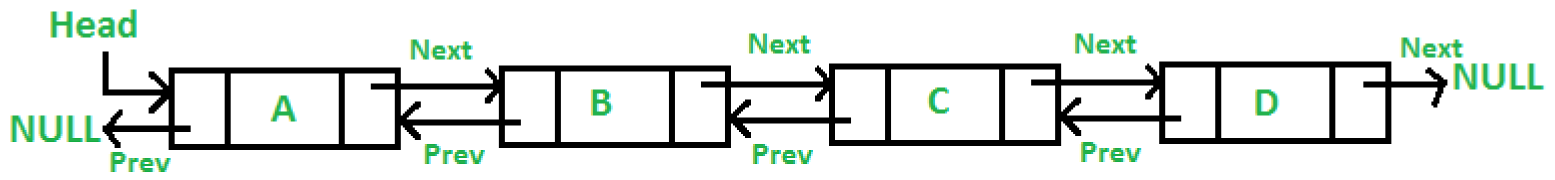
Data Structures and Algorithms

Trong-Hop Do
University of Information Technology, HCM city

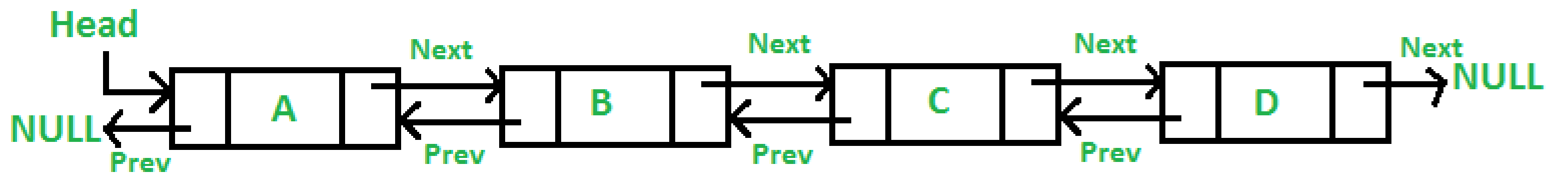
Doubly Linked List

- A Doubly Linked List (DLL) contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list.



Advantages over singly linked list

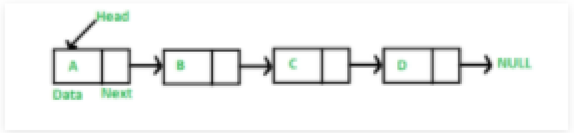
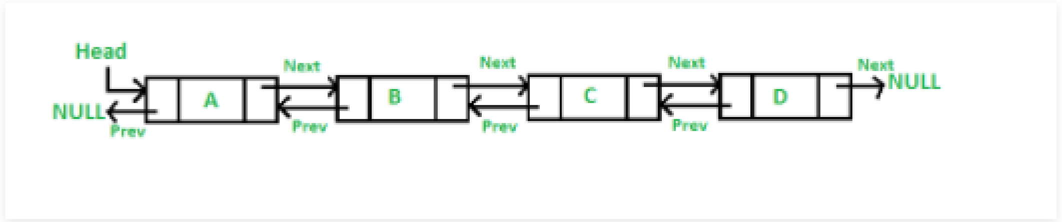
- A DLL can be traversed in both forward and backward direction.
- The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
- We can quickly insert a new node before a given node.
- In singly linked list, to delete a node, pointer to the previous node is needed. To 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

- Every node of DLL Require extra space for an previous pointer.
- 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.

Singly linked list vs Doubly linked list

SINGLY LINKED LIST (SLL)	DOUBLY LINKED LIST (DLL)
SLL has nodes with only a data field and next link field.	DLL has nodes with a data field, a previous link field and a next link field.
	
In SLL, the traversal can be done using the next node link only.	In DLL, the traversal can be done using the previous node link or the next node link.
The SLL occupies less memory than DLL as it has only 2 fields.	The DLL occupies more memory than SLL as it has 3 fields.
Less efficient access to elements.	More efficient access to elements.

Data structure

- Data structure of a Node in Doubly Linked List

```
typedef struct tagDnode
{ Data Info;
struct tagDnode *pPre;
struct tagDnode *pNext;
}DNode;
```

- Data structure of Doubly Linked List

```
Typedef struct tagDList
{ DNode *pHead;
DNode *pTail;
}DList;
```

Operations in Doubly Linked List

- Create an empty doubly linked list
- Create a node with specified data
- Insert a node to doubly linked list
- Delete a node from doubly linked list
- Find a node in doubly linked list

Create empty list

```
void CreateDList(DList &l)
{
    l.DHead=NULL;
    l.DTail=NULL;
}
```

