DATA STRUCTURES CODES

1.Infix to Postfix

2.Infix to Prefix

3.Prefix to Infix

4.Prefix to Postfix

5.Postfix to Prefix

6.Postfix to Infix

7.primitive operations of circular queue

8.primitive operations of double ended queue

9.primitive operations of priority queue

10.check string with parenthesis and validate it using stack

11.Fibonacci series using stack

12.decimal to binary conversion using stack

13.Josephus problem using stack

14.Josephus problem using linked list

15.singly linked list. (Insertion at beginning, Deletion at middle, Display)

16.singly linked list. (Insertion at middle, Deletion at end, Display)

17.singly linked list. (Insertion at end, Deletion at beginning, Display)

18.single variable polynomials using singly linked list. (Accept a sorted polynomial, Addition of two polynomials, Display)

19. accept two sorted single linked lists and merge them in a single linked list in such a way that the resultant linked list will be a sorted one

20.Postorder Traversal, Preorder Traversal, Inorder Traversal.

21.Copy a tree, Equality of two trees, Delete a node from a tree.

22.Insert a node in a tree, Display the height of the tree, Display a tree levelwise

23.Display mirror image of a tree by creating new tree, Display mirror image of a tree without creating new tree, Display leaf nodes of a tree

24. Write a C program to accept a binary search tree and display mirror image of a tree by creating new tree(part of 23)

25.leaf nodes level-wise.

26.deletion of a node in a binary search tree.

27.deletion of a node in a binary tree.

28.Threaded Binary tree and traverse it in-order.

29.Heap sort using Max heap in descending order.

30.Heap sort using Min heap in ascending order.

31.Kruskal's algorithm for min. spanning tree, in which a graph is represented using an adjacency matrix.

32.Prim's algorithm for min. spanning tree, in which a graph is represented using an adjacency list.

33.Dijkstra's algorithm in which a graph is represented using an adjacency matrix.

34.accept a graph from the user, represent it in adjacency list and traverse it in-order

35.accept a graph from user, represent in adjacency matrix and traverse it in-order

1./*INFIX TO POSTFIX EXPRESSION USING STACK*/

```
#include <stdio.h>
#include<conio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
//#include <ctype.h>
#define bool int
#define true 1
#define false 0
int stack[100];
int top = -1;
int variableValues[26];  // Store variable values
int isFull() {
return top == 100 - 1;
}
int isEmpty() {
return top == -1;
}
void push(int element) {
if (!isFull()) {
top++;
stack[top] = element;
} else {
printf("Stack Overflow!\n");
exit(1);
}
}
int pop() {
if (!isEmpty()) {
int element = stack[top];
```

```
top--;
return element;
} else {
printf("Stack is Empty\n");
exit(1);
}
return 0;
}
int preference(char op) {
if (op == '^') {
return 3;
}
if (op == '*' || op == '/') {
return 2;
}
if (op == '+' || op == '-') {
return 1;
}
return 0;
}
void infixToPostfix(char infix[], char postfix[]) {
int i, j = 0;
char symbol, next;
for (i = 0; i < (int)strlen(infix); i++) {
symbol = infix[i];
if (!isspace(symbol)) {
switch (symbol) {
case '(':
case '[':
case '{':
push(symbol);
```

```
break;
case ')':
case ']':
case '}':
while (!isEmpty() && ((next = pop()) != '(') && (next != '[') && (next != '{')) {
postfix[j++] = next;
}
break;
case '+':
case '-':
case '*':
case '/':
case '^':
while (!isEmpty() && preference(stack[top]) >= preference(symbol)) {
postfix[j++] = pop();
}
push(symbol);
break;
default:
postfix[j++] = symbol;
}
}
}
while (!isEmpty()) {
postfix[j++] = pop();
}
postfix[j] = '\0';
}
int evaluate(char postfix[]) {
int i, a, b;
int x = strlen(postfix);
```

```
for (i = 0; i < x; i++) {
if (isalpha(postfix[i])) {
push(variableValues[postfix[i] - 'A']);
} else if (isdigit(postfix[i])) {
push(postfix[i] - '0');
} else {
a = pop();
b = pop();
switch (postfix[i]) {
case '+':
push(b + a);
break;
case '-':
push(b - a);
break;
case '*':
push(b * a);
break;
case '/':
push(b / a);
break;
case '^':
push(pow(b, a));
break;
}
}
}
return pop();
}

int main() {
```

```
int i,x,result;
char postfix[100];
char variable,infix[100],highestVariable;
clrscr();
printf("\nEnter Your Expression: ");
scanf("%s", infix);
infixToPostfix(infix, postfix);
printf("Postfix Expression: %s\n", postfix);
x = strlen(postfix);
highestVariable = 'A' - 1;
for (i = 0; i < x; i++) {
if (isalpha(postfix[i])) {
if (postfix[i] > highestVariable) {
highestVariable = postfix[i];
}
}
}
for (variable = 'A'; variable <= highestVariable; variable++) {
printf("Enter value for variable %c: ", variable);
scanf("%d", &variableValues[variable - 'A']);
}
result = evaluate(postfix);
printf("The result of postfix evaluation = %d\n", result);
getch();
return 0;
}
```

2./*INFIX TO PREFIX EXPRESSION USING STACK*/

```
#include <stdio.h>

#include<conio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX_SIZE 100

// Stack implementation

typedef struct {

int top;

char items[MAX_SIZE];

} Stack;

void initialize(Stack *s) {

s->top = -1;

}

int isEmpty(Stack *s) {

return s->top == -1;

}

int isFull(Stack *s) {

return s->top == MAX_SIZE - 1;

}

void push(Stack *s, char item) {

if (isFull(s)) {

printf("Stack overflow\n");

exit(EXIT_FAILURE);

}

s->items[++s->top] = item;

}

char pop(Stack *s) {

if (isEmpty(s)) {

printf("Stack underflow\n");
```

```
exit(EXIT_FAILURE);
}
return s->items[s->top--];
}
char peek(Stack *s) {
return s->items[s->top];
}
int isOperator(char c) {
return (c == '+' || c == '-' || c == '*' || c == '/');
}
int precedence(char c) {
if (c == '+' || c == '-')
return 1;
if (c == '*' || c == '/')
return 2;
return 0;
}
void infixToPrefix(char infix[], char prefix[]) {
Stack operators, result;
int i, j = 0;
char symbol;
initialize(&operators);
initialize(&result);
for (i = strlen(infix) - 1; i >= 0; i--) {
symbol = infix[i];
if (isalnum(symbol)) {
// Operand
prefix[j++] = symbol;
} else if (symbol == ')') {
// Right parenthesis
push(&operators, symbol);
```

```
} else if (symbol == '(') {
// Left parenthesis
while (!isEmpty(&operators) && peek(&operators) != ')') {
prefix[j++] = pop(&operators);
}
pop(&operators); // Discard the ')'
} else if (isOperator(symbol)) {
// Operator
while (!isEmpty(&operators) && precedence(symbol) < precedence(peek(&operators))) {
prefix[j++] = pop(&operators);
}
push(&operators, symbol);
}
}
// Pop remaining operators from the stack
while (!isEmpty(&operators)) {
prefix[j++] = pop(&operators);
}
// Null-terminate the prefix expression
prefix[j] = '\0';
// Reverse the prefix expression to get the correct order
strrev(prefix);
}
int evaluatePrefix(char prefix[]) {
Stack s;
int i, result;
initialize(&s);
for (i = 0; prefix[i] != '\0'; i++) {
char symbol = prefix[i];
if (isdigit(symbol)) {
// Operand
```

```
push(&s, symbol - '0');
} else if (isOperator(symbol)) {
// Operator
int operand2 = pop(&s);
int operand1 = pop(&s);
switch (symbol) {
case '+':
push(&s, operand1 + operand2);
break;
case '-':
push(&s, operand1 - operand2);
break;
case '*':
push(&s, operand1 * operand2);
break;
case '/':
push(&s, operand1 / operand2);
break;
}
}
}
result = pop(&s);
return result;
}
int main() {
char infix[MAX_SIZE], prefix[MAX_SIZE];
clrscr();
printf("Enter the infix expression: ");
scanf("%s", infix);
infixToPrefix(infix, prefix);
printf("Prefix expression: %s\n", prefix);
```

```
//    int result = evaluatePrefix(prefix);
//    printf("Result after evaluation: %d\n", result);
getch();
return 0;
}
```

3./*PREFIX TO INFIX EXPRESSION USING STACK*/

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char stack[50][50];
int top = -1;
void clear_stack() {
top = -1;
}
void push(char *s) {
strcpy(stack[++top], s);
}
char* pop() {
return stack[top--];
}
int is_operator(char x) {
if (x == '+' || x == '-' || x == '*' || x == '/') {
return 1;
} else {
return 0;
}
}
// Function to Convert prefix to Postfix
void convert(char *exp) {
int i, l;
char op1[50], op2[50];
clear_stack();
l = strlen(exp);
// Scanning from right to left
for (i = l - 1; i >= 0; i--) {
// Checking if the symbol is an operator
if (is_operator(exp[i])) {
// Popping two operands from stack
strcpy(op1, pop());
strcpy(op2, pop());
// Concatenating the operands and operator
sprintf(stack[++top], "%s%c%s", op1,exp[i], op2);
} else {
// If it is an operand, push the operand to the stack
sprintf(stack[++top], "%c", exp[i]);
}
}
// Print the postfix expression
printf("%s\n", stack[top]);
}
// Main function
int main() {
char expression[50];
// Prompt the user to enter the expression
printf("Enter the prefix expression: ");
// Read the expression from the user
```

