And there can be at most one Binomial Tree of any degree.   
**Examples Binomial Heap:**

12------------10--------------------20

/ \ / | \

15 50 70 50 40

| / | |

30 80 85 65

|

100

A Binomial Heap with 13 nodes. It is a collection of 3

Binomial Trees of orders 0, 2, and 3 from left to right.

10--------------------20

/ \ / | \

15 50 70 50 40

| / | |

30 80 85 65

|

100

A Binomial Heap with 12 nodes. It is a collection of 2

Binomial Trees of orders 2 and 3 from left to right.

class BinomialHeapNode {

int key, degree;

BinomialHeapNode parent;

BinomialHeapNode sibling;

BinomialHeapNode child;

public BinomialHeapNode(int k)

{

key = k;

degree = 0;

parent = null;

sibling = null;

child = null;

}

public BinomialHeapNode reverse(BinomialHeapNode sibl)

{

BinomialHeapNode ret;

if (sibling != null)

ret = sibling.reverse(this);

else

ret = this;

sibling = sibl;

return ret;

}

public BinomialHeapNode findMinNode()

{

BinomialHeapNode x = this, y = this;

int min = x.key;

while (x != null) {

if (x.key < min) {

y = x;

min = x.key;

}

x = x.sibling;

}

return y;

}

public BinomialHeapNode findANodeWithKey(int value)

{

BinomialHeapNode temp = this, node = null;

while (temp != null) {

if (temp.key == value) {

node = temp;

break;

}

if (temp.child == null)

temp = temp.sibling;

else {

node = temp.child.findANodeWithKey(value);

if (node == null)

temp = temp.sibling;

else

break;

}

}

return node;

}

public int getSize()

{

return (1 + ((child == null) ? 0 : child.getSize())

+ ((sibling == null) ? 0 : sibling.getSize()));

}

}

class BinomialHeap {

private BinomialHeapNode Nodes;

private int size;

public BinomialHeap()

{

Nodes = null;

size = 0;

}

public boolean isEmpty() { return Nodes == null; }

public int getSize() { return size; }

public void makeEmpty()

{

Nodes = null;

size = 0;

}

public void insert(int value)

{

if (value > 0) {

BinomialHeapNode temp

= new BinomialHeapNode(value);

if (Nodes == null) {

Nodes = temp;

size = 1;

}

else {

unionNodes(temp);

size++;

}

}

}

private void merge(BinomialHeapNode binHeap)

{

BinomialHeapNode temp1 = Nodes, temp2 = binHeap;

while ((temp1 != null) && (temp2 != null)) {

if (temp1.degree == temp2.degree) {

BinomialHeapNode tmp = temp2;

temp2 = temp2.sibling;

tmp.sibling = temp1.sibling;

temp1.sibling = tmp;

temp1 = tmp.sibling;

}

else {

if (temp1.degree < temp2.degree) {

if ((temp1.sibling == null)

|| (temp1.sibling.degree

> temp2.degree)) {

BinomialHeapNode tmp = temp2;

temp2 = temp2.sibling;

tmp.sibling = temp1.sibling;

temp1.sibling = tmp;

temp1 = tmp.sibling;

}

else {

temp1 = temp1.sibling;

}

}

else {

BinomialHeapNode tmp = temp1;

temp1 = temp2;

temp2 = temp2.sibling;

temp1.sibling = tmp;

if (tmp == Nodes) {

Nodes = temp1;

}

else {

}

}

}

}

if (temp1 == null) {

temp1 = Nodes;

while (temp1.sibling != null) {

temp1 = temp1.sibling;

}

temp1.sibling = temp2;

}

else {

}

}

private void unionNodes(BinomialHeapNode binHeap)

{

merge(binHeap);

BinomialHeapNode prevTemp = null, temp = Nodes,

nextTemp = Nodes.sibling;

while (nextTemp != null) {

if ((temp.degree != nextTemp.degree)

|| ((nextTemp.sibling != null)

&& (nextTemp.sibling.degree

== temp.degree))) {

prevTemp = temp;

temp = nextTemp;

}

else {

if (temp.key <= nextTemp.key) {

temp.sibling = nextTemp.sibling;

nextTemp.parent = temp;

nextTemp.sibling = temp.child;

temp.child = nextTemp;

temp.degree++;

