KG COLLEGE OF ARTS AND SCIENCE
Autonomous Institution | Affiliated to Bharathiar University
Accredited with A++ Grade by NAAC
ISO 9001:2015 Certified Institution
KGiSL Campus, Saravanampatti, Coimbatore – 641 035

LAB MANUAL

Data Structures & Algorithms

B.Sc. CS / BCA / B.Sc. CT / B.Sc. IT / B.Sc. AI & DS

2024 BATCH

SYLLABUS

S. No.	List of Programs
1	Sample programs -Implementation of Array Operations
2	Implementation of Stack using Arrays
3	Implementation of Queue using Arrays
4	Conversion of Infix to Postfix Expression
5	Evaluation of Postfix Expression
6	Implementation of Singly Linked List
7	Implementation of Tree Traversal
8	Implementation of Depth First Search
9	Implementation of Breadth First Search
10	Implementation of Linear Search
11	Implementation of Binary Search
12	Implementation of Quick Sort
13	Implementation of Merge Sort
14	Greedy Algorithms - Activity Selection Problem
15	0-1 Knapsack Problem

- 1
Implementation of Array Operations

EXPERIMENT

AIM:

To implement various operations of array using C programming language

ALGORITHM:

Step 1: Start the process

Step 2: Open Turbo C program.

Step 3: Declare an array arr[] of fixed maximum size and other variables.

Step 4: Save the program with the correct file extension (Filename.c)

Step 5: Get the input initial array

Step 6: Perform Array operations based on choice

Step 7: Display the output

Step 8: Stop the Process

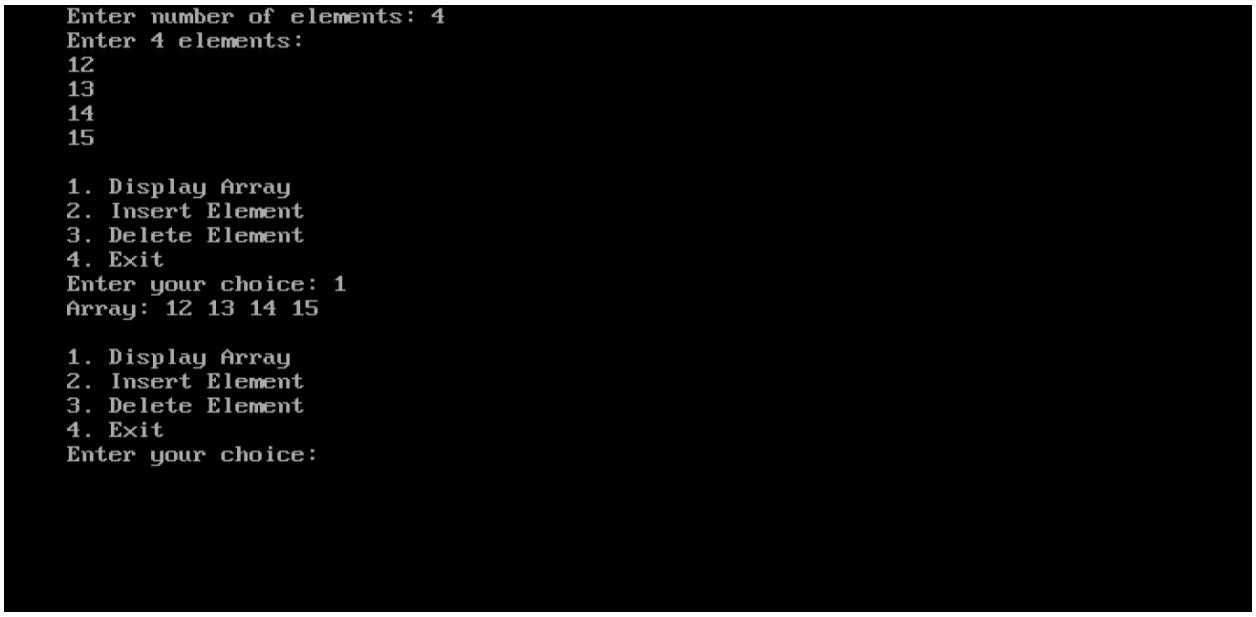
SOURCE CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#define SIZE 100

void  main()
{
        int i,arr[SIZE], n, choice, element, pos;
        printf("Enter number of elements: ");
        scanf("%d", &n);
        printf("Enter %d elements:\n", n);
        for(i = 0; i < n; i++)
        scanf("%d", &arr[i]);

        do
        {
                clrscr();
                printf("\n1. Display Array\n");
                printf("2. Insert Element\n");
                printf("3. Delete Element\n");
                printf("4. Exit\n");
                printf("Enter your choice: ");
                scanf("%d", &choice);
```

```
switch(choice)
{
        case 1:
                printf("Array: ");
                for(int i = 0; i < n; i++)
                printf("%d ", arr[i]);
                printf("\n");
                break;
        case 2:
                if (n == SIZE)
                {
                        printf("Array is full!\n");
                        break;
                }
                printf("Enter element to insert: ");
                scanf("%d", &element);
                printf("Enter position (0 to %d): ", n);
                scanf("%d", &pos);
                if(pos < 0 || pos > n)
                {
                   printf("Invalid position!\n");
                   break;
                }
                for(i = n; i > pos; i--)
                   arr[i] = arr[i - 1];
                arr[pos] = element;
                n++;
                printf("Element inserted.\n");
                break;
        case 3:
                if(n == 0)
                {
                        printf("Array is empty!\n");
                        break;
                }
                printf("Enter position to delete (0 to %d): ", n - 1);
                scanf("%d", &pos);
                if(pos < 0 || pos >= n)
                {
                        printf("Invalid position!\n");
                        break;
                }
                for( i = pos; i < n - 1; i++)
                    arr[i] = arr[i + 1];
                n--;
```

```
                printf("Element deleted.\n");
                break;
            case 4:

                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice.\n");


    }

    } while(choice != 4);

    getch();

    }
```

OUTPUT:



```
Enter your choice: 1
Array: 12 13 14 15

1. Display Array
2. Insert Element
3. Delete Element
4. Exit
Enter your choice:
2
Enter element to insert: 50
Enter position (0 to 4): 0
Element inserted.

