Aryan Nanda **221070003**

EXPERIMENT-4

Aim:

Implement the core functionalities of a linked list data structure, including:

- Taking input in the linked list.
- Reversing the linked list
- Displaying the contents of the linked list

Theory:

Linked lists are a fundamental data structure that is commonly used in computer science. They are dynamic data structures, meaning that nodes can be added and removed from the list without resizing the underlying memory. This makes linked lists particularly useful for applications where the size of the data is not known in advance or where the data needs to be frequently modified.It is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers

There are different types of linked list:

1. Singly Linked list
2. Circular Linked List
3. Doubly Linked List
4. Doubly Circular Linked List

In this experiment we are using a singly linked list.
Singly Linked List:A singly linked list is a linear data structure in which each element is connected only to its next element using a pointer.Singly linked list ensures unidirectional flow.The elements of singly linked list are accessed by traversing it from one end to other

Structure of a Linked List:

A linked list consists of a sequence of nodes, where each node contains two components:

1. **Data**: The data element stored in the node.
2. **Next** Pointer: A pointer to the next node in the sequence.

The first node in the list is called the `head` node, and the last node in the list is typically represented by a **NULL** pointer. The **start** pointer of the linked list points to the first node.

There are two ways to reverse a linked list.

1. **Reversing the links**

   The pointer-shifting method for reversing a linked list is an efficient and straightforward approach that involves manipulating the links between the nodes without copying or moving the actual data.It is considered efficient as it does not require shifting of data.

2. **Shifting Data:**

   The second approach involves copying the data from each node into a temporary array, reversing the order of the data in the array, and then creating a new linked list from the reversed data.This approach is generally avoided because it requires additional memory and is less efficient than the pointer-shifting method.

Feature	**Pointer Shifting**	**Data Shifting**
Memory Usage	Constant	Linear (proportional to the list length)
Efficiency	More efficient	Less efficient
Implementation	Simpler	More Complex

Algorithm:

Algorithm: Reverse Linked List
Class: Node
    Properties:
- data: Integer data stored in the node.

- next: Pointer to the next node.

Function: takeInput()

    Read data until -1 is encountered from the input.
    Create a new node for each data entry.
    If the list is empty, set the new node as the head and tail.
    Otherwise, link the new node to the tail and update the tail.
    Return the head of the linked list.

Function: print(head)

    Initialize a temporary pointer temp to the head.
    While temp is not NULL:
        - Print the data stored in temp.
        - Move temp to the next node.
    Print a new line after printing all elements.

Function: reverseLinkedListIterative(head)

    Initialize pointers pre, cur, and next to NULL, head, and head->next respectively.
    While next is not NULL:
        - Reverse the link between cur and pre.
        - Update pre, cur, and next to their respective next pointers.
    Update the last link and return the new head (pre).

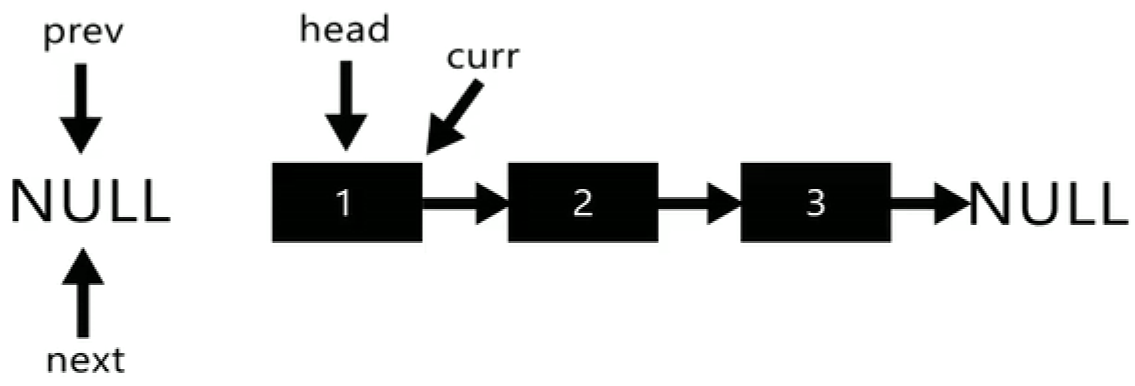
Function: reverseLinkedListRecursive(head)

