UNIT-IV

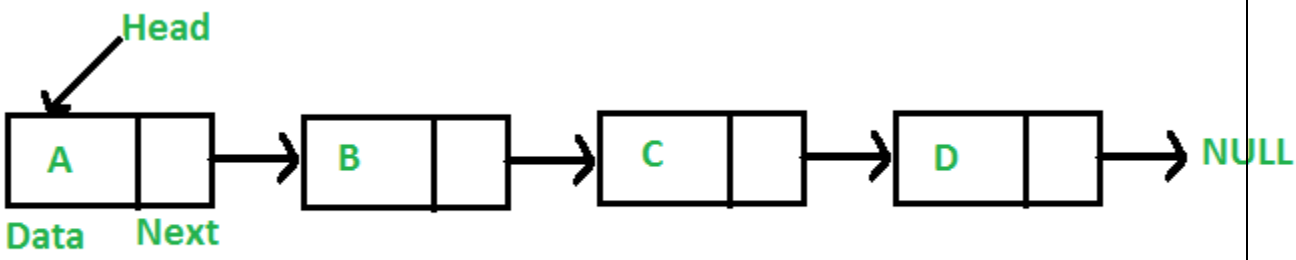
LINKED LISTS

TOPICS:-

1. **Singly linked list**
2. **dynamically linked stacks and queues**
3. **polynomials using singly linked lists, using circularly linked lists,**
4. **single linked lists and its operations - insertion, deletion and searching**
5. **doubly linked lists and its operations**
6. **circular linked lists and its operations.**

TOPIC 1: LINKED LIST AND ITS TYPES

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list. A linked list is formed when many such nodes are linked together to form a chain. Each node points to the next node present in the order. The first node is always used as a reference to traverse the list and is called HEAD. The last node points to NULL.

Declaring a Linked list :

In C language, a linked list can be implemented using structure and pointers .

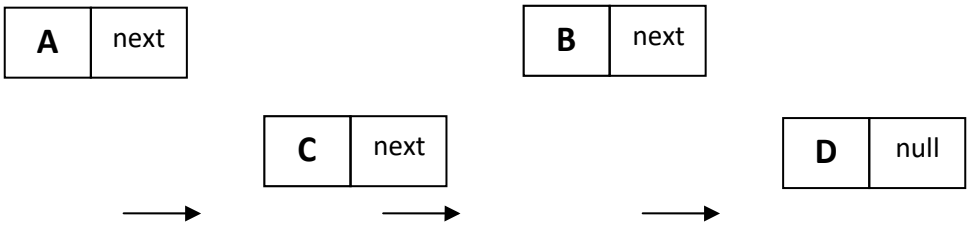
```
struct LinkedList{
   int data;
   struct LinkedList *next;
};
```

The above definition is used to create every node in the list. The data field stores the element and the next is a pointer to store the address of the next node.

Types of linked list

Singly Linked List

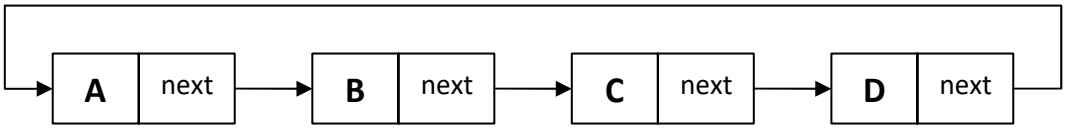
- It is basic type of linked list.
- Each node contains data and pointer to next node.
- Last node's pointer is null.
- Limitation of singly linked list is we can traverse only in one direction, forward direction.



Singly Linked List

Circular Linked List

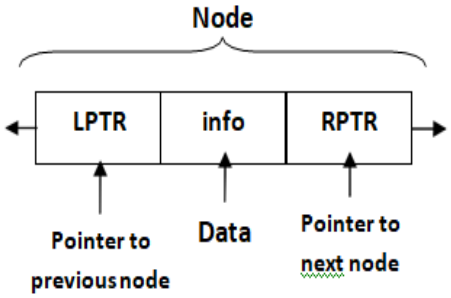
- Circular linked list is a singly linked list where last node points to first node in the list.
- It does not contain null pointers like singly linked list.
- We can traverse only in one direction that is forward direction.
- It has the biggest advantage of time saving when we want to go from last node to first node, itdirectly points to first node.
- A good example of an application where circular linked list should be used is a timesharing problemsolved by the operating system.