```
fgets(expression, sizeof(expression), stdin);
// Remove the newline character from the end of the input
expression[strcspn(expression, "\n")] = '\0';
// Convert the expression
convert(expression);
return 0;
}
```

4./*PREFIX TO POSTFIX EXPRESSION USING STACK*/

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

char stack[50][50];
int top = -1;

void clear_stack() {
top = -1;
}

void push(char *s) {
strcpy(stack[++top], s);
}

char *pop() {
return stack[top--];
}

int is_operator(char x) {
if (x == '+' || x == '-' || x == '*' || x == '/') {
return 1;
} else {
return 0;
}
}

// Function to Convert prefix to Postfix
void convert(char *exp) {
int i, l;
char op1[50], op2[50];
clear_stack();
l = strlen(exp);

// Scanning from right to left
for (i = l - 1; i >= 0; i--) {
// Checking if the symbol is an operator
if (is_operator(exp[i])) {
// Popping two operands from stack
strcpy(op1, pop());
strcpy(op2, pop());
// Concatenating the operands and operator
sprintf(stack[++top], "%s%s%c", op1, op2, exp[i]);
} else {
// If it is an operand, push the operand to the stack
sprintf(stack[++top], "%c", exp[i]);
}
}
// Print the postfix expression
printf("%s\n", stack[top]);
}
```

```
// Main function
int main() {
char expression[50];
printf("Enter the expression: ");
scanf("%s", expression);

convert(expression);

return 0;
}
```

5./*POSTFIX TO PREFIX EXPRESSION USING STACK*/

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

char stack[50][50];
int top = -1;

void clear_stack() {
top = -1;
}

void push(char *s) {
strcpy(stack[++top], s);
}

char *pop() {
return stack[top--];
}

int is_operator(char x) {
if (x == '+' || x == '-' || x == '*' || x == '/') {
return 1;
} else {
return 0;
}
}

// Function to Convert postfix to Prefix
void convert(char *exp) {
int i, l;
char op1[50], op2[50];
clear_stack();
l = strlen(exp);

// Scanning from left to right
for (i = 0; i < l; i++) {
// Checking if the symbol is an operator
if (is_operator(exp[i])) {
// Popping two operands from stack
strcpy(op2, pop());
strcpy(op1, pop());
// Concatenating the operator and operands
sprintf(stack[++top], "%c%s%s", exp[i], op1, op2);
} else {
// If it is an operand, push the operand to the stack
sprintf(stack[++top], "%c", exp[i]);
}
}
// Print the prefix expression
printf("%s\n", stack[top]);
```

```
}

// Main function
void main() {
clrscr();

// Accept expression from the user
char expression[50];
printf("Enter the postfix expression: ");
scanf("%s", expression);

// Convert and print the prefix expression
convert(expression);

getch();
}
```

6./*POSTFIX TO INFIX EXPRESSION USING STACK*/

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

// Global Variable
char stack[50];
int top = -1;

// Function to Push Elements into Stack
void push(char ch) {
stack[++top] = ch;
}

// Function to Pop Element From The Stack
char pop() {
return stack[top--];
}

// Function to reverse a string
void revstr(char str[]) {
int length = strlen(str);
int i, j;
char temp;

for (i = 0, j = length - 1; i < j; i++, j--) {
temp = str[i];
str[i] = str[j];
str[j] = temp;
}
}

// Function to convert from postfix to infix
void convert(char exp[]) {
int l, i, j = 0;
char tmp[20];
revstr(exp);
l = strlen(exp);

for (i = 0; i < 50; i++) {
stack[i] = '\0';
}

printf("\nThe Infix Expression is : ");
for (i = 0; i < l; i++) {
if (exp[i] == '+' || exp[i] == '-' || exp[i] == '*' || exp[i] == '/') {
push(exp[i]);
} else {
tmp[j++] = exp[i];
tmp[j++] = pop();
```

```
}
}

tmp[j] = exp[top--];
revstr(tmp);
puts(tmp);
}

// Main Function
int main() {
char exp[50];
clrscr();
// Taking postfix expression
printf("\nEnter the Postfix Expression : ");
gets(exp);

// Calling the function to convert the expression
convert(exp);
getch();
return 0;
}
```

7.Write a C program to implement the primitive operations of circular queue

```
#include <stdio.h>

#include <conio.h>

#define MAX_SIZE 5

// Structure to represent the circular queue

struct CircularQueue {

int items[MAX_SIZE];

int front, rear;

};

// Function to initialize the circular queue

void initializeQueue(struct CircularQueue *queue) {

queue->front = -1;

queue->rear = -1;

}

// Function to check if the queue is empty

int isEmpty(struct CircularQueue *queue) {

return (queue->front == -1 && queue->rear == -1);

}

// Function to check if the queue is full

int isFull(struct CircularQueue *queue) {

return (queue->rear + 1) % MAX_SIZE == queue->front;

}

// Function to add an element to the queue (enqueue)

void Addq(struct CircularQueue *queue, int value) {

if (isFull(queue)) {
```

```
printf("\nQueue is full. Cannot add element.\n");

} else {

if (isEmpty(queue)) {

queue->front = 0;

}

queue->rear = (queue->rear + 1) % MAX_SIZE;

queue->items[queue->rear] = value;

printf("\n%d added to the queue.\n", value);

}

}

// Function to remove an element from the queue (dequeue)

void Delq(struct CircularQueue *queue) {

if (isEmpty(queue)) {

printf("\nQueue is empty. Cannot delete element.\n");

} else {

printf("\n%d deleted from the queue.\n", queue->items[queue->front]);

if (queue->front == queue->rear) {

// Queue becomes empty after deletion

initializeQueue(queue);

} else {

queue->front = (queue->front + 1) % MAX_SIZE;

}

}

}

// Function to display the entire queue
```

```
void display(struct CircularQueue *queue) {
if (isEmpty(queue)) {
printf("\nQueue is empty.\n");
} else {
int i = queue->front;
printf("\nQueue elements: ");
do {
printf("%d ", queue->items[i]);
i = (i + 1) % MAX_SIZE;
} while (i != (queue->rear + 1) % MAX_SIZE);
printf("\n");
}
}
int main() {
struct CircularQueue queue;
initializeQueue(&queue);
int choice, value;
do {
printf("\nCircular Queue Operations:\n");
printf("1. Addq (Enqueue)\n");
printf("2. Delq (Dequeue)\n");
printf("3. Display\n");
printf("4. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
```

```
switch (choice) {

case 1:

printf("\nEnter the element to be added to the queue: ");

scanf("%d", &value);

Addq(&queue, value);

break;

case 2:

Delq(&queue);

break;

case 3:

display(&queue);

break;

case 4:

printf("\nExiting the program.\n");

break;

default:

printf("\nInvalid choice. Please enter a valid option.\n");

}

} while (choice != 4);

getch();

return 0;

}
```

8. Write a C program to implement the primitive operations of double ended queue

```
#include <stdio.h>

#include <conio.h>

#define MAX_SIZE 5

// Structure to represent the double-ended queue

struct Deque {

int items[MAX_SIZE];

int front, rear;

};

// Function to initialize the deque

void initializeDeque(struct Deque *deque) {

deque->front = -1;

deque->rear = -1;

}

// Function to check if the deque is empty

int isEmpty(struct Deque *deque) {

return (deque->front == -1 && deque->rear == -1);

}

// Function to check if the deque is full

int isFull(struct Deque *deque) {

return (deque->rear + 1) % MAX_SIZE == deque->front;

}

// Function to add an element to the front of the deque

void addFront(struct Deque *deque, int value) {

if (isFull(deque)) {
```

```
printf("\nDeque is full. Cannot add element to the front.\n");
} else {
if (isEmpty(deque)) {
deque->front = 0;
deque->rear = 0;
} else {
deque->front = (deque->front - 1 + MAX_SIZE) % MAX_SIZE;
}
deque->items[deque->front] = value;
printf("\n%d added to the front of the deque.\n", value);
}
}
// Function to add an element to the rear of the deque
void addRear(struct Deque *deque, int value) {
if (isFull(deque)) {
printf("\nDeque is full. Cannot add element to the rear.\n");
} else {
if (isEmpty(deque)) {
deque->front = 0;
deque->rear = 0;
} else {
deque->rear = (deque->rear + 1) % MAX_SIZE;
}
deque->items[deque->rear] = value;
printf("\n%d added to the rear of the deque.\n", value);
```

```
}
}
// Function to remove an element from the front of the deque
void removeFront(struct Deque *deque) {
if (isEmpty(deque)) {
printf("\nDeque is empty. Cannot remove element from the front.\n");
} else {
printf("\n%d removed from the front of the deque.\n", deque->items[deque->front]);
if (deque->front == deque->rear) {
// Deque becomes empty after removal
initializeDeque(deque);
} else {
deque->front = (deque->front + 1) % MAX_SIZE;
}
}
}
// Function to remove an element from the rear of the deque
void removeRear(struct Deque *deque) {
if (isEmpty(deque)) {
printf("\nDeque is empty. Cannot remove element from the rear.\n");
} else {
printf("\n%d removed from the rear of the deque.\n", deque->items[deque->rear]);
if (deque->front == deque->rear) {
// Deque becomes empty after removal
initializeDeque(deque);
```

```
} else {
deque->rear = (deque->rear - 1 + MAX_SIZE) % MAX_SIZE;
}
}
}
// Function to display the entire deque
void display(struct Deque *deque) {
if (isEmpty(deque)) {
printf("\nDeque is empty.\n");
} else {
int i = deque->front;
printf("\nDeque elements: ");
do {
printf("%d ", deque->items[i]);
i = (i + 1) % MAX_SIZE;
} while (i != (deque->rear + 1) % MAX_SIZE);
printf("\n");
}
}
int main() {
struct Deque deque;
initializeDeque(&deque);
int choice, value;
do {
printf("\nDouble-Ended Queue (Deque) Operations:\n");
```

```
printf("1. Add to Front\n");

printf("2. Add to Rear\n");

printf("3. Remove from Front\n");

printf("4. Remove from Rear\n");

printf("5. Display\n");

printf("6. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("\nEnter the element to be added to the front: ");

scanf("%d", &value);

addFront(&deque, value);

break;

case 2:

printf("\nEnter the element to be added to the rear: ");

scanf("%d", &value);

addRear(&deque, value);

break;

case 3:

removeFront(&deque);

break;

case 4:

removeRear(&deque);

break;
```

```
case 5:

display(&deque);

break;

case 6:

printf("\nExiting the program.\n");

break;

default:

printf("\nInvalid choice. Please enter a valid option.\n");

}

} while (choice != 6);

getch();

return 0;

}
```

