**Linked List Algorithm Questions from Programming/Coding Interviews**

Contents

[1. Add Element at Specific position 11](#_Toc20259062)

[2. create clone of a LinkedList 12](#_Toc20259063)

[3. Doubly Linked List At First 13](#_Toc20259064)

[4. Given an element, how do you find out whether that element exist in a LinkedList or not. If it exists retrieve the position of that element. 17](#_Toc20259065)

[5. Find Circular Singly Linked List Intersection Point 18](#_Toc20259066)

[} 22](#_Toc20259067)

[6. Generic Single Linked List Insert Delete First 22](#_Toc20259068)

[} 25](#_Toc20259069)

[7. get the position of last occurrence of a given element in a LinkedList 25](#_Toc20259070)

[8. Get Number Of Elements 26](#_Toc20259071)

[} 27](#_Toc20259072)

[9. In Order Traversal Of Binary Tree 27](#_Toc20259073)

[/\* Output 27](#_Toc20259074)

[In order Traversal of Binary Tree in java : 27](#_Toc20259075)

[4 2 5 1 6 3 7 27](#_Toc20259076)

[\*/ 27](#_Toc20259077)

[/\*1. What happens in InOrder traversal of Binary Tree in java > 27](#_Toc20259078)

[In InOrder traversal of Binary Tree in java first left subtree is traversed then node is processed and then right subtree is traversed. 27](#_Toc20259079)

[2. What are Steps in InOrder Traversal of Binary Tree in java > 27](#_Toc20259080)

[Steps in InOrder Traversal of Binary Tree in java - 27](#_Toc20259081)

[Traverse the left subtree 27](#_Toc20259082)

[Visit the node. 27](#_Toc20259083)

[Traverse the right subtree.\*/ 27](#_Toc20259084)

[public class InOrderTraversalOfBinaryTree { 27](#_Toc20259085)

[public static void main(String[] args) { 27](#_Toc20259086)

[InOrderTraversalOfBinaryTree inOrderTraversalOfBinaryTree = new InOrderTraversalOfBinaryTree(); 27](#_Toc20259087)

[// Now, let's start creating nodes of Binary tree 27](#_Toc20259088)

[// create node1 (which is ROOT node) 27](#_Toc20259089)

[BinaryTreeNode binaryTreeNode1 = new BinaryTreeNode(1); 27](#_Toc20259090)

[// Left node of node1 27](#_Toc20259091)

[BinaryTreeNode node2 = new BinaryTreeNode(2); 27](#_Toc20259092)

[binaryTreeNode1.leftNode = node2; 27](#_Toc20259093)

[// Right node of node1 27](#_Toc20259094)

[BinaryTreeNode node3 = new BinaryTreeNode(3); 27](#_Toc20259095)

[binaryTreeNode1.rightNode = node3; 27](#_Toc20259096)

[// Left node of node2 27](#_Toc20259097)

[BinaryTreeNode node4 = new BinaryTreeNode(4); 27](#_Toc20259098)

[node2.leftNode = node4; 27](#_Toc20259099)

[// Right node of node2 27](#_Toc20259100)

[BinaryTreeNode node5 = new BinaryTreeNode(5); 27](#_Toc20259101)

[node2.rightNode = node5; 27](#_Toc20259102)

[// Left node of node3 27](#_Toc20259103)

[BinaryTreeNode node6 = new BinaryTreeNode(6); 27](#_Toc20259104)

[node3.leftNode = node6; 27](#_Toc20259105)

[// Right node of node3 27](#_Toc20259106)

[BinaryTreeNode node7 = new BinaryTreeNode(7); 27](#_Toc20259107)

[node3.rightNode = node7; 27](#_Toc20259108)

[System.out.println("In order Traversal of Binary Tree in java : "); 27](#_Toc20259109)

[// Pass root node to in order traversal method 27](#_Toc20259110)

[inOrderTraversalOfBinaryTree.inorderTraversalMethod(binaryTreeNode1); 27](#_Toc20259111)

[} 27](#_Toc20259112)

[public static class BinaryTreeNode { 28](#_Toc20259113)

[// Left and right node of Binary tree 28](#_Toc20259114)

[BinaryTreeNode leftNode; 28](#_Toc20259115)

[BinaryTreeNode rightNode; 28](#_Toc20259116)

[// Data of BinaryTree 28](#_Toc20259117)

[int data; 28](#_Toc20259118)

[BinaryTreeNode(int data) { 28](#_Toc20259119)

[this.data = data; 28](#_Toc20259120)

[} 28](#_Toc20259121)

[} 28](#_Toc20259122)

[public void inorderTraversalMethod(BinaryTreeNode binaryTreeNode) { 28](#_Toc20259123)

[// Continue if binaryTreeNode is not null 28](#_Toc20259124)

[if (binaryTreeNode != null) { 28](#_Toc20259125)

[// pass leftNode of binary tree recursively 28](#_Toc20259126)

[inorderTraversalMethod(binaryTreeNode.leftNode); 28](#_Toc20259127)

[// Display data of current binaryTreeNode. 28](#_Toc20259128)

[System.out.print(binaryTreeNode.data + " "); 28](#_Toc20259129)

[// pass rightNode of binary tree recursively 28](#_Toc20259130)

[inorderTraversalMethod(binaryTreeNode.rightNode); 28](#_Toc20259131)

[} 28](#_Toc20259132)

[} 28](#_Toc20259133)

[} 28](#_Toc20259134)

[10. How do you insert an element at the head and tail of a LinkedList 28](#_Toc20259135)

[11. Intersection\_of\_two\_Linked\_Lists\_Hashing 28](#_Toc20259136)

[12. Intersection\_of\_two\_Linked\_Lists\_simple 30](#_Toc20259137)

[13. join an ArrayList at the end of a LinkedList 32](#_Toc20259138)

[14. Linked\_List\_get\_getFirst\_getLast 32](#_Toc20259139)

[15. Linked\_List\_indexOf\_and\_lastIndexof 33](#_Toc20259140)

[16. Linked\_List\_offer\_offerFirst\_offerLast 34](#_Toc20259141)

[17. Linked\_List\_peek\_peekFirst\_peekLast 35](#_Toc20259142)

[18. Linked\_List\_poll\_pollFirst\_pollLast 36](#_Toc20259143)

[/\*1. poll() : This method retrieves and removes the head (first element) of this list. 36](#_Toc20259144)

[This function not only returns deletes the first element, but also displays them while being deleted 37](#_Toc20259145)

[Declaration : public E poll() 37](#_Toc20259146)

[Return Value : his method returns the first element of this list, or null if this list is empty.\*/ 37](#_Toc20259147)

[/\*2.pollFirst() : This method retrieves and removes the first element of this list, or returns null if this list is empty. 37](#_Toc20259148)

[Declaration : public E pollFirst() 37](#_Toc20259149)

[Return Value : This method returns the first element of this list, or null if this list is empty\*/ 37](#_Toc20259150)

[/\*3. pollLast() : This method retrieves and removes the last element of this list, or returns null if this list is empty. 37](#_Toc20259151)

[Declaration : public E pollLast() 37](#_Toc20259152)

[Return Value : This method returns the last element of this list, or null if this list is empty.\*/ 37](#_Toc20259153)

[public class Linked\_List\_poll\_pollFirst\_pollLast { 37](#_Toc20259154)

[public static void main(String[] args) { 37](#_Toc20259155)

[LinkedList list = new LinkedList(); 37](#_Toc20259156)

[// adding elements 37](#_Toc20259157)

[list.add("Geeks"); 37](#_Toc20259158)

[list.add(4); 37](#_Toc20259159)

[list.add("Geeks"); 37](#_Toc20259160)

[list.add(8); 37](#_Toc20259161)

[// printing the list 37](#_Toc20259162)

[System.out.println("The initial Linked List is : " + list); //[Geeks, 4, Geeks, 8] 37](#_Toc20259163)

[// using poll() to retrieve and remove the head 37](#_Toc20259164)

[// removes and displays "Geeks" 37](#_Toc20259165)

[System.out.println("Head element of the list is : " + list.poll()); //Geeks 37](#_Toc20259166)

[// removes and displays "Geeks" 37](#_Toc20259167)

[System.out.println("Head element of the list is : " + list.pollFirst()); //4 37](#_Toc20259168)

[// removes and displays 8 37](#_Toc20259169)

[System.out.println("Tail element of the list is : " + list.pollLast()); //8 37](#_Toc20259170)

[// printing the resultant list 37](#_Toc20259171)

[System.out.println("Linked List after removal using poll() : " + list); //[Geeks] 37](#_Toc20259172)

[} 37](#_Toc20259173)

[} 37](#_Toc20259174)

[19. program which implements LinkedList as a Queue (FIFO) 37](#_Toc20259175)

[20. LinkedListAsStack 38](#_Toc20259176)

[21. LinkedListCircularCalculateNodes 38](#_Toc20259177)

[} 41](#_Toc20259178)

[22. Java program to traverse the elements of a LinkedList in reverse direction? 41](#_Toc20259179)

[23. LinkedListIntersectionPointExample 41](#_Toc20259180)

[//Lists are merging at :19 41](#_Toc20259181)

[import java.util.Iterator; 41](#_Toc20259182)

[import java.util.LinkedList; 41](#_Toc20259183)

[import java.util.List; 41](#_Toc20259184)

[public class LinkedListIntersectionPointExample { 41](#_Toc20259185)

[public static void main(String[] args) { 42](#_Toc20259186)

[List<Integer> l1 = new LinkedList<Integer>(); 42](#_Toc20259187)

[List<Integer> l2 = new LinkedList<Integer>(); 42](#_Toc20259188)

[l1.add(11); 42](#_Toc20259189)

[l1.add(13); 42](#_Toc20259190)

[l1.add(16); 42](#_Toc20259191)

[l1.add(19); 42](#_Toc20259192)

[l1.add(22); 42](#_Toc20259193)

[l1.add(23); 42](#_Toc20259194)

[l2.add(12); 42](#_Toc20259195)

[l2.add(14); 42](#_Toc20259196)

[l2.add(19); 42](#_Toc20259197)

[l2.add(22); 42](#_Toc20259198)

[l2.add(23); 42](#_Toc20259199)

[twoListsAreMergingOrNot(l1, l2); 42](#_Toc20259200)

[} 42](#_Toc20259201)

[public static void twoListsAreMergingOrNot(List<Integer> l1, List<Integer> l2) { 42](#_Toc20259202)

[List<Integer> smallLinkedList = l1; 42](#_Toc20259203)

[List<Integer> largeLinkedList = l2; 42](#_Toc20259204)

[Integer diffInSize = l2.size() - l1.size(); 42](#_Toc20259205)

[int ctr = 0; 42](#_Toc20259206)

[if (diffInSize < 0) { // if difference is negative, swap the references 42](#_Toc20259207)

[// of lists. 42](#_Toc20259208)

[smallLinkedList = l2; 42](#_Toc20259209)

[largeLinkedList = l1; 42](#_Toc20259210)

[diffInSize = Math.abs(diffInSize); 42](#_Toc20259211)

[} 42](#_Toc20259212)

[Iterator<Integer> smallListIterator = smallLinkedList.iterator(); 42](#_Toc20259213)

[Iterator<Integer> largeListIterator = largeLinkedList.iterator(); 42](#_Toc20259214)

[while (largeListIterator.hasNext()) { 42](#_Toc20259215)

[int listValue = largeListIterator.next(); 42](#_Toc20259216)

[if (ctr < diffInSize) { 42](#_Toc20259217)

[ctr++; 42](#_Toc20259218)

[continue; 42](#_Toc20259219)

[} 42](#_Toc20259220)

[if (listValue == smallListIterator.next()) { 42](#_Toc20259221)

[System.out.println("Lists are merging at :" + listValue); 42](#_Toc20259222)

[return; 42](#_Toc20259223)

[} 42](#_Toc20259224)

[} 42](#_Toc20259225)

[} 42](#_Toc20259226)

[} 42](#_Toc20259227)

[24. MergeSortForLinkedLists 42](#_Toc20259228)

[25. PostOrderTraversalOfBinaryTree 44](#_Toc20259229)

[/\*1. What happens in PostOrder traversal of Binary Tree in java > 44](#_Toc20259230)

[In PostOrder traversal of Binary Tree in java each node is processed after traversing its subtree. 44](#_Toc20259231)

[2. What are Steps in PostOrder Traversal of Binary Tree in java > 44](#_Toc20259232)

[Steps in PostOrder Traversal of Binary Tree in java - 44](#_Toc20259233)

[traverse left subtree 44](#_Toc20259234)

[traverse the right subtree. 44](#_Toc20259235)

[Visit the node.\*/ 44](#_Toc20259236)

[/\* Output 44](#_Toc20259237)

[Post order Traversal of Binary Tree in java : 44](#_Toc20259238)

[4 5 2 6 7 3 1 44](#_Toc20259239)

[\*/ 44](#_Toc20259240)

[public class PostOrderTraversalOfBinaryTree { 44](#_Toc20259241)

[public static void main(String[] args) { 44](#_Toc20259242)

[PostOrderTraversalOfBinaryTree postOrderTraversalOfBinaryTree = new PostOrderTraversalOfBinaryTree(); 44](#_Toc20259243)

[// Now, let's start creating nodes of Binary tree 44](#_Toc20259244)

[// create node1 (which is ROOT node) 44](#_Toc20259245)

[BinaryTreeNode binaryTreeNode1 = new BinaryTreeNode(1); 45](#_Toc20259246)

[// Left node of node1 45](#_Toc20259247)

[BinaryTreeNode node2 = new BinaryTreeNode(2); 45](#_Toc20259248)

[binaryTreeNode1.leftNode = node2; 45](#_Toc20259249)

[// Right node of node1 45](#_Toc20259250)

[BinaryTreeNode node3 = new BinaryTreeNode(3); 45](#_Toc20259251)

[binaryTreeNode1.rightNode = node3; 45](#_Toc20259252)

[// Left node of node2 45](#_Toc20259253)

[BinaryTreeNode node4 = new BinaryTreeNode(4); 45](#_Toc20259254)

[node2.leftNode = node4; 45](#_Toc20259255)

[// Right node of node2 45](#_Toc20259256)

[BinaryTreeNode node5 = new BinaryTreeNode(5); 45](#_Toc20259257)

[node2.rightNode = node5; 45](#_Toc20259258)

[// Left node of node3 45](#_Toc20259259)

[BinaryTreeNode node6 = new BinaryTreeNode(6); 45](#_Toc20259260)

[node3.leftNode = node6; 45](#_Toc20259261)

[// Right node of node3 45](#_Toc20259262)

[BinaryTreeNode node7 = new BinaryTreeNode(7); 45](#_Toc20259263)

[node3.rightNode = node7; 45](#_Toc20259264)

[System.out.println("Post order Traversal of Binary Tree in java : "); 45](#_Toc20259265)

[// Pass root node to Post order traversal method 45](#_Toc20259266)

[postOrderTraversalOfBinaryTree.postorderTraversalMethod(binaryTreeNode1); 45](#_Toc20259267)

[} 45](#_Toc20259268)

[public static class BinaryTreeNode { 45](#_Toc20259269)

[// Left and right node of Binary tree 45](#_Toc20259270)

[BinaryTreeNode leftNode; 45](#_Toc20259271)

[BinaryTreeNode rightNode; 45](#_Toc20259272)

[// Data of BinaryTree 45](#_Toc20259273)

[int data; 45](#_Toc20259274)

[BinaryTreeNode(int data) { 45](#_Toc20259275)

[this.data = data; 45](#_Toc20259276)

[} 45](#_Toc20259277)

[} 45](#_Toc20259278)

[public void postorderTraversalMethod(BinaryTreeNode binaryTreeNode) { 45](#_Toc20259279)

[// Continue if binaryTreeNode is not null 45](#_Toc20259280)

[if (binaryTreeNode != null) { // Visit the node-Printing the node data 45](#_Toc20259281)

[// pass leftNode of binary tree recursively 45](#_Toc20259282)

[postorderTraversalMethod(binaryTreeNode.leftNode); 45](#_Toc20259283)

[// pass rightNode of binary tree recursively 45](#_Toc20259284)

