CHAPTER 4: Static and Dynamic List

BY: Er. Krishna Khadka

CONTENT

LIST

- Introduction
- Static and dynamic List Structure
- Array Implementation of Lists
- Queue as a List

LINKED LIST

- Introduction
- Linked list as an ADT
- Dynamic implementation
- Insertion and deletion of node to and from a list
- Insertion and deletion and before nodes
- Linked stack and queues
- Doubly linked list and its advantages

LIST- INTRODUCTION

- A list is a collection of homogeneous set of elements or objects.
- If the size of list is fixed at compile time and do not grow or shrink at runtime then it is called static list however if the size of the list is not fixed at compile time and grow and shrink at runtime then it is called dynamic list.
- A list is said to be empty when it contains no elements
- The number of elements currently stored is called the length of the list.
- The beginning of the list is called head and the end of the list is called the tail.

STATIC IMPLEMENTATION OF LIST

- Static implementation can be implemented using arrays.
- It is very simple method but it has static implementation.
- Once a size is declared, it cannot be change during the program execution.
- It is also not efficient for memory utilization.
- When array is declared, memory allocated is equal to the size of the array.
- The vacant space of array also occupies the memory space.
- In this cases, if we store fewer arguments than declared, the memory is wasted and if more elements are stored than declared, array cannot be expanded.
- It is suitable only when exact numbers of elements are to be stored.

SOME COMMON OPERATIONS PERFORMED ON STATIC LIST

- Creating of an array
- Inserting new element at required position
- Deletion of any element
- Modification of any element
- Traversing of an array
- Merging of arrays

LIST (ARRAY) AS AN ADT

- Let A be an LIST of array implementation and it has n elements then it satisfied the following operations:
  - CREATE (A) : Create an array A
  - INSERT (A,X): Insert an element X into an array A in any location
  - DELETE (A,X) : Delete an element X from an array A
  - MODIFY (A,X,Y) : Modify element X by Y of an array A
  - TRAVERSE (A) : Access all elements of an array A
  - MERGE (A,B) : Merging elements of A and B into a third array C.

Thus by using a one dimensional array we can perform above operations thus an array acts as an ADT.

INSERTION OF AN ELEMENT IN ONE-DIMENSIONAL ARRAY

- **Insertion at the end of an array:**
  - Providing the memory space allocated for the array enough to accommodate the additional; element can easily do insertion at the end of an array.
- **Insertion at the required position**
  - For inserting the element at required position, element must be moved downwards to new locations
  - To accommodate the new element and keep the order of the elements.
  - For inserting an element into a linear array **insert(a, len, pos, num)** where **a** is a linear array, **len** be total number of elements with an array, **pos** is the position at which number **num** will be inserted.

INSERTION OF AN ELEMENT IN ONE-DIMENSIONAL ARRAY

Algorithm to insert new element in a list:

1. [initialize the value of i] set **i=len**
2. Repeat for **i= len** down to **pos**

   [shift the elements down by 1 position]

   Set **a[i+1] = a [i]**

   [end of loop]
3. [insert the element at required position]

   Set **a[pos] = num**
4. [reset len] set **len=len+1**
5. Display the new list of arrays
6. end

DELETION OF AN ELEMENT FROM ONE-DIMENSIONAL ARRAY

- Deleting an element at the end of an array presents no difficulties, but deleting element somewhere in the middle of the array would require to shift all the elements to fill the space emptied by the deletion of the element, then the element following it were moved left by one location.
- **num** is the item deleted and **MAX** is the size of array. **pos** is the position from where the data item is deleted.

Algorithm:

1. Set **num = a[pos]**
2. Repeat for **j= pos to MAX-1**
   a. Shift elements 1 position upwards
   b. Set **a[j] = a[j+1]**
3. Reset **MAX=MAX-1**
4. Display the new list of element of array
5. end

DYNAMIC IMPLEMENTATION OF LIST

- In static implementation of memory allocation, we cannot alter (increase or decrease) the size of an array and the memory allocation is fixed.
- So we have to adopt an alternative strategy to allocate memory only when it is required.
- There is a special data structure called linked list that provides a more flexible storage system and it does not required the use of array.
- The advantage of a linked-list over an array occurs when it is necessary to insert or delete an element in the middle of a group of other elements.

