Unit 2

Linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

* **Link** − Each link of a linked list can store a data called an element.
* **Next** − Each link of a linked list contains a link to the next link called Next.
* **Linked List** − A Linked List contains the connection link to the first link called First.

**Linked List Representation**

Linked list can be visualized as a chain of nodes, where every node points to the next node.

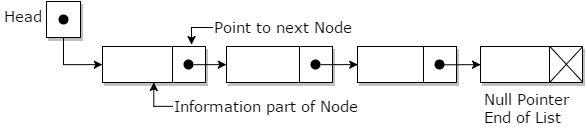


As per the above illustration, following are the important points to be considered.

* Linked List contains a link element called first.
* Each link carries a data field(s) and a link field called next.
* Each link is linked with its next link using its next link.
* Last link carries a link as null to mark the end of the list.

Each node is separated into two different parts:

* The first part holds the information of the element or node
* The second piece contains the address of the next node (link / next-pointer field) in this structure list.



Linked lists can be measured as a form of high-level standpoint as being a series of nodes where each node has at least one single pointer to the next connected node, and in the case of the last node, a null pointer is used for representing that there will be no further nodes in the linked list. In the data structure, you will be implementing the linked lists which always maintain head and tail pointers for inserting values at either the head or tail of the list is a constant time operation. Randomly inserting of values is excluded using this concept and will follow a linear operation. As such, linked lists in data structure have some characteristics which are mentioned below:

* Insertion is O(1)
* Deletion is O(n)
* Searching is O(n)

Linked lists have a few key points that usually make them very efficient for implementing. These are:

* the list is dynamic and hence can be resized based on the requirement
* Secondly, the insertion is O(1).

Types of Linked List

Following are the various types of linked list.

* **Simple Linked List** − Item navigation is forward only.
* **Doubly Linked List** − Items can be navigated forward and backward.
* **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

**Basic Operations**

Following are the basic operations supported by a list.

* **Insertion** − Adds an element at the beginning of the list.
* **Deletion** − Deletes an element at the beginning of the list.
* **Display** − Displays the complete list.
* **Search** − Searches an element using the given key.
* **Delete** − Deletes an element using the given key.

**Insertion Operation**

Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −

NewNode.next −> RightNode;

It should look like this −



Now, the next node at the left should point to the new node.

LeftNode.next −> NewNode;



This will put the new node in the middle of the two. The new list should look like this −



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

**Deletion Operation**

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



The left (previous) node of the target node now should point to the next node of the target node −

LeftNode.next −> TargetNode.next;



This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

TargetNode.next −> NULL;

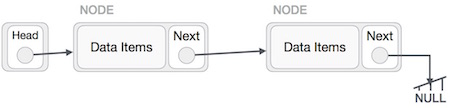


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

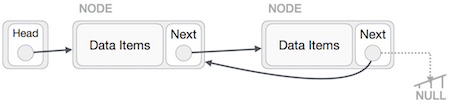


**Reverse Operation**

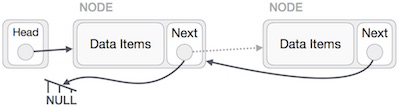
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



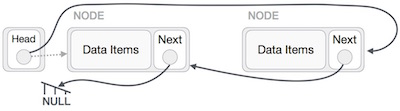
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.

# Singly Linked List

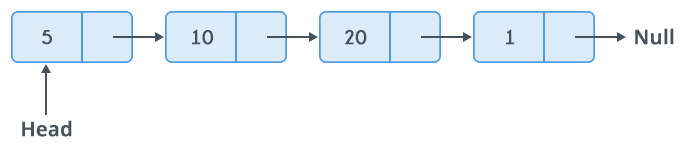
A **linked list** is a way to store a collection of elements. Like an array these can be character or integers. Each element in a linked list is stored in the form of a **node**.

**Node**:



A node is a collection of two sub-elements or parts. A **data** part that stores the element and a **next** part that stores the link to the next node.

**Linked List**:



A linked list is formed when many such nodes are linked together to form a chain. Each node points to the next node present in the order. The first node is always used as a reference to traverse the list and is called **HEAD**. The last node points to **NULL**.

**Declaring a Linked list** :

In C language, a linked list can be implemented using structure and pointers .

struct LinkedList{

int data;

struct LinkedList \*next;

};

The above definition is used to create every node in the list. The **data** field stores the element and the **next** is a pointer to store the address of the next node.

Noticed something unusual with next?

In place of a data type, **struct Linked List** is written before next. That's because it’s a **self-referencing pointer**. It means a pointer that points to whatever it is a part of. Here **next** is a part of a node and it will point to the next node.