[postorderTraversalMethod(binaryTreeNode.rightNode); 45](#_Toc20259285)

[// Display data of current binaryTreeNode. 45](#_Toc20259286)

[System.out.print(binaryTreeNode.data + " "); 45](#_Toc20259287)

[} 45](#_Toc20259288)

[} 45](#_Toc20259289)

[} 46](#_Toc20259290)

[26. PreOrderTraversalOfBinaryTree 46](#_Toc20259291)

[/\* Output 46](#_Toc20259292)

[Pre order Traversal of Binary Tree in java : 46](#_Toc20259293)

[1 2 4 5 3 6 7 46](#_Toc20259294)

[\*/ 46](#_Toc20259295)

[/\*1. What happens in PreOrder traversal of Binary Tree in java > 46](#_Toc20259296)

[In PreOrder traversal of Binary Tree in java each node is processed before traversing its subtree. 46](#_Toc20259297)

[2. What are Steps in PreOrder Traversal of Binary Tree in java > 46](#_Toc20259298)

[Steps in PreOrder Traversal of Binary Tree in java - 46](#_Toc20259299)

[Visit the node. 46](#_Toc20259300)

[Traverse the left subtree. 46](#_Toc20259301)

[Traverse the right subtree.\*/ 46](#_Toc20259302)

[public class PreOrderTraversalOfBinaryTree { 46](#_Toc20259303)

[public static void main(String[] args) { 46](#_Toc20259304)

[PreOrderTraversalOfBinaryTree preOrderTraversalOfBinaryTree = new PreOrderTraversalOfBinaryTree(); 46](#_Toc20259305)

[// Now, let's start creating nodes of Binary tree 46](#_Toc20259306)

[// create node1 (which is ROOT node) 46](#_Toc20259307)

[BinaryTreeNode binaryTreeNode1 = new BinaryTreeNode(1); 46](#_Toc20259308)

[// Left node of node1 46](#_Toc20259309)

[BinaryTreeNode node2 = new BinaryTreeNode(2); 46](#_Toc20259310)

[binaryTreeNode1.leftNode = node2; 46](#_Toc20259311)

[// Right node of node1 46](#_Toc20259312)

[BinaryTreeNode node3 = new BinaryTreeNode(3); 46](#_Toc20259313)

[binaryTreeNode1.rightNode = node3; 46](#_Toc20259314)

[// Left node of node2 46](#_Toc20259315)

[BinaryTreeNode node4 = new BinaryTreeNode(4); 46](#_Toc20259316)

[node2.leftNode = node4; 46](#_Toc20259317)

[// Right node of node2 46](#_Toc20259318)

[BinaryTreeNode node5 = new BinaryTreeNode(5); 46](#_Toc20259319)

[node2.rightNode = node5; 46](#_Toc20259320)

[// Left node of node3 46](#_Toc20259321)

[BinaryTreeNode node6 = new BinaryTreeNode(6); 46](#_Toc20259322)

[node3.leftNode = node6; 46](#_Toc20259323)

[// Right node of node3 46](#_Toc20259324)

[BinaryTreeNode node7 = new BinaryTreeNode(7); 46](#_Toc20259325)

[node3.rightNode = node7; 46](#_Toc20259326)

[System.out.println("Pre order Traversal of Binary Tree in java : "); 46](#_Toc20259327)

[// Pass root node to pre order traversal method 46](#_Toc20259328)

[preOrderTraversalOfBinaryTree.preorderTraversalMethod(binaryTreeNode1); 46](#_Toc20259329)

[} 46](#_Toc20259330)

[public static class BinaryTreeNode { 47](#_Toc20259331)

[// Left and right node of Binary tree 47](#_Toc20259332)

[BinaryTreeNode leftNode; 47](#_Toc20259333)

[BinaryTreeNode rightNode; 47](#_Toc20259334)

[// Data of BinaryTree 47](#_Toc20259335)

[int data; 47](#_Toc20259336)

[BinaryTreeNode(int data) { 47](#_Toc20259337)

[this.data = data; 47](#_Toc20259338)

[} 47](#_Toc20259339)

[} 47](#_Toc20259340)

[public void preorderTraversalMethod(BinaryTreeNode binaryTreeNode) { 47](#_Toc20259341)

[// Continue if binaryTreeNode is not null 47](#_Toc20259342)

[if (binaryTreeNode != null) { 47](#_Toc20259343)

[// Display data of current binaryTreeNode. 47](#_Toc20259344)

[System.out.print(binaryTreeNode.data + " "); 47](#_Toc20259345)

[// pass leftNode of binary tree recursively 47](#_Toc20259346)

[preorderTraversalMethod(binaryTreeNode.leftNode); 47](#_Toc20259347)

[// pass rightNode of binary tree recursively 47](#_Toc20259348)

[preorderTraversalMethod(binaryTreeNode.rightNode); 47](#_Toc20259349)

[} 47](#_Toc20259350)

[} 47](#_Toc20259351)

[} 47](#_Toc20259352)

[27. QueueImplementationUsingLinkedList 47](#_Toc20259353)

[\*/ 51](#_Toc20259354)

[28. remove the first occurrence and last occurrence of a given element in a LinkedList 51](#_Toc20259355)

[29. remove all elements of a LinkedList 51](#_Toc20259356)

[30. remove the elements of a LinkedList from both the ends 52](#_Toc20259357)

[31. replace an element at a specific position of a LinkedList with the given element 52](#_Toc20259358)

[32. retrieve but not remove the elements of a LinkedList from both the ends 53](#_Toc20259359)

[33. retrieve and remove and only retrieve an element from specific position of a LinkedList 53](#_Toc20259360)

[34. ReverseSinglyLinkedList 54](#_Toc20259361)

[35. ReverseSinglyLinkedListIterativeRecursive 55](#_Toc20259362)

[36. ReversingSinglyLinkedList 57](#_Toc20259363)

[} 59](#_Toc20259364)

[37. Segregate\_Even\_And\_Odd\_Nodes 59](#_Toc20259365)

[38. SinglyLinkedListatFirst 61](#_Toc20259366)

[Displaying LinkedList [first--->last]: 59 21 11 \*/ 63](#_Toc20259367)

[39. SinglyLinkedListatLast 63](#_Toc20259368)

[Displaying LinkedList [first--->last]: 11 21 59 \*/ 66](#_Toc20259369)

[40. SinglyLinkedListCircularOrNot 66](#_Toc20259370)

[} 67](#_Toc20259371)

[41. SinglyLinkedListDeleteSpecificNode 68](#_Toc20259372)

[} 69](#_Toc20259373)

[42. SinglyLinkedListGenericExample 69](#_Toc20259374)

[} 70](#_Toc20259375)

[43. SinglyLinkedListInsertNodeInBetween 70](#_Toc20259376)

[} 72](#_Toc20259377)

[44. SinglyLinkedListIsPalindrome\_ByReversing 72](#_Toc20259378)

[} 74](#_Toc20259379)

[45. SinglyLinkedListIsPalindrome\_Recursion 74](#_Toc20259380)

[46. SortedDoublyLinkedList 75](#_Toc20259381)

[} 78](#_Toc20259382)

[47. SortLinkedListOf0sAnd1sAnd2s 78](#_Toc20259383)

[} 80](#_Toc20259384)

[48. StackExampleGeneric 80](#_Toc20259385)

[} 81](#_Toc20259386)

[49. StackImplementationUsingLinkedList 81](#_Toc20259387)

[\*/ 83](#_Toc20259388)

[50. Union\_of\_two\_Linked\_Lists\_Hashing 83](#_Toc20259389)

[51. Given two Linked Lists, create union and intersection lists that contain union and intersection of the elements present in the given lists 85](#_Toc20259390)

[52. How to find middle element of a singly linked list in one pass? 87](#_Toc20259391)

[53. How to check if linked list contains loop in Java? How to find the starting node of the loop 88](#_Toc20259392)

[54. How to reverse a linked list in Java? 90](#_Toc20259393)

[55. How to reverse a singly linked list without recursion in Java? 92](#_Toc20259394)

[56. How would you remove a node from a doubly linked list? 92](#_Toc20259395)

[57. Write a Program to convert a binary tree into a doubly linked list? 92](#_Toc20259396)

[58. How to remove duplicate nodes in an unsorted linked list? 92](#_Toc20259397)

[59. Write a recursive method to determine whether a linked list is sorted in descending order or not? If it's sorted then return true otherwise return false. 92](#_Toc20259398)

[60. How to find the length of a singly linked list in Java? 92](#_Toc20259399)

[61. Write code to print out the data stored in each node in a singly linked list? 92](#_Toc20259400)

[62. Write a Program to print a linked list in reverse order? E.g. Print linked list from tail to head? 92](#_Toc20259401)

1. Add Element at Specific position

|  |
| --- |
| **public** **class** AddElementAtSpecificPosition {  **public** **static** **void** main(String[] args) {  LinkedList<Integer> list = **new** LinkedList<Integer>();  list.add(10);  list.add(20);  list.add(30);  list.add(40);  System.***out***.println(list); // Output : [10, 20, 30, 40]  // Adding an element at index 2  list.add(2, 9999);  // Printing the elements of list  System.***out***.println(list); // Output : [10, 20, 9999, 30, 40]  // Creating another LinkedList with elements to add  LinkedList<Integer> list1 = **new** LinkedList<Integer>();  list1.addFirst(111);  list1.addFirst(222);  list1.addFirst(333);  // Printing the elements of list1  System.***out***.println(list1); // Output : [333, 222, 111]  // Adding all elements of list1 at index 3 of list  list.addAll(3, list1);  // Printing the elements of list  System.***out***.println(list); // Output : [10, 20, 9999, 333, 222, 111, 30, 40]  }  } |

1. create clone of a LinkedList

|  |
| --- |
| **public** **class** CloneOfLinkedList {  **public** **static** **void** main(String[] args) {  LinkedList<Integer> linkedList1 = **new** LinkedList<Integer>();  linkedList1.add(10);  linkedList1.add(20);  linkedList1.add(30);  linkedList1.add(40);  linkedList1.add(50);  System.***out***.println(linkedList1); // Output : [10, 20, 30, 40, 50]  // Creating another LinkedList  LinkedList<Integer> linkedList2 = **new** LinkedList<Integer>();  // Cloning the linkedList1 into linkedList2  linkedList2 = (LinkedList<Integer>) linkedList1.clone();  System.***out***.println(linkedList2); // Output : [10, 20, 30, 40, 50]  }  } |

1. Doubly Linked List At First

|  |
| --- |
| /\*Displaying in forward direction [first--->last] : 39 14 59 21 11  Displaying in backward direction [last-->first] : 11 21 59 14 39  Deleted Nodes: 39 14  Displaying in forward direction [first--->last] : 59 21 11  Displaying in backward direction [last-->first] : 11 21 59 \*/  **public** **class** DoublyLinkedListAtFirst {  **public** **static** **void** main(String[] args) {  LinkedList4 linkedList = **new** LinkedList4(); // creation of Linked List    linkedList.insertFirst(11);  linkedList.insertFirst(21);  linkedList.insertFirst(59);  linkedList.insertFirst(14);  linkedList.insertFirst(39);  linkedList.displayFrwd(); // display DoublyLinkedList  linkedList.displayBckwrd();    System.***out***.print("Deleted Nodes: ");  Node4 deletedNode = linkedList.deleteFirst(); //delete Node  deletedNode.displayNode(); //display deleted Node.  deletedNode = linkedList.deleteFirst(); //delete Node.  deletedNode.displayNode(); //display deleted Node.    System.***out***.println();// sysout used to format output    linkedList.displayFrwd(); // display DoublyLinkedList  linkedList.displayBckwrd();  }  }  **class** LinkedList4 {  **private** Node4 first; // ref to first link on LinkedList  **private** Node4 last; // ref to last link on LinkedList    /\*\* Doubly LinkedList constructor \*/  **public** LinkedList4(){  first = **null**;  }      /\*\* Insert New Node at first position of Doubly LinkedList \*/    **public** **void** insertFirst(**int** data){ // insert at front of list  Node4 newNode = **new** Node4(data); // creation of new node.  **if** (first == **null**) // means LinkedList is empty.  last = newNode; // newNode <--- last  **else**  first.previous = newNode; // newNode <-- old first  newNode.next = first; // newNode --> old first  first = newNode; // first --> newNode  }    /\*\* Delete first node of Doubly linkedList. \*/  **public** Node4 deleteFirst() {    **if**(first==**null**){ //means LinkedList in empty, throw exception.  **throw** **new** LinkedListEmptyException("LinkedList doesn't contain any Nodes.");  }  Node4 tempNode = first;  **if** (first.next == **null**) // if only one item  last = **null**; // null <-- last  **else**  first.next.previous = **null**; // null <-- old next  first = first.next; // first --> old next  **return** tempNode;  }      /\*\* Display Doubly LinkedList in forward direction \*/  **public** **void** displayFrwd() {  System.***out***.print("Displaying in forward direction [first--->last] : ");  Node4 tempDisplay = first; // start at the beginning of linkedList  **while** (tempDisplay != **null**){ // Executes until we don't find end of list.  tempDisplay.displayNode();  tempDisplay = tempDisplay.next; // move to next Node  }  System.***out***.println("");  }      /\*\* Display Doubly LinkedList in backward direction \*/  **public** **void** displayBckwrd() {  System.***out***.print("Displaying in backward direction [last-->first] : ");  Node4 tempDisplay = last; // start at the end of linkedList  **while** (tempDisplay != **null**) {// Executes until we don't find start of list.  tempDisplay.displayNode();  tempDisplay = tempDisplay.previous; // move to previous Node  }  System.***out***.println("");  }      }  **class** Node4 {  **public** **int** data; // data in Node4.  **public** Node4 next; // points to next Node4 in list.  **public** Node4 previous; // points to previous Node4 in list.    /\*\* Constructor \*/  **public** Node4(**int** data){  **this**.data = data;  }    /\*\* Display Node4's data \*/  **public** **void** displayNode() {  System.***out***.print( data + " ");  }  } |

1. Given an element, how do you find out whether that element exist in a LinkedList or not. If it exists retrieve the position of that element.

|  |
| --- |
| **public** **class** ElementExistInLinkedList {  **public** **static** **void** main(String[] args) {  LinkedList<String> list = **new** LinkedList<String>();  list.add("JAVA");  list.add("J2EE");  list.add("JSP");  list.add("SERVLETS");  list.add("JDBC");  System.***out***.println(list); // Output : [JAVA, J2EE, JSP, SERVLETS, JDBC]  String s = "JSP";  **boolean** contains = list.contains(s);  **if** (contains) {  System.***out***.println(list.indexOf(s)); // Output : 2  }  s = "STRUTS";  contains = list.contains("STRUTS");  **if** (contains) {  System.***out***.println(list.indexOf(s));  }  }  } |

