INDEX

KUMAR 1BM22CS210

NAME: PRIYANSHO STD.: _____ SEC.: _____ ROLL NO.: _____ SUB.: DS LAB RECORD

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1.	27/12/23	1. Dynamic memory allocation, 2. Pointer swapping, 3. Stack implementation		
2.	28/12/23	1. Infix to postfix 2. Postfix evaluation 3. Queue		
3.	11/01/24	1. Circular Queue 2. Singly Linked List (Insert) 3. Leetcode		
4.	18/01/24	1. Singly Linked List (Delete), 2. Leetcode		
5.	25/01/24	1. Sorting, Reverse, Concat		
6.	01/02/24	1. Doubly Linked List 2. Leetcode		
7.	15/2/24	1. Binary search tree with pre-post, inorder traversal 2. Leetcode		
8.	22/02/24	1. DFS, 2. BFS, 3. HackerRank		
9.	22/02/24	Hashtable		

Q. WAP to swap numbers using pointers.

→
```
#include <stdio.h>
void swap(int*, int*);
void main() {
    int a, b;
    int* p, q;
    int temp;
    printf("Name: Priyanshu Kumar, USN: 1BM22CS210\n");
    printf("Enter values of a & b: \n");
    scanf("%d %d", &a, &b);
    printf("Before swapping: \n");
    printf("a = %d & b = %d \n", a, b);
    swap(&a, &b);
    printf("After swapping: \n");
    printf("a = %d & b = %d \n", a, b);
}
void swap(int* p, int* q) {
    int temp = *p;
    *p = *q;
    *q = temp;
}
```

Output:
```
Name: Priyanshu Kumar, USN: 1BM22CS210
Enter value of a & b:
15 20
Before swapping:
a = 15 & b = 20
After swapping:
a = 20 & b = 15
```

Q. WAP to demonstrate dynamic memory allocation (malloc, free, calloc, realloc).

→
```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *arr1;
    int size1 = 5;
    arr1 = (int*) malloc(size1 * sizeof(int));
    int i;
    if (arr1 == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }
    printf("Using malloc: \n");
    for(i = 0; i < size1; i++) {
        arr1[i] = i * 2;
        printf("%d", arr1[i]);
    }
    printf("\n");
    free(arr1);
```

```
int *arr2;
int size2 = 7;
arr2 = (int*) calloc(size2, sizeof(int
if (arr2 == NULL) {
    printf("Memory allocation failed\n
} return 1;

printf("\n Using calloc: \n");
for(i=0; i< size2; i++) {
    arr2[i] = i * 3;
} printf("%d", arr2[i]);
printf("\n");

int new_size = 10;
arr2 = (int*) realloc(arr2, new_size *
                      sizeof(int));

if (arr2 == NULL) {
    printf("Memory - reallocation
                failed \n");
} return1;

printf("\n Using realloc: \n")
for(i= size2; i< new_size; i++){
    arr2[i] = i * 4;
} printf("%d", arr2[i]);
printf("\n");
free(arr2);
return 0;
}
```

Output:

Using malloc:
0 2 4 6 8

Using calloc:
0 3 6 9 12 15 18

Using realloc:
28 32 36

Q. WAP to implement stack.

→

```
#include <stdio.h>

int stack[10];
int count = -1;

void push(int);
void pop();
void display();

void main() {
    for(;;){
        printf("\n1. Push value to stack\n");
        printf("2. Pop value from stack\n");
        printf("3. Display values\n");
        printf("4: Exit\n\n");

        int t;
        printf("Enter your choice: \n");
        scanf("%d", &t);
```

```
printf("\n");
if(t==1){
    int v;
    printf("Enter value to push:\n");
    scanf("%d", &v);
    push(v);
}
else if(t==2){
    pop();
}
else if(t==3){
    display();
}
else{
    exit(0);
}
}

void push(int e){
    if(count<10){
        stack[++count]=e;
    }
    else{
        printf("Stack is full!\n");
    }
}

void pop(){
    if(count!=-1){
        stack[count--]=0;
    }
    else{
        printf("stack is empty!\n");
    }
}

void display(){
    if(count==-1)
        return;
    int i;
    printf("The values are (top to bottom):\n");
    for(i=count; i>=0; i--){
        printf("%d\n", stack[i]);
    }
    printf("\n");
}
```

Output:

```
1. Push value to stack
2. Pop value from stack
3. Display values
4. Exit

Enter your choice:
1
Enter value to push:
23
:
```

Enter value to push: 50
50

Exp:7
21/12/23

28-12-23 WEEK-3

→
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_SIZE 100

int isOperator(char ch){
    return ch=='+' || ch=='-' || ch=='*'
    || ch=='/' || ch=='^';
}
int getPrecedence(char ch){
    if(ch=='+' || ch=='-') return 1;
    else if(ch=='*' || ch=='/') return 2;
    else if(ch=='^') return 3;
    else return 0;
}
void infixToPostfix(char infix[], char postfix[])
{ char stack[MAX_SIZE]; int top=-2;
    int i, j;
    for(i=0; j=0; infix[i]!='\0'; ++i){
        char ch=infix[i];
    if(isalnum(ch)){ postfix[j++]=ch;}
    else if(ch=='('){ stack[++top]=ch;}
    else if(ch==')'){
        while(top>=0 && stop[top]!='(')
{postfix[j++]=stack[top--];}
    if(top>=0 && stack[top]=='('){top--;}
    elseif(isOperator(ch)){
        while(top>=0 && getPrecedence(
        stack[top]>=getPrecedence(ch)){
            postfix[j++]=stack[top--];
        }
        stack[++top]=ch;
    }
}
while(top>=0){
    postfix[j++]=stack[top--];
}
postfix[j]='\0';
}

int main(){
    char infix[MAX_SIZE];
    char postfix[MAX_SIZE];

    printf("Enter the infix expression:");
    scanf("%s", infix);
    infixToPostfix(infix, postfix);

    printf("Postfix expression: %s\n",
                                postfix);
    return 0;
}
```

Output: Enter the infix expression: 2+(4+2)*3
Postfix Expression: 242+3*+

→ Evaluation of postfix

