**Data Structures**

* It is a data organization , management and storage format that enables efficient access and modification.
* Or , a data-structure is a particular way of organizing data in a computer so that it can be effectively used .

**Abstract data types**

* It is made up of primitive data-types , but operation logics are hidden
* Special kind of data-types whose behavior is defined by a set of values and set of instructions.
* Ex :- Lists , stack , queue

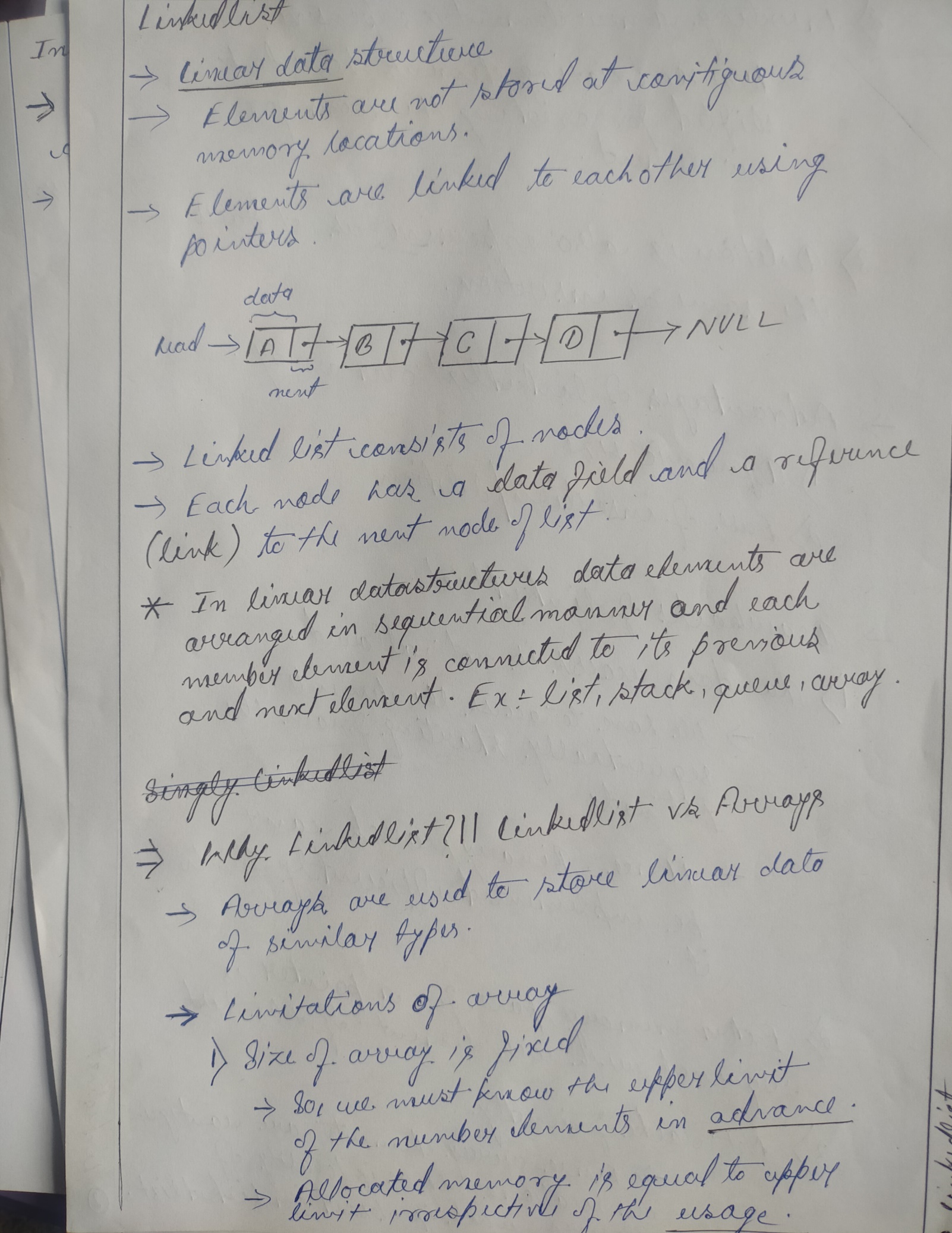
**Linked lists**

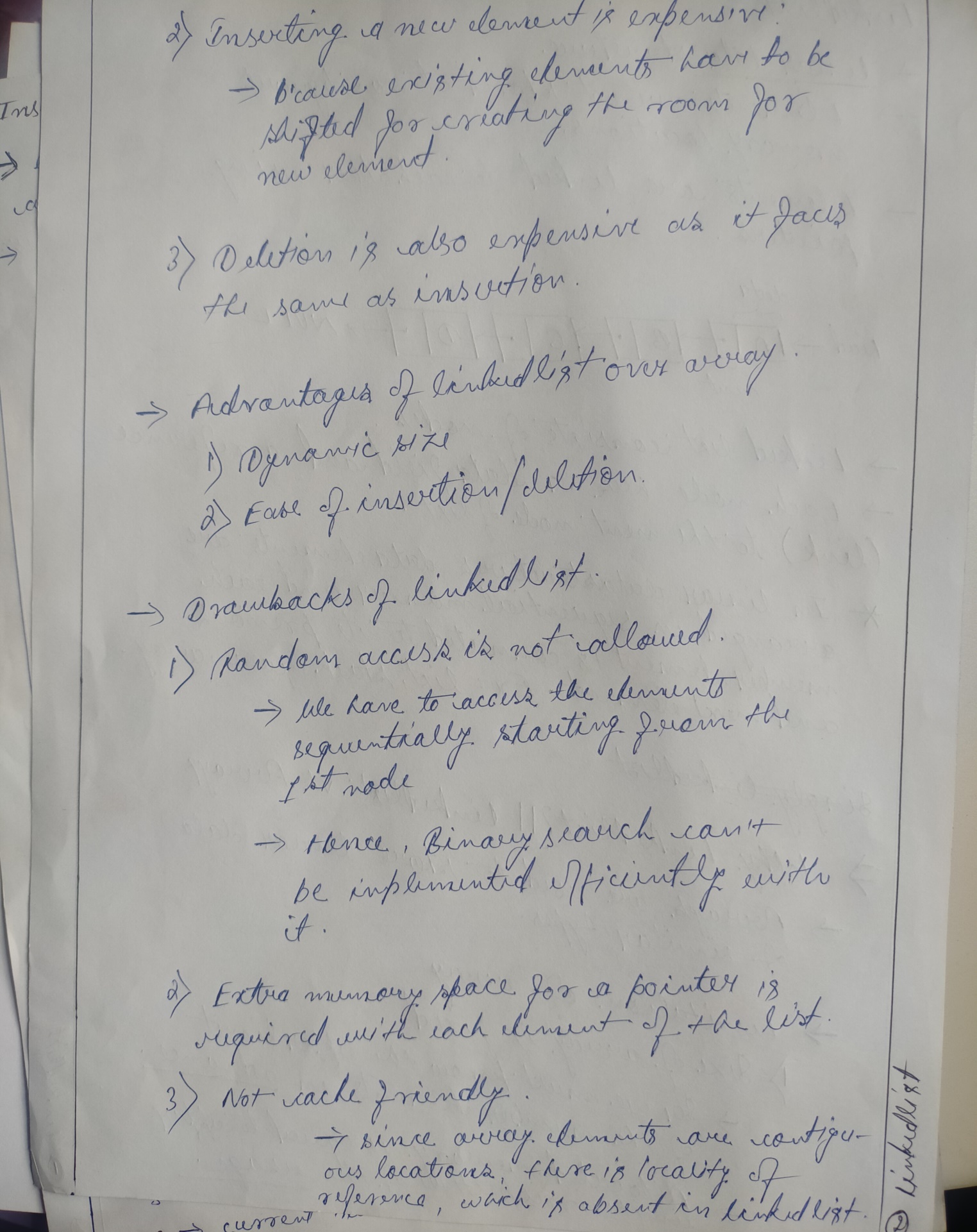
