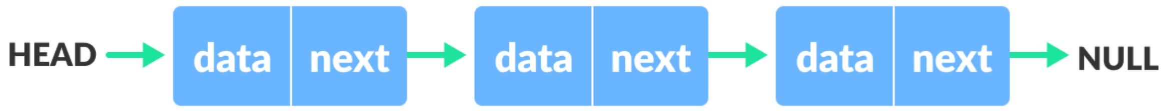
Linked List :

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. For example,



Linked list Data Structure

You have to start somewhere, so we give the address of the first node a special name called HEAD. Also, the last node in the linked list can be identified because its next portion points to NULL.

Linked lists can be of multiple types: singly, doubly, and circular linked list. In this article, we will focus on the singly linked list. To learn about other types, visit Types of Linked List.

Representation of Linked List :

Let's see how each node of the linked list is represented. 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;
};
```

Understanding the structure of a linked list node is the key to having a grasp on it.

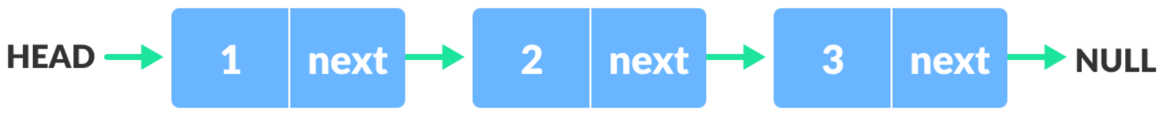
Each struct node has a data item and a pointer to another struct node. Let us create a simple Linked List with three items to understand how this works.

```
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
```

```
three = malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data=3;

/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;

/* Save address of first node in head */
head = one;
```

If you didn't understand any of the lines above, all you need is a refresher on pointers and structs.

In just a few steps, we have created a simple linked list with three nodes.



Linked list Representation

The power of a linked list comes from the ability to break the chain and rejoin it. E.g. if you wanted to put an element 4 between 1 and 2, the steps would be:

- Create a new struct node and allocate memory to it.
- Add its data value as 4
- Point its next pointer to the struct node containing 2 as the data value
- Change the next pointer of "1" to the node we just created.

Doing something similar in an array would have required shifting the positions of all the subsequent elements.

In python and Java, the linked list can be implemented using classes as shown in the codes below.

Linked List Utility :

Apart from that, linked lists are a great way to learn how pointers work. By practicing how to manipulate linked lists, you can prepare yourself to learn more advanced data structures like graphs and trees.

Linked List Implementations in c++

```cpp
#include <bits/stdc++.h>
#include <iostream>
using namespace std;

// Creating a node
class Node {
   public:
  int value;
  Node* next;
};

int main() {
  Node* head;
  Node* one = NULL;
  Node* two = NULL;
  Node* three = NULL;

  // allocate 3 nodes in the heap
  one = new Node();
  two = new Node();
  three = new Node();

  // Assign value values
  one->value = 1;
  two->value = 2;
  three->value = 3;

  // Connect nodes
  one->next = two;
  two->next = three;
  three->next = NULL;

  // print the linked list value
  head = one;
  while (head != NULL) {
    cout << head->value;
    head = head->next;
  }
}
```

Linked List Complexity :

Time Complexity

	Worst case	Average Case

Search	O(n)	O(n)
Insert	O(1)	O(1)
Deletion	O(1)	O(1)

Space Complexity: O(n)

Linked List Applications :

- **Dynamic memory allocation**
- **Implemented in stack and queue**
- **In undo functionality of softwares**
- **Hash tables, Graphs**

There are various linked list operations that allow us to perform different actions on linked lists. For example, the insertion operation adds a new element to the linked list.

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**

Before you learn about linked list operations in detail, make sure to know about Linked List first.

Things to Remember about Linked List

- **head points to the first node of the linked lis**
- **next pointer of the last node is NULL, so if the next current node is NULL, we have reached the end of the linked list.**

In all of the examples, we will assume that the linked list has three nodes 1 --->2 --->3 with node structure as below:

```
struct node {
  int data;
```

```
 struct node *next;
};
```

Traverse a Linked List :

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.

When temp is NULL, we know that we have reached the end of the linked list so we get out of the while loop.

```
struct node *temp = head;
printf("\n\nList elements are - \n");
while(temp != NULL) {
 printf("%d --->",temp->data);
 temp = temp->next;
}
```

The output of this program will be:

```
List elements are -
1 --->2 --->3 --->
```

Insert Elements to a Linked List :

You can add elements to either the beginning, middle or end of the 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

You can delete either from the beginning, end or from a particular position.

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

You can search an element on a linked list using a loop using the following steps. We are finding item 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 :

We will use a simple sorting algorithm, Bubble Sort, to sort the elements of a linked list in ascending order below.

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.

Check the article on bubble sort for better understanding of its working.

```
// 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;
    }
  }
}
```