**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

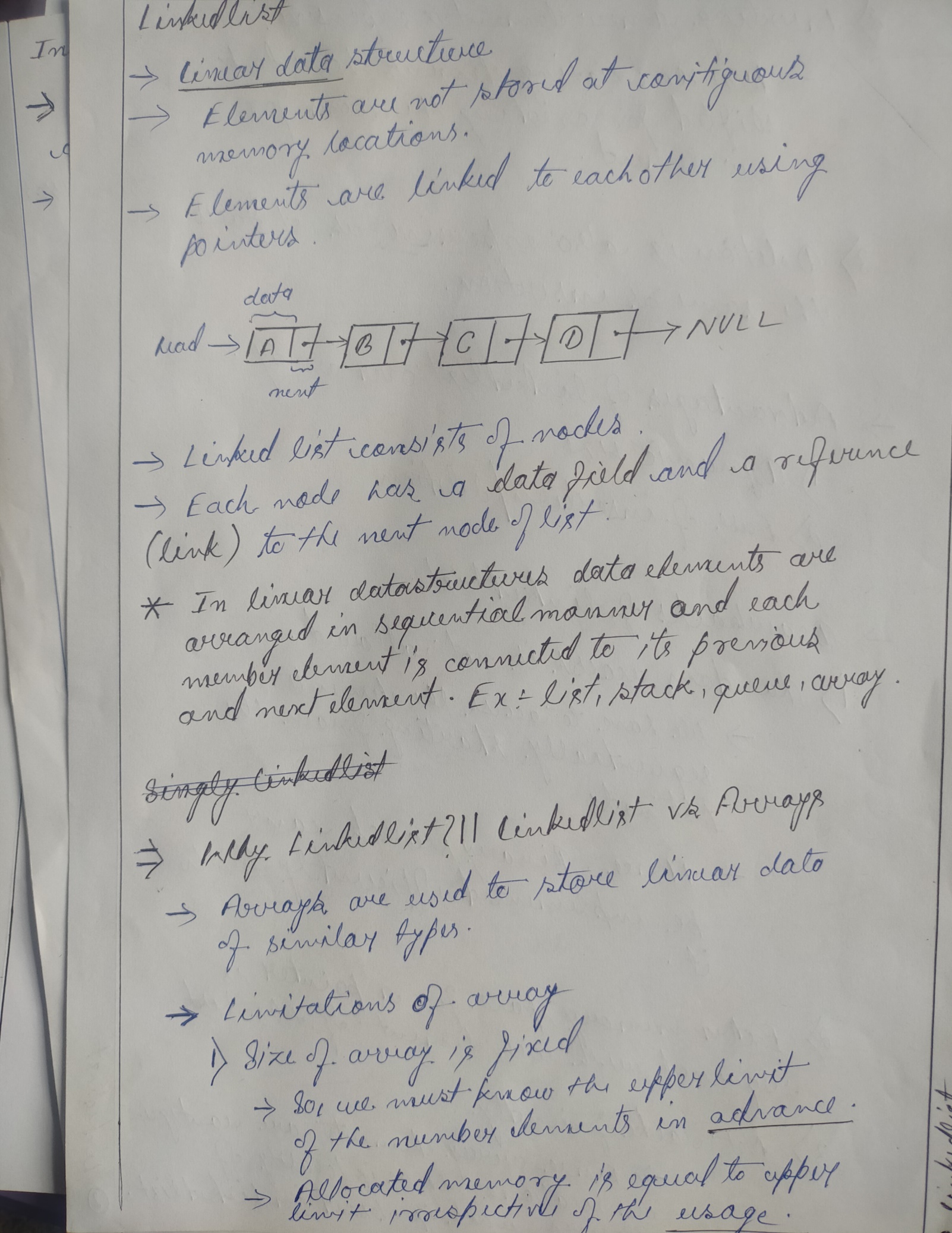
