Name	Shubhan Singh
UID no.	2022300118
Experiment No.	3

AIM:	Implementing and using singly lined list for problem solving
Program 1	
PROBLEM STATEMENT :	Implement a LinkedList ADT, and complete all the operations listed int the 'linked_list.c' file.
ALGORITHM:	1. Create a Linked List Node ->Allocate memory for a new node, set its data, and initialize its next pointer to NULL. ->Return the new node. 2. Insert at Position - >If the position is 0 or the list is empty: - >Create a new node with the given data. - >Set the new node's next pointer to the current head. - >Update the head to point to the new node. - >If the position is -1 (end): - >Traverse the list to find the last node. - >Create a new node with the given data and set its next pointer to NULL. - >Update the last node's next pointer to the new node. - >For other positions: - >Traverse the list to find the node at position (pos-1). - >Create a new node with the given data. - >Set the new node's next pointer to the current node's next. - >Update the current node's next pointer to point to the new node. 3. Delete at Position - >If the position is 0: - >Update the head to point to the second node.

- >Free the memory of the original head node.
- >If the position is -1 (end):
- >Traverse the list to find the second-to-last node.
- >Free the memory of the last node and set the second-to-last node's next pointer to NULL.
- >For other positions:
- >Traverse the list to find the node at position (pos-1).
- >Update the previous node's next pointer to skip the node at the specified position.
- >Free the memory of the removed node.

4. Delete by Value

- >If the list is empty or the value is not found in the head node:
- >Print "Element not found."
- >If the value is found in the head node:
- >Update the head to point to the node after the next node.
- >Free the memory of the original head node.
- >For other positions:
- >Traverse the list to find the node with the specified value.
- >Update the previous node's next pointer to skip the node with the specified value.
- >Free the memory of the removed node.

5. Get Node at Position

- >Traverse the list to find the node at the specified position.
- >If the position is out of range, print "Index not found" and return NULL.

6. Find First Occurrence of Value

- >Traverse the list to find the first node with the specified value.
- >If the value is not found, print "Element not found" and return NULL.

