EX.NO	DATES	LIST OF EXERCISES	MARKS	SIGN
1(A)		Array implementation of List (CBS)		
1(B)		Implementation of Singly Linked List		
1(C)		Implementation of Doubly Linked List		
2		Implementation of Polynomial ADT using Linked list		
3(A)		Array implementation of Stack ADT		
3(B)		Array implementation of Queue ADT		
4(A)		Linked list implementation of Stack ADT		
4(B)		Linked list implementation of Queue ADT		
5		Convert a given Infix expression into its Postfix using Stack		
6		Evaluate a given Postfix Expressions using Stack		
7		Implementation of Binary Search Trees		
8		Implementation of AVL Trees		
9		Implementation of Heap Trees		
10(A)		Implementation of Shortest Path algorithms :Dijkstra’s Algorithm		
10(B)		Implementation of Shortest Path algorithms : Floyd’s Algorithm		
11(A)		Implementation of Minimum Spanning Tree algorithms :Prim’s Algorithm		
11(B)		Implementation of Minimum Spanning Tree algorithms : Kruskal’s Algorithm		
12 (A)		Implementation of Linear Search		
12 (B)		Implementation of Binary Search		
13(A)		Implementation of Insertion Sort		
13(B)		Implementation of Selection Sort		
14 (A)		Implementation of Open Addressing - Linear Probing		
14 (B)		Implementation of Open Addressing - Quadratic Probing		

Note: Take this index printout and paste it on the first page of the observation notebook, or write it neatly by referring to the soft copy

EX.NO: 1(A) **Array Implementation Of List**
DATE:

Aim: To Write a program for Array based implementation of LIST ADT with Create, Insert, Search , Delete & Traverse operations

Algorithm:
1. Start the program and declare an array and variables.
2. Ask the user how many elements they want and read them into the array.
3. Ask the user for a new element and the position to insert it.
4. Check if the position is valid (between 0 and current size).
5. If valid, shift elements to the right from that position.
6. Insert the new element at the given position and increase the size.
7. Ask the user for an element to search and delete.
8. Search the array for that element and store its position if found.
9. If found, shift elements left to remove it and reduce the size.
10. Print the final array after all operations.

Program:
```
// 1 (A ) Array Implementation of List (Content Beyond Syllabus)
#include <stdio.h>

void main() {
   int a[100], n, i, p, x;

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

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

   // Insert element
   printf("\nEnter element to insert: ");
   scanf("%d", &x);
   printf("Enter position (0 to %d): ", n);
   scanf("%d", &p);

   if (p >= 0 && p <= n) {
      for (i = n; i > p; i--)
         a[i] = a[i - 1];
      a[p] = x;
      n++;
      printf("Element %d inserted at position %d.\n", x, p);
   } else {
      printf("Invalid position. Insertion skipped.\n");
   }
```

```
    // Search a element  to delete
    printf("\nEnter element to search and delete: ");
    scanf("%d", &x);

    p = -1;
    for (i = 0; i < n; i++) {
        if (a[i] == x) {
            p = i;
            break;
        }
    }

//Delete a element
    if (p != -1) {
        printf("Element %d found at position %d.\n", x, p);
        for (i = p; i < n - 1; i++)
            a[i] = a[i + 1];
        n--;
        printf("Element %d deleted.\n", x);
    } else {
        printf("Element %d not found. No deletion performed.\n", x);
    }

    // Display array
    printf("\nFinal array:\n");
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");

    }
```

Output:

Result

Thus the program for Array based implementation of LIST ADT with Create, Insert, Search , Delete & Traverse operations has been written,Executed & Output was verified.

EX.NO: 1(B) **Implementation Of Singly Linked List**
DATE:

Aim: To write a program for Singly Linked List implementation of LIST ADT with operations: Create, Insert, Search, Delete & Traverse

Algorithm:

1. Start the program and define a node structure with `data` and `next` pointer.
2. Create the list by reading `n` elements and linking each new node to the previous one.
3. Display the list by traversing from `head` and printing each node's data.
4. Insert a new node at the beginning by linking it before the current `head`.
5. Insert a new node at the end by linking it after the last node.
6. Delete the first node by moving `head` to the next node and freeing the old one.
7. Delete the last node by freeing the next of the second-last node and setting it to NULL.
8. Search for a node by traversing the list and comparing each node's data.
9. Insert a new node after a searched node by adjusting the `next` pointers.
10. Delete the node after a searched node by bypassing it and freeing its memory.