LINKED LIST

- A linked list is a linear collection of specially designed data structure, called **nodes,** linked to one another by means of **pointer.**
- Each node is divided into 2 parts: the first part contains information of the element and the second part contains address of next node in the link list.

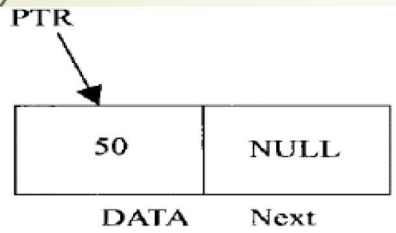


Fig1: Node

Fig2: Linked List

- The left part of each node contains the data items and the right part represents the address of the next node.
- The next pointer of the last node contains a special value, called the NULL pointer, which does not point to any address of the node.

LINKED LIST

- That is NULL pointer indicates the end of linked list.
- START pointer will hold the address of the 1st node in the list START = NULL if there is no list (i.e. NULL list or empty list)

REPRESENTATION OF LINKED LIST:

Suppose we want to store a list of integer numbers using linked list. Then it can be schematically represented as

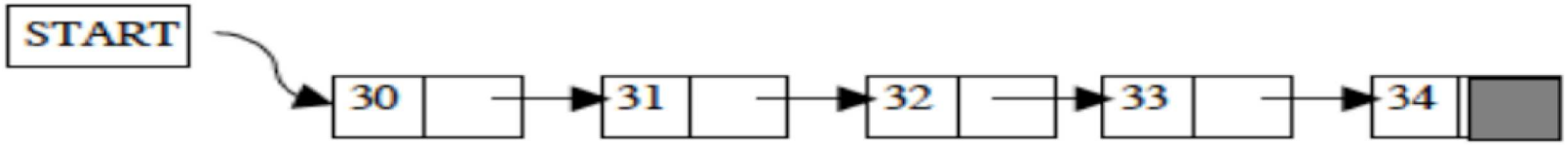


Figure 5: Representation of Linked List

We can declare linear linked list as follows

```
struct Node{
   int data; //instead of data we also use info
   struct Node *Next; //instead of Next we also use link
};
typedef struct Node *NODE;
```

ADVANTAGES AND DISADVANTAGES OF LINKED LIST

ADVANTAGES:

- Link list are dynamic data structure. That is they can grow or shrink during the execution of a program.
- Efficient memory utilization: in linked list memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated when it is not needed.
- Insertion and deletion are easier and efficient. Linked list provides flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
- Many complex applications can be easily carried out with linked list.

DISADVANTAGES:

- More memory: to store an integer number, a node with integer data and address field is allocated. That is more memory space is needed.
- Access to any arbitrary data item is little bit cumbersome and also time consuming.

OPERATIONS ON LINKED LIST

The primitive operations performed on linked list is as follows:

- **Creation :**
  - It is used to create a linked list.
  - Once a linked list is created with one node, insertion operation can be used to add more elements in a node.
- **Insertion:**
  - It is used to insert a new node at any specified location in the linked list.
  - A new node may be inserted
    - At the beginning of the linked list
    - At the end of the linked list
    - At any specified position in between in a linked list
- **Deletion:**
  - it is used to delete an item (or node) from the linked list.

OPERATIONS ON LINKED LIST

- A node may be deleted from the
  - Beginning of a linked list
  - End of a linked list
  - Specified location of the linked list
- **Traversing:**
  - It is the process of going through all the nodes from one end to another end of a linked list.
- **Concatenation:**
  - It is the process appending the second list to the end of the first list.
- **Searching:**
  - It is the process of finding the location of searched item.

TYPES OF LINKED LIST

Following are the types of Linked list depending upon the arrangements of the nodes.

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

SINGLY LINKED LIST

- All the nodes in a singly linked list are arranged sequentially by linking with pointer.
- A singly linked list can grow or shrink, because it is a dynamic data structure.
- The following figure explains the different operations on singly linked list.

