Experiment No: 1 (A)

Experiment Name:

Algorithm:

Source Code:

Experiment No: 1 (A)

Experiment Name: Write the selection sort algorithm and state its corresponding program to sort elements.

Algorithm:

1. Start
2. Set MIN to location 0
3. Search the minimum element in the list
4. Swap with value at location MIN
5. Increment MIN to point to next element
6. Repeat until list is sorted
7. Stop

Source Code:

```
#include <stdio.h>
void swap(int *a, int *b)
{
   int temp = *a;
   *a = *b;
   *b = temp;
}
void selectionSort(int arr[], int n)
{
   for (int j = 0; j < n - 1; j++)
   {
      int min_idx = j;
      for (int i = j + 1; i < n; i++)
      {
         if (arr[i] < arr[min_idx]) min_idx = i;
      }
      swap(&arr[min_idx], &arr[j]);
   }
}

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

```
int main()
{
   int arr[] = {20, 12, 10, 15, 2};
   int n = sizeof(arr) / sizeof(arr[0]);
   printArray(arr, n);
   selectionSort(arr, n);
   printf("Sorted Array: ");
   printArray(arr, n);
}
```

Output:
20 12 10 15 2
Sorted Array: 2 10 12 15 20

Experiment No: 1 (B)

Experiment Name: Write the algorithms and state their corresponding programs to insert an element at the (i) beginning (ii) middle (iii) end position in arrays.

Algorithm:

1. Start
2. Check if the array is full. If the array is full, print an error message and exit.
3. Insertion
   Insert at Begin:
      i. Shift all existing elements to the right by one position (from the last element to the first).
      ii. Insert the new element at the beginning of the array.
   Insert at Middle:
      i. Check if the specified position is valid (0 <= position <= size). - If the position is not valid, print an error message and exit.
      ii. Shift all elements from the specified position to the right by one position.
      iii. Insert the new element at the specified position.
   Insert at End:
      i. Insert the new element at the end of the array (at index size)
4. Increase the size of the array by 1.
5. Display inserted array.
6. Stop

Source Code:

```
#include <stdio.h>
#define MAX_SIZE 10

void insertFirst(int arr[], int *size, int value) {
   if (*size >= MAX_SIZE) {
      printf("Array is full. Cannot insert.\n");
      return;
   }
   for (int i = *size; i > 0; i--) {
      arr[i] = arr[i - 1];
   }
   arr[0] = value;
   (*size)++;
}

void insertMid(int arr[], int *size, int value, int pos) {
   if (*size >= MAX_SIZE) {
      printf("Array is full. Cannot insert.\n");
      return;
   }
   if (pos < 0 || pos > *size) {
      printf("Invalid position. Cannot insert.\n");
      return;
   }
   for (int i = *size; i > pos; i--) {
      arr[i] = arr[i - 1];
   }
   arr[pos] = value;
   (*size)++;
}

void insertEnd(int arr[], int *size, int value) {
   if (*size >= MAX_SIZE) {
      printf("Array is full. Cannot insert.\n");
      return;
   }
   arr[*size] = value;
   (*size)++;
}
```

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

int main() {
   int arr[] = {10, 20, 30, 40, 50};
   int size = sizeof(arr) / sizeof(arr[0]);
   int choice, value, pos;

   while (1) {
      printf("\nArray: ");
      printArray(arr, size);
      printf("Menu:\n");
      printf("1. Insert at the beginning\n");
      printf("2. Insert at the middle\n");
      printf("3. Insert at the end\n");
      printf("4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            printf("Enter value: ");
            scanf("%d", &value);
            insertFirst(arr, &size, value);
            break;
         case 2:
            printf("Enter mid position to insert: ");
            scanf("%d", &pos);
            printf("Enter value: ");
            scanf("%d", &value);
            insertMid(arr, &size, value, pos);
            break;
         case 3:
            printf("Enter value: ");
            scanf("%d", &value);
            insertEnd(arr, &size, value);
```

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

    printf("Updated Array: ");
    printArray(arr, size);
  }
  return 0;
}
```

Output:

```
Array: 10 20 30 40 50
Menu:
1. Insert at the beginning
2. Insert at the middle
3. Insert at the end
4. Exit
Enter your choice: 1
Enter value: 5
Updated Array: 5 10 20 30 40 50