Program:

```
// EX.NO.1 (B ) Implementation of SINGLY Linked List
#include <stdio.h>
#include <stdlib.h>

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

node* head = NULL, *pre, *tmp;
int n, x, i;

// Function to create the linked list
void create(int x) {
   tmp = (node*)malloc(sizeof(node));
   tmp->data = x;
   tmp->next = NULL;
   if (head == NULL) {
      head = tmp;
      pre = head; // first node
   } else {
      pre->next = tmp;
      pre = tmp; // other nodes
   }
}
```

```
// Function to display the list
void display() {
    for (tmp = head; tmp != NULL; tmp = tmp->next)
        printf("%d ", tmp->data);
    if (head == NULL)
        printf("\nList is empty");
}

// Function to insert at the beginning
void insertb(int x) {
    tmp = (node*)malloc(sizeof(node));
    tmp->data = x;
    tmp->next = head;
    head = tmp;
}

// Function to insert at the end
void inserte(int x) {
    tmp = (node*)malloc(sizeof(node));
    tmp->data = x;
    tmp->next = NULL;
    pre->next = tmp;
}

// Function to delete at the beginning
void deleteb() {
    tmp = head;
    head = head->next;
    free(tmp);
}

// Function to delete at the end
void deletee() {
    free(pre->next);
    pre->next = NULL;
}

// Function to search for a node
node* search(int x) {
    for (pre = head; pre != NULL; pre = pre->next)
        if (pre->data == x)
            return pre; // number of the node
    return NULL;
}
```

```
// Function to insert after a given node
void insertan(int x) {
    tmp = (node*)malloc(sizeof(node));
    tmp->data = x;
    tmp->next = pre->next;
    pre->next = tmp;
}

// Function to delete node after a given node
void deletean() {
    tmp = pre->next;
    pre->next = tmp->next;
    free(tmp);
}


int main() {
    // Create the list
    printf("\nEnter number of elements: ");
    scanf("%d", &n);
    for (i = 0; i < n; ++i) {
        printf("\nEnter element %d: ", i + 1);
        scanf("%d", &x);
        create(x);
    }

    // Display the list
    printf("\nList after creation: ");
    display();

    // Insert at the beginning
    printf("\nEnter element to be inserted: ");
    scanf("%d", &x);
    insertb(x);
    printf("\nList after inserting at the beginning: ");
    display();

    // Insert at the end
    printf("\nEnter element to be inserted: ");
    scanf("%d", &x);
    inserte(x);
    printf("\nList after inserting at the end: ");
    display();

    // Delete at the beginning
    deleteb();
    printf("\nList after deleting from the beginning: ");
    display();
```

```
 // Delete at the end
    deletee();
    printf("\nList after deleting from the end: ");
    display();

    // Insert after a given node
    printf("\nEnter the data of the node after which to insert: ");
    scanf("%d", &x);
    pre = search(x);
    if (pre != NULL) {
        printf("\nEnter element to be inserted: ");
        scanf("%d", &x);
        insertan(x);
    } else {
        printf("\nNode with data %d not found ", x);
    }
    printf("\nList after inserting: ");
    display();


    // Delete after a given node
    printf("\nEnter the data of the node after which to delete: ");
    scanf("%d", &x);
    pre = search(x);
    if (pre != NULL && pre->next != NULL)
        deletean();
    else
        printf("\nNode with data %d not found or no node to delete after it", x);
    printf("\nList after deleting node: ");
    display();

    return 0;
}
```

Output:

Result: Thus, the program to implement Singly Linked List with operations — Create, Insert, Search, Delete, and Traverse — has been successfully written, executed, and the output has been verified.

EX.NO: 1 (C) **Implementation Of Doubly Linked List**
DATE:

Aim: To write a program for Doubly Linked List implementation of LIST Abstract Data Type (ADT) with operations: Create, Insert, Search, Delete, and Traverse.

