

How to create Linked list using C/C++

Introduction

Prep By: Mir Waqar

Linked list is one of the fundamental data structures, and can be used to implement other data structures. In a linked list there are different numbers of nodes. Each node consists of two fields. The first field holds the value or data and the second field holds the reference to the next node or null if the linked list is empty.



Figure: Linked list

Pseudocode:

☐Collapse | [Copy Code](#)

```
LinkedList Node {  
    data // The value or data stored in the node  
    next // A reference to the next node, null for last node  
}
```

The singly-linked list is the easiest of the linked list, which has one link per node.

Pointer

To create linked list in C/C++ we must have a clear understanding about pointer. Now I will explain in brief what is pointer and how it works.

A pointer is a variable that contains the address of a variable. The question is why we need pointer? Or why it is so powerful? The answer is they have been part of the C/C++ language and so we have to use it. Using pointer we can pass argument to the functions. Generally we pass them by value as a copy. So we cannot change them. But if we pass argument using pointer, we can modify them. To understand about pointers, we must know how computer store variable and its value. Now, I will show it here in a very simple way.

Let us imagine that a computer memory is a long array and every array location has a distinct memory location.

☒Collapse | [Copy Code](#)

```
int a = 50 // initialize variable a
```

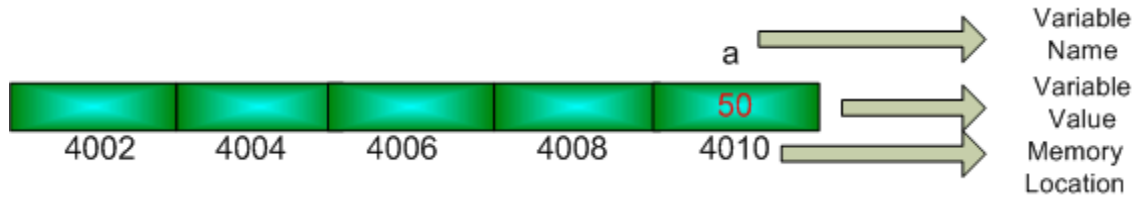


Figure: Variable value store inside an array

It is like a house which has an address and this house has only one room. So the full address is-

Name of the house: a

Name of the person/value who live here is: 50

House Number: 4010

If we want to change the person/value of this house, the conventional way is, type this code line

☒Collapse | [Copy Code](#)

```
a = 100 // new initialization
```

But using pointer we can directly go to the memory location of 'a' and change the person/value of this house without disturbing 'a'. This is the main point about pointer.

Now the question is how we can use pointer. Type this code line:

☒Collapse | [Copy Code](#)

```
int *b; // declare pointer b
```

We transfer the memory location of a to b.

☒Collapse | [Copy Code](#)

```
b = &a; // the unary operator & gives the address of an object
```

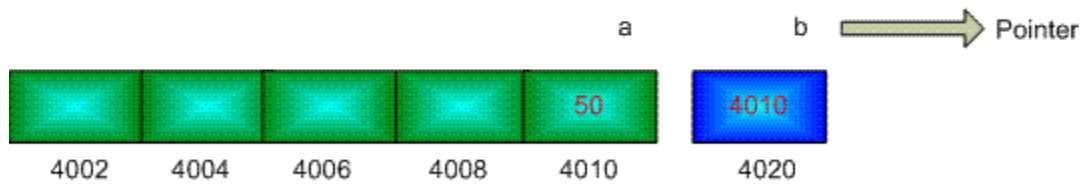


Figure: Integer pointer `b` store the address of the integer variable `a`

Now, we can change the value of `a` without accessing `a`.

☒Collapse | [Copy Code](#)

```
*b = 100; // change the value of 'a' using pointer 'b'
cout<<a; // show the output of 'a'
```

When you order the computer to access to access `*b`, it reads the content inside `b`, which is actually the address of `a` then it will follow the address and comes to the house of `a` and read `a`'s content which is 50.

Now the question is, if it is possible to read and change the content of `b` without accessing `b`? The answer is affirmative. We can create a pointer of pointer.

☒Collapse | [Copy Code](#)

```
int **c; //declare a pointer to a pointer
c = &b; //transfer the address of 'b' to 'c'
```

So, we can change the value of `a` without disturbing variable `a` and pointer `b`.