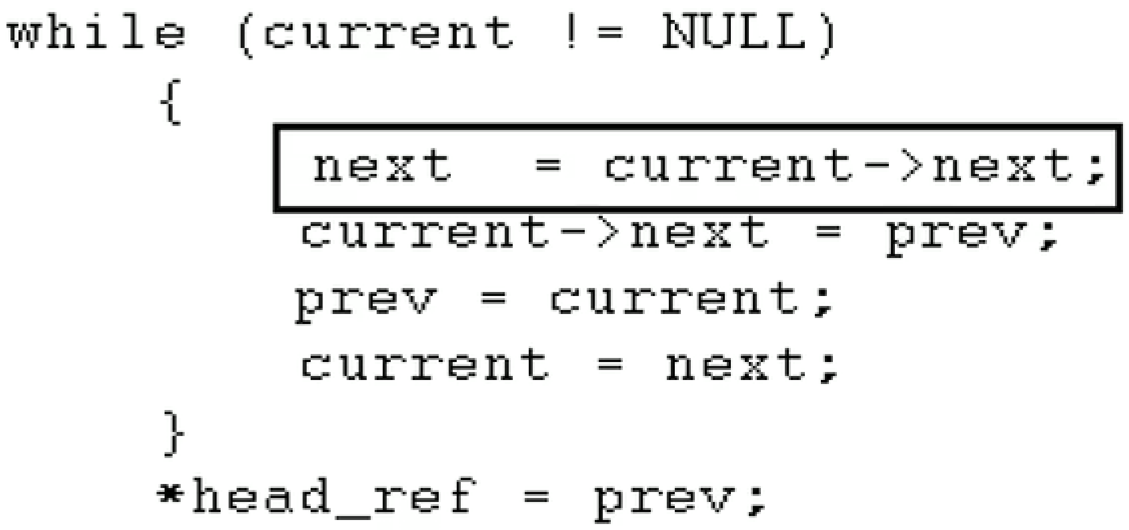
    If head is NULL or head->next is NULL, return head.
    Recursively call reverseLinkedListRecursive() for the next node and store its result in newHead.
    Link the next node's next pointer to the current head.
    Set the current head's next pointer to NULL.
    Return newHead as the new head of the reversed list.

Main Function:

    Take input to create a linked list.
    Reverse the linked list using reverseLinkedListRecursive() and assign it to head.
    Print the reversed list.
    Reverse the list again using reverseLinkedListIterative() and assign it to head.
    Print the final reversed list.

Example:





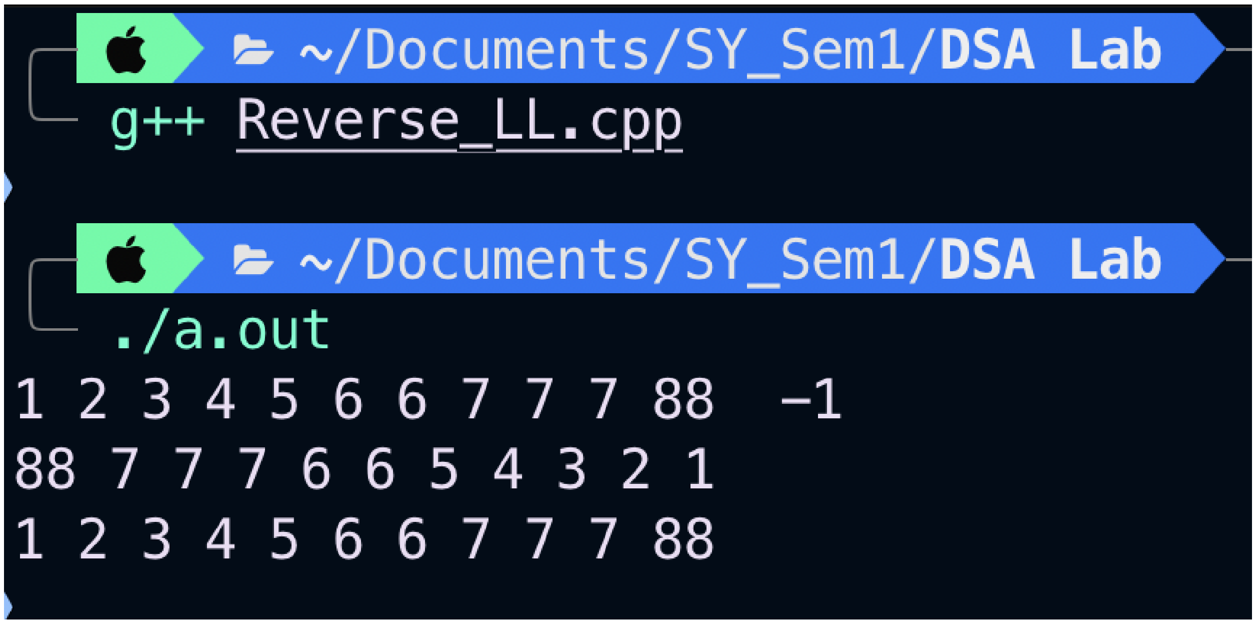
```
while (current != NULL)
      {
          next  = current->next;
          current->next = prev;
          prev = current;
          current = next;
      }
      *head_ref = prev;
```

Code:

```
#include <iostream>
using namespace std;
class Node
{
public:
   int data;
   Node *next;
   Node(int data)
   {
       this->data = data;
       this->next = NULL;
   }
};
Node *takeinput()
{
   int data;
   cin >> data;
   Node *head = NULL, *tail = NULL;
   while (data != -1)
   {
       Node *newnode = new Node(data);
       if (head == NULL)
       {
           head = newnode;
           tail = newnode;
       }
       else
       {
           tail->next = newnode;
           tail = newnode;
       }
       cin >> data;
   }
   return head;
}
void print(Node *head){
   Node* temp = head;
   while(temp!=NULL){
       cout<<temp->data<<"    ";
       temp = temp->next;
   }
   cout<<endl;
}
// -------------Iterative----------------
// -------------Iterative----------------
Node *reverseLinkedList(Node *head) {
   if(head==NULL){
       return head;
```

```
    }
    Node* pre=NULL;
    Node* cur=head;
    Node* next=head ->next;
    while(next!=NULL){
        cur ->next=pre;
        pre=cur;
        cur=next;
        next=cur ->next;
    }
    cur ->next=pre;
    pre=cur;
    return pre;
}
// -------------Recursive----------------
Node *reverseLinkedListRec(Node *head)
{
    if(head==NULL || head ->next==NULL){
        return head;
    }
    Node* newhead=reverseLinkedListRec(head ->next);
    Node* tail=head ->next;
    tail ->next=head;
    head ->next=NULL;
    return newhead;
}
int main()
{
    Node *head = takeinput();
    head = reverseLinkedListRec(head);
    print(head);
    head = reverseLinkedList(head);
    print(head);
    return 0;
}
```

Input/Output:



```
 ~/Documents/SY_Sem1/DSA Lab
g++ Reverse_LL.cpp

 ~/Documents/SY_Sem1/DSA Lab
./a.out
1 2 3 4 5 6 6 7 7 7 88  -1
88 7 7 7 6 6 5 4 3 2 1
1 2 3 4 5 6 6 7 7 7 88
```

Conclusion:

We learned how to implement the core functionalities of a linked list data structure, allowing us to insert and reverse nodes, as well as display the contents of the list.