Unit-5

Lists : (Linked list):

List is a thing that defines a sequential set of elements to which we can add new elements and remove or change existing ones. The list data structure typically has two distinctive implementations; array list and linked list. Array list is also known as contiguous list.

Array List (Contiguous list):

In array list the data structure can grow as much as it needs in case of dynamic array compared to the classical static array because it has a predefined maximum size. The advantage of array list is it improves the speed since elements are sequential in memory, they can greatly benefit from CPU cache. Array implementation is used when we have to store items without knowing how much space we will need in advance.

Array Implementation of list:

To implement a contigious list, we need to define a large enough array to hold all the items of the list. The first item of the list is placed at 0^{th} position of array and successive items in successive positions of the array. The operation of insertion and deletion of items can be done to/from anywhere within the list.

Implementation:

For array implementation let first we make assumptions for some user-defined key-words that we are going to use further.

→ N be the maximum number of elements/items in an array.
→ array[N] be an array to hold the items of list.
→ last_index is defined to indicate index of the last element.
→ Initially, last_index = -1

Now we perform array implementation for operations insertion and deletion as follows;

Insertion:

(Assumptions)

Algorithm:

1. Read item x to insert.
2. If last_index == N-1 then declare the list is full and return.
3. If last_index == -1, increment last_index and perform array[0] = x, and return.
4. Read position pos to insert (pos=0 means array[0]).
5. If pos > last_index+1, declare invalid and return.
6. i = last_index;
7. while (i >= pos)
   array[i+1] = array[i];
   i = i-1;
8. array[pos] = x;
9. Increment last_index
10. Return.

Deletion:

(Assumptions)

Algorithm:

1. If last_index == -1, declare list is empty and return.
2. Read position pos to delete.
3. If pos > last_index, declare invalid and return.
4. x = array[pos];
5. i = pos;
6. while (i < last_index)
   array[i] = array[i+1];
   i = i+1;
7. Decrement last_index
8. Return x as deleted item.

⊛. List as an ADT:-

List is a sequence of linearly ordered elements in which different kinds of operations can be performed. Some of the list ADT functions are as follows;

i) get() → Return an element from the list at any given position.
ii) insert() → Insert an element at any position of the list.
iii) remove() → Remove the first occurance of any element from a non-empty list.
iv) replace() → Replace an element at any position by another element.
v) size() → Return the number of elements in the list.
vi) IsEmpty() → Return true value if the list is empty, otherwise return false.
vii) IsFull() → Return true if the list is full, otherwise return false.

⊛. List operations:-

Let 'L' be the list of elements
e → be an element of the list
& p → be the position of an element in the list
Many kinds of operations can be performed on list some of which are as follows;

i) Insert (e, p, L)
ii) Locate (e, L)
iii) Retrive (p, L)
iv) Delete (p, L)
v) Next (p, L)
vi) END (L) etc.

Linked List:-

As we know that array is a collection of similar kinds of data types. So, the consecutive memory area is required for array which is it's disadvantage. Another disadvantage of array is that it consist of limited size and if we want to insert and delete an element then it will be complicated and expensive.

So to overcome these disadvantages linked list is created. Linked list is also a linear data structure but it is not essential to have consecutive area. It has unlimited size and it is easy to insert and delete an element in existing list.

Defⁿ → A linked list is a linear data structure. It is a collection of nodes in which each node has two parts; information part and link part.

To store a set of items in linked list representation, the information part of node contains the actual item of the list while the link part of the node contains the address of the next node in the list. The address of the last node of a linked list is NULL.

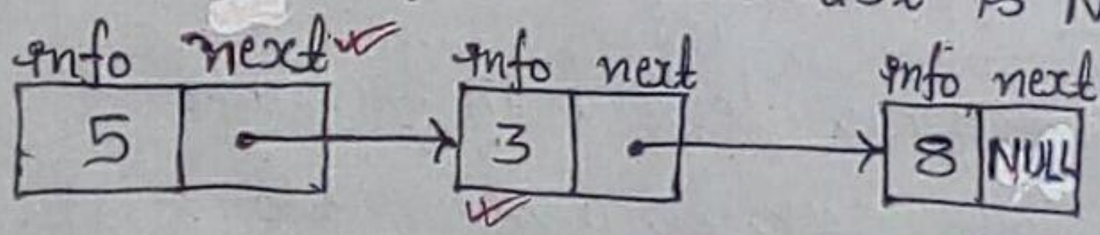


fig. Illustration of linked list

Remember that;

→ The first next (link part) in above figure contains address of next node (i.e, address of node having info 3) and similarly preceeds.
→ The nodes in a linked list are not stored contiguously in the memory.
→ You do not have to shift any element in the list.
→ Memory for each node can be allocated dynamically whenever the need arises.
→ The size of linked list can grow or shrink dynamically.

