# **CS8381 – DATA STRUCTURES LABORATORY CYCLE I**

LINEAR DATA STRUCTURES

|  |  |
| --- | --- |
| Ex. No. | 1.1.2.2 |
| NAME OF THE EXERCISE | APPLICATIONS OF LIST ADT - REVERSING A LIST |
| DATE OF EXERCISE | DD.MM.YYYY |
| DATE OF SUBMISSION | 08.12.2020 |
| STUDENT REG. NO. | 920419205025 |
| STUDENT ROLL NO. | 19UITE012 |
| STUDENT NAME | B.Naga sapta aakash |
| FACULTY IN- CHARGE | Dr. P. SUBATHRA, Prof. / IT Dr. E. VAKAIMALAR, AP/IT **Mr. D. VENDHAN, AP/IT** |



## AIM:

To reverse the element of a given linked list using the following methods:

* Non recursive or Iterative Method

* Recursive

## DESCRIPTION:

Given pointer to the head node of a linked list, the task is to reverse the linked list. We need to reverse the list by changing links between nodes.

## Examples:

**Input**: Head of following linked list 1->2->3->4->NULL

**Output**: Linked list should be changed to, 4->3->2->1->NULL

**Input**: Head of following linked list 1->2->3->4->5->NULL

**Output**: Linked list should be changed to, 5->4->3->2->1->NULL

**Input**: NULL

**Output**: NULL

**Input**: 1->NULL

**Output**: 1->NULL

The above can operations can be done using either an iterative or a recursive method.

## -0ALGORITHM

**Iterative Method**

1. Initialize three pointers prev as NULL, curr as head and next as NULL.
2. Iterate trough the linked list. In loop, do following.

// Before changing next of current, store next node next = curr->next

// Now change next of current.This is where actual reversing happens curr->next = prev

// Move prev and curr one step forward prev = curr

curr = next

## Recursive Method:

1. Divide the list in two parts - first node and rest of the linked list.
2. Call reverse for the rest of the linked list.
3. Link rest to first.
4. Fix head pointer

|  |
| --- |
|  |
|  |  |

## ONLINE CODE LINK

[**https://onlinegdb.com/B1TNaSVKw**](https://onlinegdb.com/B1TNaSVKw)

**SOURCE CODE**

//Name :B.NAGA SAPTA AAKASH Roll no: 19UITE012

//Reverse the Linked List.

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

struct node { int data;

struct node \*next;

}\*head;

void createList(int n); void reverseList(); void displayList();

int main()

{

int n, choice;

printf("\n\t\tReverse the Linked List"); printf("\n\t\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"); printf("\nEnter the total number of nodes: "); scanf("%d", &n);

createList(n);

printf("\nData in the Linked list :"); displayList();

reverseList();

printf("\nData in the Linked list after Reverse"); displayList();

return 0;

}

void createList(int n)

{

struct node \*newNode, \*temp; int data, i;

if(n <= 0)

{

printf("\nList size must be greater than zero ..."); return;

}

head = (struct node \*)malloc(sizeof(struct node));

if(head == NULL)

{

printf("\n<-- Unable to allocate memory -->");

}

else

{

printf("\nEnter the data of the NODE: "); scanf("%d", &data);

head->data = data; head->next = NULL;

temp = head; for(i=2; i<=n; i++)

{

newNode = (struct node \*)malloc(sizeof(struct node));

if(newNode == NULL)

{

printf("\nUnable to allocate memory."); break;

}

else

{

printf("\nEnter the data of node %d: ", i); scanf("%d", &data);

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

temp->next = newNode; temp = temp->next;

}

}

printf("\nSINGLY LINKED LIST CREATED SUCCESSFULLY\n");

}

}

void reverseList()

{

struct node \*prevNode, \*curNode;

if(head != NULL)

{

prevNode = head; curNode = head->next; head = head->next;

prevNode->next = NULL; while(head != NULL)

{

head = head->next; curNode->next = prevNode;

prevNode = curNode; curNode = head;

}

head = prevNode; // Make last node as head

printf("\nSUCCESSFULLY REVERSED LIST\n");

}

}

void displayList()

