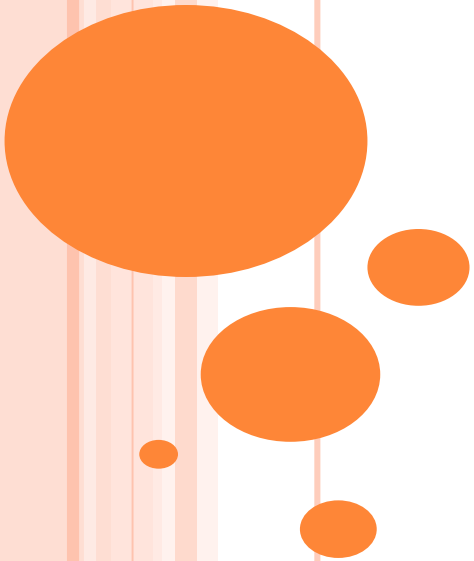


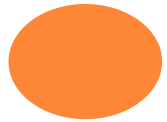
LINKED LIST

What is Data Structure?

- Whenever we want to work with a large amount of data, then organizing that data is very important.
- If that data is not organized effectively, it is very difficult to perform any task on that data.
- If it is organized effectively then any operation can be performed easily on that data.
- Linear Data Structures - If a data structure organizes the data in sequential order, then that data structure is called a Linear Data Structure.
- For example – Arrays, Linked List, Stack and queue

Linked List

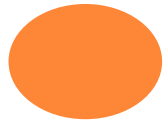
- When we want to work with an unknown number of data values, we use a linked list data structure to organize that data.
- A linked list are connected together via links.
- Linked list is a series of connected "nodes" that contains the "address" of the next node. Each node can store a data point which may be a number, a string or any other type of data.
- Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.



Why Linked List?

- Arrays can be used to store linear data of similar types, but arrays have following limitations.

1. The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2. Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.
3. Deletion is also expensive with arrays unless some special techniques are used.



Advantage

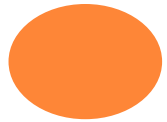
- Memory utilization is efficient as it's allocated when we add new elements to a list and list size can increase/decrease as required.
- They are useful when the size of a list is unknown and changes frequently.
- It uses a node that stores data and a pointer to the next node.

ARRAYS VS LINKED LISTS

Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]

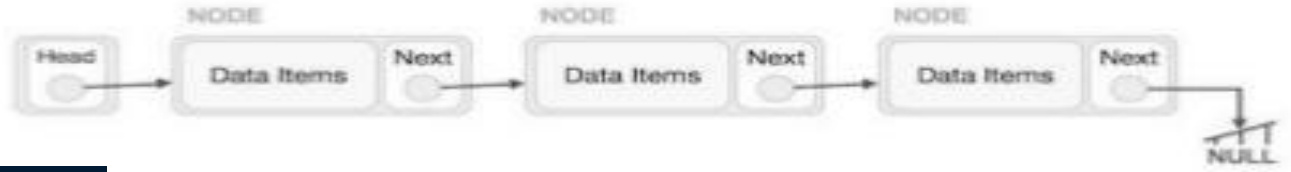
Linked List

- It is the most commonly used data structure used to store similar type of data in memory.
- The elements of a linked list are not stored in adjacent memory locations as in arrays.
- It is a linear collection of data elements, called nodes, where the linear order is implemented by means of pointers.
- In a linear or single-linked list, a node is connected to the next node by a single link.
- A node in this type of linked list contains two types of fields
  - data: which holds a list element
  - Link: which stores a link (i.e. pointer) to the next node in the list.

