|  |  |  |  |
| --- | --- | --- | --- |
| **Coventry University**  **FACULITY OF Engineering and Computing**  **Coursework Cover Sheet**  *Please ensure that you complete all relevant sections legibly*  ***First Copy:*** *Attach top copy to the front of your assignment.*  ***Second Copy:*** *Keep safety as your receipt* | | | |
| **Module Code**  **210CT** | | **Student Card ID Number**  Please print in BLOCK CAPITALS  Surname……………………………..…..……………………………  Other names…………………..……….………..........  Signature………………….……...…..…...  **197100857** | |
| **Module Title**  **Programming, Algorithms & Data Structures** | | | |
| **Deadline date**  **28 Nov 2019** | **Actual word**  **Count NA** | | **Tutor**  **CYCheng** |
| **Coursework Title/Number**  **Programming Coursework** | | | |

This assessment is all

my own work and has not

been copied in part or in whole

from any other source, except for any

clearly marked up quotations. It

complies with the university regulations

on plagiarism, which I have read

and understood.

|  |  |  |
| --- | --- | --- |
| **FEEDBACK ON MARKED WORK:** *Lecturers will complete this section when work is marked.*  *NB. All marks notified during the year are provisional until confirmed by the end of year Assessment Board* | | |
| STRENGTHS  WEAKNESSES  ADVICE ON HOW WORK COULD BE IMPROVED AND FURTHER COMMENTS | | |
| If you require more feedback, please contact your tutor or see module web.  See assignment sheet for assessment criteria for this assignment. | **MARKED AWARDED** |  |
| Less any late penalty | **-** |
| Adjusted mark if penalty |  |

**Marker’s Signature** …………………………………………………………… **Date** ……………………………………………………….

|  |
| --- |
| Students are reminded that reference must be given for any previously published  work used to gather information to help write assignments, including internet  sources, but these sources should not be copied directly. |

**Second Marker Additional comments Signature** …………………………………………… **Date** …………………………

|  |
| --- |
| VTC210CT Programming, Algorithms and Data Structures Assignment Report |
| **Student Name: Thajudeen Kamel Mohamed**  **Student ID: 197100857**  **Course: VTC20CT** |
|  |

