Batch:A2	**Roll No.:16010123032**
Experiment / assignment / tutorial No. 3	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

Title: Implementation of basic Linked List – creation, insertion, deletion, traversal, searching an element

Objective: To understand the advantage of linked list over other structures like arrays in implementing the general linear list

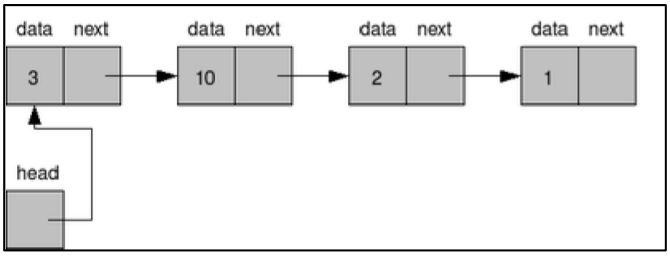
Expected Outcome of Experiment:

CO	Outcome
1	Comprehend the different data structures used in problem solving

Books/ Journals/ Websites referred:

Introduction:

A linear list is a list where each element has a unique successor. There are four common operations associated with a linear list: insertion, deletion, retrieval, and traversal. Linear list can be divided into two categories: general list and restricted list. In general list the data can be inserted or deleted without any restriction whereas in restricted list there is restrictions for these operations. Linked list and arrays are commonly used to implement general linear list. A linked list is simply a chain of structures which contain a pointer to the next element. It is dynamic in nature. Items may be added to it or deleted from it at will.



A list item has a pointer to the next element, or to NULL if the current element is the tail (end of the list). This pointer points to a structure of the same type as itself. This Structure that contains elements and pointers to the next structure is called a Node.

Related Theory: -

In computer science, a linked list is a linear collection of data elements, whose order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence. In its most basic form, each node contains: data, and a reference to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration.

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers

Linked List ADT:

Algorithm for creation, insertion, deletion, traversal and searching an element in linked list:

Step 1: Initialize

- Set head to NULL.

Step 2: Menu-Driven Operations

- Display the menu with the following options:

1. Create a new node and add it to the list.
2. Insert a node at a specific position.
3. Delete a node from a specific position.
4. Traverse the list.
5. Search for an element in the list.
6. Exit the program.

Step 3: Create Operation

- Input: Data to be inserted.
- Create a new node with the input data.
- If head is NULL, make the new node the head.
- Otherwise, traverse to the end of the list and append the new node.

Step 4: Insert at Position Operation

- Input: Data to be inserted and the position.
- If the position is 1:
  - Create a new node with the input data.
  - Set the new node's next pointer to the current head.
  - Update head to point to the new node.
- Otherwise:
  - Traverse to the node at position position-1.
  - If the node at position position-1 exists:
    - Create a new node with the input data.
    - Set the new node's next pointer to point to the next node of position-1.
    - Update the next pointer of the node at position-1 to point to the new node.
  - If the node at position-1 does not exist, print "Position out of range."

Step 5: Delete at Position Operation

- Input: Position to delete.

- If the list is empty, print "List is empty."
- If the position is 1:
  - Update head to point to the next node.
  - Free the memory of the original head.
- Otherwise:
  - Traverse to the node at position position-1.
  - If the node exists:
    - Update the next pointer of the node at position-1 to skip the node at position.
    - Free the memory of the node at position.
  - If the node at position-1 does not exist, print "Position out of range."

Step 6: Traverse Operation

- If the list is empty, print "List is empty."
- Otherwise:
  - Traverse the list, printing the data of each node followed by " -> ".
  - After the last node, print "NULL".

Step 7: Search Operation

- Input: Element to search for.
- Traverse the list, checking each node's data.
- If the element is found, print the element and its position.
- If the element is not found, print "Element not found in the list."

Step 8: Exit

- Exit the program when the user chooses the exit option.

Example Flow:

1. The user chooses "Create" and enters 10. The list now contains 10 -> NULL.
2. The user chooses "Create" and enters 20. The list now contains 10 -> 20 -> NULL.
3. The user chooses "Insert at Position" with data 15 at position 2. The list now contains 10 -> 15 -> 20 -> NULL.
4. The user chooses "Delete at Position" with position 2. The list now contains 10 -> 20 -> NULL.
5. The user chooses "Search" and enters 20. The program outputs "Element 20 found at position 2."
6. The user chooses "Exit", and the program ends.

