InterviewBit

Linked List Interview Questions

To view the live version of the page, [click here.]()

Contents

Linked List Interview Questions for Freshers

Linked List Interview Questions for Experienced

(.....Continued)

19. Given a linked list, find the length of the longest palindrome list that appears in that linked list using O(1) extra space.
20. Given a singly linked list with an additional "arbitrary" pointer at each node that currently points to NULL. What algorithm will you implement to make the "arbitrary" pointer point to the next node with a greater value?
21. In a standard Doubly Linked List, two address fields are required to contain the addresses of previous and next nodes. Can you create a doubly linked list using only one space for the address field with every node?
22. Given a linked list with each node representing a linked list and two pointers of its type given below. You need to flatten the lists into a single linked list. The flattened linked list also needs to be sorted. Discuss the approach.

Let's get Started

What is Linked List?

Linked Lists, like arrays, are linear data structures. Unlike arrays, linked list elements are not stored in a single location; instead, pointers are used to connect the elements. Learn More.

What is the purpose of a Linked list?

Arrays can be used to store comparable forms of linear data, however, they have the following drawbacks.

- Because the size of the arrays is fixed, we must know the maximum number of elements ahead of time. In addition, regardless of consumption, the allocated memory is always equal to the higher limit.
- Adding a new element to an array of elements is costly because space must be made for the new components, which requires current elements to be relocated. Deletion in arrays is also expensive unless you employ some particular procedures.

Advantages of Linked List:

- A linked list is a dynamic data structure that can grow and shrink in size at runtime by allocating and deallocating memory. As a result, there is no need to specify the linked list's initial size.
- There is no memory wastage in the linked list since the size of the linked list increases or decreases at run time, hence there is no memory wastage and no need to pre-allocate memory.
- A linked list is frequently used to build linear data structures such as stacks and queues.
- The linked list makes it much easier to insert and delete items. After an element is inserted or deleted, there is no need to move it; only the address in the next pointer needs to be updated.

Disadvantages of Linked List:

- A linked list requires more memory than an array. Because a pointer is necessary to store the address of the next entry in a linked list, it consumes additional memory.
- Traversing a linked list takes longer than traversing an array. A linked list, unlike an array by index, does not provide direct access to an entry. To get to a mode at position n, for example, you have to go through all the nodes before it.
- Reverse traversing is not possible in a singly linked list, but it is possible in a doubly-linked list since each node carries a pointer to the previously connected nodes. Extra memory is necessary for the back pointer to execute this, resulting in memory waste.
- Because of the dynamic memory allocation in a linked list, random access is not possible.

Representation:

A pointer to the first node of a linked list is used to represent it. The head is the first node in the chain. The value of the head is NULL if the linked list is empty.
A list's nodes are made up of at least two parts:

- Data/Value
- A pointer to the next node (or a reference to it).

Structures can be used to represent a node in C. A linked list node containing integer data is shown below.

A LinkedList can be written as a class in Java or C#, and a Node can be expressed as a separate class. A reference to the Node class type can be found in the LinkedList class.

Brief History:

Allen Newell, Cliff Shaw, and Herbert A. Simon of RAND Corporation created linked lists as the basic data structure for their Information Processing Language in 1955–1956. The authors utilized IPL to create the Logic Theory Machine, the General Problem Solver, and a computer chess software, among other early artificial intelligence applications. Newell and Shaw's "Programming the Logic Theory Machine" contains the now-classic diagram of blocks representing list nodes with arrows pointing to subsequent list nodes.

Hans Peter Luhn, who recommended the use of linked lists in chained hash tables in an internal IBM memo in January 1953, was another early proponent of linked lists. The efficacy of linked lists and languages that employ these structures as their principal data representation was well established by the early 1960s.

Linked List Interview Questions for Freshers

1. How will you find the middle element of a singly linked list without iterating the list more than once?

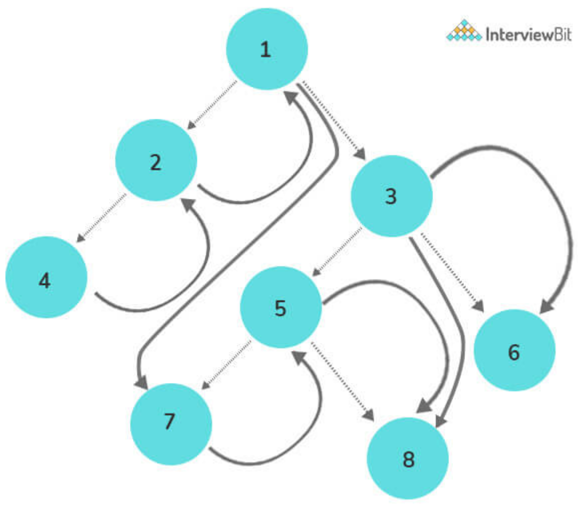
To solve this problem, we can use the two-pointer method. You have two pointers, one fast and one slow, in the two-pointer approach. The fast pointer travels two nodes per step, while the slow pointer only moves one. The slow pointer will point to the middle node of the linked list when the fast pointer points to the last node, i.e. when the next node is null.

```
Node* getMiddle(Node *head)
{
     struct Node *slow = head;
     struct Node *fast = head;

     if (head)
     {
         while (fast != NULL && fast->next != NULL)
         {
             fast = fast->next->next;
             slow = slow->next;
         }
     }
     return slow;
}
```

Time Complexity: O(n)
Space Complexity: O(1)

2. How will you convert a binary tree into a doubly-linked list?

In a converted doubly linked list, the left and right pointers in nodes of a binary tree will be used as previous and next pointers, respectively. The nodes in the doubly linked list must be in the same order as the provided Binary Tree's Inorder. The head node of the doubly linked list must be the first node of Inorder traversal (leftmost node in the binary tree).



Inorder : 4,2,1,7,5,8,3,6

```
void btToDLL(Node *root, Node **head_reference, Node **previous)
{
   if (!root)
      return;

   // Recursive conversion of left subtree
   btToDLL(root->left, head_reference, previous);

   if (*head_reference == NULL)
       *head_reference = root;
   else
   {
       root->left = *previous;
       *previous->right = root;
   }
   *previous = root;

   // Recursive conversion of right subtree
   btToDLL(root->right, head_reference, previous);
}
```

Time Complexity: O(n)
Auxiliary Space: O(1)

3. How will you remove a cycle from a linked list?

One method of identifying the cycle is Floyd's cycle detect technique, popularly known as the tortoise and hare algorithm since it uses two pointers/references that move at opposite speeds. If there is a cycle, after a limited number of steps, the two pointers (say, slow and fast) will point to the same element.

