	BANGLADESH UNIVERSITY Department of Computer Science & Engineering **CSE2104 – Data Structure**

PROJECT

SUBMITTED BY:

Student's Name : Sheoti

Student ID : 202221063026

Course Code : CSE2104

Course Title : Data Structure

SUBMITTED TO:

Pabon Shaha

Lecturer

Department of Computer Science & Engineering

Bangladesh University

Date of Submission : 07/08/2023

Marks:

Project 01

Implementation of Stack with its operations:

- **Introduction of Stack:**
  - **Stack:** A stack is a linear data structure in which elements can only be added and removed from one side of the list, known as the top.
  - **Working process:** A stack's working process adheres to the Last In, First Out (LIFO) concept. It signifies that the most recently added item to the stack is the first to be deleted. When you add an item to the stack, it takes over as the top element. When you pop an item, the top element is eliminated, and the element beneath it takes its place. Elements are added and withdrawn in sequential order in this manner, resulting in a "stack" of items. In computer science, stacks are commonly used to manage function calls, expression evaluation, memory management, and other tasks.
  - **Operations:** Here's a brief overview of the stack's basic operations:
    - **push():** Adding an element to the top of the stack.
    - **pop():** Removing the top element from the stack.
    - **peek():** Viewing the top element without removing the element.
    - **isEmpty():** Checking if the stack is empty.
    - **isFull():** Checking if the stack is full.
    - **Size():** Getting the number of elements in the stack.
  - **Advantages:**
    - Efficient memory utilization
    - Helps in function calls
    - Supports backtracking
    - Used in Compiler Design
    - Fast access time
    - Enables undo/redo operations
    - Easy implementation
  - **Disadvantages:**
    - Stack overflow and underflow
    - Memory management
    - Limited capacity
    - No random access
    - Not suitable for certain applications
    - Recursive function calls limitations

- **Brief of my project:**
  - Implementation of Stack operations
  - Clarify basic idea about Stack.

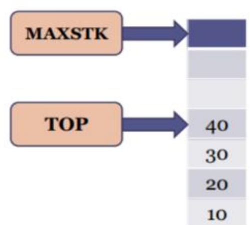
Algorithms for Stack operations:

- **push():**

1. IF(TOP=O)
2. UNDERFLOW
3. ITEM:= STACK[TOP]
4. TOP:=TOP-1

- **pop():**

1. IF(TOP=MAXSTK)
2. OVERFLOW
3. TOP:=TOP+1
4. STACK[TOP]:=ITEM



- **peek():**

```
FUNCTION peek(stack):
    IF stack is empty:
        PRINT "Stack is Empty"
    ELSE:
        RETURN stack[top]
```

- **isEmpty():**

```
FUNCTION isEmpty(stack):
    IF top is equal to -1:
        RETURN True
    ELSE:
        RETURN False
```

- **isFull():**

```
int isFull(stack){
    if(top == MAXSIZE)
        return 1;
    else
        return 0;
}
```

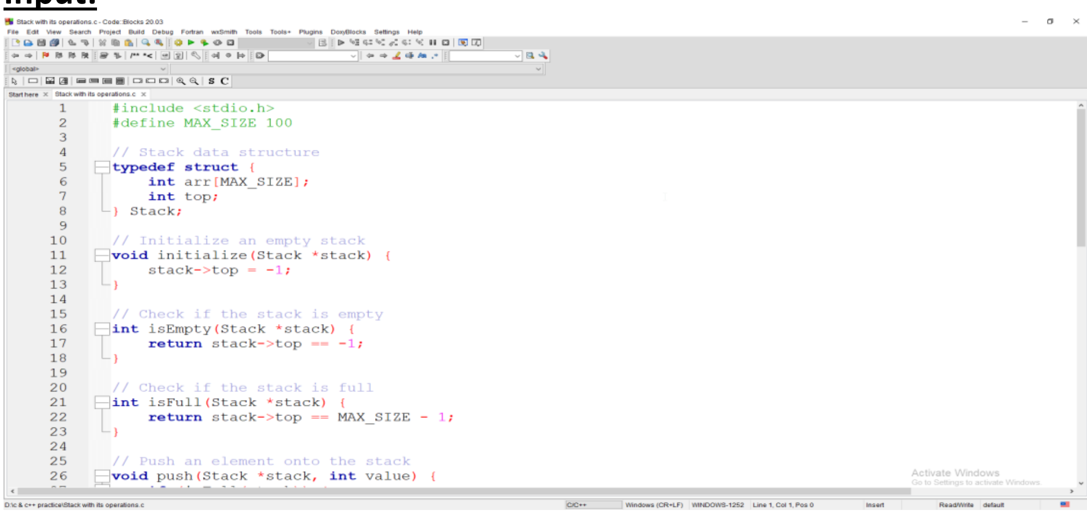
- **Size():**

```
FUNCTION size(stack):
    RETURN top + 1
```

