Array:

```
#include <stdio.h>

#define MAX_SIZE 100

int array[MAX_SIZE];
int size = 0;

void displayArray() {
    if (size == 0) {
        printf("Array is empty\n");
        return;
    }

    printf("Array elements: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

void insertAtBeginning(int element) {
    if (size >= MAX_SIZE) {
        printf("Array is full. Cannot insert element.\n");
        return;
    }

    for (int i = size; i > 0; i--) {
        array[i] = array[i - 1];
    }

    array[0] = element;
    size++;
}

void insertAtEnd(int element) {
    if (size >= MAX_SIZE) {
        printf("Array is full. Cannot insert element.\n");
        return;
    }
```

```
    array[size] = element;
    size++;
}

void insertAtPosition(int element, int position) {
    if (size >= MAX_SIZE) {
        printf("Array is full. Cannot insert element.\n");
        return;
    }

    if (position < 0 || position > size) {
        printf("Invalid position for insertion.\n");
        return;
    }

    for (int i = size; i > position; i--) {
        array[i] = array[i - 1];
    }

    array[position] = element;
    size++;
}

void deleteAtBeginning() {
    if (size == 0) {
        printf("Array is empty. Cannot delete.\n");
        return;
    }

    for (int i = 0; i < size - 1; i++) {
        array[i] = array[i + 1];
    }

    size--;
}

void deleteAtEnd() {
    if (size == 0) {
        printf("Array is empty. Cannot delete.\n");
```

```
        return;
    }

    size--;
}

void deleteAtPosition(int position) {
    if (size == 0) {
        printf("Array is empty. Cannot delete.\n");
        return;
    }

    if (position < 0 || position >= size) {
        printf("Invalid position for deletion.\n");
        return;
    }

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

    size--;
}

int main() {
    int choice, element, position;

    do {
        printf("\nArray Operations Menu:\n");
        printf("1. Insert at beginning\n");
        printf("2. Insert at end\n");
        printf("3. Insert at any position\n");
        printf("4. Delete at beginning\n");
        printf("5. Delete at end\n");
        printf("6. Delete at any position\n");
        printf("7. Display array\n");
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
switch (choice) {
    case 1:
        printf("Enter element to insert at the beginning: ");
        scanf("%d", &element);
        insertAtBeginning(element);
        break;

    case 2:
        printf("Enter element to insert at the end: ");
        scanf("%d", &element);
        insertAtEnd(element);
        break;

    case 3:
        printf("Enter element to insert: ");
        scanf("%d", &element);
        printf("Enter position for insertion: ");
        scanf("%d", &position);
        insertAtPosition(element, position);
        break;

    case 4:
        deleteAtBeginning();
        break;

    case 5:
        deleteAtEnd();
        break;

    case 6:
        printf("Enter position for deletion: ");
        scanf("%d", &position);
        deleteAtPosition(position);
        break;

    case 7:
        displayArray();
        break;

    case 8:
```

```
            printf("Exiting...\n");
            break;

        default:
            printf("Invalid choice. Please enter a valid option.\n");
      }
   } while (choice != 8);

   return 0;
}
```

Linked List:

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

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

struct Node* head = NULL;

void displayList() {
   if (head == NULL) {
      printf("List is empty\n");
      return;
   }

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

void insertAtBeginning(int element) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = element;
   newNode->next = head;
   head = newNode;
}

void insertAtEnd(int element) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = element;
   newNode->next = NULL;

   if (head == NULL) {
      head = newNode;
```

```
        return;
    }

    struct Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

void insertAtPosition(int element, int position) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = element;

    if (position <= 0) {
        printf("Invalid position for insertion\n");
        return;
    }

    if (position == 1) {
        newNode->next = head;
        head = newNode;
        return;
    }

    struct Node* temp = head;
    int count = 1;
    while (temp != NULL && count < position - 1) {
        temp = temp->next;
        count++;
    }

    if (temp == NULL) {
        printf("Invalid position for insertion\n");
    } else {
        newNode->next = temp->next;
        temp->next = newNode;
    }
}
```

