Linked List Data Structure in C++ With Illustration

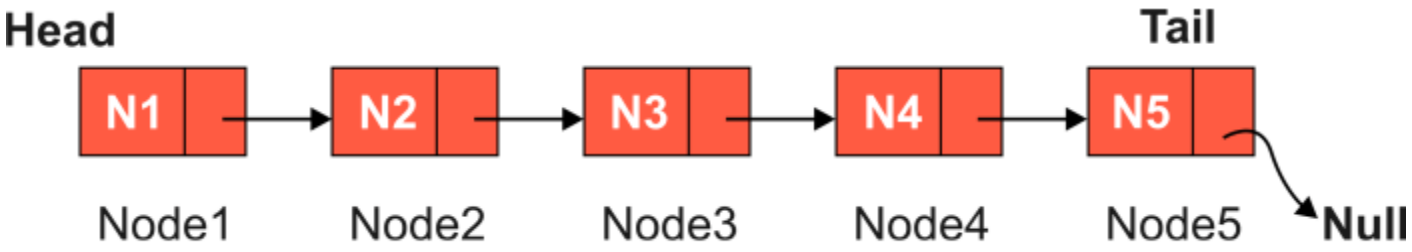
A **linked list** is a kind of linear dynamic data structure which we use to store data elements. Arrays are also a type of linear data structure where the data items are stored in continuous memory blocks.

Unlike arrays, the linked list does not need to store data elements in contiguous memory regions or blocks.

A **linked list** is composed of elements known as "Nodes" that are divided into two parts. The first component is the part where we store the actual data, and the second is a part where we store the pointer to the next node. This type of structure is known as a "**singly linked list**."

Linked List in C++

A singly linked list's structure is illustrated in the diagram below



1. As we have seen in the above part, the first node of the linked list is known as the "head," while the last node is called the "tail." It is because no memory address is specified in the last node, the final node of the linked list will have a null next pointer.
2. Because each node includes a pointer to the next node, data elements in the linked list do not need to be retained in contiguous locations. The nodes may be dispersed throughout memory. Because each node has the address of the one after it, we can access the nodes whenever we want.
3. We can quickly add and remove data items from the connected list. As a result, the linked list can increase or contract dynamically. The linked list has no maximum amount of data items it can contain. As a result, we can add as many data items as we like to the linked list as long as there is RAM available.
4. Because we don't have to specify how many items we need in the linked list in advance, the linked list saves memory space in addition to being simple to insert and delete. The only space used by a linked list is to store the pointer to the next node, which adds some cost.

Following that, we will go over the various operations that may be performed on a linked list.

1) Insertion

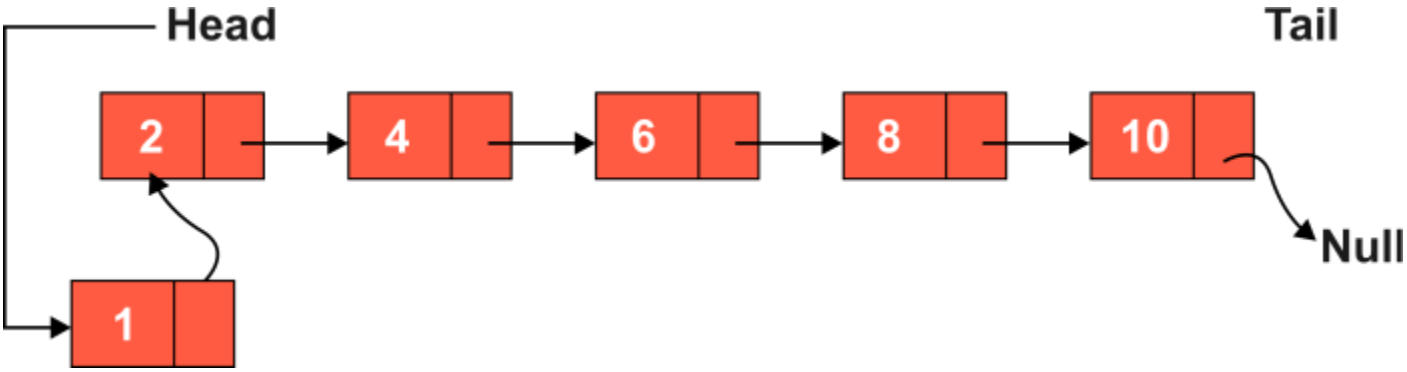
The linked list is expanded by the action of adding to it. Although it would seem simple, given the linked list's structure, we know that every time a data item is added, we must change the next pointers of the previous and next nodes of the new item that we have added.

Where the new data item will be inserted is the second aspect to think about.

