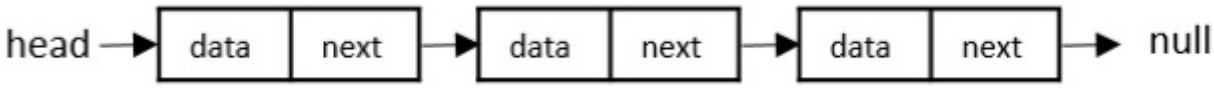
Linked List Data Structure

What is Linked List?

A linked list is a linear data structure which can store a collection of "nodes" connected together via links i.e. pointers. Linked lists nodes are not stored at a contiguous location, rather they are linked using pointers to the different memory locations. A node consists of the data value and a pointer to the address of the next node within the linked list.

A linked list is a dynamic linear data structure whose memory size can be allocated or de-allocated at run time based on the operation insertion or deletion, this helps in using system memory efficiently. Linked lists can be used to implment various data structures like a stack, queue, graph, hash maps, etc.



A linked list starts with a **head** node which points to the first node. Every node consists of data which holds the actual data (value) associated with the node and a next pointer which holds the memory address of the next node in the linked list. The last node is called the tail node in the list which points to **null** indicating the end of the list.

Linked Lists vs Arrays

In case of arrays, the size is given at the time of creation and so arrays are of fixed lenghth where as Linked lists are dynamic in size and any number of nodes can be added in the linked lists dynamically. An array can accommodate similar types of data types where as linked lists can store various nodes of different data types.

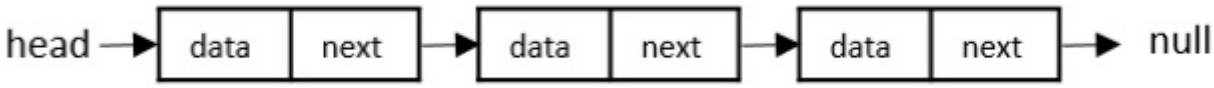
Types of Linked List

Following are the various types of linked list.

Singly Linked Lists

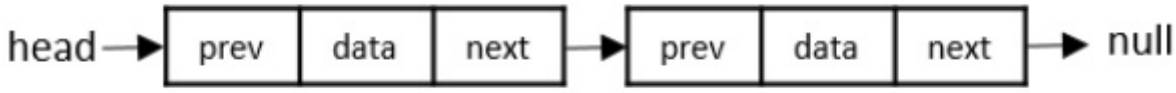
Singly linked lists contain two "buckets" in one node; one bucket holds the data and the other bucket holds the address of the next node of the list. Traversals can be

done in one direction only as there is only a single link between two nodes of the same list.



Doubly Linked Lists

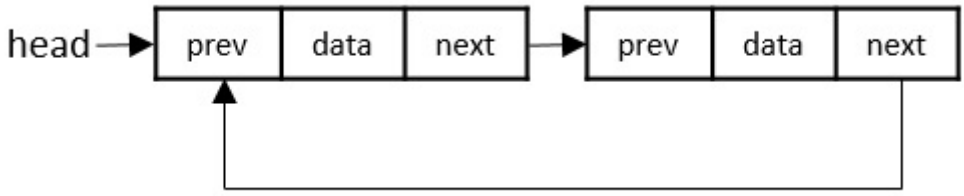
Doubly Linked Lists contain three "buckets" in one node; one bucket holds the data and the other buckets hold the addresses of the previous and next nodes in the list. The list is traversed twice as the nodes in the list are connected to each other from both sides.



Circular Linked Lists

Circular linked lists can exist in both singly linked list and doubly linked list.

Since the last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.