☒Collapse | [Copy Code](#)

```
**c = 200; // change the value of 'a' using pointer to a pointer 'c'
cout<<a; // show the output of a
```

Now the total code is:

☒Collapse | [Copy Code](#)

```
#include<span class="code-keyword"><iostream></span>
using namespace std;

int main()
{
    int a = 50; // initialize integer variable a
    cout<<"The value of 'a': "<<a<<endl; // show the output of a
```

```


int * b;           // declare an integer pointer b
b = &a;           // transfer the address of 'a' to pointer 'b'
*b = 100;         // change the value of 'a' using pointer 'b'
cout<<"The value of 'a' using *b: "<<a<<endl; // show the output of a

int **c;          // declare an integer pointer to pointer 'c'
c = &b;           // transfer the address of 'b' to pointer to pointer 'c'
**c = 200;        // change the value of 'a' using pointer to pointer 'c'
cout<<"The value of 'a' using **c: "<<a<<endl; // show the output of a

return 0;
}

```

Output:



```

MS-A
Auto
The value of 'a': 50
The value of 'a' using *b: 100
The value of 'a' using **c: 200
Press any key to continue

```

This program will give you the inside view of the pointer.

☒Collapse | [Copy Code](#)

```

#include<span class="code-keyword"><iostream></span>
using namespace std;

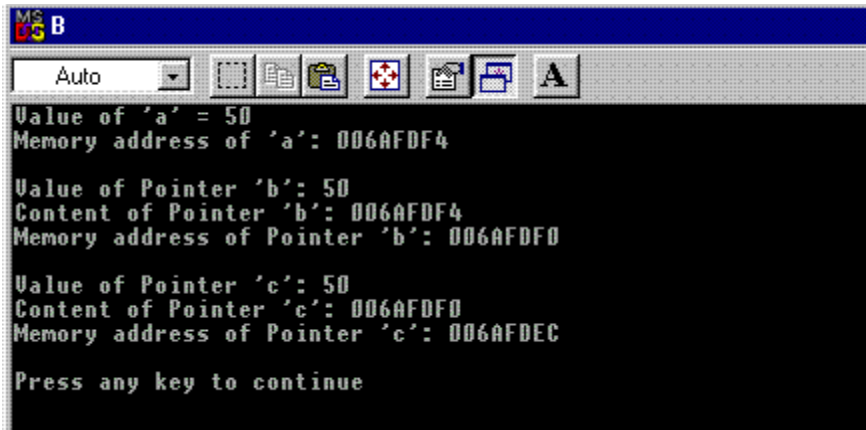
int main()
{
    int a = 50;           // initialize integer variable a
    cout<<"Value of 'a' = "<<a<<endl;           // show the output of a
    cout<<"Memory address of 'a': "<<&a<<endl; // show the address of a
    cout<<endl;

    int * b;              // declare an integer pointer b
    b = &a;               // transfer the address of 'a' to pointer 'b'
    cout<<"Value of Pointer 'b': "<<*b<<endl; // show the output of *b
    cout<<"Content of Pointer 'b': "<<b<<endl; // show the content of *b
    cout<<"Memory address of Pointer 'b': "<<&b<<endl; // show the address of *b
    cout<<endl;

    int **c;              // declare an integer pointer to a pointer
    c = &b;               // transfer the address of 'b' to 'c'
    cout<<"Value of Pointer 'c': "<<**c<<endl; // show the output of **c
    cout<<"Content of Pointer 'c': "<<c<<endl; // show the content of **c
    cout<<"Memory address of Pointer 'c': "<<&c<<endl; // show the address of **c
    cout<<endl;
    return 0;
}

```

Output:



```
MS-DOS
Auto
Value of 'a' = 50
Memory address of 'a': 006AFDF4

Value of Pointer 'b': 50
Content of Pointer 'b': 006AFDF4
Memory address of Pointer 'b': 006AFDF0

Value of Pointer 'c': 50
Content of Pointer 'c': 006AFDF0
Memory address of Pointer 'c': 006AFDEC

Press any key to continue
```

We can observe that the memory address of **a** and the content of pointer **b** is same. The content of pointer **c** and the memory address of **b** is same.

Linked list operation

Now we have a clear view about pointer. So we are ready for creating linked list.

