**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/primary 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

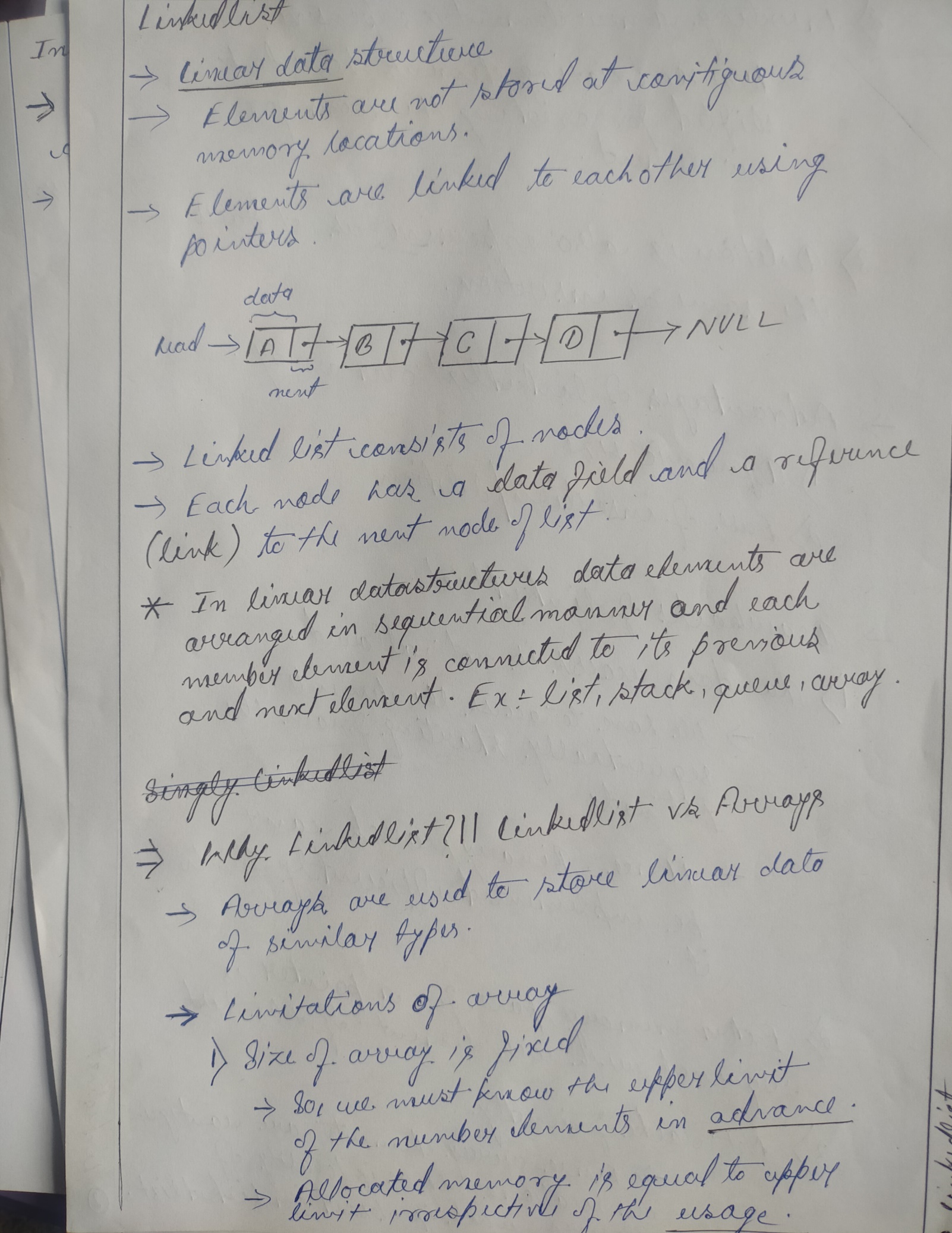
**Linked lists**

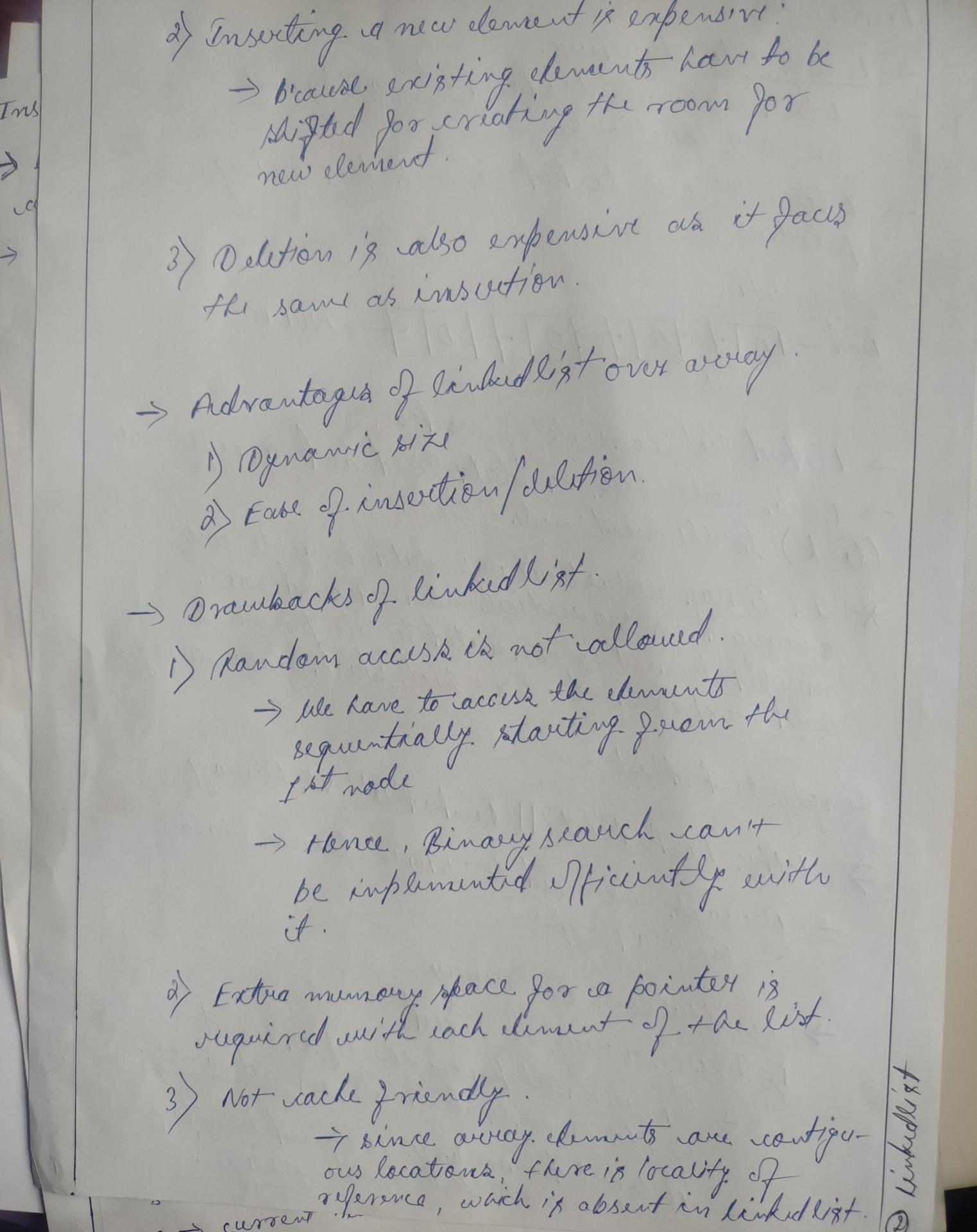
Types of linked lists

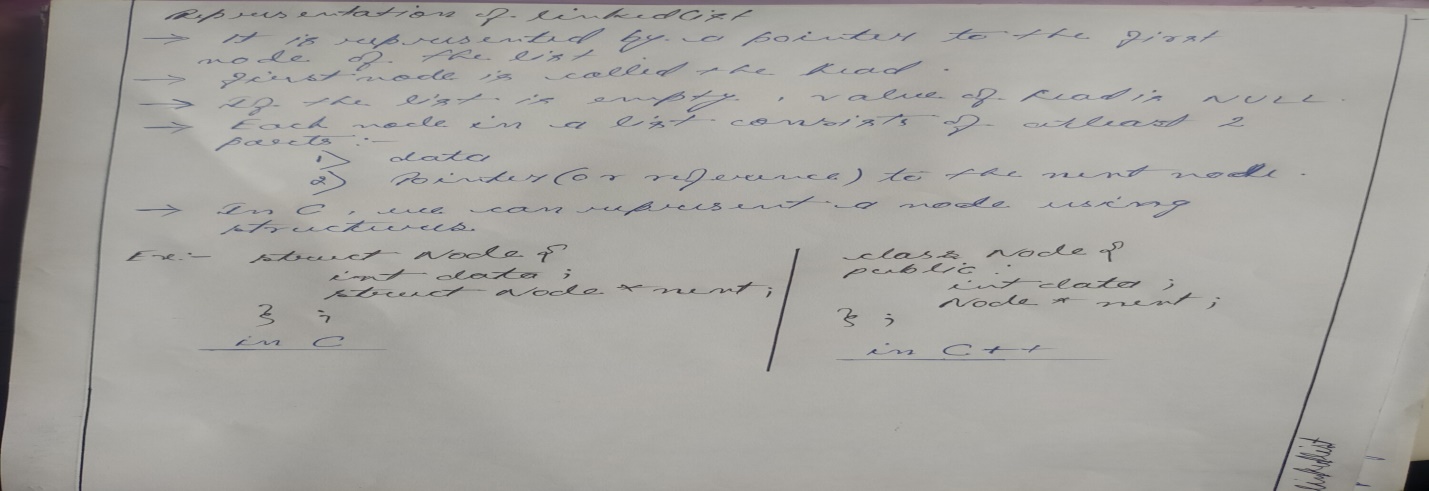
1. Singly Linked list
2. Doubly Linked list
3. Circular Linked list

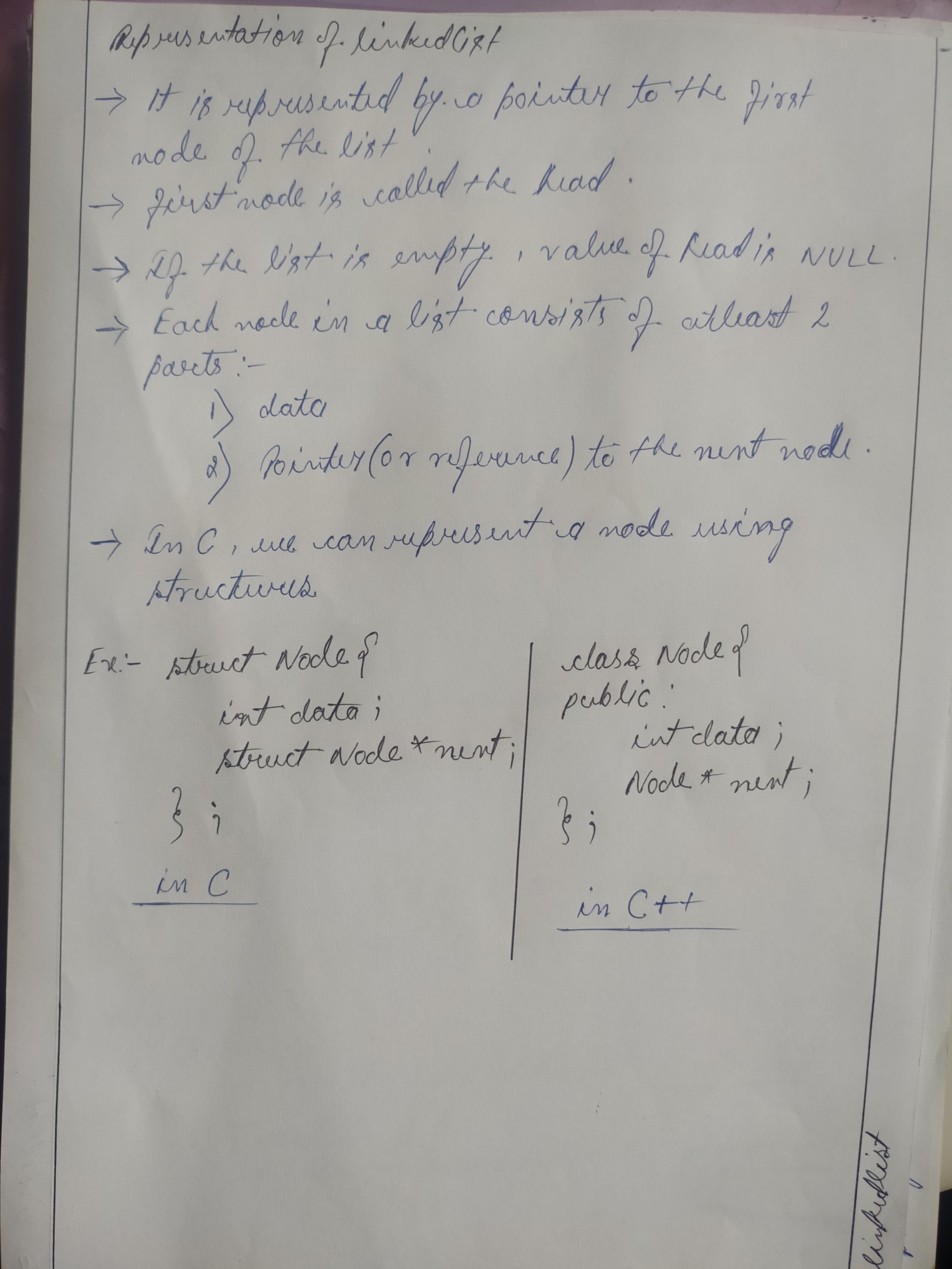
Each type will consist at least following functions to operate on the list-elements