}

else {

if (prevTemp == null) {

Nodes = nextTemp;

}

else {

prevTemp.sibling = nextTemp;

}

temp.parent = nextTemp;

temp.sibling = nextTemp.child;

nextTemp.child = temp;

nextTemp.degree++;

temp = nextTemp;

}

}

nextTemp = temp.sibling;

}

}

public int findMinimum()

{

return Nodes.findMinNode().key;

}

public void delete(int value)

{

if ((Nodes != null)

&& (Nodes.findANodeWithKey(value) != null)) {

decreaseKeyValue(value, findMinimum() - 1);

extractMin();

}

}

public void decreaseKeyValue(int old\_value,

int new\_value)

{

BinomialHeapNode temp

= Nodes.findANodeWithKey(old\_value);

if (temp == null)

return;

temp.key = new\_value;

BinomialHeapNode tempParent = temp.parent;

while ((tempParent != null)

&& (temp.key < tempParent.key)) {

int z = temp.key;

temp.key = tempParent.key;

tempParent.key = z;

temp = tempParent;

tempParent = tempParent.parent;

}

}

public int extractMin()

{

if (Nodes == null)

return -1;

BinomialHeapNode temp = Nodes, prevTemp = null;

BinomialHeapNode minNode = Nodes.findMinNode();

while (temp.key != minNode.key) {

prevTemp = temp;

temp = temp.sibling;

}

if (prevTemp == null) {

Nodes = temp.sibling;

}

else {

prevTemp.sibling = temp.sibling;

}

temp = temp.child;

BinomialHeapNode fakeNode = temp;

while (temp != null) {

temp.parent = null;

temp = temp.sibling;

}

if ((Nodes == null) && (fakeNode == null)) {

size = 0;

}

else {

if ((Nodes == null) && (fakeNode != null)) {

Nodes = fakeNode.reverse(null);

size = Nodes.getSize();

}

else {

if ((Nodes != null) && (fakeNode == null)) {

size = Nodes.getSize();

}

else {

unionNodes(fakeNode.reverse(null));

size = Nodes.getSize();

}

}

}

return minNode.key;

}

public void displayHeap()

{

System.out.print("\nHeap : ");

displayHeap(Nodes);

System.out.println("\n");

}

private void displayHeap(BinomialHeapNode r)

{

if (r != null) {

displayHeap(r.child);

System.out.print(r.key + " ");

displayHeap(r.sibling);

}

}

}

public class binomheap {

public static void main(String[] args)

{

BinomialHeap binHeap = new BinomialHeap();

binHeap.insert(12);

binHeap.insert(8);

binHeap.insert(5);

binHeap.insert(15);

binHeap.insert(7);

binHeap.insert(2);

binHeap.insert(9);

System.out.println("Size of the binomial heap is "+ binHeap.getSize());

binHeap.displayHeap();

binHeap.delete(15);

binHeap.delete(8);

System.out.println("Size of the binomial heap is "+ binHeap.getSize());

binHeap.displayHeap();

binHeap.makeEmpty();

System.out.println(binHeap.isEmpty());

}

}

**K - ary heap**

import java.util.ArrayList;

public class k\_aryheap {

private static int n;

private static ArrayList<Integer> arl = new ArrayList<Integer>();

public static int getMax() {

if (isEmpty()) {

return Integer.MIN\_VALUE;

}

return arl.get(0);

}

public static boolean isEmpty() {

return (arl.size() == 0);

}

public static void insert(int value) {

arl.add(value);

int childrenIndex = arl.size() - 1;

int parentIndex = (childrenIndex - 1) / n;

while (parentIndex >= 0 && arl.get(childrenIndex) > arl.get(parentIndex)) {

int temp = arl.get(childrenIndex);

arl.set(childrenIndex, arl.get(parentIndex));

arl.set(parentIndex, temp);

childrenIndex = parentIndex;

parentIndex = (childrenIndex - 1) / n;

}

}

public static void removeMax() {

arl.set(0, arl.get(arl.size() - 1));

arl.remove(arl.size() - 1);

int parentIndex = 0;

while (true) {

int largestValueIndex = parentIndex;

for (int i = n \* parentIndex + 1; i <= (n \* parentIndex + n) && i < arl.size(); i++) {

if (arl.get(largestValueIndex) < arl.get(i)) {

largestValueIndex = i;

