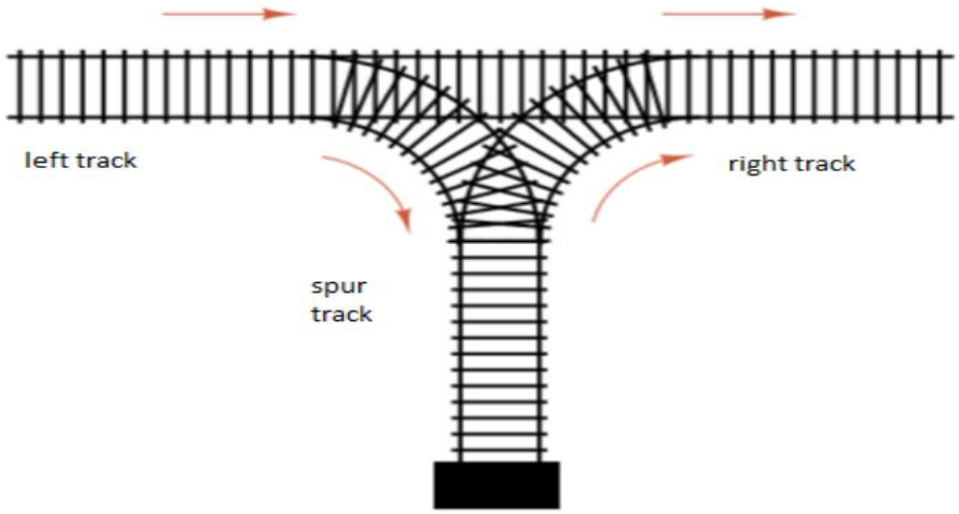
CO2:

1. Given a left, right, and a spur track as shown in the below figure. There are **N** trucks numbered 1, 2, ..., N are on the line arranged in the left track and it is desired to rearrange(permute) the trucks as they leave on the right-hand track. We can move directly N trucks to the right track but there can be more possibilities of moving the trucks to the right track using the spur track. We can move any truck to spur track and then move it to the right track. The task is to print all the possible permutation orders in which all the **N** trucks can be moved from left track to right track. Once a truck is moved from left track to right/spur track then it can't be moved to left track again. For example, if N = 3, and we have the trucks numbered 1,2,3 on the left track, then 3 first goes to the spur track. We could then send 2 to the spur, then on the way to its right, then send 3 on the way, then 1, obtaining the new order 1,3,2. Using suitable data structure write a 'C' program to print all possible permutation moves for a given N of trucks.



Solution:

```
#include <stdio.h>
#define MAX 10
typedef struct {
   int data[MAX];
   int front;
   int rear;
} Queue;
void enqueue(Queue *q, int val) {
```

```
    q->rear++;
    q->data[q->rear] = val;}

int dequeue(Queue *q) {
    return q->data[q->front++];
}

int isEmpty(Queue q) {
    return q.front > q.rear;
}
void permutation(int leftQueue[], int index, int size, Queue spur, int rightQueue[], int rightIndex) {
    if (index == size && isEmpty(spur)) {
        for (int i = 0; i < rightIndex; i++) {
            printf("%d ", rightQueue[i]);
        }
        printf("\n");
        return;
    }
   if (!isEmpty(spur)) {
        int temp = dequeue(&spur);
        rightQueue[rightIndex] = temp;
        permutation(leftQueue, index, size, spur, rightQueue, rightIndex + 1);
        enqueue(&spur, temp);
    }
    if (index < size) {
        rightQueue[rightIndex] = leftQueue[index];
        permutation(leftQueue, index + 1, size, spur, rightQueue, rightIndex + 1);
        enqueue(&spur, leftQueue[index]);
        permutation(leftQueue, index + 1, size, spur, rightQueue, rightIndex);
```

```
    dequeue(&spur);
  }}

int main() {
  int N = 3; // Number of trucks
  int leftQueue[MAX];
  Queue spur = { .front = 0, .rear = -1 };
  for (int i = 0; i < N; i++) {
    leftQueue[i] = i + 1;
  }
  int rightQueue[MAX];
  permutation(leftQueue, 0, N, spur, rightQueue, 0);
  return 0;
}
```

Write a C program to simulate a Car Parking System (CPS) with a maximum capacity of N parking slots, where no more than N-1 cars can be parked at a time. Each car is identified by its registration number and the owner's Aadhar ID. Vehicles enter the parking lot in order of their arrival and leave in the same order. If the parking lot is full when a new vehicle arrives, the system should notify that the parking lot is full. When all vehicles have exited, the system should indicate that the parking lot is empty. Cars can enter the parking lot if space becomes available due to the departure of other vehicles.

