-: Linked List :-

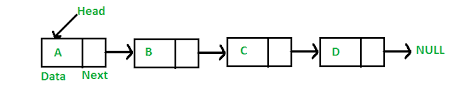
It is a dynamic collection type data storing environment. Here nodes are the building block material of linked list.

**Nodes:** These are the elements of the linked list. It consist two part one data part and one address part which is referring to the next node (next element). The last node after which there is no node present contains **null** in the address part because it is not pointing to any other node. The entry point into a linked list is called the **head** of the list.

We can create a node by using an-user defined data type i.e. structure. Here we can put the data and point the similar type of node by using self-referential structure

Structure of a Node:

|  |  |
| --- | --- |
| Data | Address of next node |



Types of linked list:

There are three types of linked list.

1. Single Linked List
2. Double Linked List
3. Circular Linked List

Difference between these three type of linked list :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Single Linked List**   1. It has two field in the node. One is data part another is link part.  |  |  | | --- | --- | | Data | Address of next node |  1. Link part of node pointing to the next node. | Double Linked List   1. It has three field in the node one data part and two link part.   C:\Users\Lenovo\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\B5BCD363.tmp   1. One link part is pointing to the previous node and another is pointing to the next node. | Circular Linked List   1. It has three field in the node one data part and two link part.   Doubly Circular Linked List | Set 1 (Introduction and Insertion) -  GeeksforGeeks   1. One link part is pointing to the previous node and another is pointing to the next node. |

**Single Linked List: -**

In single linked list the node has two fields and we are creating the node by the help of a user defined data type i.e. structure. The memory allocation is done here by dynamic memory allocation. By the help of **malloc** which returns the address of first node and that address is stored into a pointer variable.

Ex:

struct node {

int data;

struct node \*link;

};

struct node \*root;

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

Explanation: We created a structure which acts as a node in the inked list. It has two field one is data part of integer type which can hold any integer value and another is link part. This part is important and pointing to the node of same type. So it is named as **self-referential structure**.

root: is a pointer variable which can points to the struct node type. So it is used to point the first node in the list that is why memory is allocated to the root variable dynamically.

NOTE: When the list is empty i.e. not having any node then the root is assigned as NULL.

**Operation performed on single linked list: -**

1. Append the list (Adding new nodes)
2. Append at beginning
3. Append at the end
4. Append at a particular place
5. Delete the nodes
6. Delete the first node
7. Delete the last node
8. Delete any specified node
9. Length of the list
10. Reverse the list
11. Swap between two nodes
12. Sort the elements in the list
13. Searching an element in the ist

**Append the list:-**  Here we have to add elements into the list. First we have to create a linked list and have to initialize an variable which can points to the first variable. Then only we go for adding elements.

Code:-

struct node{

int data;

struct node \*link;

} ;

