➔ Binary Tree Traversal

Pseudocode :

```
Node
{
        int data;
        Node left, right;
}

Node createBinaryTree()
{
        new node = create a new node
        print("Enter data")
        x = value to insert in node
        if(x == -1)
        {
                return 0;
        }
        newNode -> data = x
        print("Enter left child")
        newNode -> left = createBinaryTree()
        print("Enter right child")
        newNode -> right = createBinaryTree()
        return newNode
}
```

Algorithm :

```
If root is NULL
  then create root node
return
```

```
If root exists then
  compare the data with node.data

  while until insertion position is located

    If data is greater than node.data
      goto right subtree
    else
      goto left subtree


  endwhile


  insert data


end If
```

1. Inorder

Pseudocode :

```
inOrder(root)
{
    if(root == 0)
    {
            return;
    }
    inOrder(root -> left)
    print root -> data
    inOrder(root -> right)
}
```

Algorithm :

Step 1: Repeat Steps 2 to 4 while TREE != NULL

Step 2: INORDER(TREE LEFT)

Step 3: Write TREE DATA

Step 4: INORDER(TREE RIGHT)

[END OF LOOP]

Step 5: END

2. Preorder

Pseudocode :

```
preOrder(root)
{
    if(root == 0)
    {
        return
    }
    print root -> data
    preOrder(root -> left)
    preOrder(root -> right   )
}
```

Algorithm :

Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: Write TREE DATA
Step 3: PREORDER(TREE LEFT)
Step 4: PREORDER(TREE RIGHT)
[END OF LOOP]
Step 5: END

3. Postorder

Pseudocode :

```
postOrder(root)
{
    if(root == 0)
```

```
    {
        return
    }
    postOrder(root -> left)
    postOrder(root -> right)
    print root -> data
}
```

Algorithm :

Step 1: Repeat Steps 2 to 4 while TREE != NULL

Step 2: POSTORDER(TREE LEFT)

Step 3: POSTORDER(TREE RIGHT)

Step 4: Write TREE DATA

[END OF LOOP]

Step 5: END

➔ Queue :

1. Enqueue :

Pseudocode (Using Array):

```
int queue[size]
enqueue(data element)
{
    if(rear == size-1)
    {
        print("overflow")
    }
    else if(front == -1 && rear == -1)
```

```
        {
                front = rear = 0
                queue[rear] = data element
        }
        else
        {
                rear++;
                queue[rear] = data element
        }
}
```

Algorithm (Using Array) :

```
Step 1: IF REAR = MAX-1
                Write OVERFLOW
                Goto step 4
        [END OF IF]
Step 2: IF FRONT=-1 and REAR=-1
            SET FRONT = REAR = 0
        ELSE
            SET REAR = REAR+1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

Pseudocode (Using Linked List):

```
enqueue(data element)
{
        if(front == 0 && rear == 0){
                front = rear = newnode
```

```
        }
        else{
                rear->next = newnode
                rear = newnode
        }
}
```

Algorithm (Using Linked List) :

```
Step 1: Allocate memory for the new node and name it as PTR
Step 2: SET PTR DATA = VAL
 Step 3: IF FRONT = NULL
            SET FRONT = REAR = PTR
            SET FRONT -> NEXT = REAR -> NEXT = NULL
        ELSE
            SET REAR -> NEXT = PTR
            SET REAR = PTR
            SET REAR -> NEXT = NULL
        [END OF IF]
Step 4: END
```

2. Dequeue

Pseudocode (Using Array):

```
dequeue()
{
        if(front == -1 && rear == -1)
        {
                print("underflow")
        }
```

```
	else if(front == rear)
	{
		front = rear = -1
	}
	else
	{
		front++
	}
}
```

Algorithm (Using Array):

```
Step 1: IF FRONT = -1 OR FRONT > REAR
			Write UNDERFLOW
	ELSE
		SET VAL = QUEUE[FRONT]
		SET FRONT = FRONT+1
	[END OF IF]
Step 2: EXIT
```

Pseudocode (Using Linked List):

```
dequeue(){
	temp=front
	if(front == 0 && rear == 0){
		print("Queue is empty")
	}
	else{
		print(front->data)
		front = front->next
		free(temp)
```

```
        }
}
```

Algorithm (Using Linked List):

```
Step 1: IF FRONT = NULL
                Write Underflow
                Go to Step 5
        [END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END
```

3. Display

Pseudocode (Using Array):