Types of linked list:

i) Singly linked list
ii) doubly linked list.
iii) Circular linked list.

Q. Differences between array and linked list.

Operations on linked list:

i) Creation → This operation is used to create a linked list.

ii) Insertion → This operation is used to insert a new node in a linked list in specified position. A new node may be inserted:
   i) At the beginning of the linked list.
   ii) At the end of linked list.
   iii) At the specified position in linked list.

iii) Deletion → The deletion operation is used to delete a node from the linked list. A node may be deleted from:
   i) The beginning of the linked list.
   ii) The end of the linked list.
   iii) The specified position in the linked list.

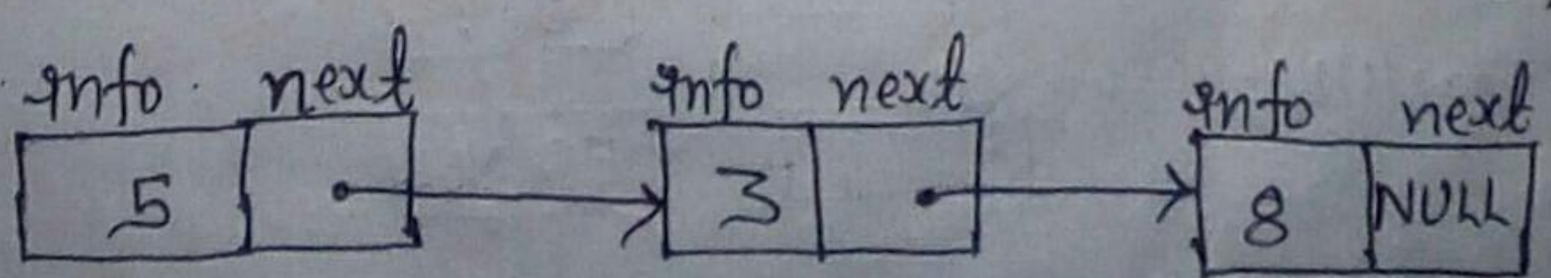
iv) Traversing → The list traversing is a process of going through all the nodes of the linked list from one end to the other end. The traversing may be either forward or backward.

v) Searching or Find → This operation is used to find an element in a linked list. If the desired element is found we say operation is successful otherwise unsuccesseful.

vi) Concatenation → It is the process of appending second list to the end of first list.

1) Singly Linked List:-

In this type of linked list each node contains two fields one is info field which is used to store data items and another is link field that is used to point the next node in the list. The last node has NULL pointer. The following example is a singly linked list that contains three elements 5, 3, 8.



⊛ C implementation (OR representation) of singly linked list:

We can create a structure for the singly linked list having two members in each node. One member is info that is used to store the data items and another is next field that store the address of next node in the list.

We can define a node as follows;

```
struct Node
{
    int info;
    struct Node *next;
};
typedef struct Node NodeType;
NodeType *head; // head is a pointer type structure variable.
```

→ This type of structure is called self-referential structure. The NULL value of the next field of the linked list indicates the last node and we define macro for NULL and set it to 0 as below:

```
#define NULL 0
```

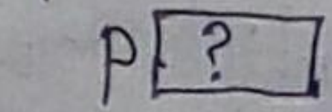
⊛ Creating a Node:

→ To create a new node, we use the malloc function to dynamically allocate the memory for new node.

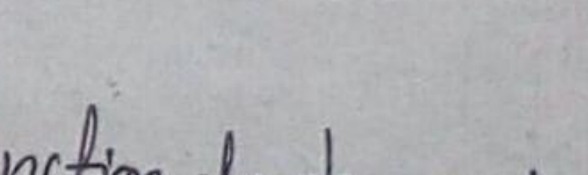
→ After creating the node we can store the new item in the node using a pointer to that node.

