A 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.
* **LinkedList** − 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.

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.

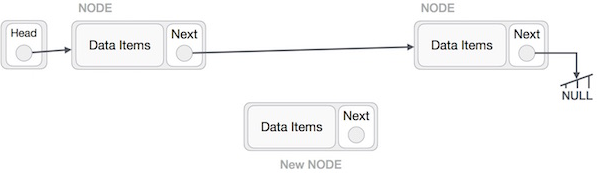
Basic Operations

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

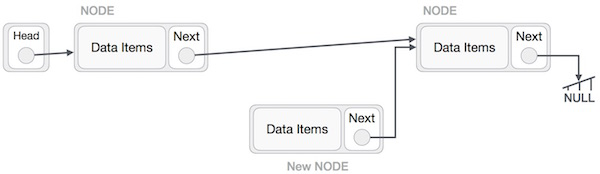
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



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

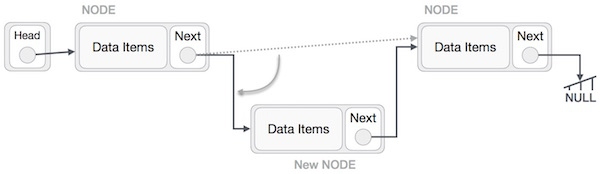
NewNode.next −> RightNode;

It should look like this −

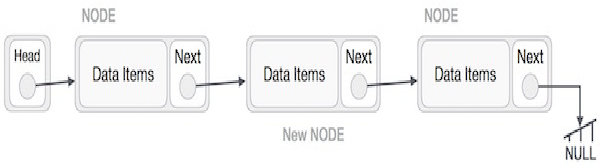


Now, the next node at the left should point to the new node.

LeftNode.next −> NewNode;



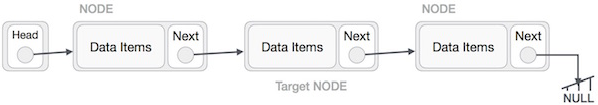
This will put the new node in the middle of the two. The new list should look like this −



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 −x

LeftNode.next −> TargetNode.next;

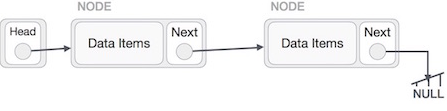


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

TargetNode.next −> NULL;

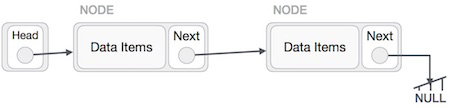


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

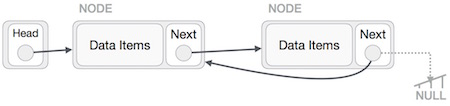


Reverse Operation

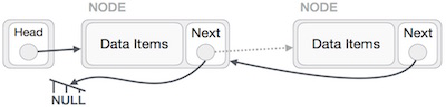
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



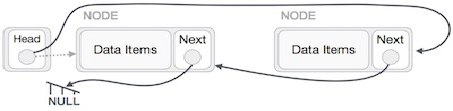
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



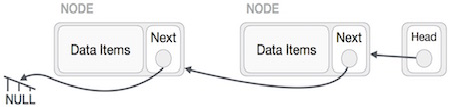
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



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.



**Introduction to linked list data structure**

A linked list is a data structure that consists of sequence of nodes. Each node is composed of two fields: **data field** and **reference field** which is a [pointer](https://zentut.com/c-tutorial/c-pointer/)that points to the next node in the sequence.

*Linked List Node*

Each node in the list is also called an element. The reference field that contains a pointer which points to the next node is called **next pointer** or **next link**.

A **head** pointer is used to track the first element in the linked list, therefore, it always points to the first element.

The following picture illustrates a linked list

The linked list data structure is designed to be efficient for insertion or removal of elements from any position in the list. However other operations such as getting the last element or finding an element that stores specific data requires scanning most or all the elements in the list.

A linked list is also used to implement other data structure such as [stack](https://zentut.com/c-tutorial/c-stack-using-pointers/) and [queue](https://zentut.com/c-tutorial/c-queue/).

C Linked List implementation

We can model a node of the linked list using a [structure](https://zentut.com/c-tutorial/c-structure/)as follows:

|  |  |
| --- | --- |
|  | typedef struct node{      int data;      struct node\* next;  } |

The node structure has two members:

* data stores the information
* next pointer holds the address of the next node.

