**Krishnendu’s cheatsheet**

**Important Concepts:**

|  |  |
| --- | --- |
| Gap traversal:  public static void main(String[] args) {  int[][] arr = {  {5, 6, 7, 8},  {0, 1, 2, 9},  {5, 2, 5, 8},  {0, 3, 4, 0},  // {2, 3, 4, 5}  };  for (int g = 0; g < arr.length; g++) {  for (int i = 0, j = g; j < arr[0].length; i++, j++) {  if (g == 0) {  System.out.print(arr[i][j] + " ");  }  }  }  } | Prefix suffix:  public static void main(String[] args) {  String str="aabb";  for (int i = 0; i < str.length(); i++) {  String pre=str.substring(0,i+1);  String suff=str.substring(i+1);  System.out.println(pre+" "+suff);  }  } |
| String swap:  static String swap(String str, int i, int j) {  char ith = str.charAt(i);  char jth = str.charAt(j);  String pre = str.substring(0, i);  String mid = str.substring(i + 1, j);  String post = str.substring(j + 1);  return pre + jth + mid + ith + post;  } | Subsequence:  public static void main(String[] args) {  String s = "abc";  sol(s, "");  }  private static void sol(String q, String a) {  if (q.length() == 0) {  System.out.println("" + a);  return;  }  char ch = q.charAt(0);  String rest = q.substring(1);  sol(rest, a);  sol(rest, ch + a);  } |
| Comparable:  class Student implements Comparable<Student>{  int id;  int age;  public Student(int id, int age) {  this.id = id;  this.age = age;  }  @Override  public int compareTo(Student s) {  if(age>s.age){  return 1;  }else{  return -1;  }  }  }  public class WhyComparable {  public static void main(String args[]){  Student s1=new Student(1,14);  Student s2=new Student(1,13);  System.out.println(""+s1.compareTo(s2));  }  } | PriorityQueue:  public static void main(String[] args) {  PriorityQueue<Integer> p = new PriorityQueue<>();  p.add(2);  p.add(3);  p.add(1);  while (p.size() > 0) {  System.out.println("" + p.remove());  }  } |

**Basic Math:**

|  |  |
| --- | --- |
| Count digits:  private static int countDigits(int n) {  int cnt = 0;  while (n != 0) {  n = n / 10;  cnt++;  }  return cnt;  } | Count digits recursive:  public static int countDigitsR(int n) {  if (n == 0) {  return 0;  } else {  return 1 + countDigitsR(n / 10);  }  } |
| Is palindrome:  static boolean isPalindrome(int n) {  int temp = n;  int rev = 0;  while (temp != 0) {  int ld = temp % 10;  rev = rev \* 10 + ld;  temp = temp / 10;  }  return (rev == n);  } | Factorial recursive:  static int factR(int n) {  if (n == 0) {  return 1;  }  return n \* factR(n - 1);  } |
| Count trailing zeroes:  static int countTrailingZeroes(int n) {  int res = 0;  for (int i = 5; i <= n; i = i \* 5) {  res = res + n / i;  }  return res;  } | Check prime:  static boolean isPrime(int n) {  if (n == 1) {  return false;  }  for (int i = 2; i \* i <= n; i++) {  if (n % i == 0) {  return false;  }  }  return true;  } |
| Calculate power:  static int power(int x, int n) {  if (n == 0) {  return 1;  }  int temp = power(x, n / 2);  temp = temp \* temp;  if (n % 2 == 0) {  return temp;  } else {  return temp \* x;  }  } | Prime seive:  public static void main(String[] args) {  boolean myseive[] = seive(20);  for (int i = 0; i < myseive.length; i++) {  System.out.println(i + " " + myseive[i]);  }  }  static boolean[] seive(int n) {  boolean[] arr = new boolean[n + 1];  Arrays.fill(arr, true);  arr[0] = false;  arr[1] = false;  for (int i = 2; i < Math.sqrt(n); i++) {  for (int j = 2 \* i; j <= n; j = j + i) {  arr[j] = false;  }  }  return arr;  } |
| Trailing zeroes:  public static void main(String[] args) {  int res=1000;  int n=7;  for(int i=5;i<=n;i=i\*5){  res=res+n/i;  }  System.out.println("zeroes: "+res);  } |  |

**Binomial:-**

|  |  |
| --- | --- |
| nCr:  static int nCr(int n, int r) {  int[] dp = new int[r + 1];  dp[0] = 1;  for (int i = 1; i <= n; i++) {  for (int j = Math.min(i, r); j > 0; j--) {  dp[j] = dp[j] + dp[j - 1];  }  }  return dp[r];  } | Combination:  static int nCr(int n, int r) {  int res = 1;  for (int i = n; i >= (n - r + 1); i--) {  res = res \* i;  }  for (int i = 2; i <= r; i++) {  res = res / i;  }  return res;  } |
| Permutation of String:  static void permute(String str, int i) {  if (i == str.length() - 1) {  System.out.println(str);  } else {  for (int j = i; j <= str.length() - 1; j++) {  str = swap(str, i, j);  permute(str, i + 1);  str = swap(str, i, j);  }  }  }  static String swap(String a, int i, int j) {  char temp;  char[] charArray = a.toCharArray();  temp = charArray[i];  charArray[i] = charArray[j];  charArray[j] = temp;  return String.valueOf(charArray);  } |  |

**Catalan:**

|  |  |
| --- | --- |
| static int catalanDP(int n) {  // Table to store results of subproblems  int dp[] = new int[n + 2];  // Initialize first two values in table  dp[0] = 1;  dp[1] = 1;  // Fill entries in dp[]  // using recursive formula  for (int i = 2; i <= n; i++) {  dp[i] = 0;  for (int j = 0; j < i; j++) {  dp[i] += dp[j] \* dp[i - j - 1];  }  }  // Return last entry  return dp[n];  } |  |

**Divisors:**

|  |  |
| --- | --- |
| All divisors:  static void printDivisors(int n) {  for (int i = 1; i \* i <= n; i++) {  if (n % i == 0) {  System.out.print(i + " ");  if (i != n / i) {  System.out.print((n / i) + " ");  }  }  }  } | Count divisors:  static int countDivisors(int n) {  int res = 1;  for (int i = 2; i \* i <= n; i++) {  int cnt = 0;  while (n % i == 0) {  n = n / i;  cnt++;  }  res = res \* (cnt + 1);  }  if (n >= 2) {  res = res \* 2;  }  return res;  } } |
|  |  |

