CSE 381: Week 02

Lists

> Every program depends on algorithms and data structures, but few programs depend on the invention of brand new ones.
>
> - Kernighan & Pike

Lists are a fundamental data structure in all strongly functional programming languages. Because of this, it is very important to understand the patterns functional languages use to deal with the many ways lists need to be used when writing code. There are many postings on the internet by people who don't understand that functional programming languages work with lists differently than other languages. Because of this lack of understanding, they draw false conclusions about the speed and memory footprint of applications written in functional programming languages.

This week we'll dive deep into how functional programming lists work and are worked on by functions. This will forearm you with acurate information. This does not mean you should engage in strong discussions or arguments regarding 'what language is best.' Such discussions are meaningless since no scope is given. These types of arguments are a form of contention.

In 1989, President Nelson gave a conference talk about how arguments destroy us. In the address he taught,

> As we dread any disease that undermines the health of the body, so should we deplore contention, which is a corroding canker of the spirit. I appreciate the counsel of Abraham Lincoln, who said: "Quarrel not at all. No man resolved to make the most of himself can spare time for personal contention. ... Better give your path to a dog than be bitten by him." (Letter to J. M. Cutts, 26 Oct. 1863, in Concise Lincoln Dictionary of Thoughts and Statements, comp. and arr. Ralph B. Winn, New York: New York Philosophical Library, 1959, p. 107.)

It is possible, and good, to have a discussion around 'my opinion as to what could be a good language' for some specific situation, but arguing over some arbitrary best language isn't pointless, it is destructive.

Internal Pieces

All languages, functional or imperative, implement lists in similar ways by simulating mathematical nodes and edges. You may recognize these terms from working with lists, trees, and graphs in your study of Discrete Mathematics, algorithms, or other mathematical reading you have done. Yet, there is a difference between those mathematical definitions and the ones implemented in computing.

Each node consists of two parts - memory for the storage of the list element and memory that indicates the location of the next node in the list. Anything can be a value in a list element. It could be data as simple as a single character or as complicated as you can imagine. Many languages allow you to store different types of elements in the same list. Others require all elements of the list to be of the same type. Regardless of if your

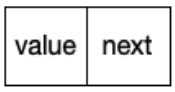

Fig.1 - The two parts of a node, the value storage and next indicator.

language's list content rules are restrictive or expansive, the list nodes have a common, basic structure.

As with all things in computing, nodes and their associated lists need to be used with discretion. If you don't need a list, don't use one. When you need a list, don't try to force something else into your code. This is a good pattern to follow for all of the data structures you will learn about during this semester and beyond.

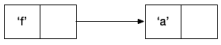

Fig.2 - A next indicator showing that the node with the element 'f' has a next node with the element 'a'.

In this course, diagrams of collections using nodes will use the arrow to indicate that a node has a next node. If there is no arrow, there is no next node and the end of the list has been reached. Figure 2 follows this pattern. In the figure there are two nodes making up the entire list. The characters 'f' and 'a' have been stored in the list. The node containing the element 'a' is the next node for the node containing the element 'f'.

It is useful to understand how these nodes are used to make lists in both imperative and functional programming languages. Especially when you are trying to decide what language or type of language would be a good option for writing all or part of a specific application.

Functional Programming Lists

A fundamental principle of functional programming is that once a value has been stored in a variable, the value can not be changed. Values can be anything. That includes lists. Yet lists need to grow, merge, and get values removed from

them. There appears to be a problem. How can the modification needs of lists align with the principle that values are fixed?

The naive, easy, but computationally and memory expensive, way is to make a copy of the list every time you want to change it. Ouch. This is not a good choice. The creators of functional programming languages know this. That's why they got outside of the box and found a solution.

Appending Lists

Consider the two lists in Figure 3. List A has the characters 'b','i',and 'g' stored in it. List B has the characters 'f','a',and 'd' stored in it.

