Guru Gobind Singh Indraprastha University East Delhi Campus, USAR

Data Structures Lab (ARI 254)

Submitted by: Sujal Singh
Enrolment Number: 04119051723
Batch: IIOT - B1
Submitted to: Dr. Anshul Bhatia

CONTENTS

5	Write a program to implement stack using arrays and linked lists.		
6	Write a program to reverse a sentence/string using stack.		
7	Write a program to check for balanced parenthesis in a given expression.		
8	Write a program to convert infix expression to prefix and postfix expression.		
9	Write a program to implement Linear Queue using Array and Linked Lists.		
10	Write a program to implement Circular Queue using Array and Linked Lists.		
11	Write a program to implement Doubly Ended Queue using Array and Linked Lists.		
12	Write a Program to implement Binary Search Tree operations.		
13	Write a program to implement Bubble Sort, Selection Sort, Heap Sort, Quick Sort, Merge Sort and Insertion Sort algorithm.		
14	Write C Programs by using Graph data structure for the following problem domains: a. Graph Traversal: BFS b. Graph Traversal: DFS		

Experiment No 1(a)

Aim: Create an array of integer with size n. Return the difference between the largest and the smallest value inside that array.

```
#include <stdio.h>
int main() {
   int arr[] = {12, 22, 33, 31, 23};
   int n = sizeof(arr) / sizeof(arr[0]); // Get the size of the array
   int max = arr[0];
   int min = arr[0];

   for (int i = 1; i < n; i++) {
      if (arr[i] > max) {
         max = arr[i];
      }
      if (arr[i] < min) {
         min = arr[i];
      }
   }

   printf("The largest element is: %d\n", max);
   printf("The smallest element is: %d\n", min);
   printf("The difference: %d\n", max - min);

   return 0;
}
```

Output:

```
The largest element is: 33
The smallest element is: 12
The difference between the largest and smallest elements is: 21
```

Experiment No 1(b)

Aim: Initializes an array with ten random integers and then prints four lines of output, containing: Every element at an even index, Every odd element, All elements in reverse order, Only the first and last element

```
#include <stdio.h>
int main() {
   int arr[] = {232, 22, 332, 12, 2212, 332, 14, 24, 44, 43};
   int n = sizeof(arr) / sizeof(arr[0]); // Get the size of the array
   printf("The even index elements are: ");
   for (int i = 0; i < n; i++) {
      if (i % 2 == 0) {
         printf("%d ", arr[i]);
      }
   }
   printf("\nThe odd index elements are: ");
   for (int i = 0; i < n; i++) {
      if (arr[i] % 2 != 0) {
         printf("%d ", arr[i]);
      }
   }
   printf("\nThe reverse of the array is: ");
   for (int i = n - 1; i >= 0; i--) {
      printf("%d ", arr[i]);
   }
   printf("\nThe first element of the array is: %d", arr[0]);
   printf("\nThe last element of the array is: %d", arr[n - 1]);
 return 0;
}
```

Output:

```
The even index elements are: 232 332 2212 14 44
The odd index elements are: 43
The reverse of the array is: 43 44 24 14 332 2212 12 332 22 232
The first element of the array is: 232The last element of the array is: 43
```

Experiment No 1(c)

Aim: Consider an integer array of size 5 and display the following: Sum of all the elements, Sum of alternate elements in the array, and second highest element in the array

```
#include <stdio.h>
int main() {
   int arr[] = {23, 21, 22, 41, 44, 32, 64, 27, 42, 45, 63, 72};
   int n = sizeof(arr) / sizeof(arr[0]); // Get the size of the array
   int sum = 0, sum1 = 0;
   for (int i = 0; i < n; i++) {
      sum += arr[i];
   }
   for (int i = 0; i < n; i += 2) {
      sum1 += arr[i];
   }
   int max = arr[0];
   int secondMax = arr[0];
 for (int i = 0; i < n; i++) {
      if (arr[i] > max) {
         secondMax = max;
         max = arr[i];
      } else if (arr[i] > secondMax && arr[i] != max) {
         secondMax = arr[i];
      }
   }
   printf("Sum of all elements: %d\n", sum);
   printf("Sum of alternate elements: %d\n", sum1);
   printf("Second highest element: %d\n", secondMax);
return 0;
}
```

Output:

```
Sum of all elements: 496
Sum of alternate elements: 258
Second highest element: 64
```

Experiment No 2

Aim: Write a program to create a singly linked list of n nodes and perform: Insertion at the beginning, Insertion at the end, Insertion at a specific location, Deletion at the beginning, Deletion at the end, Deletion At a specific location

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
   int data;
   struct Node* next;
};
void insertBeginning(struct Node** head, int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->next = *head;
   *head = newNode;
}
void insertEnd(struct Node** head, int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   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(struct Node** head, int value, int pos) {
   if (pos < 1) {
      printf("Invalid position!\n");
      return;
   }
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
  if (pos == 1) {
      newNode->next = *head;
      *head = newNode;
      return;
   }
   struct Node* temp = *head;
   for (int i = 1; i < pos - 1 && temp != NULL; i++)
      temp = temp->next;
   if (temp == NULL) {
      printf("Position out of range!\n");
```

```
        return;
    }
    newNode->next = temp->next;
    temp->next = newNode;
}
void deleteBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}
void deleteEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }
    if ((*head)->next == NULL) {
        free(*head);
        *head = NULL;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL && temp->next->next != NULL)
        temp = temp->next;
    free(temp->next);
    temp->next = NULL;
}
void deleteAtPosition(struct Node** head, int pos) {
    if (*head == NULL || pos < 1) {
        printf("Invalid position or empty list.\n");
        return;
    }
    if (pos == 1) {
        deleteBeginning(head);
        return;
    }
    struct Node* temp = *head;
    for (int i = 1; i < pos - 1 && temp != NULL; i++)
        temp = temp->next;
    if (temp == NULL || temp->next == NULL) {
        printf("Position out of range!\n");
        return;
    }
    struct Node* temp2 = temp->next;
    temp->next = temp->next->next;
    free(temp2);
}
void display(struct Node* head) {
```

```
    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");
}
int main() {
    struct Node* head = NULL;
    int choice, value, position;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert at beginning\n");
        printf("2. Insert at end\n");
        printf("3. Insert at position\n");
        printf("4. Delete at beginning\n");
        printf("5. Delete at end\n");
        printf("6. Delete at position\n");
        printf("7. Display\n");
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                insertBeginning(&head, value);
                break;
            case 2:
                printf("Enter value: ");
                scanf("%d", &value);
                insertEnd(&head, value);
                break;
            case 3:
                printf("Enter value and position: ");
                scanf("%d %d", &value, &position);
                insertAtPosition(&head, value, position);
                break;
            case 4:
                deleteBeginning(&head);
                break;
            case 5:
                deleteEnd(&head);
                break;
```