{

struct node \*temp;

if(head == NULL)

{

printf("\nList is empty.");

}

else

{

temp = head;

//printf("\nThe elements are");

while(temp != NULL)

{

printf("\n%d", temp->data); // Print the data of current node temp = temp->next; // Move to next node

}

}

}

## SAMPLE INPUT – OUTPUTS :

Reverse the Linked List

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Enter the total number of nodes: 6

Enter the data of the NODE: 1

Enter the data of node 2: 3

Enter the data of node 3: 4

Enter the data of node 4: 6

Enter the data of node 5: 7

Enter the data of node 6: 8

SINGLY LINKED LIST CREATED SUCCESSFULLY

Data in the Linked list : 1 3

4

6

7

8

SUCCESSFULLY REVERSED LIST

Data in the Linked list after Reverse 8 7

6

4

3

1

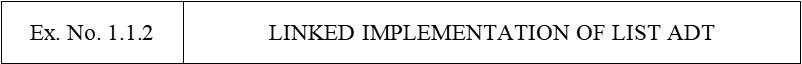
## RESULT

The Reverse List was implemented in C programming language and all the relevant operations were executed for various test cases and verified.

CS8381 – DATA STRUCTURES LABORATORY CYCLE I

LINEAR DATA STRUCTURES

|  |  |
| --- | --- |
| Ex. No. | 1.1.2 |
| NAME OF THE EXERCISE | LINKED IMPLEMENTATION OF LIST ADT |
| DATE OF EXERCISE | DD.MM.YYYY |
| DATE OF SUBMISSION | 08.12.2020 |
| STUDENT REG. NO. | 920419205025 |
| STUDENT ROLL NO. | 19UITE012 |
| STUDENT NAME | B.NAGA SAPTA AAKASH |
| FACULTY IN- CHARGE | Dr. P. SUBATHRA, Prof. / IT Dr. E. VAKAIMALAR, AP/IT **Mr. D. VENDHAN, AP/IT** |



# AIM:

To implement a List ADT using Linked representation. This program uses a singly linked structure for its operation.

# DESCRIPTION:

A linked list is a data structure used for storing data. A linked list has the following properties:

* Successive elements are connected by pointers.
* The last element points to NULL.
* Can grow or shrink in size during program execution.
* Can be made as long as needed. (Until system memory exhausts).
* Doesn’t waste memory space but takes some extra memory for pointers. It allocates memory as it grows.

Link list is an ADT and has the following operations: Main link list ADT operation:

1. **Insert:** Insert element to the list
2. **Delete:** deletes element from the list Auxiliary link list ADT operation
3. **Delete list:** delete and remove the whole list
4. **Count:** returns the number of elements in the list
5. **Find nth node:** Returns the nth node from the beginning of the list

# ALGORITHM:

**Operations on Single Linked List**

The following operations are performed on a Single Linked List

# Insertion

1. **Deletion**
2. **Display**
3. **Search**
4. **Count**
5. **Find nth node**

Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.

* **Step 1 -** Include all the **header files** which are used in the program.
* **Step 2 -** Declare all the **user defined functions**.
* **Step 3 -** Define a **Node** structure with two members **data** and **next**
* **Step 4 -** Define a Node pointer **'head'** and set it to **NULL**.
* **Step 5 -** Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

# Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

* 1. Inserting At Beginning of the list
  2. Inserting At End of the list
  3. Inserting At Specific location in the list
  4. Insert Before a Given Element in the list
  5. Insert After a Given Element in the list

# Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

* **Step 1 -** Create a **newNode** with given value.

# Step 2 – Irrespective of whether the list is EMPTY or NON EMPTY, do the following

set **newNode→next** = **head** and **head** = **newNode**.

# Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

* **Step 1 -** Create a **newNode** with given value and **newNode → next** as **NULL**.
* **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
* **Step 3 -** If it is **Empty** then, set **head** = **newNode**. (OR) **Call Insert At Beginning of the List**

module

* **Step 4 -** If it is **Not Empty** then, define a node pointer **trav** and initialize with **head**.
* **Step 5 -** Keep moving the **trav** to its next node until it reaches to the last node in the list (until **trav → next** is equal to **NULL**).