}

}

if (largestValueIndex == parentIndex) {

break;

}

else {

int temp = arl.get(parentIndex);

arl.set(parentIndex, arl.get(largestValueIndex));

arl.set(largestValueIndex, temp);

parentIndex = largestValueIndex;

}

}

}

public static void main(String[] args) {

n=3;

ArrayList<Integer> arr = new ArrayList<Integer>();

arr.add(4);

arr.add(5);

arr.add(6);

arr.add(7);

arr.add(8);

arr.add(9);

arr.add(10);

for (int i = 0; i < arr.size(); i++) {

insert(arr.get(i));

}

System.out.println("The maximum Element in the heap is: " + getMax());

removeMax();

System.out.println("The maximum Element in the heap is: " + getMax());

}

}

**HashMap to TreeMap**

import java.util.\*;

class hashmapToTreemap {

public static void main(String args[])

{

Map<Integer, String> hashMap = new HashMap<>();

hashMap.put(1, "Geeks");

hashMap.put(2, "forGeeks");

hashMap.put(3, "A computer Portal");

System.out.println("HashMap: " + hashMap);

Map<Integer, String> treeMap = new TreeMap<>();

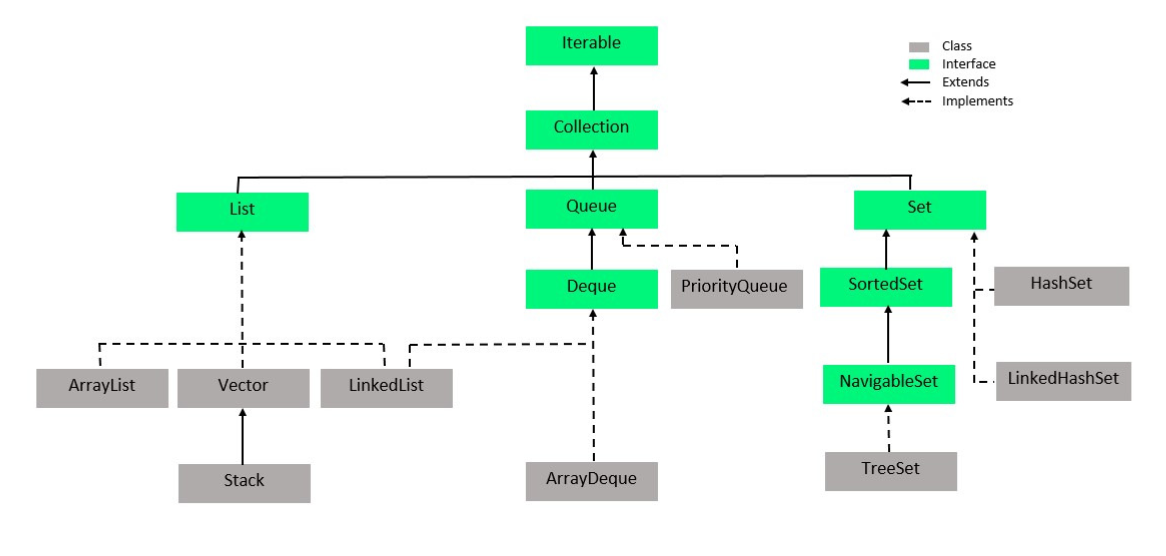
treeMap.putAll(hashMap);

System.out.println("TreeMap: " + treeMap);

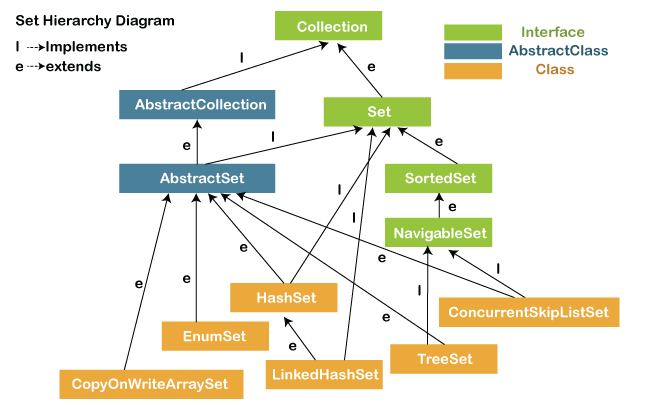
}

}

# Set in Java



The **set** is an interface available in the **java.util** package. The **set** interface extends the Collection interface. An unordered collection or list in which duplicates are not allowed is referred to as a **collection interface**. The set interface is used to create the mathematical set. The set interface use collection interface's methods to avoid the insertion of the same elements. **SortedSet** and **NavigableSet** are two interfaces that extend the set implementation.