Types of linked lists

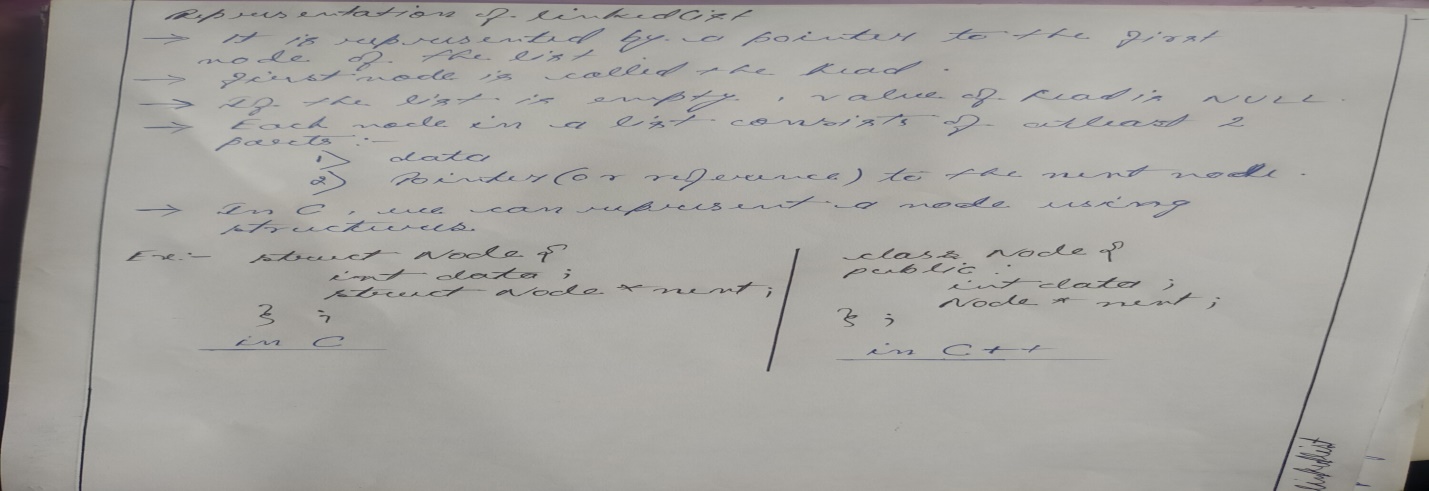
1. Singly Linked list
2. Doubly Linked list
3. Circular Linked list

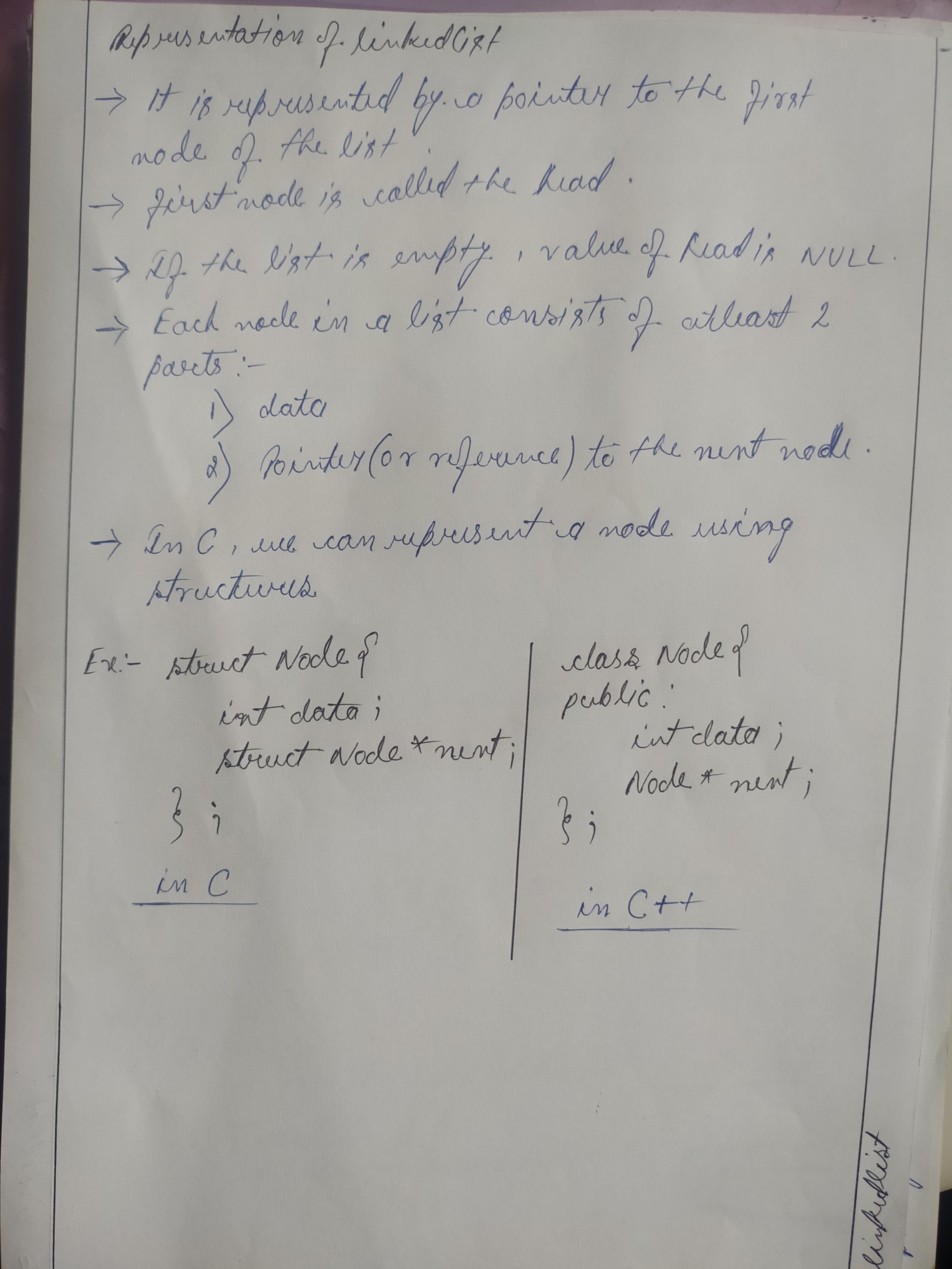
Each type will consist at least following functions to operate on the list-elements

