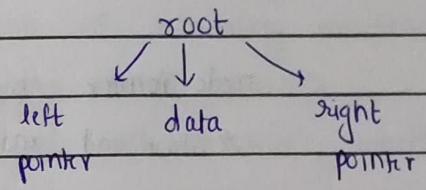
TUTORIAL - 13 DATA STRUCTURE

U19CS012

Q.1.) Write Algorithm for:

(a) (i) Pre-order Traversal



Preorder (node pointer root)

1) Start
2) If root == NULL then return
3) display root → data
4) Preorder(root → left)
5) Preorder (root → right)
6) End

(ii) Inorder Traversal

inorder (node pointer root)

1) Start
2) If root == NULL then return
3) inorder (root → left)
4) display root → data
5) inorder (root → right)
6) End

(iii) Postorder Traversal

Postorder (node pointer root)

1) Start
2) If root == NULL then return
3) Postorder (root → left)
4) Postorder (root → right)
5) ~~Postorder~~ & display root → data
6) End

(b) (i) Construct Tree Using inorder and preorder

(A) Each node Pointer contains Pointer left and Pointer right + data. the operator new is used for the dynamic memory allocation of node.

```
node Pointer buildtree ( inorder arrayin , preorder p , start , end )
// start and end will be starting and ending address of array

1) Begin
2) Declare Static Prindex = 0
3) If start > end
        return NULL
4) set temp = new node ( P[preindex] )
5) Preindex = Preindex + 1
6) If start = end
        return temp

7) Set inindex = search ( in, start, end , temp -> data )
8) temp -> left = buildtree ( in , p, start, inIndex - 1)
9) temp -> right = buildtree ( in, p, inIndex+1, end )
10) return temp
11) End
```

```
Search ( inorder array , start, end, data )
1) Begin
2) Set i = start
3) while ( i <= end )
        if in[i] = data
            return 1
        i = i + 1
4) End
```

ii) Inorder and Postorder sequence

- Create (in[], post[], inst, inend, Index)

Let in[] be string of inorder expression, post[] be string of postorder expression inst be starting index of remaining inorder sequence, inend → ending index of remaining inorder sequence sequence = length(in) - 1 Let index = pointer of post expression which will form a node.

1) Start
2) If inst > inend
   then return NULL
3) Set tree = createNode(post[index])
4) index = index + 1
5) if inst = inend
   then return tree
6) declare eindex = n and i = inst
7) while (i <= eindex) repeat step 8 & 9
8) If in[i] >= tree->data then break
9) else i = i+1
10) Set index = i
11) tree->right = create (in, post, index+1, inend, index)
12) tree->left = create (in, post, inst, index-1, index)
13) End

PTO →

c) Expression evaluation from given

i) Preorder sequence

let pre be the prefix expression string and s be the stack.

1) Start
2) set len = length(pre) - 1
3) while len >= 0 Repeat step 4 and 5
4) If pre[len] is an operand then s.push(pre[ln])
5) else let a = S.pop() , b = S.pop()
   S.push(a operator b)
6) return S.top()
7) End

ii) Postorder sequence

let post be the postfix expression string and s be the stack

1) Start
2) Set ln = 0
3) while ln < length(post) Repeat Steps 4 and 5
4) I: If post[ln] is an operand then s.push(post[ln])
5) Else Set a = S.pop() , b = S.pop()
   Set s.push(a operator b)
6) Return S.top()
7) end

iii) Inorder Sequence

Let "in" be the infix expression and operator & operand be two stacks and "calc" be an utility function.

→ calc()

1) Pop out two values from operand stack A and B
2) pop out a value from operator stack
3) Push (A operator B) in operand stack

→ Required Function

1) set len = 0
2) while len < length(in), Repeat steps 3 to 5
3) If IsEmpty(operator) then operator.push(In[len])
   Else If IsEmpty(operator) <> True → [precedence]
      If prec(In[len]) >= prec(operator.top())
         then operator.push(In[len])
      Else (operator)
         while (IsEmpty <> true and prec(In(len) < prec(operator.top())
         perform calc()
      End
4) If In[len] == '(' then
      operator.push(In[len])
5) If In[len] == ')' then
      while (IsEmpty(operator) <> true and In[len] <> '('
         calc()
      End
      operator.pop()
6) End