**Add a node at the beginning of the linked list**

First, we declare a head pointer that always points to the first node of the list.

|  |  |
| --- | --- |
| 1 | node\* head; |

To add a node at the beginning of the list:

First, we need to create a new node. We will need to create a new node each time we want to insert a new node into the list so we can develop a function that creates a new node and return it.

|  |  |
| --- | --- |
|  | node\* create(int data,node\* next)  {      node\* new\_node = (node\*)malloc(sizeof(node));      if(new\_node == NULL)      {          printf("Error creating a new node.\n");          exit(0);      }      new\_node->data = data;      new\_node->next = next;      return new\_node;  } |

Second, we need to point the next pointer of the new node to the head pointer and point he headpointer to the new node. It works for both empty and non-empty linked list.

|  |  |
| --- | --- |
|  | node\* prepend(node\* head,int data)  {      node\* new\_node = create(data,head);      head = new\_node;      return head;  } |

The following picture illustrates how to insert a node when the list is empty:

*Add a node into an empty linked list*

The following picture illustrates how to insert a node at the beginning of a non-empty linked list:

*Add a new node at the beginning of a non-empty linked list*

**Traverse the linked list**

Sometimes we may want to traverse the linked list to get the data stored in each node for further manipulation e.g., display the node information. To traverse the linked list, we start from the first node, and move to the next node until we reach a NULL pointer.

To make the traverse() function more general, we can add a [function pointer](https://zentut.com/c-tutorial/c-function-pointer/) that points to a function for linked list node manipulation as a parameter of the traverse() function.

We define a call back function for manipulating a node of the linked list:

|  |  |
| --- | --- |
| 1 | typedef void (\*callback)(node\* data); |

The following is the traverse() function:

|  |  |
| --- | --- |
|  | void traverse(node\* head,callback f)  {      node\* cursor = head;      while(cursor != NULL)      {          f(cursor);          cursor = cursor->next;      }  } |

**Count the elements of the linked list**

We can use the same traversing technique to count the number of elements in a linked list. See the following count() function:

|  |  |
| --- | --- |
|  | int count(node \*head)  {      node \*cursor = head;      int c = 0;      while(cursor != NULL)      {          c++;          cursor = cursor->next;      }      return c;  } |

**Add a new node at the end of the linked list**

To add a new node at the end of the linked list:

First, we need to find the last node of the list. Starting the first node which is indicated by the headpointer, we traverse the list until the next pointer reaches NULL.

|  |  |
| --- | --- |
|  | node \*cursor = head;  while(cursor->next != NULL)     cursor = cursor->next; |

Second, we create a new node with the next pointer points to NULL , and the next pointer of the cursor points to the new node:

|  |  |
| --- | --- |
|  | node\* new\_node =  create(data,NULL);  cursor->next = new\_node; |
|  |  |

***Add a new node at the end of the linked list***

The following is the append() function that adds a new node at the end of the linked list.

|  |  |
| --- | --- |
|  | node\* append(node\* head, int data)  {      /\* go to the last node \*/      node \*cursor = head;      while(cursor->next != NULL)          cursor = cursor->next;      /\* create a new node \*/      node\* new\_node =  create(data,NULL);      cursor->next = new\_node;      return head;  } |

**Insert a new node after a particular node**

To insert a new node after a particular node, we need to:

First, verify if the node exists in the list, we call this node is prev node

|  |  |
| --- | --- |
|  | node \*cursor = head;  while(cursor != prev)     cursor = cursor->next; |

If it exists, we point the next pointer of the new node to the next node that the next pointer of the prev node points to, and point the next pointer of the prev node to the new node.

|  |  |
| --- | --- |
|  | node\* new\_node = create(data,cursor->next);  cursor->next = new\_node; |

The following picture illustrates how to insert a new node after a particular node in the linked list:

*Insert a new after a particular node in the linked list*

The insert\_after() function is as follows:

|  |  |
| --- | --- |
|  | /\*      insert a new node after the prev node  \*/  node\* insert\_after(node \*head, int data, node\* prev)  {      /\* find the prev node, starting from the first node\*/      node \*cursor = head;      while(cursor != prev)          cursor = cursor->next;      if(cursor != NULL)      {          node\* new\_node = create(data,cursor->next);          cursor->next = new\_node;          return head;      }      else      {          return NULL;      }  } |

**Insert a new node before a particular node**

To insert a new node before a particular node named nxt we need to:

* First, if the nxt node is the first node, we can call the prepend() function to add a new node at the beginning of the list, otherwise, we find the previous node of the nxt node, suppose it is the cursor node.
* Second, point the next pointer of the new node to the node that the next pointer of the cursor points to and point the next pointer of the cursor to the new node.

The following picture illustrates how to insert a node before a particular node:

*Insert a node before a particular node*

The following is the insert\_before() function that inserts a new node before a particular node in the linked list:

|  |  |
| --- | --- |
|  | node\* insert\_before(node \*head, int data, node\* nxt)  {      if(nxt == NULL || head == NULL)          return NULL;      if(head == nxt)      {          head = prepend(head,data);          return head;      }      /\* find the prev node, starting from the first node\*/      node \*cursor = head;      while(cursor != NULL)      {          if(cursor->next == nxt)              break;          cursor = cursor->next;      }      if(cursor != NULL)      {          node\* new\_node = create(data,cursor->next);          cursor->next = new\_node;          return head;      }      else      {          return NULL;      }  } |

**Search for a node**

To search for a node that stores a given data, we scan the whole list and return the first node that stores the searched data. The following illustrates the search function:

|  |  |
| --- | --- |
|  | node\* search(node\* head,int data)  {      node \*cursor = head;      while(cursor!=NULL)      {          if(cursor->data == data)              return cursor;          cursor = cursor->next;      }      return NULL;  } |

The search() function returns NULL if no node stores the input data.

