**01 - Arrays and Abstract Data Type\V06-aadt.c**

```c
//Implementing Array as an Abstract Data Type
#include<stdio.h>
#include<stdlib.h>
// Structure to represent a dynamic array
struct myArray {
    int total_size; // Total size of the array
    int used_size;  // Number of elements used in the array
    int *ptr;       // Pointer to the dynamically allocated array
};
// Function to create an array with specified total size and used size
void createArray(struct myArray *a, int tsize, int usize){
    (*a).total_size = tsize;
    (*a).used_size  = usize;
    (*a).ptr = (int*)malloc(tsize * sizeof(int)); // Allocate memory for the array
    /*
    //same thing diff syntax.....
    a->total_size = tsize;
    a->used_size  = usize;
    a->ptr = (int*)malloc(tsize  * sizeof(int));
    */
}
// Function to display the elements of the array
void show(struct myArray *a){
    for (int i = 0; i < (*a).used_size; i++) {
        printf("%d,", (*a).ptr[i]);
    }
}
// Function to set values for each element of the array
void setVal(struct myArray *a){
    int n;
    for (int i = 0; i < (*a).used_size; i++) {
        printf("Enter element %d: ", i + 1);
        scanf("%d", &n);
        (*a).ptr[i] = n;
    }
}
int main() {
    struct myArray marks;
    createArray(&marks,10,5);
    printf("we are runnin setval now\n");
    setVal(&marks);
    printf("we re running show now\n");
    show(&marks);
    free(marks.ptr);    // Free the dynamically allocated memory before exiting
    return 0;
}
```

**02 - Operations on Arrays in Data Structures\V10-ins.c**

```c
//Insertion Operation in Array
#include <stdio.h>
#include <stdlib.h>
// Function to display the elements of an array
void display(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d,", arr[i]);
    }
    printf("\n");
}
// Function to insert an element at a specified index in the array
int index_insertion(int arr[], int size, int element_to_ins, int at_index, int capacity) {
    // Check if the array is already at full capacity
    if (size >= capacity) {
        return -1;
    }
    // Shift elements to the right to create space for the new element
    for (int i = size - 1; i >= at_index; i--) {
        arr[i + 1] = arr[i];
    }
    // Insert the new element at the specified index
    arr[at_index] = element_to_ins;
    return 1;
}
int main() {
    // Initialize an array with some values
    int arr[100] = {1, 2, 3, 4, 5, 6, 8, 9, 10};
    // Initial size and capacity of the array
    int size = 9, capacity = 100;
    // Element to insert and the index at which to insert
    int element_to_ins = 7, at_index = 6;
    printf("Array before insertion: ");
    display(arr, size);
    index_insertion(arr, size, element_to_ins, at_index, capacity);
    size++;
    printf("Array after  insertion: ");
    display(arr, size);
    return 0;
}
```

**02 - Operations on Arrays in Data Structures\V11-del.c**

```c
//-------------------------------------------------------------------------------------------------------------------
//Deletion Operation in Array

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

// Function to display the elements of an array
void display(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d,", arr[i]);
    }
    printf("\n");
}

int index_deletion(int arr[], int size, int index_to_del) {

    for (int i = index_to_del; i <  size -1 ; i++)
    {
        arr[i] = arr[i+1];
    }

    return 1;
}

int main() {
    // Initialize an array with some values
    int arr[100] = {1, 2, 3, 4, 5};
    // Initial size and capacity of the array
    int size = 5, capacity = 100;
    // Element to insert and the index at which to insert
    int index_to_del = 2;

    printf("Array before deletion: ");
    display(arr, size);

    index_deletion(arr, size, index_to_del);
    size--;

    printf("Array after  deletion: ");
    display(arr, size);

    return 0;

}

//-------------------------------------------------------------------------------------------------------------------
```

**03 - LinearSearch & BinarySearch\V12-BS.c**

```c
//-------------------------------------------------------------------------------------------------------
//Binary search

#include<stdio.h>

int binarySearch(int arr[],int low,int high, int key){
    if(low<=high){
        int mid = low + (high-low)/2;
        if (arr[mid]== key)
        {
            return mid;
        }
        if (arr[mid]>key)
        {
            return binarySearch(arr,low,mid -1,key);
        }
        if (arr[mid]<key)
        {
            return binarySearch(arr,mid+1,high,key);
        }
    }
    return -1;
}

int main() {
    int arr[] = {2,4,6,8,12,34,56,78,90,99};
    int size = sizeof(arr)/sizeof(arr[0]);
    // printf("%d",size);
    int key = 8;
    int indx = binarySearch(arr,0,size-1,key);
    if (indx==-1)
    {
        printf("not found");
    }
    else{
        printf("indx = %d",indx);
    }
    return 0;
}
```

**03 - LinearSearch & BinarySearch\V12-LS.c**

```c
//-----------------------------------------------------------------------------------------------------------------
//Linear search

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

int LinearSearch(int arr[],int key, int size){

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

int main() {
    int arr[] = {5,6,7,8,23,52,536,35635,6356,546,4565,4654,6,45654,654,654,645,6546,45,645,23,343,6,99,45,87,674,63442,11,25,345,463,436};
    int size = sizeof(arr)/sizeof(arr[0]);
    int key = 99;
    int indx = LinearSearch(arr,key,size);
    printf("%d is present in the index %d",key,indx);
    return 0;
}
```

**04 - Linked List\V14-LL.c**

```c
// ------------------------------------------------------------------------
// linked list

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

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

void llTraversal(struct node *temp)
{
    while (temp != NULL)
    {
        printf("elems: %d\n", temp->data);
        temp = temp->next;
    }
}

int main()
{

    struct node *head;
    struct node *second;
    struct node *third;

    // allocate memmory for nodes in heap
    head = (struct node *)malloc(sizeof(struct node));
    second = (struct node *)malloc(sizeof(struct node));
    third = (struct node *)malloc(sizeof(struct node));

    head->data = 10;
    head->next = second;

    second->data = 100;
    second->next = third;

    third->data = 1000;
    third->next = NULL;

    llTraversal(head);

    return 0;
}
```