* 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;

**/\*We can create a list in this block only , by assigning head to newnode\*/**

}

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 ;

**/\*We can create a list in**

**this block only ,**

**by assigning head to newnode\*/**

}

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 , \*newnode ;

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)

{

locp = loc ;

loc = loc->next;

}

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)

{

locp = loc ;

loc = loc -> next ;

}

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(curr!=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 )->prev = newnode ;

( \*head) = newnode ;

}

else

{

newnode->next = loc->next ;

newnode->next->prev = newnode ;

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 see 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); <-**OR one can write whole delethead() function here**

else

{

temp = \*head ;

count = 1 ;

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

{

temp=temp->next;

count++ ;

}

if(temp==NULL) {

printf(“Invalid Position\n”);

return ;

}

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 ++ ;**

**}**

**if(temp==NULL) {**

**printf(“Invalid Position\n”);**

**return ;**

**}**

**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 ;

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

newnode->next = (\*head) ;

**(\*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 ; r

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 or 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 (while using array and linkedlist both)

***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**

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...

***#define STACKSIZE 100***

***typedef struct stk{***

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

***int top ;***

***}stack;***

***int main()***

***{stack st ; stack \*p = &st ; p->top = -1;}***

int isEmpty(stack\*p)

{

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

***class stack{***

***int top ;***

***public:***

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

***stack(){***

***top= -1;}***

***bool isEmpty();***

***bool isFull();***

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

***int pop();***

***int Top();***

***};***

***In this way we don’t use stack pointers , like*** *stack\*p = &st ;* ***, in***

***main() function . Hence , We also don’t have to pass the pointer argument to each functions***

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];

}

**Implementation of Stack using Linkedlist**

Advantages:-

a . It is possible to implement a stack that can shrink or grow as much as needed as In linked list implementation each new node is dynamically allocated unlike array implementation in which the max-capacity is pre-fixed .

c . Since , using linkedlist the maximum size isn’t pre-fixed hence , the possibility of OVERFLOW doesn’t occur and hence we don’t need to check for it on every push operation .

**typedef struct node{**

**int data ;**

**struct node \*next ;**

**tp is the node pointer pointing over the last pushed node and it would be initially NULL initialised in main() i.e .** nod \*tp = NULL

**}nod;**

**void push(nod \*\*tp , int item)**

{

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

newnode->data = item ;

newnode->next = (\*tp) ;

\*tp = newnode ;

return ;

}

**void pop(nod \*\*tp)**

{

if(empty(\*tp)) return;

nod \*temp = (\*tp) ;

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

temp->next = NULL ;

free(temp);

return;

}

**int top(nod \*tp)**

{

if(empty(tp)) return tp ;

return tp->data ;

}

**int empty(nod \*tp)**{ return tp==NULL ;}

**Queue**

- Linear Data Structure

- Deletions can take place only at one end , denoted by the pointer **front**

- Insertions can take place at the other end , denoted by the pointer **back/rear**.

- It uses **FIFO** order of operations , as first element inserted would be taken out first .

- It has O(1) complexity for all operations and as well in case of **deque**

Operations in QUEUE :-

enQue() - an element is inserted at back position.

deQue() - element discarded from the the front position.

peek() - It returns the front value

empty() - it tells whether the queue is empty or not .

**Linear Array implementation**

**typedef struct queue{**

**int front , rear ;**

**int arr[MAX];**

**}que;**

int main()

{

que qp ;

que \*q = &qp ;

q->front = -1 ; q->rear = -1 ;

enQue(q,1);

enQue(q,2);

}

**void enQue(que\* q , int n)**

{

if(q->rear == MAX-1) // This can be managed by creating a specific isFull() function

{

printf(“OVERFLOW\n”);

return ;

}

q->arr[++(q->rear)] = n ;

if(q->front == -1){q->front++;}

}

**void deQue(que\* q)**

{

if(empty(q))

{

printf(“UNDERFLOW\n”);

return ;

}

q->front++ ;

}

**int empty(que \*q)**

{

if(q->front == -1 || q->front > q->rear) return 1 ;

else return 0;

}

**int peek(que \*q)**

{

if(empty(q))

{

printf(“UNDERFLOW\n”);

return -9999 ;

}

**These vacant positions can’t be filled in linear array implementation**

return q->arr[q->front];

front

rear

}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | 30 | 40 | 25 | 60 |

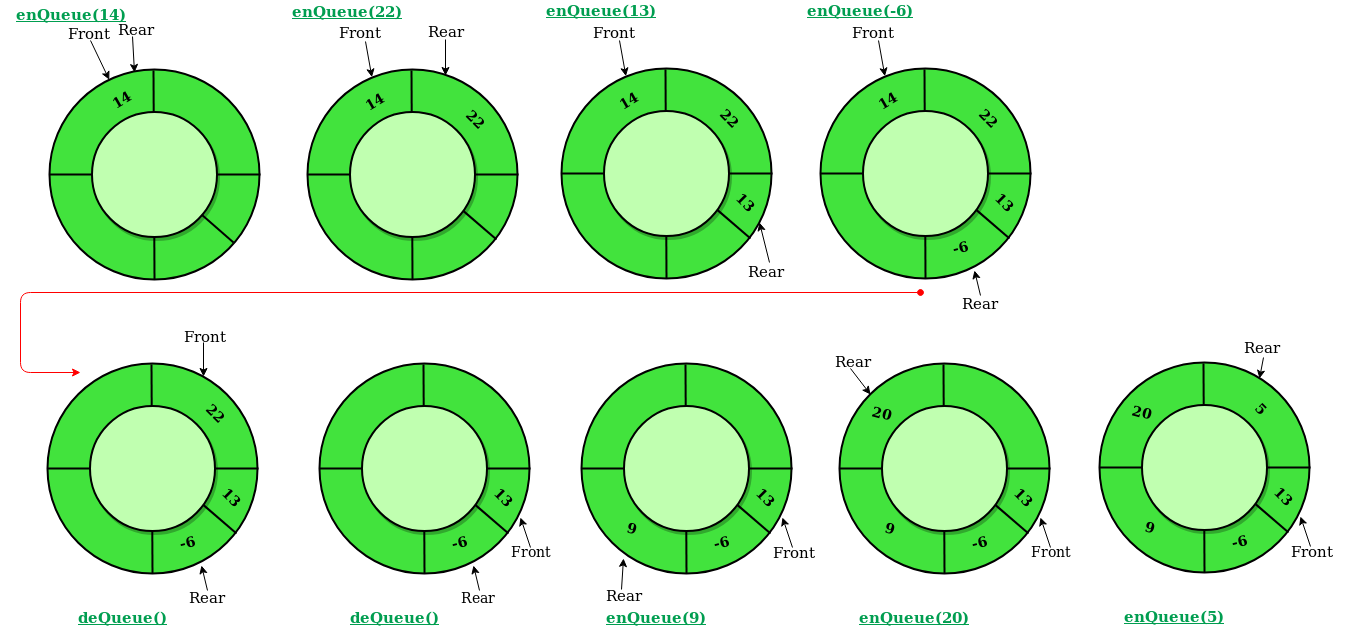
**Limitations of Linear Array implementation of Queue:-**

- If the last position of queue is occupied , it is not possible to insert any element any more in the queue even though some positions are vacant towards the front position of the queue .

- In order to overcome this limitation we can use **Circular Array Implementation**

**Circular Array Implementation**

- Almost similar to linear array , only the last position of the array is connected back to the first position to make a circle .

**int empty(que \*q)**

{

if(q->front == -1 && q->rear == -1) return 1 ;

else return 0 ;

}

**int full(que \*q)**

{

if(q->front == 0 && q->rear == MAX -1 || q->front = q->rear+1) return 1;

else return 0 ;

}

**int peek(que \*q)**

{

if(empty(q)) return -9999;

else return q->arr[q->front];

}

**void** **enQue(que \*q , int n)**

{

if(full(q))

{

printf(“OVERFLOW\n”);

return ;

}

if(empty(q))

{

q->front = q->rear = 0 ;

}

else if(q->rear == MAX-1) q->rear = 0;

else q->rear++ ;

q->arr[q->rear] = n ;

}

**void deQue(que \*q)**

{

if(empty(q))

{

printf(“UNDERFLOW\n”);

return ;

}

if(q->front == q->rear)

{

q->front = q->rear = -1 ;

}

else if(q->front == MAX-1) q->front = 0 ;

else q->front++ ;

return ;

}

**Singly Linkedlist Implementation**

Drawbacks of a sequential(array) representation of a queue data structure:-

- Finite capacity of queue

- Checking for Overflow condition every time an enQue() operation is done .

**typedef struct node{**

**int info ;**

**struct node \*next ;**

**}nod;**

**int main()**

**{**

**nod \*front , \*rear ;**

**front = rear = NULL ;**

**enQue(&front , &rear , 10);**

**enQue(&front , &rear , 8);**

**enQue(&front , &rear , 5) ;**

**deQue(&front , &rear);**

**printf(“%d”,peek(front));**

**}**

**void enQue(nod \*\*f, nod \*\*r , int n)**

{

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

newnode->info = n ;

newnode->next = NULL ;

if(\*f == NULL)

{

\*f = newnode ;

\*r = newnode ;

}

else

{

\*rear->next = newnode;

\*rear = newnode ;

}

return ;

}

**void deQue(nod \*\*f , nod \*\*r)**

{

if(\*f == NULL)

{

printf(“UNDERFLOW\n”);

return ;

}

nod \*temp = \*f ;

\*f = \*f->next ;

temp->next = NULL;

if(temp == \*r) \*r = NULL;

free(temp);

return ;