Q2> Write down the algorithms and implement the following heap operations by assumming max-heap structure.

a) Build max heap

1) Start
2) set sid = (n/2) - 1
3) For i = sid to 0 Repeat Step 4.
4) Set large = i
   Set l = 2*i+1
   Set r = 2*i+2
   if (l<n and arr[l] > arr[large])
      then large = l
   if (r<n and arr[r] >= arr[large])
      then large = r
   If large != i
      swap (arr[i], arr[large])
      Heapify (arr, n, large)
   End If
5) End

b) Heapify (int arr[], int n, int i)

0.) Start
1.) Set large = i
2.) set l = 2*i+1 , r = 2*i+2
3.) if (l<r and arr[l] > arr[large])
      large = l
4) if (r<n and arr[r] > arr[large])
      large = r
5) if large != i, swap (arr[i], arr[large])
6) call Heapify (arr, n, large)
End if
(7) (End)

c) Insert a new Element in Existing Heap

Let 'data' be value to be inserted

1) Start
2) Set n = n+1 and i = n
3) set key = element to be inserted (data)
4) while i > 1 and A[parent[i]] < A[i]
   swap (A[i], A[parent[i]])
   i = parent[i]
5) End

d) Extract Node

1) Start
2) If n < 1
   print "Underflow"
3) Set max = A[0]
4) A[0] = A[n-1]
5) n = n-1
6) Heapify (A,0)
7) Return max
8) End

e) Heap sort (Arr, n)

1.) For i = 0 to n-1, Repeat step 2
2) Insert_heap(Arr, n, arr[i])
3) Repeat Step 4 while n > 0
   Extract_max (Arr, n, data)
   n = n-1
4) end

TUTORIAL XIII:
Heap Implementation
U19CS012 [D-12]

Implement the following operations in context to ***Heap Data Structure***:

1) Build Max Heap

2) Heapify Procedure

3) Insert a New Element in the Existing Heap

4.) Extract Max or Delete an Element from Max Heap

5.) Heap Sort

Code:

```c
// Implementation Of Heap Operations
// a. Build max heap
// b. Heapify procedure
// c. Insert a new element in the existing heap
// d. Delete_Max max or delete an element from the max heap
// e. Heap Sort

#include <stdio.h>
#include <stdlib.h>

// Defines the Maximum Size of Heap [Stored in Form Of Array]
#define MAX 1005

// Global Iterator
int i;

// Utility Function to Swap
void swap(int *x, int *y);

// Heapify Procedure to Comlpete Binary Tree to Heap
void heapify(int heap[], int n, int i);

// Utility Function for Insert
void create(int heap[], int n);

// To Display the Max Heap
void Display_Max_Heap(int heap[], int n);

// To Insert Element in Max-Heap
void Insert(int heap[], int *n, int val);
```