```
void deleteAtBeginning() {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }

    struct Node* temp = head;
    head = head->next;
    free(temp);
}

void deleteAtEnd() {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }

    if (head->next == NULL) {
        free(head);
        head = NULL;
        return;
    }

    struct Node* temp = head;
    while (temp->next->next != NULL) {
        temp = temp->next;
    }
    free(temp->next);
    temp->next = NULL;
}

void deleteAtPosition(int position) {
    if (head == NULL || position <= 0) {
        printf("List is empty or invalid position for deletion\n");
        return;
    }

    struct Node* temp = head;
    if (position == 1) {
        head = temp->next;
```

```
        free(temp);
        return;
    }

    int count = 1;
    struct Node* prev = NULL;
    while (temp != NULL && count < position) {
        prev = temp;
        temp = temp->next;
        count++;
    }

    if (temp == NULL) {
        printf("Invalid position for deletion\n");
    } else {
        prev->next = temp->next;
        free(temp);
    }
}

int main() {
    int choice, element, position;

    do {
        printf("\nLinked List Operations Menu:\n");
        printf("1. Insert at beginning\n");
        printf("2. Insert at end\n");
        printf("3. Insert at any position\n");
        printf("4. Delete at beginning\n");
        printf("5. Delete at end\n");
        printf("6. Delete at any position\n");
        printf("7. Display list\n");
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter element to insert at the beginning: ");
                scanf("%d", &element);
```

```
            insertAtBeginning(element);
            break;

        case 2:
            printf("Enter element to insert at the end: ");
            scanf("%d", &element);
            insertAtEnd(element);
            break;

        case 3:
            printf("Enter element to insert: ");
            scanf("%d", &element);
            printf("Enter position for insertion: ");
            scanf("%d", &position);
            insertAtPosition(element, position);
            break;

        case 4:
            deleteAtBeginning();
            break;

        case 5:
            deleteAtEnd();
            break;

        case 6:
            printf("Enter position for deletion: ");
            scanf("%d", &position);
            deleteAtPosition(position);
            break;

        case 7:
            displayList();
            break;

        case 8:
            printf("Exiting...\n");
            break;

        default:
```

```
            printf("Invalid choice. Please enter a valid option.\n");
        }
    } while (choice != 8);