It's interesting to note that the element where they meet will be the same distance from the loop's start (continuing to traverse the list in the same, forward direction) as the loop's start is from the list's head. That is, if the linear component of the list contains k elements, the two pointers will meet inside a loop of length m at a location m-k from the loop's start or k elements from the loop's 'end' (of course, it's a loop, so there is no 'end' - it's just the ‘start' again). That gives us a technique to find the loop's beginning.

Once a cycle has been detected, keep fast pointing to the element where the loop for the previous step ended, but reset slow to point back to the beginning of the list. Now, one element at a time, move each pointer. Fast will keep looping because it started inside the loop. Slow and fast will meet again after k steps (equivalent to the distance between the start of the loop and the head of the list). This will serve as a pointer to the beginning of the loop.

It's now simple to set slow (or fast) to point to the loop's starting element and traverse the loop until slow returns to the starting element. Slow is referring to the 'last' element list at this point, and its next pointer can be adjusted to null.

Implementation:

```
Node* getLastNode(Node* head)
{
   Node* slow = head;
   Node* fast = head;
   // find the intersection point using Tortoise and Hare algorithm
   while (fast->next != NULL)
   {
       slow = slow->next;
       fast = fast->next->next;
       if (slow == fast)
           break;
   }
   //check if there is no loop
   if (fast->next == NULL)
   {
       return NULL;
   }
   slow = head;
   // run this loop till both the references are one short of the start of the loop
   while (slow->next != fast->next)
   {
       slow = slow->next;
       fast = fast->next;
   }
   // Now the fast pointer is pointing to the start of the loop
   return fast;
}
Node* getLastNode = findStartOfLoop(head);
getLastNode->next = null;
```

Time Complexity: O(n)
Space Complexity: O(1)

4. What algorithm will you implement to find similar elements from two Linked Lists given and return the result in the form of a Linked List? Assume there are no duplicates.

Create an empty hash table and set the result list to NULL. While traversing List1, insert the element in the hash table for each element visited in List1. While traversing List2, look for the entries in the hash table for each element visited in List2. If the element is already existing, add it to the result list. If the element isn't present, it is to be ignored.

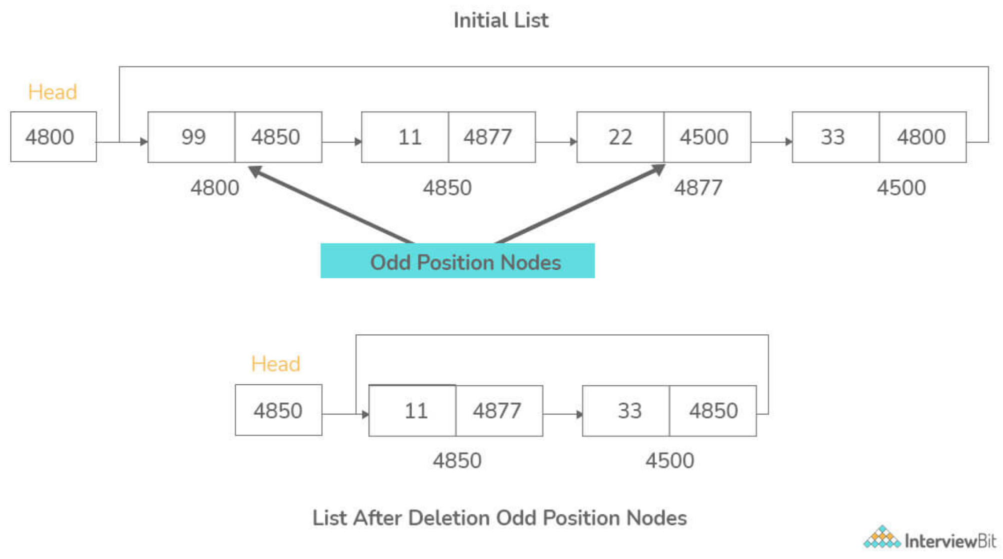
The overall time and space complexity are linear.

```
Node* getIntersection(Node* head1, Node* head2)
{
       unordered_map < int > m;
       Node* n1 = head1;
       Node* n2 = head2;
       Node* head = NULL;

       // loop stores all the elements of list1 in hset
       while (n1)
       {
           m[n1->value] = 1;
           n1 = n1->next;
       }

       // For every element of list2 present in hset
       // loop inserts the element into the result
       while (n2 != null)
       {
           if (m[n2->value] == 1)
           {
               Node* temp = new Node();
               temp->value = n2->value;
               temp->next = head;
               head = temp;
           }
           n2 = n2->next;
       }
       return head;
}
```

5. Why is merge sort a better option than quicksort for linked lists?

- When it comes to linked lists, there are a few things to keep in mind. The issue is unique due to the memory allocation differences between arrays and linked lists. Unlike arrays, linked list nodes in memory may not be adjacent.
- We can insert items in the middle of a linked list in O(1) extra space and O(1) time if we are given a reference/pointer to the previous node, unlike an array. As a result, the merge sort operation can be accomplished without the need for additional linked list space.
- We can do random access in arrays since the elements are continuous in memory. In contrast to arrays, we can't access a linked list at random.
- Quick Sort necessitates a great deal of this type of access. Because we don't have a continuous block of memory, we have to travel from the head to the i'th node to get to the i'th index in a linked list. Merge sort accesses data in a sequential manner, with less requirement for random access.

6. Write a program to delete all odd positioned nodes from a circular linked list. (Consider 1-based indexing).



The approach is to begin traversing the circular linked list by keeping track of the current node's position using a count variable. Delete the current node if it is at an odd position.