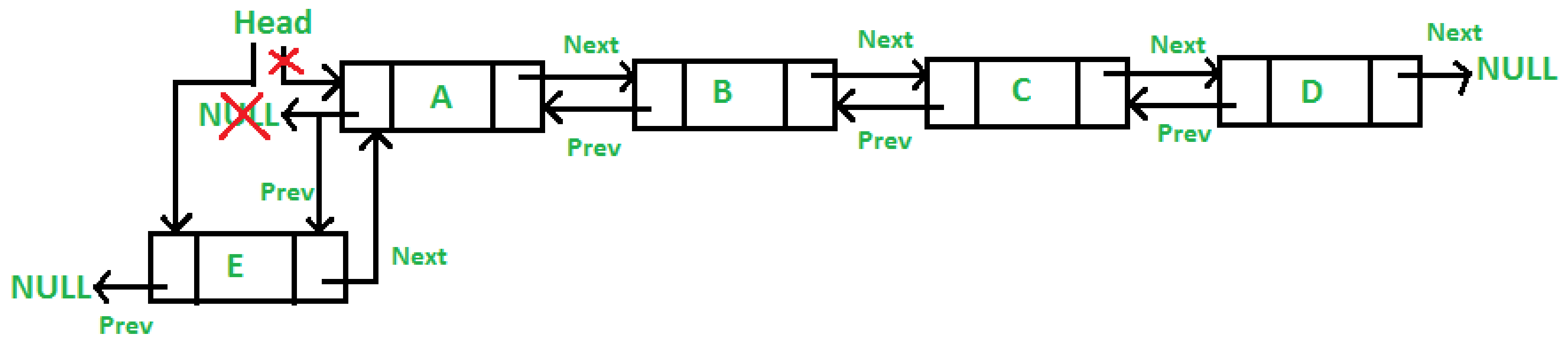
Create a node with specified data x

```
DNode *CreateDNode(int x)
{       DNode *tam;
        tam=new DNode;
        if(tam==NULL)
        {       printf("khong con du bo nho");
                exit(1);
        }
        else
        {       tam->Info=x;
                tam->pNext=NULL;
                tam->pPre=NULL;
                return tam;
        }
}
```

Insert a node to doubly linked list

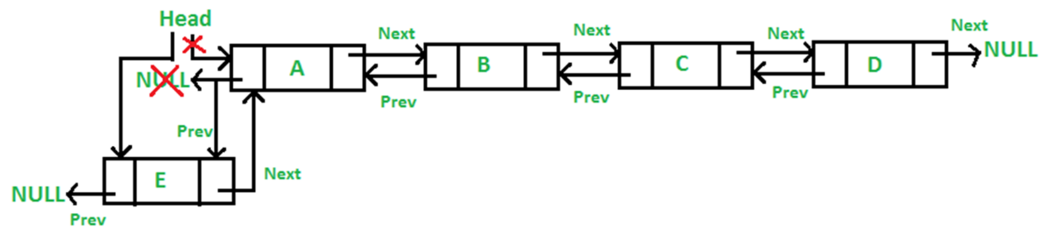
- At the front
- After a given node
- At the end
- Before a given node

Insert a node at the front of doubly linked list



Insert a node at the front of doubly linked list

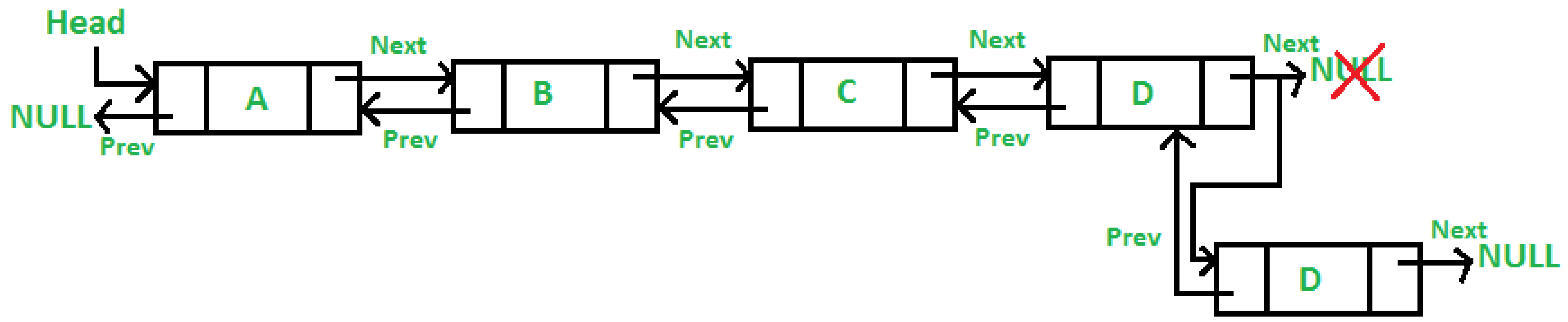
```
void AddFront(DList &l, DNode *temp)
{
        if(l.pHead==NULL)//empty list
        {
                l.pHead=temp;
                l.pTail=l.pHead;
        }
        else
        {
                temp->pNext=l.pHead;
                l.pHead->pPre=temp;
                l.pHead=temp;
        }
}
```