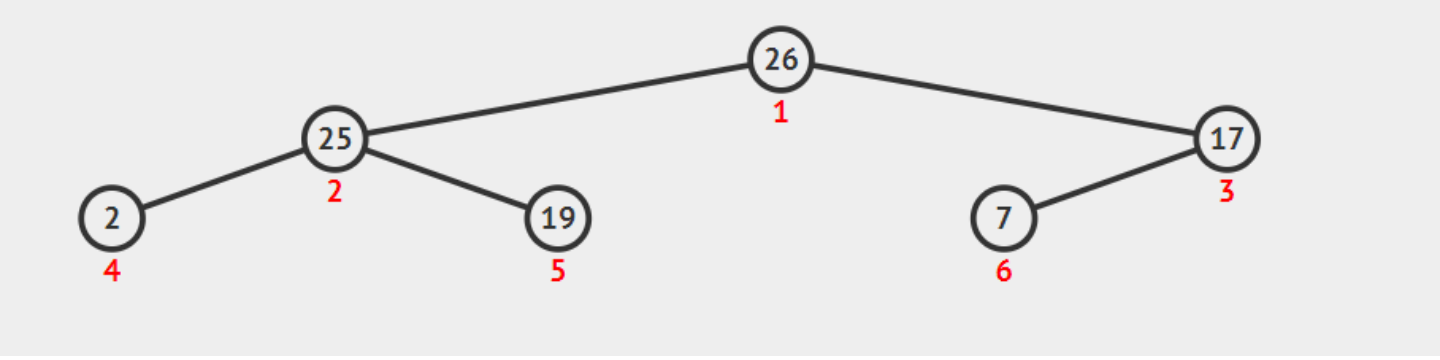
```c
// Delete the Maximum Element in Max Heap
void Delete_Max(int heap[], int *n);

// Function to Implement Heap Sort Algorithm
void Heap_Sort(int heap[], int n);

int main()
{
    int heap[MAX];
    int len = 0;

    int choice;
    printf("\nHEAP\n");

    printf(" 1 -> Insert a New Node in Heap\n");
    printf(" 2 -> Delete Element in Max-Heap\n");
    printf(" 3 -> Heap Sort\n");
    printf(" 4 -> Display Inorder Traversal of Max Heap\n");
    printf(" 5 -> Exit\n");
    int x;

    while (1)
    {
        printf("Enter your choice : ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            printf("Enter Node Value : ");
            scanf("%d", &x);
            Insert(heap, &len, x);
            break;
        case 2:
            Delete_Max(heap, &len);
            break;
        case 3:
            Heap_Sort(heap, len);
            break;
        case 4:
            Display_Max_Heap(heap, len);
            break;
        case 5:
            exit(0);
            break;
        default:
            printf("Enter a Valid Choice!");
            break;
        }
    }
```

```
    return 0;
}

// Utility Function to Swap
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// Heapify Procedure to Comlpete Binary Tree to Heap
void heapify(int heap[], int n, int i)
{
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    int large = i;
    if (l < n && heap[l] > heap[large])
    {
        large = l;
    }
    if (r < n && heap[r] > heap[large])
    {
        large = r;
    }
    if (i != large)
    {
        swap(&heap[i], &heap[large]);
        heapify(heap, n, large);
    }
}

// Utility Function for Insert
void create(int heap[], int n)
{
    for (i = n / 2 - 1; i >= 0; --i)
    {
        heapify(heap, n, i);
    }
}

// To Display the Max Heap
void Display_Max_Heap(int heap[], int n)
{
    printf("MAX HEAP : ");
    for (i = 0; i < n; ++i)
    {
        printf("%d ", heap[i]);
    }
```

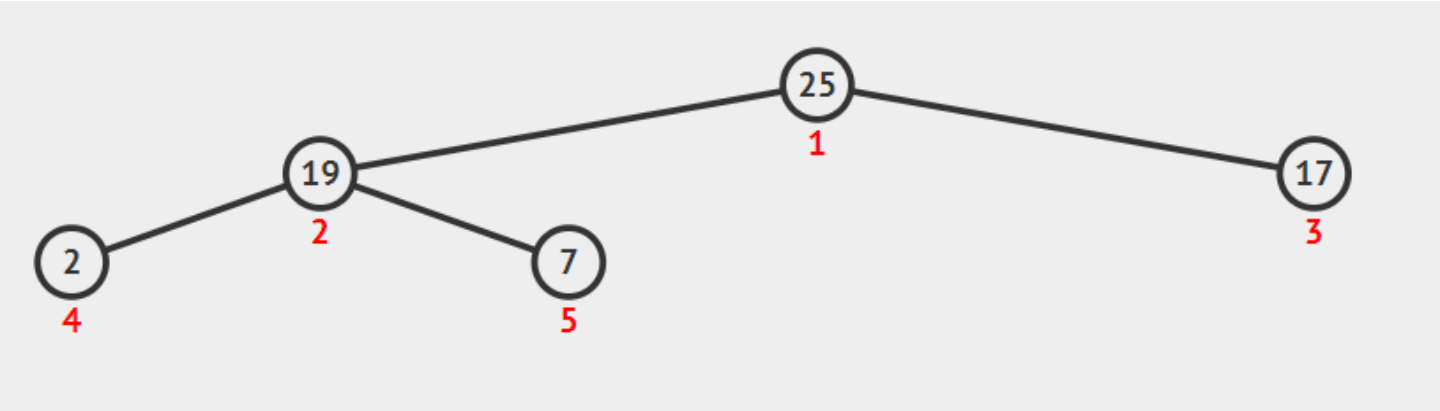
```
    printf("\n");
}

// To Insert Element in Max-Heap
void Insert(int heap[], int *n, int val)
{
    *n = *n + 1;
    heap[*n - 1] = val;
    create(heap, *n);
}

// Delete the Maximum Element in Max Heap
void Delete_Max(int heap[], int *n)
{
    heap[0] = heap[*n - 1];
    *n = *n - 1;
    heapify(heap, *n, 0);
}

// Function to Implement Heap Sort Algorithm
void Heap_Sort(int heap[], int n)
{
    for (i = n - 1; i > 0; --i)
    {
        swap(&heap[0], &heap[i]);
        heapify(heap, i, 0);
    }
    for (i = 0; i < n / 2; ++i)
    {
        swap(&heap[i], &heap[n - 1 - i]);
    }
}
```