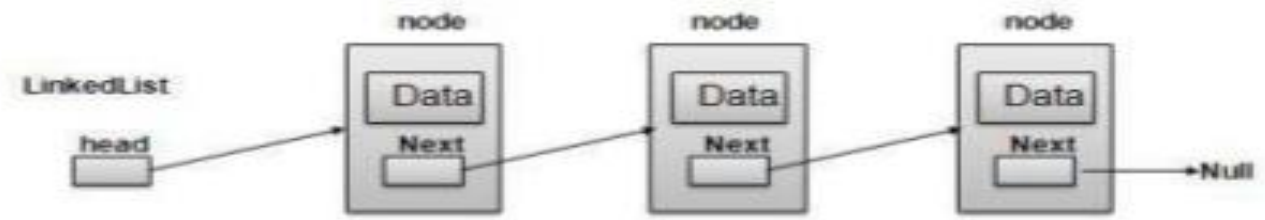


Linked List

- The structure defined for a single linked list is implemented as follows:
  - The structure declared for linear linked list holds two members
  - An integer type variable ‘data’ which holds the elements and
  - Another type ‘node’, which has next, which stores the address of the next node in the list.




```
struct Node
{
        int info;
        struct Node* next;
};
```

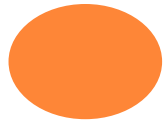


Linked List Types: Node and Pointer

- Node :The type for the nodes which will make up the body of the list.
- These are allocated in the heap. Each node contains a single client data element and a pointer to the next node in the list.
- Type: struct node struct node

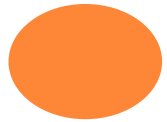
```
{   int data;
    struct node* next; }; //called as self referen
```

- Node Pointer :The type for pointers to nodes.
- This will be the type of the head pointer and the .next fields inside each node. In C and C++, no separate type declaration is required since the pointer type is just the node type followed by a '*'.
- Type: struct node*



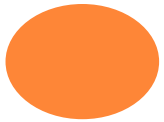
Properties of Linked list

- The nodes in a linked list are not stored contiguously in the memory
- You don’t have to shift any element in the list
- Memory for each node can be allocated dynamically whenever the need arises.
- The size of a linked list can grow or shrink dynamically.



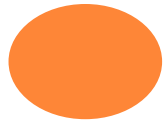
Operations on Linked List

- Creation: This operation is used to create a linked list
- Insertion / Deletion :
  - At/From the beginning of the linked list
  - At/From the end of the linked list
  - At/From the specified position in a linked list
- Traversing: Traversing may be either forward or backward
- Searching: Finding an element in a linked list
- Concatenation: The process of appending second list to the end of the first list.



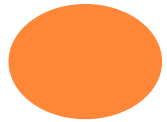
Types of Linked List

- Singly Linked List
- Doubly linked list
- Circular linked list
- Circular doubly linked list



Singly Linked List

- A singly linked list is a dynamic data structure which may grow or shrink, and growing and shrinking depends on the operation made.
- In this type of linked list each node contains two fields one is data field which is used to store the data items and another is next field that is used to point the next node in the list.
- **Linked list is known as simple or singly linked list because a simple linked list can be traversed in only one direction from head to the last node.**



Creating a Linked List

- The head pointer is used to create and access unnamed nodes.
- The above statement obtains memory to store a node and assigns its address to head which is a pointer variable.

```
struct Node
{
        int info;
        struct Node* next;
};
typedef struct Node NodeType;
NodeType* head;
head=(NodeType *) malloc (sizeof( NodeType ) );
```

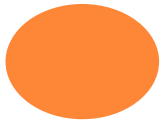
Creating a Node

- To create a new node, we use the malloc function to dynamically allocate memory for the new node.
- After creating the node, we can store the new item in the node using a pointer to that node.
- Note that p is not a node; instead it is a pointer to a node.

