# Lesson 15: Singly linked lists in C

By Alex Allain

Linked lists are a way to store data with structures so that the programmer can automatically create a new place to store data whenever necessary. Specifically, the programmer writes a struct definition that contains variables holding information about something and that has a pointer to a struct of its same type (it has to be a pointer--otherwise, every time an element was created, it would create a new element, infinitely). Each of these individual structs or classes in the list is commonly known as a node or element of the list.

One way to visualize a linked list is as though it were a train. The programmer always stores the first node of the list in a pointer he won't lose access to. This would be the engine of the train. The pointer itself is the connector between cars of the train. Every time the train adds a car, it uses the connectors to add a new car. This is like a programmer using malloc to create a pointer to a new struct.

In memory a linked list is often described as looking like this:

```
----------         ----------
- Data   -         - Data   -
----------         ----------
- Pointer- - - -> - Pointer-
----------         ----------
```

The representation isn't completely accurate in all of its details, but it will suffice for our purposes. Each of the big blocks is a struct that has a pointer to another one. Remember that the pointer only *stores* the memory location of something--it is not that thing itself--so the arrow points to the next struct. At the end of the list, there is nothing for the pointer to point to, so it does not point to anything; it should be a null pointer or a dummy node to prevent the node from accidentally pointing to a random location in memory (which is very bad).

So far we know what the node struct should look like:

```
#include <stdlib.h>

struct node {
  int x;
  struct node *next;
};

int main()
{
    /* This will be the unchanging first node */
    struct node *root;

    /* Now root points to a node struct */
    root = malloc( sizeof(struct node) );

    /* The node root points to has its next pointer equal to a null pointer
       set */
    root->next = 0;
    /* By using the -> operator, you can modify what the node,
       a pointer, (root in this case) points to. */
    root->x = 5;
}
```

This so far is not very useful for doing anything. It is necessary to understand how to traverse (go through) the linked list before it really becomes useful. This will allow us to store some data in the list and later find it without knowing exactly where it is located.

Think back to the train. Let's imagine a conductor who can only enter the train through the first car and can walk through the train down the line as long as the connector connects to another car. This is how the program will traverse the linked list. The conductor will be a pointer to node, and it will first point to root, and then, if the root's pointer to the next node is pointing to something, the "conductor" (not a technical term) will be set to point to the next node. In this fashion, the list can be traversed.

Now, as long as there is a pointer to something, the traversal will continue. Once it reaches a null pointer (or dummy node), meaning there are no more nodes (train cars) then it will be at the end of the list, and a new node can subsequently be added if so desired.

Here's what that looks like:

```
#include <stdio.h>
#include <stdlib.h>

struct node {
  int x;
  struct node *next;
};

int main()
{
    /* This won't change, or we would lose the list in memory */
    struct node *root;
    /* This will point to each node as it traverses the list */
    struct node *conductor;

    root = malloc( sizeof(struct node) );
    root->next = 0;
    root->x = 12;
    conductor = root;
    if ( conductor != 0 ) {
        while ( conductor->next != 0)
        {
            conductor = conductor->next;
        }
    }
    /* Creates a node at the end of the list */
    conductor->next = malloc( sizeof(struct node) );

    conductor = conductor->next;

    if ( conductor == 0 )
    {
        printf( "Out of memory" );
        return 0;
    }
    /* initialize the new memory */
    conductor->next = 0;
    conductor->x = 42;

    return 0;
}
```

That is the basic code for traversing a list. The if statement ensures that the memory was properly allocated before traversing the list. If the condition in the if statement evaluates to true, then it is okay to try and access the node pointed to by conductor. The while loop will continue as long as there is another pointer in the next. The conductor simply moves along. It changes what it points to by getting the address of conductor->next.

Finally, the code at the end can be used to add a new node to the end. Once the while loop as finished, the conductor will point to the last node in the array. (Remember the conductor of the train will move on until there is nothing to move on to? It works the same way in the while loop.) Therefore, conductor->next is set to null, so it is okay to allocate a new area of memory for it to point to (if it weren't NULL, then storing something else in the pointer would cause us to lose the memory that it pointed to). When we allocate the memory, we do a quick check to ensure that we're not out of memory, and then the conductor traverses one more element (like a train conductor moving on to the newly added car) and makes sure that it has its pointer to next set to 0 so that the list has an end. The 0 functions like a period; it means there is no more beyond. Finally, the new node has its x value set. (It can be set through user input. I simply wrote in the '=42' as an example.)

To print a linked list, the traversal function is almost the same. In our first example, it is necessary to ensure that the last element is printed after the while loop terminates. (See if you can think of a better way before reading the second code example.)

For example:

```
conductor = root;
if ( conductor != 0 ) { /* Makes sure there is a place to start */
```

```
    while ( conductor->next != 0 ) {
        printf( "%d\n", conductor->x );
        conductor = conductor->next;
    }
    printf( "%d\n", conductor->x );
}
```

The final output is necessary because the while loop will not run once it reaches the last node, but it will still be necessary to output the contents of the next node. Consequently, the last output deals with this. We can avoid this redundancy by allowing the conductor to walk off of the back of the train. Bad for the conductor (if it were a real person), but the code is simpler as it also allows us to remove the initial check for null (if root is null, then conductor will be immediately set to null and the loop will never begin):

```
conductor = root;
while ( conductor != NULL ) {
    printf( "%d\n", conductor->x );
    conductor = conductor->next;
}
```

Still not getting it? Ask an expert!
Previous: Accepting command-line arguments
Next: Recursion
Back to C Tutorial Index

---

**Related articles**

**Learn how linked lists help with the Huffman encoding algorithm**