Array: 5 10 20 30 40 50
Menu:
1. Insert at the beginning
2. Insert at the middle
3. Insert at the end
4. Exit
Enter your choice: 4
Exiting.
```

Experiment No: 2 (A)

Experiment Name: Write the insertion sort algorithm and state its corresponding program to sort elements.

Algorithm:

1. Start with the second element (index 1) and consider it as the current element.
2. Compare the current element with the elements before it in the sorted portion.
3. Move elements in the sorted portion that are greater than the current element one position to the right.
4. Insert the current element into its correct position in the sorted portion.
5. Repeat steps 2-4 for all remaining elements in the array.

Source Code:

```
#include <stdio.h>
void insertionSort(int arr[], int n)
{
   int i, j, key;
   for (i = 1; i < n; i++)
   {
      key = arr[i];
      j = i - 1;
      // Move elements of the sorted portion that are greater than the key
      while (j >= 0 && arr[j] > key)
      {
         arr[j + 1] = arr[j];
         j = j - 1;
      }
      // Insert the key into its correct position in the sorted portion
      arr[j + 1] = key;
   }
}
void printArray(int arr[], int n)
{
   for (int i = 0; i < n; i++)
   {
      printf("%d ", arr[i]);
   }
   printf("\n");
}
```

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

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

   insertionSort(arr, n);

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

   return 0;
}
```

Output:
Original array: 12 11 13 5 6
Sorted array: 5 6 11 12 13

Experiment No: 2 (B)

Experiment Name: Write the algorithms and state their corresponding programs to delete an element from the (i) beginning (ii) middle (iii) end position in the arrays.

Algorithm:

1. Start
2. Check if the array is empty. If the array is empty, print an error message and exit.
   Delete at Begin:
      i. Shift all elements one position to the left, starting from the second element to the last element.
   Delete at Middle:
      i. Check if the specified position is valid (0 <= position < size). If the position is not valid, print an error message and exit.
      ii. Shift all elements from the specified position + 1 to the left, effectively overwriting the element at the specified position.
   Delete at End:
      i. Check if the array is empty (size is 0). If it's empty, print an error message and exit.

3. Decrease the size of the array by 1.
4. Display deleted array.
5. Stop

Source Code:

```
#include <stdio.h>
#define MAX_SIZE 10
void deleteFirst(int arr[], int *size)
{
   if (*size <= 0)
   {
      printf("Array is empty. Cannot delete.\n");
      return;
   }
   for (int i = 0; i < *size - 1; i++)
   {
      arr[i] = arr[i + 1];
   }
   (*size)--;
}

void deleteMid(int arr[], int *size, int pos)
{
   if (*size <= 0)
   {
      printf("Array is empty. Cannot delete.\n");
      return;
   }
   if (pos < 0 || pos >= *size)
   {
      printf("Invalid position. Cannot delete.\n");
      return;
   }
   for (int i = pos; i < *size - 1; i++)
   {
      arr[i] = arr[i + 1];
   }
   (*size)--;
}
```

```
void deleteEnd(int arr[], int *size)
{
   if (*size <= 0)
   {
      printf("Array is empty. Cannot delete.\n");
      return;
   }
   (*size)--;
}

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

int main()
{
   int arr[] = {10, 20, 30, 40, 50, 60, 70};
   int size = sizeof(arr) / sizeof(arr[0]);
   int choice, pos;

   while (1)
   {
      printf("\nArray: ");
      printArray(arr, size);
      printf("Menu:\n");
      printf("1. Delete from the beginning\n");
      printf("2. Delete from the middle\n");
      printf("3. Delete from the end\n");
      printf("4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice)
      {
```

```
        case 1:
            deleteFirst(arr, &size);
            break;
        case 2:
            printf("Enter mid position to delete: ");
            scanf("%d", &pos);
            deleteMid(arr, &size, pos);
            break;
        case 3:
            deleteEnd(arr, &size);
            break;
        case 4:
            printf("Exiting.\n");
            return 0;
        default:
            printf("Invalid choice.\n\n");
        }

        printf("Updated Array: ");
        printArray(arr, size);
    }
    return 0;
}
```

Output:

```
Array: 10 20 30 40 50 60 70
Menu:
1. Delete from the beginning
2. Delete from the middle
3. Delete from the end
4. Exit
Enter your choice: 3
Updated Array: 10 20 30 40 50 60

Array: 10 20 30 40 50 60
Menu:
1. Delete from the beginning
2. Delete from the middle
3. Delete from the end
```

4. Exit
Enter your choice: 3
Updated Array: 10 20 30 40 50

Array: 10 20 30 40 50
Menu:
1. Delete from the beginning
2. Delete from the middle
3. Delete from the end
4. Exit
Enter your choice: 4
Exiting.