➢ Requirement analysis:

- Laptop
- CodeBlocks
- Ms word 16
- Sniping tool
- Mouse
- Keyboard

➢ Input:



```c
// Push an element onto the stack
void push(Stack *stack, int value) {
    if (isFull(stack)) {
        printf("Stack overflow. Cannot push more elements.\n");
    } else {
        stack->arr[++(stack->top)] = value;
    }
}

// Pop an element from the stack
int pop(Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow. Cannot pop from an empty stack.\n");
        return -1;
    } else {
        return stack->arr[(stack->top)--];
    }
}

// Peek the top element of the stack without removing it
int pick(Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty.\n");
        return -1;
    } else {
        return stack->arr[stack->top];
```

```c
int pick(Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty.\n");
        return -1;
    } else {
        return stack->arr[stack->top];
    }
}

int main() {
    Stack stack;
    initialize(&stack);

    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);

    printf("Top element of the stack: %d\n", pick(&stack));

    while (!isEmpty(&stack)) {
        printf("Popped: %d\n", pop(&stack));
    }

    return 0;
}
```

➢ **Output:**

"D:\c & c++ practice\Stack with its operations.exe"

```
Top element of the stack: 30
Popped: 30
Popped: 20
Popped: 10

Process returned 0 (0x0)   execution time : 0.022 s
Press any key to continue.
```

Project 02

Implementation of Queue with its operations:

- **Introduction of Queue:**
  - **Queue:** A Queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.
  - **Working process:** A stack's working process adheres to the First In, First Out (FIFO) concept. Two-pointers FRONT and REAR.FRONT track the first element of the queue. REAR tracks the last element of the queue. Initially, set value of FRONT and REAR to -1.
  - **Operations:** Here's a brief overview of the Queue's basic operations:
    - **enqueue():** Adding an element to the end of the queue.
    - **dequeue():** Removing the front element from the queue.
    - **peek():** Viewing the top element without removing the element.
    - **isEmpty():** Checking if the queue is empty.
    - **isFull():** Checking if the queue is full.
    - **Size():** Getting the number of elements in the queue.
  - **Advantages:**
    - Queues can be used in the implementation of other data structures.
    - Queues are fast in speed for data inter-process communication.
    - Operations such as insertion and deletion can be performed with ease as it follows the first in first out rule.
    - A large amount of data can be managed efficiently with ease.
    - Queues are useful when a particular service is used by multiple consumers.
  - **Disadvantages:**
    - Maximum size of a queue must be defined prior.
    - Searching an element takes O(N) time.
    - Limited Space.
    - In a classical queue, a new element can only be inserted when the existing elements are deleted from the queue.
    - The operations such as insertion and deletion of elements from the middle are time-consuming.
  - **Brief of my project:**
    - Implementation of Queue operations
    - Clarify basic idea about Queue.

Algorithms for Queue operations:

enqueue():

- check if the queue is full
- for the first element, set the value of FRONT to 0
- increase the REAR index by 1
- add the new element in the position pointed to by REAR

