

Non-Recursive Single Linked List

COP 2335

April 3, 2015

By

Adnan Zejnilovic

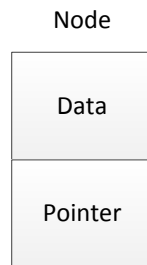


Wolfson Campus

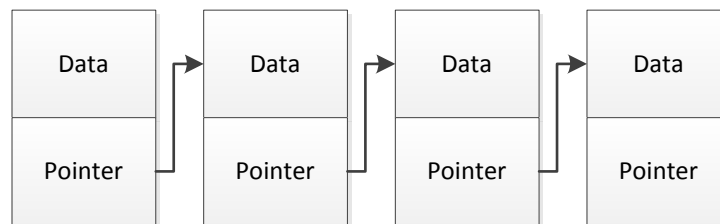
Preliminaries	3
Getting Started.....	4
Adding Nodes.....	5
Adding a Node to the Beginning of a List.....	5
Adding a Node to the End of the List	8
Using Pointers to Keep Track of Positions in the List.....	13
addLast with “tail”:	14
Adding in the Middle of the List.....	16
Deleting a Node from the List.....	19
Only One Node in the List	19
Removing the First Node	19
Removing the Last Node	21
Deleting a Node in the Middle of the List.....	23

Preliminaries

Unlike an array data structure, a linked list consists of “nodes” that are linked to each other via pointers. Depending on the type of the list (Single Linked List, Double Linked List, Circular Linked List (single or double)), the node may look a bit different. In general, it consists of the *Data* part and the *Pointer* part:



Normally, the *Pointer* part of the node is linked to the *Data* part of the next node:



As its name implies, the *Data* part contains the data which can be any data (data type) - from a single character, to a more complex programmer defined types such as structures and classes. The *Pointer* part is just that, a pointer to the *Data*. Thus we have a case of recursion here.

Let's review the following code segment:

```
const int SIZE = 35;

struct Entry
{
    char name[SIZE];
    int age;
    Entry *next;
};
```

The *Data* part consists of:

```
char name[SIZE];
int age;
```

while the *Pointer* part is:

```
Entry *next;
```

What is the data type of structure member “next”? It is of type *Entry* which means that we are defining a pointer to a data type that we have not finished defining yet. Thus we have recursion in our data type (structure of type *Entry* one of its members pointing to itself).

Getting Started

After defining our structure, we are ready to start implementing our linked list. The first step in the process is to obtain a node that will capture user data. We can accomplish that by writing a function to “get a new node”:

```
Entry* getNewEntry()
{
    char name[SIZE];
    cout << "Enter a new name (ENTER to quit):";
    cin.getline(name, SIZE);
    if (strlen(name) == 0)
    {
        cout << "You pressed ENTER" << endl;
        return NULL;
    }

    Entry *newOne = new Entry;
    strcpy(newOne->name, name);
    cout << "Age: ";
    cin >> newOne->age;
    cin.ignore(1024, '\n');
    newOne->next = NULL;
    return newOne;
}
```

The function allocates a new Entry structure on the heap, and populates its members with user supplied data. The function returns a NULL pointer if the user presses the ENTER key when entering person's name. The test case for this function follows:

```
int main(int argc, char *argv[])
{
    Entry* newOne = getNewEntry();
    if (newOne)
        displayEntry(newOne);

    system("PAUSE");
    return EXIT_SUCCESS;
}

void displayEntry(Entry* e)
{
    cout << e->name << "    " << e->age << endl;
}
```

This code is safe and should work correctly. The problem is that the “newOne” points to only one node, nothing else. We need a mechanism to insert this newly created node (Entry) into the list. What we need is a function to build a list.

There will be cases that we will need to:

- add a newly allocated node to the beginning of the list,
- Add a newly allocated node to the end of the list, or

- Add a newly allocated node somewhere in the middle of the list.

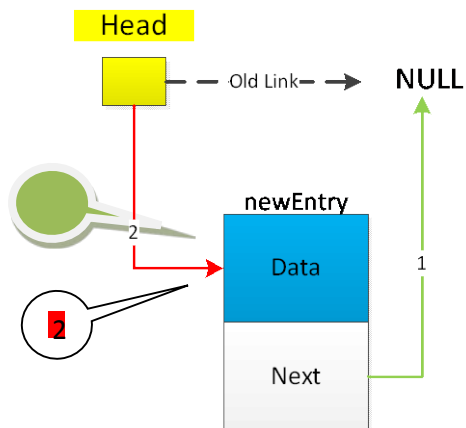
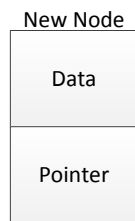
We are going to explore each case in greater detail.

Adding Nodes

A node can be added to the beginning, end, or somewhere in the middle of the list. We will examine details of each case next.

Adding a Node to the Beginning of a List

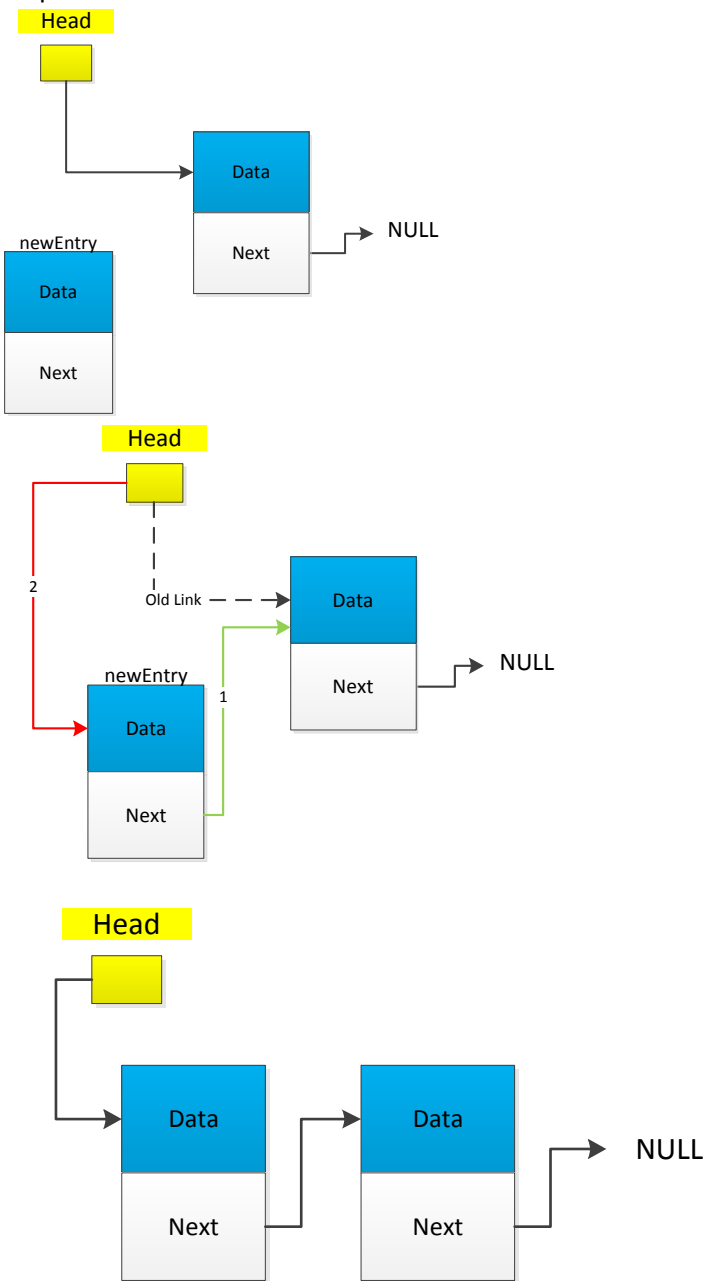
In order to add a node to the beginning of a list, we need to connect it correctly to the head of the list. There are several variations on this depending on the implementation. For example, in a very simple case, only the head pointer is present (and not the tail):



```
bool addFirst(Entry* newEntry, Entry* &head)
{
    if (newEntry == NULL)
        return false;

    newEntry->next = head;
    head = newEntry;
    return true;
}
```