Experiment No: 3 (A)

Experiment Name: Design the algorithms and write their corresponding programs to (i) insert (ii) delete an element in arrays.

Algorithm:

1. Start
2. *Insertion:*
    i. Check if the array is full. If it's full, print an error message and exit.
    ii. Ask the user for the value to insert.
    iii. Ask the user for the position where the element should be inserted (0 <= position <= size).
    iv. Shift all elements from the specified position to the right by one position.
    v. Insert the new element at the specified position.
    vi. Increase the size of the array by 1.

    Deletion:
    i. Check if the array is empty (size is 0). If it's empty, print an error message and exit.
    ii. Ask the user for the position of the element to delete (0 <= position < size).
    iii. Shift all elements from the specified position + 1 to the left, effectively overwriting the element at the specified position.
    iv. Decrease the size of the array by 1.

3. Display Updated array.
4. Stop

Source Code:

```
#include <stdio.h>
#define MAX_SIZE 10
void insertElement(int arr[], int *size) {
   if (*size >= MAX_SIZE) {
      printf("Array is full. Cannot insert.\n");
      return;
   }
   int position, value;
   printf("Enter the value to insert: ");
   scanf("%d", &value);
   printf("Enter the position to insert (0 <= position <= %d): ", *size);
   scanf("%d", &position);

   if (position < 0 || position > *size) {
      printf("Invalid position. Cannot insert.\n");
      return;
   }
   // Shift elements to the right
   for (int i = *size; i > position; i--) {
      arr[i] = arr[i - 1];
   }
   // Insert the value at the specified position
   arr[position] = value;
   (*size)++;
}

void deleteElement(int arr[], int *size) {
   if (*size <= 0) {
      printf("Array is empty. Cannot delete.\n");
      return;
   }
   int position;
   printf("Enter the position to delete (0 <= position < %d): ", *size);
   scanf("%d", &position);

   if (position < 0 || position >= *size) {
      printf("Invalid position. Cannot delete.\n");
      return;
   }
```

```
    // Shift elements to the left
    for (int i = position; i < *size - 1; i++) {
        arr[i] = arr[i + 1];
    }
    (*size)--;
}

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

int main() {
    int arr[] = {10, 20, 30, 40, 50, 60};
    int size = sizeof(arr) / sizeof(arr[0]);
    int choice;

    while (1) {
        printf("\nOriginal Array: ");
        printArray(arr,size);
        printf("Menu:\n");
        printf("1. Insert an element\n");
        printf("2. Delete an element\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                insertElement(arr, &size);
                break;
            case 2:
                deleteElement(arr, &size);
                break;
            case 3:
                printf("Exiting.\n");
                return 0;
```

```
            default:
                printf("Invalid choice.\n\n");
        }
        printf("Updated Array: ");
        printArray(arr,size);
    }
    return 0;
}
```

Output:

```
Original Array: 10 20 30 40 50 60
Menu:
1. Insert an element
2. Delete an element
3. Exit
Enter your choice: 1
Enter the value to insert: 70
Enter the position to insert (0 <= position <= 6): 6
Updated Array: 10 20 30 40 50 60 70

Original Array: 10 20 30 40 50 60 70
Menu:
1. Insert an element
2. Delete an element
3. Exit
Enter your choice: 2
Enter the position to delete (0 <= position < 7): 6
Updated Array: 10 20 30 40 50 60

Original Array: 10 20 30 40 50 60
Menu:
1. Insert an element
2. Delete an element
3. Exit
Enter your choice: 3
Exiting.
```

Experiment No: 3 (B)

Experiment Name: Design an algorithm and state its corresponding program to understand pointer to structure.

Algorithm:

1. Define a structure with one or more members.
2. Declare a pointer to the structure.
3. Allocate memory for the structure using **malloc** or a similar function.
4. Use the pointer to access and modify the structure members.
5. Print the original and modified structure members.
6. Deallocate memory to prevent memory leaks.