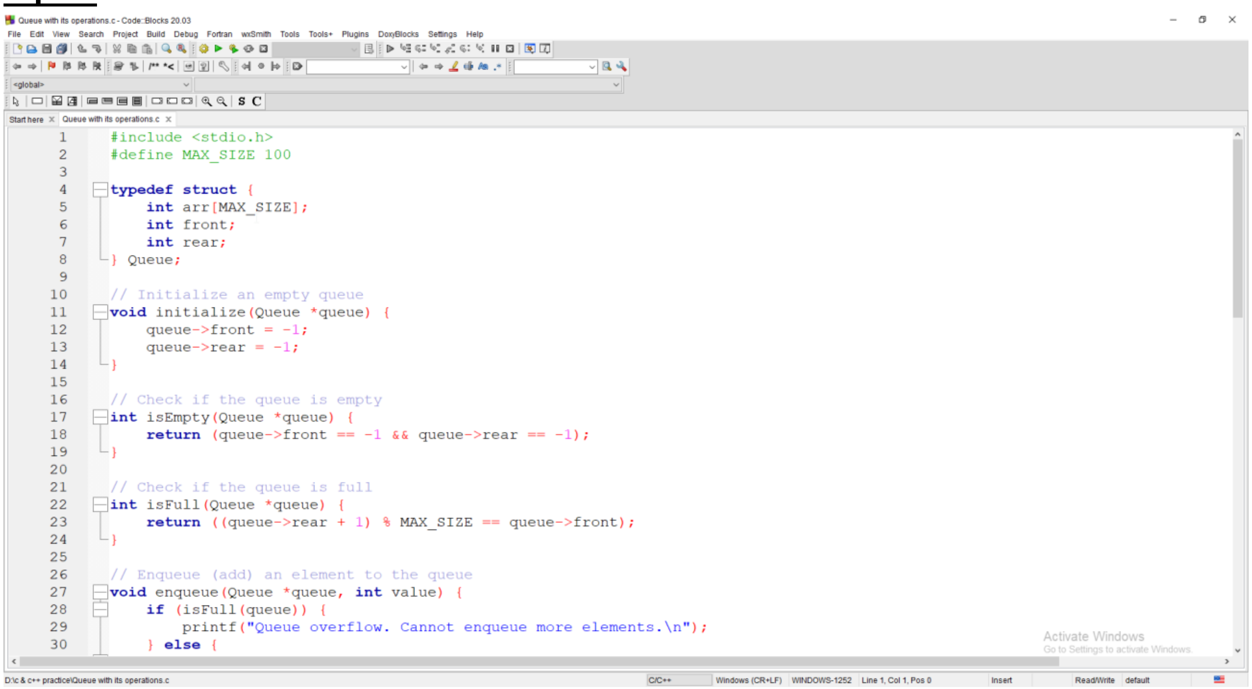
dequeue():

- check if the queue is empty
- return the value pointed by FRONT
- increase the FRONT index by 1
- for the last element, reset the values of FRONT and REAR to -1

Requirement analysis:

- Laptop
- CodeBlocks
- Ms word 16
- Sniping tool
- Mouse
- Keyboard

Input:



```c

// Enqueue (add) an element to the queue
void enqueue(Queue *queue, int value) {
    if (isFull(queue)) {
        printf("Queue overflow. Cannot enqueue more elements.\n");
    } else {
        if (isEmpty(queue)) {
            queue->front = 0;
            queue->rear = 0;
        } else {
            queue->rear = (queue->rear + 1) % MAX_SIZE;
        }
        queue->arr[queue->rear] = value;
    }
}

// Dequeue (remove) an element from the queue
int dequeue(Queue *queue) {
    if (isEmpty(queue)) {
        printf("Queue underflow. Cannot dequeue from an empty queue.\n");
        return -1;
    } else {
        int value = queue->arr[queue->front];
        if (queue->front == queue->rear) {
            // Only one element in the queue, reset to empty queue state
            queue->front = -1;
            queue->rear = -1;
        } else {
            queue->front = (queue->front + 1) % MAX_SIZE;
        }
```

```c

// Dequeue (remove) an element from the queue
int dequeue(Queue *queue) {
    if (isEmpty(queue)) {
        printf("Queue underflow. Cannot dequeue from an empty queue.\n");
        return -1;
    } else {
        int value = queue->arr[queue->front];
        if (queue->front == queue->rear) {
            // Only one element in the queue, reset to empty queue state
            queue->front = -1;
            queue->rear = -1;
        } else {
            queue->front = (queue->front + 1) % MAX_SIZE;
        }
        return value;
    }
}

// Peek the front element of the queue without removing it
int peek(Queue *queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty.\n");
        return -1;
    } else {
        return queue->arr[queue->front];
    }
}

int main() {
```

```c
    }
}

// Peek the front element of the queue without removing it
int peek(Queue *queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty.\n");
        return -1;
    } else {
        return queue->arr[queue->front];
    }
}

int main() {
    Queue queue;
    initialize(&queue);

    enqueue(&queue, 10);
    enqueue(&queue, 20);
    enqueue(&queue, 30);

    printf("Front element of the queue: %d\n", peek(&queue));

    while (!isEmpty(&queue)) {
        printf("Dequeued: %d\n", dequeue(&queue));
    }

    return 0;
}
```