**04 - Linked List\V16-I.c**

```c
// ------------------------------------------------------------
// insertion in linked list
/*
case 1 - insert at beginning O(1)
case 2 - insert at between O(n)
case 3 - insert at the end O(n)
case 4 - insert after a given node O(1)
*/
#include <stdio.h>
#include <stdlib.h>
// creating structure node
struct node
{
    int data;
    struct node *next;
};
// calling fucntion
void linkedListTraversal(struct node *ptr);
struct node *insertAtFirst(struct node *head, int data);
struct node *insertAtIndex(struct node *head, int data, int index);
struct node *insertAtEnd(struct node *head, int data);
struct node *insertAfterNode(struct node *head, struct node *prevNode, int data);
int main()
{
    /* //one way
        struct node *head;
        struct node *second;
        struct node *third;
        struct node *fourth;

        //allocate memory for nodes in the linked list in heap
        head = (struct node *)malloc(sizeof(struct node));
        second = (struct node *)malloc(sizeof(struct node));
        third = (struct node *)malloc(sizeof(struct node));
        fourth = (struct node *)malloc(sizeof(struct node));
    */
    // declare and allocate memory for nodes in the linked list in heap
    struct node *head = (struct node *)malloc(sizeof(struct node));
    struct node *second = (struct node *)malloc(sizeof(struct node));
    struct node *third = (struct node *)malloc(sizeof(struct node));
    struct node *fourth = (struct node *)malloc(sizeof(struct node));
    // creating linked list chain
    //  link first and second nodes
    head->data = 10;
    head->next = second;
    // link second and third nodes
    second->data = 20;
    second->next = third;
    // link third and fourth nodes
    third->data = 30;
    third->next = fourth;
    // terminating the list with null in 4th
    fourth->data = 40;
    fourth->next = NULL;
    printf("\nlist before insertion----------\n");
    linkedListTraversal(head);
    // head = insertAtFirst(head, 99);
```

```c
    // head = insertAtIndex(head, 99, 2);
    // head = insertAtEnd(head, 99);
    // head = insertAfterNode(head, third,99);
    printf("\nlist after insertion----------\n");
    linkedListTraversal(head);
    return 0;
}
void linkedListTraversal(struct node *ptr)
{
    while (ptr != NULL)
    {
        printf("elemnets: %d\n", ptr->data);
        ptr = ptr->next;
    }
}
struct node *insertAtFirst(struct node *head, int data)
{
    struct node *ptr = (struct node *)malloc(sizeof(struct node));
    ptr->data = data;
    ptr->next = head;
    return ptr;
}
struct node *insertAtIndex(struct node *head, int data, int index)
{
    struct node *ptr = (struct node *)malloc(sizeof(struct node));
    struct node *p = head;
    int i = 0;
    while (i != index - 1)
    {
        p = p->next;
        i++;
    }
    ptr->data = data;
    ptr->next = p->next;
    p->next = ptr;
    return head;
}
struct node *insertAtEnd(struct node *head, int data)
{
    struct node *ptr = (struct node *)malloc(sizeof(struct node));
    ptr->data = data;
    struct node *p = head;
    while (p->next != NULL)
    {
        p = p->next;
    }
    p->next = ptr;
    ptr->next = NULL;
    return head;
}
struct node * insertAfterNode(struct node* head,struct node * prevNode,int data){
    struct node * ptr = (struct node *)malloc(sizeof(struct node));
    ptr->data = data;
    ptr->next = prevNode->next;
    prevNode->next = ptr;
    return head;
}
```

**04 - Linked List\V17-D.c**

```c
//------------------------------------------------
//deletion of linked list
#include <stdio.h>
#include <stdlib.h>
struct Node{
    int data;
    struct Node *next;
};
void linkedListTraversal(struct Node *ptr){
    while (ptr != NULL){
        printf("Element: %d\n", ptr->data);
        ptr = ptr->next;
    }
}
// Case 1: Deleting the first element from the linked list
struct Node * deleteFirst(struct Node * head){
    struct Node * ptr = head;
    head = head->next;
    free(ptr);
    return head;
}
// Case 2: Deleting the element at a given index from the linked list
struct Node * deleteAtIndex(struct Node * head, int index){
    struct Node *p = head;
    struct Node *q = head->next;
    for (int i = 0; i < index-1; i++){
        p = p->next;
        q = q->next;
    }

    p->next = q->next;
    free(q);
    return head;
}
// Case 3: Deleting the last element
struct Node * deleteAtLast(struct Node * head){
    struct Node *p = head;
    struct Node *q = head->next;
    while(q->next !=NULL)  {
        p = p->next;
        q = q->next;
    }
    p->next = NULL;
    free(q);
    return head;
}
// Case 4: Deleting the element with a given value from the linked list
struct Node * deleteAtIndex(struct Node * head, int value){
    struct Node *p = head;
    struct Node *q = head->next;
    while(q->data!=value && q->next!= NULL){
        p = p->next;
        q = q->next;
    }
    if(q->data == value){
        p->next = q->next;
        free(q);
```

```c
    }
    return head;
}
int main(){
    struct Node *head;
    struct Node *second;
    struct Node *third;
    struct Node *fourth;
    // Allocate memory for nodes in the linked list in Heap
    head = (struct Node *)malloc(sizeof(struct Node));
    second = (struct Node *)malloc(sizeof(struct Node));
    third = (struct Node *)malloc(sizeof(struct Node));
    fourth = (struct Node *)malloc(sizeof(struct Node));
    // Link first and second nodes
    head->data = 4;
    head->next = second;
    // Link second and third nodes
    second->data = 3;
    second->next = third;
    // Link third and fourth nodes
    third->data = 8;
    third->next = fourth;
    // Terminate the list at the third node
    fourth->data = 1;
    fourth->next = NULL;
    printf("Linked list before deletion\n");
    linkedListTraversal(head);
    // head = deleteFirst(head); // For deleting first element of the linked list
    // head = deleteAtIndex(head, 2);
    head = deleteAtLast(head);
    printf("Linked list after deletion\n");
    linkedListTraversal(head);
    return 0;
}
```

**04 - Linked List\V20-CLL.c**

```c
// ------------------------------------------------------------
// circular linked list
#include <stdio.h>
#include <stdlib.h>
struct node{
    int data;
    struct node *next;
};
void linkedListTraversal(struct node *head){
    struct node *ptr = head;
    do{
        printf("element: %d\n", ptr->data);
        ptr = ptr->next;
    } while (ptr != head);
}
struct node * insertAtFirst(struct node * head, int data){
    struct node * ptr = (struct node *)malloc(sizeof(struct node));
    ptr->data = data;
    struct node * p = head->next;
    while(p->next != head){
        p = p->next;
    } //no its point to a bofore head
    p->next = ptr;
    ptr->next = head;
    head = ptr;
    return head;
}
int main(){
    struct node *head = (struct node *)malloc(sizeof(struct node));
    struct node *second = (struct node *)malloc(sizeof(struct node));
    struct node *third = (struct node *)malloc(sizeof(struct node));
    struct node *fourth = (struct node *)malloc(sizeof(struct node));
    head->data = 4;
    head->next = second;
    second->data = 3;
    second->next = third;
    third->data = 6;
    third->next = fourth;
    fourth->data = 1;
    fourth->next = head;
    printf("circular linked list traversal before insertion\n");
    linkedListTraversal(head);
    head = insertAtFirst(head,80);
    printf("circular linked list traversal after insertion\n");
    linkedListTraversal(head);
    return 0;
}
```

