[Date]

Complete placement course

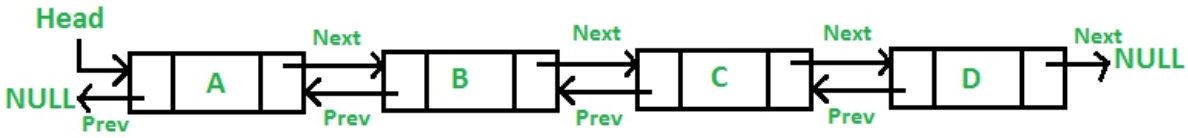
Doubly linked list

ANIL KUMAR DWIVEDI
BADA VIDYALAY

Doubly Linked List

introduction and Insertion

A **D**oubly **L**inked **L**ist (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.



Following is representation of a DLL node in C language.

```
/* Node of a doubly linked list */
struct Node {
    int data;
    struct Node* next; // Pointer to next node in DLL
    struct Node* prev; // Pointer to previous node in DLL
};
```

Advantages over singly linked list

1) A DLL can be traversed in both forward and backward direction.
2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
3) We can quickly insert a new node before a given node.
In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Disadvantages over singly linked list

1) Every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer though (See this and this).

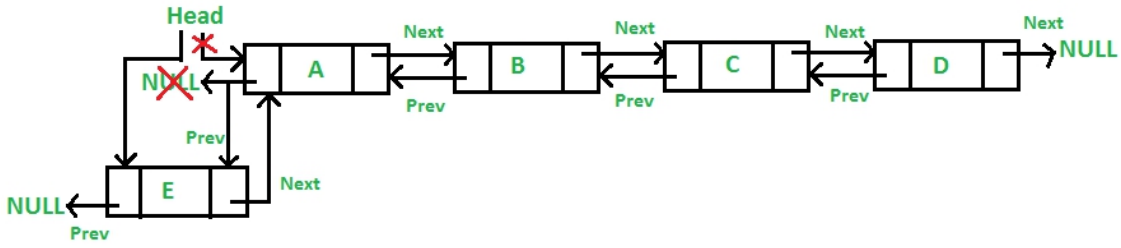
2) All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.

Insertion

A node can be added in four ways
1) At the front of the DLL
2) After a given node.
3) At the end of the DLL
4) Before a given node.

1) Add a node at the front: (A 5 steps process)

The new node is always added before the head of the given Linked List. And newly added node becomes the new head of DLL.
For example if the given Linked List is 10152025 and we add an item 5 at the front, then the Linked List becomes 510152025.
Let us call the function that adds at the front of the list is push().
The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node

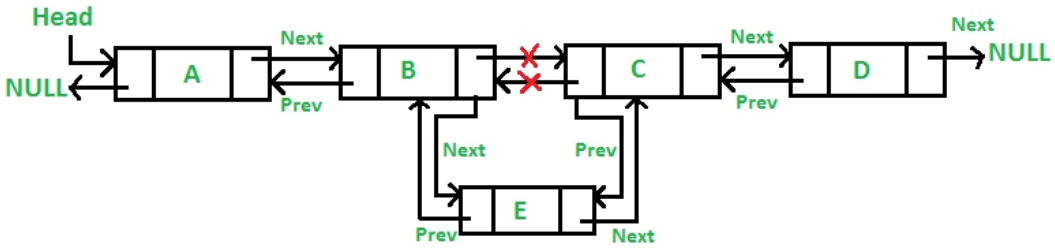


Following are the 5 steps to add node at the front.

```c
/* Given a reference (pointer to pointer) to the head of a list
   and an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* 2. put in the data  */
    new_node->data = new_data;

    /* 3. Make next of new node as head and previous as NULL */
    new_node->next = (*head_ref);
    new_node->prev = NULL;

    /* 4. change prev of head node to new node */
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    /* 5. move the head to point to the new node */
    (*head_ref) = new_node;
}
```

2) Add a node after a given node.: (A 7 steps process)

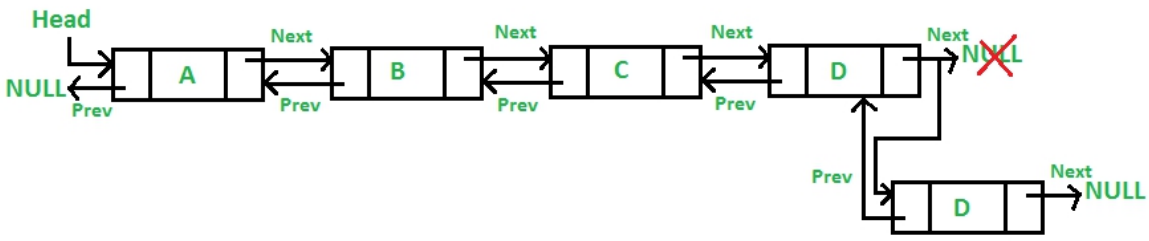
We are given pointer to a node as prev_node, and the new node is inserted after the given node.



```c
/* Given a node as prev_node, insert a new node after the given node */
void insertAfter(struct Node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL) {
        printf("the given previous node cannot be NULL");
        return;
    }

    /* 2. allocate new node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* 3. put in the data  */
    new_node->data = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. Make the next of prev_node as new_node */
    prev_node->next = new_node;

    /* 6. Make prev_node as previous of new_node */
    new_node->prev = prev_node;

    /* 7. Change previous of new_node's next node */
    if (new_node->next != NULL)
        new_node->next->prev = new_node;
}
```

3) Add a node at the end: (7 steps process)

The new node is always added after the last node of the given Linked List. For example if the given DLL is 510152025 and we add an item 30 at the end, then the DLL becomes 51015202530.
Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.

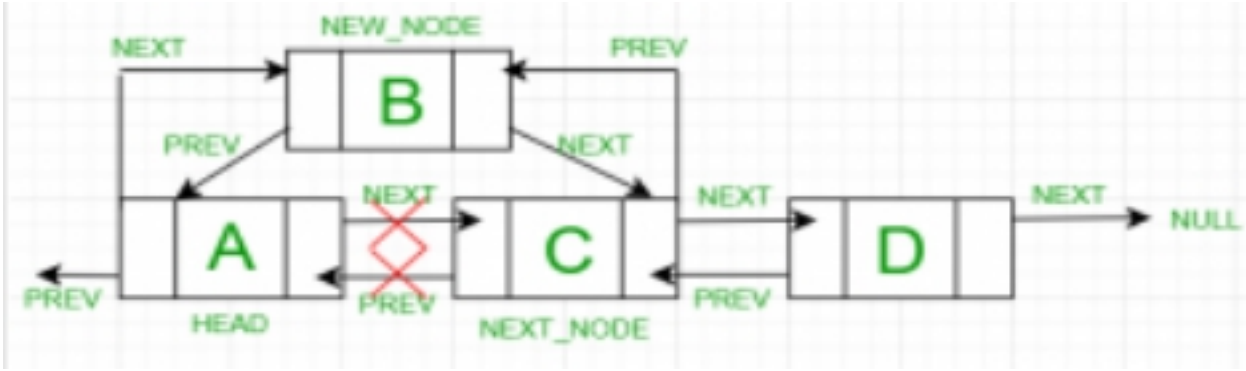


```c
/* Given a reference (pointer to pointer) to the head
of a DLL and an int, appends a new node at the end */
void append(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    struct Node* last = *head_ref; /* used in step 5*/
    /* 2. put in the data */
    new_node->data = new_data;
    /* 3. This new node is going to be the last node, so
        make next of it as NULL*/
    new_node->next = NULL;
    /* 4. If the Linked List is empty, then make the new
        node as head */
    if (*head_ref == NULL) {
        new_node->prev = NULL;
        *head_ref = new_node;
        return;
    }
    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;
    /* 6. Change the next of last node */
    last->next = new_node;
    /* 7. Make last node as previous of new node */
    new_node->prev = last;
    return;
}

```

4) Add a node before a given node:

Steps
Let the pointer to this given node be next_node and the data of the new node to be added as new_data.

1. Check if the next_node is NULL or not. If it's NULL, return from the function because any new node can not be added before a NULL
2. Allocate memory for the new node, let it be called new_node
3. Set new_node->data = new_data
4. Set the previous pointer of this new_node as the previous node of the next_node, new_node->prev = next_node->prev
5. Set the previous pointer of the next_node as the new_node, next_node->prev = new_node
6. Set the next pointer of this new_node as the next_node, new_node->next = next_node;
7. If the previous node of the new_node is not NULL, then set the next pointer of this previous node as new_node, new_node->prev->next = new_node
8. Else, if the prev of new_node is NULL, it will be the new head node. So, make (*head_ref) = new_node.



Below is the implementation of the above approach:

```c
// A complete working C program to demonstrate all
// insertion before a given node
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

/* Given a reference (pointer to pointer) to the head of a
list and an int, inserts a new node on the front of the
list. */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node
        = (struct Node*)malloc(sizeof(struct Node));

    new_node->data = new_data;

    new_node->next = (*head_ref);
    new_node->prev = NULL;

    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    (*head_ref) = new_node;
}

/* Given a node as next_node, insert a new node before the
* given node */
void insertBefore(struct Node** head_ref,
                  struct Node* next_node, int new_data)
{
    /*1. check if the given next_node is NULL */
    if (next_node == NULL) {
        printf("the given next node cannot be NULL");
        return;
    }

```

```
    /* 2. allocate new node */
    struct Node* new_node
        = (struct Node*)malloc(sizeof(struct Node));

    /* 3. put in the data */
    new_node->data = new_data;

    /* 4. Make prev of new node as prev of next_node */
    new_node->prev = next_node->prev;

    /* 5. Make the prev of next_node as new_node */
    next_node->prev = new_node;

    /* 6. Make next_node as next of new_node */
    new_node->next = next_node;

    /* 7. Change next of new_node's previous node */
    if (new_node->prev != NULL)
        new_node->prev->next = new_node;
    /* 8. If the prev of new_node is NULL, it will be
    the new head node */
    else
        (*head_ref) = new_node;
}

// This function prints contents of linked list starting
// from the given node
void printList(struct Node* node)
{
    struct Node* last;
    printf("\nTraversal in forward direction \n");
    while (node != NULL) {
        printf(" %d ", node->data);
        last = node;
        node = node->next;
    }

    printf("\nTraversal in reverse direction \n");
    while (last != NULL) {
        printf(" %d ", last->data);
        last = last->prev;
    }
}
```

```
86
87  /* Driver program to test above functions*/
88  int main()
89  {
90      /* Start with the empty list */
91      struct Node* head = NULL;
92      push(&head, 7);
93
94      push(&head, 1);
95
96      push(&head, 4);
97
98      // Insert 8, before 1. So linked list becomes
99      // 4->8->1->7->NULL
100     insertBefore(&head, head->next, 8);
101
102     printf("Created DLL is: ");
103     printList(head);
104
105     getchar();
106     return 0;
107 }
```

Output:

```
Created DLL is:
Traversal in forward direction
 4  8  1  7
Traversal in reverse direction
 7  1  8  4
```

A complete working program to test above functions.
Following is complete program to test above functions.

```cpp
// A complete working C++ program to
// demonstrate all insertion methods
#include <bits/stdc++.h>
using namespace std;

// A linked list node
class Node
{
    public:
    int data;
    Node* next;
    Node* prev;
};

/* Given a reference (pointer to pointer)
to the head of a list
and an int, inserts a new node on the
front of the list. */
void push(Node** head_ref, int new_data)
{
    /* 1. allocate node */
    Node* new_node = new Node();

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head
    and previous as NULL */
    new_node->next = (*head_ref);
    new_node->prev = NULL;

    /* 4. change prev of head node to new node */
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    /* 5. move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Given a node as prev_node, insert
a new node after the given node */
void insertAfter(Node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
        cout<<"the given previous node cannot be NULL";
```

```
48          return;
49      }
50
51      /* 2. allocate new node */
52      Node* new_node = new Node();
53
54      /* 3. put in the data */
55      new_node->data = new_data;
56
57      /* 4. Make next of new node as next of prev_node */
58      new_node->next = prev_node->next;
59
60      /* 5. Make the next of prev_node as new_node */
61      prev_node->next = new_node;
62
63      /* 6. Make prev_node as previous of new_node */
64      new_node->prev = prev_node;
65
66      /* 7. Change previous of new_node's next node */
67      if (new_node->next != NULL)
68          new_node->next->prev = new_node;
69  }
70
71  /* Given a reference (pointer to pointer) to the head
72  of a DLL and an int, appends a new node at the end */
73  void append(Node** head_ref, int new_data)
74  {
75      /* 1. allocate node */
76      Node* new_node = new Node();
77
78      Node* last = *head_ref; /* used in step 5*/
79
80      /* 2. put in the data */
81      new_node->data = new_data;
82
83      /* 3. This new node is going to be the last node, so
84          make next of it as NULL*/
85      new_node->next = NULL;
86
87      /* 4. If the Linked List is empty, then make the new
88          node as head */
89      if (*head_ref == NULL)
90      {
91          new_node->prev = NULL;
92          *head_ref = new_node;
93          return;
```

```
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;

    /* 7. Make last node as previous of new node */
    new_node->prev = last;

    return;
}

// This function prints contents of
// linked list starting from the given node
void printList(Node* node)
{
    Node* last;
    cout<<"\nTraversal in forward direction \n";
    while (node != NULL)
    {
        cout<<" "<<node->data<<" ";
        last = node;
        node = node->next;
    }

    cout<<"\nTraversal in reverse direction \n";
    while (last != NULL)
    {
        cout<<" "<<last->data<<" ";
        last = last->prev;
    }
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    Node* head = NULL;

    // Insert 6. So linked list becomes 6->NULL
    append(&head, 6);