    return 0;
}
```

Singly Linked list:

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

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

struct Node* head = NULL;

void displayList() {
   if (head == NULL) {
      printf("List is empty\n");
      return;
   }

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

void insertAtBeginning(int element) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = element;
   newNode->next = head;
   head = newNode;
}

void insertAtEnd(int element) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = element;
   newNode->next = NULL;

   if (head == NULL) {
      head = newNode;
      return;
   }

   struct Node* temp = head;
   while (temp->next != NULL) {
      temp = temp->next;
```

```
    }
    temp->next = newNode;
}

void insertAtPosition(int element, int position) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = element;

    if (position <= 0) {
        printf("Invalid position for insertion\n");
        return;
    }

    if (position == 1) {
        newNode->next = head;
        head = newNode;
        return;
    }

    struct Node* temp = head;
    int count = 1;
    while (temp != NULL && count < position - 1) {
        temp = temp->next;
        count++;
    }

    if (temp == NULL) {
        printf("Invalid position for insertion\n");
    } else {
        newNode->next = temp->next;
        temp->next = newNode;
    }
}

void deleteAtBeginning() {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }

    struct Node* temp = head;
    head = head->next;
    free(temp);
}

void deleteAtEnd() {
```

```
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }

    if (head->next == NULL) {
        free(head);
        head = NULL;
        return;
    }

    struct Node* temp = head;
    while (temp->next->next != NULL) {
        temp = temp->next;
    }
    free(temp->next);
    temp->next = NULL;
}

void deleteAtPosition(int position) {
    if (head == NULL || position <= 0) {
        printf("List is empty or invalid position for deletion\n");
        return;
    }

    struct Node* temp = head;
    if (position == 1) {
        head = temp->next;
        free(temp);
        return;
    }

    int count = 1;
    struct Node* prev = NULL;
    while (temp != NULL && count < position) {
        prev = temp;
        temp = temp->next;
        count++;
    }

    if (temp == NULL) {
        printf("Invalid position for deletion\n");
    } else {
        prev->next = temp->next;
        free(temp);
    }
```

```
}

int main() {
    int choice, element, position;

    do {
        printf("\nLinked List Operations Menu:\n");
        printf("1. Insert at beginning\n");
        printf("2. Insert at end\n");
        printf("3. Insert at any position\n");
        printf("4. Delete at beginning\n");
        printf("5. Delete at end\n");
        printf("6. Delete at any position\n");
        printf("7. Display list\n");
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter element to insert at the beginning: ");
                scanf("%d", &element);
                insertAtBeginning(element);
                break;

            case 2:
                printf("Enter element to insert at the end: ");
                scanf("%d", &element);
                insertAtEnd(element);
                break;

            case 3:
                printf("Enter element to insert: ");
                scanf("%d", &element);
                printf("Enter position for insertion: ");
                scanf("%d", &position);
                insertAtPosition(element, position);
                break;

            case 4:
                deleteAtBeginning();
                break;

            case 5:
                deleteAtEnd();
                break;
```

```
            case 6:
                printf("Enter position for deletion: ");
                scanf("%d", &position);
                deleteAtPosition(position);
                break;

            case 7:
                displayList();
                break;

            case 8:
                printf("Exiting...\n");
                break;

            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    } while (choice != 8);

    return 0;
}
```

Doubly Linked list:

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

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

struct Node* head = NULL;

void displayList() {
   if (head == NULL) {
      printf("List is empty\n");
      return;
   }

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

void insertAtBeginning(int element) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = element;
   newNode->prev = NULL;

   if (head == NULL) {
      newNode->next = NULL;
   } else {
      newNode->next = head;
      head->prev = newNode;
   }

   head = newNode;
}

void insertAtEnd(int element) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = element;
   newNode->next = NULL;
```

```
    if (head == NULL) {
        newNode->prev = NULL;
        head = newNode;
        return;
    }

    struct Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }

    temp->next = newNode;
    newNode->prev = temp;
}

void insertAtPosition(int element, int position) {
    if (position == 1) {
        insertAtBeginning(element);
        return;
    }

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = element;

    if (head == NULL) {
        printf("List is empty. Invalid position for insertion\n");
        free(newNode);
        return;
    }

    struct Node* temp = head;
    int count = 1;
    while (temp != NULL && count < position - 1) {
        temp = temp->next;
        count++;
    }

    if (temp == NULL) {
        printf("Invalid position for insertion\n");
        free(newNode);
        return;
    }

    newNode->next = temp->next;
    newNode->prev = temp;
```

```
    if (temp->next != NULL) {
        temp->next->prev = newNode;
    }
    temp->next = newNode;
}

void deleteAtPosition(int position) {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }

    struct Node* temp = head;
    if (position == 1) {
        head = head->next;
        if (head != NULL) {
            head->prev = NULL;
        }
        free(temp);
        return;
    }

    int count = 1;
    while (temp != NULL && count < position) {
        temp = temp->next;
        count++;
    }

    if (temp == NULL) {
        printf("Invalid position for deletion\n");
        return;
    }

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

void deleteAtEnd() {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }
```

```
    struct Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }

    if (temp == head) {
        head = NULL;
    } else {
        temp->prev->next = NULL;
    }
    free(temp);
}

void deleteAtBeginning() {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }

    struct Node* temp = head;
    head = head->next;
    if (head != NULL) {
        head->prev = NULL;
    }
    free(temp);
}

int main() {
    int choice, element, position;

    do {
        printf("\nDoubly Linked List Operations Menu:\n");
        printf("1. Insert at beginning\n");
        printf("2. Insert at end\n");
        printf("3. Insert at any position\n");
        printf("4. Delete at any position\n");
        printf("5. Delete at end\n");
        printf("6. Delete at beginning\n");
        printf("7. Display list\n");
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter element to insert at the beginning: ");
```

