What is a Linked List?

A linked list is a linear data structure that stores a collection of data elements dynamically.

Nodes represent those data elements, and links or pointers connect each node.

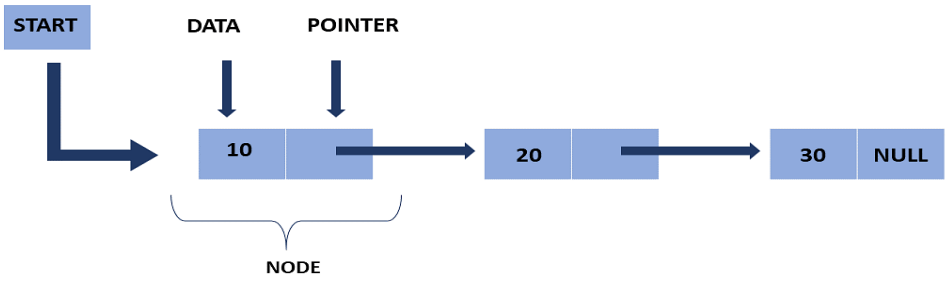
Each node consists of two fields, the information stored in a linked list and a pointer that stores the address of its next node.

The last node contains null in its second field because it will point to no node.

A linked list can grow and shrink its size, as per the requirement. It does not waste memory space.

Representation of a Linked List

This representation of a linked list depicts that each node consists of two fields. The first field consists of data, and the second field consists of pointers that point to another node.



Here, the start pointer stores the address of the first node, and at the end, there is a null pointer that states the end of the Linked List.

Key Differences Between Linked Lists and Arrays

What Are They?

An array is a data structure that holds a fixed number of elements of the same type in a continuous block of memory. You can think of it as a row of boxes where each box contains a value. In contrast, a linked list is made up of nodes, each containing data and a pointer to the next node. This forms a chain, allowing elements to be scattered throughout memory.

Size Matters

Arrays have a fixed size. You decide how many elements to store upfront, and that's it. If you need more space later, you'll have to create a new array and move the elements. Linked lists,

on the other hand, can grow or shrink as needed. You can easily add or remove nodes without worrying about the total size.

Memory Allocation

Arrays use contiguous memory, meaning all elements are stored next to each other. This makes accessing elements fast. Linked lists use separate memory for each node, which can lead to some inefficiency because of the extra memory needed for pointers.

Memory Usage

Arrays are generally more memory-efficient since they don't need extra pointers. Linked lists require more memory due to the pointers that link each node together.

Insertion and Deletion

When inserting or deleting elements, arrays can be slow, taking O(n) time, especially if you're working with elements in the middle or at the beginning. This is because you may need to shift other elements. Linked lists allow for quick insertions at the beginning (O(1)), but inserting at the end can take O(n) time if you don't have a direct pointer to the last node.

Searching for Elements

Searching in arrays can take O(n) time in the worst case. If the array is sorted, you can use binary search, which is faster at O(log n). For linked lists, you also typically spend O(n) time searching since you need to traverse each node until you find what you're looking for.

Accessing Elements

Accessing elements is quick in arrays, with a time complexity of O(1). You can jump directly to any index. In linked lists, accessing an element requires O(n) time because you need to follow the chain of nodes from the start.

Deleting Elements

In arrays, deleting an element from the beginning or middle can take O(n) time since other elements need to be shifted. However, deleting from the end can be done in O(1). For linked lists, removing a node at the beginning is O(1), but it takes O(n) to remove from the middle or end, as you have to find the node first.

Creation of Node and Declaration of Linked Lists:

```
struct node
{
 int data;
 struct node * next;
};
struct node * n;
```

n=(struct node*)malloc(sizeof(struct node*));

It is a declaration of a node that consists of the first variable as data and the next as a pointer, which will keep the address of the next node. Here you need to use the malloc function to allocate memory for the nodes dynamically.

Essential Operation on Linked Lists

- Traversing: To traverse all nodes one by one.
- Insertion: To insert new nodes at specific positions.
- Deletion: To delete nodes from specific positions.
- Searching: To search for an element from the linked list.

