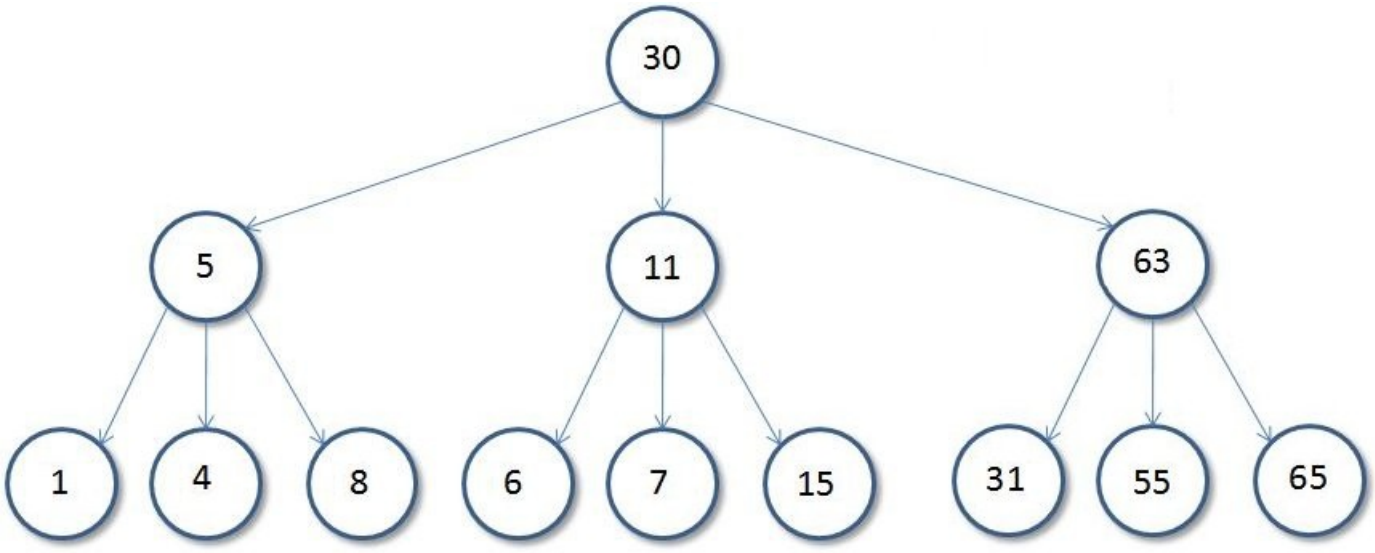
Given a ternary tree, create a doubly linked list out of it. A ternary tree is just like binary tree but instead of having two nodes, it has three nodes i.e. left, middle, right.

The doubly linked list should holds following properties â€“

1. Left pointer of ternary tree should act as prev pointer of doubly linked list.
2. Middle pointer of ternary tree should not point to anything.
3. Right pointer of ternary tree should act as next pointer of doubly linked list.
4. Each node of ternary tree is inserted into doubly linked list before its subtrees and for any node, its left child will be inserted first, followed by mid and right child (if any).

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



We strongly recommend you to minimize your browser and try this yourself first.

The idea is to traverse the tree in preoder fashion similar to binary tree preorder traversal. Here, when we visit a node, we will insert it into doubly linked list in the end using a tail pointer. That we use to maintain the required insertion order. We then recursively call for left child, middle child and right child in that order.

Below is C++ implementation of this idea.

```
// C++ program to create a doubly linked list out
// of given a ternary tree.
#include <bits/stdc++.h>
using namespace std;

/* A ternary tree */
struct Node
{
    int data;
    struct Node *left, *middle, *right;
};

/* Helper function that allocates a new node with the
   given data and assign NULL to left, middle and right
   pointers.*/
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->middle = node->right = NULL;
    return node;
}

/* Utility function that constructs doubly linked list
by inserting current node at the end of the doubly
linked list by using a tail pointer */
void push(Node** tail_ref, Node* node)
{
    // initilize tail pointer
    if (*tail_ref == NULL)
    {
        *tail_ref = node;
```

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

        return;
    }

    // insert node in the end using tail pointer
    (*tail_ref)->right = node;

    // set prev of node
    node->left = (*tail_ref);

    // set middle and right child to point to NULL
    node->right = node->middle = NULL;

    // now tail pointer will point to inserted node
    (*tail_ref) = node;
}

/* Create a doubly linked list out of given a ternary tree.
by traversing the tree in preoder fashion. */
Node* TernaryTreeToList(Node* root, Node** head_ref)
{
    // Base case
    if (root == NULL)
        return NULL;

    //create a static tail pointer
    static Node* tail = NULL;

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

    // set head of the doubly linked list
    // head will be root of the ternary tree
    if (*head_ref == NULL)
        *head_ref = root;

    // push current node in the end of DLL
    push(&tail, root);

    //recurse for left, middle and right child
    TernaryTreeToList(left, head_ref);
    TernaryTreeToList(middle, head_ref);
    TernaryTreeToList(right, head_ref);
}

// Utility function for printing double linked list.
void printList(Node* head)
{
    printf("Created Double Linked list is:\n");
    while (head)
    {
        printf("%d ", head->data);
        head = head->right;
    }
}

// Driver program to test above functions
int main()
{
    // Construting ternary tree as shown in above figure
    Node* root = newNode(30);

    root->left = newNode(5);
    root->middle = newNode(11);
    root->right = newNode(63);

    root->left->left = newNode(1);
    root->left->middle = newNode(4);
    root->left->right = newNode(8);
```