```
        case 6:
          printf("Enter position: ");
          scanf("%d", &position);
          deleteAtPosition(&head, position);
          break;
        case 7:
          display(head);
          break;
        case 8:
          exit(0);
        default:
          printf("Invalid choice.\n");
      }
    }
    return 0;
}
```

Output:

```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at beginning
5. Delete at end
6. Delete at position
7. Display
8. Exit
Enter your choice: 1
Enter value: 34

Menu:
1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at beginning
5. Delete at end
6. Delete at position
7. Display
8. Exit
Enter your choice: 7
Linked List: 34 -> NULL
```

Experiment No 3

Aim: Write a program to create a doubly linked list of n nodes and perform: Insertion at the beginning, Insertion at the end, Insertion at a specific location, Deletion at the beginning, Deletion at the end, Deletion At a specific location

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
  int data;
  struct Node* prev;
  struct Node* next;
};
void createList(int n, struct Node** head) {
  for (int i = 0; i < n; i++) {
    int data;
    printf("Enter data for node %d: ", i + 1);
    scanf("%d", &data);
    insertAtEnd(head, data);
  }
}
void insertAtBeginning(struct Node** head, int data) {
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = data;
  newNode->prev = NULL;
  newNode->next = *head;

  if (*head != NULL) {
    (*head)->prev = newNode;
  }
  *head = newNode;
}

void insertAtEnd(struct Node** head, int data) {
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = data;
  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(struct Node** head, int data, int pos) {
  if (pos <= 1) {
    insertAtBeginning(head, data);
    return;
  }
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = data;
  struct Node* temp = *head;
  for (int i = 1; i < pos - 1 && temp != NULL; i++)
    temp = temp->next;
  if (temp == NULL) {
    insertAtEnd(head, data);
  } else {
    newNode->next = temp->next;
    newNode->prev = temp;

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

    temp->next = newNode;
  }
}

void deleteAtBeginning(struct Node** head) {
  if (*head == NULL) {
    printf("List is empty.\n");
    return;
  }
  struct Node* temp = *head;
  *head = (*head)->next;
  if (*head != NULL)
    (*head)->prev = NULL;
  free(temp);
}
void deleteAtEnd(struct Node** head) {
  if (*head == NULL) {
    printf("List is empty.\n");
    return;
  }
  if ((*head)->next == NULL) {
    free(*head);
    *head = NULL;
    return;
  }
  struct Node* temp = *head;
  while (temp->next != NULL)
    temp = temp->next;
  temp->prev->next = NULL;
  free(temp);
}
```

```
void deleteAtPosition(struct Node** head, int pos) {
   if (*head == NULL) {
      printf("List is empty.\n");
      return;
   }
   if (pos == 1) {
      deleteAtBeginning(head);
      return;
   }
   struct Node* temp = *head;
   for (int i = 1; i < pos && temp != NULL; i++)
      temp = temp->next;
   if (temp == NULL) {
      printf("Position out of range.\n");
      return;
   }
   if (temp->next != NULL)
      temp->next->prev = temp->prev;
   if (temp->prev != NULL)
      temp->prev->next = temp->next;
   free(temp);
}
void display(struct Node* head) {
   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");
}
int main() {
   struct Node* head = NULL;
   int n, value, position;
   printf("Enter number of nodes: ");
   scanf("%d", &n);
   createList(n, &head);
   display(head);
   insertAtBeginning(&head, 10);
   display(head);
   insertAtEnd(&head, 99);
   display(head);
   insertAtPosition(&head, 77, 3);
   display(head);
   deleteAtBeginning(&head);
   display(head);
   deleteAtEnd(&head);
```

```
    display(head);
    deleteAtPosition(&head, 2);
    display(head);
 return 0;
}
```

Output:

```
Enter number of nodes: 2
Enter data for node 1: 25
Enter data for node 2: 32
Doubly Linked List: 25 <-> 32 <-> NULL
Doubly Linked List: 10 <-> 25 <-> 32 <-> NULL
Doubly Linked List: 10 <-> 25 <-> 32 <-> 99 <-> NULL
Doubly Linked List: 10 <-> 25 <-> 77 <-> 32 <-> 99 <-> NULL
Doubly Linked List: 25 <-> 77 <-> 32 <-> 99 <-> NULL
Doubly Linked List: 25 <-> 77 <-> 32 <-> NULL
Doubly Linked List: 25 <-> 32 <-> NULL
```

Experiment No 4

Aim: Write a program to create a singly circular and doubly linked list of n nodes and perform:
Insertion at the beginning, Insertion at the end, Insertion at a specific location, Deletion at the beginning, Deletion at the end, Deletion At a specific location

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
  int data;
  struct Node* next;
};
void insertBeginning(struct Node** head, int data) {
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = data;
  if (*head == NULL) {
    newNode->next = newNode;  // Point to itself if the list is empty
    *head = newNode;
    return;
  }
  struct Node* temp = *head;
  while (temp->next != *head) {
    temp = temp->next;
  }
  newNode->next = *head;
  temp->next = newNode;
  *head = newNode;
}
void insertEnd(struct Node** head, int data) {
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = data;
  if (*head == NULL) {
    newNode->next = newNode;  // Point to itself if the list is empty
    *head = newNode;
    return;
  }
  struct Node* temp = *head;
  while (temp->next != *head) {
    temp = temp->next;
  }
  temp->next = newNode;
  newNode->next = *head;
}
void insertAtPosition(struct Node** head, int data, int position) {
  if (position <= 0) {
    printf("Invalid position.\n");
    return;
```

```
    }
    if (position == 1) {
        insertBeginning(head, data);
        return;
    }

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    struct Node* temp = *head;
    int count = 1;
    while (count < position - 1 && temp->next != *head) {
        temp = temp->next;
        count++;
    }
    if (count != position - 1) {
        printf("Position out of bounds.\n");
        return;
    }
    newNode->next = temp->next;
    temp->next = newNode;
}
void deleteBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }
    if ((*head)->next == *head) {
        free(*head);
        *head = NULL;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != *head) {
        temp = temp->next;
    }
    temp->next = (*head)->next;
    free(*head);
    *head = temp->next;
}
void deleteEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }
    if ((*head)->next == *head) {
        free(*head);
        *head = NULL;
        return;
    }
    struct Node* temp = *head;
    struct Node* prev = NULL;
```