1. Display Array
2. Insert Element
3. Delete Element
4. Exit
Enter your choice: 1
Array: 50 12 13 14 15

1. Display Array
2. Insert Element
3. Delete Element
4. Exit
Enter your choice: _
```

```
3. Delete Element
4. Exit
Enter your choice: 1
Array: 50 12 13 14 15

1. Display Array
2. Insert Element
3. Delete Element
4. Exit
Enter your choice: 3
Enter position to delete (0 to 4): 1
Element deleted.

1. Display Array
2. Insert Element
3. Delete Element
4. Exit
Enter your choice: 1
Array: 50 13 14 15

1. Display Array
2. Insert Element
3. Delete Element
4. Exit
Enter your choice:
```

EXPERIMENT – 2
Implementation of Stack using Arrays.

AIM:

To implement stack operations using array.

ALGORITHM :

Step 1: Start the process
Step 2: Open Turbo C program
Step 3: Edit the program
Step 4: Save the program with the correct file extension (Filename.c)
Step 5: Define a array *stack* of size *max* = 5 , Initialize *top* = -1
Step 6: If choice = 2 then, If top < max -1, Increment top, Store element at current position of top
else Print Stack overflow
Step 7: If top < 0 then Print Stack underflow Else Display current top element
Step 8: If choice = 3 then ,Display stack elements starting from top
Step 9: Display the output
Step 10: Stop the Process

SOURCE CODE:

```
#include <stdio.h>
#include <conio.h>
#include<stdlib.h>
#define max 5
static int stack[max];
int top = -1;
        void push(int x)
        {
                stack[++top] = x;
        }
        int pop()
        {
                return (stack[top--]);
        }
        void view()
        {
                int i;
                if (top < 0)
                        printf("\n Stack Empty \n");
                else
                {
                        printf("\n Top-->");
```

```
                for(i=top; i>=0; i--)
                        printf("%4d", stack[i]);


                printf("\n");
        }
    }
void main()
{
    int ch=0, val;
    clrscr();
    while(ch != 4)
    {
        clrscr();
        printf("\n STACK OPERATIONs \n");
        printf("1.PUSH \t ");
        printf("2.POP \t");
        printf("3.VIEW \t ");
        printf("4.QUIT \n");
        printf("Enter Choice : ");
        scanf("%d", &ch);
        switch(ch)
        {
            case 1:
                if(top < max-1)
                {
                    printf("\nEnter Stack element : ");
                    scanf("%d", &val);
                    push(val);
                }
                else
                        printf("\n Stack Overflow \n");
                break;
            case 2:
                if(top < 0)
                        printf("\n Stack Underflow \n");
                else
                {
                        val = pop();
                        printf("\n Popped element is %d\n", val);
                }
                break;
            case 3:
                view();
                break;
            case 4:
                exit(0);
            default:
                printf("\n Invalid Choice \n");
        }
```

```
        }
        getch();
    }
```

OUTPUT:

```
STACK OPERATION
1.PUSH      2.POP       3.VIEW      4.QUIT
Enter Choice : 1
Enter Stack element : 12
STACK OPERATION
1.PUSH      2.POP       3.VIEW      4.QUIT
Enter Choice : 1
Enter Stack element : 23
STACK OPERATION
1.PUSH      2.POP       3.VIEW      4.QUIT
Enter Choice : 1
Enter Stack element : 34
STACK OPERATION
1.PUSH      2.POP       3.VIEW      4.QUIT
Enter Choice : 1
Enter Stack element : 45
STACK OPERATION
1.PUSH      2.POP       3.VIEW      4.QUIT
Enter Choice : 3
Top--> 45 34 23 12
STACK OPERATION
1.PUSH      2.POP       3.VIEW      4.QUIT
Enter Choice : 2
Popped element is 45
STACK OPERATION
1.PUSH      2.POP       3.VIEW      4.QUIT
Enter Choice : 3
Top--> 34 23 12
STACK OPERATION
1.PUSH      2.POP       3.VIEW      4.QUIT
Enter Choice : 4
```

EXPERIMENT – 3

Implementation of Queue using Arrays

AIM:

To implement queue operations using array.

ALGORITHM:

Step 1: Start the process
Step 2: Open Turbo C program
Step 3: Save the program with the correct file extension **(Filename.c)**
Step 4: Define a array *queue* of size *max* = 5
Step 5: Initialize *front* = *rear* = –1
Step 6: If choice = 1 then, If rear < max -1, Increment rear
Store element at current position of rear Else Print Queue Full
Step 7: If choice = 2 then, If front = –1 then Print Queue empty
Else Display current front element, Increment front
Step 8: If choice = 3 then, Display queue elements starting from front to rear.
Step 9: Display the output
Step 10: Stop the Process

SOURCE CODE:

```
#include <stdio.h>
#include <conio.h>
#include<stdlib.h>
#define max 5
static int queue[max];
int front = -1;
int rear = -1;
        void insert(int x)
        {
                queue[++rear] = x;
                if (front == -1)
                front = 0;
        }
        int remove()
        {
                int val;
                val = queue[front];
                if (front==rear && rear==max-1)
                front = rear = -1;
```

```
                else
                front ++;
                return (val);
        }
        void view()
        {       int i;
                if (front == -1)

                printf("\n Queue Empty \n");
                else
                {
                        printf("\n Front-->");
                        for(i=front; i<=rear; i++)
                        printf("%4d", queue[i]);
                        printf(" <--Rear\n");
                }
        }

main()
        {
                int ch= 0,val;
                while(ch != 4)
                {
                        printf("\n QUEUE OPERATION \n");
                        printf("1.INSERT \t ");
                        printf("2.DELETE \t ");
                        printf("3.VIEW \t ");
                        printf("4.QUIT \t\n");
                        printf("\n Enter Choice : ");
                        scanf("%d", &ch);
                switch(ch)
                {
                        case 1:
                                if(rear< max-1)
                                {
                                        printf("\n Enter element to be inserted : ");
                                        scanf("%d", &val);
                                        insert(val);
                                }
                                else
                                        printf("\n Queue Full \n");
                                        break;
                        case 2:
                                if(front == -1)
                                        printf("\n Queue Empty \n");
                                else
                                {
                                        val = remove();
                                        printf("\n Element deleted : %d \n", val);
                                }
```

```
                        break;
                case 3:
                        view();
                        break;
                case 4:
                        exit(0);
                default:
                        printf("\n Invalid Choice \n");
                        }
        }
}
```

OUTPUT:

```
QUEUE OPERATION
1.INSERT    2.DELETE    3.VIEW      4.QUIT
Enter Choice : 1
Enter element to be inserted : 12
QUEUE OPERATION
1.INSERT    2.DELETE    3.VIEW      4.QUIT
Enter Choice : 1
Enter element to be inserted : 23
QUEUE OPERATION
1.INSERT    2.DELETE    3.VIEW      4.QUIT
Enter Choice : 1
Enter element to be inserted : 34
QUEUE OPERATION
1.INSERT    2.DELETE     3.VIEW     4.QUIT
Enter Choice : 1
Enter element to be inserted : 45
QUEUE OPERATION
1.INSERT    2.DELETE    3.VIEW      4.QUIT
Enter Choice : 1
Enter element to be inserted : 56
QUEUE OPERATION
1.INSERT    2.DELETE    3.VIEW      4.QUIT
Enter Choice : 1
Queue Full
QUEUE OPERATION
1.INSERT     2.DELETE    3.VIEW     4.QUIT
Enter Choice : 3
Front--> 12 23 34 45 56 <--Rear
```

EXPERIMENT–4

CONVERSION OF INFIX TO POSTFIX EXPRESSION

AIM:

To implement Conversion of infix to postfix expression using C programming language

ALGORITHM:

Step1: Startthe process

Step2: OpenTurbo C program.

Step3: Initialize an empty stack for operators and an empty string for the output (postfix).

Step 4: Scan the infix expression from left to right.

Step 5: Initialize an empty stack for operators and an empty string for the output (postfix).

Step 6 : Scan the infix expression from left to right.

Step 7: For each symbol in the expression:

- **Operand**: Add it directly to the postfix output.
- **Left Parenthesis** (: Push it onto the stack.
- **Right Parenthesis**): Pop from the stack and add to postfix until
- **Operator (+, -, *, /, ^)**:

Step 8: While the stack is not empty and the precedence of the top of the stack is **greater than or equal to** the current operator (consider associativity), **pop** from the stack and add to postfix.

Step 9: Push the current operator onto the stack.

Step 10 : After scanning, pop any remaining operators from the stack and add to the postfix expression.

Step 11: Display the output

Step 12 : Stop the Process

SOURCE CODE:

```
#include <stdio.h>
#include <ctype.h>  // for isalnum()
#include <string.h>
#include <stdlib.h>
#define SIZE 100
```

```
// Stack implementation
char stack[SIZE];
int top = -1;
void push(char ch) {
    if (top >= SIZE - 1) {
        printf("Stack overflow\n");
        return;
    }
    stack[++top] = ch;
}
char pop() {
    if (top == -1) {
        return '\0';
    }
    return stack[top--];
}
char peek() {
    if (top == -1) {
        return '\0';
    }
    return stack[top];
}
// Precedence function
int precedence(char ch) {
    switch (ch) {
        case '^': return 3;
        case '*':
        case '/': return 2;
        case '+':
        case '-': return 1;
        default: return 0;
    }
}
// Infix to Postfix conversion function
void infixToPostfix(char* infix, char* postfix) {
    int i, k = 0;
    char ch;

    for (i = 0; infix[i] != '\0'; i++) {
        ch = infix[i];
        if (isalnum(ch)) {  // operand
            postfix[k++] = ch;
        }
        else if (ch == '(') {
            push(ch);
        }
        else if (ch == ')') {
```

```
            while (peek() != '(' && top != -1) {
                postfix[k++] = pop();
            }
            pop(); // remove '('
        }
        else { // operator
            while (precedence(ch) <= precedence(peek()) && top != -1) {
                postfix[k++] = pop();
            }
            push(ch);
        }
    }
    // Pop remaining operators
    while (top != -1) {
        postfix[k++] = pop();
    }
    postfix[k] = '\0';
}
int main() {
    char infix[SIZE], postfix[SIZE];
    printf("Enter infix expression: ");
    scanf("%s", infix);
    infixToPostfix(infix, postfix);
    printf("Postfix expression: %s\n", postfix);
    return 0;}
```

OUTPUT:

```
Enter infix expression: A+B*C/D
Postfix expression: ABC*D/+


=== Code Execution Successful ===
```

EXPERIMENT– 5

EVALUATION OF POSTFIX EXPRESSION

AIM:

To implement Evaluation of postfix expression using C programming language

ALGORITHM:

Step1: Start the process
Step2:OpenTurbo C program.
Step3: **Initialize an empty stack**.
Step4: **Scan the postfix expression** from left to right.
Step5 : For each symbol in the expression: If the symbol is an **operand** (digit), **push it** onto the stack.
Step6: If the symbol is an **operator (+, -, *, /)**:

- **Pop two elements** from the stack (say op2 and op1)
- **Perform** the operation op1 operator op2
- **Push the result** back to the stack.

Step7: **After the expression is scanned**, the stack will have only **one element**, which is the result.
Step8: Return that value.
Step 9: Display the output
Step 10 :Stop the Process

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define MAX_SIZE 100
int stack[MAX_SIZE];
int top = -1;
void push(int item) {
   if (top >= MAX_SIZE - 1) {
      printf("Stack Overflow\n");
      exit(1);
   }
   stack[++top] = item;
}
int pop() {
   if (top < 0) {
      printf("Stack Underflow\n");
      exit(1);
   }
   return stack[top--];
```

```
}
int evaluatePostfix(char* exp) {
    int i, operand1, operand2, result;

    for (i = 0; exp[i] != '\0'; i++) {
        if (isdigit(exp[i])) {
            push(exp[i] - '0');
        }
        else {
            operand2 = pop();
            operand1 = pop();

            switch (exp[i]) {
                case '+': result = operand1 + operand2; break;
                case '-': result = operand1 - operand2; break;
                case '*': result = operand1 * operand2; break;
                case '/': result = operand1 / operand2; break;
            }

            push(result);
        }
    }

    return pop();
}


int main() {
    char postfix[MAX_SIZE];

    printf("Enter a postfix expression: ");
    scanf("%s", postfix);

    printf("Evaluated result: %d\n", evaluatePostfix(postfix));

    return 0;
}
```

OUTPUT:

```
Enter a postfix expression: 987+*
Evaluated result: 135


=== Code Execution Successful ===
```

EXPERIMENT–6

SINGLY LINKED LIST

AIM:

To implement Singly Linked List operations in C.

ALGORITHM:

Step 1: Initialize and Define a node structure with:
- An integer data
- A pointer link to the next node

- Initialize the head pointer to NULL

Step 2: Loop infinitely until user chooses to exit
- Display menu options:
  1. Insert at beginning
  2. Insert at end
  3. Insert after a node
  4. Delete a node
  5. Display list
  6. Exit

Step 3:
If choice = 1 (Insert at Beginning) follows the steps below
- Read value from user
- Create a new node with given value
- Set new node's link to current head
- Update head to point to new node

If choice = 2 (Insert at End) follow the steps:
- Read value from user
- Create a new node with given value and link = NULL
- If list is empty, update head to new node
- Else, traverse to the last node and set its link to the new node

If choice = 3 (Insert after a Node) follow the steps
- Read key and value from user
- Traverse list to find node with data == key
  - If found:
    - Create a new node with value
    - Set its link to key node's link
    - Update key node's link to new node

- If not found, show error message

If choice = 4 (Delete a Node)follow the steps

- Read value from user
- If list is empty, show error
- If head->data == value:
  - Update head to next node
  - Free the deleted node
- Else, traverse to find the node
  - If found, update previous node's link
  - Free the deleted node
  - If not found, show error

If choice = 5 (Display List) follow the steps

- If list is empty, show message
- Else, traverse from head and print each node's data

If choice = 6 Exit from the program

Step 4:

Stop the proces

SOURCE CODE:

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

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

struct node *head = NULL;

// Function to insert at beginning
void insert_at_front(int value)
{
   struct node *newnode = (struct node *)malloc(sizeof(struct node));
   newnode->data = value;
   newnode->link = head;
   head = newnode;
}

// Function to insert at end
void insert_at_end(int value)
{
   struct node *newnode = (struct node *)malloc(sizeof(struct node));
```

```
    newnode->data = value;
    newnode->link = NULL;

    if (head == NULL)
    {
        head = newnode;
    }
    else
    {
        struct node *temp = head;
        while (temp->link != NULL)
        {
            temp = temp->link;
        }
        temp->link = newnode;
    }
}

// Function to insert after a given value
void insert_middle(int key, int value)
{
    struct node *newnode = (struct node *)malloc(sizeof(struct node));
    struct node *temp = head;
    while (temp != NULL && temp->data != key)
    {
        temp = temp->link;
    }

    if (temp == NULL)
    {
        printf("Node with value %d not found.\n", key);
        return;
    }


    newnode->data = value;
    newnode->link = temp->link;
    temp->link = newnode;
}

// Function to delete a node by value
void delete_node(int value)
{
    struct node *temp = head, *prev = NULL;

    if (temp != NULL && temp->data == value)
    {
        head = temp->link;
```

```
        free(temp);
        printf("Node with value %d deleted.\n", value);
        return;
    }

    while (temp != NULL && temp->data != value)
    {
        prev = temp;
        temp = temp->link;
    }

    if (temp == NULL)
    {
        printf("Node with value %d not found.\n", value);
        return;
    }

    prev->link = temp->link;
    free(temp);
    printf("Node with value %d deleted.\n", value);
}