Insert a node with specified data at the front

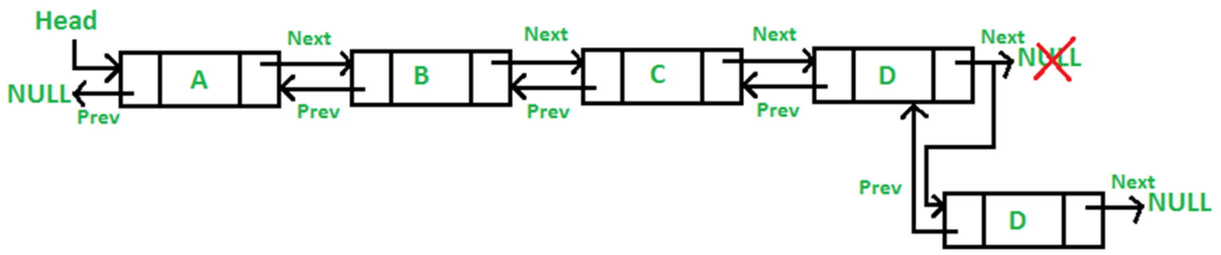
```
/* Given a reference (pointer to pointer) to the head of a list
   and an int, inserts a new node on the front of the list. */
void AddFront(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    /* 2. put in the 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;
}
```

Insert a node at the end of doubly linked list



Insert a node at the end of doubly linked list

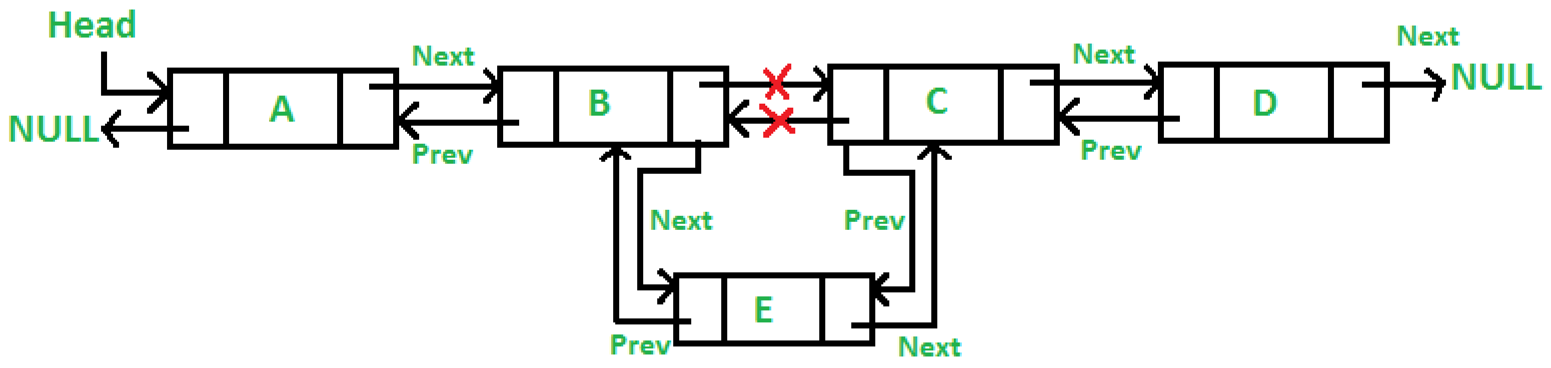
```
void AddEnd(DList &l,DNode *temp)
{
        if(l.pHead==NULL)
        {
                l.pHead=temp;
                l.pTail=l.pHead;
        }
        else
        {
                temp->pPre=l.pTail;
                l.pTail->pNext=temp;
                temp=l.pTail;
        }
}
```



Insert a node with specified data at the end

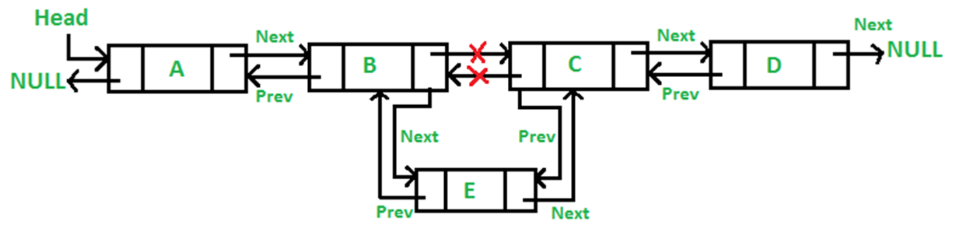
```
void AddEnd(struct Node** head_ref, int new_data)
{
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node)); /* 1. allocate */
    struct Node* last = *head_ref; /* used in step 5*/
    new_node->data = new_data; /* 2. put in the data  */
    new_node->next = NULL; /* 3. This new node is the last node, so next node is 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;
    }
    while (last->next != NULL) /* 5. Else traverse till the last node */
        last = last->next;
    last->next = new_node; /* 6. Change the next of last node */
    new_node->prev = last; /* 7. Make last node as previous of new node */
    return;
}
```