The following steps clearly shows the required steps to create a node and storing an item.

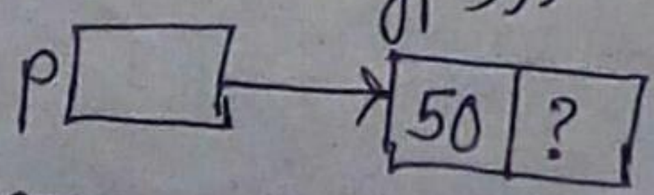
```
Nodetype *p;
```



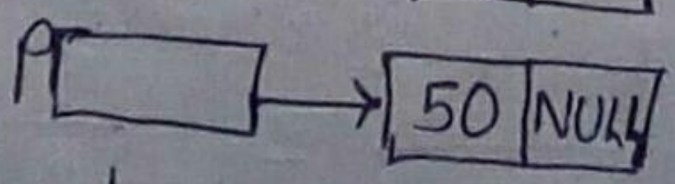

```
p=(NodeType*)malloc(sizeof(Nodetype));
```




```
p->info=50;
```




```
p->next=NULL;
```



Note that p is not a node; instead it is a pointer to a node.

@ The getNode function:

We can define a function getNode() to allocate the memory for a node dynamically. It is user-defined function that return a pointer to the newly created node.

```
Nodetype *getNode()
{
    NodeType *p;
    P==(NodeType*)malloc(sizeof(NodeType));
    return(p);
}
```

ⓑ Creating the empty list:

```
void createEmptylist (NodeType *head)
{
    head = NULL;
}
```

⊛ Inserting Nodes:

To insert an element or a node in a linked list, the following three things to be done;

i) Allocating a node
ii) Assigning a data to info field of the node.
iii) Adjusting a pointer and a new node may be inserted.

@. An algorithm to insert a node at the beginning of singly Linked list:

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

1. Create a node using malloc function.

```
NewNode=(NodeType*)malloc(sizeof(NodeType));
```

2. Assign data to info field of new node.

```
NewNode->info=newItem;
```

3. Set next of new node to head

```
NewNode->next=head;
```

4. Set the head pointer to the new node

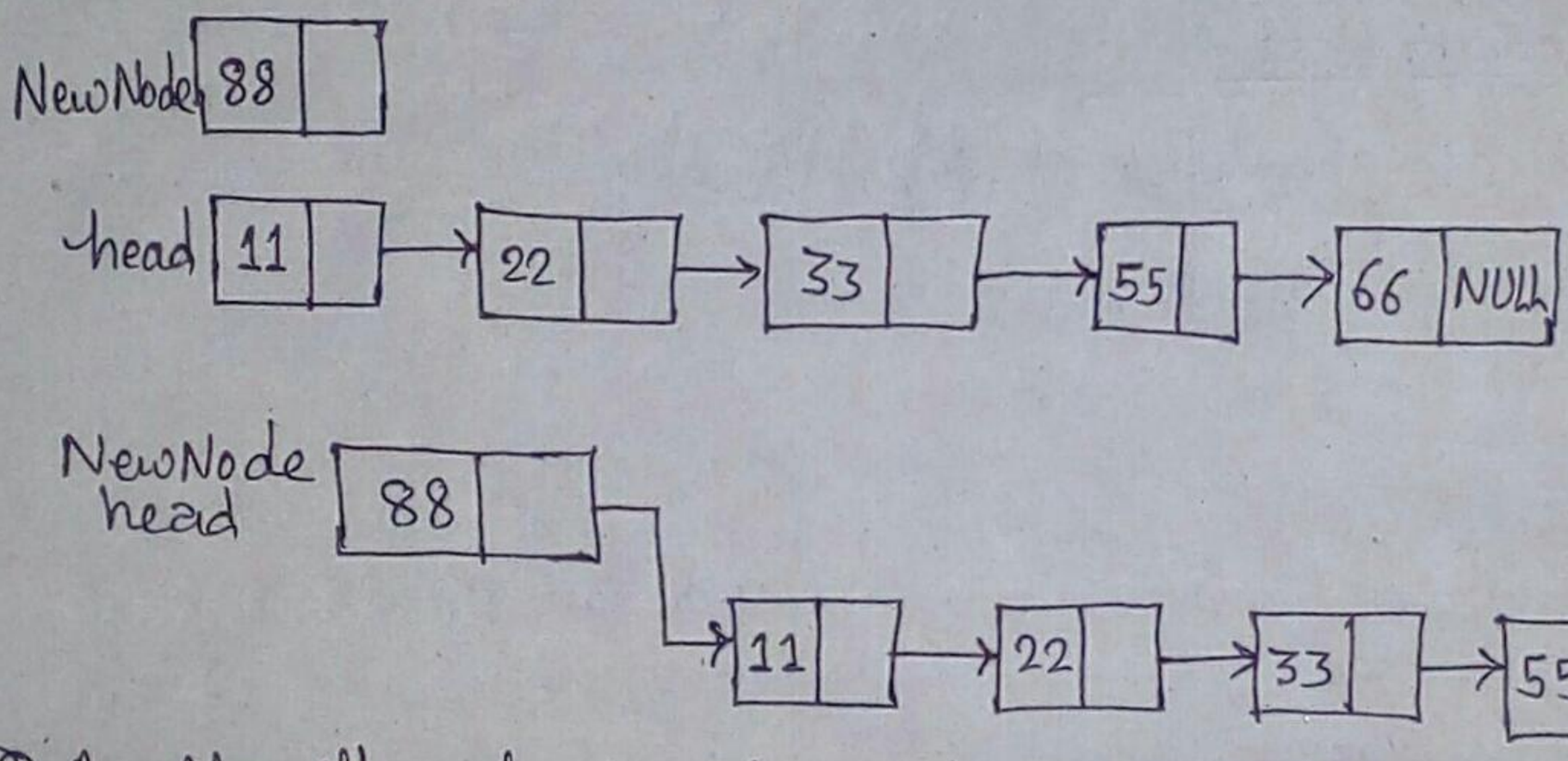
```
head=NewNode;
```

5. End.

for understanding → not imp for exam point of view

The C function to insert a node at the beginning of singly linked list:

```
void InsertAtBeg(int newItem)
{
    NodeType *NewNode;
    NewNode = getNode();
    NewNode->info = newItem;
    NewNode->next = head;
    head = NewNode;
}
```



(b) An Algorithm to insert a node at the end of singly linked list:

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

1. Create a new node using malloc function.
   NewNode = (NodeType*)malloc(sizeof(NodeType));
2. Assign data to the info field of new node.
   NewNode->info = newItem;
3. Set next of new node to NULL.
   NewNode->next = NULL;
4. if (head == NULL) then
   Set head = NewNode and exist.
5. Set temp = head;
6. while (temp->next != NULL)
   temp = temp->next; // increment temp
7. Set temp->next = NewNode;
8. End.

The C function to insert a node at the end of linked list:

```
VoidInsert AtEnd (int newItem)
 {
  NodeType *NewNode;
  NewNode = getNode();
  NewNode->next=NULL;
   If(head == NULL)
      { head = NewNode;
      }
else  {
      { temp=head;
  while(temp->next != NULL)
            { temp=temp->next;
            }
        temp->next= NewNode;
      }
 }
```

(c) An Algorithm to insert a node at the specified position in a singly Linked list:

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

1. Create a new node using malloc function.

   NewNode=(NodeType*)malloc (sizeof (NodeType));

2. Assign data to info field of new node.

   NewNode->info=newItem;

3. Entering the position of a node at which you want to insert a new node. Let this position is pos.
4. Set temp=head;
5. If(head==NULL) then.

   printf("void insertion") and exit.

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

   temp=temp->next;

7. Set NewNode->next=temp->next;

   Set temp->next=NewNode.

8. End.

⊛. Deleting Nodes:-

@. An Algorithm for deleting the first node of the singly Linked list:

Let *head be the pointer to first node in 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 reversed by temp variable.
   free (temp);
5. End.

(b). An Algorithm for deleting the last node of the singly linked list:

Let *head be the pointer to first node on the current list.