9.Write a C program to implement the primitive operations of priority queue

```
#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#define MAX_SIZE 100

// Structure to represent the priority queue

struct PriorityQueue {

int data[MAX_SIZE];

int priority[MAX_SIZE];

int front, rear;

};

// Function to create a new priority queue

struct PriorityQueue* createPriorityQueue() {

struct PriorityQueue* pq = (struct PriorityQueue*)malloc(sizeof(struct PriorityQueue));

pq->front = -1;

pq->rear = -1;

return pq;

}

// Function to check if the priority queue is empty

int isEmpty(struct PriorityQueue* pq) {

return pq->front == -1;

}

// Function to check if the priority queue is full

int isFull(struct PriorityQueue* pq) {

return pq->rear == MAX_SIZE - 1;

}

// Function to insert a new node with given priority into the priority queue

void enqueue(struct PriorityQueue* pq, int d, int p) {

if (isFull(pq)) {

printf("Queue is full. Cannot enqueue.\n");
```

```
return;
}
if (isEmpty(pq)) {
pq->front = 0;  // Adjust front pointer if the queue was empty
}
// Increment the rear index
pq->rear++;
// Add data and priority to the arrays
pq->data[pq->rear] = d;
pq->priority[pq->rear] = p;
}
// Function to display the elements of the priority queue
void display(struct PriorityQueue* pq) {
int i;
if (isEmpty(pq)) {
printf("Priority Queue is empty.\n");
return;
}
printf("Priority Queue: ");
for (i = pq->front; i <= pq->rear; ++i) {
printf("(%d, %d) ", pq->data[i], pq->priority[i]);
}
printf("\n");
}
int main() {
struct PriorityQueue* pq = createPriorityQueue();
int i;
// Get user input and enqueue elements with priorities
int data, priority;
for (i = 0; i < 3; ++i) {
printf("Enter data and priority (e.g., 10 2): ");
```

```
scanf("%d %d", &data, &priority);
enqueue(pq, data, priority);
}
// Display the priority queue
display(pq);
// Clear screen before exiting
clrscr();
getch();
return 0;
}
```

10.Write a C program to check string with parenthesis and validate it using stack

```
#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#define MAX_SIZE 100

// Structure to represent a stack

struct Stack {

char items[MAX_SIZE];

int top;

};

// Function to initialize an empty stack

void initialize(struct Stack* stack) {

stack->top = -1;

}

// Function to check if the stack is empty

int isEmpty(struct Stack* stack) {

return stack->top == -1;

}

// Function to push an item onto the stack

void push(struct Stack* stack, char item) {

if (stack->top == MAX_SIZE - 1) {

printf("Stack Overflow\n");

exit(1);

}

stack->items[++stack->top] = item;

}

// Function to pop an item from the stack

char pop(struct Stack* stack) {

if (isEmpty(stack)) {

printf("Stack Underflow\n");
```

```
exit(1);
}
return stack->items[stack->top--];
}
// Function to check if a string with parentheses is valid
int isValidString(char str[]) {
struct Stack stack;
initialize(&stack);
int i;
for (i = 0; str[i] != '\0'; i++) {
if (str[i] == '(' || str[i] == '[' || str[i] == '{') {
push(&stack, str[i]);
} else if (str[i] == ')' || str[i] == ']' || str[i] == '}') {
if (isEmpty(&stack)) {
return 0;  // Unmatched closing parenthesis
}
char popped = pop(&stack);
if ((str[i] == ')' && popped != '(') ||
(str[i] == ']' && popped != '[') ||
(str[i] == '}' && popped != '{')) {
return 0;  // Mismatched parenthesis
}
}
}
return isEmpty(&stack);  // Stack should be empty for a valid string
}
int main() {
char str[MAX_SIZE];
// Input string from the user
printf("Enter a string with parentheses: ");
gets(str);
```

```
// Check if the string is valid
if (isValidString(str)) {
printf("The string is valid.\n");
} else {
printf("The string is not valid.\n");
}
// Clear screen before exiting
getch();  // Wait for a key press
clrscr();  // Clear screen
return 0;
}
```

11.Write a C program to generate the Fibonacci series using stack

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define MAX_SIZE 100
// Structure to represent a stack
struct Stack {
int items[MAX_SIZE];
int top;
};
// Function to initialize an empty stack
void initialize(struct Stack* stack) {
stack->top = -1;
}
// Function to check if the stack is empty
int isEmpty(struct Stack* stack) {
return stack->top == -1;
}
// Function to push an item onto the stack
void push(struct Stack* stack, int item) {
if (stack->top == MAX_SIZE - 1) {
printf("Stack Overflow\n");
getch();  // Wait for a key press before exiting
exit(1);
}
stack->items[++stack->top] = item;
}

// Function to pop an item from the stack
int pop(struct Stack* stack) {
if (isEmpty(stack)) {
```

```
printf("Stack Underflow\n");
getch();  // Wait for a key press before exiting
exit(1);
}
return stack->items[stack->top--];
}  // Function to generate Fibonacci series using a stack
void generateFibonacci(int n) {
struct Stack fibStack;
initialize(&fibStack);
printf("Fibonacci Series: ");
int a = 0, b = 1;
int i;
for (i = 0; i < n; ++i) {
push(&fibStack, a);
int next = a + b;
a = b;
b = next;
}
while (!isEmpty(&fibStack)) {
printf("%d ", pop(&fibStack));
}
printf("\n");
}
int main() {
int n;  // Input from the user
clrscr();  // Clear screen
printf("Enter the number of terms in the Fibonacci series: ");
scanf("%d", &n);   // Generate and display the Fibonacci series
generateFibonacci(n);  // Wait for a key press before exiting
getch();  // Wait for a key press
clrscr();  // Clear screen
```

```
return 0;
}
```

12.Write a C program to implement the decimal to binary conversion using stack

```
#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#define MAX_SIZE 100  // Structure to represent a stack

struct Stack {

int items[MAX_SIZE];

int top;

};  // Function to initialize an empty stack

void initialize(struct Stack* stack) {

stack->top = -1;

}  //   Function to check if the stack is empty

int isEmpty(struct Stack* stack) {

return stack->top == -1;

}// Function to push an item onto the stack

void push(struct Stack* stack, int item) {

if (stack->top == MAX_SIZE - 1) {

printf("Stack Overflow\n");

getch();  // Wait for a key press before exiting

exit(1);

}

stack->items[++stack->top] = item;

}  // Function to pop an item from the stack

int pop(struct Stack* stack) {

if (isEmpty(stack)) {

printf("Stack Underflow\n");

getch();  // Wait for a key press before exiting

exit(1);

}

return stack->items[stack->top--];
```

```
} // Function to convert decimal to binary using a stack
void decimalToBinary(int decimal) {
struct Stack binaryStack;
initialize(&binaryStack);
printf("Binary Equivalent: ");
while (decimal > 0) {
int remainder = decimal % 2;
push(&binaryStack, remainder);
decimal /= 2;
}
while (!isEmpty(&binaryStack)) {
printf("%d", pop(&binaryStack));
}
printf("\n");
}
int main() {
int decimal;   // Input from the user
clrscr();  // Clear screen
printf("Enter a decimal number: ");
scanf("%d", &decimal);   // Convert and display the binary equivalent
decimalToBinary(decimal);   // Wait for a key press before exiting
getch();  // Wait for a key press
clrscr();  // Clear screen
return 0;
}
```

13. Write a C program to implement Josephus problem using stack

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
struct Stack {
int top;
int capacity;
int* array;
};
struct Stack* createStack(int capacity) {
struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
stack->capacity = capacity;
stack->top = -1;
stack->array = (int*)malloc(capacity * sizeof(int));
return stack;
}
int isFull(struct Stack* stack) {
return stack->top == stack->capacity - 1;
}
int isEmpty(struct Stack* stack) {
return stack->top == -1;
}
void push(struct Stack* stack, int item) {
if (isFull(stack)) {
printf("Stack Overflow\n");
return;
}
stack->array[++stack->top] = item;
}
int pop(struct Stack* stack) {
if (isEmpty(stack)) {
printf("Stack Underflow\n");
return -1;
}
return stack->array[stack->top--];
}
int josephus(int n, int k) {
struct Stack* stack = createStack(n);
for (int i = n; i >= 1; i--) {
push(stack, i);
}
int count = 0;
int survivor = 0;
while (!isEmpty(stack)) {
int popped = pop(stack);
count++;
if (count == k-1) {
count = 0;
survivor = popped-k;
```

```
} else {
push(stack, popped);
}
}
return survivor;
}
int main() {
int n, k;
printf("Enter the number of people (n): ");
scanf("%d", &n);
printf("Enter the counting interval (k): ");
scanf("%d", &k);
int survivor = josephus(n, k);
printf("The survivor is at position %d.\n", survivor);
return 0;
}
```