**Fibonacci:**

|  |  |
| --- | --- |
| Fib:  //Fm+n=FmFn+1+Fm-1Fn;  static int fib(int n) {  if (n == 0 || n == 1) {  return 1;  } else {  return fib(n - 1) + fib(n - 2);  }  } | //Cassini  //Fn+1\*Fn-1-Fn^2=(-1)^n;  double nplusone = fib(5);  double nminusone = fib(3);  double nsqr = Math.pow(fib(4), 2);  double res=Math.pow(-1,4);  double result = nplusone \* nminusone - nsqr;  System.out.println("" + result);  System.out.println("" + res); |
| DudenyCow:  static int count(int n) {  if (n == 0 || n == 1) {  return 1;  } else {  return count(n - 1) + count(n - 2);  }  } | FibomodM:  static int fibMod(int n, int m)  {  ArrayList<Integer> fib=new ArrayList<>();  fib.add(0);  fib.add(1);  int pis=1;  for(int i=2;i<m\*m;i++)  {  fib.add(fib.get(i-1)+fib.get(i-2));  if(fib.get(i)%m==1 && fib.get(i-1)%m==0){  pis=i-1;  break;  }  }  return fib.get(n%pis)%m;    } |
| GCD(Fm,Fn)=Fgcd(m,n):-  static int gcd(int a, int b)  {  // Everything divides 0  if (a == 0)  return b;  if (b == 0)  return a;    if (a == b)  return a;  if (a > b)  return gcd(a-b, b);  return gcd(a, b-a);  } | Binary strings with no two consecutive ones:-  public class BinaryStringswithNo2consecutive1 {  static int fib(int n) {  if (n == 0 || n == 1) {  return 1;  } else {  return fib(n - 1) + fib(n - 2);  }  }  public static void main(String[] args) {  int n=3;  int count=fib(n+2);  System.out.println(""+count);  }  } |

**GCD & LCM:**

|  |  |
| --- | --- |
| Euclid:  static int gcd(int a, int b)  {  if (b == 0)  return a;  return gcd(b, a % b);  } | Extended Euclid:  public static int gcdExtended(int a, int b, int x, int y)  {  // Base Case  if (a == 0)  {  x = 0;  y = 1;  return b;  }  int x1=1, y1=1;  // To store results of recursive call  int gcd = gcdExtended(b%a, a, x1, y1);  // Update x and y using results of recursive  // call  x = y1 - (b/a) \* x1;  y = x1;  return gcd;  } |
| // Recursive function to return gcd of a and b:  static int gcd(int a, int b)  {  // Everything divides 0  if (a == 0)  return b;  if (b == 0)  return a;    // base case  if (a == b)  return a;    // a is greater  if (a > b)  return gcd(a-b, b);  return gcd(a, b-a);  } | LCM:  static int lcm(int a, int b)  {  return (a / gcd(a, b)) \* b;  } |

**Geometric:**

|  |  |
| --- | --- |
| Orientation:  public static int orientation(Point p, Point q, Point r) {  int val = (q.y - p.y) \* (r.x - q.x)  - (q.x - p.x) \* (r.y - q.y);  if (val == 0) {  return 0; // collinear  }  return (val > 0) ? 1 : 2; // clock or counterclock wise  } |  |

**Number Theory:**

|  |  |
| --- | --- |
| // Function to return GCD of a and b  static int gcd(int a, int b)  {  if (a == 0)  return b;  return gcd(b % a, a);  }  // A simple method to evaluate  // Euler Totient Function  static int phi(int n)  {  int result = 1;  for (int i = 2; i < n; i++)  if (gcd(i, n) == 1)  result++;  return result;  } |  |

**Prime:**

|  |  |
| --- | --- |
| Check prime:  static boolean isPrime(int n) {  if (n == 1) {  return false;  }  if (n == 2 || n == 3) {  return true;  }  if (n % 2 == 0 || n % 3 == 0) {  return false;  }  for (int i = 5; i \* i <= n; i = i + 6) {  if (n % i == 0 || n % (i + 2) == 0) {  return false;  }  }  return true;  } |  |

**Prime Factorization:**

|  |  |
| --- | --- |
| Prime factors:  static void printPrimeFactors(int n) {  if (n <= 1) {  return;  }  while (n % 2 == 0) {  System.out.print(2 + " ");  n = n / 2;  }  while (n % 3 == 0) {  System.out.print(3 + " ");  n = n / 3;  }  for (int i = 5; i \* i <= n; i = i + 6) {  while (n % i == 0) {  System.out.print(i + " ");  n = n / i;  }  while (n % (i + 2) == 0) {  System.out.print((i + 2) + " ");  n = n / (i + 2);  }  }  if (n > 3) {  System.out.print(n + " ");  }  System.out.println();  } | Smallest prime factor:  public static void smallestPrimeFactor(int n) {  // Create a vector to store least primes.  // Initialize all entries as 0.  int[] least\_prime = new int[n + 1];  // We need to print 1 for 1.  System.out.print(1 + " ");  for (int i = 2; i <= n; i++) {  // least\_prime[i] == 0  // means it i is prime  if (least\_prime[i] == 0) {  System.out.print(i + " ");  // mark it as a divisor for all its  // multiples if not already marked  for (int j = i \* i; j <= n; j += i) {  if (least\_prime[j] == 0) {  least\_prime[j] = i;  }  }  } else {  System.out.print(least\_prime[i] + " ");  }  }  } |

**Bit magic:**

|  |  |
| --- | --- |
| Operations:  System.out.println(a&b);  System.out.println(a|b);  System.out.println(a^b);  // left shift  System.out.println(a << 1);  System.out.println(a << 2);    // right shift  System.out.println(a >> 1);  System.out.println(a >> 2);  // negation Operator  System.out.println(~a); | Binary to gray:  public static int greyConverter(int n) {  return n ^ (n >> 1);  }  Gray to binary:  public static int binaryConverter(int n) {  int res = 0;  while (n > 0) {  res ^= n;  n >>= 1;  }  return res;  } |
| Check Kth bit:  public static void isKthBitSet(int n,  int k) {  if ((n & (1 << (k - 1))) == 1) {  System.out.print("SET");  } else {  System.out.print("NOT SET");  }  } | Count set bits:  static int countSetBits(int n) {  int count = 0;  while (n > 0) {  n &= (n - 1);  count++;  }  return count;  } |
| Power of two:  static boolean isPow2(int n) {  if (n == 0) {  return true;  }  return ((n & (n - 1)) == 0);  } | Power:  static int power(int x, int y) {  // Initialize result  int res = 1;  while (y > 0) {  // If y is odd,  // multiply  // x with result  if ((y & 1) == 1) {  res = res \* x;  }  // y must be even now  y = y >> 1; // y = y/2  x = x \* x; // Change x to x^2  }  return res;  } |
| Even odd:  static void evenodd(int a) {  System.out.println((a & 1) == 0 ? "EVEN" : "ODD");  } | Mul by 2:  static void mulby2(int a) {  System.out.println(a << 1);  } |
| Div by 2:  static void divby2(int a) {  System.out.println(a >> 1);  } | Swap:  static void swap(int a, int b) {  a ^= b;  b ^= a;  a ^= b;  System.out.println(a + " "+b);  } |