- **Output:**

"D:\c & c++ practice\Queue with its operations.exe"

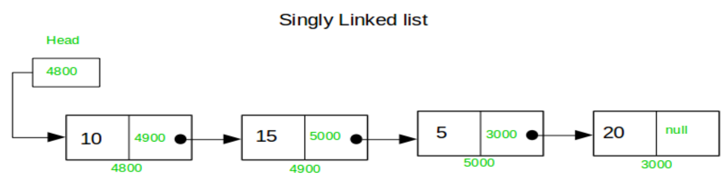
```
Front element of the queue: 10
Dequeued: 10
Dequeued: 20
Dequeued: 30

Process returned 0 (0x0)   execution time : 0.021 s
Press any key to continue.
```

Project 03

Describe the Singly Linked List with proper diagram and definition:

➢ **Linked List:** A linked list is a linear data structure that includes a series of connected nodes.

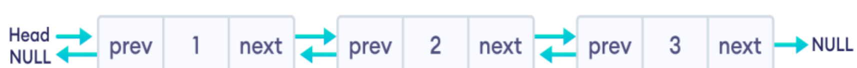
➢ **Singly Linked List:** The singly linked list is a linear data structure in which each element of the list contains a pointer that points to the next element in the list. Each element in the singly linked list is called a node. Each node has two components: data and a pointer next which points to the next node in the list.



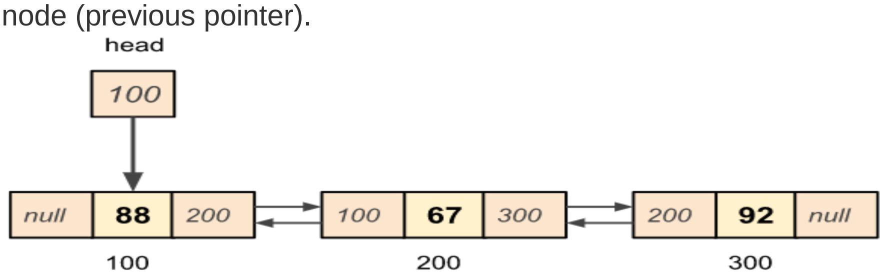
Project 04

Describe the Doubly Linked List with proper diagram and definition:

➢ **Linked List:** A linked list is a linear data structure that includes a series of connected nodes.



➢ **Doubly Linked List:** Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), and pointer to the previous node (previous pointer).

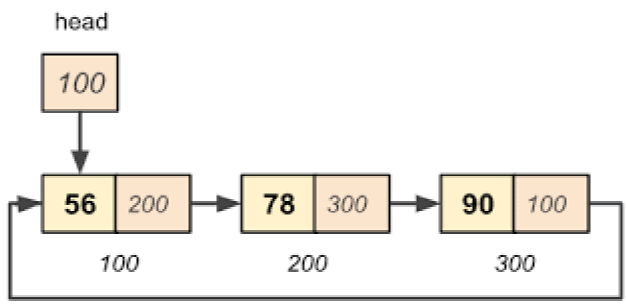


Project 05

Describe the Circular Linked List with proper diagram and definition:

- **Linked List:** A linked list is a linear data structure that includes a series of connected nodes.