```
    root->middle->left = newNode(6);
    root->middle->middle = newNode(7);
    root->middle->right = newNode(15);

    root->right->left = newNode(31);
    root->right->middle = newNode(55);
    root->right->right = newNode(65);

    Node* head = NULL;

    TernaryTreeToList(root, &head);

    printList(head);

    return 0;
}
```

Output:

```
Created Double Linked list is:
30 5 1 4 8 11 6 7 15 63 31 55 65
```

This article is contributed by **Aditya Goel**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Company Wise Coding Practice Â Â Â Topic Wise Coding Practice

Given two linked lists that represent two large positive numbers. Subtract the smaller number from larger one and return the difference as a linked list. Note that the input lists may be in any order, but we always need to subtract smaller from larger one.

It may be assumed that there are no extra leading zeros in input lists.

Examples

```
Input  : l1 = 1 -> 0 -> 0 -> NULL,  l2 = 1 -> NULL
Output : 0->9->9->NULL

Input  : l1 = 1 -> 0 -> 0 -> NULL,  l2 = 1 -> NULL
Output : 0->9->9->NULL

Input  : l1 = 7-> 8 -> 6 -> NULL,  l2 = 7 -> 8 -> 9 NULL
Output : 3->NULL
```

Following are the steps.
1) Calculate sizes of given two linked lists.
2) If sizes not are same, then append zeros in smaller linked list.
3) If size are same, then follow below steps:
â€¦.a) Find the smaller valued linked list.
â€¦.b) One by one subtract nodes of smaller sized linked list from larger size. Keep track of borrow while subtracting.

Following is C implementation of the above approach.

```
// C++ program to subtract smaller valued list from
// larger valued list and return result as a list.
#include<bits/stdc++.h>
using namespace std;

// A linked List Node
struct Node
{
    int data;
    struct Node* next;
};

// A utility which creates Node.
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->next = NULL;
    return temp;
}

/* A utility function to get length of linked list */
int getLength(Node *Node)
{
    int size = 0;
    while (Node != NULL)
    {
        Node = Node->next;
        size++;
    }
    return size;
}

/* A Utility that padds zeros in front of the
   Node, with the given diff */
Node* paddZeros(Node* sNode, int diff)
{
    if (sNode == NULL)
        return NULL;

    Node* zHead = newNode(0);
    diff--;
    Node* temp = zHead;
    while (diff--)
    {
        temp->next = newNode(0);
        temp = temp->next;
    }
    temp->next = sNode;
    return zHead;
}

/* Subtract LinkedList Helper is a recursive function,
   move till the last Node,  and subtract the digits and
   create the Node and return the Node. If d1 < d2, we
   borrow the number from previous digit. */
Node* subtractLinkedListHelper(Node* l1, Node* l2, bool& borrow)
{
    if (l1 == NULL && l2 == NULL && borrow == 0)
        return NULL;

    Node* previous = subtractLinkedListHelper(l1 ? l1->next : NULL,
                                    l2 ? l2->next : NULL, borrow);

    int d1 = l1->data;
    int d2 = l2->data;
    int sub = 0;

    /* if you have given the value value to next digit then
       reduce the d1 by 1 */
    if (borrow)
    {
        d1--;
        borrow = false;
    }
```

```
    /* If d1 < d2 , then borrow the number from previous digit.
       Add 10 to d1 and set borrow = true; */
    if (d1 < d2)
    {
        borrow = true;
        d1 = d1 + 10;
    }

    /* subtract the digits */
    sub = d1 - d2;

    /* Create a Node with sub value */
    Node* current = newNode(sub);

    /* Set the Next pointer as Previous */
    current->next = previous;

    return current;
}

/* This API subtracts two linked lists and returns the
   linked list which shall  have the subtracted result. */
Node* subtractLinkedList(Node* l1, Node* l2)
{
    // Base Case.
    if (l1 == NULL &&  l2 == NULL)
        return NULL;

    // In either of the case, get the lengths of both
    // Linked list.
    int len1 = getLength(l1);
    int len2 = getLength(l2);

    Node *lNode = NULL, *sNode = NULL;

    Node* temp1 = l1;
    Node* temp2 = l2;

    // If lengths differ, calculate the smaller Node
    // and padd zeros for smaller Node and ensure both
    // larger Node and smaller Node has equal length.
    if (len1 != len2)
    {
        lNode = len1 > len2 ? l1 : l2;
        sNode = len1 > len2 ? l2 : l1;
        sNode = paddZeros(sNode, abs(len1 - len2));
    }

    else
    {
        // If both list lengths are equal, then calculate
        // the larger and smaller list. If 5-6-7 & 5-6-8
        // are linked list, then walk through linked list
        // at last Node as 7 < 8, larger Node is 5-6-8
        // and smaller Node is 5-6-7.
        while (l1 && l2)
        {
            if (l1->data != l2->data)
            {
                lNode = l1->data > l2->data ? temp1 : temp2;
                sNode = l1->data > l2->data ? temp2 : temp1;
                break;
            }
            l1 = l1->next;
            l2 = l2->next;
        }
    }

    // After calculating larger and smaller Node, call
    // subtractLinkedListHelper which returns the subtracted
    // linked list.
    bool borrow = false;
    return subtractLinkedListHelper(lNode, sNode, borrow);
}
```

```
/* A utility function to print linked list */
void printList(struct Node *Node)
{
    while (Node != NULL)
    {
        printf("%d ", Node->data);
        Node = Node->next;
    }
    printf("\n");
}

// Driver program to test above functions
int main()
{
    Node* head1 = newNode(1);
    head1->next = newNode(0);
    head1->next->next = newNode(0);

    Node* head2 = newNode(1);

    Node* result = subtractLinkedList(head1, head2);

    printList(result);

    return 0;
}
```

Output :

```
0 9 9
```

This article is contributed by **Mu Ven**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Company Wise Coding Practice Â Â Â Topic Wise Coding Practice

Given a singly linked list of 0s and 1s find its decimal equivalent.