7. Display Linked List

	- >Traverse the list from the head and print each element. 8. Free Linked List - >Traverse the list, freeing the memory for each node. 9. Reverse Linked List - >Traverse the list while reversing the direction of next pointers. - >Return the new head of the reversed list.
PROGRAM:	

```c
// Create a Linked List ADT using the Struct 'Node'.
// The Linked List should support all operations that are listed
as functions in this file

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

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


// create a LinkedList node with 'data'
Node* create_node(int data){
    Node* genesis_node=malloc(sizeof(Node));
    genesis_node->data=data;
    genesis_node->next=NULL;
    return genesis_node;
}

// pos=-1 indicates insert at end
// pos=0 indicates add at beginning
// This creates a new LinkedList node and inserts it at position
```

```
'pos' in the current linked list originating from head
void insert_at_pos(Node **head, int pos, int data){
    Node* iterator=*(head);
    Node* temp;
    Node* prevnode=NULL;
    if(pos<0){pos=INT_MAX;}
    for(int i=0;i<pos;i++){
        prevnode=iterator;
        temp=iterator->next;
        if(temp==NULL){
            pos=-1;
            break;
        }
        iterator=temp;
    }
    if(pos<0){
        iterator->next=malloc(sizeof(Node));
        iterator=iterator->next;
        iterator->data=data;
        iterator->next=NULL;
    }
    else if(pos==0){
        Node* newhead=malloc(sizeof(Node));
        newhead->data=data;
        newhead->next=iterator;
        head=&newhead;
    }
    else{
        Node* newnode=malloc(sizeof(Node));
        newnode->data=data;
        newnode->next=iterator;
        prevnode->next=newnode;
    }
}
```

```
// pos=-1 indicates delete last node
// pos=0 indicates delete first node
// This deletes the LinkedList node at position 'pos' in the
current linked list originating from head
void delete_at_pos(Node **head, int pos){
    Node* iterator=*(head);
    if(iterator->next==NULL){
        printf("Cannot delete only element in the list\n");
        return;
    }
    Node* temp;
    Node* prevnode=NULL;
    if(pos<0){pos=INT_MAX;}
    for(int i=0;i<pos;i++){
        temp=iterator->next;
        if(temp==NULL){
            pos=-1;
            break;
        }
        prevnode=iterator;
        iterator=temp;
    }
    if(pos<0){
        free(iterator);
        prevnode->next=NULL;
    }
    else if(pos==0){
        *head=iterator->next;
        free(iterator);
    }
    else{
        prevnode->next=iterator->next;
        free(iterator);
    }
}
```

```
// delete linkedlist node with first occurrence of given value
from linked list originating from 'head'
void delete_by_value(Node **head, int value){
    Node* prevnode=*(head);
    if(prevnode->data==value){
        head=&(prevnode->next);
        free(prevnode);
        return;
    }
    while(1){
        if(prevnode->next==NULL){
            printf("This value does not exist in the list\n");
            return;
        }
        if(prevnode->next->data==value){
            Node* temp=prevnode->next;
            prevnode->next=temp->next;
            free(temp);
            return;
        }
        prevnode=prevnode->next;
    }
}


// gets the node at position 'pos' in linkedlist originating fro
'head'
// return 'null' if no node found along with informative message
Node* get_node_at_pos(Node **head, int pos){//assuming 0 based
indexing
    int isnull=0;;
    Node* iterator=*(head);
    for(int i=0;i<pos;i++){
        if(iterator->next!=NULL)
```

```
        iterator=iterator->next;
        else{
            isnull=1;
            break;
        }
    }
    if(!isnull){
        return iterator;
    }
    else{
        printf("This index doesn't exist in the list\n");
        return NULL;
    }
}

// Return the node with the first occurrence of value
// return 'null' if no node found along with informative message
Node* find_first(Node **head, int value){
    Node* iterator=*(head);
    while(iterator->data!=value){
        iterator=iterator->next;
        if(iterator==NULL){
            printf("This value doesn'e exist in the list\n");
            return NULL;
        }
    }
    return iterator;
}

// display entire linked list, starting from head, in a well-
formatted way
void display(Node *head){
    Node* iterator=head;
    while(iterator!=NULL){
        printf("%d ",iterator->data);
```

```
        iterator=iterator->next;
    }
    printf("\n");
}


// deallocate the memory being used by the entire linkedlist,
starting from head
void free_linkedlist(Node *head){
    Node* current=head;
    Node* next=current->next;
    while(next!=NULL){
        free(current);
        current=next;
        next=current->next;
    }
    free(current);
}


// reverse a linkedlist - reverse a given linked list
Node* reverse(Node *head){
    Node* next=head->next;
    if(next==NULL){
        return head;
    }
    head->next=NULL;
    Node* current=head;
    Node* twoahead=next->next;
    while(twoahead!=NULL){
        next->next=current;
        current=next;
        next=twoahead;
        twoahead=next->next;
    }
```

```
    next->next=current;
    head=next;
    return head;
}

int main(){
    printf("Enter data for head of list\n");
    int data,pos;
    scanf("%d",&data);
    Node* head=create_node(data);
    int choose=1;
    while(choose){
        printf("Enter\n1.To insert into list\n2.To delete by
position\n3.To delete by value\n4.To get node at pos\n5.Find
first occurence of value\n6.Display list\n7.Reverse list\n8.Free
list\nanything else To exit\n");
        scanf("%d",&choose);
        switch(choose){
            case 1:
            printf("Enter value to insert into list and position
to insert it at(-1 for inserting at end)\n");
            scanf("%d%d",&data,&pos);
            insert_at_pos(&head,pos,data);
            break;
            case 2:
            printf("Enter position to delete(-1 for end)\n");
            scanf("%d",&pos);
            delete_at_pos(&head,pos);
            break;
            case 3:
            printf("Enter value to delete from list(only first
occurence will be deleted)\n");
            scanf("%d",&data);
            delete_by_value(&head,data);
            break;
```

```
            case 4:
            printf("Enter position to fetch node from\n");
            scanf("%d",&pos);
            Node* temp=get_node_at_pos(&head,pos);
            if(temp!=NULL)
            printf("The fetched node had data: %d\n",temp->data)
            break;
            case 5:
            printf("Enter a value to get node with the first
occurence of that value\n");
            scanf("%d",&data);
            Node* templ= find_first(&head,data);
            if(templ!=NULL)
            printf("The fetched node had data: %d\n",templ-
>data);
            break;
            case 6:
            printf("The list from head to tail is:\n");
            display(head);
            break;
            case 7:
            head=reverse(head);
            printf("The reversed list is:\n");
            display(head);
            break;
            case 8:
            free_linkedlist(head);
            choose=0;
            break;
            default:
            choose=0;
            break;
        }
    }
    free_linkedlist(head);
```

```
}
```