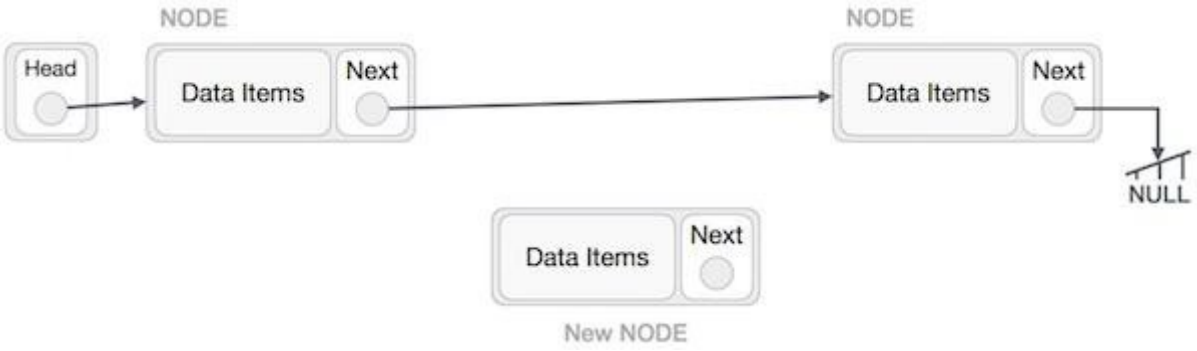
Basic Operations in Linked List

The basic operations in the linked lists are insertion, deletion, searching, display, and deleting an element at a given key. These operations are performed on Singly Linked Lists as given below –

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key.

Linked List - Insertion Operation

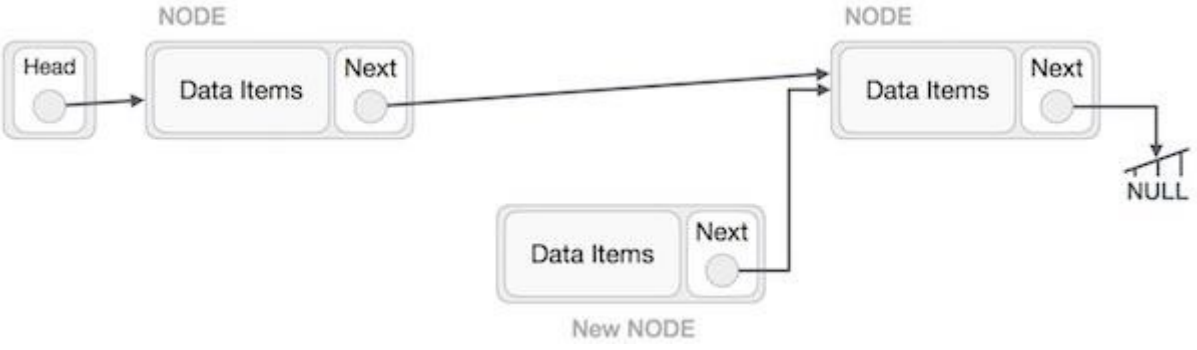
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode). Then point B.next to C –

```
NewNode.next -> RightNode;
```

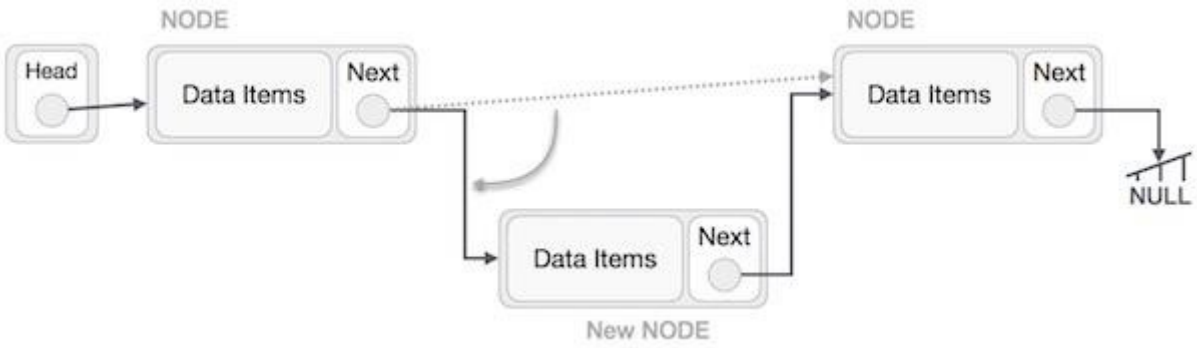
It should look like this –



Now, the next node at the left should point to the new node.

```
LeftNode.next -> NewNode;
```

This will put the new node in the middle of the two. The new list should look like this –



Insertion in linked list can be done in three different ways. They are explained as follows –

Insertion at Beginning

In this operation, we are adding an element at the beginning of the list.