**04 - Linked List\V21-DLL.c**

```c
// -----------------------------------------------------------------
// doubly linked list
#include <stdio.h>
#include <stdlib.h>
struct node{
    struct node *prev;
    int data;
    struct node *next;
};
void linkedListTraversal(struct node *head){
    struct node *ptr = head;
    while (ptr != NULL){
        printf("elemnts: %d\n", ptr->data);
        ptr = ptr->next;
    }
    printf("-----\n");
    do{ //error here
        printf("element: %d\n", ptr->data);
        ptr = ptr->prev;
    } while (ptr != NULL);
}
int main(){
    struct node *head = (struct node *)malloc(sizeof(struct node));
    struct node *n2 = (struct node *)malloc(sizeof(struct node));
    struct node *n3 = (struct node *)malloc(sizeof(struct node));
    struct node *n4 = (struct node *)malloc(sizeof(struct node));
    head->next = n2;
    head->prev = NULL;
    head->data = 10;
    n2->next = n3;
    n2->prev = head;
    n2->data = 20;
    n3->next = n4;
    n3->prev = n2;
    n3->data = 30;
    n4->next = NULL;
    n4->prev = n3;
    n4->data = 40;
    linkedListTraversal(head);
    return 0;
}
```

**05- Stack\V24-stack.c**

```c
// -----------------------------------------------------
//stack
#include<stdio.h>
#include<stdlib.h>
struct stack{
    int size;
    int top;
    int *arr;
};
int isEmpty(struct stack * ptr){
    if(ptr->top == -1){
        return 1;
    }
    else{
        return 0;
    }
}
int isFull(struct stack * ptr){
    if (ptr->top == ptr->size-1){
        return 1;
    }
    else{
        return 0;
    }
}
void push(struct stack * ptr,int val){
    if(isFull(ptr)){
        printf("overflow");
    }
    else{
        ptr->top++;
        ptr->arr[ptr->top] = val;
    }
}
int pop(struct stack * ptr){
    if (isEmpty(ptr)){
        printf("underflow");
        return -1;
    }
    else{
        int val = ptr->arr[ptr->top];
        ptr->top--;
        return val;
    }
}
int peek(struct stack * ptr, int i){
    if (ptr->top-i+1<0){
        printf("invalid positon");
        return -1;
    }
    else{
        return ptr->arr[ptr->top-i+1];
    }
}
void display(struct stack * ptr){
    if (isEmpty(ptr)){
        printf("empty");
```

```c
    }
    else{
        for (int i = ptr->size-1; i >= 0; i--){
            printf("%d,",ptr->arr[i]);
        }
    }
}
int stackTop(struct stack * ptr){
    return ptr->arr[ptr->top];
}
int stackBottom(struct stack * ptr){
    return ptr->arr[0];
}
int main() {
    struct stack * S;
    S->size = 5;
    S->top = -1;
    S->arr = (int*)malloc(S->size * sizeof(int));
    printf("%d\n", isEmpty(S));
    printf("%d\n", isFull(S));
    push(S,50);
    push(S,50);
    push(S,50);
    push(S,50);
    push(S,50);
    printf("%d\n", isEmpty(S));
    printf("%d\n", isFull(S));
    printf("%d\n",peek(S,2));
    printf("%d",S->arr[S->top]);
    printf("\n----\n");
    display(S);
    return 0;
}
```

**05- Stack\V29-stack_LinkedList.c**

```c
//-------------------------------------------------------------
//stack using linked list
#include<stdio.h>
#include<stdlib.h>
struct stack{
    int size;
    int top;
    int * arr[];
};
struct node{
    int data;
    struct node * next;
};
void linkedListTraversal(struct node * top){
    while (top != NULL){
        printf("element: %d\n",top->data);
        top = top->next;
    }

}
int isEmpty(struct node * top){
    if (top==NULL){
        return 1;
    }
    else{
        return 0;
    }
}
int isFull(struct node * top){
    struct node * ptr = (struct node *)malloc(sizeof(struct node));
    if (ptr == NULL){
        return 1;
    }
    else{
        return 0;
    }
}
struct node * push(struct node * top, int x){
    if (isFull(top)){
        printf("overflow");
    }
    else{
        struct node * n  = (struct node *)malloc(sizeof(struct node));
        n->data = x;
        n->next = top;
        top = n;
        return top;
    }
}
int pop(struct node ** top){
    if (isEmpty(*top)){
        printf("underflow");
    }
    else{
        struct node * n = *top;
        *top = (*top)->next;
        int x = n->data;
```

```c
        free(n);
        return x;
    }
}
int peek(struct node *top, int pos) {
    struct node *ptr = top;
    for (int i = 0; (i < pos - 1 && ptr != NULL); i++) {
        ptr = ptr->next;
    }
    if (ptr != NULL) {
        return ptr->data;
    } else {
        return -1;
    }
}
int main() {
    struct node * top = NULL;
    top = push(top,10);
    top = push(top,11);
    top = push(top,12);
    top = push(top,13 );
    printf("\nbefore:----------------\n");
    linkedListTraversal(top);
    int el_p = pop(&top);
    printf("\nafter:----------------\n");
    linkedListTraversal(top);
    printf("poped elem is%d",el_p);
    return 0;
}
```

**05- Stack\V33-paranthesisMatching.c**

```c
//---------------------------------------------------------------------------
// application of stack -> paranthesis matching in an expresion
#include <stdio.h>
#include <stdlib.h>
struct stack
{
    int size;
    int top;
    char *arr;
};
int isEmpty(struct stack *ptr){
    if (ptr->top == -1){
        return 1;
    }
    else{
        return 0;
    }
}
int isFull(struct stack *ptr){
    if (ptr->top == ptr->size - 1){
        return 1;
    }
    else{
        return 0;
    }
}
void push(struct stack *ptr, char val){
    if (isFull(ptr)){
        printf("overflow");
    }
    else{
        ptr->top++;
        ptr->arr[ptr->top] = val;
    }
}
char pop(struct stack *ptr){
    if (isEmpty(ptr)){
        printf("underflow");
    }
    else{
        char val = ptr->arr[ptr->top];
        ptr->top--;
        return val;
    }
}
int parenthesisMatch(char *exp){
    struct stack * sp;
    sp->size = 100;
    sp->top = -1;
    sp->arr = (char *)malloc(sp->size * sizeof(char));
    for (int i = 0; exp[i]!= '\0'; i++){
        if (exp[i]=='('){
            push(sp,'(');
        }
        else if(exp[i]==')'){
            if (isEmpty(sp)){
                return 0;
```

```
            }
            pop(sp);
        }
    }
        if (isEmpty(sp)){
        return 1;
    }
    else{
        return 0;
    }
}
int main(){
    char * exp = "((45343ty536v2t4(2vtc32r)))";
    if (parenthesisMatch(exp)){
        printf("paranthesis is matching");
    }
    else{
        printf("paranthesis is not matching");
    }
    return 0;
}
```