Source Code:

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

struct Student {
   int id;
   char name[50];
   float gpa;
};
int main() {
   struct Student *ptr;
   ptr = (struct Student *)malloc(sizeof(struct Student));
   if (ptr == NULL) {
      printf("Memory allocation failed.\n");
      return 1;
   }
   ptr->id = 101;
   strcpy(ptr->name , "Sajjad Hossain");
   ptr->gpa = 4.00;
   printf("Student Data:\n");
   printf("ID: %d\n", ptr->id);
   printf("Name: %s\n", ptr->name);
   printf("GPA: %.2f\n", ptr->gpa);
   free(ptr);
   return 0;
}
```

Output:

```
Student Data:
ID: 101
Name: Sajjad Hossain
GPA: 4.00
```

Experiment No: 4 (A)

Experiment Name: Design the algorithms and write their corresponding programs to (i) insert (ii) delete an element in arrays.

Algorithm:

1. Start
2. *Search:*
    i. Prompt the user to enter the element to search for.
    ii. Iterate through the array elements to find a match with the input element.
    iii. If a match is found, print the index where the element was found.
    iv. If no match is found, print a message indicating that the element was not found.

    Update:
    i. Prompt the user to enter the element to update and the new value.
    ii. Iterate through the array elements to find a match with the input element.
    iii. If a match is found, update the element with the new value.
    iv. If no match is found, print a message indicating that the element was not found.
3. Stop

Source Code:

```
#include <stdio.h>

int search(int arr[], int size, int value)
{
   for (int i = 0; i < size; i++)
   {
      if (arr[i] == value)
      {
         printf("Element %d found at index %d.\n", value, i);
         return i;
      }
   }
   printf("Element %d not found in the array.\n", value);
   return -1;
}
```

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

int main()
{
   int arr[] = {10, 20, 30, 40, 50};
   int size = sizeof(arr) / sizeof(arr[0]);
   int value, newValue;

   printf("Original Array: ");
   printArray(arr, size);

   // Search for an element
   printf("Enter the element to search for: ");
   scanf("%d", &value);
   search(arr, size, value);

   //Update an element
   printf("Enter the element to update: ");
   scanf("%d", &value);
   int foundIndex = search(arr, size, value);

   if (foundIndex != -1)
   {
      printf("Enter the new value to update: ");
      scanf("%d", &newValue);
      arr[foundIndex]=newValue;
      printf("Updated Array: ");
      printArray(arr, size);
   }

   return 0;
}
```

Output:

```
Original Array: 10 20 30 40 50
Enter the element to search for: 60
Element 60 not found in the array.
Enter the element to update: 30
Element 30 found at index 2.
Enter the new value for element 30: 60
Updated Array: 10 20 60 40 50
```

Experiment No: 4 (B)

Experiment Name: Design an algorithm and state its corresponding program to understand pointer to arrays

Algorithm:

1. Start
2. Declare an integer array.
3. Declare a pointer to an integer.
4. Assign the address of the array to the pointer.
5. Use the pointer to access and modify elements in the array.
6. Print the original and modified array elements.
7. End

Source Code:

```
#include <stdio.h>
void printArray(int arr[], int n)
{
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
}
int main() {
   int arr[] = {10, 20, 30, 40, 50};
   int n = sizeof(arr)/sizeof(arr[0]);
   int *ptr = arr; // Declare a pointer and assign the array's address

   printf("Original Array: ");
   printArray(arr,n);

   // Use the pointer to modify elements in the array
   ptr[1] = 99;
   ptr[3] = 77;

   printf("Modified Array: ");
   printArray(arr,n);

   return 0;
}
```

Output:

Original Array: 10 20 30 40 50
Modified Array: 10 99 30 77 50

Experiment No: 5 (A)

Experiment Name: Design the algorithms and write their corresponding programs to (i) add elements (ii) access elements in a vector data structure

Algorithm:

1. Start
2. Include the header file <vector> into the program.
3. Initialize v as vector.
4. To add element in vector, use push_back() library function.
5. Use value 0 to stop adding element.
6. To access element in vector, use at() library function
7. Print the elements.
8. Stop