```

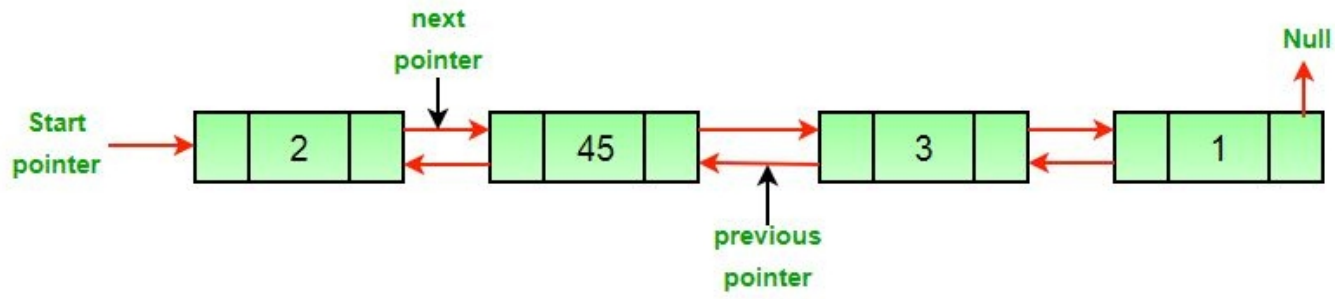
```
139     // Insert 7 at the beginning. So
140     // linked list becomes 7->6->NULL
141     push(&head, 7);
142
143     // Insert 1 at the beginning. So
144     // linked list becomes 1->7->6->NULL
145     push(&head, 1);
146
147     // Insert 4 at the end. So linked
148     // list becomes 1->7->6->4->NULL
149     append(&head, 4);
150
151     // Insert 8, after 7. So linked
152     // list becomes 1->7->8->6->4->NULL
153     insertAfter(head->next, 8);
154
155     cout << "Created DLL is: ";
156     printList(head);
157
158     return 0;
159 }
```

Output:

```
 Created DLL is:
Traversal in forward direction
 1  7  8  6  4
Traversal in reverse direction
 4  6  8  7  1
```

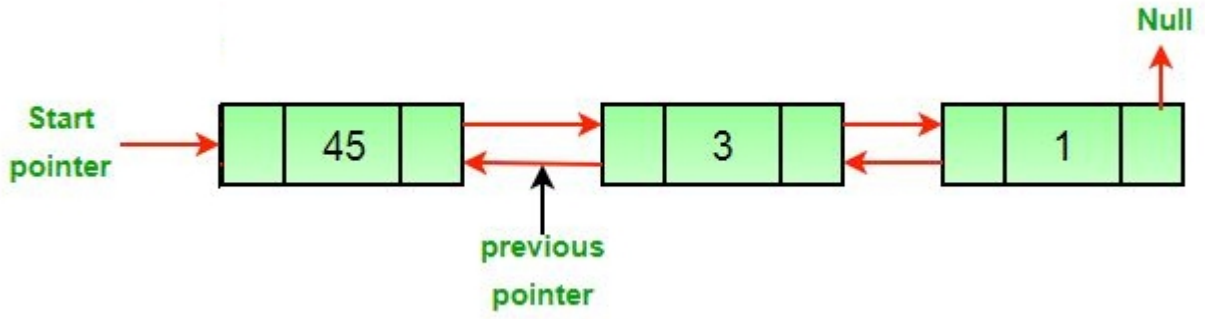
Delete a node in a Doubly Linked List

Write a function to delete a given node in a doubly-linked list.
Original Doubly Linked List

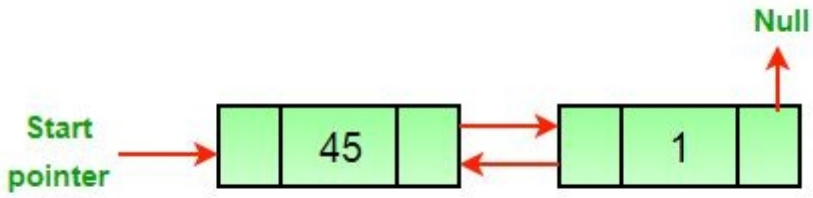


Approach: The deletion of a node in a doubly-linked list can be divided into three main categories:

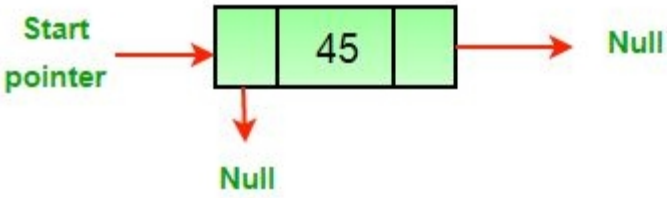
- **After the deletion of the head node.**



- **After the deletion of the middle node.**

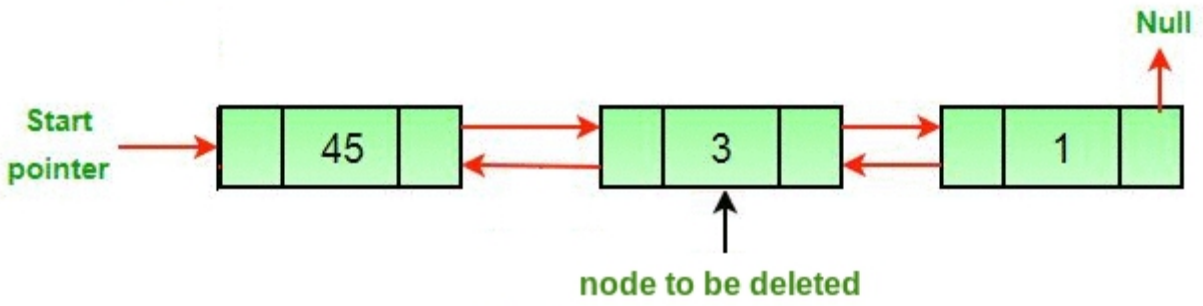


- **After the deletion of the last node.**



All three mentioned cases can be handled in two steps if the pointer of the node to be deleted and the head pointer is known.

1. If the node to be deleted is the head node then make the next node as head.
2. If a node is deleted, connect the next and previous node of the deleted node.



Algorithm

- Let the node to be deleted be *del*.
- If node to be deleted is head node, then change the head pointer to next current head.

```
if headnode == del then
      headnode =  del.nextNode
```

- Set *next* of previous to *del*, if previous to *del* exists.

```
if del.nextNode != none
      del.nextNode.previousNode = del.previousNode
```

- Set *prev* of next to *del*, if next to *del* exists.

```
if del.previousNode != none
      del.previousNode.nextNode = del.next
```

Complexity Analysis:

- **Time Complexity:** O(1).
  Since traversal of the linked list is not required so the time complexity is constant.
- **Space Complexity:** O(1).
  As no extra space is required, so the space complexity is constant.

```cpp
// C++ program to delete a node from
// Doubly Linked List
#include <bits/stdc++.h>
using namespace std;

/* a node of the doubly linked list */
class Node
{
    public:
    int data;
    Node* next;
    Node* prev;
};

/* Function to delete a node in a Doubly Linked List.
head_ref --> pointer to head node pointer.
del --> pointer to node to be deleted. */
void deleteNode(Node** head_ref, Node* del)
{
    /* base case */
    if (*head_ref == NULL || del == NULL)
        return;

    /* If node to be deleted is head node */
    if (*head_ref == del)
        *head_ref = del->next;

    /* Change next only if node to be
    deleted is NOT the last node */
    if (del->next != NULL)
        del->next->prev = del->prev;

    /* Change prev only if node to be
    deleted is NOT the first node */
    if (del->prev != NULL)
        del->prev->next = del->next;

    /* Finally, free the memory occupied by del*/
    free(del);
    return;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the
beginning of the Doubly Linked List */
void push(Node** head_ref, int new_data)
{
```

```
48      /* allocate node */
49      Node* new_node = new Node();
50
51      /* put in the data */
52      new_node->data = new_data;
53
54      /* since we are adding at the beginning,
55      prev is always NULL */
56      new_node->prev = NULL;
57
58      /* link the old list off the new node */
59      new_node->next = (*head_ref);
60
61      /* change prev of head node to new node */
62      if ((*head_ref) != NULL)
63          (*head_ref)->prev = new_node;
64
65      /* move the head to point to the new node */
66      (*head_ref) = new_node;
67  }
68
69  /* Function to print nodes in a given doubly linked list
70  This function is same as printList() of singly linked list */
71  void printList(Node* node)
72  {
73      while (node != NULL)
74      {
75          cout << node->data << " ";
76          node = node->next;
77      }
78  }
79
80  /* Driver code*/
81  int main()
82  {
83      /* Start with the empty list */
84      Node* head = NULL;
85
86      /* Let us create the doubly linked list 10<->8<->4<->2 */
87      push(&head, 2);
88      push(&head, 4);
89      push(&head, 8);
90      push(&head, 10);
91
92      cout << "Original Linked list ";
93      printList(head);
94
95      /* delete nodes from the doubly linked list */
96      deleteNode(&head, head); /*delete first node*/
97      deleteNode(&head, head->next); /*delete middle node*/
98      deleteNode(&head, head->next); /*delete last node*/
99
100     /* Modified linked list will be NULL<-8->NULL */
101     cout << "\nModified Linked list ";
102     printList(head);
103
104     return 0;
105 }
```

Output:

```
Original Linked list 10 8 4 2
Modified Linked list 8
```

Reverse a Doubly Linked List

Here is a simple method for reversing a Doubly Linked List. All we need to do is swap prev and next pointers for all nodes, change prev of the head (or start) and change the head pointer in the end.

```cpp
/* C++ program to reverse a doubly linked list */
#include <bits/stdc++.h>
using namespace std;

/* a node of the doubly linked list */
class Node
{
    public:
    int data;
    Node *next;
    Node *prev;
};

/* Function to reverse a Doubly Linked List */
void reverse(Node **head_ref)
{
    Node *temp = NULL;
    Node *current = *head_ref;

    /* swap next and prev for all nodes of
    doubly linked list */
    while (current != NULL)
    {
        temp = current->prev;
        current->prev = current->next;
        current->next = temp;
        current = current->prev;
    }

    /* Before changing the head, check for the cases like empty
        list and list with only one node */
    if(temp != NULL )
        *head_ref = temp->prev;
}
```

```
35
36
37
38  /* UTILITY FUNCTIONS */
39  /* Function to insert a node at the
40  beginging of the Doubly Linked List */
41  void push(Node** head_ref, int new_data)
42  {
43      /* allocate node */
44      Node* new_node = new Node();
45
46      /* put in the data */
47      new_node->data = new_data;
48
49      /* since we are adding at the beginning,
50      prev is always NULL */
51      new_node->prev = NULL;
52
53      /* link the old list off the new node */
54      new_node->next = (*head_ref);
55
56      /* change prev of head node to new node */
57      if((*head_ref) != NULL)
58      (*head_ref)->prev = new_node ;
59
60      /* move the head to point to the new node */
61      (*head_ref) = new_node;
62  }
63
64  /* Function to print nodes in a given doubly linked list
65  This function is same as printList() of singly linked list */
66  void printList(Node *node)
67  {
68      while(node != NULL)
69      {
70          cout << node->data << " ";
71          node = node->next;
72      }
73  }
74
75  /* Driver code */
76  int main()
77  {
78      /* Start with the empty list */
79      Node* head = NULL;
```

```
80
81      /* Let us create a sorted linked list to test the functions
82      Created linked list will be 10->8->4->2 */
83      push(&head, 2);
84      push(&head, 4);
85      push(&head, 8);
86      push(&head, 10);
87
88      cout << "Original Linked list" << endl;
89      printList(head);
90
91      /* Reverse doubly linked list */
92      reverse(&head);
93
94      cout << "\nReversed Linked list" << endl;
95      printList(head);
96
97      return 0;
98  }
99
```