```
    while (temp->next != *head) {
        prev = temp;
        temp = temp->next;
    }
    prev->next = *head;
    free(temp);
}
void deleteAtPosition(struct Node** head, int position) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }
    if (position <= 0) {
        printf("Invalid position.\n");
        return;
    }
    if (position == 1) {
        deleteBeginning(head);
        return;
    }
    struct Node* temp = *head;
    struct Node* prev = NULL;
    int count = 1;
    while (count < position && temp->next != *head) {
        prev = temp;
        temp = temp->next;
        count++;
    }
    if (count != position) {
        printf("Position out of bounds.\n");
        return;
    }
    prev->next = temp->next;
    free(temp);
}
void display(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = head;
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("(Back to Head)\n");
}
int main() {
    struct Node* head = NULL;
    int value, position;
    insertEnd(&head, 10);
```

```
    insertEnd(&head, 20);
    insertEnd(&head, 30);
    display(head);
    insertBeginning(&head, 5);
    display(head);
    insertAtPosition(&head, 15, 3);
    display(head);
    deleteBeginning(&head);
    display(head);
    deleteEnd(&head);
    display(head);
    deleteAtPosition(&head, 2);
    display(head);
    return 0;
}
```

```
10 -> 20 -> 30 -> (Back to Head)
5 -> 10 -> 20 -> 30 -> (Back to Head)
5 -> 10 -> 15 -> 20 -> 30 -> (Back to Head)
10 -> 15 -> 20 -> 30 -> (Back to Head)
10 -> 15 -> 20 -> (Back to Head)
10 -> 20 -> (Back to Head)
```

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

struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
void insertBeginning(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;

    if (*head == NULL) {
        newNode->next = newNode->prev = newNode; // Point to itself if list is empty
        *head = newNode;
        return;
    }

    struct Node* tail = (*head)->prev;
    newNode->next = *head;
    newNode->prev = tail;
    (*head)->prev = newNode;
    tail->next = newNode;
    *head = newNode;
}
void insertEnd(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
    newNode->data = data;

    if (*head == NULL) {
        newNode->next = newNode->prev = newNode; // Point to itself if list is empty
        *head = newNode;
        return;
    }

    struct Node* tail = (*head)->prev;
    newNode->next = *head;
    newNode->prev = tail;
    tail->next = newNode;
    (*head)->prev = newNode;
}
void insertAtPosition(struct Node** head, int data, int position) {
    if (position <= 0) {
        printf("Invalid position.\n");
        return;
    }
    if (position == 1) {
        insertBeginning(head, data);
        return;
    }

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    struct Node* temp = *head;
    int count = 1;

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

    if (count != position - 1) {
        printf("Position out of bounds.\n");
        return;
    }

    struct Node* nextNode = temp->next;
    temp->next = newNode;
    newNode->prev = temp;
    newNode->next = nextNode;
    nextNode->prev = newNode;
}
void deleteBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }
    if ((*head)->next == *head) {
```

```
    free(*head);
    *head = NULL;
    return;
  }

  struct Node* tail = (*head)->prev;
  *head = (*head)->next;
  (*head)->prev = tail;
  tail->next = *head;
  free((*head)->prev);
}
void deleteEnd(struct Node** head) {
  if (*head == NULL) {
    printf("List is empty.\n");
    return;
  }
  if ((*head)->next == *head) {
    free(*head);
    *head = NULL;
    return;
  }

  struct Node* tail = (*head)->prev;
  struct Node* newTail = tail->prev;
  newTail->next = *head;
  (*head)->prev = newTail;
  free(tail);
}
void deleteAtPosition(struct Node** head, int position) {
  if (*head == NULL || position <= 0) {
    printf("Invalid operation.\n");
    return;
  }
  if (position == 1) {
    deleteBeginning(head);
    return;
  }

  struct Node* temp = *head;
  int count = 1;

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

  if (count != position) {
    printf("Position out of bounds.\n");
    return;
  }
```

```
    struct Node* prevNode = temp->prev;
    struct Node* nextNode = temp->next;
    prevNode->next = nextNode;
    nextNode->prev = prevNode;
    free(temp);
}

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

    struct Node* temp = head;
    printf("Forward: ");
    do {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("(Back to Head)\n");
}

int main() {
    struct Node* head = NULL;
    int value, position;
    insertEnd(&head, 10);
    insertEnd(&head, 20);
    insertEnd(&head, 30);
    display(head);
    insertBeginning(&head, 5);
    display(head);
    insertAtPosition(&head, 15, 3);
    display(head);
    deleteBeginning(&head);
    display(head);
    deleteEnd(&head);
    display(head);
    deleteAtPosition(&head, 2);
    display(head);
    return 0;
}
```

```
Forward: 10 <-> 20 <-> 30 <-> (Back to Head)
Forward: 5 <-> 10 <-> 20 <-> 30 <-> (Back to Head)
Forward: 5 <-> 10 <-> 15 <-> 20 <-> 30 <-> (Back to Head)
Forward: 10 <-> 15 <-> 20 <-> 30 <-> (Back to Head)
Forward: 10 <-> 15 <-> 20 <-> (Back to Head)
Forward: 10 <-> 20 <-> (Back to Head)
```

Experiment No 5

Aim: Write a program to implement stack using arrays and linked lists.

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

// Stack using Array
#define MAX_SIZE 5

// StackArray structure
struct StackArray {
  int top;
  int stack[MAX_SIZE];
};

// Function to initialize the stack (Array version)
void initStackArray(struct StackArray* stack) {
  stack->top = -1;
}

// Function to push an element into the array stack
void pushArray(struct StackArray* stack, int data) {
  if (stack->top == MAX_SIZE - 1) {
    printf("Stack Overflow\n");
    return;
  }
  stack->stack[++stack->top] = data;
}

// Function to pop an element from the array stack
void popArray(struct StackArray* stack) {
  if (stack->top == -1) {
    printf("Stack Underflow\n");
    return;
  }
  printf("Popped: %d\n", stack->stack[stack->top--]);
}

// Function to peek the top element of the array stack
void peekArray(struct StackArray* stack) {
  if (stack->top != -1) {
    printf("Top: %d\n", stack->stack[stack->top]);
  } else {
    printf("Stack is empty\n");
  }
}
void displayArray(struct StackArray* stack) {
  if (stack->top == -1) {
    printf("Stack is empty.\n");
```