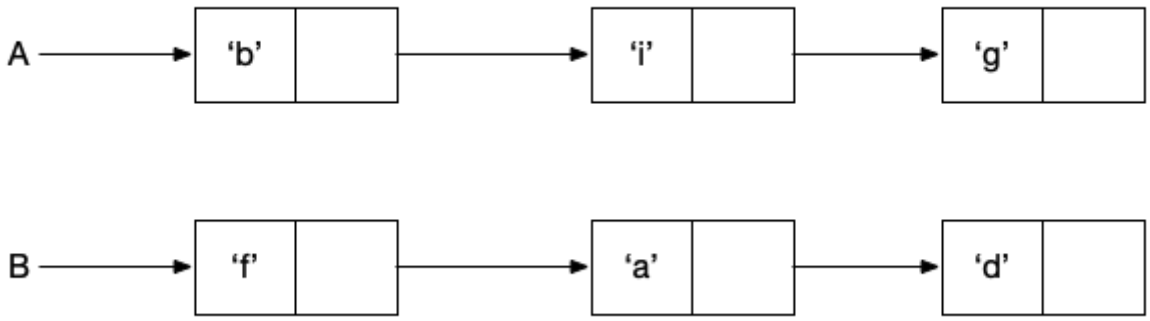

Figure 3. - Lists A and B with their elements.

Appending the lists to form a list C, `A:B→C`, using the functional programming pattern requires only copying list A, not both. List A had to be copied since the last element was changed to refer to the first node in List B. List B doesn't need to be copied since it wasn't changed.

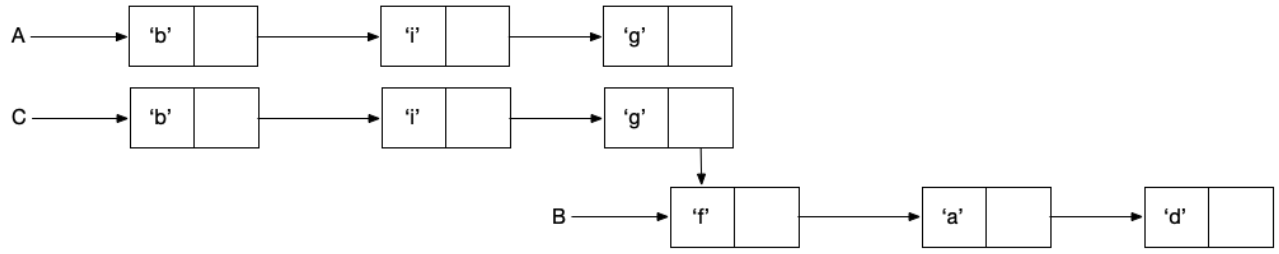

Figure 4. - List C, `A:B⟶C`.

Copying all of list A may seem like a big waste of memory and computation time, but please keep in mind that merging two lists of significant size is not a common programming behavior, let alone a functional programming one. If your app, in any language, is written to merge large lists, it is designed wrong and needs a good reworking. Let's look at another example of the impact of the accepted functional programming pattern.

Modifying a Value in a List

Say you want to change a value in a list. What happens then? Imagine you had a list named N storing the numbers 7, 13, 0, 5, -3, and 15 in that order and you wanted to change the 0 to be 6 and named the changed List U, `f::0 6 N ⟶U` where `f` is a function that modifies N. The third node, containing the value 0, has to be changed. Therefore it has to be copied and modified. That triggers a cascade of requirements to copy all of the proceeding nodes. The list U, then, consists of copies of the first two nodes, a copied node updated to contain the value 6, and three uncopied nodes from list N like you see in Figure 5.

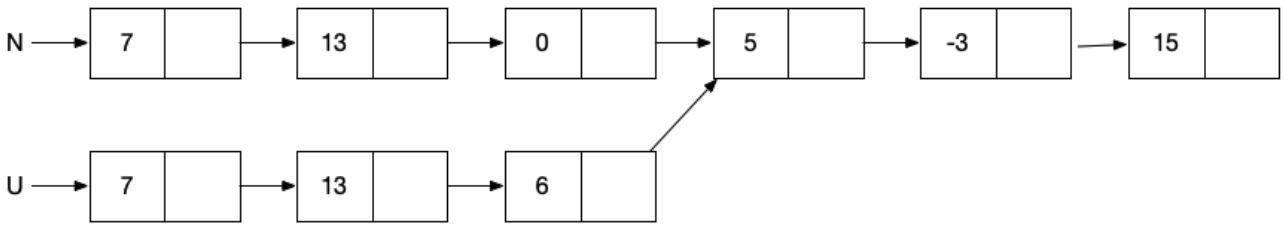

Figure 5. - List U,the modified version of list N.