Source Code:

```
#include <iostream>

#include <vector>

using namespace std;

int main()

{

   vector <int> v; // Declare a dynamic array (vector) of integers

   cout << "Enter values to add (0 to stop):\n";

   while (true)

   {

      int value;

      cout << "Enter value: ";

      cin >> value;

      if (value == 0) break;

      v.push_back(value); // Add the element to the vector

   }
```

```
    cout << "Vector elements:\n";
    for (int i = 0; i < v.size(); i++)
    {
        cout << v.at(i) << " "; // Access and print elements using at() library function
    }
    cout << endl;

    return 0;
}
```

Output:

```
Enter values to add (0 to stop):
Enter value: 1
Enter value: 2
Enter value: 3
Enter value: 4
Enter value: 0
Vector elements:
1 2 3 4
```

Experiment No: 5 (B)

Experiment Name: Write the algorithms and state their corresponding programs to delete a node from the (i) Beginning (ii) Middle (iii) End position in a linked list.

Algorithm:

1. Start
2. Check if the linked list is empty. If it is, print an error message and return.
3. Deletion Process

    Deleting from the Beginning of a Linked List:
    i. Create a temporary pointer to the first node.
    ii. Update the head of the linked list to point to the next node.
    iii. Delete the temporary pointer.

    Deleting from the Middle of a Linked List:
    i. Prompt the user to enter the position (0-based) of the node to delete.
    ii. Traverse the linked list until the previous node of the target node is reached.
    iii. Update the **next** pointer of the previous node to skip the target node.
    iv. Delete the target node.

    Deleting from the End of a Linked List:
    i. Traverse the linked list until the second-to-last node is reached.
    ii. Update the **next** pointer of the second-to-last node to **NULL**.
    iii. Delete the last node.

4. Decrease the size of the linked list.
5. Stop

Source Code:

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

// Define the structure for a singly linked list node
struct Node
{
   int data;
   struct Node *next;
};
```

```
//Checking Linked List Empty or Not
void checkEmpty(struct Node **head)
{
   if (*head == NULL)
   {
      printf("Linked list is empty. Cannot delete.\n");
      return;
   }
}

void deleteFromBeginning(struct Node **head)
{
   checkEmpty(head);
   struct Node *temp = *head;
   *head = (*head)->next;
   free(temp);
}

void deleteFromMiddle(struct Node **head, int position)
{
   checkEmpty(head);
   if (position < 0)
   {
      printf("Invalid position. Cannot delete.\n");
      return;
   }
   if (position == 0)
   {
      deleteFromBeginning(head);
      return;
   }
   struct Node *current = *head;
   struct Node *prev = NULL;
   int count = 0;
   while (current != NULL && count < position)
   {
      prev = current;
      current = current->next;
      count++;
   }
```

```
    if (current == NULL)
    {
        printf("Invalid position. Cannot delete.\n");
        return;
    }

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

void deleteFromEnd(struct Node **head)
{
   checkEmpty(head);

    struct Node *current = *head;
    struct Node *prev = NULL;

    while (current->next != NULL)
    {
        prev = current;
        current = current->next;
    }

    if (prev == NULL)
    {
        free(*head); // Only one node in the list
        *head = NULL;
    }
    else
    {
        prev->next = NULL;
        free(current);
    }
}
```

```
void printLinkedList(struct Node *head)
{
   struct Node *current = head;

   while (current != NULL)
   {
      printf("%d ", current->data);
      current = current->next;
   }
   printf("\n");
}

int main()
{
   // Creating Nodes and memory Allocation
   struct Node *head = malloc(sizeof(struct Node));
   struct Node *one = malloc(sizeof(struct Node));
   struct Node *two = malloc(sizeof(struct Node));
   struct Node *three = malloc(sizeof(struct Node));
   struct Node *four = malloc(sizeof(struct Node));

   // Asign Values
   one->data = 10;
   two->data = 20;
   three->data = 30;
   four->data = 40;

   // Connect
   one->next = two;
   two->next = three;
   three->next = four;
   four->next = NULL;
   head = one;

   // Print the original linked list
   printf("Linked List: ");
   printLinkedList(head);

   int choice, position;
```

```
    while (1)
    {
        printf("\nMenu:\n");
        printf("1. Delete from the beginning\n");
        printf("2. Delete from the middle\n");
        printf("3. Delete from the end\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            deleteFromBeginning(&head);
            break;
        case 2:
            printf("Enter the position (0-based) to delete: ");
            scanf("%d", &position);
            deleteFromMiddle(&head, position);
            break;
        case 3:
            deleteFromEnd(&head);
            break;
        case 4:
            printf("Exiting.\n");
            return 0;
        default:
            printf("Invalid choice.\n\n");
        }