14 .Write a C program to implement Josephus problem using linked list

```
#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(struct Node));

newNode->data = data;

newNode->next = NULL;

return newNode;

}

struct Node* buildCircularLinkedList(int n) {

if (n <= 0) {

return NULL;

}

struct Node* head = createNode(1);

struct Node* current = head;

for (int i = 2; i <= n; i++) {

current->next = createNode(i);

current = current->next;

}

current->next = head; // Make it circular

return head;

}

void printList(struct Node* head) {

if (!head) {

return;

}

struct Node* current = head;
```

```
do {
printf("%d ", current->data);
current = current->next;
} while (current != head);
printf("\n");
}
int josephus(struct Node** head, int k) {
if (!(*head)) {
return -1;
}
struct Node* current = *head;
struct Node* prev = NULL;
while (current->next != current) {
// Find the k-th node
for (int i = 1; i < k; i++) {
prev = current;
current = current->next;
}
// Remove the k-th node
if (prev != NULL) {
prev->next = current->next;
} else {
// If prev is NULL, it means we are removing the head node
*head = current->next;
}
struct Node* temp = current;
current = current->next;
free(temp);
}
int survivor = current->data;
free(current);
```

```
*head = NULL;
return survivor;
}
int main() {
int n, k;
printf("Enter the number of people (n): ");
scanf("%d", &n);
printf("Enter the counting interval (k): ");
scanf("%d", &k);
struct Node* head = buildCircularLinkedList(n);
printf("Initial List: ");
printList(head);
int survivor = josephus(&head, k);
printf("The survivor is at position %d.\n", survivor);
return 0;
}
```

15.Write a C program to perform following operations on singly linked list.
Insertion at beginning

Deletion at middle

Display

```
#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node* next;

};

struct Node* createNode(int value) {

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

if (newNode) {

newNode->data = value;

newNode->next = NULL;

}

return newNode;

}

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

struct Node* newNode = createNode(value);

if (newNode) {

newNode->next = *head;

*head = newNode;

printf("Inserted %d at the beginning.\n", value);

} else {

printf("Memory allocation failed.\n");

}

}

void deleteAtMiddle(struct Node** head) {

if (*head == NULL || (*head)->next == NULL) {

printf("List is empty or has only one element, cannot delete from middle.\n");
```

```
return;
}
struct Node* slow = *head;
struct Node* fast = *head;
struct Node* prev = NULL;
while (fast != NULL && fast->next != NULL) {
prev = slow;
slow = slow->next;
fast = fast->next->next;
}
if (prev != NULL) {
prev->next = slow->next;
free(slow);
printf("Deleted middle element.\n");
} else {
printf("List has only one element, cannot delete from middle.\n");
}
}
void display(struct Node* head) {
printf("List: ");
while (head != NULL) {
printf("%d -> ", head->data);
head = head->next;
}
printf("NULL\n");
}
void freeList(struct Node* head) {
struct Node* temp;
while (head != NULL) {
temp = head;
head = head->next;
```

```
free(temp);
}
}
int main() {
struct Node* head = NULL;
int value, choice;
do {
printf("\n1. Insert at the beginning\n");
printf("2. Delete at the middle\n");
printf("3. Display\n");
printf("4. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:
printf("Enter the value to insert: ");
scanf("%d", &value);
insertAtBeginning(&head, value);
break;
case 2:
deleteAtMiddle(&head);
break;
case 3:
display(head);
break;
case 4:
printf("Exiting the program.\n");
break;
default:
printf("Invalid choice. Please enter a valid option.\n");
}
```

```
} while (choice != 4);
// Free the memory
freeList(head);
return 0;
}
```

16. Write a C program to perform the following operations on a singly linked list.

Insertion at middle

Deletion at end

Display

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
int data;
struct Node* next;
};
struct Node* createNode(int value) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
if (newNode) {
newNode->data = value;
newNode->next = NULL;
}
return newNode;
}
void insertAtMiddle(struct Node** head, int value) {
struct Node* newNode = createNode(value);
if (newNode) {
if (*head == NULL) {
// First insertion when the list is empty
*head = newNode;
printf("Inserted %d at the middle.\n", value);
return;
}
struct Node* slow = *head;
struct Node* fast = *head;
struct Node* prev = NULL;
```

```
while (fast != NULL && fast->next != NULL) {
prev = slow;
slow = slow->next;
fast = fast->next->next;
}
if (prev != NULL) {
prev->next = newNode;
} else {
// Insert at the beginning if the list has only one element
newNode->next = *head;
*head = newNode;
}
newNode->next = slow;
printf("Inserted %d at the middle.\n", value);
} else {
printf("Memory allocation failed.\n");
}
}
void deleteAtEnd(struct Node** head) {
if (*head == NULL) {
printf("List is empty, cannot delete from end.\n");
return;
}
struct Node* temp = *head;
struct Node* prev = NULL;
while (temp->next != NULL) {
prev = temp;
temp = temp->next;
}
if (prev != NULL) {
prev->next = NULL;
```

```
free(temp);
printf("Deleted from the end.\n");
} else {
free(temp);
*head = NULL;
printf("List is empty after deletion.\n");
}
}
void display(struct Node* head) {
printf("List: ");
while (head != NULL) {
printf("%d -> ", head->data);
head = head->next;
}
printf("NULL\n");
}
void freeList(struct Node* head) {
struct Node* temp;
while (head != NULL) {
temp = head;
head = head->next;
free(temp);
}
}
int main() {
struct Node* head = NULL;
int value, choice;
do {
printf("\n1. Insert at the middle\n");
printf("2. Delete at the end\n");
printf("3. Display\n");
```

```
printf("4. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:
printf("Enter the value to insert at the middle: ");
scanf("%d", &value);
insertAtMiddle(&head, value);
break;
case 2:
deleteAtEnd(&head);
break;
case 3:
display(head);
break;
case 4:
printf("Exiting the program.\n");
break;
default:
printf("Invalid choice. Please enter a valid option.\n");
}
} while (choice != 4);
// Free the memory
freeList(head);
return 0;
}
```

17 . C program to perform the following operations on a singly linked list.

Insertion at end

Deletion at beginning

Display

```
#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node* next;

};

struct Node* createNode(int value) {

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

if (newNode) {

newNode->data = value;

newNode->next = NULL;

}

return newNode;

}

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

struct Node* newNode = createNode(value);

if (newNode) {

if (*head == NULL) {

// If the list is empty, make the new node the head

*head = newNode;

} else {

struct Node* temp = *head;

// Traverse to the end of the list

while (temp->next != NULL) {

temp = temp->next;

}
```

```
// Insert the new node at the end
temp->next = newNode;
}
printf("Inserted %d at the end.\n", value);
} else {
printf("Memory allocation failed.\n");
}
}
void deleteAtBeginning(struct Node** head) {
if (*head == NULL) {
printf("List is empty, cannot delete from beginning.\n");
return;
}
struct Node* temp = *head;
*head = (*head)->next;
free(temp);
printf("Deleted from the beginning.\n");
}
void display(struct Node* head) {
printf("List: ");
while (head != NULL) {
printf("%d -> ", head->data);
head = head->next;
}
printf("NULL\n");
}
void freeList(struct Node* head) {
struct Node* temp;
while (head != NULL) {
temp = head;
head = head->next;
```

```
free(temp);
}
}
int main() {
struct Node* head = NULL;
int value, choice;
do {
printf("\n1. Insert at the end\n");
printf("2. Delete at the beginning\n");
printf("3. Display\n");
printf("4. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:
printf("Enter the value to insert at the end: ");
scanf("%d", &value);
insertAtEnd(&head, value);
break;
case 2:
deleteAtBeginning(&head);
break;
case 3:
display(head);
break;
case 4:
printf("Exiting the program.\n");
break;
default:
printf("Invalid choice. Please enter a valid option.\n");
}
```

```
} while (choice != 4);
// Free the memory
freeList(head);
return 0;
}
```

18.Write a C program to perform the following operations on single variable polynomials using singly linked list.
Accept a sorted polynomial

Addition of two polynomials

Display

```
#include <stdio.h>

#include <stdlib.h>

struct Term {

int coef, exp;

struct Term *next;

};

struct Term *createTerm(int coef, int exp) {

struct Term *term = (struct Term *)malloc(sizeof(struct Term));

term->coef = coef;

term->exp = exp;

term->next = NULL;

return term;

}

void insertTerm(struct Term **head, int coef, int exp) {

struct Term *term = createTerm(coef, exp);

if (*head == NULL || exp > (*head)->exp) {

term->next = *head;

*head = term;

} else {

struct Term *current = *head;

while (current->next != NULL && exp < current->next->exp) {

current = current->next;

}

term->next = current->next;

current->next = term;

}
```

```
}
void displayPolynomial(struct Term *head) {
while (head != NULL) {
printf("%dx^%d", head->coef, head->exp);
head = head->next;
if (head != NULL) {
printf(" + ");
}
}
printf("\n");
}
struct Term *addPolynomials(struct Term *poly1, struct Term *poly2) {
struct Term *result = NULL;
while (poly1 != NULL && poly2 != NULL) {
if (poly1->exp > poly2->exp) {
insertTerm(&result, poly1->coef, poly1->exp);
poly1 = poly1->next;
} else if (poly1->exp < poly2->exp) {
insertTerm(&result, poly2->coef, poly2->exp);
poly2 = poly2->next;
} else {
// Exponents are equal, add coefficients
insertTerm(&result, poly1->coef + poly2->coef, poly1->exp);
poly1 = poly1->next;
poly2 = poly2->next;
}
}
while (poly1 != NULL) {
insertTerm(&result, poly1->coef, poly1->exp);
poly1 = poly1->next;
}
```

```
while (poly2 != NULL) {
insertTerm(&result, poly2->coef, poly2->exp);
poly2 = poly2->next;
}
return result;
}
int main() {
struct Term *poly1 = NULL, *poly2 = NULL, *result = NULL;
int n, i, coef, exp;
printf("Enter the number of terms in the first polynomial: ");
scanf("%d", &n);
printf("Enter the terms for the first polynomial (sorted by exponent):\n");
for (i = 0; i < n; i++) {
printf("Coefficient: ");
scanf("%d", &coef);
printf("Exponent: ");
scanf("%d", &exp);
insertTerm(&poly1, coef, exp);
}
printf("Enter the number of terms in the second polynomial: ");
scanf("%d", &n);
printf("Enter the terms for the second polynomial (sorted by exponent):\n");
for (i = 0; i < n; i++) {
printf("Coefficient: ");
scanf("%d", &coef);
printf("Exponent: ");
scanf("%d", &exp);
insertTerm(&poly2, coef, exp);
}
printf("First polynomial: ");
displayPolynomial(poly1);
```

