DATA STRUCTURE USING C LABORATORY		SEMESTER	I
Course Code	BCAL208	CIE Marks	50
Teaching Hours/Week (L: P: SDA)	1:2:0	SEE Marks	50
Total Hours of Pedagogy	14 Sessions	Total Marks	100
Credits	02	Exam Hours	03
Type of Course SEE	Practical		

COURSE OBJECTIVES:

- Equip students with the skills to implement and manipulate fundamental data structures like arrays, stacks, queues, and linked lists.
- Enable students to solve complex problems using recursion, matrix operations, and tree traversal techniques.

PART-A

1. **Write a program to implement and demonstrate the following operations on an array: Insertion, Deletion, Traversing, and Searching.**

Algorithm.

Array - Insertion Operation:

```
1. Start

2. Create an Array of a desired datatype and size.

3. Initialize a variable 'i' as 0.

4. Enter the element at ith index of the array.

5. Increment i by 1.

6. Repeat Steps 4 & 5 until the end of the array.

7. Stop
```

Array - Deletion Operation:

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Array - Search Operation:

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Array - Traversal Operation:

```
1 Start
2. Initialize an Array of certain size and datatype.
3. Initialize another variable i with 0.
4. Print the ith value in the array and increment i.
5. Repeat Step 4 until the end of the array is reached.
6. End
```

Program:

```
#include <stdio.h>
#define SIZE 5

void display(int arr[], int n) {
   printf("Array: ");
   for(int i=0; i<n; i++) printf("%d ", arr[i]);
   printf("\n");
}

int search(int arr[], int n, int key) {
   for(int i=0; i<n; i++)
      if(arr[i] == key) return i;
   return -1;
}

void insert(int arr[], int *n, int pos, int val) {
   if(*n >= SIZE) {
      printf("Overflow!\n");
      return;
   }
   for(int i=*n; i>pos; i--) arr[i] = arr[i-1];
   arr[pos] = val;
   (*n)++;
}

void delete(int arr[], int *n, int pos) {
   if(*n <= 0) {
```

```
        printf("Underflow!\n");
        return;
    }
    for(int i=pos; i<*n-1; i++) arr[i] = arr[i+1];
    (*n)--;
}
int main() {
    int arr[SIZE] = {1, 2, 3}, n = 3;
    display(arr, n);
    insert(arr, &n, 1, 5);
    printf("After insertion: ");
    display(arr, n);

    delete(arr, &n, 2);
    printf("After deletion: ");
    display(arr, n);

    int key = 5;
    int pos = search(arr, n, key);
    if(pos != -1) printf("%d found at index %d\n", key, pos);
    else printf("%d not found\n", key);
    return 0;
}
```

Output:

Array: 1 2 3

After insertion: Array: 1 5 2 3

After deletion: Array: 1 5 3

5 found at index 1

2. Write a recursive program to solve Towers of Hanoi problem for n disks

Algorithm.

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, **source**, **destination** and **aux** (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks –

- First, we move the smaller (top) disk to aux peg.
- Then, we move the larger (bottom) disk to destination peg.
- And finally, we move the smaller disk from aux to destination peg.

So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (n^{th} disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk **n** from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are –

```
Step 1 - Move n-1 disks from source to aux
Step 2 - Move nth disk from source to dest
Step 3 - Move n-1 disks from aux to dest
```

A recursive algorithm for Tower of Hanoi can be driven as follows –

```
START
Procedure Hanoi(disk, source, dest, aux)
    IF disk == 1, THEN
        move disk from source to dest
    ELSE
        Hanoi(disk - 1, source, aux, dest) // Step 1
        move disk from source to dest // Step 2
        Hanoi(disk - 1, aux, dest, source) // Step 3
    END IF
```