```
#include <stdio.h>
#include <string.h>
#define MAX_SIZE

void push(int item) {
    if (top >= MAX_SIZE - 1) {
        printf("Stack overflow \n");
    }
    top++;
    stack[top] = item;
}

int pop() {
    if (top < 0) {
        printf("Stack underflow \n");
    }
    int item = stack[top];
    top--;
    return item;
}

int isOperator(char symbol) {
    return (symbol == '+' || symbol == '-' ||
            symbol == '*' || symbol == '/');
}

int evaluate(char expression[]) {
    int i=0;
    char symbol = expression[i];
    int operand1, operand2, result;
    while (symbol != '\0') {
        if (symbol >= '0' && symbol <= '9') {
            int num = symbol - '0';
            push(num);
        } else if (isOperator(symbol)) {
            op2 = pop(); op1 = pop();
            switch (symbol) {
                case '+': result = op1 + op2; break;
                case '-': result = op1 - op2; break;
                case '*': result = op1 * op2; break;
                case '/': result = op1 / op2; break;
            }
            push(result);
        }
        symbol = expression[i];
    }
    result = pop();
    return result;
}

int main() {
    char expression[] = "5 6 7 + * 8 -";
    int result = evaluate(expression);
    printf("Result = %d \n", result);
    return 0;
}
```

Output:
Result = 57

→ Queue Implementation

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 5

void enqueue(int);
void dequeue();
void display();

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

int main(){
    dequeue();
    enqueue(1);
    enqueue(3);
    enqueue(4);
    enqueue(5);
    enqueue(6);
    display();
    dequeue();
    display();
    return 0;
}

void enqueue(int value){
    if(rear == SIZE-1)
        printf("\n Queue is Full!");
    else{
        if(front == -1)
            front=0;
        rear++;
        items[rear]=value;
        printf("\n Inserted -> %d", value);
    }
}

void dequeue(){
    if(front==-1)
        printf("\n Queue is empty!");
    else{
        printf("\n Deleted : %d",
                items[front]);
        front++;
        if(front>rear)
            front=rear=-1;
    }
}

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

Output:

```
Queue is empty
Queue elements are: 1 3 4 5 6
Deleted 1
Queue elements are 3 4 5 6
```

WEEK-7

11-1-24

Circular Queue

```
#include <stdio.h>

#define SIZE 5

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

int isFull(){
  if ((front == rear+1) || (front == 0 &&
      rear == SIZE-1))
    return 1;
  else
    return 0;
}

int isEmpty(){
  if (front == -1) return 1;
  else return 0;
}

void enqueue(int e){
  if (isFull())
    printf("\n Que is full !! \n");
  else {
    if (front == -1) front = 0;
    rear = (rear+1) % SIZE;
    items[rear] = element;
    printf("\n Inserted %d", e);
  }
}

int dequeue(){
  int e;
  if(isEmpty()){
    printf("\n Queue is empty !! \n");
    return(-1);
  }
  else {
    e = items[front];
    if(front == rear){
      front = -1;
      rear = -1;
    }
    else {
      front = (front+1) % SIZE
    };
    printf("\n Deleted element %d", e);
    return e;
  }
}

void display(){
  int i;
  if (isEmpty())
    printf("\n Empty Queue \n");
  else{
    printf("\n Front: %d", front);
    printf("\n Items:");
    for(i = front; i != rear; i = (i+2)%SIZE) {
      printf("%d", items[i]);
    }
    printf("%d", items[i]);
    printf("\n Rear: %d \n", rear);
  }
}
```

```
int main(){
    deQueue();

    enqueue(5);

    enqueue(7);

    enqueue(10);

    display();

    dequeue();

    display();
}
```

Output:

Queue is empty!!

Inserted 5

Inserted 7

Inserted 10

Front → 0

Linked list

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

struct Node {
    int data;
    int* next;
};

void insertAtBeginning(struct Node** head_ref,
                       int new_data)
{
    struct Node* new_Node = (struct Node*)
        malloc(sizeof(struct Node));

    new_Node -> data = new_data;

    new_Node -> next = (*head_ref);

    (*head_ref) = new_Node;
}

void insertAfter(struct Node* prev_node,
                 int new_data){
    if(prev_node == NULL){
        printf("the given previous node
            cannot be NULL");
    return;
}

struct Node* new_node = (struct Node*)
    malloc(sizeof(struct Node));

new_node -> data = new_data;
prev_node -> next;
```

```
    prev_node -> next = new_node;
}

void insertAtEnd(struct Node** head_ref,
                 int new_data) {

    struct Node* new_node = (struct Node*)
        malloc(sizeof(struct Node));
    struct Node* last = *head_ref;

    new_node -> data = new_data;
    new_node -> next = NULL;

    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }

    while (last -> next != NULL) last
            = last -> next;

    last -> next = new_node;
    return;
}

void printList(struct Node* node) {
    while (node != NULL) {
        printf("%d", node -> data);
        node = node -> next;
    }
}
```

```
int main() {
    struct Node* head = NULL;

    insertAtEnd(&head, 1);
    insertAtBeginning(&head, 2);
    insertAtBeginning(&head, 3);
    insertAtEnd(&head, 4);
    insertAfter(head -> next, 5);

    print("Linked list:");
    printList(head);

}
```

Output:

Linked list: 3 2 5 1 4

18-1-24

WEEK-3

→ Deletion of element

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

struct Node{
    int data;
    struct Node* next;
};

void insertAtBeginning(struct Node** head_ref, int new_data) {

    struct Node* new_node = (struct Node*)
        malloc(sizeof(struct Node));

    new_node->data = new_data;

    new_node->next = (*head_ref);