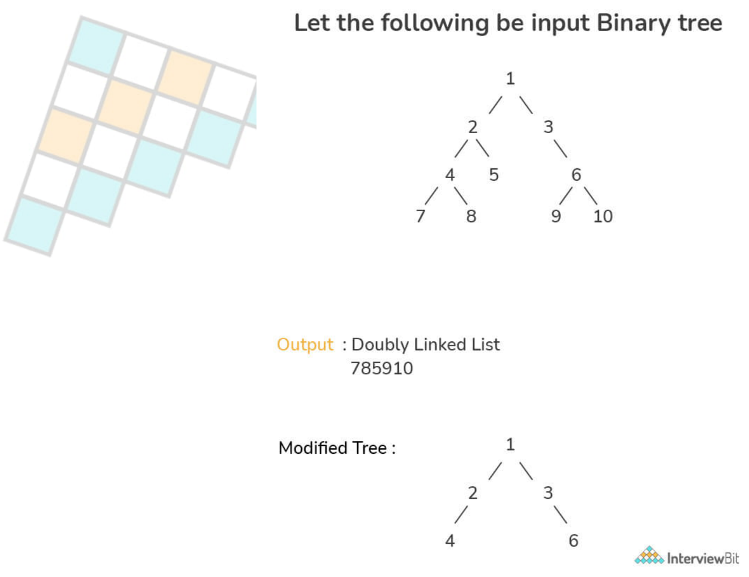
Implementation:

```
// l is the length of the linked list
void DeleteAllOddNodes(struct Node** head_reference, int l)
{
   int cnt = 0;
   struct Node *prev = *head_reference, *next = *head_reference;

   // check if the list is empty
   if (*head_reference == NULL)
   {
       cout<<"List is empty"<<endl;
       return;
   }

   // check if there is a single node in the list
   if (l == 1)
   {
       // Function to delete first node
       *head_reference=NULL;
       return;
   }

   while (l > 0)
   {
       // delete first position node as it is odd
       if (cnt == 0)
       {
           struct Node *t1 = *head_reference, *t2 = *head_reference;
           if (t1->next == t1)
           {
              *head_reference = NULL;
           }
           while(t1->next!=*head_reference)
           {
              t1 = t1->next;
              t2 = t1->next;
           }
           t1->next = t2->next;
           *head_reference = t1->next;
           free(t2);
       }

       // if the position is odd, delete that node
       if (cnt % 2 == 0 && cnt != 0)
       {
           struct Node* tmp = head_reference;
           if (head_reference == prev)
           {
               head_reference = prev->next;
           }
           while (tmp->next != prev)
           {
               tmp = tmp->next;
           }
           tmp->next = prev->next;
           free(prev);
       }
```

Time Complexity: O(n^2)
Space Complexity: O(1)

7. Extract all leaves from a Binary Tree into a Doubly Linked List (DLL).

It's worth noting that the DLL must be created in place. Assume that the DLL and Binary Tree node structures are identical, with the exception that the meanings of the left and right pointers. Left denotes the previous pointer, whereas right denotes the next pointer in DLL.


Let the following be input Binary tree
1
2
3
4
5
6
7
8
9
10
Output : Doubly Linked List
785910
Modified Tree :
1
2
3
4
6
InterviewBit

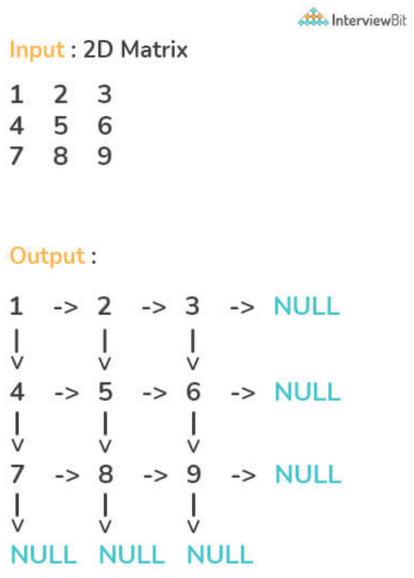
All of the leaves must be traversed and connected by adjusting the left and right pointers. By adjusting the left or right pointers in parent nodes, we can also delete them from the Binary Tree. There are numerous options for resolving this issue. We add leaves to the beginning of the current linked list and update the list's head using the pointer to head pointer in the following code. We must process leaves in reverse order because we insert from the beginning. We traverse the right subtree first, then the left subtree, in reverse order. To update the left and right pointers in parent nodes, we employ return values.

Implementation:

```
// The function extracts all the
// leaves from a given Binary Tree.
// The function returns new root of
// the Binary Tree. The function also sets
// *head_reference as the head of the DLL.
// The left pointer of the tree is used as prev
// and the right pointer is used as next in the DLL.
Node* extractLeaf(Node **head_reference, Node *root)
{
  if (!root)
      return NULL;

  if (root->right == NULL && root->left == NULL)
  {
      // This node will be added to the doubly linked
      // list of leaves. We will have to
      // set the right pointer of this node
      // as the previous head of DLL. We
      // don't need to set the left pointer
      // as the left is already NULL.
      root->right = *head_reference;

      // Change the left pointer of previous head
      if (*head_reference)
         (*head_reference)->left = root;

      // Change the head of the linked list
      *head_reference = root;

      return NULL;
  }

  // Recursion for right and left subtrees
  root->right = extractLeaf(&head_reference, root->right);
  root->left = extractLeaf(&head_reference, root->lef);

  return root;
}
```

Time Complexity: O(n)
Space Complexity: O(1), ignoring the recursion stack space

8. Given a 2-D matrix. You need to convert it into a linked list matrix such that each node is linked to its next right and down node and display it.



Input : 2D Matrix

```
1  2  3
4  5  6
7  8  9
```

Output :

```
1    ->  2    ->  3    ->  NULL
|        |        |
v        v        v
4    ->  5    ->  6    ->  NULL
|        |        |
v        v        v
7    ->  8    ->  9    ->  NULL
|        |        |
v        v        v
NULL     NULL     NULL
```

The idea is to create a new node for each element of the matrix and then create its next down and right nodes in a recursive manner.

Implementation:

```
Node* construct(int A[][3], int m, int n, int i, int j)
{
   // check if i or j is out of bounds
   if (i > n - 1 || j > m - 1)
       return NULL;

   // a new node for current i and j is created
   // and its down and right pointers are
   //recursively allocated
   Node* t = new Node();
   t->value = A[i][j];
   t->right = construct(A, m, n, i, j + 1);
   t->down  = construct(A, m, n, i + 1, j);
   return t;
}

// function to display linked list data
void printData(Node* head)
{
   // pointer to move down
   Node* d;
   // pointer to move down

   Node* r;

   // loop till node->down is not NULL
   for( d = head; d!=NULL; d = d->down)
   {
       for( r = d; r!=NULL; r = r->right)
       {
       // loop till node->right is not NULL
           cout << r->value << " ";

       }
       cout << endl;

   }
}
```

Time Complexity: O(m * n)
Space Complexity: O(1), ignoring the space required for the final answer.

9. The task is to determine pairs in a doubly-linked list whose sum equals the provided value ‘val’ without consuming any extra space for a given sorted doubly-linked list of positive distinct entries. The expected complexities are O(n) time and O(1) space.

The approach is as follows:

- Two pointer variables are to be initialized to find the possible elements in the sorted doubly linked list. Initialize num1 with the head of the doubly linked list,i.e., num1=head, and num2 with the last node of the doubly linked list, i.e., num2=lastNode.
- If the current sum of num1 and num2 is less than Val, then we advance num1 in the forward direction. If the current total of the num1 and num2 is greater than x, then num2 is moved in the backward direction.
- When the two pointers cross each other (num2->next = num1) or they become equal (num1 == num2), the loop ends. The condition "num1==num2" will handle the circumstance where no such pairs are present.