**05- Stack\V34-multiParanthesis.c**

```c
//-------------------------------------------------------------
// multiple paranthesis matching
#include <stdio.h>
#include <stdlib.h>
struct stack{
    int size;
    int top;
    char *arr;
};
int isEmpty(struct stack *ptr){
    if (ptr->top == -1){
        return 1;
    }
    else{
        return 0;
    }
}
int isFull(struct stack *ptr){
    if (ptr->top == ptr->size - 1){
        return 1;
    }
    else{
        return 0;
    }
}
void push(struct stack *ptr, char val){
    if (isFull(ptr)){
        printf("overflowed");
    }
    else{
        ptr->top++;
        ptr->arr[ptr->top] = val;
    }
}
char pop(struct stack *ptr){
    if (isEmpty(ptr)){
        printf("underflow");
        return -1;
    }
    else{
        char val = ptr->arr[ptr->top];
        ptr->top--;
        return val;
    }
}
char stackTOP(struct stack *ptr){
    return ptr->arr[ptr->top];
}

int match(char a, char b){
    if (a == '(' && b == ')'){
        return 1;
    }
    if (a == '{' && b == '}'){
        return 1;
    }
    if (a == '[' && b == ']'){
```

```c
        return 1;
    }
    return 0;
}
int parenthesisMatch(char *exp){
    char popped_ch;
    struct stack *mmp = (struct stack *)malloc(sizeof(struct stack));
    mmp->size = 100;
    mmp->top = -1;
    mmp->arr = (char *)malloc(mmp->size * sizeof(char));
    for (int i = 0; exp[i] != '\0'; i++){
        if (exp[i] == '(' || exp[i] == '{' || exp[i] == '['){
            push(mmp, exp[i]);
        }
        else if (exp[i] == ')' || exp[i] == '}' || exp[i] == ']'){
            if (isEmpty(mmp)){
                return 0;
            }
            popped_ch = pop(mmp);
            if (!match(popped_ch, exp[i])){
                return 0;
            }
        }
    }
    if (isEmpty(mmp)){
        return 1;
    }
    else{
        return 0;
    }
}
int main(){
    char *exp = "{3424[343]}";
    if (parenthesisMatch(exp)){
        printf("the paranthesis is balanced");
    }
    else{
        printf("the patenthesis is not balanced");
    }
    return 0;
}
```

**05- Stack\V37-infixPrefixPostfix.c**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct stack{
    int size;
    int top;
    char *arr;
};
int stackTop(struct stack* sp){
    return sp->arr[sp->top];
}
int isEmpty(struct stack *ptr){
    if (ptr->top == -1){
        return 1;
    }
    else{
        return 0;
    }
}
int isFull(struct stack *ptr){
    if (ptr->top == ptr->size - 1){
        return 1;
    }
    else{
        return 0;
    }
}
void push(struct stack* ptr, char val){
    if(isFull(ptr)){
        printf("Stack Overflow! Cannot push %d to the stack\n", val);
    }
    else{
        ptr->top++;
        ptr->arr[ptr->top] = val;
    }
}
char pop(struct stack* ptr){
    if(isEmpty(ptr)){
        printf("Stack Underflow! Cannot pop from the stack\n");
        return -1;
    }
    else{
        char val = ptr->arr[ptr->top];
        ptr->top--;
        return val;
    }
}
int precedence(char ch){
    if(ch == '*' || ch=='/')
        return 3;
    else if(ch == '+' || ch=='-')
        return 2;
    else
        return 0;
}
int isOperator(char ch){
    if(ch=='+' || ch=='-' ||ch=='*' || ch=='/')
```

```c
        return 1;
    else
        return 0;
}
char* infixToPostfix(char* infix){
    struct stack * sp = (struct stack *) malloc(sizeof(struct stack));
    sp->size = 10;
    sp->top = -1;
    sp->arr = (char *) malloc(sp->size * sizeof(char));
    char * postfix = (char *) malloc((strlen(infix)+1) * sizeof(char));
    int i=0; // Track infix traversal
    int j = 0; // Track postfix addition
    while (infix[i]!='\0'){
        if(!isOperator(infix[i])){
            postfix[j] = infix[i];
            j++;
            i++;
        }
        else{
            if(precedence(infix[i])> precedence(stackTop(sp))){
                push(sp, infix[i]);
                i++;
            }
            else{
                postfix[j] = pop(sp);
                j++;
            }
        }
    }
    while (!isEmpty(sp))    {
        postfix[j] = pop(sp);
        j++;
    }
    postfix[j] = '\0';
    return postfix;
}
int main(){
    char * infix = "x-y/z-k*d";
    printf("postfix is %s", infixToPostfix(infix));
    return 0;
}
```

**06 - Queue\V41-Queue.c**

```c
#include<stdio.h>
#include<stdlib.h>
struct queue{
    int size,f,r;
    int* arr;
};
int isEmpty(struct queue *q){
    if(q->r==q->f){
        return 1;
    }
    return 0;
}
int isFull(struct queue *q){
    if(q->r==q->size-1){
        return 1;
    }
    return 0;
}
void enqueue(struct queue *q, int val){
    if(isFull(q)){
        printf("This Queue is full\n");
    }
    else{
        q->r++;
        q->arr[q->r] = val;
        printf("Enqued element: %d\n", val);
    }
}
int dequeue(struct queue *q){
    int a = -1;
    if(isEmpty(q)){
        printf("This Queue is empty\n");
    }
    else{
        q->f++;
        a = q->arr[q->f];
    }
    return a;
}
int main(){
    struct queue q;
    q.size = 4;
    q.f = q.r = 0;
    q.arr = (int*) malloc(q.size*sizeof(int));
    enqueue(&q, 12);                            // Enqueue few elements
    printf("Dequeuing element %d\n", dequeue(&q));
    enqueue(&q, 45);
    if(isEmpty(&q)){
        printf("Queue is empty\n");
    }
    if(isFull(&q)){
        printf("Queue is full\n");
    }
    return 0;
}
```

**06 - Queue\V44-CQ.c**