# Step 6 - **Set** trav → next **=** newNode**.**

**Inserting At Specific location in the list (After a Node)**

We can use the following steps to insert a new node after a node in the single linked list...

* **Step 1 -** Create a **newNode** with given value.
* **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
* **Step 3 -** If it is **Empty** then, print EMPTY list and return.
* **Step 4 -** If it is **Not Empty** then, define a node pointer **trav** and initialize with **head**.
* **Step 5 -** Keep moving the **trav** to its next node until it reaches to the node after which we want to insert the newNode (until **trav → data** is equal to (**location-1)**, here location is the node value after which we want to insert the newNode).
* **Step 6 -** Every time check whether **trav** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **trav** to next node.
* **Step 7 -** Finally, Set '**newNode → next** = **trav → next**' and **trav → next** = **newNode**'

# Inserting Before a Given Element in the List

We can use the following steps to insert a new node after a node in the single linked list...

* **Step 1 -** Create a **newNode** with given value.
* **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
* **Step 3 -** If it is **Empty** then, print EMPTY list and return.
* **Step 4 –** If the GIVEN\_ELEMENT is available in the first node of the list, then use INSERT\_FIRST procedure.
* **Step 5 –** If not, define a node pointer **trav** and initialize with **head**.
* **Step 6 -** Keep moving the **trav** to its next node until it reaches to the node where **(trav->next-> data = GIVEN\_ELEMENT)**
* **Step 7 -** Every time check whether **trav->next** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **trav** to next node.
* **Step 8 -** Finally, now the new node can be inserted next to the **trav** as **(newNode-> next=trav-**

# >next) **and** trav->next = newNode

**Inserting After a Given Element in the List**

We can use the following steps to insert a new node after a node in the single linked list...

* **Step 1 -** Create a **newNode** with given value.
* **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
* **Step 3 -** If it is **Empty** then, print EMPTY list and return.
* **Step 4 -** If it is **Not Empty** then, define a node pointer **trav** and initialize with **head**.
* **Step 5 -** Keep moving the **trav** to its next node until it reaches to the node where **trav->data** is equal to **GIVEN\_ELEMENT**
* **Step 6 -** Every time check whether **trav** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **trav** to next node.
* **Step 7 -** Finally, Set '**newNode → next** = **trav → next**' and **trav → next** = **newNode**'

# Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

* 1. Deleting from Beginning of the list
  2. Deleting from End of the list
  3. Deleting a Specific Node

# Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

* **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
* **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
* **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
* **Step 4 -** Check whether list is having only one node (**temp → next** == **NULL**)
* **Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)
* **Step 6 -** If it is **FALSE** then set **head** = **temp → next**, and delete **temp**.

# Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

* **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
* **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
* **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
* **Step 4 -** Check whether list has only one Node (**temp1 → next** == **NULL**)
* **Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
* **Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1 → next** == **NULL**)
* **Step 7 -** Finally, Set **temp2 → next** = **NULL** and delete **temp1**.

# Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

* **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
* **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
* **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
* **Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
* **Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
* **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
* **Step 7 -** If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
* **Step 8 -** If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
* **Step 9 -** If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.
* **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next**

**== NULL**).

* **Step 11 -** If **temp1** is last node then set **temp2 → next** = **NULL** and delete **temp1** (**free(temp1)**).
* **Step 12 -** If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

# Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

* **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
* **Step 2 -** If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.
* **Step 3 -** If it is **Not Empty** then, define a Node pointer **'trav'** and initialize with **head**.
* **Step 4 -** Keep displaying **trav → data** with an arrow (**--->**) until **trav** reaches to the last node
* **Step 5 -** Finally display **trav → data** with arrow pointing to **NULL** (**trav → data ---> NULL**).

# Searching for an Element in a Singly linked List

We can use the following steps to search for an element in a single linked list...