1. Find Circular Singly Linked List Intersection Point

|  |
| --- |
| /\*LinkedList has been succesfully converted into CircularLinkedList  Displaying LinkedList [first--->last]: 11 22 33 44 55 66 33 44 55 66 33 44  LinkedList is circular at Node: 33\*/  **public** **class** FindCircularSinglyLinkedListInterscetionPointExample {  **public** **static** **void** main(String[] args) {  LinkedList8 linkedList=**new** LinkedList8();  linkedList.insert(66);  linkedList.insert(55);  linkedList.insert(44);  linkedList.insert(33);  linkedList.insert(22);  linkedList.insert(11);    linkedList.makeLinkedListCircular();  linkedList.displayLinkedList();    linkedList.findIntersectionPointOfCircularLikedList();  }  }  **class** LinkedList8{    Node first=**null**;  Node circularPoint1; //points used to make LinkedList circular.  Node circularPoint2;    Node slowPointer; //will step over LinkedList by 1 Node.  Node fastPointer; //will step over LinkedList by 2 Node.      /\*\*  \* Insert New Node at first position  \*/  **public** **void** insert(**int** data){    Node newNode=**new** Node(data);  newNode.next=first;  first=newNode;    /\*  \* Below we have kept track of two Nodes so that later we can make LinkedList circular(If required).  \* Note:- I have kept track of below two nodes just for demonstration purpose. You may provide some other implementation for making LinkedList circular.  \*/  **if**(data==33)  circularPoint1=newNode;  **if**(data==66)  circularPoint2=newNode;    }    /\*\*  \*This method makes LikedList circular- by making end Node point to some middle Node of LinkedList.  \*end Node--->middle Node.  \*/  **public** **void** makeLinkedListCircular(){  circularPoint2.next=circularPoint1;  System.***out***.println("LinkedList has been succesfully converted into CircularLinkedList");  }    /\*\*  \* method finds out intersection point of circular LikedList.  \*/  **public** **void** findIntersectionPointOfCircularLikedList(){  slowPointer=first;  fastPointer=first;    **while**( (slowPointer!=fastPointer || slowPointer==first) //when first time condition is checked slowPointer is equal to fastPointer -so that does not means LinkedList is circular and we exit while loop  //we got to be cautious by keeping extra check that whether we are on first node or not.  && fastPointer.next!=**null** //used to avoid NullPointerException(in case we are are on last Node- than next is null, calling further next on null will cause NPE.)  && fastPointer.next.next!=**null**){  slowPointer=slowPointer.next; // step over LinkedList by 1 Node.  fastPointer=fastPointer.next.next; // step over LinkedList by 2 Node.  }    //we will exit above while loop when we have detected LinkedList is circular      /\*make one of the pointer point to first, and let other pointer continue to point to same node.  \* Than make both pointers step over LinkedList by 1 Node, they will meet at intersection point.  \*/  slowPointer=first;  **while**(slowPointer!=fastPointer){  slowPointer=slowPointer.next; // step over LinkedList by 1 Node.  fastPointer=fastPointer.next; // step over LinkedList by 1 Node.  }  System.***out***.println("LinkedList is circular at Node: "+slowPointer.data);    }      /\*\*  \* Display LinkedList  \*/  **public** **void** displayLinkedList(){  Node tempDisplay=first;  **int** displayLimiterCtr=0; //as our LinkedList is circular it will keep on displaying nodes till infinity...  //so this variable will help us in limiting the display to specific count.  System.***out***.print("Displaying LinkedList [first--->last]: ");  **while**(tempDisplay!=**null**){  tempDisplay.displayNode();  tempDisplay=tempDisplay.next;  **if**(++displayLimiterCtr >= 12) //stops displaying after 12 Nodes.  **break**;  }  System.***out***.println();  }    } |

1. Generic Single Linked List Insert Delete First

|  |
| --- |
| /\*Displaying LinkedList [first--->last]: 39 14 59 21 11  Deleted Nodes: 39 14  Displaying LinkedList [first--->last]: 59 21 11 \*/  **public** **class** GenericSingleLinkedListInsertDeleteFirstExample {  **public** **static** **void** main(String[] args) {  LinkedList10<Integer> linkedList = **new** LinkedList10<Integer>(); // creation of Linked List    linkedList.insertFirst(11);  linkedList.insertFirst(21);  linkedList.insertFirst(59);  linkedList.insertFirst(14);  linkedList.insertFirst(39);  linkedList.displayLinkedList(); // display LinkedList    System.***out***.print("Deleted Nodes: ");  Node10<Integer> deletedNode = linkedList.deleteFirst(); //delete Node  deletedNode.displayNode(); //display deleted Node.  deletedNode = linkedList.deleteFirst(); //delete Node.  deletedNode.displayNode(); //display deleted Node.    System.***out***.println();// sysout used to format output  linkedList.displayLinkedList(); //Again display LinkedList  }  }  **class** LinkedList10<T> {  **private** Node10<T> first; // ref to first link on list    /\*\*  \* generic Single LinkedList constructor  \*/  **public** LinkedList10(){  first = **null**;  }    /\*\*  \* Insert New Node at first position of generic Single LinkedList  \*/  **public** **void** insertFirst(T data) {  Node10<T> newNode = **new** Node10<T>(data); //Creation of New Node.  newNode.next = first; //newLink ---> old first  first = newNode; //first ---> newNode  }    /\*\*  \* Deletes first Node of generic Single LinkedList  \*/  **public** Node10<T> deleteFirst()  {  **if**(first==**null**){ //means LinkedList in empty, throw exception.  **throw** **new** LinkedListEmptyException("LinkedList doesn't contain any Nodes.");  }  Node10<T> tempNode = first; // save reference to first Node in tempNode- so that we could return saved reference.  first = first.next; // delete first Node (make first point to second node)  **return** tempNode; // return tempNode (i.e. deleted Node)  }      /\*\*  \* Display generic Single LinkedList  \*/  **public** **void** displayLinkedList() {  System.***out***.print("Displaying LinkedList [first--->last]: ");  Node10<T> tempDisplay = first; // start at the beginning of linkedList  **while** (tempDisplay != **null**){ // Executes until we don't find end of list.  tempDisplay.displayNode();  tempDisplay = tempDisplay.next; // move to next Node  }  System.***out***.println();    }    }  **class** Node10<T> {  **public** T data; // data in Node.  **public** Node10<T> next; // points to next Node in list.    /\*\*  \* Constructor  \*/  **public** Node10(T data){  **this**.data = data;  }    /\*\*  \* Display Node's data  \*/  **public** **void** displayNode() {  System.***out***.print( data + " ");  }  } |

1. get the position of last occurrence of a given element in a LinkedList

|  |
| --- |
| **public** **class** Get\_position\_of\_last\_occurrence {  **public** **static** **void** main(String[] args) {  LinkedList<String> linkedList = **new** LinkedList<String>();  linkedList.add("AAA");  linkedList.add("BBB");  linkedList.add("CCC");  linkedList.add("BBB");  linkedList.add("FFF");  linkedList.add("BBB");  System.***out***.println(linkedList); // Output : [AAA, BBB, CCC, BBB, FFF, BBB]  // Getting the position of last occurrence of "BBB"  System.***out***.println(linkedList.lastIndexOf("BBB")); // Output : 5  }  } |

1. Get Number Of Elements

|  |
| --- |
| **public** **class** GetNumberOfElements {  **public** **static** **void** main(String[] args) {  LinkedList<String> list = **new** LinkedList<String>();  list.add("JAVA");  list.add("J2EE");  list.add("JSP");  list.add("SERVLETS");  list.add("JDBC");  System.***out***.println(list); // Output : [JAVA, J2EE, JSP, SERVLETS, JDBC]  // Getting the number of elements in list  System.***out***.println(list.size()); // Output : 5  }  } |

1. In Order Traversal Of Binary Tree

|  |
| --- |
| /\* Output  In order Traversal of Binary Tree in java :  4 2 5 1 6 3 7    \*/  /\*1. What happens in InOrder traversal of Binary Tree in java >  In InOrder traversal of Binary Tree in java first left subtree is traversed then node is processed and then right subtree is traversed.  2. What are Steps in InOrder Traversal of Binary Tree in java >  Steps in InOrder Traversal of Binary Tree in java -  Traverse the left subtree  Visit the node.  Traverse the right subtree.\*/  public class InOrderTraversalOfBinaryTree {  public static void main(String[] args) {  InOrderTraversalOfBinaryTree inOrderTraversalOfBinaryTree = new InOrderTraversalOfBinaryTree();  // Now, let's start creating nodes of Binary tree  // create node1 (which is ROOT node)  BinaryTreeNode binaryTreeNode1 = new BinaryTreeNode(1);  // Left node of node1  BinaryTreeNode node2 = new BinaryTreeNode(2);  binaryTreeNode1.leftNode = node2;  // Right node of node1  BinaryTreeNode node3 = new BinaryTreeNode(3);  binaryTreeNode1.rightNode = node3;  // Left node of node2  BinaryTreeNode node4 = new BinaryTreeNode(4);  node2.leftNode = node4;  // Right node of node2  BinaryTreeNode node5 = new BinaryTreeNode(5);  node2.rightNode = node5;  // Left node of node3  BinaryTreeNode node6 = new BinaryTreeNode(6);  node3.leftNode = node6;  // Right node of node3  BinaryTreeNode node7 = new BinaryTreeNode(7);  node3.rightNode = node7;  System.out.println("In order Traversal of Binary Tree in java : ");  // Pass root node to in order traversal method  inOrderTraversalOfBinaryTree.inorderTraversalMethod(binaryTreeNode1);  }  public static class BinaryTreeNode {  // Left and right node of Binary tree  BinaryTreeNode leftNode;  BinaryTreeNode rightNode;  // Data of BinaryTree  int data;  BinaryTreeNode(int data) {  this.data = data;  }  }  public void inorderTraversalMethod(BinaryTreeNode binaryTreeNode) {  // Continue if binaryTreeNode is not null  if (binaryTreeNode != null) {  // pass leftNode of binary tree recursively  inorderTraversalMethod(binaryTreeNode.leftNode);  // Display data of current binaryTreeNode.  System.out.print(binaryTreeNode.data + " ");  // pass rightNode of binary tree recursively  inorderTraversalMethod(binaryTreeNode.rightNode);  }  }  } |

1. How do you insert an element at the head and tail of a LinkedList

|  |
| --- |
| **public** **class** InsertAnElement {  **public** **static** **void** main(String[] args) {  LinkedList<Integer> list = **new** LinkedList<Integer>();  list.add(10);  list.addLast(20);  list.offer(30);  list.offerLast(40);  // Printing the elements of list  System.***out***.println(list); // Output : [10, 20, 30, 40]  // Adding elements at the beginning of the list  list.offerFirst(1);  list.addFirst(2);  // Printing the elements of list  System.***out***.println(list); // Output : [2, 1, 10, 20, 30, 40]  }  } |

1. Intersection\_of\_two\_Linked\_Lists\_Hashing

|  |
| --- |
| **public** **class** Intersection\_of\_two\_Linked\_Lists\_Hashing {  Node head; // head of list  **class** Node {  **int** data;  Node next;  Node(**int** d) {  data = d;  next = **null**;  }  }  **public** **static** **void** main(String[] args) {  Intersection\_of\_two\_Linked\_Lists\_Hashing llist1 = **new** Intersection\_of\_two\_Linked\_Lists\_Hashing();  Intersection\_of\_two\_Linked\_Lists\_Hashing llist2 = **new** Intersection\_of\_two\_Linked\_Lists\_Hashing();  Intersection\_of\_two\_Linked\_Lists\_Hashing intersection = **new** Intersection\_of\_two\_Linked\_Lists\_Hashing();  /\* create a linked lits 10->15->5->20 \*/  llist1.push(20);  llist1.push(4);  llist1.push(15);  llist1.push(10);  /\* create a linked lits 8->4->2->10 \*/  llist2.push(10);  llist2.push(2);  llist2.push(4);  llist2.push(8);  intersection = intersection.getIntersection(llist1.head, llist2.head);  System.***out***.println("First List is");  llist1.printList();  System.***out***.println("Second List is");  llist2.printList();  System.***out***.println("Intersection List is");  intersection.printList();  }  **void** push(**int** new\_data) {  Node new\_node = **new** Node(new\_data);  new\_node.next = head;  head = new\_node;  }  Intersection\_of\_two\_Linked\_Lists\_Hashing getIntersection(Node head1, Node head2) {  HashSet<Integer> hset = **new** HashSet<>();  Node n1 = head1;  Node n2 = head2;  Intersection\_of\_two\_Linked\_Lists\_Hashing result = **new** Intersection\_of\_two\_Linked\_Lists\_Hashing();  **while** (n1 != **null**) {  **if** (hset.contains(n1.data)) {  hset.add(n1.data);  } **else** {  hset.add(n1.data);  }  n1 = n1.next;  }  **while** (n2 != **null**) {  **if** (hset.contains(n2.data)) {  result.push(n2.data);  }  n2 = n2.next;  }  **return** result;  }  **void** printList() {  Node temp = head;  **while** (temp != **null**) {  System.***out***.print(temp.data + " ");  temp = temp.next;  }  System.***out***.println();  }  } |

1. Intersection\_of\_two\_Linked\_Lists\_simple

|  |
| --- |
| /\*Input:  List1: 10->15->4->20  lsit2: 8->4->2->10  Output:  Intersection List: 4->10  Time Complexity: O(mn)\*/  **public** **class** Intersection\_of\_two\_Linked\_Lists\_simple {  Node head; // head of list  **class** Node {  **int** data;  Node next;  Node(**int** d) {  data = d;  next = **null**;  }  }  **public** **static** **void** main(String[] args) {  Intersection\_of\_two\_Linked\_Lists\_simple llist1 = **new** Intersection\_of\_two\_Linked\_Lists\_simple();  Intersection\_of\_two\_Linked\_Lists\_simple llist2 = **new** Intersection\_of\_two\_Linked\_Lists\_simple();  Intersection\_of\_two\_Linked\_Lists\_simple intersecn = **new** Intersection\_of\_two\_Linked\_Lists\_simple();  /\* create a linked lits 10->15->5->20 \*/  llist1.push(20);  llist1.push(4);  llist1.push(15);  llist1.push(10);  /\* create a linked lits 8->4->2->10 \*/  llist2.push(10);  llist2.push(2);  llist2.push(4);  llist2.push(8);  intersecn.getIntersection(llist1.head, llist2.head);  System.***out***.println("First List is");  llist1.printList();  System.***out***.println("Second List is");  llist2.printList();  System.***out***.println("Intersection List is");  intersecn.printList();  }  **void** push(**int** new\_data) {  Node new\_node = **new** Node(new\_data);  new\_node.next = head;  head = new\_node;  }  **void** getIntersection(Node head1, Node head2) {  Node result = **null**;  Node t1 = head1;  **while** (t1 != **null**) {  **if** (isPresent(head2, t1.data))  push(t1.data);  t1 = t1.next;  }  }  **boolean** isPresent(Node head, **int** data) {  Node t = head;  **while** (t != **null**) {  **if** (t.data == data)  **return** **true**;  t = t.next;  }  **return** **false**;  }  **void** printList() {  Node temp = head;  **while** (temp != **null**) {  System.***out***.print(temp.data + " ");  temp = temp.next;  }  System.***out***.println();  }  } |

1. join an ArrayList at the end of a LinkedList

|  |
| --- |
| **public** **class** JoinArrayListAtEnd {  **public** **static** **void** main(String[] args) {  LinkedList<String> linkedList = **new** LinkedList<String>();  linkedList.add("ONE");  linkedList.add("TWO");  linkedList.add("THREE");  linkedList.add("FOUR");  linkedList.add("FIVE");  System.***out***.println(linkedList); // Output : [ONE, TWO, THREE, FOUR, FIVE]  ArrayList<String> arrayList = **new** ArrayList<String>();  arrayList.add("SIX");  arrayList.add("SEVEN");  arrayList.add("EIGHT");  arrayList.add("NINE");  System.***out***.println(arrayList); // Output : [SIX, SEVEN, EIGHT, NINE]  linkedList.addAll(arrayList);  System.***out***.println(linkedList); // Output : [ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE]  }  } |

1. Linked\_List\_get\_getFirst\_getLast

|  |
| --- |
| /\*1. get(int index) : This method returns the element at the specified position in this list.  Declaration : public E get(int index)  Parameters : index : index of the element to return  Return Value : This method returns the element at the specified position in this list  Exception : IndexOutOfBoundsException : if the index is out of range  2. getFirst() : This method returns the first element in this list.  Declaration : public E getFirst()  Return Value : This method returns the first element in this list  Exceptions : NoSuchElementException : if this list is empty  3. getLast() : This method returns the last element in this list.  Declaration : public E getLast()  Return Value : This method returns the last element in this list  Exceptions : NoSuchElementException : if this list is empty\*/  **public** **class** Linked\_List\_get\_getFirst\_getLast {  **public** **static** **void** main(String[] args) {  LinkedList list = **new** LinkedList();  // adding elements using add()  list.add("Geeks");  list.add(4);  list.add("Geeks");  list.add("8");  // printing the whole list  System.***out***.println("The elements in List are : " + list); //[Geeks, 4, Geeks, 8]  // using get() to print element at index 3 prints 8  System.***out***.println("Element at index 3 is : " + list.get(3)); //8  // using get() to print element at first index prints "Geeks"  System.***out***.println("Element at 1st index is : " + list.getFirst()); // Geeks  // using get() to print element at last index prints "8"  System.***out***.println("Element at last index is : " + list.getLast()); //8  }  } |