Test Cases:

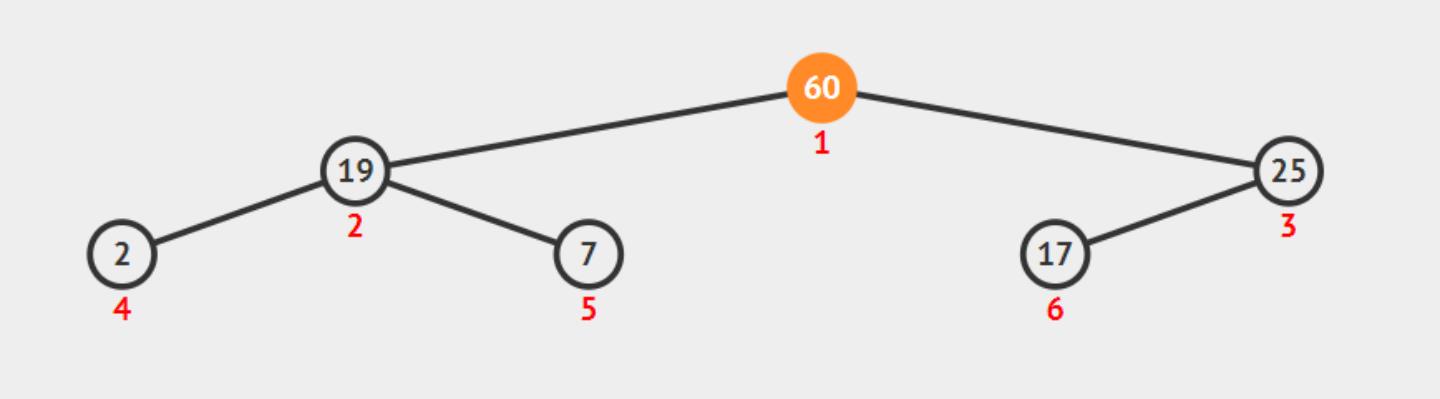
A.) Insertion in Max Heap to form below Heap



B.) After Extract Max or Delete Max Element



C.) After Inserting "60"



D.) In Heap Sort

It will Insert all Elements and Remove the Highest One-by-One.

Resulting in

"60 25 19 17 7 2"

Execution

```
HEAP
 1 -> Insert a New Node in Heap
 2 -> Delete Element in Max-Heap
 3 -> Heap Sort
 4 -> Display Inorder Traversal of Max Heap
 5 -> Exit
Enter your choice : 1
Enter Node Value : 2
Enter your choice : 1
Enter Node Value : 7
Enter your choice : 1
Enter Node Value : 26
Enter your choice : 1
Enter Node Value : 25
Enter your choice : 1
Enter Node Value : 19
Enter your choice : 1
Enter Node Value : 17
Enter your choice : 4
MAX HEAP : 26 25 17 2 19 7
Enter your choice : 2
Enter your choice : 4
MAX HEAP : 25 19 17 2 7
Enter your choice : 1
Enter Node Value : 60
Enter your choice : 4
MAX HEAP : 60 19 25 2 7 17
Enter your choice : 3
Enter your choice : 4
MAX HEAP : 60 25 19 17 7 2
Enter your choice : 5
```