**Graph algo:**

|  |  |
| --- | --- |
| DFS:  public class DFS {  static void addEdge(ArrayList<ArrayList<Integer>> adj, int u, int v) {  adj.get(u).add(v);  adj.get(v).add(u);  }  static void DFSRec(ArrayList<ArrayList<Integer>> adj, int s, boolean[] visited) {  visited[s]=true;  System.out.print(" "+s);  for(int nbr:adj.get(s)){  if(visited[nbr]==false){  visited[nbr]=true;  DFSRec(adj,nbr,visited);  }  }  }  static void DFS(ArrayList<ArrayList<Integer>> adj, int V, int s) {  boolean[] visited = new boolean[V];  DFSRec(adj, s, visited);  }  public static void main(String[] args) {  int V = 7;  ArrayList<ArrayList<Integer>> adj = new ArrayList<ArrayList<Integer>>(V);  for (int i = 0; i < V; i++) {  adj.add(new ArrayList<Integer>());  }  addEdge(adj, 0, 1);  addEdge(adj, 0, 2);  addEdge(adj, 2, 3);  addEdge(adj, 1, 3);  addEdge(adj, 1, 4);  addEdge(adj, 3, 4);  System.out.println("Following is Depth First Traversal: ");  DFS(adj, V, 0);  }  } | BFS:  public class BFS {  static void addEdge(ArrayList<ArrayList<Integer>> adj, int u, int v) {  adj.get(u).add(v);  adj.get(v).add(u);  }  static void BFS(ArrayList<ArrayList<Integer>> adj, int V, int s) {  boolean[] visited = new boolean[V];  visited[s] = true;  //System.out.println(""+s);  Queue<Integer> q = new LinkedList<>();  q.add(s);  while (q.size() > 0) {  int f=q.poll();  System.out.println(f);  for (int nbr : adj.get(f)) {  if(visited[nbr]==false){  visited[nbr]=true;  q.add(nbr);  }  }  }  }  public static void main(String[] args) {  int V = 5;  ArrayList<ArrayList<Integer>> adj = new ArrayList<ArrayList<Integer>>(V);  for (int i = 0; i < V; i++) {  adj.add(new ArrayList<Integer>());  }  addEdge(adj, 0, 1);  addEdge(adj, 0, 2);  addEdge(adj, 1, 2);  addEdge(adj, 2, 3);  addEdge(adj, 1, 3);  addEdge(adj, 3, 4);  addEdge(adj, 2, 4);  System.out.println("Following is Breadth First Traversal: ");  BFS(adj, V, 0);  }  } |
| Shortest path(Dijkastra):  public class ShortestPathinWeights {  static class Edge {  int src;  int nbr;  int wt;  Edge(int src, int nbr, int wt) {  this.src = src;  this.nbr = nbr;  this.wt = wt;  }  }  public static class Pair implements Comparable<Pair> {  int v;  String psf;  int wsf;  Pair(int v, String psf, int wsf) {  this.v = v;  this.psf = psf;  this.wsf = wsf;  }  public int compareTo(Pair o) {  return this.wsf - o.wsf;  }  }  static void addEdge(ArrayList<Edge>[] graph, int v1, int v2, int wt) {  graph[v1].add(new Edge(v1, v2, wt));  graph[v2].add(new Edge(v2, v1, wt));  }  public static void main(String[] args) throws Exception {  int vtces = 7;  int edges = 9;  ArrayList<Edge>[] graph = new ArrayList[vtces];  for (int i = 0; i < vtces; i++) {  graph[i] = new ArrayList<>();  }  addEdge(graph, 0, 1, 10);  addEdge(graph, 1, 2, 10);  addEdge(graph, 2, 3, 10);  addEdge(graph, 0, 3, 40);  addEdge(graph, 3, 4, 2);  addEdge(graph, 4, 5, 3);  addEdge(graph, 5, 6, 3);  addEdge(graph, 4, 6, 8);  addEdge(graph, 2, 5, 5);  int src = 0;  boolean[] visited=new boolean[vtces];  PriorityQueue<Pair> pq=new PriorityQueue<>();  pq.add(new Pair(src,src+"",0));    while(pq.size()>0){  Pair rem=pq.remove();  if(visited[rem.v]==true){  continue;  }  visited[rem.v]=true;  System.out.println(rem.v+" via "+rem.psf+" @ "+rem.wsf);  for(Edge e:graph[rem.v]){  if(visited[e.nbr]==false){  pq.add(new Pair(e.nbr,rem.psf+e.nbr,rem.wsf+e.wt));  }  }  }  }  } | BFS Path:  public class BFSPath {  static class Edge {  int src;  int nbr;  Edge(int src, int nbr) {  this.src = src;  this.nbr = nbr;  }  }  static class Pair {  int v;  String psf;  Pair(int v, String psf) {  this.v = v;  this.psf = psf;  }  }  static void addEdge(ArrayList<Edge>[] graph, int v1, int v2) {  graph[v1].add(new Edge(v1, v2));  graph[v2].add(new Edge(v2, v1));  }  public static void main(String[] args) throws Exception {  int vtces = 7;  ArrayList<Edge>[] graph = new ArrayList[vtces];  for (int i = 0; i < vtces; i++) {  graph[i] = new ArrayList<>();  }  int edges = 8;  addEdge(graph, 0, 1);  addEdge(graph, 1, 2);  addEdge(graph, 2, 3);  addEdge(graph, 0, 3);  addEdge(graph, 3, 4);  addEdge(graph, 4, 5);  addEdge(graph, 5, 6);  addEdge(graph, 4, 6);  int src = 2;  ArrayDeque<Pair> q = new ArrayDeque<>();  boolean[] visited = new boolean[vtces];  //visited[src]=true;  q.add(new Pair(src, src + ""));  while (!q.isEmpty()) {  Pair rem = q.removeFirst();  if (visited[rem.v] == true) {  continue;  }  visited[rem.v] = true;  System.out.println(rem.v + "@" + rem.psf);  for (Edge e : graph[rem.v]) {  if (visited[e.nbr] == false) {  q.add(new Pair(e.nbr, rem.psf + e.nbr));  }  }  }  }  } |