Again, this type of behavior is expensive in ALL types of languages and should be avoided if at all possible in all languages. Both of these behaviors are included in the list pattern for all languages so they are available if there is no other way to create what you need. Rarely is that the case.

Prepending a Value to a List

Now let's take a look at a list behavior where functional programming really shines. Imagine you wanted to prepend the value 50 to the begining of list N and call the result P, `50:N→P`. In Figure 6 you see that no copying is required. All that is needed is to create a node to contain the value 50 and have it refer to the first node in list N. Done!

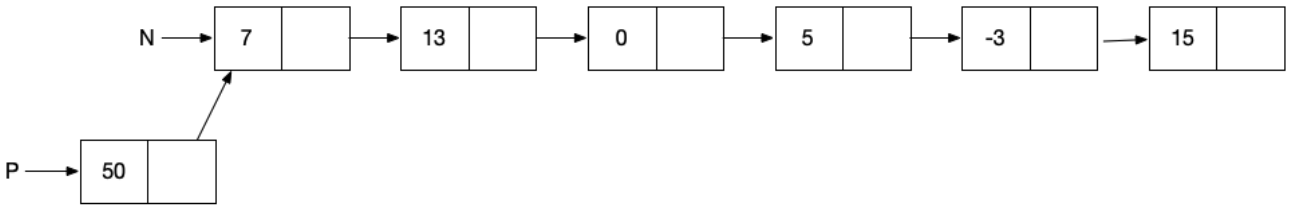

Figure 5. - The value 50 prepended to list N.

Notice that list N still has a value of 0 in its third node, not 6. That's because all modifications for lists in functional programming languages, including the value change done above, are non-destructive. Any change you make doesn't destroy the original list unlike non-functional languages such as Python. Non-destructive list modification is very helpful, especially when you are writing concurrent code. (More on that later in another course.)

Appending a Value to a List

If you are interested in concurrent computing, and let's

face it...who isn't 😎, BYU-Idaho offers a course subsequent to this one you can take. It is CSE 481.

Having seen that when you modify a value in a list a cascade of copying is triggered of all previous nodes in a list, it should come as no suprise that appending to a list triggers the same behavior. Let's append a 75 to list N and put the result in list E, `N:75⟶E.`

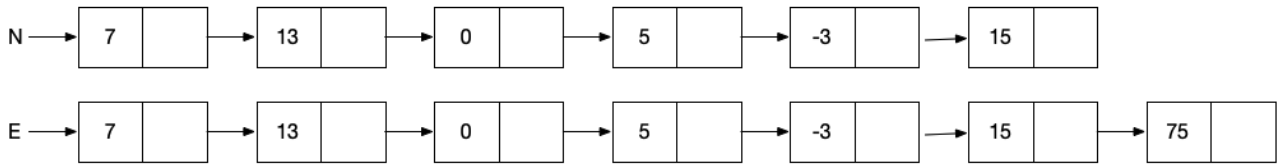

Figure 6. - The value 75 appended to list N.

Notice in Figure 6 that every node in list N was copied. This is very expensive. That's why, if you are building a list by repeatedly adding elements to the list, most functional programming languages urge you to not build lists by repeated appending. Instead, if you need the appended list ordering, they advise you to prepend each additional value to a list and then reverse the list. That makes only one copy and then when the function in which this copy is made, the memory for the original list is freed up.