        // Print the updated linked list
        printf("Updated Linked List: ");
        printLinkedList(head);
    }

    return 0;
}
```

Output:

```
Linked List: 10 20 30 40

Menu:
1. Delete from the beginning
2. Delete from the middle
3. Delete from the end
4. Exit
Enter your choice: 1
Updated Linked List: 20 30 40

Menu:
1. Delete from the beginning
2. Delete from the middle
3. Delete from the end
4. Exit
Enter your choice: 3
Updated Linked List: 20 30

Menu:
1. Delete from the beginning
2. Delete from the middle
3. Delete from the end
4. Exit
Enter your choice: 2
Enter the position (0-based) to delete: 1
Updated Linked List: 20

Menu:
1. Delete from the beginning
2. Delete from the middle
3. Delete from the end
4. Exit
Enter your choice: 1
Updated Linked List:

Menu:
1. Delete from the beginning
2. Delete from the middle
3. Delete from the end
4. Exit
Enter your choice: 1
Linked list is empty. Cannot delete.
```

Experiment No: 6 (A)

Experiment Name: Design the algorithms and state their corresponding programs to (i) update vector elements (ii) delete vector elements in a vector data structure.

Algorithm:

1. Start
2. Include the header file <vector> into the program.
3. Initialize v as vector.
4. To update element in vector, use at() library function
5. To delete last element in vector, use pop_back() library function.
6. Print the elements.
7. Stop

Source Code:

```
#include <iostream>
#include <vector>
using namespace std;

void printVector(const vector<int> &v)
{
   for (int i = 0; i < v.size(); i++)
   {
      cout << v.at(i) << " ";
   }
   cout << endl;
}

// Function to update a vector element at a given index
void updateElement(vector<int> &v, int index, int newValue)
{
   if (index >= 0 && index < v.size())
   {
      v.at(index) = newValue;
      cout << "Element updated.\n";
   }
   else cout << "Invalid index. Element not updated.\n";
}
```

```
// Function to delete the last element from the vector
void deleteLastElement(vector<int> &v)
{
   if (!v.empty())
   {
      v.pop_back();
      cout << "Last element deleted.\n";
   }
   else cout << "Vector is empty. Cannot delete.\n";
}

int main()
{
   vector<int> v = {10, 20, 30, 40, 50};

   printf("Initial Vector: ");
   printVector(v);

   int choice, index, newValue;

   while (true)
   {
      cout << "\nMenu:\n";
      cout << "1. Update element\n";
      cout << "2. Delete last element\n";
      cout << "3. Exit\n";
      cout << "Enter your choice: ";
      cin >> choice;

      switch (choice)
      {
      case 1:
         cout << "Enter the index of the element to update: ";
         cin >> index;
         cout << "Enter the new value: ";
         cin >> newValue;
         updateElement(v, index, newValue);
         break;
```

Output:

```
Initial Vector: 10 20 30 40 50

Menu:
1. Update element
2. Delete last element
3. Exit
Enter your choice: 1
Enter the index of the element to update: 1
Enter the new value: 25
Element updated.
Updated Vector: 10 25 30 40 50

Menu:
1. Update element
2. Delete last element
3. Exit
Enter your choice: 2
Last element deleted.
Updated Vector: 10 25 30 40

Menu:
1. Update element
2. Delete last element
3. Exit
Enter your choice: 3
Exiting.
```

```
      case 2:
        deleteLastElement(v);
        break;

      case 3:
        cout << "Exiting.\n";
        return 0;