**Basic Exercise**

|  |
| --- |
| GitHub link: https://github.com/tkamel55/210CT\_Assignment.git |

|  |
| --- |
| **Week 1**  **Q1. Program to check if a 3-digit number is an Armstrong number.**  **import** java.util.Scanner;  /\*q1 is about checking whether the number entered is  an 3 digit armstrong number or not  \*/  **public** **class** Q1  {  **public** **static** **void** main(String args[])  {  **int** num, onum, total=0, rem;  Scanner scan = **new** Scanner(System.***in***);    System.***out***.print("Enter 3 digit number : ");  num = scan.nextInt();    onum = num;    **while**(onum != 0)  {  //formula for finding the armstrong number  rem = onum%10;  total = total + rem\*rem\*rem;  onum = onum/10;  }  **if**(total == num)  {  System.***out***.print(num+ " is an Armstrong Number");  }  **else**  {  System.***out***.print(num + " is not an Armstrong Number");  }  }  }  **Output of Q1:**    **Q2. Program to check whether the string is a substring.**  **import** java.util.Scanner;  /\*Q2 is about entering two strings and checking  whether the second string is the substring or not  \*/  **public** **class** Q2 {  **public** **static** **void** main(String[] args) {    Scanner in = **new** Scanner(System.***in***);    System.***out***.print("Enter string s1: ");  String s1 = in.nextLine();    System.***out***.print("Enter string s2: ");  String s2 = in.nextLine();    **if** (s1.indexOf(s2) != -1) {  System.***out***.printf(s2 + " is a substring of " + s1);  }  **else**{  System.***out***.printf(s2 + " is not a substring of " + s1);  }  }  }  **Output of Q2:**    **Week 2**  **Q3.**   1. **Circle class**   **public** **class** Circle **extends** Shape {    **private** Point center ;  **private** **double** radius;    **public** Circle(String n, Point center, **double** radius) {  **super**(n);  **this**.center = center;  **this**.radius = radius;  // **TODO** Auto-generated constructor stub  }  **public** **double** setRadius() {  **return** radius ;  }    **public** **double** getRadius() {  **return** radius;  }    **public** Point setCenter() {  **return** center;  }    **public** Point getCenter() {  **return** center;  }  **public** **double** getArea() {  // **TODO** Auto-generated method stub  **return** Math.***PI*** \* radius \* radius;  }    **public** String toString() {  **return** "Circle [center=" + center + ", radius=" + radius + "]";  }  }   1. **Square class**   **public** **class** Square **extends** Shape {  **private** Point topLeft;  **private** **double** width;    **public** Square(String n, Point topLeft, **double** width) {  **super**(n);    **this**.width = width;  **this**.topLeft = topLeft;  // **TODO** Auto-generated constructor stub  }  **public** **double** getArea() {  // **TODO** Auto-generated method stub  **return** width\*width;  }    **public** **double** setWidth() {  **return** width ;  }    **public** **double** getWidth() {  **return** width ;  }    **public** Point setTopLeft() {  **return** topLeft ;  }    **public** Point getTopLeft() {    **return** topLeft ;  }  **public** String toString() {  **return** "Square [topLeft=" + topLeft + " width=" + width + "]";  }  }  **Shape class**  **public** **abstract** **class** Shape {    **protected** String name;  **public** Shape(String n) {  name = **new** String(n);  }  **public** **abstract** **double** getArea();  **public** String getName() {    **return** name  }  }  **Class Point**  **public** **class** Point {  **private** **double** x, y; // coordinates of the Point  // constructor  **public** Point( **double** a, **double** b ) { setPoint( a, b ); }  // Set x and y coordinates of Point  **public** **void** setPoint( **double** a, **double** b ) {  x = a;  y = b;  }  // get x coordinate  **public** **double** getX() { **return** x; }  // get y coordinate  **public** **double** getY() { **return** y; }  // convert the point into a String representation  **public** String toString()  { **return** "[" + x + ", " + y + "]"; }  }  **Output of Q3:**    **Week 3**  **Q4. LinkedList method**  // q4 using linked list method to test the cases.  **class** ListNode {  **private** Object data;  **private** ListNode next;  ListNode(Object o) { data = o; next = **null**; }  ListNode(Object o, ListNode nextNode)  { data = o; next = nextNode; }  Object getData() { **return** data; }  ListNode getNext() { **return** next; }  **void** setData(Object data){  **this**.data = data;  }  **void** setNext(ListNode next){  **this**.next = next;  }  } // class ListNode    **class** EmptyListException **extends** RuntimeException {  **public** EmptyListException () { **super** ("List is empty"); }  } // class EmptyListException    **class** LinkedList {  **private** ListNode head;  **private** ListNode tail;  **public** LinkedList() { head = tail = **null**; }  **public** **boolean** isEmpty() { **return** head == **null**; }  **public** **void** addToHead(Object item) {  // TO BE COMPLETED  **if**(isEmpty())  head = tail = **new** ListNode(item);  **else**  head = **new** ListNode(item,head);  }  **public** **void** addToTail(Object item) {  // TO BE COMPLETED  **if**(isEmpty())  head = tail = **new** ListNode(item);  **else** {  tail.setNext(**new** ListNode(item));  tail = tail.getNext();  }  }    **public** Object removeFromHead() **throws** EmptyListException {  // TO BE COMPLETED  Object item = **null**;  **if**(isEmpty())  **throw** **new** EmptyListException();  item = head.getData();  **if**(head == tail)  head = tail = **null**;  **else**  head = head.getNext();  **return** item;  }  **public** Object removeFromTail() **throws** EmptyListException {  // TO BE COMPLETED  Object item = **null**;  **if**(isEmpty())  **throw** **new** EmptyListException();  item = tail.getData();  **if**(head == tail)  head = tail = **null**;  **else**{  ListNode current = head;  **while**(current.getNext() != tail)  current = current.getNext();  tail = current;  current.setNext(**null**);  }  **return** item;  }  **public** String toString () {  String s = "[ ";  ListNode current = head;  **while** (current != **null**) {  s += current.getData() + " ";  current = current.getNext();  }  **return** s + "]";  }  }  **public** **class** TestLinkedList {  **public** **static** **void** main (String args[]) {  LinkedList s = **new** LinkedList();  System.***out***.println(s);    s.addToTail (**new** Integer(42));  System.***out***.println(s);  s.addToTail (**new** Character('n'));  System.***out***.println(s);  s.addToTail (**new** String("hello"));  System.***out***.println(s);    **while** (!s.isEmpty()) {  System.***out***.println("removed:" + s.removeFromHead());  System.***out***.println(s);  }    s.addToHead (**new** Integer(42));  System.***out***.println(s);  s.addToHead (**new** Character('n'));  System.***out***.println(s);  s.addToHead (**new** String("hello"));  System.***out***.println(s);    **while** (!s.isEmpty()) {  System.***out***.println("removed:" + s.removeFromTail());  System.***out***.println(s);  }  }  }  **Output of Q4:**    **Week 4**  **Q6. ListStack**  **//using the same LinkedList class from the previous question**  // Q6 about stacks using push, pop, peek and search method.  **public** **class** Q6 **extends** LinkedList {  **public** Q6() {  **super**();  }  **public** **boolean** empty() {  **if** (isEmpty()) {  **return** **true**;  } **else** {  **return** **false**;  }  }  **public** Object push(Object item) {  addToHead(item);  **return** item;  }  **public** Object pop() {  Object item = removeFromHead();  **return** item;  }  **public** Object peek() {  Object item = get(0);  **return** item;  }  **public** **int** search(Object item) {  ListNode current = head;  **int** num = -1;  **for** (**int** i = 0; i < length; i++) {  **if** (item.equals(current.getData())) {  **return** i;  } **else** {  current = current.getNext();  }  }  **return** num;  }    **public** **static** **void** main (String args[]) {  Q6 s = **new** Q6();    System.***out***.println(s);  System.***out***.println("Patrick is at " + s.search("Patrick"));  s.push(**new** Character('A'));  System.***out***.println(s);  s.push(**new** Character('B'));  System.***out***.println(s);  s.push("Cat");  System.***out***.println(s);  s.push("Dog");  System.***out***.println(s);  s.push(**new** Integer(123));  System.***out***.println(s);  s.push("Patrick");  System.