```
   Input  : 0->0->0->1->1->0->0->1->0
   Output : 50

   Input  : 1->0->0
   Output : 4
```

Decimal Value of an empty linked list is considered as 0.

Initialize result as 0. Traverse the linked list and for each node, multiply the result by 2 and add nodeâ€™s data to it.

```
// C++ Program to find decimal value of
// binary linked list
#include<iostream>
using namespace std;
```

```
/* Link list Node */
struct Node
{
    bool data;
    struct Node* next;
};

/* Returns decimal value of binary linked list */
int decimalValue(struct Node *head)
{
    // Initialized result
    int  res = 0;

    // Traverse linked list
    while (head != NULL)
    {
        // Multiply result by 2 and add
        // head's data
        res = (res  << 1) + head->data;

        // Move next
        head = head->next;
    }
    return res;
}

// Utility function to create a new node.
Node *newNode(bool data)
{
    struct Node *temp = new Node;
    temp->data = data;
    temp->next = NULL;
    return temp;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = newNode(1);
    head->next = newNode(0);
    head->next->next = newNode(1);
    head->next->next->next = newNode(1);

    cout << "Decimal value is "
         << decimalValue(head);

    return 0;
}
```

Output :

```
Decimal value is 11
```

This article is contributed by **Shivam Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Company Wise Coding Practice Â Â Â Topic Wise Coding Practice

Given K sorted linked lists of size N each, merge them and print the sorted output.

Example:

```
Input: k = 3, n =  4
list1 = 1->3->5->7->NULL
list2 = 2->4->6->8->NULL
list3 = 0->9->10->11

Output:
0->1->2->3->4->5->6->7->8->9->10->11
```

Â

Method 1 (Simple)

A Simple Solution is to initialize result as first list. Now traverse all lists starting from second list. Insert every node of currently traversed list into result in a sorted way. Time complexity of this solution is $O(N^2)$ where N is total number of nodes, i.e., N = kn.
Â

Method 2 (Using Min Heap)

A **Better solution** is to use Min Heap based solution which is discussed here for arrays. Time complexity of this solution would be O(nk Log k)
Â

Method 3 (Using Divide and Conquer))

In this post, **Divide and Conquer** approach is discussed. This approach doesnâ€™t require extra space for heap and works in O(nk Log k)
We already know that merging of two linked lists can be done in O(n) time and O(1) space (For arrays O(n) space is required). The idea is to pair up K lists and merge each pair in linear time using O(1) space. After first cycle, K/2 lists are left each of size 2*N. After second cycle, K/4 lists are left each of size 4*N and so on. We repeat the procedure until we have only one list left.

Below is C++ implementation of the above idea.

```
// C++ program to merge k sorted arrays of size n each
#include <bits/stdc++.h>
using namespace std;

// A Linked List node
struct Node
{
    int data;
    Node* next;
};

/* Function to print nodes in a given linked list */
void printList(Node* node)
{
    while (node != NULL)
    {
```

```
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Takes two lists sorted in increasing order, and merge
   their nodes together to make one big sorted list. Below
   function takes O(Log n) extra space for recursive calls,
   but it can be easily modified to work with same time and
   O(1) extra space  */
Node* SortedMerge(Node* a, Node* b)
{
    Node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return (b);
    else if(b == NULL)
        return (a);

    /* Pick either a or b, and recur */
    if(a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }

    return result;
}

// The main function that takes an array of lists
// arr[0..last] and generates the sorted output
Node* mergeKLists(Node* arr[], int last)
{
    // repeat until only one list is left
    while (last != 0)
    {
        int i = 0, j = last;

        // (i, j) forms a pair
        while (i < j)
        {
            // merge List i with List j and store
            // merged list in List i
            arr[i] = SortedMerge(arr[i], arr[j]);

            // consider next pair
            i++, j--;

            // If all pairs are merged, update last
            if (i >= j)
                last = j;
        }
    }

    return arr[0];
}

// Utility function to create a new node.
Node *newNode(int data)
{
    struct Node *temp = new Node;
    temp->data = data;
    temp->next = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    int k = 3; // Number of linked lists
```

```
    int n = 4; // Number of elements in each list

    // an array of pointers storing the head nodes
    // of the linked lists
    Node* arr[k];

    arr[0] = newNode(1);
    arr[0]->next = newNode(3);
    arr[0]->next->next = newNode(5);
    arr[0]->next->next->next = newNode(7);

    arr[1] = newNode(2);
    arr[1]->next = newNode(4);
    arr[1]->next->next = newNode(6);
    arr[1]->next->next->next = newNode(8);

    arr[2] = newNode(0);
    arr[2]->next = newNode(9);
    arr[2]->next->next = newNode(10);
    arr[2]->next->next->next = newNode(11);