Algorithm:

1. **Start the program** and declare the node structure with `data`, `left`, and `right` pointers.
2. **Create the list** by dynamically allocating nodes and linking them using `left` and `right` pointers.
3. **Display the list** by traversing from `head` to the end using the `right` pointer.
4. **Insert at the beginning** by creating a new node and linking it before the current `head`.
5. **Insert at the end** by traversing to the last node and linking the new node after it.
6. **Delete from the beginning** by moving `head` to the next node and freeing the old one.
7. **Delete from the end** by traversing to the second-last node and freeing the last node.
8. **Search for a node** by comparing each node's data with the target value.
9. **Insert or delete after a specific node** by first searching for it, then adjusting links accordingly.

Program:

```
// EX.NO.1 (C) Implementation of Doubly Linked List
#include <stdio.h>
#include <stdlib.h>

// Declaration
typedef struct node {
   int data;
   struct node* left, *right;
} node;

node* head = NULL, *pre, *tmp;

// Function to create the linked list
void Create(int x) {
   tmp = (node*)malloc(sizeof(node));
   tmp->data = x;
   tmp->right = NULL;
   if (head == NULL) {
      head = tmp;
      pre = head; // first node
   } else {
      pre->right = tmp;
      tmp->left = pre; // other nodes
      pre = tmp;
   }
}
```

```
// Function to display the list
void display() {
   for (pre = head; pre != NULL; pre = pre->right)
      printf("%d ", pre->data);
   if (head == NULL)
      printf("\nList is empty");
}

// Function to insert at the beginning
void insertb(int x) {
   tmp = (node*)malloc(sizeof(node));
   tmp->data = x;
   tmp->left = NULL;
   tmp->right = head;
   head->left = tmp;
   head = tmp;
}

// Function to insert at the end
void inserte(int x) {
   tmp = (node*)malloc(sizeof(node));
   tmp->data = x;
   tmp->right = NULL;
   pre->right = tmp;
   tmp->left = pre;
}

// Function to delete at the beginning
void deleteb() {
   tmp = head;
   head = head->right;
   free(tmp);
   head->left = NULL;
}

// Function to delete at the end
void deletee() {
   free(pre->right);
   pre->right = NULL;
}

// Function to search for a node
node* search(int x) {
   for (pre = head; pre != NULL; pre = pre->right)
      if (pre->data == x)
         return pre; // number of the node
   return NULL;
}
```

```
// Function to insert after a given node
void insertan(int x) {
    tmp = (node*)malloc(sizeof(node));
    tmp->data = x;
    tmp->right = pre->right;
    tmp->left = pre;
    pre->right = tmp;
}

// Function to delete node after a given node

void deletean() {
    tmp = pre->right;
    pre->right = tmp->right;
    tmp->right->left = pre;
    free(tmp);
}


// Main function with precondition checks
int main() {
    int n, x, i;

    // Create the list
    printf("\nEnter number of elements: ");
    scanf("%d", &n);
    for (i = 0; i < n; ++i) {
        printf("\nEnter element %d: ", i + 1);
        scanf("%d", &x);
        Create(x);
    }

    printf("\nList after creation: ");
    display();

    // Insert at beginning
    printf("\n\nEnter element to be inserted at beginning: ");
    scanf("%d", &x);
    if (head != NULL)
        insertb(x);
    else {
        head = (node*)malloc(sizeof(node));
        head->data = x;
        head->left = NULL;
        head->right = NULL;
    }
    printf("\nList after inserting at the beginning: ");
    display();
    // Insert at end
    printf("\n\nEnter element to be inserted at end: ");
    scanf("%d", &x);
    for (pre = head; pre->right != NULL; pre = pre->right);
    inserte(x);
    printf("\nList after inserting at the end: ");
    display();

    // Delete at beginning
    if (head != NULL && head->right != NULL)
        deleteb();
    printf("\n\nList after deleting from the beginning: ");
    display();

    // Delete at end
    for (pre = head; pre->right->right != NULL; pre = pre->right);
    deletee();
    printf("\n\nList after deleting from the end: ");
    display();

    // Insert after a given node
    printf("\n\nEnter the data of the node after which to insert: ");
    scanf("%d", &x);
    pre = search(x);
    if (pre != NULL) {
        printf("Enter element to be inserted: ");
        scanf("%d", &x);
        insertan(x);
    }
    printf("\nList after inserting: ");
    display();

  // Delete after a given node
  printf("\nEnter the data of the node after which to delete: ");
  scanf("%d", &x);
  pre = search(x);
  if (pre != NULL && pre->right != NULL) {
    if (pre->right->right != NULL)
        deletean(); // safe to access tmp->right->left
    else {
        tmp = pre->right;
        pre->right = NULL;
        free(tmp);
    }
  } else {
    printf("\nNode with data %d not found or no node to delete after it", x);
  }
  printf("\nList after deleting node: ");
  display();
return 0;
  }
```