1. If (head==NULL) then,
   print "Void deletion" and exist.
2. else if (head->next==NULL) then //if list has only one node
   Set temp=head;
   print deleted item as:
   printf ("%d", head->info);
   head=NULL;
   free (temp);
3. else
   Set temp=head;
   while (temp->next->next!=NULL)
   Set temp=temp->next;
   End of while
   free (temp->next);
   Set temp->next=NULL
4. End.

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

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

1. If head==NULL
   print "void deletion" and exit.
2. Enter position of a node at which you want to delete a new node.
   Let this position is pos.
3. Set temp=head
   declare a pointer of structure let it be *p;
4. If (head==NULL) then
   print "void deletion" and exit
   otherwise;
5. for(i=1; i<pos-1; i++)
   temp=temp->next;
6. Print deleted item is temp->next->info
7. Set p=temp->next;
8. Set temp->next=temp->next->next;
9. free(p);
10. End.

2). Doubly linked list:

A linked list in which all nodes are linked together by multiple number of links i.e, each node contains three fields (two pointer fields and one data field) is called doubly linked list. It provides bidirectional traversal.

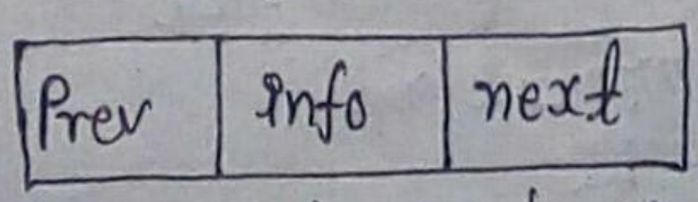


fig. A node in doubly linked list

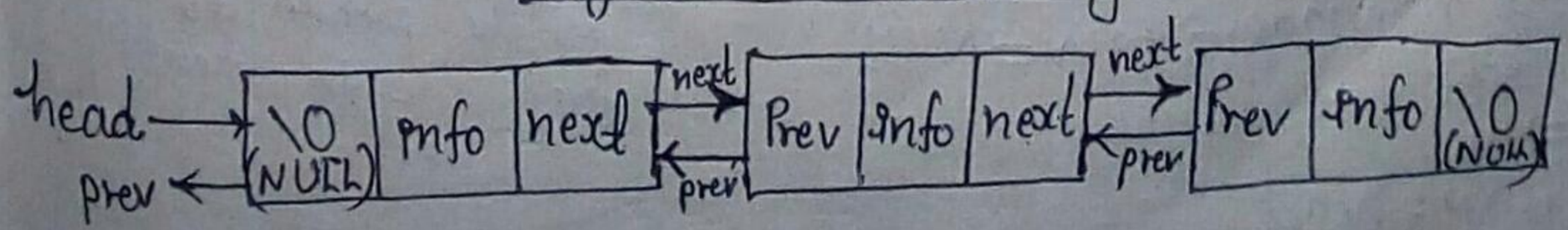


fig. A doubly linked list with three nodes.

In singly linked list we cannot access predecessor node from current node. This problem is overcomed by doubly linked list.

C representation of doubly linked list:

```
struct node
    {
        int info;
        struct node *prev;
        struct node *next;
    };
typedef struct node NodeType;
NodeType *head = NULL.
```

⊛. Inserting Nodes:- (In doubly linked list):

(a). Algorithm to insert a node at the beginning of a doubly linked list:

1. Allocate memory for the new node as:
   newnode = (NodeType*) malloc (sizeof (NodeType))
2. Assign value to info field of a new node.
   set newnode → info = item.
3. Set newnode → prev. = newnode → next = NULL
4. Set newnode → next = head.
5. Set head → prev = newnode.
6. Set head = newnode
7. End.

*head is assumed as the pointer to first node in current list

(b) Algorithm to insert a node at the end of a doubly linked list:

1. Allocate memory for the new node as;
   newnode = (NodeType*) malloc (sizeof (NodeType))
2. Assign value to info field of new node
   Set newnode → info = item
3. Set newnode → next = NULL
4. If head == NULL
   set newnode → prev = NULL.
   set head = newnode
5. If head != NULL
   set temp = head
   while (temp → next != NULL)
   temp = temp → next
   end while

*head is assumed as the pointer to first node in current list.

set temp->next = newnode
set newnode->prev = temp
6. End.