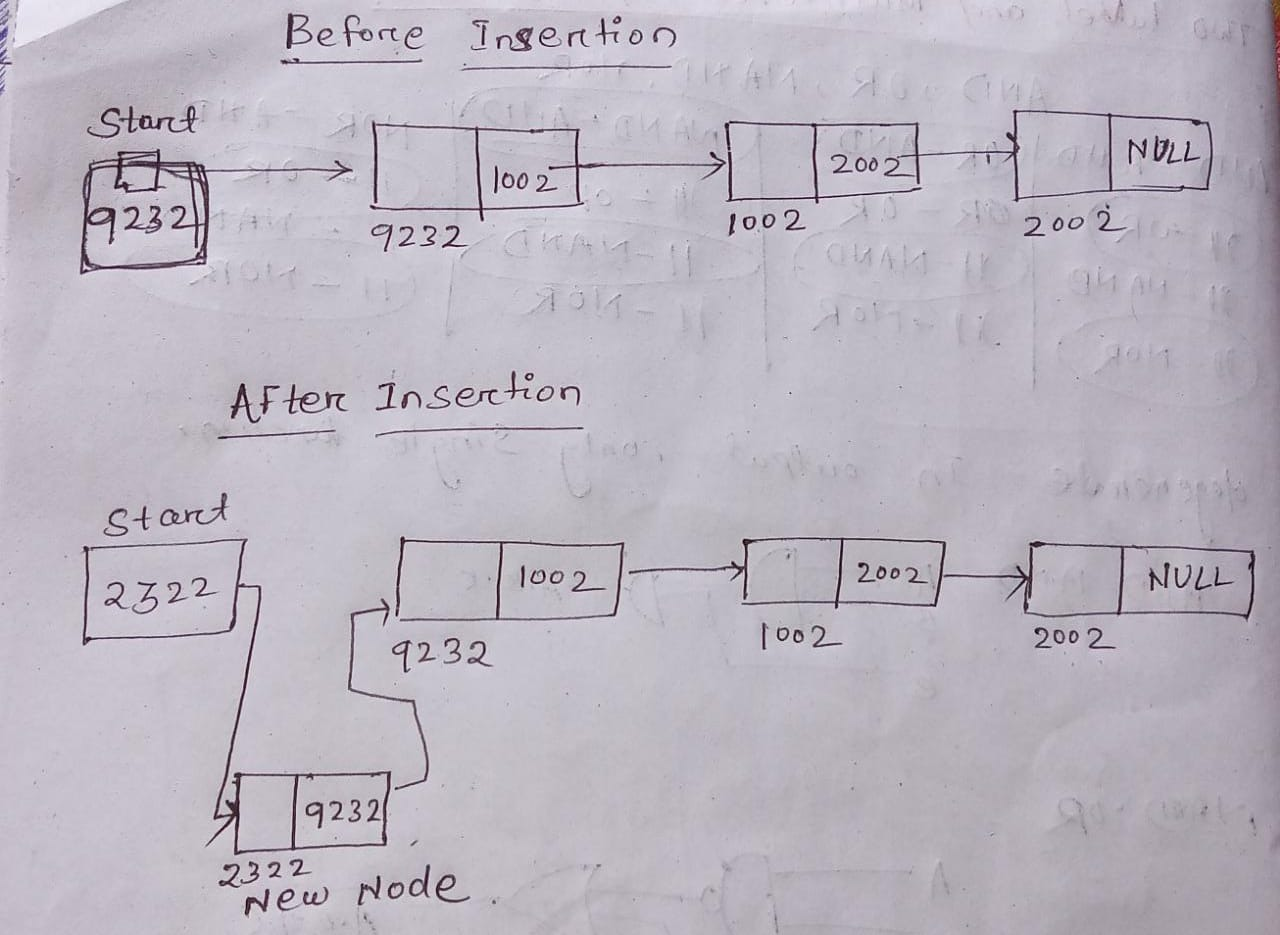
struct node \*start;

// this structure and variable are global variable.

Then we have different case to insert element in different positions.

1. **Insertion of node at the beginning:**

Before insertion we have a linked list of particular number of nodes and start is pointing to the first node in the list. After insertion the start should point to the inserted new node and link part of new node should point to the first node of old list.



**Algorithm:**

Here we have to take an temporary node which is of structure node type and memory allocated dynamically to it. Then we have to input the data part by temp->data and it should points to the second element in the list. So this can be done by temp->link = start; this is because at this stage start is pointing to the second. Then we should make start to the new node by start = temp;

**Code:**

void addBeginning (struct node \*start , int value )

{

struct node \*temp;

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

temp->data = value;