**Linked lists**

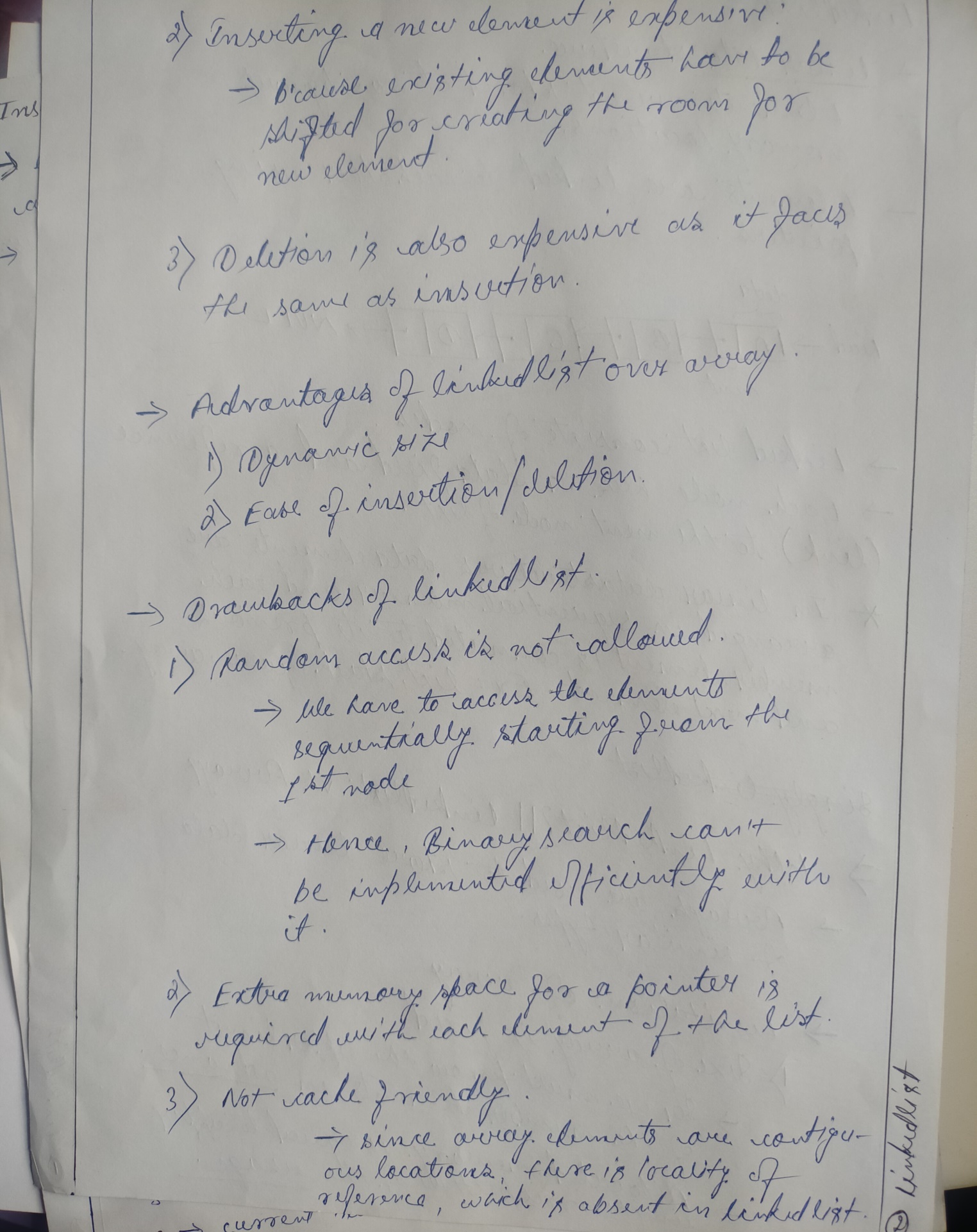
Types of linked lists