***out***.println(s);  s.push(**new** Character('E'));  System.***out***.println(s);  s.push(**new** Double(789.123));  System.***out***.println(s);  System.***out***.println("peek() returns: " + s.peek());  System.***out***.println("Patrick is at " + s.search("Patrick"));  System.***out***.println("A is at " + s.search(**new** Character('A')));  System.***out***.println("789.123 is at " + s.search(**new** Double(789.123)));  System.***out***.println("Peter is at " + s.search("Peter"));  System.***out***.println();  }  }  **Output of Q6:**    **Q7. Using linkedlist class to develop queue class**  // q7 is about queue  **public** **class** Q7 **extends** LinkedList {  **public** Q7() {  **super**();  }  **public** **void** enqueue(Object item) {  **if** (head == **null**) {  head = **new** ListNode(item);  tail = head;  } **else** {  tail.setNext(**new** ListNode(item));  tail = tail.getNext();  }  }  **public** Object dequeue() {  Object item;  **if** (head == **null**)  **return** **null**;  item = head.getData();  head = head.getNext();  **if** (head == **null**)  tail = **null**;  **return** item;  }  **public** **boolean** empty() {  **return** (head == **null**);  }    // to test the program  **public** **static** **void** main(String[] args) {  Q7 q = **new** Q7();      q.enqueue(10);  q.enqueue(20);  q.dequeue();  q.dequeue();  q.enqueue(30);  q.enqueue(40);  q.enqueue(50);    System.***out***.println("Dequeued item is " + q.dequeue());  }  }  **Output of Q7:**    **Week 5**  **Q8. Binary search algorithm**  // question no.8 about binary search  **public** **class** Q8 {  // it Returns index of x if the value is presented in arr[]  // r], else return -1  **int** binarySearch(**int** arr[], **int** l, **int** r, **int** x)  {  **if** (r >= l) {  **int** middle = l + (r - l) / 2;    // If the element is present at the  // middle itself  **if** (arr[middle] == x)  **return** middle;    // If element is smaller than mid, then  // it can only be present in left subarray  **if** (arr[middle] > x)  **return** binarySearch(arr, l, middle - 1, x);    // Else the element can only be present  // in right subarray  **return** binarySearch(arr, middle + 1, r, x);  }  // We reach here when element is not present  // in array  **return** -1;  }  // to test the program  **public** **static** **void** main(String args[])  {  Q8 ob = **new** Q8();  **int** arr[] = { 11, 12, 34, 55, 66, 77, 88 };  **int** n = arr.length;  **int** x = 77;  **int** result = ob.binarySearch(arr, 0, n - 1, x);  **if** (result == -1)  System.***out***.println("Number can't be found!");  **else**  System.***out***.println("Number " + x + " founded in index " + result);  }  }  **Output of Q8:**    **Q9. Binary Search tree with all necessary methods**  // q9 about binary search tree with traverse  //this class represent the nodes  **class** Node {    **private** Node left;  **private** Node right;  **private** Integer data;    **public** Node(Integer data) {  **this**.data = data;  }    **public** Node getLeft() {  **return** left;  }  **public** **void** setLeft(Node left) {  **this**.left = left;  }  **public** Node getRight() {  **return** right;  }  **public** **void** setRight(Node right) {  **this**.right = right;  }    **public** Integer getData() {  **return** data;  }  }    **public** **class** Q9 {    **private** Node root;    **public** **boolean** isEmpty() {    **return** (**this**.root == **null**);  }    **public** **void** insert(Integer data) {    System.***out***.print("[input: "+data+"]");  **if**(root == **null**) {  **this**.root = **new** Node(data);  System.***out***.println(" -> inserted: "+data);  **return**;  }    insertNode(**this**.root, data);  System.***out***.print(" -> inserted: "+data);  System.***out***.println();  }    **private** Node insertNode(Node root, Integer data) {    Node tmpNode = **null**;  System.***out***.print(" ->"+root.getData());  **if**(root.getData() >= data) {  System.***out***.print(" [L]");  **if**(root.getLeft() == **null**) {  root.setLeft(**new** Node(data));  **return** root.getLeft();  } **else** {  tmpNode = root.getLeft();  }  } **else** {  System.