RESULT:

```
Enter data for head of list
1
Enter
1.To insert into list
2.To delete by position
3.To delete by value
4.To get node at pos
5.Find first occurence of value
6.Display list
7.Reverse list
8.Free list
anything else To exit
1
Enter value to insert into list and position to insert it at(-1 for inserting at end)
2 -1
Enter
1.To insert into list
2.To delete by position
3.To delete by value
4.To get node at pos
5.Find first occurence of value
6.Display list
7.Reverse list
8.Free list
anything else To exit
1
Enter value to insert into list and position to insert it at(-1 for inserting at end)
3 -1
Enter
1.To insert into list
2.To delete by position
3.To delete by value
4.To get node at pos
5.Find first occurence of value
6.Display list
7.Reverse list
8.Free list
anything else To exit
6
The list from head to tail is:
1 2 3
```

```
Enter value to insert into list and position to insert it at(-1 for inserting at end)
4 1
Enter
1.To insert into list
2.To delete by position
3.To delete by value
4.To get node at pos
5.Find first occurence of value
6.Display list
7.Reverse list
8.Free list
anything else To exit
6
The list from head to tail is:
1 4 2 3
Enter
1.To insert into list
2.To delete by position
3.To delete by value
4.To get node at pos
5.Find first occurence of value
6.Display list
7.Reverse list
8.Free list
anything else To exit
2
Enter position to delete(-1 for end)
2
Enter
1.To insert into list
2.To delete by position
3.To delete by value
4.To get node at pos
5.Find first occurence of value
6.Display list
7.Reverse list
8.Free list
anything else To exit
6
The list from head to tail is:
1 4 3
```

```
The list from head to tail is:
1 4 3
Enter
1.To insert into list
2.To delete by position
3.To delete by value
4.To get node at pos
5.Find first occurence of value
6.Display list
7.Reverse list
8.Free list
anything else To exit
3
Enter value to delete from list(only first occurence will be deleted)
4
Enter
1.To insert into list
2.To delete by position
3.To delete by value
4.To get node at pos
5.Find first occurence of value
6.Display list
7.Reverse list
8.Free list
anything else To exit
6
The list from head to tail is:
1 3
Enter
1.To insert into list
2.To delete by position
3.To delete by value
4.To get node at pos
5.Find first occurence of value
6.Display list
7.Reverse list
8.Free list
anything else To exit
4
Enter position to fetch node from
1
The fetched node had data: 3
```

```
1
Enter value to insert into list and position to insert it at(-1 for inserting at end)
6 -1
Enter
1.To insert into list
2.To delete by position
3.To delete by value
4.To get node at pos
5.Find first occurence of value
6.Display list
7.Reverse list
8.Free list
anything else To exit
1
Enter value to insert into list and position to insert it at(-1 for inserting at end)
7 -1
Enter
1.To insert into list
2.To delete by position
3.To delete by value
4.To get node at pos
5.Find first occurence of value
6.Display list
7.Reverse list
8.Free list
anything else To exit
6
The list from head to tail is:
1 3 6 7
Enter
1.To insert into list
2.To delete by position
3.To delete by value
4.To get node at pos
5.Find first occurence of value
6.Display list
7.Reverse list
8.Free list
anything else To exit
7
The reversed list is:
7 6 3 1
Enter
1.To insert into list
2.To delete by position
3.To delete by value
4.To get node at pos
5.Find first occurence of value
6.Display list
7.Reverse list
8.Free list
anything else To exit
8
PS C:\Users\shubh\OneDrive - Bhar
```

Program 2	
PROBLEM STATEMENT :	swap nodes in pairs- Given a linked list, swap every two adjacent nodes and return the head of the new linked list.