Output:

Result: Thus, the program to implement Doubly Linked List with operations such as create, insert, search, delete, and traverse has been written, executed, and output verified.

EX.NO:2 **Implementation Of Polynomial ADT Using Linked List**
DATE:

Aim: To write a program to implement Polynomial Manipulation using Linked List to add add two polynomial .

Algorithm

1. **Start the program** and define a node structure with fields for coefficient, power, and next pointer.
2. **Initialize three linked lists**: one for each input polynomial and one for the result.
3. **Ask the user** for the number of terms in each polynomial.
4. **Input each term's coefficient and power** for both polynomials and insert them into their respective lists.
5. **Traverse both polynomials simultaneously** to compare powers of each term.
6. **If powers are equal**, add the coefficients and insert the result into the result list.
7. **If powers are unequal**, insert the term with the higher power into the result list and move its pointer forward.
8. **Continue until all terms are processed** from both polynomials.
9. **Display the final result** by printing each term in the result polynomial.

Program

```
// EX.NO.2 Implementation of Polynomial ADT using Linked list
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

// Define the node structure
typedef struct node {
   int co, po; // Coefficient & Power (exponent)
   struct node* next; // Pointer to the next node
} node;

struct node *poly = NULL, *poly1 = NULL, *poly2 = NULL, *tmp, *cur;

// Function to insert a term into the result polynomial
void insert(node** poly, int co, int po) {
   tmp = (node*)malloc(sizeof(node));
   tmp->co = co;
   tmp->po = po;
   tmp->next = NULL;
   if (*poly == NULL) {
      *poly = tmp;
   } else {
      for (cur = *poly; cur->next != NULL; cur = cur->next);
      cur->next = tmp;
   }
}
```

```
int main() {
    int i, n, co1, co2, po;

    // Input for both polynomials simultaneously
    printf("Enter the number of terms (same for both polynomials): ");
    scanf("%d", &n);

    printf("Enter coefficient and power for each term:\n");
    for (i = 0; i < n; ++i) {
        printf("Term %d:\n", i + 1);
        printf("  Power: "); scanf("%d", &po);
        printf("  Coeff of Poly1: "); scanf("%d", &co1);
        printf("  Coeff of Poly2: "); scanf("%d", &co2);
        insert(&poly1, co1, po);
        insert(&poly2, co2, po);
    }

    // Perform polynomial addition
    while (poly1 != NULL && poly2 != NULL) {
        if (poly1->po == poly2->po) {
            insert(&poly, poly1->co + poly2->co, poly1->po);
            poly1 = poly1->next;
            poly2 = poly2->next;
        } else if (poly1->po > poly2->po) {
            insert(&poly, poly1->co, poly1->po);
            poly1 = poly1->next;
        } else {
            insert(&poly, poly2->co, poly2->po);
            poly2 = poly2->next;
        }
    }

    // Print the result
    printf("Resultant polynomial: ");
    for (cur = poly; cur != NULL; cur = cur->next) {
        printf("(%dx^%d)", cur->co, cur->po);
        if (cur->next != NULL) {
            printf(" + ");
        }
    }

   }
```

Output:

Result: Thus, the program to implement Polynomial Manipulation using Linked List by adding two polynomial has been written, executed, and output verified.

EX.NO:3 (A) **Array Implementation Of Stack ADT**
DATE:

Aim : To Write a program to implement array implementation of Stack ADT with push, pop, and display operations.