Linked list structure

☐ Collapse | [Copy Code](#)

```
typedef struct node
{
    int data;                // will store information
    node *next;              // the reference to the next node
};
```

First we create a structure “node”. It has two members and first is **int data** which will store the information and second is **node *next** which will hold the address of the next node. Linked list structure is complete so now we will create linked list. We can insert data in the linked list from 'front' and at the same time from 'back'. Now we will examine how we can insert data from front in the linked list.

1) Insert from front

At first initialize node type.

☐ Collapse | [Copy Code](#)

```
node *head = NULL;           //empty linked list
```

Then we take the data input from the user and store in the **node info** variable. Create a temporary node **node *temp** and allocate space for it.

☒Collapse | [Copy Code](#)

```
node *temp;           //create a temporary node
temp = (node*)malloc(sizeof(node)); //allocate space for node
```

Then place `info` to `temp->data`. So the first field of the `node *temp` is filled. Now `temp->next` must become a part of the remaining linked list (although now linked list is empty but imagine that we have a 2 node linked list and head is pointed at the front) So `temp->next` must copy the address of the `*head` (Because we want insert at first) and we also want that `*head` will always point at front. So `*head` must copy the address of the `node *temp`.

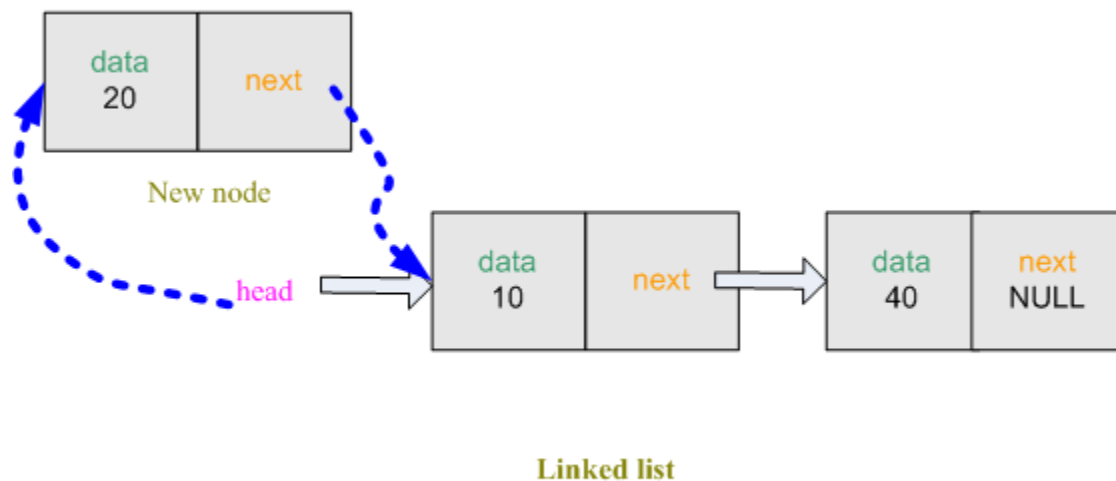


Figure: Insert at first

☒Collapse | [Copy Code](#)

```
temp->data = info;           // store data(first field)
temp->next=head; // store the address of the pointer head(second field)
head = temp;                 // transfer the address of 'temp' to 'head'
```

2) Traverse

Now we want to see the information stored inside the linked list. We create node `*temp1`. Transfer the address of `*head` to `*temp1`. So `*temp1` is also pointed at the front of the linked list. Linked list has 3 nodes.

We can get the data from first node using `temp1->data`. To get data from second node, we shift `*temp1` to the second node. Now we can get the data from second node.

☒Collapse | [Copy Code](#)

```
while( temp1!=NULL )
{
    cout<< temp1->data<<" "; // show the data in the linked list
    temp1 = temp1->next;    // tranfer the address of 'temp->next' to 'temp'
}
```