}

**int peek(nod \*f)**

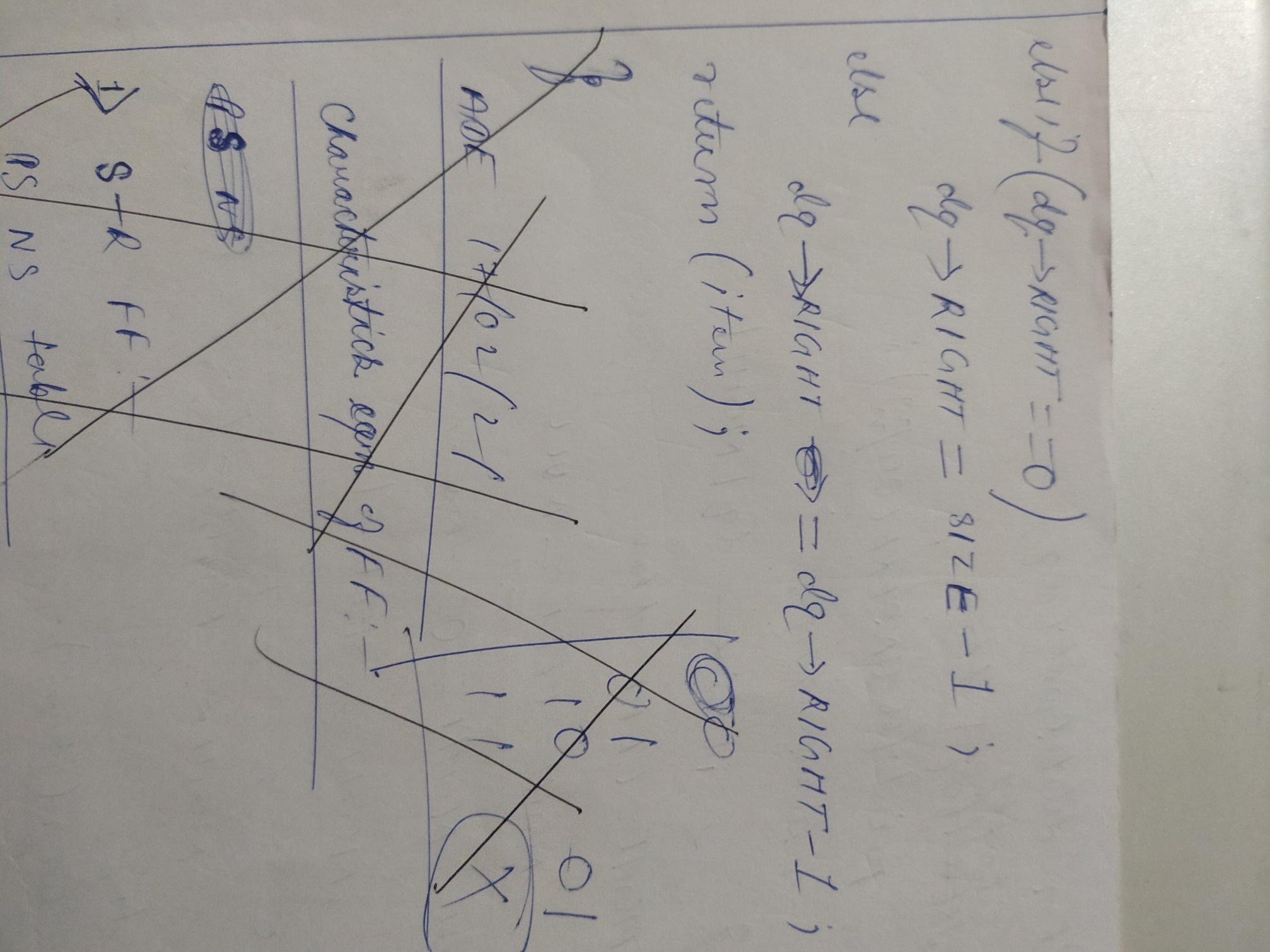
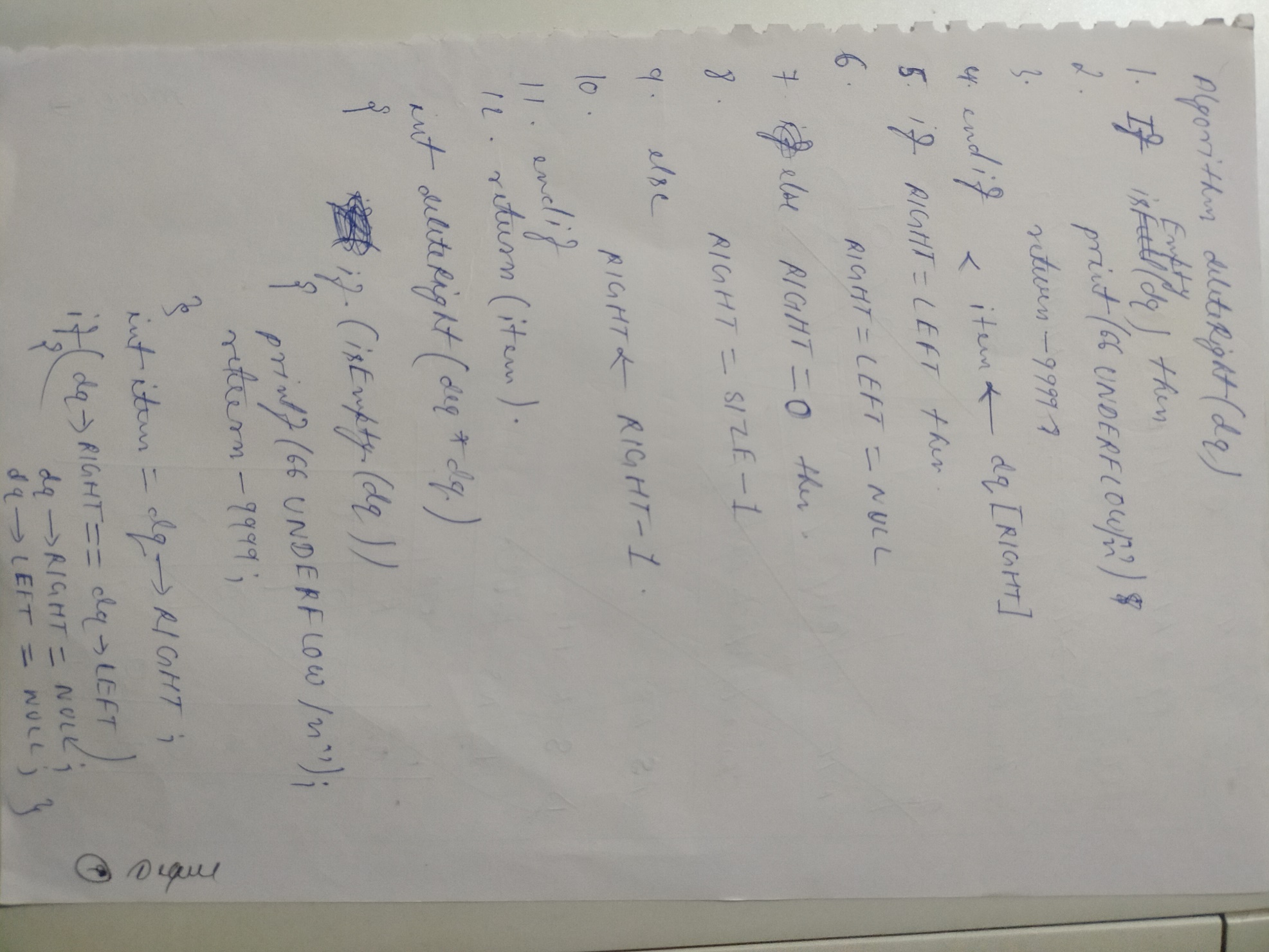
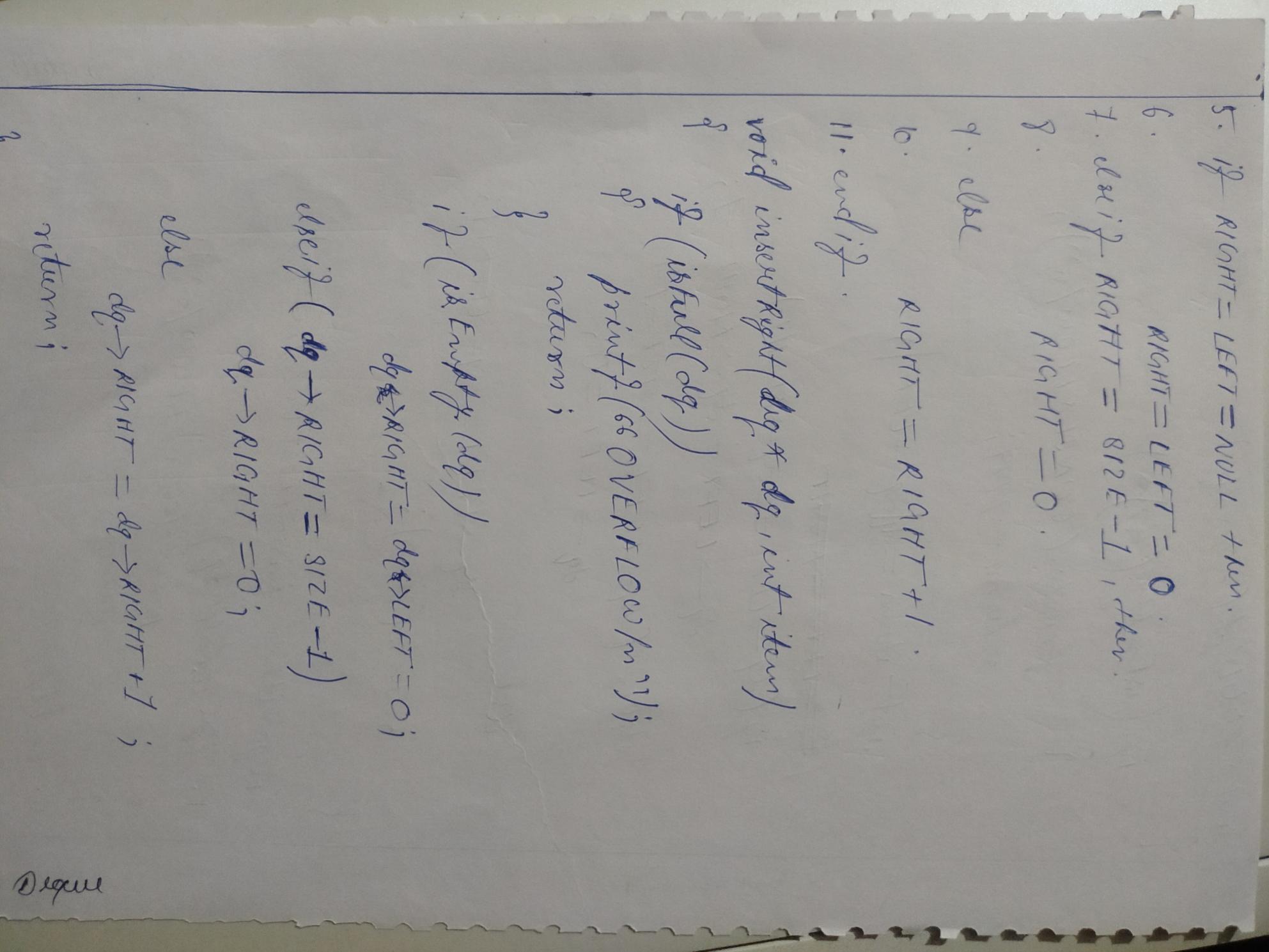