The process is the same if there are some elements in the list:



To test this function, we need to call it inside of the buildList function:

```
Entry* buildList()
{
    Entry *listHead = NULL; // start of the list
    while(true)
    {
        Entry *newOne = getNewEntry();

        // add to the beginning
        if(!addFirst(newOne, listHead))
            break;
    }

    return listHead;
}
```

We also introduce the displayList function at this point:

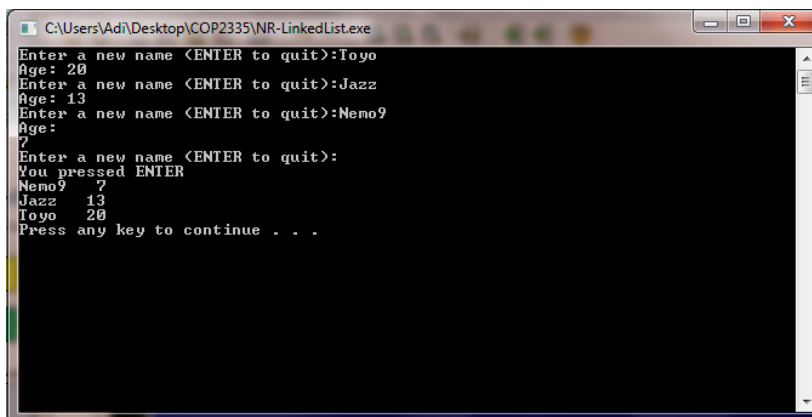
```
void displayList(Entry *list)
{
    for (Entry *current = list; current!=NULL; current = current->next)
        displayEntry(current);
}
```

And the main function is basically calling the buildList function, and the displayList function:

```
int main(int argc, char *argv[])
{
    Entry* listHead = buildList();
    displayList(listHead);

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

After compiling and running the program, we get the following:

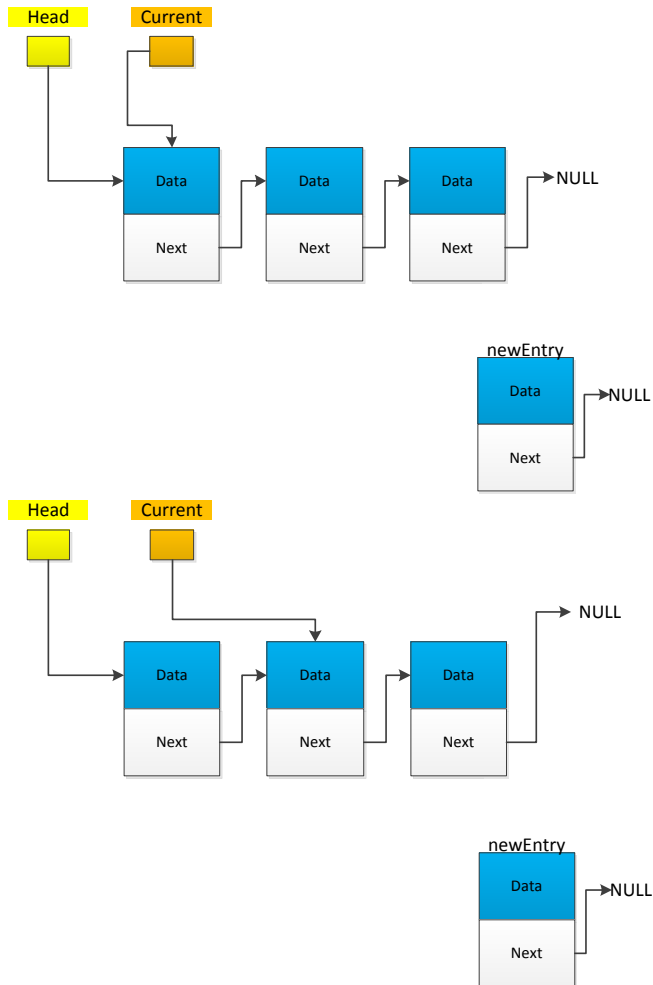


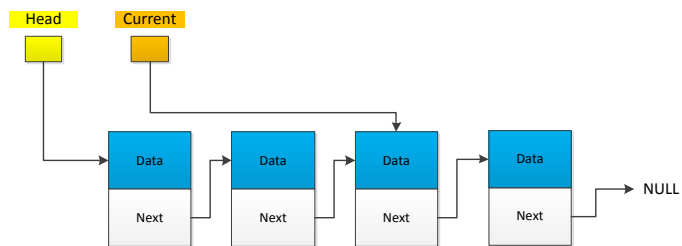
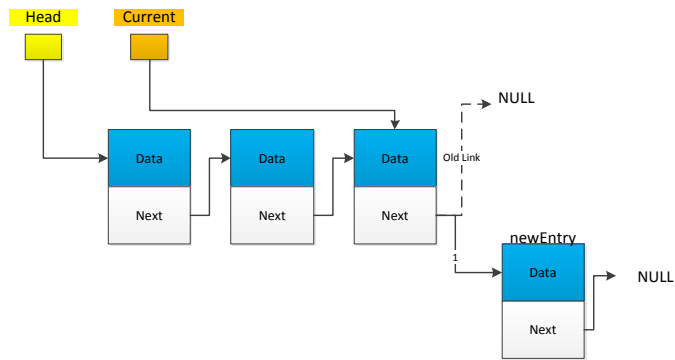
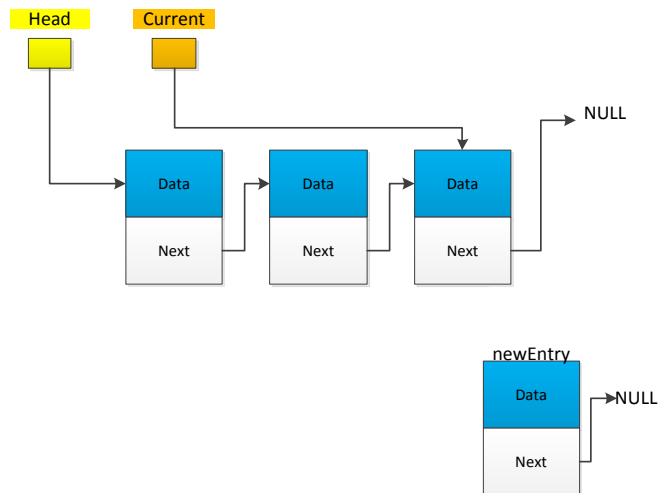
```
C:\Users\Adn\Desktop\COP2335\NR-LinkedList.exe
Enter a new name <ENTER to quit>:Toyo
Age: 20
Enter a new name <ENTER to quit>:Jazz
Age: 13
Enter a new name <ENTER to quit>:Nemo9
Age: 7
?
Enter a new name <ENTER to quit>:
You pressed ENTER
Nemo9 7
Jazz 13
Toyo 20
Press any key to continue . . .
```

This is Ok, the list appears as if it is being printed backward but that is not the case, as each node we added has been added to the front of the list thus pushing the rest of the nodes to the end of the list.

Adding a Node to the End of the List

With the current design (keeping track of the head pointer only), how would we add a node to the end of the list? Well, we would have to write a function to “traverse” our list. Basically, we would visit each node, until we reached the end of the list (until the “next” was pointing to a NULL). At that point, we could add a node to the end of the list.





The following function called `addLast` demonstrates that:

```
bool addLast(Entry* newEntry, Entry* &head)
{
    if (newEntry == NULL)
        return false;
    Entry* current;

    current = head;
    do
    {
        current= current->next;
    }while(current==NULL);

    // Set the current's next to the newEntry
    current->next = newEntry;
    return true;
}
```

And here are the changes to the main function:

```
int main(int argc, char *argv[])
{
    Entry* listHead = buildList();
    displayList(listHead);

    Entry* lastEntry = getNewEntry();
    if (!addLast(lastEntry, listHead))
    {
        return EXIT_FAILURE;;
    }
    cout << "Will display the list now: " << endl;
    displayList(listHead);

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Compile and run the program and then enter the following info:

Name: Toyo
Age: 20
Name Jazz
Age: 14

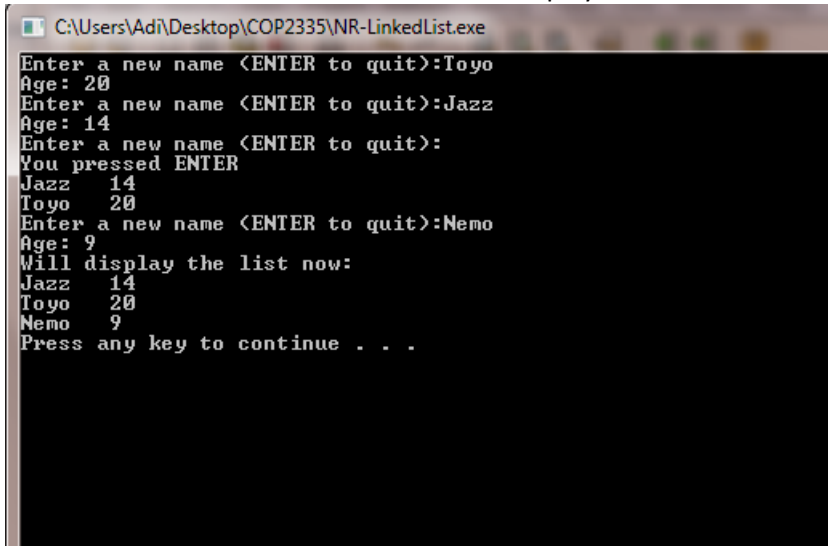
Press the ENTER key to signify the end of input.

At this point, the program will ask for one more name so a new node can be added to the end of the list. Provide the following info:

Name: Nemo

Age: 9

And it will add it to the end of the list and display the list:



```
C:\Users\Adi\Desktop\COP2335\NR-LinkedList.exe
Enter a new name <ENTER to quit>:Toyo
Age: 20
Enter a new name <ENTER to quit>:Jazz
Age: 14
Enter a new name <ENTER to quit>:
You pressed ENTER
Jazz 14
Toyo 20
Enter a new name <ENTER to quit>:Nemo
Age: 9
Will display the list now:
Jazz 14
Toyo 20
Nemo 9
Press any key to continue . . .
```

The displayed results are correct as the node containing Nemo was correctly added to the end of the list. Node with Jazz info, which was added 2nd, is the first member of the list as we added it using the `addFirst` function. The `addLast` function needs to be improved, as its design assumes that there are some elements in the list. What if the list is an empty list? What would we need to change in the design/implementation in order to be able to add a node to an empty list?