* **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
* **Step 2 -** If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.
* **Step 3 -** If it is **Not Empty** then, define a Node pointer **'trav'** and initialize with **head**.
* **Step 4 -** Keep checking if GIVEN\_ELEMENT is equal to **temp → data** until **trav** reaches to the last node
* **Step 5 –** If **GIVEN\_ELEMENT** is found, display the result and return
* **Step 6 –** If the search reaches the last node unsuccessfully, display **(ELEMENT NOT AVAILABLE IN THE LIST)**

# Find the Nth node in the Singly Linked List

We can use the following steps to find the Nth node in a single linked list...

* **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
* **Step 2 -** If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.
* **Step 3 -** If it is **Not Empty** then, define a Node pointer **'trav'** and initialize with **head**.
* **Step 4 –** Initialize **COUNTER =0**
* **Step 5 -** Keep traversing through the list with **trav=trav-> next** and increment **COUNTER** for each of the traversal
* **Step 6 –** If **COUNTER** is equal to **N**, display **trav->data**.
* **Step 7–** If the search reaches the last node and still COUNTER is less than N, display **(Nth POSITION NOT AVAILABLE IN THE LIST)**

# ONLINE CODE LINK

**https://onlinegdb.com/BJevMgNAw**

**SOURCE CODE**

//Name : B.NAGA SAPTA AAKASH Roll no : 19UITE012

//Singly Linked list

#include<stdio.h> #include<stdlib.h> struct node

{

int data;

struct node\*next;//create a node.

};

struct node\*head=NULL;//Asign head is null.Initially. void printList()

{

if(head==NULL)

{

printf("\nList is Empty...Nothing to print");

}

/\*else if(head!=NULL)

{

}\*/ else

{

struct node\*trav=head; printf("\t..%d",trav->data);

struct node\*trav=head; //trav is a temperory pointer variable points the head. while(trav!=NULL)

{

printf("\t..%d",trav->data); trav=trav->next;

}

}

}

void insertFirst()

{

struct node\*temp=(struct node\*)malloc(sizeof(struct node)); printf("\nEnter the element to be inserted first: "); scanf("%d",&temp->data);

temp->next=head; head=temp;

}

void insertLast()

{

struct node\*temp=(struct node\*)malloc(sizeof(struct node));

printf("\nEnter the element to be inserted Last: "); scanf("%d",&temp->data);

temp->next=NULL; if(head==NULL)

{

}

else

{

head=temp;

struct node\*trav; trav=head;

while(trav->next!=NULL)

{

trav=trav->next;

}

trav->next=temp;

}

}

void insertBeforeElement ()

{

int x;

{

if (head == NULL)

printf("\nList is Empty, CANNOT perform the operation Insert Before Element"); else

struct node \*temp = (struct node \*) malloc (sizeof (struct node)); printf ("\n Element to be inserted =");

scanf ("%d", &temp->data);

printf ("\n Before which element:"); scanf ("%d", &x);

printf ("\nList before INSERT BEFORE ELEMENT"); printList ();

if (head->data == x)

{

temp->next = head; head = temp;

}

else

{

struct node \*prev = head; struct node \*curr = head->next;

while (curr->data != x && curr->next != NULL)

{

curr = curr->next; prev = prev->next;

}

if (curr->data == x)

{

temp->next = curr; //temp->next=prev->next; prev->next = temp;

}

>data);

else

printf("\n%d is not available in the list so Element %d cannot be inserted ",x, temp-

}

printf ("\nList after INSERT BEFORE ELEMENT"); printList ();

}

}

void insertAfterElement ()

{

int x;

if (head == NULL)

printf("\nList is Empty, CANNOT perform the operation Insert Before Element"); else

{

struct node \*temp = (struct node \*) malloc (sizeof (struct node)); printf ("\n Element to be inserted :");

scanf ("%d", &temp->data); printf ("\n After which element:"); scanf ("%d", &x);

printf ("\nList before INSERT AFTER ELEMENT"); printList ();

struct node \*curr = head;

while (curr->data != x && curr->next != NULL)

{

curr = curr->next;

}

if (curr->data == x)

{

}

temp->next = curr->next; curr->next = temp;

else temp->data);

printf("\nExpected element %dis not availablein the list so Element %d cannot be inserted ",x,

printf ("\nList after INSERT AFTER ELEMENT"); printList ();

}

}

void insertAtPosition()