```c
printf("Second polynomial: ");
displayPolynomial(poly2);
result = addPolynomials(poly1, poly2);
printf("Resultant Polynomial (Sum): ");
displayPolynomial(result);
// Free memory
free(poly1);
free(poly2);
free(result);
return 0;
}
```

19.Write a C program to accept two sorted single linked lists and merge them in a single linked list in such a way that the resultant linked list will be a sorted one

```
#include <stdio.h>
#include<stdlib.h>
struct Term{
struct Term *next;
int data;
};
void insert(struct Term **head,int data){
struct Term term=(struct Term)malloc(sizeof(struct Term));
term->data=data;
term->next=NULL;
if(*head==NULL){
*head=term;
return;
}
struct Term *current=*head;
while(current->next!=NULL){
current=current->next;
};
current->next=term;
}
struct Term *mergesortlist(struct Term *list1,struct Term *list2){
struct Term *mergelist=NULL;
while(list1!=NULL && list2!=NULL){
if(list1->data < list2->data){
insert(&mergelist,list1->data);
list1=list1->next;
}else{
insert(&mergelist,list2->data);
```

```
list2=list2->next;
}
}
while(list1!=NULL){
insert(&mergelist,list1->data);
list1=list1->next;
}
while(list2!=NULL){
insert(&mergelist,list2->data);
list2=list2->next;
}
return mergelist;
}
void display(struct Term *head){
struct Term *current=head;
while(current!=NULL){
printf("%d ->",current->data);
current=current->next;
} printf("NULL\n");
}
int main(){
struct Term *list1=NULL;
struct Term *list2=NULL;
struct Term *mergelist=NULL;
int i,n1,n2,data;
printf("enter the number of elements in 1st list");
scanf("%d",&n1);
printf("the elements should be in asceneding order: \n");
for(i=0;i<n1;i++){
scanf("%d",&data);
insert(&list1,data);
```

```
}
printf("enter the number of elements in 2nd list");
scanf("%d",&n2);
printf("the elements should be in asceneding order: \n");
for(i=0;i<n2;i++){
scanf("%d",&data);
insert(&list2,data);
}
printf("first sorted lisr\n");
display(list1);
printf("2nd sorted lisr\n");
display(list2);
mergelist=mergesortlist(list1,list2);
printf("merged sorted list:");
display(mergelist);
return 0;
}
```

20.Postorder Traversal, Preorder Traversal, Inorder Traversal.

```
#include <stdio.h>

#include <conio.h>

#include <alloc.h>

struct Node {

int data;

struct Node* left;

struct Node* right;

};

struct Node* createNode(int data) {

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

newNode->data = data;

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

return newNode;

}

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

if (root == NULL)

return createNode(data);

struct Node* temp = root;

struct Node* parent = NULL;

while (temp != NULL) {

parent = temp;

if (data < temp->data)

temp = temp->left;

else

temp = temp->right;

}

if (data < parent->data)

parent->left = createNode(data);

else

parent->right = createNode(data);
```

```
return root;
}
void postorderTraversal(struct Node* root) {
if (root == NULL)
return;
struct Node* stack[100];
int top = -1;
struct Node* prev = NULL;
do {
while (root != NULL) {
stack[++top] = root;
root = root->left;
}
while (root == NULL && top != -1) {
root = stack[top];
if (root->right == NULL || root->right == prev) {
printf("%d ", root->data);
top--;
prev = root;
root = NULL;
} else {
root = root->right;
}
}
} while (top != -1);
}
void preorderTraversal(struct Node* root) {
if (root == NULL)
return;
struct Node* stack[100];
int top = -1;
```

```
stack[++top] = root;
while (top >= 0) {
struct Node* node = stack[top--];
printf("%d ", node->data);
if (node->right != NULL)
stack[++top] = node->right;
if (node->left != NULL)
stack[++top] = node->left;
}
}
void inorderTraversal(struct Node* root) {
if (root == NULL)
return;
struct Node* stack[100];
int top = -1;
while (root != NULL || top != -1) {
while (root != NULL) {
stack[++top] = root;
root = root->left;
}
if (top != -1) {
root = stack[top--];
printf("%d ", root->data);
root = root->right;
}
}
}
int main() {
struct Node* root = NULL;
int choice, data;
do {
```

```
printf("\n1. Insert Node\n");
printf("2. Postorder Traversal\n");
printf("3. Preorder Traversal\n");
printf("4. Inorder Traversal\n");
printf("5. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:
printf("Enter data to insert: ");
scanf("%d", &data);
root = insertNode(root, data);
break;
case 2:
printf("Postorder Traversal: ");
postorderTraversal(root);
printf("\n");
break;
case 3:
printf("Preorder Traversal: ");
preorderTraversal(root);
printf("\n");
break;
case 4:
printf("Inorder Traversal: ");
inorderTraversal(root);
printf("\n");
break;
case 5:
printf("Exiting program.\n");
break;
```

```
default:
printf("Invalid choice. Please enter a valid option.\n");
}
} while (choice != 5);
getch();
return 0;
}
```

21.Copy a tree, 2. Equality of two trees, 3. Delete a node from a tree.

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
struct Node {
int data;
struct Node* left;
struct Node* right;
};
struct Node* createNode(int data) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = data;
newNode->left = newNode->right = NULL;
return newNode;
}
struct Node* insertNode(struct Node* root, int data) {
if (root == NULL)
return createNode(data);
struct Node* temp = root;
struct Node* parent = NULL;
while (temp != NULL) {
parent = temp;
if (data < temp->data)
temp = temp->left;
else
temp = temp->right;
}
if (data < parent->data)
parent->left = createNode(data);
else
parent->right = createNode(data);
```

```
return root;
}
void inorderTraversal(struct Node* root) {
if (root == NULL)
return;
struct Node* stack[100];
int top = -1;
while (root != NULL || top != -1) {
while (root != NULL) {
stack[++top] = root;
root = root->left;
}
if (top != -1) {
root = stack[top--];
printf("%d ", root->data);
root = root->right;
}
}
}
struct Node* copyTree(struct Node* root) {
if (root == NULL)
return NULL;
struct Node* newRoot = createNode(root->data);
struct Node* stack[100];
struct Node* newStack[100];
int top = -1;
stack[++top] = root;
newStack[++top] = newRoot;
while (top >= 0) {
struct Node* node = stack[top];
struct Node* newNode = newStack[top--];
```

```
if (node->right != NULL) {
stack[++top] = node->right;
newNode->right = createNode(node->right->data);
newStack[++top] = newNode->right;
}
if (node->left != NULL) {
stack[++top] = node->left;
newNode->left = createNode(node->left->data);
newStack[++top] = newNode->left;
}
}
return newRoot;
}
int areTreesEqual(struct Node* root1, struct Node* root2) {
if (root1 == NULL && root2 == NULL)
return 1;
if (root1 == NULL || root2 == NULL)
return 0;
struct Node* stack1[100];
int top1 = -1;
struct Node* stack2[100];
int top2 = -1;
while (root1 != NULL || top1 != -1) {
while (root1 != NULL) {
stack1[++top1] = root1;
root1 = root1->left;
}
while (root2 != NULL) {
stack2[++top2] = root2;
root2 = root2->left;
}
```

```
if (top1 != -1 && top2 != -1) {
root1 = stack1[top1--];
root2 = stack2[top2--];
if (root1->data != root2->data)
return 0;
root1 = root1->right;
root2 = root2->right;
} else {
return (top1 == -1 && top2 == -1);
}
}
return 1;
}
struct Node* deleteNode(struct Node* root, int key) {
struct Node* parent = NULL;
struct Node* current = root;
while (current != NULL && current->data != key) {
parent = current;
if (key < current->data)
current = current->left;
else
current = current->right;
}
if (current == NULL) {
printf("Node with key %d not found.\n", key);
return root;
}
// Case 1: Node with only one child or no child
if (current->left == NULL) {
struct Node* temp = current->right;
if (parent == NULL)
```

```
return temp; // Current is the root
if (current == parent->left)
parent->left = temp;
else
parent->right = temp;
free(current);
} else if (current->right == NULL) {
struct Node* temp = current->left;
if (parent == NULL)
return temp; // Current is the root
if (current == parent->left)
parent->left = temp;
else
parent->right = temp;
free(current);
}
// Case 2: Node with two children
else {
struct Node* successor = current->right;
struct Node* successorParent = NULL;
while (successor->left != NULL) {
successorParent = successor;
successor = successor->left;
}
if (successorParent != NULL)
successorParent->left = successor->right;
else
current->right = successor->right;
current->data = successor->data;
free(successor);
}
```

```
printf("Node with key %d deleted.\n", key);
return root;
}
int main() {
struct Node* root = NULL;
struct Node* copyRoot = NULL;
int choice, data;
do {
printf("\n1. Insert Node\n");
printf("2. Inorder Traversal\n");
printf("3. Copy a Tree\n");
printf("4. Check Equality of Two Trees\n");
printf("5. Delete a Node\n");
printf("6. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:
printf("Enter data to insert: ");
scanf("%d", &data);
root = insertNode(root, data);
break;
case 2:
printf("Inorder Traversal: ");
inorderTraversal(root);
printf("\n");
break;
case 3:
copyRoot = copyTree(root);
printf("Tree Copied.\n");
break;
```

```
case 4:
if (areTreesEqual(root, copyRoot))
printf("The trees are equal.\n");
else
printf("The trees are not equal.\n");
break;
case 5:
printf("Enter the key to delete: ");
scanf("%d", &data);
root = deleteNode(root, data);
break;
case 6:
printf("Exiting program.\n");
break;
default:
printf("Invalid choice. Please enter a valid option.\n");
}
} while (choice != 6);
getch();
return 0;
}
```