```
            scanf("%d", &element);
            insertAtBeginning(element);
            break;

        case 2:
            printf("Enter element to insert at the end: ");
            scanf("%d", &element);
            insertAtEnd(element);
            break;

        case 3:
            printf("Enter element to insert: ");
            scanf("%d", &element);
            printf("Enter position for insertion: ");
            scanf("%d", &position);
            insertAtPosition(element, position);
            break;

        case 4:
            printf("Enter position for deletion: ");
            scanf("%d", &position);
            deleteAtPosition(position);
            break;

        case 5:
            deleteAtEnd();
            break;

        case 6:
            deleteAtBeginning();
            break;

        case 7:
            displayList();
            break;

        case 8:
            printf("Exiting...\n");
            break;

        default:
            printf("Invalid choice. Please enter a valid option.\n");
    }
} while (choice != 8);

return 0;}
```

Circular Linked list:

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

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

struct Node* head = NULL;

void displayList() {
   if (head == NULL) {
      printf("List is empty\n");
      return;
   }

   struct Node* temp = head;
   printf("Circular Linked List: ");
   do {
      printf("%d -> ", temp->data);
      temp = temp->next;
   } while (temp != head);
   printf(" (head)\n");
}

void insertAtBeginning(int element) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = element;

   if (head == NULL) {
      head = newNode;
      newNode->next = head;
   } else {
      struct Node* temp = head;
      while (temp->next != head) {
         temp = temp->next;
      }
      newNode->next = head;
      head = newNode;
      temp->next = head;
   }
}

void insertAtEnd(int element) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
    newNode->data = element;

    if (head == NULL) {
        head = newNode;
        newNode->next = head;
    } else {
        struct Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;
    }
}

void insertAtPosition(int element, int position) {
    if (position == 1) {
        insertAtBeginning(element);
        return;
    }

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = element;

    if (head == NULL) {
        printf("List is empty. Invalid position for insertion\n");
        free(newNode);
        return;
    }

    struct Node* temp = head;
    int count = 1;
    while (temp->next != head && count < position - 1) {
        temp = temp->next;
        count++;
    }

    if (count != position - 1) {
        printf("Invalid position for insertion\n");
        free(newNode);
        return;
    }

    newNode->next = temp->next;
    temp->next = newNode;
}
```

```
void deleteAtPosition(int position) {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }

    if (position == 1) {
        if (head->next == head) {
            free(head);
            head = NULL;
        } else {
            struct Node* temp = head;
            while (temp->next != head)
                temp = temp->next;

            struct Node* del = head;
            head = head->next;
            temp->next = head;
            free(del);
        }
    } else {
        struct Node* temp = head;
        int count = 1;
        struct Node* prev = NULL;
        while (temp->next != head && count < position) {
            prev = temp;
            temp = temp->next;
            count++;
        }

        if (count != position) {
            printf("Invalid position for deletion\n");
            return;
        }

        prev->next = temp->next;
        free(temp);
    }
}

void deleteAtEnd() {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }
```

```
    if (head->next == head) {
        free(head);
        head = NULL;
    } else {
        struct Node* temp = head;
        struct Node* prev = NULL;
        while (temp->next != head) {
            prev = temp;
            temp = temp->next;
        }
        prev->next = head;
        free(temp);
    }
}

void deleteAtBeginning() {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }

    if (head->next == head) {
        free(head);
        head = NULL;
    } else {
        struct Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }
        struct Node* del = head;
        head = head->next;
        temp->next = head;
        free(del);
    }
}

int main() {
    int choice, element, position;

    do {
        printf("\nCircular Linked List Operations Menu:\n");
        printf("1. Insert at beginning\n");
        printf("2. Insert at end\n");
        printf("3. Insert at any position\n");
        printf("4. Delete at any position\n");
```

```
printf("5. Delete at end\n");
printf("6. Delete at beginning\n");
printf("7. Display list\n");
printf("8. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
   case 1:
      printf("Enter element to insert at the beginning: ");
      scanf("%d", &element);
      insertAtBeginning(element);
      break;

   case 2:
      printf("Enter element to insert at the end: ");
      scanf("%d", &element);
      insertAtEnd(element);
      break;

   case 3:
      printf("Enter element to insert: ");
      scanf("%d", &element);
      printf("Enter position for insertion: ");
      scanf("%d", &position);
      insertAtPosition(element, position);
      break;

   case 4:
      printf("Enter position for deletion: ");
      scanf("%d", &position);
      deleteAtPosition(position);
      break;

   case 5:
      deleteAtEnd();
      break;

   case 6:
      deleteAtBeginning();
      break;

   case 7:
      displayList();
      break;
```

```
            case 8:
                printf("Exiting...\n");
                break;

            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    } while (choice != 8);

    return 0;
}
```