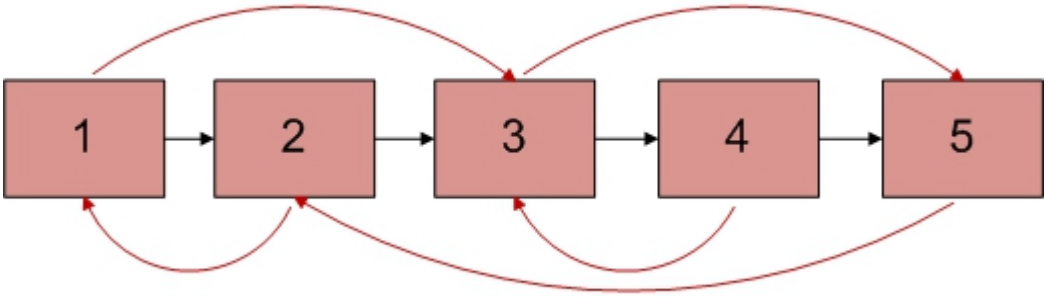
Output:

```
Original linked list
10 8 4 2
The reversed Linked List is
2 4 8 10
```

Time Complexity: *O(N), where N denotes the number of nodes in the doubly linked list.*
Auxiliary Space: O(1)

Clone a linked list with next and random pointer

You are given a Double Link List with one pointer of each node pointing to the next node just like in a single link list. The second pointer however CAN point to any node in the list and not just the previous node. Now write a program in **O(n) time** to duplicate this list. That is, write a program which will create a copy of this list.
Let us call the second pointer as arbit pointer as it can point to any arbitrary node in the linked list.



Uses Constant Extra Space

1) Create the copy of node 1 and insert it between node 1 & node 2 in original Linked List, create the copy of 2 and insert it between 2 & 3.. Continue in this fashion, add the copy of N afte the Nth node
2) Now copy the arbitrary link in this fashion

```
     original->next->arbitrary = original->arbitrary->next;  /*TRAVERSE
TWO NODES*/
```

This works because original->next is nothing but copy of original and Original->arbitrary->next is nothing but copy of arbitrary.
3) Now restore the original and copy linked lists in this fashion in a single loop.

```
     original->next = original->next->next;
     copy->next = copy->next->next;
```

4) Make sure that last element of original->next is NULL.

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

```cpp
// C++ program to clone a linked list with next
// and arbit pointers in O(n) time
#include <bits/stdc++.h>
using namespace std;

// Structure of linked list Node
struct Node
{
    int data;
    Node *next,*random;
    Node(int x)
    {
        data = x;
        next = random = NULL;
    }
};

// Utility function to print the list.
void print(Node *start)
{
    Node *ptr = start;
    while (ptr)
    {
        cout << "Data = " << ptr->data << ", Random = "
             << ptr->random->data << endl;
        ptr = ptr->next;
    }
}

// This function clones a given linked list
// in O(1) space
Node* clone(Node *start)
{
    Node* curr = start, *temp;

    // insert additional node after
    // every node of original list
    while (curr)
    {
        temp = curr->next;

        // Inserting node
        curr->next = new Node(curr->data);
        curr->next->next = temp;
        curr = temp;
    }

```

```
    curr = start;

    // adjust the random pointers of the
    // newly added nodes
    while (curr)
    {
        if(curr->next)
            curr->next->random = curr->random ?
                                 curr->random->next : curr->random;

        // move to the next newly added node by
        // skipping an original node
        curr = curr->next?curr->next->next:curr->next;
    }

    Node* original = start, *copy = start->next;

    // save the start of copied linked list
    temp = copy;

    // now separate the original list and copied list
    while (original && copy)
    {
        original->next =
        original->next? original->next->next : original->next;

        copy->next = copy->next?copy->next->next:copy->next;
        original = original->next;
        copy = copy->next;
    }

    return temp;
}

// Driver code
int main()
{
    Node* start = new Node(1);
    start->next = new Node(2);
    start->next->next = new Node(3);
    start->next->next->next = new Node(4);
    start->next->next->next->next = new Node(5);

    // 1's random points to 3
    start->random = start->next->next;

```

```
94      // 2's random points to 1
95      start->next->random = start;
96
97      // 3's and 4's random points to 5
98      start->next->next->random =
99                      start->next->next->next->next;
100     start->next->next->next->random =
101                     start->next->next->next->next;
102
103     // 5's random points to 2
104     start->next->next->next->next->random =
105                                     start->next;
106
107     cout << "Original list : \n";
108     print(start);
109
110     cout << "\nCloned list : \n";
111     Node *cloned_list = clone(start);
112     print(cloned_list);
113
114     return 0;
115 }
```

Output

```
Original list :
Data = 1, Random  = 3
Data = 2, Random  = 1
Data = 3, Random  = 5
Data = 4, Random  = 5
Data = 5, Random  = 2

Cloned list :
Data = 1, Random  = 3
Data = 2, Random  = 1
Data = 3, Random  = 5
Data = 4, Random  = 5
Data = 5, Random  = 2
```

QuickSort on Doubly Linked List

Following is a typical recursive implementation of quick sort for arrays. The implementation uses last element as pivot.

```
/* A typical recursive implementation of Quicksort for array*/

/* This function takes last element as pivot, places the pivot element at its
   correct position in sorted array, and places all smaller (smaller than
   pivot) to left of pivot and all greater elements to right of pivot */
int partition (int arr[], int l, int h)
{
    int x = arr[h];
    int i = (l - 1);


    for (int j = l; j <= h- 1; j++)
    {
        if (arr[j] <= x)
        {
            i++;
            swap (&arr[i], &arr[j]);
        }
    }
    swap (&arr[i + 1], &arr[h]);
    return (i + 1);
}

/* A[] --> Array to be sorted, l  --> Starting index, h  --> Ending index */
void quickSort(int A[], int l, int h)
{
    if (l < h)
    {
        int p = partition(A, l, h); /* Partitioning index */
        quickSort(A, l, p - 1);
        quickSort(A, p + 1, h);
    }
}
```

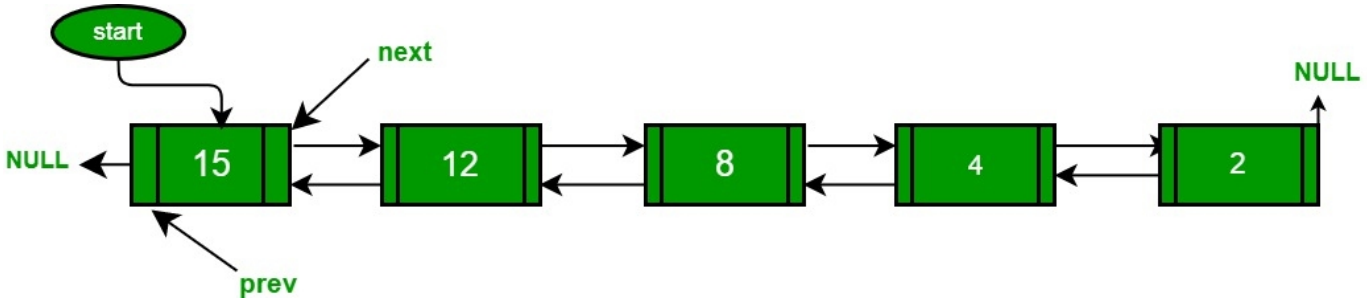
Can we use the same algorithm for Linked List?

Following is C++ implementation for the doubly linked list. The idea is simple, we first find out pointer to the last node.

Once we have a pointer to the last node, we can recursively sort the linked list using pointers to first and last nodes of a linked list, similar to the above recursive function where we pass indexes of first and last array elements.

The partition function for a linked list is also similar to partition for arrays. Instead of returning index of the pivot element, it returns a pointer to the pivot element.

In the following implementation, quickSort() is just a wrapper function, the main recursive function is _quickSort() which is similar to quickSort() for array implementation.




```cpp
// A C++ program to sort a linked list using Quicksort
#include <bits/stdc++.h>
using namespace std;

/* a node of the doubly linked list */
class Node
{
    public:
    int data;
    Node *next;
    Node *prev;
};

/* A utility function to swap two elements */
void swap ( int* a, int* b )
{ int t = *a; *a = *b; *b = t; }

// A utility function to find
// last node of linked list
Node *lastNode(Node *root)
{
    while (root && root->next)
        root = root->next;
    return root;
}

/* Considers last element as pivot,
places the pivot element at its
correct position in sorted array,
and places all smaller (smaller than
pivot) to left of pivot and all greater
elements to right of pivot */
Node* partition(Node *l, Node *h)
{
    // set pivot as h element
    int x = h->data;

    // similar to i = l-1 for array implementation
    Node *i = l->prev;

    // Similar to "for (int j = l; j <= h- 1; j++)"
    for (Node *j = l; j != h; j = j->next)
    {
        if (j->data <= x)
        {
            // Similar to i++ for array
            i = (i == NULL)? l : i->next;
```

```
48
49                  swap(&(i->data), &(j->data));
50              }
51          }
52          i = (i == NULL)? l : i->next; // Similar to i++
53          swap(&(i->data), &(h->data));
54          return i;
55      }
56
57      /* A recursive implementation
58      of quicksort for linked list */
59      void _quickSort(Node* l, Node *h)
60      {
61          if (h != NULL && l != h && l != h->next)
62          {
63              Node *p = partition(l, h);
64              _quickSort(l, p->prev);
65              _quickSort(p->next, h);
66          }
67      }
68
69      // The main function to sort a linked list.
70      // It mainly calls _quickSort()
71      void quickSort(Node *head)
72      {
73          // Find last node
74          Node *h = lastNode(head);
75
76          // Call the recursive QuickSort
77          _quickSort(head, h);
78      }
79
80      // A utility function to print contents of arr
81      void printList(Node *head)
82      {
83          while (head)
84          {
85              cout << head->data << " ";
86              head = head->next;
87          }
88          cout << endl;
89      }
90
91      /* Function to insert a node at the
92      beginging of the Doubly Linked List */
93      void push(Node** head_ref, int new_data)
```

```
{
    Node* new_node = new Node; /* allocate node */
    new_node->data = new_data;

    /* since we are adding at the
    beginning, prev is always NULL */
    new_node->prev = NULL;

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

    /* change prev of head node to new node */
    if ((*head_ref) != NULL) (*head_ref)->prev = new_node ;

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

/* Driver code */
int main()
{
    Node *a = NULL;
    push(&a, 5);
    push(&a, 20);
    push(&a, 4);
    push(&a, 3);
    push(&a, 30);

    cout << "Linked List before sorting \n";
    printList(a);

    quickSort(a);

    cout << "Linked List after sorting \n";
    printList(a);

    return 0;
}
```

Output

```
Linked List before sorting
30  3  4  20  5
Linked List after sorting
3  4  5  20  30
```

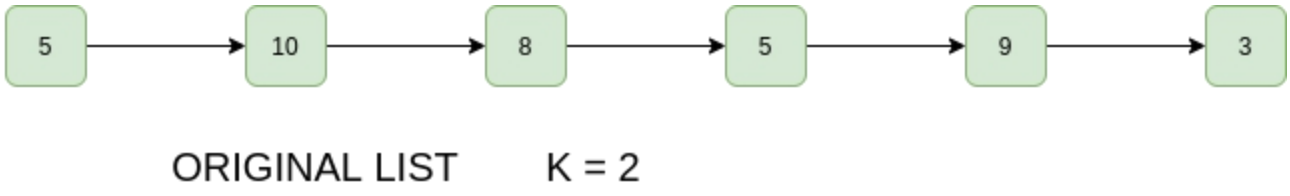
Swap Kth node from beginning with Kth node from end in a Linked List

Given a singly linked list, swap kth node from beginning with kth node from end. **Swapping of data is not allowed, only pointers should be changed.** This requirement may be logical in many situations where the linked list data part is huge (For example student details line Name, RollNo, Address, ..etc). The pointers are always fixed (4 bytes for most of the compilers).