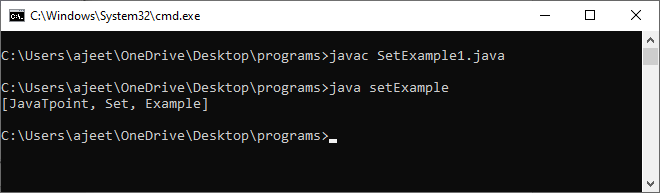


In the above diagram, the **NavigableSet** and **SortedSet** are both the interfaces. The **NavigableSet** extends the SortedSet, so it will not retain the insertion order and store the data in a sorted way.

### SetExample1.java

1. **import** java.util.\*;
2. **public** **class** setExample{
3. **public** **static** **void** main(String[] args)
4. {
5. // creating LinkedHashSet using the Set
6. Set<String> data = **new** LinkedHashSet<String>();
8. data.add("JavaTpoint");
9. data.add("Set");
10. data.add("Example");
11. data.add("Set");
13. System.out.println(data);
14. }
15. }

**Output:**

****

#### Note: Throughout the section, we have compiled the program with file name and run the program with class name. Because the file name and the class name are different.

## Operations on the Set Interface

On the Set, we can perform all the basic mathematical operations like intersection, union and difference.

Suppose, we have two sets, i.e., set1 = [22, 45, 33, 66, 55, 34, 77] and set2 = [33, 2, 83, 45, 3, 12, 55]. We can perform the following operation on the Set:

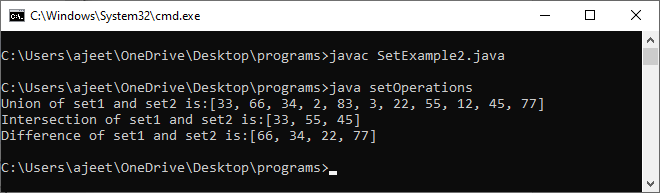
* **Intersection:** The intersection operation returns all those elements which are present in both the set. The intersection of set1 and set2 will be [33, 45, 55].
* **Union:** The union operation returns all the elements of set1 and set2 in a single set, and that set can either be set1 or set2. The union of set1 and set2 will be [2, 3, 12, 22, 33, 34, 45, 55, 66, 77, 83].
* **Difference:** The difference operation deletes the values from the set which are present in another set. The difference of the set1 and set2 will be [66, 34, 22, 77].

In set, **addAll()** method is used to perform the union, **retainAll()** method is used to perform the intersection and **removeAll()** method is used to perform difference. Let's take an example to understand how these methods are used to perform the intersection, union, and difference operations.

### SetExample2.java

1. **import** java.util.\*;
2. **public** **class** SetOperations
3. {
4. **public** **static** **void** main(String args[])
5. {
6. Integer[] A = {22, 45,33, 66, 55, 34, 77};
7. Integer[] B = {33, 2, 83, 45, 3, 12, 55};
8. Set<Integer> set1 = **new** HashSet<Integer>();
9. set1.addAll(Arrays.asList(A));
10. Set<Integer> set2 = **new** HashSet<Integer>();
11. set2.addAll(Arrays.asList(B));
13. // Finding Union of set1 and set2
14. Set<Integer> union\_data = **new** HashSet<Integer>(set1);
15. union\_data.addAll(set2);
16. System.out.print("Union of set1 and set2 is:");
17. System.out.println(union\_data);
19. // Finding Intersection of set1 and set2
20. Set<Integer> intersection\_data = **new** HashSet<Integer>(set1);
21. intersection\_data.retainAll(set2);
22. System.out.print("Intersection of set1 and set2 is:");
23. System.out.println(intersection\_data);
25. // Finding Difference of set1 and set2
26. Set<Integer> difference\_data = **new** HashSet<Integer>(set1);
27. difference\_data.removeAll(set2);
28. System.out.print("Difference of set1 and set2 is:");
29. System.out.println(difference\_data);
30. }
31. }

**Output:**

****

### Description:

In the above code, first, we create two arrays, i.e., A and B of type integer. After that, we create two set, i.e., set1 and set2 of type integer. We convert both the array into a list and add the elements of array A into set1 and elements of array B into set2.