Traversal

In this operation, you will display all the nodes in the linked list. When the temp is null, it means you traversed all the nodes, and you reach the end of the linked list and get out from the while loop.

```
struct node * temp = start;

printf("\n list empty are-");

while (temp!= NULL)

{

 printf('%d ", temp -> data)

 temp=temp -> next;

}
```

Insertion

You can add a node at the beginning, middle, and end.

Insert at the Beginning

- Create a memory for a new node.
- Store data in a new node.
- Change next to the new node to point to start.
- Change starts to tell the recently created node.

```
struct node *NewNode;

NewNode=malloc(sizeof(struct node));

NewNode -> data = 40;
```

```
NewNode -> next= start;
start= NewNode;
```

Insert at the End

- Insert a new node and store data in it.
- Traverse the last node of a linked list
- Change the next pointer of the last node to the newly created node.

```
struct node *NewNode;
NewNode=malloc(sizeof(struct node));
NewNode-> data = 40;
NewNode->next = NULL;
struct node *temp = start;
while( temp->next ! = NULL){
temp=temp -> next;
   }
 temp -> next = NewNode;
```

Insert at the Middle

- Allocate memory and store data in the new node.
- Traverse the node, which is just before the new node.
- Change the next pointer to add a new node in between.

```
struct node *NewNode;
NewNode= malloc(sizeof(struct node));
NewNode -> data = 40;
struct node - > temp = start;
for(int i=2; i<position; i++){
if (temp -> next!= NULL)
temp = temp -> next;
} }
NewNode -> next = temp -> next;
temp -> next = NewNode;
```

Deletion

You can also do deletion in the linked list in three ways either from the end, beginning, or from a specific position.

Delete from the Beginning

- The point starts at the second node.

```
start = start -> next;
```

Delete from the End

- Traverse the second last element in the linked list.
- Change its next pointer to null.

```
struct node * temp = start;

while(temp -> next -> next!= NULL){

temp=temp -> next;
```

Delete from the Middle

- Traverse the element before the element to be deleted.
- Change the next pointer to exclude the node from the linked list.

```
for (int i = 2; i, position; i++){

if (temp -> next ! = NULL)

temp = temp -> next;

        }

}

temp-> next = temp -> next -> next;
```

Searching

The search operation is done to find a particular element in the linked list. If the element is found in any location, then it returns. Else, it will return null.

Operations on Linked-Lists

Code to Insert at the Beginning of the Linked List

```
void insertatbeg()

{

   struct node *NewNode;

   int item;
```

```
    NewNode = (struct node *) malloc(sizeof(struct node *));
    if(NewNode == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value\n");
        scanf("%d",&item);
        NewNode->data = item;
        NewNode->next = start;
        start = NewNode;
        printf("\nNode inserted");
    }
}
```

Code to Insert at the End of the Linked List

```
void insertatend()
{
    struct node *NewNode,*temp;
    int item;
    NewNode = (struct node*)malloc(sizeof(struct node));
    if(NewNode == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
```

```
        NewNode->data = item;
        if(start == NULL)
        {
           NewNode -> next = NULL;
           start = NewNode;
           printf("\nNode inserted");
        }
        else
        {
           temp = start;
           while (temp -> next != NULL)
           {
              temp = temp -> next;
           }
           temp->next = NewNode;
           NewNode->next = NULL;
           printf("\nNode inserted");
        }
     }
}
```

Code to Insert at the Middle of the Linked List

```
void insertmiddle()
{
   int i,position,item;
   struct node *NewNode, *temp;
   NewNode = (struct node *) malloc (sizeof(struct node));
   if(NewNode == NULL)
   {
      printf("\nOVERFLOW");
```

```
    }
    else
    {
        printf("\nEnter element value");
        scanf("%d",&item);
        NewNode-> data = item;
        printf("\n Enter the location after which you want to insert an element ");
        scanf("\n%d",&position);
        temp=start;
        for(i=0;i<position;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                printf("\can't insert\n");
                return;
            }
        }
        NewNode ->next = temp ->next;
        temp ->next = NewNode;
        printf("\nNode inserted");
    }
}
```

