# Linked List

Linked list is a data structure consisting of a group of memory space which together represents a list i.e. a sequence of data. Each data is stored in a separate memory space/block (called cell/node). Each memory block contains the data along with link/location/address to the memory location for the next data in the list. The linked list uses dynamic memory allocation, that is, it allocates memory for new list elements as needed.

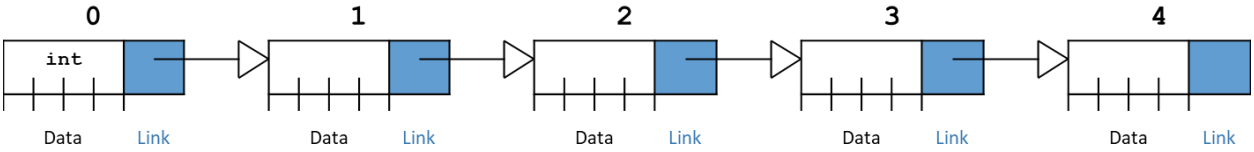


**Figure 6.1: Linked List**

Figure 6.1 shows a graphical depiction for a linked list storing five integers (Data). The value stored in a pointer variable (Link) is indicated by an arrow "pointing" to something.

## 1.1 Array and Linked List

A sequence of data can also be represented as an array. But in an array, data are stored consecutively in the memory. A linked list also represents and stores a sequence of data. But in a linked list the data may not be stored consecutively in the memory. Figure 6.2 shows a depiction of array and linked list representation. Same sequence of five values is stored in both an array and a linked list. In array, values are stored in continuous memory locations (FF00 to FF04) whereas in the linked list values are stored in random memory locations.

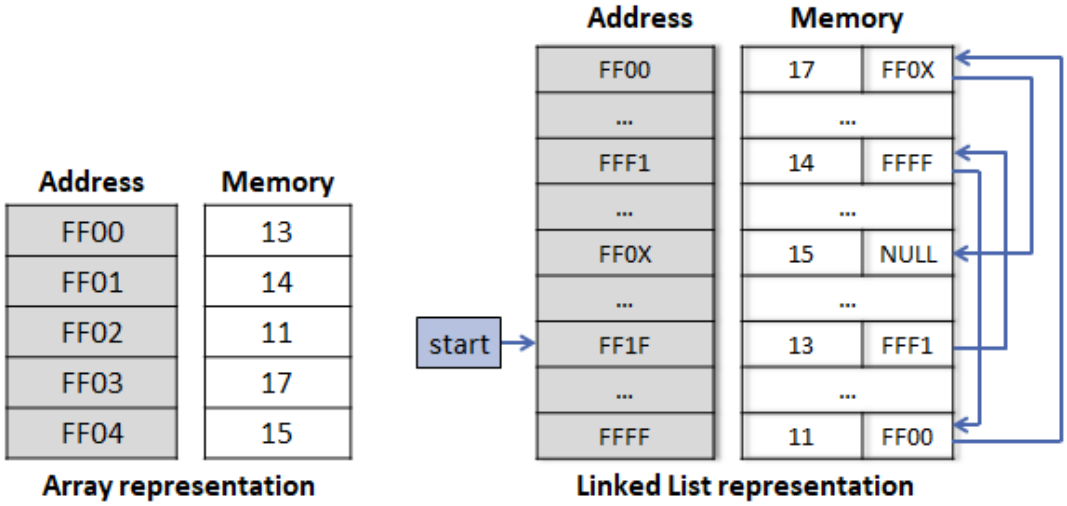


**Figure 6.2: Array and Linked List representation in memory**

When comparing array and linked list array allows direct access value at specific position whereas linked list requires sequential access. As an example, if we want to access the value17 from the linked list we also need to access all the previous values (ie. 13, 14 and 11).

## 1.2 Applications of Linked List in Computer Science

**Implementation of stacks and queues:** Stack and Queue both can be implemented using linked list. In fact the linked list implementation is more efficient in compare to array based implementation of Stack and Queue.

**Implementation of graphs:** Adjacency list representation of graphs is most popular which uses linked list to store adjacent vertices.

**Dynamic memory allocation:** We use linked list of free blocks which allows memory allocation on demand.

**Image viewer**: Previous and next images are linked, hence can be accessed by next and previous button.

**Previous and next page in web browser:** We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.

**Music Player:** Songs in music player are linked to previous and next song. You can play songs either from starting or ending of the list.