This reversing of the list does require a temporary increase in the amount of memory being used and has a cost computationally. The developers of the Erlang language and compiler stated that they have resolved the 'appending problem' in their latest, as of 2020, version of Erlang. They advise Erlang programmers to preppend when that makes sense engineering-wise, and append otherwise.👍

Removing Values

Along with adding a value to the begining of a list, a very common behavior is removing, usually repeatedly as part of a recursive function, the head of a list `f::N →h|t`. Here `h` is the first element of the original list, the head, and `t` is the rest of the original list, the tail.

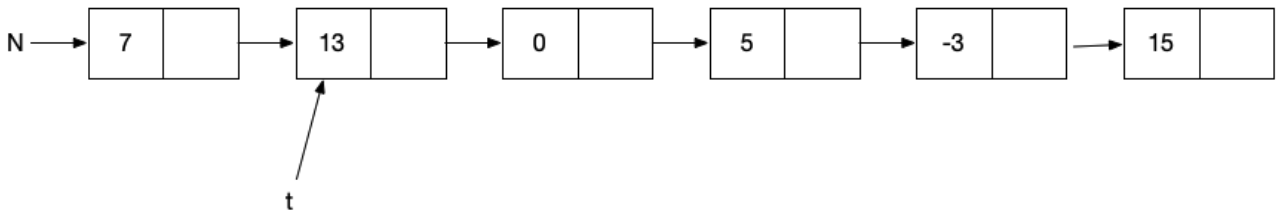

Figure 7. - The tail, `t`, of the list after retrieving the head value.

It is unsuprising that this very common behavior is highly optimized. Notice that list t contains no copies and list N has not been modified.

Removing a value from the end of a list is a very different matter. Let R be the list N with the last element removed, `f::N→R` where `f` is a function that removes the last element from N.

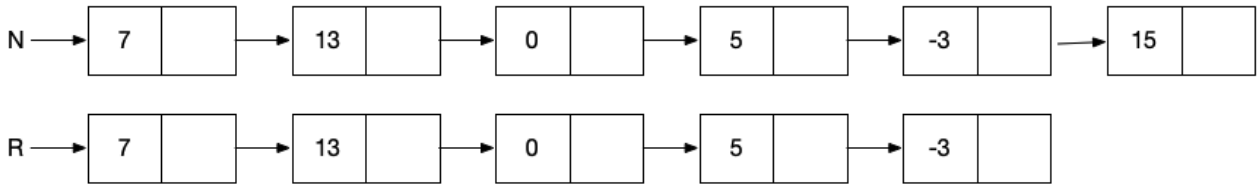

Figure 7. - R consists of copies of all of list N's nodes except the last.

For the list R to end before list N does, the next to the last element of N has to be copied and modified. This, again, triggers a cascade of copies and changes for all of the nodes of N that are before the next to the last one. This behavior is expensive and should be engineered out of applications if at all possible.

Non-Functional Programming Lists

So now, let's compare what you've learned about how functional programming languages create, modify, and use lists with non-functional programming languages of which Python is one. In these languages lists are usually NOT a fundimental part of the language. Often they are an addition in some sort of library you can choose to include in your application or not. This second-class status does not always relegate these lists to the dust bin. Let's take a look at how these languages treat lists following the same order of exploration as we did with functional programming language lists.

Before diving in please understand that values, including lists, in non-functional programming languages can be changed. Take Python as an example. There you can change the values of variables all you want. Also, when you use list methods like append, remove, pop, sort, and others, the original list is changed. This is a destructive use of the list since the list as it was originally no longer exists. Because of this, the images below will have a before and after section showing you how the list(s) have changed.

Appending Lists

Let's use the same two lists, A and B, that we did before. Just like before, A contains the characters 'b','i', 'g' and B contains the characters 'f', 'a', and 'd'. Let C be the list that is B appended to A, `A:B→C`.

What happens when C is created, is that, without duplication, the last element of list A is changed so that it refers to the first element of list B. This means that there is no list A that ends as it did previously. This is why merging lists in this non-functional way is destructive. A, by itself, can no longer be used. That's why it would be dangerous to keep the A list variable around. Using it would lead to a lot of serious bugs in the code. Destructive modification also has a lot

of serious implications for concurrent applications. That's why the A list variable has been removed from Figure 8. It's what you have to do in these languages to keep the bugs at bay.