```
display()
{
    if(front == -1 && rear == -1)
    {
            print("empty")
    }
    else{
            for(i = front; i<rear+1; i++){
                    print queue[i]
            }
    }
}
```

Algorithm (Using Array):

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.

- **Step 4 -** Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

Pseudocode (Using Linked List):

```
display(){
    if(front == 0 && rear == 0){
        print("Queue is empty")
    }
    else{
        temp = first
        while(temp != 0){
            print(temp->data)
            temp = temp->next
        }
    }
}
```

Algorithm (Using Linked List):

4. Peek

Pseudocode :

```
peek()
{
    if(front == 0 && rear == 0){
        print("Queue is empty")
    }
```

else{

print(front->data)

}

}

Algorithm :

```
begin procedure peek
   return queue[front]
end procedure
```

5. isFull

Pseudocode :

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

Algorithm :

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

6. isEmpty :

Pseudocode :

```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

Algorithm :

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

7. Enqueue(Circular Queue)

Pseudocode (Using Array):

```
enqueue(data element){
if(front == -1 && rear == -1){
        front = rear = 0
        queue[rear] = data
}
else if(((rear + 1) % n) == front){
        print("Queue is full")
}
else{
        rear = (rear + 1) % n
        queue[rear] = data
}
}
```

Algorithm (Using Array):

Step 1: IF FRONT = and Rear = MAX-1

Write OVERFLOW

Goto step 4

[End OF IF]

Step 2: IF FRONT=-1 and REAR=-1

SET FRONT = REAR = 0

ELSE IF REAR = MAX-1 and FRONT != 0

SET REAR = 0

ELSE

SET REAR = REAR+1

[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT

Pseudocode (Using Linked List):

```
enqueue(data element){

    if(rear == 0){
        front = rear = newnode
        rear->next = front
    }
    else{
        rear->next = newnode
        rear = newnode
        rear->next = front
    }
}
```

Algorithm (Using Linked List):

8. Dequeue(Circular Queue) :

Pseudocode (Using Array):

```
dequeue(){
	if(front == -1 && rear == -1){
		print("Queue is empty")
	}
	else if(front == rear)
	{
		print(queue[front])
		front = rear = 1
	}
	else
	{
		front = (front + 1) % n
	}
}
```

Algorithm (Using Array):

```
Step 1: IF FRONT=-1
		Write UNDERFLOW
		Goto Step 4
	[END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
		SET FRONT = REAR=-1
	ELSE
		IF FRONT = MAX -1
			SET FRONT = 0
		ELSE
			SET FRONT = FRONT+1
		[END of IF]
	[END OF IF]
```

Step 4: EXIT

Pseudocode (Using Linked List):

```
temp=front
dequeue(){
        if(front == 0 && rear == 0){
                print("Queue is empty")
        }
        else if(front == rear) {
                front = rear = 0
                free(temp)
        }
        else{
                front = front->next
                rear->next = front
                free(temp)
        }
}
```

Algorithm (Using Linked List):

9. Display (Circular Queue)

Pseudocode (Using Array):

```
display(){
        i = front
        if(front == -1 && rear == -1){
                print("Queue is empty")
```

```
        }
        else{
                while(i != rear){
                        print(queue[i])
                        i = (i+1) % n
                }
                print(queue[i])
        }
}
```

Pseudocode (Using Linked List):

```
        display(){
        if(front == 0 && rear == 0){
                print("Queue is empty")
        }
        else{
                while(temp->next != front){
                        print(temp->data)
                        temp = temp->data
                }
         print(temp->data)
}
```

Algorithm (Using Linked List):

➔ Stack :

1. Push :

Pseudocode (Using Array):

```
int stack[size]
push(int data)
{
	if(top == size-1){
		print("overflow")
	}
	else
	{
		top++
		stack[top] = data
	}
}
```

Pseudocode (Using Linked List):

Algorithm (Using Array):

```
begin procedure push : stack, data
	if stack is full
		return null
	endif
	top <- top + 1
	stack[top] <- data
end procedure
```

Algorithm (Using Linked List):

push(value)

1. Create a new Node and set data part = value

2. Check whether stack is Empty (top=== NULL)

3. If it is Empty, then set

new Node->next = NULL

4. If it is NOT Empty, then set new Node ->next = top

5. Then, set top = new Node

2. Pop

Pseudocode (Using Array):

```
pop()
{
        int item;
        if(top == -1){
                print("underflow")
        }
        else
        {
                item = stack[top]
                top--
        }
}
```

Algorithm (Using Array):

```
begin procedure pop: Stack
        if stack is empty
                return null
```

endif

data ← stack [top]

top & top - 1

return data

end procedure

Pseudocode (Using Linked List):

Algorithm (Using Linked List):

pop()

1. Check whether stack is Empty (top == =NULL)
2. If it is Empty, then display "Stack undertlow / Stack is Empty"
3. If it is NOT Empty, then define a Node pointer 'temp' and Set it to 'top'.
4. Then, set top = top->next
5. Finally, delete 'temp'.