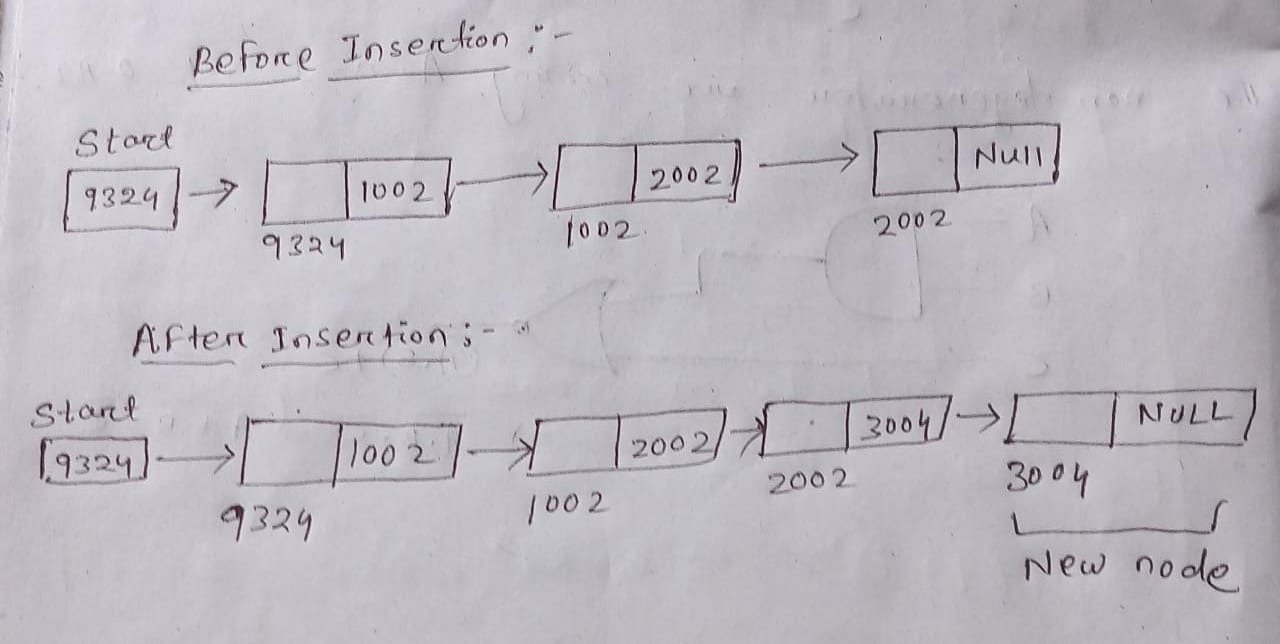
temp->link = start;

start = temp;

}

1. **Append at the end:**

Insertion at the end means we have a list with some elements. Before insertion the last element contains null as link part. After insertion the new element will be the last element and the previous element should point to the last element.



**Algorithm:**

* First we have to take two pointer variable i.e. temp and p. temp will hold the address of new node and p will go with while loop an at last it points to the last node.
* Then we input the value into the temp->data. And the penultimate node will point to the last node by p->link = temp .
* After that temp->link will hold the null.

**Code:**

void addend(int value)

{

Struct node \*temp,\*p;

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

temp->data = value;

p=start;

while(p->link != NULL){

p = p->link;

}

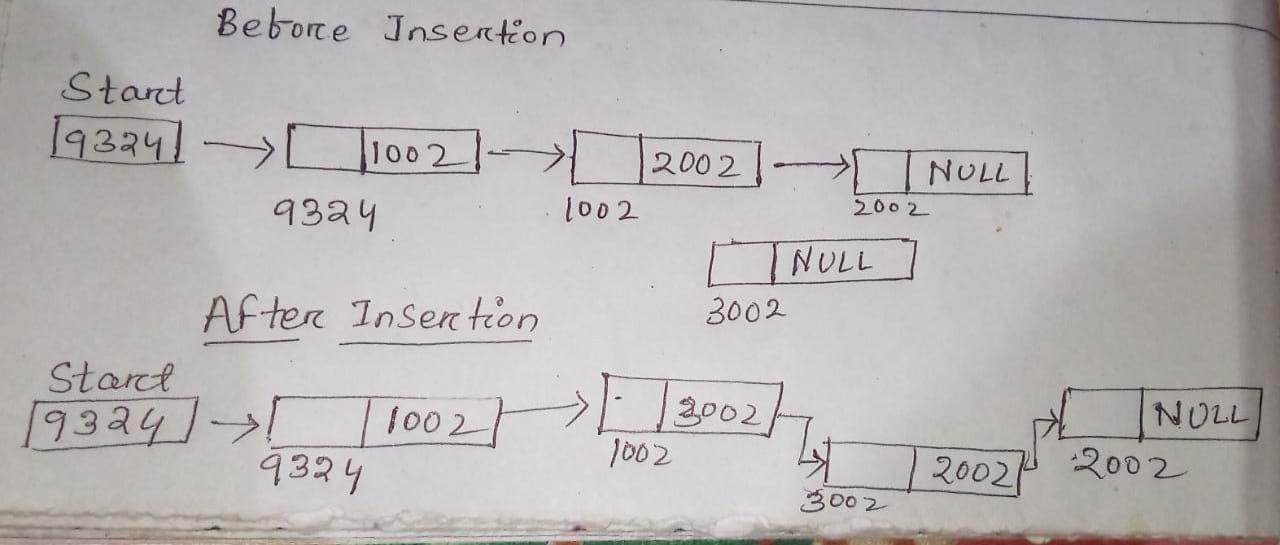
p->link = temp;

temp->link = NULL;

}

1. **Append between two nodes:**

Here we have to insert a node in between the two nodes.