Insert a node after a given node



Insert a node after a given node

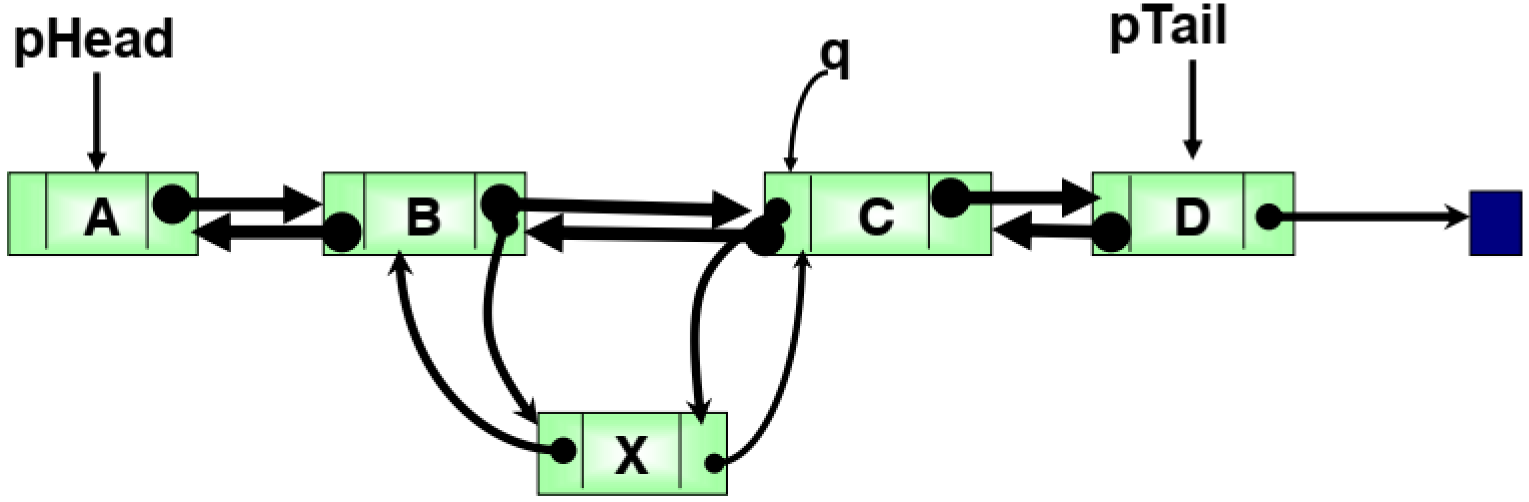
```
void AddAfter(DList &l,DNode *temp, DNode *q)
{
        DNode *p;
        p=q->pNext;
        if(q!=NULL)//list is not empty
        {
                temp->pNext=p;
                temp->pPre=q;
                q->pNext=temp;
                if(p!=NULL)
                        p->pPre=temp;
                if(q==l.pTail)
                        l.pTail=temp;
        }
        else
                AddFirst(l,temp);
}
```



Insert a node with specified data after a given node

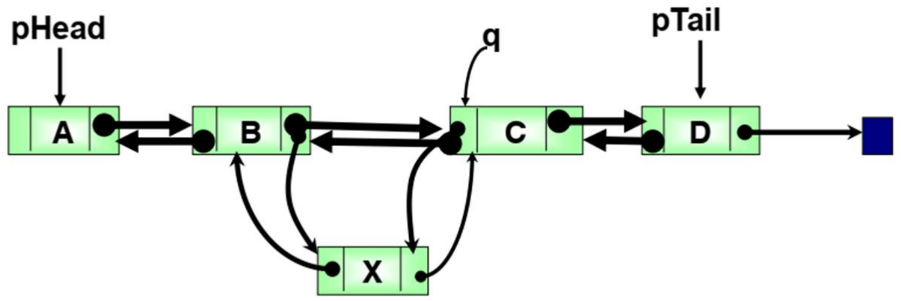
```
void AddAfter(struct Node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL) {
        printf("the given previous node cannot be NULL"); return;
    }
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node)); /* 2.
allocate new node */
    new_node->data = new_data;  /* 3. put in the data  */
    new_node->next = prev_node->next; /* 4. Make next of new node as next of
prev_node */
    prev_node->next = new_node; /* 5. Make the next of prev_node as new_node */
    new_node->prev = prev_node; /* 6. Make prev_node as previous of new_node */
    /* 7. Change previous of new_node's next node */
    if (new_node->next != NULL)
        new_node->next->prev = new_node;
}
```