For performing the union, we create a new set **union\_data** with the same element of the set1. We then call the addAll() method of set and pass the set2 as an argument to it. This method will add all those elements to the **union\_data** which are not present in it and gives the union of both sets.

For performing the intersection, we create a new set **intersection\_data** with the same element of the set1. We then call the retainAll() method of set and pass the set2 as an argument to it. This method will get all those elements from the **intersection\_data** which are present in set2 and store it in the intersection\_data. Now, the intersection\_data contains the intersect value of both the sets.

For performing the difference, we create a new set **difference\_data** with the same element of the set1. We then call the removeAll() method of set and pass the set2 as an argument to it. This method will remove all those elements from the **difference\_data** which are present in the set2 and gives the difference of both the sets.

## Set Methods

There are several methods available in the set interface which we can use to perform a certain operation on our sets. These methods are as follows:

### 1) add()

The add() method insert a new value to the set. The method returns true and false depending on the presence of the insertion element. It returns false if the element is already present in the set and returns true if it is not present in the set.

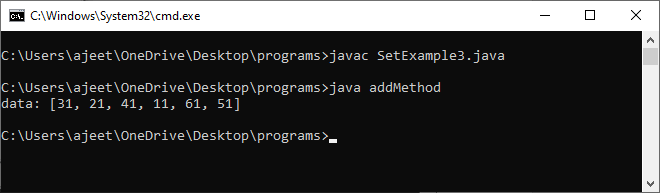
**Syntax:**

1. **boolean** add(type element).

**SetExample3.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **public** **class** addMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(11);
11. data.add(61);
12. data.add(51);
13. System.out.println("data: " + data);
14. }
15. }

**Output:**

****

### 2) addAll()

The addAll() method appends all the elements of the specified collection to the set.

AD

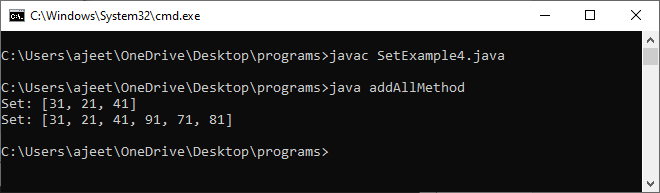
**Syntax:**

1. **boolean** addAll(Collection data)

**SetExample4.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** addAllMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. System.out.println("Set: " + data);
11. ArrayList<Integer> newData = **new** ArrayList<Integer>();
12. newData.add(91);
13. newData.add(71);
14. newData.add(81);
15. data.addAll(newData);
16. System.out.println("Set: " + data);
17. }
18. }

**Output:**

****

### 3) clear()

The method removes all the elements from the set. It doesn't delete the reference of the set. It only deletes the elements of the set.

**Syntax:**

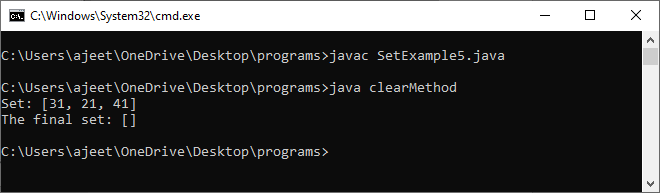
AD

1. **void** clear()

**SetExample5.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **public** **class** clearMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
8. data.add(31);
9. data.add(21);
10. data.add(41);
11. System.out.println("Set: " + data);
13. data.clear();
14. System.out.println("The final set: " + data);
15. }
16. }

**Output:**

****

### 4) contains()

The contains() method is used to know the presence of an element in the set. Its return value is true or false depending on the presence of the element.