- **Circular Linked List:** Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

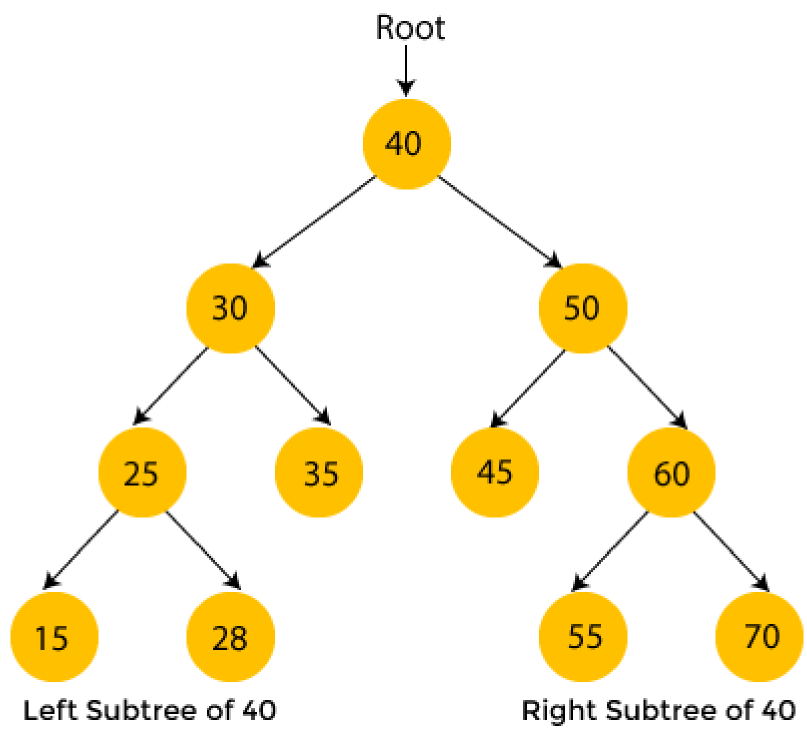


Project 06

Implementation the Binary Search Tree with linked list:

➢ Binary search tree:

- A Binary Search Tree (BST) is a special type of binary tree in which-
- The left child of a node has a value less than the parent node's value.
- The right child has a value greater than the parent node's value.
- This property is called the BST property.
- It makes it possible to efficiently search, insert, and delete elements in the tree.



➢ Requirement analysis:

- Laptop
- CodeBlocks
- Ms word 16
- Sniping tool
- Mouse
- Keyboard

- **Input:**

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

// Binary Search Tree (BST) node structure
typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
} Node;

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to insert a node in the BST
Node* insert(Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }

    if (data < root->data) {
        root->left = insert(root->left, data);
    } else if (data > root->data) {
        root->right = insert(root->right, data);
    }

    return root;
}

```

```c

// Function to search for a value in the BST
Node* search(Node* root, int key) {
    if (root == NULL || root->data == key) {
        return root;
    }

    if (key < root->data) {
        return search(root->left, key);
    } else {
        return search(root->right, key);
    }
}

// Function to perform an inorder traversal of the BST (left-root-right)
void inorderTraversal(Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}

// Function to perform a preorder traversal of the BST (root-left-right)
void preorderTraversal(Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}

// Function to perform a postorder traversal of the BST (left-right-root)
void postorderTraversal(Node* root) {
```

```c
}

// Function to perform a postorder traversal of the BST (left-right-root)
void postorderTraversal(Node* root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->data);
    }
}

int main() {
    Node* root = NULL;

    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    printf("Inorder Traversal: ");
    inorderTraversal(root);
    printf("\n");

    printf("Preorder Traversal: ");
    preorderTraversal(root);
    printf("\n");

    printf("Postorder Traversal: ");
    postorderTraversal(root);
    printf("\n");
```


```c
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    printf("Inorder Traversal: ");
    inorderTraversal(root);
    printf("\n");

    printf("Preorder Traversal: ");
    preorderTraversal(root);
    printf("\n");

    printf("Postorder Traversal: ");
    postorderTraversal(root);
    printf("\n");
```


```c

    int key;
    printf("Search a number in the BST: ");
    scanf("%d", &key);
    Node* result = search(root, key);
    if (result != NULL) {
        printf("%d found in the BST.\n", key);
    } else {
        printf("%d not found in the BST.\n", key);
    }