1. Linked\_List\_indexOf\_and\_lastIndexof

|  |
| --- |
| /\*1. indexOf(Object o) : This method returns the index of the first occurrence of the specified element in this list,  or -1 if this list does not contain the element.  Declaration : public int indexOf(Object o)  Parameters : o : element to search for  Return Value : This method returns the index of the first occurrence of the specified element in this list,  or -1 if this list does not contain the element.  2. lastIndexOf(Object o) : This method returns the index of the last occurrence of the specified element in this list,  or -1 if this list does not contain the element.  Declaration : public int lastIndexOf(Object o)  Parameters : o : element to search for  Return Value : This method returns the index of the last occurrence of the specified element in this list, or -1 if this  list does not contain the element\*/  **public** **class** Linked\_List\_indexOf\_and\_lastIndexof {  **public** **static** **void** main(String[] args) {  LinkedList list = **new** LinkedList();  // adding elements  list.add("Geeks");  list.add(4);  list.add("Geeks");  list.add(8);  // printing the initial list  System.***out***.println("The initial Linked List is : " + list); //[Geeks, 4, Geeks, 8]  // Retrieving index of 1st occurrence of "Geeks" Prints 0  System.***out***.println("Index of 1st occurrence of Geeks : " + list.indexOf("Geeks")); //0  // Retrieving index of last occurrence of "Astha" Prints -1 element not  // present  System.***out***.println("Index of last occurrence of Astha : " + list.lastIndexOf("Astha")); //-1  }  } |

1. Linked\_List\_offer\_offerFirst\_offerLast

|  |
| --- |
| /\*Linked list also has a function that does the work of flexible addition of elements  1. offer(E e) : This method adds the specified element as the tail (last element) of this list.  Declaration : public boolean offer(E e)  Parameters : e: the element to add  Return Value : This method returns true  2. offerFirst(E e) : This method inserts the specified element at the front of this list.  Declaration : public boolean offerFirst(E e)  Parameters : e : the element to add  Return Value : This method returns true  3. offerLast(E e) : This method inserts the specified element at the end of this list.  Declaration : public boolean offerLast(E e)  Parameters : e:the element to add  Return Value : This method returns true\*/  **public** **class** Linked\_List\_offer\_offerFirst\_offerLast {  **public** **static** **void** main(String[] args) {  LinkedList list = **new** LinkedList();  // adding elements  list.add("Geeks");  list.add(4);  list.add("Geeks");  list.add(8);  // printing the list  System.***out***.println("The initial Linked list is : " + list); //[Geeks, 4, Geeks, 8]  // offering a new element adds element at tail.  list.offer("Astha");  // offering a new element adds element at head.  list.offerFirst("Astha");  // offering a new element adds element at end.  list.offerLast("Astha");  // printing the new list  System.***out***.println("LinkedList after insertion using offer() : " + list); //[Astha, Geeks, 4, Geeks, 8, Astha, Astha]  }  } |

1. Linked\_List\_peek\_peekFirst\_peekLast

|  |
| --- |
| /\*Linked list class offers the functionality to “look into” the first and last elements of the list and hence can be  useful in cases where only the retrieval is required and not necessarily the deletion is required.  1. peek() : This method retrieves, but does not remove, the head (first element) of this list.  Declaration : public E peek()  Return Value : This method returns the head of this list, or null if this list is empty.  2. peekFirst() : This method retrieves, but does not remove, the first element of this list, or returns null if  this list is empty. This works similar to peek().  Declaration : public E peekFirst()  Return Value : This method returns the first element of this list, or null if this list is empty  3. peekLast() : This method retrieves, but does not remove, the last element of this list, or returns null if this list is empty.  Declaration : public E peekLast()  Return Value : This method returns the last element of this list, or null if this list is empty\*/  **public** **class** Linked\_List\_peek\_peekFirst\_peekLast {  **public** **static** **void** main(String[] args) {  LinkedList list = **new** LinkedList();  // adding elements  list.add("Geeks");  list.add(4);  list.add("Geeks");  list.add("8");  // printing the list  System.***out***.println("The initial list is :" + list); //[Geeks, 4, Geeks, 8]  // peek at the head of the list Prints 1st element, "Geeks"  System.***out***.println("Head of the list : " + list.peek()); //Geeks  // peek at the first element of the list Prints 1st element, "Geeks"  System.***out***.println("First element of the list is : " + list.peekFirst()); //Geeks  // peek at the last element of the list Prints last element, 8  System.***out***.println("Last element of the list is : " + list.peekLast()); //8  }  } |

1. Linked\_List\_poll\_pollFirst\_pollLast

|  |
| --- |
| /\* poll() : This method retrieves and removes the head (first element) of this list.This function not only returns deletes the first element, but also displays them while being deleted  Declaration : public E poll()  Return Value : his method returns the first element of this list, or null if this list is empty.\*/  /\*pollFirst() : This method retrieves and removes the first element of this list, or returns null if this list is empty.  Declaration : public E pollFirst()  Return Value : This method returns the first element of this list, or null if this list is empty\*/  /\*pollLast() : This method retrieves and removes the last element of this list, or returns null if this list is empty.  Declaration : public E pollLast()  Return Value : This method returns the last element of this list, or null if this list is empty.\*/  public class Linked\_List\_poll\_pollFirst\_pollLast {  public static void main(String[] args) {  LinkedList list = new LinkedList();  // adding elements  list.add("Geeks");  list.add(4);  list.add("Geeks");  list.add(8);  // printing the list  System.out.println("The initial Linked List is : " + list); //[Geeks, 4, Geeks, 8]  // using poll() to retrieve and remove the head  // removes and displays "Geeks"  System.out.println("Head element of the list is : " + list.poll()); //Geeks  // removes and displays "Geeks"  System.out.println("Head element of the list is : " + list.pollFirst()); //4  // removes and displays 8  System.out.println("Tail element of the list is : " + list.pollLast()); //8  // printing the resultant list  System.out.println("Linked List after removal using poll() : " + list); //[Geeks]  }  } |

1. program which implements LinkedList as a Queue (FIFO)

|  |
| --- |
| **public** **class** LinkedListAsQueue {  **public** **static** **void** main(String[] args) {  LinkedList<Integer> queue = **new** LinkedList<Integer>();  queue.offer(10);  queue.offer(20);  queue.offer(30);  queue.offer(40);  System.***out***.println(queue); // Output : [10, 20, 30, 40]  // Removing the elements from the queue  System.***out***.println(queue.poll()); // Output : 10  System.***out***.println(queue.poll()); // Output : 20  }  } |

1. LinkedListAsStack

|  |
| --- |
| **public** **class** LinkedListAsStack {  **public** **static** **void** main(String[] args) {  LinkedList<Integer> stack = **new** LinkedList<Integer>();  stack.push(10);  stack.push(20);  stack.push(30);  stack.push(40);  System.***out***.println(stack); // Output : [40, 30, 20, 10]  // Poping out the elements from the stack  System.***out***.println(stack.pop()); // Output : 40  System.***out***.println(stack.pop()); // Output : 30  }  } |

1. LinkedListCircularCalculateNodes

|  |
| --- |
| /\*LinkedList has been succesfully converted into CircularLinkedList  Displaying LinkedList [first--->last]: 11 22 33 44 55 66 33 44 55 66 33 44  LinkedList is circular at Node: 33  Total number of nodes in LinkedList are: 6\*/  **public** **class** LinkedListCircularCalculateNodes {  **public** **static** **void** main(String[] args) {  LinkedList9 linkedList=**new** LinkedList9();  linkedList.insert(66);  linkedList.insert(55);  linkedList.insert(44);  linkedList.insert(33);  linkedList.insert(22);  linkedList.insert(11);    linkedList.makeLinkedListCircular();  linkedList.displayLinkedList();    linkedList.calculateNodesInCircularLikedList();  }  }  **class** LinkedList9{    Node first=**null**;  Node circularPoint1; //points used to make LinkedList circular.  Node circularPoint2;    Node slowPointer; //will step over LinkedList by 1 Node.  Node fastPointer; //will step over LinkedList by 2 Node.      /\*\*  \* Insert New Node at first position  \*/  **public** **void** insert(**int** data){    Node newNode=**new** Node(data);  newNode.next=first;  first=newNode;    /\*  \* Below we have kept track of two Nodes so that later we can make LinkedList circular(If required).  \* Note:- I have kept track of below two nodes just for demonstration purpose. You may provide some other implementation for making LinkedList circular.  \*/  **if**(data==33)  circularPoint1=newNode;  **if**(data==66)  circularPoint2=newNode;    }    /\*\*  \*This method makes LikedList circular- by making end Node point to some middle Node of LinkedList.  \*end Node--->middle Node.  \*/  **public** **void** makeLinkedListCircular(){  circularPoint2.next=circularPoint1;  System.***out***.println("LinkedList has been succesfully converted into CircularLinkedList");  }    /\*\*  \* method calculates number of nodes in circular LinkedList.  \*/  **public** **void** calculateNodesInCircularLikedList(){  slowPointer=first;  fastPointer=first;    **while**( (slowPointer!=fastPointer || slowPointer==first) //when first time condition is checked slowPointer is equal to fastPointer -so that does not means LinkedList is circular and we exit while loop  //we got to be cautious by keeping extra check that whether we are on first node or not.  && fastPointer.next!=**null** //used to avoid NullPointerException(in case we are are on last Node- than next is null, calling further next on null will cause NPE.)  && fastPointer.next.next!=**null**){  slowPointer=slowPointer.next; // step over LinkedList by 1 Node.  fastPointer=fastPointer.next.next; // step over LinkedList by 2 Node.  }    //we will exit above while loop when we have detected LinkedList is circular.    **int** noOfNodes=0; //initially keep number of nodes to zero.    /\*make one of the pointer point to first,  \* & than make both pointers step over LinkedList by 1 Node, they will meet at intersection point giving number of elements before loop formation.  \*/  slowPointer=first;  **while**(slowPointer!=fastPointer){  slowPointer=slowPointer.next; // step over LinkedList by 1 Node.  fastPointer=fastPointer.next; // step over LinkedList by 1 Node.  noOfNodes++;  }  System.***out***.println("LinkedList is circular at Node: "+slowPointer.data);    /\*  \*Make slow pointer step over LinkedList by 1 Node.  \*& Make slow pointer step over LinkedList by 2 Node.  \*They both will meet after certain number of iterations giving length of loop.  \*/  **boolean** pointersJustmet=**true**;  **while**(slowPointer!=fastPointer || pointersJustmet){ //initially slowPointer is equal to fastPointer so boolean variable ensures that we don't exit in first iteration.  slowPointer=slowPointer.next; // step over LinkedList by 1 Node.  fastPointer=fastPointer.next.next; // step over LinkedList by 2 Node.  noOfNodes++;  pointersJustmet=**false**;  }  System.***out***.println("Total number of nodes in LinkedList are: "+noOfNodes);  }      /\*\*  \* Display LinkedList  \*/  **public** **void** displayLinkedList(){  Node tempDisplay=first;  **int** displayLimiterCtr=0; //as our LinkedList is circular it will keep on displaying nodes till infinity...  //so this variable will help us in limiting the display to specific count.  System.***out***.print("Displaying LinkedList [first--->last]: ");  **while**(tempDisplay!=**null**){  tempDisplay.displayNode();  tempDisplay=tempDisplay.next;  **if**(++displayLimiterCtr >= 12) //stops displaying after 12 Nodes.  **break**;  }  System.***out***.println();  }    } |

1. Java program to traverse the elements of a LinkedList in reverse direction?

|  |
| --- |
| **public** **class** LinkedListInReverse {  **public** **static** **void** main(String[] args) {  LinkedList<String> list = **new** LinkedList<String>();  list.add("JAVA");  list.add("J2EE");  list.add("JSP");  list.add("SERVLETS");  list.add("JDBC");  System.***out***.println(list); // Output : [JAVA, J2EE, JSP, SERVLETS, JDBC]  Iterator<String> it = list.descendingIterator();  **while** (it.hasNext()) {  System.***out***.println(it.next());  }  }  } |

1. LinkedListIntersectionPointExample

|  |
| --- |
| //Lists are merging at :19  import java.util.Iterator;  import java.util.LinkedList;  import java.util.List;  public class LinkedListIntersectionPointExample {  public static void main(String[] args) {  List<Integer> l1 = new LinkedList<Integer>();  List<Integer> l2 = new LinkedList<Integer>();  l1.add(11);  l1.add(13);  l1.add(16);  l1.add(19);  l1.add(22);  l1.add(23);  l2.add(12);  l2.add(14);  l2.add(19);  l2.add(22);  l2.add(23);  twoListsAreMergingOrNot(l1, l2);  }  public static void twoListsAreMergingOrNot(List<Integer> l1, List<Integer> l2) {  List<Integer> smallLinkedList = l1;  List<Integer> largeLinkedList = l2;  Integer diffInSize = l2.size() - l1.size();  int ctr = 0;  if (diffInSize < 0) { // if difference is negative, swap the references  // of lists.  smallLinkedList = l2;  largeLinkedList = l1;  diffInSize = Math.abs(diffInSize);  }  Iterator<Integer> smallListIterator = smallLinkedList.iterator();  Iterator<Integer> largeListIterator = largeLinkedList.iterator();  while (largeListIterator.hasNext()) {  int listValue = largeListIterator.next();  if (ctr < diffInSize) {  ctr++;  continue;  }  if (listValue == smallListIterator.next()) {  System.out.println("Lists are merging at :" + listValue);  return;  }  }  }  } |

1. MergeSortForLinkedLists

|  |
| --- |
| //Time Complexity: O(n Log n)  **public** **class** MergeSortForLinkedLists {  node head = **null**;  **static** **class** node {  **int** val;  node next;  **public** node(**int** val) {  **this**.val = val;  }  }  **public** **static** **void** main(String[] args) {  MergeSortForLinkedLists li = **new** MergeSortForLinkedLists();  li.push(15);  li.push(10);  li.push(5);  li.push(20);  li.push(3);  li.push(2);  System.***out***.println("Linked List without sorting is :");  li.printList(li.head);  li.head = li.mergeSort(li.head);  System.***out***.print("\n Sorted Linked List is: \n");  li.printList(li.head);  }  **void** push(**int** new\_data) {  node new\_node = **new** node(new\_data);  new\_node.next = head;  head = new\_node;  }  **void** printList(node headref) {  **while** (headref != **null**) {  System.***out***.print(headref.val + " ");  headref = headref.next;  }  }  node mergeSort(node h) {  **if** (h == **null** || h.next == **null**) {  **return** h;  }  node middle = getMiddle(h);  node nextofmiddle = middle.next;  middle.next = **null**;  node left = mergeSort(h);  node right = mergeSort(nextofmiddle);  node sortedlist = sortedMerge(left, right);  **return** sortedlist;  }  node getMiddle(node h) {  **if** (h == **null**)  **return** h;  node fastptr = h.next;  node slowptr = h;  **while** (fastptr != **null**) {  fastptr = fastptr.next;  **if** (fastptr != **null**) {  slowptr = slowptr.next;  fastptr = fastptr.next;  }  }  **return** slowptr;  }  node sortedMerge(node a, node b) {  node result = **null**;  **if** (a == **null**)  **return** b;  **if** (b == **null**)  **return** a;  **if** (a.val <= b.val) {  result = a;  result.next = sortedMerge(a.next, b);  } **else** {  result = b;  result.next = sortedMerge(a, b.next);  }  **return** result;  }  } |

