Name	Sujal Dingankar
UID no.	DSE24100720
Experiment No.	3

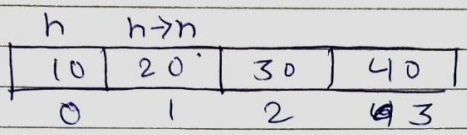
AIM:	Swap the nodes in pairs from linked list.
THEORY:	A linked list is a linear data structure where elements, known as nodes, are not stored in contiguous memory locations. Instead, each node contains two parts: the data and a reference (or link) to the next node in the sequence. This allows dynamic memory allocation and efficient insertion and deletion operations. We use linked lists because they provide dynamic memory allocation, allowing efficient insertion and deletion of elements without requiring contiguous memory. Unlike arrays, linked lists can grow or shrink in size as needed, making them more flexible. They are ideal for scenarios where frequent additions and deletions occur, such as implementing stacks, queues, or managing polynomial arithmetic. Linked lists also eliminate the need for resizing, avoiding the overhead of shifting elements, and are particularly useful in memory-constrained environments. **Types of Linked Lists:** 1. **Singly Linked List**: Each node points to the next node in the list, and the last node points to null. 2. **Doubly Linked List**: Each node contains two references, one to the next node and another to the previous node. 3. **Circular Linked List**: The last node's next reference points back to the first node, forming a loop. **Operations on Linked List:** 1. **Insertion Operation**: Adds a new node at the beginning, end, or at a specified position in the list. The references between nodes are adjusted accordingly. o **At Beginning**: The new node becomes the head of the list, and its next points to the old head. o **At End**: The new node's next is set to null, and the last node's next points to the new node. o **At Specific Position**: The new node is placed at the specified position, with references of adjacent nodes updated.

	2. **Deletion Operation**: Removes a node from the beginning, end, or a specific position in the list. References between nodes are adjusted to bypass the deleted node.
	  - **From Beginning**: The head node is removed, and the second node becomes the new head.
	  - **From End**: The last node is removed, and the second-last node's next is set to null.
	  - **From Specific Position**: The node at the specified position is removed, and references of adjacent nodes are updated to bypass it.
	3. **Traversal Operation**: Iterates through the linked list to access each node starting from the head node until the end (null) is reached.
	4. **Search Operation**: Searches for a specific element in the linked list by traversing node by node.
	5. **Reverse Operation**: Reverses the order of nodes in the linked list by adjusting the references of the nodes so that the first node becomes the last and vice versa.
	
	Applications of Linked List:
	1. **Dynamic Memory Allocation**: Linked lists efficiently handle dynamic memory allocation as they do not require pre-allocated memory.
	2. **Implementing Stacks and Queues**: Linked lists can be used to implement stack (LIFO) and queue (FIFO) data structures.
	3. **Graph Representation**: Adjacency lists, which represent graphs, are often implemented using linked lists.
	4. **Handling Polynomial Arithmetic**: Linked lists can represent polynomials, with each node storing a term (coefficient and exponent).
	5. **Memory-Efficient Storage**: Useful in scenarios where elements need to be frequently inserted or removed, as the memory overhead for rearranging elements is low.

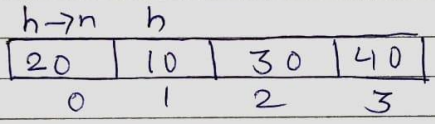
ALGORITHM:	**1. Input the number of nodes and data:** **o Ask the user for the number of nodes in the linked list.** **o For each node, read its data and create a new node.** **o Link each node to form the list.** **2. Initialize the linked list:** **o Set the head pointer to the first node.** **o Link all the subsequent nodes by setting their next pointers.** **3. Print the linked list before swapping:** **o Traverse the linked list from the head and print each node's data.** **4. Define a swap function:** **o Create a function to swap the data of two nodes (using pointers).** **5. Traverse the list to swap adjacent nodes:** **o Start from the head of the list.** **o While traversing the list, for each adjacent pair of nodes:** **▪ Swap the data of the current node and the next node.** **▪ Move the pointer two steps forward (to the node after the swapped pair).** **6. Edge case handling:** **o Ensure that the swap operation stops if there is no adjacent node left (i.e., in case of an odd number of nodes, the last node remains unswapped).** **7. Print the linked list after swapping:** **o Traverse the modified linked list and print the updated data of each node.** **8. End of program:** **o The list has now been modified with all adjacent pairs of nodes swapped. Exit the program.**

PROBLEM SOLVING:

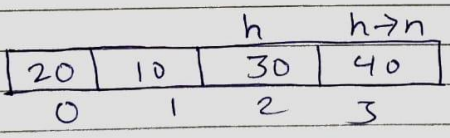
Initially, head = 0



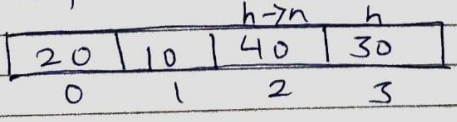
1st iteration :-



Before 2nd Iteration we jump head by 2



Swap h with h→n,

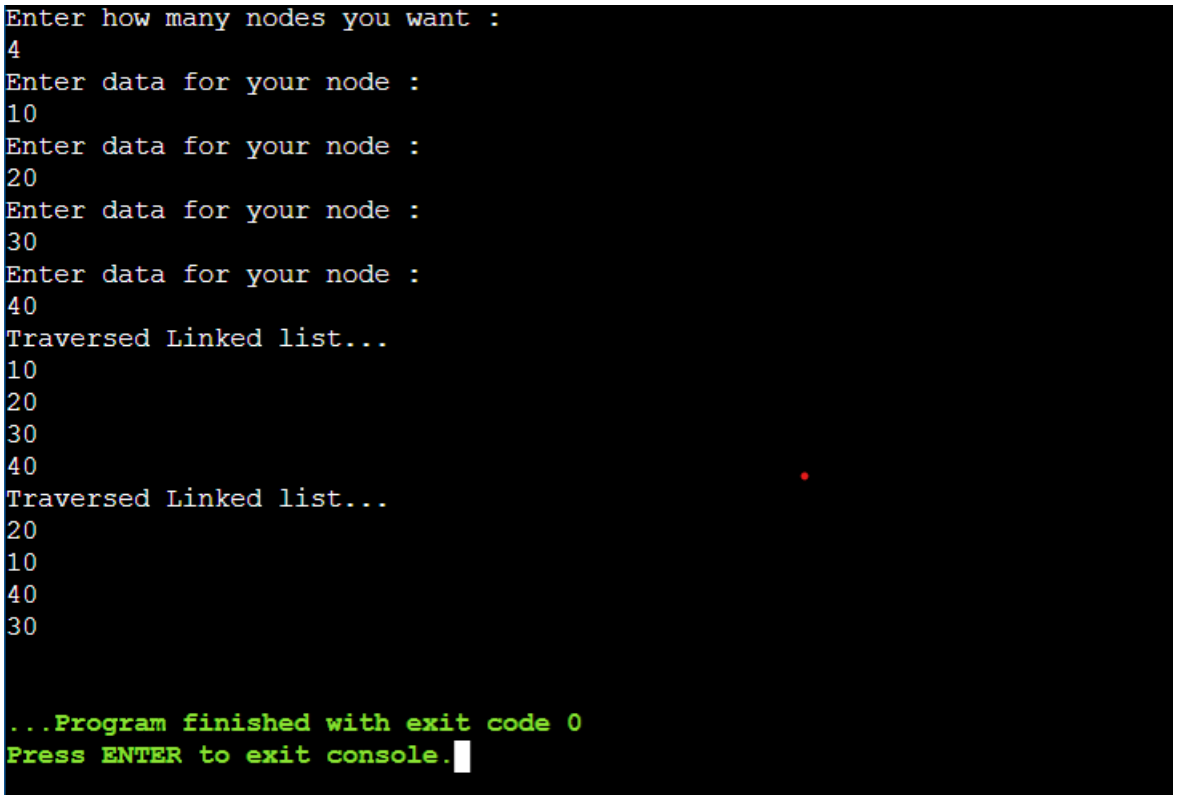


PROGRAM :-

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

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

void printLinkedList(struct Node *head){
    printf("Traversed Linked list...\n");
    while(head != NULL){
        printf("%d\n",head->data);
        head = head->next;
    }
};

void swap(int *a,int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
};

void adjacentPairSwap(struct Node* head){
    struct Node *temp = head;
    while (temp!=NULL && temp->next !=NULL)
    {
        swap(&temp->data,&temp->next->data);
        temp = temp->next->next;
    }

}

int main(){
    int noOfNodes,value;
    struct Node *head = NULL;
    struct Node *newNode = NULL;
    struct Node *temp = NULL;


    printf("Enter how many nodes you want : \n");
    scanf("%d",&noOfNodes);

    for(int i=0;i<noOfNodes;i++){
        newNode = (struct Node*)malloc(sizeof(struct Node));

        printf("Enter data for your node :\n");
        scanf("%d",&value);

        newNode->data = value;
        newNode->next = NULL;

        if (head == NULL)
        {
            head = newNode;
        }else{
            temp->next = newNode;
        }

        temp = newNode;

    }
```

```
    printLinkedList(head);
    adjacentPairSwap(head);
    printLinkedList(head);

    return 0;
}
```

RESULT :-



CONCLUSION : From this experiment with linked lists and adjacent pair swapping, I gained practical experience in manipulating linked lists and understanding their node-based structure. By implementing functions to create, traverse, and modify linked lists, including swapping adjacent pairs, I learned how to manage pointers and dynamically allocated memory effectively. The use of a swap function to interchange node values highlighted the importance of pointer manipulation and linked list traversal techniques. This exercise improved my ability to manage linked list operations and will be valuable for solving problems involving dynamic data structures.