Algorithm

```
1. START
2. Create a node to store the data
3. Check if the list is empty
4. If the list is empty, add the data to the node and
   assign the head pointer to it.
5. If the list is not empty, add the data to a node and link to the
   current head. Assign the head to the newly added node.
6. END
```

Example

Following are the implementations of this operation in various programming languages –

C | C++ | Java | Python

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
```

```
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void main(){
   int k=0;
   insertatbegin(12);
   insertatbegin(22);
   insertatbegin(30);
   insertatbegin(44);
   insertatbegin(50);
   printf("Linked List: ");

   // print list
   printList();
}
```

Output

```
Linked List:
[ 50  44  30  22  12 ]
```

Insertion at Ending

In this operation, we are adding an element at the ending of the list.

Algorithm

```
1. START
2. Create a new node and assign the data
3. Find the last node
4. Point the last node to new node
5. END
```

Example

Following are the implementations of this operation in various programming languages –

C | C++ | Java | Python

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
```

```
   }
   printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void insertatend(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;
   struct node *linkedlist = head;

   // point it to old first node
   while(linkedlist->next != NULL)
      linkedlist = linkedlist->next;

   //point first to new first node
   linkedlist->next = lk;
}
void main(){
   int k=0;
   insertatbegin(12);
   insertatend(22);
   insertatend(30);
   insertatend(44);
   insertatend(50);
   printf("Linked List: ");

   // print list
```

```
   printList();
}
```

Output

```
Linked List:
[ 12 22 30 44 50 ]
```

Insertion at a Given Position

In this operation, we are adding an element at any position within the list.

Algorithm

```
1. START
2. Create a new node and assign data to it
3. Iterate until the node at position is found
4. Point first to new first node
5. END
```

Example

Following are the implementations of this operation in various programming languages –

C | C++ | Java | Python

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");
```

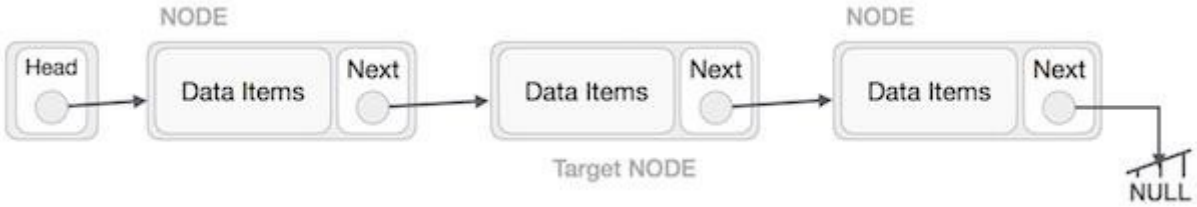
```
   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void insertafternode(struct node *list, int data){
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;
   lk->next = list->next;
   list->next = lk;
}
void main(){
   int k=0;
   insertatbegin(12);
   insertatbegin(22);
   insertafternode(head->next, 30);
   printf("Linked List: ");

   // print list
   printList();
}
```

Output

```
Linked List:
[ 22 12 30 ]
```

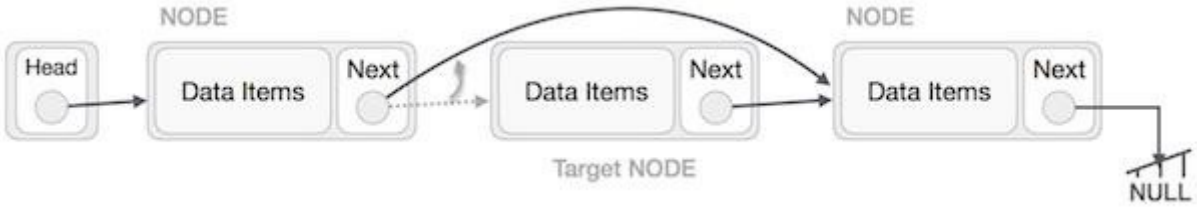
Linked List - Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



The left (previous) node of the target node now should point to the next node of the target node –

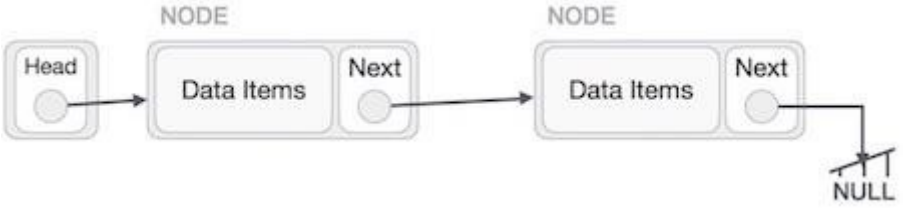
```
LeftNode.next -> TargetNode.next;
```



This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next -> NULL;
```