1. PostOrderTraversalOfBinaryTree

|  |
| --- |
| /\*1. What happens in PostOrder traversal of Binary Tree in java >  In PostOrder traversal of Binary Tree in java each node is processed after traversing its subtree.  2. What are Steps in PostOrder Traversal of Binary Tree in java >  Steps in PostOrder Traversal of Binary Tree in java -  traverse left subtree  traverse the right subtree.  Visit the node.\*/  /\* Output  Post order Traversal of Binary Tree in java :  4 5 2 6 7 3 1    \*/  public class PostOrderTraversalOfBinaryTree {  public static void main(String[] args) {  PostOrderTraversalOfBinaryTree postOrderTraversalOfBinaryTree = new PostOrderTraversalOfBinaryTree();  // Now, let's start creating nodes of Binary tree  // create node1 (which is ROOT node)  BinaryTreeNode binaryTreeNode1 = new BinaryTreeNode(1);  // Left node of node1  BinaryTreeNode node2 = new BinaryTreeNode(2);  binaryTreeNode1.leftNode = node2;  // Right node of node1  BinaryTreeNode node3 = new BinaryTreeNode(3);  binaryTreeNode1.rightNode = node3;  // Left node of node2  BinaryTreeNode node4 = new BinaryTreeNode(4);  node2.leftNode = node4;  // Right node of node2  BinaryTreeNode node5 = new BinaryTreeNode(5);  node2.rightNode = node5;  // Left node of node3  BinaryTreeNode node6 = new BinaryTreeNode(6);  node3.leftNode = node6;  // Right node of node3  BinaryTreeNode node7 = new BinaryTreeNode(7);  node3.rightNode = node7;  System.out.println("Post order Traversal of Binary Tree in java : ");  // Pass root node to Post order traversal method  postOrderTraversalOfBinaryTree.postorderTraversalMethod(binaryTreeNode1);  }  public static class BinaryTreeNode {  // Left and right node of Binary tree  BinaryTreeNode leftNode;  BinaryTreeNode rightNode;  // Data of BinaryTree  int data;  BinaryTreeNode(int data) {  this.data = data;  }  }  public void postorderTraversalMethod(BinaryTreeNode binaryTreeNode) {  // Continue if binaryTreeNode is not null  if (binaryTreeNode != null) { // Visit the node-Printing the node data  // pass leftNode of binary tree recursively  postorderTraversalMethod(binaryTreeNode.leftNode);  // pass rightNode of binary tree recursively  postorderTraversalMethod(binaryTreeNode.rightNode);  // Display data of current binaryTreeNode.  System.out.print(binaryTreeNode.data + " ");  }  }  } |

1. PreOrderTraversalOfBinaryTree

|  |
| --- |
| /\* Output  Pre order Traversal of Binary Tree in java :  1 2 4 5 3 6 7    \*/  /\*1. What happens in PreOrder traversal of Binary Tree in java >  In PreOrder traversal of Binary Tree in java each node is processed before traversing its subtree.  2. What are Steps in PreOrder Traversal of Binary Tree in java >  Steps in PreOrder Traversal of Binary Tree in java -  Visit the node.  Traverse the left subtree.  Traverse the right subtree.\*/  public class PreOrderTraversalOfBinaryTree {  public static void main(String[] args) {  PreOrderTraversalOfBinaryTree preOrderTraversalOfBinaryTree = new PreOrderTraversalOfBinaryTree();  // Now, let's start creating nodes of Binary tree  // create node1 (which is ROOT node)  BinaryTreeNode binaryTreeNode1 = new BinaryTreeNode(1);  // Left node of node1  BinaryTreeNode node2 = new BinaryTreeNode(2);  binaryTreeNode1.leftNode = node2;  // Right node of node1  BinaryTreeNode node3 = new BinaryTreeNode(3);  binaryTreeNode1.rightNode = node3;  // Left node of node2  BinaryTreeNode node4 = new BinaryTreeNode(4);  node2.leftNode = node4;  // Right node of node2  BinaryTreeNode node5 = new BinaryTreeNode(5);  node2.rightNode = node5;  // Left node of node3  BinaryTreeNode node6 = new BinaryTreeNode(6);  node3.leftNode = node6;  // Right node of node3  BinaryTreeNode node7 = new BinaryTreeNode(7);  node3.rightNode = node7;  System.out.println("Pre order Traversal of Binary Tree in java : ");  // Pass root node to pre order traversal method  preOrderTraversalOfBinaryTree.preorderTraversalMethod(binaryTreeNode1);  }  public static class BinaryTreeNode {  // Left and right node of Binary tree  BinaryTreeNode leftNode;  BinaryTreeNode rightNode;  // Data of BinaryTree  int data;  BinaryTreeNode(int data) {  this.data = data;  }  }  public void preorderTraversalMethod(BinaryTreeNode binaryTreeNode) {  // Continue if binaryTreeNode is not null  if (binaryTreeNode != null) {  // Display data of current binaryTreeNode.  System.out.print(binaryTreeNode.data + " ");  // pass leftNode of binary tree recursively  preorderTraversalMethod(binaryTreeNode.leftNode);  // pass rightNode of binary tree recursively  preorderTraversalMethod(binaryTreeNode.rightNode);  }  }  } |

1. QueueImplementationUsingLinkedList

|  |
| --- |
| **public** **static** **void** main(String[] args) {  QueueLinkedList queueLinkedList = **new** QueueLinkedList();  System.***out***.println("INSERTING AT LAST (REAR) IN QUEUE IMPLEMENTED USING LINKED LIST ");  queueLinkedList.insert(11); // insert node.  queueLinkedList.displayStack(); // display QUEUE IMPLEMENTED USING  // LINKED LIST  queueLinkedList.insert(71); // insert node.  queueLinkedList.displayStack(); // display QUEUE IMPLEMENTED USING  // LINKED LIST  queueLinkedList.insert(39); // insert node.  queueLinkedList.displayStack(); // display QUEUE IMPLEMENTED USING  // LINKED LIST  queueLinkedList.insert(31); // insert node.  queueLinkedList.displayStack(); // display QUEUE IMPLEMENTED USING  // LINKED LIST  System.***out***.println("\nDELETING FROM FIRST (FRONT) OF QUEUE IMPLEMENTED USING LINKED LIST ");  queueLinkedList.remove(); // remove Node  queueLinkedList.displayStack(); // display QUEUE IMPLEMENTED USING  // LINKED LIST  queueLinkedList.remove(); // remove Node  queueLinkedList.displayStack(); // display QUEUE IMPLEMENTED USING  // LINKED LIST  queueLinkedList.remove(); // remove Node  queueLinkedList.displayStack(); // display QUEUE IMPLEMENTED USING  // LINKED LIST  queueLinkedList.remove(); // remove Node  queueLinkedList.displayStack(); // display QUEUE IMPLEMENTED USING  // LINKED LIST  }  }  **class** QueueLinkedList {  LinkedList13 linkedList = **new** LinkedList13(); // creation of Linked List  /\*\*  \* Insert element at rear in Queue  \*/  **public** **void** insert(**int** value) {  linkedList.insertLast(value);  }  /\*\*  \* Removes elements from front of Queue  \*/  **public** **void** remove() **throws** QueueEmptyException {  **try** {  linkedList.deleteFirst();  } **catch** (LinkedListEmptyException llee) {  **throw** **new** QueueEmptyException();  }  }  /\*\*  \* Display queue.  \*/  **public** **void** displayStack() {  System.***out***.print("Displaying Queue> Front to Rear: ");  linkedList.displayLinkedList();  }  }  **class** LinkedList13 {  **private** Node first; // ref to first link on list  /\*\*  \* LinkedList constructor  \*/  **public** LinkedList13() {  first = **null**;  }  /\*\*  \* Inserts new Node at last of LinkedList.  \*/  **public** **void** insertLast(**int** data) {  Node newNode = **new** Node(data); // Creation of New Node.  **if** (first == **null**) { // means LinkedList is empty, make first point to  // new Node.  first = newNode; // first ---> newNode  **return**;  }  Node tempNode = first; // save reference to first Node in tempNode- so  // that we could return saved reference.  **while** (tempNode.next != **null**) { // Executes until we don't find last  // Node of LinkedList.  // If next of some Node is pointing to  // null, that means it's a last Node.  tempNode = tempNode.next; // move to next Node.  }  tempNode.next = newNode; // make last's Node next point to new Node  }  /\*\*  \* Deletes first Node  \*/  **public** Node deleteFirst() {  **if** (first == **null**) { // means LinkedList in empty, throw exception.  **throw** **new** LinkedListEmptyException("LinkedList doesn't contain any Nodes.");  }  Node tempNode = first; // save reference to first Node in tempNode- so  // that we could return saved reference.  first = first.next; // delete first Node (make first point to second  // node)  **return** tempNode; // return tempNode (i.e. deleted Node)  }  /\*\*  \* Display LinkedList  \*/  **public** **void** displayLinkedList() {  Node tempDisplay = first; // start at the beginning of linkedList  **while** (tempDisplay != **null**) { // Executes until we don't find end of  // list.  tempDisplay.displayNode();  tempDisplay = tempDisplay.next; // move to next Node  }  System.***out***.println();  }  }  **class** QueueEmptyException **extends** RuntimeException {  **public** QueueEmptyException() {  **super**();  }  **public** QueueEmptyException(String message) {  **super**(message);  }  }  /\*  \* OUTPUT  \*  \* INSERTING AT LAST (REAR) IN QUEUE IMPLEMENTED USING LINKED LIST Displaying  \* Queue> Front to Rear: 11 Displaying Queue> Front to Rear: 11 71 Displaying  \* Queue> Front to Rear: 11 71 39 Displaying Queue> Front to Rear: 11 71 39 31  \*  \* DELETING FROM FIRST (FRONT) OF QUEUE IMPLEMENTED USING LINKED LIST Displaying  \* Queue> Front to Rear: 71 39 31 Displaying Queue> Front to Rear: 39 31  \* Displaying Queue> Front to Rear: 31 Displaying Queue> Front to Rear:  \*  \*/ |

1. remove the first occurrence and last occurrence of a given element in a LinkedList

|  |
| --- |
| **public** **class** Remove\_first\_and\_last\_occurrence {  **public** **static** **void** main(String[] args) {  LinkedList<String> list = **new** LinkedList<String>();  list.add("JAVA");  list.add("J2EE");  list.add("JSP");  list.add("J2EE");  list.add("JDBC");  System.***out***.println(list); // Output : [JAVA, J2EE, JSP, J2EE, JDBC]  // Removing the first occurrence of "J2EE"  list.removeFirstOccurrence("J2EE");  System.***out***.println(list); // Output : [JAVA, JSP, J2EE, JDBC]  // Removing the last occurrence of "J2EE"  list.removeLastOccurrence("J2EE");  System.***out***.println(list); // Output : [JAVA, JSP, JDBC]  }  } |

1. remove all elements of a LinkedList

|  |
| --- |
| **public** **class** RemoveAllElements {  **public** **static** **void** main(String[] args) {  LinkedList<Integer> linkedList = **new** LinkedList<Integer>();  linkedList.add(10);  linkedList.add(20);  linkedList.add(30);  linkedList.add(40);  linkedList.add(50);  System.***out***.println(linkedList); // Output : [10, 20, 30, 40, 50]  // Removing all elements of linkedList  linkedList.clear();  System.***out***.println(linkedList); // Output : []  }  } |

1. remove the elements of a LinkedList from both the ends

|  |
| --- |
| **public** **class** RemoveElements {  **public** **static** **void** main(String[] args) {  LinkedList<String> list = **new** LinkedList<String>();  list.add("ONE");  list.add("TWO");  list.add("THREE");  list.add("FOUR");  list.add("FIVE");  list.add("SIX");  list.add("SEVEN");  System.***out***.println(list); // Output : [ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN]  // Removing the elements from the head of the LinkedList  list.poll();  list.pollFirst();  list.remove();  list.removeFirst();  // Printing the elements of list  System.***out***.println(list); // Output : [FIVE, SIX, SEVEN]  // Removing elements from the end of the LinkedList  list.pollLast();  list.removeLast();  // Printing the elements of list  System.***out***.println(list); // Output : [FIVE]  }  } |

1. replace an element at a specific position of a LinkedList with the given element

|  |
| --- |
| **public** **class** ReplaceElement {  **public** **static** **void** main(String[] args) {  LinkedList<String> list = **new** LinkedList<String>();  list.add("ONE");  list.add("TWO");  list.add("THREE");  list.add("FOUR");  System.***out***.println(list); // Output : [ONE, TWO, THREE, FOUR]  // Replacing an element at index 2 with "ZERO"  list.set(2, "ZERO");  System.***out***.println(list); // Output : [ONE, TWO, ZERO, FOUR]  }  } |

1. retrieve but not remove the elements of a LinkedList from both the ends

|  |
| --- |
| **public** **class** Retrieve\_But\_Not\_Remove {  **public** **static** **void** main(String[] args) {  LinkedList<String> list = **new** LinkedList<String>();  list.add("FIRST");  list.add("SECOND");  list.add("THIRD");  list.add("FOURTH");  list.add("FIFTH");  System.***out***.println(list); // Output : [FIRST, SECOND, THIRD, FOURTH, FIFTH]  // Retrieving the elements from the head  System.***out***.println(list.element()); // Output : FIRST  System.***out***.println(list.getFirst()); // Output : FIRST  System.***out***.println(list.peek()); // Output : FIRST  System.***out***.println(list.peekFirst()); // Output : FIRST  // Retrieving the elements from the tail  System.***out***.println(list.peekLast()); // Output : FIFTH  System.***out***.println(list.getLast()); // Output : FIFTH  }  } |

1. retrieve and remove and only retrieve an element from specific position of a LinkedList

|  |
| --- |
| **public** **class** RetrieveAndRemove {  **public** **static** **void** main(String[] args) {  LinkedList<String> list = **new** LinkedList<String>();  list.add("JAVA");  list.add("J2EE");  list.add("JSP");  list.add("SERVLETS");  list.add("JDBC");  System.***out***.println(list); // Output : [JAVA, J2EE, JSP, SERVLETS, JDBC]  // Retrieving and removing an element at index 2 of the list  System.***out***.println(list.remove(2)); // Output : JSP  System.***out***.println(list); // Output : [JAVA, J2EE, SERVLETS, JDBC]  // Only retrieving an element at index 2 of the list  System.***out***.println(list.get(2)); // Output : SERVLETS  }  } |

1. ReverseSinglyLinkedList

|  |
| --- |
| **public** **class** ReverseSinglyLinkedList {  **public** **static** **void** main(String[] args) {  SinglyLinkedList.Node head = **new** SinglyLinkedList.Node(1);  SinglyLinkedList linkedlist = **new** SinglyLinkedList(head);  linkedlist.add(**new** SinglyLinkedList.Node(2));  linkedlist.add(**new** SinglyLinkedList.Node(3));  linkedlist.print();  linkedlist.reverse();  linkedlist.print();  }  }  **class** SinglyLinkedList {  **static** **class** Node {  **private** **int** data;  **private** Node next;  **public** Node(**int** data) {  **this**.data = data;  }  **public** **int** data() {  **return** data;  }  **public** Node next() {  **return** next;  }  }  **private** Node head;  **public** SinglyLinkedList(Node head) {  **this**.head = head;  }  **public** **void** add(Node node) {  Node current = head;  **while** (current != **null**) {  **if** (current.next == **null**) {  current.next = node;  **break**;  }  current = current.next;  }  }  **public** **void** print() {  Node node = head;  **while** (node != **null**) {  System.***out***.print(node.data() + " ");  node = node.next();  }  System.***out***.println("");  }  **public** **void** reverse() {  Node pointer = head;  Node previous = **null**, current = **null**;  **while** (pointer != **null**) {  current = pointer;  pointer = pointer.next;  // reverse the link  current.next = previous;  previous = current;  head = current;  }  }  }  /\*  \* Output 1 2 3  \* 3 2 1  \*/ |