Implementation:

```
vector<pair<int,int>> sumPair(struct Node *head, int val)
{
   // Two pointers are to be set, one to the beginning
   // and the other to the last of the DLL.
   struct Node *num1 = head;
   struct Node *num2 = head;
   while (num2->next != NULL)  //to get to the last node
       num2 = num2->next;

   vector<pair<int,int>> ans;
   // The loop ends when two pointers
   // cross each other or they are equal
   while (num1 != num2 && num2->next != num1)
   {
       if ((num1->value + num2->value) == val)
       {
           ans.push_back(make_pair(num1->value,num2->value));

           // move num1 in the forward direction
           num1 = num1->next;

           // move num2 in the backward direction
           num2 = num2->prev;
       }
       else
       {
           if ((num1->value + num2->value) > val)
               num2 = num2->prev;
           else
               num1 = num1->next;
       }
   }

  return ans;
}
```

Time Complexity: O(n)
Space Complexity: O(1)

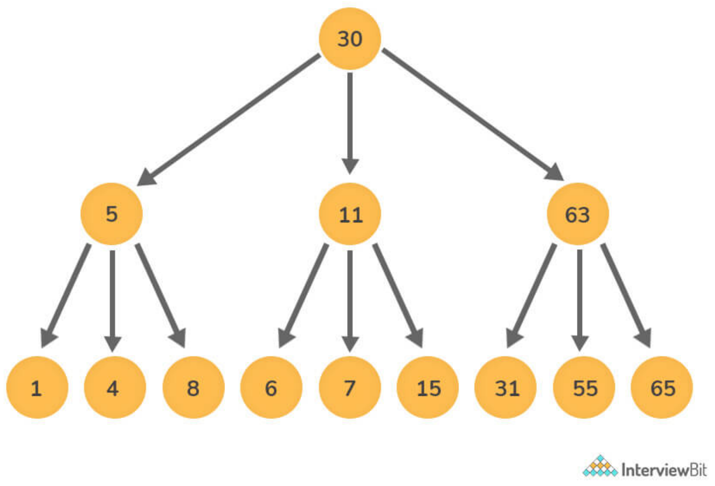
10. Construct a doubly linked list out of a ternary tree.

A ternary tree is similar to a binary tree, except that instead of two nodes, it has three: left, middle, and right.

The attributes of the doubly linked list should be as follows:

- The ternary tree's left pointer should be used as the previous pointer in a doubly-linked list.
- The ternary tree's middle pointer should not point to anything.
- The ternary tree's right pointer should be the doubly linked list's next pointer.
- Each ternary tree node is entered into a doubly-linked list before its subtrees, with the left child of each node being inserted first, followed by the middle and right child (if any).

For the above example, the linked list for below tree should be NULL <-30<->5<->1<->4<->8<->11<->6<->7<->15<->63<->31<->55<->65->NULL



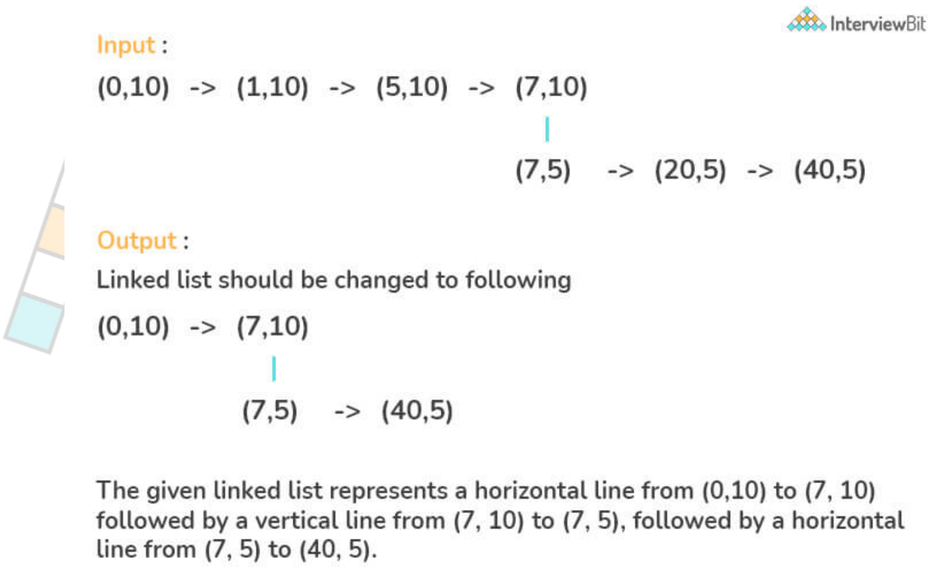
The approach is to make a preorder traversal of the tree. When we visit a node, we will use a tail pointer to insert it into a doubly-linked list at the end. That's how we keep the required insertion order. Then, in that order, we call for the left child, middle child, and right child.

Implementation:

```
//Utility function that creates a doubly linked list
//by inserting the current node at the end of the doubly
//linked list by employing a tail pointer
void push(Node** tail_reference, Node* n)
{
   // the tail pointer is to be initialized
   if (*tail_reference == NULL)
   {
       *tail_reference = n;

       // set left, middle and right child to point
       // to NULL
       n->left = NULL;
       n->middle = NULL;
       n->right = NULL;

       return;
   }

   // using tail pointer, insert node in the end
   (*tail_reference)->right = n;
   // the middle and right child are set to point to NULL
   n->right = NULL;
   n->middle = NULL;

   // set previous of the node
   n->left = (*tail_reference);


   // now tail pointer is pointing to the inserted node
   (*tail_reference) = n;
}

// From a ternary tree, create a doubly linked list,
// by making a preorder traversal of the tree
Node* ternaryTreeToList(Node** head_reference, Node* root)
{
   // Base case
   if (!root)
       return NULL;

   //a static tail pointer to be created
   static Node* tail = NULL;

   // left, middle and right nodes to be stored
   // for future calls.
   Node* left = root->left;
   Node* middle = root->middle;
   Node* right = root->right;

   // set the head of the doubly linked list
   // as the root of the ternary tree
   if (*head_reference == NULL)
       *head_reference = root;

   // push the current node in the end of the doubly linked list
   push(&tail, root);
```

Time Complexity: O(n)
Space Complexity: O(1)

11. A linked list of coordinates is given, with neighboring points forming either a vertical or horizontal line. Remove points in between the starting and ending points of the horizontal or vertical line from the linked list.