Stack using Arrays:

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

#define MAX_SIZE 100

int stack[MAX_SIZE];
int top = -1;

// Function to check if the stack is empty
int isEmpty() {
   return top == -1;
}

// Function to check if the stack is full
int isFull() {
   return top == MAX_SIZE - 1;
}

// Function to push (insert) an element onto the stack
void push(int element) {
   if (isFull()) {
      printf("Stack overflow, cannot push\n");
      return;
   }

   stack[++top] = element;
   printf("%d pushed to the stack\n", element);
}

// Function to pop (remove) an element from the stack
int pop() {
   if (isEmpty()) {
      printf("Stack underflow, cannot pop\n");
      return -1;
   }

   int popped = stack[top--];
   printf("Popped element: %d\n", popped);
   return popped;
}

// Function to peek at the top element of the stack without removing it
int peek() {
   if (isEmpty()) {
      printf("Stack is empty, cannot peek\n");
```

```
        return -1;
    }

    return stack[top];
}

// Function to display the stack elements
void displayStack() {
    if (isEmpty()) {
        printf("Stack is empty\n");
        return;
    }

    printf("Stack elements: ");
    for (int i = 0; i <= top; i++) {
        printf("%d ", stack[i]);
    }
    printf("\n");
}

int main() {
    int choice, element;

    do {
        printf("\nStack Operations Menu:\n");
        printf("1. Push (Insert) Element\n");
        printf("2. Pop (Remove) Element\n");
        printf("3. Peek (View Top) Element\n");
        printf("4. Display Stack\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter element to push: ");
                scanf("%d", &element);
                push(element);
                break;

            case 2:
                pop();
                break;

            case 3:
                printf("Top element: %d\n", peek());
```

```
            break;

        case 4:
            displayStack();
            break;

        case 5:
            printf("Exiting...\n");
            break;

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

   return 0;
}
```

Stack using linked lists:

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

// Define a structure for the stack node
struct Node {
    int data;
    struct Node* next;
};

struct Node* top = NULL; // Initialize an empty stack

// Function to check if the stack is empty
int isEmpty() {
    return top == NULL;
}

// Function to push (insert) an element onto the stack
void push(int element) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Memory allocation error\n");
        return;
    }
    newNode->data = element;
    newNode->next = top;
    top = newNode;
    printf("%d pushed to the stack\n", element);
}

// Function to pop (remove) an element from the stack
int pop() {
    if (isEmpty()) {
        printf("Stack is empty, cannot pop\n");
        return -1;
    }
    struct Node* temp = top;
    int popped = temp->data;
    top = top->next;
    free(temp);
    return popped;
}

// Function to peek at the top element of the stack without removing it
```

```
int peek() {
    if (isEmpty()) {
        printf("Stack is empty, cannot peek\n");
        return -1;
    }
    return top->data;
}

// Function to display the stack elements
void displayStack() {
    if (isEmpty()) {
        printf("Stack is empty\n");
        return;
    }
    struct Node* temp = top;
    printf("Stack elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    int choice, element;

    do {
        printf("\nStack Operations Menu:\n");
        printf("1. Push (Insert) Element\n");
        printf("2. Pop (Remove) Element\n");
        printf("3. Peek (View Top) Element\n");
        printf("4. Display Stack\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter element to push: ");
                scanf("%d", &element);
                push(element);
                break;

            case 2:
                printf("Popped element: %d\n", pop());
                break;
```

```
            case 3:
                printf("Top element: %d\n", peek());
                break;

            case 4:
                displayStack();
                break;

            case 5:
                printf("Exiting...\n");
                break;

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

    return 0;
}
```