```c
//Circular Queue
#include<stdio.h>
#include<stdlib.h>
struct circularQueue{
    int size;
    int f;
    int r;
    int* arr;
};
int isEmpty(struct circularQueue *q){
    if(q->r==q->f){
        return 1;
    }
    return 0;
}
int isFull(struct circularQueue *q){
    if((q->r+1)%q->size == q->f){
        return 1;
    }
    return 0;
}
void enqueue(struct circularQueue *q, int val){
    if(isFull(q)){
        printf("This Queue is full");
    }
    else{
        q->r = (q->r +1)%q->size;
        q->arr[q->r] = val;
        printf("Enqued element: %d\n", val);
    }
}
int dequeue(struct circularQueue *q){
    int a = -1;
    if(isEmpty(q)){
        printf("This Queue is empty");
    }
    else{
        q->f = (q->f +1)%q->size;
        a = q->arr[q->f];
    }
    return a;
}
int main(){
    struct circularQueue q;
    q.size = 4;
    q.f = q.r = 0;
    q.arr = (int*) malloc(q.size*sizeof(int));
    enqueue(&q, 12);    // Enqueue few elements
    printf("Dequeuing element %d\n", dequeue(&q));
    enqueue(&q, 45);
    return 0;
}
```

**06 - Queue\V46-LLQ.c**

```c
//Queue with linked list
#include <stdio.h>
#include <stdlib.h>
struct Node *f = NULL;
struct Node *r = NULL;
struct Node{
    int data;
    struct Node *next;
};
void linkedListTraversal(struct Node *ptr){
    printf("Printing the elements of this linked list\n");
    while (ptr != NULL){
        printf("Element: %d\n", ptr->data);
        ptr = ptr->next;
    }
}
void enqueue(int val){
    struct Node *n = (struct Node *) malloc(sizeof(struct Node));
    if(n==NULL){
        printf("Queue is Full");
    }
    else{
        n->data = val;
        n->next = NULL;
        if(f==NULL){
            f=r=n;
        }
        else{
            r->next = n;
            r=n;
        }
    }
}
int dequeue(){
    int val = -1;
    struct Node *ptr = f;
    if(f==NULL){
        printf("Queue is Empty\n");
    }
    else{
        f = f->next;
        val = ptr->data;
        free(ptr);
    }
    return val;
}
int main(){
    linkedListTraversal(f);
    printf("Dequeuing element %d\n", dequeue());
    enqueue(34);
    printf("Dequeuing element %d\n", dequeue());
    linkedListTraversal(f);
    return 0;
}
```

**06 - Sortings\V51-BBS.c**

```c
//Bubble sort
#include<stdio.h>
void printArray(int* A, int n){
    for (int i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
}
void bubbleSort(int *A, int n){
    int temp;
    int isSorted = 0;
    for (int i = 0; i < n-1; i++) // For number of pass
    {
        printf("Working on pass number %d\n", i+1);
        for (int j = 0; j <n-1-i ; j++) // For comparison in each pass
        {
            if(A[j]>A[j+1]){
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
void bubbleSortAdaptive(int *A, int n){
    int temp;
    int isSorted = 0;
    for (int i = 0; i < n-1; i++){  // For number of pass
        printf("Working on pass number %d\n", i+1);
        isSorted = 1;
        for (int j = 0; j <n-1-i ; j++){    // For comparison in each pass
            if(A[j]>A[j+1]){
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
                isSorted = 0;
            }
        }
        if(isSorted){
            return;
        }
    }
}
int main(){
    // int A[] = {12, 54, 65, 7, 23, 9};
    int A[] = {1, 2, 5, 6, 12, 54, 625, 7, 23, 9, 987};
    // int A[] = {1, 2, 3, 4, 5, 6};
    int n = 11;
    printArray(A, n); // Printing the array before sorting
    bubbleSort(A, n); // Function to sort the array
    printArray(A, n); // Printing the array before sorting
    return 0;
}
```

**06 - Sortings\V53-IS.c**

```c
//insersion sort
#include<stdio.h>
void printArray(int* A, int n){
    for (int i = 0; i < n; i++){
        printf("%d ", A[i]);
    }
    printf("\n");
}
void insertionSort(int *A, int n){
    int key, j;
    for (int i = 1; i <= n-1; i++){ // Loop for passes
        key = A[i];
        j = i-1;
        while(j>=0 && A[j] > key){  // Loop for each pass
            A[j+1] = A[j];
            j--;
        }
        A[j+1] = key;
    }
}
int main(){
   // -1   0    1   2   3   4   5
   //      12,| 54, 65, 07, 23, 09 --> i=1, key=54, j=0
   //      12,| 54, 65, 07, 23, 09 --> 1st pass done (i=1)!
   //      12, 54,| 65, 07, 23, 09 --> i=2, key=65, j=1
   //      12, 54,| 65, 07, 23, 09 --> 2nd pass done (i=2)!
   //      12, 54, 65,| 07, 23, 09 --> i=3, key=7, j=2
   //      12, 54, 65,| 65, 23, 09 --> i=3, key=7, j=1
   //      12, 54, 54,| 65, 23, 09 --> i=3, key=7, j=0
   //      12, 12, 54,| 65, 23, 09 --> i=3, key=7, j=-1
   //      07, 12, 54,| 65, 23, 09 --> i=3, key=7, j=-1--> 3rd pass done (i=3)!
   // Fast forwarding and 4th and 5th pass will give:
   //      07, 12, 54, 65,| 23, 09 --> i=4, key=23, j=3
   //      07, 12, 23, 54,| 65, 09 --> After the 4th pass
   //      07, 12, 23, 54, 65,| 09 --> i=5, key=09, j=4
   //      07, 09, 12, 23, 54, 65| --> After the 5th pass
    int A[] = {12, 54, 65, 7, 23, 9};
    int n = 6;
    printArray(A, n);
    insertionSort(A, n);
    printArray(A, n);
    return 0;
}
```

**06 - Sortings\V55-SS.c**

```c
//selection sort
#include<stdio.h>
void printArray(int* A, int n){
    for (int i = 0; i < n; i++){
        printf("%d ", A[i]);
    }
    printf("\n");
}
void selectionSort(int *A, int n){
    int indexOfMin, temp;
    printf("Running Selection sort...\n");
    for (int i = 0; i < n-1; i++){
        indexOfMin = i;
        for (int j = i+1; j < n; j++){
            if(A[j] < A[indexOfMin]){
                indexOfMin = j;
            }
        }
        // Swap A[i] and A[indexOfMin]
        temp = A[i];
        A[i] = A[indexOfMin];
        A[indexOfMin] = temp;
    }
}
int main(){
    // Input Array (There will be total n-1 passes. 5-1 = 4 in this case!)
    //  00  01  02  03  04
    // |03, 05, 02, 13, 12
    // After first pass
    //  00  01  02  03  04
    //  02,|05, 03, 13, 12
    // After second pass
    // 00  01  02  03  04
    // 02, 03,|05, 13, 12
    // After third pass
    // 00  01  02  03  04
    // 02, 03, 05,|13, 12
    // After fourth pass
    // 00  01  02  03  04
    // 02, 03, 05, 12,|13
    int A[] = {3, 5, 2, 13, 12};
    int n = 5;
    printArray(A, n);
    selectionSort(A, n);
    printArray(A, n);
    return 0;
}
```

**06 - Sortings\V56-QS.c**