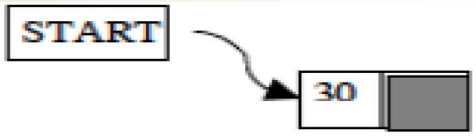


Fig4: Create a node with Data (30)

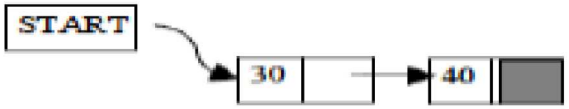


Fig5: Insert a node with Data (40) at the end

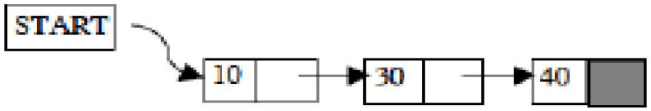


Fig6: Insert a node with Data (10) at the beginning position

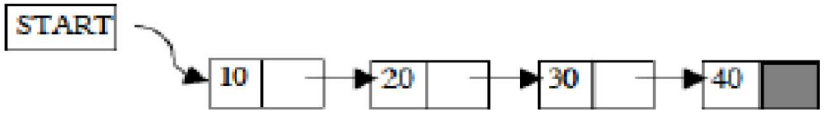


Fig7: Insert a node with Data (20) at 2nd

SINGLY LINKED LIST

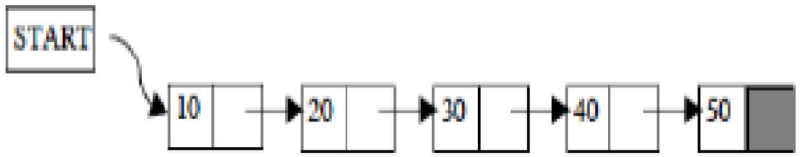


Fig8: Insert a node with Data (50) at the end

Output → 10, 20, 30, 40, 50

Fig9: Traversing the nodes from left to right

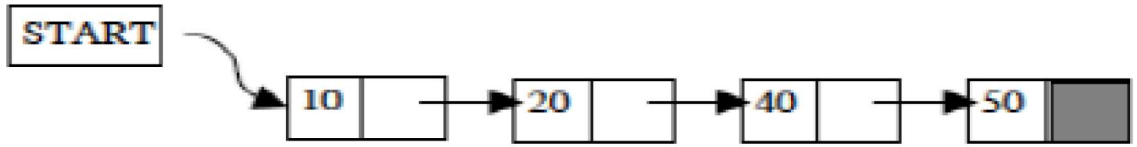


Fig10: Delete 3rd node from the

ALGORITHM FOR INSERTION A NODE IN SLL

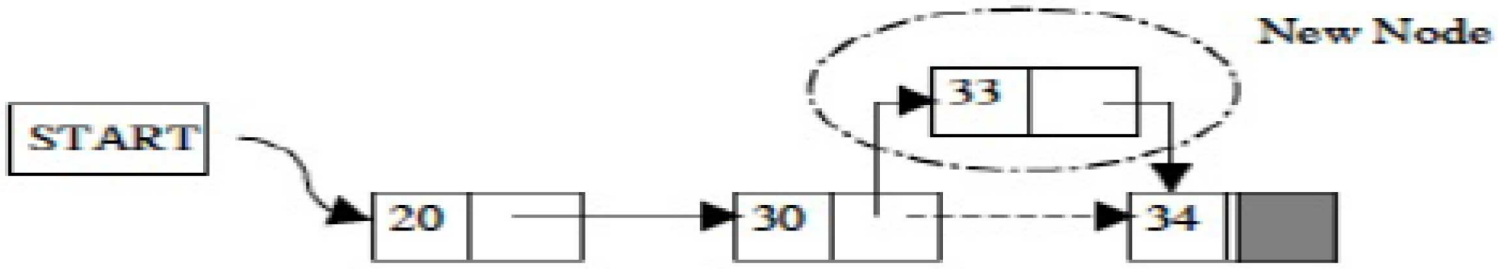


Fig11:

Figure 6: Insertion of New Node at specific position of SLL

- Suppose START is the first position in linked list. Let DATA be the element to be inserted in the new node. POS is the position where the new node is to be inserted. TEMP is a temporary pointer to hold the node address.