Input :

```
(0,10) -> (1,10) -> (5,10) -> (7,10)
                                |
                              (7,5) -> (20,5) -> (40,5)
```

Output :

Linked list should be changed to following

```
(0,10) -> (7,10)
            |
          (7,5) -> (40,5)
```

The given linked list represents a horizontal line from (0,10) to (7, 10) followed by a vertical line from (7, 10) to (7, 5), followed by a horizontal line from (7, 5) to (40, 5).

The objective is to keep track of the current node, the next node, and the node after that. Continue deleting the next node when it is the same as the next-next node. We must keep a watch on the shifting of pointers and check for NULL values throughout the entire procedure.

Time Complexity is O(n).
Space Complexity is O(1)

Implementation:

```
void deleteNode(Node *head, Node *t)
{
   head->next = t->next;
   t->next = NULL;
   free(t);
}

// This function deletes the intermediate nodes in a sequence of
// horizontal and vertical line segments represented by a
// linked list.
Node* deleteMiddleNodes(Node *head)
{
   if (head == NULL || head->next == NULL || head->next->next == NULL)
       return head;

   Node* t = head->next;
   Node *tt = t->next;

   // Check whether this is a vertical line or horizontal line
   if (t->y == head->y)
   {
       // Find intermediate nodes with same y value, and delete them
       while (tt != NULL && t->y == tt->y)
       {
           deleteNode(head, t);

           // Update t and tt for the next iteration
           t = tt;
           tt = tt->next;
       }
   }
       // check for vertical line
   else if (t->x == head->x)
   {
       // Find intermediate nodes with same x value, and delete them
       while (tt != NULL && t->x == tt->x)
       {
           deleteNode(head, t);

           // Update t and tt for the next iteration
           t = tt;
           tt = tt->next;
       }
   }
   // Adjacent points must either have same x or same y
   else
   {
       cout<<"Given linked list is not valid";
       return NULL;
   }

   // Recursion for the next segment
   deleteMiddleNodes(head->next);

   return head;
}
```

12. How will you modify a linked list of integers so that all even numbers appear before all odd numbers in the modified linked list? Also, keep the even and odd numbers in the same order.

Example:
Input: 17->15->8->12->10->5->4->1->7->6->NULL
Output: 8->12->10->4->6->17->15->5->1->7->NULL

Algorithm:
The approach is to divide the linked list into two sections, one with all even nodes and the other with all odd nodes. In order to split the Linked List, traverse the original Linked List and move all odd nodes to a new Linked List. The original list will include all the even nodes at the end of the loop, while the odd node list will have all the odd nodes. We must place all the odd nodes at the end of the odd node list to maintain the same ordering of all nodes. And, in order to do it in real-time, we'll need to maintain track of the last pointer in the odd node list. Finally, the odd node linked list is to be attached after the even node linked list.

Implementation:

```
Node* separateEvenOdd(struct Node *head)
{
   // Starting node of the list having
   // even values
   Node *evenStart = NULL;

   // Starting node of the list having odd values
   Node *oddStart = NULL;

   // Ending node of the list having even values
   Node *evenEnd = NULL;

   // Ending node of the list having odd values
   Node *oddEnd = NULL;

   // Node for list traversal.
   Node *currentNode;

   for(currentNode = head; currentNode != NULL; currentNode = currentNode -> next)
   {
       int value = currentNode -> value;

       // If the current value is even, add
       // it to the list of even values.
       if(value % 2 != 0)
       {
          if(oddStart != NULL)
           {
               oddEnd -> next = currentNode;
               oddEnd = oddEnd -> next;
           }
           else
           {
               oddStart = currentNode;
               oddEnd = oddStart;
           }
       }

       // If current value is odd, add
       // it to the list of odd values.
       else
       {
           if(evenStart != NULL)
           {
               evenEnd -> next = currentNode;
               evenEnd = evenEnd -> next;
           }

           else
           {
              evenStart = currentNode;
              evenEnd = evenStart;
           }
       }

  }
```

Time Complexity: O(n)
Space Complexity: O(1)

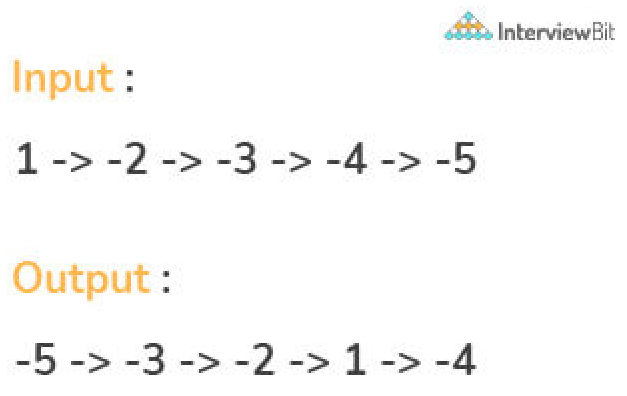
13. Given a linked list and a number n, you have to find the sum of the last n nodes of the linked list in a single traversal. Explain your approach in brief.

The use of two-pointers will require a single traversal. We will have to maintain two pointers – reference pointer and main pointer. Both these pointers will be initialized to head. First, the reference pointer will be moved to n nodes from the head, and while traversing, we will keep adding the values and store them into a variable called sum1. Now both the pointers will move simultaneously until the reference pointer reaches the end of the list and while traversing, we will keep adding the values of the nodes. The reference pointer is storing this sum in the variable sum1, and the main pointer will be storing it in sum2. Now, (sum1 – sum2) is the answer, that is the required sum of the last n nodes.

```
int getSum(Node* head, int n)
{
   if (n <= 0)
       return 0;

   int sum1 = 0, sum2 = 0;
   struct Node* ptr1 = head;
   struct Node* ptr2 = head;

   // the sum of the first n nodes is to be calculated
   for (ptr1 = head; ptr1 != NULL; ptr1 = ptr1->next;)
   {
       sum += ptr1->value;
       n--;
       if(n == 0)
         break;
   }
   // now there is a distance of n nodes between the two pointers
   // move to the end of the linked list
   while (ptr1 != NULL)
   {
       // sum of all the nodes
       sum1 += ptr1->value;
       // sum of the first length -  n nodes
       sum2 += ptr2->value;

       ptr1 = ptr2->next;
       ptr2 = ptr2->next;
   }

   // returning the required sum
   return (sum1 - sum2);
}
```

Time Complexity is O(n) and space complexity is O(1).

14. A given linked list is sorted based on absolute values. Write a function to sort the list based on actual values in O(n) time.