**Creating a Node**:

Let's define a data type of struct LinkedListto make code cleaner.

typedef struct LinkedList \*node; //Define node as pointer of data type struct LinkedList

node createNode(){

node temp; // declare a node

temp = (node)malloc(sizeof(struct LinkedList)); // allocate memory using malloc()

temp->next = NULL;// make next point to NULL

return temp;//return the new node

}

**typedef** is used to define a data type in C.

**malloc()** is used to dynamically allocate a single block of memory in C, it is available in the header file stdlib.h.

**sizeof()** is used to determine size in bytes of an element in C. Here it is used to determine size of each node and sent as a parameter to malloc.

The above code will create a node with data as value and next pointing to NULL.

Let's see how to **add a node to the linked list**:

node addNode(node head, int value){

node temp,p;// declare two nodes temp and p

temp = createNode();//createNode will return a new node with data = value and next pointing to NULL.

temp->data = value; // add element's value to data part of node

if(head == NULL){

head = temp; //when linked list is empty

}

else{

p = head;//assign head to p

while(p->next != NULL){

p = p->next;//traverse the list until p is the last node.The last node always points to NULL.

}

p->next = temp;//Point the previous last node to the new node created.

}

return head;

}

Here the new node will always be added after the last node. This is known as **inserting a node at the rear end**.

**Food for thought**

This type of linked list is known as **simple or singly linked list**. A simple linked list can be traversed in only one direction from **head** to the last node.

The last node is checked by the condition :

p->next = NULL;

Here -> is used to access **next** sub element of node p. **NULL** denotes no node exists after the current node , i.e. its the end of the list.

**Traversing the list**:

The linked list can be traversed in a while loop by using the **head** node as a starting reference:

node p;

p = head;

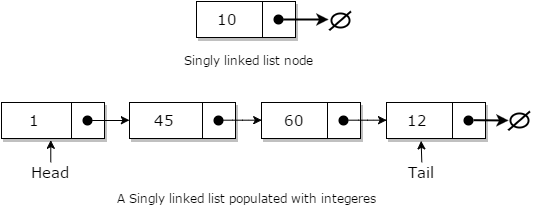
while(p != NULL){

p = p->next;

}

In general, when people talk about insertion concerning linked lists of any form, they implicitly refer to the adding of a node to the tail of the list.

Singly linked lists are one of the most primitive data structures you will learn in this tutorial. Here each node makes up a singly linked list and consists of a value and a reference to the next node (if any) in the list.



Adding a node to a singly linked list has only two cases:

* head = fi, in which case the node we are adding is now both the head and tail of the list; or
* we simply need to append our node onto the end of the list updating the tail reference appropriately

### Algorithm for inserting values to Linked List

* also Add(value)
* Pre: value is the value to be added to the list
* Post: value has been placed at the tail of the list
* n <- node(value)
* if head = fi
* head <- n
* tail<- n
* else
* Next <- n
* tail <- n
* end if
* end Add

### Example

//insert link at the first location

void insertFirst(int key, int data) {

//create a link

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

link->key = key;

link->data = data;

if(isEmpty()) {

//make it the last link

last = link;

} else {

//update first prev link

head->prev = link;

}

//point it to old first link

link->next = head;

//point first to new first link

head = link;

}

## Searching in Linked Lists

Searching a linked list is straightforward: we simply traverse the list checking the value we are looking for with the value of each node in the linked list.

* also Contains(head, value)
* Pre: the head is the head node in the list
* value is the value to search for
* Post: the item is either in the linked list, true; otherwise false
* n <- head
* While n 6= fi and n.Value 6= value
* n <- n.Next
* end while
* if n = fi
* return false
* end if
* return true
* end Contains

## Deletion in Linked Lists

Deleting a node from a linked list is straight-forward, but there are some cases that you need to account for:

* the list is empty; or
* the node to remove is the only node in the linked list; or
* we are removing the head node; or
* we are removing the tail node; or
* the node to remove is somewhere in between the head and tail; or
* the item to remove doesn't exist in the linked list

### Algorithm for Deletion

* algorithm Remove(head, value)
* Pre: the head is the head node in the list
* value is the value to remove from the list
* Post: value is removed from the list, true; otherwise false
* if head = fi
* return false
* end if
* n <- head
* If n.Value = value
* if the head = tail
* head <- fi
* tail <- fi
* else
* HHead <- head.Next
* end if
* return true
* end if
* while n.Next 6= fi and n.Next.Value 6= value
* n <- n.Next
* end while
* if n.Next 6= fi
* if n.Next = tail
* tail <- n
* end if
* // this is only case 5 if the conditional on line 25 was false
* Next <- n.Next.Next
* return true
* end if
* return false
* end Remove

### Example

//delete first item

struct node\* deleteFirst() {

//save reference to first link

struct node \*tempLink = head;

//if only one link

if(head->next == NULL) {

last = NULL;

} else {

head->next->prev = NULL;

}

head = head->next;

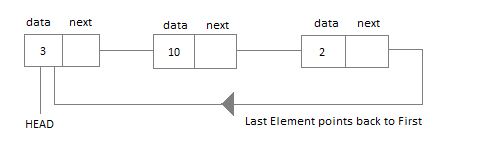
//return the deleted link

return tempLink;

}

## Circular Linked List

Circular Linked List is little more complicated linked data structure. In the circular linked list we can insert elements anywhere in the list whereas in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked list the previous element stores the address of the next element and the last element stores the address of the starting element. The elements points to each other in a circular way which forms a circular chain. The circular linked list has a dynamic size which means the memory can be allocated when it is required.