    (*head_ref) = new_node;
}

void insertAfter(struct Node* prev_node,
        int new_data) {
    if (prev_node == NULL) {
        printf("The previous node cannot
            be null");
        return;
    }
    struct Node* new_node = (struct Node*)
        malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = prev_node->next;
    prev_node->next = new_node;
}

void insertAtEnd(struct Node** head_ref
        , int new_data) {

    struct Node* new_node = (struct Node*)
        malloc(sizeof(struct Node));

    struct Node* last = *head_ref;
    new_node->data = new_data;
    new_node->next = NULL;

    if(*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }

    while(last->next != NULL)
        last = last->next;

    last->next = new_node;
    return;
}

void deleteNode(struct Node** head_ref,
        int key) {
    struct Node *temp = *head_ref, *prev;

    if (temp != NULL && temp->data == key)
    {
        *head_ref = temp->next;
        free(temp);
        return;
```

```
while (temp != NULL && temp -> data
                        != key) {
    prev = temp;
    temp = temp -> next;
}

if (temp == NULL) return;

prev -> next = temp -> next;

free (temp);
```

Executed 18/11/24

```
void displayList (struct Node * node) {
    while (node != NULL) {
        printf ("%d", node -> data);
        node = node -> next;
    }
}

int main () {
    struct Node * head = NULL;

    insertAtEnd (&head, 1);
    insertAtBeginning (&head, 2);
    insertAtBeginning (&head, 3);
    insertAtEnd (&head, 4);
    insertAfter (head -> next, 5);

    printf ("Linked list: ");
    displayList (head);
    printf ("\n After deleting an
                element: ");
    deleteNode (&head, 3);

    displayList (head);
}
```

Output:
Linked list: 3 2 5 14
After deleting an element: 2 5 14

Leetcode

→ the leetcode prm min stack

```
#define MAX_SIZE 1000

typedef struct {
    int *stack;
    int *min_stack;
    int top;
} MinStack;

MinStack * minStackCreate () {

    MinStack * obj = (MinStack *) malloc
                (sizeof (MinStack));

    obj -> stack = (int *) malloc (MAX_SIZE
                * sizeof (int));

    obj -> min_stack = (int *) malloc
                (MAX_SIZE * sizeof (int));
    obj -> top = -1;
    return obj;
}
```

Executed 18/11/24

```
void minStackPush(MinStack * obj, int val){
    if (obj -> top == -1) {
        obj -> top ++;
        obj -> stack [obj -> top] = val;
        obj -> min_stack [obj -> top] = val;
    } else if(obj -> top < MAX_SIZE - 1){
        obj -> top ++;
        obj -> stack [obj -> top] = val;
        if(obj -> min_stack[obj -> top - 1] < val)
            obj -> min_stack [obj -> top] =
            obj -> min_stack [obj -> top - 1];
        else
            obj -> min_stack [obj -> top] = val;
    }
}

void minStackPop(MinStack * obj) {
    if(obj -> top == -1) return;
        obj -> top --;
}
```

```
int minStackTop (MinStack * obj) {
        return obj -> stack [obj -> top];
}

int minStackGetMin (MinStack * obj) {
    return obj -> min_stack [obj -> top];
}

void minStackFree (MinStack * obj) {
    free (obj -> stack);
    free (obj -> min_stack);
    free (obj);
}
```

Reverse Linked List

```
struct ListNode* createNode(int val){
    struct ListNode* newNode
    =(struct ListNode*) malloc(sizeof
        (struct ListNode));
    newNode->val = val;
    newNode->next = NULL;
    return newNode;
}

struct ListNode* reverseBetween(struct
    ListNode* head, int left, int right){
if(head ==NULL || left == right){
    return head;
}
struct ListNode dummy;
dummy.next = head;
struct ListNode* prev = &dummy;
for(int i=1; i<left; ++i){
    prev = prev->next;
}
struct ListNode* current = prev->next;
struct ListNode* next = NULL;
struct ListNode* tail = current;
for(int i=left; i<=right; i++){
    struct ListNode* temp = current->next;
    current->next = next;
    next = current;
    current = temp;
}
prev->next = next;
tail->next = current;
return dummy.next;
}
```

25-01

→

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

struct Node {
    int data;
    struct Node* next;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)
        malloc(sizeof(structNode));

    newNode->data = data;

    newNode->next = NULL;

    return newNode;
}

void displayList(struct Node* head){
    struct Node* temp = head;

    while(temp != NULL) {
        printf("%d -> ", temp->data);

        temp = temp->next;
    }
    printf("NULL\n");
}

void sortList(struct Node* head) {
    struct Node *prev, *current, *nextNode;
    prev = NULL;
    current = head;
    while(current != NULL) {
        nextNode = current->next;
        current->next = prev;
        prev = current;
        current = nextNode;
    }

    return prev;
}

void concatenateLists(struct Node* list1,
        struct Node* list2) {
    while(list->next != NULL) {

        list1 = list1->next;
    }
    list1->next = list1;
}

int main() {
    struct Node* head1 = createNode(3);
    head1->next = createNode(1);
    head1->next->next = createNode(5);

    printf("Original Linked List 1: ");
    displayList(head1);

    sortList(head1);
    printf("Sorted Linked List 1: ");
    displayList(head1);

    head1 = reverseList(head1);
    printf("Reversed Linked List 1: ");
    displayList(head1);
```

```
    struct Node* head2 = createNode(2);
    head2->next = createNode(4);
    head2->next->next = createNode(6);

    printf("Original Linked List 2:");
    displayList(head2);

    concatenateLists(head1, head2);
    printf("Concatenated Linked List:");
    displayList(head1);

    return 0;
}
```

Output:

Original Linked List 1: 3 → 1 → 5 → NULL
Sorted Linked List: 1 → 3 → 5 → NULL
Reversed Linked List: 5 → 3 → 1 → NULL
Original Linked list: 2 → 4 → 6 → NULL
Concatenated List: 5 → 3 → 1 → 2 → 4 → 6 → NULL

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

struct Node {
    int data;
    struct Node* next;
};

struct LinkedList {
    struct Node* head;
};

struct Node* createNode(int data){
    struct Node* newNode = (struct Node*)
        malloc(sizeof(struct Node));

    newNode->data = data;

    newNode->next = NULL;
    return newNode;
}

struct LinkedList* initializeList(){
    struct LinkedList* newList
    = (struct LinkedList*) malloc(sizeof(
        struct LinkedList));
    newList->head = NULL;
    return newList;
}
```

```
void push(struct LinkedList *, int);
int pop(struct LinkedList *);
void enqueue(struct LinkedList *, int);
int dequeue(struct LinkedList *);
void display(struct LinkedList *);
void freeList(struct LinkedList *);

void push(struct LinkedList * list,
          int data) {
    struct Node * newNode = createNode
                            (data);
    newNode -> next = list -> head;
    list -> head = newNode;
}

int pop(struct LinkedList * list) {
    if (list -> head == NULL) {
        printf("Stack/Queue is empty\n");
        return -1;
    }
    struct Node* temp = list -> head;
    int poppedData = temp -> data;
    list -> head = temp -> next;
    free(temp);
    return poppedData;
}

void enqueue(struct LinkedList* list,
             int data) {
    struct Node * newNode = createNode(data);
    if (list -> head == NULL) {
        list -> head = newNode;
        return;
    }
    struct Node* lastNode = list -> head;
    while (lastNode -> next != NULL) {
        lastNode = lastNode -> next;
    }
    lastNode -> next = newNode;
}

int dequeue(struct LinkedList* list) {
    return pop(list);
}

void display(struct LinkedList* list) {
    struct Node * current = list -> head;
    while (current != NULL) {
        printf("%d ", current -> data);
        current = current -> next;
    }
    printf("\n");
}

void freeList(struct LinkedList * list) {
    struct Node* current = list -> head;
    struct Node* nextNode;
    while (current != NULL) {
        nextNode = current -> next;
        free(current);
        current = nextNode;
    }
    free(list);
}

int main() {
    struct LinkedList * linkedlist =
```

```
push(linkedList, 1);
push(linkedList, 2);
push(linkedList, 3);
printf("Stack:");
display(linkedList);
printf("Popped from stack:
        %d\n", pop(linkedList));
display(linkedList);

enqueue(linkedList, 4);
enqueue(linkedList, 5);
enqueue(linkedList, 6);
display(linkedList);
printf("Dequeued from queue:
        %d\n", dequeue(linkedList
                        ));

freeList(linkedList);

return 0;
}
```

Output:
Stack: 3 2 1
Popped from stack: 3
2 1
Queue: 2 1 4 5 6
Dequeued from queue: 2
1 4 5 6

01-02

Doubly linked list

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

struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)
        malloc(sizeof(struct Node));

    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
    } else {
        newNode->data = value;
        newNode->prev = NULL;
        newNode->next = NULL;
        return newNode;
    }
}

void insertToLeft(struct Node** head,
    struct Node* target, int value){
    struct Node* newNode = createNode(value);

    if(target->prev != NULL){
        target->prev->next = newNode;
        newNode->prev = target->prev;
    }else{
        *head = newNode;
    }
    newNode->next = target;
```

```
    target->prev = newNode;
}

void deleteNodeByValue(struct Node** head,
    int value) {

    struct Node* current = *head;

    while (current != NULL) {
        if (current->data == value) {
            current->prev->next = current->
                                  next;
        } else {
            *head = current->next;
        }

        if (current->next != NULL) {
            current->next->prev = current->prev;
        }

        free(current);
        return;
    }
    current = current->next;
}
    printf("Node with value %d not
        found in the list.\n", value);
}

void displayList(struct Node* head) {
    printf("Doubly Linked List: ");
    while (head != NULL) {
        printf("%d", head->data);
        head = head->next;
    }
    printf("\n");
}

int main() {
    struct Node* head = NULL;
    head = createNode(1);
    struct Node* second = createNode(2);
    struct Node* third = createNode(8);

    head->next = second;
    second->prev = head;
    second->next = third;
    third->prev = second;

    displayList(head);

    insertToLeft(&head, second, 5);
    displayList(head);

    deleteNodeByValue(&head, 2);
    displayList(head);

    return 0;
}
```

```
Output:
Doubly Linked List: 123
Doubly Linked List: 1523
Doubly Linked List: 153
```

→ Split Linked List in Parts

```
struct ListNode** splitListToParts(
    struct ListNode* head, int k,
    int* returnSize) {
```

```
int length = getLength(head);
int width = length/k;
int remainder = length % k;
struct ListNode** result =
    (struct ListNode**) malloc(k *
        sizeof(struct ListNode**));
struct ListNode* current = head;
for(int i = 0; i < k; i++) {
    result[i] = current;
    int partSize = width + (i < remainder
                    ? 1 : 0);
    for(int j=0; j< remainder partSize-1;
                j++) {
        if (current != NULL) {
            current = current -> next;
        }
    }
    if(current != NULL) {
        struct ListNode* temp = current;
        current = current -> next;
        temp -> next = NULL;
    }
}
*returnSize = k;
return result;
}
```

15/2/24

```
int getLength(struct ListNode* head){
    int length =0;
    while(head != NULL) {
        length++;
        head = head -> next;
    }
    return length;
}
```