Write a C program to check if a string is palindrome or not using a double ended queue.

Solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>

#define MAX 100
```

```
// Deque structure
typedef struct {
   char data[MAX];
   int front, rear;
} Deque;

// Initialize deque
void initDeque(Deque *deque) {
   deque->front = -1;
   deque->rear = -1;
}

// Check if deque is empty
bool isEmpty(Deque *deque) {
   return deque->front == -1;
}

// Insert at rear
void insertRear(Deque *deque, char item) {
   if (deque->rear == MAX - 1) {
      printf("Deque overflow\n");
      return;
   }
   if (isEmpty(deque))
      deque->front = 0;
   deque->data[++deque->rear] = item;
}

// Remove from front
```

```
char deleteFront(Deque *deque) {
  if (isEmpty(deque)) {
    printf("Deque underflow\n");
    return '\0';
  }
  char item = deque->data[deque->front];
  if (deque->front == deque->rear)
    deque->front = deque->rear = -1;
  else
    deque->front++;
  return item;
}

// Remove from rear
char deleteRear(Deque *deque) {
  if (isEmpty(deque)) {
    printf("Deque underflow\n");
    return '\0';
  }
  char item = deque->data[deque->rear];
  if (deque->front == deque->rear)
    deque->front = deque->rear = -1;
  else
    deque->rear--;
  return item;
}

// Palindrome check function
bool isPalindrome(char *str) {
```

```
    Deque deque;
    initDeque(&deque);

    // Add each character to deque
    for (int i = 0; i < strlen(str); i++) {
        if (isalpha(str[i]))  // Only consider alphabet characters
            insertRear(&deque, tolower(str[i]));
    }

    // Check palindrome by comparing front and rear characters
    while (!isEmpty(&deque)) {
        char frontChar = deleteFront(&deque);
        if (isEmpty(&deque))  // Odd length case, middle character doesn't need a match
            break;
        char rearChar = deleteRear(&deque);
        if (frontChar != rearChar)
            return false;
    }
    return true;
}

// Main function
int main() {
    char str[MAX];
    printf("Enter a string: ");
    fgets(str, MAX, stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline character
    if (isPalindrome(str))
        printf("The string is a palindrome.\n");
```

```
    else
        printf("The string is not a palindrome.\n");
    return 0;
}
```

write a program to manage an emergency room capable of occupying 50 patients, using a priority queue to manage incoming patients. Patients with more severe levels (e.g., critical=4, serious=3, moderate=2, mild =1) conditions have higher priority and should be seen before others, even if they arrive later. Write a function that adds patients to the queue (with patient name and sever level of a patient) to a priority queue based on their severity when they arrive and retrieves them in the correct order for treatment.

Solution:

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

#define MAX_PATIENTS 100

// Enum to define severity levels
typedef enum {
    MILD = 1,
    MODERATE = 2,
    SERIOUS = 3,
    CRITICAL = 4
} SeverityLevel;

typedef struct {
    char name[50];
    SeverityLevel severity;
} Patient;
```

```
// Priority Queue structure
typedef struct {
   Patient patients[MAX_PATIENTS];
   int front;
   int rear;
} PriorityQueue;

// Initialize the priority queue
void initQueue(PriorityQueue *pq) {
   pq->front = -1;
   pq->rear = -1;
}

// Check if the queue is empty
int isEmpty(PriorityQueue *pq) {
   return pq->front == -1;
}

// Check if the queue is full
int isFull(PriorityQueue *pq) {
   return pq->rear == MAX_PATIENTS - 1;
}

// Function to add a new patient to the priority queue
void addPatient(PriorityQueue *pq, const char *name, SeverityLevel severity) {
   if (isFull(pq)) {
      printf("The emergency room is full!\n");
      return;
   }
```

```
    Patient newPatient;
    strcpy(newPatient.name, name);
    newPatient.severity = severity;

    // If the queue is empty, add the first patient
    if (isEmpty(pq)) {
        pq->front = pq->rear = 0;
        pq->patients[pq->rear] = newPatient;
    } else {
        // Find the correct position to insert the new patient based on severity
        int i;
        for (i = pq->rear; i >= pq->front && pq->patients[i].severity < severity; i--) {
            pq->patients[i + 1] = pq->patients[i];
        }
        pq->patients[i + 1] = newPatient;
        pq->rear++;
    }
    printf("Patient %s with severity %d added to the queue.\n", name, severity);
}

// Function to retrieve the highest-priority patient for treatment
void treatPatient(PriorityQueue *pq) {
    if (isEmpty(pq)) {
        printf("No patients in the emergency room.\n");
        return;
    }

    Patient patient = pq->patients[pq->front];
```

```
    printf("Treating patient %s with severity %d.\n", patient.name, patient.severity);

    // Shift the front pointer forward
    if (pq->front == pq->rear) {
        pq->front = pq->rear = -1;  // Queue becomes empty
    } else {
        pq->front++;
    }
}

// Main function to test the priority queue for ER
int main() {
    PriorityQueue pq;
    initQueue(&pq);

    addPatient(&pq, "KIM", CRITICAL);
    addPatient(&pq, "DIM", MODERATE);
    addPatient(&pq, "DUBBU", SERIOUS);
    addPatient(&pq, "SUBBU", MILD);
    addPatient(&pq, "GUBBI", CRITICAL);

    printf("\nTreating patients in order of severity:\n");
    while (!isEmpty(&pq)) {
        treatPatient(&pq);
    }

    return 0;
}
```