{

int position;

struct node\*previous=NULL; struct node\*current=NULL;

printf("\nEnter the position to insert the value"); scanf("%d",&position);

struct node\*temp=(struct node\*)malloc(sizeof(struct node)); printf("\nEnter the element to be inserted first: "); scanf("%d",&temp->data);

if(head==NULL)

{

printf("\nLIST is empty we cannot process INSERT at a POSITION :");

}

else if(head->next==NULL)

{

if(position==1)

{

}

}

else

{

temp->next=head; head=temp;

previous=head; current=head->next; int count=1;

while(count<(position-1) && current->next!=NULL)

{

current=current->next; previous=previous->next; count++;

}

if(count==(position-1))

{

}

else

{

}

}

}

temp->next = current; //temp->next=prev->next; previous->next = temp;

printf("\n Position is not available in the List");

int deleteFirst()

{

if(head==NULL)

{

printf("\nTher is no element to DELETE from the LIST"); printf("\nIgnore the RETURN value ");

return -999;

}

else

{

struct node\*trav=head;//collect the whole node. int temp=head->data;

head=head->next; free(trav); //freeup the node

trav=NULL; //make it as null.. return temp;

}

}

int deleteLast()

{

if(head==NULL)

{

printf("\nTher is no element to DELETE from the LIST"); printf("\nIgnore the RETURN value ");

return -999;

}

else if (head->next==NULL)

{

int x=head->data;

struct node\*temp=head;//collect the whole node. head=head->next;

free(temp); //freeup the node temp=NULL; //make it as null.. return x;

}

else

{

struct node\*previous=head; struct node\*current=head->next; while(current->next!=NULL)

{

current=current->next; previous=previous->next;

}

int x=current->data;

previous->next=NULL; //make it as null.. free(current); //freeup the node current=NULL;

return x;

}

}

int deleteElement()

{

int element,count;

printf("Enter the search element :"); scanf("%d",&element);

struct node\*previous=NULL; struct node\*current=NULL;

if(head==NULL)

{

printf("\nTher is no element to DELETE from the LIST"); printf("\nIgnore the RETURN value ");

return -999;

}

else if(head->next==NULL)

{

if(head->data==element)

{

}

else

{

}

current=head; head=head->next; free(current); current=NULL; return count=0;

printf("\nThe Search element is not found\nIgnore the RETURN value..."); return -999;

}

else

{

previous=head; current=head->next; count=1;

while(current->next!=NULL && current->data!=element)

{

current=current->next; previous=previous->next; count++;

}

if(current->data==element)

{

}

else

{

int x=current->data;

previous->next=current->next; //make it as null.. free(current); //freeup the node

current=NULL; return count;

printf("\nThe Search element is not found\nIgnore the RETURN value. ");

return -999;

}

}

}

int deletePosition ()

{

if (head == NULL)

{

printf ("\n List is Empty, we cannot perform DELETE Position Operation"); printf ("\n Ignore the return value");

return (-99);

}

else

{

int posi, x, count;

printf ("\n Enter the Position to be deleted:"); scanf ("%d", &posi);

if (posi < 1)

{

}

printf ("\n Invalid Position..!"); printf ("\n Ignore the return value"); return (-999);

else if (posi == 1)

{

x = deleteFirst ();

else

return (x);

}

{

printf ("\nList before Deletion"); printList ();

struct node \*prev = head; struct node \*curr = head->next; count = 1;

while (count < (posi - 1) && curr->next != NULL)

{

prev = prev->next; curr = curr->next; count++;

printf ("\n%d", count);

}

if (count == (posi - 1))

{

x = curr->data;

prev->next = curr->next;

free (curr);

curr = NULL;

printf ("\nList after Deletion"); printList ();

return (x);

}

else

{

printf("\n Position is not available in the List"); printf ("\n Ignore the return value");

return (-999);

}

}

}

}

int main()