15-02

→ // Binary Search Tree

```
struct TreeNode {
    int data;
    struct TreeNode *left;
    struct TreeNode *right;
};

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

struct TreeNode* insertNode (struct
    TreeNode* root, int data) {

    if (root == NULL) {
        root = createNode (data);
    } else if (data <= root -> data) {
        root -> left = insertNode (root -> left,
                                   data);
    } else {
        root -> right = insertNode (root -> right,
                                    data);
    }
    return root;
}
```

```
void inorderTraversal (struct TreeNode*
                       root) {
    if (root != NULL) {
        inorderTraversal (root -> left);
        printf ("%d ", root -> data);
        inorderTraversal (root -> right);
    }
}

void preorderTraversal (struct TreeNode*
                        root) {
    if (root != NULL) {
        postorderTraversal (root -> left);
        postorderTraversal (root -> right);
        printf ("%d", root -> data);
    }
}

void postorderTraversal (struct TreeNode*
                         root) {
    if (root != NULL) {
        postorderTraversal (root -> left);
        postorderTraversal (root -> right);
        printf ("%d", root -> data);
    }
}

void display (struct TreeNode* root) {
    printf ("Elements in the tree: ");
    inorderTraversal (root);
    printf ("\n");
}
```

15/24

```
int main() {
    struct TreeNode* root = NULL;

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

    display(root);

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

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

    printf("Postorder traversal: ");
    postorderTraversal(root);
    printf("\n");

    return 0;
}
```

Output:

Elements in the tree: 20 30 40 50 60 70 80

Inorder traversal: 20 30 40 50 60 70 80

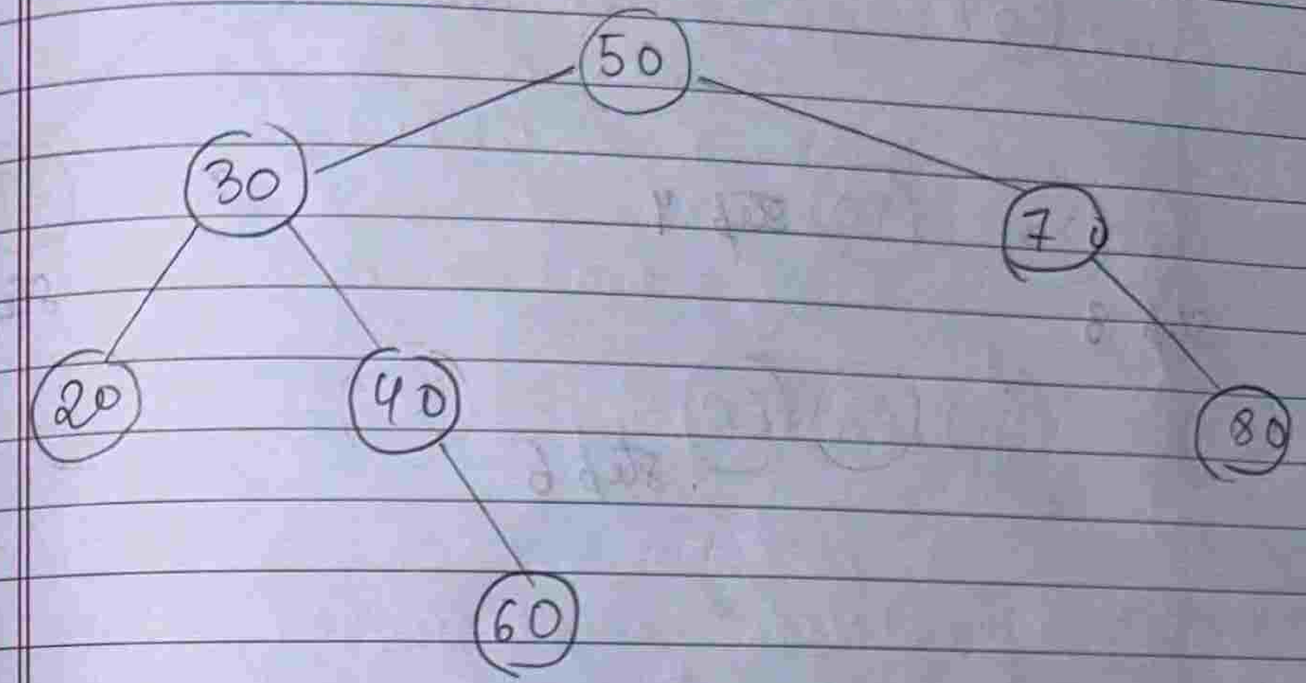
Preorder traversal: 50 30 20 40 70 60 80