    // Merge all lists
    Node* head = mergeKLists(arr, k - 1);

    printList(head);

    return 0;
}
```

Output :

```
0 1 2 3 4 5 6 7 8 9 10 11
```

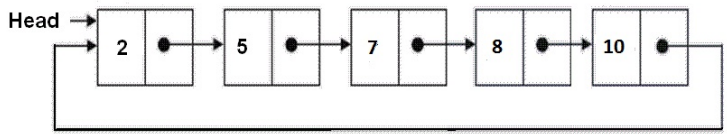
Time Complexity of above algorithm is O(nk logk) as outer while loop in function mergeKLists() runs log k times and every time we are processing nk elements.

This article is contributed by **Aditya Goel**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Company Wise Coding Practice Â Â Â Topic Wise Coding Practice

Given a singly linked list, find if the linked list is circular or not. A linked list is called circular if it not NULL terminated and all nodes are connected in the form of a cycle. Below is an example of circular linked list.



An empty linked list is considered as circular.

Note that this problem is different from cycle detection problem, here all nodes have to be part of cycle.

The idea is to store head of the linked list and traverse it. If we reach NULL, linked list is not circular. If reach head again, linked list is circular.

```
// C++ program to check if linked list is circular
#include<bits/stdc++.h>
using namespace std;

/* Link list Node */
struct Node
{
    int data;
    struct Node* next;
};

/* This function returns true if given linked
   list is circular, else false. */
bool isCircular(struct Node *head)
{
    // An empty linked list is circular
    if (head == NULL)
       return true;

    // Next of head
    struct Node *node = head->next;

    // This loop would stope in both cases (1) If
    // Circular (2) Not circular
    while (node != NULL && node != head)
       node = node->next;

    // If loop stopped because of circular
    // condition
    return (node == head);
}

// Utility function to create a new node.
Node *newNode(int data)
{
    struct Node *temp = new Node;
    temp->data = data;
    temp->next = NULL;
    return temp;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);

    isCircular(head)? cout << "Yes\n" :
                      cout << "No\n" ;

    // Making linked list circular
    head->next->next->next->next = head;

    isCircular(head)? cout << "Yes\n" :
                      cout << "No\n" ;

    return 0;
}
```

Output :

```
No
Yes
```

This article is contributed by **Shivam Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Company Wise Coding Practice Â Â Â Topic Wise Coding Practice

Given a list of integers, rearrange the list such that it consists of alternating minimum maximum elements **using only list operations**. The first element of the list should be minimum and second element should be maximum of all elements present in the list. Similarly, third element will be next minimum element and fourth element is next maximum element and so on. Use of extra space is not permitted.

Examples:

```
Input:  [1 3 8 2 7 5 6 4]
Output: [1 8 2 7 3 6 4 5]

Input:  [1 2 3 4 5 6 7]
Output: [1 7 2 6 3 5 4]

Input:  [1 6 2 5 3 4]
Output: [1 6 2 5 3 4]
```

The idea is to sort the list in ascending order first. Then we start popping elements from the end of the list and insert them into their correct position in the list.

Below is C++ implementation of above idea â€“

C/C++

```
// C++ program to rearrange a given list such that it
// consists of alternating minimum maximum elements
#include <bits/stdc++.h>
using namespace std;
```

```
// Function to rearrange a given list such that it
// consists of alternating minimum maximum elements
void alternateSort(list<int>& inp)
{
    // sort the list in ascending order
    inp.sort();

    // get iterator to first element of the list
    list<int>::iterator it = inp.begin();
    it++;

    for (int i=1; i<(inp.size() + 1)/2; i++)
    {
        // pop last element (next greatest)
        int val = inp.back();
        inp.pop_back();

        // insert it after next minimum element
        inp.insert(it, val);

        // increment the pointer for next pair
        ++it;
    }
}

// Driver code
int main()
{
    // input list
    list<int> inp({ 1, 3, 8, 2, 7, 5, 6, 4 });

    // rearrange the given list
    alternateSort(inp);

    // print the modified list
    for (int i : inp)
        cout << i << " ";

    return 0;
}
```

Java

```
// Java program to rearrange a given list such that it
// consists of alternating minimum maximum elements
import java.util.*;

class AlternateSort
{
 // Function to rearrange a given list such that it
    // consists of alternating minimum maximum elements
    // using LinkedList
 public static void alternateSort(LinkedList<Integer> ll)
 {
  Collections.sort(ll);

  for (int i = 1; i < (ll.size() + 1)/2; i++)
     {
      Integer x = ll.getLast();
      ll.removeLast();
      ll.add(2*i - 1, x);
  }

  System.out.println(ll);
 }

 public static void main (String[] args) throws java.lang.Exception
 {
  // input list
  Integer arr[] = {1, 3, 8, 2, 7, 5, 6, 4};

  // convert array to LinkedList
```

```
        LinkedList<Integer> ll = new LinkedList<Integer>(Arrays.asList(arr));

  // rearrange the given list
  alternateSort(ll);
 }
}
```

Output:

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

This article is contributed by **Aditya Goel**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Company Wise Coding Practice Â Â Â Topic Wise Coding Practice

Given a singly linked list, delete middle of the linked list. For example, if given linked list is 1->2->3->4->5 then linked list should be modified to 1->2->4->5

If there are even nodes, then there would be two middle nodes, we need to delete the second middle element. For example, if given linked list is 1->2->3->4->5->6 then it should be modified to 1->2->3->5->6.

If the input linked list is NULL, then it should remain NULL.

If the input linked list has 1 node, then this node should be deleted and new head should be returned.

A Simple Solution is to first count number of nodes in linked list, then delete n/2â€™th node using the simple deletion process.

The above solution requires two traversals of linked list. We can delete middle node using one traversal. The idea is to use two pointers, slow_ptr and fast_ptr. Both pointers start from head of list. When fast_ptr reaches end, slow_ptr reaches middle. This idea is same as the one used in method 2 of this post. The additional thing in this post is to keep track of previous of middle so that we can delete middle.

Below is C++ implementation.