Circular Linked List

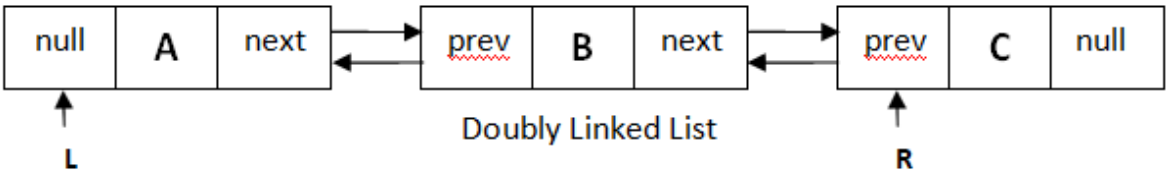
Doubly Linked list

- Each node of doubly linked list contains data and two pointers to point previous (LPTR) and next (RPTR) node.



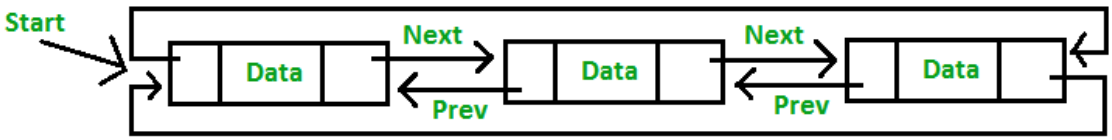

```
// C Structure to represent a node
struct node
{
      int info
      struct node *lptr;
      struct node *rptr;
};
```

- Main advantage of doubly linked list is we can traverse in any direction, forward or reverse.
- Other advantage of doubly linked list is we can delete a node with little trouble, since we have pointers to the previous and next nodes. A node on a singly linked list cannot be removed unless we have the pointer to its predecessor.
- Drawback of doubly linked list is it requires more memory compared to singly linked list because we need an extra pointer to point previous node.
- L and R in image denotes left most and right most nodes in the list.
- Left link of L node and right link of R node is NULL, indicating the end of list for each direction.



Circular Double 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.



Advantages:

- 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).

Disadvantages

- 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.

----*****----

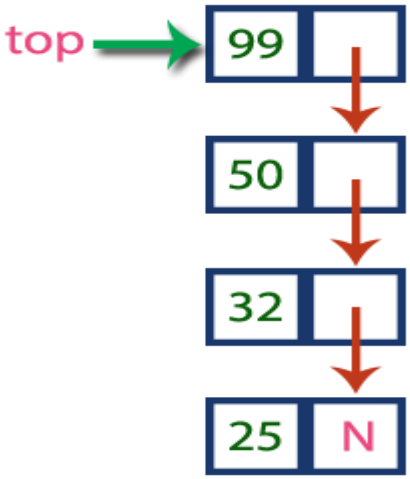
TOPIC 2: DYNAMICALLY LINKED STACKS AND QUEUES

Implementing Stack functionalities using Linked List

The major problem with the stack implemented using an array is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.

Example



In the above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

Stack Operations using Linked List

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3 -** If it is **Empty**, then set **newNode** → **next** = **NULL**.
- **Step 4 -** If it is **Not Empty**, then set **newNode** → **next** = **top**.
- **Step 5 -** Finally, set **top** = **newNode**.

pop( ) - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- **Step 1 -** Check whether **stack** is **Empty** (**top == NULL**).