⊛. Deleting Nodes (In doubly linked list):-

@. Algorithm to delete a node from beginning of a doubly linked list:

1. if head == NULL then
   print "empty list" and exit.
2. else
   set hold = head.
   set head = head->next
   set head->prev = NULL
   free (hold).
3. End.

ⓑ Algorithm to delete a node from end of a doubly linked list:

1. if head == NULL then
   print "empty list" and exist.
2. else if (head->next == NULL) then,
   set hold = head
   set head = NULL
   free (hold).
3. else
   set temp = head;
   while (temp->next->next != NULL)
   temp = temp->next
   end while
   set hold = temp->next
   set temp->next = NULL
   free (hold).
4. End.

3) Circular linked list:

A circular linked list is a list where the link field of last node points to the very first node of the list.

Circular linked lists can be used to help the traverse the same list again and again if needed. A circular list is very similar to the linear list where in the circular list the pointer of the last node points not NULL but the first node.

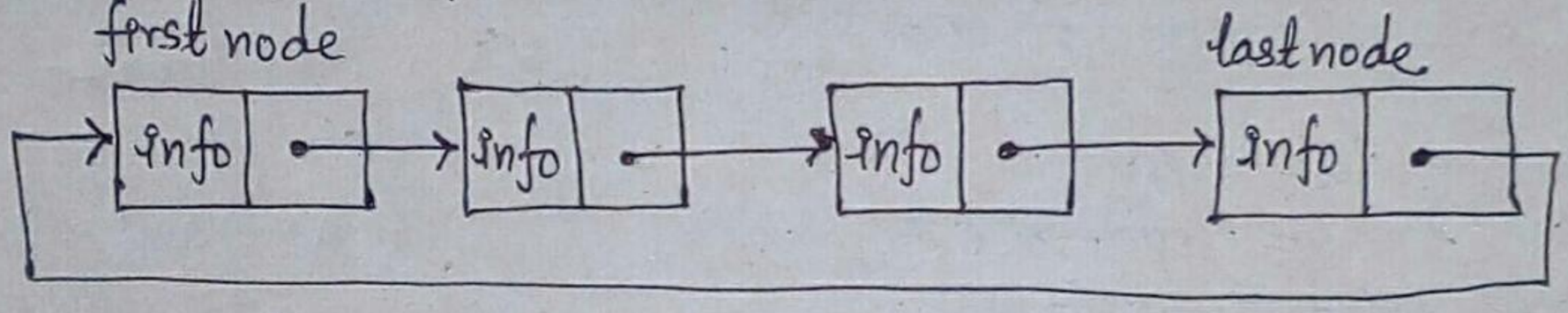


fig. A circular linked list without header node.

In a circular linked list there are two methods to know if a node is the first node or not:

i) Either a external pointer, List, points the first node or

ii) A header node is placed as the first node of the circular list.

The header node can be seperated from the others by either having a sentinel value (i.e, dummy data) as the info part or having a dedicated flag variable to specify if the node is a header node or not.

C representation of circular linked list:

We declare the structure for the circular linked list in the same way as we declared it for linear linked list.

```
struct node
{
  int info;
  struct node *next;
};
typedef struct node NodeType;
NodeType *start = NULL;
NodeType *last = NULL;
```

⑧. Inserting Nodes: (In Circular linked list):

ⓐ. Algorithm to insert a node at the beginning of a circular linked list:

1. Create a new node as:
   newnode = (NodeType*)malloc(sizeof(NodeType));
2. If start=NULL then,
   set newnode->info = item
   set newnode->next=newnode.
   set start=newnode
   set last->newnode
   end if
3. else
   set newnode->info = item
   set newnode->next = start
   set start=newnode
   set last->next=newnode
   end else
4. End.

ⓑ. Algorithm to insert a node at the end of circular linked list:

1. Create a newnode as:
   newnode=(NodeType*)malloc(sizeof(NodeType));
2. If start==NULL then,
   set newnode->info = item
   set newnode->next=newnode.
   set start = newnode.
   set last->newnode.
   end if
3. else
   set newnode->info = item
   set last->next = newnode
   set last=newnode.
   set last->next=start
   end else
4. End.

✱. Deleting Nodes (In Circular linked list):

(a) Algorithm to delete a node from beginning of circular linked list:

1. If start == NULL then.
   print "empty list" and exit.
2. else
   set temp = start
   Set start = start → next
   print the deleted element = temp → info
   set last → next = start
   free (temp)
   end else.
3. End.