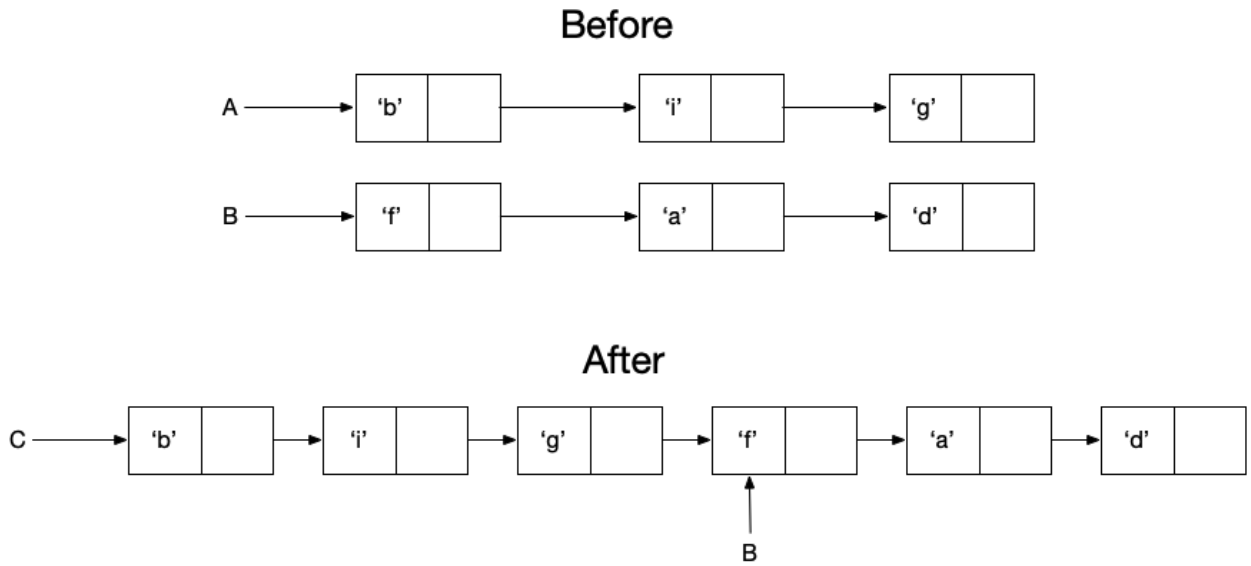

Figure 8. - List C consists of all of list A's and all of list B's nodes.

Modifying a Value in a List

This time, let's reuse the list N to show how non-functional modification works. Initially N contains the numbers 7, 13, 0, 5, -3, and 15 in that order. As before, let's change the 0 to be a 6.

What happens here is called 'modify in place.' That means no new list is created,`f:: 0 6 N→N`. Instead, the node containing the 0 value is modified to contain a 6 instead as you can see in Figure 9.

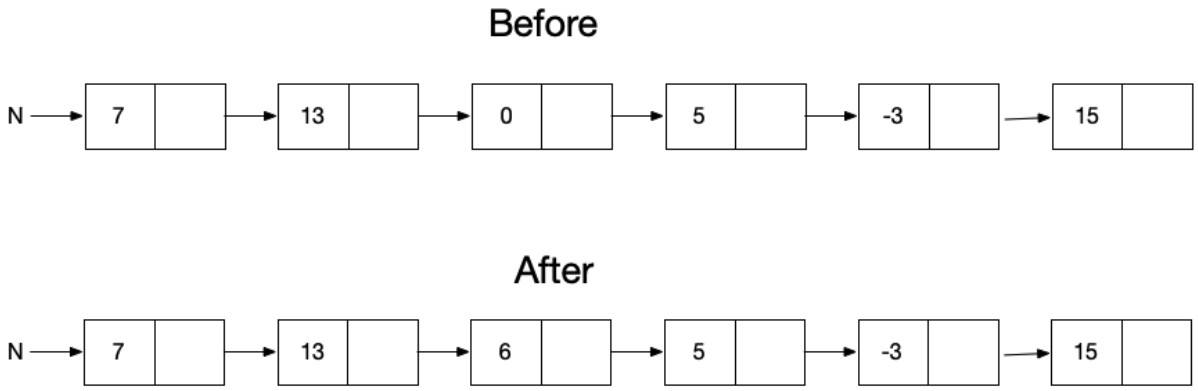

Figure 9. - List N modified in place.