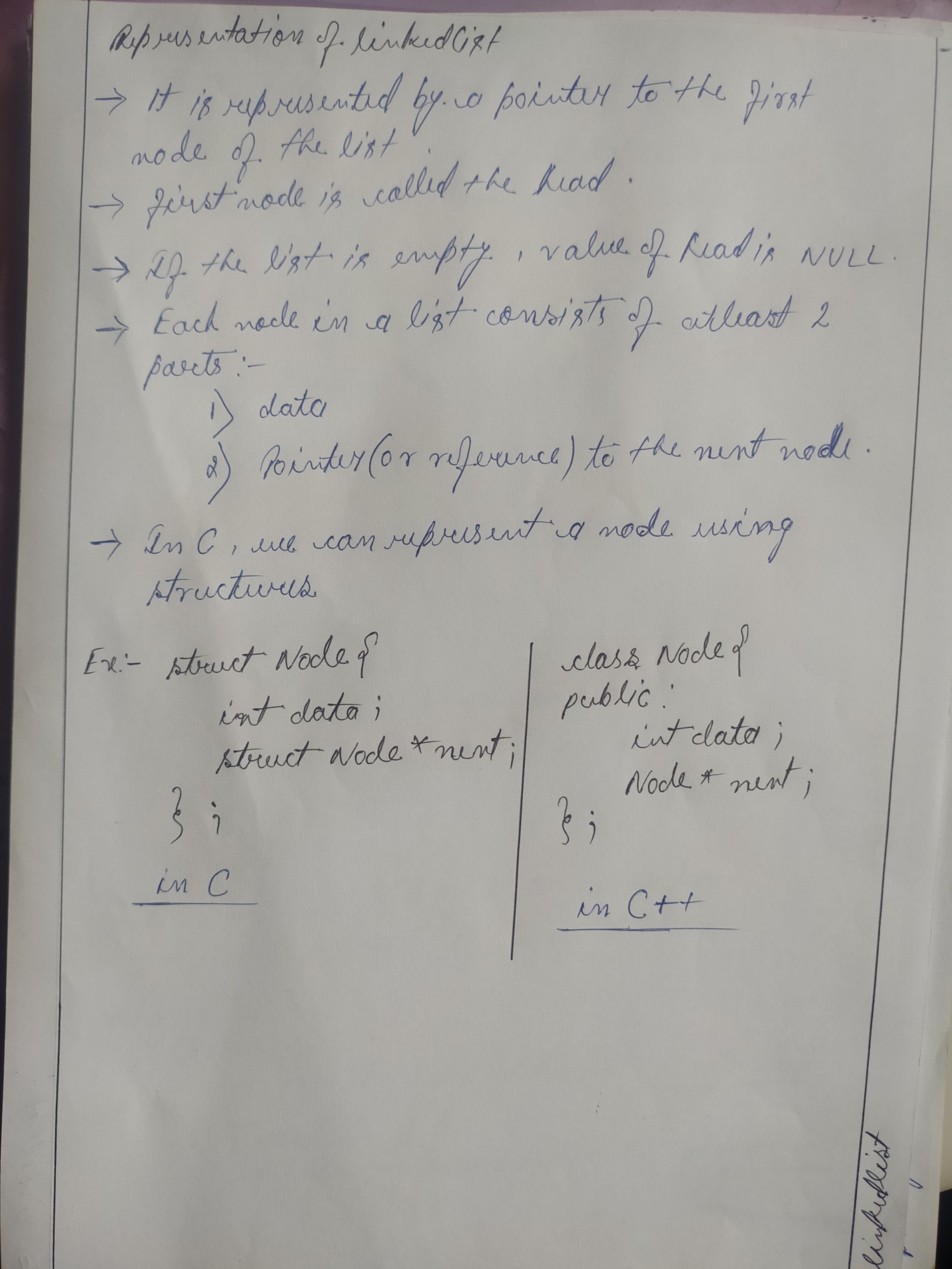
1. Singly Linked list
2. Doubly Linked list
3. Circular Linked list