ALGORITHM FOR INSERTION A NODE IN SLL

Insert a Node at the beginning of Linked List:

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode→ DATA = DATA
4. If(START == NULL)
    a. NewNode→next = NULL
5. Else
    a. NewNode→next = START
6. START = NewNode
7. Exit

ALGORITHM FOR INSERTION A NODE IN SLL

Insert a Node at the end of Linked List:

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode→ DATA = DATA
4. NewNode→next = NULL
5. If(START == NULL)
   a. START = NewNode
6. Else
   a. TEMP = START
   b. While (TEMP→ next!=NULL)
      i. TEMP=TEMP→next
   c. TEMP→next = NewNode
7. Exit

ALGORITHM FOR INSERTION A NODE IN SLL

Insert a Node at any specified position of Linked List:

1. Input DATA and POS to be inserted
2. Initialize TEMP=START and i=1
3. Repeat the step 3 while (i<POS-1)
   a. TEMP = TEMP→ next
   b. If(TEMP ==NULL)
      a. Display "Node in the list less than the position"
      b. Exit
   c. i=i+1
4. Create a NewNode
5. NewNode→ DATA =DATA
6. NewNode → next=TEMP→next
7. TEMP→next =NewNode
8. Exit

ALGORITHM FOR DISPLAY ALL NODES

Suppose START is the address of the first node in the linked list.

1. If (START ==NULL)
   a. Display “The list is Empty”
   b. Exit
2. Initialize TEMP = START
3. Repeat the step 4 and 5 until (TEMP==NULL)
4. Display “TEMP→DATA”
5. TEMP=TEMP→next
6. Exit

Deleting Nodes in SLL:

- A node may be deleted:
- From the beginning of the linked list
- from the end of the linked list
- from the specified position in a linked list

Deleting first node of the linked list:

let *head/START(in the earlier section) be the pointer to first node in the current list

1. If(head==NULL) then

   print "Void deletion" and exit

2. Store the address of first node in a temporary variable **temp.**

   temp=head;

3. Set head to next of head.

   head=head->next;

4. Free the memory reserved by temp variable.

   free(temp);

5. End

Deleting the last node of the linked list:

let *head be the pointer to first node in the current list

1. If(head==NULL) then //if list is empty

   print "Void deletion" and exit

2. else if(head->next==NULL) then //if list has only one node

   Set temp=head;

   print deleted item as,

   printf("%d" ,head->info);

   // DATA and info are same

   head=NULL;

   free(temp);

3. else

   set temp=hold=head;

   while(temp->next!=NULL)//added

   {

   hold=temp;

   temp=temp->next;

   }

   hold->next=NULL;

   free(temp);

   temp=NULL;

4. End

An algorithm to delete a node at the specified position in a singly linked list:

let *head be the pointer to first node in the current list

1. *Read position of a node which to be deleted, let it be pos.*

2. if head==NULL

   print "void deletion" and exit

3. Enter position of a node at which you want to delete a new node. Let this position is pos.

4. Set temp=head

declare a pointer of a structure let it be *hold

5. if (head ==NULL)then

   print "void deletion" and exit

otherwise;.

6. for(i=1;i<pos-1;i++)

   temp=temp->next;

7. hold=temp->next;

   temp->next=hold->next;

8. free(hold);

11. End

ALGORITHM FOR SEARCHING A NODE

Suppose START is the address of the first node in the linked list and DATA is the information to be searched. After searching, if the DATA is found, POS will contain the corresponding position in the list.

1. Input the DATA to be searched
2. Initialize TEMP = START; POS =1
3. Repeat the step 4, 5 and 6 until (TEMP == NULL)
4. If (TEMP → DATA ==DATA)
   a. Display "The data is found at POS"
   b. Exit
5. TEMP = TEMP→next
6. POS = POS+1
7. If (TEMP == NULL)
   a. Display "The data is not found in the list"
8. Exit

STACK USING LINKED LIST

The following figures shows that the implementation of stack using linked list.