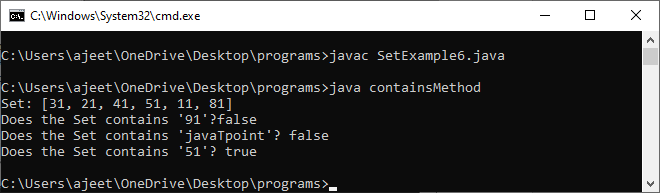
**Syntax:**

1. **boolean** contains(Object element)

**SetExample6.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** containsMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(51);
11. data.add(11);
12. data.add(81);
13. System.out.println("Set: " + data);
14. System.out.println("Does the Set contains '91'?" + data.contains(91));
15. System.out.println("Does the Set contains 'javaTpoint'? " + data.contains("4"));
16. System.out.println("Does the Set contains '51'? " + data.contains(51));
17. }
18. }

**Output:**

****

### 5) containsAll()

The method is used to check whether all the elements of the collection are available in the existing set or not. It returns true if all the elements of the collection are present in the set and returns false even if one of the elements is missing in the existing set.

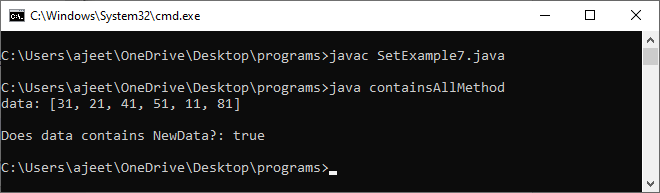
**Syntax:**

1. **public** **boolean** containsAll(Collection data)

**SetExample7.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** containsAllMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(51);
11. data.add(11);
12. data.add(81);
14. System.out.println("data: " + data);
16. Set<Integer> newData = **new** LinkedHashSet<Integer>();
17. newData.add(31);
18. newData.add(21);
19. newData.add(41);
21. System.out.println("\nDoes data contains newData?: "+ data.containsAll(newData));
23. }
24. }

**Output:**

****

### 6) hashCode()

The method is used to derive the hash code value for the current instance of the set. It returns hash code value of integer type.

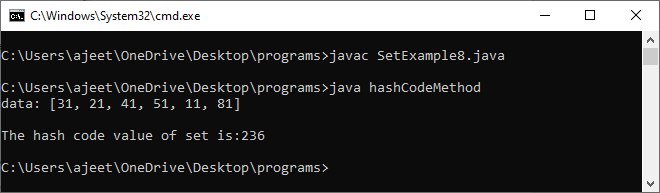
**Syntax:**

1. **public** **int** hashCode()

**SetExample8.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** hashCodeMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(51);
11. data.add(11);
12. data.add(81);
13. System.out.println("data: " + data);
14. System.out.println("\nThe hash code value of set is:"+ data.hashCode());
15. }
16. }

**Output:**

****

### 7) isEmpty()

The isEmpty() method is used to identify the emptiness of the set . It returns true if the set is empty and returns false if the set is not empty.

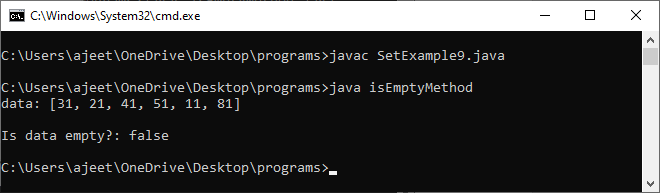
**Syntax:**

1. **boolean** isEmpty()

**SetExample9.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** isEmptyMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(51);
11. data.add(11);
12. data.add(81);
13. System.out.println("data: " + data);
14. System.out.println("\nIs data empty?: "+ data.isEmpty());
15. }
16. }

**Output:**

****

### 8) iterator()

The iterator() method is used to find the iterator of the set. The iterator is used to get the element one by one.

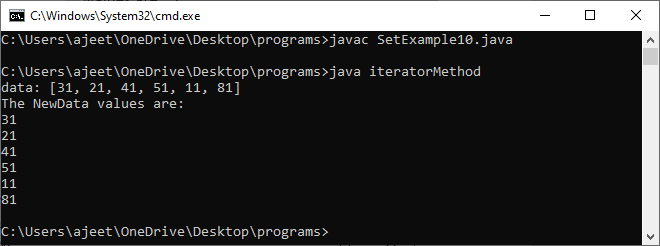
**Syntax:**

1. Iterator iterate\_value = set1.iterator();

**SetExample10.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** iteratorMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(51);
11. data.add(11);
12. data.add(81);
13. System.out.println("data: " + data);
15. Iterator newData = data.iterator();
16. System.out.println("The NewData values are: ");
17. **while** (newData.hasNext()) {
18. System.out.println(newData.next());
19. }
20. }
21. }

**Output:**

****

### 9) remove()

The method is used to remove a specified element from the Set. Its return value depends on the availability of the element. It returns true if the element is available in the set and returns false if it is unavailable in the set.