Postorder traversal: 20 40 30 60 80 70 50

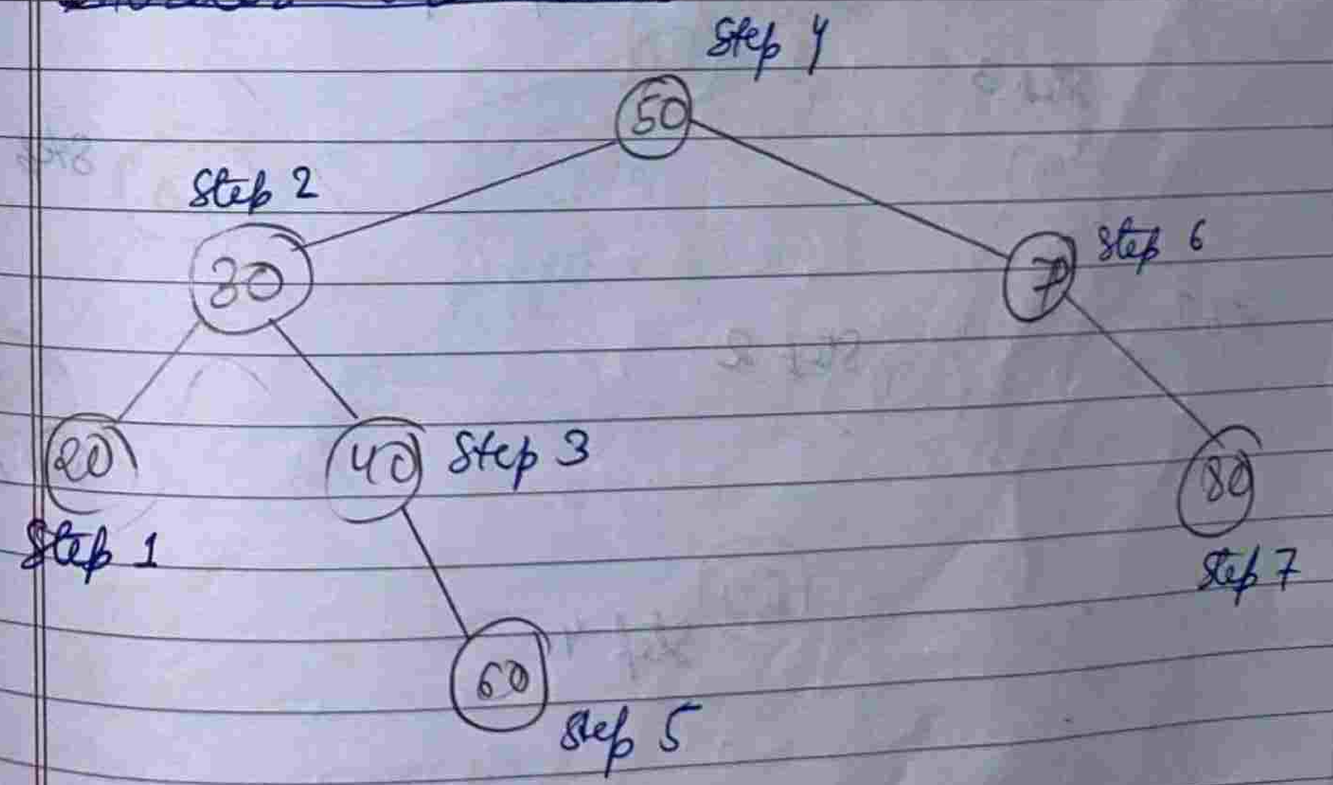
Leetcode:

Spacing.

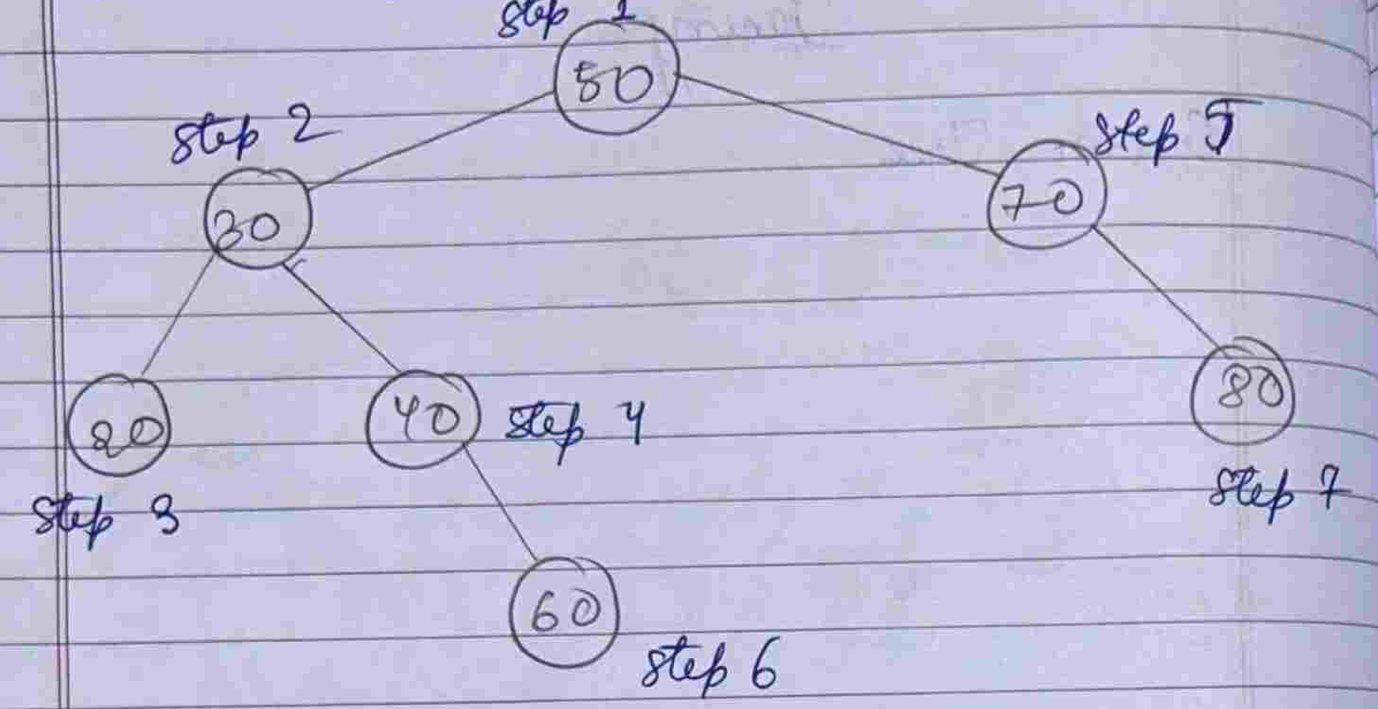
Given Tree:



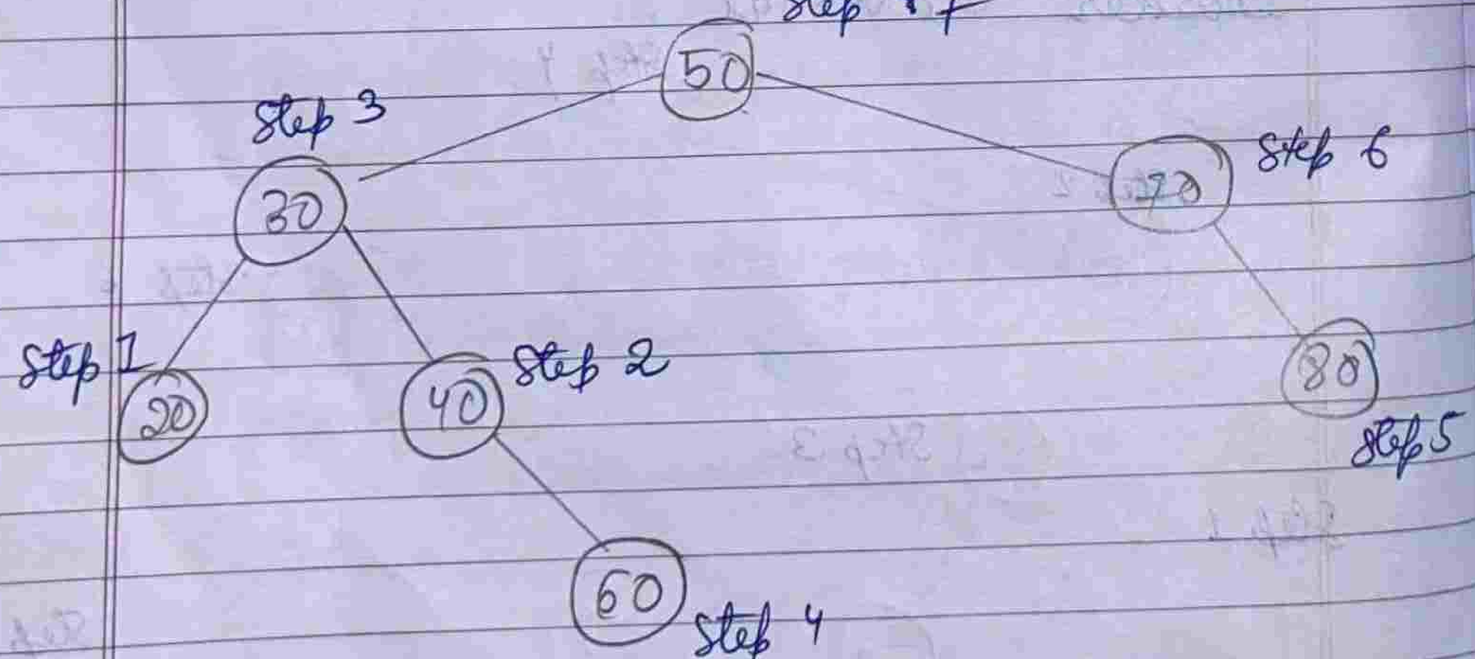
Inorder Traversal:



Pre-order Traversal



Post-order Traversal



→ ~~LeetCode~~

→ // LeetCode: Rotate List

```
int getLength(struct ListNode* head){
    int length = 0;
    while (head != NULL) {
        length++;
        head = head -> next;
    }
    return length;
}

struct ListNode* rotateRight(struct
        ListNode* head, int k) {

    if (head == NULL || head->next == NULL
        || k == 0) {
        return head;
    }

    int length = getLength(head);
    k = k % length;
    if(k == 0) {
        return head;
    }

    struct ListNode* tail = head;
    for(int i = 0; i < length - k - 1; i++){
        tail = tail -> next;
    }

    struct ListNode* newHead = tail->next;
    tail -> next = NULL;
    struct ListNode* current = newHead;
    while (current -> next != NULL) {
        current = current -> next;
    }
    current -> next = head;

    return newHead;
}
```

22 → //BFS

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

#define MAX_NODES 100

struct Node {
    int data;
    struct Node* next;
};

struct Graph {
    int num_vertices;
    struct Node* next;
};

struct Node* createNode(int data){
    struct Node* newNode = (struct Node*)
        malloc(sizeof(struct Node));

    newNode -> data = data;

    newNode -> next = NULL;

    return newNode;
}

struct Graph* createGraph(int num_vertices)
{
    struct Graph* graph = (struct Graph*)
        malloc(sizeof(struct Graph));

    graph -> num_vertices = num_vertices;

    for(int i=0; i< num_vertices; ++i){
```

```
void addEdge(struct Graph* graph, int src,
             int dest)
{
    struct Node* newNode = createNode(dest);
    newNode -> next = graph -> adjacency_list
                              [src];
    graph -> adjacency_list[src] = newNode;

    newNode = createNode(src);
    newNode -> next = graph -> adjacency_list
                              [dest];
    graph -> adjacency_list[dest] = newNode;
}

void BFS(struct Graph* graph, int start){
    int visited[MAX_NODES] = {0};
    int queue[MAX_NODES];
    int front = -1, rear = -1;

    queue[++rear] = start;
    visited[start] = 1;

    while(front < rear){
        int current = queue[++front];
        printf("%d", current);

        struct Node* temp = graph -> adjacency
                            _list[current];

        while(temp){
            int adj_vertex = temp -> data;
            if(!visited[adj_vertex]){
                queue[++rear] = adj_vertex;
                visited[adj_vertex] = 1;
            }
            temp = temp -> next;
```

```
int main() {
    struct Graph *graph = createGraph(4);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 0);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 3);

    printf("BFS traversal starting
            from vertex 2:\n");
    BFS(graph, 2);

    return 0;
}
//DFS
#include <stdio.h>
#include <stdlib.h>

#define MAX_NODES 100

struct Graph {

struct Node {
    int data;
    struct Node * next;
};

struct Graph {
    int num_vertices;
    struct Node * adjacency_list[MAX_NODES];
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)
        malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int num_vertices)
{
    struct Graph* graph = (struct Graph*)
        malloc(sizeof(struct Graph));
    graph->num_vertices = num_vertices;

    for(int i=0; i< num_vertices; i++)
        graph->adjacency_list[i] = NULL;

    return graph;
}

void addEdge(struct Graph* graph,
    int src, int dest) {

    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjacency
                    _list[dest];
    graph->adjacency_list[dest] = newNode;
}

void DFS(struct Graph* graph, int
    vertex, int visited[]) {
    visited[vertex] = 1;
```