All the negative elements can be found in the reverse order. Therefore, as we traverse the list, whenever we find an element that is out of order, it is moved to the front of the linked list.

Auxiliary Space: O(1)

Implementation:

```
void sortList(Node** head)
{
   // Initialize the previous and the current nodes
   Node* previous = (*head);
   Node* current;

   // list traversal
   for(current = (*head)->next; current != NULL; current = current->next)
   {
       // continue if the current element
       // is at its right place
       if (current->value >= previous->value)
       {
          previous = current;
       }

       // If current is smaller than previous, then
       // it must be moved to head
       else
       {
          // Detach current from the linked list
           previous->next = current->next;

           // Move the current node to the beginning
           current->next = (*head);
           (*head) = current;

           // Update current
           current = previous;

        }
   }
}
```

Linked List Interview Questions for Experienced

15. How will you find the length of a linked list which contains a cycle?

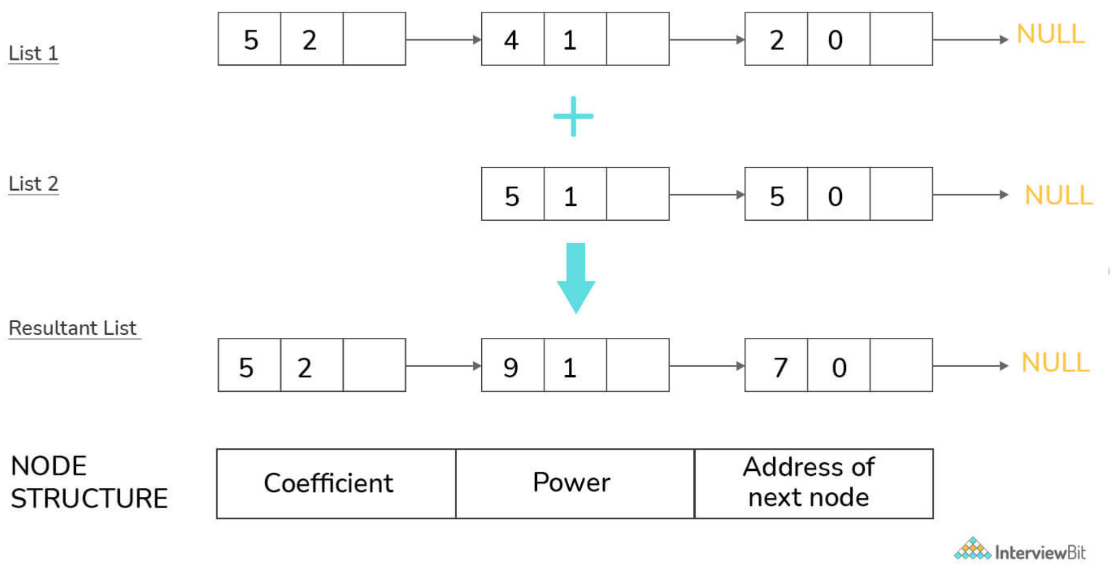
The algorithm to determine the length of the list:

- At a time, pointer A advances one node while pointer B advances two nodes.
- Starting from the head, they move until B reaches null (no loop) or A and B refer to the same node.
- Now, if A simply advances, A will run into B again. The length of the loop, let's call it x, may be calculated from this.
- Begin again from the head, but this time have a pointer C which has moved x nodes, followed by a pointer D behind it. Both will move one node further at a time.
- When they meet, the length of the linked list with a loop is equal to the number of nodes traversed by D plus x.

```
int calcLen( Node* head )
{
  struct Node *A = head, *B = head;

   while ( A && B && B->next )
   {
       A = A->next;
       B = B->next->next;

       /* If slow_p and fast_p meet at
       some point then there is a loop */
       if (A == B)
       {
          int x = 1;
          struct Node *t = A;
          while (t->next != A)
          {
              x++;
              t = t->next;
          }
       }
   }

   struct Node *C = head, *D = head;
   int y = 0;
   for ( int i = 0; i < x; i++ )
   {
      C = C->next;
   }
   while( C != D )
   {
      y++;
      C = C->next;
      D = D->next;
   }

   return x+y;
}
```

Time Complexity: O(n)
Space Complexity: O(1)

16. Given two polynomial expressions represented by linked lists. You need to write a function that adds these lists, that is, adds the coefficients that have the same variable powers.

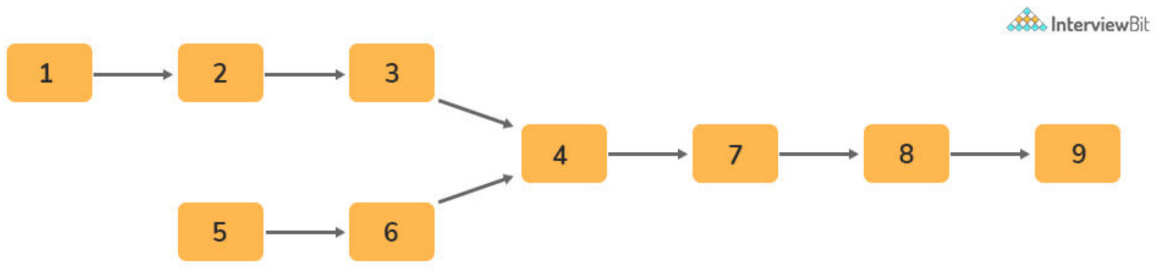


Implementation:

```
struct Node
{
   int coefficient;
   int power;
   struct Node* next;
};

void createNode(int x, int y, struct Node** t)
{
   struct Node *v, *z;
   z = *t;
   if (z == NULL)
   {
       v = new Node;
       v->coefficient = x;
       v->power = y;
       *t = v;
       v->next = new Node;
       v = v->next;
       v->next = NULL;
   }
   else
   {
       v->coefficient = x;
       v->power = y;
       v->next = new Node;
       v = v->next;
       v->next = NULL;
   }
}

// Function to add two polynomial expressions
void polyAdd(struct Node* poly1, struct Node* poly2,
            struct Node* poly)
{
   while (poly1->next && poly2->next)
   {
       // If the power of the 1st polynomial is greater than that of 2nd,
       // then store 1st as it is and move its pointer
       if (poly2->power < poly1->power)
       {
           poly->power = poly1->power;
           poly->coefficient = poly1->coefficient;
           poly1 = poly1->next;
       }

       // If the power of the 2nd polynomial is greater than that of 1st,
       // then store 2nd as it is and move its pointer
       else if (poly1->power < poly2->power)
       {
           poly->power = poly2->power;
           poly->coefficient = poly2->coefficient;
           poly2 = poly2->next;
       }

       // If power of both polynomial expressions is same then
       // add their coefficients
```

Time Complexity: O(m + n), where m and n are the respective numbers of nodes in the first and second lists.