3. Peek

Pseudocode :

```
int peek() {
   return stack[top];
}
```

Algorithm :

begin procedure peek

return stack [top]

end procedure

4. isFull

Pseudocode :

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

Algorithm :

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

5. isEmpty

Pseudocode :

```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

Algorithm :

```
begin procedure isempty

   if top less than 1
      return true
   else
      return false
   endif
```

```
end procedure
```

➔ Heap

1. Insertion(Max Heap)

Pseudocode :

```
insertHeap(A,n, value)
{
        n = n + 1
        A[n] = value
        i=n
        white(i>1)
        {
                parent = [i/2]
                if (A[parent] < A[i])
                {
                        Swap(A [ parent], A [i]]
                        i= Parent
                }
                else{ return }
        }
}
```

Algorithm :

Step 1: [Add the new value and set its POS]

SETN=N+1, POS=N

Step 2: SET HEAP[N] = VAL

```
Step 3: [Find appropriate location of VAL]
        Repeat Steps 4 and 5 while POS>1
Step 4:         SET PAR = POS/2
Step 5:         IF HEAP[POS] <= HEAP[PAR],
                then Goto
Step 6.         ELSE
                        SWAP HEAP[POS], HEAP[PAR]
                        POS = PAR
                [END OF IF]
        [END OF LOOP]
Step 6: RETURN
```

2. Heapsort

Pseudocode :

```
Heapsort (A, n)
{
    for (i=n/2; i >= 1; i--)
    {
            мaxHeapify(A, n, i);
    }
    for(i = n; i>=1; i--)
    {
            Swap (A [1], A [i]);
            Max Heapify (A, n, 1);
    }
}
```

Algorithm :

⇨

```
HeapSort(arr)
BuildMaxHeap(arr)
for i = length(arr) to 2
    swap arr[1] with arr[i]
        heap_size[arr] = heap_size[arr] ? 1
        MaxHeapify(arr,1)
End
```

BuildMaxHeap(arr)

```
BuildMaxHeap(arr)
    heap_size(arr) = length(arr)
    for i = length(arr)/2 to 1
MaxHeapify(arr,i)
End
```

MaxHeapify(arr,i)

```
MaxHeapify(arr,i)
L = left(i)
R = right(i)
if L ? heap_size[arr] and arr[L] > arr[i]
largest = L
else
largest = i
if R ? heap_size[arr] and arr[R] > arr[largest]
largest = R
if largest != i
swap arr[i] with arr[largest]
MaxHeapify(arr,largest)
End
```

3. MaxHeapify

Pseudocode :

```
Max Heapify (A, n, i)
{
    int largest = i;
    int l = 2 * i;
    int r = 2 * i + 1;
    while (l ≤ n && A[l] > A[largest])
    {
            largest = l;
    }
    while (r ≤ n && A[r] > A [largest])
    {
            largest r;
    }

    if (largest != i)
    {
            swap(A[largest], A[i]);
            MaxHeapify (A, n, largest);
    }
}
```

Algorithm :

MaxHeapify(arr,i)

1. MaxHeapify(arr,i)
2. L = left(i)

3. R = right(i)
4. if L ? heap_size[arr] and arr[L] > arr[i]
5. largest = L
6. else
7. largest = i
8. if R ? heap_size[arr] and arr[R] > arr[largest]
9. largest = R
10. if largest != i
11. swap arr[i] with arr[largest]
12. MaxHeapify(arr,largest)
13. End

4. Deletion :

Algorithm :