END Procedure

STOP

Program.

```
#include <stdio.h>
void towers(int n, char from, char to, char aux) {
   if(n == 1) {
      printf("Move disk 1 from %c to %c\n", from, to);
      return;
   }
   towers(n-1, from, aux, to);
   printf("Move disk %d from %c to %c\n", n, from, to);
   towers(n-1, aux, to, from);
}
int main() {
   int n = 3;
   printf("Tower of Hanoi solution for %d disks:\n", n);
   towers(n, 'A', 'C', 'B');
   return 0;
}
```

Output:

Tower of Hanoi solution for 3 disks:

Move disk 1 from A to C

Move disk 2 from A to B

Move disk 1 from C to B

Move disk 3 from A to C

Move disk 1 from B to A

Move disk 2 from B to C

Move disk 1 from A to C

Explanation:

The recursive solution moves n disks from source peg to destination peg using an auxiliary peg:

1. Move n-1 disks from source to auxiliary peg

2. Move nth disk from source to destination

3. Move n-1 disks from auxiliary to destination peg

3. Write a recursive program to calculate Greatest Common Divisor (GCD) of two numbers.

Algorithm.

Refer an algorithm given below to find the greatest common divisor (GCD) for the given two numbers by using the recursive function.

Step 1 – Define the recursive function.

Step 2 – Read the two integers a and b.

Step 3 – Call recursive function.

a. if i>j

b. then return the function with parameters i,j

c. if i==0

d. then return j

e. else return the function with parameters i, j%i.

Program:

```
#include <stdio.h>
int gcd(int a, int b) {
   if(b == 0) return a;
   return gcd(b, a % b);
}
int main() {
   int a = 48, b = 18;
   printf("GCD of %d and %d is %d\n", a, b, gcd(a, b));
   return 0;
}
```

Output:

GCD of 48 and 18 is 6

Explanation:

This program calculates GCD using Euclid's algorithm:

- If b=0, GCD is a

- Else GCD is GCD of b and a%b (remainder of a/b)

4. Write a recursive program to calculate factorial of a number.

Alogorithm.

Problem Statement:

Given a non-negative integer `n`, compute the factorial of `n` (denoted as `n!`) using a recursive approach.

Factorial Definition:

- `n! = n * (n-1) * (n-2) * ... * 1`
- `0! = 1` (by definition)

Recursive Approach:

1. **Base Case:** If `n` is 0 or 1, return 1 (since `0! = 1` and `1! = 1`).
2. **Recursive Case:** For `n > 1`, return `n * factorial(n - 1)`.

Algorithm Steps:

1. Check if `n == 0` or `n == 1`:
   - If yes, return 1.
2. Otherwise:
   - Return `n * factorial(n - 1)`.

Program:

```
#include <stdio.h>
int fact(int n) {
   if(n == 0) return 1;
   return n * fact(n-1);
}
int main() {
   int n = 5;
   printf("Factorial of %d is %d\n", n, fact(n));
   return 0;
}
```

Output:

Factorial of 5 is 120

Explanation:

- Base case: 0! = 1
- Recursive case: n! = n × (n-1)!

5. Write a program to implement linear searching technique on an array.

Algorithm:

Algorithm

```
Linear_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search

Step 1: set pos = -1
Step 2: set i = 1
Step 3: repeat step 4 while i <= n
Step 4: if a[i] == val
set pos = i
print pos
go to step 6
[end of if]
set ii = i + 1
[end of loop]
Step 5: if pos = -1
print "value is not present in the array "
[end of if]
Step 6: exit
```

Program:

```
#include <stdio.h>
int linearSearch(int arr[], int n, int key) {
   for(int i=0; i<n; i++)
      if(arr[i] == key) return i;
   return -1;
}
int main() {
   int arr[] = {10, 20, 30, 40, 50};
   int n = sizeof(arr)/sizeof(arr[0]);
   int key = 30;
```

```
    int pos = linearSearch(arr, n, key);
    if(pos != -1) printf("%d found at index %d\n", key, pos);
    else printf("%d not found\n", key);

    return 0;
}
```

Output:

30 found at index 2

Explanation:

Linear search checks each element sequentially until the key is found or end of array is reached.