```
Output:
Elements in the tree: 20 30 40 50 60 70 80
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
```

```
struct Node* temp = graph->
                adjacency_list[vertex];
while(temp){
  int adj_vertex = temp->data;
  if(!visited[adj_vertex])
      DFS(graph, adj_vertex, visited);
  temp = temp->next;
}
}

int isConnected(struct Graph* graph){
  int visited[MAX_NODES] = {0};
  DFS(graph, 0, visited);

  for(int i=0; i < graph->num_
                vertices; ++i){
    if(!visited[i])
        return 0;
  }
  return 1;
}

int main(){
  struct Graph* graph = createGraph(5);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 2);
  addEdge(graph, 3, 4);

  if(isConnected(graph))
      printf("The graph is connected.\n");
  else
      printf("The graph is not
                connected.\n");
  return 0;
```

```
// Leetcode:

// HackerRank: Swap Nodes

typedef struct Node{
    int data;
    struct Node* left;
    struct Node* right;
}Node;

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

void swapSubtrees(Node* root, int k,
                int depth){
  if(root == NULL)
      return;

  if(depth % k == 0){
    Node* temp = root->left;
    root->left = root->right;
    root->right = temp;
  }

  swapSubtrees(root->left, k, depth+1);
  swapSubtrees(root->right, k, depth+1);
}
```

```
void inorderTraversal(Node* root, int** result, int* index) {
    if (root == NULL) {
        return;
    }
    inorderTraversal(root->left, result, index);
    (*result)[(*index)++] = root->data;
    inorderTraversal(root->right, result, index);
}

int** swapNodes(int indexes_rows, int indexes_columns, int** indexes, int queries_count, int* queries, int* result_rows, int* result_columns) {
    Node* root = createNode(1);
    Node* queue[indexes_rows];
    int front = -1, rear = -1;
    queue[++rear] = root;

    for (int i = 0; i < indexes_rows; i++) {
        Node* curr = queue[++front];
        int leftData = indexes[i][0];
        int rightData = indexes[i][1];

        if (leftData != -1) {
            curr->left = createNode(leftData);
            queue[++rear] = curr->left;
        }

        if (rightData != -1) {
            curr->right = createNode(rightData);
            queue[++rear] = curr->right;
        }
    }

    int** resultArray = (int**)malloc(queries_count * sizeof(int*));
    *result_rows = queries_count;
    *result_columns = indexes_rows;

    int resultIndex = 0;
    for (int i = 0; i < queries_count; i++) {
        int k = queries[i];
        swapSubtrees(root, k, 1);

        int* result = (int*)malloc(indexes_rows * sizeof(int));
        int index = 0;
        inorderTraversal(root, &result, &index);
        resultArray[resultIndex++] = result;
    }

    return resultArray;
}
```

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

#define MAX_EMPLOYEES 100
#define HT_SIZE 10

typedef struct {
    int key;
    char name[50];
} EmployeeRecord;

typedef struct {
    int key;
    int address;
} HashTableEntry;

HashTableEntry hashTable[HT_SIZE];

int hashFunction(int key) {
    return key % HT_SIZE;
}

void initializeHashTable() {
    for(int i = 0; i < HT_SIZE; i++) {
        hashTable[i].key = -1;
        hashTable[i].address = -1;
    }
}

void insertRecord(EmployeeRecord record) {
    int hashIndex = hashFunction(record.key);
    while(hashTable[hashIndex].key != -1) {
        hashIndex = (hashIndex + 1) % HT_SIZE;
    }

    hashTable[hashIndex].key = record.key;
    hashTable[hashIndex].address =
                    hashIndex;
}

int searchRecord(int key) {
    int hashIndex = hashFunction(key);
    while (hashTable[hashIndex] != key &&
        hashTable[hashIndex].key != -1) {
        hashIndex = (hashIndex + 1)
                % HT_SIZE;
    }
    if (hashTable[hashIndex].key == key) {
        return hashIndex;
    }
    else {
        return -1;
    }
}

void displayRecord(int index) {
    if (index != -1) {
        printf("Employee Record Found:\n");
        printf("Key: %d\n", hashTable[index]
                .key);
        printf("Address %d\n", hashTable[index]
                .address);
    }
```

```
    else {
        printf("Employee Record Not
                Found.\n");
    }
}

int main() {
    int choice, key;
    EmployeeRecord records[MAX_EMPLOYEES];
    int numRecords = 0;

    initializeHashTable();

    do {
        printf("\n Employee Record
                Management System\n");
        printf("1. Insert Record\n");
        printf("2. Search Record\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                if (numRecords < MAX_RECORDS) {
                    printf("Enter employee key: ");
                    scanf("%d", &records
                            [numRecords].key);
                    printf("Enter employe name: ");
                    scanf("%s", records[numRecords]
                            .name);
                    insertRecord(records[numRecords]);
                    numRecords++;
                    printf("Record inserted successfully.\n");
                } else {
                    printf("Maximum number of
                            records reached.\n");
                }
                break;
            case 2:
                printf("Enter employee key to
                        search: ");
                scanf("%d", &key);
                int index = searchRecord(key);
                displayRecord(index);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try
                        again.\n");
        } while(choice != 3);

        return 0;
    }
```

Output:

```
Employee Record Management System
1. Insert Record 2. Search Record 3. Exit
Enter your choice: 1
Enter employee key: 4444
Enter employee name: aman
Record entered successfully.
```