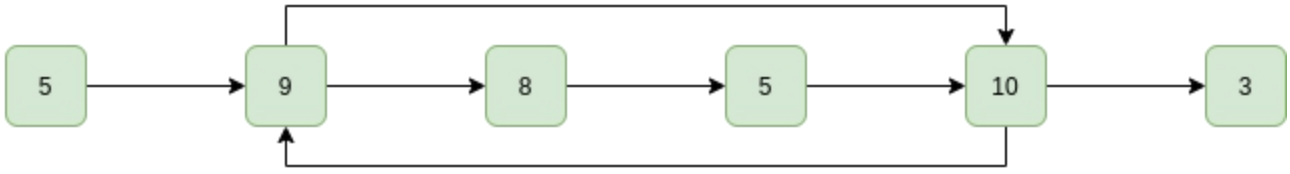
Example:

```
Input: 1 -> 2 -> 3 -> 4 -> 5, K = 2
Output: 1 -> 4 -> 3 -> 2 -> 5
Explanation: The 2nd node from 1st is 2 and
2nd node from last is 4, so swap them.


Input: 1 -> 2 -> 3 -> 4 -> 5, K = 5
Output: 5 -> 2 -> 3 -> 4 -> 1
Explanation: The 5th node from 1st is 5 and
5th node from last is 1, so swap them.
```

Illustration:



ORIGINAL LIST K = 2



NEW LIST

Approach
: The idea is very simple find the k th node from the start and the kth node from last is n-k+1 th node from start. Swap both the nodes.
However there are some corner cases, which must be handled

1. Y is next to X
2. X is next to Y
3. X and Y are same
4. X and Y don't exist (k is more than number of nodes in linked list)

Below is the implementation of the above approach.

```cpp
// A C++ program to swap Kth node
// from beginning with kth node from end
#include <bits/stdc++.h>
using namespace std;

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

/* Utility function to insert
a node at the beginning */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node
        = (struct Node*)malloc(
            sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Utility function for displaying linked list */
void printList(struct Node* node)
{
    while (node != NULL) {
        cout << node->data << " ";
        node = node->next;
    }
    cout << endl;
}

/* Utility function for calculating
length of linked list */
int countNodes(struct Node* s)
{
    int count = 0;
    while (s != NULL) {
        count++;
        s = s->next;
    }
    return count;
}

```

```
/* Function for swapping kth nodes
from both ends of linked list */
void swapKth(struct Node** head_ref, int k)
{
    // Count nodes in linked list
    int n = countNodes(*head_ref);

    // Check if k is valid
    if (n < k)
        return;

    // If x (kth node from start) and
    // y(kth node from end) are same
    if (2 * k - 1 == n)
        return;

    // Find the kth node from the beginning of
    // the linked list. We also find
    // previous of kth node because we
    // need to update next pointer of
    // the previous.
    Node* x = *head_ref;
    Node* x_prev = NULL;
    for (int i = 1; i < k; i++) {
        x_prev = x;
        x = x->next;
    }

    // Similarly, find the kth node from
    // end and its previous. kth node
    // from end is (n-k+1)th node from beginning
    Node* y = *head_ref;
    Node* y_prev = NULL;
    for (int i = 1; i < n - k + 1; i++) {
        y_prev = y;
        y = y->next;
    }

    // If x_prev exists, then new next of
    // it will be y. Consider the case
    // when y->next is x, in this case,
    // x_prev and y are same. So the statement
    // "x_prev->next = y" creates a self loop.
    // This self loop will be broken
    // when we change y->next.
    if (x_prev)
```

```
        x_prev->next = y;

    // Same thing applies to y_prev
    if (y_prev)
        y_prev->next = x;

    // Swap next pointers of x and y.
    // These statements also break self
    // loop if x->next is y or y->next is x
    Node* temp = x->next;
    x->next = y->next;
    y->next = temp;

    // Change head pointers when k is 1 or n
    if (k == 1)
        *head_ref = y;
    if (k == n)
        *head_ref = x;
}

// Driver program to test above functions
int main()
{
    // Let us create the following
    // linked list for testing
    // 1->2->3->4->5->6->7->8
    struct Node* head = NULL;
    for (int i = 8; i >= 1; i--)
        push(&head, i);

    cout << "Original Linked List: ";
    printList(head);

    for (int k = 1; k < 9; k++) {
        swapKth(&head, k);
        cout << "\nModified List for k = " << k << endl;
        printList(head);
    }

    return 0;
}
```

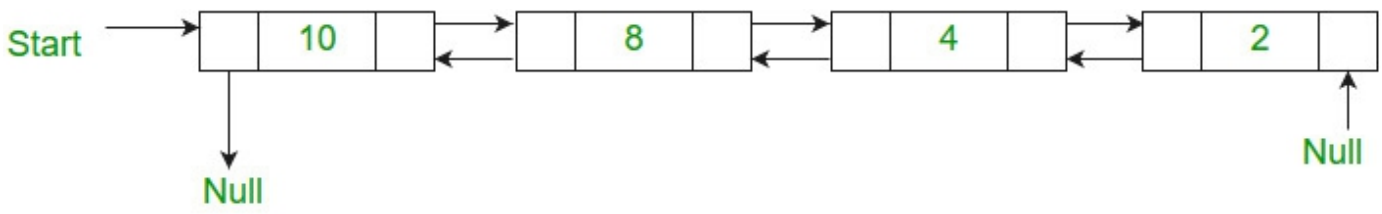
Output:

```
Original Linked List: 1 2 3 4 5 6 7 8

Modified List for k = 1
8 2 3 4 5 6 7 1

Modified List for k = 2
8 7 3 4 5 6 2 1

Modified List for k = 3
8 7 6 4 5 3 2 1

Modified List for k = 4
8 7 6 5 4 3 2 1

Modified List for k = 5
8 7 6 4 5 3 2 1

Modified List for k = 6
8 7 3 4 5 6 2 1

Modified List for k = 7
8 2 3 4 5 6 7 1

Modified List for k = 8
1 2 3 4 5 6 7 8
```

Complexity Analysis:

- **Time Complexity:** O(n), where n is the length of the list.
  One traversal of the list is needed.
- **Auxiliary Space:** O(1).
  No extra space is required.

Merge Sort for Doubly Linked List



Below is the implementation of merge sort for doubly linked list.

```cpp
// C++ program for merge sort on doubly linked list
#include <bits/stdc++.h>
using namespace std;
class Node
{
    public:
    int data;
    Node *next, *prev;
};

Node *split(Node *head);

// Function to merge two linked lists
Node *merge(Node *first, Node *second)
{
    // If first linked list is empty
    if (!first)
        return second;

    // If second linked list is empty
    if (!second)
        return first;

    // Pick the smaller value
    if (first->data < second->data)
    {
        first->next = merge(first->next,second);
        first->next->prev = first;
        first->prev = NULL;
        return first;
    }
    else
    {
        second->next = merge(first,second->next);
        second->next->prev = second;
        second->prev = NULL;
        return second;
    }
}
```

```
40
41  // Function to do merge sort
42  Node *mergeSort(Node *head)
43  {
44      if (!head || !head->next)
45          return head;
46      Node *second = split(head);
47
48      // Recur for left and right halves
49      head = mergeSort(head);
50      second = mergeSort(second);
51
52      // Merge the two sorted halves
53      return merge(head,second);
54  }
55
56  // A utility function to insert a new node at the
57  // beginning of doubly linked list
58  void insert(Node **head, int data)
59  {
60      Node *temp = new Node();
61      temp->data = data;
62      temp->next = temp->prev = NULL;
63      if (!(*head))
64          (*head) = temp;
65      else
66      {
67          temp->next = *head;
68          (*head)->prev = temp;
69          (*head) = temp;
70      }
71  }
72
73  // A utility function to print a doubly linked list in
74  // both forward and backward directions
75  void print(Node *head)
76  {
77      Node *temp = head;
78      cout<<"Forward Traversal using next poitner\n";
79      while (head)
80      {
81          cout << head->data << " ";
82          temp = head;
83          head = head->next;
84      }
85      cout << "\nBackward Traversal using prev pointer\n";
86      while (temp)
87      {
88          cout << temp->data << " ";
89          temp = temp->prev;
90      }
91  }
92
```

```
// Utility function to swap two integers
void swap(int *A, int *B)
{
    int temp = *A;
    *A = *B;
    *B = temp;
}

// Split a doubly linked list (DLL) into 2 DLLs of
// half sizes
Node *split(Node *head)
{
    Node *fast = head,*slow = head;
    while (fast->next && fast->next->next)
    {
        fast = fast->next->next;
        slow = slow->next;
    }
    Node *temp = slow->next;
    slow->next = NULL;
    return temp;
}

// Driver program
int main(void)
{
    Node *head = NULL;
    insert(&head, 5);
    insert(&head, 20);
    insert(&head, 4);
    insert(&head, 3);
    insert(&head, 30);
    insert(&head, 10);
    head = mergeSort(head);
    cout << "Linked List after sorting\n";
    print(head);
    return 0;
}

```

Output

```
Linked List after sorting
Forward Traversal using next pointer
3 4 5 10 20 30
Backward Traversal using prev pointer
30 20 10 5 4 3
```

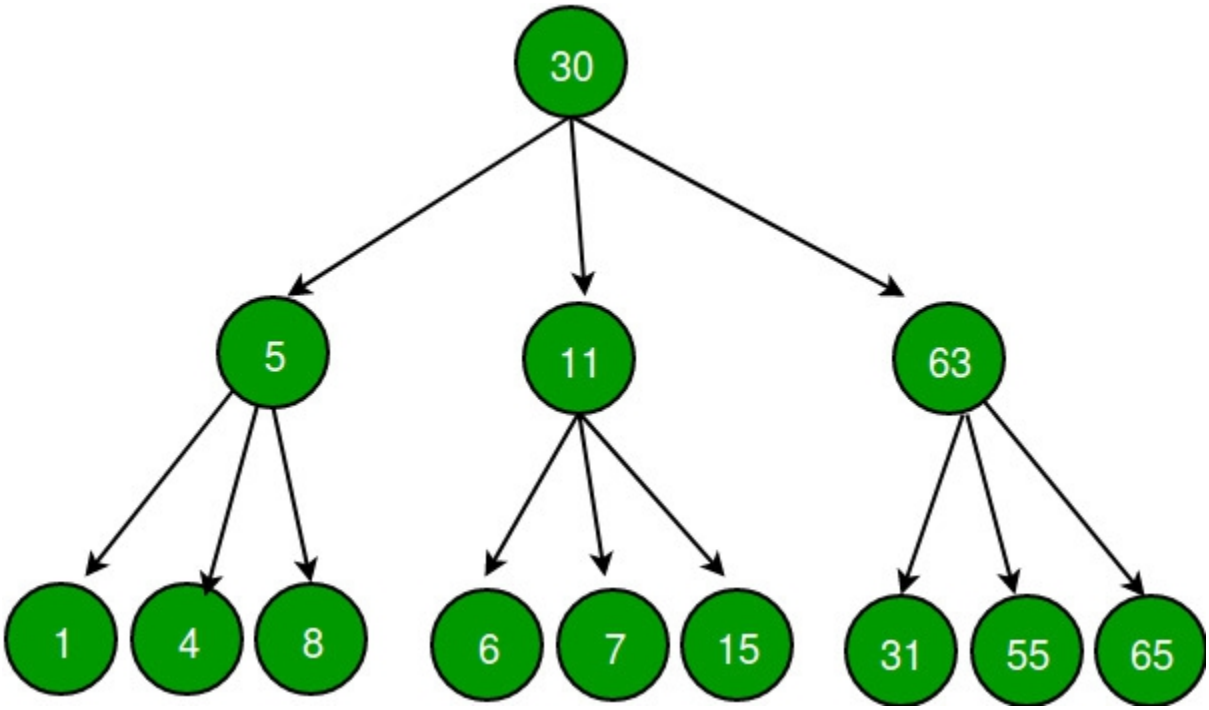
Create a Doubly Linked List from a Ternary Tree

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 –

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 <- 30 <-> 5 <-> 1 <-> 4 <-> 8 <-> 11 <-> 6 <-> 7 <-> 15 <-> 63 <-> 31 <-> 55 <-> 65 -> NULL



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.

```cpp
// 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)
{
    // initialize 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;
}

```

```c
/* 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
```

Find pairs with given sum in doubly linked list

Given a sorted doubly linked list of positive distinct elements, the task is to find pairs in a doubly-linked list whose sum is equal to given value x, without using any extra space?