// Function to display the list
void display()
{
    struct node *temp = head;
    if (temp == NULL) {
        printf("List is empty.\n");
        return;
    }

    printf("Linked List: ");
    while (temp != NULL)
    {
        printf("%d -> ", temp->data);
        temp = temp->link;
    }
    printf("NULL\n");
}


void main()
{
    int choice, value, key;

    while (1)
    {
        clrscr();
```

```
        printf("\n--- Singly Linked List Menu ---\n");
        printf("1. Insert at beginning\n");
        printf("2. Insert at end\n");
        printf("3. Insert after a node\n");
        printf("4. Delete a node\n");
        printf("5. Display list\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            printf("Enter value to insert at beginning: ");
            scanf("%d", &value);
            insert_at_front(value);
            break;
        case 2:
            printf("Enter value to insert at end: ");
            scanf("%d", &value);
            insert_at_end(value);
            break;
        case 3:
            printf("Enter key after which to insert: ");
            scanf("%d", &key);
            printf("Enter value to insert: ");
            scanf("%d", &value);
            insert_middle(key, value);
            break;
        case 4:
            printf("Enter value to delete: ");
            scanf("%d", &value);
            delete_node(value);
            break;
        case 5:
            display();
            break;
        case 6:
            printf("Exiting program.\n");
            exit(0);
        default:
            printf("Invalid choice. Try again.\n");
        }
    }

}
```

Output:

--- Singly Linked List Menu ---
1. Insert at beginning
2. Insert at end
3. Insert after a node
4. Delete a node
5. Display list
6. Exit
Enter your choice: 1
Enter value to insert at beginning: 10

--- Singly Linked List Menu ---
1. Insert at beginning
2. Insert at end
3. Insert after a node
4. Delete a node
5. Display list
6. Exit
Enter your choice: 1
Enter value to insert at beginning:
20

--- Singly Linked List Menu ---
1. Insert at beginning
2. Insert at end
3. Insert after a node
4. Delete a node
5. Display list
6. Exit
Enter your choice:
2
Enter value to insert at end: 30

--- Singly Linked List Menu ---
1. Insert at beginning
2. Insert at end
3. Insert after a node
4. Delete a node
5. Display list
6. Exit
Enter your choice: 3
Enter key after which to insert: 10
Enter value to insert: 40

--- Singly Linked List Menu ---
1. Insert at beginning
2. Insert at end
3. Insert after a node
4. Delete a node
5. Display list
6. Exit
Enter your choice: 5
Linked List: 20 -> 10 -> 40 -> 30 -> NULL

--- Singly Linked List Menu ---
1. Insert at beginning
2. Insert at end
3. Insert after a node
4. Delete a node
5. Display list
6. Exit
Enter your choice: 4
Enter value to delete: 10
Node with value 10 deleted.

--- Singly Linked List Menu ---
1. Insert at beginning
2. Insert at end
3. Insert after a node
4. Delete a node
5. Display list
6. Exit
Enter your choice: 5
Linked List: 20 -> 40 -> 30 -> NULL

--- Singly Linked List Menu ---
1. Insert at beginning
2. Insert at end
3. Insert after a node
4. Delete a node
5. Display list
6. Exit
Enter your choice:

EXPERIMENT 7

IMPLEMENTING TREE TRAVERSAL

AIM

To implement three types of tree traversals (Inorder, Preorder, and Postorder) on a binary tree using recursion in C programming.

ALGORITHM:

Step 1: Start the process.
Step 2: Create a structure of the Node which contains an integer data to store the value and Two pointers left and right pointing to the left and right child nodes respectively.
Step 3: Create a function createNode(data):
Step 4: Construct the binary tree by creating nodes and linking them.
Step 5: In Inorder Traversal (Left, Root, Right), If the current node is not NULL then Recursively traverse the left subtree and Print the data of the current node and then Recursively traverse the right subtree.
Step 6: In Preorder Traversal (Root, Left, Right), If the current node is not NULL then Print the data of the current node and recursively traverse the left and right subtree.
Step 7: In Postorder Traversal (Left, Right, Root), If the current node is not NULL then recursively traverse the left subtree and right subtree and Print the data of the current node.
Step 8: Call each traversal function with the root node and print the traversal order.
Step 9: Stop the process.

PROGRAM :

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

// Define the structure for a tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node with the given data
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;
}
```

```
// Inorder Traversal (Left, Root, Right)
void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}

// Preorder Traversal (Root, Left, Right)
void preorderTraversal(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}

// Postorder Traversal (Left, Right, Root)
void postorderTraversal(struct Node* root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->data);
    }
}

// Driver function to test the tree traversal functions
int main() {
    // Create the root node and other nodes
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    // Perform and display each traversal
    printf("Inorder Traversal: ");
    inorderTraversal(root);
    printf("\n");
    printf("Preorder Traversal: ");
    preorderTraversal(root);
    printf("\n");
    printf("Postorder Traversal: ");
    postorderTraversal(root);
    printf("\n");
    return 0;
}
```

OUTPUT

```
Inorder Traversal: 4 2 5 1 3
Preorder Traversal: 1 2 4 5 3
Postorder Traversal: 4 5 2 3 1
```

EXPERIMENT 8

IMPLEMENTATION OF DEPTH FIRST SEARCH

AIM:

To implement Depth First Search (DFS) traversal on a graph using adjacency lists and recursion in C.