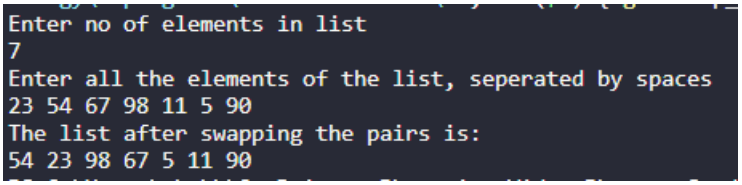
	You must solve the problem without modifying the values in the list's nodes.
ALGORITHM:	->Initialize head_temp as a pointer to the head of the linked list. ->Initialize pos and value to keep track of the current position and temporary value. ->Check if the linked list contains only one node (i.e., head->next == NULL). If so, no swaps ar necessary, and the function returns the original head. ->Enter a loop to process the linked list nodes in pairs: ->Get the node at the current position pos using the get_node_at_pos function. ->If the get_node_at_pos function returns NULL, it means we have reached the end of the list, so exit the loop. ->Check if the next node (temp->next) is NULL. If it is, this means there's only one node left, so exit the loop. ->Extract the data value (value) from the current node (temp). ->Delete the current node at position pos using the delete_at_pos function, effectively removing it from the list. ->Increment pos by 1 to move to the next position. ->Insert the extracted value at the new position (pos) in the linked list using the insert_at_pos function. This effectively swaps the current node and the next node. ->Increment pos by 1 to move to the position after the swapped pair. ->Once the loop completes, return the head_temp pointer, which now points to the head of the modified linked list with adjacent nodes swapped.
Test Cases:	Example 1: input: 1 -> 2 -> 3 -> 4 output: 2 -> 1 -> 4 -> 3 Example 2: input: 1 -> 2 -> 3 output: 2 -> 1 -> 3 Example 3:

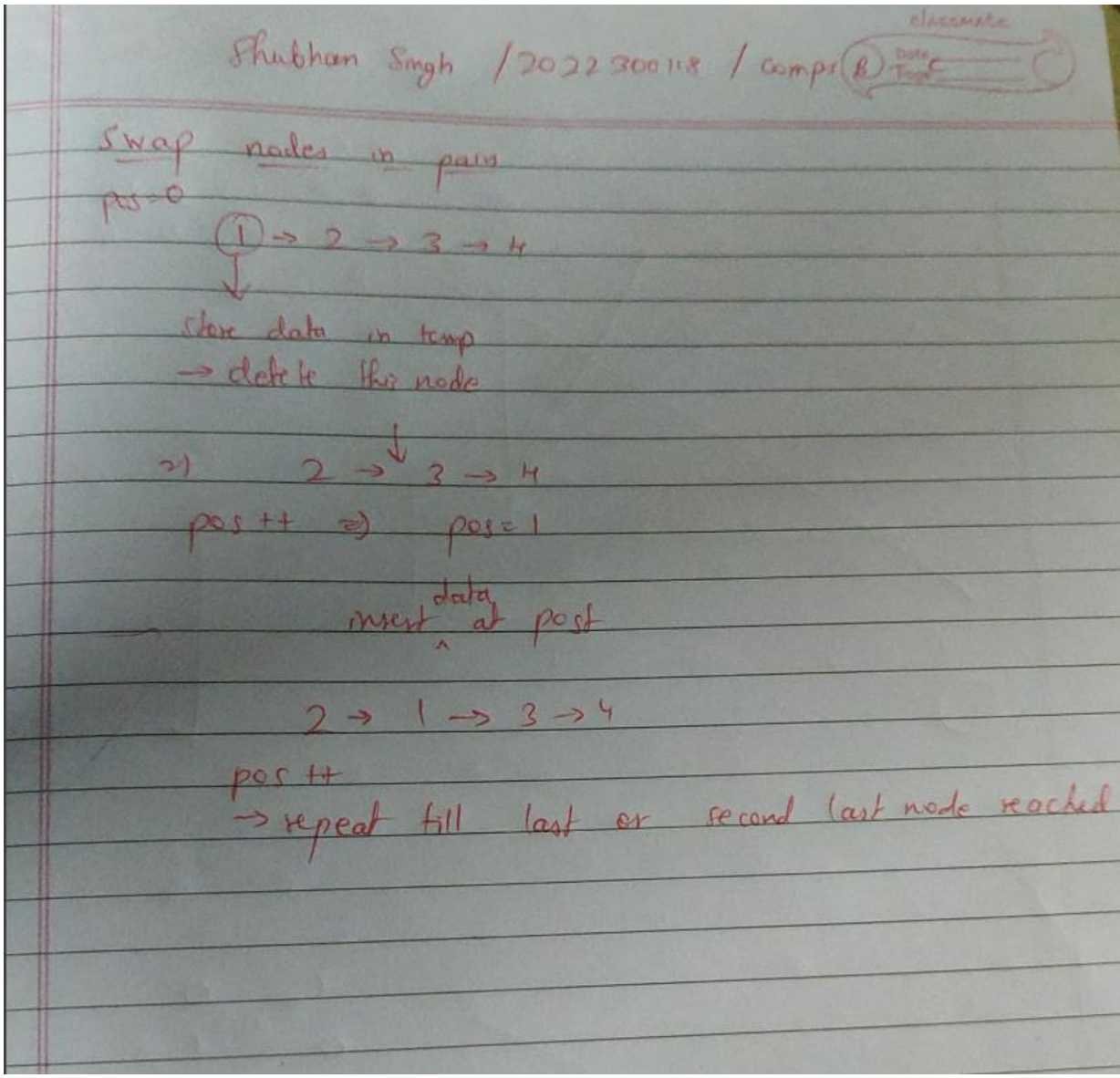
	input: 1 output: 1
PROGRAM:	

```c
#include "linked_list.c"


// This function should take a head node of a linked list, swap nodes in pairs, and return the new head node
Node* swap_pairs(Node* head){
    int pos=0;
    Node* temp;
    int tem;
    while(1){
        temp=get_node_at_pos(&head,pos);
        if(temp==NULL || temp->next==NULL){
            break;
        }
        tem=temp->data;
        delete_at_pos(&head,pos);
        pos++;
        insert_at_pos(&head,pos,tem);
        pos++;
    }
    return head;
}

int main(){
    int n,dat;
    printf("Enter no of elements in list\n");
    scanf("%d",&n);
    printf("Enter all the elements of the list, seperated by spaces\n");
    scanf("%d",&dat);
    Node* list=create_node(dat);
    for(int i=1;i<n;i++){
```

```
        scanf("%d",&dat);
        insert_at_pos(&list,-1,dat);
    }
    list=swap_pairs(list);
    printf("The list after swapping the pairs is:\n");
    display(list);
    free_linkedlist(list);
    return 0;
}
```

RESULT:



Rough work:	
Theory:	Linked List: Linked lists are fundamental data structures used in computer science and programming for organizing and storing a collection of elements. There are several types of linked lists, including singly linked lists, doubly linked lists, and circular linked lists. **Singly Linked List :** A singly linked list is a linear data structure in which elements, called nodes, are connected via pointers. Each node contains two part stores the actual data or value associated with the node an a reference (usually a memory address) to the next node in the sequence.The last node in the list typically has a null reference as its next pointer, indicating the end of the list. 1. **Head**: The first node of the list is called the head. It serves as the starting point for traversing the list.

	2. **Traversal**: To access elements in a singly linked list, you start at the head and follow the next pointers from one node to the next until you reach the desired node. 3. **Insertion**: You can insert a new node at various positions in a singly linked list, such as at the beginning (insertion at the head), at the end, or at a specific position within the list. To do this, you update the next pointers of the relevant nodes. **Deletion**: Nodes can be removed from a singly linked list by updating the next pointers of the preceding nodes to bypass the node you want to delete. Once no references exist to a node, it becomes eligible for garbage collection in languages.
Coclusion:	• Dynamic Size: Linked lists can grow or shrink as needed during runtime. • Efficient Insertions and Deletions: Insertions and deletions at the beginning are faster in linked lists (O(1)). • Used in implementing dynamic data structures like stacks and queues. • Memory management in operating systems and lower-level programming languages. • Symbol tables in compilers and interpreters. • Undo functionality in software applications. • Representing polynomial expressions in algebraic computations.