    return 0;
}
```

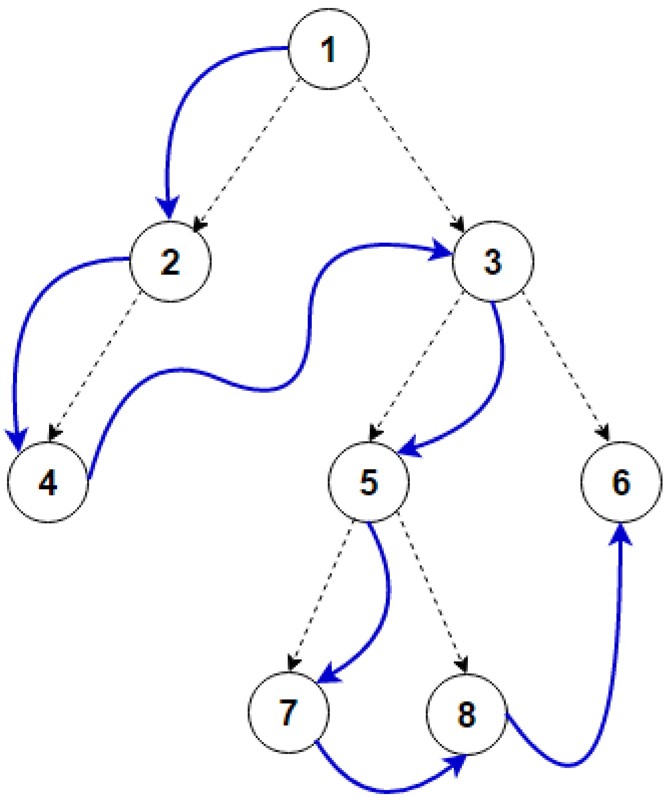
Output:

```
"D:\c & c++ practice\The Binary Search Tree with LinkedList.exe"
Inorder Traversal: 20 30 40 50 60 70 80
Preorder Traversal: 50 30 20 40 70 60 80
Postorder Traversal: 20 40 30 60 80 70 50
Search a number in the BST: 30
30 found in the BST.