```
        return;
    }
    printf("Stack (Array): ");
    for (int i = stack->top; i >= 0; i--) {
        printf("%d ", stack->stack[i]);
    }
    printf("\n");
}
struct Node {
    int data;
    struct Node* next;
};
struct StackLinkedList {
    struct Node* top;
};
void initStackLinkedList(struct StackLinkedList* stack) {
    stack->top = NULL;
}
void pushLinkedList(struct StackLinkedList* stack, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = stack->top;
    stack->top = newNode;
}
void popLinkedList(struct StackLinkedList* stack) {
    if (stack->top == NULL) {
        printf("Stack Underflow\n");
        return;
    }
    printf("Popped: %d\n", stack->top->data);
    struct Node* temp = stack->top;
    stack->top = stack->top->next;
    free(temp);
}
void peekLinkedList(struct StackLinkedList* stack) {
    if (stack->top != NULL) {
        printf("Top: %d\n", stack->top->data);
    } else {
        printf("Stack is empty\n");
    }
}
void displayLinkedList(struct StackLinkedList* stack) {
    if (stack->top == NULL) {
        printf("Stack is empty.\n");
        return;
    }
    struct Node* temp = stack->top;
    printf("Stack (Linked List): ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
```

```
    }
    printf("\n");
}
int main() {
    struct StackArray arrayStack;
    struct StackLinkedList linkedStack;
    initStackArray(&arrayStack);
    initStackLinkedList(&linkedStack);
    printf("Pushing elements to both stacks:\n");
    for (int i = 1; i <= 5; i++) {
        pushArray(&arrayStack, i * 10);
        pushLinkedList(&linkedStack, i * 10);
    }
    displayArray(&arrayStack);
    displayLinkedList(&linkedStack);
    printf("\nPop from Array Stack:\n");
    popArray(&arrayStack);
    displayArray(&arrayStack);

    printf("\nPop from Linked List Stack:\n");
    popLinkedList(&linkedStack);
    displayLinkedList(&linkedStack);

    printf("\nPeek operations:\n");
    peekArray(&arrayStack);
    peekLinkedList(&linkedStack);

    return 0;
}
```

Output:

```
Pushing elements to both stacks:
Stack (Array): 50 40 30 20 10
Stack (Linked List): 50 40 30 20 10

Pop from Array Stack:
Popped: 50
Stack (Array): 40 30 20 10

Pop from Linked List Stack:
Popped: 50
Stack (Linked List): 40 30 20 10

Peek operations:
Top: 40
Top: 40
```

Experiment No 6

Aim: Write a program to reverse a sentence/string using stack.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 1000

struct Stack {
   int top;
   char items[MAX];
};

void initStack(struct Stack* stack) {
   stack->top = -1;
}

int isFull(struct Stack* stack) {
   return stack->top == MAX - 1;
}

int isEmpty(struct Stack* stack) {
   return stack->top == -1;
}

void push(struct Stack* stack, char ch) {
   if (!isFull(stack)) {
      stack->items[++(stack->top)] = ch;
   }
}

char pop(struct Stack* stack) {
   if (!isEmpty(stack)) {
      return stack->items[(stack->top)--];
   }
   return '\0';
}

int main() {
   char input[MAX];
   struct Stack stack;
   initStack(&stack);

   printf("Enter a sentence or string:\n");
   fgets(input, MAX, stdin);

   size_t len = strlen(input);
   if (len > 0 && input[len - 1] == '\n') {
      input[len - 1] = '\0';
   }
```

```
    for (int i = 0; input[i] != '\0'; i++) {
        push(&stack, input[i]);
    }

    printf("Reversed string:\n");
    while (!isEmpty(&stack)) {
        printf("%c", pop(&stack));
    }

    printf("\n");
    return 0;
}
```

Output:

```
Enter a sentence or string:
Hello this is String reversal using stack
Reversed string:
kcats gnisu lasrever gnirtS si siht olleH
```

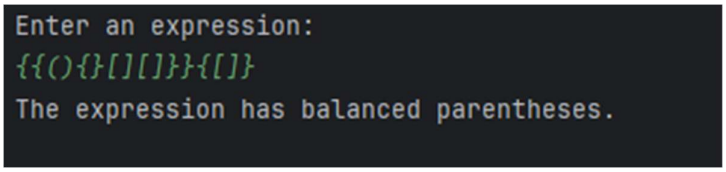
Experiment No 7

Aim: Write a program to check for balanced parenthesis in each expression.

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

#define MAX 1000

struct Stack {
   int top;
   char items[MAX];
};

void initStack(struct Stack* stack) {
   stack->top = -1;
}

int isEmpty(struct Stack* stack) {
   return stack->top == -1;
}

void push(struct Stack* stack, char ch) {
   if (stack->top < MAX - 1) {
      stack->items[++(stack->top)] = ch;
   }
}

char pop(struct Stack* stack) {
   if (!isEmpty(stack)) {
      return stack->items[(stack->top)--];
   }
   return '\0';
}

char peek(struct Stack* stack) {
   if (!isEmpty(stack)) {
      return stack->items[stack->top];
   }
   return '\0';
}

int isBalanced(const char* expr) {
   struct Stack stack;
   initStack(&stack);
   for (int i = 0; expr[i] != '\0'; i++) {
      char ch = expr[i];
      if (ch == '(' || ch == '{' || ch == '[') {
         push(&stack, ch);
      } else if (ch == ')' || ch == '}' || ch == ']') {
```

```
            if (isEmpty(&stack)) return 0;
            char top = pop(&stack);
            if ((ch == ')' && top != '(') ||
                (ch == '}' && top != '{') ||
                (ch == ']' && top != '[')) {
                return 0;
            }
        }
    }
    return isEmpty(&stack);
}