- **Step 2 -** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- **Step 3 -** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4 -** Then set '**top = top → next**'.
- **Step 5 -** Finally, delete '**temp**'. (**free(temp)**).

display( ) - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

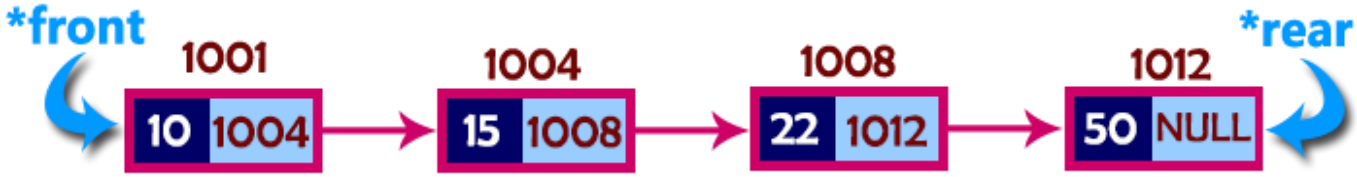
- **Step 1 -** Check whether stack is **Empty** (**top** == **NULL**).
- **Step 2 -** If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.
- **Step 4 -** Display '**temp → data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp → next** != **NULL**).
- **Step 5 -** Finally! Display '**temp → data** ---> **NULL**'.

Implementing Queue functionalities using Linked List

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

Operations

To implement queue using linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

- **Step 1 -** Create a **newNode** with given value and set '**newNode → next**' to **NULL**.
- **Step 2 -** Check whether queue is **Empty** (**rear** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **rear → next** = **newNode** and **rear** = **newNode**.

deQueue( ) - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

- **Step 1 -** Check whether **queue** is **Empty** (**front == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function
- **Step 3 -** If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
- **Step 4 -** Then set '**front** = **front → next**' and delete '**temp**' (**free(temp)**).

display( ) - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

- **Step 1 -** Check whether queue is **Empty** (**front** == **NULL**).
- **Step 2 -** If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.
- **Step 4 -** Display '**temp → data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp → next** != **NULL**).
- **Step 5 -** Finally! Display '**temp → data** ---> **NULL**'.

----****----

TOPIC 4: SINGLE LINKED LISTS AND ITS OPERATIONS

Single linked list is a sequence of elements in which every element has link to its next element in the sequence.

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.
- Step 5 - Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

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

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. 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'

Deletion

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

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. 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.
- 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)).

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).

---***---

TOPIC 5: DOUBLE LINKED LIST AND ITS OPERATIONS

Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

Operations on Double Linked List

In a double linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Insertion

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

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. 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 double linked list...

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

Inserting At End of the list

We can use the following steps to insert a new node at end of the double 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 assign NULL to newNode → previous and newNode to head.
- 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 - Assign newNode to temp → next and temp to newNode → previous.

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 double 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, assign NULL to both newNode → previous & newNode → next and set newNode to head.
- Step 4 - If it is not Empty then, define two node pointers temp1 & temp2 and initialize temp1 with head.
- Step 5 - Keep moving the temp1 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 temp1 is reached to the last node. If it is reached to the last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp1 to next node.
- Step 7 - Assign temp1 → next to temp2, newNode to temp1 → next, temp1 to newNode → previous, temp2 to newNode → next and newNode to temp2 → previous.

Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double 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 → previous is equal to temp → next)
- Step 5 - If it is TRUE, then set head to NULL and delete temp (Setting Empty list conditions)
- Step 6 - If it is FALSE, then assign temp → next to head, NULL to head → previous and delete temp.

Deleting from End of the list

We can use the following steps to delete a node from end of the double 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 has only one Node (temp → previous and temp → next both are NULL)
- Step 5 - If it is TRUE, then assign NULL to head and delete temp. And terminate from the function. (Setting Empty list condition)
- Step 6 - If it is FALSE, then keep moving temp until it reaches to the last node in the list. (until temp → next is equal to NULL)
- Step 7 - Assign NULL to temp → previous → next and delete temp.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double 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 - Keep moving the temp until it reaches to the exact node to be deleted or to the last 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 fuction.
- 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 which is to be deleted then set head to NULL and delete temp (free(temp)).
- Step 8 - If list contains multiple nodes, then check whether temp is the first node in the list (temp == head).
- Step 9 - If temp is the first node, then move the head to the next node (head = head → next), set head of previous to NULL (head → previous = NULL) and delete temp.