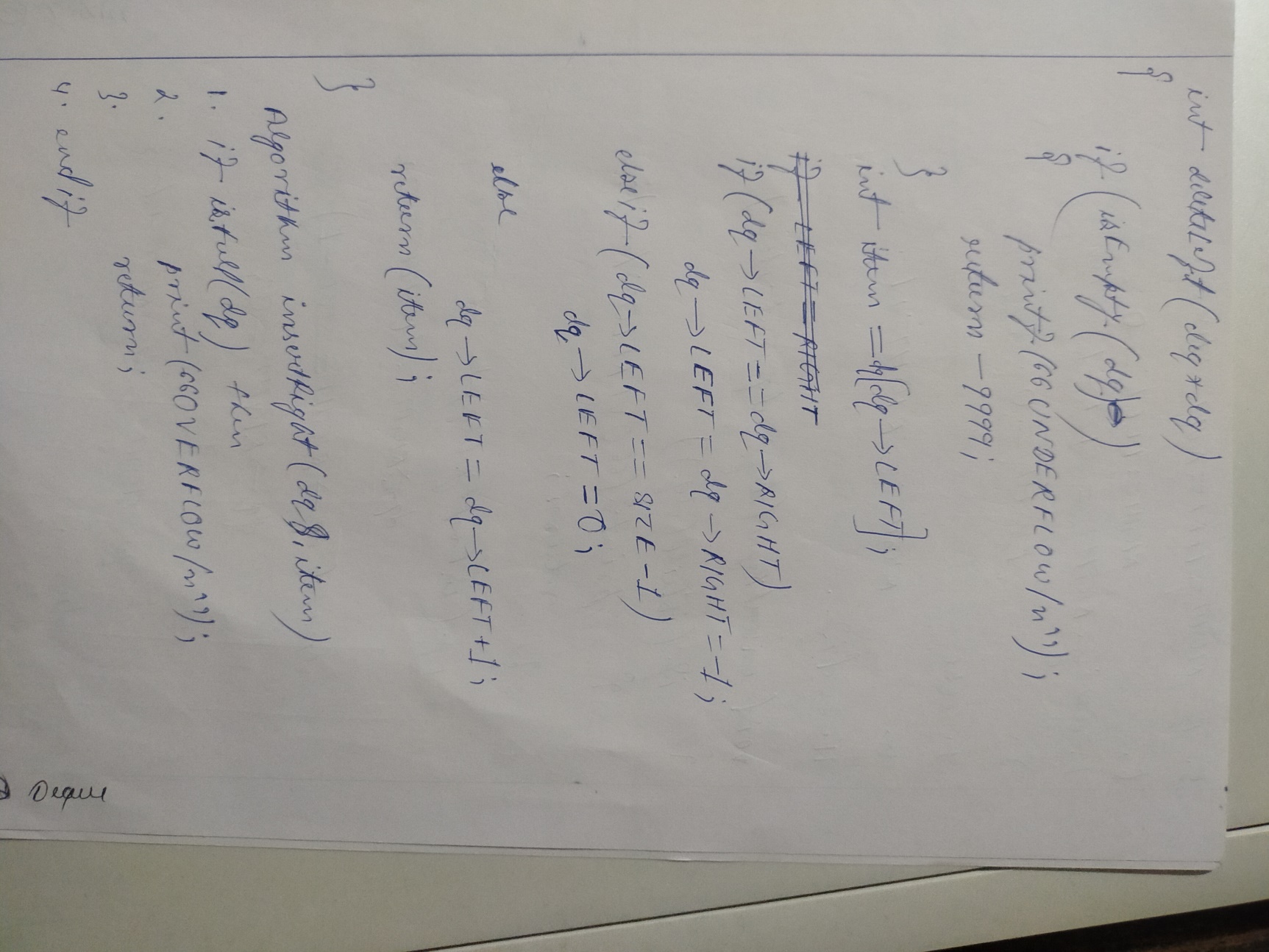
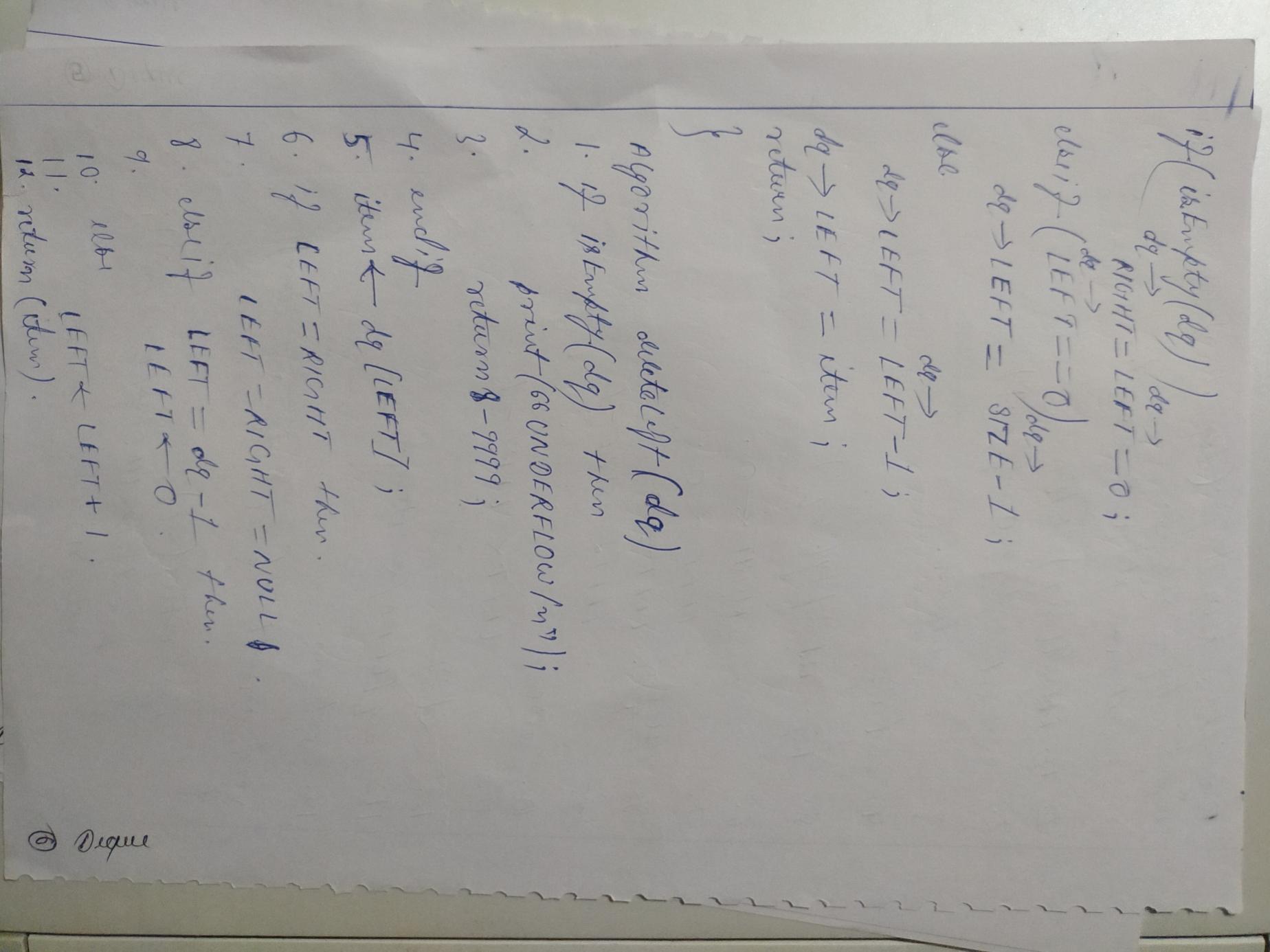
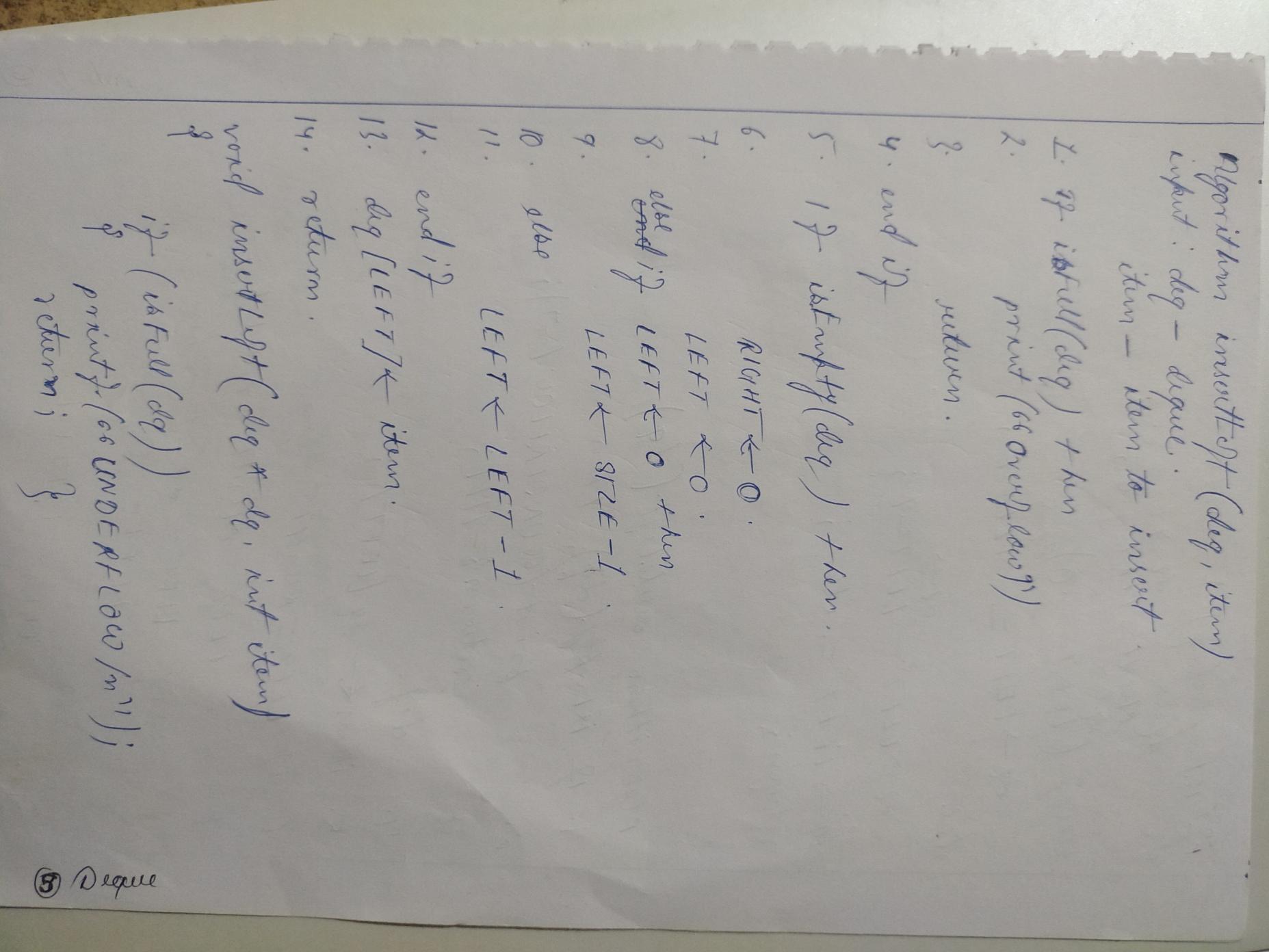
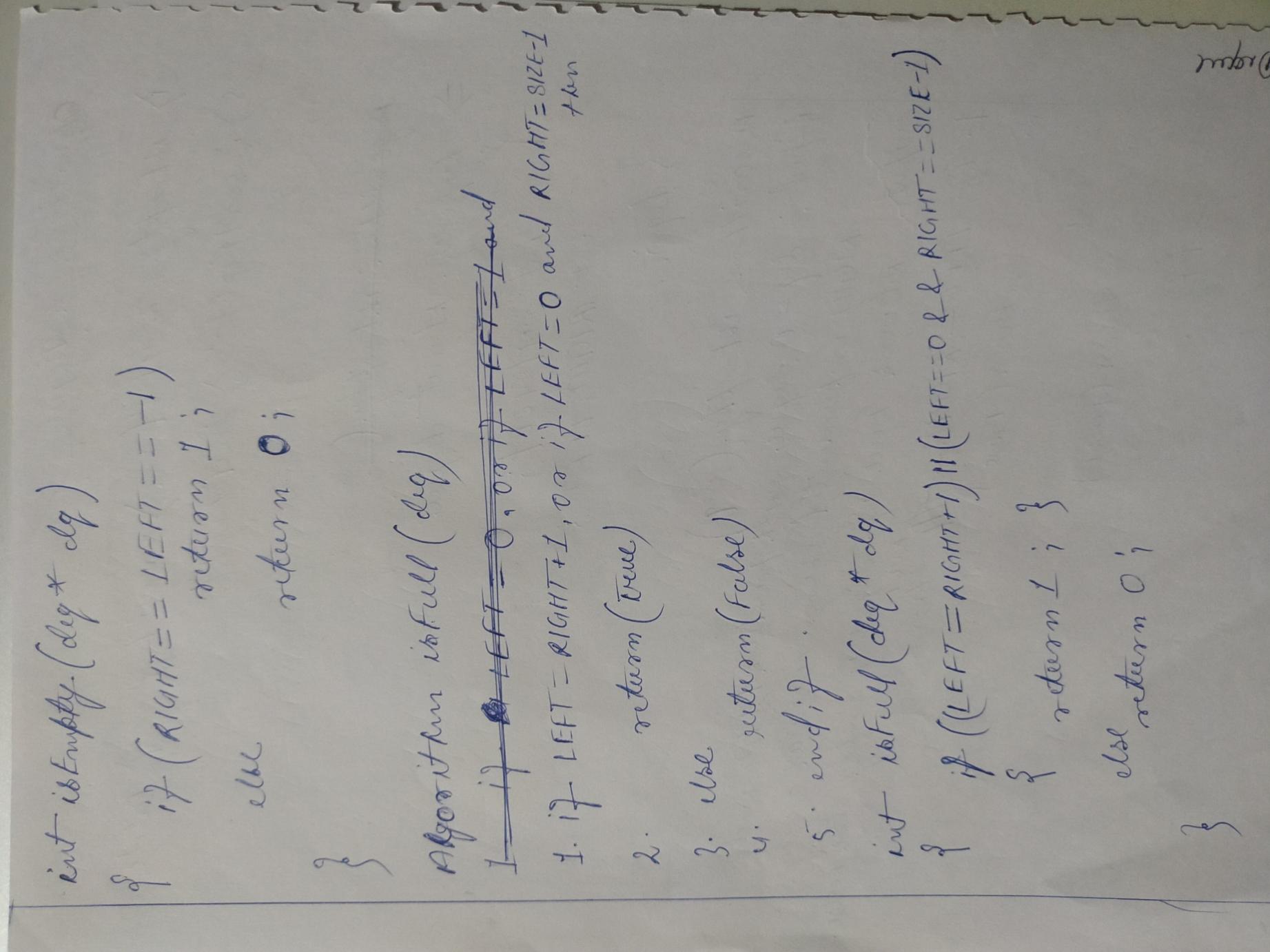