Example:

```
Input : head : 1 <-> 2 <-> 4 <-> 5 <-> 6 <-> 8 <-> 9
        x = 7
Output: (6, 1), (5,2)
```

The expected time complexity is O(n) and auxiliary space is O(1).

- A **simple approach** for this problem is to one by one pick each node and find a second element whose sum is equal to x in the remaining list by traversing in the forward direction. The time complexity for this problem will be O(n^2), n is the total number of nodes in the doubly linked list. Initialize two pointer variables to find the candidate elements in the sorted doubly linked list. Initialize **first** with the start of the doubly linked list i.e; **first=head** and initialize **second** with the last node of the doubly linked list i.e; **second=last_node**.
- We initialize **first** and **second** pointers as first and last nodes. Here we don't have random access, so to find the second pointer, we traverse the list to initialize the second.
- If current sum of **first** and **second** is less than x, then we move **first** in forward direction. If current sum of **first** and **second** element is greater than x, then we move **second** in backward direction.
- Loop termination conditions are also different from arrays. The loop terminates when two pointers cross each other (second->next = first), or they become the same (first == second).
- The case when no pairs are present will be handled by the condition "first==second"

```cpp
// C++ program to find a pair with given sum x.
#include<bits/stdc++.h>
using namespace std;

// structure of node of doubly linked list
struct Node
{
    int data;
    struct Node *next, *prev;
};

// Function to find pair whose sum equal to given value x.
void pairSum(struct Node *head, int x)
{
    // Set two pointers, first to the beginning of DLL
    // and second to the end of DLL.
    struct Node *first = head;
    struct Node *second = head;
    while (second->next != NULL)
        second = second->next;

    // To track if we find a pair or not
    bool found = false;

    // The loop terminates when two pointers
    // cross each other (second->next
    // == first), or they become same (first == second)
    while (first != second && second->next != first)
    {
```

```
        if ((first->data + second->data) == x)
        {
            found = true;
            cout << "(" << first->data<< ", "
                << second->data << ")" << endl;

            // move first in forward direction
            first = first->next;

            // move second in backward direction
            second = second->prev;
        }
        else
        {
            if ((first->data + second->data) < x)
                first = first->next;
            else
                second = second->prev;
        }
    }

    // if pair is not present
    if (found == false)
        cout << "No pair found";
}

// A utility function to insert a new node at the
// beginning of doubly linked list
void insert(struct Node **head, int data)
{
    struct Node *temp = new Node;
    temp->data = data;
    temp->next = temp->prev = NULL;
    if (!(*head))
        (*head) = temp;
    else
    {
        temp->next = *head;
        (*head)->prev = temp;
        (*head) = temp;
    }
}

// Driver program
int main()
```

```
{
    struct Node *head = NULL;
    insert(&head, 9);
    insert(&head, 8);
    insert(&head, 6);
    insert(&head, 5);
    insert(&head, 4);
    insert(&head, 2);
    insert(&head, 1);
    int x = 7;

    pairSum(head, x);

    return 0;
}
```

Output:

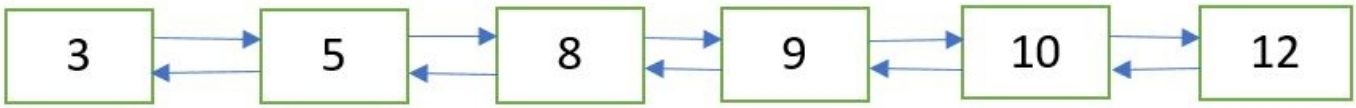
```
(1,6)
(2,5)
```

Time complexity : O(n)
Auxiliary space : O(1)
If linked list is not sorted, then we can sort the list as a first step. But in that case overall time complexity would become O(n Log n).

Insert value in sorted way in a sorted doubly linked list

Given a sorted doubly linked list and a value to insert, write a function to insert the value in sorted way.
Initial doubly linked list

Doubly Linked List after insertion of 9



Algorithm:
Let input doubly linked list is sorted in increasing order.
New node passed to the function contains data in the data part and previous and next link are set to NULL

```
sortedInsert(head_ref, newNode)
      if (head_ref == NULL)
      head_ref = newNode

      else if head_ref->data >= newNode->data
          newNode->next = head_ref
      newNode->next->prev = newNode
      head_ref = newNode

      else
      Initialize current = head_ref
      while (current->next != NULL and
                current->next->data < newNode->data)
        current = current->next

      newNode->next = current->next
      if current->next != NULL
        newNode->next->prev = newNode

          current->next = newNode
      newNode->prev = current
```

```cpp
// C++ implementation to insert value in sorted way
// in a sorted doubly linked list
#include <bits/stdc++.h>

using namespace std;

// Node of a doubly linked list
struct Node {
    int data;
    struct Node* prev, *next;
};

```

```
// function to create and return a new node
// of a doubly linked list
struct Node* getNode(int data)
{
    // allocate node
    struct Node* newNode =
        (struct Node*)malloc(sizeof(struct Node));

    // put in the data
    newNode->data = data;
    newNode->prev = newNode->next = NULL;
    return newNode;
}

// function to insert a new node in sorted way in
// a sorted doubly linked list
void sortedInsert(struct Node** head_ref, struct Node* newNode)
{
    struct Node* current;

    // if list is empty
    if (*head_ref == NULL)
        *head_ref = newNode;

    // if the node is to be inserted at the beginning
    // of the doubly linked list
    else if ((*head_ref)->data >= newNode->data) {
        newNode->next = *head_ref;
        newNode->next->prev = newNode;
        *head_ref = newNode;
    }

    else {
        current = *head_ref;

        // locate the node after which the new node
        // is to be inserted
        while (current->next != NULL &&
            current->next->data < newNode->data)
            current = current->next;

        /* Make the appropriate links */
        newNode->next = current->next;

        // if the new node is not inserted
        // at the end of the list
        if (current->next != NULL)
            newNode->next->prev = newNode;

         current->next = newNode;
         newNode->prev = current;
     }
}

```

```
67  // function to print the doubly linked list
68  void printList(struct Node* head)
69  {
70      while (head != NULL) {
71          cout << head->data << " ";
72          head = head->next;
73      }
74  }
75
76  // Driver program to test above
77  int main()
78  {
79      /* start with the empty doubly linked list */
80      struct Node* head = NULL;
81
82      // insert the following nodes in sorted way
83      struct Node* new_node = getNode(8);
84      sortedInsert(&head, new_node);
85      new_node = getNode(5);
86      sortedInsert(&head, new_node);
87      new_node = getNode(3);
88      sortedInsert(&head, new_node);
89      new_node = getNode(10);
90      sortedInsert(&head, new_node);
91      new_node = getNode(12);
92      sortedInsert(&head, new_node);
93      new_node = getNode(9);
94      sortedInsert(&head, new_node);
95
96      cout << "Created Doubly Linked Listn";
97      printList(head);
98      return 0;
99  }
100
```

Output:

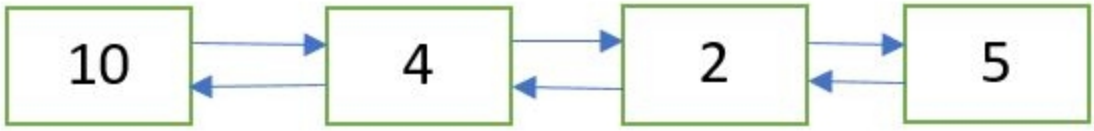
```
Created Doubly Linked List
3 5 8 9 10 12
```

Time Complexity: O(n)

Delete a Doubly Linked List node at a given position

Given a doubly linked list and a position **n**. The task is to delete the node at the given position **n** from the beginning.
Initial doubly linked list

Doubly Linked List after deletion of node at position **n** = 2



Approach: Following are the steps:

1. Get the pointer to the node at position **n** by traversing the doubly linked list up to the **nth** node from the beginning.
2. Delete the node using the pointer obtained in Step 1.

```cpp
/* C++ implementation to delete a doubly Linked List node
at the given position */
#include <bits/stdc++.h>

using namespace std;

/* a node of the doubly linked list */
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

/* Function to delete a node in a Doubly Linked List.
head_ref --> pointer to head node pointer.
del --> pointer to node to be deleted. */
void deleteNode(struct Node** head_ref, struct Node* del)
{
```

```
    /* base case */
    if (*head_ref == NULL || del == NULL)
        return;

    /* If node to be deleted is head node */
    if (*head_ref == del)
        *head_ref = del->next;

    /* Change next only if node to be deleted is NOT
    the last node */
    if (del->next != NULL)
        del->next->prev = del->prev;

    /* Change prev only if node to be deleted is NOT
    the first node */
    if (del->prev != NULL)
        del->prev->next = del->next;

    /* Finally, free the memory occupied by del*/
    free(del);
}

/* Function to delete the node at the given position
in the doubly linked list */
void deleteNodeAtGivenPos(struct Node** head_ref, int n)
{
    /* if list in NULL or invalid position is given */
    if (*head_ref == NULL || n <= 0)
        return;

    struct Node* current = *head_ref;
    int i;

    /* traverse up to the node at position 'n' from
    the beginning */
    for (int i = 1; current != NULL && i < n; i++)
        current = current->next;

    /* if 'n' is greater than the number of nodes
    in the doubly linked list */
    if (current == NULL)
        return;

    /* delete the node pointed to by 'current' */
    deleteNode(head_ref, current);
}

/* Function to insert a node at the beginning
of the Doubly Linked 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;

    /* since we are adding at the beginning,
    prev is always NULL */
    new_node->prev = NULL;

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

    /* change prev of head node to new node */
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

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

/* Function to print nodes in a given doubly
linked list */
void printList(struct Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Create the doubly linked list 10<->8<->4<->2<->5 */
    push(&head, 5);
    push(&head, 2);
    push(&head, 4);
    push(&head, 8);
    push(&head, 10);

    cout << "Doubly linked list before deletion:n";
    printList(head);

    int n = 2;

    /* delete node at the given position 'n' */
    deleteNodeAtGivenPos(&head, n);

    cout << "\nDoubly linked list after deletion:n";
    printList(head);

    return 0;
}
```

Output:

```
Doubly linked list before deletion:
10 8 4 2 5
Doubly linked list after deletion:
10 4 2 5
```

Count triplets in a sorted doubly linked list whose sum is equal to a given value x

Given a sorted doubly linked list of distinct nodes(no two nodes have the same data) and a value **x**. Count triplets in the list that sum up to a given value **x**.
Examples:

```
Input : DLL:  1 <-> 2 <-> 4 <-> 5 <-> 6 <-> 8 <-> 9
        x = 17
Output : 2
The triplets are:
(2, 6, 9) and (4, 5, 8)

Input : DLL:  1 <-> 2 <-> 4 <-> 5 <-> 6 <-> 8 <-> 9
        x = 15
Output : 5
```

Efficient Approach(Use of two pointers):

Traverse the doubly linked list from left to right. For each **current** node during the traversal, initialize two pointers **first** = pointer to the node next to the **current** node and **last** = pointer to the last node of the list. Now, count pairs in the list from **first** to **last** pointer that sum up to value **(x – current node's data)** (algorithm described in post). Add this count to the **total_count** of triplets. Pointer to the **last** node can be found only once in the beginning.

```cpp
// C++ implementation to count triplets in a sorted doubly linked
// whose sum is equal to a given value 'x'
#include <bits/stdc++.h>

using namespace std;

// structure of node of doubly linked list
struct Node {
    int data;
    struct Node* next, *prev;
};

// function to count pairs whose sum equal to given 'value'
int countPairs(struct Node* first, struct Node* second, int value)
{
    int count = 0;

    // The loop terminates when either of two pointers
    // become NULL, or they cross each other (second->next
    // == first), or they become same (first == second)
    while (first != NULL && second != NULL &&
        first != second && second->next != first) {

        // pair found
        if ((first->data + second->data) == value) {

            // increment count
            count++;

            // move first in forward direction
            first = first->next;

            // move second in backward direction
            second = second->prev;
        }

        // if sum is greater than 'value'
        // move second in backward direction
        else if ((first->data + second->data) > value)
            second = second->prev;

        // else move first in forward direction
        else
            first = first->next;
    }

    // required count of pairs
    return count;
}

// function to count triplets in a sorted doubly linked list
// whose sum is equal to a given value 'x'
int countTriplets(struct Node* head, int x)
{
```

```
    // if list is empty
    if (head == NULL)
        return 0;

    struct Node* current, *first, *last;
    int count = 0;

    // get pointer to the last node of
    // the doubly linked list
    last = head;
    while (last->next != NULL)
        last = last->next;

    // traversing the doubly linked list
    for (current = head; current != NULL; current = current->next) {

        // for each current node
        first = current->next;

        // count pairs with sum(x - current->data) in the range
        // first to last and add it to the 'count' of triplets
        count += countPairs(first, last, x - current->data);
    }

    // required count of triplets
    return count;
}

// A utility function to insert a new node at the
// beginning of doubly linked list
void insert(struct Node** head, int data)
{
    // allocate node
    struct Node* temp = new Node();

    // put in the data
    temp->data = data;
    temp->next = temp->prev = NULL;

    if ((*head) == NULL)
        (*head) = temp;
    else {
        temp->next = *head;
        (*head)->prev = temp;
        (*head) = temp;
    }
}

// Driver program to test above
int main()
{
    // start with an empty doubly linked list
    struct Node* head = NULL;

    // insert values in sorted order
```

```
110     insert(&head, 9);
111     insert(&head, 8);
112     insert(&head, 6);
113     insert(&head, 5);
114     insert(&head, 4);
115     insert(&head, 2);
116     insert(&head, 1);
117
118     int x = 17;
119
120     cout << "Count = "
121         << countTriplets(head, x);
122     return 0;
123 }
124
```

Output:

```
Count = 2
```

Time Complexity: $O(n^2)$

Auxiliary Space: $O(1)$

Remove duplicates from a sorted doubly linked list

Given a sorted doubly linked list containing **n** nodes. The problem is removing duplicate nodes from the given list.