There are three places where a data item can be added to the linked list.

a. Beginning with the linked list

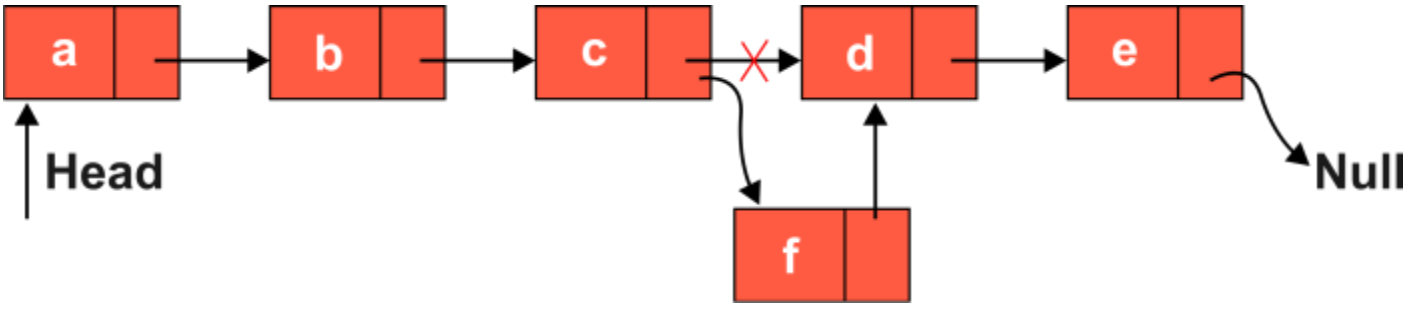
Below is a connected list of the numbers 2->4->6->8->10. The head pointing to node 2 will now point to node 1, and the next pointer of node 1 will have the memory address of node 2, as shown in the illustration below, if we add a new node 1 as the first node in the list.



As a result, the new linked list is 1->2->4->6->8->10.

b. After the given Node

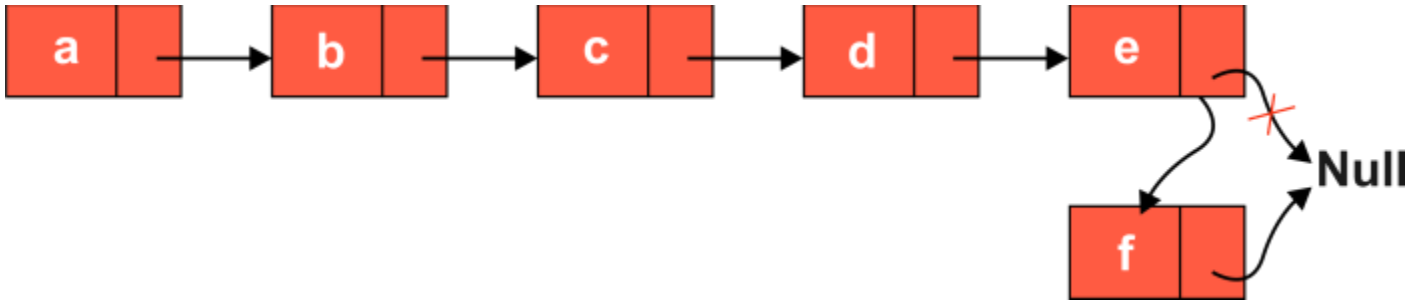
In this case, we are given a node and must add a new node behind it. The linked list will seem as follows if node f is added to the linked list a->b->c->d->e after node c:



We therefore check to see if the specified node is present in the diagram above. If it is present, a new node f is created. After that, we point node c's next pointer at the brand-new node f. The node f's next pointer now points to node d.

c. The Linked List's last item

In the third case, a new node is added to the end of the linked list. Take into account the linked list below: a->b->c->d->e, with the addition of node f at the end. After adding the node, the linked list will appear like this.



As a result, we construct a new node f. The tail pointer leading to null is then pointed to f, and node f's next pointer is pointed to null. In the Programming language below, we have generated all three types of insert functions.

A linked list can be declared as a structure or as a class in C++. A linked list declared as a structure is a classic C-style statement. A linked list is used as a class in modern C++, mainly when using the standard template library.

Structure was used in the following application to declare and generate a linked list. Its members will be data and a pointer to the following element.

C++ Program:

```
#include <iostream>
using namespace std;

struct Node
{
  int data;
  struct Node *next;
};
void push ( struct Node** head, int nodeData )
```

```
{
  struct Node* newNode1 = new Node;

  newNode1 -> data = nodeData;
  newNode1 -> next = (*head);

  (*head) = newNode1;
}
void insertAfter ( struct Node* prevNode, int nodeData )
{
if ( prevNode == NULL )
{
  cout << "the given previous node is required,cannot be NULL";
  return;

}
  struct Node* newNode1 =new Node;
  newNode1 -> data = nodeData;
  newNode1 -> next = prevNode -> next;
   prevNode -> next = newNode1;
}
void append ( struct Node** head, int nodeData )
{
struct Node* newNode1 = new Node;

struct Node *last = *head;
newNode1 -> data = nodeData;
newNode1 -> next = NULL;
if ( *head == NULL )
{
*head = newNode1;
return;
}
while ( last -> next != NULL )
last = last -> next;
last -> next = newNode1;
return;
}
```

```
void displayList ( struct Node *node )
{
  while ( node != NULL )
  {
    cout << node -> data << "-->";
    node = node -> next;
  }

if ( node== NULL)
cout<<"null";
}
int main ()
{
struct Node* head = NULL;
append ( &head, 15 );
push ( &head, 25 );
push ( &head, 35 );
append ( &head, 45 );
insertAfter ( head -> next, 55 );

cout << "Final linked list: " << endl;
displayList (head);

return 0;
}
```