* Create – to create the list (Although , one can create a simple list directly in the main function without any user-defined function)
* InsertAtHead – To insert the element at the beginning of the list
* InsertAtPos – To insert the node at specific position
* InsertAtTail – to insert at the last
* DeleteHead
* DeleteTail
* DeleteElement – Delete the node with specific data
* Reverse
* Traverse – to go through the nodes in order to operate them or print them

****

****

****

****

*- Array is a single block of memory with partition while Linked list is a multiple blocks of memory linked to each other .*

**Singly Linked list**

* It is the standard type of linkedlist , in which there are two fields in each element one is data and another one is next node pointer .
* Unlike doubly linked list , this is an one-way list .

***typedef struct node{***

***int info ;***

***struct node \* next ;***

***We can create a list without a specific create() function also , i.e with insertAtTail() function .***

***}node;***

**Algorithm create(head , n)**

input : head – pointer to the first node of the list (*initially head = NULL*)

n – number of nodes/elements in the list

1 . if head != NULL then

2 . print(“List is already created”)

3 . return

4 . end if

5 . for i <- 1 to n do

6 . input item

7 . newnode <- getNode()

8 . info[newnode] <- item

9 . next[newnode] <- NULL

10 . if head = NULL then

11 . head <- newnode

12 . else

13 . next[temp] <- newnode

14 . temp <- newnode

15 . end for

16 . return

**void create(node \*\* head , int n) Time-comp : O(n)**

{

node \* newnode , \* temp ;

int item , i ;

if(\*head!= NULL)

{

printf(“List created\n”);

return ;

}

for(i=1 ; i<=n ; i++)

{

printf(“Enter item = ”);

scanf(“%d”,&item);

newnode=(node\*)

malloc(sizeof(node));

newnode -> info = item ;

newnode -> next = NULL ;

if((\*head)==NULL)

(\*head) = newnode ;

else

temp -> next = newnode ;

temp = newnode ;

}

return ;

}

**void insertAtHead(node \*\* head , int item)**

Here , head is being/might get modified , hence ***head is passed by reference*** ,instead of value , by double pointer .

Same concept is used in deletion and reverse of the list .

{

if((\*head) == NULL)

{

Time-complexity : O(1)

printf(“List Empty\n”);

return;

}

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

newnode -> info = item ;

**In CPP , we create a node by using class and constructors in it ,**

**class node{**

**int data ;**

**node \* next ;**

(constructor is optional ; it is used to create a node instantly , instead manually)

**node(int val){**

**data = val ;**

**next = NULL ;**

**}**

**};**

**node\* newnode = new node(data) ; 🡨 Node creation in CPP**

newnode -> next = \*head

(\*head) -> newnode ;

return;

}

**void insertAtTail(node\*\*head , int item)**

{

if((\*head) == NULL)

{

printf(“List is empty\n”);

return ;

}

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

newnode->info = item ;

newnode -> next = NULL ;

Time-complexity : O(n)

Can be optimized to O(1) : GFG

node \* loc = \*head ;

while(loc->next!=NULL)

loc = loc->next ;

loc->next = newnode ;

return ;

}

**void traverse(node\*head)**

Time-complexity : O(n)

{

if(head==NULL)

{

printf(“List is empty\n”);

return ;

}

node \* temp = head ;

while(temp!=NULL)

{

printf(“%d ” , temp->info);

temp = temp->next ;

}

return;

}

**int search(node \* head , int key)**

{

if(head == NULL)

{

printf(“List is empty\n”);