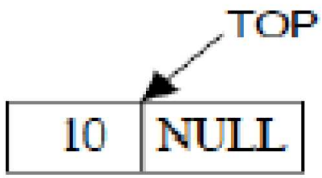


Fig13: Push (10)

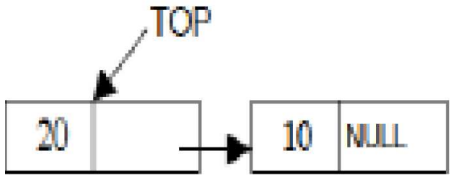


Fig14: Push (20)

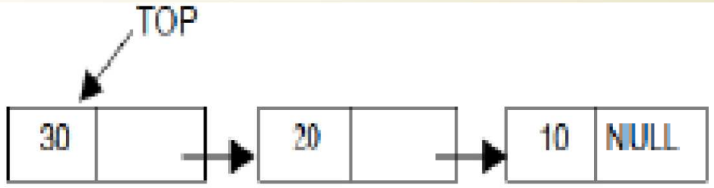


Fig15: Push (30)

Fig16: X = Pop (i.e. X=30)

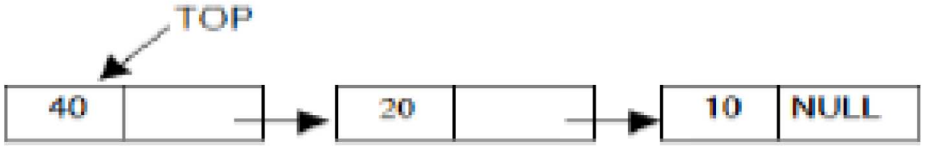


Fig17: Push (40)

STACK USING LINKED LIST

Algorithm for PUSH operation:

- Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. DATA is the data item to be pushed.

1. Input the DATA to be pushed
2. Create a new Node
3. NewNode→DATA=DATA
4. NewNode→next = TOP
5. TOP=NewNode
6. exit

STACK USING LINKED LIST

Algorithm for POP operation:

- Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. TEMP is pointer variable to hold any nodes address. DATA is the information on the node which is just deleted.

1. If (TOP ==NULL)
   a. Display "The stack is empty"
2. Else
   a. TEMP = TOP
   b. Disply "The popped elemet TOP→DATA"
   c. TOP = TEMP→Next //new TOP
   d. TEMP → next = NULL
   e. Free the TEMP node
3. Exit

QUEUE USING LINKED LIST

The following figure shows that the implementation issues of Queue using linked list.

Fig18: Enqueue (10)

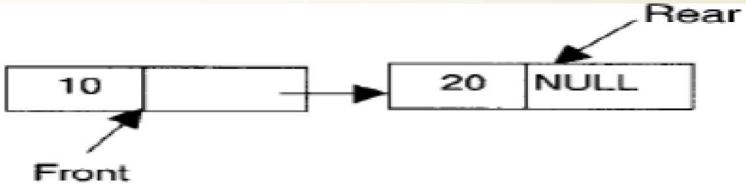


Fig19: Enqueue (20)

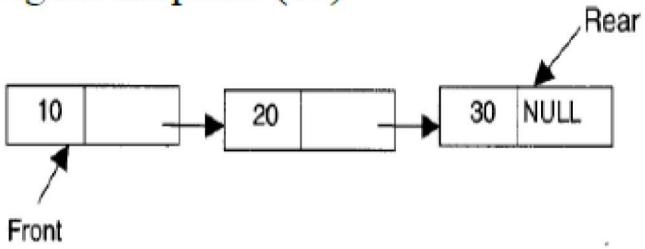


Fig20: Enqueue (30)

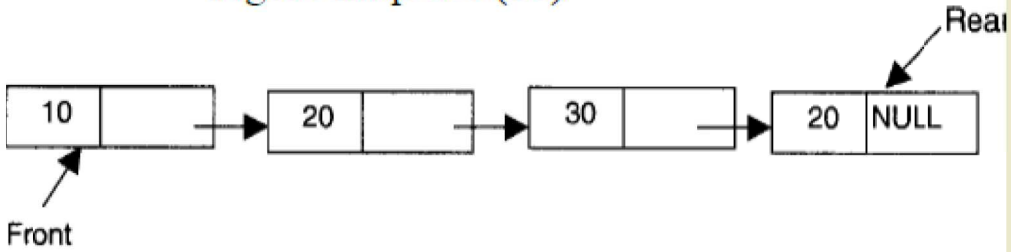


Fig21: Enqueue (20)