```c
//Quick sort
#include <stdio.h>
void printArray(int *A, int n){
    for (int i = 0; i < n; i++){
        printf("%d ", A[i]);
    }
    printf("\n");
}
int partition(int A[], int low, int high){
    int pivot = A[low];
    int i = low + 1;
    int j = high;
    int temp;
    do{
        while (A[i] <= pivot){
            i++;
        }
        while (A[j] > pivot){
            j--;
        }
        if (i < j){
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    } while (i < j);
    temp = A[low];// Swap A[low] and A[j]
    A[low] = A[j];
    A[j] = temp;
    return j;
}
void quickSort(int A[], int low, int high){
    int partitionIndex; // Index of pivot after partition
    if (low < high){
        partitionIndex = partition(A, low, high);
        quickSort(A, low, partitionIndex - 1);  // sort left subarray
        quickSort(A, partitionIndex + 1, high); // sort right subarray
    }
}
int main(){
    //int A[] = {3, 5, 2, 13, 12, 3, 2, 13, 45};
    int A[] = {9, 4, 4, 8, 7, 5, 6};
    // 3, 5, 2, 13, 12, 3, 2, 13, 45
    // 3, 2, 2, 13i, 12, 3j, 5, 13, 45
    // 3, 2, 2, 3j, 12i, 13, 5, 13, 45 --> first call to partition returns 3
    int n = 9;
    n =7;
    printArray(A, n);
    quickSort(A, 0, n - 1);
    printArray(A, n);
    return 0;
}
```

**06 - Sortings\V59-MS.c**

```c
//merge sort
#include <stdio.h>
void printArray(int *A, int n){
    for (int i = 0; i < n; i++){
        printf("%d ", A[i]);
    }
    printf("\n");
}
void merge(int A[], int mid, int low, int high){
    int i, j, k, B[100];
    i = low,j = mid + 1,k = low;
    while (i <= mid && j <= high){
        if (A[i] < A[j]){
            B[k] = A[i];
            i++;
            k++;
        }
        else{
            B[k] = A[j];
            j++;
            k++;
        }
    }
    while (i <= mid){
        B[k] = A[i];
        k++;
        i++;
    }
    while (j <= high){
        B[k] = A[j];
        k++;
        j++;
    }
    for (int i = low; i <= high; i++){
        A[i] = B[i];
    }
}
void mergeSort(int A[], int low, int high){
    int mid;
    if(low<high){
        mid = (low + high) /2;
        mergeSort(A, low, mid);
        mergeSort(A, mid+1, high);
        merge(A, mid, low, high);
    }
}
int main(){
    int A[] = {9, 1, 4, 14, 4, 15, 6};
    int n = 7;
    printArray(A, n);
    mergeSort(A, 0, 6);
    printArray(A, n);
    return 0;
}
```

**06 - Sortings\V60-CS.c**

```c
//count sort
#include<stdio.h>
#include<limits.h>
#include<stdlib.h>
void printArray(int *A, int n){
    for (int i = 0; i < n; i++){
        printf("%d ", A[i]);
    }
    printf("\n");
}
int maximum(int A[], int n){
    int max = INT_MIN;
    for (int i = 0; i < n; i++){
        if (max < A[i]){
            max = A[i];
        }
    }
    return max;
}
void countSort(int * A, int n){
    int i, j;
    // Find the maximum element in A
    int max = maximum(A, n);
    // Create the count array
    int* count = (int *) malloc((max+1)*sizeof(int));
    // Initialize the array elements to 0
    for (i = 0; i < max+1; i++){
        count[i] = 0;
    }
    // Increment the corresponding index in the count array
    for (i = 0; i < n; i++){
        count[A[i]] = count[A[i]] + 1;
    }
    i =0; // counter for count array
    j =0; // counter for given array A
    while(i<= max){
        if(count[i]>0){
            A[j] = i;
            count[i] = count[i] - 1;
            j++;
        }
        else{
            i++;
        }
    }
}
int main(){
    int A[] = {9, 1, 4, 14, 4, 15, 6};
    int n = 7;
    printArray(A, n);
    countSort(A, n);
    printArray(A, n);
    return 0;
}
```

**07 - Binary Tree\V65-BT.c**

```c
//Linked list representation of binary tree
#include<stdio.h>
#include<malloc.h>
struct node{
    int data;
    struct node* left;
    struct node* right;
};
struct node* createNode(int data){
    struct node *n; // creating a node pointer
    n = (struct node *) malloc(sizeof(struct node)); // Allocating memory in the heap
    n->data = data; // Setting the data
    n->left = NULL; // Setting the left and right children to NULL
    n->right = NULL; // Setting the left and right children to NULL
    return n; // Finally returning the created node
}
int main(){
    /*
    // Constructing the root node
    struct node *p;
    p = (struct node *) malloc(sizeof(struct node));
    p->data = 2;
    p->left = NULL;
    p->right = NULL;

    // Constructing the second node
    struct node *p1;
    p1 = (struct node *) malloc(sizeof(struct node));
    p->data = 1;
    p1->left = NULL;
    p1->right = NULL;

    // Constructing the third node
    struct node *p2;
    p2 = (struct node *) malloc(sizeof(struct node));
    p->data = 4;
    p2->left = NULL;
    p2->right = NULL;
    */
    // Constructing the root node - Using Function (Recommended)
    struct node *p = createNode(2);
    struct node *p1 = createNode(1);
    struct node *p2 = createNode(4);
    // Linking the root node with left and right children
    p->left = p1;
    p->right = p2;
    return 0;
}
```

**07 - Binary Tree\V70-OT.c**

```c
//PreOrder, PostOrder and InOrder tranversal in Tree
#include<stdio.h>
#include<malloc.h>
struct node{
    int data;
    struct node* left;
    struct node* right;
};
struct node* createNode(int data){
    struct node *n; // creating a node pointer
    n = (struct node *) malloc(sizeof(struct node)); // Allocating memory in the heap
    n->data = data; // Setting the data
    n->left = NULL; // Setting the left and right children to NULL
    n->right = NULL; // Setting the left and right children to NULL
    return n; // Finally returning the created node
}
void preOrder(struct  node* root){
    if(root!=NULL){
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}
void postOrder(struct  node* root){
    if(root!=NULL){
        postOrder(root->left);
        postOrder(root->right);
        printf("%d ", root->data);
    }
}
void inOrder(struct  node* root){
    if(root!=NULL){
        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);
    }
}
int main(){
    // Constructing the root node - Using Function (Recommended)    // Finally The tree looks like this:
    struct node *p = createNode(4);                                 //      4
    struct node *p1 = createNode(1);                                //     / |
    struct node *p2 = createNode(6);                                //    1   6
    struct node *p3 = createNode(5);                                //   / |
    struct node *p4 = createNode(2);                                //  5  2
    // Linking the root node with left and right children
    p->left = p1;
    p->right = p2;
    p1->left = p3;
    p1->right = p4;
    // preOrder(p); printf("\n"); postOrder(p);
    inOrder(p);
    return 0;
}
```

**07 - Binary Tree\V72-!BST.c**