22.Insert a node in a tree, 2. Display the height of the tree, 3. Display a tree levelwise

```
#include <stdio.h>

#include <conio.h>

#include <alloc.h>

struct Node {

int data;

struct Node* left;

struct Node* right;

};

struct Node* createNode(int data) {

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

newNode->data = data;

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

return newNode;

}

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

if (root == NULL)

return createNode(data);

struct Node* temp = root;

struct Node* parent = NULL;

while (temp != NULL) {

parent = temp;

if (data < temp->data)

temp = temp->left;

else

temp = temp->right;

}

if (data < parent->data)

parent->left = createNode(data);

else
```

```
parent->right = createNode(data);
return root;
}
int heightOfTree(struct Node* root) {
if (root == NULL)
return 0;
struct Node* stack[100];
int top = -1;
int height = 0;
int maxDepth = 0;
stack[++top] = root;
while (top >= 0) {
struct Node* node = stack[top--];
if (node->right != NULL)
stack[++top] = node->right;
if (node->left != NULL)
stack[++top] = node->left;
if (node->left == NULL && node->right == NULL) {
// Leaf node, calculate depth
if (top + 1 > maxDepth)
maxDepth = top + 1;
}
}
return maxDepth;
}
void displayLevelWise(struct Node* root) {
if (root == NULL)
return;
struct Node* queue[100];
int front = -1;
int rear = -1;
```

```
queue[++rear] = root;
while (front != rear) {
struct Node* node = queue[++front];
printf("%d ", node->data);
if (node->left != NULL)
queue[++rear] = node->left;
if (node->right != NULL)
queue[++rear] = node->right;
}
}
int main() {
struct Node* root = NULL;
int choice, data;
do {
printf("\n1. Insert Node\n");
printf("2. Display Height of Tree\n");
printf("3. Display Tree Levelwise\n");
printf("4. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:
printf("Enter data to insert: ");
scanf("%d", &data);
root = insertNode(root, data);
break;
case 2:
printf("Height of the tree: %d\n", heightOfTree(root));
break;
case 3:
printf("Tree Levelwise: ");
```

```
displayLevelWise(root);
printf("\n");
break;
case 4:
printf("Exiting program.\n");
break;
default:
printf("Invalid choice. Please enter a valid option.\n");
}
} while (choice != 4);
getch();
return 0;
}
```

23.Display mirror image of a tree by creating new tree, 2. Display mirror image of a tree without creating new tree, 3. Display leaf nodes of a tree

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
struct Node {
int data;
struct Node* left;
struct Node* right;
};
struct Node* createNode(int data) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = data;
newNode->left = newNode->right = NULL;
return newNode;
}

struct Node* insertNode(struct Node* root, int data) {
if (root == NULL)
return createNode(data);
struct Node* temp = root;
struct Node* parent = NULL;
while (temp != NULL) {
parent = temp;
if (data < temp->data)
temp = temp->left;
else
temp = temp->right;
}
if (data < parent->data)
parent->left = createNode(data);
```

```
else
parent->right = createNode(data);
return root;
}
void displayMirrorImage(struct Node* root) {
if (root == NULL)
return;
struct Node* stack[100];
int top = -1;
stack[++top] = root;
while (top >= 0) {
struct Node* node = stack[top--];
printf("%d ", node->data);
if (node->left != NULL)
stack[++top] = node->left;
if (node->right != NULL)
stack[++top] = node->right;
}
}
void mirrorImageWithoutCreatingNew(struct Node* root) {
if (root == NULL)
return;
struct Node* stack[100];
int top = -1;
stack[++top] = root;
while (top >= 0) {
struct Node* node = stack[top--];
// Swap the left and right children
struct Node* temp = node->left;
node->left = node->right;
node->right = temp;
```

```
if (node->right != NULL)
stack[++top] = node->right;
if (node->left != NULL)
stack[++top] = node->left;
}
}
void displayLeafNodes(struct Node* root) {
if (root == NULL)
return;
struct Node* stack[100];
int top = -1;
stack[++top] = root;
while (top >= 0) {
struct Node* node = stack[top--];
if (node->right != NULL)
stack[++top] = node->right;
if (node->left != NULL)
stack[++top] = node->left;
if (node->left == NULL && node->right == NULL)
printf("%d ", node->data);
}
}
int main() {
struct Node* root = NULL;
int choice, data;
do {
printf("\n1. Insert Node\n");
printf("2. Display Mirror Image (Creating New Tree)\n");
printf("3. Display Mirror Image (Without Creating New Tree)\n");
printf("4. Display Leaf Nodes\n");
printf("5. Exit\n");
```

```
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:
printf("Enter data to insert: ");
scanf("%d", &data);
root = insertNode(root, data);
break;
case 2:
printf("Mirror Image (Creating New Tree): ");
displayMirrorImage(root);
printf("\n");
break;
case 3:
printf("Mirror Image (Without Creating New Tree): ");
mirrorImageWithoutCreatingNew(root);
displayMirrorImage(root);
printf("\n");
break;
case 4:
printf("Leaf Nodes: ");
displayLeafNodes(root);
printf("\n");
break;
case 5:
printf("Exiting program.\n");
break;
default:
printf("Invalid choice. Please enter a valid option.\n");
}
} while (choice != 5);
```

```
getch();
return 0;
}
```

25. Write a C program to display the leaf nodes level-wise

```
#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in a binary tree

struct Node {

int data;

struct Node* left;

struct Node* right;

};

struct Node* queue[100];

int front = -1, rear = -1;

// Function to create a new node

struct Node* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function to print leaf nodes level-wise

void printLeafNodesLevelWise(struct Node* root) {

if (root == NULL) {

return;

}

// Create a queue for level order traversal

queue[++rear] = root;

queue[++rear] = NULL; // Using NULL as a marker for the end of a level

printf("Binary Tree (Level Order):\n");

while (front < rear) {

struct Node* current = queue[++front];

if (current == NULL) {
```

```
if (front < rear) {
// If the current level is not finished, add a marker for the next level
queue[++rear] = NULL;
printf("\n");
}
} else {
// Display the data of the current node
printf("%d ", current->data);
// Enqueue the left and right children if they exist
if (current->left != NULL) {
queue[++rear] = current->left;
}
if (current->right != NULL) {
queue[++rear] = current->right;
}
}
}
}
// Function to display leaf nodes
void displayLeafNodes(struct Node* root) {
if (root == NULL) {
return;
}
if (root->left == NULL && root->right == NULL) {
// If the current node is a leaf node, print its data
printf("%d ", root->data);
}
// Recursively display leaf nodes in the left and right subtrees
displayLeafNodes(root->left);
displayLeafNodes(root->right);
}
```

```
int main() {
// Constructing a sample binary tree
struct Node* root = createNode(1);
root->left = createNode(2);
root->right = createNode(3);
root->left->left = createNode(4);
root->left->right = createNode(5);
root->right->left = createNode(6);
root->right->right = createNode(7);
root->right->right->left = createNode(8);
// Displaying the binary tree in level order
printLeafNodesLevelWise(root);
// Displaying leaf nodes
printf("\nLeaf nodes: ");
displayLeafNodes(root);
getch();
return 0;
}
```

26. Write a C program to perform all primitive operations of deletion of a node in a binary search tree

```
#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in a binary search tree

struct Node {

int data;

struct Node* left;

struct Node* right;

};

// Function to create a new node

struct Node* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function to insert a new node into the binary search tree

struct Node* insert(struct Node* root, int data) {

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

root->left = insert(root->left, data);

} else if (data > root->data) {

root->right = insert(root->right, data);

}

return root;

}

// Function to find the node with the minimum value in a given binary search tree

struct Node* findMinNode(struct Node* root) {
```

```
while (root->left != NULL) {
root = root->left;
}
return root;
}
// Function to delete a node with a given key from the binary search tree
struct Node* deleteNode(struct Node* root, int key) {
struct Node* temp;
if (root == NULL) {
return root;
}
if (key < root->data) {
root->left = deleteNode(root->left, key);
} else if (key > root->data) {
root->right = deleteNode(root->right, key);
} else {
// Node with only one child or no child
if (root->left == NULL) {
struct Node* temp = root->right;
free(root);
return temp;
} else if (root->right == NULL) {
struct Node* temp = root->left;
free(root);
return temp;
}
// Node with two children
temp = findMinNode(root->right);
root->data = temp->data;
root->right = deleteNode(root->right, temp->data);
}
```

```
return root;
}
// Function to perform an in-order traversal of the binary search tree
void inorderTraversal(struct Node* root) {
if (root != NULL) {
inorderTraversal(root->left);
printf("%d ", root->data);
inorderTraversal(root->right);
}
}
int main() {
struct Node* root = NULL;
int choice, data;
do {
printf("\n1. Insert\n2. Delete\n3. Display\n4. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:
printf("Enter data to insert: ");
scanf("%d", &data);
root = insert(root, data);
break;
case 2:
printf("Enter data to delete: ");
scanf("%d", &data);
root = deleteNode(root, data);
break;
case 3:
printf("Binary Search Tree: ");
inorderTraversal(root);
```

```
printf("\n");
break;
case 4:
printf("Exiting program.\n");
break;
default:
printf("Invalid choice.\n");
}
} while (choice != 4);
return 0;
}
```

27. Write a C program to perform all primitive operations of deletion of a node in a binary tree

```
#include <stdio.h>

#include <conio.h>

#include <alloc.h>

struct Node {

int data;

struct Node *left, *right;

};

struct Node *createNode(int data) {

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

newNode->data = data;

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

return newNode;

}

struct Node *insert(struct Node *root, int data) {

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

root->left = insert(root->left, data);