int main() {
    char expression[MAX];
    printf("Enter an expression:\n");
    fgets(expression, MAX, stdin);
    size_t len = strlen(expression);
    if (len > 0 && expression[len - 1] == '\n') {
        expression[len - 1] = '\0';
    }
    if (isBalanced(expression)) {
        printf("The expression has balanced parentheses.\n");
    } else {
        printf("The expression does NOT have balanced parentheses.\n");
    }
    return 0;
}
```

Output:



Experiment No 8

Aim: Write a program to convert infix expression to prefix and postfix expression.

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

#define MAX 100

typedef struct {
   char items[MAX];
   int top;
} Stack;

void push(Stack *s, char c) {
   if (s->top < MAX - 1)
      s->items[++s->top] = c;
}

char pop(Stack *s) {
   if (s->top >= 0)
      return s->items[s->top--];
   return '\0';
}

char peek(Stack *s) {
   if (s->top >= 0)
      return s->items[s->top];
   return '\0';
}

int isEmpty(Stack *s) {
   return s->top == -1;
}

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

int precedence(char c) {
   switch (c) {
      case '^': return 3;
      case '*': case '/': return 2;
      case '+': case '-': return 1;
      default: return -1;
   }
}
```

```
void replaceParentheses(char *str) {
   for (int i = 0; i < strlen(str); i++) {
      if (str[i] == '(')
         str[i] = ')';
      else if (str[i] == ')')
         str[i] = '(';
   }
}

char* infixToPostfix(const char *infix) {
   Stack stack;
   stack.top = -1;
   char *result = malloc(strlen(infix) + 1);
   int k = 0;

   for (int i = 0; i < strlen(infix); i++) {
      char ch = infix[i];
      if (isalnum(ch)) {
         result[k++] = ch;
      } else if (ch == '(') {
         push(&stack, ch);
      } else if (ch == ')') {
         while (!isEmpty(&stack) && peek(&stack) != '('
            result[k++] = pop(&stack);
         pop(&stack);
      } else if (isOperator(ch)) {
         while (!isEmpty(&stack) && precedence(ch) <= precedence(peek(&stack)))
            result[k++] = pop(&stack);
         push(&stack, ch);
      }
   }
   while (!isEmpty(&stack))
      result[k++] = pop(&stack);
   result[k] = '\0';
   return result;
}

char* infixToPrefix(char *infix) {
   reverse(infix);
   replaceParentheses(infix);

   Stack stack;
   stack.top = -1;
   char *result = malloc(strlen(infix) + 1);
   int k = 0;

   for (int i = 0; i < strlen(infix); i++) {
      char ch = infix[i];
      if (isalnum(ch)) {
         result[k++] = ch;
      } else if (ch == '(') {
```

```
            push(&stack, ch);
        } else if (ch == ')') {
            while (!isEmpty(&stack) && peek(&stack) != '(')
                result[k++] = pop(&stack);
            pop(&stack);
        } else if (isOperator(ch)) {
            while (!isEmpty(&stack) && precedence(ch) < precedence(peek(&stack)))
                result[k++] = pop(&stack);
            push(&stack, ch);
        }
    }
    while (!isEmpty(&stack))
        result[k++] = pop(&stack);
    result[k] = '\0';
    reverse(result);
    return result;
}

int main() {
    char infix[MAX];
    printf("Enter infix expression:\n");
    fgets(infix, MAX, stdin);
    infix[strcspn(infix, "\n")] = '\0';

    char cleaned[MAX];
    int j = 0;
    for (int i = 0; i < strlen(infix); i++) {
        if (!isspace(infix[i]))
            cleaned[j++] = infix[i];
    }
    cleaned[j] = '\0';

    char *postfix = infixToPostfix(cleaned);
    char *prefix = infixToPrefix(cleaned);
    printf("Postfix: %s\n", postfix);
    printf("Prefix: %s\n", prefix);

    free(postfix);
    free(prefix);
    return 0;
}
```

Output:

```
Enter infix expression:
(8 + 2) * (3 - 1) / (4 + (6 / 3))
Postfix: 82+31-*463/+/
Prefix: /*+82-31+4/63
```

Experiment No 9

Aim: Write a program to implement Linear Queue using Array and Linked Lists.

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

#define SIZE 5

typedef struct {
   int front, rear;
   int queue[SIZE];
} ArrayQueue;

void initArrayQueue(ArrayQueue *q) {
   q->front = q->rear = -1;
}

void enqueueArray(ArrayQueue *q, int data) {
   if (q->rear == SIZE - 1) {
      printf("Queue Overflow\n");
      return;
   }
   if (q->front == -1) q->front = 0;
   q->queue[++q->rear] = data;
}

void dequeueArray(ArrayQueue *q) {
   if (q->front == -1 || q->front > q->rear) {
      printf("Queue Underflow\n");
      return;
   }
   printf("Dequeued: %d\n", q->queue[q->front]);
   q->front++;
   if (q->front > q->rear) q->front = q->rear = -1;
}

void displayArray(ArrayQueue *q) {
   if (q->front == -1 || q->front > q->rear) {
      printf("Queue is empty\n");
      return;
   }
   printf("Queue (Array): ");
   for (int i = q->front; i <= q->rear; i++) {
      printf("%d ", q->queue[i]);
   }
   printf("\n");
}

typedef struct Node {
```

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

typedef struct {
    Node *front, *rear;
} LinkedQueue;

void initLinkedQueue(LinkedQueue *q) {
    q->front = q->rear = NULL;
}

void enqueueLinked(LinkedQueue *q, int data) {
    Node *newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = newNode;
        return;
    }
    q->rear->next = newNode;
    q->rear = newNode;
}

void dequeueLinked(LinkedQueue *q) {
    if (q->front == NULL) {
        printf("Queue Underflow\n");
        return;
    }
    printf("Dequeued: %d\n", q->front->data);
    Node *temp = q->front;
    q->front = q->front->next;
    if (q->front == NULL) q->rear = NULL;
    free(temp);
}

void displayLinked(LinkedQueue *q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue (Linked List): ");
    Node *temp = q->front;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
```

```
    ArrayQueue arrayQueue;
    LinkedQueue linkedQueue;
    initArrayQueue(&arrayQueue);
    initLinkedQueue(&linkedQueue);

    printf("Enqueueing elements to both queues:\n");
    for (int i = 1; i <= 5; i++) {
        enqueueArray(&arrayQueue, i * 10);
        enqueueLinked(&linkedQueue, i * 10);
    }

    displayArray(&arrayQueue);
    displayLinked(&linkedQueue);

    printf("\nDequeuing once from both:\n");
    dequeueArray(&arrayQueue);
    dequeueLinked(&linkedQueue);

    displayArray(&arrayQueue);
    displayLinked(&linkedQueue);

    return 0;
}
```

Output:

```
Enqueueing elements to both queues:
Queue (Array): 10 20 30 40 50
Queue (Linked List): 10 20 30 40 50

Dequeuing once from both:
Dequeued: 10
Dequeued: 10
Queue (Array): 20 30 40 50
Queue (Linked List): 20 30 40 50
```

Experiment No 10

Aim: Write a program to implement Circular Queue using Array and Linked Lists.