```
// C++ program to delete middle of a linked list
#include<bits/stdc++.h>
using namespace std;

/* Link list Node */
struct Node
{
    int data;
    struct Node* next;
};

// Deletes middle node and returns head of the
// modified list
struct Node* deleteMid(struct Node *head)
{
    // Base cases
    if (head == NULL)
        return NULL;
    if (head->next == NULL)
```

```
    {
        delete head;
        return NULL;
    }

    // Initialize slow and fast pointers to reach
    // middle of linked list
    struct Node *slow_ptr = head;
    struct Node *fast_ptr = head;

    // Find the middle and previous of middle.
    struct Node *prev; // To store previous of slow_ptr
    while (fast_ptr != NULL && fast_ptr->next != NULL)
    {
        fast_ptr = fast_ptr->next->next;
        prev = slow_ptr;
        slow_ptr = slow_ptr->next;
    }

    //Delete the middle node
    prev->next = slow_ptr->next;
    delete slow_ptr;

    return head;
}

// A utility function to print a given linked list
void printList(struct Node *ptr)
{
    while (ptr != NULL)
    {
        cout << ptr->data << "->";
        ptr = ptr->next;
    }
    cout << "NULL\n";
}

// Utility function to create a new node.
Node *newNode(int data)
{
    struct Node *temp = new Node;
    temp->data = data;
    temp->next = NULL;
    return temp;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);

    cout << "Gven Linked List\n";
    printList(head);

    head = deleteMid(head);

    cout << "Linked List after deletion of middle\n";
    printList(head);

    return 0;
}
```

Output :

```
Gven Linked List
1->2->3->4->NULL
Linked List after deletion of middle
1->2->4->NULL
```

This article is contributed by **Piyush Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your

article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Company Wise Coding Practice Â Â Â Topic Wise Coding Practice

Given two sorted singly linked lists having n and m elements each, merge them using constant space. First n smallest elements in both the lists should become part of first list and rest elements should be part of second list. Sorted order should be maintained. We are not allowed to change pointers of first linked list.

For example,

```
Input:
First List: 2->4->7->8->10
Second List: 1->3->12

Output:
First List: 1->2->3->4->7
Second List: 8->10->12
```

We strongly recommend you to minimize your browser and try this yourself first.

The problem becomes very simple if weâ€™re allowed to change pointers of first linked list. If we are allowed to change links, we can simply do something like merge of merge-sort algorithm. We assign first n smallest elements to the first linked list where n is the number of elements in first linked list and the rest to second linked list. We can achieve this in O(m + n) time and O(1) space, but this solution violates the requirement that we canâ€™t change links of first list.

The problem becomes a little tricky as weâ€™re not allowed to change pointers in first linked list. The idea is something similar to this post but as we are given singly linked list, we canâ€™t proceed backwards with the last element of LL2.

The idea is for each element of LL1, we compare it with first element of LL2. If LL1 has a greater element than first element of LL2, then we swap the two elements involved. To keep LL2 sorted, we need to place first element of LL2 at its correct position. We can find mismatch by traversing LL2 once and correcting the pointers.

Below is C++ implementation of this idea.

```
// Program to merge two sorted linked lists without
// using any extra space and without changing links
// of first list
#include <bits/stdc++.h>
using namespace std;

 /* Structure for a linked list node */
struct node
{
    int data;
    struct node *next;
};

/* 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(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
```

```
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

// Function to merge two sorted linked lists
// LL1 and LL2 without using any extra space.
void mergeLists(struct node *a, struct node * &b)
{
    // run till either one of a or b runs out
    while (a && b)
    {
        // for each element of LL1,
        // compare it with first element of LL2.
        if (a->data > b->data)
        {
            // swap the two elements involved
            // if LL1 has a greater element
            swap(a->data, b->data);

            struct node *temp = b;

            // To keep LL2 sorted, place first
            // element of LL2 at its correct place
            if (b->next && b->data > b->next->data)
            {
                b = b->next;
                struct node *ptr= b, *prev = NULL;

                // find mismatch by traversing the
                // second linked list once
                while (ptr && ptr->data < temp->data)
                {
                    prev = ptr;
                    ptr = ptr -> next;
                }

                // correct the pointers
                prev->next = temp;
                temp->next = ptr;
            }
        }

        // move LL1 pointer to next element
        a = a->next;
    }
}

// Code to print the linked link
void printList(struct node *head)
{
    while (head)
    {
        cout << head->data << "->" ;
        head = head->next;
    }
    cout << "NULL" << endl;
}

// Driver code
int main()
{
    struct node *a = NULL;
    push(&a, 10);
    push(&a, 8);
    push(&a, 7);
    push(&a, 4);
    push(&a, 2);
```

```
    struct node *b = NULL;
    push(&b, 12);
    push(&b, 3);
    push(&b, 1);

    mergeLists(a, b);

    cout << "First List: ";
    printList(a);

    cout << "Second List: ";
    printList(b);

    return 0;
}
```

Output :

```
First List: 1->2->3->4->7->NULL
Second List: 8->10->12->NULL
```

Time Complexity : O(mn)

This article is contributed by **Aditya Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Company Wise Coding Practice Â Â Â Topic Wise Coding Practice

We have discussed flattening of a multi-level linked list where nodes have two pointers down and next. In the previous post, we flattened the linked list level wise. How to flatten a linked list when we always need to process down pointer before next at every node.