Examples:

```
Input : DLL: 4<->4<->4<->4<->6<->8<->8<->10<->12<->12
Output : 4<->6<->8<->10<->12
```

Algorithm:

```
removeDuplicates(head_ref, x)
      if head_ref == NULL
          return
      Initialize current = head_ref
      while current->next != NULL
           if current->data == current->next->data
               deleteNode(head_ref, current->next)
           else
               current = current->next
```

```cpp
/* C++ implementation to remove duplicates from a
sorted doubly linked list */
#include <bits/stdc++.h>

using namespace std;

/* a node of the doubly linked list */
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

/* Function to delete a node in a Doubly Linked List.
head_ref --> pointer to head node pointer.
del --> pointer to node to be deleted. */
void deleteNode(struct Node** head_ref, struct Node* del)
{
    /* base case */
    if (*head_ref == NULL || del == NULL)
        return;

    /* If node to be deleted is head node */
    if (*head_ref == del)
        *head_ref = del->next;

    /* Change next only if node to be deleted
        is NOT the last node */
    if (del->next != NULL)
        del->next->prev = del->prev;

    /* Change prev only if node to be deleted
    is NOT the first node */
    if (del->prev != NULL)
        del->prev->next = del->next;

    /* Finally, free the memory occupied by del*/
    free(del);
}

```

```c
/* function to remove duplicates from a
sorted doubly linked list */
void removeDuplicates(struct Node** head_ref)
{
    /* if list is empty */
    if ((*head_ref) == NULL)
        return;

    struct Node* current = *head_ref;
    struct Node* next;

    /* traverse the list till the last node */
    while (current->next != NULL) {

        /* Compare current node with next node */
        if (current->data == current->next->data)

            /* delete the node pointed to by
            'current->next' */
            deleteNode(head_ref, current->next);

        /* else simply move to the next node */
        else
            current = current->next;
    }
}

/* Function to insert a node at the beginning
of the Doubly Linked 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;

    /* since we are adding at the beginning,
    prev is always NULL */
    new_node->prev = NULL;

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

    /* change prev of head node to new node */
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

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

```
94  /* Function to print nodes in a given doubly linked list */
95  void printList(struct Node* head)
96  {
97      /* if list is empty */
98      if (head == NULL)
99          cout << "Doubly Linked list empty";
100
101     while (head != NULL) {
102         cout << head->data << " ";
103         head = head->next;
104     }
105 }
106
107 /* Driver program to test above functions*/
108 int main()
109 {
110     /* Start with the empty list */
111     struct Node* head = NULL;
112
113     /* Create the doubly linked list:
114     4<->4<->4<->4<->6<->8<->8<->10<->12<->12 */
115     push(&head, 12);
116     push(&head, 12);
117     push(&head, 10);
118     push(&head, 8);
119     push(&head, 8);
120     push(&head, 6);
121     push(&head, 4);
122     push(&head, 4);
123     push(&head, 4);
124     push(&head, 4);
125
126     cout << "Original Doubly linked list:n";
127     printList(head);
128
129     /* remove duplicate nodes */
130     removeDuplicates(&head);
131
132     cout << "\nDoubly linked list after"
133             " removing duplicates:n";
134     printList(head);
135
136     return 0;
137 }
138
```

Output:

```
Original Doubly linked list:
4 4 4 4 6 8 8 10 12 12
Doubly linked list after removing duplicates:
4 6 8 10 12
```

Time Complexity: O(n)

Delete all occurrences of a given key in a doubly linked list

Given a doubly linked list and a key **x**. The problem is to delete all occurrences of the given key **x** from the doubly linked list.
Examples:

```
Input : DLL: 2 <-> 2 <-> 10 <-> 8 <-> 4 <-> 2 <-> 5 <-> 2
        x = 2
Output : 10 <-> 8 <-> 4 <-> 5
```

Algorithm:

```
delAllOccurOfGivenKey(head_ref, x)
      if head_ref == NULL
          return
      Initialize current = head_ref
      Declare next
      while current != NULL
           if current->data == x
               next = current->next
               deleteNode(head_ref, current)
               current = next
           else
               current = current->next
```

Time Complexity: O(n)

```cpp
/* C++ implementation to delete all occurrences
of a given key in a doubly linked list */
#include <bits/stdc++.h>

using namespace std;

/* a node of the doubly linked list */
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

/* Function to delete a node in a Doubly Linked List.
head_ref --> pointer to head node pointer.
del --> pointer to node to be deleted. */
void deleteNode(struct Node** head_ref, struct Node* del)
{
    /* base case */
    if (*head_ref == NULL || del == NULL)
        return;

    /* If node to be deleted is head node */
    if (*head_ref == del)
        *head_ref = del->next;

    /* Change next only if node to be deleted
    is NOT the last node */
    if (del->next != NULL)
        del->next->prev = del->prev;

    /* Change prev only if node to be deleted
    is NOT the first node */
    if (del->prev != NULL)
        del->prev->next = del->next;

    /* Finally, free the memory occupied by del*/
    free(del);
}

/* function to delete all occurrences of the given
    key 'x' */
void deleteAllOccurOfX(struct Node** head_ref, int x)
{
    /* if list is empty */
    if ((*head_ref) == NULL)
        return;

    struct Node* current = *head_ref;
    struct Node* next;

    /* traverse the list up to the end */
    while (current != NULL) {

        /* if node found with the value 'x' */
        if (current->data == x) {

```

```
            /* save current's next node in the
            pointer 'next' */
            next = current->next;

            /* delete the node pointed to by
            'current' */
            deleteNode(head_ref, current);

            /* update current */
            current = next;
        }

        /* else simply move to the next node */
        else
            current = current->next;
    }
}

/* Function to insert a node at the beginning
of the Doubly Linked 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;

    /* since we are adding at the beginning,
    prev is always NULL */
    new_node->prev = NULL;

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

    /* change prev of head node to new node */
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

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

/* Function to print nodes in a given doubly
linked list */
void printList(struct Node* head)
{
    /* if list is empty */
    if (head == NULL)
        cout << "Doubly Linked list empty";

    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}
```

```
115
116 /* Driver program to test above functions*/
117 int main()
118 {
119     /* Start with the empty list */
120     struct Node* head = NULL;
121
122     /* Create the doubly linked list:
123         2<->2<->10<->8<->4<->2<->5<->2 */
124     push(&head, 2);
125     push(&head, 5);
126     push(&head, 2);
127     push(&head, 4);
128     push(&head, 8);
129     push(&head, 10);
130     push(&head, 2);
131     push(&head, 2);
132
133     cout << "Original Doubly linked list:n";
134     printList(head);
135
136     int x = 2;
137
138     /* delete all occurrences of 'x' */
139     deleteAllOccurOfX(&head, x);
140
141     cout << "\nDoubly linked list after deletion of "
142         << x << ":n";
143     printList(head);
144
145     return 0;
146 }
147
```

Output:

```
Original Doubly linked list:
2 2 10 8 4 2 5 2
Doubly linked list after deletion of 2:
10 8 4 5
```

Remove duplicates from an unsorted doubly linked list

Given an unsorted doubly linked list containing **n** nodes. The problem is to remove duplicate nodes from the given list.

```
Input : DLL: 8<->4<->4<->6<->4<->8<->4<->10<->12<->12
Output : 8<->4<->6<->10<->12
```

```cpp
// C++ implementation to remove duplicates from an
// unsorted doubly linked list
#include <bits/stdc++.h>

using namespace std;

// a node of the doubly linked list
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

// Function to delete a node in a Doubly Linked List.
// head_ref --> pointer to head node pointer.
// del --> pointer to node to be deleted.
void deleteNode(struct Node** head_ref, struct Node* del)
{
    // base case
    if (*head_ref == NULL || del == NULL)
        return;

    // If node to be deleted is head node
    if (*head_ref == del)
        *head_ref = del->next;

    // Change next only if node to be deleted
    // is NOT the last node
    if (del->next != NULL)
        del->next->prev = del->prev;

    // Change prev only if node to be deleted
    // is NOT the first node
    if (del->prev != NULL)
        del->prev->next = del->next;

    // Finally, free the memory occupied by del
    free(del);
}
```

```
40
41  // function to remove duplicates from
42  // an unsorted doubly linked list
43  void removeDuplicates(struct Node** head_ref)
44  {
45      // if DLL is empty or if it contains only
46      // a single node
47      if ((*head_ref) == NULL ||
48          (*head_ref)->next == NULL)
49          return;
50
51      struct Node* ptr1, *ptr2;
52
53      // pick elements one by one
54      for (ptr1 = *head_ref; ptr1 != NULL; ptr1 = ptr1->next) {
55          ptr2 = ptr1->next;
56
57          // Compare the picked element with the
58          // rest of the elements
59          while (ptr2 != NULL) {
60
61              // if duplicate, then delete it
62              if (ptr1->data == ptr2->data) {
63
64                  // store pointer to the node next to 'ptr2'
65                  struct Node* next = ptr2->next;
66
67                  // delete node pointed to by 'ptr2'
68                  deleteNode(head_ref, ptr2);
69
70                  // update 'ptr2'
71                  ptr2 = next;
72              }
73
74              // else simply move to the next node
75              else
76                  ptr2 = ptr2->next;
77          }
78      }
79  }
80
81  // Function to insert a node at the beginning
82  // of the Doubly Linked List
83  void push(struct Node** head_ref, int new_data)
84  {
85      // allocate node
86      struct Node* new_node =
87          (struct Node*)malloc(sizeof(struct Node));
88
89      // put in the data
90      new_node->data = new_data;
91
92      // since we are adding at the beginning,
93      // prev is always NULL
94      new_node->prev = NULL;
95
```

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

    // change prev of head node to new node
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

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

// Function to print nodes in a given doubly
// linked list
void printList(struct Node* head)
{
    // if list is empty
    if (head == NULL)
        cout << "Doubly Linked list empty";

    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Driver program to test above
int main()
{
    struct Node* head = NULL;

    // Create the doubly linked list:
    // 8<->4<->4<->6<->4<->8<->4<->10<->12<->12
    push(&head, 12);
    push(&head, 12);
    push(&head, 10);
    push(&head, 4);
    push(&head, 8);
    push(&head, 4);
    push(&head, 6);
    push(&head, 4);
    push(&head, 4);
    push(&head, 8);

    cout << "Original Doubly linked list:n";
    printList(head);

    /* remove duplicate nodes */
    removeDuplicates(&head);

    cout << "\nDoubly linked list after "
            "removing duplicates:n";
    printList(head);

    return 0;
}
```

Output

```
Original Doubly linked list:
8 4 4 6 4 8 4 10 12 12
Doubly linked list after removing duplicates:
8 4 6 10 12
```

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

Sort the biotonic doubly linked list

Sort the given biotonic doubly linked list. A biotonic doubly linked list is a doubly linked list which is first increasing and then decreasing. A strictly increasing or a strictly decreasing list is also a biotonic doubly linked list.

```
Input : DLL: 2<->5<->7<->12<->10<->6<->4<->1
Output : 1<->2<->4<->5<->6<->7<->10<->12

Input : DLL: 20<->17<->14<->8<->3
Output : 3<->8<->14<->17<->20
```