***out***.print(" [R]");  **if**(root.getRight() == **null**) {  root.setRight(**new** Node(data));  **return** root.getRight();  } **else** {  tmpNode = root.getRight();  }  }    **return** insertNode(tmpNode, data);  }    **public** **void** postOrderTraversal() {  doPostOrder(**this**.root);  }    //implementing postorder method  **private** **void** doPostOrder(Node root) {    **if**(root == **null**) **return**;  doPostOrder(root.getLeft());  doPostOrder(root.getRight());  System.***out***.print(root.getData()+" ");  }    **public** **void** inOrderTraversal() {  doInOrder(**this**.root);  }    //implementing inorder method  **private** **void** doInOrder(Node root) {    **if**(root == **null**) **return**;  doInOrder(root.getLeft());  System.***out***.print(root.getData()+" ");  doInOrder(root.getRight());  }    **public** **void** preOrderTraversal() {  doPreOrder(**this**.root);  }    // Implementing preorder method  **private** **void** doPreOrder(Node root) {    **if**(root == **null**) **return**;  System.***out***.print(root.getData()+" ");  doPreOrder(root.getLeft());  doPreOrder(root.getRight());  }    // to test the program  **public** **static** **void** main(String a[]) {    Q9 bst = **new** Q9();  bst.insert(7);  bst.insert(9);  bst.insert(4);  bst.insert(1);  bst.insert(5);  bst.insert(3);  bst.insert(6);  bst.insert(0);  bst.insert(8);  //For spacing  System.***out***.println("");    System.***out***.println("In-order Traverse:");  bst.inOrderTraversal();    System.***out***.println("");    System.***out***.println("Post-order Traverse:");  bst.postOrderTraversal();    System.***out***.println("");    System.***out***.println("Pre-order Traverse:");  bst.preOrderTraversal();    }  }  **Output of Q9:**    **Week 6**  **Q10. Program to delete the node from the binary search tree.**  // q9 about binary search tree with traverse  //this class represent the nodes  **class** Node {    **private** Node left;  **private** Node right;  **private** Integer data;    **public** Node(Integer data) {  **this**.data = data;  }    **public** Node getLeft() {  **return** left;  }  **public** **void** setLeft(Node left) {  **this**.left = left;  }  **public** Node getRight() {  **return** right;  }  **public** **void** setRight(Node right) {  **this**.right = right;  }  **public** **void** setData(Integer data) {  **this**.data = data;  }    **public** Integer getData() {  **return** data;  }  }    **public** **class** Q10 {    **private** Node root;    **public** **boolean** isEmpty() {    **return** (**this**.root == **null**);  }    **public** **void** insert(Integer data) {    System.***out***.print("[input: "+data+"]");  **if**(root == **null**) {  **this**.root = **new** Node(data);  System.***out***.println(" -> inserted: "+data);  **return**;  }    insertNode(**this**.root, data);  System.***out***.print(" -> inserted: "+data);  System.***out***.println();  }    **private** Node insertNode(Node root, Integer data) {    Node tmpNode = **null**;  System.***out***.print(" ->"+root.getData());  **if**(root.getData() >= data) {  System.***out***.print(" [L]");  **if**(root.getLeft() == **null**) {  root.setLeft(**new** Node(data));  **return** root.getLeft();  } **else** {  tmpNode = root.getLeft();  }  } **else** {  System.***out***.print(" [R]");  **if**(root.getRight() == **null**) {  root.setRight(**new** Node(data));  **return** root.getRight();  } **else** {  tmpNode = root.getRight();  }  }    **return** insertNode(tmpNode, data);  }      **public** **void** delete(Integer data) {    deleteNode(**this**.root, data);  }    **private** Node deleteNode(Node root, Integer data) {    **if**(root == **null**) **return** root;    **if**(data < root.getData()) {  root.setLeft(deleteNode(root.getLeft(), data));  } **else** **if**(data > root.getData()) {  root.setRight(deleteNode(root.getRight(), data));  } **else** {  // node with no leaf nodes  **if**(root.getLeft() == **null** && root.getRight() == **null**) {  System.***out***.println("After deleting "+data+":");  **return** **null**;  } **else** **if**(root.getLeft() == **null**) {  // node with one node (no left node)  System.