1. ReverseSinglyLinkedListIterativeRecursive

|  |
| --- |
| /\*Output: linked list before reversing : A-->B-->C-->D-->E-->F  linked list after reversing : F-->E-->D-->C-->B-->A  linked list after reversing recursively: A-->B-->C-->D-->E-->F  \*/  **public** **class** ReverseSinglyLinkedListIterativeRecursive {  **public** **static** **void** main(String args[]) {  SinglyLinkedList11 linkedlist = *getDefaultList*();  System.***out***.println("linked list before reversing : " + linkedlist);  linkedlist.reverseIteratively();  System.***out***.println("linked list after reversing : " + linkedlist);  linkedlist.reverseRecursively();  System.***out***.println("linked list after reversing recursively: " + linkedlist);  }  **private** **static** SinglyLinkedList11 getDefaultList() {  SinglyLinkedList11 linkedlist = **new** SinglyLinkedList11();  linkedlist.append("A");  linkedlist.append("B");  linkedlist.append("C");  linkedlist.append("D");  linkedlist.append("E");  linkedlist.append("F");  **return** linkedlist;  }  }  **class** SinglyLinkedList11<T> {  **private** Node11 head;  **public** SinglyLinkedList11() {  // **TODO** Auto-generated constructor stub  }  **public** **void** append(T data) {  **if** (head == **null**) {  head = **new** Node11(data);  **return**;  }  tail().next = **new** Node11(data);  }  **public** **void** reverseRecursively() {  head = reverseRecursively(head);  }  **private** Node11 reverseRecursively(Node11 head2) {  Node11 newHead;  **if** ((head2.next == **null**)) {  **return** head2;  }  newHead = reverseRecursively(head2.next);  head2.next.next = head2;  head2.next = **null**;  **return** newHead;  }  **public** **void** reverseIteratively() {  Node11 current = head;  Node11 previous = **null**;  Node11 forward = **null**;  **while** (current.next != **null**) {  forward = current.next;  current.next = previous;  previous = current;  current = forward;  }  head = current;  head.next = previous;  }  **private** Node11 tail() {  Node11 tail = head;  **while** (tail.next != **null**) {  tail = tail.next;  }  **return** tail;  }  @Override  **public** String toString(){  StringBuilder sb = **new** StringBuilder();  Node11 current = head;  **while**(current != **null**){  sb.append(current).append("-->");  current = current.next;  }  **if**(sb.length()>=3){  sb.delete(sb.length() - 3, sb.length());  // to remove --> from last node  }  **return** sb.toString();  }  **class** Node11 {  **private** Node11 next;  **private** T data;  **public** Node11(T data) {  **this**.data = data;  }  @Override  **public** String toString() {  **return** data.toString();  }  }  } |

1. ReversingSinglyLinkedList

|  |
| --- |
| /\*Displaying LinkedList [first--->last]: 1 2 3 4  LinkedList has been reversed successfully.  Displaying LinkedList [first--->last]: 4 3 2 1\*/  **public** **class** ReversingSinglyLinkedList {  **public** **static** **void** main(String[] args) {  LinkedList6 linkedList = **new** LinkedList6();    linkedList.insertFirst(4);  linkedList.insertFirst(3);  linkedList.insertFirst(2);  linkedList.insertFirst(1);  linkedList.displayLinkedList();  linkedList.reverseLinkedList(); //REVERSE LinkedList  linkedList.displayLinkedList();  }  }  **class** LinkedList6 {  **private** Node first; // ref to first link on list    /\*\*  \* Singly LinkedList constructor  \*/  **public** LinkedList6(){  first = **null**;  }    /\*\*  \* REVERSE linkedList.  \*/  **public** **void** reverseLinkedList() {  //Using 3 pointers for reversing LinkedList.  Node previousNode=**null**;  Node currentNode=first;  Node nextNode=first;    **while**(nextNode!=**null**){  nextNode=nextNode.next; //make nextNode point to next node.  currentNode.next=previousNode; //make current node's next point to previous node.  previousNode=currentNode; //make previousNode point to currentNode.  currentNode=nextNode; //make currentNode point to nextNode.  }  //by this stage of program all the nodes are pointing to previous nodes(which has helped us generating reverse of LinkedList.)  first=previousNode; //now make first point to previous node(i.e. last node).    System.***out***.println("LinkedList has been reversed successfully.");  }        /\*\*  \* Insert New Node at first position  \*/  **public** **void** insertFirst(**int** data) {  Node newNode = **new** Node(data); //Creation of New Node.  newNode.next = first; //newLink ---> old first  first = newNode; //first ---> newNode  }        /\*\*  \* Display Singly LinkedList  \*/  **public** **void** displayLinkedList() {  System.***out***.print("Displaying LinkedList [first--->last]: ");  Node tempDisplay = first; // start at the beginning of linkedList  **while** (tempDisplay != **null**){ // Executes until we don't find end of list.  tempDisplay.displayNode();  tempDisplay = tempDisplay.next; // move to next Node  }  System.***out***.println();    }    } |

1. Segregate\_Even\_And\_Odd\_Nodes

|  |
| --- |
| /\*Examples:  Input: 17->15->8->12->10->5->4->1->7->6->NULL  Output: 8->12->10->4->6->17->15->5->1->7->NULL  Input: 8->12->10->5->4->1->6->NULL  Output: 8->12->10->4->6->5->1->NULL  // If all numbers are even then do not change the list  Input: 8->12->10->NULL  Output: 8->12->10->NULL  // If all numbers are odd then do not change the list  Input: 1->3->5->7->NULL  Output: 1->3->5->7->NULL\*/  **class** Segregate\_Even\_And\_Odd\_Nodes {  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;  }  }  // Move head pointer one step in forward direction  currentNode = currentNode.next;  }  **if** (oddStart == **null** || evenStart == **null**) {  **return**;  }  evenEnd.next = oddStart;  oddEnd.next = **null**;  head = evenStart;  }  /\*  \* Given a reference (pointer to pointer) to the head of a list and an int,  \* push a new node on the front of the list.  \*/  **void** push(**int** new\_data) {  /\*  \* 1 & 2: Allocate the Node & Put in the data  \*/  Node new\_node = **new** Node(new\_data);  /\* 3. Make next of new Node as head \*/  new\_node.next = head;  /\* 4. Move the head to point to new Node \*/  head = new\_node;  }  // Utility function to print a linked list  **void** printList() {  Node temp = head;  **while** (temp != **null**) {  System.***out***.print(temp.data + " ");  temp = temp.next;  }  System.***out***.println();  }  /\* Drier program to test above functions \*/  **public** **static** **void** main(String args[]) {  Segregate\_Even\_And\_Odd\_Nodes llist = **new** Segregate\_Even\_And\_Odd\_Nodes();  llist.push(11);  llist.push(10);  llist.push(9);  llist.push(6);  llist.push(4);  llist.push(1);  llist.push(0);  System.***out***.println("Origional Linked List");  llist.printList();  llist.segregateEvenOdd();  System.***out***.println("Modified Linked List");  llist.printList();  }  } |

1. SinglyLinkedListatFirst

|  |
| --- |
| **public** **class** SinglyLinkedListatFirst {  **public** **static** **void** main(String[] args) {  LinkedList linkedList = **new** LinkedList(); // creation of Linked List  linkedList.insertFirst(11);  linkedList.insertFirst(21);  linkedList.insertFirst(59);  linkedList.insertFirst(14);  linkedList.insertFirst(39);  linkedList.displayLinkedList(); // display LinkedList  System.***out***.print("Deleted Nodes: ");  Node deletedNode = linkedList.deleteFirst(); // delete Node  deletedNode.displayNode(); // display deleted Node.  deletedNode = linkedList.deleteFirst(); // delete Node.  deletedNode.displayNode(); // display deleted Node.  System.***out***.println();// sysout used to format output  linkedList.displayLinkedList(); // Again display LinkedList  }  }  **class** LinkedList {  **private** Node first; // ref to first link on list  **public** LinkedList() {  first = **null**;  }  /\*\* Insert New Node at first position in Singly LinkedList \*/  **public** **void** insertFirst(**int** data) {  Node newNode = **new** Node(data); // Creation of New Node.  newNode.next = first; // newLink ---> old first  first = newNode; // first ---> newNode  }  /\*\* Deletes first Node of Singly LinkedList \*/  **public** Node deleteFirst() {  **if** (first == **null**) { // means LinkedList in empty, throw exception.  **throw** **new** LinkedListEmptyException("LinkedList doesn't contain any Nodes.");  }  Node tempNode = first; // save reference to first Node in tempNode- so  // that we could return saved reference.  first = first.next; // delete first Node (make first point to second  // node)  **return** tempNode; // return tempNode (i.e. deleted Node)  }  /\*\* Display Singly LinkedList \*/  **public** **void** displayLinkedList() {  System.***out***.print("Displaying LinkedList [first--->last]: ");  Node tempDisplay = first; // start at the beginning of linkedList  **while** (tempDisplay != **null**) { // Executes until we don't find end of  // list.  tempDisplay.displayNode();  tempDisplay = tempDisplay.next; // move to next Node  }  System.***out***.println();  }  }  **class** Node {  **public** **int** data; // data in Node.  **public** Node next; // points to next Node in list.  **public** Node(**int** data) {  **this**.data = data;  }  /\*\* Display Node's data \*/  **public** **void** displayNode() {  System.***out***.print(data + " ");  }  }  /\*\* Exception to indicate that Singly LinkedList is empty. \*/  **class** LinkedListEmptyException **extends** RuntimeException {  **public** LinkedListEmptyException() {  **super**();  }  **public** LinkedListEmptyException(String message) {  **super**(message);  }  }  /\*Displaying LinkedList [first--->last]: 39 14 59 21 11  Deleted Nodes: 39 14  Displaying LinkedList [first--->last]: 59 21 11 \*/ |

1. SinglyLinkedListatLast

|  |
| --- |
| **public** **class** SinglyLinkedListatLast {  **public** **static** **void** main(String[] args) {  LinkedList1 linkedList = **new** LinkedList1(); // creation of Linked List  linkedList.insertLast(11);  linkedList.insertLast(21);  linkedList.insertLast(59);  linkedList.insertLast(14);  linkedList.insertLast(39);  linkedList.displayLinkedList(); // display LinkedList  System.***out***.print("Deleted Nodes: ");  Node1 deletedNode = linkedList.deleteLast(); // delete Node  deletedNode.displayNode(); // display deleted Node.  deletedNode = linkedList.deleteLast(); // delete Node  deletedNode.displayNode(); // display deleted Node.  System.***out***.println();// sysout used to format output  linkedList.displayLinkedList(); // Again display LinkedList  }  }  **class** LinkedList1 {  **private** Node1 first; // ref to first link on list  /\*\* LinkedList constructor \*/  **public** LinkedList1() {  first = **null**;  }  /\*\* Inserts new Node at last of Singly LinkedList. \*/  **public** **void** insertLast(**int** data) {  Node1 newNode = **new** Node1(data); // Creation of New Node.  **if** (first == **null**) { // means LinkedList is empty, make first point to  // new Node.  first = newNode; // first ---> newNode  **return**;  }  Node1 tempNode = first; // save reference to first Node in tempNode- so  // that we could return saved reference.  **while** (tempNode.next != **null**) { // Executes until we don't find last  // Node of LinkedList.  // If next of some Node is pointing to  // null, that means it's a last Node.  tempNode = tempNode.next; // move to next Node.  }  tempNode.next = newNode; // make last's Node next point to new Node  }  /\*\* Deletes last Node from Singly LinkedList \*/  **public** Node1 deleteLast() {  // Case1: when there is no element in LinkedList  **if** (first == **null**) { // means LinkedList in empty, throw exception.  **throw** **new** LinkedListEmptyException1("LinkedList doesn't contain any Nodes.");  }  // Case2: when there is only one element in LinkedList  **if** (first.next == **null**) { // means LinkedList consists of only one  // element, delete that.  Node1 tempNode = first; // save reference to first Node in tempNode-  // so that we could return saved reference.  first = first.next; // delete firstNode (make first point to  // secondNode)  **return** tempNode; // return deleted Node.  }  // Case3: when there are atLeast two elements in LinkedList  Node1 previous = **null**;  Node1 current = first;  **while** (current.next != **null**) {// Executes until we don't find last Node  // of LinkedList.  // If next of some Node is pointing to  // null, that means it's a last Node.  previous = current;  current = current.next; // move to next node.  }  previous.next = **null**; // Now, previous is pointing to second last Node  // of LinkiedList,  // make it point to null [it byPasses current  // Node(last Node of LinkedList) which was in  // between]  **return** current;  }  /\*\* Display LinkedList \*/  **public** **void** displayLinkedList() {  System.***out***.print("Displaying LinkedList [first--->last]: ");  Node1 tempDisplay = first; // start at the beginning of linkedList  **while** (tempDisplay != **null**) { // Executes until we don't find end of  // list.  tempDisplay.displayNode();  tempDisplay = tempDisplay.next; // move to next Node  }  System.***out***.println();  }  }  **class** Node1 {  **public** **int** data; // data in Node.  **public** Node1 next; // points to next Node in list.  /\*\*  \* Constructor  \*/  **public** Node1(**int** data) {  **this**.data = data;  }  /\*\*  \* Display Node's data  \*/  **public** **void** displayNode() {  System.***out***.print(data + " ");  }  }  **class** LinkedListEmptyException1 **extends** RuntimeException {  **public** LinkedListEmptyException1() {  **super**();  }  **public** LinkedListEmptyException1(String message) {  **super**(message);  }  }  /\*Displaying LinkedList [first--->last]: 11 21 59 14 39  Deleted Nodes: 39 14  Displaying LinkedList [first--->last]: 11 21 59 \*/ |

1. SinglyLinkedListCircularOrNot

|  |
| --- |
| /\*Displaying LinkedList [first--->last]: 11 22 33 44 55 66 LinkedList is NOT CIRCULAR  LinkedList has been succesfully converted into CircularLinkedList  Displaying LinkedList [first--->last]: 11 22 33 44 55 66 33 44 55 66 33 44 LinkedList is CIRCULAR\*/  **public** **class** SinglyLinkedListCircularOrNot {  **public** **static** **void** main(String[] args) {  LinkedList7 linkedList = **new** LinkedList7();  linkedList.insert(66);  linkedList.insert(55);  linkedList.insert(44);  linkedList.insert(33);  linkedList.insert(22);  linkedList.insert(11);  linkedList.displayLinkedList();  linkedList.findCircularOrNot();  System.***out***.println();  linkedList.makeLinkedListCircular();  linkedList.displayLinkedList();  linkedList.findCircularOrNot();  }  }  **class** LinkedList7 {  Node first = **null**;  Node circularPoint1;  Node circularPoint2;  Node slowPointer;  Node fastPointer;  **public** **void** insert(**int** data) {  Node newNode = **new** Node(data);  newNode.next = first;  first = newNode;  **if** (data == 33)  circularPoint1 = newNode;  **if** (data == 66)  circularPoint2 = newNode;  }  **public** **void** makeLinkedListCircular() {  circularPoint2.next = circularPoint1;  System.***out***.println("LinkedList has been succesfully converted into CircularLinkedList");  }  **public** **void** findCircularOrNot() {  slowPointer = first;  fastPointer = first;  **while** ((slowPointer != fastPointer || slowPointer == first)  && fastPointer.next != **null**  && fastPointer.next.next != **null**) {  slowPointer = slowPointer.next;  fastPointer = fastPointer.next.next;  }  **if** (slowPointer == fastPointer)  System.***out***.println("LinkedList is CIRCULAR");  **else**  System.***out***.println("LinkedList is NOT CIRCULAR");  }  **public** **void** displayLinkedList() {  Node tempDisplay = first;  **int** displayLimiterCtr = 0;  System.***out***.print("Displaying LinkedList [first--->last]: ");  **while** (tempDisplay != **null**) {  tempDisplay.displayNode();  tempDisplay = tempDisplay.next;  **if** (++displayLimiterCtr >= 12)  **break**;  }  System.***out***.println();  }  } |