6. Write a program to implement binary searching technique on an array.

Algorithm:

```
Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
Step 2: repeat steps 3 and 4 while beg <=end
Step 3: set mid = (beg + end)/2
Step 4: if a[mid] = val
set pos = mid
print pos
go to step 6
else if a[mid] > val
set end = mid - 1
else
set beg = mid + 1
[end of if]
[end of loop]
Step 5: if pos = -1
print "value is not present in the array"
[end of if]
Step 6: exit
```

Program:

```
#include <stdio.h>
int binarySearch(int arr[], int l, int r, int key) {
   while(l <= r) {
      int m = l + (r-l)/2;
      if(arr[m] == key) return m;
      if(arr[m] < key) l = m+1;
      else r = m-1;
   }
   return -1;
}
```

```
int main() {
   int arr[] = {10, 20, 30, 40, 50};
   int n = sizeof(arr)/sizeof(arr[0]);
   int key = 40;
   int pos = binarySearch(arr, 0, n-1, key);
   if(pos != -1) printf("%d found at index %d\n", key, pos);
   else printf("%d not found\n", key);
   return 0;
}
```

Output:

40 found at index 3

Explanation:

Binary search works on sorted arrays by repeatedly dividing the search interval in half.

7. Write a program to implement a stack using an array. Include the operations Push, Pop, and display the current stack.

Algorithm:

Algorithm

1. Checks if the stack is full.
2. If the stack is full, produces an error and exit.
3. If the stack is not full, increments top to point next empty space.
4. Adds data element to the stack location, where top is pointing.
5. Returns success.

Algorithm

1. Checks if the stack is empty.
2. If the stack is empty, produces an error and exit.
3. If the stack is not empty, accesses the data element at which top is pointing.
4. Decreases the value of top by 1.

`5. Returns success.`

Program:

```
#include <stdio.h>
#define SIZE 5

int stack[SIZE], top = -1;
void push(int val) {
   if(top >= SIZE-1) printf("Stack Overflow!\n");
   else stack[++top] = val;
}
void pop() {
   if(top < 0) printf("Stack Underflow!\n");
   else printf("Popped: %d\n", stack[top--]);
}
void display() {
   if(top < 0) printf("Stack is empty!\n");
   else {
      printf("Stack: ");
      for(int i=0; i<=top; i++) printf("%d ", stack[i]);
      printf("\n");
   }
}

int main() {
   push(10);
   push(20);
   push(30);
   display();
```

```
    pop();
    display();
    return 0;
}
```

Output:

Stack: 10 20 30

Popped: 30

Stack: 10 20

Explanation:

This implements a stack with:

- Push: Adds element at top

- Pop: Removes top element

- Display: Shows stack contents.

PART-B

1. Write a program to implement bubble sort technique.

Algorithm:

```
bubbleSort(array)
n = length(array)
repeat
  swapped = false
  for i = 1 to n - 1
      if array[i - 1] > array[i], then
      swap(array[i - 1], array[i])
      swapped = true
      end if
   end for
   n = n - 1
 until not swapped
end bubbleSort
```

Steps of Bubble Sort

Step 1: Compare the first two elements of the input array.

Step 2: If the first element is greater than the second element, swap them; otherwise, swapping is not required.

Step 3: According to step 2, do the comparison and swapping of the next pair of elements of the input array (the second and the third element), i.e. if the second element is greater than the third element, do the swapping; otherwise, not.

Step 4: Continue this process till the end of the input array is reached. At this point, the largest element will be "bubble up" and get the end of the list.

Step 5: Repeat the steps from 1 to 4, excluding the last element, as it has already achieved its correct position.

Program:

```
#include <stdio.h>
void bubbleSort(int arr[], int n) {
   for(int i=0; i<n-1; i++)
      for(int j=0; j<n-i-1; j++)
         if(arr[j] > arr[j+1]) {
            int temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
}

int main() {
   int arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr)/sizeof(arr[0]);
   bubbleSort(arr, n);
   printf("Sorted array: ");
   for(int i=0; i<n; i++) printf("%d ", arr[i]);

   return 0;
```

}

Output:

Sorted array: 11 12 22 25 34 64 90

Explanation:

Bubble sort repeatedly swaps adjacent elements if they are in wrong order.