Insert a node before a node



Insert a node before a node

```
void AddBefore(DList &l,DNode *temp,DNode *q)
{ DNode *p;
       p=q->pPre;
       if(q!=NULL)
       {
              temp->pNext=q;
              q->pPre=temp;
              temp->pPre=p;
              if(p!=NULL)
                     p->pNext=temp;
              if(q==l.pHead)
                     l.pHead = temp;
       }
       else
              AddEnd(l,temp);
}
```



Insert a node with specified data before a node

```
void insertBefore(struct Node** head_ref, struct Node* next_node, int new_data)
{
    if (next_node == NULL) {  /*1. check if the given next_node is NULL */
        printf("the given next node cannot be NULL");
        return;
    }
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));  /* 2. allocate new node */
    new_node->data = new_data;  /* 3. put in the data */
    new_node->prev = next_node->prev; /* 4. Make prev of new node as prev of next_node */
    next_node->prev = new_node;  /* 5. Make the prev of next_node as new_node */
    new_node->next = next_node;   /* 6. Make next_node as next of new_node */
    if (new_node->prev != NULL)  /* 7. Change next of new_node's previous node */
        new_node->prev->next = new_node;
    else /* 8. If the prev of new_node is NULL, it will be the new head node */
        (*head_ref) = new_node;
}
```

Delete a node at the front of doubly linked list

```
void DeleteFront(DList &l)
{
        DNode *p;
        if(l.pHead!=NULL)
        {
                p=l.pHead;
                l.pHead=l.pHead->pNext;
                l.pHead->pPre=NULL;
                delete p;
                if(l.pHead==NULL)
                        l.pTail=NULL;
        }
}
```

Delete a node at the end of doubly linked list

```
void DeleteEnd(DList &l )
{
        DNode *p;
        if(l.pHead!=NULL)
        {
                p=l.pTail;
                l.pTail=l.pTail->Pre;
                l.pTail->pNext=NULL;
                delete p;
                if(l.pTail==NULL)
                l.pHead=NULL;
        }
}
```

Delete a node before a specified node

```
void DeleteBefore(DList &l,DNode *q)
{
       DNode *p;
       if(q!=NULL) //node q exist
       {
              p=q->pPre;
              if(p!=NULL)
              {
                     q->pPre=p->pPre;
                     if(p==l.pHead)//p is the head Node
                            l.pHead=q;
                     else //p is not the head Node
                            p->pPre->pNext=q;
                     delete p;
              }
       }
       else
              DeleteEnd(l);
}
```

Delete a node with specified data x

```
int DeleteX(DList &l,int x)
{
        DNode *p;
        DNode *q;
        q=NULL;
        p=l.pHead;
        while(p!=NULL)
        {
                if(p->Info==x)
                break;
                q=p;
                p=p->pNext;
        }
        if(q==NULL) return 0;//no node with Info = x found
        if(q!=NULL)
                DeleteEnd(l,q);
        else
                DeleteFront(l);
        return 1;
}
```

Delete a specified node in doubly linked list

```
void deleteNode(struct Node** head_ref, struct Node* del)
{
    if (*head_ref == NULL || del == NULL)
        return;
    if (*head_ref == del) /* If node to be deleted is head node */
        *head_ref = del->next;
    if (del->next != NULL) /* Change next only if node to be deleted is NOT the
last node */
        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;
    free(del); /* Finally, free the memory occupied by del*/
}
```

Delete all node with specified data x

```c
void deleteAllX(struct Node** head, int x)
{
    if ((*head) == NULL)/* if list is empty */
        return;
    struct Node* current = *head;
    struct Node* next;
    while (current != NULL) { /* traverse the list up to the end */
        if (current->data == x) { /* if node found with the value 'x' */
            next = current->next; /* save current's next node in the pointer 'next' */
            deleteNode(head, current); /* delete node pointed to by 'current' */
            current = next; /* update current */
        }
        else /* else simply move to the next node */
            current = current->next;
    }
}
```