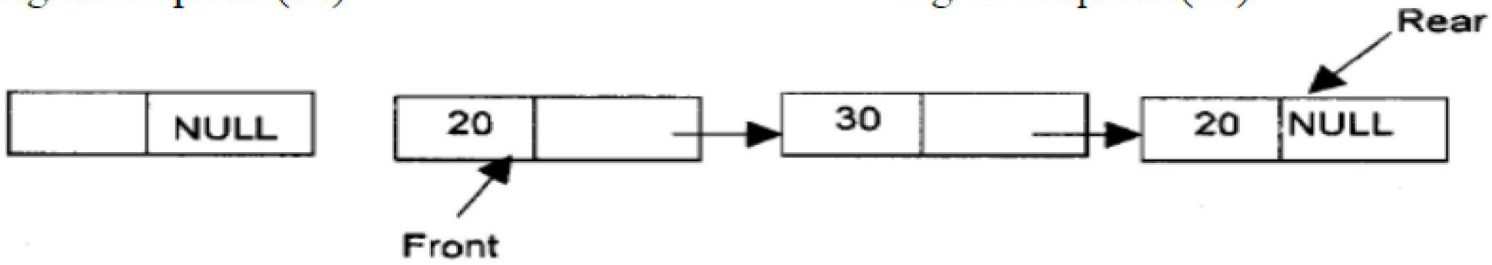


Fig22: X = Dequeue (i.e. X = 10)

ALGORITH FOR ENQUEUE AN ELEMENT INTO A QUEUE

- REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element to be pushed.

1. Input the DATA element to be pushed
2. Create a New Node
3. NewNode → DATA = DATA
4. NewNode → next = NULL
5. If (REAR != NULL)
   a. REAR→next = NewNode
6. REAR = NewNode
7. Exit

ALGORITH FOR DEQUEUE AN ELEMENT FROM A QUEUE

REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element popped from the queue.

1. If (FRONT == NULL)
    a. Display “The queue is empty”
2. Else
    a. Display “ the popped element is FRONT → DATA”
    b. If (FRONT != REAR)
        i. FRONT = FRONT → Next
    c. Else,
        i. Front = NULL
3. exit

ADVANTAGE AND DISADVANTAGE OF SINGLY LINKED LIST

Advantages:

- Accessibility of a node in the forward direction is easier
- Insertion and deletion of node are easier, it doesn't need movement of elements for insertion and deletion
- Size is not fixed and wastage of memory issue is solved. It can be extended and reduced during runtime according to our requirements.

Disadvantages:

- It requires more space as pointers are also stored with information.
- Can traverse list only in the forward direction.

DOUBLY LINKED LIST

- A doubly linked list is one in which all nodes are linked together by multiple links which help in accessing both the successor (next) and predecessor (previous) node for any arbitrary node within the list.
- Every nodes in the doubly linked list has three fields: LeftPointer (Prev), RightPointer (Next) and DATA
- **Prev** will point to the node in the left side i.e. **Prev** will hold the address of the previous node.
- **Next** will point to the node in the right side i.e. **Next** will hold the address of next node.
- **DATA** will store the information of the node.

Fig23: Typical Doubly Linked list node

Figure 8: Representation of Doubly Linked List (DLL)

REPRESENTATION OF DOUBLY LINKED LIST

Following declaration can represent doubly linked list

```
struct Node
{
    int data;
    struct Node *Next;
    struct Node *Prev;
};
typedef struct Node *NODE;
```

All the primitive operations performed on singly linked list can also be performed on doubly linked list. The following figures shows that the insertion and deletion of nodes.