1. SinglyLinkedListDeleteSpecificNode

|  |
| --- |
| /\*Displaying LinkedList [first--->last]: 29 19 20 92  Node with data=29 was found on first and has been deleted.  Node with data=11 wasn't found for deletion.  Displaying LinkedList [first--->last]: 19 20 92 \*/  **public** **class** SinglyLinkedListDeleteSpecificNode {  **public** **static** **void** main(String[] args) {  LinkedList3 linkedList = **new** LinkedList3();  linkedList.insertFirst(92);  linkedList.insertFirst(20);  linkedList.insertFirst(19);  linkedList.insertFirst(29);  linkedList.displayLinkedList();  linkedList.deleteSpecificNode(29);  linkedList.deleteSpecificNode(11);  linkedList.displayLinkedList();  }  }  **class** LinkedList3 {  **private** Node first;  **public** LinkedList3() {  first = **null**;  }  **public** **void** insertFirst(**int** data) {  Node newNode = **new** Node(data);  newNode.next = first;  first = newNode;  }  **public** Node deleteSpecificNode(**int** deleteKey) {  **if** (first == **null**) {  **throw** **new** LinkedListEmptyException("LinkedList doesn't contain any Nodes.");  }  **if** (first.data == deleteKey) {  Node tempNode = first;  first = first.next;  System.***out***.println("Node with data=" + tempNode.data + " was found on first and has been deleted.");  **return** tempNode;  }  Node previous = **null**;  Node current = first;  **while** (current != **null**) {  **if** (current.data == deleteKey) {  System.***out***.println("Node with data=" + current.data + " has been deleted.");  previous.next = current.next;  **return** current;  } **else** {  **if** (current.next == **null**) {  System.***out***.println("Node with data=" + deleteKey + " wasn't found for deletion.");  **return** **null**;  }  previous = current;  current = current.next;  }  }  **return** **null**;  }  **public** **void** displayLinkedList() {  System.***out***.print("Displaying LinkedList [first--->last]: ");  Node tempDisplay = first;  **while** (tempDisplay != **null**) {  tempDisplay.displayNode();  tempDisplay = tempDisplay.next;  }  System.***out***.println();  }  } |

1. SinglyLinkedListGenericExample

|  |
| --- |
| /\*Displaying LinkedList [first--->last]: Employee [id=39, name=pat] Employee [id=14, name=sai] Employee [id=59, name=katy]  Employee [id=21, name=amy] Employee [id=11, name=sam]  Deleted Nodes: Employee [id=39, name=pat] Employee [id=14, name=sai]  Displaying LinkedList [first--->last]: Employee [id=59, name=katy] Employee [id=21, name=amy] Employee [id=11, name=sam] \*/  **public** **class** SinglyLinkedListGenericExample {  **public** **static** **void** main(String[] args) {  LinkedList11<Employee11> linkedList = **new** LinkedList11<Employee11>();  linkedList.insertFirst(**new** Employee11("11", "sam"));  linkedList.insertFirst(**new** Employee11("21", "amy"));  linkedList.insertFirst(**new** Employee11("59", "katy"));  linkedList.insertFirst(**new** Employee11("14", "sai"));  linkedList.insertFirst(**new** Employee11("39", "pat"));  linkedList.displayLinkedList();  System.***out***.print("Deleted Nodes: ");  Node11<Employee11> deletedNode = linkedList.deleteFirst();  deletedNode.displayNode();  deletedNode = linkedList.deleteFirst();  deletedNode.displayNode();  System.***out***.println();  linkedList.displayLinkedList();  }  }  **class** LinkedList11<T> {  **private** Node11<T> first;  **public** LinkedList11() {  first = **null**;  }  **public** **void** insertFirst(T data) {  Node11<T> newNode = **new** Node11<T>(data);  newNode.next = first;  first = newNode;  }  **public** Node11<T> deleteFirst() {  **if** (first == **null**) {  **throw** **new** LinkedListEmptyException("LinkedList doesn't contain any Nodes.");  }  Node11<T> tempNode = first;  first = first.next;  **return** tempNode;  }  **public** **void** displayLinkedList() {  System.***out***.print("Displaying LinkedList [first--->last]: ");  Node11<T> tempDisplay = first;  **while** (tempDisplay != **null**) {  tempDisplay.displayNode();  tempDisplay = tempDisplay.next;  }  System.***out***.println();  }  }  **class** Node11<T> {  **public** T data;  **public** Node11<T> next;  **public** Node11(T data) {  **this**.data = data;  }  **public** **void** displayNode() {  System.***out***.print(data + " ");  }  }  **class** Employee11 {  **private** String id;  **private** String name;  **public** Employee11(String id, String name) {  **this**.id = id;  **this**.name = name;  }  @Override  **public** String toString() {  **return** "Employee [id=" + id + ", name=" + name + "] ";  }  } |

1. SinglyLinkedListInsertNodeInBetween

|  |
| --- |
| /\*Displaying LinkedList (first--->last):  Node with data=92 inserted at first.  Node with data=20 inserted at first Node, beacuse newNode is smallest of existing Nodes.  Node with data=19 inserted at first Node, beacuse newNode is smallest of existing Nodes.  Node with data=29 inserted successfully in middle of LinkedList.  Node with data=99 inserted successfully at last of LinkedList.  Displaying LinkedList (first--->last): 19 20 29 92 99 \*/  **public** **class** SinglyLinkedListInsertNodeInBetween {  **public** **static** **void** main(String[] args) {  LinkedList2 linkedList = **new** LinkedList2();  linkedList.displayLinkedList();  linkedList.insertNodeInSortedLinkedList(92);  linkedList.insertNodeInSortedLinkedList(20);  linkedList.insertNodeInSortedLinkedList(19);  linkedList.insertNodeInSortedLinkedList(29);  linkedList.insertNodeInSortedLinkedList(99);  linkedList.displayLinkedList();  }  }  **class** LinkedList2 {  **private** Node first;  **public** LinkedList2() {  first = **null**;  }  **public** **void** insertNodeInSortedLinkedList(**int** data) {  Node newNode = **new** Node(data);  **if** (first == **null**) {  first = newNode;  System.***out***.println("Node with data=" + newNode.data + " inserted at first.");  **return**;  }  **if** (first.data >= newNode.data) {  newNode.next = first;  first = newNode;  System.***out***.println("Node with data=" + newNode.data  + " inserted at first Node, beacuse newNode is smallest of existing Nodes.");  **return**;  }  Node current = first;  Node previous = **null**;  **while** (current != **null**) {  **if** (current.data < newNode.data) {  **if** (current.next == **null**) {  current.next = newNode;  System.***out***.println(  "Node with data=" + newNode.data + " inserted successfully at last of LinkedList.");  **return**;  }  previous = current;  current = current.next;  } **else** {  newNode.next = previous.next;  previous.next = newNode;  System.***out***  .println("Node with data=" + newNode.data + " inserted successfully in middle of LinkedList.");  **return**;  }  }  }  **public** **void** displayLinkedList() {  System.***out***.print("Displaying LinkedList (first--->last): ");  Node tempDisplay = first;  **while** (tempDisplay != **null**) {  tempDisplay.displayNode();  tempDisplay = tempDisplay.next;  }  System.***out***.println();  }  } |

1. SinglyLinkedListIsPalindrome\_ByReversing

|  |
| --- |
| /\*Given a singly linked list of characters, write a function that returns true if the given list is palindrome, else false.  Time Complexity O(n)  Auxiliary Space: O(1)\*/  **public** **class** SinglyLinkedListIsPalindrome\_ByReversing {  Node head;  Node slow\_ptr, fast\_ptr, second\_half;  **class** Node {  **char** data;  Node next;  Node(**char** d) {  data = d;  next = **null**;  }  }  **public** **static** **void** main(String[] args) {  SinglyLinkedListIsPalindrome\_ByReversing llist = **new** SinglyLinkedListIsPalindrome\_ByReversing();  **char** str[] = { 'a', 'b', 'a', 'c', 'a', 'b', 'a' };  String string = **new** String(str);  **for** (**int** i = 0; i < 7; i++) {  llist.push(str[i]);  llist.printList(llist.head);  **if** (llist.isPalindrome(llist.head) != **false**) {  System.***out***.println("Is Palindrome");  System.***out***.println("");  } **else** {  System.***out***.println("Not Palindrome");  System.***out***.println("");  }  }  }  **public** **void** push(**char** new\_data) {  Node new\_node = **new** Node(new\_data);  new\_node.next = head;  head = new\_node;  }  **void** printList(Node ptr) {  **while** (ptr != **null**) {  System.***out***.print(ptr.data + "->");  ptr = ptr.next;  }  System.***out***.println("NULL");  }  **boolean** isPalindrome(Node head) {  slow\_ptr = head;  fast\_ptr = head;  Node prev\_of\_slow\_ptr = head;  Node midnode = **null**;  **boolean** res = **true**;  **if** (head != **null** && head.next != **null**) {  **while** (fast\_ptr != **null** && fast\_ptr.next != **null**) {  fast\_ptr = fast\_ptr.next.next;  prev\_of\_slow\_ptr = slow\_ptr;  slow\_ptr = slow\_ptr.next;  }  **if** (fast\_ptr != **null**) {  midnode = slow\_ptr;  slow\_ptr = slow\_ptr.next;  }  second\_half = slow\_ptr;  prev\_of\_slow\_ptr.next = **null**;  reverse();  res = compareLists(head, second\_half);  reverse();  **if** (midnode != **null**) {  prev\_of\_slow\_ptr.next = midnode;  midnode.next = second\_half;  } **else**  prev\_of\_slow\_ptr.next = second\_half;  }  **return** res;  }  **void** reverse() {  Node prev = **null**;  Node current = second\_half;  Node next;  **while** (current != **null**) {  next = current.next;  current.next = prev;  prev = current;  current = next;  }  second\_half = prev;  }  **boolean** compareLists(Node head1, Node head2) {  Node temp1 = head1;  Node temp2 = head2;  **while** (temp1 != **null** && temp2 != **null**) {  **if** (temp1.data == temp2.data) {  temp1 = temp1.next;  temp2 = temp2.next;  } **else**  **return** **false**;  }  **if** (temp1 == **null** && temp2 == **null**)  **return** **true**;  **return** **false**;  }  } |

1. SinglyLinkedListIsPalindrome\_Recursion

|  |
| --- |
| /\*Time Complexity: O(n)  Auxiliary Space: O(n)\*/  **public** **class** SinglyLinkedListIsPalindrome\_Recursion {  Node head;  Node left;  **class** Node {  **char** data;  Node next;  Node(**char** d) {  data = d;  next = **null**;  }  }  **public** **static** **void** main(String[] args) {  SinglyLinkedListIsPalindrome\_Recursion llist = **new** SinglyLinkedListIsPalindrome\_Recursion();  **char** str[] = { 'a', 'b', 'a', 'c', 'a', 'b', 'a' };  String string = **new** String(str);  **for** (**int** i = 0; i < 7; i++) {  llist.push(str[i]);  llist.printList(llist.head);  **if** (llist.isPalindrome(llist.head) != **false**) {  System.***out***.println("Is Palindrome");  System.***out***.println("");  } **else** {  System.***out***.println("Not Palindrome");  System.***out***.println("");  }  }  }  **public** **void** push(**char** new\_data) {  Node new\_node = **new** Node(new\_data);  new\_node.next = head;  head = new\_node;  }  **void** printList(Node ptr) {  **while** (ptr != **null**) {  System.***out***.print(ptr.data + "->");  ptr = ptr.next;  }  System.***out***.println("NULL");  }  **boolean** isPalindrome(Node head) {  **boolean** result = isPalindromeUtil(head);  **return** result;  }  **boolean** isPalindromeUtil(Node right) {  left = head;  **if** (right == **null**)  **return** **true**;  **boolean** isp = isPalindromeUtil(right.next);  **if** (isp == **false**)  **return** **false**;  **boolean** isp1 = (right.data == (left).data);  left = left.next;  **return** isp1;  }  } |

1. SortedDoublyLinkedList

|  |
| --- |
| /\*Node with data=11 inserted at first.  Node with data=21 inserted successfully at last of LinkedList.  Node with data=59 inserted successfully at last of LinkedList.  Node with data=14 inserted successfully in middle of LinkedList.  Node with data=39 inserted successfully in middle of LinkedList.  Node with data=66 inserted successfully at last of LinkedList.  Node with data=33 inserted successfully in middle of LinkedList.  Displaying in forward direction [first--->last] : 11 14 21 33 39 59 66  Displaying in backward direction [last-->first] : 66 59 39 33 21 14 11  Node with data=11 was found on first and has been deleted.  Node with data=21 has been deleted.  Node with data=29 wasn't found for deletion.  Node with data=59 has been deleted.  Displaying in forward direction [first--->last] : 14 33 39 66  Displaying in backward direction [last-->first] : 66 39 33 14\*/  **public** **class** SortedDoublyLinkedList {  **public** **static** **void** main(String[] args) {  LinkedList5 linkedList = **new** LinkedList5();  linkedList.insertSorted(11);  linkedList.insertSorted(21);  linkedList.insertSorted(59);  linkedList.insertSorted(14);  linkedList.insertSorted(39);  linkedList.insertSorted(66);  linkedList.insertSorted(33);  linkedList.displayFrwd();  linkedList.displayBckwrd();  System.***out***.println();  linkedList.deleteSpecificNode(11);  linkedList.deleteSpecificNode(21);  linkedList.deleteSpecificNode(29);  linkedList.deleteSpecificNode(59);  linkedList.displayFrwd();  linkedList.displayBckwrd();  }  }  **class** LinkedList5 {  **private** Node4 first;  **private** Node4 last;  **public** LinkedList5() {  first = **null**;  }  **public** **void** insertFirst(**int** data) {  Node4 newNode = **new** Node4(data);  **if** (first == **null**)  last = newNode;  **else**  first.previous = newNode;  newNode.next = first;  first = newNode;  }  **public** Node4 deleteFirst() {  Node4 tempNode = first;  **if** (first.next == **null**)  last = **null**;  **else**  first.next.previous = **null**;  first = first.next;  **return** tempNode;  }  **public** **void** insertSorted(**int** newKey) {  Node4 newNode = **new** Node4(newKey);  **if** (first == **null**) {  first = last = newNode;  System.***out***.println("Node with data=" + newNode.data + " inserted at first.");  **return**;  }  Node4 current = first;  **if** (current.data >= newNode.data) {  newNode.next = first;  first.previous = newNode;  first = newNode;  System.***out***.println("Node with data=" + newNode.data  + " inserted at first Node, beacuse newNode is smallest of existing Nodes.");  **return**;  }  **while** (**true**) {  **if** (newNode.data > current.data) {  **if** (current.next == **null**) {  last.next = newNode;  newNode.previous = last;  last = newNode;  System.***out***.println(  "Node with data=" + newNode.data + " inserted successfully at last of LinkedList.");  **return**;  }  current = current.next;  } **else** {  current = current.previous;  newNode.next = current.next;  current.next.previous = newNode;  newNode.previous = current;  current.next = newNode;  System.***out***.println("Node with data=" + newNode.data + " inserted successfully in middle of LinkedList.");  **return**;  }  }  }  **public** **void** deleteSpecificNode(**int** deleteKey) {  **if** (first == **null**) {  **throw** **new** LinkedListEmptyException("LinkedList doesn't contain any Nodes.");  }  Node4 current = first;  **while** (current.data != deleteKey) {  **if** (current.next == **null**) {  System.***out***.println("Node with data=" + deleteKey + " wasn't found for deletion.");  **return**;  }  current = current.next;  }  **if** (current == first) {  System.***out***.println("Node with data=" + current.data + " was found on first and has been deleted.");  first = first.next;  first.previous = **null**;  } **else** **if** (current == last) {  System.***out***.println("Node with data=" + current.data + " was found on last has been deleted.");  last = last.previous;  last.next = **null**;  } **else** {  current.previous.next = current.next;  current.next.previous = current.previous;  System.***out***.println("Node with data=" + current.data + " has been deleted.");  }  }  **public** **void** displayFrwd() {  System.***out***.print("Displaying in forward direction [first--->last] : ");  Node4 tempDisplay = first;  **while** (tempDisplay != **null**) {  tempDisplay.displayNode();  tempDisplay = tempDisplay.next;  }  System.***out***.println("");  }  **public** **void** displayBckwrd() {  System.***out***.print("Displaying in backward direction [last-->first] : ");  Node4 tempDisplay = last;  **while** (tempDisplay != **null**) {  tempDisplay.displayNode();  tempDisplay = tempDisplay.previous;  }  System.***out***.println("");  }  } |