(b). Algorithm to delete a node at the end of circular linked list:

1. If start == NULL then,
   "empty list" and exist
2. else if start == last
   set temp == start
   print deleted element = temp → info
   free (temp)
   start = last = NULL.
3. else
   set temp = start
   while (temp → next != last)
   set temp = temp → next
   end while.
   set hold = temp → next
   set last = temp
   set last → next = start
   print the deleted element = hold → info
   free (hold)
   end else.
4. End.

C) Linked list Implementation of Stack and Queue:

1) Singly linked list:

@ Implementation of stack:

Push function:

let *top be the top of the stack or pointer to the first node of the list.

```
void push (item)
{
  NodeType *nnode;
   int data;
  nnode=(NodeType*)malloc (sizeof (NodeType));
  if (top==0)
    { nnode->info=item;
      nnode->next=NULL;
       top=nnode;
    }
else
    {nnode->info=item;
     nnode->next=top;
     top=nnode;
    }
```

Pop function:

let *be the top of the stack or pointer to first node of the list.

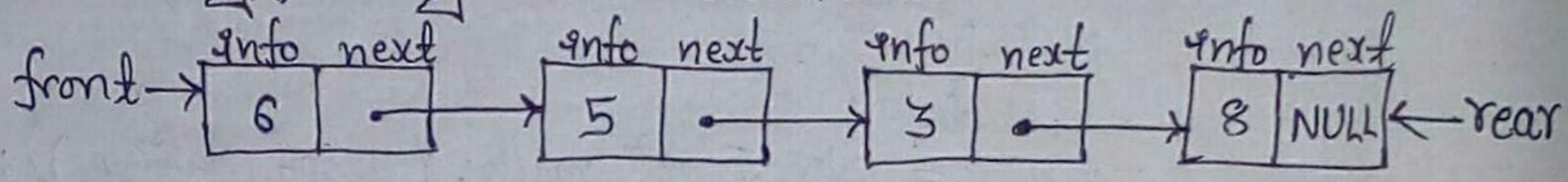
```
void pop()
  {
    NodeType *temp;
    if (top==0)
          { printf ("Stack contain no elements : \n");
            return;
          }
else
    { temp=top;
      top=top->next;
      printf ("\n deleted item is %d\t", temp->info);
       free (temp);
    }
}
```

(b) Implementation of Queue:

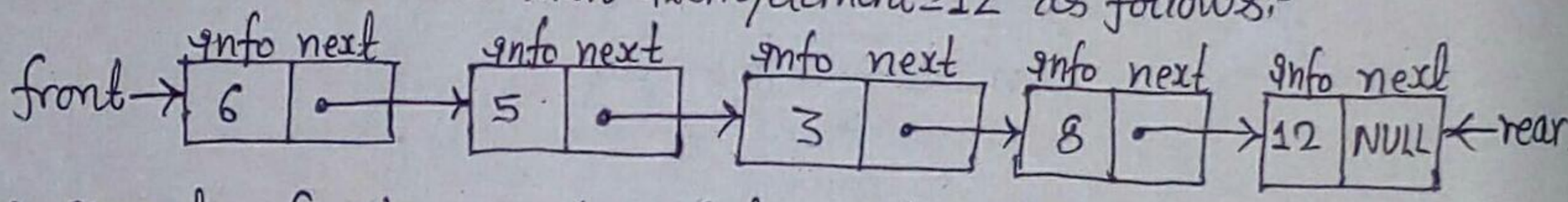
(a) Insert Function:

Let *rear and *front are pointers to the first node of the list initially and insertion of node in linked list done at the rear part and deletion of node from the linked list done from front part.

Let initially following be the linked list.



Let we insert the node with item/element=12 as follows:-



The insertion function implementation is as follows:-

```
void insert (int item)
  {
  NodeType *nnode;
  nnode=(NodeType*) malloc (sizeof (NodeType));
  if (rear==0)
      { nnode->info = item;
        nnode->next=NULL
        rear->next =nnode;
      }
  rear=nnode;
}
```

(b) Delete function:

Let *rear and *front are pointers to the first node of the list initially and insertion of node in linked list done at the rear part and deletion of node from the linked list done from front part.