Queue using Arrays:

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

#define MAX_SIZE 100

int queue[MAX_SIZE];
int front = -1, rear = -1;

// Function to check if the queue is empty
int isEmpty() {
   return front == -1 || front > rear;
}

// Function to check if the queue is full
int isFull() {
   return rear == MAX_SIZE - 1;
}

// Function to enqueue (insert) an element into the queue
void enqueue(int element) {
   if (isFull()) {
      printf("Queue is full, cannot enqueue\n");
      return;
   }

   if (front == -1) {
      front = 0;
   }

   rear++;
   queue[rear] = element;
   printf("%d enqueued to the queue\n", element);
}

// Function to dequeue (remove) an element from the queue
int dequeue() {
   if (isEmpty()) {
      printf("Queue is empty, cannot dequeue\n");
      return -1;
   }

   int dequeued = queue[front];
   front++;
   printf("Dequeued element: %d\n", dequeued);
   return dequeued;
```

```
}

// Function to display the queue elements
void displayQueue() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }

    printf("Queue elements: ");
    for (int i = front; i <= rear; i++) {
        printf("%d ", queue[i]);
    }
    printf("\n");
}

int main() {
    int choice, element;

    do {
        printf("\nQueue Operations Menu:\n");
        printf("1. Enqueue (Insert) Element\n");
        printf("2. Dequeue (Remove) Element\n");
        printf("3. Display Queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter element to enqueue: ");
                scanf("%d", &element);
                enqueue(element);
                break;

            case 2:
                dequeue();
                break;

            case 3:
                displayQueue();
                break;

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

```
            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    } while (choice != 4);

    return 0;
}
```

Queue using linked lists:

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

// Define a structure for the queue node
struct Node {
    int data;
    struct Node* next;
};

struct Node* front = NULL;
struct Node* rear = NULL;

// Function to check if the queue is empty
int isEmpty() {
    return front == NULL;
}

// Function to enqueue (insert) an element into the queue
void enqueue(int element) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Memory allocation error\n");
        return;
    }
    newNode->data = element;
    newNode->next = NULL;

    if (rear == NULL) {
        front = rear = newNode;
        return;
    }

    rear->next = newNode;
    rear = newNode;
    printf("%d enqueued to the queue\n", element);
}

// Function to dequeue (remove) an element from the queue
int dequeue() {
    if (isEmpty()) {
        printf("Queue is empty, cannot dequeue\n");
        return -1;
    }
    struct Node* temp = front;
    int dequeued = temp->data;
```

```
    front = front->next;

    if (front == NULL) {
        rear = NULL;
    }

    free(temp);
    return dequeued;
}

// Function to display the queue elements
void displayQueue() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }

    struct Node* temp = front;
    printf("Queue elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    int choice, element;

    do {
        printf("\nQueue Operations Menu:\n");
        printf("1. Enqueue (Insert) Element\n");
        printf("2. Dequeue (Remove) Element\n");
        printf("3. Display Queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter element to enqueue: ");
                scanf("%d", &element);
                enqueue(element);
                break;

            case 2:
```

```
            printf("Dequeued element: %d\n", dequeue());
            break;

        case 3:
            displayQueue();
            break;

        case 4:
            printf("Exiting...\n");
            break;

        default:
            printf("Invalid choice. Please enter a valid option.\n");
      }
   } while (choice != 4);

   return 0;
}
```

BFS CODE

```
// BFS algorithm in c
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40
struct queue {
  int items[SIZE];
  int front;
  int rear;
};
struct queue* createQueue();
void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);
struct node {
  int vertex;
  struct node* next;
};
struct node* createNode(int);
struct Graph {
  int numVertices;
  struct node** adjLists;
  int* visited;
};
// BFS algorithm
```