Output:

```
Final linked list:
35-->25-->55-->15-->45-->null
```

2) Deletion

Similar to insertion, deleting a node from a linked list requires many points from which the node might be eliminated. We can remove the linked list's first, last, or kth node at random. We must correctly update the next pointer and all other linked list pointers in order to maintain the linked list after deletion.

In the following C++ implementation, we have two deletion methods: deleting the list's initial node and deleting the list's last node. We begin by adding nodes to the head of the list. The list's contents are then shown following each addition and deletion.

C++ Program:

```
#include <iostream>
using namespace std;
struct Node {
int data;
struct Node* next;
};
Node* deletingFirstNode ( struct Node* head )
{
if ( head == NULL )
return NULL;
Node* tempNode = head;
head = head -> next;
delete tempNode;

return head;
}
Node* removingLastNode ( st

ruct Node* head )
{
if ( head == NULL )
return NULL;

if ( head -> next == NULL ) {
delete head;
return NULL;
}
Node* secondLast = head;
while ( secondLast -> next -> next != NULL )
secondLast = secondLast->next;
delete ( secondLast -> next );
secondLast -> next = NULL;

return head;
```

```
}
void push ( struct Node** head, int newData )
{
struct Node* newNode1 = new Node;
newNode1 -> data = newData;
newNode1 -> next = ( *head );
( *head ) = newNode1;
}
int main()
{
Node* head = NULL;
push ( &head, 25 );
push ( &head, 45 );
push ( &head, 65);
push ( &head, 85 );
push ( &head, 95 );

Node* temp;

cout << "Linked list created " << endl; for ( temp = head; temp != NULL; temp = temp -> next )
cout << temp->data << "-->";
if ( temp == NULL )
cout << "NULL" << endl;
head = deletingFirstNode (head);
cout << "Linked list after deleting head node" << endl; for ( temp = head; temp != NULL; temp = temp -
> next )
cout << temp->data << "-->";
if ( temp == NULL )
cout<<"NULL"<<endl;
head = removingLastNode (head);
cout << "Linked list after deleting last node" << endl; for ( temp = head; temp != NULL; temp = temp -
> next )
cout << temp -> data << "-->";
if ( temp == NULL )
cout << "NULL";

return 0;
```

}

Output:

```
Linked list created
95-->85-->65-->45-->25-->NULL
Linked list after deleting head node
85-->65-->45-->25-->NULL
Linked list after deleting last node
85-->65-->45-->NULL
```

Node Count

While traversing the linked list, the process of counting the number of nodes can be performed. In the preceding approach, we saw that if we needed to insert/delete a node or display the contents of the linked list, we had to traverse the linked list from the beginning.

Setting a counter and incrementing as well as we traverse each node will provide us the number of nodes in the linked list.

Differences between Array and Linked list:

Array	Linked list
Arrays have a defined size.	The size of the linked list is variable.
Inserting a new element is difficult.	Insertion and deletion are simpler.
Access is permitted at random.	No random access is possible.
Elements are in relatively close or contiguous.	The elements are not contiguous.
No additional room is required for the following pointer.	The following pointer requires additional memory.

Functionality

Since linked lists and arrays are both linear data structures that hold objects, they can be utilised in similar ways for the majority of applications.

The following are some examples of linked list applications:

- Stacks and queues can be implemented using linked lists.
- When we need to express graphs as adjacency lists, we can use a linked list to implement them.
- We can also use a linked list to contain a mathematical polynomial.

- In the case of hashing, linked lists are employed to implement the buckets.
- When a programmer requires dynamic memory allocation, we can utilize a linked list because linked lists are more efficient in this instance.

Conclusion

Linked lists are data structures used to hold data elements in a linear but non-contiguous form. A linked list is made up of nodes with two components each. The first component is made up of data, while the second half has a pointer that stores the memory address of the following member of the list.

As a sign that the linked list has ended, the last item in the list has its next pointer set to NULL. The Head is the first item on the list. The linked list allows for a variety of actions such as insertion, deletion, traversal, and so on. Linked lists are favoured over arrays for dynamic memory allocation.

Linked lists are hard to print or traverse because we can't access the elements randomly like arrays. When compared to arrays, insertion-deletion procedures are less expensive.

In this tutorial, we learned everything there is to know about linear linked lists. Linked lists can also be doubly linked or circular. In our forthcoming tutorials, we will go through these lists in detail.