Approach: Find the first node in the list which is smaller than its previous node. Let it be **current**. If no such node is present then list is already sorted. Else split the list into two lists, **first** starting from **head** node till the **current's** previous node and **second** starting from **current** node till the end of the list. Reverse the **second** doubly linked list. Now merge the **first** and **second** sorted doubly linked list.. The final merged list is the required sorted doubly linked list.

```cpp
// C++ implementation to sort the biotonic doubly linked list
#include <bits/stdc++.h>

using namespace std;

// a node of the doubly linked list
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

// Function to reverse a Doubly Linked List
void reverse(struct Node** head_ref)
{
```

```
    struct Node* temp = NULL;
    struct Node* current = *head_ref;

    // swap next and prev for all nodes
    // of doubly linked list
    while (current != NULL) {
        temp = current->prev;
        current->prev = current->next;
        current->next = temp;
        current = current->prev;
    }

    // Before changing head, check for the cases
    // like empty list and list with only one node
    if (temp != NULL)
        *head_ref = temp->prev;
}

// Function to merge two sorted doubly linked lists
struct Node* merge(struct Node* first, struct Node* second)
{
    // If first linked list is empty
    if (!first)
        return second;

    // If second linked list is empty
    if (!second)
        return first;

    // Pick the smaller value
    if (first->data < second->data) {
        first->next = merge(first->next, second);
        first->next->prev = first;
        first->prev = NULL;
        return first;
    } else {
        second->next = merge(first, second->next);
        second->next->prev = second;
        second->prev = NULL;
        return second;
    }
}

// function to sort a biotonic doubly linked list
struct Node* sort(struct Node* head)
{
    // if list is empty or if it contains a single
    // node only
    if (head == NULL || head->next == NULL)
        return head;

    struct Node* current = head->next;

    while (current != NULL) {

```

```
        // if true, then 'current' is the first node
        // which is smaller than its previous node
        if (current->data < current->prev->data)
            break;

        // move to the next node
        current = current->next;
    }

    // if true, then list is already sorted
    if (current == NULL)
        return head;

    // spilt into two lists, one starting with 'head'
    // and other starting with 'current'
    current->prev->next = NULL;
    current->prev = NULL;

    // reverse the list starting with 'current'
    reverse(&t);

    // merge the two lists and return the
    // final merged doubly linked list
    return merge(head, current);
}

// Function to insert a node at the beginning
// of the Doubly Linked 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;

    // since we are adding at the beginning,
    // prev is always NULL
    new_node->prev = NULL;

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

    // change prev of head node to new node
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

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

// Function to print nodes in a given doubly
// linked list
void printList(struct Node* head)
```

```
{
    // if list is empty
    if (head == NULL)
        cout << "Doubly Linked list empty";

    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Driver program to test above
int main()
{
    struct Node* head = NULL;

    // Create the doubly linked list:
    // 2<->5<->7<->12<->10<->6<->4<->1
    push(&head, 1);
    push(&head, 4);
    push(&head, 6);
    push(&head, 10);
    push(&head, 12);
    push(&head, 7);
    push(&head, 5);
    push(&head, 2);

    cout << "Original Doubly linked list:n";
    printList(head);

    // sort the biotonic DLL
    head = sort(head);

    cout << "\nDoubly linked list after sorting:n";
    printList(head);

    return 0;
}

```

Output

```
Original Doubly linked list:
2 5 7 12 10 6 4 1
Doubly linked list after sorting:
1 2 4 5 6 7 10 12
```

Time Complexity: O(n)

Sort a k sorted doubly linked list

Given a doubly linked list containing **n** nodes, where each node is at most **k** away from its target position in the list. The problem is to sort the given doubly linked list.
For example, let us consider **k** is 2, a node at position 7 in the sorted doubly linked list, can be at positions 5, 6, 7, 8, 9 in the given doubly linked list.
Examples:

```
Input : DLL: 3 <-> 6 <-> 2 <-> 12 <-> 56 <-> 8
        k = 2
Output : 2 <-> 3 <-> 6 <-> 8 <-> 12 <-> 56
```

Naive Approach: Sort the given doubly linked list using the insertion sort technique. While inserting each element in the sorted part of the list, there will be at most k swaps to place the element to its correct position since it is at most k steps away from its correct position.

```cpp
// C++ implementation to sort a k sorted doubly
// linked list
#include<bits/stdc++.h>
using namespace std;

// a node of the doubly linked list
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};


// function to sort a k sorted doubly linked list
struct Node* sortAKSortedDLL(struct Node* head, int k)
{
    if(head == NULL || head->next == NULL)
        return head;

    // perform on all the nodes in list
    for(Node *i = head->next; i != NULL; i = i->next) {
        Node *j = i;
        // There will be atmost k swaps for each element in the list
        // since each node is k steps away from its correct position
        while(j->prev != NULL && j->data < j->prev->data) {
            // swap j and j.prev node
            Node* temp = j->prev->prev;
            Node* temp2 = j->prev;
            Node *temp3 = j->next;
            j->prev->next = temp3;
            j->prev->prev = j;
            j->prev = temp;
```

```
            j->next = temp2;
            if(temp != NULL)
                temp->next = j;
            if(temp3 != NULL)
                temp3->prev = temp2;
        }
        // if j is now the new head
    // then reset head
        if(j->prev == NULL)
            head = j;
    }
    return head;
}

// Function to insert a node at the beginning
// of the Doubly Linked 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;

    // since we are adding at the beginning,
    // prev is always NULL
    new_node->prev = NULL;

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

    // change prev of head node to new node
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

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

// Function to print nodes in a given doubly linked list
void printList(struct Node* head)
{
    // if list is empty
    if (head == NULL)
        cout << "Doubly Linked list empty";

    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

```

```
// Driver program to test above
int main()
{
    struct Node* head = NULL;

    // Create the doubly linked list:
    // 3<->6<->2<->12<->56<->8
    push(&head, 8);
    push(&head, 56);
    push(&head, 12);
    push(&head, 2);
    push(&head, 6);
    push(&head, 3);

    int k = 2;

    cout << "Original Doubly linked list:\n";
    printList(head);

    // sort the biotonic DLL
    head = sortAKSortedDLL(head, k);

    cout << "\nDoubly linked list after sorting:\n";
    printList(head);

    return 0;
}

// This code is contributed by sachinejain74754.

```

Output

```
Original Doubly linked list:
3 6 2 12 56 8
Doubly Linked List after sorting:
2 3 6 8 12 56
```

Time Complexity: O(n*k)

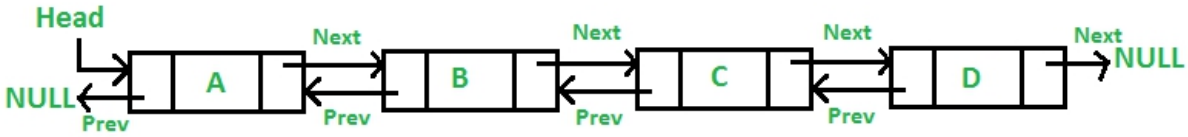
Auxiliary Space: O(1)

Go through below problem..

https://www.geeksforgeeks.org/convert-a-given-binary-tree-to-doubly-linked-list-set-4/

Program to find size of Doubly Linked List

Given a doubly linked list , the task is to find the size of that doubly linked list. For example, size of below linked list is 4.



A doubly linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes.
Traversal of a doubly linked list can be in either direction. In fact, the direction of traversal can change many times, if desired.

Algorithm :

1) Initialize size to 0.

2) Initialize a node pointer, temp = head.

3) Do following while temp is not NULL

.....a) temp = temp -> next

.....b) size++;

4) Return size.

```cpp
// A complete working C++ program to
// find size of doubly linked list.
#include <bits/stdc++.h>
using namespace std;

// A linked list node
struct Node
{
    int data;
    struct Node *next;
    struct Node *prev;
};

/* Function to add a node to front of doubly
linked list */
```

```
16  void push(struct Node** head_ref, int new_data)
17  {
18      struct Node* new_node = new Node;
19      new_node->data = new_data;
20      new_node->next = (*head_ref);
21      new_node->prev = NULL;
22      if ((*head_ref) != NULL)
23      (*head_ref)->prev = new_node ;
24      (*head_ref) = new_node;
25  }
26
27  // This function returns size of linked list
28  int findSize(struct Node *node)
29  {
30  int res = 0;
31  while (node != NULL)
32  {
33      res++;
34      node = node->next;
35  }
36  return res;
37  }
38
39  /* Driver program to test above functions*/
40  int main()
41  {
42      struct Node* head = NULL;
43      push(&head, 4);
44      push(&head, 3);
45      push(&head, 2);
46      push(&head, 1);
47      cout << findSize(head);
48      return 0;
49  }
```

Output:

```
4
```

Sorted insert in a doubly linked list with head and tail pointers

A doubly linked list is a linked list that consists of a set of sequentially linked records called

nodes. Each node contains two fields that are references to the previous and to the next node in the sequence of nodes.
The task is to create a doubly linked list by inserting nodes such that list remains in ascending order on printing from left to right. Also, we need to maintain two pointers, head (points to first node) and tail (points to last node).
Examples:

```
Input : 40 50 10 45 90 100 95
Output :10 40 45 50 90 95 100

Input : 30 10 50 43 56 12
Output :10 12 30 43 50 56
```

Algorithm:
The task can be accomplished as:

1. If Linked list is empty then make both the left and right pointers point to the node to be inserted and make its previous and next field point to NULL.
2. If node to be inserted has value less than the value of first node of linked list then connect that node from previous field of first node.
3. If node to be inserted has value more than the value of last node of linked list then connect that node from next field of last node.
4. If node to be inserted has value in between the value of first and last node, then check for appropriate position and make connections.

```cpp
/* C++ program to insetail nodes in doubly
linked list such that list remains in
ascending order on printing from left
to right */
#include <bits/stdc++.h>
using namespace std;

// A linked list node
class Node
{
    public:
    Node *prev;
    int info;
    Node *next;
};

// Function to insetail new node
void nodeInsetail(Node **head,
                  Node **tail,
                  int key)
{
```

```
22
23      Node *p = new Node();
24      p->info = key;
25      p->next = NULL;
26
27      // If first node to be insetailed in doubly
28      // linked list
29      if ((*head) == NULL)
30      {
31          (*head) = p;
32          (*tail) = p;
33          (*head)->prev = NULL;
34          return;
35      }
36
37      // If node to be insetailed has value less
38      // than first node
39      if ((p->info) < ((*head)->info))
40      {
41          p->prev = NULL;
42          (*head)->prev = p;
43          p->next = (*head);
44          (*head) = p;
45          return;
46      }
47
48      // If node to be insetailed has value more
49      // than last node
50      if ((p->info) > ((*tail)->info))
51      {
52          p->prev = (*tail);
53          (*tail)->next = p;
54          (*tail) = p;
55          return;
56      }
57
58      // Find the node before which we need to
59      // insert p.
60      Node *temp = (*head)->next;
61      while ((temp->info) < (p->info))
62          temp = temp->next;
63
64      // Insert new node before temp
65      (temp->prev)->next = p;
66      p->prev = temp->prev;
67      temp->prev = p;
68      p->next = temp;
69  }
70
71  // Function to print nodes in from left to right
72  void printList(Node *temp)
73  {
74      while (temp != NULL)
75      {
76          cout << temp->info << " ";
77          temp = temp->next;
```

```
78      }
79  }
80
81  // Driver program to test above functions
82  int main()
83  {
84      Node *left = NULL, *right = NULL;
85      nodeInsetail(&left, &right, 30);
86      nodeInsetail(&left, &right, 50);
87      nodeInsetail(&left, &right, 90);
88      nodeInsetail(&left, &right, 10);
89      nodeInsetail(&left, &right, 40);
90      nodeInsetail(&left, &right, 110);
91      nodeInsetail(&left, &right, 60);
92      nodeInsetail(&left, &right, 95);
93      nodeInsetail(&left, &right, 23);
94
95      cout<<"Doubly linked list on printing"
96          " from left to right\n";
97      printList(left);
98
99      return 0;
100 }
101
102 // This is code is contributed by rathbhupendra
103
```