All we need to do is check whether the list is empty:

```
bool addLast(Entry* newEntry, Entry* &head)
{
    if (newEntry == NULL)
        return false;
    else
    {
        Entry* current;
        current = head;

        if (head == NULL)
        {
            head = newEntry;
        }
        else
        {
            // traverse the list to get to the end
            while(current->next != NULL)
            {
                current = current->next;
            }

            // Set the current's next to the newEntry
            current->next = newEntry;
        }
    }

    return true;
}
```

We also make a minimal change to the buildList function:

```
Entry* buildList()
{
    Entry *listHead = NULL; // start of the list
    while(true)
    {
        Entry *newOne = getNewEntry();

        // add to the beginning
        if(!addLast(newOne, listHead))
            break;
    }

    return listHead;
}
```

And to int main:

```
int main(int argc, char *argv[])
{
    Entry* listHead = buildList();
    displayList(listHead);

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Now if we enter the following data:

Name:Toyo

Age: 20

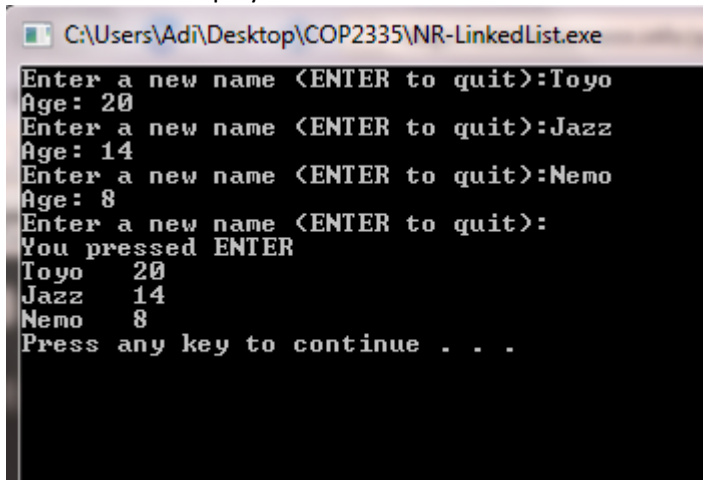
Name: Jazz

Age: 14

Name: Nemo

Age: 8

The list will be displayed in the correct order:



```
C:\Users\Adi\Desktop\COP2335\NR-LinkedList.exe
Enter a new name (ENTER to quit):Toyo
Age: 20
Enter a new name (ENTER to quit):Jazz
Age: 14
Enter a new name (ENTER to quit):Nemo
Age: 8
Enter a new name (ENTER to quit):
You pressed ENTER
Toyo 20
Jazz 14
Nemo 8
Press any key to continue . . .
```

Using Pointers to Keep Track of Positions in the List

So far, we only dealt with the head (of the list) pointer, but what if we need to keep track of the end ("tail") of the list as well? What are the benefits of implementing a list with both head and tail pointers? for a few extra lines of code, we can get additional flexibility and speed to be able to add the new node to the list immediately rather than traversing through the list.

This can be accomplished by keeping track of the head and tail position, and also, the current, and the previous positions in the list. We will rewrite our functions to demonstrate that:

```
addFirst with "tail"
bool addFirst(Entry* newEntry, Entry* &head, Entry* &tail)
{
    if (newEntry == NULL)
        return false;
    else
    {
        if (head == NULL)
        {
            head = newEntry;
            tail = newEntry;
        }
        else
        {
            newEntry->next = head;
            head = newEntry;
        }
    }

    return true;
}
```

We also make a change in the buildList:

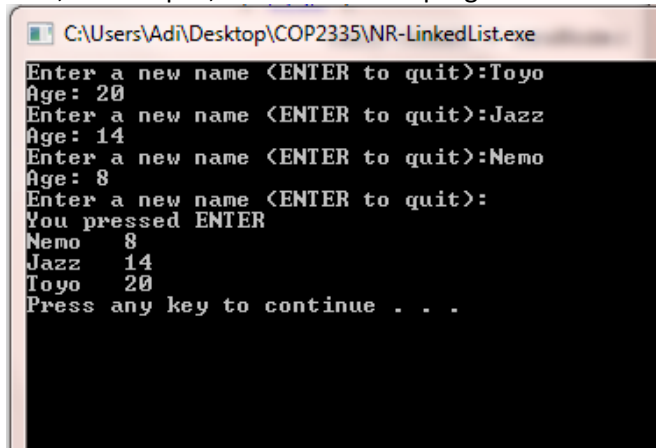
```
Entry* buildList()
{
    Entry *listHead = NULL; // start of the list
    Entry* listTail = NULL;

    while(true)
    {
        Entry *newOne = getNewEntry();

        // add to the beginning
        if(!addFirst(newOne, listHead, listTail))
            break;
    }

    return listHead;
}
```

Next, we compile, and execute our program with the familiar input:



```
C:\Users\Adi\Desktop\COP2335\NR-LinkedList.exe
Enter a new name <ENTER to quit>:Toyo
Age: 20
Enter a new name <ENTER to quit>:Jazz
Age: 14
Enter a new name <ENTER to quit>:Nemo
Age: 8
Enter a new name <ENTER to quit>:
You pressed ENTER
Nemo 8
Jazz 14
Toyo 20
Press any key to continue . . .
```

addLast with “tail”:

```
bool addLast(Entry* newEntry, Entry* &head, Entry* &tail)
{
    if (newEntry == NULL)
        return false;
    else
    {
        Entry* current;
        current = head;

        if (head == NULL)
        {
            head = newEntry;
            tail = newEntry;
        }
        else
        {
            tail->next = newEntry;
            tail = newEntry;
        }
    }

    return true;
}
```

Changes in the buildList:

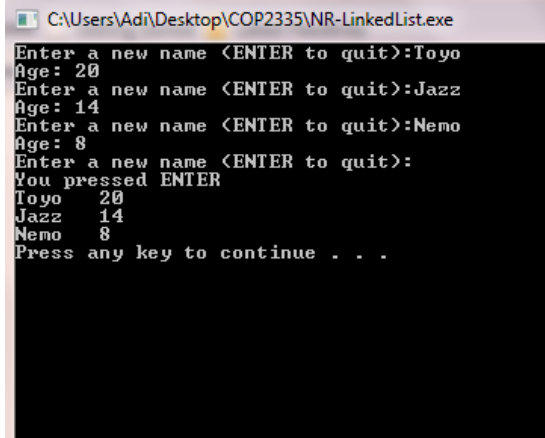
```
Entry* buildList()
{
    Entry *listHead = NULL; // start of the list
    Entry* listTail = NULL;

    while(true)
    {
        Entry *newOne = getNewEntry();

        // add to the beginning
        if(!addLast(newOne, listHead, listTail))
            break;
    }

    return listHead;
}
```

And the output:



```
C:\Users\Adi\Desktop\COP2335\NR-LinkedList.exe
Enter a new name <ENTER to quit>:Toyo
Age: 20
Enter a new name <ENTER to quit>:Jazz
Age: 14
Enter a new name <ENTER to quit>:Nemo
Age: 8
Enter a new name <ENTER to quit>:
You pressed ENTER
Toyo 20
Jazz 14
Nemo 8
Press any key to continue . . .
```

It is evident that the code with the tail pointer is simpler, as we do not need to traverse the list to find the last member of the list because the tail pointer keeps track of this for us.

Next, we explore the case when a node needs to be inserted anywhere in the list. We will refer to it as adding in the middle of the list.

Adding in the Middle of the List

The idea is that you will be traversing through the list looking for the correct place to insert newly added node. We can assume the list is sorted for this example. Once the correct insertion position has been found, we will insert the new node into the list simply by wiring the pointers.

In order to implement this correctly, we need to consider the following cases:

- The new node may need to be inserted in the first position
- The new node may need to be inserted in the last position
- The new node may need to be inserted anywhere in the list between the first and the last position

As we traverse through the list, we need to keep track of the current location in the list and the previous position in the list. Thus we will need two pointers called `current`, and `previous` to help us “inch-worm” through the list. As soon as we find the correct place to insert the new node, these two pointers will play crucial role in the wiring of the new node. We also need to be aware that once we find the correct position in the list, we may need to insert either before or after the current node. For this example, we will concentrate on inserting “before” the current node” (right after the “previous”):

```
void traverseList(Entry* &head, Entry* &tail, Entry* newEntry)
{
    Entry* current;
    Entry* previous;
    bool found = false;

    current = head;
    previous = NULL;

    while ((current != NULL) && !found)
    {
        if (newEntry->age >= current->age)
        {
            found = true;
        }
        else
        {
            previous = current;
            current = current->next;
        }
    }

    if (!addAnywhere(previous, newEntry, head, tail))
    {
        cout << "Failed to addAnywhere!" << endl;
        found = false;
    }
}
```



```

bool addAnywhere(Entry* &previous, Entry* newEntry, Entry* &head, Entry*
&tail)
{
    if (newEntry == NULL)
        return false;
    else
    {
        if (previous == NULL)
        {
            addFirst(newEntry, head, tail);
        }
        else if (previous == tail)
        {
            addLast(newEntry, head, tail);
        }
        else
        {
            Entry* next = previous->next;
            previous->next = newEntry;
            newEntry->next = next;
        }

        return true;
    }
}

Entry* buildList(Entry* &tail)
{
    Entry *listHead = NULL; // start of the list

    while(true)
    {
        Entry *newOne = getNewEntry();

        // add to the end
        if(!addLast(newOne, listHead, tail))
            break;
    }

    return listHead;
}

int main(int argc, char *argv[])
{
    Entry* listTail = NULL;
    Entry* listHead = buildList(listTail);
    displayList(listHead);

    // get an entry and add it
    Entry* newOne = getNewEntry();

    if (newOne)
        traverseList(listHead, listTail, newOne);

    displayList(listHead);

    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Compile and run the program with the following data:

Name: Toyo

Age: 20

Name: Jazz

Age: 14

Name: Nemo

Age: 9

Name: <ENTER>

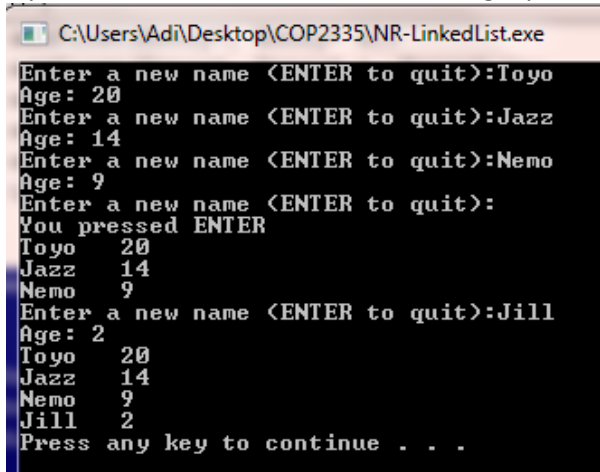
When prompted for new input enter the following:

Name: Jill

Age: 28

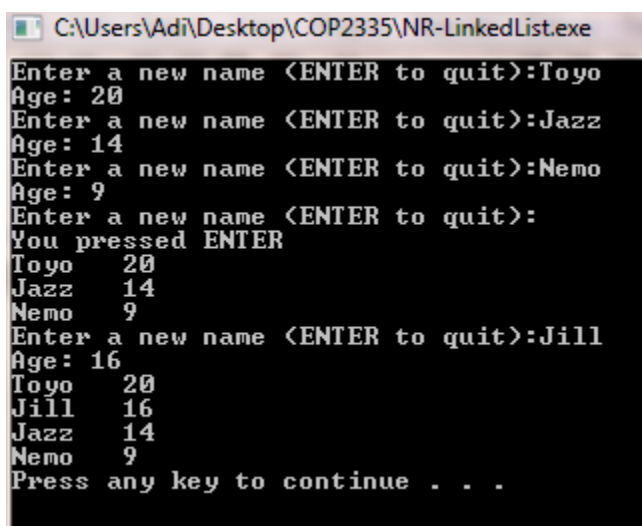
The program correctly inserts the node in the list based on “age” member.

Try several other variations with Jill being 2 years old:

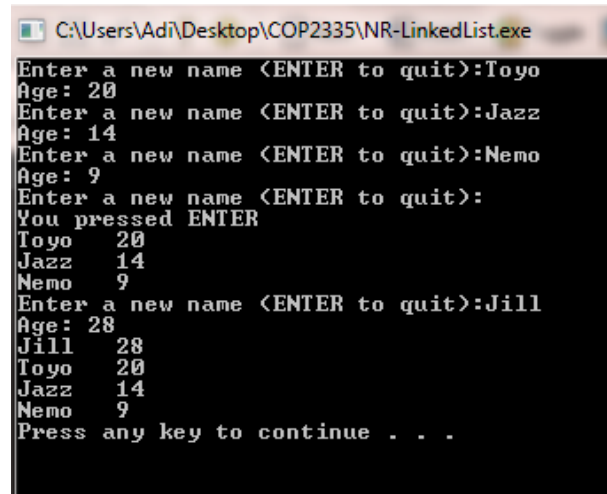


```
C:\Users\Adi\Desktop\COP2335\NR-LinkedList.exe
Enter a new name <ENTER to quit>:Toyo
Age: 20
Enter a new name <ENTER to quit>:Jazz
Age: 14
Enter a new name <ENTER to quit>:Nemo
Age: 9
Enter a new name <ENTER to quit>:
You pressed ENTER
Toyo 20
Jazz 14
Nemo 9
Enter a new name <ENTER to quit>:Jill
Age: 2
Toyo 20
Jazz 14
Nemo 9
Jill 2
Press any key to continue . . .
```