Delete a node at given a position

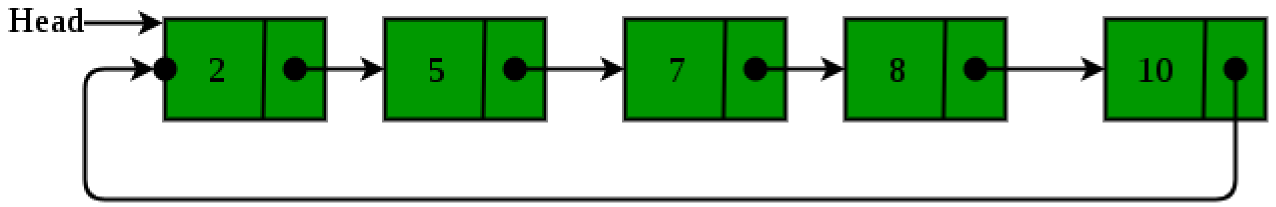
```
void deleteNodeAtGivenPos(struct Node** head_ref, int n)
{
    if (*head_ref == NULL || n <= 0)
        return;
    struct Node* current = *head_ref;
    int i;
    /* traverse up to the node at position 'n' from the beginning */
    for (int i = 1; current != NULL && i < n; i++)
        current = current->next;

    if (current == NULL) /* if 'n' is greater than the number of nodes in the
list */
        return;
    deleteNode(head_ref, current); /* delete the node pointed to by 'current'
*/
}
```

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;
}
```

Circular Linked List

- Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



Advantages of Circular Linked Lists:

- Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

- Useful for implementation of queue. Unlike the implementation using singly linked list, we don’t need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

- Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

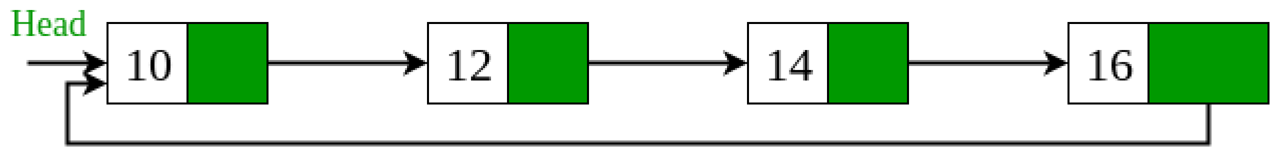
- Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

Create new node in circular linked list

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

Check if a linked list is Circular Linked List

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



- An empty linked list is considered as circular.
- Note that this problem is different from cycle detection problem, here all nodes have to be part of cycle.

Check if a linked list is Circular Linked List

```
bool isCircular(struct Node *head)
{
    // An empty linked list is circular
    if (head == NULL)
       return true;
    // Next of head
    struct Node *node = head->next;
    // This loop would stop in both cases (1) If Circular (2) Not circular
    while (node != NULL && node != head)
       node = node->next;

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

Convert singly linked list into circular linked list

```
// Function that convert singly linked list into circular linked list.
struct Node* circular(struct Node* head)
{
    // declare a node variable start and assign head node into start node.
    struct Node* start = head;

    // check that while head->next not equal to NULL then head points to
next node.
    while (head->next != NULL)
        head = head->next;

    // if head->next points to NULL then start assign to the head->next
node.
    head->next = start;
    return start;
}
```

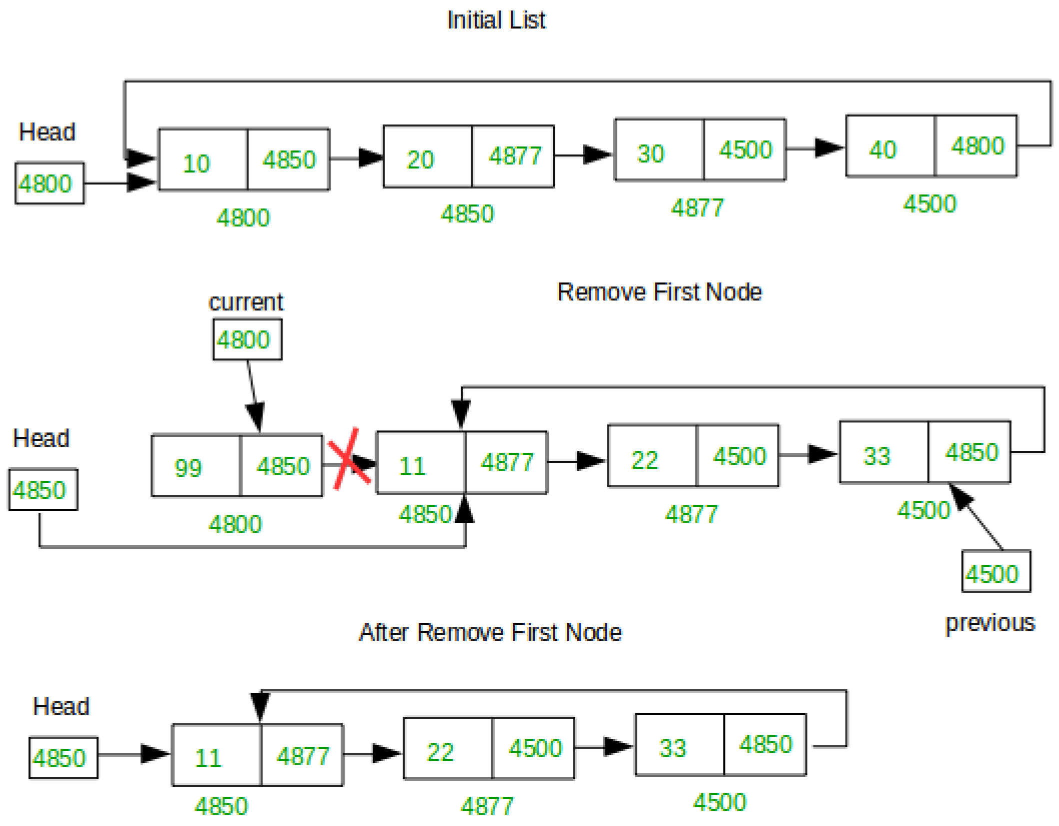
Traverse a given Circular linked list

```c
void printList(struct Node *first)
{
    struct Node *temp = first;

    // If linked list is not empty
    if (first != NULL)
    {
        // Keep printing nodes till we reach the first node again
        do
        {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        while (temp != first);
    }
}
```