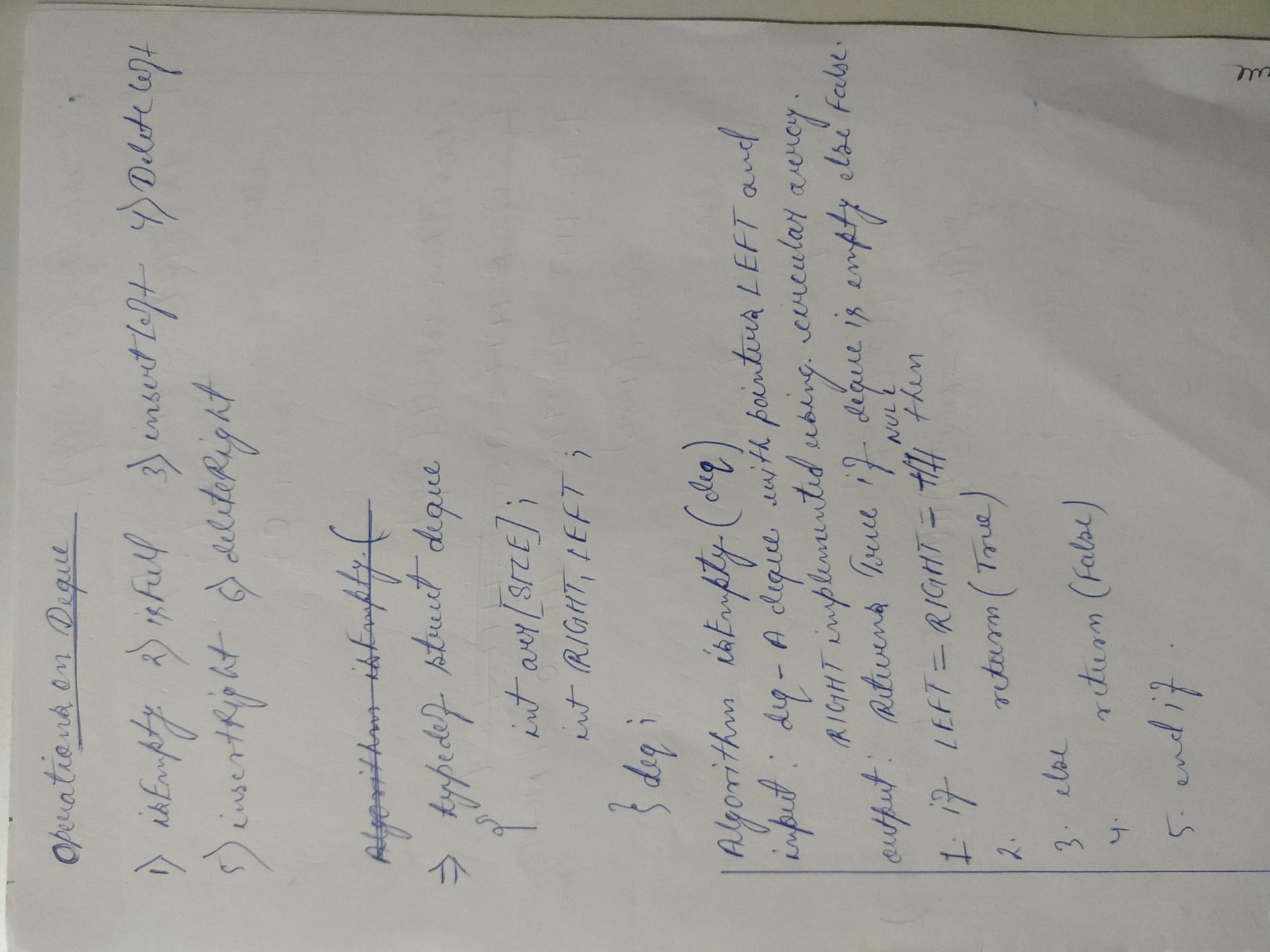
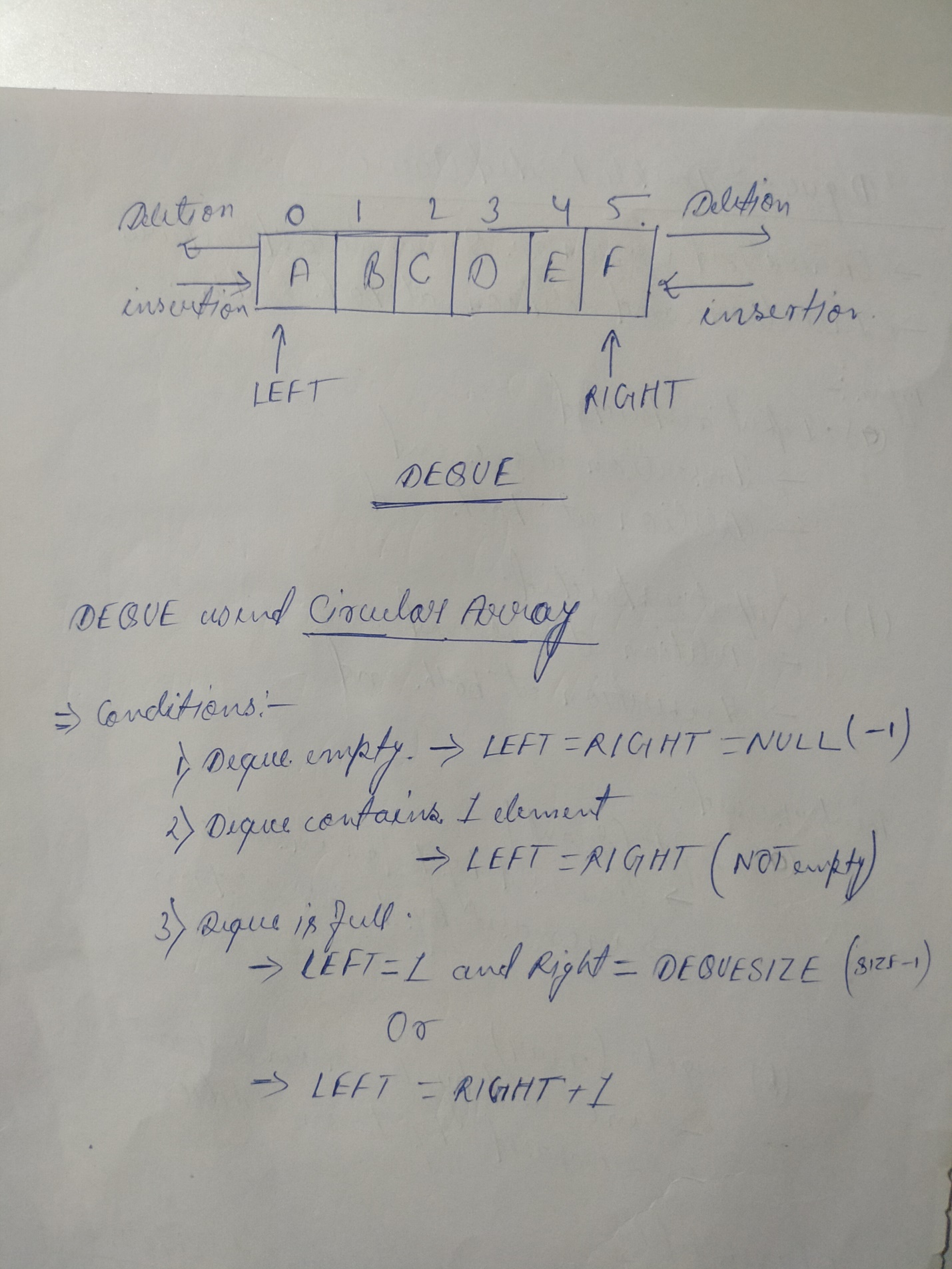
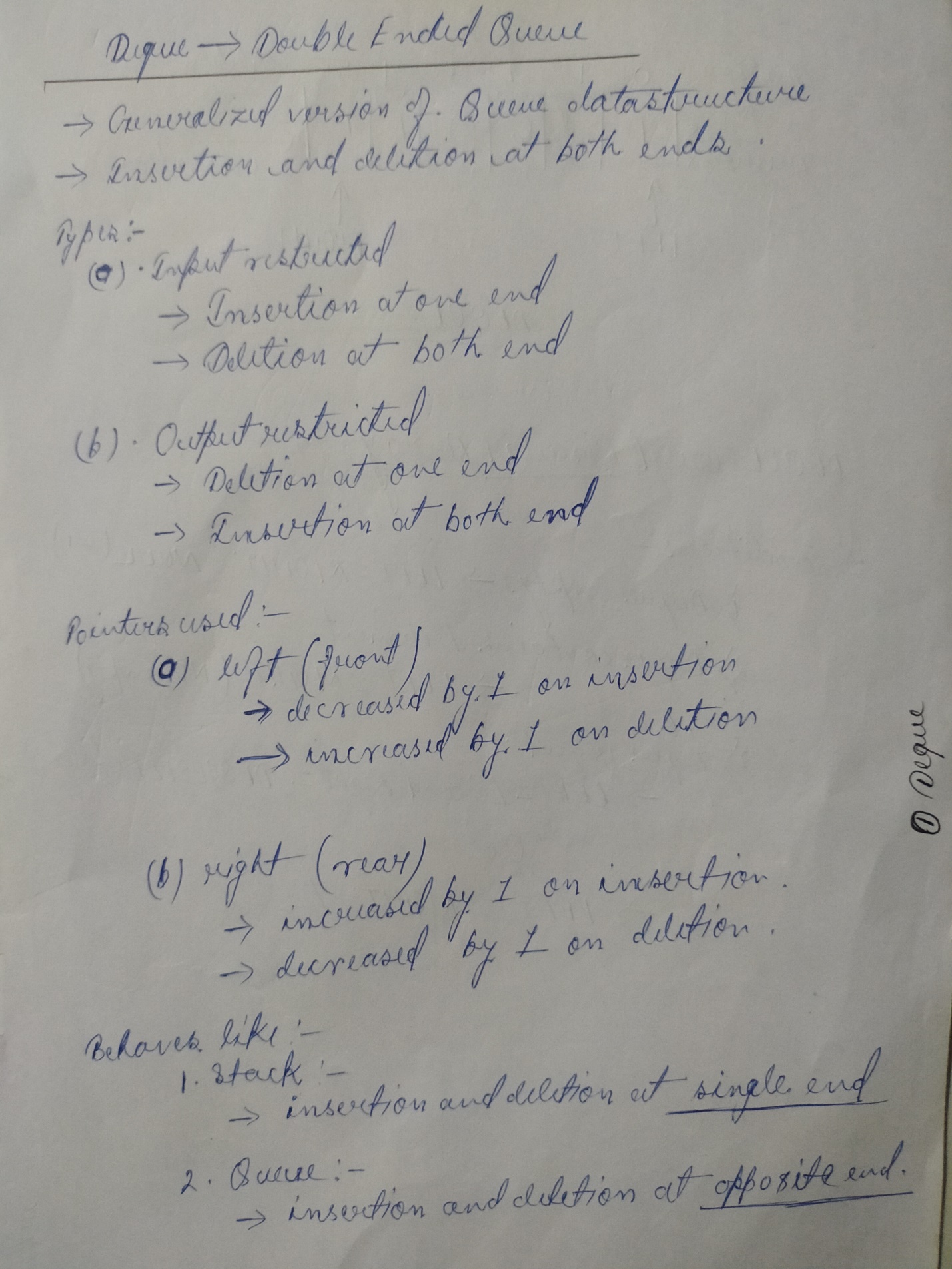
{  
 if(f==NULL) return -9999 ;