There are many other applications such as maintaining directory of names, performing arithmetic operations on long integers, manipulation of polynomials by storing constants in the node of linked list and representing sparse matrices etc.

## 1.3 Representation of a linked list node

A linked list node can be defined using structure. The structure is named like a normal variable which contains **data** and **link** to the next node (here it is named as next).

```
struct ListNode{
        int data;
        ListNode *next;
};
ListNode node;
```

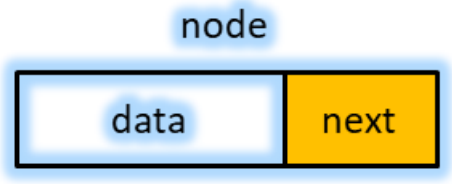


**Figure 6.3: Representation of a linked list node in C/C++**

Notice that node stores two types of information namely data and address to next node. In a linked list, there will be many nodes which will be connected thorough the addresses (links). A node variable can be created using the structure name. In Figure 6.3, **node** is a variable of type **ListNode**.

## 1.4 Linked List Traversal

Traversal is a basic operation on any data structure which requires accessing all data elements stored in the data structure. As we mentioned earlier in the chapter that linked list doesn't allow direct access though.

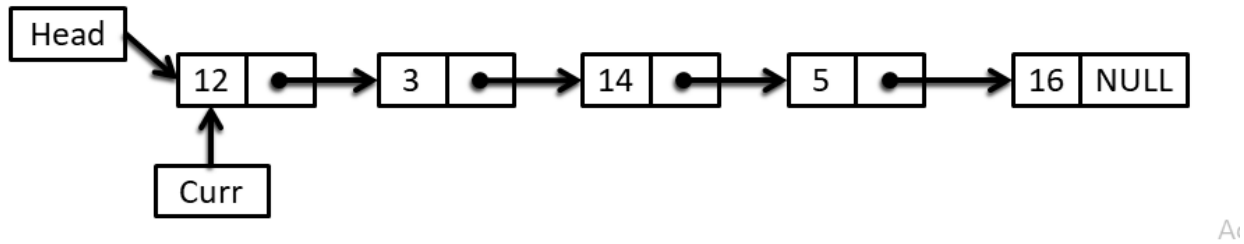


**Figure 6.4: Linked List Traversal**

Figure 6.4 shows a linked list which has 5 elements stored. Since linked list is always given with the address of first node we need to start from the first node only. A temporary pointer **Curr** is used to store address of visiting node. Initially the **Curr** will point to the first node (here the node containing 12). In the next step, we need to visit second node (the node containing 3). Since the first node (now pointed by **Curr**) contains the address of second node we can update or forward the **Curr** pointer to second node with the following operations:

**Curr = Curr →next**

Now **Curr** points to second node. In the same manner we can update **Curr** and visit to next nodes (third node, fourth node and fifth node). When **Curr** points to fifth node and we update it **Curr** will certainly become **NULL** which indicates the end of the linked list. The algorithm for linked list traversal is given as follows:

```
Algorithm (Linked List Traversal):
Input: Head (the address of first node)
Curr = Head
Step 1: if Curr == NULL exit otherwise access current node (with
address Curr)
Step 2: move Curr to next node and go to step 1
```

## 1.5 Searching in Linked List

Searching in linked list involves visiting each element starting from the first node and comparing each element with the searching element. Once the searching item is found the process stops otherwise keep visiting the next node. Basically, searching involves traversing the linked list partially or fully (in the worst case). The full list is visited if the searching item is absent in the list. The complexity of searching in linked list mostly like searching in an array.