} else if (data > root->data) {

root->right = insert(root->right, data);

}

return root;

}

struct Node *findMin(struct Node *root) {

while (root->left != NULL) {

root = root->left;

}

return root;

}
```

```
struct Node *deleteNode(struct Node *root, int key) {
struct Node *temp;
if (root == NULL) {
return root;
}
if (key < root->data) {
root->left = deleteNode(root->left, key);
} else if (key > root->data) {
root->right = deleteNode(root->right, key);
} else {
// Node with only one child or no child
if (root->left == NULL) {
struct Node *temp = root->right;
free(root);
return temp;
} else if (root->right == NULL) {
struct Node *temp = root->left;
free(root);
return temp;
}
// Node with two children: Get the inorder successor (smallest
// in the right subtree)
temp = findMin(root->right);
// Copy the inorder successor's data to this node
root->data = temp->data;
// Delete the inorder successor
root->right = deleteNode(root->right, temp->data);
}
return root;
}
void deleteTree(struct Node *root) {
```

```
if (root != NULL) {
deleteTree(root->left);
deleteTree(root->right);
free(root);
}
}
void inOrderTraversal(struct Node *root) {
if (root != NULL) {
inOrderTraversal(root->left);
printf("%d ", root->data);
inOrderTraversal(root->right);
}
}
int main() {
struct Node *root = NULL;
int choice, keyToDelete, value;
int i;
do {
printf("\nBinary Tree Menu:\n");
printf("1. Insert Node\n");
printf("2. Display In-order Traversal\n");
printf("3. Delete Node\n");
printf("4. Delete Entire Tree\n");
printf("5. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:
printf("Enter the value to insert: ");
scanf("%d", &value);
root = insert(root, value);
```

```
printf("Node %d inserted.\n", value);
break;
case 2:
printf("In-order Traversal: ");
inOrderTraversal(root);
printf("\n");
break;
case 3:
printf("Enter the key to delete: ");
scanf("%d", &keyToDelete);
root = deleteNode(root, keyToDelete);
printf("Node %d deleted.\n", keyToDelete);
break;
case 4:
deleteTree(root);
printf("Tree deleted.\n");
break;
case 5:
printf("Exiting program.\n");
break;
default:
printf("Invalid choice. Please enter a valid option.\n");
break;
}
} while (choice != 5);
getch();
return 0;
}
```

28. Write a C program to implement a Threaded Binary tree and traverse it in-order

```
#include <stdio.h>

#include <stdlib.h>

int te = 0, b[40], size = 0;

struct node {

int data;

struct node* left;

struct node* right;

int lf;

int rf;

};

struct node* newNode(int data) {

struct node* new = (struct node*)malloc(sizeof(struct node));

new->data = data;

new->left = NULL;

new->right = NULL;

new->lf = 0;

new->rf = 0;

return new;

}

void insert(struct node** root, int data) {

struct node* n = newNode(data);

struct node* temp = *root;

struct node* parent = NULL;

if (*root == NULL) {

*root = n;

te++;

return;

}

while (1) {

parent = temp;
```

```
if (data < temp->data) {
if (temp->left == NULL || temp->lf == 1) {
temp->lf = 0;
temp->left = n;
n->rf = 1;
n->lf = 1;
n->right = temp;
n->left = root;
te++;
return;
} else {
temp = temp->left;
}
} else if (data > temp->data) {
if (temp->right == NULL || temp->rf == 1) {
temp->rf = 0;
temp->right = n;
n->rf = 1;
n->lf = 1;
n->left = root;
n->right = temp;
te++;
return;
} else {
temp = temp->right;
}
}
}
}
void right(struct node** root, int data) {
struct node* n = newNode(data);
```

```
struct node* temp = *root;
struct node* parent = NULL;
if (*root == NULL) {
*root = n;
te++;
return;
}
while (1) {
parent = temp;
if (data < temp->data) {
if (temp->left == NULL) {
temp->left = n;
n->rf = 1;
n->right = temp;
te++;
return;
} else {
temp = temp->left;
}
} else if (data > temp->data) {
if (temp->right == NULL || temp->rf == 1) {
temp->rf = 0;
temp->right = n;
n->rf = 1;
n->right = root;
te++;
return;
} else {
temp = temp->right;
}
}
```

```
}
}
void left(struct node** root, int data) {
struct node* n = newNode(data);
struct node* temp = *root;
struct node* parent = NULL;
if (*root == NULL) {
*root = n;
te++;
return;
}
while (1) {
parent = temp;
if (data < temp->data) {
if (temp->left == NULL || temp->lf == 1) {
temp->lf = 0;
temp->left = n;
n->lf = 1;
n->left = root;
te++;
return;
} else {
temp = temp->left;
}
} else if (data > temp->data) {
if (temp->right == NULL) {
temp->right = n;
n->lf = 1;
n->left = temp;
te++;
return;
```

```
} else {
temp = temp->right;
}
}
}
}
void inorder(struct node* root) {
if (root == NULL) {
return;
}
if (root->lf == 0) {
inorder(root->left);
}
b[size] = root->data;
size++;
printf("%d ", root->data);
if (root->rf == 0) {
inorder(root->right);
}
}
void postorder(struct node* root) {
if (root == NULL) {
return;
}
if (root->lf == 0) {
postorder(root->left);
}
if (root->rf == 0) {
postorder(root->right);
}
printf("%d ", root->data);
```

```
}
void preorder(struct node* root) {
if (root == NULL) {
return;
}
printf("%d ", root->data);
if (root->lf == 0) {
preorder(root->left);
}
if (root->rf == 0) {
preorder(root->right);
}
}
int main() {
struct node* root = NULL;
struct node* root1 = NULL;
struct node* root2 = NULL;
int no,i, p;
printf("Enter 5 numbers:\n");
for (i = 0; i < 5; ++i) {
scanf("%d", &no);
insert(&root, no);
right(&root1, no);
left(&root2, no);
}
printf("\nPreorder: ");
preorder(root);
printf("\nInorder: ");
inorder(root);
printf("\nPostorder: ");
postorder(root);
```

```
size = 0;
printf("\nRight: ");
inorder(root1);
printf("\nLeft: ");
size = 0;
inorder(root2);
return 0;
}
```

29. Write a C Program to implement Heap sort using Max heap in descending order

```
#include <stdio.h>

void heapify(int arr[],int n,int i){

int largest=i;

int left=2*i+1;

int right=2*i+2;

if(left<n && arr[left]>arr[largest])

largest=left;

if(right<n && arr[right]>arr[largest])

largest=right;

if(largest!=i){

int temp=arr[i];

arr[i]=arr[largest];

arr[largest]=temp;

heapify(arr,n,largest);

}}

void heapsort(int arr[],int n){

int i;

for(i=n/2-1;i>=0;i--){

heapify(arr,n,i);

}

for(i=n-1;i>0;i--){

int temp=arr[0];

arr[0]=arr[i];

arr[i]=temp;

heapify(arr,i,0);

}

}
```

30. Write a C Program to implement Heap sort using Min heap in ascending order

```
int main(){

int i;

int arr[]={13,12,8,67,7};

int n=sizeof(arr)/sizeof(arr[0]);

clrscr();

printf("original array:");

for(i=0;i<n;i++){

printf("%d",arr[i]);

printf("\n");

}

heapsort(arr,n);

printf("descending order");

for(i=n-1;i>=0;i--){

printf("%d ", arr[i]);

}

getch();

return 0;

}

#include <stdio.h>

void heapify(int arr[],int n,int i){

int smallest=i;

int left=2*i+1;

int right=2*i+2;

if(left<n && arr[left]<arr[smallest])

smallest=left;

if(right<n && arr[right]<arr[smallest])

smallest=right;


if(smallest!=i){

int temp=arr[i];
```

```
arr[i]=arr[smallest];
arr[smallest]=temp;
heapify(arr,n,smallest);
}}
void heapsort(int arr[],int n){
int i;
for(i=n/2-1;i>=0;i--){
heapify(arr,n,i);
}
for(i=n-1;i>0;i--){
int temp=arr[0];
arr[0]=arr[i];
arr[i]=temp;
heapify(arr,i,0);
}
}
int main(){
int i;
int arr[]={13,12,8,67,7};
int n=sizeof(arr)/sizeof(arr[0]);
clrscr();
printf("original array:");
for(i=0;i<n;i++){
printf("%d",arr[i]);
printf("\n");
}
heapsort(arr,n);
printf("ascending order");
for(i=n-1;i>=0;i--){
printf("%d ", arr[i]);
}
```

```
getch();
return 0;
}
```

31.Write a C Program to implement Kruskal’s algorithm for min. spanning tree, in which a graph is represented using an adjacency matrix.