| Get connected components:  public class GetConnectedComponents {  static void addEdge(ArrayList<ArrayList<Integer>> adj, int u, int v) {  adj.get(u).add(v);  adj.get(v).add(u);  }  public static void main(String[] args) {  int V = 7;  ArrayList<ArrayList<Integer>> adj = new ArrayList<ArrayList<Integer>>(V);  for (int i = 0; i < V; i++) {  adj.add(new ArrayList<Integer>());  }  addEdge(adj, 0, 1);  addEdge(adj, 0, 2);  addEdge(adj, 2, 3);  addEdge(adj, 1, 3);  addEdge(adj, 1, 4);  addEdge(adj, 3, 4);  System.out.println("Following is Depth First Traversal: ");  //DFS(adj, V, 0);  ArrayList<ArrayList<Integer>> comps = new ArrayList<>();  boolean[] visited = new boolean[V];  for (int i = 0; i < V; i++) {  if (visited[i] == false) {  ArrayList<Integer> comp = new ArrayList<>();  drawTree(adj, i, comp, visited);  comps.add(comp);  }  }  System.out.println(comps);  }  static void drawTree(ArrayList<ArrayList<Integer>> adj, int src, ArrayList<Integer> comp, boolean[] visited) {  visited[src] = true;  comp.add(src);  for (int nbr : adj.get(src)) {  if (visited[nbr] == false) {  drawTree(adj, nbr, comp, visited);  }  }  }  } | Is connected:  public class IsConnected {  static void addEdge(ArrayList<ArrayList<Integer>> adj, int u, int v) {  adj.get(u).add(v);  adj.get(v).add(u);  }  public static void main(String[] args) {  int V = 7;  ArrayList<ArrayList<Integer>> adj = new ArrayList<ArrayList<Integer>>(V);  for (int i = 0; i < V; i++) {  adj.add(new ArrayList<Integer>());  }  addEdge(adj, 0, 1);  addEdge(adj, 0, 2);  addEdge(adj, 2, 3);  addEdge(adj, 1, 3);  addEdge(adj, 1, 4);  addEdge(adj, 3, 4);    ArrayList<ArrayList<Integer>> comps = new ArrayList<>();  boolean[] visited = new boolean[V];  for (int i = 0; i < V; i++) {  if (visited[i] == false) {  ArrayList<Integer> comp = new ArrayList<>();  drawTree(adj, i, comp, visited);  comps.add(comp);  }  }  System.out.println(comps.size()==1);  }    static void drawTree(ArrayList<ArrayList<Integer>> adj, int src, ArrayList<Integer> comp, boolean[] visited) {  visited[src] = true;  comp.add(src);  for (int nbr : adj.get(src)) {  if (visited[nbr] == false) {  drawTree(adj, nbr, comp, visited);  }  }  }  } |
| Is bipartite:  public class IsBipartite {  static class Edge {  int src;  int nbr;  int wt;  Edge(int src, int nbr, int wt) {  this.src = src;  this.nbr = nbr;  this.wt = wt;  }  }  public static class Pair {  int v, level;  String psf;  Pair(int v, String psf, int level) {  this.v = v;  this.psf = psf;  this.level = level;  }  }  static void addEdge(ArrayList<Edge>[] graph, int v1, int v2, int wt) {  graph[v1].add(new Edge(v1, v2, wt));  graph[v2].add(new Edge(v2, v1, wt));  }  public static void main(String[] args) {  int vtces = 7;  ArrayList<Edge>[] graph = new ArrayList[vtces];  for (int i = 0; i < vtces; i++) {  graph[i] = new ArrayList<>(0);  }  int edges = 9;  addEdge(graph, 0, 1, 10);  addEdge(graph, 1, 2, 10);  addEdge(graph, 2, 3, 10);  addEdge(graph, 0, 3, 10);  addEdge(graph, 3, 4, 10);  addEdge(graph, 4, 5, 10);  addEdge(graph, 5, 6, 10);  addEdge(graph, 4, 6, 10);  int[] visited = new int[vtces];  Arrays.fill(visited, -1);  for (int i = 0; i < vtces; i++) {  if (visited[i] == -1) {  boolean isBi = check(graph, i, visited);  if (isBi == false) {  System.out.println(false);  return;  }  }  }  System.out.println(true);  }  static boolean check(ArrayList<Edge>[] graph, int src, int[] visited) {  ArrayDeque<Pair> q = new ArrayDeque<>();  q.add(new Pair(src, src + "", 0));  while (q.size() > 0) {  Pair rem = q.removeFirst();  if (visited[rem.v] != -1) {  if (rem.level != visited[rem.v]) {  return false;  }  } else {  visited[rem.v] = rem.level;  }  for (Edge e : graph[rem.v]) {  if (visited[e.nbr] == -1) {  q.add(new Pair(e.nbr, rem.psf + e.nbr, rem.level + 1));  }  }  }  return true;  }  } | Hamiltonian:  public class Hamiltonian {  static void addEdge(ArrayList<ArrayList<Integer>> adj, int u, int v) {  adj.get(u).add(v);  adj.get(v).add(u);  }  public static void main(String[] args) {  int V = 7;  ArrayList<ArrayList<Integer>> adj = new ArrayList<ArrayList<Integer>>(V);  for (int i = 0; i < V; i++) {  adj.add(new ArrayList<Integer>());  }  addEdge(adj, 0, 1);  addEdge(adj, 1, 2);  addEdge(adj, 2, 3);  addEdge(adj, 0, 3);  addEdge(adj, 3, 4);  addEdge(adj, 4, 5);  addEdge(adj, 5, 6);  addEdge(adj, 4, 6);  addEdge(adj, 2, 5);  int src = 0;  HashSet<Integer> visited = new HashSet<>();  hamil(adj, src, visited, src + "", src);  }  public static void hamil(ArrayList<ArrayList<Integer>> adj, int src, HashSet<Integer> visited,  String psf, int osrc) {  if (visited.size() == adj.size() - 1) {  System.out.print(psf);  boolean closingEdge = false;  for (int nbr : adj.get(src)) {  if (nbr == osrc) {  closingEdge = true;  break;  }  }  if (closingEdge == true) {  System.out.println("\*");  } else {  System.out.println(".");  }  return;  }  visited.add(src);  for (int nbr : adj.get(src)) {  if (visited.contains(nbr) == false) {  hamil(adj, nbr, visited, psf + nbr, osrc);  }  }  visited.remove(src);  }  } |