```
void bfs(struct Graph* graph, int startVertex) {
  struct queue* q = createQueue();
  graph->visited[startVertex] = 1;
  enqueue(q, startVertex);
  while (!isEmpty(q)) {
    printQueue(q);
    int currentVertex = dequeue(q);
    printf("Visited %d\n", currentVertex);
    struct node* temp = graph->adjLists[currentVertex];
    while (temp) {
      int adjVertex = temp->vertex;
      if (graph->visited[adjVertex] == 0) {
        graph->visited[adjVertex] = 1;
        enqueue(q, adjVertex);
      }
      temp = temp->next;
    }
  }
}
// Creating a node
struct node* createNode(int v) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}
// Creating a 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;
}
// Create a queue
struct queue* createQueue() {
  struct queue* q = malloc(sizeof(struct queue));
  q->front = -1;
```

```
  q->rear = -1;
  return q;
}
// Check if the queue is empty
int isEmpty(struct queue* q) {
  if (q->rear == -1)
    return 1;
  else
    return 0;
}
// Adding elements into queue
void enqueue(struct queue* q, int value) {
  if (q->rear == SIZE - 1)
    printf("\nQueue is Full!!");
  else {
    if (q->front == -1)
      q->front = 0;
    q->rear++;
    q->items[q->rear] = value;
  }
}
// Removing elements from queue
int dequeue(struct queue* q) {
  int item;
  if (isEmpty(q)) {
    printf("Queue is empty");
    item = -1;
```

```
  } else {
    item = q->items[q->front];
    q->front++;
    if (q->front > q->rear) {
      printf("Resetting queue ");
      q->front = q->rear = -1;
    }
  }
  return item;
}
// Print the queue
void printQueue(struct queue* q) {
  int i = q->front;
  if (isEmpty(q)) {
    printf("Queue is empty");
  } else {
    printf("\nQueue contains \n");
    for (i = q->front; i < q->rear + 1; i++) {
      printf("%d ", q->items[i]);
    }
  }
}
int main() {
  struct Graph* graph = createGraph(6);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 2);
```

```
  addEdge(graph, 1, 4);
  addEdge(graph, 1, 3);
  addEdge(graph, 2, 4);
  addEdge(graph, 3, 4);
  bfs(graph, 0);
  return 0;
}
```

DFS CODE

```
// 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;
}
```

TREE TRAVERSAL CODE

```
// Tree traversal in C
#include <stdio.h>
#include <stdlib.h>
struct node {
  int item;
  struct node* left;
  struct node* right;
};
// Inorder traversal
void inorderTraversal(struct node* root) {
  if (root == NULL) return;
  inorderTraversal(root->left);
  printf("%d ->", root->item);
```

```
  inorderTraversal(root->right);
}
// preorderTraversal traversal
void preorderTraversal(struct node* root) {
  if (root == NULL) return;
  printf("%d ->", root->item);
  preorderTraversal(root->left);
  preorderTraversal(root->right);
}
// postorderTraversal traversal
void postorderTraversal(struct node* root) {
  if (root == NULL) return;
  postorderTraversal(root->left);
  postorderTraversal(root->right);
  printf("%d ->", root->item);
}
// Create a new Node
struct node* createNode(value) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->item = value;
  newNode->left = NULL;
  newNode->right = NULL;
  return newNode;
}
// Insert on the left of the node
struct node* insertLeft(struct node* root, int value) {
  root->left = createNode(value);
```