Time-complexity : O(n)

Worst case :

Best case :

return 0;

}

node \* temp = head ;

while(temp!=NULL)

{

if(temp->info == key)

return 1;

temp = temp->next ;

}

return 0 ;

}

**void insertAtPos(node\*\*head , int pos , int item)**

{

if(\*head == NULL)

{

Time-complexity : O(n)

Can be optimized to O(1) : GFG

printf(“List is empty\n”);

return ;

}

node \*loc , \*locp , \*newnode ;

preloc = NULL ;

loc = \*head ;

int totalnode = 0 , currentnode ;

while(loc != NULL)

{

totalnode = totalnode + 1 ;

loc = loc -> next ;

}

if(pos > totalnode + 1||pos<=0)

{

printf(“Your element is not there\n”);

return ;

}

currentnode = 1 ;

loc = \*head ;

while(currentnode < pos-1 && loc!=NULL)

{

currentnode +=1 ;

loc = loc->next ;

}

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

newnode->info = item ;

if(pos == 1)

{

newnode->next = \*head ;

\*head = newnode ;

}

else{

newnode->next = loc->next ;

loc->next = newnode ;

}

return ;

}

**int deleteHead(node \*\* head )**

{

if(\*head = NULL)

{

printf(“The list is empty\n”);

return -9999;

}

node \* temp = \*head ;

int val = (\*head)->info ;

\*head = \*head->next ;

temp->next = NULL;

In CPP , instead of **free()** function to delete a node , we use **delete** function . e. g **delete temp ;**

free(temp);

return val;

}

**void deleteTail(node\*\*head)**

{

if(\*head == NULL)

{

printf(“List is empty\n”);

return ;

}

node \*loc , \*locp ;

loc = \*head ;

locp = NULL ;

while(loc->next!=NULL)

{

loc = loc->next;

locp = loc ;

}

printf(“%d is deleted \n” , loc->info);

if(loc==\*head)

\*head = loc->next;

else

locp->next = loc->next;

free(loc);

return;

}

**void deleteElement(node\*\*head , int key)**

Elements can be deleted by specific position also and for this head to doubly linked list

{

if(\*head == NULL)

{

printf(“List is empty\n”);

return ;

}

node \* loc , \*locp ;

loc = \*head ;

locp = NULL ;

while(loc!=NULL && loc->info!=key)

{

loc = loc -> next ;

locp = locp->loc ;

}

if(loc==NULL)

{

printf(“%d is not in the list\n”,key);

return ;

}

else if(loc == (\*head))

\*head = loc->next ;

else

locp->next = loc->next ;

loc ->next = NULL ;

free(loc);

return ;

}

**void reverse(node\*\*head)** *(Iterative method)*

Time-complexity : O(n)

{

if(\*head==NULL || \*head->next ==NULL)

{

printf(“The list is empty or it has only single node\n”);

return ;

}

node \*pre , \*curr , \*nex ;

pre = NULL ;

curr = (\*head) ;

while(loc!=NULL)

{

nex = curr->next ;

curr->next = pre ;

pre = curr ;

curr = nex ;

}

\*head = pre ;

return ;

}

**node \* reverse(node \*head)** (***recursive method)***

{

if(head==NULL || head->next ==NULL)

Time-complexity : O(n)

return head ;

node \* newhead = reverse(head->next);

head->next->next = head ;

head->next = NULL ;

In this function head value is not gonna change , hence we can call the function by value , hence \*\*head is not required

return newhead ;

}

**void insertAfter(node\*head , int data , int pos)**

{

/\*Is to finish: head to **GFG**\*/

}

**Doubly Linked list**

- Each node contains an extra data-field which occupies an extra pointer which holds the address of the previous node .

- The previous pointer of the first node will point over NULL

*Representation of Node in a DLL*

***typedef struct node{***

***int info ;***

***struct node \* next ;***

***struct node \* prev ;***

***}node;***

*Advantages Of DLL over Singly Linked List:-*

- A DLL can be traversed in both forward and backward direction

- Deletion of node in a DLL by the given address/pointer is possible

- We can quickly insert a new node before a given node : In singly linked list to insert a node , pointer of the previous node is needed , which is accessed by traversal . But , In DLL we can get the previous node using previous pointer .

*Disadvantages Of DLL over Singly Linked List:-*

- Each node of DLL require extra space for a previous pointer **(DLL with single pointer is possible : head to GFG)**

- Insertion and Deletion operations require an extra pointer *prevoius* to be maintained which results 1 or 2 extra operations/steps leading to longer time .