Process returned 0 (0x0)   execution time : 4.757 s
Press any key to continue.
```

Project 07

Describe the traverse tree in preorder method:

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.



Preorder: 1, 2, 4, 3, 5, 7, 8, 6

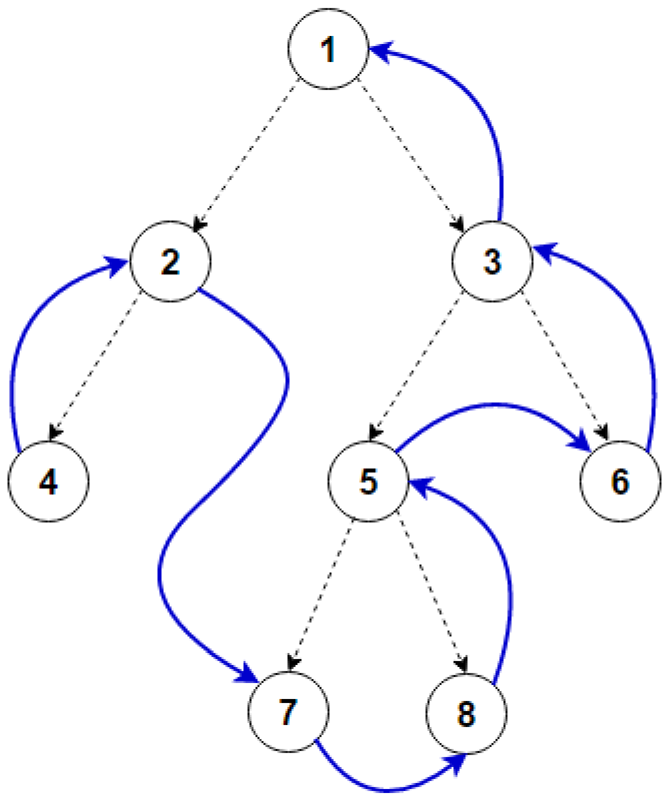
- **Pre-order Traversal:**
  - Start with the root node 1.
  - First, print 1 and then recursively traverse the left subtree.
  - Now, move to the left subtree.
  - For the left subtree, the root node is 2.
  - Print 2, and move towards the left subtree of 2.
  - In the left subtree of 2, there is an element 4.

- Print 4, and 4 has no subtree.
- So that, move towards the right subtree of 2. But, 2 has no right subtree.
- So, move towards the right subtree of 1.
- Now, move to the right subtree.
- For the right subtree, the root node is 3.
- Print 3, and move towards the left subtree of 3.
- In the left subtree of 3, there is an element 5.
- Print 5, and move towards the left subtree of 5.
- In the left subtree of 5, there is an element 7.
- Print 7, and 7 has no subtree.
- So that, move towards the right subtree of 5.
- In the right subtree of 5, there is an element 8.
- Print 8, and 8 has no subtree.
- So that, move towards the right subtree of 3.
- In the right subtree of 3, there is an element 6.
- Print 6, and 6 has no subtree.

Project 08

Describe the traverse tree in post-order method:

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.



Postorder: 4, 2, 7, 8, 5, 6, 3, 1

➢ **Post-order Traversal:**

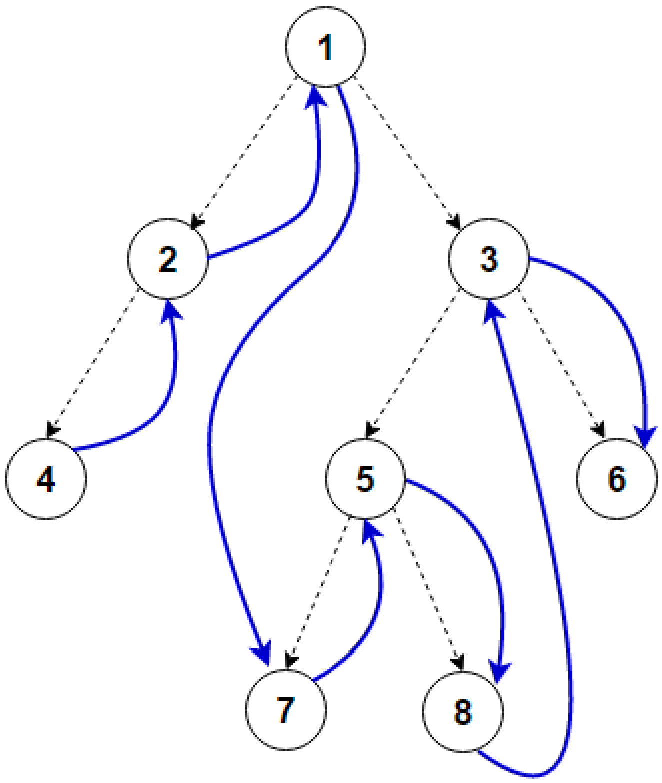
- Start with the root node 1.
- First, visit the left subtree.
- Now, move to the left subtree.
- For the left subtree, the root node is 2.
- Move towards the left subtree of 2.
- In the left subtree of 2, there is an element 4.
- 4 has no subtree and print 4.

- So that, move towards the right subtree of 2. But, 2 has no right subtree.
- Print 2 and move towards the right subtree of 1.
- Now, move to the right subtree.
- For the right subtree, the root node is 3.
- Move towards the left subtree of 3.
- In the left subtree of 3, there is an element 5.
- Move towards the left subtree of 5.
- In the left subtree of 5, there is an element 7.
- 7 has no subtree and print 7.
- Now, move towards the right subtree of 5.
- In the right subtree of 5, there is an element 8.
- 8 has no subtree and print 8 then print 5.
- Now, move towards the right subtree of 3.
- In the right subtree of 3, there is an element 6.
- 6 has no subtree and print 6.
- Then print 3 and at the end print 1

<u>**Project 09**</u>

Describe the traverse tree in In-order method:

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.



Inorder: 4, 2, 1, 7, 5, 8, 3, 6

➢ <u>**In-order Traversal:**</u>

- Start with the root node 1.
- First, visit the left subtree.
- Now, move to the left subtree.
- For the left subtree, the root node is 2.
- Move towards the left subtree of 2.
- In the left subtree of 2, there is an element 4.
- 4 has no subtree and print 4.

- So that, move towards the right subtree of 2. But, 2 has no right subtree. So, print 2.
- Then print 1 and move towards the right subtree of 1.
- Now, move to the right subtree.
- For the right subtree, the root node is 3.
- Move towards the left subtree of 3.
- In the left subtree of 3, there is an element 5.
- Move towards the left subtree of 5.
- In the left subtree of 5, there is an element 7.
- 7 has no subtree and print 7.
- Now, print 5 and move towards the right subtree of 5.
- In the right subtree of 5, there is an element 8.
- 8 has no subtree and print 8.
- Now, print 3 and move towards the right subtree of 3.
- In the right subtree of 3, there is an element 6.
- 6 has no subtree and print 6.

Project 10

Describe the infix, prefix, postfix expression notation with proper example and algorithm:

- **Infix Notation**: In infix notation, operators are written between the operands. This is the common notation used in mathematics. For example, the expression "2 + 3" is written in infix notation.