We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion in linked lists is also performed in three different ways. They are as follows –

Deletion at Beginning

In this deletion operation of the linked, we are deleting an element from the beginning of the list. For this, we point the head to the second node.

Algorithm

```
1. START
2. Assign the head pointer to the next node in the list
3. END
```

Example

Following are the implementations of this operation in various programming languages –

C | C++ | Java | Python

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
```

```
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void deleteatbegin(){
   head = head->next;
}
int main(){
   int k=0;
   insertatbegin(12);
   insertatbegin(22);
   insertatbegin(30);
   insertatbegin(40);
   insertatbegin(55);
   printf("Linked List: ");

   // print list
   printList();
   deleteatbegin();
   printf("\nLinked List after deletion: ");

   // print list
```

```
   printList();
}
```

Output

```
Linked List:
[ 55  40  30  22  12 ]
Linked List after deletion:
[ 40  30  22  12 ]
```

Deletion at Ending

In this deletion operation of the linked, we are deleting an element from the ending of the list.

Algorithm

```
1. START
2. Iterate until you find the second last element in the list.
3. Assign NULL to the second last element in the list.
4. END
```

Example

Following are the implementations of this operation in various programming languages –

C | C++ | Java | Python

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
```

```
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void deleteatend(){
   struct node *linkedlist = head;
   while (linkedlist->next->next != NULL)
      linkedlist = linkedlist->next;
   linkedlist->next = NULL;
}
void main(){
   int k=0;
   insertatbegin(12);
   insertatbegin(22);
   insertatbegin(30);
   insertatbegin(40);
   insertatbegin(55);
   printf("Linked List: ");

   // print list
   printList();
   deleteatend();
   printf("\nLinked List after deletion: ");
```

```
   // print list
   printList();
}
```

Output

```
Linked List:
[ 55  40  30  22  12 ]
Linked List after deletion:
[ 55  40  30  22 ]
```

Deletion at a Given Position

In this deletion operation of the linked, we are deleting an element at any position of the list.

Algorithm

```
1. START
2. Iterate until find the current node at position in the list.
3. Assign the adjacent node of current node in the list
   to its previous node.
4. END
```

Example

Following are the implementations of this operation in various programming languages –

C | C++ | Java | Python

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;
```

```c
// display the list
void printList(){
   struct node *p = head;
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void deletenode(int key){
   struct node *temp = head, *prev;
   if (temp != NULL && temp->data == key) {
      head = temp->next;
      return;
   }

   // Find the key to be deleted
   while (temp != NULL && temp->data != key) {
      prev = temp;
      temp = temp->next;
   }

   // If the key is not present
   if (temp == NULL) return;
```

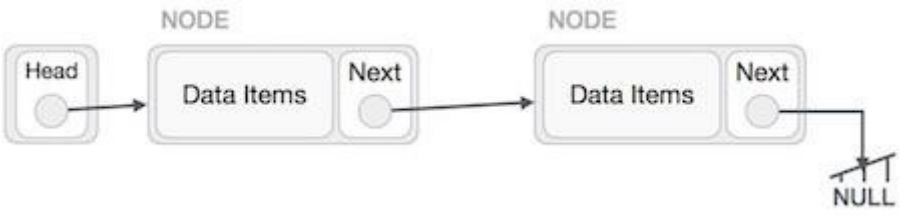
```
    // Remove the node
    prev->next = temp->next;
}
void main(){
    int k=0;
    insertatbegin(12);
    insertatbegin(22);
    insertatbegin(30);
    insertatbegin(40);
    insertatbegin(55);
    printf("Linked List: ");