| Print All paths:  public class PrintAllPaths {  static void addEdge(ArrayList<ArrayList<Integer>> adj, int u, int v) {  adj.get(u).add(v);  adj.get(v).add(u);  }  static void dfs(ArrayList<ArrayList<Integer>> adj, int V, int src, int dest) {  boolean[] visited = new boolean[V];  printAllPaths(adj, src, dest, visited, "");  }  static void printAllPaths(ArrayList<ArrayList<Integer>> adj, int src, int dest, boolean[] visited, String psf) {  if (src == dest) {  System.out.println("" + psf);  return;  }  visited[src] = true;  for (int nbr : adj.get(src)) {  if (visited[nbr] == false) {  printAllPaths(adj, nbr, dest, visited, psf + " " + nbr);  }  }  }  public static void main(String[] args) {  int V = 7;  ArrayList<ArrayList<Integer>> adj = new ArrayList<ArrayList<Integer>>(V);  for (int i = 0; i < V; i++) {  adj.add(new ArrayList<Integer>());  }  addEdge(adj, 0, 1);  addEdge(adj, 0, 2);  addEdge(adj, 2, 3);  addEdge(adj, 1, 3);  addEdge(adj, 1, 4);  addEdge(adj, 3, 4);  addEdge(adj, 2, 4);  System.out.println("Following is Depth First Traversal: ");  dfs(adj, V, 0, 4);  }  } | Prim(MST):  public class MinimumWireConnectPCs {  static class Edge {  int src;  int nbr;  int wt;  Edge(int src, int nbr, int wt) {  this.src = src;  this.nbr = nbr;  this.wt = wt;  }  }  static class Pair implements Comparable<Pair> {  int v;  int av;  int wt;  Pair(int v, int av, int wt) {  this.v = v;  this.av = av;  this.wt = wt;  }  public int compareTo(Pair o) {  return this.wt - o.wt;  }  }  static void addEdge(ArrayList<Edge>[] graph, int v1, int v2, int wt) {  graph[v1].add(new Edge(v1, v2, wt));  graph[v2].add(new Edge(v2, v1, wt));  }  public static void main(String[] args) throws Exception {  int vtces = 7;  ArrayList<Edge>[] graph = new ArrayList[vtces];  for (int i = 0; i < vtces; i++) {  graph[i] = new ArrayList<>();  }  int edges = 8;  addEdge(graph, 0, 1, 10);  addEdge(graph, 1, 2, 10);  addEdge(graph, 2, 3, 10);  addEdge(graph, 0, 3, 40);  addEdge(graph, 3, 4, 2);  addEdge(graph, 4, 5, 3);  addEdge(graph, 5, 6, 3);  addEdge(graph, 4, 6, 8);  PriorityQueue<Pair> pq = new PriorityQueue<>();  pq.add(new Pair(0, -1, 0));  boolean[] visited = new boolean[vtces];  while (pq.size() > 0) {  Pair rem = pq.remove();  if (visited[rem.v] == true) {  continue;  }  visited[rem.v] = true;  if (rem.av != -1) {  System.out.println("[" + rem.v + "-" + rem.av + "@" + rem.wt + "]");  }  for (Edge e : graph[rem.v]) {  if (visited[e.nbr] == false) {  pq.add(new Pair(e.nbr, rem.v, e.wt));  }  }  }  }  } |
| Topological sort:  public class OrderOfCompilation {  static class Edge {  int src;  int nbr;  Edge(int src, int nbr) {  this.src = src;  this.nbr = nbr;  }  }  static void addEdge(ArrayList<Edge>[] graph, int v1, int v2) {  graph[v1].add(new Edge(v1, v2));  graph[v2].add(new Edge(v2, v1));  }  public static void main(String[] args) throws Exception {  int vtces = 7;  ArrayList<Edge>[] graph = new ArrayList[vtces];  for (int i = 0; i < vtces; i++) {  graph[i] = new ArrayList<>();  }  int edges = 7;  addEdge(graph, 0, 1);  addEdge(graph, 1, 2);  addEdge(graph, 2, 3);  addEdge(graph, 0, 3);  addEdge(graph, 4, 5);  addEdge(graph, 5, 6);  addEdge(graph, 4, 6);  boolean[] visited = new boolean[vtces];  Stack<Integer> st = new Stack<>();  for (int i = 0; i < vtces; i++) {  if (visited[i] == false) {  topologicalSort(graph, i, visited, st);  }  }  while (st.size() > 0) {  System.out.println(st.pop());  }  }  public static void topologicalSort(ArrayList<Edge>[] graph, int src, boolean[] visited, Stack<Integer> st) {  visited[src] = true;  for (Edge e : graph[src]) {  if (visited[e.nbr] == false) {  topologicalSort(graph, e.nbr, visited, st);  }  }  st.push(src);  }  } | Spread Infection: (Time dependent)  public class SpreadofInfection {  static class Edge {  int src;  int nbr;  int wt;  Edge(int src, int nbr, int wt) {  this.src = src;  this.nbr = nbr;  this.wt = wt;  }  }  static class Pair {  int v;  int time;  Pair(int v, int time) {  this.v = v;  this.time = time;  }  }  static void addEdge(ArrayList<Edge>[] graph, int v1, int v2, int wt) {  graph[v1].add(new Edge(v1, v2, wt));  graph[v2].add(new Edge(v2, v1, wt));  }  public static void main(String[] args) throws Exception {  int vtces = 7;  ArrayList<Edge>[] graph = new ArrayList[vtces];  for (int i = 0; i < vtces; i++) {  graph[i] = new ArrayList<>(0);  }  int edges = 8;  addEdge(graph, 0, 1, 10);  addEdge(graph, 1, 2, 10);  addEdge(graph, 2, 3, 10);  addEdge(graph, 0, 3, 10);  addEdge(graph, 3, 4, 10);  addEdge(graph, 4, 5, 10);  addEdge(graph, 5, 6, 10);  addEdge(graph, 4, 6, 10);  int src = 6;  int t = 3;  ArrayDeque<Pair> q = new ArrayDeque<>();  q.add(new Pair(src, 1));  int[] visited = new int[vtces];  int count = 0;  while (q.size() > 0) {  Pair rem = q.removeFirst();  if (visited[rem.v] > 0) {  continue;  }  visited[rem.v] = rem.time;  if (rem.time > t) {  break;  }  count++;  for (Edge e : graph[rem.v]) {  if (visited[e.nbr] == 0) {  q.add(new Pair(e.nbr, rem.time + 1));  }  }  }  System.out.println("" + count);  }  } |