***out***.println("After deleting "+data+":");  **return** root.getRight();  } **else** **if**(root.getRight() == **null**) {  // node with one node (no right node)  System.***out***.println("After deleting "+data+":");  **return** root.getLeft();  } **else** {  // nodes with two nodes  // search for min number in right sub tree  Integer minValue = minValue(root.getRight());  root.setData(minValue);  root.setRight(deleteNode(root.getRight(), minValue));  System.***out***.println("After deleting "+data+":");  }  }    **return** root;  }  // this method is to find the minimum number after the nodes been deleted  **private** Integer minValue(Node node) {    **if**(node.getLeft() != **null**) {  **return** minValue(node.getLeft());  }  **return** node.getData();  }    **public** **void** inOrderTraversal() {  doInOrder(**this**.root);  }    //implementing inorder method  **private** **void** doInOrder(Node root) {    **if**(root == **null**) **return**;  doInOrder(root.getLeft());  System.***out***.print(root.getData()+" ");  doInOrder(root.getRight());  }  // to test the program  **public** **static** **void** main(String a[]) {    Q10 bst = **new** Q10();  // to inset data into the node  bst.insert(7);  bst.insert(9);  bst.insert(4);  bst.insert(1);  bst.insert(5);  bst.insert(3);  bst.insert(6);  bst.insert(0);  bst.insert(8);  //For spacing  System.***out***.println("");  // to print output in ascending order  System.***out***.println("In-order Traverse:");  bst.inOrderTraversal();    System.***out***.println();  //deleting the node  bst.delete(9);  bst.inOrderTraversal();  }  }  **Output of Q10:**    **Week 7**  **Q11. Merge sort algorithm**  **import** java.util.Arrays;  //q11 about sorting  **public** **class** Q11  {  // using the merge sorting method  **public** **static** **void** merge(Integer[] a,  **int** iLeft, **int** iMiddle, **int** iRight,  Integer[] tmp)  {  **int** i, j, k;  i = iLeft;  j = iMiddle;  k = iLeft;  **while** ( i < iMiddle || j < iRight )  {  **if** ( i < iMiddle && j < iRight )  { // Both array have elements  **if** ( a[i] < a[j] )  tmp[k++] = a[i++];  **else**  tmp[k++] = a[j++];  }  **else** **if** ( i == iMiddle )  tmp[k++] = a[j++]; // a is empty  **else** **if** ( j == iRight )  tmp[k++] = a[i++]; // b is empty  }  }  **public** **static** **void** sort(Integer[] a, Integer[] tmp)  {  **int** width;  **for** ( width = 1; width < a.length; width = 2\*width )  {  // Combine sections of array a of width "width"  **int** i;  **for** ( i = 0; i < a.length; i = i + 2\*width )  {  **int** left, middle, right;  left = i;  middle = i + width;  right = i + 2\*width;  *merge*( a, left, middle, right, tmp );  }  // Copy tmp[ ] back to a[ ] for next iteration    **for** ( i = 0; i < a.length; i++ )  a[i] = tmp[i];  System.***out***.println("Steps: " + Arrays.*toString*(a) );  }  }  // run the program  **public** **static** **void** main( String[] args )  {  Integer[] x = {99, 22, 11,3} ;  Integer[] help = **new** Integer[x.length];  System.***out***.println("Before sorting: " + Arrays.*toString*(x) );  //for spacing  System.***out***.println("");  Q11.*sort*( x, help ); // Merge sort  System.***out***.println("\nAfter sorting: " + Arrays.*toString*(x) );  }  }  **Output of Q11:**    **Q12. Program to sort sequence with quick sorting method**  **import** java.util.Arrays;  //Q12 is about using the quick sort method  **public** **class** Q12 {  **public** **static** **void** main(String[] args) {  **int** input[] = **new** **int**[] {99, 22, 3, 11};  *sort*(input, 0, input.length - 1);    System.***out***.println("After sorting: " + Arrays.*toString*(input));  }  **private** **static** **void** sort(**int**[] array, **int** left, **int** right) {  **if** (left >= right) {  **return**;  }  //breakdown each part  System.***out***.print("Partition: " + *printArray*(array, left, right));  **int** pindex = *partition*(array, left, right);  System.***out***.println("\nSorted: " + *printArray*(array, left, right) + "\n");  // Sort left portion - recursive call  *sort*(array, left, pindex - 1);    // Sort right portion - recursive call  *sort*(array, pindex, right);  }    **private** **static** **int** partition(**int**[] arr, **int** left, **int** right) {  // Consider middle index as pivot  **int** pivot = arr[(left + right) / 2];    System.