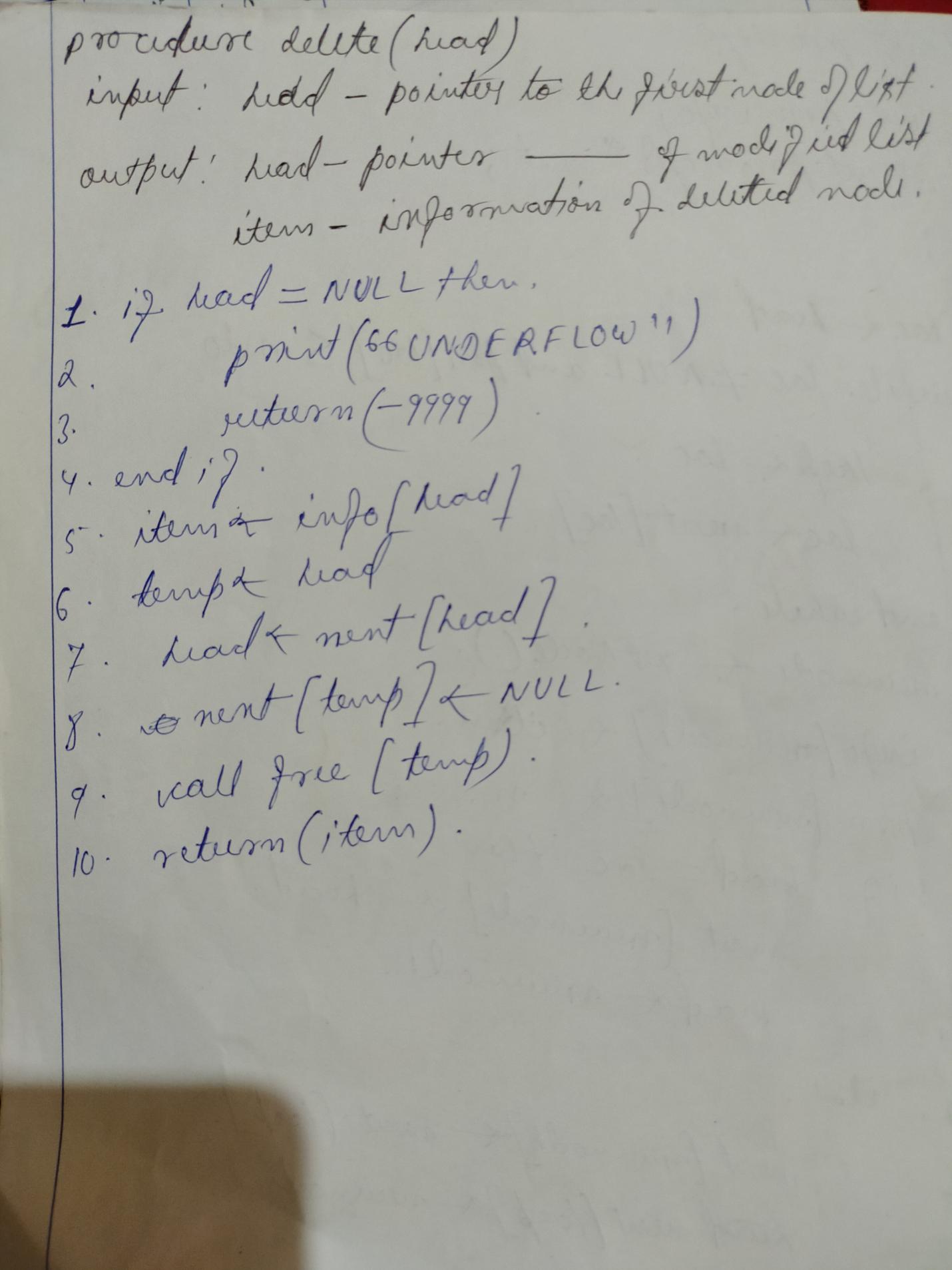
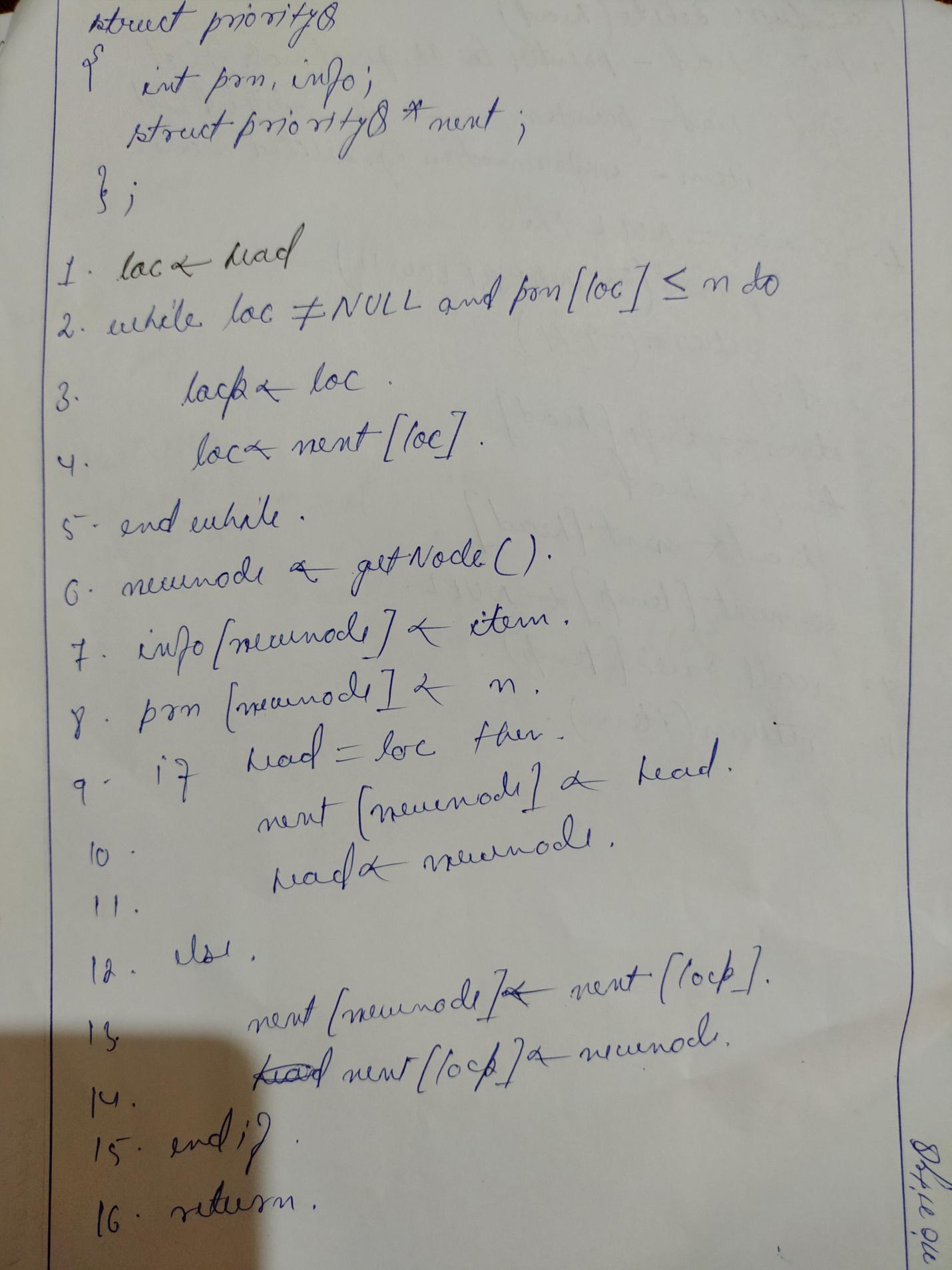
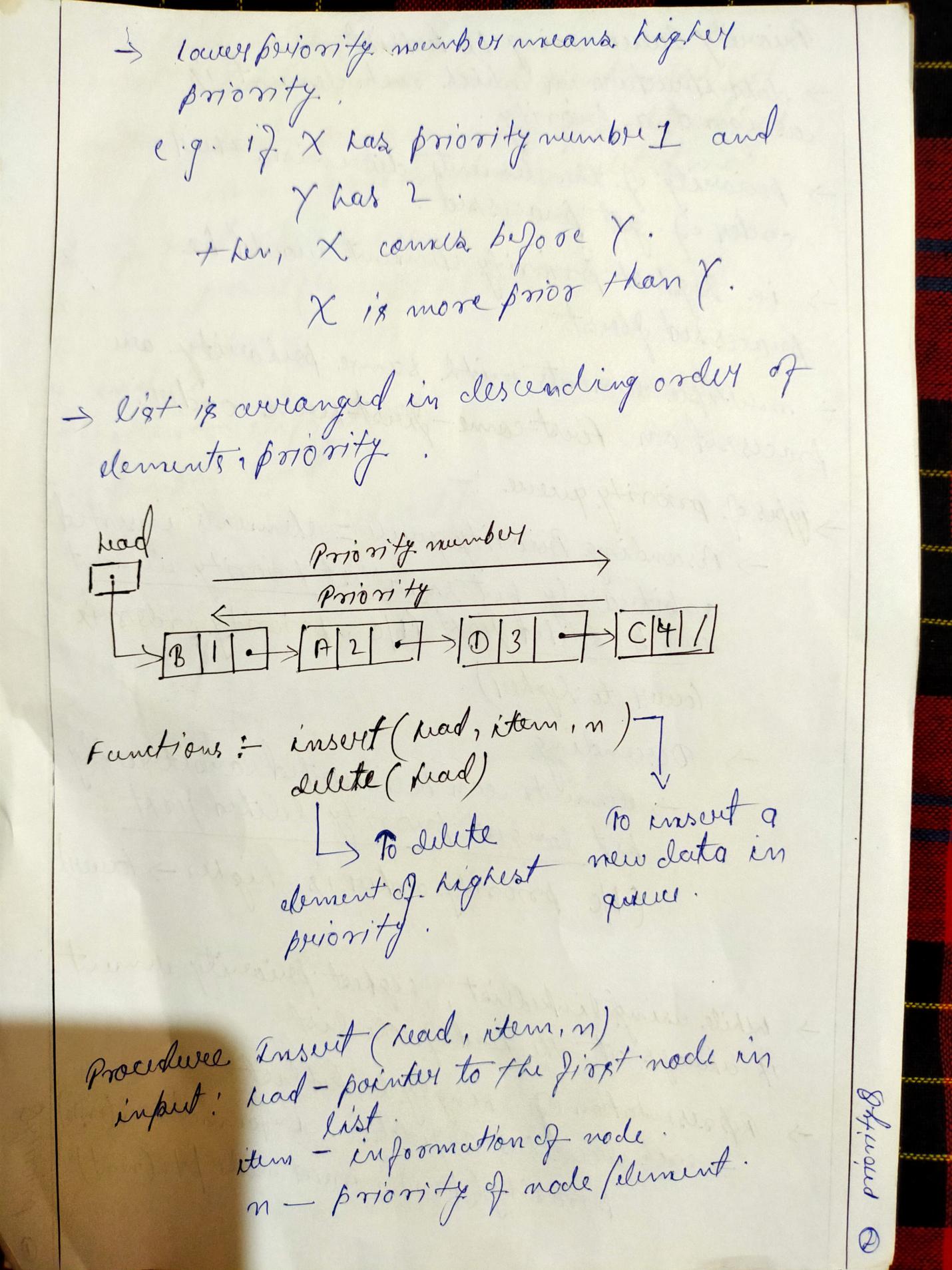
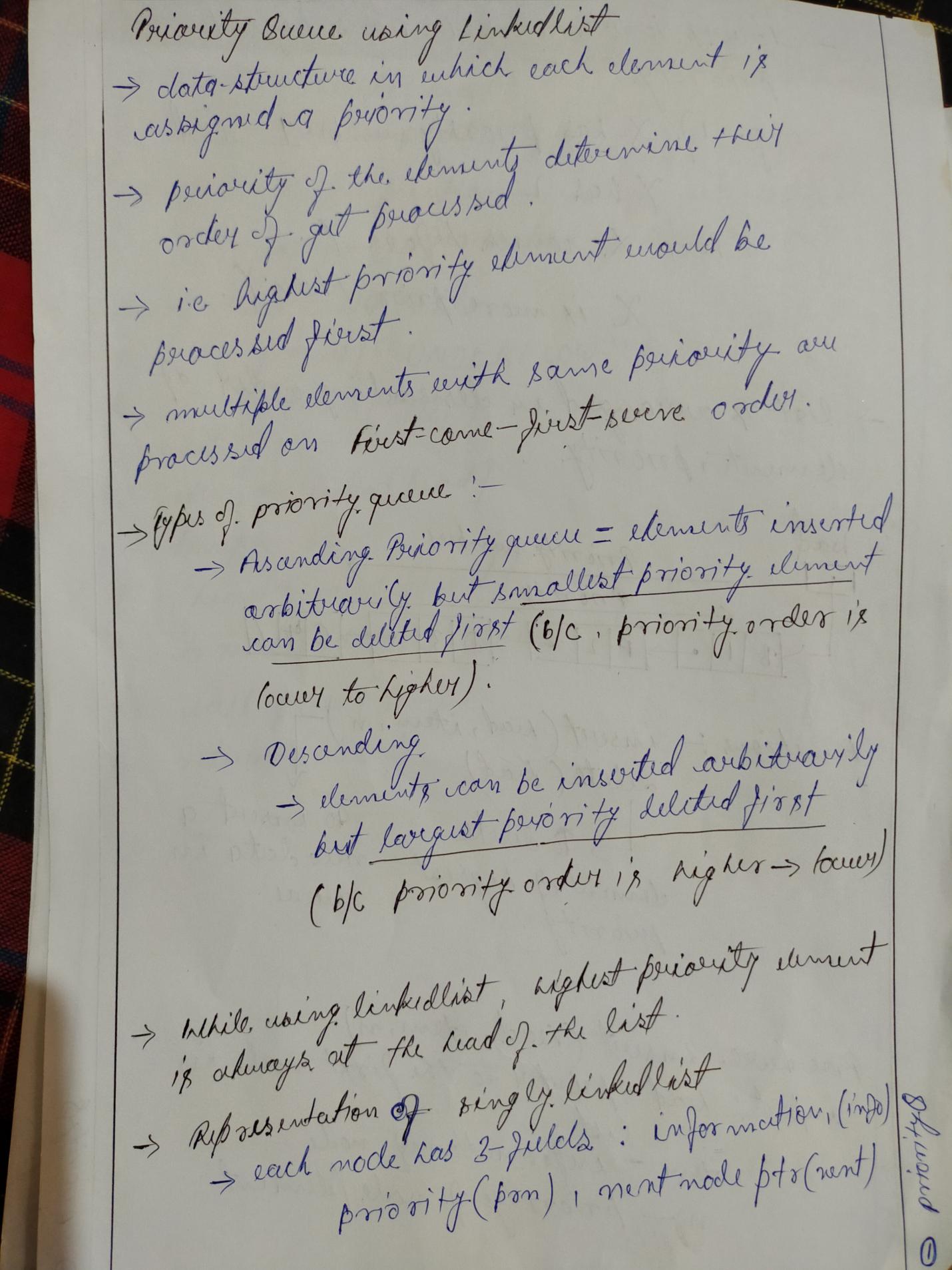
return f->info ;

