**Linked list questions**

1. Write a java method that receives the head of a linked list then check if the linked list sorted or not (sorted in any manner)

Ex. [ 1 2 3 4 5 6 7] 🡪 sorted ascending order

[ 1 2 5 8 7] 🡪 not sorted

[ 9 5 3 2 1] 🡪 sorted in descending order

**The solution should be in O(N)**

**// Java program to check Linked List is sorted**

**// in descending order or not**

|  |
| --- |
| // Java program to check Linked List is sorted  // in descending order or not  **class** GFG  {    /\* Linked list node \*/  **static** **class** Node  {  **int** data;      Node next;  };    // function to Check Linked List is  // sorted in descending order or not  **static** **boolean** isSortedDesc(Node head)  {  **if** (head == **null**)  **return** **true**;        // Traverse the list till last node and return      // false if a node is smaller than or equal      // its next.  **for** (Node t = head; t.next != **null**; t = t.next)  **if** (t.data <= t.next.data)  **return** **false**;  **return** **true**;  }  **static** **boolean** isSortedAsce(Node head)  {  **if** (head == **null**)  **return** **true**;        // Traverse the list till last node and return      // false if a node is smaller than or equal      // its next.  **for** (Node t = head; t.next != **null**; t = t.next)  **if** (t.data >= t.next.data)  **return** **false**;  **return** **true**;  }    **static** Node newNode(**int** data)  {      Node temp = **new** Node();      temp.next = **null**;      temp.data = data;  **return** temp;  }    // Driver Code  **public** **static** **void** main(String[] args)  {      Node head = newNode(7);      head.next = newNode(5);      head.next.next = newNode(4);      head.next.next.next = newNode(3);    **if**(isSortedDesc(head))          System.out.println("Yes is desc");  Else **if**(isSortedAsce (head))          System.out.println("Yes is Asce");  **else**          System.out.println("No");  }  } |

1. Write a program of two methods that receives the head of a linked list then convert binary number found to an integer

**First method:** receives the head of a linked list, find the binary number stored in it as a String and pass it to the other method.

**Note**: The linked list could contain any digit not only 0’s and 1’s. The nodes with any number either than 0 or 1 should be ignored.

The series of 0’s and 1’s found in the linked list should be considered as a binary number represented as a string and passed to method2.

**Second method:** should receive the String represents the binary number and convert it to the corresponding decimal representation.

Ex.

Input: [ 1 0 0 1 5 1 ]

The binary number is: 10011

The equivalent decimal is: 19

Ex.

Input: [ 5 8 1 4 0 7]

The binary number is: 10.

The equivalent decimal is: 2

Ex. [2 5 4 7]

The binary number is: No binary number found

The equivalent decimal: No decimal number calculated.

1. Given a linked list containing 0's, 1's, and 2's, sort the linked list by doing a single traversal of it. (The time complexity should be O(N))

For example,

Input:  0 —> 1 —> 2 —> 2 —> 1 —> 0 —> 0 —> 2 —> 0 —> 1 —> 1 —> 0 —> NULL  
   
Output: 0 —> 0 —> 0 —> 0 —> 0 —> 1 —> 1 —> 1 —> 1 —> 2 —> 2 —> 2 —> NULL

|  |
| --- |
| // Java program to sort a linked list of 0, 1 and 2  **class** LinkedList  {      Node head;  // head of list        /\* Linked list Node\*/  **class** Node      {  **int** data;          Node next;          Node(**int** d) {data = d; next = **null**; }      }    **void** sortList()      {         // initialise count of 0 1 and 2 as 0  **int** count[] = {0, 0, 0};           Node ptr = head;           /\* count total number of '0', '1' and '2'          \* count[0] will store total number of '0's          \* count[1] will store total number of '1's          \* count[2] will store total number of '2's  \*/  **while** (ptr != **null**)         {              count[ptr.data]++;              ptr = ptr.next;         }    **int** i = 0;         ptr = head;           /\* Let say count[0] = n1, count[1] = n2 and count[2] = n3          \* now start traversing list from head node,          \* 1) fill the list with 0, till n1 > 0          \* 2) fill the list with 1, till n2 > 0          \* 3) fill the list with 2, till n3 > 0  \*/  **while** (ptr != **null**)          {  **if** (count[i] == 0)                  i++;  **else**              {                 ptr.data= i;                 --count[i];                 ptr = ptr.next;              }           }      }          /\* Utility functions \*/        /\* Inserts a new Node at front of the list. \*/  **public** **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;      }        /\* Function to print linked list \*/  **void** printList()      {          Node temp = head;  **while** (temp != **null**)          {             System.out.print(temp.data+" ");             temp = temp.next;          }          System.out.println();      }         /\* Driver program to test above functions \*/  **public** **static** **void** main(String args[])      {          LinkedList llist = **new** LinkedList();            /\* Constructed Linked List is 1->2->3->4->5->6->7->             8->8->9->null \*/          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();      }  } |

1. Given a single-digit number k and a singly linked list whose nodes stores digits of a non-negative number, add k to the linked list.