**Sort a linked list using insertion sort**

We can sort a linked list using the [insertion sort algorithm](https://zentut.com/c-tutorial/insertion-sort-in-c/). The following is the insertion\_sort()function that sorts a linked list:

|  |  |
| --- | --- |
|  | node\* insertion\_sort(node\* head)  {      node \*x, \*y, \*e;      x = head;      head = NULL;      while(x != NULL)      {          e = x;          x = x->next;          if (head != NULL)          {              if(e->data > head->data)              {                  y = head;                  while ((y->next != NULL) && (e->data> y->next->data))                  {                      y = y->next;                  }                  e->next = y->next;                  y->next = e;              }              else              {                  e->next = head;                  head = e ;              }          }          else          {              e->next = NULL;              head = e ;          }      }      return head;  } |

**Reverse linked list**

To reverse a linked list, you change the next pointer of each node from the next node to the previous node. The following reverse() function reverses a linked list.

|  |  |
| --- | --- |
|  | node\* reverse(node\* head)  {      node\* prev    = NULL;      node\* current = head;      node\* next;      while (current != NULL)      {          next  = current->next;          current->next = prev;          prev = current;          current = next;      }      head = prev;      return head;  } |

**Delete a node from the front of the linked list**

To delete a node from the front of the linked list is relatively simple. We point the head to the next node and remove the node that the head pointed to.

Notice that if the list has only one node, we should set the head pointer to NULL.

*Remove from the front of the list*

The following is the remove\_front() function:

|  |  |
| --- | --- |
|  | node\* remove\_front(node\* head)  {      if(head == NULL)          return NULL;      node \*front = head;      head = head->next;      front->next = NULL;      /\* is this the last node in the list \*/      if(front == head)          head = NULL;      free(front);      return head;  } |

**Delete a node from the back of the linked list**

To remove a node from the back of the linked list, we need to:

* Use two pointers: cursor and back to track the node.
* Start from the first node until the cursor pointer reaches the last node and the back pointer reaches the node before the last node.
* Set the next pointer of the back to NULL and delete the node that the cursor points to.
* If the node has only 1 element, set the head pointer to NULL before removing the node.

*Delete a node from the back of the linked list*

The following is the function to remove a node from the back of the linked list:

|  |  |
| --- | --- |
|  | node\* remove\_back(node\* head)  {      if(head == NULL)          return NULL;      node \*cursor = head;      node \*back = NULL;      while(cursor->next != NULL)      {          back = cursor;          cursor = cursor->next;      }      if(back != NULL)          back->next = NULL;      /\* if this is the last node in the list\*/      if(cursor == head)          head = NULL;      free(cursor);      return head;  } |

**Delete a node in the middle of the linked list**

To delete a node in the middle of the linked list:  
If the node is the first node, call remove\_front() function to remove it.  
If the node is the last node, call remove\_back() function to remove it.  
If the node is in the middle of the list:

* Traverse from the first node, use the cursor pointer to point to the node before the node that needs to be removed e.g., in the picture, the node that needs to be removed is 2 and the cursor pointer points to node 5.
* Use a tmp pointer to point to the node that needs to be removed.
* Set the next pointer of the cursor point to the node that the next pointer of the tmp points to.
* Remove the node that the tmp pointer points to.

The following picture illustrates how to delete a node in the middle of the linked list:

*Remove the middle node of a linked list*

The following is the function that can be used to remove any node in the linked list:

|  |  |
| --- | --- |
|  | node\* remove\_any(node\* head,node\* nd)  {      /\* if the node is the first node \*/      if(nd == head)      {          head = remove\_front(head);          return head;      }      /\* if the node is the last node \*/      if(nd->next == NULL)      {          head = remove\_back(head);          return head;      }      /\* if the node is in the middle \*/      node\* cursor = head;      while(cursor != NULL)      {          if(cursor->next = nd)              break;          cursor = cursor->next;      }      if(cursor != NULL)      {          node\* tmp = cursor->next;          cursor->next = tmp->next;          tmp->next = NULL;          free(tmp);      }      return head;  } |

**Delete the whole linked list**

It is important to remove all nodes of the linked list when you no longer use it. The following dispose() function releases memory allocated for all the nodes in a linked list:

|  |  |
| --- | --- |
|  | void dispose(node \*head)  {      node \*cursor, \*tmp;        if(head != NULL)      {          cursor = head->next;          head->next = NULL;          while(cursor != NULL)          {              tmp = cursor->next;              free(cursor);              cursor = tmp;          }      }  } |