}

Deque

- It has O(1) complexity for all the operations

****



**- Priority queue has O(1) complexity for enqueue() operations , O(n) for dequeue() and for peek() operations**

**Trees**

- A tree is a finite set of nodes , such that there is a specially designated node called root of the tree

- It is a non-linear data-structure and hence there exist a hierarchial relationship between its elements

- Trees and Graphs are the non-linear data-structure , Ex:-

Book

C1 C2 C3

S1.1 S1.2 S2.1 S2.2 S2.3

S2.1.1 S2.1.2

Book

Chapter1

S1.1

S1.2

Chapter2

S2

S2.1.1

S2.1.2

S2

Chapter 3

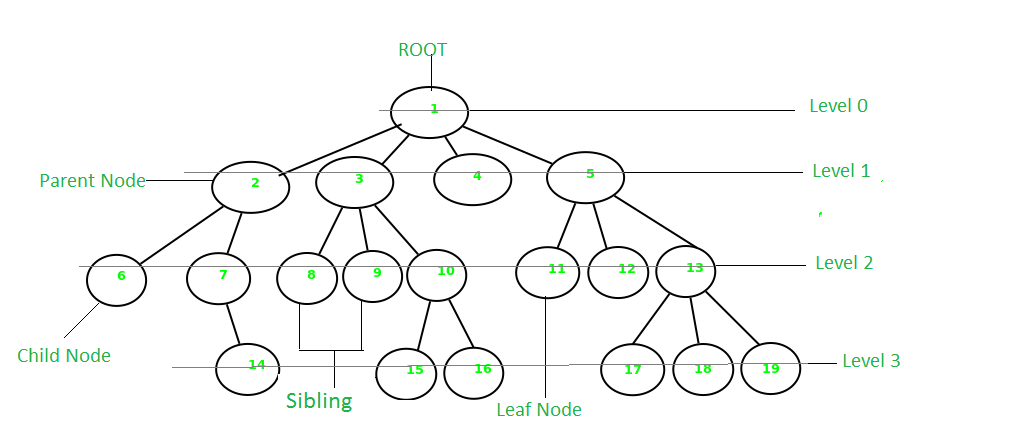
- here , C1 , C2 and C3 are the subtrees of Book tree

Applications:-

1 . Directory structure in OS

2 . Used to organize informations in Databases Systems

3 . Ideal for representing hierarchical data



Some key points:-

- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy

- It is a non-linear data structure because it doesn’t store elements in a sequential manner .

- It is a hierarchical structure as elements in a Tree are arranged in multiple levels

- topmost node is called root .

- Each node contains some data , and data can be of any type i.e string , int etc.

- Each node contains some data and the link or reference of other nodes that are called children

**Some tree Terminologies:-**

**root** : the topmost node . It doesn’t have any parent . ex: 1

**child node** : descendant of any node : 2 , 3 , 4 etc

**parent** : having any sub-node . ex:- 2 , 3 , 5 etc

**sibling** : nodes that have the same parent . ex:- 2,3,4,5 etc

**leaf/terminal/external-node** : node with no children . ex: - 14 , 15 , 16 . 11 etc

**non-leaf/non-terminal/internal** : node with at least one child . ex:- 7 , 10 etc

- root node has level 0

- level of any other node is one more than the level of its parent

- Depth(height) of a tree is the **maximum level of any leaf node** in the tree

- Thus the height equals the length of the longest path from root to any leaf node

- The height or depth of a binary tree can be defined as the maximum or the largest number of edges from leaf node to the root node or root node to leaf node

- Degree of a node in a tree is determined by the **number of its children** like , 1 is degree-4 , 2 is degree-2 and so on

- A node or vertex carries information

-A link b/w parent and its child is called **edge**

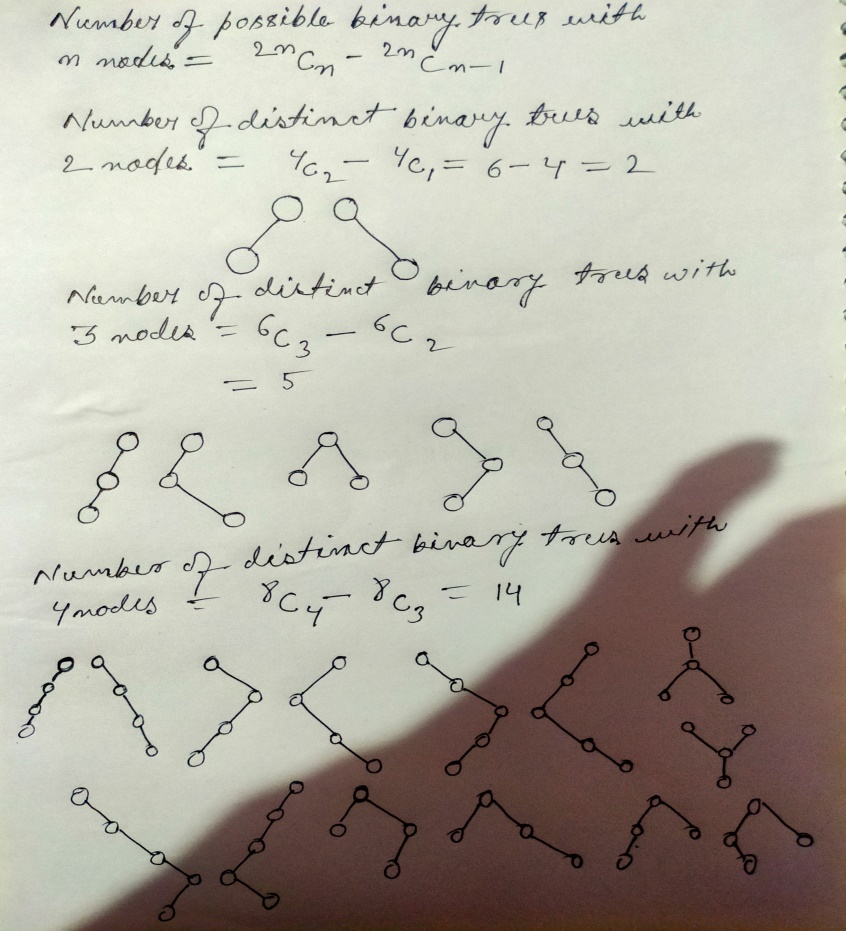
- If there are n nodes, then number of edges = **n-1**

**Binary Tree**

- A tree whose elements have at most 2 children is called a binary tree .

- Since , each element in a binary can have 2 children , we name them the left-child/left sub-tree and right-child/right sub-tree

- A binary tree node contains Data , Pointer to left child and pointer to right child



**Types of Binary Trees**

1 . Full or Strict Binary Tree : All nodes have either 0 or 2 children

2 . Perfect Binary Tree : If internal nodes have 2 children as well as all leaf nodes are on same level

3 . Complete Binary Tree : All levels are completely filled except possibly the last level and if the last level is not completely filled then all the nodes at last level must be on the left as much as possible.

Incomplete BT

Complete BT

4 . Degenerate Tree : Every parent node has exactly one child

a . Left Skewed : All nodes of a degenerate tree ar on left

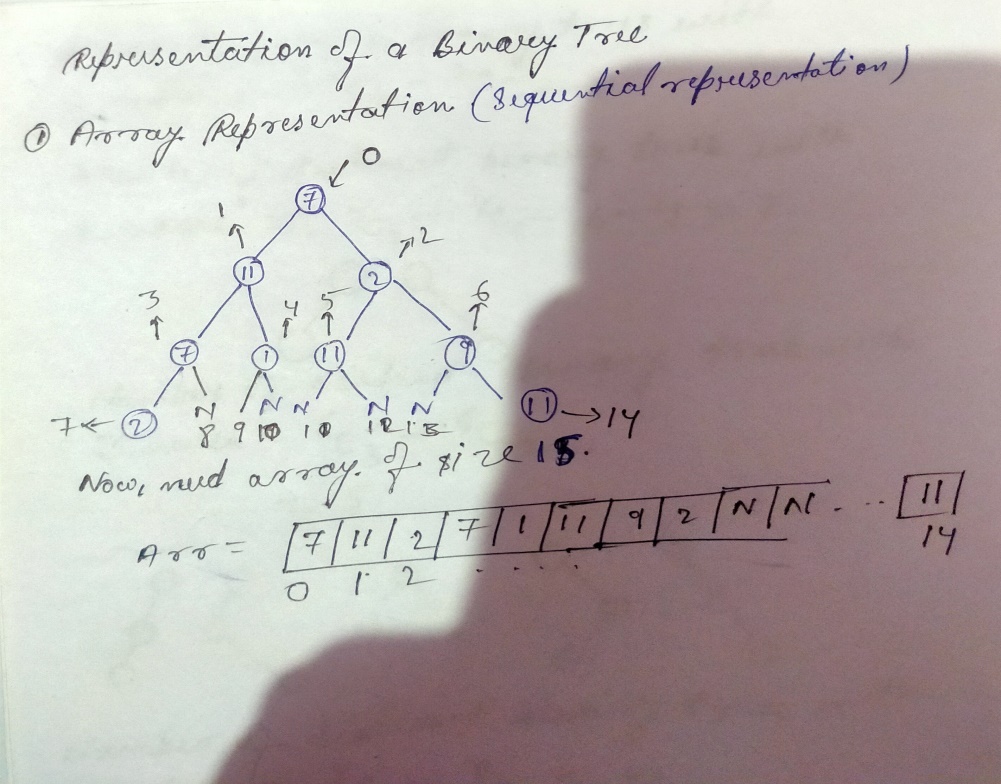
b . Right Skewed : All nodes are at right side

**Representation of Binary Trees**

1 . Array Representation (Sequential Representation)

2 . Dynamic Node Representation (Linked Representation)

**Array Representation**



Drawbacks of Array(Sequential) representation

- Modification , insertion ,deletion kind of opperations are costly

- Size extension is also costly

**Linked Representation**

- It is not called linked list representation because list is linear and Binary trees are non-linear

**Tarversal of Binary-tree**

#include<stdio.h>

#include<malloc.h>

struct node

{

int info ;

struct node \*left ;

struct node \*right;

**};**

**struct node\* createNode(int data)**

{

struct node\* n = (struct node\*)malloc(sizeof(struct node));

n->info = data;

n->left = NULL;

n->right = NULL;

return n ;

}

**void preorder(struct node\* p)**

{

if(p!=NULL)

{

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

preorder(p->left);

preorder(p->right);

}

}

**void postorder(struct node\*p)**

{

if(p!=NULL)

{

postorder(p->left);

postorder(p->right);

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

}

}

**void inorder(struct node\*p)**

{

if(p!=NULL)

{

inorder(p->left);

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

inorder(p->right);

}

}

int main()

{

struct node \*p = createNode(11);

struct node \*p1 = createNode(5);

struct node \*p2 = createNode(6);

struct node \*p3 = createNode(2);

struct node \*p4 = createNode(2);

p->left = p1 ;

p->right = p2 ;

p1->left = p3;

p1->right = p4;

preorder(p);

postorder(p);

inorder(p);

return 0 ;

