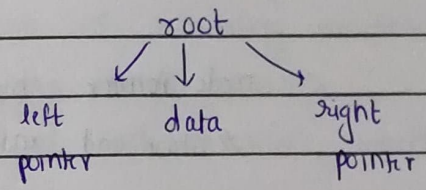
TUTORIAL - 13 DATA STRUCTURE

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