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

#define MAX_CAPACITY 5

typedef struct CircularArrayQueue {
   int queue[MAX_CAPACITY];
   int front, rear;
} CircularArrayQueue;

void initArrayQueue(CircularArrayQueue* q) {
   q->front = q->rear = -1;
}

void enqueueArrayQueue(CircularArrayQueue* q, int data) {
   if ((q->rear + 1) % MAX_CAPACITY == q->front) {
      printf("Queue Overflow (Full)\n");
      return;
   }
   if (q->front == -1) q->front = 0;
   q->rear = (q->rear + 1) % MAX_CAPACITY;
   q->queue[q->rear] = data;
}

void dequeueArrayQueue(CircularArrayQueue* q) {
   if (q->front == -1) {
      printf("Queue Underflow (Empty)\n");
      return;
   }
   printf("Dequeued: %d\n", q->queue[q->front]);
   if (q->front == q->rear) {
      q->front = q->rear = -1;
   } else {
      q->front = (q->front + 1) % MAX_CAPACITY;
   }
}

void displayArrayQueue(CircularArrayQueue* q) {
   if (q->front == -1) {
      printf("Queue is empty.\n");
      return;
   }
   printf("Circular Queue (Array): ");
   int i = q->front;
   while (1) {
      printf("%d ", q->queue[i]);
```

```
        if (i == q->rear) break;
        i = (i + 1) % MAX_CAPACITY;
    }
    printf("\n");
}

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

typedef struct CircularLinkedQueue {
    Node* front;
    Node* rear;
} CircularLinkedQueue;

void initLinkedQueue(CircularLinkedQueue* q) {
    q->front = q->rear = NULL;
}

void enqueueLinkedQueue(CircularLinkedQueue* q, int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    if (q->front == NULL) {
        q->front = q->rear = newNode;
        q->rear->next = q->front;
    } else {
        q->rear->next = newNode;
        q->rear = newNode;
        q->rear->next = q->front;
    }
}

void dequeueLinkedQueue(CircularLinkedQueue* q) {
    if (q->front == NULL) {
        printf("Queue Underflow (Empty)\n");
        return;
    }
    printf("Dequeued: %d\n", q->front->data);
    if (q->front == q->rear) {
        free(q->front);
        q->front = q->rear = NULL;
    } else {
        Node* temp = q->front;
        q->front = q->front->next;
        q->rear->next = q->front;
        free(temp);
    }
}

void displayLinkedQueue(CircularLinkedQueue* q) {
```

```
    if (q->front == NULL) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Circular Queue (Linked List): ");
    Node* temp = q->front;
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != q->front);
    printf("\n");
}

int main() {
    CircularArrayQueue arrayQueue;
    CircularLinkedQueue linkedQueue;

    initArrayQueue(&arrayQueue);
    initLinkedQueue(&linkedQueue);

    printf("Enqueueing into both circular queues:\n");
    for (int i = 1; i <= 4; i++) {
        enqueueArrayQueue(&arrayQueue, i * 10);
        enqueueLinkedQueue(&linkedQueue, i * 10);
    }

    displayArrayQueue(&arrayQueue);
    displayLinkedQueue(&linkedQueue);

    printf("\nDequeuing once from both:\n");
    dequeueArrayQueue(&arrayQueue);
    dequeueLinkedQueue(&linkedQueue);

    displayArrayQueue(&arrayQueue);
    displayLinkedQueue(&linkedQueue);

    return 0;
}
```

Output:

```
Enqueueing into both circular queues:
Circular Queue (Array): 10 20 30 40
Circular Queue (Linked List): 10 20 30 40

Dequeuing once from both:
Dequeued: 10
Dequeued: 10
Circular Queue (Array): 20 30 40
Circular Queue (Linked List): 20 30 40
```

Experiment No 11

Aim: Write a program to implement Doubly Ended Queue using Array and Linked Lists.

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

#define MAX 5

typedef struct DequeArray {
   int deque[MAX];
   int front, rear, size;
} DequeArray;

void initDequeArray(DequeArray* d) {
   d->size = MAX;
   d->front = -1;
   d->rear = 0;
}

void insertFront(DequeArray* d, int data) {
   if ((d->front == 0 && d->rear == d->size - 1) || (d->front == d->rear + 1)) {
      printf("Deque Overflow\n");
      return;
   }
   if (d->front == -1) {
      d->front = d->rear = 0;
   } else if (d->front == 0) {
      d->front = d->size - 1;
   } else {
      d->front--;
   }
   d->deque[d->front] = data;
}

void insertRear(DequeArray* d, int data) {
   if ((d->front == 0 && d->rear == d->size - 1) || (d->front == d->rear + 1)) {
      printf("Deque Overflow\n");
      return;
   }
   if (d->front == -1) {
      d->front = d->rear = 0;
   } else if (d->rear == d->size - 1) {
      d->rear = 0;
   } else {
      d->rear++;
   }
   d->deque[d->rear] = data;
}
```

```
void deleteFront(DequeArray* d) {
   if (d->front == -1) {
      printf("Deque Underflow\n");
      return;
   }
   printf("Deleted from front: %d\n", d->deque[d->front]);
   if (d->front == d->rear) {
      d->front = d->rear = -1;
   } else if (d->front == d->size - 1) {
      d->front = 0;
   } else {
      d->front++;
   }
}

void deleteRear(DequeArray* d) {
   if (d->front == -1) {
      printf("Deque Underflow\n");
      return;
   }
   printf("Deleted from rear: %d\n", d->deque[d->rear]);
   if (d->front == d->rear) {
      d->front = d->rear = -1;
   } else if (d->rear == 0) {
      d->rear = d->size - 1;
   } else {
      d->rear--;
   }
}

void displayDequeArray(DequeArray* d) {
   if (d->front == -1) {
      printf("Deque is empty.\n");
      return;
   }
   printf("Deque (Array): ");
   int i = d->front;
   while (1) {
      printf("%d ", d->deque[i]);
      if (i == d->rear) break;
      i = (i + 1) % d->size;
   }
   printf("\n");
}

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

```
typedef struct DequeLinkedList {
   Node* front;
   Node* rear;
} DequeLinkedList;

void initDequeLinkedList(DequeLinkedList* d) {
   d->front = d->rear = NULL;
}

void insertFrontLL(DequeLinkedList* d, int data) {
   Node* newNode = (Node*)malloc(sizeof(Node));
   newNode->data = data;
   newNode->prev = NULL;
   newNode->next = d->front;
   if (d->front == NULL) {
      d->front = d->rear = newNode;
   } else {
      d->front->prev = newNode;
      d->front = newNode;
   }
}

void deleteFrontLL(DequeLinkedList* d) {
   if (d->front == NULL) {
      printf("Deque Underflow\n");
      return;
   }
   printf("Deleted from front: %d\n", d->front->data);
   Node* temp = d->front;
   if (d->front == d->rear) {
      d->front = d->rear = NULL;
   } else {
      d->front = d->front->next;
      d->front->prev = NULL;
   }
   free(temp);
}

void deleteRearLL(DequeLinkedList* d) {
   if (d->rear == NULL) {
      printf("Deque Underflow\n");
      return;
   }
   printf("Deleted from rear: %d\n", d->rear->data);
   Node* temp = d->rear;
   if (d->front == d->rear) {
      d->front = d->rear = NULL;
   } else {
      d->rear = d->rear->prev;
      d->rear->next = NULL;
   }
```