- **Prefix Notation:** In prefix notation, operators are written before the operands. This is also known as Polish notation. For example, the expression "+ 2 3" is written in prefix notation.
- **Postfix Notation:** In postfix notation, operators are written after the operands. This is also known as Reverse Polish notation. For example, the expression "2 3 +" is written in postfix notation.
- **Algorithm to Convert Infix to Postfix:**
  - Create an empty stack to hold operators.
  - Scan the infix expression from left to right.
  - If an operand is encountered, append it to the output string.
  - If an operator (+, -, *, /, etc.) is encountered:
    - Pop all the higher or equal precedence operators from the stack and append them to the output string.
    - Push the current operator onto the stack.
  - If a left parenthesis "(" is encountered, push it onto the stack.
  - If a right parenthesis ")" is encountered:
    - Pop operators from the stack and append them to output until a left parenthesis "(" is found.
    - Pop out and discard "(" from stack.
  - Repeat steps 3-6 until all characters are scanned.
  - Pop any remaining operators from stack and append them to output string.
  - The resulting postfix expression will be stored in output string.
- **Example:** Converting Infix Expression "2 + 3 * 4" to Postfix
  - Input: 2 + 3 * 4
  - Output: 2 3 4 * +

- **Step-by-step process:**
  - Read '2', since it's an operand add it directly to output string => Output: "2"
  - Read '+', push it onto stack => Stack: "+"
  - Read '3', since it's an operand add it to output string => Output: "2 3"
  - Read '*', push it onto stack => Stack: "+ *"
  - Read '4', since it's an operand add it to output string => Output: "2 3 4"
  - No more input left, pop remaining operators from stack and add them to output string => Output: "2 3 4 * +"

Hence, the infix expression "2 + 3 * 4" is converted to the postfix expression "2 3 4 * +".

- **To convert an infix expression to a prefix expression, you can use the following algorithm:**
  - Initialize two stacks: operator stack and output stack.
  - Read the infix expression from right to left.
  - If the current character is whitespace, ignore it and move to the next character.
  - If the current character is an operand (number or variable), push it onto the output stack.
  - If the current character is a closing parenthesis ')', push it onto the operator stack.
  - If the current character is an operator (+, -, *, /, etc.), do the following:
    - While the top of the operator stack is not an opening parenthesis '(' and has higher precedence or has equal precedence and is left-associative (except for '^'), pop the operators from the stack and push them onto the output stack.
    - Push the current operator onto the operator stack.
  - If the current character is an opening parenthesis '(', do the following:

- ❖ While the top of the operator stack is not a closing parenthesis ')', pop the operators from the stack and push them onto the output stack.
- ❖ Pop and discard the closing parenthesis from the operator stack.

- Repeat steps 2-7 until all characters in the infix expression are processed.
- While there are operators left on the operator stack, pop them and push them onto the output stack.
- Reverse the order of the characters in the output stack to obtain the prefix expression.
- Done.

➢ **Let's illustrate the algorithm with an example:**

Infix expression: (5 + 7) * 2

- **Step 1:**

  Operator Stack: empty

  Output Stack: empty

- **Step 2-4:**

  Operator Stack: empty

  Output Stack: 2 7 5

- **Step 5:**

  Operator Stack:)

  Output Stack: + 2 7 5

- **Step 6:**

  Operator Stack:) +

  Output Stack: * + 2 7 5

- **Step 7:**

Operator Stack: (

Output Stack: * + 2 7 5

- **Step 9:**

  Operator Stack: empty

  Output Stack: * + 2 7 5

- **Step 10:**

  Prefix expression: * + 2 7 5

So, the infix expression "(5 + 7) * 2" is converted to prefix expression "* + 2 7 5" using the algorithm.