Trees Co4

Filesystems can be represented as binary trees where folders are internal nodes, and files are leaf nodes. Use 1 to represent the folder name along with the name of the folder and 0 to represent the file name respectively. Write a C program to simulate a simple filesystem hierarchy using a binary tree and display its contents using iterative post-order traversal.

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


// Define a node structure for the filesystem

typedef struct Node {

   char name[50];

   int isFolder;  // 1 if folder, 0 if file

   struct Node *left, *right;

} Node;


// Function to create a new node

Node* createNode(const char* name, int isFolder) {

   Node* newNode = (Node*)malloc(sizeof(Node));

   strcpy(newNode->name, name);

   newNode->isFolder = isFolder;

   newNode->left = newNode->right = NULL;

   return newNode;

}


// Stack structure for iterative traversal

typedef struct Stack {

   Node* data;

   int visited;  // flag to check if the node has been processed

   struct Stack* next;

} Stack;
```

```
// Push a node onto the stack
void push(Stack** top, Node* node, int visited) {
   Stack* newStackNode = (Stack*)malloc(sizeof(Stack));
   newStackNode->data = node;
   newStackNode->visited = visited;
   newStackNode->next = *top;
   *top = newStackNode;
}

// Pop a node from the stack
Node* pop(Stack** top, int* visited) {
   if (*top == NULL) return NULL;
   Stack* temp = *top;
   Node* node = temp->data;
   *visited = temp->visited;
   *top = (*top)->next;
   free(temp);
   return node;
}

// Check if the stack is empty
int isStackEmpty(Stack* top) {
   return top == NULL;
}

// Iterative Postorder Traversal of the filesystem tree
void iterativePostorder(Node* root) {
   Stack* stack = NULL;
   Node* current = root;
```

```
int visited;

printf("Filesystem contents (Postorder traversal):\n");

// Start traversal
do {
  // Go down the tree and push right and left children
  while (current != NULL) {
    push(&stack, current, 0);
    current = current->left;
  }

  // Retrieve the top node from the stack
  current = pop(&stack, &visited);

  // Process the node if it hasn't been visited
  if (visited == 0) {
    // Mark node as visited and push it back to stack
    push(&stack, current, 1);
    current = current->right;  // Move to the right subtree
  } else {
    // Visit the node
    if (current->isFolder) {
      printf("Folder: %s\n", current->name);
    } else {
      printf("File: %s\n", current->name);
    }
    current = NULL;
  }
```

```
    } while (!isStackEmpty(stack));
}

// Example filesystem hierarchy construction
Node* createFilesystem() {
    Node* root = createNode("root", 1);            // Root folder
    root->left = createNode("home", 1);            // Home folder
    root->right = createNode("etc", 1);            // Etc folder
    root->left->left = createNode("user1", 1);      // User1 folder in home
    root->left->right = createNode("user2", 1);     // User2 folder in home
    root->left->left->left = createNode("file1.txt", 0);  // File in user1 folder
    root->left->left->right = createNode("file2.txt", 0); // File in user1 folder
    root->left->right->left = createNode("file3.txt", 0); // File in user2 folder
    root->right->left = createNode("config", 0);    // File in etc folder
    root->right->right = createNode("settings", 0);   // File in etc folder
    return root;
}
// Main function
int main() {
    Node* filesystem = createFilesystem();
    iterativePostorder(filesystem);
    return 0;
}
```

Write a C program to construct an expression tree for the expression for the given postfix expression ABCD ^*+ and to evaluate it.

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