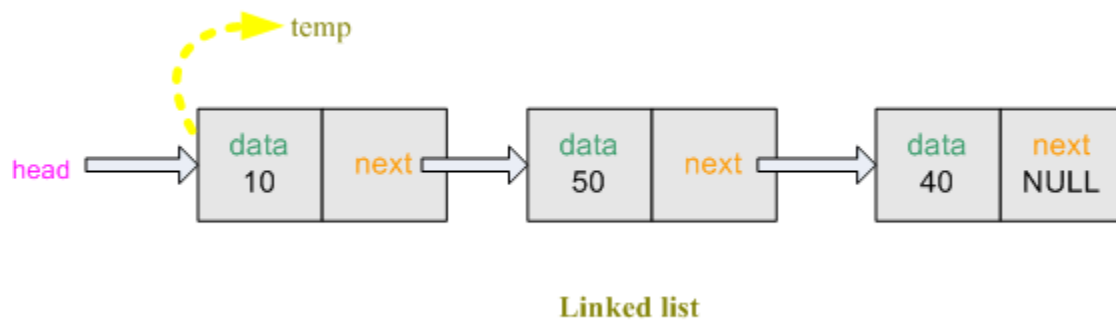


Figure: Traverse

This process will run until the linked list's next is NULL.

3) Insert from back

Insert data from back is very similar to the insert from front in the linked list. Here the extra job is to find the last node of the linked list.

☒Collapse | [Copy Code](#)

```
node *temp1; // create a temporary node
temp1=(node*)malloc(sizeof(node)); // allocate space for node
temp1 = head; // transfer the address of 'head' to 'temp1'
while(temp1->next!=NULL) // go to the last node
    temp1 = temp1->next; // tranfer the address of 'temp1->next' to 'temp1'
```

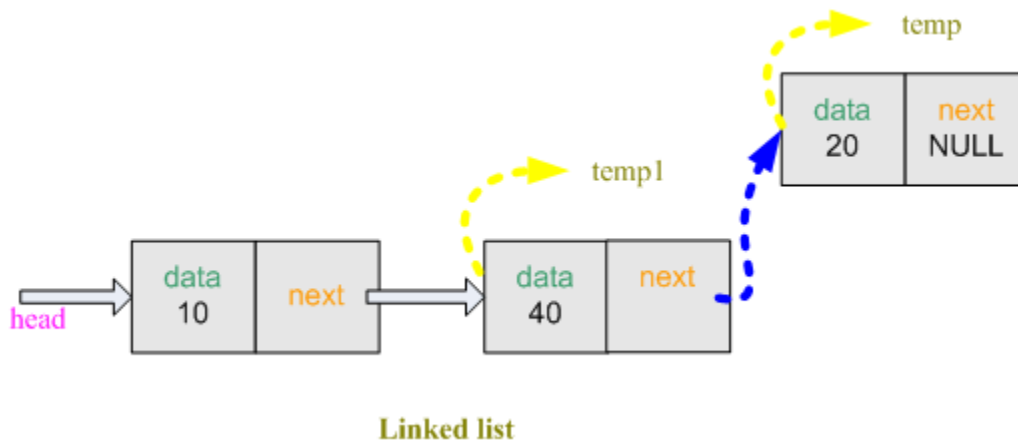
Now, Create a temporary node `node *temp` and allocate space for it. Then place `info` to `temp->data`, so the first field of the node `node *temp` is filled. `node *temp` will be the last node of the linked list. For this reason, `temp->next` will be NULL. To create a connection between linked list and the new node, the last node of the existing linked list `node *temp1`'s second field `temp1->next` is pointed to `node *temp`.



Figure: Insert at last

☒Collapse | [Copy Code](#)

```
node *temp; // create a temporary node
temp = (node*)malloc(sizeof(node)); // allocate space for node
temp->data = info; // store data(first field)
temp->next = NULL; // second field will be null(last node)
temp1->next = temp; // 'temp' node will be the last node
```



4) Insert after specified number of nodes

Insert data in the linked list after specified number of node is a little bit complicated. But the idea is simple. Suppose, we want to add a node after 2nd position. So, the new node must be in 3rd position. The first step is to go the specified number of node. Let, node *temp1 is pointed to the 2nd node now.

☒Collapse | [Copy Code](#)

```
cout<<"ENTER THE NODE NUMBER:";
cin>>node_number; // take the node number from user

node *temp1; // create a temporary node
temp1 = (node*)malloc(sizeof(node)); // allocate space for node
temp1 = head;

for( int i = 1 ; i < node_number ; i++ )
{
    temp1 = temp1->next; // go to the next node

    if( temp1 == NULL )
```



```

    {
        cout<<node_number<<" node is not exist"<< endl;
        break;
    }
}

```

Now, Create a temporary node `node *temp` and allocate space for it. Then place `info` to `temp->next` , so the first field of the node `node *temp` is filled.