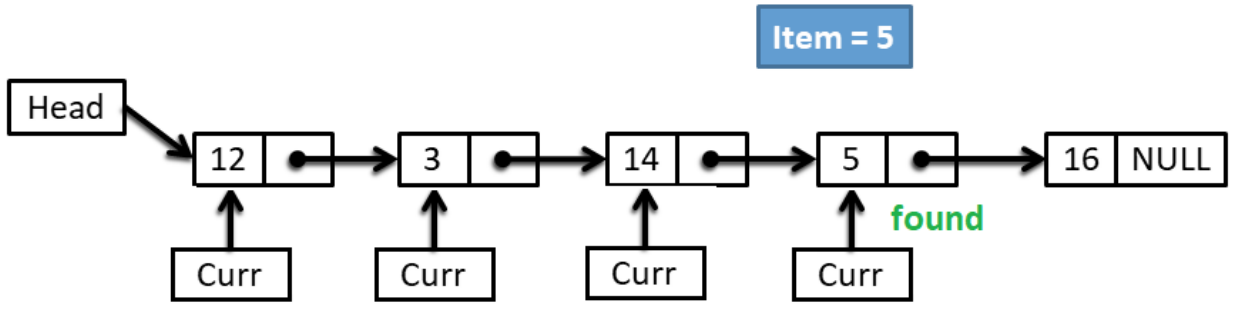


**Figure 6.5: Searching in a linked list**

Figure 6.5 represents an example of searching in a linked list. Like traversing a pointer Curr is assigned the address of first node (containing 12) and the data is compare with the searching element (5). If the data of node is a mismatch Curr is updated to point to the next node and the process goes on until 5 is found in 4th node.

```
Algorithm (Searching in linked list)
Input: Head (the address of first node)
Curr = Head
Step 1: if Curr == NULL print not found and exit
If Curr->data = item print found and exit
Step 2: move Curr to next node and go to step 1
```

## 1.6 Insertion in Linked List

Insertion in a linked list doesn't require any physical movements of data unlike insertion in an array. In a linked list, once the data is inserted the physical memory location is fixed. To insert a new data element, a new node is created in the memory and the data is placed inside the node. This doesn't complete the insertion process because the newly created node must be placed (logically) in the linked list. This process is done by updating the links (or we call it pointer, next etc.).

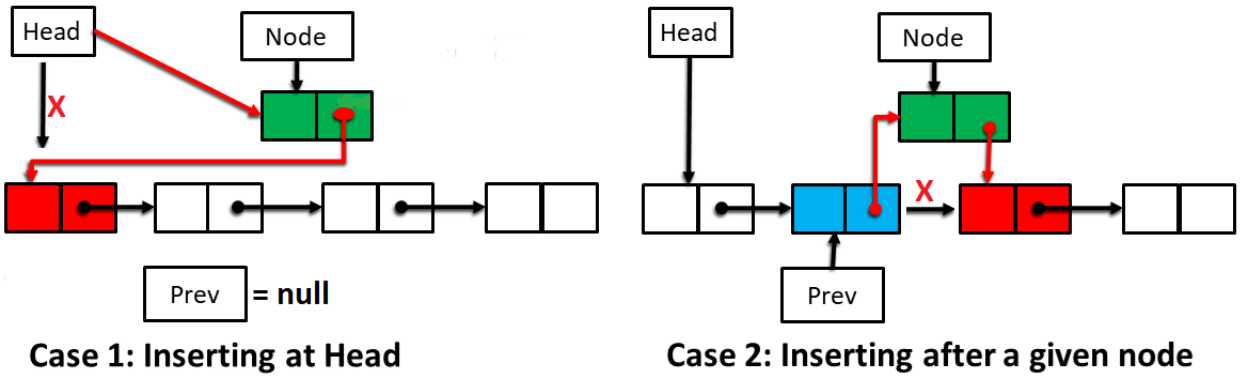


**Figure 6.6: Insertion in a linked list**

Figure 6.6 shows an illustrative example of insertion operation in a linked list. As already explained earlier in the section, a new node (colored green in the figure) is created which contains the inserting data. Now, there are two cases such as **case 1)** the new node is inserted as the first node and **case 2)** the new node is inserted after a given node (blue colored in the figure).

In case 1, at first we need to make a link (colored red) between the new node and the first node and then update Head by the address of newly created node. The link is established by replacing the value at Node→next by the address of the first node (colored red).

In case 2, we need to make two new links (colored red). At first make a link between the new node and the next node (colored red) then another link between the previous node and the new node. The first link is established by replacing the value at Node→next by the address of next node (stored in Prev→next). And the second link is established by replacing the value at Prev→next by the address of the new node.

```
Algorithm (insertion in linked list)
Input:
```

```
Head (the address of first node),
Node (inserting node),
Prev (address of previous node)

Case 1:
if Prev != NULL then go to Case 2
Make a link from Node to first node
Make Node as the Head
Exit
Case 2:
Make a link from Node to the node next to Prev
Make another link from Prev to Node
```

## 1.7 Deletion in Linked List

Deletion in the linked list involves updating links between nodes. Like insertion, there are two cases in deletion. Case 1 occurs when we want to delete the first node and Case 2 occurs when we want to delete any other node which has a previous node.

