**Linked List in Data Structure**

A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the **data** and the **address** of the next node.



**Representation of Linked List**

 Each node consists:

* A data item
* An address of another node

We wrap both the data item and the next node reference in a struct as:

struct node

{

int data;

struct node \*next;

};

## Linked List Operations:

## Traverse, Insert and Delete

Here's a list of basic linked list operations that we will cover in this article.

* 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

## Traverse a Linked List

struct node \*temp = head;

printf("\n\nList elements are - \n");

while(temp != NULL) {

printf("%d --->",temp->data);

temp = temp->next;

}

## Insert Elements to a Linked List

### 1. Insert at the beginning

* Allocate memory for new node
* Store data
* Change next of new node to point to head
* Change head to point to recently created node
* struct node \*newNode;
* newNode = malloc(sizeof(struct node));
* newNode->data = 4;
* newNode->next = head;
* head = newNode;

### 2. Insert at the End

* Allocate memory for new node
* Store data
* Traverse to last node
* Change next of last node to recently created node

struct node \*newNode;

newNode = malloc(sizeof(struct node));

newNode->data = 4;

newNode->next = NULL;

struct node \*temp = head;

while(temp->next != NULL){

temp = temp->next;

}

temp->next = newNode;

### 3. Insert at the Middle

* Allocate memory and store data for new node
* Traverse to node just before the required position of new node
* Change next pointers to include new node in between

struct node \*newNode;

newNode = malloc(sizeof(struct node));

newNode->data = 4;

struct node \*temp = head;

for(int i=2; i < position; i++) {

if(temp->next != NULL) {

temp = temp->next;

}

}

newNode->next = temp->next;

temp->next = newNode;

## Delete from a Linked List

### 1. Delete from beginning

* Point head to the second node

head = head->next;

### 2. Delete from end

* Traverse to second last element
* Change its next pointer to null

struct node\* temp = head;

while(temp->next->next!=NULL){

temp = temp->next;

}

temp->next = NULL;

### 3. Delete from middle

* Traverse to element before the element to be deleted
* Change next pointers to exclude the node from the chain

for(int i=2; i< position; i++) {

if(temp->next!=NULL) {

temp = temp->next;

}

}

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

## Search an Element on a Linked List

* Make head as the current node.
* Run a loop until the current node is NULL because the last element points to NULL.
* In each iteration, check if the key of the node is equal to item. If it the key matches the item, return true otherwise return false.

// Search a node

bool searchNode(struct Node\*\* head\_ref, int key) {

struct Node\* current = \*head\_ref;

while (current != NULL) {

if (current->data == key) return true;

current = current->next;

}

return false;

}

## Sort Elements of a Linked List

1. Make the head as the current node and create another node index for later use.
2. If head is null, return.
3. Else, run a loop till the last node (i.e. NULL).
4. In each iteration, follow the following step 5-6.
5. Store the next node of current in index.
6. Check if the data of the current node is greater than the next node. If it is greater, swap current and index.

// 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 points to the node next to current

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;

}

}

}

**Complete Program**

**// Linked list operations in C++**

**#include <stdlib.h>**

**#include <iostream>**

**using namespace std;**

**// Create a node**

**struct Node {**

**int data;**

**struct Node\* next;**

**};**

**void insertAtBeginning(struct Node\*\* head\_ref, int new\_data) {**

**// Allocate memory to a node**

**struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));**

**// insert the data**

**new\_node->data = new\_data;**

**new\_node->next = (\*head\_ref);**

**// Move head to new node**

**(\*head\_ref) = new\_node;**

**}**

**// Insert a node after a node**

**void insertAfter(struct Node\* prev\_node, int new\_data) {**

**if (prev\_node == NULL) {**

**cout << "the given previous node cannot be NULL";**

**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; /\* used in step 5\*/**

**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;**

**return;**

**}**

**// Delete a node**

**void deleteNode(struct Node\*\* head\_ref, int key) {**

**struct Node \*temp = \*head\_ref, \*prev;**

**if (temp != NULL && temp->data == key) {**

**\*head\_ref = temp->next;**

**free(temp);**

**return;**

**}**

**// Find the key to be deleted**

**while (temp != NULL && temp->data != key) {**

**prev = temp;**

**temp = temp->next;**

**}**

**// If the key is not present**

**if (temp == NULL) return;**

**// Remove the node**

**prev->next = temp->next;**

**free(temp);**

**}**

**// Search a node**

**bool searchNode(struct Node\*\* head\_ref, int key) {**

**struct Node\* current = \*head\_ref;**

**while (current != NULL) {**

**if (current->data == key) return true;**

**current = current->next;**

**}**

**return false;**

**}**

**// 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 points to the node next to current**

**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) {**

**cout << node->data << " ";**

**node = node->next;**

**}**

**}**

**// Driver program**

**int main() {**

**struct Node\* head = NULL;**

**insertAtEnd(&head, 1);**

**insertAtBeginning(&head, 2);**

**insertAtBeginning(&head, 3);**

**insertAtEnd(&head, 4);**

**insertAfter(head->next, 5);**

**cout << "Linked list: ";**

**printList(head);**

**cout << "\nAfter deleting an element: ";**

**deleteNode(&head, 3);**

**printList(head);**

**int item\_to\_find = 3;**

**if (searchNode(&head, item\_to\_find)) {**

**cout << endl << item\_to\_find << " is found";**

**} else {**

**cout << endl << item\_to\_find << " is not found";**

**}**

**sortLinkedList(&head);**

**cout << "\nSorted List: ";**

**printList(head);**

**}**

# Types of Linked List

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

# **Singly Linked List**

Singly linked list is a type of data structure that is made up of nodes that are created using self-referential structures. Each of these nodes contain two parts, namely the data and the reference to the next list node. Only the reference to the first list node is required to access the whole linked list

Program

struct Node {

   int data;

   struct Node \*next;

};

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Inserting Data \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

struct Node\* head = NULL;

void insert(int new\_data) {

   struct Node\* new\_node = (struct Node\*) malloc(sizeof(struct Node));

   new\_node->data = new\_data;

   new\_node->next = head;

   head = new\_node;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Display Data \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void display() {

   struct Node\* ptr;

   ptr = head;

   while (ptr != NULL) {

      cout<< ptr->data <<" ";

      ptr = ptr->next;

   }

}