**LINKED LISTS**

**UNIT –II**

**Introduction - Singly linked list - Representation of a linked list in memory - Operations on a singly linked list - Merging two singly linked lists into one list - Reversing a singly linked list - Applications of singly linked list to represent polynomial - Advantages and disadvantages of singly linked list - Circular linked list - Doubly linked list - Circular Doubly Linked List.**

**INTRODUCTION**

Linked lists and arrays are similar since they both store collections of data. Array is the most common data structure used to store collection of similar elements. Arrays are convenient to declare and provide the easy syntax to access any element by its index number. Once the array is set up, access to any element is convenient and fast. But arrays are suffer from the following limitations:

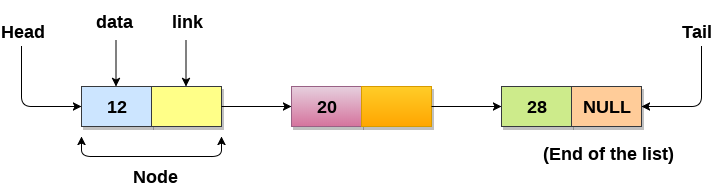
* Arrays have a fixed dimension. Once the size of an array is decided it cannot be increased or decreased during execution. For example, if we construct an array of 100 elements and then try to stuff more than 100 elements in it, our program may crash. On the other hand, if we use only 10 elements then the space for balance 90 elements goes waste.
* Array elements are always stored in contiguous memory locations. At times it might so happen that enough contiguous locations might not be available for the array that we are trying to create. Even though the total space requirement of the array can be met through a combination of non-contiguous blocks of memory, we would still not be allowed to create the array.
* Operations like insertion of a new element in an array or deletion of an existing element from the array are pretty tedious. This is because during insertion or deletion each element after the specified position has to be shifted one position to the right (in case of insertion) or one position to the left (in case of deletion).

Linked list overcomes all these disadvantages. A linked list can grow and shrink in size during its lifetime. In other words, there is no maximum size of a linked list. The second advantage of linked lists is that, as node (elements) are stored at different memory locations it hardly happens that we fall short of memory when required. The third advantage is that, unlike arrays, while inserting or deleting the nodes of the linked list, shifting of nodes is not required.

What is a Linked list?

Linked list is a very common data structure often used to store similar data in memory. While the elements of an array occupy contiguous memory locations, those of a linked list are not constrained to be stored in adjacent locations. The individual elements are stored “somewhere” in memory, rather like a family dispersed, but still bound together. The order of the elements is maintained by explicit links between them. \*\*-

* Linked List can be defined as collection of objects called nodes that are randomly stored in the memory.
* A node contains two fields i.e. data and pointer which contains the address of the next node in the memory.
* Linked list requires more memory compared to array because along with value it stores pointer to next node.
* Linked list are among the simplest and most common data structures. They can be used to implement other data structures like stacks,queues and symbolic expressions,etc…
* The last node of the list contains pointer to the null.
* Typically, a linked list, in its simplest form looks like the following



Few salient features

* + There is a pointer (called header) points the first element (also called node)
  + Successive nodes are connected by pointers.
  + Last element points to NULL.
  + Linked lists have efficient memory utilization. Here, memory is not pre allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
  + Linked lists are dynamic data structures. i.e., It can grow or shrink in size during execution of a program.
  + Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
  + Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node when compared to array.

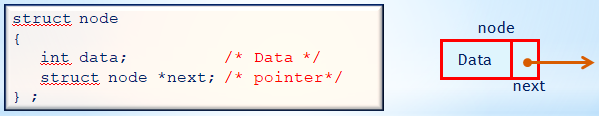
2. Searching a particular element in list is difficult and also time consuming.

**Defining a Node of a Linked List**

Each structure of the list is called a node, and consists of two fields:

* Item (or) data
* Address or pointer to the next node in the list

How to define a node of a linked list?



**Note:**

Such structures which contain a member field pointing to the same structure type are called self-referential structures.

**Initialize the head node**

Struct node \*head=NULL;

**Memory allocation for a node**

Struct node \*newnode=(struct node \*)malloc(sizeof(struct node));

**Creation of node**

Steps to create node:

1. Defining a structure of a node
2. Allocate memory dynamically.
3. Read the data into data field
4. Assign null value to its next

struct node

{

int data;

struct node \*next; //defining node structure

};

struct node \*head=null,\*newnode; //initialize the head node

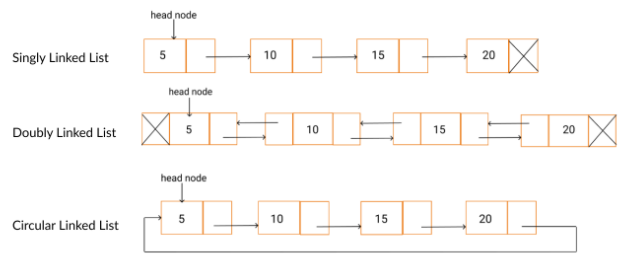
newnode=(struct node \*)malloc(sizeof(struct node)) // memory allocation for a newnode

newnode->data =data // Read data

newnode-> next =null;

There are 3 different implementations of Linked List available, they are:

* Singly Linked List
* Doubly Linked List
* Circular Linked List



Singly linked list

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we cannot traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.