```c
//Checking if a binary tree is a binary search tree or not
#include<stdio.h>
#include<malloc.h>
struct node{
    int data;struct node* left;struct node* right;
};
struct node* createNode(int data){
    struct node *n; // creating a node pointer
    n = (struct node *) malloc(sizeof(struct node)); // Allocating memory in the heap
    n->data = data; // Setting the data
    n->left = NULL; // Setting the left and right children to NULL
    n->right = NULL; // Setting the left and right children to NULL
    return n; // Finally returning the created node
}
void preOrder(struct  node* root);      //Functions
void postOrder(struct  node* root);     //Functions
void inOrder(struct  node* root);       //Functions
int isBST(struct  node* root){
    static struct node *prev = NULL;
    if(root!=NULL){
        if(!isBST(root->left)){
            return 0;
        }
        if(prev!=NULL && root->data <= prev->data){
            return 0;
        }
        prev = root;
        return isBST(root->right);
    }
    else{
        return 1;
    }
}
int main(){
    struct node *p = createNode(5);                                          // Constructing the root node - Using Function (Recommended)
    struct node *p1 = createNode(3);                                         // Finally The tree looks like this:
    struct node *p2 = createNode(6);                                         //      5
    struct node *p3 = createNode(1);                                         //     / |
    struct node *p4 = createNode(4);                                         //    3   6
    p->left = p1;                                                            //   / |
    p->right = p2;  // Linking the root node with left and right children   //  1   4
    p1->left = p3;
    p1->right = p4;
    inOrder(p);     // preOrder(p); printf("\n"); postOrder(p);
    printf("\n");   // printf("%d", isBST(p));
    if(isBST(p)){
        printf("This is a bst" );
    }
    else{
        printf("This is not a bst");
    }
    return 0;}
```

**07 - Binary Tree\V74-BST.c**

```c
// Searching in a BST
#include<stdio.h>
#include<malloc.h>
struct node{
    int data;
    struct node* left;
    struct node* right;
};
struct node* createNode(int data){
    struct node *n; // creating a node pointer
    n = (struct node *) malloc(sizeof(struct node)); // Allocating memory in the heap
    n->data = data; // Setting the data
    n->left = NULL; // Setting the left and right children to NULL
    n->right = NULL; // Setting the left and right children to NULL
    return n; // Finally returning the created node
}
int isBST(struct  node* root);          //FUCNTION
struct node * search(struct node* root, int key){
    if(root==NULL){
        return NULL;
    }
    if(key==root->data){
        return root;
    }
    else if(key<root->data){
        return search(root->left, key);
    }
    else{
        return search(root->right, key);
    }
}
int main(){
    struct node *p = createNode(5);         // Constructing the root node - Using Function (Recommended)
    struct node *p1 = createNode(3);        // Finally The tree looks like this:
    struct node *p2 = createNode(6);        //      5
    struct node *p3 = createNode(1);        //     / |
    struct node *p4 = createNode(4);        //    3   6
    p->left = p1;                           //   / |
    p->right = p2;                          //  1   4
    p1->left = p3;                          // Linking the root node with left and right children
    p1->right = p4;
    struct node* n = search(p, 10);
    if(n!=NULL){
    printf("Found: %d", n->data);
    }
    else{
        printf("Element not found");
    }
    return 0;
}
```

**07 - Binary Tree\V75-ISBST.c**

```c
//Iterative Search in a Binary Search Tree
#include<stdio.h>
#include<malloc.h>
struct node{
    int data;
    struct node* left;
    struct node* right;
};
struct node* createNode(int data){
    struct node *n; // creating a node pointer
    n = (struct node *) malloc(sizeof(struct node)); // Allocating memory in the heap
    n->data = data; // Setting the data
    n->left = NULL; // Setting the left and right children to NULL
    n->right = NULL; // Setting the left and right children to NULL
    return n; // Finally returning the created node
}
int isBST(struct  node* root);                          //FUNCTION
struct node * searchIter(struct node* root, int key){
    while(root!=NULL){
        if(key == root->data){
            return root;
        }
        else if(key<root->data){
            root = root->left;
        }
        else{
            root = root->right;
        }
    }
    return NULL;
}
int main(){
    struct node *p = createNode(5);   //      5             // Finally The tree looks like this:
    struct node *p1 = createNode(3);  //     / |            // Constructing the root node - Using Function (Recommended)
    struct node *p2 = createNode(6);  //    3   6
    struct node *p3 = createNode(1);  //   / |
    struct node *p4 = createNode(4);  //  1   4
    p->left = p1;// Linking the root node with left and right children
    p->right = p2;
    p1->left = p3;
    p1->right = p4;
    struct node* n = searchIter(p, 6);
    if(n!=NULL){
    printf("Found: %d", n->data);
    }
    else{
        printf("Element not found");
    }
    return 0;
}
```

**07 - Binary Tree\V76-insBST.c**

```c
//insertion in BST
#include<stdio.h>
#include<malloc.h>
struct node{
    int data;struct node* left;struct node* right;
};
struct node* createNode(int data);  //FUNCTION
int isBST(struct  node* root);      //FUNCTION
struct node * searchIter(struct node* root, int key){
    while(root!=NULL){
        if(key == root->data){
            return root;
        }
        else if(key<root->data){
            root = root->left;
        }
        else{
            root = root->right;
        }
    }
    return NULL;
}
void insert(struct node *root, int key){
   struct node *prev = NULL;
   while(root!=NULL){
       prev = root;
       if(key==root->data){
           printf("Cannot insert %d, already in BST", key);
           return;
       }
       else if(key<root->data){
           root = root->left;
       }
       else{
           root = root->right;
       }
   }
   struct node* new = createNode(key);
   if(key<prev->data){
       prev->left = new;
   }
   else{
       prev->right = new;
   }
}
int main(){
    struct node *p = createNode(5);
    struct node *p1 = createNode(3);struct node *p2 = createNode(6);
    struct node *p3 = createNode(1);struct node *p4 = createNode(4);
    p->left = p1;p->right = p2;p1->left = p3;p1->right = p4;
    insert(p, 16);
    printf("%d", p->right->right->data);
    return 0;
}
```

**07 - Binary Tree\V78-delBST.c**