**Trees:**

|  |  |
| --- | --- |
| Postorder:  private void printPostorder(Node1 node) {  if (node == null) {  return;  }  printPostorder(node.left);  printPostorder(node.right);  System.out.print(node.key + " ");  } |  |
| Inorder:  private void printPostorder(Node1 node) {  if (node == null) {  return;  }  printPostorder(node.left);  System.out.print(node.key + " ");  printPostorder(node.right);  } | Iterative inorder:  static void inorderIter(Node1 root) {  if (root == null) {  return;  }  Stack<Node1> s = new Stack<Node1>();  Node1 curr = root;  while (curr != null & !s.isEmpty()) {  while (curr != null) {  s.push(curr);  curr = curr.left;  }  curr = s.pop();  System.out.print(" " + curr.key);  curr = curr.right;  }  } |
| Preorder:  private void printPostorder(Node1 node) {  if (node == null) {  return;  }  System.out.print(node.key + " ");  printPostorder(node.left);  printPostorder(node.right);  } | Size:  public int size(Node1 root) {  if (root == null) {  return 0;  } else {  return 1 + size(root.left) + size(root.right);  }  } |
| Max:  public int max(Node1 root) {  if (root == null) {  return -1;  } else {  return Math.max(root.key, Math.max(max(root.left), max(root.right)));  }  } | Height:  public int height(Node1 root) {  if (root == null) {  return 0;  } else {  return (Math.max(height(root.left), height(root.right))) + 1;  }  } |
| Print at distance K:  public void printKdist(Node1 root, int k) {  if (root == null) {  return;  }  if (k == 0) {  System.out.print(root.key + " ");  } else {  printKdist(root.left, k - 1);  printKdist(root.right, k - 1);  }  } | Diameter:  int diameter(Node root) {  if (root == null) {  return 0;  }  int d1 = 1 + height(root.left) + height(root.right);  int d2 = diameter(root.left);  int d3 = diameter(root.right);  return Math.max(d1, Math.max(d2, d3));  } |
| Find a node:  static Node1 find(Node1 root, int x) {  if (root == null) {  return null;  }  if (root.key == x) {  return root;  }  Node1 left = find(root.left, x);  if (left != null) {  return left;  }  Node1 right = find(root.right, x);  if (right != null) {  return right;  }  return null;  } | Pair sum:  static void pairSum(Node1 curr, int sum, Node1 root) {  if (curr == null) {  return;  }  int s = sum - curr.key;  Node1 f = null;  if (curr.key < s) {  f = find(root, s);  }  if (f != null) {  System.out.println(curr.key + " " + f.key);  }  pairSum(curr.left, sum, root);  pairSum(curr.right, sum, root);  } |
| Maximum width:  int maxWidth(Node1 root) {  Queue<Node1> q = new LinkedList<Node1>();  q.add(root);  int res = 0;  while (!q.isEmpty()) {  int size = q.size();  res = Math.max(res, size);  for (int i = 0; i < size; i++) {  Node1 curr = q.poll();  if (curr.left != null) {  q.add(curr.left);  }  if (curr.right != null) {  q.add(curr.right);  }  }  }  return res;  } | Is balanced:  int isBalanced(Node1 root) {  if (root == null) {  return 0;  }  int lh = isBalanced(root.left);  if (lh == -1) {  return -1;  }  int rh = isBalanced(root.right);  if (rh == -1) {  return -1;  }  if (Math.abs(lh - rh) > 1) {  return -1;  } else {  return (Math.max(lh, rh) + 1);  }  } |
| Left view:  public void leftView(Node1 root) {  Queue<Node1> q = new LinkedList<Node1>();  q.add(root);  while (!q.isEmpty()) {  int cnt = q.size();  for (int i = 0; i < cnt; i++) {  Node1 curr = q.poll();  if (i == 0) {  System.out.print(curr.key + " ");  }  if (curr.left != null) {  q.add(curr.left);  }  if (curr.right != null) {  q.add(curr.right);  }  }  }  } | Level order line by line:  public void levelOrderLinebyLine(Node1 root) {  Queue<Node1> q = new LinkedList<Node1>();  q.add(root);  q.add(null);  while (q.size() > 1) {  Node1 curr = q.poll();  if (curr == null) {  System.out.println();  q.add(null);  continue;  }  System.out.print(curr.key + " ");  if (curr.left != null) {  q.add(curr.left);  }  if (curr.right != null) {  q.add(curr.right);  }  }  } |
| Print spiral:  public void printSpiral(Node1 root) {  if (root == null) {  return;  }  Queue<Node1> q = new LinkedList<Node1>();  Stack<Integer> s = new Stack<Integer>();  boolean reverse = false;  q.add(root);  while (q.isEmpty() == false) {  int cnt = q.size();  for (int i = 0; i < cnt; i++) {  Node1 curr = q.poll();  if (reverse) {  s.push(curr.key);  } else {  System.out.print(curr.key + " ");  }  if (curr.left != null) {  q.add(curr.left);  }  if (curr.right != null) {  q.add(curr.right);  }  if (reverse) {  while (s.isEmpty() == false) {  System.out.print(s.pop() + " ");  }  }  reverse = !reverse;  //System.out.println();  }  }  } | Is children sum:  boolean isChildrenSum(Node1 root) {  if (root == null) {  return true;  }  if (root.left == null && root.right == null) {  return true;  }  int sum = 0;  if (root.left != null) {  sum = sum + root.left.key;  }  if (root.right != null) {  sum = sum + root.right.key;  }  return (root.key == sum && isChildrenSum(root.left) && isChildrenSum(root.right));  } |
| Binary tree to Linked List:  static Node BinTree2LL(Node root) {  if (root == null) {  return root;  }  Node head = BinTree2LL(root.left);  if (prev == null) {  head = root;  } else {  root.left = prev;  prev.right = root;  }  prev = root;  BinTree2LL(root.right);  return head;  } | LCA:  static Node LCA(Node root, int n1, int n2) {  if (root == null) {  return null;  }  if (root.key == n1 || root.key == n2) {  return root;  }  Node lca1 = LCA(root.left, n1, n2);  Node lca2 = LCA(root.right, n1, n2);  if (lca1 != null && lca2 != null) {  return root;  }  if (lca1 != null) {  return lca1;  } else {  return lca2;  }  } |
| Left clone:  private static Node leftclone(Node root) {  if (root == null) {  return null;  }  Node lcr = leftclone(root.left);  Node rcr = leftclone(root.right);  Node newNode = new Node(root.key, lcr, null);  root.left = newNode;  root.right = rcr;  return root;  } | Right clone:  private static Node rightclone(Node root) {  if (root == null) {  return null;  }  Node lcr = leftclone(root.left);  Node rcr = leftclone(root.right);  Node newNode = new Node(root.key, null, rcr);  root.left = lcr;  root.right = newNode;  return root;  } |
| Level Order line by line LeftView:  static void levelOrderlinebylineLeftView(Node root,int k) {  Queue<Node> q = new ArrayDeque<>();  q.add(root);  while (!q.isEmpty()) {  int size = q.size();  for (int i = 0; i < size; i++) {  Node f = q.poll();  if(i==0){  System.out.print(f.key+" ");  }    if(f.left!=null){  q.add(f.left);  }  if(f.right!=null){  q.add(f.right);  }  }  System.out.println("");  }  } | Level Order line by line RightView:  static void levelOrderlinebylineRightView(Node root,int k) {  Queue<Node> q = new ArrayDeque<>();  q.add(root);  while (!q.isEmpty()) {  int size = q.size();  for (int i = 0; i < size; i++) {  Node f = q.poll();  if(i==size-1){  System.out.print(f.key+" ");  }    if(f.left!=null){  q.add(f.left);  }  if(f.right!=null){  q.add(f.right);  }  }  System.out.println("");  }  } |
| Node to root path:  static boolean path(Node root, int data, ArrayList<Integer> path) {  if (root == null) {  return false;  }  if (root.key == data) {  path.add(root.key);  return true;  }  boolean leftf = path(root.left, data, path);  if (leftf == true) {  path.add(root.key);  return true;  }  boolean rightf = path(root.right, data, path);  if (rightf == true) {  path.add(root.key);  return true;  }  return false;  } | Remove leaf nodes:  static Node sol(Node node) {  if (node == null) {  return null;  }  if (node.left == null && node.right == null) {  return null;  }  node.left = sol(node.left);  node.right = sol(node.right);  return node;  } |
| Single child node:  static void sol(Node root, Node parent) {  if (root == null) {  return;  }  if (parent!=null && parent.left == root && parent.right == null) {  System.out.println("" + root.key);  } else if (parent!=null && parent.right == root && parent.left == null) {  System.out.println("" + root.key);  }  sol(root.left, root);  sol(root.right, root);  } |  |