***out***.print(" Pivot : "+ pivot);  **while** (left <= right) {  // find a number which is greater than pivot  **while** (arr[left] < pivot) {  left++;  }  // find a number which is less than pivot  **while** (arr[right] > pivot) {  right--;  }  /\*  \* Swap the left and right numbers  \* cause pivot left side numbers should be less than pivot and right side numbers should be greater than pivot  \*/  **if** (left <= right) {  **if** (left != right) {  System.***out***.print(" => Swap : " + arr[left] + ", " + arr[right]);  **int** temp = arr[left];  arr[left] = arr[right];  arr[right] = temp;  }  left++;  right--;  }  }  // this is partition index - actually pivot index  **return** left;  }  // Utility method organize the output well  **private** **static** String printArray(**int**[] arr, **int** sindex, **int** eindex) {  String s = "";    **for** (**int** i = 0; i < arr.length; i++) {    **if**(i == sindex) s = s + " {";  **if**(!s.equals("")) s = s + " ";  s = s + arr[i];  **if**(i == eindex) s = s + " }";    }  **return** s;  }  }  **Output of Q12:**    **Week 8**  **Q13. Implementation of BFS traversal graph**  **import** java.util.\*;  //q13 implementation of the BFS traversal graph  //implementation of the graph  **public** **class** Q13  {  **public** **void** bfs(Node root)  {  //Since queue is a interface  Queue<Node> queue = **new** LinkedList<Node>();  **if**(root == **null**) **return**;  root.state = State.***Visited***;  //Adds to end of queue  queue.add(root);  **while**(!queue.isEmpty())  {  //removes from front of queue  Node r = queue.remove();  System.***out***.print(r.getVertex() + "\t");  //Visit child first before grandchild  **for**(Node n: r.getChild())  {  **if**(n.state == State.***Unvisited***)  {  queue.add(n);  n.state = State.***Visited***;  }  }  }  }  // create graph  **public** **static** Graph createNewGraph()  {  Graph g = **new** Graph();  Node[] temp = **new** Node[8];  temp[0] = **new** Node("A", 3);  temp[1] = **new** Node("B", 3);  temp[2] = **new** Node("C", 1);  temp[3] = **new** Node("D", 1);  temp[4] = **new** Node("E", 1);  temp[5] = **new** Node("F", 1);  temp[0].addChildNode(temp[1]);  temp[0].addChildNode(temp[2]);  temp[0].addChildNode(temp[3]);  temp[1].addChildNode(temp[0]);  temp[1].addChildNode(temp[4]);  temp[1].addChildNode(temp[5]);  temp[2].addChildNode(temp[0]);  temp[3].addChildNode(temp[0]);  temp[4].addChildNode(temp[1]);  temp[5].addChildNode(temp[1]);  **for** (**int** i = 0; i < 7; i++)  {  g.addNode(temp[i]);  }  **return** g;  }  // to test the answer  **public** **static** **void** main(String[] args) {    Q13 s = **new** Q13();    Graph gBfs = *createNewGraph*();  System.***out***.println("BFS Traversal for this graph: ");  System.***out***.println("");  s.bfs(gBfs.getNode()[0]);  }  }  // this class representing a simple graph with vertices  **public** **class** Graph {  **public** **int** count; // num of vertices  **private** Node vertices[];  **public** Graph()  {  vertices = **new** Node[8];  count = 0;  }  **public** **void** addNode(Node n)  {  **if**(count < 10)  {  vertices[count] = n;  count++;  }  **else**  {  System.***out***.println("graph full");  }  }  **public** Node[] getNode()  {  **return** vertices;  }  }  // this class is made to get vertex from the node  **public** **class** Node {  **public** Node[] child;  **public** **int** childCount;  **private** String vertex;  **public** State state;  **public** Node(String vertex)  {  **this**.vertex = vertex;  }  **public** Node(String vertex, **int** childlen)  {  **this**.vertex = vertex;  childCount = 0;  child = **new** Node[childlen];  }  **public** **void** addChildNode(Node adj)  {  adj.state = State.***Unvisited***;  **if**(childCount < 30)  {  **this**.child[childCount] = adj;  childCount++;  }  }  **public** Node[] getChild()  {  **return** child;  }  **public** String getVertex()  {  **return** vertex;  }  }  // this class is made to check whether the node is visited or not  **public** **enum** State {  ***Unvisited***,***Visiting***,***Visited***;  }  **Output of Q13:** |