1. SortLinkedListOf0sAnd1sAnd2s

|  |
| --- |
| /\*Time Complexity: O(n) where n is number of nodes in linked list.  Auxiliary Space: O(1)\*/  **public** **class** SortLinkedListOf0sAnd1sAnd2s {  Node head; // head of list  **class** Node {  **int** data;  Node next;  Node(**int** d) {  data = d;  next = **null**;  }  }  **public** **static** **void** main(String[] args) {  SortLinkedListOf0sAnd1sAnd2s llist = **new** SortLinkedListOf0sAnd1sAnd2s();  llist.push(0);  llist.push(1);  llist.push(0);  llist.push(2);  llist.push(1);  llist.push(1);  llist.push(2);  llist.push(1);  llist.push(2);  System.***out***.println("Linked List before sorting");  llist.printList();  llist.sortList();  System.***out***.println("Linked List after sorting");  llist.printList();  }  **public** **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();  }  **void** sortList() {  **int** count[] = { 0, 0, 0 };  Node ptr = head;  **while** (ptr != **null**) {  count[ptr.data]++;  ptr = ptr.next;  }  **int** i = 0;  ptr = head;  **while** (ptr != **null**) {  **if** (count[i] == 0)  i++;  **else** {  ptr.data = i;  --count[i];  ptr = ptr.next;  }  }  }  } |

1. StackExampleGeneric

|  |
| --- |
| /\* OUTPUT  \* Popped items: Employee [id=11, name=sam] Employee [id=44, name=sam] Employee  \* [id=33, name=sam] Employee [id=22, name=sam] Employee [id=11, name=sam]  \*/  **public** **class** StackExampleGeneric {  **public** **static** **void** main(String[] args) {  Stack<Employee> stack = **new** Stack<Employee>(10);  stack.push(**new** Employee("11", "sam"));  stack.push(**new** Employee("22", "sam"));  stack.push(**new** Employee("33", "sam"));  stack.push(**new** Employee("44", "sam"));  stack.push(**new** Employee("11", "sam"));  System.***out***.print("Popped items: ");  System.***out***.print(stack.pop() + " ");  System.***out***.print(stack.pop() + " ");  System.***out***.print(stack.pop() + " ");  System.***out***.print(stack.pop() + " ");  System.***out***.print(stack.pop() + " ");  }  }  **class** Employee {  **private** String id;  **private** String name;  **public** Employee(String id, String name) {  **this**.id = id;  **this**.name = name;  }  @Override  **public** String toString() {  **return** "Employee [id=" + id + ", name=" + name + "] ";  }  }  **class** StackFullException **extends** RuntimeException {  **public** StackFullException() {  **super**();  }  **public** StackFullException(String message) {  **super**(message);  }  }  **class** StackEmptyException **extends** RuntimeException {  **public** StackEmptyException() {  **super**();  }  **public** StackEmptyException(String message) {  **super**(message);  }  }  **class** Stack<T> {  **private** **int** size;  **private** T[] stackAr;  **private** **int** top;  @SuppressWarnings("unchecked")  **public** Stack(**int** size) {  **this**.size = size;  stackAr = (T[]) **new** Object[size];  top = -1;  }  **public** **void** push(T value) {  **if** (isFull()) {  **throw** **new** StackFullException("Cannot push " + value + ", Stack is full");  }  stackAr[++top] = value;  }  **public** T pop() {  **if** (isEmpty()) {  **throw** **new** StackEmptyException("Stack is empty");  }  **return** stackAr[top--];  }  **public** **boolean** isEmpty() {  **return** (top == -1);  }  **public** **boolean** isFull() {  **return** (top == size - 1);  }  } |

1. StackImplementationUsingLinkedList

|  |
| --- |
| **public** **class** StackImplementationUsingLinkedList {  **public** **static** **void** main(String[] args) {  StackLinkedList stackLinkedList = **new** StackLinkedList();  System.***out***.println("INSERTING AT FIRST (TOP) IN STACK IMPLEMENTED USING LINKED LIST ");  stackLinkedList.push(39);  stackLinkedList.displayStack();  stackLinkedList.push(71);  stackLinkedList.displayStack();  stackLinkedList.push(11);  stackLinkedList.displayStack();  stackLinkedList.push(76);  stackLinkedList.displayStack();  System.***out***.println("\nDELETING FROM FIRST (TOP) FROM STACK IMPLEMENTED USING LINKED LIST ");  stackLinkedList.pop();  stackLinkedList.displayStack();  stackLinkedList.pop();  stackLinkedList.displayStack();  stackLinkedList.pop();  stackLinkedList.displayStack();  stackLinkedList.pop();  stackLinkedList.displayStack();  }  }  **class** StackLinkedList {  LinkedList12 linkedList = **new** LinkedList12();  **public** **void** push(**int** value) {  linkedList.insertFirst(value);  }  **public** **void** pop() **throws** StackEmptyException {  **try** {  linkedList.deleteFirst();  } **catch** (LinkedListEmptyException llee) {  **throw** **new** StackEmptyException();  }  }  **public** **void** displayStack() {  System.***out***.print("Displaying Stack > Top to Bottom : ");  linkedList.displayLinkedList();  }  }  **class** LinkedList12 {  **private** Node first;  **public** LinkedList12() {  first = **null**;  }  **public** **void** insertFirst(**int** data) {  Node newNode = **new** Node(data);  newNode.next = first;  first = newNode;  }  **public** Node deleteFirst() {  **if** (first == **null**) {  **throw** **new** LinkedListEmptyException("LinkedList doesn't contain any Nodes.");  }  Node tempNode = first;  first = first.next;  **return** tempNode;  }  **public** **void** displayLinkedList() {  Node tempDisplay = first;  **while** (tempDisplay != **null**) {  tempDisplay.displayNode();  tempDisplay = tempDisplay.next;  }  System.***out***.println();  }  }  /\*  \* INSERTING AT FIRST (TOP) IN STACK IMPLEMENTED USING LINKED LIST Displaying  \* Stack > Top to Bottom : 39 Displaying Stack > Top to Bottom : 71 39  \* Displaying Stack > Top to Bottom : 11 71 39 Displaying Stack > Top to Bottom  \* : 76 11 71 39  \*  \* DELETING FROM FIRST (TOP) FROM STACK IMPLEMENTED USING LINKED LIST Displaying  \* Stack > Top to Bottom : 11 71 39 Displaying Stack > Top to Bottom : 71 39  \* Displaying Stack > Top to Bottom : 39 Displaying Stack > Top to Bottom :  \*/ |

1. Union\_of\_two\_Linked\_Lists\_Hashing

|  |
| --- |
| **public** **class** Union\_of\_two\_Linked\_Lists\_Hashing {  Node head; // head of list  /\* Linked list Node \*/  **class** Node {  **int** data;  Node next;  Node(**int** d) {  data = d;  next = **null**;  }  }  **public** **static** **void** main(String[] args) {  Union\_of\_two\_Linked\_Lists\_Hashing llist1 = **new** Union\_of\_two\_Linked\_Lists\_Hashing();  Union\_of\_two\_Linked\_Lists\_Hashing llist2 = **new** Union\_of\_two\_Linked\_Lists\_Hashing();  Union\_of\_two\_Linked\_Lists\_Hashing union = **new** Union\_of\_two\_Linked\_Lists\_Hashing();  /\* create a linked list 10->15->4->20 \*/  llist1.push(20);  llist1.push(4);  llist1.push(15);  llist1.push(10);  /\* create a linked list 8->4->2->10 \*/  llist2.push(10);  llist2.push(2);  llist2.push(4);  llist2.push(8);  union = union.getUnion(llist1.head, llist2.head);  System.***out***.println("First List is");  llist1.printList();  System.***out***.println("Second List is");  llist2.printList();  System.***out***.println("Union List is");  union.printList();  }  Union\_of\_two\_Linked\_Lists\_Hashing getUnion(Node head1, Node head2) {  HashMap<Integer, Integer> hmap = **new** HashMap<>();  Node n1 = head1;  Node n2 = head2;  Union\_of\_two\_Linked\_Lists\_Hashing result = **new** Union\_of\_two\_Linked\_Lists\_Hashing();  **while** (n1 != **null**) {  **if** (hmap.containsKey(n1.data)) {  **int** val = hmap.get(n1.data);  hmap.put(n1.data, val + 1);  } **else** {  hmap.put(n1.data, 1);  }  n1 = n1.next;  }  **while** (n2 != **null**) {  **if** (hmap.containsKey(n2.data)) {  **int** val = hmap.get(n2.data);  hmap.put(n2.data, val + 1);  } **else** {  hmap.put(n2.data, 1);  }  n2 = n2.next;  }  **for** (**int** a : hmap.keySet()) {  result.append(a);  }  **return** result;  }  **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();  }  **boolean** isPresent(Node head, **int** data) {  Node t = head;  **while** (t != **null**) {  **if** (t.data == data)  **return** **true**;  t = t.next;  }  **return** **false**;  }  **public** **void** append(**int** new\_data) {  **if** (**this**.head == **null**) {  Node n = **new** Node(new\_data);  **this**.head = n;  **return**;  }  Node n1 = **this**.head;  Node n2 = **new** Node(new\_data);  **while** (n1.next != **null**) {  n1 = n1.next;  }  n1.next = n2;  n2.next = **null**;  }  } |

1. Given two Linked Lists, create union and intersection lists that contain union and intersection of the elements present in the given lists

|  |
| --- |
| /\*Given two Linked Lists, create union and intersection lists that contain union and intersection of the elements  present in the given lists.  Input:  List1: 10->15->4->20  lsit2: 8->4->2->10  Output:  Union List: 2->8->20->4->15->10  Time Complexity: O(mn)\*/  **public** **class** Union\_of\_two\_Linked\_Lists\_simple {  Node head; // head of list  **class** Node {  **int** data;  Node next;  Node(**int** d) {  data = d;  next = **null**;  }  }  **public** **static** **void** main(String[] args) {  Union\_of\_two\_Linked\_Lists\_simple llist1 = **new** Union\_of\_two\_Linked\_Lists\_simple();  Union\_of\_two\_Linked\_Lists\_simple llist2 = **new** Union\_of\_two\_Linked\_Lists\_simple();  Union\_of\_two\_Linked\_Lists\_simple unin = **new** Union\_of\_two\_Linked\_Lists\_simple();  /\* create a linked lits 10->15->5->20 \*/  llist1.push(20);  llist1.push(4);  llist1.push(15);  llist1.push(10);  /\* create a linked lits 8->4->2->10 \*/  llist2.push(10);  llist2.push(2);  llist2.push(4);  llist2.push(8);  unin.getUnion(llist1.head, llist2.head);  System.***out***.println("First List is");  llist1.printList();  System.***out***.println("Second List is");  llist2.printList();  System.***out***.println("Union List is");  unin.printList();  }  **void** push(**int** new\_data) {  Node new\_node = **new** Node(new\_data);  new\_node.next = head;  head = new\_node;  }  **void** getUnion(Node head1, Node head2) {  Node t1 = head1, t2 = head2;  **while** (t1 != **null**) {  push(t1.data);  t1 = t1.next;  }  **while** (t2 != **null**) {  **if** (!isPresent(head, t2.data))  push(t2.data);  t2 = t2.next;  }  }  **boolean** isPresent(Node head, **int** data) {  Node t = head;  **while** (t != **null**) {  **if** (t.data == data)  **return** **true**;  t = t.next;  }  **return** **false**;  }  **void** printList() {  Node temp = head;  **while** (temp != **null**) {  System.***out***.print(temp.data + " ");  temp = temp.next;  }  System.***out***.println();  }  } |

1. How to find middle element of a singly linked list in one pass?

|  |
| --- |
| public class LinkedListTest {         public static void main(String args[]) {       LinkedList linkedList = new LinkedList();       LinkedList.Node head = linkedList.head();       linkedList.add( new LinkedList.Node("1"));       linkedList.add( new LinkedList.Node("2"));       linkedList.add( new LinkedList.Node("3"));       linkedList.add( new LinkedList.Node("4"));       LinkedList.Node current = head;       int length = 0;       LinkedList.Node middle = head;           while(current.next() != null){           length++;           if(length%2 ==0){               middle = middle.next();           }           current = current.next();       }           if(length%2 == 1){           middle = middle.next();       }       System.out.println("length of LinkedList: " + length);       System.out.println("middle element of LinkedList : " + middle);           }    } class LinkedList{     private Node head;     private Node tail;       public LinkedList(){         this.head = new Node("head");         tail = head;     }       public Node head(){         return head;     }       public void add(Node node){         tail.next = node;         tail = node;     }       public static class Node{         private Node next;         private String data;         public Node(String data){             this.data = data;         }               public String data() {             return data;         }         public void setData(String data) {             this.data = data;         }         public Node next() {             return next;         }         public void setNext(Node next) {             this.next = next;         }               public String toString(){             return this.data;         }     } }Output: length of LinkedList: 4 middle element of LinkedList : 2 |

1. How to check if linked list contains loop in Java? How to find the starting node of the loop

|  |
| --- |
| public class *LinkedList* {  private Node head;  public LinkedList() { *this*.head = new Node("head"); }  public Node head() { return head;}  public void appendIntoTail(Node *node*) {  Node current = head;  while(current.next() != *null*){  current = current.next();  }  current.setNext(node);  }  public boolean isCyclic(){  Node fast = head;  Node slow = head;  while(fast!= *null* && fast.next != *null*){  fast = fast.next.next;  slow = slow.next;  if(fast == slow ){  return *true*;  }  }  return *false*;  }  @Override  public String toString(){  StringBuilder sb = new StringBuilder();  Node current = head.next();  while(current != *null*){  sb.append(current).append("-->");  current = current.next();  }  sb.delete(sb.length() - *3*, sb.length());  return sb.toString();  }  public static class *Node* {  private Node next;  private String data;  public Node(String *data*) {  *this*.data = data;  }  public String data() { return data; }  public void setData(String *data*) { *this*.data = data;}  public Node next() { return next; }  public void setNext(Node *next*) { *this*.next = next; }  @Override  public String toString() {  return *this*.data;  }  }  } |

1. How to reverse a linked list in Java?

|  |
| --- |
| **public** **class** ReverseSinglyLinkedList {  **public** **static** **void** main(String[] args) {  SinglyLinkedList.Node head = **new** SinglyLinkedList.Node(1);  SinglyLinkedList linkedlist = **new** SinglyLinkedList(head);  linkedlist.add(**new** SinglyLinkedList.Node(2));  linkedlist.add(**new** SinglyLinkedList.Node(3));  linkedlist.print();  linkedlist.reverse();  linkedlist.print();  }  }  **class** SinglyLinkedList {  **static** **class** Node {  **private** **int** data;  **private** Node next;  **public** Node(**int** data) {  **this**.data = data;  }  **public** **int** data() {  **return** data;  }  **public** Node next() {  **return** next;  }  }  **private** Node head;  **public** SinglyLinkedList(Node head) {  **this**.head = head;  }  **public** **void** add(Node node) {  Node current = head;  **while** (current != **null**) {  **if** (current.next == **null**) {  current.next = node;  **break**;  }  current = current.next;  }  }  **public** **void** print() {  Node node = head;  **while** (node != **null**) {  System.***out***.print(node.data() + " ");  node = node.next();  }  System.***out***.println("");  }  **public** **void** reverse() {  Node pointer = head;  Node previous = **null**, current = **null**;  **while** (pointer != **null**) {  current = pointer;  pointer = pointer.next;  current.next = previous;  previous = current;  head = current;  }  }  }  /\*  \* Output 1 2 3  \* 3 2 1  \*/ |

1. How to reverse a singly linked list without recursion in Java?

|  |
| --- |
|  |

1. How would you remove a node from a doubly linked list?

|  |
| --- |
|  |

1. Write a Program to convert a binary tree into a doubly linked list?

|  |
| --- |
|  |

1. How to remove duplicate nodes in an unsorted linked list?

|  |
| --- |
|  |

1. Write a recursive method to determine whether a linked list is sorted in descending order or not? If it's sorted then return true otherwise return false.

|  |
| --- |
|  |

1. How to find the length of a singly linked list in Java?

|  |
| --- |
|  |

1. Write code to print out the data stored in each node in a singly linked list?

|  |
| --- |
|  |

1. Write a Program to print a linked list in reverse order? E.g. Print linked list from tail to head?

|  |
| --- |
|  |