Insert a node at the beginning of a Circular linked list

```
void AddFront(Node **head_ref, int data)
{
    Node *ptr1 = new Node();
    Node *temp = *head_ref;
    ptr1->data = data;
    ptr1->next = *head_ref;
    /* If linked list is not NULL then set the next of last node */
    if (*head_ref != NULL)
    {
        while (temp->next != *head_ref)
            temp = temp->next;
        temp->next = ptr1;
    }
    else
        ptr1->next = ptr1; /*For the first node */

    *head_ref = ptr1;
}
```

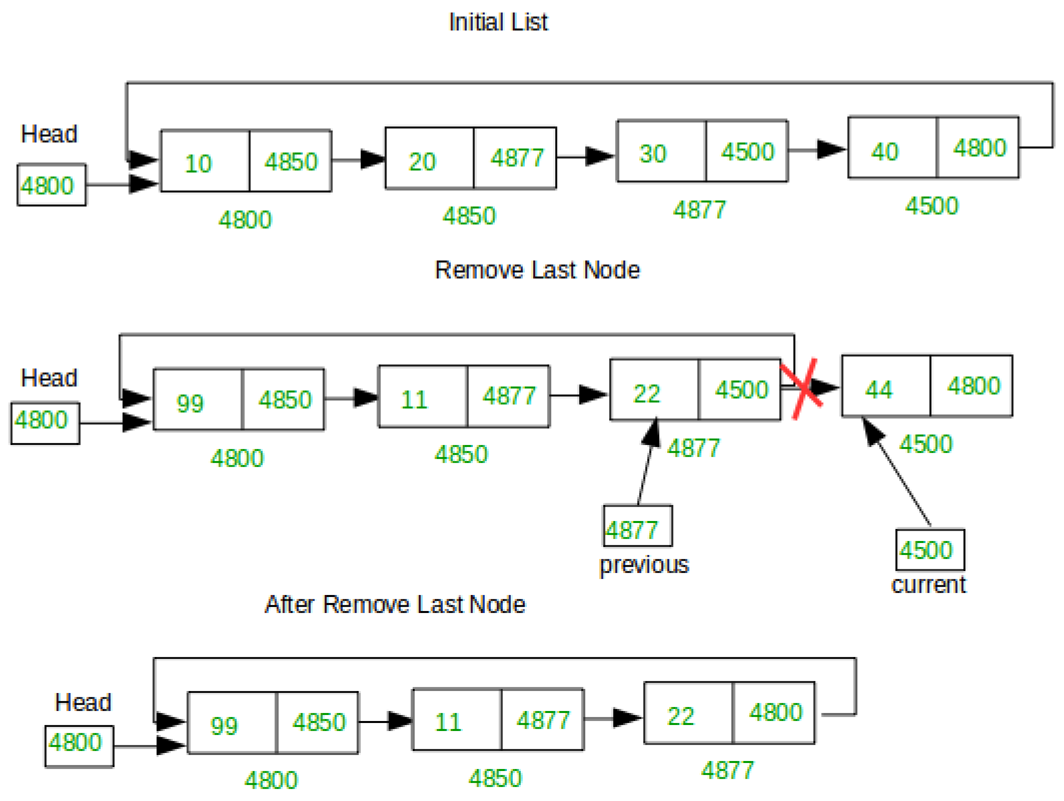
Delete a node at the front



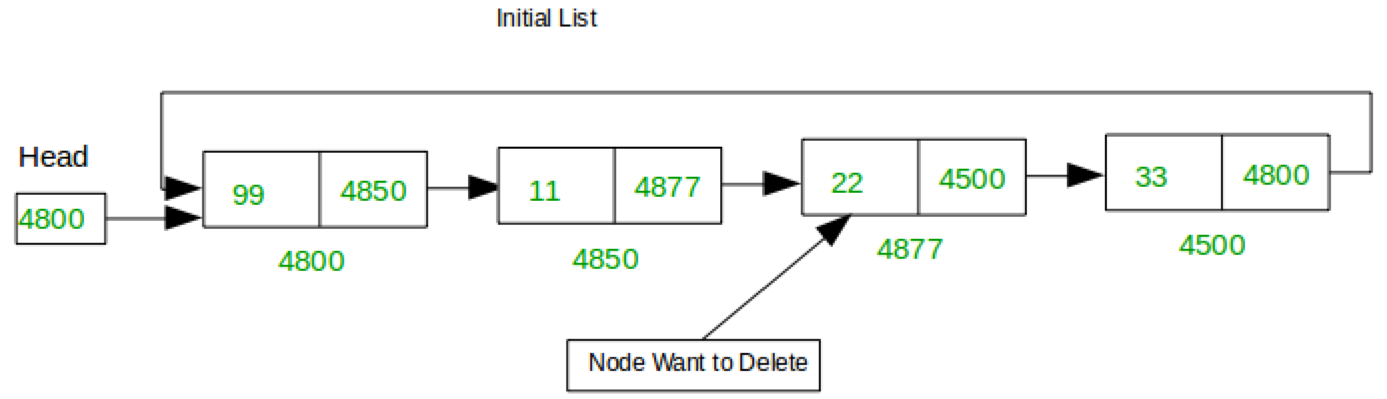
Delete a node at the front

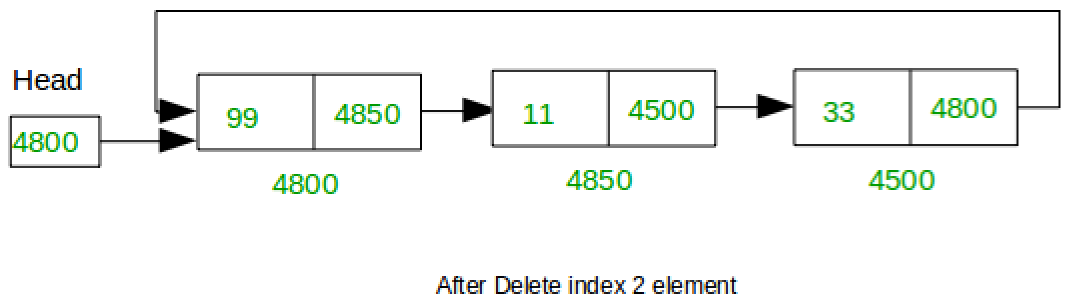
```
void DeleteFirst(struct Node** head)
{   struct Node *previous = *head, *firstNode = *head;
    // check if list doesn't have any node if not then return
    if (*head == NULL) {
        printf("\nList is empty\n");
        return;
    }
    // check if list have single node if yes then delete it and return
    if (previous->next == previous) {
        *head = NULL;
        return;
    }
    while (previous->next != *head) { // traverse second node to first
        previous = previous->next;
    }
    // now previous is last node and