**Algorithm:**

**Code:**

void addmiddle () {

struct node \*temp, \*p;

int loc,i=1;

if(loc >= length()){

printf(“Invalid input! ”); }

else{

p=start;

while(i < loc){

p = p->link;

i++; }

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

printf(“Enter the data part: ”);

scanf(“%d”,&temp->link);

temp->link = p->link;

p->link = temp;

}

}

**Length of the list:**

Length is the number of nodes present in a list. We can get the length of the list by traversing the list.

int length () {

struct node \*p;

int count = 0;

p = start;

while(p != NULL){

p = p->link;

count++;}

Printf(“Length of List : %d”, count);

return count;

}

**Delete node from the list:**

Here deleting a node means we should stop pointing the previous node to the node to be deleted and we should release the memory taken by the node. Mainly there are 3 cases in deleting the node i.e. deletion of first node, deletion of last node and deletion at a particular place. There are two different logic one for deletion of first node and another for both deletion of last and for a particular location.

**Deletion of first node:**

Algorithm:

Here we have to know the location where we want to delete the node. If the location is 1 then we have to do operation on 1st node. We have to take a temporary variable temp which should point to the first location. The address of 2nd node i.e. link part of first node should be stored into start which means start should point to the second node. Then the first node should not point to the 2nd node because there shouldn’t any link between first and second node which helps us in release the memory for the first node. After this operation the start point to the second node which is now the first node for the list.

**Code :**

void delete () {

struct node \*temp;

int loc;

printf(“Enter the location of node to be deleted\n”);

scanf(“%d”,&loc);

if (loc > length ()) {

printf(“Invalid location”); }

elseif (loc = 1) {

temp = start;

start = temp->link;

temp->link = NULL;

free(temp);

}

else {

// here logic for deletion of last or middle node should be written

}

}

**Deletion of last or middle node :**

**Algorithm:**

There is same logic for the deletion of last and middle node. We have to input the location of the node. The element which is to be deleted should be not have any relation between the previous and next node. We have to store the address of next node in the link part previous node. So that the previous node should point to the next node.

* We have to take two pointer variable p and q of struct node type. p should points to the previous node and q should point to the node to be deleted.
* Then we have to store the address of next node i.e. link part of q to link part of p so that previous node should point to the next node.
* The node to be delete i.e. q is pointing to the next node so to stop pointing we have to store NULL at the q->link. Then free(q) is done to release the memory.

**Code:**

else {

struct node \*p=start, \*q;

int i = 1;

while(i < loc-1){

p = p->link;

i++; }

q=p->link;

p->link = q->link;

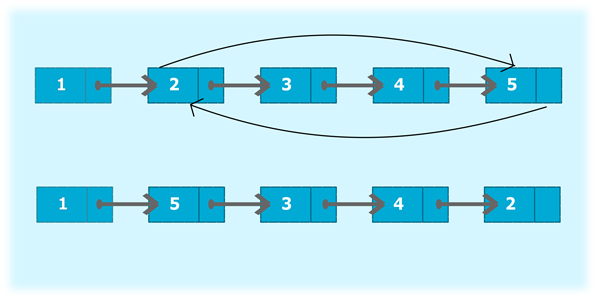
q->link = NULL;

free(q);

}

**Swap two nodes in the linked list:**

Swapping of nodes is occurred in a list having more than one element. Interchange of the values are done in between the nodes. Here we are swapping in between two adjacent nodes.



**Algorithm:**

* At a time we can swap only two positions to swap many more positions in the list we have to repeat the operation.
* We have to take pointers of the two node which we are going to swap and the previous node.
* Then we have to preform the below operation to swap the node.

**Code:**

void swap (int loc){

int i = 1;

struct node \*p, \*q, \*r;

p=start;

while(i < loc-1){

p = p -> link;

i++;

}

q->link = r->link;

r->link = q;

p->link = r;

}