In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

## Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

* Traversing the list
* Inserting a node into the list
* Deleting a node from the list
* Copying the list to make a duplicate of it
* Merging the linked list with another one to make a larger list
* Searching for an element in the list.

|  |  |  |
| --- | --- | --- |
| SN | Operation | Description |
| 1 | Insertion at beginning | It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list. |
| 2 | Insertion at end of the list | It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario. |
| 3 | Insertion after specified node | It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. . |

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

# Insertion in singly linked list at beginning

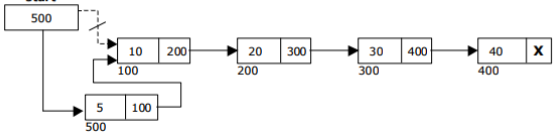
Inserting a node into a single linked list There are various positions where a node can be inserted:

1. Inserting at the front (as a first element)
2. Inserting at the end (as a last element)
3. Inserting at any other position.

**Inserting a node at the front of a single linked list**

Inserting a new element into a singly linked list at beginning is quite simple. We just need to make a few adjustments in the node links adjustments to make the new node as the first node of the list. There are the following steps which need to be followed in order to insert a new node in the list at beginning.

* Allocate the space for the new node and store data into the data part of the node.
* Make the link part of the new node pointing to the existing first node of the list.
* At the last, we need to make the new node as the first node of the list.



**Fig: Inserting a node at the front of a single linked list**

The algorithm lnsertatfront: is used to insert a node at the front of a single linked list.

**Algorithm INSERT\_FRONT(head,X)**

**Input:** Head is the pointer to the first node and X is the data of the node to be inserted.

**Output:** A singly linked list with newly inserted node in the front of the list.

1. newnode = create newnode // Create a newnode and store its pointer in newnode

newnode->data =data // Read data

newnode-> next =null;

1. if(head!= NULL)

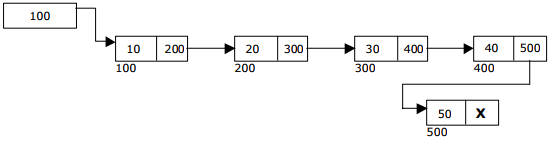
newnode->next=head

head=newnode

1. endif
2. stop

**Inserting a node at the end of a single linked list**

* We need to declare a temporary pointer temp in order to traverse through the list. **temp** is made to point the first node of the list. At the end of the loop, the temp will be pointing to the last node of the list. Now, allocate the space for the new node, and assign the item to its data part. Since, the new node is going to be the last node of the list hence, the next part of this node needs to be pointing to the null. We need to make the next part of the temp node (which is currently the last node of the list) to null .



**Fig: Inserting a node at the end of a single linked list**

The algorithm Insert\_End is used to insert a node at the end of a single linked list

**Algorithm INSERT\_END(head,X)**

**Input:** Head is the pointer to the first node and X is the data of the node to be inserted.

**Output:** A singly linked list with newly inserted node at the end of a linked list

* 1. newnode = create newnode // Create a newnode and store its pointer in newnode

newnode->data =data // Read data

newnode-> next =null

2. temp = head

while (temp -> next != NULL) do

temp = temp -> next

Endwhile

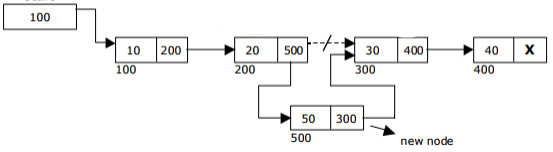
temp->next = newnode

newnode->next=null

stop

**Inserting a node into a single linked list at any position in the list**

In order to insert an element after the specified number of nodes into the linked list, we need to skip the desired number of elements in the list to move the pointer at the position after which the node will be inserted.



**Fig:Inserting a node into a single linked list at any position in the list**

The algorithm InsertAnyPosition is used to insert a node into a single linked list at any position in the list.

Algorithm INSERT\_ANYPOSITION(Head,X,pos)

1. temp=head;

while(i<pos)

temp=temp->next;

i++

Endwhile

* 1. newnode = create newnode // Create a newnode and store its pointer in newnode

newnode->data =data // Read data

newnode->next=temp->next

temp->next=newnode