```
  return root->left;
}
// Insert on the right of the node
struct node* insertRight(struct node* root, int value) {
  root->right = createNode(value);
  return root->right;
}
int main() {
  struct node* root = createNode(1);
  insertLeft(root, 12);
  insertRight(root, 9);
  insertLeft(root->left, 5);
  insertRight(root->left, 6);
  printf("Inorder traversal \n");
  inorderTraversal(root);
  printf("\nPreorder traversal \n");
  preorderTraversal(root);
  printf("\nPostorder traversal \n");
  postorderTraversal(root);
}
```

LINEAR SEARCH

```
// Linear Search in C
#include <stdio.h>
int search(int array[], int n, int x) {
  // Going through array sequencially
  for (int i = 0; i < n; i++)
```

```
    if (array[i] == x)
      return i;
  return -1;
}
int main() {
  int array[] = {2, 4, 0, 1, 9};
  int x = 1;
  int n = sizeof(array) / sizeof(array[0]);
  int result = search(array, n, x);
  (result == -1) ? printf("Element not found") : printf("Element found at index: %d",
result);
}
```

BINARY SEARCH

```
// Binary Search in C
#include <stdio.h>
int binarySearch(int array[], int x, int low, int high) {
  if (high >= low) {
    int mid = low + (high - low) / 2;
    // If found at mid, then return it
    if (x == array[mid])
      return mid;
    // Search the right half
    if (x > array[mid])
      return binarySearch(array, x, mid + 1, high);
    // Search the left half
    return binarySearch(array, x, low, mid - 1);
  }
```

```
  return -1;
}
int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x = 4;
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
}
```

SELECTION SORT

```
// Selection sort in C
#include <stdio.h>
// function to swap the the position of two elements
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
void selectionSort(int array[], int size) {
  for (int step = 0; step < size - 1; step++) {
    int min_idx = step;
    for (int i = step + 1; i < size; i++) {
      // To sort in descending order, change > to < in this line.
```

```
      // Select the minimum element in each loop.
      if (array[i] < array[min_idx])
        min_idx = i;
    }
    // put min at the correct position
    swap(&array[min_idx], &array[step]);
  }
}
// function to print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}
// driver code
int main() {
  int data[] = {20, 12, 10, 15, 2};
  int size = sizeof(data) / sizeof(data[0]);
  selectionSort(data, size);
  printf("Sorted array in Acsending Order:\n");
  printArray(data, size);
}
```

INSERTION SORT

```
// Insertion sort in C
#include <stdio.h>
```

```
// Function to print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; i++) {
    printf("%d ", array[i]);
  }
  printf("\n");
}
void insertionSort(int array[], int size) {
  for (int step = 1; step < size; step++) {
    int key = array[step];
    int j = step - 1;
    // Compare key with each element on the left of it until an element smaller than
    // it is found.
    // For descending order, change key<array[j] to key>array[j].
    while (j >=0 && key < array[j]) {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = key;
  }
}
// Driver code
int main() {
  int data[] = {9, 5, 1, 4, 3};
  int size = sizeof(data) / sizeof(data[0]);
  insertionSort(data, size);
  printf("Sorted array in ascending order:\n");
```

```
  printArray(data, size);
}
```

BUBBLE SORT

```
// Bubble sort in C
#include <stdio.h>
// perform the bubble sort
void bubbleSort(int array[], int size) {
  // loop to access each array element
  for (int step = 0; step < size - 1; ++step) {
    // loop to compare array elements
    for (int i = 0; i < size - step - 1; ++i) {
      // compare two adjacent elements
      // change > to < to sort in descending order
      if (array[i] > array[i + 1]) {
        // swapping occurs if elements
        // are not in the intended order
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}
// print array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
```

```
    printf("%d  ", array[i]);
  }
  printf("\n");
}
int main() {
  int data[] = {-2, 45, 0, 11, -9};
  // find the array's length
  int size = sizeof(data) / sizeof(data[0]);
  bubbleSort(data, size);
  printf("Sorted Array in Ascending Order:\n");
  printArray(data, size);    }
```