#### Application of Circular Linked List

* The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
* Another example can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.
* Circular Linked List can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.

### Implementing Circular Linked List

Implementing a circular linked list is very easy and almost similar to linear linked list implementation, with the only difference being that, in circular linked list the last **Node** will have it's **next** point to the **Head** of the List. In Linear linked list the last Node simply holds NULL in it's next pointer.

So this will be oue Node class, as we have already studied in the lesson, it will be used to form the List.

class Node {

public:

int data;

//pointer to the next node

node\* next;

node() {

data = 0;

next = NULL;

}

node(int x) {

data = x;

next = NULL;

}

}

Circular Linked List

Circular Linked List class will be almost same as the Linked List class that we studied in the previous lesson, with a few difference in the implementation of class methods.

class CircularLinkedList {

public:

node \*head;

//declaring the functions

//function to add Node at front

int addAtFront(node \*n);

//function to check whether Linked list is empty

int isEmpty();

//function to add Node at the End of list

int addAtEnd(node \*n);

//function to search a value

node\* search(int k);

//function to delete any Node

node\* deleteNode(int x);

CircularLinkedList() {

head = NULL;

}

}

#### Insertion at the Beginning

Steps to insert a Node at beginning :

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the fisrt Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

int CircularLinkedList :: addAtFront(node \*n) {

int i = 0;

/\* If the list is empty \*/

if(head == NULL) {

n->next = head;

//making the new Node as Head

head = n;

i++;

}

else {

n->next = head;

//get the Last Node and make its next point to new Node

Node\* last = getLastNode();

last->next = n;

//also make the head point to the new first Node

head = n;

i++;

}

//returning the position where Node is added

return i;

}

#### Insertion at the End

Steps to insert a Node at the end :

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, and make the next of the Newly added Node point to the Head of the List.

int CircularLinkedList :: addAtEnd(node \*n) {

//If list is empty

if(head == NULL) {

//making the new Node as Head

head = n;

//making the next pointer of the new Node as Null

n->next = NULL;

}

else {

//getting the last node

node \*last = getLastNode();

last->next = n;

//making the next pointer of new node point to head

n->next = head;

}

}

#### Searching for an Element in the List

In searhing we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the **data** of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

node\* CircularLinkedList :: search(int x) {

node \*ptr = head;

while(ptr != NULL && ptr->data != x) {

//until we reach the end or we find a Node with data x, we keep moving

ptr = ptr->next;

}

return ptr;

}

#### Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following :

* If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted. And update the next pointer of the Last Node as well.
* If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.
* If the Node is at the end, then remove it and make the new last node point to the head.

node\* CircularLinkedList :: deleteNode(int x) {

//searching the Node with data x

node \*n = search(x);

node \*ptr = head;

if(ptr == NULL) {

cout << "List is empty";

return NULL;

}

else if(ptr == n) {

ptr->next = n->next;

return n;

}

else {

while(ptr->next != n) {

ptr = ptr->next;

}

ptr->next = n->next;

return n;

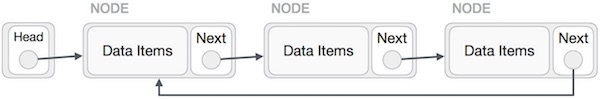
}

}

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

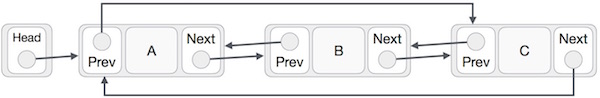
## Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



## Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



As per the above illustration, following are the important points to be considered.

* The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
* The first link's previous points to the last of the list in case of doubly linked list.

## Basic Operations

Following are the important operations supported by a circular list.

* **insert** − Inserts an element at the start of the list.
* **delete** − Deletes an element from the start of the list.
* **display** − Displays the list.

## Insertion Operation

Following code demonstrates the insertion operation in a circular linked list based on single linked list.

### Example

//insert link at the first location

void insertFirst(int key, int data) {

//create a link

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

link->key = key;

link->data= data;

if (isEmpty()) {

head = link;

head->next = head;

} else {

//point it to old first node

link->next = head;

//point first to new first node

head = link;

}

}

## Deletion Operation

Following code demonstrates the deletion operation in a circular linked list based on single linked list.