}

- Complexities of all traversals : O(n)

**Binary Search Tree**

**Checking the Binary Search Tree**

**int isBST(struct node\* root)**

{

static struct node\* prev = NULL ;

if(root!=NULL)

{

if(!isBST(root->left)) return 0;

if(prev!=NULL && root->info <= prev->info) return 0;

prev = root;

return isBST(root->right);

}

else return 1;

}

\*Binary tree with 0 node is BST

// To check BST we are basically using the modified version of inorder traversal . SInce inorder traversal gives sorted ascending order array ,we just check this out for checking BST

**Searching in Binary Search Tree**

**struct node\* search(struct node\* root , int key) // Recursive**

{

if(root==NULL) return NULL;

if(key == root->info) return root;

else if(key < root->info) return search(root->left , key);

else return search(root->right , key);

}

**struct node \*search\_itr(struct node \*root , int key) // Iterative**

{

while(root !=NULL)

{

if(root->info == key) return root;

else if(root->info > key) root = root->left;

else root = root->right;

}

return NULL;

}

Time-complexity : O(n) , n is the total number of nodes

Best Case : O(logn)

**void insert (struct node \*root , int key)**

{

struct node \*prev , \*ptr ;

while(root!=NULL)

{

prev = root;

if(root->info == key) return;

else if(key < root->info) root = root->left;

else root =root->right;

}

ptr = createNode(key);

if(key < prev->info) prev->left = ptr;

else prev->right = ptr;

}

**Deletion of node in BST**

case 1 : leaf node

step-1 : Search the node and delete

case 2 : non-leaf node but the child nodes are leaf nodes

step-1 : Search node

step-2 : get the inorder pre and inorder post of the node

step-3 : Either replace the node with inorder pre or inorder post(leaf-node will be more prior)

case 3 : root node / any other node inbetween the BST

step - 1 : search the node

step - 2 : get inorder pre and inorder post of the node

step - 3 : - if the the replaced node is a leaf node then no worries but if it is non-leaf node then a void would be created at its previous position and hence we need to fill that void with again either with inorder-pre or inorder-post of the replaced node AND keep doing this until the tree has no empty nodes

**struct node\* delete(struct node \*root , int key)**

{

struct node\* ipre ;

if(root == NULL) return NULL;

if(root->left == NULL && root->right==NULL) {free(root); return NULL;}

if(key < root->info) {

root->left = delete(root->left , key);

}

else if(key > root->info) root->right = delete(root->right , key);

else{

ipre = getPred(root);

root->info = ipre->info;

root->left = delete(root->left , ipre->info); **// because the inorder pre is in left-subtree**

}

return root;

}

- All the operations in BST have complexity : O(h) , h is the height of the BST

\*\*HEAD OVER TO Javatpoint FOR COMPLEXITIES TABLE OF BINARY SEARCH TREE\*\*

**AVL Tree**

**Complexities**

|  |  |  |
| --- | --- | --- |
| Operations | Average case | Worst case |
| Space | O(n) | O(n) |
| Search | O(logn) | O(logn) |
| Insert | O(logn) | O(logn) |
| Delete | O(logn) | O(logn) |

**Balancing Imbalanced AVL tree**

typedef struct node{

int data;

struct node \*left;

struct node \*right;

int height;

}nod;

int height(nod \*root){

if(root==NULL) return 0;

else return root->height;

}

int max(int a , int b){

return a>b ? a : b;

}

nod \*create(int data){

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

node->data = data;

node->left = NULL;

node->right = NULL;

node->height = 1;

return node;

}

int **getBalancedFactor(nod \*n)**{

if(n==NULL) return 0;

else return height(n->left) - height(n->right);

}

nod **\*rotateRight(nod \*y)**{

nod \*x = y->left;

nod \*T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(height(y->right),height(y->left))+1;

x->height = max(height(x->right),height(x->left))+1;

return x;

}

nod **\*rotateLeft(nod \*x)**{

nod \*y = x->right;

nod \*T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(height(x->right),height(x->left))+1;

y->height = max(height(y->right),height(y->left))+1;

return y;

}

nod **\*insert(nod \*root , int key)**{

if(root==NULL)

return create(key);

if(key<root->data)

root->left = insert(root->left,key);

else if(key>root->data)

root->right = insert(root->right,key);

root->height = 1 + max(height(root->left),height(root->left));

int bf = getBalancedFactor(root);

//LL

if(bf>1 && key < root->left->data)

return rotateRight(root);

//RR

if(bf<1 && key > root->right->data)

return rotateLeft(root);

//RL

if(bf<1 && key<root->right->data)

{

root->right = rotateRight(root->right);

return rotateLeft(root);

}

//LR

if(bf>1 && key>root->left->data)

{

root->left = rotateLeft(root->left);

return rotateRight(root);

}

return root;

}

int main(){

nod \*root = NULL;

root = insert(root,1);

root = insert(root,2);

root = insert(root,4);

root = insert(root,5);

root = insert(root,6);

root = insert(root,3);

return 0;

}

**Graphs**

**Traversal**

**BFS(Breadth First Search)**

- We make an array for visited nodes and a queue for the keys to be explored

- As we visit all the connected nodes of a particular node then it means the node is explored

#include<stdio.h>

#include<stdlib.h>

#define MAX 100

typedef struct que{

int arr[MAX];

int front;

int rear;

}q;

int isEmpty(q \*n)

{

if(n->front == -1) return 1;

else return 0;

}

int isFull(q \*n)

{

if(n->rear == MAX-1) return 1;

else return 0 ;

}

void enque(q \*n , int key)

{

if(isFull(n)) return;

n->rear++;

n->arr[n->rear] = key;

if(n->front == -1) n->front++;

return;

}

int deque(q \*n)

{

if(isEmpty(n)) return -9999;

int item = n->arr[n->front] ;

if(n->front == n->rear)

{

n->front = n->rear = -1;

}

else n->front++;

return item;

}

int main()

{

q qu , \*p ;

p = &qu;

p->rear = p->front = -1;

int node , i=0 , j;

int visited[7] = {0,0,0,0,0,0,0};

int a[7][7] = {

(0,1,1,1,0,0,0),

(1,0,1,0,0,0,0),

(1,1,0,1,1,0,0),

(1,0,1,0,1,0,0),

(0,0,1,1,0,1,1),

(0,0,0,0,1,0,0),

(0,0,0,0,1,0,0)

} ;

printf("%d ",i); // prints the visited or traversed node

visited[i] = 1 ; // contains the info of visited node or traversed node

enque(p , i) ;// nodes are enqued to get explored

while(!isEmpty(p))

{

node = deque(p);

for(j=0;j<7;j++)

{

if(a[node][j] == 1 && visited[j] == 0)

{

printf("%d ",j);

visited[j] = 1 ;

enque(p,j);

}

}

}

return 0;

}

**DFS(Depth First Search)**

#include<stdio.h>

int visited[7] = {0,0,0,0,0,0,0};

int A[7][7] = {

{0,1,1,1,0,0,0},

{1,0,1,0,0,0,0},

{1,1,0,1,1,0,0},

{1,0,1,0,1,0,0},

{0,0,1,1,0,1,1},

{0,0,0,0,1,0,0},

{0,0,0,0,1,0,0}

};

void DFS(int i)

{

int j ;

printf("%d " , i);

visited[i] = 1 ;

for(j=0;j<7;j++)

{

if(A[i][j] == 1 && !visited[j]) DFS(j);

}

}

int main()

{

DFS(1);

return 0;

}

time-complexity of BFS and DFS : O(v+e) , v = number of vertices and e = no. of edges

**Heap**

int main()

{

**heap h ;**

**h.insert(50);**

**h.insert(55);**

**h.insert(53);**

**h.insert(52);**

**h.insert(54);**

**h.printHeap();**

return 0;

}

**Heap definition , Insertion and Printing**

class heap{

public:

**int arr[100];**

**int size;**

**heap()**

**{**

**arr[0] = -1;**

**size = 0 ;**

**}**

**void insert(int val)**

{

size++;

int index = size , parent;

complexity : O(logn)

arr[index] = val;

while(index>1)

{

parent = index/2;

if(arr[parent]<arr[index])

{

swap(arr[parent],arr[index]);

index = parent ;

}

else return;

}

return;

}

**void deleteHeap()**

complexity : O(logn)

{

if(size==0) return;

//step1 : Put last element into first index and delete the last element

arr[1] = arr[size];

size-- ;

//step2: take root node to its correct position

int i = 1 ;

while(i<size)

{

int leftIndex = 2\*i ;

int rightIndex = 2\*i+1;

if(leftIndex < size and arr[i]<arr[leftIndex])

{

swap(arr[i],arr[leftIndex]);

i = leftIndex;

}

else if(rightIndex < size and arr[i]<arr[rightIndex])

{

swap(arr[i],arr[rightIndex]);

i = rightIndex;

}

else return;

}

}

**void printHeap()**

{

int i ;

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

cout << arr[i] << " " ;

cout << endl;

}

};

**void heapify(int arr[] , int n , int i) (Max heap)**

{

int largest = i ;

complexity : O(logn)

O(n) is also possible

int leftIndex = i\*2;

int rightIndex = i\*2+1;

if(leftIndex<n and arr[largest]<arr[leftIndex])

largest = leftIndex;

if(rightIndex<n and arr[largest]<arr[rightIndex])

largest = rightIndex;

if(largest!=i)

{

swap(arr[largest] , arr[i]);

heapify(arr , n , largest);

}

}

**void heapifyMin(int arr[] , int n , int i) (Max heap)**

{

int smallest = i , leftIndex = 2\*i , rightIndex = 2\*i+1 ;

if(leftIndex < n and arr[smallest] > arr[leftIndex])

smallest = leftIndex;

if(rightIndex < n and arr[rightIndex] < arr[smallest])

smallest = rightIndex;

if(smallest != i)

{

swap(arr[smallest] , arr[i]);

heapifyMin(arr , n , smallest);

}

}

**void heapSort(int arr[] , int n)**

{

int t = n ;

while(t>1)

complexity : O(nlogn)

{

swap(arr[t] , arr[1]);

t--;

heapify(arr , t , 1);

}

}

int main()

{

**int arr[6] = {-1 , 54 , 53 , 55 , 52 , 50}; // regular array**

int n = 5 ;

**for(int i = n/2 ; i>0 ; i--) // At first create a max-heap using heapify algo with above array**

**heapify(arr , n , i);**

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

cout << arr[i] << " " ;

cout << endl;

**heapSort(arr , n); // Then , sort the heap in ascending order**

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

cout << arr[i] << " " ;

cout << endl;

return 0;

}

- we can make min-heap and max-heap **using STL** by calling queue header file

- Priority Queue is a Max-heap by-default

#include<iostream>

**#include<queue>**

using namespace std;

int main()

{

**priority\_queue<int> maxheap ;**

maxheap.push(2);

maxheap.push(6);

maxheap.push(1);

maxheap.push(4);

cout << maxheap.top() << endl;

**priority\_queue<int , vector<int> , greater<int>> minheap;**

minheap.push(2);

minheap.push(4);

minheap.push(1);

minheap.push(6);

cout << minheap.top()<< endl;

return 0;

}

**HashMap / HASHTABLE / HASHING**

**Mapping**

- **ordered mapping** and **unordered mapping**

- In a map data is in a key-value pair

**Operations in ordered and unordered maps**

#include<iostream>

#include<map>

#include<unordered\_map>

#include<iterator>

using namespace std;

int main()

{

// CREATION

unordered\_map<string,int> m;

//INSERTION

//1

pair<string,int> p = make\_pair("operations",3);

m.insert(p);

//2

pair<string,int> p2("operation",1);

m.insert(p2);

//3

m["operate"] = 2 ;

//SEARCH

cout << m["operate"] << endl;

cout << m.at("operate") <<endl;

cout << m["new"]; // it will make entry with value 0 , if there is not any

cout<<m.at("new"); // it won't create entry but only print the corresponding value of the provided key

//SIZE

cout << m.size() << endl;

//CHECK PRESENCE OF A KEY

cout << m.count("operation") << endl ; // It will give 1 if this key is present else 0

//ERASE A KEY

m.erase("operate");

//TRAVERSE THE KEY-VALUE PAIRS

//1

for(auto i:m)

cout << i.first << " " <<i.second << endl;

//2 , using iterators

unordered\_map<string,int> :: iterator it = m .begin();

while(it!=m.end()){

cout << it->first << " " << it->second << endl;

it++;

}

return 0;

}