Code to Delete From the Beginning of the Linked List

```
void deleteatbeg()
{
    struct node *NewNode;
    if(start == NULL)
    {
```

```
        printf("\nList is empty\n");
    }
    else
    {
        NewNode = start;
        start = NewNode->next;
        free(NewNode);
        printf("\nNode deleted from the beginning\n");
    }
}
```

Code to Delete From the End of the Linked List

```
void deleteatend()
{
    struct node *NewNode,*NewNode1;
    if(start == NULL)
    {
        printf("\list is empty");
    }
    else if(start -> next == NULL)
    {
        start = NULL;
        free(NewNode);
        printf("\n node of the list deleted\n");
    }
          else
    {
        NewNode = start;
        while(NewNode->next != NULL)
        {
```

```
            NewNode1 = NewNode;
            NewNode = NewNode ->next;
        }
        NewNode1->next = NULL;
        free(NewNode);
        printf("\nDeleted Node from the last\n");
    }
}
```

Code to Delete From the Middle of the Linked List

```
void deletemiddle()
{
    struct node *NewNode,*NewNode1;
    int position,i;
    printf("\n what location of the node \n");
    scanf("%d",&position);
    NewNode=start;
    for(i=0;i<position;i++)
    {
        NewNode1 = NewNode;
        NewNode = NewNode->next;
        if(NewNode == NULL)
        {
            printf("\nCan't delete");
            return;
        }
    }
    NewNode1 ->next = NewNode ->next;
    free(NewNode);
    printf("\nDeleted node %d ",position+1);
```

```
}
```

Code to Search for an Element From the Linked List

```
void search()
{
   struct node *NewNode;
   int item,i=0,flag;
   NewNode = start;
   if(NewNode == NULL)
   {
      printf("\nEmpty List\n");
   }
   else
   {
      printf("\nEnter an item which you want to search\n");
      scanf("%d",&item);
      while (NewNode!=NULL)
      {
         if(NewNode->data == item)
         {
            printf("item found at position%d ",i+1);
            flag=0;
         }
         else
         {
            flag=1;
         }
         i++;
         NewNode = NewNode -> next;
      }
```

```
        if(flag==1)
        {
            printf("Item not found\n");
        }
    }
}
```

Code to Show Data Elements of the Linked-List

```
void show()
{
    struct node *NewNode;
    NewNode = start;
    if(NewNode == NULL)
    {
        printf("Nothing to print");
    }
    else
    {
        printf("\nprinting values\n");
        while (NewNode!=NULL)
        {
            printf("\n%d",NewNode->data);
            NewNode = NewNode -> next;
        }
    }
}
```

Application of a Linked List

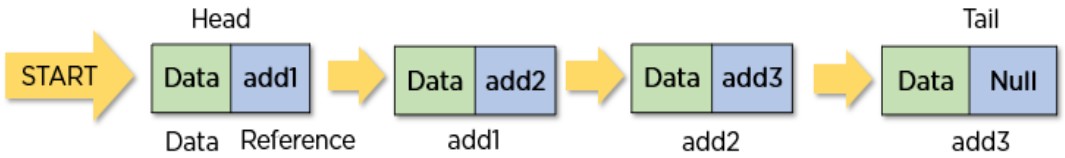
- A linked list is used to implement stacks and queues.
- A linked list also helps to implement an adjacency matrix graph.
- It is used for the dynamic memory location.

- The linked list makes it easy to deal with the addition and multiplication of polynomial operations.
- Implementing a hash table, each bucket of the hash table itself behaves as a linked list.
- It is used in a functionality known as undo in Photoshop and Word.

How to Implement A Singly Linked List in Data Structures

A singly linked list is like a train system, where it connects each bogie to the next bogie. A singly linked list is a unidirectional linked list; i.e., you can only traverse it from head node to tail node. It is used to do a slideshow or some basic operations on a notepad like undo and redo.