17. Let's say there are two lists of varying lengths that merge at a certain point; how do we know where the merging point is?



You begin with List 1 and assume that the NULL at the end is a pointer to the start of List 2, giving the illusion of a cyclic list. After that, the algorithm will inform you how far down List 1 the merging is.

- Make an iterator pointer that runs forward until it reaches the end of the list, then jumps to the start of the opposite list, and so on.
- Make two of them, each pointing to two different heads.
- Each time you advance the pointers by one, they will eventually meet in the intersection point (IP). After one or two passes, this will occur.

Count the number of nodes traversed from head1-> tail1-> head2 -> intersection point and head2-> tail2-> head1 -> intersection point to have a better understanding. Both will be equal (Draw diff types of linked lists to verify this). The reason for this is that both pointers must traverse identical distances (head1->IP + head2->IP) before returning to IP. As a result, by the time it reaches IP, both pointers will be equal, and we will have arrived at the merging point.

```
Node* getIP(Node* head1, Node* head2)
{
   // two pointers ptr1 and ptr2
   // at the heads of the two lists
   Node* p1 = head1;
   Node* p2 = head2;

   if (p1 == NULL || p2 == NULL)
      return NULL;

   // the two lists are to be traversed until we reach the IP
   while (p1 != p2)
   {
       p1 = p1->next;
       p2 = p2->next;

       // When p1 reaches the end of the list, it is
       // redirected to head2.
       if (p1 == NULL)
           p1 = head2;

       // When p2 reaches the end of the list, it is
       // redirected to head1.
       if (p2 == NULL)
           p2 = head1;

       // If at any node p1 meets p2, we have got our IP.
       if (p1 == p2)
           return p2;

   }

   return p2;
}
```

OR

- Make a circular linked list by traversing the first linked list (counting the elements). (Keep track of the last node so we can break the circle later.)
- Reframe the issue as locating the loop in the second linked list.
- We already know the length of the loop which is equal to the length of the first linked list. We have to traverse those many numbers of nodes in the second list first, and then another pointer should be started from the beginning of the second list. We have to continue traversing until they meet, and that is the required intersection point.
- The circle should be removed from the linked list.

Both the approaches have O(n+m) time and O(1) space complexity.

18. Given a value x and a sorted doubly linked list of different nodes (no two nodes have the same data). Count the number of triplets in the list that add up to x. The expected time complexity is O(n^2) and the expected space complexity is O(1).

Following the approach os using two pointers:

From left to right, traverse the doubly linked list. Initialize two pointers for each current node during the traversal: first = pointer to the node next to the current node, and last = pointer to the list's last node. Count the number of pairs in the list that add up to the value (x – the current node's data) from the first to the last pointer (algorithm described in Q8). This number should be added to the total count of triplets. Pointer to the last node can be retrieved only once at the beginning.

Implementation:

```
int pairCount(struct Node* num1, struct Node* num2, int val)
{
   int cnt = 0;
   // The loop terminates when either of two the pointers
   // become NULL, or they cross each other or they become equal
   while (num1 != NULL && num2 != NULL &&
          num1 != num2 && num2->next != num1)
   {

       // pair found
       if ((num1->value + num2->value) == val)
       {

           cnt++;

           // second is moved in backward direction
           num2 = num2->prev;
           // first is moved in forward direction
           num1 = num1->next;
       }
       // else first is moved in forward direction
       else if ((num1->value + num2->value) < val)
           num1 = num1->next;

       // if sum is greater than 'value'
       // second is moved in backward direction
       else
           num2 = num2->prev;
     }

   // required number of pairs
   return cnt;
}

// function to count triplets in a sorted DLL
// whose sum equals a given value 'x'
int tripletCount(struct Node* head, int x)
{
   // check if the list is empty
   if (!head)
       return 0;
   int cnt = 0;
   struct Node* current = head;
   struct Node* last = head;
   struct Node* first;
   // get pointer to the last node of the doubly linked list

   while (last->next)
       last = last->next;

   // traverse the doubly linked list

   while (current)
   {
       // for every current node
       first = current->next;
```

19. Given a linked list, find the length of the longest palindrome list that appears in that linked list using O(1) extra space.

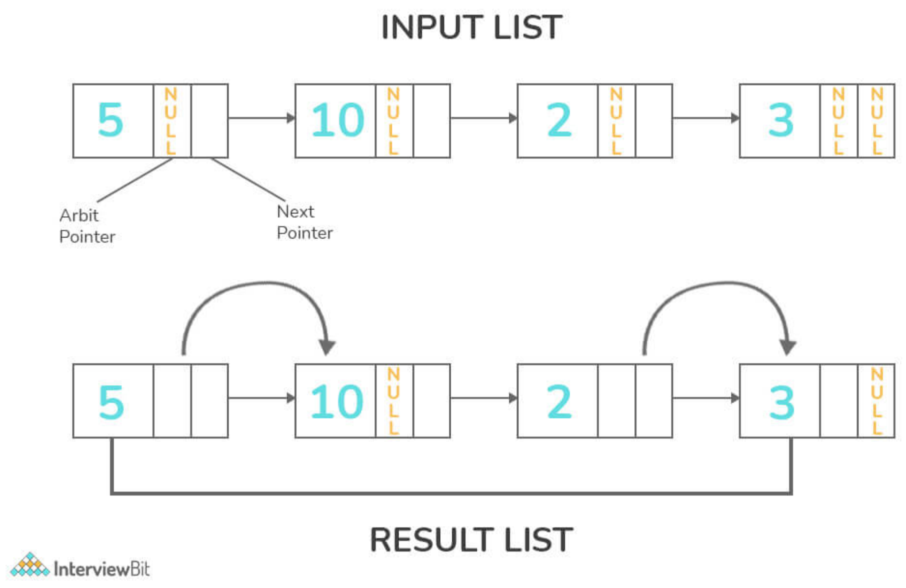
The concept is built on reversing a linked list iteratively. We loop through the given linked list, reversing each prefix of the linked list one by one from the left. We discover the longest common list beginning with the reversed prefix and the list after the reversed prefix, after reversing a prefix.

Time complexity is O(n^2).