```
// Structure for a tree node
struct Node {
   char op;           // Operator or operand
   struct Node* left;   // Pointer to left child
   struct Node* right;  // Pointer to right child
};
// Function to create a new tree node
struct Node* newNode(char op) {
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
   node->op = op;
   node->left = NULL;
   node->right = NULL;
   return node;
}

// Function to evaluate the expression tree
double evaluate(struct Node* root) {
   // Base case: if the node is a leaf node (operand)
   if (!root->left && !root->right) {
      return (double)(root->op - 'A' + 1);  // Assuming A=1, B=2, C=3, D=4
   }

   // Recursively evaluate left and right subtrees
   double leftEval = evaluate(root->left);
   double rightEval = evaluate(root->right);

   // Apply the operator
   switch (root->op) {
      case '+': return leftEval + rightEval;
```

```
        case '*': return leftEval * rightEval;
        case '^': return pow(leftEval, rightEval);
        default: return 0; // Should not reach here
    }
}

// Function to construct the expression tree from postfix expression
struct Node* constructTree(char* postfix) {
    struct Node** stack = (struct Node**)malloc(sizeof(struct Node*) * strlen(postfix));
    int stackIndex = -1;

    for (int i = 0; postfix[i] != '\0'; i++) {
        char token = postfix[i];

        // If the token is an operand (A, B, C, D)
        if (isalnum(token)) {
            stack[++stackIndex] = newNode(token);
        } else {
            // The token is an operator
            struct Node* node = newNode(token);
            node->right = stack[stackIndex--]; // Right child
            node->left = stack[stackIndex--];  // Left child
            stack[++stackIndex] = node;        // Push new node onto stack
        }
    }

    // The last item on the stack is the root of the expression tree
    struct Node* root = stack[stackIndex];
    free(stack);
```

```
    return root;
}

// Function to free the expression tree
void freeTree(struct Node* root) {
    if (root) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}

// Example usage
int main() {
    char postfix[] = "ABCD^*+";

    // Construct the expression tree from the postfix expression
    struct Node* root = constructTree(postfix);

    // Evaluate the expression tree
    double result = evaluate(root);

    // Print the result
    printf("Result: %.2f\n", result);

    // Free the memory allocated for the tree
    freeTree(root);

    return 0;
```

}

2. Write a C program to implement a simple spell check suggestion system using a binary tree, where each node contains a common word. Use Inorder traversal to suggest words in alphabetical order when given a prefix.

Solution:

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

// Define a node structure for the binary tree
typedef struct Node {
  char word[50];
  struct Node *left, *right;
} Node;

// Function to create a new node
Node* createNode(const char* word) {
  Node* newNode = (Node*)malloc(sizeof(Node));
  strcpy(newNode->word, word);
  newNode->left = newNode->right = NULL;
  return newNode;
}

// Function to insert a word into the binary tree
Node* insert(Node* root, const char* word) {
  if (root == NULL) {
    return createNode(word);
  }

  if (strcmp(word, root->word) < 0) {
```

```
        root->left = insert(root->left, word);
    } else if (strcmp(word, root->word) > 0) {
        root->right = insert(root->right, word);
    }

    return root;
}

// Function to perform Inorder traversal and suggest words based on prefix
void suggestWords(Node* root, const char* prefix) {
    if (root == NULL) {
        return;
    }

    // Traverse the left subtree
    suggestWords(root->left, prefix);

    // Check if the current word starts with the given prefix
    if (strncmp(root->word, prefix, strlen(prefix)) == 0) {
        printf("%s\n", root->word);  // Suggest the word
    }

    // Traverse the right subtree
    suggestWords(root->right, prefix);
}

// Function to free the allocated memory for the tree
void freeTree(Node* root) {
    if (root != NULL) {
```

```
    freeTree(root->left);
    freeTree(root->right);
    free(root);
  }
}

// Main function
int main() {
  Node* root = NULL;
  char words[][50] = {
    "blackmulberry","apple", "banana", "grape", "orange", "strawberry", "kiwi", "blueberry", "blackberry"
  };
  int i,n = sizeof(words) / sizeof(words[0]);

  // Insert words into the binary tree
  for ( i = 0; i < n; i++) {
    root = insert(root, words[i]);
  }

  char prefix[50];
  printf("Enter a prefix to search for suggestions: ");
  scanf("%s", prefix);

  printf("Suggested words:\n");
  suggestWords(root, prefix);

  // Free the memory allocated for the tree
  freeTree(root);
  return 0;
```

```
}
```