BST:

|  |  |
| --- | --- |
| Insert node recursive:  Node insertRec(Node root, int key) {  if (root == null) {  root = new Node(key);  return root;  }  if (key < root.key) {  root.left = insertRec(root.left, key);  root.lcount++;  }  if (key > root.key) {  root.right = insertRec(root.right, key);  }  return root;  } | Search Recursive:  static boolean searchRec(Node root, int x) {  if (root == null) {  return false;  }  if (x == root.key) {  return true;  }  if (x < root.key) {  return searchRec(root.left, x);  }  if (x > root.key) {  return searchRec(root.right, x);  }  return false;  } |
| Search Iterative:  static boolean searchI(Node root, int x) {  Node curr = root;  while (curr != null) {  if (x == curr.key) {  return true;  }  if (x < curr.key) {  curr = curr.left;  }  if (x > curr.key) {  curr = curr.right;  }  }  return false;  } | Floor:  static Node floor(Node root, int x) {  Node res = null;  while (root != null) {  if (x == root.key) {  return root;  } else if (x < root.key) {  root = root.left;  } else {  res = root;  root = root.right;  }  }  return res;  } |
| Ceiling:  static Node ceiling(Node root, int x) {  Node res = null;  while (root != null) {  if (x == root.key) {  return root;  } else if (x < root.key) {  res = root;  root = root.left;  } else {  root = root.right;  }  }  return res;  } | Is BST:  static boolean isBST(Node root, int min, int max) {  if (root == null) {  return true;  }  return (root.key > Integer.MIN\_VALUE && root.key < Integer.MAX\_VALUE  && isBST(root.left, Integer.MIN\_VALUE, root.key)  && isBST(root.right, root.key, Integer.MAX\_VALUE));  } |
| Is pair sum:  public static boolean isPairSum(Node root, int sum, HashSet<Integer> s)  {  if(root==null)return false;  if(isPairSum(root.left,sum,s)==true){  return true;  }  if(s.contains(sum-root.key)){  return true;  }else{  s.add(root.key);  }  return isPairSum(root.right,sum,s);  } | Kth smallest:  public static Node kthSmallest(Node root, int k) {  if (root == null) {  return null;  }  int count = root.lcount + 1;  if (count == k) {  return root;  }  if (count > k) {  return kthSmallest(root.left, k);  }  return kthSmallest(root.right, k - count);  } |
| Get successor:  static Node getSucc(Node root) {  Node curr = root.right;  while (curr != null && curr.left != null) {  curr = curr.left;  }  return curr;  } | Delete:  static Node delete(Node root, int x) {  if (root == null) {  return null;  }  if (root.key > x) {  root.left = delete(root.left, x);  } else if (root.key < x) {  root.right = delete(root.right, x);  } else {  if (root.left == null) {  return root.right;  } else if (root.right == null) {  return root.left;  } else {  Node succ = getSucc(root);  root.key = succ.key;  root.right = delete(root.right, succ.key);  }  }  return root;  } |