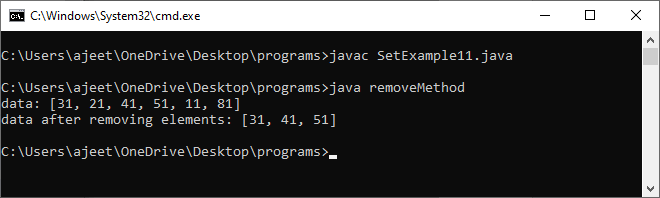
**Syntax:**

1. **boolean** remove(Object O)

**SetExample11.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** removeMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(51);
11. data.add(11);
12. data.add(81);
13. System.out.println("data: " + data);
15. data.remove(81);
16. data.remove(21);
17. data.remove(11);
18. System.out.println("data after removing elements: " + data);
19. }
20. }

**Output:**

****

### 11) removeAll()

The method removes all the elements of the existing set from the specified collection.

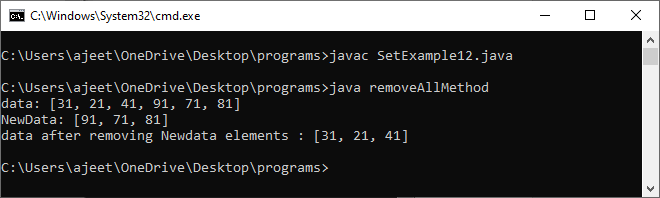
**Syntax:**

1. **public** **boolean** removeAll(Collection data)

**SetExample12.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** removeAllMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(91);
11. data.add(71);
12. data.add(81);
13. System.out.println("data: " + data);
15. ArrayList<Integer> newData = **new** ArrayList<Integer>();
16. newData.add(91);
17. newData.add(71);
18. newData.add(81);
19. System.out.println("NewData: " + newData);
21. data.removeAll(newData);
22. System.out.println("data after removing Newdata elements : " + data);
23. }
24. }

**Output:**

****

### 11) retainAll()

The method retains all the elements from the set specified in the given collection.

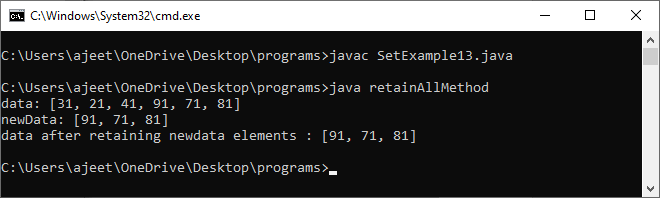
**Syntax:**

1. **public** **boolean** retainAll(Collection data)

**SetExample13.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** retainAllMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(91);
11. data.add(71);
12. data.add(81);
13. System.out.println("data: " + data);
15. ArrayList<Integer> newData = **new** ArrayList<Integer>();
16. newData.add(91);
17. newData.add(71);
18. newData.add(81);
19. System.out.println("newData: " + newData);
21. data.retainAll(newData);
22. System.out.println("data after retaining newdata elements : " + data);
23. }
24. }

**Output:**

****

### 12) size()

The method returns the size of the set.

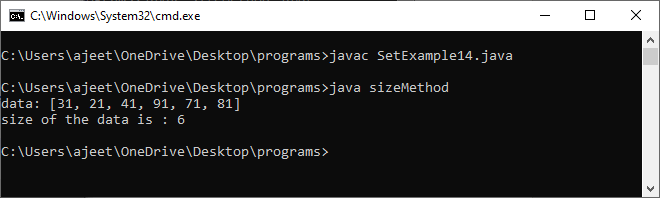
**Syntax:**

1. **int** size()

**SetExample14.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** sizeMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(91);
11. data.add(71);
12. data.add(81);
13. System.out.println("data: " + data);
15. System.out.println("size of the data is : " + data.size());
16. }
17. }

**Output:**

****

### 13) removeAll()

The method is used to create an array with the same elements of the set.