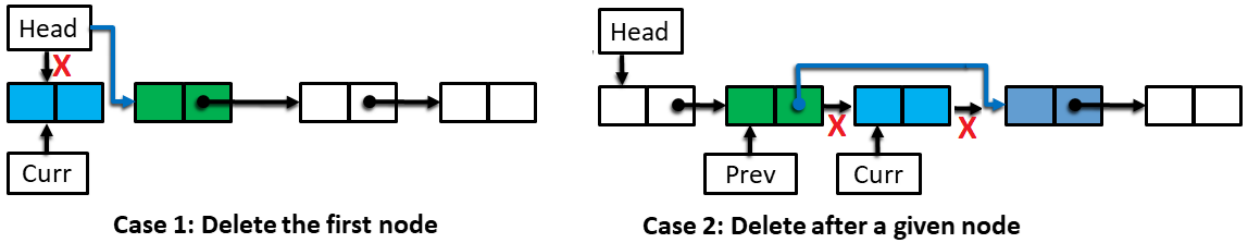


**Figure 6.7: Deletion in linked list**

Figure 6.7 illustrates the two cases of deletion. In the first case, the head pointer is updated by the address of second node which automatically discards the first node. In case 2, a node (pointed by Curr) is deleted by making another link between the previous node and the next node. This is done by simply replacing the value at Prev→next by the value at Curr→next (the address of next node).

```
Algorithm (deletion in linked list)
Input:
Head (the address of first node),
Prev (address of previous node)
Case 1:
if Prev = NULL then go to Case 2
Curr = Prev->next
Make a link from Prev to the node next to Curr
Exit
Case 2:
Make the node next to Head as new Head
```

## 1.8 Doubly Linked List

The linked list which has been so far discussed is called Singly Linked List. In Singly Linked List, a node contains data and the address of the next node which allows traversing in one direction starting from the first node to the last. Other way traversal is not possible. In contrast, a Doubly Linked List allows traversing to the both directions. Unlike a singly linked list, a doubly linked list contains data plus two pointers namely prev and next, which are addresses of the previous node and the next node respectively.

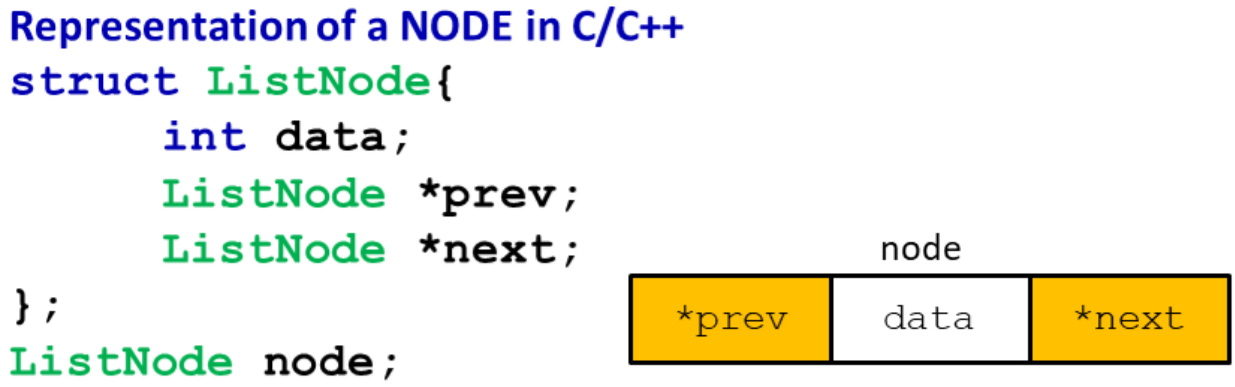


**Figure 6.8: Representation of Doubly Linked List node**

Figure 6.8 shows the representation of a node in doubly linked list. It is clear from the representation that from a given node traversing can be done in both directions.

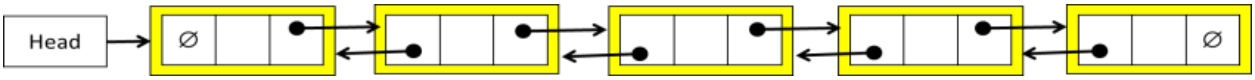


**Figure 6.9: A doubly linked list**

Figure 6.9 shows a representation of a doubly linked list. Note that prev of the first node is NULL and the next of the last node is NULL

## 1.9 References


- Kruse, R. L., Ryba, A. J., & Ryba, A. (1999). Data structures and program design in C++ (p. 280). New York: Prentice Hall.
- Shaffer, C. A. (2012). Data structures and algorithm analysis. Update, 3, 0-3.
- Dale, N. B. (2003). C++ plus data structures. Jones & Bartlett Learning.