    // print list
    printList();
    deletenode(30);
    printf("\nLinked List after deletion: ");

    // print list
    printList();
}
```

Output

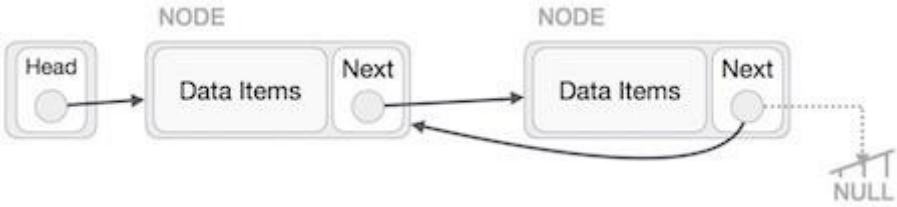
```
Linked List:
[ 55  40  30  22  12 ]
Linked List after deletion:
[ 55  40  22  12 ]
```

Linked List - Reversal Operation

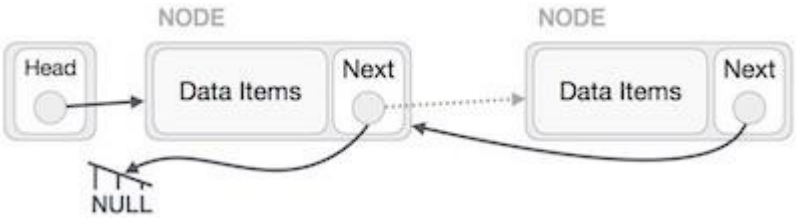
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



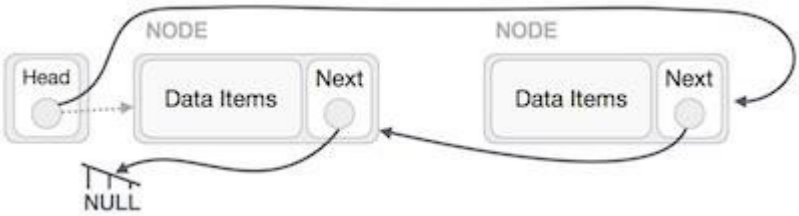
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



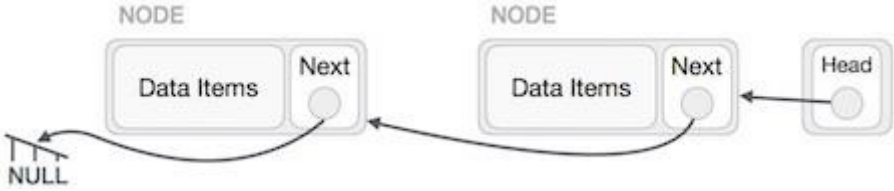
We have to make sure that the last node is not the last node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.



Algorithm

Step by step process to reverse a linked list is as follows –

```
1. START
2. We use three pointers to perform the reversing:
   prev, next, head.
3. Point the current node to head and assign its next value to
   the prev node.
4. Iteratively repeat the step 3 for all the nodes in the list.
5. Assign head to the prev node.
```

Example

Following are the implementations of this operation in various programming languages –

C | C++ | Java | Python

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
```

```
}
void reverseList(struct node** head){
   struct node *prev = NULL, *cur=*head, *tmp;
   while(cur!= NULL) {
      tmp = cur->next;
      cur->next = prev;
      prev = cur;
      cur = tmp;
   }
   *head = prev;
}
void main(){
   int k=0;
   insertatbegin(12);
   insertatbegin(22);
   insertatbegin(30);
   insertatbegin(40);
   insertatbegin(55);
   printf("Linked List: ");

   // print list
   printList();
   reverseList(&head);
   printf("\nReversed Linked List: ");
   printList();
}
```

Output

```
Linked List:
[ 55  40  30  22  12 ]
Reversed Linked List:
[ 12  22  30  40  55 ]
```

Linked List - Search Operation

Searching for an element in the list using a key element. This operation is done in the same way as array search; comparing every element in the list with the key element given.