For example, consider the linked list 9 —> 9 —> 9 —> 3 —> NULL which represents the number 9993. Adding a single-digit number 7 to it should result in the linked list 1 —> 0 —> 0 —> 0 —> 0 —> NULL which corresponds to the number 10000.

|  |
| --- |
| // Java implementation of the approach    // Node structure containing data  // and pointer to the next Node  **class** node  {  **int** key;      node next;        node(**int** n)      {          key = n;          next = **null**;      }  };    // Linked list class  **class** LinkedList  {  **static** node head;        // Default constructor for      // creating empty list  **public** LinkedList()      {          // Empty List          head = **null**;      }        // Function to insert a node at the      // head of the linked list  **void** insert(node n)      {            // Empty List  **if** (head == **null**)              head = n;            // Insert in the beginning of the list  **else**          {              n.next = head;              head = n;          }      }        // Function to print the linked list  **void** printList()      {          node ptr = head;    **while** (ptr != **null**)          {              System.out.print(ptr.key + "->");              ptr = ptr.next;          }          System.out.print("null" + "\n");      }        // Function to add a digit to the integer      // represented as a linked list  **void** addDigit(**int** n)      {            // To keep track of the last node          // whose value is less than 9          node lastNode = **null**;          node curr = head;    **while** (curr.next != **null**)          {                // If found a node with value              // less than 9  **if** (curr.key < 9)                  lastNode = curr;                // Otherwise keep traversing              // the list till end              curr = curr.next;          }            // Add the given digit to the last node          curr.key = curr.key + n;            // In case of overflow in the last node  **if** (curr.key > 9)          {              curr.key = curr.key % 10;                // If the list is of the              // form 9.9.9....  **if** (lastNode == **null**)              {                    // Insert a node at the beginning as                  // there would be overflow in the                  // head in this case                  insert(**new** node(1));                    // Adjust the lastNode pointer to                  // propagate the carry effect to                  // all the nodes of the list                  lastNode = head.next;              }                // Forward propagate carry effect  **while** (lastNode != curr)              {                  lastNode.key = (lastNode.key + 1) % 10;                  lastNode = lastNode.next;              }          }      }        // Driver code  **public** **static** **void** main(String[] args)      {            // Creating the linked list          LinkedList l1 = **new** LinkedList();            // Adding elements to the linked list          l1.insert(**new** node(9));          l1.insert(**new** node(9));          l1.insert(**new** node(1));            // Printing the original list          l1.printList();            // Adding the digit          l1.addDigit(5);            // Printing the modified list          l1.printList();      }  } |

1. Write a method that receives the head of two linked lists then find the joint node between them.

Ex.

A picture containing text, clock

Description automatically generated

**Output: 15**

**class** GFG {

**static** **class** Node {

**int** data;

        Node next;

        Node(**int** d)

        {

            data = d;

            next = **null**;

        }

    }

    /\* function to get the intersection point of two linked

    lists head1 and head2 \*/

**public** Node getIntersectionNode(Node head1, Node head2)

    {

**while** (head2 != **null**) {

            Node temp = head1;

**while** (temp != **null**) {

                // if both Nodes are same

**if** (temp == head2) {

**return** head2;

                }

                temp = temp.next;

            }

            head2 = head2.next;

        }

        // If intersection is not present between the lists,

        // return NULL.

**return** **null**;

    }

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

    {

        GFG list = **new** GFG();

        Node head1, head2;

        /\*

                Create two linked lists

                1st 3->6->9->15->30

                2nd 10->15->30

                15 is the intersection point

        \*/

        head1 = **new** Node(10);

        head2 = **new** Node(3);

        Node newNode = **new** Node(6);

        head2.next = newNode;

        newNode = **new** Node(9);

        head2.next.next = newNode;

        newNode = **new** Node(15);

        head1.next = newNode;

        head2.next.next.next = newNode;

        newNode = **new** Node(30);

        head1.next.next = newNode;

        head1.next.next.next = **null**;

        Node intersectionPoint

            = list.getIntersectionNode(head1, head2);

**if** (intersectionPoint == **null**) {

            System.out.print(" No Intersection Point \n");

        }

**else** {

            System.out.print("Intersection Point: "

                             + intersectionPoint.data);

        }

    }

}