**Linked List:**

|  |  |
| --- | --- |
| Print list:  void printList(Node head) {  Node temp = head;  while (temp != null) {  System.out.print(temp.data + " ");  temp = temp.next;  }  System.out.println();  } | Push:  void push(int new\_data) {  Node new\_node = new Node(new\_data);  new\_node.next = head;  head = new\_node;  } |
| Insert at beginning:  void insertAtBeg(int x) {  Node1 temp = new Node1(x);  temp.next = head;  head = temp;  } | Insert at end:  void insertEnd(int x) {  Node1 temp = new Node1(x);  if (head == null) {  head = temp;  return;  }  Node1 curr = head;  while (curr.next != null) {  curr = curr.next;  }  curr.next = temp;  } |
| Middle node:  void middleNode(){  Node1 fast=head,slow=head;  if(head==null){  return;  }  while(fast!=null && fast.next!=null){  fast=fast.next.next;  slow=slow.next;  }  System.out.println("\n"+slow.data + " is middle node");  } | Print nth from end:  void printNthfromEnd(int n){  int len=0;  for(Node1 curr=head;curr!=null;curr=curr.next){  len++;  }  System.out.println(len);  if(len<n)return;  Node1 curr=head;  for(int i=0;i<len-n;i++){  curr=curr.next;  }  System.out.println("\n"+curr.data + " is "+n+"th node");  } |
| Detect loop:  boolean detectLoop(){  Node1 fast=head,slow=head;  while(fast!=null && fast.next!=null){  slow=slow.next;  fast=fast.next.next;  if(slow==fast){  return true;  }  }return false;  } | Detect and remove loop:  void detectandRemoveLoop(){  Node1 fast=head,slow=head;  while(fast!=null && fast.next!=null){  slow=slow.next;  fast=fast.next.next;  if(slow==fast){  break;  }  }  if(slow!=fast){  return;  }  slow=head;  while(slow.next!=fast.next){  slow=slow.next;  fast=fast.next;  }  fast.next=null;  } |
| No of nodes:  int getCount(Node node) {  Node current = node;  int count = 0;  while (current != null) {  count++;  current = current.next;  }  return count;  } | Get node:  int getNode() {  int c1 = getCount(head1);  int c2 = getCount(head2);  int d;  if (c1 > c2) {  d = c1 - c2;  return getIntesectionNode(d, head1, head2);  } else {  d = c2 - c1;  return getIntesectionNode(d, head2, head1);  }  } |
| Intersection of two LL:  int getIntesectionNode(int d, Node node1, Node node2) {  int i;  Node current1 = node1;  Node current2 = node2;  for (i = 0; i < d; i++) {  if (current1 == null) {  return -1;  }  current1 = current1.next;  }  while (current1 != null && current2 != null) {  if (current1.data == current2.data) {  return current1.data;  }  current1 = current1.next;  current2 = current2.next;  }  return -1;  } | Pairwise swap:  Node pairWiseSwap(Node node)  {  if (node == null || node.next == null) {  return node;  }  Node remaing = node.next.next;  Node newhead = node.next;  node.next.next = node;  node.next = pairWiseSwap(remaing);  return newhead;  } |
| Recursive reverse:  static Node reverse(Node head) {  if (head == null || head.next == null) {  return head;  }  Node smallAns = reverse(head.next);  head.next.next = head;  head.next = null;  return smallAns;  } | Reverse:  Node reverse(Node head) {  Node prev = null;  Node curr = head;  Node next;  while (curr != null) {  next = curr.next;  curr.next = prev;  prev = curr;  curr = next;  }  return prev;  } |
| Is palindrome:  static boolean isPalindrome(Node head) {  Node slow = head;  boolean ispalin = true;  Stack<Integer> stack = new Stack<Integer>();  while (slow != null) {  stack.push(slow.data);  slow = slow.ptr;  }  while (head != null) {  int i = stack.pop();  if (head.data == i) {  ispalin = true;  } else {  ispalin = false;  break;  }  head = head.ptr;  }  return ispalin;  } | Merge LL:  static Node sortedMerge(Node headA, Node headB) {  Node dummyNode = new Node(0);  Node tail = dummyNode;  while (true) {  if (headA == null) {  tail.next = headB;  break;  }  if (headB == null) {  tail.next = headA;  break;  }  if (headA.data <= headB.data) {  tail.next = headA;  headA = headA.next;  } else {  tail.next = headB;  headB = headB.next;  }  tail = tail.next;  }  return dummyNode.next;  } |

Generic Tree:

|  |  |
| --- | --- |
| private static void display(Node root) {  String str = root.data + "->";  for (Node t : root.children) {  str = str + t.data + ",";  }  str = str + ".";  System.out.println("" + str);  for (Node t : root.children) {  display(t);;  }  } | private static int size(Node root) {  int s = 0;  for (Node t : root.children) {  int cs = size(t);  s = s + cs;  }  s = s + 1;  return s;  } |
| private static int max(Node root) {  //int max=Integer.MIN\_VALUE;  int m = root.data;  for (Node t : root.children) {  int cm = max(t);  if (cm > m) {  m = cm;  }  }  return m;  } | private static int height(Node root) {  int ht = -1;  for (Node t : root.children) {  int chld = height(t);  ht = Math.max(ht, chld);  }  ht = ht + 1;  return ht;  } |
| private static void reverse(Node root) {  for(Node t:root.children){  reverse(t);  }  Collections.reverse(root.children);  } | Level order line by line:  private static void levelOrderlinebyline(Node root) {  Queue<Node> q = new LinkedList<>();  Queue<Node> cq = new LinkedList<>();  q.add(root);  while (q.size() > 0) {  Node front = q.poll();  System.out.print(front.data + " ");  for (Node t : front.children) {  cq.add(t);  }  if (q.size() == 0) {  q = cq;  cq = new LinkedList<>();  System.out.println("");  }  }  } |
| Level order zig jag:  private static void levelOrderZigZag(Node root) {  Stack<Node> ms = new Stack<>();  Stack<Node> cs = new Stack<>();  ms.push(root);  int level = 1;  while (ms.size() > 0) {  root = ms.pop();  System.out.print(root.data + " ");  if (level % 2 == 1) {  for (int i = 0; i < root.children.size(); i++) {  Node t = root.children.get(i);  cs.push(t);  }  } else {  for (int i = root.children.size() - 1; i >= 0; i--) {  Node t = root.children.get(i);  cs.push(t);  }  }  if (ms.size() == 0) {  ms = cs;  cs = new Stack<>();  level++;  System.out.println();  }  }  } |  |