Algorithm:

1. **Start the program** and declare an array `s[]` to hold stack elements and a variable `top = -1`.
2. **Display a menu** with choices for push, pop, display, and exit.
3. **If the user chooses push**, check if the stack is full (`top == size - 1`).
4. **If not full**, increment `top` and insert the value at `s[top]`.
5. **If the user chooses pop**, check if the stack is empty (`top == -1`).
6. **If not empty**, remove the top element and decrement `top`.
7. **If the user chooses display**, print all elements from `top` to `0`.
8. **Repeat the menu until the user chooses exit**, then stop the program.

Program

```
// 3(A) Array implementation of Stack ADT
 #include <stdio.h>
 #include <stdlib.h>
#define size 5
int s[size], top = -1, i, x;
int main() {
    int ch;
     printf("Array Implementation of Stack ");
    while (1) {
        printf("\n 1. Push 2. Pop 3. Display 4. Exit");
        printf("\n Enter your choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1:  printf("Enter value to push: ");
                     scanf("%d", &x);
                     if (top == size - 1) {
                     printf("Stack is FULL\n");
                     } else {
                     top = top + 1;
                     s[top] = x;
                     printf("%d pushed onto stack\n", x);
                     }
                     break;
            case 2:if (top == -1) {
                    printf("Stack is empty\n");
                    } else {
                    x = s[top];
                   top = top - 1;
                   printf("%d popped from stack\n", x);
                   }
                   break;
            case 3: if (top == -1) {
                     printf("Stack is empty\n");
                    } else {
                    printf("Stack elements are:\n");
                   for (i = top; i >= 0; i--) {
                    printf("%d ", s[i]);
                   }
                   printf("\n");
                   }
                   break;
            case 4: exit(0);
                   default:
                   printf("Invalid choice\n");
        }
    }

 return 0;
}
```

Output:

Result: Thus, the program to implement Stack using Array with push, pop, and display operations. has been written, executed, and output verified

EX.NO. 3(B) **Array Implementation Of Queue ADT**
DATE:

Aim : To Write a program to implement array implementation of Queue ADT with enqueue, dequeue, and display operations.

Algorithm:

1. Start the program and declare an array q[size] to store queue elements.
2. Initialize front = 0 and rear = -1 to track the queue boundaries.
3. Display a menu with choices: Enqueue, Dequeue, Display, and Exit.
4. If the user chooses Enqueue, check if rear == size - 1 (queue full).
5. If not full, increment rear and insert the new element at q[rear].
6. If the user chooses Dequeue, check if front > rear (queue empty).
7. If not empty, remove the element at q[front] and increment front.
8. If the user chooses Display, check if queue is empty (front > rear).
9. If not empty, print all elements from q[front] to q[rear].
10. Repeat steps 3–9 until the user chooses Exit to terminate the program.

Program

```
//EX.NO 3  (B)ARRAY IMPLEMENTATION OF QUEUE
#include <stdio.h>
#include <stdlib.h>
#define size 5
int main() {
   int q[size], front = 0, rear = -1, ch, i, x;
    printf("\nArray Implementation of Queue\n");
   while (1) {
      printf("\n1. Enqueue 2. Dequeue 3. Display 4. Exit: ");
      printf("\nEnter your choice: ");
      scanf("%d", &ch);
      switch (ch) {
         case 1:              if (rear == size - 1) {
               printf("Queue is full\n");
            } else {
               printf("Enter element to enqueue: ");
               scanf("%d", &x);
               rear = rear + 1;
               q[rear] = x;
            }
            break;
         case 2:              if (front > rear) {
               printf("Queue is empty\n");
            } else {
               x = q[front];
               front = front + 1;
               printf("Dequeued element: %d\n", x);
            }
            break;
```

```
        case 3:             if (front > rear) {
                  printf("Queue is empty\n");
              } else {
                  printf("Queue elements: ");
                  for (i = front; i <= rear; i++) {
                      printf("%d ", q[i]);
                  }
                  printf("\n");
              }
              break;

        case 4:
              exit(0);

        default:
              printf("Invalid choice\n");
      }
   }
   return 0;
}
```

Output:

Result : Thus, the program to implement Queue using Array with enqueue, dequeue, and display operations has been written, executed, and output verified.