Algorithm

```
1 START
2 If the list is not empty, iteratively check if the list
  contains the key
3 If the key element is not present in the list, unsuccessful
  search
4 END
```

Example

Following are the implementations of this operation in various programming languages –

C | C++ | Java | Python

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
```

```
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
int searchlist(int key){
   struct node *temp = head;
   while(temp != NULL) {
      if (temp->data == key) {
         return 1;
      }
      temp=temp->next;
   }
   return 0;
}
void main(){
   int k=0;
   insertatbegin(12);
   insertatbegin(22);
   insertatbegin(30);
   insertatbegin(40);
   insertatbegin(55);
   printf("Linked List: ");

   // print list
   printList();
   int ele = 30;
   printf("\nElement to be searched is: %d", ele);
   k = searchlist(30);
   if (k == 1)
      printf("\nElement is found");
   else
      printf("\nElement is not found in the list");
}
```

Output

```
Linked List:
[ 55  40  30  22  12 ]
```

```
Element to be searched is: 30
Element is found
```

Linked List - Traversal Operation

The traversal operation walks through all the elements of the list in an order and displays the elements in that order.

Algorithm

```
1. START
2. While the list is not empty and did not reach the end of the list,
   print the data in each node
3. END
```

Example

Following are the implementations of this operation in various programming languages –

C | C++ | Java | Python

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

   // display the list
   void printList(){
   struct node *p = head;
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
```

```
      p = p->next;
   }
   printf("]");
}

   //insertion at the beginning
   void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void main(){
   int k=0;
   insertatbegin(12);
   insertatbegin(22);
   insertatbegin(30);
   printf("Linked List: ");

   // print list
   printList();
}
```

Output

```
Linked List:
[ 30  22  12 ]
```

Linked List - Complete implementation

Following are the complete implementations of Linked List in various programming languages –

C | C++ | Java | Python

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");
   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}
//insertion at the beginning
void insertatbegin(int data){
   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;
   // point it to old first node
   lk->next = head;
   //point first to new first node
   head = lk;
}
void insertatend(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;
   struct node *linkedlist = head;

   // point it to old first node
   while(linkedlist->next != NULL)
      linkedlist = linkedlist->next;
```

```
   //point first to new first node
   linkedlist->next = lk;
}
void insertafternode(struct node *list, int data){
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;
   lk->next = list->next;
   list->next = lk;
}
void deleteatbegin(){
   head = head->next;
}
void deleteatend(){
   struct node *linkedlist = head;
   while (linkedlist->next->next != NULL)
      linkedlist = linkedlist->next;
   linkedlist->next = NULL;
}
void deletenode(int key){
   struct node *temp = head, *prev;
   if (temp != NULL && temp->data == key) {
      head = temp->next;
      return;
   }

   // Find the key to be deleted
   while (temp != NULL && temp->data != key) {
      prev = temp;
      temp = temp->next;
   }

   // If the key is not present
   if (temp == NULL) return;

   // Remove the node
   prev->next = temp->next;
}
int searchlist(int key){
   struct node *temp = head;
   while(temp != NULL) {
      if (temp->data == key) {
```

```
            return 1;
        }
        temp=temp->next;
    }
    return 0;
}
void main(){
    int k=0;
    insertatbegin(12);
    insertatbegin(22);
    insertatend(30);
    insertatend(44);
    insertatbegin(50);
    insertafternode(head->next->next, 33);
    printf("Linked List: ");

    // print list
    printList();
    deleteatbegin();
    deleteatend();
    deletenode(12);
    printf("\nLinked List after deletion: ");

    // print list
    printList();
    insertatbegin(4);
    insertatbegin(16);
    printf("\nUpdated Linked List: ");
    printList();
    k = searchlist(16);
    if (k == 1)
        printf("\nElement is found");
    else
        printf("\nElement is not present in the list");
}
```

Output

```
Linked List:
[ 50  22  12  33  30  44 ]
Linked List after deletion:
[ 22  33  30 ]
Updated Linked List:
```

```
[ 16  4  22  33  30 ]
Element is found
```