The deletion function is implemented as follows:-

```
void delete()
  { NodeType *temp;
    if (front==0)
     { printf("Queue contain no elements:\n");
       return;
     }
    else if(front->next==NULL)
     { temp=front;
       rear=front=NULL;
       printf("\n Deleted item is %d \n", temp->info);
       free(temp);
     }
else
     { temp=front;
       front=front->next;
       printf("\n Deleted item is %d \n", temp->info);
       free(temp);
     }
  }
```

2) Circular Linked List:

@ Stack as circular list:

To implement a stack in a circular linked list, let pstack be a pointer to the last node of a circular list. Actually there is no any end of a list but for convention let us assume that the first node (rightmost node of a list) is the top of the stack. An empty stack is represented by a null list.

The structure for the circular linked list implementation of stack is:

```
struct node
   { int info;
     struct node *next;
   };
typedef struct node NodeType;
NodeType *pstack=NULL;
```

(a). PUSH function:

```
void PUSH(int item)
 {
    NodeType newnode;
    newnode=(NodeType*)malloc (sizeof (NodeType));
    newnode -> info = item;
    if (pstack == NULL)
    {
      pstack=newnode;
      pstack->next =pstack;
    }
else
    { newnode ->next = pstack -> next;
      pstack -> next = newnode;
    }
}
```

(b) POP function:

```
void POP ()
    {
      NodeType *temp;
      if (pstack == NULL)
      {
        printf ("Stack underflow \n");
          exit;
      }
else if (pstack->next==pstack) //for only one node.
    { printf ("poped item=%d", pstack->info);
      pstack =NULL;
    }
else
    { temp = pstack->next;
      pstack->next = temp->next;
      printf ("poped item = %d", temp->info);
       free (temp);
    }
}.
```

(b) Queue as a circular list:

It is easier to represent a queue as a circular list than as a linear list. As a linear list a queue is specified by two pointers, one to the front of the list and other to its rear. However, by using a circular list, a queue may be specified by a single pointer q to that list. node(q) is the rear of the queue and the following node is its front.

(a) Insertion function:

```
void insert (int item)
  {
    NodeType *nnode;
    nnode = (NodeType *)malloc (sizeof (NodeType));
    nnode->info = item;
    if (pq == NULL)
            pq=nnode;
    else
        {
          nnode->next = pq->next
          pq->next = nnode;
          pq = nnode;
        }
  }
```

(b) Deletion function:

```
void delete (int item)
    {
      NodeType *temp;
      if (pq == NULL)
      { printf ("void deletion \n");
          exit;
      }
else if (pq->next == pq) //for only one node.
      { printf ("poped item = %d", pq-> info);
        pq = NULL;
      }
else
    { temp = pq-> next;
      pq->next = temp->next;
      printf ("poped item = %d", temp->info);
    } free(temp);
}.
```

⊛ Advantages and disadvantages of circular linked list:

Advantages:

i) In linear linked list it is not possible to go to previous node but in circular linked list we can go to any node.

ii) It saves time when we have to go to the first node from the last node. Since, it can be done in single step.

Disadvantages:

i) It is not easy to reverse the linked list.

ii) If proper care is not taken, then the problem of infinite loop can occur.

iii) If we are not at any node then we can not go to its previous node in single step instead we have to complete the entire circle by going through the between nodes.

What is the difference between Linked List and Linear Array?

S.No.	Array	Linked List
1.	Insertions and deletions are difficult.	Insertions and deletions can be done easily.
2.	It needs movements of elements for insertion and deletion	It does not need movement of nodes for insertion and deletion.
3.	In it space is wasted	In it space is not wasted.
4.	It is more expensive	It is less expensive.
5.	It requires less space as only information is stored	It requires more space as pointers are also stored along with information.
6.	Its size is fixed	Its size is not fixed.
7.	It cannot be extended or reduced according to requirements	It can be extended or reduced according to requirements.
8.	Same amount of time is required to access each element	Different amount of time is required to access each element.
9.	Elements are stored in consecutive memory locations.	Elements may or may not be stored in consecutive memory locations.
10.	If have to go to a particular element then we can reach their directly.	If we have to go to a particular node then we have to go through all those nodes that come before that node.