{

int choice,cont,Z;

do

{

printf("\t\tSINGLY LINKED LIST OPERATIONS\n");

printf("\t\t "); printf("\n1. Print List");

printf("\n2. Insert First");

printf("\n3. Insert Last"); printf("\n4. Insert Before an Element");

printf("\n5. Insert After an Element"); printf("\n6. Insert at a Position");

printf("\n7. Delete First"); printf("\n8. Delete Last");

printf("\n9. Delete Element"); printf("\n10. Delete Position");

printf("\n "); printf("\n\nChoose the List Operation??\t(1-10) :"); scanf("%d", &choice);

switch(choice)

{

case 1:

printList(); break;

case 2:

insertFirst(); break;

case 3:

insertLast(); break;

case 4:

insertBeforeElement(); break;

case 5:

insertAfterElement(); break;

case 6:

insertAtPosition(); break;

case 7:

Z=deleteFirst();

printf("\nDeleted Element is: %d", Z); break;

case 8:

Z=deleteLast();

printf("\nDeleted Element is: %d", Z); break;

case 9:

Z=deleteElement(); if(Z!=0)

printf("\nPosition of the Deleted Element is: %d", Z); break;

case 10:

Z=deletePosition(); if(Z!=-999)

printf("\nElement in the Deleted Positionis: %d", Z); break;

default:

printf("\nEnter a valid option\n");

}

printf("\nDo you Continue List Operations??\t(1/0)\t:"); scanf("%d", &cont);

}while(cont==1); return 0;

}

# SAMPLE OUTPUT :

SINGLY LINKED LIST OPERATIONS

|  |
| --- |
|  |
|  |  |

1. Print List

1. Insert First

1. Insert Last

1. Insert Before an Element

1. Insert After an Element

1. Insert at a Position

1. Delete First

1. Delete Last

1. Delete Element

1. Delete Position

|  |
| --- |
|  |
|  |  |

Choose the List Operation?? (1-10) :2

Enter the element to be inserted first: 3

Do you Continue List Operations?? (1/0) :1

SINGLY LINKED LIST OPERATIONS

|  |
| --- |
|  |
|  |  |

1. Print List

1. Insert First

1. Insert Last

1. Insert Before an Element
2. Insert After an Element

1. Insert at a Position

1. Delete First

1. Delete Last

1. Delete Element

1. Delete Position

|  |
| --- |
|  |
|  |  |

Choose the List Operation?? (1-10) :4

Element to be inserted =5

Before which element:3

List before INSERT BEFORE ELEMENT ..3

List after INSERT BEFORE ELEMENT ..5 ..3

Do you Continue List Operations?? (1/0) :1

SINGLY LINKED LIST OPERATIONS

|  |
| --- |
|  |
|  |  |

1. Print List

1. Insert First

1. Insert Last

1. Insert Before an Element

1. Insert After an Element

1. Insert at a Position

1. Delete First
2. Delete Last

1. Delete Element

1. Delete Position

|  |
| --- |
|  |
|  |  |

Choose the List Operation?? (1-10) :3

Enter the element to be inserted Last: 6

Do you Continue List Operations?? (1/0) :1

SINGLY LINKED LIST OPERATIONS

|  |
| --- |
|  |
|  |  |

1. Print List

1. Insert First

1. Insert Last

1. Insert Before an Element

1. Insert After an Element

1. Insert at a Position

1. Delete First

1. Delete Last

1. Delete Element

1. Delete Position

|  |
| --- |
|  |
|  |  |

Choose the List Operation?? (1-10) :4 5

Element to be inserted :5 7 4

After which element:5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| List before INSERT AFTER ELEMENT | ..5 | ..3 | ..6 |  |
| List after INSERT AFTER ELEMENT  Do you Continue List Operations?? (1/0) | ..5  :1 | ..4 | ..3 | ..6 |