2. Write a program to implement selection sort technique.

Algorithm:

```
// Function to do selection sort on the array 'a' of size 'n'
function selectionSort(a, s)
 // Outer loop: for iterating through the array
 // The last element is not included
 for j = 0 to s - 1
  // Assuming that the current element is the minimum
  // storing its index.
  minIdx = j
  // Inner loop: for finding the minimum element in the unsorted portion
  for k = j + 1 to s - 1
   // If a smaller element is found
   if a[k] < a[minIdx]
    // Updating the index of the element, which is the minimum
    minIdx = k
  // Swapping the minimum element found with the current element
  if minIdx != j
   swap(a[j], a[minIdx])
 // The array is now sorted. return it
 return a
```

Steps of Selection Sort Algorithm

Step 1: Find the smallest element and swap it with the first element of the input array. If the first element is the smallest, then swapping is not required.

Step 2: Find the smallest element from the remaining elements (the second smallest element) and swap it with the second element. Again, here also swapping is not required if the second smallest element is the second element of the array.

Step 3: Repeat the process for the remaining elements of the input array until all elements achieve the correct position.

Program:

```
#include <stdio.h>
void selectionSort(int arr[], int n) {
   for(int i=0; i<n-1; i++) {
      int min_idx = i;
      for(int j=i+1; j<n; j++)
         if(arr[j] < arr[min_idx]) min_idx = j;

      int temp = arr[min_idx];
      arr[min_idx] = arr[i];
      arr[i] = temp;
   }
}
int main() {
   int arr[] = {64, 25, 12, 22, 11};
   int n = sizeof(arr)/sizeof(arr[0]);
   selectionSort(arr, n);
   printf("Sorted array: ");
   for(int i=0; i<n; i++) printf("%d ", arr[i]);
   return 0;
}
```

Output:

Sorted array: 11 12 22 25 64

Explanation:

Selection sort finds the minimum element and swaps it with the first unsorted element.

3. Write a program to implement insertion sort technique.

Algorithm:

The simple steps of achieving the insertion sort are listed as follows -

Step 1: If the element is the first element, assume that it is already sorted. Return 1.

Step 2: Pick the next element and store it separately in a **key.**

Step 3: Now, compare the **key** with all elements in the sorted array.

Step 4: If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5: Insert the value.

Step 6: Repeat until the array is sorted.

```
1. INSERTION-SORT(A)
2. for i ← 1 to length(A) - 1 do
3. key ← A[i]
4. j ← i - 1
5. while j ≥ 0 and A[j] > key do
6. A[j + 1] ← A[j]
7. j ← j - 1
8. A[j + 1] ← key
```

Program:

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
   for(int i=1; i<n; i++) {
      int key = arr[i];
      int j = i-1;
      while(j >= 0 && arr[j] > key) {
         arr[j+1] = arr[j];
         j--;
```

```
        }
        arr[j+1] = key;
    }
}
int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    insertionSort(arr, n);
    printf("Sorted array: ");
    for(int i=0; i<n; i++) printf("%d ", arr[i]);
    return 0;
}
```

Output:

Sorted array: 5 6 11 12 13

Explanation:

Insertion sort builds the final array one item at a time by inserting each element in its correct position.

4. Write a program to convert an infix expression to a postfix expression using stack.

Algorithm:

```
Step 1. Scan the expression from left to right

Step 2. If it is an operand push it to stack

Step 3. If it is an operator pull operand from stack and perform
operation

Step 4. Store the output of step 3, back to stack
```

Step 5. Scan the expression until all operands are consumed

Step 6. Pop the stack and perform operation.

Program:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define SIZE 50

char stack[SIZE];
int top = -1;

void push(char elem) { stack[++top] = elem; }
char pop() { return stack[top--]; }
int precedence(char op) {
   Switch(op) {
      Case '+': case '-': return 1;
      Case '*': case '/': return 2;
      Case '^': return 3;
   }
   Return 0;
}

Void infixToPostfix(char infix[], char postfix[]) {
   Int I = 0, j = 0;
   Char ch, x;
   Push('(');
   Strcat(infix, ")");
```

```
  While((ch = infix[i++]) != '\0') {
    If(ch == '(') push(ch);
    Else if(isalnum(ch)) postfix[j++] = ch;
    Else if(ch == ')') {
      While((x = pop()) != '(') postfix[j++] = x;
    }
    Else {
      While(precedence(stack[top]) >= precedence(ch))
        Postfix[j++] = pop();
      Push(ch);
    }
  }
  Postfix[j] = '\0';
}
int main() {
  Char infix[SIZE] = "a+b*(c^d-e)^(f+g*h)-I";
  Char postfix[SIZE];

  infixToPostfix(infix, postfix);
  printf("Infix: %s\nPostfix: %s\n", infix, postfix);
  return 0;
}
```

Output:

Infix: a+b*(c^d-e)^(f+g*h)-i

Postfix: abcd^e-fgh*+^*+i-

Explanation:

This converts infix expressions to postfix notation using stack to handle operator precedence.

5. Write a program to implement an ordinary queue using an array. Include the operations Enqueue,Dequeue,and Display.

Algorithm:

Queue Operations:

1. **Enqueue(item):** Add an item to the rear of the queue.
2. **Dequeue():** Remove and return the item from the front of the queue.
3. **Display():** Print all elements in the queue from front to rear.

Data Structures Used:

- An array `queue[]` to store elements.
- Two pointers:
  - `front` (index of the front element).
  - `rear` (index of the rear element).
- A constant `MAX_SIZE` to define the maximum capacity of the queue.

Algorithm Steps:

1. Initialize the Queue:

- Set `front = -1` and `rear = -1` (empty queue).

2. Enqueue(item):

1. Check if the queue is full (`rear == MAX_SIZE - 1`):
   - If full, print "Queue Overflow" and return.
2. If the queue is empty (`front == -1`), set `front = 0`.
3. Increment `rear` by 1.
4. Insert `item` at `queue[rear]`.

3. Dequeue():

1. Check if the queue is empty (front == -1 or front > rear):
   - If empty, print "Queue Underflow" and return None.
2. Store queue[front] in a temporary variable.
3. Increment front by 1.
4. If front > rear (queue becomes empty), reset front = rear = -1.
5. Return the stored item.

4. Display():

1. Check if the queue is empty (front == -1 or front > rear):
   - If empty, print "Queue is empty".
2. Else, loop from front to rear and print each element.

Program:

```
#include <stdio.h>
#define SIZE 5

int queue[SIZE], front = -1, rear = -1;

void enqueue(int val) {
   if(rear == SIZE-1) printf("Queue is full!\n");
   else {
      if(front == -1) front = 0;
      Queue[++rear] = val;
   }
}

void dequeue() {
```

```
    if(front == -1 || front > rear) printf("Queue is empty!\n");
    else printf("Dequeued: %d\n", queue[front++]);
}

void display() {
    if(front == -1) printf("Queue is empty!\n");
    else {
        Printf("Queue: ");
        For(int i=front; i<=rear; i++) printf("%d ", queue[i]);
        Printf("\n");
    }
}

int main() {
    Enqueue(10);
    Enqueue(20);
    Enqueue(30);
    Display();
    Dequeue();
    Display();

    Return 0;
}
```

Output:

Queue: 10 20 30

Dequeued: 10

Queue: 20 30

Explanation:

This implements a queue with:

- Enqueue: Adds element at rear
- Dequeue: Removes element from front
- Display: Shows queue contents

6. Write a program to create a singly linked list and perform the following operations: Insertion at the beginning ,Deletion from the beginning ,and Traversal of the linked list.