```c
//deletion in BST
#include<stdio.h>
#include<malloc.h>
struct node{
    int data;struct node* left;struct node* right;
};
struct node* createNode(int data);                      //FUNCTION
int isBST(struct  node* root);                          //FUNCTION
struct node * searchIter(struct node* root, int key);   //FUNCTION
struct node *inOrderPredecessor(struct node* root){
    root = root->left;
    while (root->right!=NULL){
        root = root->right;
    }
    return root;
}
struct node *deleteNode(struct node *root, int value){
    struct node* iPre;
    if (root == NULL){
        return NULL;
    }
    if (root->left==NULL&&root->right==NULL){
        free(root);
        return NULL;
    }
    if (value < root->data){        //searching for the node to be deleted
        root-> left = deleteNode(root->left,value);
    }
    else if (value > root->data){
        root-> right = deleteNode(root->right,value);
    }
    else{                           //deletion strategy when the node is found
        iPre = inOrderPredecessor(root);
        root->data = iPre->data;
        root->left = deleteNode(root->left, iPre->data);
    }
    return root;
}
int main(){
    struct node *p = createNode(5);struct node *p1 = createNode(3);
    struct node *p2 = createNode(6);struct node *p3 = createNode(1);
    struct node *p4 = createNode(4);
    p->left = p1;p->right = p2;p1->left = p3;p1->right = p4;
    inOrder(p);
    printf("\n");
    deleteNode(p, 3);
    inOrder(p);
    return 0;
}
```

**08 - AVL tree\V82-IRAVL.c**

```c
//AVL Tree Insertion & Rotation
#include <stdio.h>
#include <stdlib.h>
struct Node{
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};
int getHeight(struct Node *n){
    if(n==NULL)
        return 0;
    return n->height;
}
struct Node *createNode(int key){
    struct Node* node = (struct Node *) malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return node;
}
int max (int a, int b){
    return (a>b)?a:b;
}
int getBalanceFactor(struct Node * n){
    if(n==NULL){
        return 0;
    }
    return getHeight(n->left) - getHeight(n->right);
}
struct Node* rightRotate(struct Node* y){
    struct Node* x = y->left;
    struct Node* T2 = x->right;
    x->right = y;
    y->left = T2;
    x->height = max(getHeight(x->right), getHeight(x->left)) + 1;
    y->height = max(getHeight(y->right), getHeight(y->left)) + 1;
    return x;
}
struct Node* leftRotate(struct Node* x){
    struct Node* y = x->right;
    struct Node* T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = max(getHeight(x->right), getHeight(x->left)) + 1;
    y->height = max(getHeight(y->right), getHeight(y->left)) + 1;
    return y;
}
struct Node *insert(struct Node* node, int key){
    if (node == NULL)
        return  createNode(key);
    if (key < node->key)
        node->left  = insert(node->left, key);
```

```
    else if (key > node->key)
        node->right = insert(node->right, key);
        node->height = 1 + max(getHeight(node->left), getHeight(node->right));
    int bf = getBalanceFactor(node);
        if(bf>1 && key < node->left->key){          // Left Left Case
            return rightRotate(node);
        }
        if(bf<-1 && key > node->right->key){        // Right Right Case
            return leftRotate(node);
        }
    if(bf>1 && key > node->left->key){              // Left Right Case
            node->left = leftRotate(node->left);
            return rightRotate(node);
        }
    if(bf<-1 && key < node->right->key){            // Right Left Case
            node->right = rightRotate(node->right);
            return leftRotate(node);
        }
    return node;
}
void preOrder(struct Node *root){
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}
int main(){
    struct Node * root = NULL;
    root = insert(root, 1);
    root = insert(root, 2);
    root = insert(root, 4);
    root = insert(root, 5);
    root = insert(root, 6);
    root = insert(root, 3);
    preOrder(root);
    return 0;
}
```

**09 - Graphs\V87-BFS.c**

```c
// BFS Implementation

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

struct queue
{
    int size;
    int f;
    int r;
    int* arr;
};


int isEmpty(struct queue *q){
    if(q->r==q->f){
        return 1;
    }
    return 0;
}

int isFull(struct queue *q){
    if(q->r==q->size-1){
        return 1;
    }
    return 0;
}

void enqueue(struct queue *q, int val){
    if(isFull(q)){
        printf("This Queue is full\n");
    }
    else{
        q->r++;
        q->arr[q->r] = val;
        // printf("Enqued element: %d\n", val);
    }
}

int dequeue(struct queue *q){
    int a = -1;
    if(isEmpty(q)){
        printf("This Queue is empty\n");
    }
    else{
        q->f++;
        a = q->arr[q->f];
    }
    return a;
}

int main(){
    // Initializing Queue (Array Implementation)
    struct queue q;
```

```c
    q.size = 400;
    q.f = q.r = 0;
    q.arr = (int*) malloc(q.size*sizeof(int));

    // BFS Implementation
    int node;
    int i = 1;
    int visited[7] = {0,0,0,0,0,0,0};
    int a [7][7] = {
        {0,1,1,1,0,0,0},
        {1,0,1,0,0,0,0},
        {1,1,0,1,1,0,0},
        {1,0,1,0,1,0,0},
        {0,0,1,1,0,1,1},
        {0,0,0,0,1,0,0},
        {0,0,0,0,1,0,0}
    };
    printf("%d", i);
    visited[i] = 1;
    enqueue(&q, i); // Enqueue i for exploration
    while (!isEmpty(&q))
    {
        int node = dequeue(&q);
        for (int j = 0; j < 7; j++)
        {
            if(a[node][j] ==1 && visited[j] == 0){
                printf("%d", j);
                visited[j] = 1;
                enqueue(&q, j);
            }
        }
    }
    return 0;
}
```

**09 - Graphs\V89-DFS.c**

```c
// DFS Implementation

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

int visited[7] = {0,0,0,0,0,0,0};
    int A [7][7] = {
        {0,1,1,1,0,0,0},
        {1,0,1,0,0,0,0},
        {1,1,0,1,1,0,0},
        {1,0,1,0,1,0,0},
        {0,0,1,1,0,1,1},
        {0,0,0,0,1,0,0},
        {0,0,0,0,1,0,0}
    };

void DFS(int i){
    printf("%d ", i);
    visited[i] = 1;
    for (int j = 0; j < 7; j++)
    {
        if(A[i][j]==1 && !visited[j]){
            DFS(j);
        }
    }
}

int main(){
    // DFS Implementation
    DFS(0);
    return 0;
}
```