ALGORITHM:

Step 1: Start the process.
Step 2: Define the graph structure and create a struct node to represent each adjacency list node containing the vertex and a pointer to the next node.
Step 3: Create function createNode (int v) to allocate memory for a new adjacency list node and initialize it.
Step 4: Create function createGraph(int vertices) to allocate memory for the graph, adjacency lists, and visited array, initializing all lists as empty and visited flags as 0.
Step 5: Create function addEdge(Graph* graph, int src, int dest) to add an undirected edge by adding nodes in both adjacency lists for src and dest.
Step 6: Mark the current vertex as visited and Print or process the current vertex and recursively visit all adjacent vertices that are not yet visited.
Step 7: Build a sample graph with vertices and edges.
Step 8: Call DFS starting from a chosen vertex.
Step 9: Stop the process.

PROGRAM:

```
// DFS algorithm in C
#include <stdio.h>
#include <stdlib.h>

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

struct node* createNode(int v);

struct Graph {
  int numVertices;
  int* visited;
  // We need int** to store a two dimensional array.
  // Similary, we need struct node** to store an array of Linked lists
  struct node** adjLists;
};
```

```
// DFS algo
void DFS(struct Graph* graph, int vertex) {
  struct node* adjList = graph->adjLists[vertex];
  struct node* temp = adjList;

  graph->visited[vertex] = 1;
  printf("Visited %d \n", vertex);

  while (temp != NULL) {
    int connectedVertex = temp->vertex;

    if (graph->visited[connectedVertex] == 0) {
      DFS(graph, connectedVertex);
    }
    temp = temp->next;
  }
}

// Create a node
struct node* createNode(int v) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}

// Create graph
struct Graph* createGraph(int vertices) {
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices * sizeof(struct node*));
 graph->visited = malloc(vertices * sizeof(int));

  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }
  return graph;
}

// Add edge
void addEdge(struct Graph* graph, int src, int dest) {
  // Add edge from src to dest
  struct node* newNode = createNode(dest);
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;
```

```
  // Add edge from dest to src
  newNode = createNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}

// Print the graph
void printGraph(struct Graph* graph) {
  int v;
  for (v = 0; v < graph->numVertices; v++) {
    struct node* temp = graph->adjLists[v];
    printf("\n Adjacency list of vertex %d\n ", v);
    while (temp) {
      printf("%d -> ", temp->vertex);
      temp = temp->next;
    }
    printf("\n");
  }
}

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

OUTPUT

```
Adjacency list of vertex 0
 2 -> 1 ->

 Adjacency list of vertex 1
 2 -> 0 ->

 Adjacency list of vertex 2
 3 -> 1 -> 0 ->

 Adjacency list of vertex 3
 2 ->
Visited 2
Visited 3
Visited 1
Visited 0
```

EXPERIMENT 9
Implementation of Breadth First Search.

AIM:
To implement Breadth First Search (BFS) traversal on an undirected graph represented using adjacency lists in C, and to traverse the graph starting from a given vertex.

ALGORITHM:
Step 1: Start the process
Step 2: Create a Node structure to represent each adjacency list node with an integer vertex to store the vertex number and a pointer next to the next adjacency node.
Step 3: Create a Graph structure with an integer numVertices to store the number of vertices.
Step 4: Initialize the graph and allocate memory for the graph structure, adjacency lists and the visited array.
Step 5: Add edges to the graph
Step 6: Perform BFS traversal
Step 7: Stop the Process.

PROGRAM:
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Structure to represent a node in the adjacency list
typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

// Structure to represent the graph
typedef struct Graph {
    int numVertices;
    Node** adjLists;
    bool* visited;
} Graph;

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

// Function to create a graph
Graph* createGraph(int vertices) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->numVertices = vertices;
```

```
    graph->adjLists = (Node**)malloc(vertices * sizeof(Node*));
    graph->visited = (bool*)malloc(vertices * sizeof(bool));

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = false;
    }
    return graph;
}

// Function to add an edge to the graph
void addEdge(Graph* graph, int src, int dest) {
    Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

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

// Function to implement BFS
void bfs(Graph* graph, int startVertex) {
    int queue[graph->numVertices];
    int front = 0, rear = 0;

    graph->visited[startVertex] = true;
    queue[rear++] = startVertex;

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

        Node* temp = graph->adjLists[currentVertex];
        while (temp) {
            int adjVertex = temp->vertex;
            if (!graph->visited[adjVertex]) {
                graph->visited[adjVertex] = true;
                queue[rear++] = adjVertex;
            }
            temp = temp->next;
        }
    }
}

int main() {
    int numVertices = 6;
    Graph* graph = createGraph(numVertices);
```

```
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 5);

    printf("BFS Traversal starting from vertex 0: ");
    bfs(graph, 0);
    printf("\n");

    return 0;
}
```

OUTPUT

BFS Traversal starting from vertex 0: 0 2 1 5 4 3

EXPERIMENT 10

Implementation of Linear Search

AIM:

To write a C program to search for an element in an array using the **Linear Search** technique.

ALGORITHM:

Step 1: Start the program
Step 2: Input the size of the array.
Step 3: Input the array elements.
Step 4: Input the element to be searched (key).
Step 6: Traverse the array from the beginning:
- If the current element matches the key, print the index and stop.

Step 7: If the element is not found after traversing the whole array, print "Element not found."
Step 8: Stop the program

PROGRAM :

```
#include <stdio.h>

int main() {
   int array[100], n, i, key, found = 0;

   // Input array size
   printf("Enter number of elements in array: ");
   scanf("%d", &n);

   // Input array elements
   printf("Enter %d elements:\n", n);
   for(i = 0; i < n; i++) {
      scanf("%d", &array[i]);
   }

   // Input element to be searched
   printf("Enter the element to search: ");
   scanf("%d", &key);

   // Linear Search
   for(i = 0; i < n; i++) {
      if(array[i] == key) {
         printf("Element found at position %d (index %d)\n", i + 1, i);
```

```
            found = 1;
            break;
        }
    }

    // If not found
    if(!found) {
        printf("Element not found in the array.\n");
    }

    return 0;
}
```

OUTPUT:

```
Enter number of elements in array: 5
Enter 5 elements:
10 20 30 40 50
Enter the element to search: 30
Element found at position 3 (index 2)
```

EXPERIMENT 11

Implementing Binary Search

AIM:

To write a C program to implement Binary Search to find the position of a given element in an array of integers sorted in ascending order.