☒Collapse | [Copy Code](#)

```

node *temp; // create a temporary node
temp = (node*)malloc(sizeof(node)); // allocate space for node
temp->data = info; // store data(first field)

```

To establish the connection between new node and the existing linked list, new node's `next` must pointed to the 2nd node's (temp1) `next` . The 2nd node's (temp1) next must pointed to the new node(temp).

☒Collapse | [Copy Code](#)

```

temp->next = temp1->next; //transfer the address of temp1->next to temp->next
temp1->next = temp; //transfer the address of temp to temp1->next

```

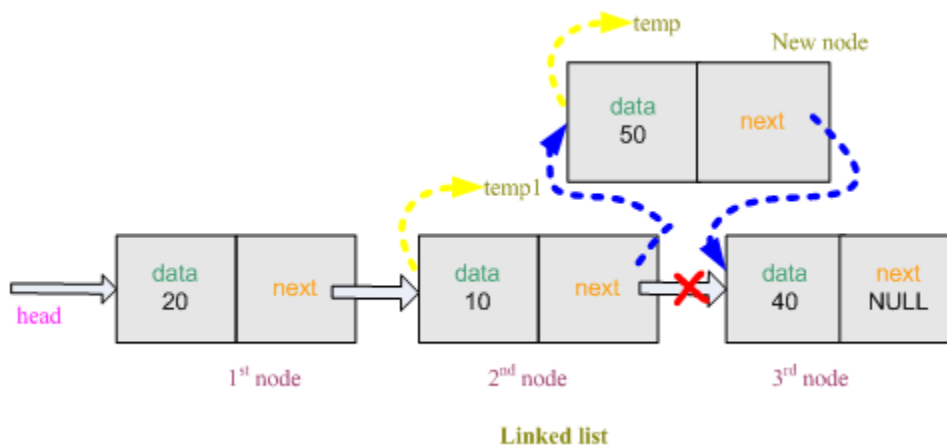


Figure: Insert after specified number of nodes

5) Delete from front

Delete a node from linked list is relatively easy. First, we create node `*temp`. Transfer the address of `*head` to `*temp`. So `*temp` is pointed at the front of the linked list. We want to delete the first node. So transfer the address of `temp->next` to `head` so that it now pointed to the second node. Now free the space allocated for first node.

☒Collapse | [Copy Code](#)

```
node *temp; // create a temporary node
temp = (node*)malloc(sizeof(node)); // allocate space for node
temp = head; // transfer the address of 'head' to 'temp'
head = temp->next; // transfer the address of 'temp->next' to 'head'
free(temp);
```

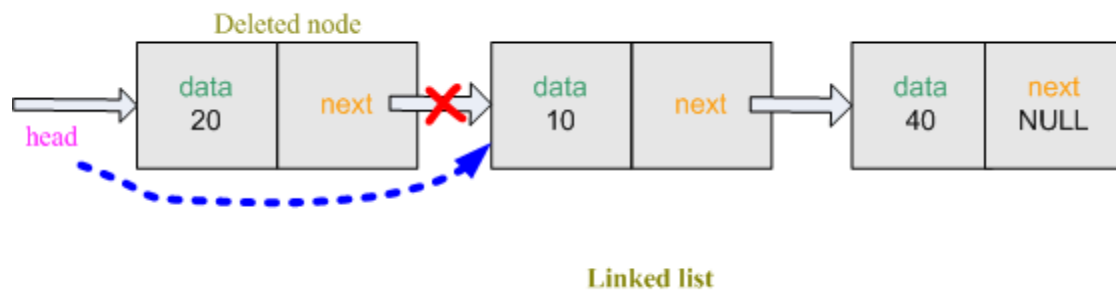


Figure: Delete at first node

6) Delete from back

The last node's `next` of the linked list always pointed to NULL. So when we will delete the last node, the previous node of last node is now pointed at NULL. So, we will track last node and previous node of the last node in the linked list. Create temporary `node * temp1` and `*old_temp`.