//delete first item

struct node \* deleteFirst() {

//save reference to first link

struct node \*tempLink = head;

if(head->next == head) {

head = NULL;

return tempLink;

}

//mark next to first link as first

head = head->next;

//return the deleted link

return tempLink;

}

## Display List Operation

Following code demonstrates the display list operation in a circular linked list.

//display the list

void printList() {

struct node \*ptr = head;

printf("\n[ ");

//start from the beginning

if(head != NULL) {

while(ptr->next != ptr) {

printf("(%d,%d) ",ptr->key,ptr->data);

ptr = ptr->next;

}

}

printf(" ]");

}

# Linked List vs Array

Both [Arrays](https://www.geeksforgeeks.org/array-data-structure/) and [Linked List](https://www.geeksforgeeks.org/data-structures/linked-list/)can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.





Key Differences Between Array and Linked List  
1. An array is the data structure that contains a collection of similar type data elements whereas the Linked list is considered as non-primitive data structure contains a collection of unordered linked elements known as nodes.  
2. In the array the elements belong to indexes, i.e., if you want to get into the fourth element you have to write the variable name with its index or location within the square bracket.  
3. In a linked list though, you have to start from the head and work your way through until you get to the fourth element.  
4. Accessing an element in an array is fast, while Linked list takes linear time, so it is quite a bit slower.  
5. Operations like insertion and deletion in arrays consume a lot of time. On the other hand, the performance of these operations in Linked lists is fast.  
6. Arrays are of fixed size. In contrast, Linked lists are dynamic and flexible and can expand and contract its size.  
7. In an array, memory is assigned during compile time while in a Linked list it is allocated during execution or runtime.  
9. Elements are stored consecutively in arrays whereas it is stored randomly in Linked lists.  
10. The requirement of memory is less due to actual data being stored within the index in the array. As against, there is a need for more memory in Linked Lists due to storage of additional next and previous referencing elements.  
11. In addition memory utilization is inefficient in the array. Conversely, memory utilization is efficient in the linked list.

Following are the points in favor of Linked Lists.

(1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, the upper limit is rarely reached.

(2) Inserting a new element in an array of elements is expensive because a room has to be created for the new elements and to create room existing elements have to be shifted.

For example, suppose we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040, …..].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

So Linked list provides the following two advantages over arrays  
1) Dynamic size  
2) Ease of insertion/deletion

Linked lists have following drawbacks:  
1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do a binary search with linked lists.  
2) Extra memory space for a pointer is required with each element of the list.  
3) Arrays have better cache locality that can make a pretty big difference in performance

* Linked List vs. Array
* Array is a datatype which is widely implemented as a default type, in almost all the modern programming languages, and is used to store data of similar type.
* But there are many usecases, like the one where we don't know the quantity of data to be stored, for which advanced data structures are required, and one such data structure is **linked list**.
* Let's understand how array is different from Linked list.

|  |  |
| --- | --- |
| **ARRAY** | **LINKED LIST** |
| Array is a collection of elements of similar data type. | Linked List is an ordered collection of elements of same type, which are connected to each other using pointers. |
| Array supports **Random Access**, which means elements can be accessed directly using their index, like arr[0] for 1st element, arr[6] for 7th element etc.  Hence, accessing elements in an array is **fast** with a constant time complexity of O(1). | Linked List supports **Sequential Access**, which means to access any element/node in a linked list, we have to sequentially traverse the complete linked list, upto that element.  To access **nth** element of a linked list, time complexity is O(n). |
| In an array, elements are stored in **contiguous memory location** or consecutive manner in the memory. | In a linked list, new elements can be stored anywhere in the memory.  Address of the memory location allocated to the new element is stored in the previous node of linked list, hence formaing a link between the two nodes/elements. |
| In array, **Insertion and Deletion** operation takes more time, as the memory locations are consecutive and fixed. | In case of linked list, a new element is stored at the first free and available memory location, with only a single overhead step of storing the address of memory location in the previous node of linked list.  Insertion and Deletion operations are **fast** in linked list. |
| Memory is allocated as soon as the array is declared, at **compile time**. It's also known as **Static Memory Allocation**. | Memory is allocated at **runtime**, as and when a new node is added. It's also known as **Dynamic Memory Allocation**. |
| In array, each element is independent and can be accessed using it's index value. | In case of a linked list, each node/element points to the next, previous, or maybe both nodes. |
| Array can **single dimensional**, **two dimensional** or **multidimensional** | Linked list can be **Linear(Singly)**, **Doubly** or **Circular** linked list. |
| Size of the array must be specified at time of array decalaration. | Size of a Linked list is variable. It grows at runtime, as more nodes are added to it. |
| Array gets memory allocated in the **Stack** section. | Whereas, linked list gets memory allocated in **Heap** section. |