    // first node(firstNode) link address
    // put in last node(previous) link
    previous->next = firstNode->next;
    *head = previous->next; // make second node as head node
    free(firstNode);
    return;
}
```

Delete a node at the end



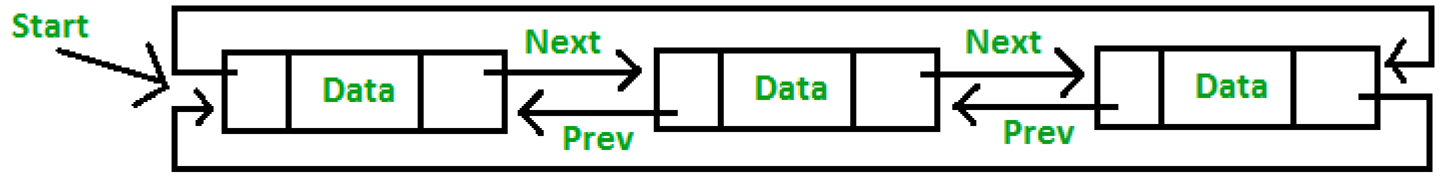
Delete a node at given index





Doubly Circular Linked List

- Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by previous and next pointer and the last node points to first node by next pointer and also the first node points to last node by previous pointer.



Advantage of Doubly Circular Linked List

- List can be traversed from both the directions i.e. from head to tail or from tail to head.
- Jumping from head to tail or from tail to head is done in constant time O(1).
- Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

Disadvantages of Doubly Circular Linked List

- It takes slightly extra memory in each node to accommodate previous pointer.
- Lots of pointers involved while implementing or doing operations on a list. So, pointers should be handled carefully otherwise data of the list may get lost.

Applications of Circular doubly linked list

- Managing songs playlist in media player applications.
- Managing shopping cart in online shopping.