- Step 10 - If temp is not the first node, then check whether it is the last node in the list (temp → next == NULL).
- Step 11 - If temp is the last node then set temp of previous of next to NULL (temp → previous → next = NULL) and delete temp (free(temp)).
- Step 12 - If temp is not the first node and not the last node, then set temp of previous of next to temp of next (temp → previous → next = temp → next), temp of next of previous to temp of previous (temp → next → previous = temp → previous) and delete temp (free(temp)).

Displaying a Double Linked List

We can use the following steps to display the elements of a double 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 - Display 'NULL <--- '.
- Step 5 - Keep displaying temp → data with an arrow (<===>) until temp reaches to the last node
- Step 6 - Finally, display temp → data with arrow pointing to NULL (temp → data ---> NULL).

---***---

TOPIC 6: CIRCULAR LINKED LIST AND ITS OPERATIONS

A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.

Operations

In a circular linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Before we implement actual operations, first we need to setup 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.
- Step 5 - Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

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

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. 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 circular 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 head = newNode and newNode→next = head .
- 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 (until 'temp → next == head').
- Step 6 - Set 'newNode → next =head', 'head = newNode' and 'temp → next = head'.

Inserting At End of the list

We can use the following steps to insert a new node at end of the circular 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 head = newNode and newNode → next = head.
- 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 == head).
- Step 6 - Set temp → next = newNode and newNode → next = head.

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 circular 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 head = newNode and newNode → next = head.
- 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 the 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 - If temp is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
- Step 8 - If temp is last node then set temp → next = newNode and newNode → next = head.
- Step 8 - If temp is not last node then set newNode → next = temp → next and temp → next = newNode.

Deletion

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular 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 both 'temp1' and 'temp2' with head.
- Step 4 - Check whether list is having only one node (temp1 → next == head)
- Step 5 - If it is TRUE then set head = NULL and delete temp1 (Setting Empty list conditions)
- Step 6 - If it is FALSE move the temp1 until it reaches to the last node. (until temp1 → next == head)
- Step 7 - Then set head = temp2 → next, temp1 → next = head and delete temp2.

Deleting from End of the list

We can use the following steps to delete a node from end of the circular 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 == head)

- Step 5 - If it is TRUE. Then, set head = NULL and delete temp1. And terminate from 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 temp1 reaches to the last node in the list. (until temp1 → next == head)
- Step 7 - Set temp2 → next = head and delete temp1.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular 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.
- Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node (temp1 → next == head)
- 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 set temp2 = head and keep moving temp2 to its next node until temp2 reaches to the last node. Then set head = head → next, temp2 → next = head and delete temp1.
- Step 10 - If temp1 is not first node then check whether it is last node in the list (temp1 → next == head).
- Step 1 1- If temp1 is last node then set temp2 → next = head 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)).

Displaying a circular Linked List

We can use the following steps to display the elements of a circular 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 head → data.

---***---

TOPIC 3: POLYNOMIALS USING SINGLE AND CIRCULAR LINKED LIST

Polynomials and Sparse Matrix are two important applications of arrays and linked lists. A polynomial is composed of different terms where each of them holds a coefficient and an exponent.

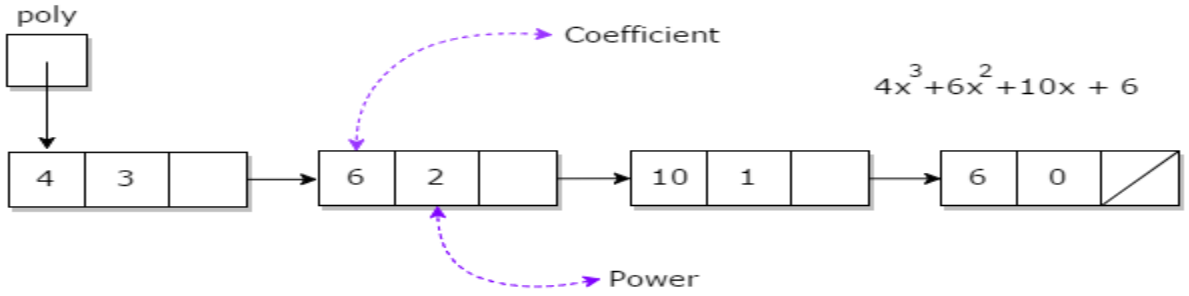
A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + ... + jx + k$), where a, b, c ...., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

- one is the coefficient
- other is the exponent

Example:

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.



