Data Structures : Algorithms and Applications(Lab)

EXPERIMENT No: 5

AIM: To implement and demonstrate various operations on a singly linked list, including insertion, deletion, traversal, searching, and sorting of elements in the list.

THEORY:

A linked list is a sequential data structure that stores the data in non-contiguous memory locations unlike array. In a linked list, each element is referred to as a node. Each node in the linked list stores data and a pointer to the next node present in the list.

There are various linked list operations that allow us to perform different actions on linked lists.

Here's a list of basic linked list operations:

Traversal - access each element of the linked list

Insertion - adds a new element to the linked list

Deletion - removes the existing elements

Search - find a node in the linked list

Sort - sort the nodes of the linked list

Advantages of Singly Linked List:

1) Dynamic Data Structure:

Linked List being a dynamic data structure can shrink and grow at the runtime by deallocating or allocating memory, so there is no need for an initial size in linked list.

2) No Memory Wastage:

As the size of a linked list can grow or shrink at runtime, there is no memory wastage. Only the required memory is allocated.

3) Implementation:

Some very helpful data structures like queues and stacks can be easily implemented using a Linked List.

4) Insertion and Deletion Operation:

In a Linked List, insertion and deletion operations are quite easy, as there is no need to shift every element after insertion or deletion. Only the address present in the pointers needs to be updated.

Disadvantages of Singly Linked List:
1)Reverse Traversal:
The reverse traversal is not possible in the Singly LinkedList because we don't have the memory address of the previous pointers. It is possible in the Doubly LinkedList, but again it consumes more memory as we need to store the memory address of the previous pointers.
2)Accessing an element:
We can not access any element of the LinkedList directly. We don't have direct access to every element of LinkedList. If we want to access the ith element of the LinkedList, then we need to traverse the LinkedList till the ith index.

CODE:

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

// Create a node
struct Node {
    int data;
    struct Node* next;
};

// Insert at the beginning
void insertAtBeginning(struct Node** head_ref, int new_data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

// Insert a node after a node
void insertAfter(struct Node* prev_node, int new_data) {
    if (prev_node == NULL) {
        printf("The given previous node cannot be NULL\n");
```

```
        return;
    }
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = prev_node->next;
    prev_node->next = new_node;
}

// Insert at the end
void insertAtEnd(struct Node** head_ref, int new_data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    struct Node* last = *head_ref;

    new_node->data = new_data;
    new_node->next = NULL;

    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }

    while (last->next != NULL)
        last = last->next;

    last->next = new_node;
}

// Delete at the beginning
void deleteAtBeginning(struct Node** head_ref) {
    if (*head_ref == NULL) {
        printf("List is empty, nothing to delete\n");
        return;
    }
    struct Node* temp = *head_ref;
    *head_ref = temp->next;
    free(temp);
}
```

```
// Delete at a specific position
void deleteAtPosition(struct Node** head_ref, int position) {
    if (*head_ref == NULL) {
        printf("List is empty, nothing to delete\n");
        return;
    }

    struct Node* temp = *head_ref;

    if (position == 0) {
        *head_ref = temp->next;
        free(temp);
        return;
    }

    for (int i = 0; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }

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

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

// Delete at the end
void deleteAtEnd(struct Node** head_ref) {
    if (*head_ref == NULL) {
        printf("List is empty, nothing to delete\n");
        return;
    }
```

```
    struct Node* temp = *head_ref;
    struct Node* prev = NULL;

    if (temp->next == NULL) {
        *head_ref = NULL;
        free(temp);
        return;
    }

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

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

// Search a node
int searchNode(struct Node** head_ref, int key) {
    struct Node* current = *head_ref;

    while (current != NULL) {
        if (current->data == key)
            return 1;
        current = current->next;
    }
    return 0;
}

// Sort the linked list
void sortLinkedList(struct Node** head_ref) {
    struct Node *current = *head_ref, *index = NULL;
    int temp;

    if (*head_ref == NULL) {
        return;
```

```
    } else {
        while (current != NULL) {
            index = current->next;
            while (index != NULL) {
                if (current->data > index->data) {
                    temp = current->data;
                    current->data = index->data;
                    index->data = temp;
                }
                index = index->next;
            }
            current = current->next;
        }
    }
}

// Print the linked list
void printList(struct Node* node) {
    while (node != NULL) {
        printf(" %d ", node->data);
        node = node->next;
    }
    printf("\n");
}

int main() {
    struct Node* head = NULL;
    int choice, data, position, key;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert at beginning\n");
        printf("2. Insert at end\n");
        printf("3. Insert after a specific position\n");
        printf("4. Delete at beginning\n");
        printf("5. Delete at specific position\n");
        printf("6. Delete at end\n");
```