**void create(node \*\*head , int n)**

{

if((\*head)!=NULL)

{

printf(“Already created\n”);

return ;

}

*Time-complexity : O(n)*

int i , item;

node \*newnode , \*temp ;

for(i=1;i<=n;i++)

{

printf(“Enter the item = ”);

scanf(“%d”,&item);

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

newnode->info = item ;

newnode->next = NULL;

if(\*head==NULL)

{

\*head = newnode ;

newnode->prev = NULL ;

}

else

{

temp->next = newnode ;

newnode->prev = temp ;

}

temp = newnode ;

}

return ;

}

**void traverse(node\*head)**

{

if(head==NULL)

{

printf(“List is empty\n”);

return ;

}

node \*loc , \*current ;

*Time-complexity : O(n)*

loc = head ;

while(loc!=NULL)

{

printf(“%d”,loc->info);

current = loc ;

loc = loc->next ;

}

loc = current ;

while(loc!=NULL)

{

printf(“%d”,loc->info);

loc = loc->prev ;

}

return ;

}

**void insertAtHead(node\*\*head , int item)**

{

if(\*head == NULL)

*Time-complexity : O(1)*

{

printf(“List is empty\n”);

return;

}

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

newnode->info = item ;

newnode->next = (\*head) ;

newnode->prev = NULL ;

\*head->prev = newnode ;

\*head = newnode ;

return ;

}

**void insertAtTail(node\*head , int item)**

{

if(head==NULL)

{

printf(“List is empty\n”);

return ;

}

node \*newnode , \*temp ;

*Time-complexity : O(n)*

temp = head ;

while(temp->next != NULL)

{

temp = temp->next;

}

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

temp->next = newnode ;

newnode->next = NULL;

newnode->prev = temp ;

return ;

}

**void insertAtPosition(node\*\*head , int data , int pos)**

{

if(\*head == NULL)

*Time-complexity(Worst case) : O(n)*

*Best-case : O(1)*

{

printf(“List is empty”);

return ;

}

node \*loc , \*newnode ;

int totalNode , currentNode ;

totalNode = 0 ;

loc = \*head ;

while(loc!=NULL)

{

totalNode++ ;

loc = loc->next ;

}

if(pos<=0 || pos > totalNode+1)

{

printf(“Invalid position entered\n”);

return ;

}

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

newnode->info = data ;

loc = \*head ;

currentNode = 1 ;

while(currentNode < pos – 1 && loc!=NULL)

{

loc = loc->next ;

currentNode++;

}

if(pos==1)

{

newnode->next = \*head ;

newnode->prev = NULL ;

\*head = newnode ;

}

else

{

newnode->next = loc->next ;

newnode->prev = loc ;

loc->next = newnode ;

}

return ;

}

**void insertAfter(node\*\*head , int data , int pos) /\* FINISH THESE : HEAD TO GFG \*/**

**void insertBefore(node \*\*head , int data , int pos)**

**int deleteHead(node\*\*head)**

{

if(\*head == NULL)

*Time-complexity : O(1)*

{

printf(“List is empty\n”);

return -9999 ;

}

node \* temp = \*head ;

int deletedData = temp->info;

\*head = \*head->next ;

\*head->prev = NULL ;

temp->next = NULL ;

free(temp) ;

return deletedData;

}

**int deleteTail(node\*\*head)**

{

if(\*head == NULL)

{

printf(“List is Empty\n”);

return -9999;

*Time-complexity : O(n)*

}

node \* temp = \*head ;

while(temp->next!=NULL)

{

temp = temp->next ;

}

int deletedData = temp->info ;

if(temp==\*head)

\*head = temp->next ;

else

{

temp->prev->next = temp->next ;

temp->prev = NULL ;

}

free(temp) ;

return deletedData ;

Elements can be deleted by specific data/key also and for this head to singly linked list

}

**int deleteAtPos(node\*\*head , int pos)**

{

if(\*head == NULL)

{

printf(“List is empty\n”);

return ;

}

int deletedData , count;

node \* temp ;

if(pos==1)

deletedData = deleteHead(head);

else

{

temp = \*head ;

count = 1 ;

while(temp!=NULL && count != pos)

{

temp=temp->next;

count++ ;

}

deletedData = temp->info ;

temp->prev->next = temp->next ; // last node deletion is covered by this statement only

if(temp->next!=NULL)

temp->next->prev = temp->prev ;

temp->next = temp->prev = NULL ;

free(temp);

}

return deletedData ;