Implementation:

```
int calcCommon(Node *x, Node *y)
{
   int cnt = 0;

   // count common in the list starting
   // from node x and y
  while (1)
  {
       // increase the count by one for same values
       if (x->value == y->value)
           cnt++;
       else
           break;
       x = x->next;
       y = y->next;
  }
  return cnt;
}

// Returns length of the longest palindrome sublist
int maxPalindrome(Node *head)
{
   Node *previous = NULL, *current = head;
   int answer = 0;
   // loop running till the end of the linked list
   while (1)
   {
       if(current==NULL)
          break;
       // reversed sublist from head to current
       Node *next = current->next;
       current->next = previous;

       // check for odd length palindrome
       answer = max(result, 2*calcCommon(previous, next)+1);

       // check for even length palindrome
       answer = max(result, 2*calcCommon(current, next));

       // update previous and current for next iteration
       previous = current;
       current = next;
   }
   return answer;
}
```

20. Given a singly linked list with an additional "arbitrary" pointer at each node that currently points to NULL. What algorithm will you implement to make the "arbitrary" pointer point to the next node with a greater value?



A simple solution is to go through all nodes one by one, finding the node with the next bigger value than the current node and changing the next pointer for each node. This solution has an O time complexity (n^2).

An Efficient Solution takes O(nLogn) time to complete. The approach is to use Merge Sort for linked lists.

- Traverse input list and for each node, copy the next pointer to arbit pointer.
- Sort the linked list formed by arbit pointers using Merge Sort.

Here, all of the merge sort methods are altered to work with arbit pointers rather than the next pointers.

21. In a standard Doubly Linked List, two address fields are required to contain the addresses of previous and next nodes. Can you create a doubly linked list using only one space for the address field with every node?

Yes, there is a memory-saving version of the Doubly Linked List that uses only one space for the address field in each node. Because the list uses the bitwise XOR operation to save space for one address, it is known as the XOR Linked List or Memory Efficient. Instead of storing actual memory addresses, each node in the XOR linked list stores the XOR of previous and next node addresses.

In the XOR representation, let's call the address variable npx (XOR of next and previous).

We can traverse the XOR Linked List in both forward and reverse directions while traversing it. We must remember the address of the previously visited node when traversing the list in order to determine the address of the next node.

Node W:
npx = 0 XOR add(X)

Node X:
npx = add(W) XOR add(Y)

Node Y:
npx = add(X) XOR add(Z)

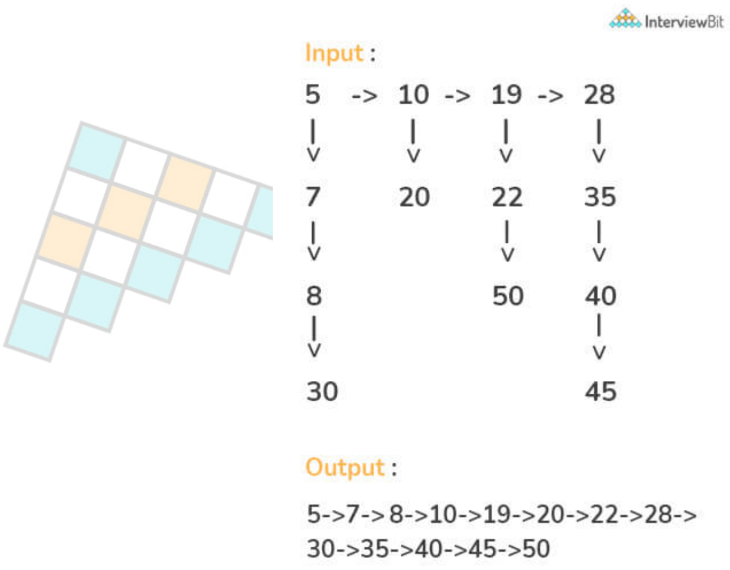
Node Z:
npx = add(Y) XOR 0

Calculation:

```
npx(Y) XOR add(X)
=> (add(X) XOR add(Z)) XOR add(X)          // npx(Y) = add(X) XOR add(Z)
=> add(X) XOR add(Z) XOR add(X)            // w^x = w^x and (w^x)^y = w^(x^y)
=> add(Z) XOR 0                            // x^x = 0
=> add(Z)                                  // x^0 = x
```

22. Given a linked list with each node representing a linked list and two pointers of its type given below. You need to flatten the lists into a single linked list. The flattened linked list also needs to be sorted. Discuss the approach.

- **Pointer to the next node in the main list (the "right" pointer).**
- **Pointer to a linked list, this node being the head (the 'down' pointer).**



Input :

```
5   -> 10 -> 19 -> 28
|       |     |     |
v       v     v     v
7      20    22    35
|             |     |
v             v     v
8            50    40
|                   |
v                   v
30                 45
```

Output :

```
5->7->8->10->19->20->22->28->
30->35->40->45->50
```

The approach is to use the merge() procedure of merge sort for linked lists. To merge lists one by one, we use merge(). We merge() the current list with the flattened list recursively.
The flattened list's nodes are linked via the down pointer.

Another approach is to use heaps. It is important to notice that there are N nodes connecting in a downward manner from each top node, but those downward nodes are in sorted order. As a result, the objective is to sort everything in ascending order (or decreasing order).

- Push all the heads of the linked lists in the priority queue's downward list.
- Pop the node with the smallest priority from the priority queue.
- Check the node's location so that the next node which is being pointed by the current node can be inserted into the priority queue.
- Pop the smallest element once more and push the next node pointed by the current node until the heap is empty.
- Continue to add node data to a new linked list that is popped out to the new list.
- Print the above-mentioned linked list.

```
struct cmp {
   bool operator()(Node* x, Node* y)
   {
       return x->value > y->value;
   }
};

//the following function returns the flattened linked list’s root
Node* flattenList(Node* root)
{
   Node* p = root;
   Node* head = NULL;

   // this min heap returns the current smallest element in the heap
   priority_queue < Node*, vector <Node*>, cmp > pqueue;

   // the head nodes of every
   // downward linked list is pushed into the heap
   while (p)
   {
       pqueue.push(p);
       p = p->right;
   }

   while (!pqueue.empty())
   {
       // pop out the topmost node
       Node* t = pqueue.top();
       pqueue.pop();

       if (t->down)
       {
           pqueue.push(t->down);
       }

       // Create the required linked list
       if (head != NULL)
           p->down = t;
       else
           head = t;

       p = t;
       p->right = NULL;

   }

   // Pointer to head node
   return head;
}
void printList(Node* head)
{
   while (head != NULL)
   {
       cout << head->value << ” “ << endl;
       head = head->down;
   }
}
```

Time Complexity : O(nlogn)
Space Complexity : O(n)

Conclusion:

We hope that this article has helped you learn the fundamentals and advanced questions of linked lists. These questions cover the most important concepts related to linked lists which will help you in both interview and understanding this data structure in depth.

Links to More Interview Questions