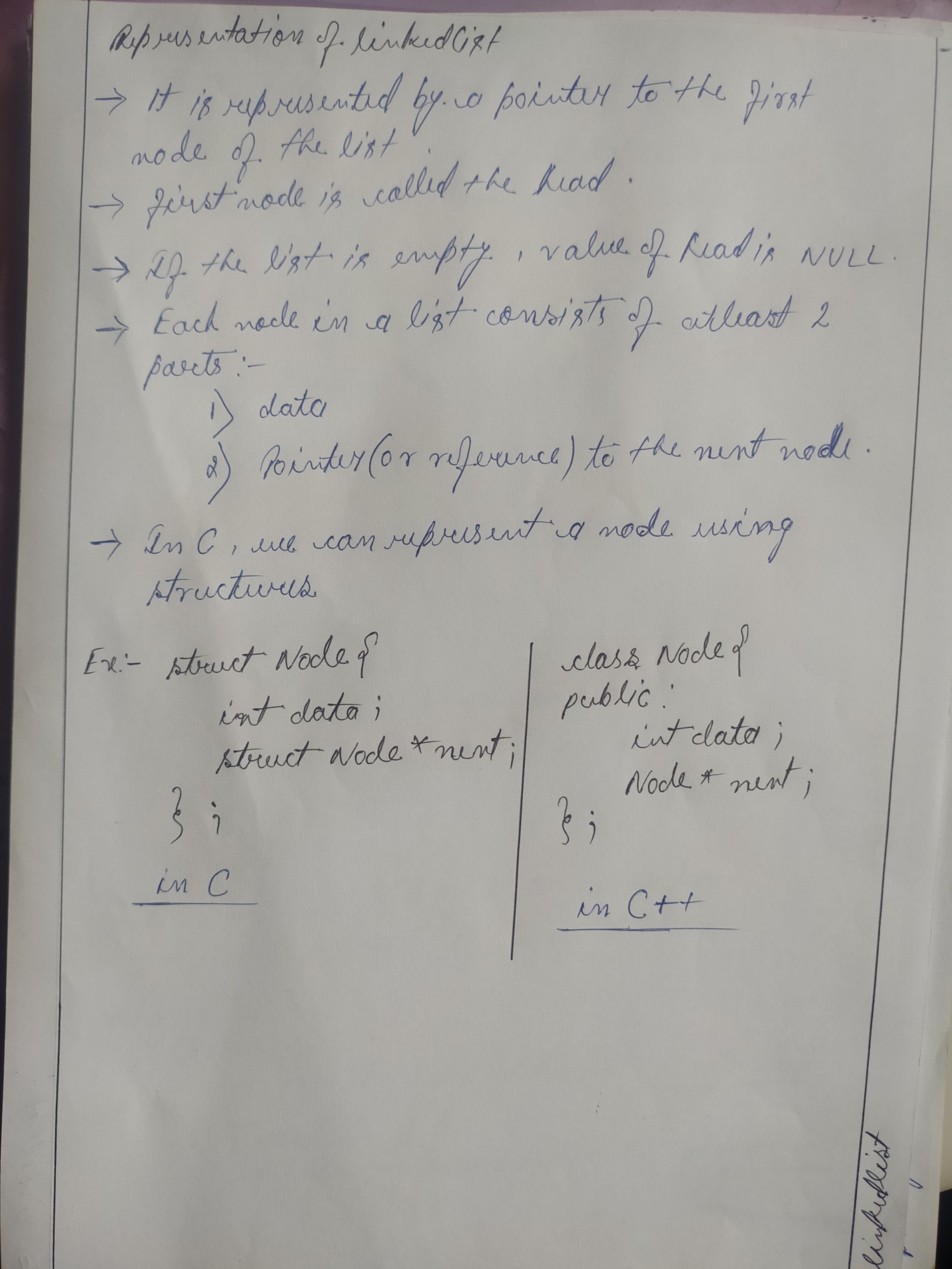
Each type will consist at least following functions to operate on the list-elements

* 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 ;***

***}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)**

{

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 , list is being 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)

{

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 ;

node \* loc = \*head ;

while(loc->next!=NULL)

loc = loc->next ;

loc->next = newnode ;

return ;

}

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

{

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”);

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)

{

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(loc==\*head)

{

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)**

{

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)*

{

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)

return head ;

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

head->next->next = head ;

head->next = NULL ;

return newhead ;

}