How to Implement a Singly Linked List?



You can create nodes of singly linked lists using classes or structures. And you link them using the next pointer.

Code:

```
// implementation of singly linked list

#include <bits/stdc++.h>

using namespace std;

//A class to create node

class Node {

public:

int data;

Node* next;

};

// A function to print the given linked list

// starting from the given node

void printList(Node* n)

{

while (n != NULL)

 {
```

```
cout << n->data << " ";
n = n->next;
 }
}
int main()
{
// creating nodes
Node* head = NULL;
Node* second = NULL;
Node* third = NULL;
Node* tail = NULL;
// allocate four nodes
head = new Node();
second = new Node();
third = new Node();
tail = new Node();
head->data = 2; // assign data in head node
head->next = second; // Link first node with second
second->data = 3; // assign data to second node
second->next = third; //Link second node with third
third->data = 5; // assign data to third node
third->next = tail; // Link third node with tail
tail->data = 7; // assign data to tail node
tail->next=NULL; // link tail node with NULL
// printing singly linked list
cout<<"Created singly linked list: "<<endl;
printList(head);
return 0;
}
```

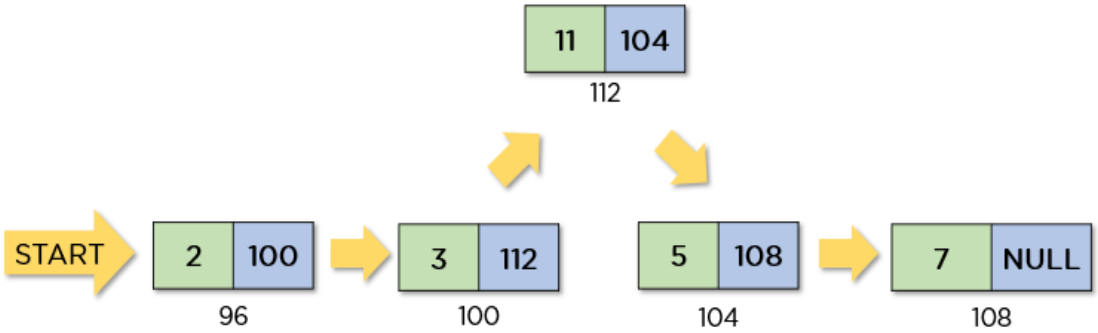
```
Created singly linked list:
2 3 5 7
--------------------------------
Process exited after 0.4099 seconds with return value 0
Press any key to continue . . .
```

What Operations Can You Perform on a Singly Linked List?

You can perform two operations on a singly linked list:

- Insertion
- Deletion

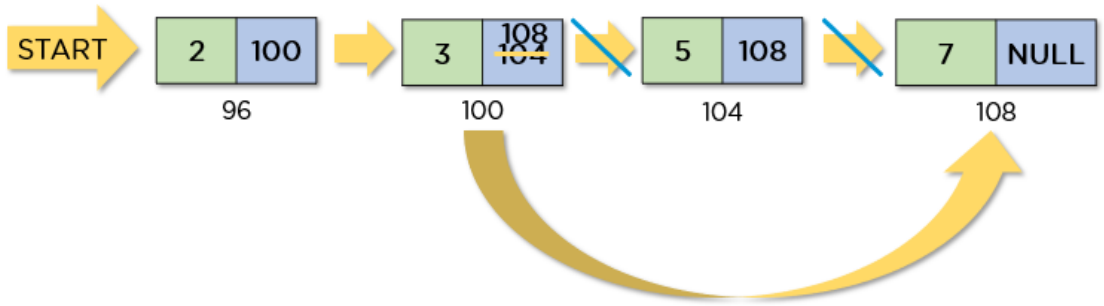
How to Insert a Node in a Singly Linked List?



You can insert a node at three different positions, they are:

- At the beginning
- At the end
- At a specific position after a node

How to Remove a Node From a Singly Linked List?



You can delete a node from 3 different locations, they are:

- From the beginning
- From the last
- From a specific position after a given node