}

**int deleteAtPos(node\*\*head , int pos)**

**{**

**if(\*head == NULL)**

**{**

**cout << "List is empty" <<endl ;**

**return -9999 ;**

**}**

**int item , count ;**

**node \*temp = (\*head);**

**if(pos == 1)**

**{**

**item = (\*head)->data ;**

**\*head = (\*head)->next ;**

**(\*head)->prev = NULL ;**

**}**

**else**

**{**

**count = 1 ;**

**while(temp!=NULL && count!=pos)**

**{**

**temp = temp->next ;**

**count ++ ;**

**}**

**item = temp->data ;**

**temp->prev->next = temp->next ;**

**if(temp->next!=NULL)**

**temp->next->prev = temp->prev ;**

**temp->next = temp->prev = NULL ;**

**}**

**delete temp ;**

**return item ;**

**}**

**Another way to write the deleteAtPos() function**

**Circular Linked list**

- There is no NULL at the end .

- next of last node is pointing over first node

*Advantages*

- Any node can be a starting point .

- We can traverse the whole list by starting from any point and we need to stop the first visited node is visited again .

***visit gfg to explore more advantages and applications***

**typedef struct node{**

**int info ;**

**struct node \*next ;**

**}nod;**

**void create(nod\*\*head , int n)**

{

if(\*head!=NULL)

{

printf(“List is already created\n”);

return ;

}

*Time-complexity : O(n)*

int i ,item;

nod \*temp , \*newnode ;

for(i=1 ; i<=n ; i++)

{

printf(“Enter the node data = ”);

scanf(“%d”,&item);

newnode = (nod\*)malloc(sizeof(nod));

newnode->info = item ;

if(\*head==NULL)

\*head = newnode ;

else

temp->next = newnode ;

temp = newnode ;

}

newnode->next = (\*head);

return ;

}

**void display(nod\*head)**

{

if(head==NULL)

*Time-complexity : O(n)*

{

printf(“List is empty”);

return ;

}

nod\*temp = head ;

do{

printf(“%d ”, temp->info);

temp = temp->next ;

}while(temp!=head);

printf(“\n”);

return ;

}

**void insertAtHead(nod\*\*head , int item)**

{

if((\*head)==NULL)

{

printf(“List is Empty\n”);

return ;

}

nod \*newnode , \*temp ;

newnode = (nod\*)malloc(sizeof(nod));

*Time-complexity : O(n)*

newnode->info = item ;

temp = (\*head) ;

while(temp->next!=(\*head))

{

temp = temp->next ;

}

temp->next = newnode ;

newnode->next = (\*head) ;

Only head pointer determines the first and last node in a circular linked list , otherwise rest code is same for inserting at head and tail both

(\*head) = newnode ;

return ;

}

**void insertAtTail(nod\*\*head , int item)**

{

if(\*head == NULL)

{

printf(“List is empty”);

return ;

}

*Time-complexity : O(n)*

nod \*newnode , \*temp ;

newnode = (nod\*)malloc(sizeof(nod)) ;

newnode->info = item ;

temp = (\*head) ;

while(temp->next != (\*head))

{

temp = tem->next ;

}

temp->next = newnode ;

newnode->next = (\*head) ;

return ;

}

**int deleteHead(nod\*\*head)**

{

if(\*head==NULL)

{

printf(“List is empty\n”);

return -9999 ;

}

*Time-complexity : O(n)*

nod \*temp , \*temp2 ;

temp = (\*head) ;

temp2 = (\*head);

while(temp->next!=(\*head))

{

temp = temp->next ;

}

int item = \*head->info ;

if(temp == \*head)

\*head = NULL ;

else

{

\*head = (\*head)->next ;

temp->next = \*head ;

}

temp2->next = NULL ;

free(temp2);

return item;

}

**int deleteTail(nod\*\*head)**

{

if(\*head==NULL)

{

printf(“List is empty\n”);

return -9999 ;

}

nod \*pre , \*cur ;

pre = NULL ;

cur = \*head ;

while(cur->next!=(\*head))

*Time-complexity : O(n)*

{

pre = cur

cur = cur->next ;

}

int item = cur->info ;

if(cur == \*head)

\*head = NULL ;

else

pre->next = \*head ;

cur->next = NULL ;

free(cur) ;

return item ;

}

**int count(node\*head)**

{

node\*temp = head ;

int nodeCount = 1 ;

*Time-complexity : O(n)*

while(temp-next!=head)

{

temp=temp->next ;

nodeCount++ ;

}

return nodeCount ;