```
Step 1: [Remove the last node from the heap]
            SET LAST = HEAP[N], SETN=N-1
Step 2: [Initialization]
            SET PTR=1, LEFT=2, RIGHT=3
Step 3: SET HEAP[PTR] = LAST
Step 4: Repeat Steps 5 to 7 while LEFT <= N
Step 5: IF HEAP[PTR] >= HEAP[LEFT] AND
           HEAP[PTR] >= HEAP[RIGHT]
                Go to Step 8
           [END OF IF]
Step 6: IF HEAP[RIGHT] <= HEAP[LEFT]
                SWAP HEAP[PTR], HEAP[LEFT]
                SET PTR = LEFT
            ELSE
                SWAP HEAP[PTR], HEAP[RIGHT]
                SET PTR = RIGHT
```

[END OF IF]

Step 7: SET LEFT=2* PTR and RIGHT = LEFT+1

[END OF LOOP]

Step 8: RETURN

➔ Linked List(singly) :

1. Traverse :

   Pseudocode :

   Algorithm :

   Step 1: [INITIALIZE] SET PTR = START

   Step 2: Repeat Steps 3 and 4 while PTR != NULL

   Step 3: Apply Process to PTR -> DATA

   Step 4: SET PTR = PTR-> NEXT

   [END OF LOOP]

   Step 5: EXIT

2. Insertion at beginning :

   Pseudocode :

   Algorithm :

   Step 1: IF AVAIL = NULL

   Write OVERFLOW

   Go to Step 7

   [END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

Step 4: SET NEW_NODE-> DATA = VAL

Step 5: SET NEW_NODE-> NEXT = START

Step 6: SET START = NEW_NODE

Step 7: EXIT

3. Insertion at the end :

Pseudocode :

Algorithm :

Step 1: IF AVAIL = NULL
    Write OVERFLOW
    Go to Step 10

  [END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL-> NEXT

Step 4: SET NEW_NODE-> DATA = VAL

Step 5: SET NEW_NODE -> NEXT = NULL

Step 6: SET PTR = START

Step 7: Repeat Step 8 while PTR-> NEXT != NULL

Step 8: SET PTR = PTR -> NEXT

  [END OF LOOP]

Step 9: SET PTR-> NEXT = NEW_NODE;

Step 10: EXIT

4. Delete at the start :

Pseudocode :

Algorithm :

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 5

[END OF IF]

Step 2: SET PTR = START

Step 3: SET START = START-> NEXT

Step 4: FREE PTR

Step 5: EXIT

5. Delete at the end :

Pseudocode :

Algorithm :

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 8

[END OF IF]

Step 2: SET PTR = START

Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL

Step 4: SET PREPTR = PTR

Step 5: SET PTR = PTR -> NEXT

[END OF LOOP]

Step 6: SET PREPTR-> NEXT = NULL

Step 7: FREE PTR

Step 8: EXIT

➔ Linked List (Circular)

1. Insertion at beginning :

Algorithm :

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 11

[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL ->NEXT

Step 4: SET NEW_NODE-> DATA = VAL

Step 5: SET PTR = START

Step 6: Repeat Step 7 while PTR NEXT != START

Step 7: PTR = PTR-> NEXT

[END OF LOOP]

Step 8: SET NEW_NODE->NEXT = START

Step 9: SET PTR NEXT = NEW_NODE

Step 10 : SET START = NEW_NODE

Step 11: EXIT

1. Deletion at end :

Algorithm :

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 10

[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL ->NEXT

Step 4: SET NEW_NODE-> DATA = VAL

Step 5: SET NEW_NODE-> NEXT = START

Step 6: SET PTR = START

Step 7: Repeat Step 8 while PTR ->NEXT != START

Step 8: SET PTR = PTR-> NEXT

[END OF LOOP]

Step 9: SET PTR-> NEXT = NEW_NODE

Step 10 : EXIT

➔ DFS :

Pseudocode :

```
Initialize visited array
DFS(v){
    visited[v] = true;
    print(v);

    for each 'u' adjacent to 'v'
    {
        if(visited[u] = false)
            DFS(u)
    }
}
```

Algorithm :

Step 1: SET STATUS=1 (ready state) for each node in G

Step 2: Push the starting nodeAon the stack and set its STATUS=2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS=3 (processed state)

Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS=1) and set their STATUS=2 (waiting state)

[END OF LOOP]

Step 6: EXIT

➔ BFS :

Pseudocode :

```
BFS(G,S){
    Initialize visited array
            q.enqueue(s);
    while(!q.empty())
    {
            v = q.dequeue();
            print(v)
            visited[v] = true;
            for all u adjacent to v
            {
                    if(u is not visited & u not already in queue)
                    {
                            q.enqueue(u);
                    }
            }
    }
}
```

Algorithm :

Step 1: SET STATUS=1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS=2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS=3 (processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS=1) and set their STATUS=2 (waiting state)

[END OF LOOP]

Step 6: EXIT