ALGORITHM:

1. Start the program
2. Declare variables: array[100], n, search, first, last, middle, c.
3. Prompt the user to enter the number of elements n.
4. Read n.
5. Prompt the user to enter n integers in ascending order.
6. Use a loop to read n integers into the array.
7. Prompt the user to enter the value to be searched (search).
8. Initialize:
    - first = 0
    - last = n - 1
    - middle = (first + last) / 2
9. Repeat steps 10 to 13 while first <= last:
10. If array[middle] == search:
    - Print the position as middle + 1.
    - Exit loop.
11. If array[middle] < search:
    - Set first = middle + 1.
12. Else:
    - Set last = middle - 1.
13. Update middle = (first + last) / 2.
14. If first > last, print "Not found! search isn't present in the list."
15. Stop

PROGRAM:

```
#include <stdio.h>
int main()
{
  int c, first, last, middle, n, search, array[100];
 printf("Enter number of elements\n");
 scanf("%d", &n);
 printf("Enter %d integers in ascending order\n", n);
 for (c = 0; c < n; c++)
        scanf("%d", &array[c]);
printf("Enter value to find\n");
scanf("%d", &search);
first = 0;
last = n - 1;
middle = (first+last)/2;
```

```
 while (first <= last) {
 if (array[middle] < search)
first = middle + 1;
 else if (array[middle] == search) {
 printf("%d found at location %d \n", search, middle+1);
 break;
 }
else
last = middle - 1;
middle = (first + last)/2;
 }
 if (first > last)
printf("Not found! %d isn't present in the list.\n", search);
return 0;
}
```

OUTPUT:

```
Enter number of elements
5
Enter 5 integers in ascending order
23
45
55
67
78
Enter value to find
78
78 found at location 5
```

EXPERIMENT 12
Implementing Binary Search

AIM:
To write a C program to sort an array of integers using the Quick Sort algorithm.

ALGORITHM:
1. **Start.**
2. Read the number of elements n from the user.
3. Read n integer elements into an array arr[].
4. Call the quickSort function with parameters arr, low = 0, and high = n - 1.
5. In the quickSort function:
    - If low < high, do the following:
        - Call the partition function:
            - Choose the last element as the pivot.
            - Initialize i = low - 1.
            - For each j from low to high - 1:
                - If arr[j] < pivot, increment i and swap arr[i] with arr[j].
            - After the loop, swap arr[i+1] with arr[high] (placing the pivot in the correct position).
            - Return the index pi = i + 1.
        - Recursively call quickSort(arr, low, pi - 1) and quickSort(arr, pi + 1, high) to sort the sub-arrays.
6. After sorting, print the sorted array using printArray().
7. **Stop.**

PROGRAM:
```
#include <stdio.h>

// Function to swap two numbers
void swap(int* a, int* b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

// Function to partition the array
int partition(int arr[], int low, int high) {
   int pivot = arr[high];  // Pivot is the last element
   int i = (low - 1);      // Index of the smaller element

   for (int j = low; j <= high - 1; j++) {
      // If current element is smaller than the pivot
      if (arr[j] < pivot) {
```

```
            i++;  // Increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// QuickSort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // pi is partitioning index, arr[pi] is now at right place
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

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

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter the elements: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    quickSort(arr, 0, n - 1);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

OUTPUT:

```
Enter the number of elements: 5
Enter the elements:
12
2
9
3
4
Sorted array: 2 3 4 9 12
```

EXPERIMENT 13

MERGE SORT

AIM:

To implement **Merge Sort** algorithm using C language in Turbo C compiler and sort an array of integers in ascending order.

ALGORITHM

1. **Start**
2. Input the number of elements n and the array a[n]
3. Call mergeSort(a, low, high) where low = 0 and high = n - 1
4. In mergeSort(a, low, high):
    - If low < high, do the following:
        - Find the middle index: mid = (low + high) / 2
        - Recursively call mergeSort(a, low, mid)
        - Recursively call mergeSort(a, mid + 1, high)
        - Call merge(a, low, mid, high) to merge the sorted sub-arrays
5. In merge(a, low, mid, high):
    - Create a temporary array
    - Compare elements from both sub-arrays and copy the smaller one into the temp array
    - Copy remaining elements (if any) from both sub-arrays
    - Copy merged elements back into the original array
6. Display the sorted array
7. **End**

PROGRAM

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

void merge(int a[], int low, int mid, int high) {
   int i, j, k;
   int b[50]; // Temporary array, assuming max size of 50
   i = low;
   j = mid + 1;
   k = low;

   while (i <= mid && j <= high) {
      if (a[i] <= a[j]) {
         b[k] = a[i];
         i++;
      } else {
         b[k] = a[j];
```

```
            j++;
        }
        k++;
    }

    while (i <= mid) {
        b[k] = a[i];
        i++;
        k++;
    }

    while (j <= high) {
        b[k] = a[j];
        j++;
        k++;
    }

    for (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, low, mid, high);
    }
}

void main() {
    int a[50], n, i;

  clrscr();
    printf("Enter number of elements (max 50): ");
    scanf("%d", &n);

    if (n > 50) {
        printf("Maximum allowed elements is 50.\n");
        return;
    }

    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }
```

```
    mergeSort(a, 0, n - 1);

    printf("Sorted array:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }

 getch();
}
```

OUTPUT

```
Enter number of elements (max 50): 5
Enter 5 elements:
54
76
86
23
11
Sorted array:
11 23 54 76 86
```

EXPERIMENT 14
ACTIVITY SELECTION PROBLEM (Greedy Method)

AIM:

To implement the **Activity Selection Problem** using a greedy algorithm in C language on Turbo C, and select the maximum number of non-overlapping activities that a person can perform.