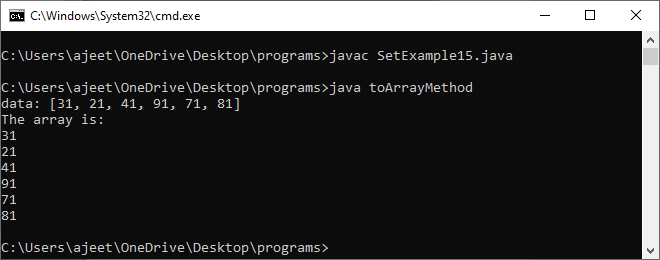
**Syntax:**

1. Object[] toArray()

**SetExample15.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** toArrayMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(91);
11. data.add(71);
12. data.add(81);
13. System.out.println("data: " + data);
15. Object[] array\_data = data.toArray();
16. System.out.println("The array is:");
17. **for** (**int** i = 0; i < array\_data.length; i++)
18. System.out.println(array\_data[i]);
19. }
20. }

**Output:**

****

**Class 1:** [**HashSet**](https://www.geeksforgeeks.org/hashset-in-java/)

**HashSet class which is implemented in the** [**collection framework**](https://www.geeksforgeeks.org/collections-in-java-2/) **is an inherent implementation of the** [**hash table**](https://www.geeksforgeeks.org/data-structure-gq/hash-gq/) **data structure. The objects that we insert into the HashSet do not guarantee to be inserted in the same order. The objects are inserted based on their hashcode. This class also allows the insertion of NULL elements.**

**Class 2:** [**EnumSet**](https://www.geeksforgeeks.org/enumset-class-java/)

**EnumSet class which is implemented in the** [**collections framework**](https://www.geeksforgeeks.org/collections-in-java-2/) **is one of the specialized implementations of the Set interface for use with the** [**enumeration type**](https://www.geeksforgeeks.org/enum-in-java/)**. It is a high-performance set implementation, much faster than HashSet. All of the elements in an enum set must come from a single enumeration type that is specified when the set is created either explicitly or implicitly.**

**Class 3:** [**LinkedHashSet**](https://www.geeksforgeeks.org/linkedhashset-in-java-with-examples/)

**LinkedHashSet class which is implemented in the** [**collections framework**](https://www.geeksforgeeks.org/collections-in-java-2/) **is an ordered version of HashSet that maintains a** [**doubly-linked List**](https://www.geeksforgeeks.org/doubly-linked-list/) **across all elements. When the iteration order is needed to be maintained this class is used. When iterating through a HashSet the order is unpredictable, while a LinkedHashSet lets us iterate through the elements in the order in which they were inserted.**

**Class 4:** [**TreeSet**](https://www.geeksforgeeks.org/treeset-in-java/)

**TreeSet class which is implemented in the** [**collections framework**](https://www.geeksforgeeks.org/collections-in-java-2/) **and implementation of the** [**SortedSet Interface**](https://www.geeksforgeeks.org/sortedset-java-examples/) **and SortedSet extends Set Interface. It behaves like a simple set with the exception that it stores elements in a sorted format. TreeSet uses a tree data structure for storage. Objects are stored in sorted, ascending order. But we can iterate in descending order using the method TreeSet.descendingIterator(). Let’s see how to create a set object using this class.**

**Distributing items when a person cannot take more than two items of same type**

import java.util.HashMap;

import java.util.Map;

public class distribute

{

static boolean checkCount(int arr[], int n, int k)

{

HashMap <Integer, Integer> hash = new HashMap<>();

for (int i = 0; i < n; i++)

{

if (!hash.containsKey(arr[i]))

hash.put(arr[i], 0);

hash.put(arr[i], hash.get(arr[i]) + 1);

}

for (Map.Entry x : hash.entrySet())

if ((int)x.getValue() > 2 \* k)

return false;

return true;

}

public static void main(String []args)

{

int arr[] = { 1, 1, 2, 3, 1 };

int n = arr.length;

int k = 2;

if (checkCount(arr, n, k))

System.out.println("Yes");

else

System.out.println("No");

}

}