```
#include <stdio.h>

#define MAX_EDGES 50  // Adjust the maximum number of edges as needed

struct edge {

int src, dest, weight;

};

struct Graph {

int nv, ne;

struct edge edges[MAX_EDGES];

};

void swap(struct edge *a, struct edge *b) {

struct edge temp = *a;

*a = *b;

*b = temp;

}

void bubble(struct edge *edges, int n) {

int i, j;

for (i = 0; i < n - 1; i++) {

for (j = 0; j < n - i - 1; j++) {

if (edges[j].weight > edges[j + 1].weight) {

swap(&edges[j], &edges[j + 1]);

}

}

}

}

int find(int parent[], int i) {

if (parent[i] == -1) {

// Do something when the condition is met (optional)

// For example, print the index

printf("Found: %d\n", i);
```

```
return i;
} else {
return find(parent, parent[i]);
}
}
void connect(int parent[], int x, int y) {
parent[x] = y;
}
void krushkal(struct Graph *graph) {
int nv = graph->nv;
struct edge result[MAX_EDGES];  // Adjust the maximum edges as needed
int e = 0;
int i = 0;
int v;
int parent[MAX_EDGES];
bubble(graph->edges, graph->ne);
// int parent[MAX_EDGES];  // Adjust the maximum edges as needed
for (v = 0; v < nv; v++) {
parent[v] = -1;
}
while (e < nv - 1 && i < graph->ne) {
struct edge next = graph->edges[i++];
int x = find(parent, next.src);
int y = find(parent, next.dest);
if (x != y) {
result[e++] = next;
connect(parent, x, y);
}
}
printf("Minimum Spanning Tree:\n");
for (i = 0; i < e; i++) {
```

```
printf("(%d, %d) Weight: %d\n", result[i].src, result[i].dest, result[i].weight);
}
}
int main() {
struct Graph graph;
int nv, ne, i;
printf("Enter the number of vertices: ");
scanf("%d", &nv);
if (nv <= 0 || nv >= MAX_EDGES) {
printf("Invalid number of vertices.\n");
return 1;
}
printf("Enter the number of edges: ");
scanf("%d", &ne);
if (ne <= 0 || ne >= MAX_EDGES) {
printf("Invalid number of edges.\n");
return 1;
}
graph.nv = nv;
graph.ne = ne;
printf("Enter edges (source, destination, weight):\n");
for (i = 0; i < ne; i++) {
printf("Edge %d: ", i + 1);
scanf("%d %d %d", &graph.edges[i].src, &graph.edges[i].dest, &graph.edges[i].weight);
}
krushkal(&graph);
return 0;
}
```

32.Write a C Program to implement Prim's algorithm for min. spanning tree, in which a graph is represented using an adjacency list.
// Prim's Algorithm in C

```
#include <stdio.h>

#include <limits.h>

#define INF INT_MAX  // Use INT_MAX instead of 9999999 for portability

// Number of vertices in the graph

#define V 5

// Create a 2D array of size 5x5

// for the adjacency matrix to represent the graph

int G[V][V] = {

{0, 9, 75, 0, 0},

{9, 0, 95, 19, 42},

{75, 95, 0, 51, 66},

{0, 19, 51, 0, 31},

{0, 42, 66, 31, 0}};

int main() {

int no_edge;  // Number of edges

int x, y, j, i;

// Create an array to track selected vertices

// selected will become true otherwise false

int selected[V];  // Use int instead of bool for boolean type

// Set selected false initially

for (i = 0; i < V; i++)

selected[i] = 0;

// Set the number of edges to 0

no_edge = 0;

// The number of edges in the minimum spanning tree will be

// always less than (V - 1), where V is the number of vertices in the graph

// Choose 0th vertex and make it true

selected[0] = 1;
```

```
// Print for edge and weight
printf("Edge : Weight\n");
while (no_edge < V - 1) {
// For every vertex in the set S, find all adjacent vertices
// , calculate the distance from the vertex selected at step 1.
// if the vertex is already in the set S, discard it otherwise
// choose another vertex nearest to the selected vertex at step 1.
int min = INF;
x = 0;
y = 0;
for (i = 0; i < V; i++) {
if (selected[i]) {
for (j = 0; j < V; j++) {
if (!selected[j] && G[i][j]) {  // Not in selected and there is an edge
if (min > G[i][j]) {
min = G[i][j];
x = i;
y = j;
}
}
}
}
}
printf("%d - %d : %d\n", x, y, G[x][y]);
selected[y] = 1;
no_edge++;
}
return 0;
}
```

33.Write a C Program to implement Dijkstra’s algorithm in which a graph is represented using an adjacency matrix.

// C program for Dijkstra's single source shortest path

// algorithm. The program is for adjacency matrix

// representation of the graph

```
#include <limits.h>
#include <stdio.h>
// Define boolean values
#define true 1
#define false 0
// Number of vertices in the graph
#define V 9
// A utility function to find the vertex with the minimum
// distance value, from the set of vertices not yet included
// in the shortest path tree
int minDistance(int dist[], int sptSet[])
{
// Initialize min value
int min = INT_MAX, min_index;
int v;
for ( v = 0; v < V; v++)
if (sptSet[v] == false && dist[v] <= min)
min = dist[v], min_index = v;
return min_index;
}
// A utility function to print the constructed distance
// array
void printSolution(int dist[])
{
int i;
```

```
printf("Vertex \t\t Distance from Source\n");
for (i = 0; i < V; i++)
printf("%d \t\t\t\t %d\n", i, dist[i]);
}
// Function that implements Dijkstra's single source
// shortest path algorithm for a graph represented using
// an adjacency matrix
void dijkstra(int graph[V][V], int src)
{
int i,v,u;
int count;
int dist[V]; // The output array. dist[i] will hold the
// shortest distance from src to i
int sptSet[V]; // sptSet[i] will be true if vertex i is
// included in the shortest
// path tree or the shortest distance from src to i is
// finalized
// Initialize all distances as INFINITE and sptSet[] as
// false
for (i = 0; i < V; i++)
dist[i] = INT_MAX, sptSet[i] = false;
// Distance of the source vertex from itself is always 0
dist[src] = 0;
// Initialize count outside the loop
// Find the shortest path for all vertices
for (count = 0; count < V - 1; count++)
{
// Pick the minimum distance vertex from the set of
// vertices not yet processed. u is always equal to
// src in the first iteration.
u = minDistance(dist, sptSet);
```

```
// Mark the picked vertex as processed
sptSet[u] = true;
// Update dist value of the adjacent vertices of the
// picked vertex.
for ( v = 0; v < V; v++)
// Update dist[v] only if it is not in sptSet,
// there is an edge from u to v, and the total
// weight of the path from src to v through u is
// smaller than the current value of dist[v]
if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX &&
dist[u] + graph[u][v] < dist[v])
dist[v] = dist[u] + graph[u][v];
}
// Print the constructed distance array
printSolution(dist);
}
// Driver's code
int main()
{
// Let us create the example graph discussed above
int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
{4, 0, 8, 0, 0, 0, 0, 11, 0},
{0, 8, 0, 7, 0, 4, 0, 0, 2},
{0, 0, 7, 0, 9, 14, 0, 0, 0},
{0, 0, 0, 9, 0, 10, 0, 0, 0},
{0, 0, 4, 14, 10, 0, 2, 0, 0},
{0, 0, 0, 0, 0, 2, 0, 1, 6},
{8, 11, 0, 0, 0, 0, 1, 0, 7},
{0, 0, 2, 0, 0, 0, 6, 7, 0}};
// Function call
dijkstra(graph, 0);
```

```
return 0;
}
```

34.Write a C program to accept a graph from the user, represent it in adjacency list and traverse it in-order

```
#include <stdio.h>

#include <stdlib.h>

// Node structure to represent vertices in the adjacency list

struct Node {

int data;

struct Node* next;

};

// Graph structure with an array of linked lists

struct Graph {

int vertices;

struct Node** adjList;

};

// Function to create a new node

struct Node* createNode(int vertex) {

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

newNode->data = vertex;

newNode->next = NULL;

return newNode;

}

// Function to add an edge to the adjacency list

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

struct Node* newNode = createNode(dest);

newNode->next = graph->adjList[src];

graph->adjList[src] = newNode;

}

// Function to traverse the graph in the order of linked lists

void inOrderTraversal(struct Graph* graph) {

int i;

printf("In-Order Traversal:\n");
```

```
for (i = 0; i < graph->vertices; i++) {
struct Node* temp = graph->adjList[i];
while (temp != NULL) {
printf("%d -> ", temp->data);
temp = temp->next;
}
printf("NULL\n");
}
}
int main() {
int i, k, src, dest, numEdges;
// Create a graph and add edges based on user input
struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
printf("Enter the number of vertices: ");
scanf("%d", &(graph->vertices));
graph->adjList = (struct Node**)malloc(graph->vertices * sizeof(struct Node*));
// Initialize adjacency list
for (i = 0; i < graph->vertices; i++) {
graph->adjList[i] = NULL;
}
printf("Enter the number of edges: ");
scanf("%d", &numEdges);
printf("Enter the edges (src dest):\n");
for (k = 0; k < numEdges; k++) {
scanf("%d %d", &src, &dest);
addEdge(graph, src, dest);
}
// Perform in-order traversal
inOrderTraversal(graph);
return 0;
}
```

35.Write a C program to accept a graph from user, represent in adjacency matrix and traverse it in-order

```
#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#define MAX_VERTICES 100 // Adjust the maximum vertices as needed

// Function to traverse the graph in in-order using adjacency matrix

void inOrderTraversal(int adjMatrix[MAX_VERTICES][MAX_VERTICES], int vertices, int currentVertex,
int *visited) {

int i;

visited[currentVertex] = 1;

printf("In-Order Traversal: %d\n", currentVertex + 1);

for (i = 0; i < vertices; i++) {

if (adjMatrix[currentVertex][i] && !visited[i]) {

inOrderTraversal(adjMatrix, vertices, i, visited);

}

}

}

int main() {

int visited[MAX_VERTICES] = {0};

int vertices, edges, i, j, k;

int adjMatrix[MAX_VERTICES][MAX_VERTICES] = {0};

printf("Enter the number of vertices: ");

scanf("%d", &vertices);

// Create adjacency matrix

// Accept edges from the user

printf("Enter the number of edges: ");

scanf("%d", &edges);

printf("Enter the edges (src dest):\n");

for (k = 0; k < edges; k++) {

scanf("%d %d", &i, &j);
```

```
// Check if vertices are within the valid range
if (i >= 1 && i <= vertices && j >= 1 && j <= vertices) {
adjMatrix[i - 1][j - 1] = 1;
} else {
printf("Invalid vertices. Please enter valid vertices.\n");
k--; // Decrement k to re-enter the edge
}
}
// Perform in-order traversal
for (i = 0; i < vertices; i++) {
if (!visited[i]) {
inOrderTraversal(adjMatrix, vertices, i, visited);
}
}
return 0;
}
```