Algorithm:

1. Node Structure

- Each node has:
  - `data`: Stores the value.
  - `next`: Pointer to the next node.

2. Linked List Operations

- **Insertion at Beginning (**`insert_at_beginning`**)**

1. Create a new node with the given data.
2. Point the new node's `next` to the current `head`.
3. Update `head` to the new node.

- **Deletion from Beginning (**`delete_from_beginning`**)**

1. Check if the list is empty (`head == None`). If yes, return.
2. Store the `head` node in a temporary variable.
3. Move `head` to `head.next`.
4. Free the temporary node (optional in Python due to garbage collection).

- **Traversal (**`display`**)**

1. Start from `head`.
2. Traverse each node while printing its `data`.
3. Stop when `current.next == None`.

Program:

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

Struct Node {
   Int data;
   Struct Node *next;
};

Struct Node *head = NULL;

Void insertAtBeginning(int val) {
   Struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = val;
   newNode->next = head;
   head = newNode;
}

Void deleteFromBeginning() {
   If(head == NULL) printf("List is empty!\n");
   Else {
      Struct Node *temp = head;
      Head = head->next;
      Printf("Deleted: %d\n", temp->data);
      Free(temp);
   }
}
```

```
Void display() {
   Struct Node *current = head;
   Printf("List: ");
   While(current != NULL) {
      Printf("%d ", current->data);
      Current = current->next;
   }
   Printf("\n");
}

Int main() {
   insertAtBeginning(30);
   insertAtBeginning(20);
   insertAtBeginning(10);
   display();

   deleteFromBeginning();
   display();

   return 0;
}
```

Output:

List: 10 20 30

Deleted: 10

List: 20 30

Explanation:

This implements a singly linked list with:

- Insertion at beginning

- Deletion from beginning

- Traversal to display list

7. Write a program to represent a binary tree and implement Pre-order ,In-order and Post-order traversal methods.

Algorithm:

1. Binary Tree Node Structure

- Each node contains:
  - data (integer value)
  - left (pointer to the left child)
  - right (pointer to the right child)

2. Tree Traversal Methods

1. **Pre-order Traversal (Root → Left → Right)**
   - Visit the root.
   - Traverse the left subtree.
   - Traverse the right subtree.
2. **In-order Traversal (Left → Root → Right)**
   - Traverse the left subtree.
   - Visit the root.
   - Traverse the right subtree.
3. **Post-order Traversal (Left → Right → Root)**
   - Traverse the left subtree.
   - Traverse the right subtree.
   - Visit the root.

Program:

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

Struct Node {
   Int data;
   Struct Node *left, *right;
};

Struct Node* createNode(int val) {
   Struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = val;
   newNode->left = newNode->right = NULL;
   return newNode;
}

Void preorder(struct Node* root) {
   If(root == NULL) return;
   Printf("%d ", root->data);
   Preorder(root->left);
   Preorder(root->right);
}

Void inorder(struct Node* root) {
   If(root == NULL) return;
   Inorder(root->left);
   Printf("%d ", root->data);
   Inorder(root->right);
```

```
}

Void postorder(struct Node* root) {
   If(root == NULL) return;
   Postorder(root->left);
   Postorder(root->right);
   Printf("%d ", root->data);
}
Int main() {
   Struct Node* root = createNode(1);
   Root->left = createNode(2);
   Root->right = createNode(3);
   Root->left->left = createNode(4);
   Root->left->right = createNode(5);

   Printf("Preorder: "); preorder(root); printf("\n");
   Printf("Inorder: "); inorder(root); printf("\n");
   Printf("Postorder: "); postorder(root); printf("\n");

   Return 0;
}
```

Output:

Preorder: 1 2 4 5 3

Inorder: 4 2 5 1 3

Postorder: 4 5 2 3 1

Explanation:

This demonstrates three tree traversal methods:

- Preorder: Root, Left, Right

- Inorder: Left, Root, Right

- Postorder: Left, Right, Root