Fig25: Add (20)

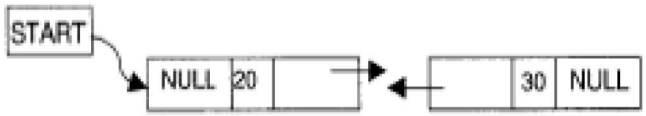


Fig26: Insert (30) at the end

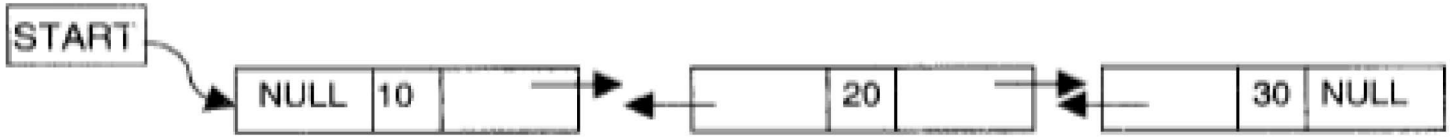


Fig27: Insert (10) at the Beginning

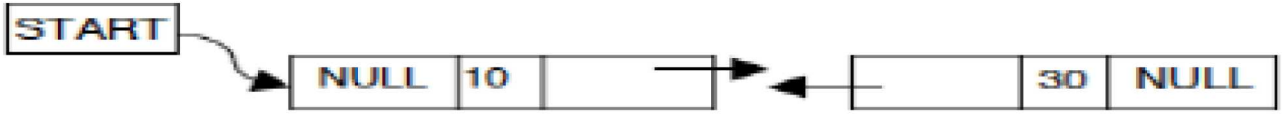


Fig28: Delete a node at the 2nd position (Delete 20 at 2nd position)

ALGORITHM FOR INSERTING A NODE INTO DLL

- Suppose START is the first position in linked list. Let DATA be the element to be inserted in the new node. TEMP is a temporary pointer to hold the node address and POS is the position where the NewNode is to be inserted.

At the beginning:

1. Input DATA element to be inserted
2. Create New Node
3. NewNode → DATA =DATA
4. If (SART == NULL)
   a. NewNode → next = NULL
   b. NewNode→prev = NULL
5. Else
   a. TEMP = START
   b. NewNode→ prev = NULL
   c. NewNode→next = TEMP
   d. TEMP→prev = NewNode
6. Start=NewNode
7. exit

ALGORITHM FOR INSERTING A NODE INTO DLL

At the end:

1. Input DATA element to be inserted
2. Create New Node
3. NewNode → DATA =DATA
4. If (SART == NULL)
   a. NewNode → next = NULL
   b. NewNode→prev = NULL
   c. START=NewNode
5. Else
   a. TEMP = START
   b. While (TEMP→next != NULL)
      a. TEMP = TEMP → next
   c. NewNode→ prev = TEMP
   d. NewNode→next = NULL
   e. TEMP→ next= NewNode
6. exit

ALGORITHM FOR INSERTING A NODE INTO DLL

At the specified location:

1. Input DATA and POS
2. Create New Node
3. NewNode → DATA =DATA
4. TEMP = START; i=1
5. while((i< POS-1) && (TEMP!=NULL))
   a. TEMP = TEMP→ next ; i=i+1
6. If ((TEMP !=NULL) && (i=POS-1))
   a. NewNode→ prev = TEMP
   b. NewNode→next = TEMP→next
   c. (TEMP→ next)→prev = NewNode
   d. TEMP→next = NewNode
7. Else
   a. Display "Position not found"
8. exit

ALGORITHM FOR DELETING A NODE FROM DLL

- Suppose START is the address of the first node in the linked list. Let DATA is the number to be deleted. TEMP and PTR is the temporary pointer to hold the address of the node.

1. If (START→ DATA == DATA) **(delete from beginning)**
    a. TEMP = START
    b. START = START→ next
    c. START → Prev = NULL
    d. Free the TEMP
2. PTR = START
3. Repeat until((PTR –> next)→ next !=NULL) **(Delete from the position)**
    a. If ((PTR → next)→ DATA == DATA)// repeat until this condition is matched
        i. TEMP = PTR→ next