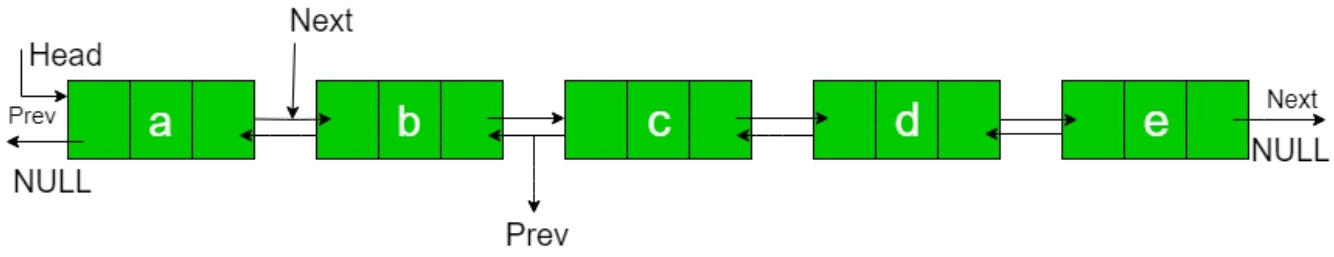
Output

Output:

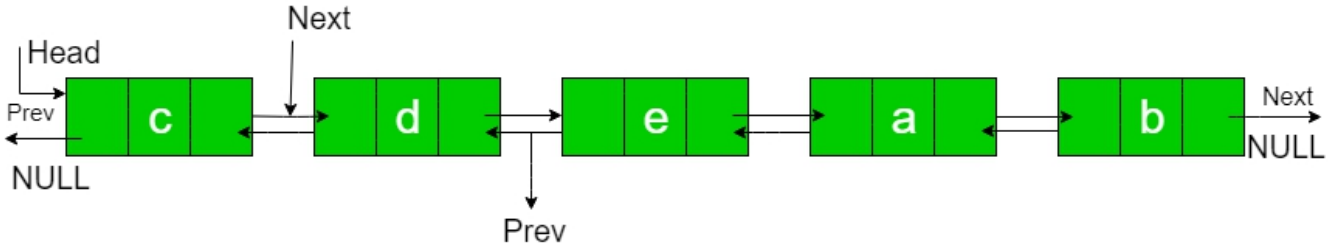
```
Doubly linked list on printing from left to right
10 23 30 40 50 60 90 95 110
```

Rotate Doubly linked list by N

Given a doubly linked list, rotate the linked list counter-clockwise by N nodes. Here N is a given positive integer and is smaller than the count of nodes in linked list.



N = 2
Rotated List:



Examples:

```
Input : a  b  c  d  e   N = 2
Output : c  d  e  a  b

Input : a  b  c  d  e  f  g  h   N = 4
Output : e  f  g  h  a  b  c  d
```

- Asked in amazon

To rotate the Doubly linked list, we need to change next of Nth node to NULL, next of last node to previous head node, and prev of head node to last node and finally change head to (N+1)th node and prev of new head node to NULL (Prev of Head node in doubly linked list is NULL)
So we need to get hold of three nodes: Nth node, (N+1)th node and last node. Traverse the list from beginning and stop at Nth node. Store pointer to Nth node. We can get (N+1)th node using NthNode->next. Keep traversing till end and store pointer to last node also. Finally, change pointers as stated above and at Last Print Rotated List using PrintList Function.

```cpp
// C++ program to rotate a Doubly linked
// list counter clock wise by N times
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node {
    char data;
    struct Node* prev;
    struct Node* next;
};

// This function rotates a doubly linked
// list counter-clockwise and updates the
// head. The function assumes that N is
```

```
// smallerthan size of linked list. It
// doesn't modify the list if N is greater
// than or equal to size
void rotate(struct Node** head_ref, int N)
{
    if (N == 0)
        return;

    // Let us understand the below code
    // for example N = 2 and
    // list = a <-> b <-> c <-> d <-> e.
    struct Node* current = *head_ref;

    // current will either point to Nth
    // or NULL after this loop. Current
    // will point to node 'b' in the
    // above example
    int count = 1;
    while (count < N && current != NULL) {
        current = current->next;
        count++;
    }

    // If current is NULL, N is greater
    // than or equal to count of nodes
    // in linked list. Don't change the
    // list in this case
    if (current == NULL)
        return;

    // current points to Nth node. Store
    // it in a variable. NthNode points to
    // node 'b' in the above example
    struct Node* NthNode = current;

    // current will point to last node
    // after this loop current will point
    // to node 'e' in the above example
    while (current->next != NULL)
        current = current->next;

    // Change next of last node to previous
    // head. Next of 'e' is now changed to
    // node 'a'
    current->next = *head_ref;

    // Change prev of Head node to current
    // Prev of 'a' is now changed to node 'e'
    (*head_ref)->prev = current;

    // Change head to (N+1)th node
    // head is now changed to node 'c'
    *head_ref = NthNode->next;

    // Change prev of New Head node to NULL
    // Because Prev of Head Node in Doubly
```

```
    // linked list is NULL
    (*head_ref)->prev = NULL;

    // change next of Nth node to NULL
    // next of 'b' is now NULL
    NthNode->next = NULL;
}

// Function to insert a node at the
// beginning of the Doubly Linked List
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = new Node;
    new_node->data = new_data;
    new_node->prev = NULL;
    new_node->next = (*head_ref);
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;
*head_ref = new_node;
}

/* Function to print linked list */
void printList(struct Node* node)
{
    while (node->next != NULL) {
        cout << node->data << " "
            << "<=>"
            << " ";
        node = node->next;
    }
    cout << node->data;
}

// Driver's Code
int main(void)
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Let us create the doubly
    linked list a<->b<->c<->d<->e */
    push(&head, 'e');
    push(&head, 'd');
    push(&head, 'c');
    push(&head, 'b');
    push(&head, 'a');

    int N = 2;

    cout << "Given linked list \n";
    printList(head);
    rotate(&head, N);

    cout << "\nRotated Linked list \n";
    printList(head);

    return 0;
}
```

Output:

```
Given linked list
a  b  c  d  e
Rotated Linked list
c  d  e  a  b
```

Priority Queue using Doubly Linked List

Given Nodes with their priority, implement a priority queue using doubly linked list.

- push(): This function is used to insert a new data into the queue.
- pop(): This function removes the element with the lowest priority value form the queue.
- peek() / top(): This function is used to get the lowest priority element in the queue without removing it from the queue.

Approach :
1. Create a doubly linked list having fields info(hold the information of the Node), priority(hold the priority of the Node), prev(point to previous Node), next(point to next Node).
2. Insert the element and priority in the Node.
3. Arrange the Nodes in the increasing order of priority.

```cpp
// C++ code to implement priority
// queue using doubly linked list
#include <bits/stdc++.h>
using namespace std;

// Linked List Node
struct Node {
    int info;
    int priority;
    struct Node *prev, *next;
};

// Function to insert a new Node
void push(Node** fr, Node** rr, int n, int p)
{
    Node* news = (Node*)malloc(sizeof(Node));
    news->info = n;
    news->priority = p;

```

```
    // If linked list is empty
    if (*fr == NULL) {
        *fr = news;
        *rr = news;
        news->next = NULL;
    }
    else {
        // If p is less than or equal front
        // node's priority, then insert at
        // the front.
        if (p <= (*fr)->priority) {
            news->next = *fr;
            (*fr)->prev = news->next;
            *fr = news;
        }

        // If p is more rear node's priority,
        // then insert after the rear.
        else if (p > (*rr)->priority) {
            news->next = NULL;
            (*rr)->next = news;
            news->prev = (*rr)->next;
            *rr = news;
        }

        // Handle other cases
        else {

            // Find position where we need to
            // insert.
            Node* start = (*fr)->next;
            while (start->priority > p)
                start = start->next;
            (start->prev)->next = news;
            news->next = start->prev;
            news->prev = (start->prev)->next;
            start->prev = news->next;
        }
    }
}

// Return the value at rear
int peek(Node* fr) { return fr->info; }

bool isEmpty(Node* fr) { return (fr == NULL); }

// Removes the element with the
// least priority value form the list
int pop(Node** fr, Node** rr)
{
    Node* temp = *fr;
    int res = temp->info;
    (*fr) = (*fr)->next;
    free(temp);
    if (*fr == NULL)
```

```
        *rr = NULL;
    return res;
}

// Driver code
int main()
{
    Node *front = NULL, *rear = NULL;
    push(&front, &rear, 2, 3);
    push(&front, &rear, 3, 4);
    push(&front, &rear, 4, 5);
    push(&front, &rear, 5, 6);
    push(&front, &rear, 6, 7);
    push(&front, &rear, 1, 2);

    cout << pop(&front, &rear) << endl;
    cout << peek(front);

    return 0;
}

```

Output:

```
1
2
```

Reverse a doubly linked list in groups of given size

Given a doubly linked list containing **n** nodes. The problem is to reverse every group of **k** nodes in the list.

```
Input : DLL: 10<->8<->4<->2
        k = 2
Output : 8<->10<->2<->4

Input : 1<->2<->3<->4<->5<->6<->7<->8
        k = 3
Output : 3<->2<->1<->6<->5<->4<->8<->7
```

Approach: Create a recursive function say **reverse(head, k)**. This function receives the head or the first node of each group of **k** nodes After reversing the group of **k** nodes the function checks whether next group of nodes exists in the list or not. If group exists then it makes a recursive call to itself with the first node of the next group and makes the necessary adjustments with the next and previous links of that group. Finally it returns the new head node of the reversed group.

```cpp
// C++ implementation to reverse a doubly linked list
// in groups of given size
#include <bits/stdc++.h>

using namespace std;

// a node of the doubly linked list
struct Node {
    int data;
    Node *next, *prev;
};

// function to get a new node
Node* getNode(int data)
{
    // allocate space
    Node* new_node = (Node*)malloc(sizeof(Node));

    // put in the data
    new_node->data = data;
    new_node->next = new_node->prev = NULL;
    return new_node;
}

// function to insert a node at the beginging
// of the Doubly Linked List
void push(Node** head_ref, Node* new_node)
{
    // since we are adding at the beginning,
    // prev is always NULL
    new_node->prev = NULL;

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

    // change prev of head node to new node
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

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

// function to reverse a doubly linked list
// in groups of given size
Node* revListInGroupOfGivenSize(Node* head, int k)
{
```

```
48      Node *current = head;
49      Node* next = NULL;
50      Node* newHead = NULL;
51      int count = 0;
52
53      // reversing the current group of k
54      // or less than k nodes by adding
55      // them at the beginning of list
56      // 'newHead'
57      while (current != NULL && count < k)
58      {
59          next = current->next;
60          push(&newHead, current);
61          current = next;
62          count++;
63      }
64
65      // if next group exists then making the desired
66      // adjustments in the link
67      if (next != NULL)
68      {
69          head->next = revListInGroupOfGivenSize(next, k);
70          head->next->prev = head;
71      }
72
73      // pointer to the new head of the
74      // reversed group
75      return newHead;
76  }
77
78  // Function to print nodes in a
79  // given doubly linked list
80  void printList(Node* head)
81  {
82      while (head != NULL) {
83          cout << head->data << " ";
84          head = head->next;
85      }
86  }
87
88  // Driver program to test above
89  int main()
90  {
91      // Start with the empty list
92      Node* head = NULL;
93
94      // Create doubly linked: 10<->8<->4<->2
95      push(&head, getNode(2));
96      push(&head, getNode(4));
97      push(&head, getNode(8));
98      push(&head, getNode(10));
99
100     int k = 2;
```

```
101
102     cout << "Original list: ";
103     printList(head);
104
105     // Reverse doubly linked list in groups of
106     // size 'k'
107     head = revListInGroupOfGivenSize(head, k);
108
109     cout << "\nModified list: ";
110     printList(head);
111
112     return 0;
113 }
114
```

Output

```
Original list: 10 8 4 2
Modified list: 8 10 2 4
```

Time Complexity: O(n).

We can further simplify the implementation of this algorithm using the same idea with recursion in just one function.

```
// function to Reverse a doubly linked list
// in groups of given size
Node* reverseByN(Node* head, int k)
{
    if (!head)
        return NULL;
    head->prev = NULL;
    Node *temp, *curr = head, *newHead;
    int count = 0;
    while (curr != NULL && count < k) {
        newHead = curr;
        temp = curr->prev;
        curr->prev = curr->next;
        curr->next = temp;
        curr = curr->prev;
        count++;
    }
    // checking if the reversed LinkedList size is
    // equal to K or not
    // if it is not equal to k that means we have reversed
    // the last set of size K and we don't need to call the
    // recursive function
    if (count >= k) {
        head->next = reverseByN(curr, k);
    }
    return newHead;
}
```

Thankyou

To be continued….