SINGLY LINKED LIST OPERATIONS

|  |
| --- |
|  |
|  |  |

1. Print List

1. Insert First

1. Insert Last

1. Insert Before an Element

1. Insert After an Element

1. Insert at a Position

1. Delete First

1. Delete Last

1. Delete Element

1. Delete Position

|  |
| --- |
|  |
|  |  |

Choose the List Operation?? (1-10) :6

Enter the position to insert the value4

Enter the element to be inserted first: 2

Do you Continue List Operations?? (1/0) :1

SINGLY LINKED LIST OPERATIONS

|  |
| --- |
|  |
|  |  |

1. Print List

1. Insert First

1. Insert Last

1. Insert Before an Element

1. Insert After an Element

1. Insert at a Position

1. Delete First

1. Delete Last

1. Delete Element

1. Delete Position

|  |
| --- |
|  |
|  |  |

Choose the List Operation?? (1-10) :1

..5 ..4 ..3 ..2 ..6

Do you Continue List Operations?? (1/0) :1

SINGLY LINKED LIST OPERATIONS

|  |
| --- |
|  |
|  |  |

1. Print List

1. Insert First

1. Insert Last

1. Insert Before an Element

1. Insert After an Element
2. Insert at a Position

1. Delete First

1. Delete Last

1. Delete Element

1. Delete Position

|  |
| --- |
|  |
|  |  |

Choose the List Operation?? (1-10) :7

Deleted Element is: 5

Do you Continue List Operations?? (1/0) :1

SINGLY LINKED LIST OPERATIONS

|  |
| --- |
|  |
|  |  |

1. Print List

1. Insert First

1. Insert Last

1. Insert Before an Element

1. Insert After an Element

1. Insert at a Position

1. Delete First

1. Delete Last

1. Delete Element

1. Delete Position

|  |
| --- |
|  |
|  |  |

Choose the List Operation?? (1-10) :8

Deleted Element is: 6

Do you Continue List Operations?? (1/0) :1

SINGLY LINKED LIST OPERATIONS

|  |
| --- |
|  |
|  |  |

1. Print List

1. Insert First

1. Insert Last

1. Insert Before an Element

1. Insert After an Element

1. Insert at a Position

1. Delete First

1. Delete Last

1. Delete Element

1. Delete Position

|  |
| --- |
|  |
|  |  |

Choose the List Operation?? (1-10) :9 Enter the search element :3

Position of the Deleted Element is: 1

Do you Continue List Operations?? (1/0) :1

SINGLY LINKED LIST OPERATIONS

|  |
| --- |
|  |
|  |  |

1. Print List

1. Insert First

1. Insert Last

1. Insert Before an Element

1. Insert After an Element

1. Insert at a Position

1. Delete First

1. Delete Last

1. Delete Element

1. Delete Position

|  |
| --- |
|  |
|  |  |

Choose the List Operation?? (1-10) :1

..4 ..2

Do you Continue List Operations?? (1/0) :0

# RESULT

The Singly linked List was implemented in C programming language and all the relevant operations were executed for various test cases and verified.

# **CS8381 – DATA STRUCTURES LABORATORY CYCLE I**

LINEAR DATA STRUCTURES

|  |  |
| --- | --- |
| Ex. No. | 1.1.2.1 |
| NAME OF THE EXERCISE | APPLICATIONS OF LIST ADT - MERGING TWO LISTS |
| DATE OF EXERCISE | DD.MM.YYYY |
| DATE OF SUBMISSION | 08.12.2020 |
| STUDENT REG. NO. | 920419205025 |
| STUDENT ROLL NO. | 19UITE012 |
| STUDENT NAME | B.NAGA SAPTA AAKASH |
| FACULTY IN- CHARGE | Dr. P. SUBATHRA, Prof. / IT Dr. E. VAKAIMALAR, AP/IT **Mr. D. VENDHAN, AP/IT** |



**AIM:**

To merge the given two sorted lists into a single linked list by maintaining the sorted order.

**DESCRIPTION:**

The goal here is merge two linked lists that are already sorted.

Example:

List a : →2→6→18 List b: →1→3→17→19

Merged List: →1→2→3→6→17→18→19

**ALGORITHM:**

**Approaches:**

**Without Recursion:**

* Create a new node say result
* Navigate through both the linked lists at the same time, starting from head
* Compare the first node values of both the linked lists
* Whichever is smaller, add it to the result node
* Move the head pointer of the linked list whose value was smaller
* Again compare the node values
* Keep doing until one list gets over
* Copy the rest of the nodes of unfinished list to the result

**With Recursion:**

* Base Case :

If List A gets finished , return List B. If List B gets finished, return List A.