}

*Time-complexity : O(n)*

*best case : O(1)*

*worst case : O(n)*

**int deletion(node\*\*head , int pos)**

{

if(\*head == NULL)

{

printf(“List is empty\n”);

return -9999 ;

}

if(pos<=0 || pos>count(\*head))

{

printf(“Invalid position\n”);

return -9999 ;

}

if(pos == 1)

return deleteHead(head);

node \*pre , \*toDelete ;

pre = NULL ;

toDelete = \*head ;

int cnt = 1 ;

while(cnt!=pos)

{

***Another approach***

node \* temp = \*head

cnt = 1 ;

while(cnt!=pos-1)

{

temp=temp->next ;

cnt++ ;

}

node \*toDelete = temp->next ;

int item = toDelete->info ;

temp->next = toDelete->next ;

toDelete->next =NULL ;

free(toDelete);

pre = toDelete ;

toDelete = toDelete->next ;

cnt++ ;

}

int item = toDelete->info ;

pre->next = toDelete->next ;

toDelete->next = NULL ;

free(toDelete);

return item ;

}

**void insertAtPosition(node\*\*head , int item , int pos)**

{

if(\*head == NULL)

{

printf(“List is empty\n”);

return ;

}

int totalNode , cnt ;

nod \*temp , \*newnode ;

temp = \*head ;

totalNode = 0 ;

do

{

you can use count() method here

totalNode++ ;

temp = temp ->next ;

} while(temp!=(\*head)) ;

if(pos <=0 || pos>totalNode + 1)

{

printf(“Invalid position\n”);

return ;

}

temp = (\*head) ;

cnt = 1 ;

while(cnt !=pos-1)

{

temp = temp->next ;

cnt++ ;

}

newnode = (nod\*)malloc(sizeof(nod));

newnode->info = item ;

if(pos == 1)

{

nod \*loc = \*head ;

while(loc->next != \*head)

you can use insertAtHead() method here

{

loc = loc->next ;

}

loc->next = newnode ;

newnode->next = (\*head) ;

\*head = newnode ;

time-complexity : O(n)

}

else{

newnode->next = temp->next ;

temp->next = newnode ;

}

return ;

}

**void revese(node\*\*head)**

{

if(\*head == NULL || (\*head) -> next == (\*head))

{

printf(“List is empty of it has only single node”);

return ;

}

nod \*prev , \*curr , \*nex ;

time-complexity : O(n)

best case : O(1) , when head=Null or there is a single node

prev = NULL ;

curr = (\*head) ;

do

{

nex = curr -> next ;

curr->next = prev ;

prev = curr ;

curr = nex ;

}while(curr!=(\*head));

(\*head)->next = prev ;

\*head = prev ;

return ;

}

**Stack**

- It is a linear datastructure , which stores a list of items in which an item can be added or removed from a single end only .

- It follows the LIFO(Last in First Out)/FILO order

- The current-accessible end is called the top

- All the operations below take O(1) time as there is no any loop present

***head to GFG for more explanations and applications of stack , pros&cons***

Operations :-

1.isEmpty() 2 .isFull 3.push() 4.pop() 5.top()

**Implementation of Stack using Array**

***#define STACKSIZE 100***

***In C++ , implementation of stack***

***class stack{***

***int top ;***

***public:***

***int arr[MAX];***

***stack(){***

***top= -1;***

***bool isEmpty();***

***bool isFull();***

***void push(int x);***

***int pop();***

***int Top();***

***}***

***};***

***typedef struct stk{***

***int arr[STACKSIZE] ;***

***int top ;***

In CPP , the implementation is slightly different , as we don’t use the pointers in it and some other stuffs

Function definitions:-

**bool stack::isEmpty()**

**bool stack::isFull()** likewise...

***}stack;***

int isEmpty(stack\*p)

{

if(p->top == -1) return 1 ;

else return 0;

}

int isFull(stack\*p)

{

if(p->top == MAX-1) return 1;

else return 0;

}

void push(stack\*p , int item)

{

if(isFull(p))

{

printf(“OVERFLOW\n”);

return ;

}

p->top++ ;

p->arr[p->top] = item ;

return;

}

int pop(stack\*p)

{

if(isEmpty(p))

{

printf(“UNDERFLOW\n”);

return -9999 ;

}

int item = p->arr[p->top];

p->top-- ;

return item ;

}

int Top(stack\*p)

{

if(isEmpty(p))

{

printf(“UNDERFLOW\n”);

return -9999;

}

return p->arr[p->top];

}