```
printf("7. Search for a node\n");
printf("8. Sort the list\n");
printf("9. Print the list\n");
printf("10. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

struct Node* current = head;

switch (choice) {
    case 1:
        printf("Enter the value to insert at beginning: ");
        scanf("%d", &data);
        insertAtBeginning(&head, data);
        break;
    case 2:
        printf("Enter the value to insert at end: ");
        scanf("%d", &data);
        insertAtEnd(&head, data);
        break;
    case 3:
        printf("Enter the position after which to insert (0 for head): ");
        scanf("%d", &position);
        printf("Enter the value to insert: ");
        scanf("%d", &data);
        for (int i = 0; i < position && current != NULL; i++) {
            current = current->next;
        }
        if (current == NULL && position != 0) {
            printf("Invalid position\n");
        } else {
            insertAfter(current, data);
        }
        break;
    case 4:
        deleteAtBeginning(&head);
        printf("Deleted at beginning\n");
```

```
            break;
        case 5:
            printf("Enter the position to delete (0 for head): ");
            scanf("%d", &position);
            deleteAtPosition(&head, position);
            printf("Deleted at position %d\n", position);
            break;
        case 6:
            deleteAtEnd(&head);
            printf("Deleted at end\n");
            break;
        case 7:
            printf("Enter the value to search: ");
            scanf("%d", &key);
            if (searchNode(&head, key)) {
                printf("%d is found\n", key);
            } else {
                printf("%d is not found\n", key);
            }
            break;
        case 8:
            sortLinkedList(&head);
            printf("List sorted\n");
            break;
        case 9:
            printf("Linked list: ");
            printList(head);
            break;
        case 10:
            exit(0);
        default:
            printf("Invalid choice\n");
    }
  }

  return 0;
}
```

OUTPUT :

```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert after a specific position
4. Delete at beginning
5. Delete at specific position
6. Delete at end
7. Search for a node
8. Sort the list
9. Print the list
10. Exit
Enter your choice: 1
Enter the value to insert at beginning: 56
```

```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert after a specific position
4. Delete at beginning
5. Delete at specific position
6. Delete at end
7. Search for a node
8. Sort the list
9. Print the list
10. Exit
Enter your choice: 1
Enter the value to insert at beginning: 67
```

```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert after a specific position
4. Delete at beginning
5. Delete at specific position
6. Delete at end
7. Search for a node
8. Sort the list
9. Print the list
10. Exit
Enter your choice: 3
Enter the position after which to insert (0 for head): 1
Enter the value to insert: 98
```

```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert after a specific position
4. Delete at beginning
5. Delete at specific position
6. Delete at end
7. Search for a node
8. Sort the list
9. Print the list
10. Exit
Enter your choice: 1
Enter the value to insert at beginning: 87
```

```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert after a specific position
4. Delete at beginning
5. Delete at specific position
6. Delete at end
7. Search for a node
8. Sort the list
9. Print the list
10. Exit
Enter your choice: 1
Enter the value to insert at beginning: 59
```

```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert after a specific position
4. Delete at beginning
5. Delete at specific position
6. Delete at end
7. Search for a node
8. Sort the list
9. Print the list
10. Exit
Enter your choice: 4
Deleted at beginning
```

```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert after a specific position
4. Delete at beginning
5. Delete at specific position
6. Delete at end
7. Search for a node
8. Sort the list
9. Print the list
10. Exit
Enter your choice:
5
Enter the position to delete (0 for head): 2
Deleted at position 2
```

```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert after a specific position
4. Delete at beginning
5. Delete at specific position
6. Delete at end
7. Search for a node
8. Sort the list
9. Print the list
10. Exit
Enter your choice: 6
Deleted at end
```

```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert after a specific position
4. Delete at beginning
5. Delete at specific position
6. Delete at end
7. Search for a node
8. Sort the list
9. Print the list
10. Exit
Enter your choice: 9
Linked list:  87  67
```

```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert after a specific position
4. Delete at beginning
5. Delete at specific position
6. Delete at end
7. Search for a node
8. Sort the list
9. Print the list
10. Exit
Enter your choice: 8
List sorted
```

```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert after a specific position
4. Delete at beginning
5. Delete at specific position
6. Delete at end
7. Search for a node
8. Sort the list
9. Print the list
10. Exit
Enter your choice: 9
Linked list:  67  87
```

```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert after a specific position
4. Delete at beginning
5. Delete at specific position
6. Delete at end
7. Search for a node
8. Sort the list
9. Print the list
10. Exit
Enter your choice: 7
Enter the value to search: 87
87 is found
```

```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert after a specific position
4. Delete at beginning
5. Delete at specific position
6. Delete at end
7. Search for a node
8. Sort the list
9. Print the list
10. Exit
Enter your choice: 10


=== Code Execution Successful ===
```

CONCLUSION:

This practical demonstrated the implementation of a singly linked list with operations like insertion, deletion, traversal, searching, and sorting. It highlighted the dynamic nature of linked lists, allowing efficient memory usage without wastage. The advantages, such as easy insertion and deletion, were observed, along with limitations like the inability to traverse in reverse. Overall, the exercise provided a clear understanding of linked list operations and their efficient data management capabilities.