* Create a result node and initialize it as NULL
* Check which node (List A or List B) has a smaller value.
* Whichever is smaller, add it to the result node.
* Make recursive call and add the return node as result.next result.next = recurrsionMerge(nA.next, nB)

|  |
| --- |
|  |
|  |  |

1.1.2.1 Application of List ADT – Merging Two Lists Page 1

**ONLINE CODE LINK :**

[**https://onlinegdb.com/BJ-qar4Yv**](https://onlinegdb.com/BJ-qar4Yv)

**SOURCE CODE :**

//Name :B.NAGA SAPTA AAKASH Roll no :19UITE012

//Program name :Merge two Linked List #include <stdio.h>

#include<stdlib.h> struct node

{

int data;

struct node \*next;

};

struct node \*head1=NULL; struct node \*head2=NULL;

void list1()

{

struct node \*temp=(struct node\*)malloc(sizeof(struct node)); scanf("%d",&temp->data);

temp->next=NULL; if(head1==NULL)

{

head1=temp;

}

else

{

struct node \*trave=head1;

while(trave->next!=NULL)

{

trave=trave->next;

}

trave->next=temp;

}

}

void list2()

{

struct node \*temp=(struct node\*)malloc(sizeof(struct node));

scanf("%d",&temp->data); temp->next=NULL; if(head2==NULL)

{

head2=temp;

}

else

{

struct node \*trave=head2;

while(trave->next!=NULL)

{

trave=trave->next;

}

trave->next=temp;

}

}

void merge\_List()

{

struct node\* trav; trav=head1;

while(trav->next!=NULL)

{

trav=trav->next;

}

trav->next=head2;

}

void displayList()

{

if(head1==NULL)

{

printf("\nList is Empty nothing to PRINT");

}

else

{

struct node \*trave=head1; printf("\nThe List elements are \n");

while(trave!=NULL)

{

printf("\t...%d",trave->data); trave=trave->next;

}

}

}

void sortafterMerge()

{

struct node \*t; struct node \*s; int x; t=head1;

while(t!=NULL)

{

s=t->next; while(s!=NULL)

{

if(t->data >s->data)

{

x=t->data;

t->data=s->data; s->data=x;

}

s=s->next;

}

t=t->next;

}

t=head1;

printf("\nSorted list -->\n"); while(t!=NULL)

{

printf(" \t...%d ",t->data); t=t->next;

}

}

int main()

{

printf("\n\t\tMerge two sorted Linked List"); printf("\n\t\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n"); int i,n1,n2;

printf("\nEnter number of elements for the first list :"); scanf("%d",&n1);

printf("\nEnter %d elements :",n1); for(i=0;i<n1;i++)

{

list1();

}

printf("\nEnter number of elements for the second list :"); scanf("%d",&n2);

printf("\nEnter %d elements :",n2); for(i=0;i<n2;i++)

{

list2();

}

merge\_List();

printf("\nAfter merging -->\n"); displayList();

sortafterMerge(); return 0;

}

**SAMPLE OUTPUT :**

Merge two sorted Linked List

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Enter number of elements for the first list :2

Enter 2 elements :3

4

Enter number of elements for the second list :3 Enter 3 elements :5

6

7

After merging --> The List elements are

...3 ...4 ...5 ...6 ...7

Sorted list -->

...3 ...4 ...5 ...6 ...7

**RESULT**

The Merging two list was implemented in C programming language and all the relevant operations were executed for various test cases and verified.

|  |  |
| --- | --- |
| Ex. No. | 1.1.2.2 |
| NAME OF THE EXERCISE | APPLICATIONS OF LIST ADT - REVERSING A LIST |
| DATE OF EXERCISE | DD.MM.YYYY |
| DATE OF SUBMISSION | 07.12.2020 |
| STUDENT REG. NO. | 920419205025 |
| STUDENT ROLL NO. | 19UITE012 |
| STUDENT NAME | RAHUL VISWANATH G |
| FACULTY IN- CHARGE | Dr. P. SUBATHRA, Prof. / IT Dr. E. VAKAIMALAR, AP/IT **Mr. D. VENDHAN, AP/IT** |