ALGORITHM:

1. **Start**
2. Input the number of activities n
3. Input start[] and finish[] times for all activities
4. Sort the activities in ascending order of **finish times**
5. Select the first activity (it always gets selected)
6. For each remaining activity i from 1 to n-1:
    - If start[i] >= finish[prev_selected], then:
        - Select activity i
        - Update prev_selected = i
7. Display the selected activities
8. **End**

PROGRAM

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

void sortActivities(int n, int start[], int finish[])
{
   int i, j, temp;
   for (i = 0; i < n - 1; i++) {
      for (j = i + 1; j < n; j++) {
         if (finish[i] > finish[j]) {
            // Swap finish times
            temp = finish[i];
            finish[i] = finish[j];
            finish[j] = temp;
            // Swap start times to keep pair intact
            temp = start[i];
            start[i] = start[j];
            start[j] = temp;
         }
      }
   }
```

```
}

void activitySelection(int n, int start[], int finish[]) {
   int i, last = 0;

   printf("Selected activities (0-based index):\n");
   printf("%d ", 0); // Always select first activity

   for (i = 1; i < n; i++) {
      if (start[i] >= finish[last]) {
         printf("%d ", i);
         last = i;
      }
   }
}

void main() {
   int n, i, start[50], finish[50];

 clrscr();
   printf("Enter number of activities: ");
   scanf("%d", &n);

   printf("Enter start times:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &start[i]);
   }

   printf("Enter finish times:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &finish[i]);
   }

   sortActivities(n, start, finish);
   activitySelection(n, start, finish);

   getch();
}
```

OUTPUT

```
Enter number of activities: 6
Enter start times:
1
3
0
5
8
```

```
5
Enter finish times:
2
4
6
7
9
9
Selected activities (0-based index):
0 1 3 4
```

EXPERIMENT 15
0-1 KNAPSACK PROBLEM

AIM:

To implement the **0-1 Knapsack Problem** using **dynamic programming** in C language (Turbo C), and to determine the maximum total value that can be placed into a knapsack of limited capacity.

ALGORITHM:

1. **Start.**
2. **Input** number of items n.
3. **Input**:
   - Weight of each item into array weight[].
   - Value of each item into array value[].
4. **Input** maximum capacity of the knapsack W.
5. Create a 2D array K[n+1][W+1] to store intermediate results.
6. Loop i from 0 to n:
   - Loop w from 0 to W:
     - If i == 0 or w == 0, set K[i][w] = 0.
     - Else if weight[i-1] <= w, then
       K[i][w] = max(value[i-1] + K[i-1][w - weight[i-1]], K[i-1][w])
     - Else
       K[i][w] = K[i-1][w]
7. The value K[n][W] will be the maximum value that can be put in the knapsack.
8. Trace back through the matrix to find which items were selected.
9. **Display** the maximum value and selected items.
10. **End.**

PROGRAM

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

#define MAX 20

int max(int a, int b) {
   return (a > b) ? a : b;
}

void knapsack(int n, int weight[], int value[], int W) {
   int i, w;
   int K[MAX][MAX];

   // Build table K[][] in bottom-up manner
   for (i = 0; i <= n; i++) {
```

```
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (weight[i - 1] <= w)
                K[i][w] = max(value[i - 1] + K[i - 1][w - weight[i - 1]], K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }
    // Final result
    printf("\nMaximum value in Knapsack = %d\n", K[n][W]);

    // Traceback to find selected items
    printf("Items included:\n");
    w = W;
    for (i = n; i > 0 && w > 0; i--) {
        if (K[i][w] != K[i - 1][w]) {
            printf("Item %d (Weight = %d, Value = %d)\n", i, weight[i - 1], value[i - 1]);
            w -= weight[i - 1];
        }
    }
}
void main() {
    int weight[MAX], value[MAX], n, W, i;
   clrscr();

    printf("Enter number of items: ");
    scanf("%d", &n);

    printf("Enter weights of items:\n");
    for (i = 0; i < n; i++) {
        printf("Weight of item %d: ", i + 1);
        scanf("%d", &weight[i]);
    }
    printf("Enter values of items:\n");
    for (i = 0; i < n; i++) {
        printf("Value of item %d: ", i + 1);
        scanf("%d", &value[i]);
    }
    printf("Enter maximum capacity of knapsack: ");
    scanf("%d", &W);
    knapsack(n, weight, value, W);

 getch();
}
```

OUTPUT

```
Enter number of items: 4
Enter weights of items:
Weight of item 1: 2
Weight of item 2: 3
Weight of item 3: 4
Weight of item 4: 5
Enter values of items:
Value of item 1: 3
Value of item 2: 4
Value of item 3: 5
Value of item 4: 6
Enter maximum capacity of knapsack: 5

Maximum value in Knapsack = 7
Items included:
Item 2 (Weight = 3, Value = 4)
Item 1 (Weight = 2, Value = 3)
```

****