Finally, test again with Jill being 16:



```
C:\Users\Adi\Desktop\COP2335\NR-LinkedList.exe
Enter a new name <ENTER to quit>:Toyo
Age: 20
Enter a new name <ENTER to quit>:Jazz
Age: 14
Enter a new name <ENTER to quit>:Nemo
Age: 9
Enter a new name <ENTER to quit>:
You pressed ENTER
Toyo 20
Jazz 14
Nemo 9
Enter a new name <ENTER to quit>:Jill
Age: 16
Toyo 20
Jill 16
Jazz 14
Nemo 9
Press any key to continue . . .
```



```
C:\Users\Adi\Desktop\COP2335\NR-LinkedList.exe
Enter a new name <ENTER to quit>:Toyo
Age: 20
Enter a new name <ENTER to quit>:Jazz
Age: 14
Enter a new name <ENTER to quit>:Nemo
Age: 9
Enter a new name <ENTER to quit>:
You pressed ENTER
Toyo 20
Jazz 14
Nemo 9
Enter a new name <ENTER to quit>:Jill
Age: 28
Jill 28
Toyo 20
Jazz 14
Nemo 9
Press any key to continue . . .
```

Deleting a Node from the List

The deletion operation is identical to the addition operation except it is performed in reverse. Just like with the addition, the following cases need to be considered:

- The list contains only one node
- Removing the first node
- Removing the last node
- Removing a node somewhere in the list

We will examine all four cases in greater detail.

Only One Node in the List

This is very simple case. Both, the head and the tail point to the same node. The node is deleted and both head and tail are set to NULL. The following code segment illustrates this:

```
Entry* toBeDeleted;  
if (head == tail)  
{  
    toBeDeleted = head;  
    Head = NULL;  
    Tail = NULL;  
}  
delete toBeDeleted;
```

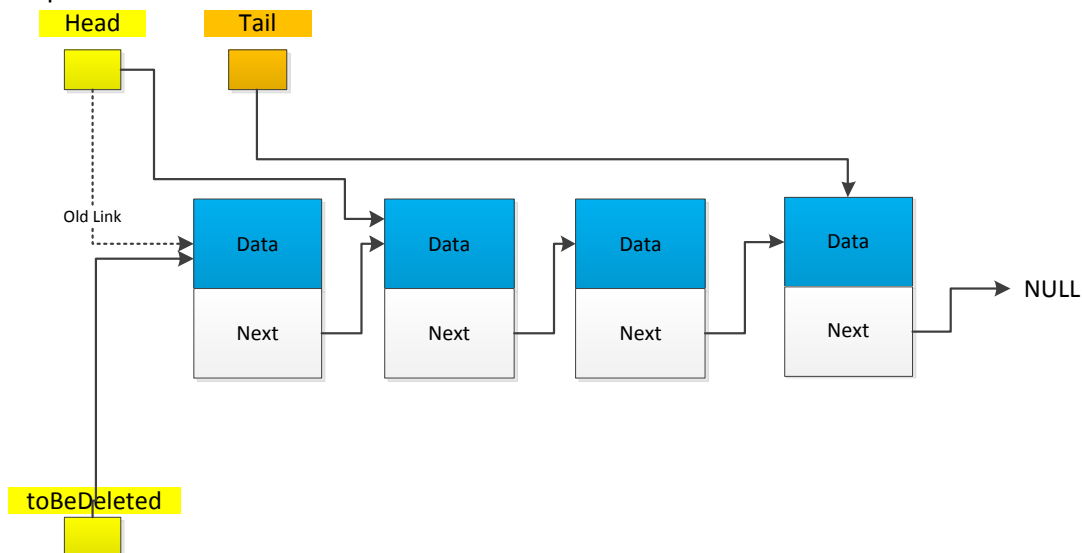
Rather than writing a separate function, this code segment is used in other delete functions.

Removing the First Node

In order to remove the first node in a single linked list, it is necessary to wire the links that point to the head pointer correctly. Usually, this is a two step process:

1. Step 1: Allocate a new pointer variable, say “toBeDeleted” and set it to the current head. Update the head pointer to point to the next node (the one that the head=>next was pointing to).
2. Step 2: delete the node pointed to by “toBeDeleted”

This is depicted below:



And the code to accomplish this is listed below:

```
bool removeFirst(Entry* &head, Entry* &tail)
{
    if (head == NULL)
        return false;
    else
    {
        Entry* toBeDeleted;
        toBeDeleted = head;

        if (head == tail)
        {
            head = NULL;
            tail = NULL;
        }
        else
        {
            head = head->next;
        }

        delete toBeDeleted;
    }

    return true;
}
```

We also need to change int main in order to test the removeFirst function. We still need to populate the list,

```
int main(int argc, char *argv[])
{
    Entry* listTail = NULL;
    Entry* listHead = buildList(listTail);
    displayList(listHead);

    removeFirst(listHead, listTail);

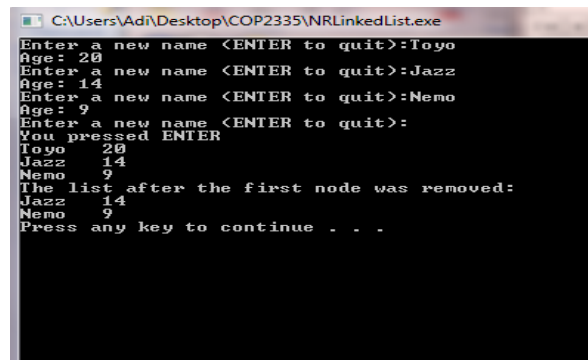
    cout << "The list after the first node was removed:" << endl;
    displayList(listHead);

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Compile and run the program with the following data:

Name: Toyo
Age: 20
Name: Jazz
Age: 14
Name: Nemo
Age: 9
Name: <ENTER>

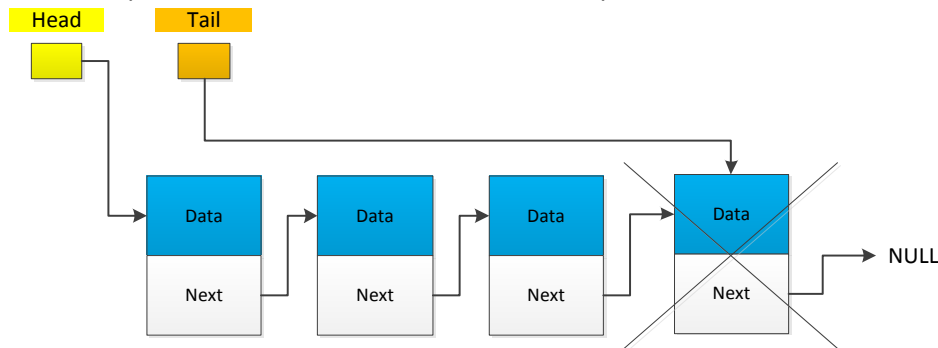
What is the output of your program? It should like something like the figure on the right:



```
C:\Users\Adi\Desktop\COP2335\NRLinkedList.exe
Enter a new name <ENTER to quit>:Toyo
Age: 20
Enter a new name <ENTER to quit>:Jazz
Age: 14
Enter a new name <ENTER to quit>:Nemo
Age: 9
Enter a new name <ENTER to quit>:
You pressed ENTER
Toyo 20
Jazz 14
Nemo 9
The list after the first node was removed:
Jazz 14
Nemo 9
Press any key to continue . . .
```

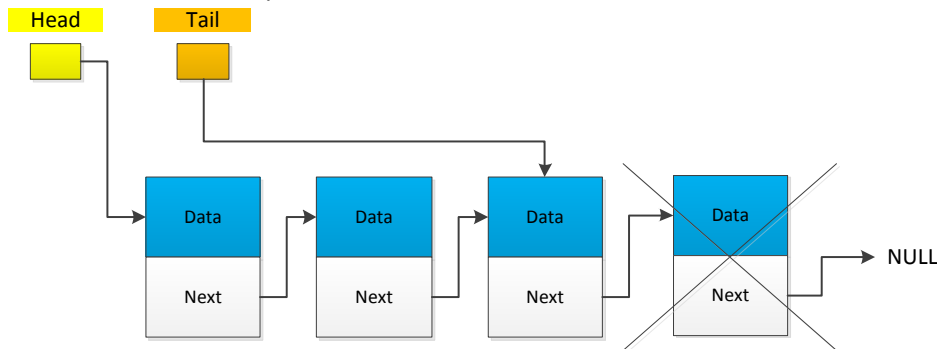
Removing the Last Node

Removing the last node entails removing the current tail node and setting the new tail node to the node that was “previous” node to the old tail. This is depicted below:

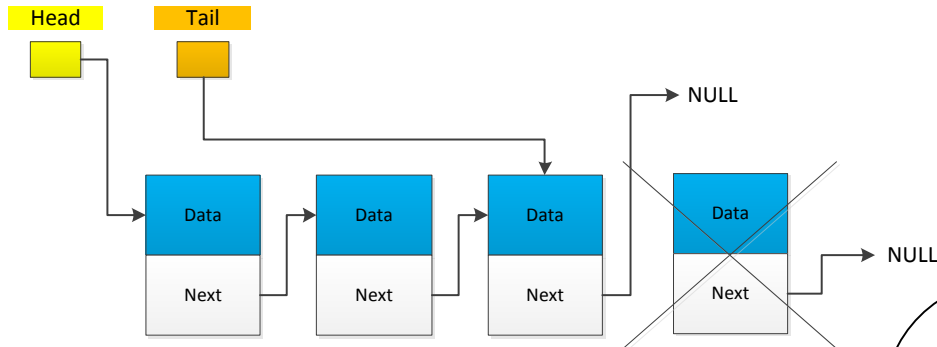


To accomplish this we need to do the following:

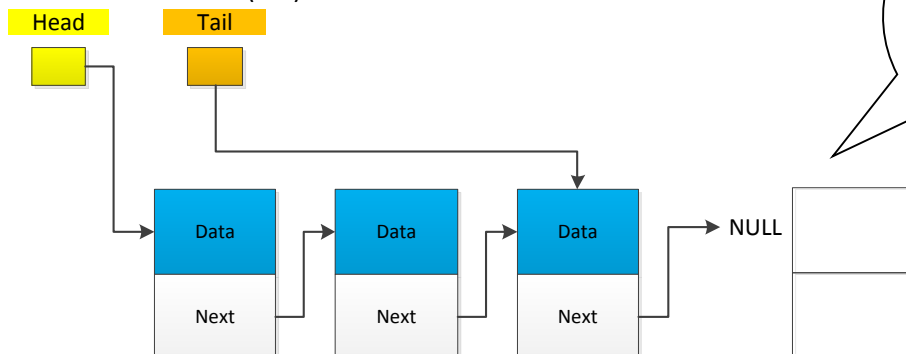
- Since this is a singly linked list, we need to start from the head of the list and traverse the list to find the node that is “previous” to the tail.



- Once we have the “previous”, we need to set the new tail’s next to NULL



- Delete the unlinked (old) tail



Let's write some code to make this happen:

```
bool removeLast(Entry* &head, Entry* &tail)
{
    if (tail == NULL)
        return false;
    else
    {
        Entry* toBeDeleted;
        toBeDeleted = tail;

        if (head == tail)
        {
            head = NULL;
            tail = NULL;
        }
        else
        {
            Entry* previous;
            previous = head;

            while (previous->next != tail)
                previous = previous->next;
            tail = previous;
            tail->next = NULL;
        }

        delete toBeDeleted;
    }
}
```

We also make small changes to the main program:

```
int main(int argc, char *argv[])
{
    Entry* listTail = NULL;
    Entry* listHead = buildList(listTail);
    displayList(listHead);

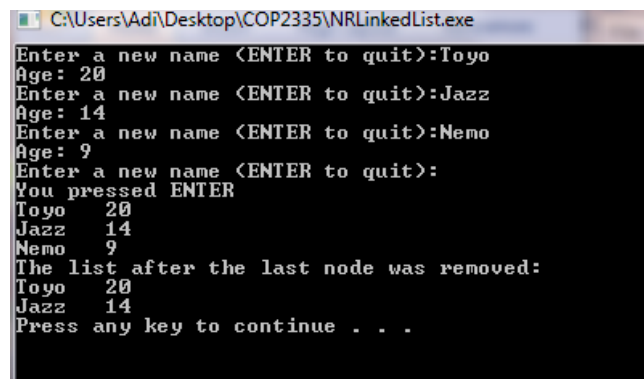
    removeLast(listHead, listTail);

    cout << "The list after the last node was removed:" << endl;
    displayList(listHead);

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Compile and run the program with the following data:

Name: Toyo
Age: 20
Name: Jazz
Age: 14
Name: Nemo
Age: 9
Name: <ENTER>

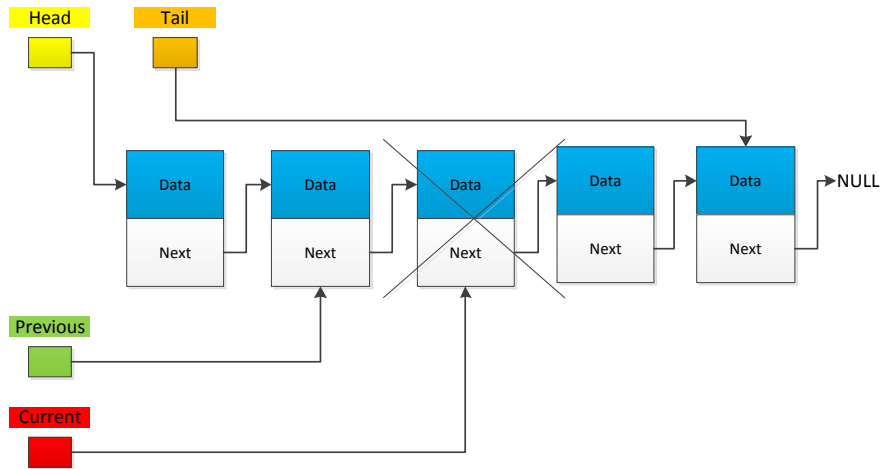


```
C:\Users\Adi\Desktop\COP2335\NRLinkedList.exe
Enter a new name <ENTER to quit>:Toyo
Age: 20
Enter a new name <ENTER to quit>:Jazz
Age: 14
Enter a new name <ENTER to quit>:Nemo
Age: 9
Enter a new name <ENTER to quit>:
You pressed ENTER
Toyo 20
Jazz 14
Nemo 9
The list after the last node was removed:
Toyo 20
Jazz 14
Press any key to continue . . .
```

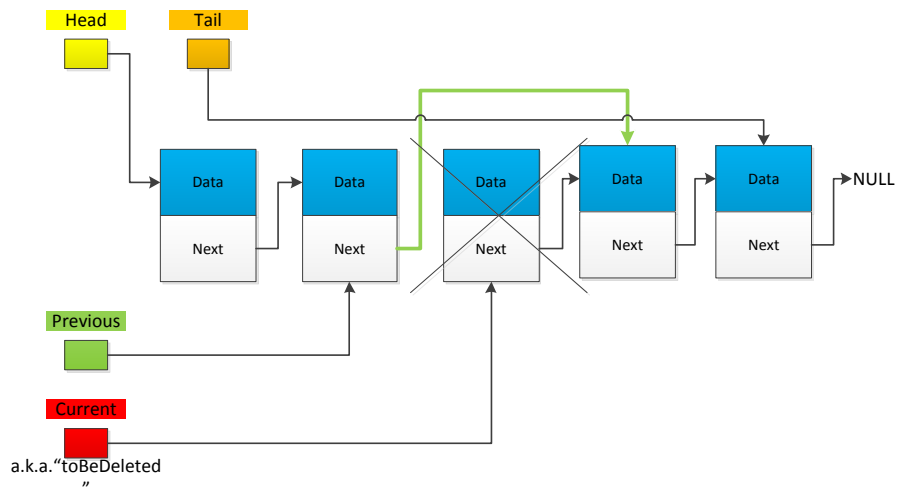
And you should get the results as depicted in the screenshot on the right. Notice that the "last" node with Nemo data was successfully deleted.

Deleting a Node in the Middle of the List

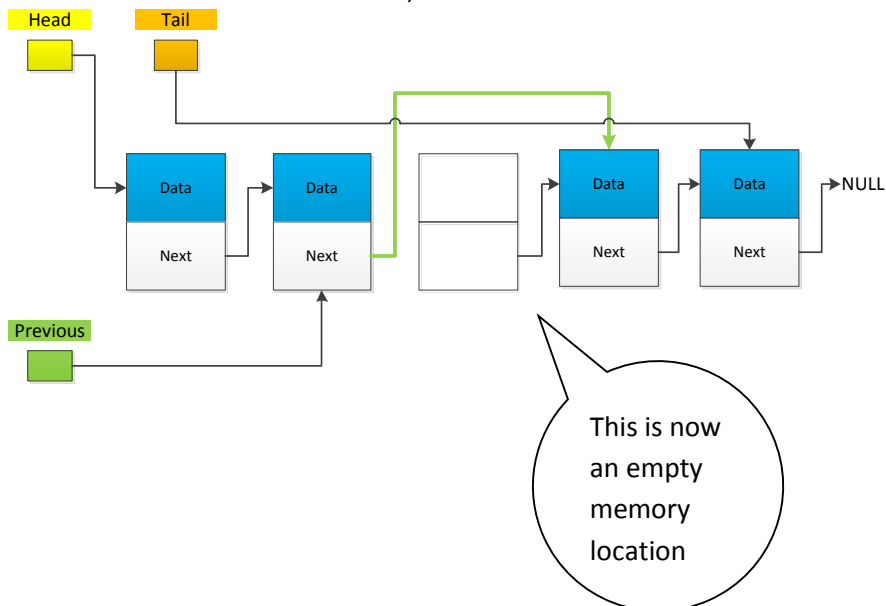
In this case, it is assumed that a node to be deleted is located between two nodes:



We can successfully delete a node in the middle of a list, by wiring the previous node's pointer to the next node relative to the current ("toBeRemoved") node:



Once we established the connections, we delete the node and free the memory location:



Finally, we write a function `removeAnywhere` to accomplish this:

```
bool removeAnywhere(Entry* &previous, int age, Entry* &head, Entry* &tail)
{
    if (previous == NULL)
        removeFirst(head, tail);
    else if (previous->next == tail)
    {
        Entry* toBeRemoved;
        toBeRemoved = previous->next;
        tail = previous;
        tail->next = NULL;
        delete toBeRemoved;
    }
    else if (previous == tail)
    {
        Entry* toBeRemoved;
        toBeRemoved = previous->next;
        previous->next = toBeRemoved->next;
        delete toBeRemoved;
    }
}
```

Since the `removeAnywhere` function removes a node when a match between members called "age" is found, we need to make a small change in `traverseList` function:

```
void traverseList(Entry* &head, Entry* &tail, Entry* newEntry)
{
    Entry* current;
    Entry* previous;
    bool found = false;

    current = head;
    previous = NULL;

    while ((current != NULL) && !found)
    {
        if (newEntry->age >= current->age)
        {
            found = true;
        }
        else
        {
            previous = current;
            current = current->next;
        }
    }

    if (!removeAnywhere(previous, newEntry->age, head, tail))
    {
        cout << "Failed to addAnywhere!" << endl;
        found = false;
    }
}
```