Addition of two polynomials

Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients who have same variable powers.

Example:

Input:

1st number = $5x^2 + 4x^1 + 2x^0$

2nd number = $-5x^1 - 5x^0$

Output:

$5x^2-1x^1-3x^0$

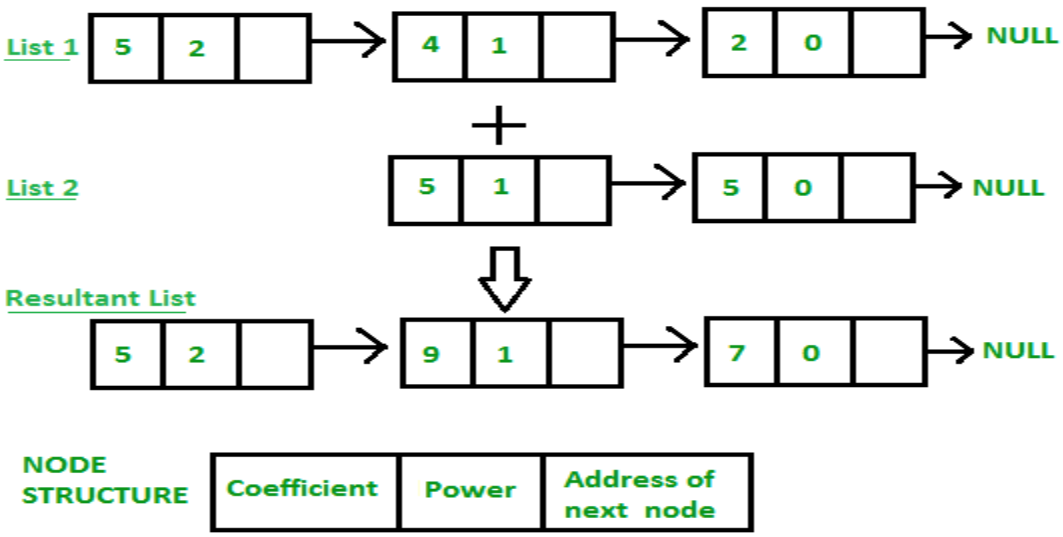
Input:

1st number = $5x^3 + 4x^2 + 2x^0$

2nd number = 5x^1 - 5x^0

Output:

$5x^3 + 4x^2 + 5x^1 - 3x^0$



Polynomials using circular linked list

1. Create two circular linked lists, where each node will consist of the coefficient, power of **x**, power of **y** and pointer to the next node.
2. Traverse both the polynomials and check the following conditions:
   - If power of **x** of 1st polynomial is greater than power of **x** of second polynomial then store node of first polynomial in resultant polynomial and increase counter of polynomial **1**.
   - If power of **x** of 1st polynomial is less than power of **x** of second polynomial then store the node of second polynomial in resultant polynomial and increase counter of polynomial **2**.
   - If power of **x** of 1st polynomial is equal to power of **x** of second polynomial and power of **y** of 1st polynomial is greater than power of **y** of 2nd polynomial then store the node of first polynomial in resultant polynomial and increase counter of polynomial **1**.
   - If power of **x** of 1st polynomial is equal to power of **x** of second polynomial and power of **y** of 1st polynomial is equal to power of **y** of 2nd polynomial then store the sum of coefficient of both polynomial in resultant polynomial and increase counter of both polynomial **1** and polynomial **2**.
3. If there are nodes left to be traversed in 1st polynomial or in 2nd polynomial then append them in resultant polynomial.
4. Finally, print the resultant polynomial.

----**** THE END****----