```
    free(temp);
}

void displayDequeLinkedList(DequeLinkedList* d) {
    if (d->front == NULL) {
        printf("Deque is empty.\n");
        return;
    }
    printf("Deque (Linked List): ");
    Node* temp = d->front;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    DequeArray arrayDeque;
    DequeLinkedList linkedDeque;
    initDequeArray(&arrayDeque);
    initDequeLinkedList(&linkedDeque);

    insertRear(&arrayDeque, 10);
    insertRearLL(&linkedDeque, 10);
    insertFrontLL(&linkedDeque, 5);
    insertRearLL(&linkedDeque, 15);
    displayDequeLinkedList(&linkedDeque);
    deleteFrontLL(&linkedDeque);
    deleteRearLL(&linkedDeque);
    displayDequeLinkedList(&linkedDeque);

    return 0;
}
```

Output:

```
Deque (Array): 5 10 15
Deleted from front: 5
Deleted from rear: 15
Deque (Array): 10
Deque (Linked List): 5 10 15
Deleted from front: 5
Deleted from rear: 15
Deque (Linked List): 10
```

Experiment No 12

Aim: Write a Program to implement Binary Search Tree operations.

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

struct Node {
  int key;
  struct Node *left, *right;
};

struct BST {
  struct Node *root;
};

struct Node* newNode(int item) {
  struct Node* node = (struct Node*)malloc(sizeof(struct Node));
  node->key = item;
  node->left = node->right = NULL;
  return node;
}

struct Node* insert(struct Node* root, int key) {
  if (root == NULL) {
    return newNode(key);
  }
  if (key < root->key) {
    root->left = insert(root->left, key);
  } else if (key > root->key) {
    root->right = insert(root->right, key);
  }
  return root;
}

void inorder(struct Node* root) {
  if (root != NULL) {
    inorder(root->left);
    printf("%d ", root->key);
    inorder(root->right);
  }
}

struct Node* search(struct Node* root, int key) {
  if (root == NULL || root->key == key) {
    return root;
  }
  if (key < root->key) {
    return search(root->left, key);
  }
```

```
    return search(root->right, key);
}

struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current && current->left != NULL) {
        current = current->left;
    }
    return current;
}

struct Node* delete(struct Node* root, int key) {
    if (root == NULL) {
        return root;
    }
    if (key < root->key) {
        root->left = delete(root->left, key);
    } else if (key > root->key) {
        root->right = delete(root->right, key);
    } else {
        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;
        }

        struct Node* temp = minValueNode(root->right);
        root->key = temp->key;
        root->right = delete(root->right, temp->key);
    }
    return root;
}

int main() {
    struct BST tree;
    tree.root = NULL;

    int keys[] = {50, 30, 20, 40, 70, 60, 80};
    int n = sizeof(keys) / sizeof(keys[0]);

    for (int i = 0; i < n; i++) {
        tree.root = insert(tree.root, keys[i]);
    }

    printf("Inorder traversal of the BST:\n");
    inorder(tree.root);
```

```
    int searchKey = 40;
    struct Node* result = search(tree.root, searchKey);
    printf("\n\nSearch for %d: %s\n", searchKey, (result != NULL ? "Found" : "Not Found"));

    int deleteKey = 30;
    printf("\nDeleting %d\n", deleteKey);
    tree.root = delete(tree.root, deleteKey);

    printf("Inorder traversal after deletion:\n");
    inorder(tree.root);

    return 0;
}
```

Output:

```
Inorder traversal of the BST:
20 30 40 50 60 70 80

Search for 40: Found

Deleting 30
Inorder traversal after deletion:
20 40 50 60 70 80
```

Experiment No 13

Aim: Write a Program to implement Bubble Sort, Selection Sort, Heap Sort, Quick Sort, Merge Sort and Insertion Sort algorithm.

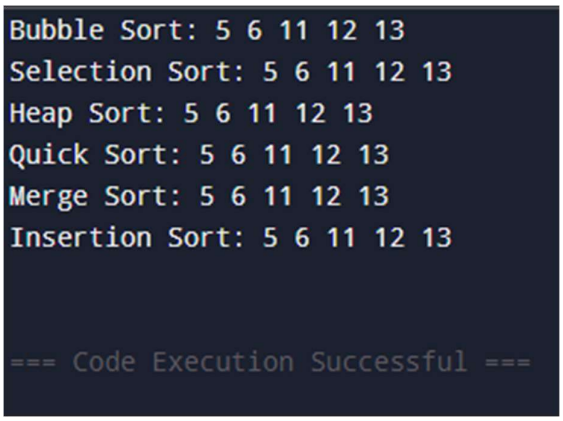
```
#include <stdio.h>

void bubbleSort(int arr[], int n) {
   for (int i = 0; i < n-1; i++) {
      for (int j = 0; j < n-i-1; j++) {
         if (arr[j] > arr[j+1]) {
            int temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
   }
}

void selectionSort(int arr[], int n) {
   for (int i = 0; i < n-1; i++) {
      int minIdx = i;
      for (int j = i+1; j < n; j++) {
         if (arr[j] < arr[minIdx]) {
            minIdx = j;
         }
      }
      int temp = arr[minIdx];
      arr[minIdx] = arr[i];
      arr[i] = temp;
   }
}

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) {
   for (int i = n/2 - 1; i >= 0; i--) {
      heapify(arr, n, i);
   }
```

```
  for (int i = n-1; i >= 0; i--) {
    int temp = arr[0];
    arr[0] = arr[i];
    arr[i] = temp;
    heapify(arr, i, 0);
  }
}

int partition(int arr[], int low, int high) {
  int pivot = arr[high];
  int i = low - 1;
  for (int j = low; j < high; j++) {
    if (arr[j] < pivot) {
      i++;
      int temp = arr[i];
      arr[i] = arr[j];
      arr[j] = temp;
    }
  }
  int temp = arr[i+1];
  arr[i+1] = arr[high];
  arr[high] = temp;
  return i+1;
}

void quickSort(int arr[], int low, int high) {
  if (low < high) {
    int pi = partition(arr, low, high);
    quickSort(arr, low, pi-1);
    quickSort(arr, pi+1, high);
  }
}

void merge(int arr[], int l, int m, int r) {
  int n1 = m - l + 1;
  int n2 = r - m;
  int L[n1], R[n2];
  for (int i = 0; i < n1; i++) L[i] = arr[l + i];
  for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
  int i = 0, j = 0, k = l;
  while (i < n1 && j < n2) {
    if (L[i] <= R[j]) arr[k++] = L[i++];
    else arr[k++] = R[j++];
  }
  while (i < n1) arr[k++] = L[i++];
  while (j < n2) arr[k++] = R[j++];
}

void mergeSort(int arr[], int l, int r) {
  if (l < r) {
    int m = l + (r - l) / 2;
```

```
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);
    merge(arr, l, m, r);
  }
}

void insertionSort(int arr[], int n) {
  for (int i = 1; i < n; i++) {
    int key = arr[i];
    int j = i - 1;
    while (j >= 0 && arr[j] > key) {
      arr[j + 1] = arr[j];
      j--;
    }
    arr[j + 1] = key;
  }
}

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

int main() {
  int arr[] = {12, 11, 13, 5, 6};
  int n = sizeof(arr)/sizeof(arr[0]);

  int arr1[] = {12, 11, 13, 5, 6};
  bubbleSort(arr1, n);
  printf("Bubble Sort: ");
  printArray(arr1, n);

  int arr2[] = {12, 11, 13, 5, 6};
  selectionSort(arr2, n);
  printf("Selection Sort: ");
  printArray(arr2, n);

  int arr3[] = {12, 11, 13, 5, 6};
  heapSort(arr3, n);
  printf("Heap Sort: ");
  printArray(arr3, n);

  int arr4[] = {12, 11, 13, 5, 6};
  quickSort(arr4, 0, n-1);
  printf("Quick Sort: ");
  printArray(arr4, n);

  int arr5[] = {12, 11, 13, 5, 6};
  mergeSort(arr5, 0, n-1);
```

```
    printf("Merge Sort: ");
    printArray(arr5, n);

    int arr6[] = {12, 11, 13, 5, 6};
    insertionSort(arr6, n);
    printf("Insertion Sort: ");
    printArray(arr6, n);

    return 0;
}
```