```
Input:
1 - 2 - 3 - 4
    |
    7 -  8 - 10 - 12
    |    |    |
    9    16   11
    |    |
    14   17 - 18 - 19 - 20
    |                   |
    15 - 23             21
         |
         24

Output:
Linked List to be flattened to
1 - 2 - 7 - 9 - 14 - 15 - 23 - 24 - 8
 - 16 - 17 - 18 - 19 - 20 - 21 - 10 -
11 - 12 - 3 - 4
Note : 9 appears before 8 (When we are
at a node, we process down pointer before
right pointer)
```

Source : Oracle Interview

If we take a closer look, we can notice that this problem is similar to tree to linked list conversion. We recursively flatten a linked list with

following steps.

1) If node is NULL, return NULL.
2) Store next node of current node (used in step 4).
3) Recursively flatten down list. While flattening, keep track of last visited node, so that the next list can be linked after it.
4) Recursively flatten next list (we get the next list from pointer stored in step 2) and attach it after last visited node.

Below is C++ implementation of above idea.

```
// C++ program to flatten a multilevel linked list
#include <bits/stdc++.h>
using namespace std;

// A Linked List Node
struct Node
{
    int data;
    struct Node *next;
    struct Node *down;
};

// Flattens a multi-level linked list depth wise
Node* flattenList(Node* node)
{
    // Base case
    if (node == NULL)
        return NULL;

    // To keep track of last visited node
    // (NOTE: This is static)
    static Node *last;
    last = node;

    // Store next pointer
    Node *next = node->next;

    // If down list exists, process it first
    // Add down list as next of current node
    if (node->down)
       node->next = flattenList(node->down);

    // If next exists, add it after the next
    // of last added node
    if (next)
       last->next = flattenList(next);

    return node;
}

// Utility method to print a linked list
void printFlattenNodes(Node* head)
{
    while (head)
    {
        printf("%d ", head->data);
        head = head->next;
    }
}

// Utility function to create a new node
Node* newNode(int new_data)
{
    Node* new_node = new Node;
    new_node->data = new_data;
    new_node->next = new_node->down = NULL;
    return new_node;
}

// Driver code
int main()
{
    // Creating above example list
    Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
```

```
    head->next->next->next = newNode(4);
    head->next->down = newNode(7);
    head->next->down->down = newNode(9);
    head->next->down->down->down = newNode(14);
    head->next->down->down->down->down
                                    = newNode(15);
    head->next->down->down->down->down->next
                                    = newNode(23);
    head->next->down->down->down->down->next->down
                                     = newNode(24);
    head->next->down->next = newNode(8);
    head->next->down->next->down = newNode(16);
    head->next->down->next->down->down = newNode(17);
    head->next->down->next->down->down->next
                                     = newNode(18);
    head->next->down->next->down->down->next->next
                                     = newNode(19);
    head->next->down->next->down->down->next->next->next
                                     = newNode(20);
    head->next->down->next->down->down->next->next->next->down
                                     = newNode(21);
    head->next->down->next->next = newNode(10);
    head->next->down->next->next->down = newNode(11);

    head->next->down->next->next->next = newNode(12);

    // Flatten list and print modified list
    head = flattenList(head);
    printFlattenNodes(head);

    return 0;
}
```

Output:

```
1 2 7 9 14 15 23 24 8 16 17 18 19 20 21 10 11 12 3 4
```

This article is contributed by **Mu Ven**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

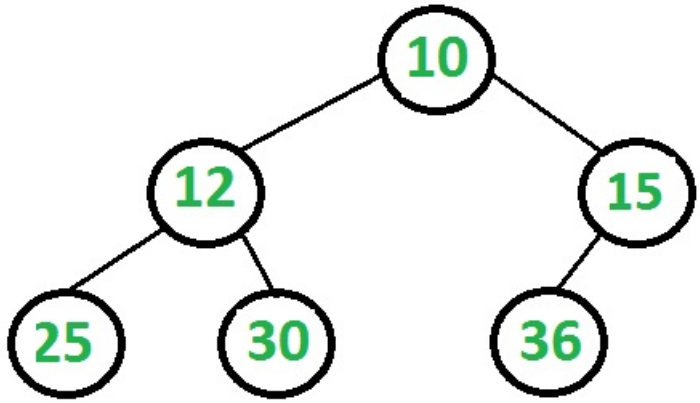
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Company Wise Coding Practice Â Â Â Topic Wise Coding Practice

Given a Binary Tree, convert it to a Circular Doubly Linked List (In-Place).

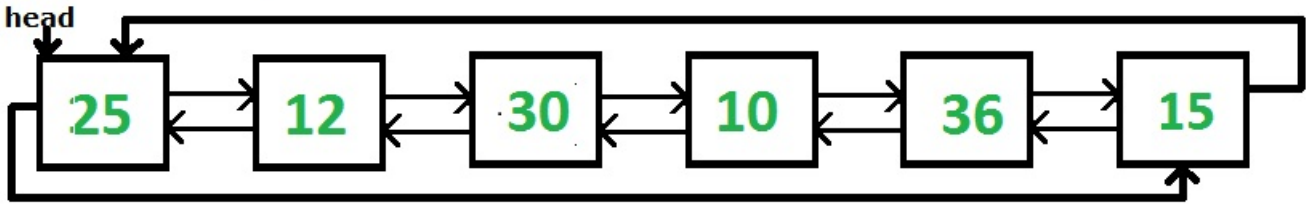
- The left and right pointers in nodes are to be used as previous and next pointers respectively in converted Circular Linked List.
- The order of nodes in List must be same as Inorder of the given Binary Tree.
- The first node of Inorder traversal must be head node of the Circular List.

Example:



The above tree should be in-place converted to following

Circular Doubly Linked List



The idea can be described using below steps.
1) Write a general purpose function that concatenates two given circular doubly lists (This function is explained below).
2) Now traverse the given tree
â€¦.a) Recursively convert left subtree to a circular DLL. Let the converted list be leftList.
â€¦.a) Recursively convert right subtree to a circular DLL. Let the converted list be rightList.
â€¦.c) Make a circular linked list of root of the tree, make left and right of root to point to itself.
â€¦.d) Concatenate leftList with list of single root node.
â€¦.e) Concatenate the list produced in step above (d) with rightList.

Note that the above code traverses tree in Postorder fashion. We can traverse in inorder fashion also. We can first concatenate left subtree and root, then recur for right subtree and concatenate the result with left-root concatenation.

How to Concatenate two circular DLLs?

- Get the last node of the left list. Retrieving the last node is an O(1) operation, since the prev pointer of the head points to the last node of the list.
- Connect it with the first node of the right list
- Get the last node of the second list
- Connect it with the head of the list.

Below are implementations of above idea.

C++

```
// C++ Program to convert a Binary Tree
// to a Circular Doubly Linked List
#include<iostream>
using namespace std;

// To represents a node of a Binary Tree
struct Node
{
    struct Node *left, *right;
    int data;
};

// A function that appends rightList at the end
// of leftList.
Node *concatenate(Node *leftList, Node *rightList)
{
    // If either of the list is empty
```

```
    // then return the other list
    if (leftList == NULL)
        return rightList;
    if (rightList == NULL)
        return leftList;

    // Store the last Node of left List
    Node *leftLast = leftList->left;

    // Store the last Node of right List
    Node *rightLast = rightList->left;

    // Connect the last node of Left List
    // with the first Node of the right List
    leftLast->right = rightList;
    rightList->left = leftLast;

    // Left of first node points to
    // the last node in the list
    leftList->left = rightLast;

    // Right of last node refers to the first
    // node of the List
    rightLast->right = leftList;

    return leftList;
}

// Function converts a tree to a circular Linked List
// and then returns the head of the Linked List
Node *bTreeToCList(Node *root)
{
    if (root == NULL)
        return NULL;

    // Recursively convert left and right subtrees
    Node *left = bTreeToCList(root->left);
    Node *right = bTreeToCList(root->right);

    // Make a circular linked list of single node
    // (or root). To do so, make the right and
    // left pointers of this node point to itself
    root->left = root->right = root;

    // Step 1 (concatenate the left list with the list
    //         with single node, i.e., current node)
    // Step 2 (concatenate the returned list with the
    //         right List)
    return concatenate(concatenate(left, root), right);
}

// Display Circular Link List
void displayCList(Node *head)
{
    cout << "Circular Linked List is :\n";
    Node *itr = head;
    do
    {
        cout << itr->data <<" ";
        itr = itr->right;
    } while (head!=itr);
    cout << "\n";
}


// Create a new Node and return its address
Node *newNode(int data)
{
    Node *temp = new Node();
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver Program to test above function
int main()
```

```
{
    Node *root = newNode(10);
    root->left = newNode(12);
    root->right = newNode(15);
    root->left->left = newNode(25);
    root->left->right = newNode(30);
    root->right->left = newNode(36);

    Node *head = bTreeToCList(root);
    displayCList(head);

    return 0;
}
```

Java

```
// Java Program to convert a Binary Tree to a
// Circular Doubly Linked List

// Node class represents a Node of a Tree
class Node
{
    int val;
    Node left,right;

    public Node(int val)
    {
        this.val = val;
        left = right = null;
    }
}

// A class to represent a tree
class Tree
{
    Node root;
    public Tree()
    {
        root = null;
    }

    // concatenate both the lists and returns the head
    // of the List
    public Node concatenate(Node leftList,Node rightList)
    {
        // If either of the list is empty, then
        // return the other list
        if (leftList == null)
            return rightList;
        if (rightList == null)
            return leftList;

        // Store the last Node of left List
        Node leftLast = leftList.left;

        // Store the last Node of right List
        Node rightLast = rightList.left;

        // Connect the last node of Left List
        // with the first Node of the right List
        leftLast.right = rightList;
        rightList.left = leftLast;

        // left of first node refers to
        // the last node in the list
        leftList.left = rightLast;

        // Right of last node refers to the first
        // node of the List
        rightLast.right = leftList;