☒Collapse | [Copy Code](#)

```
// create a temporary node
node *temp1;
temp1 = (node*)malloc(sizeof(node)); // allocate space for node
temp1 = head; // transfer the address of head to temp1
node *old_temp; // create a temporary node
old_temp = (node*)malloc(sizeof(node)); // allocate space for node

while(temp1->next!=NULL) // go to the last node
{
    old_temp = temp1; // transfer the address of 'temp1' to 'old_temp'
    temp1 = temp1->next; // transfer the address of 'temp1->next' to 'temp1'
}
```

Now `node *temp1` is now pointed at the last node and `*old_temp` is pointed at the previous node of the last node. Now rest of the work is very simple. Previous node of the last node `old_temp` will be NULL so it become the last node of the linked list. Free the space allocated for last node.

☒Collapse | [Copy Code](#)

```
old_temp->next = NULL; // previous node of the last node is null
free(temp1);
```

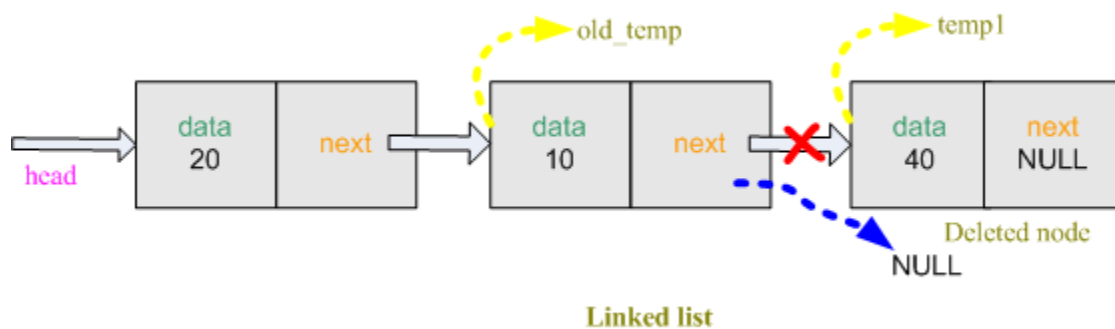


Figure: Delete at first last

7) Delete specified number of node

To delete a specified node in the linked list, we also require to find the specified node and previous node of the specified node. Create temporary `node * temp1`, `*old_temp` and allocate space for it. Take the input from user to know the number of the node.

☒Collapse | [Copy Code](#)

```
node *temp1; // create a temporary node
temp1 = (node*)malloc(sizeof(node)); // allocate space for node
temp1 = head; // transfer the address of 'head' to 'temp1'

node *old_temp; // create a temporary node
old_temp = (node*)malloc(sizeof(node)); // allocate space for node
old_temp = temp1; // transfer the address of 'temp1' to 'old_temp'
```

☒Collapse | [Copy Code](#)

```
cout<<"ENTER THE NODE NUMBER:";
cin>>node_number; // take location
```

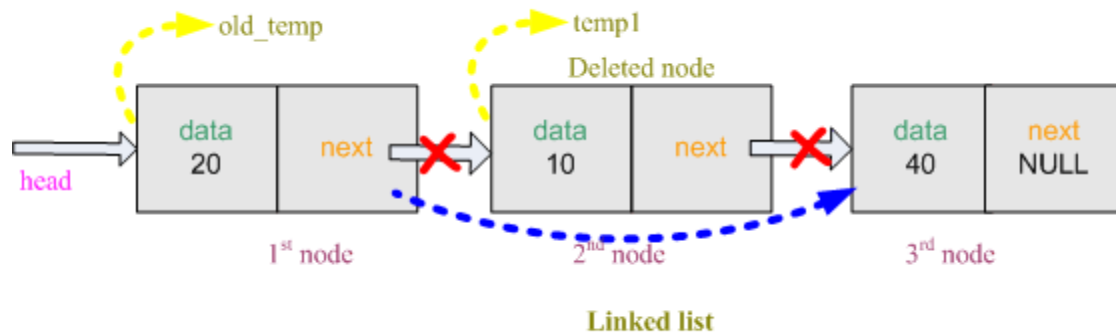
☒Collapse | [Copy Code](#)

```
for( int i = 1 ; i < node_number ; i++ )
{
    old_temp = temp1; // store previous node
    temp1 = temp1->next; // store current node
}
```

Now `node *temp1` is now pointed at the specified node and `*old_temp` is pointed at the previous node of the specified node. The previous node of the specified node must connect to the rest of the linked list so we transfer the address of `temp1->next` to `old_temp->next`. Now free the space allocated for the specified node.