      default:
        cout << "Invalid choice.\n";
      }
      printf("Updated Vector: ");
      printVector(v);
    }
    return 0;
}
```

Experiment No: 6 (B)

Experiment Name: Write the algorithms and state their corresponding programs to insert a node in the (i) Beginning (ii) Middle (iii) End position in a linked list

Algorithm:

1. Start
2. *Insertion at the Beginning of a Linked List:*
   i. Create a new node with the given data.
   ii. Set the new node's next pointer to point to the current head of the linked list.
   iii. Update the head of the linked list to point to the new node.

   Insertion in the Middle of a Linked List:
   i. Prompt the user to enter the position (0-based) at which to insert the node.
   ii. Create a new node with the given data.
   iii. Traverse the linked list until the previous node of the target position is reached.
   iv. Update the next pointer of the previous node to point to the new node.
   v. Set the new node's next pointer to point to the node originally at the target position.

Insertion at the End of a Linked List:

i. Create a new node with the given data.
ii. If the linked list is empty, set the head to point to the new node.
iii. Otherwise, traverse the linked list until the last node is reached. iv. Update the next pointer of the last node to point to the new node.

3. Increase the size of the linked list.
4. Stop

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

struct Node

{

   int data;

   struct Node *next;

};


void insertAtBeginning(struct Node **head, int data)

{

   // Creating New Node

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

   newNode->data = data;


   // Inserting

   newNode->next = *head;

   *head = newNode;

}


void insertAtMiddle(struct Node **head, int position, int data)

{
```

```
if (position < 0)
{
   printf("Invalid position. Cannot insert.\n");
   return;
}
if (position == 0)
{
   insertAtBeginning(head, data);
   return;
}
struct Node *current = *head;
struct Node *prev = NULL;
int count = 0;

// Traverse to find the previous node of the target position
while (current != NULL && count < position)
{
   prev = current;
   current = current->next;
   count++;
}
if (current == NULL && count != position)
{
   printf("Invalid position. Cannot insert.\n");
   return;
}

// Creating New Node
```

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

    // Inserting between prev and current
    prev->next = newNode;
    newNode->next = current;
}
void insertAtEnd(struct Node **head, int data)
{
    // Creating New Node
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    if (*head == NULL)
    {
        // If the linked list is empty, set the head to the new node
        *head = newNode;
        return;
    }
    // Traverse to find the last node
    struct Node *current = *head;
    while (current->next != NULL)
    {
        current = current->next;
    }
    // Update the last node's next pointer to point to the new node
    current->next = newNode;
}
```

```
void printLinkedList(struct Node *head)
{
   struct Node *current = head;
   while (current != NULL)
   {
      printf("%d ", current->data);
      current = current->next;
   }
   printf("\n");
}
int main()
{
   // Creating Nodes and memory Allocation
   struct Node *head = malloc(sizeof(struct Node));
   struct Node *one = malloc(sizeof(struct Node));
   struct Node *two = malloc(sizeof(struct Node));
   struct Node *three = malloc(sizeof(struct Node));

  // Asign Values
   one->data = 10;
   two->data = 20;
   three->data = 30;

   // Connect
   one->next = two;
   two->next = three;
   three->next = NULL;
   head = one;
```

```
// Print the original linked list
printf("Linked List: ");
printLinkedList(head);
int choice, position, data;
while (1)
{
   printf("\nMenu:\n");
   printf("1. Insert at the beginning\n");
   printf("2. Insert at the middle\n");
   printf("3. Insert at the end\n");
   printf("4. Exit\n");
   printf("Enter your choice: ");
   scanf("%d", &choice);
   if (choice != 4)
   {
      printf("Enter data to insert: ");
      scanf("%d", &data);
   }
   switch (choice)
   {
   case 1:
      insertAtBeginning(&head, data);
      break;
   case 2:
      printf("Enter position (0-based) to insert: ");
      scanf("%d", &position);
      insertAtMiddle(&head, position, data);
      break;
```

Output:

Linked List: 10 20 30

Menu:

1. Insert at the beginning

2. Insert at the middle

3. Insert at the end

4. Exit

Enter your choice: 1

Enter data to insert: 5

Updated Linked List: 5 10 20 30

Menu:

1. Insert at the beginning

2. Insert at the middle

3. Insert at the end

4. Exit

Enter your choice: 3

Enter data to insert: 40

Updated Linked List: 5 10 20 30 40

Menu:

1. Insert at the beginning

2. Insert at the middle

3. Insert at the end

4. Exit

Enter your choice: 2

Enter data to insert: 25

Enter position (0-based) to insert: 3

Updated Linked List: 5 10 20 25 30 40

Menu:

1. Insert at the beginning

2. Insert at the middle

3. Insert at the end

4. Exit

Enter your choice: 4

Exiting.

```
        case 3:
            insertAtEnd(&head, data);
            break;
        case 4:
            printf("Exiting.\n");
            return 0;
        default:
            printf("Invalid choice.\n\n");
        }

        // Print the updated linked list
        printf("Updated Linked List: ");
        printLinkedList(head);
    }
    return 0;
}
```