ALGORITHM FOR DELETING A NODE FROM DLL

   ii. PTR→next = TEMP → next

   iii. (TEMP → next) → prev = PTR

   iv. Free the TEMP

b. PTR= PTR→next

4. If ((PTR—>next)→DATA == DATA) **(Delete from the last)**

   a. TEMP = PTR→next

   b. Free the TEMP

   c. PTR→next == NULL

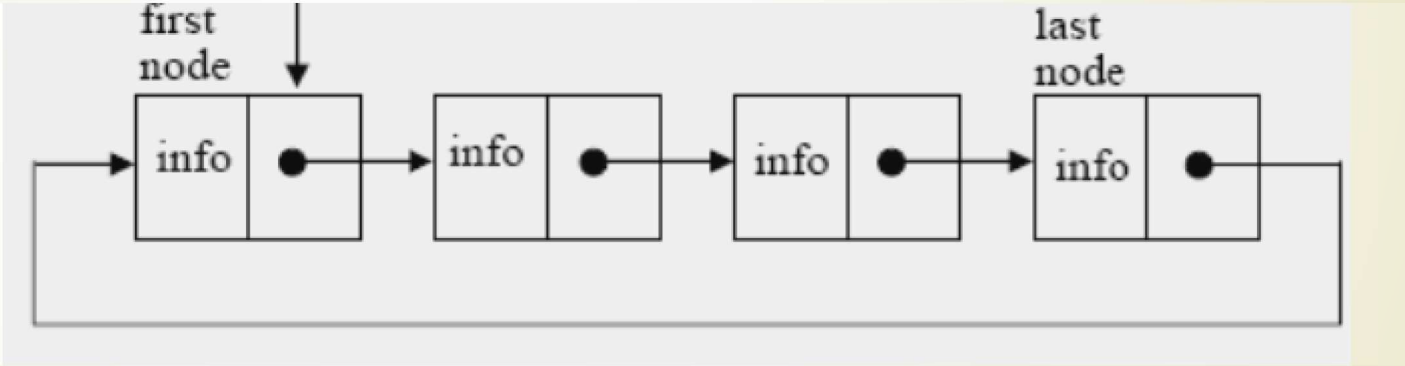
5. Else

   a. Display “Data not Fund”

6. Exit

CIRCULAR LINKED LIST

- A circular linked list is one, which has no beginning and no end.
- A singly linked list can be made a circular linked list by simply storing the address of the very first node in the linked field of the last node.
- Circular linked lists also make our implementation easier, because they eliminate the boundary conditions associated with the beginning and end of the list.



INSERTION ALGORITHM INTO CIRCULAR LINKED LIST

- Suppose START is the first position in linked list. Let DATA be the element to be inserted in the new node. LAST indicated the last node.

At the beginning:

1. Create a New Node
2. NewNode→DATA = DATA
3. If (START == NULL)
   a. NewNode→next = NewNode
   b. START = NewNode
   c. LAST = NewNode
4. Else
   a. NewNode → next = START
   b. START = NewNode
   c. LAST → next = NewNode
5. Exit

INSERTION ALGORITHM INTO CIRCULAR LINKED LIST

At the end:

1. Create a New Node
2. NewNode→DATA = DATA
3. If (START == NULL)
    a. NewNode→next = NewNode
    b. START = NewNode
    c. LAST = NewNode
4. Else
    a. LAST → next = NewNode
    b. LAST = NewNode
    c. LAST → next = START
5. Exit

DELETION ALGORITHM FROM CIRCULAR LINKED LIST

At the beginning:

1. Declare a temporary node , PTR
2. If (START == NULL)
    a. Display " Empty Circular queue"
    b. Exit
3. PTR = START
4. START = START →next
5. Print , element deleted is PTR→ DATA
6. LAST→next = START
7. Free (PTR)
8. Exit

DELETION ALGORITHM FROM CIRCULAR LINKED LIST

At the end:

1. Declare a temporary node , PTR
2. If (START == NULL)
    a. Display " Empty Circular queue"
    b. Exit
3. PTR = START
4. While (PTR! = LAST)
    a. PTR1 = PTR// to make the second last node to make new last node
    b. PTR = PTR →next
5. Print , element deleted is PTR→ DATA
6. LAST = PTR1
7. LAST→next = START
8. Free PTR

APPLICATION OF LINKED LIST IN COMPUTER SCIENCE

- Implementation of stacks and queues
- Implementation of graphs : Adjacency list representation of graphs is most popular which uses linked list to store adjacent vertices.
- Maintaining directory of names

APPLICATION OF LINKED LIST IN REAL WORLD

- Image viewer – Previous and next images are linked, hence can be accessed by next and previous button.
- Previous and next page in web browser – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
- Music Player – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

Thank you