Program source code:

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

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

struct Node* head = NULL;

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

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

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

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

   struct Node* temp = head;
   for (int i = 1; i < position - 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 deleteAtPosition(int position) {
```

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

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

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

    if (temp->next == NULL) {
        printf("Position out of range\n");
        return;
    }

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

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

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

void search(int key) {
   struct Node* temp = head;
   int position = 1;

   while (temp != NULL) {
      if (temp->data == key) {
         printf("Element %d found at position %d\n", key, position);
         return;
      }
      temp = temp->next;
      position++;
```

```
    }
    printf("Element %d not found in the list\n", key);
}

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

    while (1) {
        printf("\nMenu:\n");
        printf("1. Create\n");
        printf("2. Insert at Position\n");
        printf("3. Delete at Position\n");
        printf("4. Traverse\n");
        printf("5. Search\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to create: ");
                scanf("%d", &data);
                create(data);
                break;
            case 2:
                printf("Enter data to insert at position: ");
```

```
            scanf("%d", &data);
            printf("Enter position: ");
            scanf("%d", &position);
            insertAtPosition(data, position);
            break;
        case 3:
            printf("Enter position to delete: ");
            scanf("%d", &position);
            deleteAtPosition(position);
            break;
        case 4:
            traverse();
            break;
        case 5:
            printf("Enter element to search: ");
            scanf("%d", &data);
            search(data);
            break;
        case 6:
            exit(0);
        default:
            printf("Invalid choice\n");
    }
}

return 0;
```

}

Output Screenshots:

```
Menu:
1. Create
2. Insert at Position
3. Delete at Position
4. Traverse
5. Search
6. Exit
Enter your choice: 1
Enter data to create: 1

Menu:
1. Create
2. Insert at Position
3. Delete at Position
4. Traverse
5. Search
6. Exit
Enter your choice: 1
Enter data to create: 2

Menu:
1. Create
2. Insert at Position
3. Delete at Position
4. Traverse
5. Search
6. Exit
Enter your choice: 1
Enter data to create: 3

Menu:
1. Create
2. Insert at Position
3. Delete at Position
4. Traverse
5. Search
6. Exit
```

```
Enter your choice: 2
Enter data to insert at position: 3
Enter position: 4

Menu:
1. Create
2. Insert at Position
3. Delete at Position
4. Traverse
5. Search
6. Exit
Enter your choice: 4
1 -> 2 -> 3 -> 3 -> NULL

Menu:
1. Create
2. Insert at Position
3. Delete at Position
4. Traverse
5. Search
6. Exit
Enter your choice: 3
Enter position to delete: 4

Menu:
1. Create
2. Insert at Position
3. Delete at Position
4. Traverse
5. Search
6. Exit
Enter your choice: 4
1 -> 2 -> 3 -> NULL
```

```
Menu:
1. Create
2. Insert at Position
3. Delete at Position
4. Traverse
5. Search
6. Exit
Enter your choice: 5
Enter element to search: 1
Element 1 found at position 1

Menu:
1. Create
2. Insert at Position
3. Delete at Position
4. Traverse
5. Search
6. Exit
Enter your choice: 6
```

Conclusion:-

Learnt various things about linked list

Post lab questions:

1. Write the differences between linked list and linear array
2. Name some applications which uses linked list.

Differences Between Linked List and Linear Array

1. *Structure*:

   - *Linked List*: Composed of nodes, where each node contains a data element and a reference (or link) to the next node.

   - *Linear Array*: A contiguous block of memory where elements are stored sequentially.

2. *Size*:

   - *Linked List*: Dynamic in size. It can grow or shrink in size during runtime as nodes are added or removed.

   - *Linear Array*: Fixed in size. The size of the array is determined at the time of declaration and cannot be changed.

3. *Memory Allocation*:

   - *Linked List*: Memory is allocated dynamically, with each node being allocated individually in the heap.

   - *Linear Array*: Memory is allocated in a contiguous block, typically in the stack or heap.

4. *Access Time*:

   - *Linked List*: Access time is linear (O(n)) since you need to traverse the list from the beginning to access a specific element.

   - *Linear Array*: Access time is constant (O(1)) because elements can be accessed directly using an index.

5. *Insertion/Deletion*:

- *Linked List*: Easier and faster for insertion and deletion of elements, especially at the beginning or middle (O(1) for insertion/deletion if the node reference is known).

- *Linear Array*: Insertion and deletion are slower (O(n)) because elements need to be shifted to maintain the array structure.

6. *Memory Usage*:

- *Linked List*: Requires extra memory for storing pointers (references) to the next node, so it has additional memory overhead.

- *Linear Array*: No extra memory is needed beyond the space required for the elements themselves.

Applications of Linked List

1. *Dynamic Memory Allocation*: Linked lists are often used in memory management systems where memory allocation and deallocation are frequent, such as in operating systems.

2. *Implementation of Stacks and Queues*: Linked lists are used to implement stacks and queues efficiently, especially when the size of the stack or queue is not known in advance.

3. *Graph Representations*: Linked lists are used to represent graphs through adjacency lists, making it easier to manage sparse graphs.

4. *Handling Collisions in Hash Tables*: Linked lists are commonly used in hash tables to handle collisions through chaining, where each slot in the hash table points to a linked list of entries.

5. *Undo Functionality in Applications*: Linked lists can be used to implement the undo functionality, where each change is stored as a node in a list.

6. *Music Playlist Applications*: Many music players use linked lists to manage playlists, allowing easy addition, deletion, and reordering of songs.