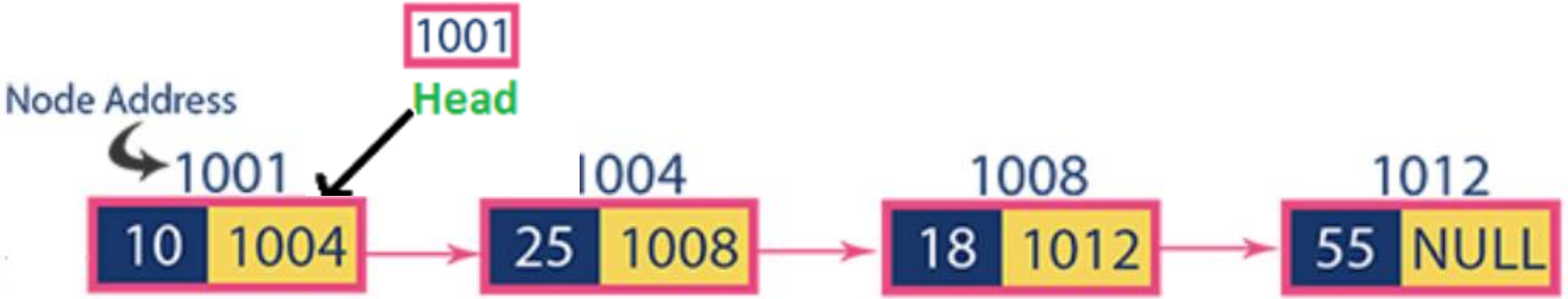
```
Nodetype *p;
p=(NodeType *) malloc (sizeof( NodeType ) );
p->info=50;
p->next = NULL;
```

Representation

- Every "Node" contains two fields, data field, and the next field. The data field is used to store actual value of the node and next field is used to store the address of next node in the sequence.
- The first node is called the head.
- If the linked list is empty, then the value of the head is NULL.

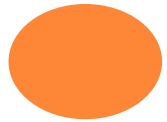



Stores Address of next node
Data
Link
Stores Actual value


1001
Node Address
Head
1001
1004
1008
1012
10
1004
25
1008
18
1012
55
NULL

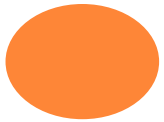
- We wrap both the data item and the next node reference in a struct as:

```
struct node
{
int data;
struct node *next; // self-referential structure
};
```



Pointers to Structures as structure members

- Any pointer can be a member of a structure.
- This includes a pointer that points to a structure.
- A pointer structure member that points to the same type of structure is also permitted. This is called a **self-referential structure.**




HEAD
1
Next
2
Next
3
Next
NULL

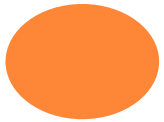
Operations on Single Linked List

- The following operations are performed on a Single Linked List

Insertion

Deletion

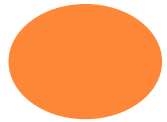
Display



Before we implement actual operations, first we need to set up an empty list.

First, perform the following steps before implementing actual operations.

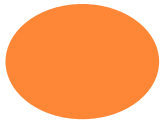
- Step 1 - Include all the header files which are used in the program.
- Step 2 - Declare all the user defined functions.
- Step 3 - Define a Node structure with two members data and next
- Step 4 - Define a Node pointer 'head' and set it to NULL.



Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

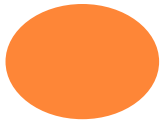
- Inserting At Beginning of the list
- Inserting At End of the list
- Inserting At Specific location in the list



Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

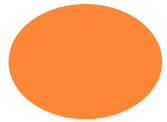
- Step 1 - Create a newNode with given value.
- Step 2 - Check whether list is Empty (head == NULL)
- Step 3 - If it is Empty then, set newNode→next = NULL and head = newNode.
- Step 4 - If it is Not Empty then, set newNode→next = head and head = newNode.



Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

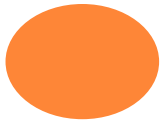
- Step 1 - Create a newNode with given value and newNode → next as NULL.
- Step 2 - Check whether list is Empty (head == NULL).
- Step 3 - If it is Empty then, set head = newNode.
- Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.
- Step 5 - Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).
- Step 6 - Set temp → next = newNode.



Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode** → **next** = **NULL** and **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1** → **data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp** is reached to last node or not. If it is reached to last node then display '**Given node is not found in the list!!! Insertion not possible!!!**' and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** Finally, Set '**newNode** → **next** = **temp** → **next**' and '**temp** → **next** = **newNode**



Displaying a Single Linked List

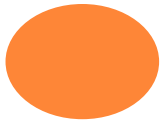
We can use the following steps to display the elements of a single linked list...

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty then, display 'List is Empty!!!' and terminate the function.
- Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
- Step 4 - Keep displaying temp → data with an arrow (--->) until temp reaches to the last node
- Step 5 - Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).

Deletion

The deletion operation can be performed in three ways :

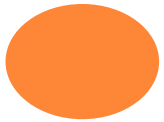
- Deleting from Beginning of the list
- Deleting from End of the list
- Deleting a Specific Node



Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

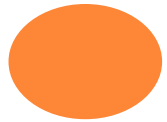
- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
- Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
- Step 4 - Check whether list is having only one node (temp → next == NULL)
- Step 5 - If it is TRUE then set head = NULL and delete temp (Setting Empty list conditions)
- Step 6 - If it is FALSE then set head = temp → next, and delete temp.



Deleting from End of the list

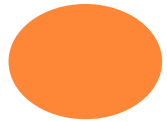
We can use the following steps to delete a node from end of the single linked list.

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
- Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
- Step 4 - Check whether list has only one Node (temp1 → next == NULL)
- Step 5 - If it is TRUE. Then, set head = NULL and delete temp1. And terminate the function. (Setting Empty list condition)
- Step 6 - If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until it reaches to the last node in the list. (until temp1 → next == NULL)
- Step 7 - Finally, Set temp2 → next = NULL and delete temp1.



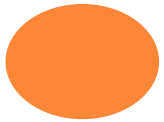
Deleting a Specific Node from the list

- We can use the following steps to delete a specific node from the single linked list...
- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
- Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
- Step 4 - Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.
- Step 5 - If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.



Deleting a Specific Node from the list-contd

- Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- Step 7 - If list has only one node and that is the node to be deleted, then set head = NULL and delete temp1 (free(temp1)).
- Step 8 - If list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == head).
- Step 9 - If temp1 is the first node then move the head to the next node (head = head → next) and delete temp1.
- Step 10 - If temp1 is not first node then check whether it is last node in the list (temp1 → next == NULL).
- Step 11 - If temp1 is last node then set temp2 → next = NULL and delete temp1 (free(temp1)).
- Step 12 - If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).



TRAVERSING

- If the traversal start from the last node towards the first node , it is called back word traversing.
- Searching operation is a process of accessing the desired node in the list.
- We start searching node –by-node and compare the data of the node with the key.
- Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.
- When temp is NULL, we know that we have reached the end of linked list so we get out of the while loop.

```
struct node *temp = head;
 printf("\n\nList elements are - \n");
while(temp != NULL) {
 printf("%d --->",temp->data);
 temp = temp->next; }
```

REVERSING , SORTING

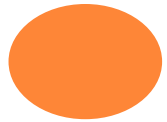
- REVERSING THE LINKS :

  Reversing means the last node becomes the first node and the first becomes the last.

- SORTING OF A LINKED LIST :
  - Sorting means to arrange the elements either in ascending or descending order.
  - To sort the elements of a linked list we use any of the standard sorting algorithms for carrying out the sorting.
  - While performing the sorting, when it is time to exchange two elements, we can adopt any of the following two strategies

    1)Exchange the data part of the two nodes, keeping the links intact.

    2) Keep the data in the nodes intact. Simply readjust the links such that effectively the order of the nodes changes



SEARCHING AN ITEM IN A LINKED LIST

- Let *head be the pointer to first node in the current list

1. If head==Null

   Print “Empty List”

2. Else, enter an item to be searched as key

3. Set temp==head

4. While temp!=Null

   If (temp->info == key)

   Print “search success”

   temp=temp->next;

5. If temp==Null

- Print “Unsuccessful search”