Output:



Experiment No 14(a)

Aim: Write C Programs by using Graph data structures for Graph Traversal using BFS

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

#define MAX 100

int graph[MAX][MAX];
bool visited[MAX];
int queue[MAX];
int front = -1, rear = -1;

void enqueue(int vertex) {
  if (rear == MAX - 1) {
    printf("Queue is full\n");
  } else {
    if (front == -1) front = 0;
    rear++;
    queue[rear] = vertex;
  }
}

int dequeue() {
  int vertex;
  if (front == -1) {
    printf("Queue is empty\n");
    return -1;
  } else {
    vertex = queue[front];
    front++;
    if (front > rear) front = rear = -1;
    return vertex;
  }
}

void bfs(int start, int n) {
  enqueue(start);
  visited[start] = true;
  while (front != -1) {
    int vertex = dequeue();
    printf("%d ", vertex);
    for (int i = 0; i < n; i++) {
      if (graph[vertex][i] == 1 && !visited[i]) {
        enqueue(i);
        visited[i] = true;
      }
    }
```

```
    }
}

int main() {
    int n, edges, u, v;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter number of edges: ");
    scanf("%d", &edges);

    for (int i = 0; i < edges; i++) {
        printf("Enter edge (u v): ");
        scanf("%d %d", &u, &v);
        graph[u][v] = 1;
        graph[v][u] = 1;
    }

    printf("Enter starting vertex: ");
    int start;
    scanf("%d", &start);

    printf("BFS Traversal: ");
    for (int i = 0; i < n; i++) visited[i] = false;
    bfs(start, n);
    printf("\n");

    return 0;
}
```

Output:

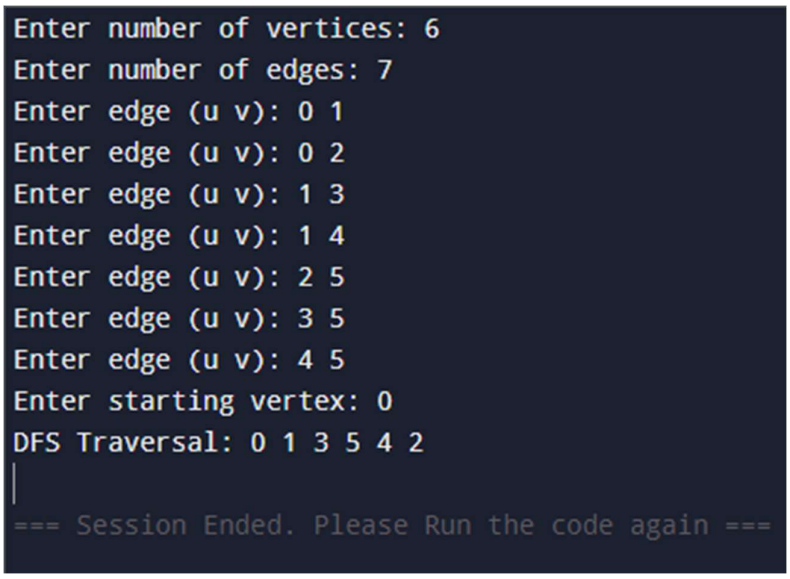
```
Enter number of vertices: 6
Enter number of edges: 7
Enter edge (u v): 0 1
Enter edge (u v): 0 2
Enter edge (u v): 1 3
Enter edge (u v): 1 4
Enter edge (u v): 2 5
Enter edge (u v): 3 5
Enter edge (u v): 4 5
Enter starting vertex: 0
BFS Traversal: 0 1 2 3 4 5
```

Experiment No 14(b)

Aim: Write C Programs by using Graph data structures for Graph Traversal using DFS

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

#define MAX 100

int graph[MAX][MAX];
bool visited[MAX];

void dfs(int vertex, int n) {
   visited[vertex] = true;
   printf("%d ", vertex);
   for (int i = 0; i < n; i++) {
      if (graph[vertex][i] == 1 && !visited[i]) {
         dfs(i, n);
      }
   }
}

int main() {
   int n, edges, u, v;
   printf("Enter number of vertices: ");
   scanf("%d", &n);
   printf("Enter number of edges: ");
   scanf("%d", &edges);

   for (int i = 0; i < edges; i++) {
      printf("Enter edge (u v): ");
      scanf("%d %d", &u, &v);
      graph[u][v] = 1;
      graph[v][u] = 1;
   }

   printf("Enter starting vertex: ");
   int start;
   scanf("%d", &start);

   printf("DFS Traversal: ");
   for (int i = 0; i < n; i++) visited[i] = false;
   dfs(start, n);
   printf("\n");

   return 0;
}
```

Output:


Enter number of vertices: 6
Enter number of edges: 7
Enter edge (u v): 0 1
Enter edge (u v): 0 2
Enter edge (u v): 1 3
Enter edge (u v): 1 4
Enter edge (u v): 2 5
Enter edge (u v): 3 5
Enter edge (u v): 4 5
Enter starting vertex: 0
DFS Traversal: 0 1 3 5 4 2
=== Session Ended. Please Run the code again ===