        // Return the Head of the List
        return leftList;
```

```
    }

    // Method converts a tree to a circular
    // Link List and then returns the head
    // of the Link List
    public Node bTreeToCList(Node root)
    {
        if (root == null)
            return null;

        // Recursively convert left and right subtrees
        Node left = bTreeToCList(root.left);
        Node right = bTreeToCList(root.right);

        // Make a circular linked list of single node
        // (or root). To do so, make the right and
        // left pointers of this node point to itself
        root.left = root.right = root;

        // Step 1 (concatenate the left list with the list
        //         with single node, i.e., current node)
        // Step 2 (concatenate the returned list with the
        //         right List)
        return concatenate(concatenate(left, root), right);
    }

    // Display Circular Link List
    public void display(Node head)
    {
        System.out.println("Circular Linked List is :");
        Node itr = head;
        do
        {
            System.out.print(itr.val+ " " );
            itr = itr.right;
        }
        while (itr != head);
        System.out.println();
    }
}

// Driver Code
class Main
{
    public static void main(String args[])
    {
        // Build the tree
        Tree tree = new Tree();
        tree.root = new Node(10);
        tree.root.left = new Node(12);
        tree.root.right = new Node(15);
        tree.root.left.left = new Node(25);
        tree.root.left.right = new Node(30);
        tree.root.right.left = new Node(36);

        // head refers to the head of the Link List
        Node head = tree.bTreeToCList(tree.root);

        // Display the Circular LinkedList
        tree.display(head);
    }
}
```

Output:

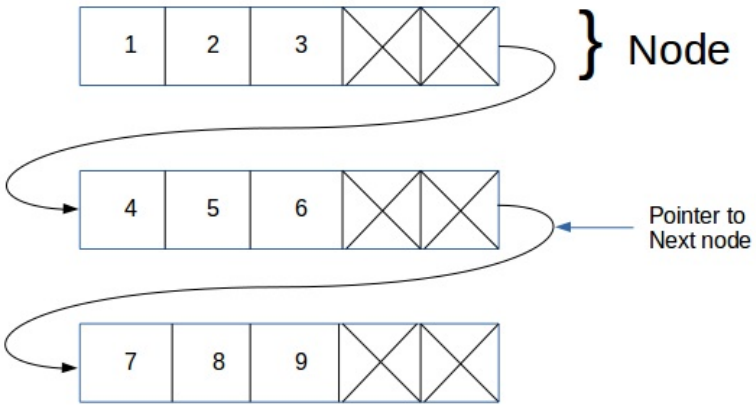
```
Circular Linked List is :
25 12 30 10 36 15
```

This article is contributed by **Chirag Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Company Wise Coding Practice Â Â Â Topic Wise Coding Practice

Like array and linked list, unrolled Linked List is also a linear data structure and is a variant of linked list. Unlike simple linked list, it stores multiple elements at each node. That is, instead of storing single element at a node, unrolled linked lists store an array of elements at a node. Unrolled linked list covers advantages of both array and linked list as it reduces the memory overhead in comparison to simple linked lists by storing multiple elements at each node and it also has the advantage of fast insertion and deletion as that of a linked list.



Advantages:

- Because of the Cache behavior, linear search is much faster in unrolled linked lists.
- In comparison to ordinary linked list, it requires less storage space for pointers/references.
- It performs operations like insertion, deletion and traversal more quickly than ordinary linked lists (because search is faster).

Disadvantages:

- The overhead per node is comparatively high than singly linked lists. Refer an example node in below code.

Simple Implementation in C

The below program creates a simple unrolled linked list with 3 nodes containing variable number of elements in each. It also traverses the created list.

```
// C program to implement unrolled linked list
// and traversing it.
#include<stdio.h>
#include<stdlib.h>
#define maxElements 4

// Unrolled Linked List Node
struct Node
{
    int numElements;
    int array[maxElements];
    struct Node *next;
};

/* Function to traverse am unrolled linked list
   and print all the elements*/
```

```
void printUnrolledList(struct Node *n)
{
    while (n != NULL)
    {
        // Print elements in current node
        for (int i=0; i<n->numElements; i++)
            printf("%d ", n->array[i]);

        // Move to next node
        n = n->next;
    }
}

// Program to create an unrolled linked list
// with 3 Nodes
int main()
{
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;

    // allocate 3 Nodes
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));

    // Let us put some values in second node (Number
    // of values must be less than or equal to
    // maxElement)
    head->numElements = 3;
    head->array[0] = 1;
    head->array[1] = 2;
    head->array[2] = 3;

    // Link first Node with the second Node
    head->next = second;

    // Let us put some values in second node (Number
    // of values must be less than or equal to
    // maxElement)
    second->numElements = 3;
    second->array[0] = 4;
    second->array[1] = 5;
    second->array[2] = 6;

    // Link second Node with the third Node
    second->next = third;

    // Let us put some values in third node (Number
    // of values must be less than or equal to
    // maxElement)
    third->numElements = 3;
    third->array[0] = 7;
    third->array[1] = 8;
    third->array[2] = 9;
    third->next = NULL;

    printUnrolledList(head);

    return 0;
}
```

Output:

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

In this article, we have introduced unrolled list and advantages of it. We have also shown how to traverse the list. In the next article, we will be discussing insertion, deletion and values of maxElements/numElements in detail.

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Company Wise Coding Practice Â Â Â Topic Wise Coding Practice