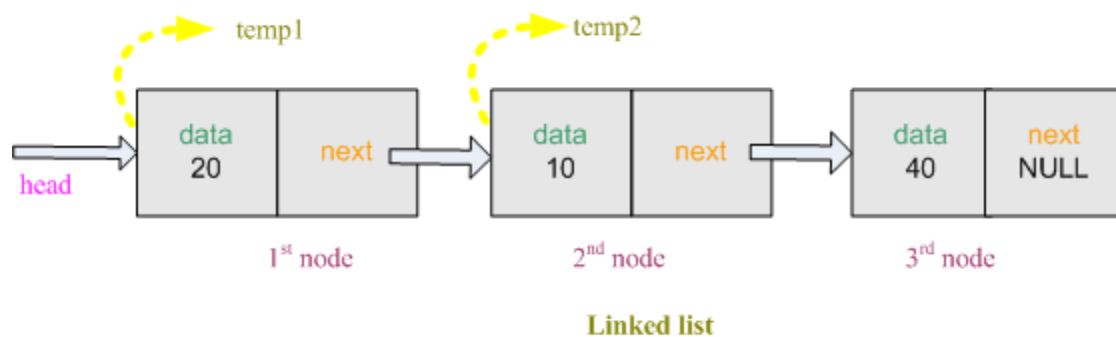
☒Collapse | [Copy Code](#)

```
old_temp->next = temp1->next; // transfer the address of 'temp1->next' to 'old_temp->next'
free(temp1);
```



8) Sort nodes

Linked list sorting is very simple. It is just like ordinary array sorting. First we create two temporary node `node *temp1, *temp2` and allocate space for it. Transfer the address of first node to `temp1` and address of second node to `temp2`. Now check if `temp1->data` is greater than `temp2->data`. If yes then exchange the data. Similarly, we perform this checking for all the nodes.



☒Collapse | [Copy Code](#)

```
node *temp1; // create a temporary node
temp1 = (node*)malloc(sizeof(node)); // allocate space for node

node *temp2; // create a temporary node
temp2 = (node*)malloc(sizeof(node)); // allocate space for node

int temp = 0; // store temporary data value

for( temp1 = head ; temp1!=NULL ; temp1 = temp1->next )
{
```

```

for( temp2 = temp1->next ; temp2!=NULL ; temp2 = temp2->next )
{
    if( temp1->data > temp2->data )
    {
        temp = temp1->data;
        temp1->data = temp2->data;
        temp2->data = temp;
    }
}
}

```

Conclusion

From the above discussions, I hope that everybody understands what linked list is and how we can create it. Now we can easily modify linked list according to our program requirement and try to use it for some real tasks. Those who still have some confusion about linked list, for them I will now tell you a story.

Once upon a time, an old man lived in a small village. The old man was very wise and knew many solutions about different problems. He had also special powers so he could talk to genie and spirits, and sometimes they granted his wish by using their special powers. Oneday a witch with a broom came to talk with him and ask difficult and complex issues about global warming. He was very surprised but patiently explained her about green house model and gave her advice about using biofuel in her broom. The witch was very rude and greedy but she always liked to preach her nobility. So at the time of her departure, she wanted to grant only two wishes. The old man asked her why she only granted two wishes. He also reminded her that whenever genie comes he granted two wishes." What I am look like" the witch asked angrily," A blank check?" The old man brewed a plan. He told her that his first wish was to get a super computer." It is granted", the witch announced loudly." Then my second wish is to have another two wishes", the old man said very slowly. The witch was shell shocked. "It is also granted", the witch said and left the place very quickly with her broom.

You may ask yourself why the witch was surprised. First, the witch granted two wishes. One was fulfilled and for the second wish, the old man wanted another two wish. "What's the big idea?" you can ask me," The witch can also fulfill this wish". Certainly, the witch can grant his wish. But what will happen if the old man wants to extend his second wish to another two wish set. So, the process will never end unless, the old man want to stop. The idea is same for linked list. First you have a node where you can imagine that `data` is the first wish and `node*next` is the second wish by which you can create second node just like first. This process will continue until you put NULL in the `*next`.