Again, this is destructive modification because the original list containing a 0 value no longer exists.

Prepending a Value to a List

It is possible to do a non-destructive prepend to a non-functional programming list. This, however is rarely done since it would break the pattern. It would be bad engineering-wise to have some actions on a list be destructive and others not. It would be too hard to remember which actions do what. So prepending 50 to N is destructive, `50:N⟶N`.

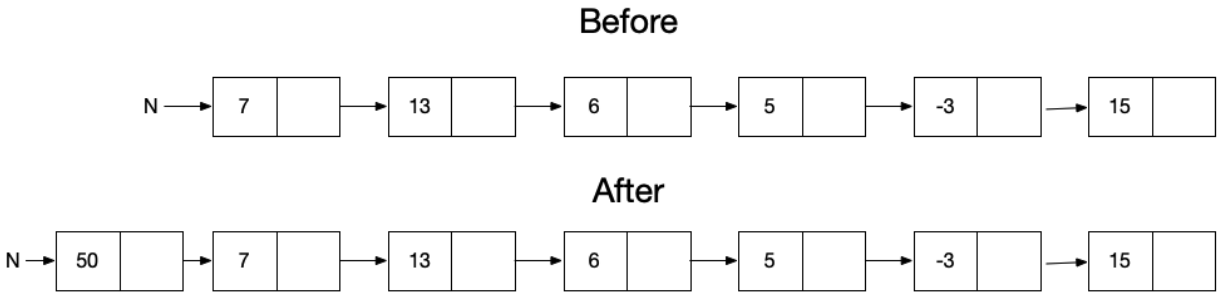

Figure 10. - List N modified in place.

Notice in Figure 10 that list N has it's 0 value replace by a 6 due to the previous modification. If you go back and examine this same change in the functional

programming section, you will find that the value is still 0 since that change was non-destructive.

Appending a Value to a List

It is no suprise that appending the value 75 to N is also destructive, `N:75⟶N`.

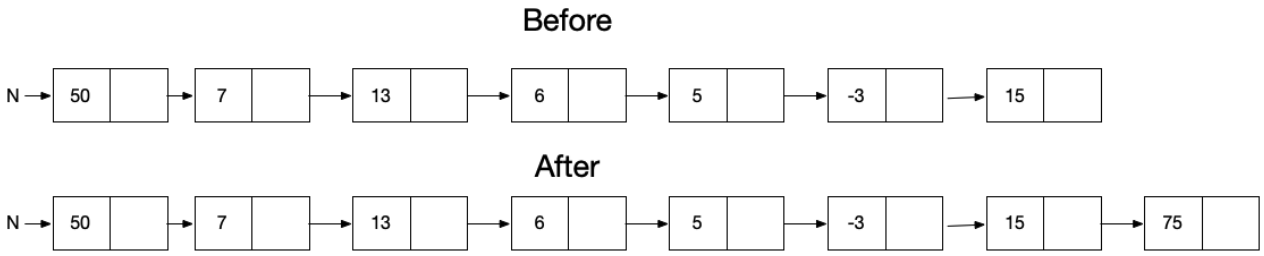

Figure 11. - List N modified in place.

Notice yet again that, in Figure 11, the list N starts off with all of the modifications due to the previous actions that are part of this tutorial.

Removing Values

Once again, let's use `h` to represent the head value. Being destructive, this behavior does not produce a tail, `f::N ⟶h|N`.

In this case, the N variable is changed to refer to what was the second node in the list and the memory for the first node is freed up for reuse after the value it contains is stored in a variable.

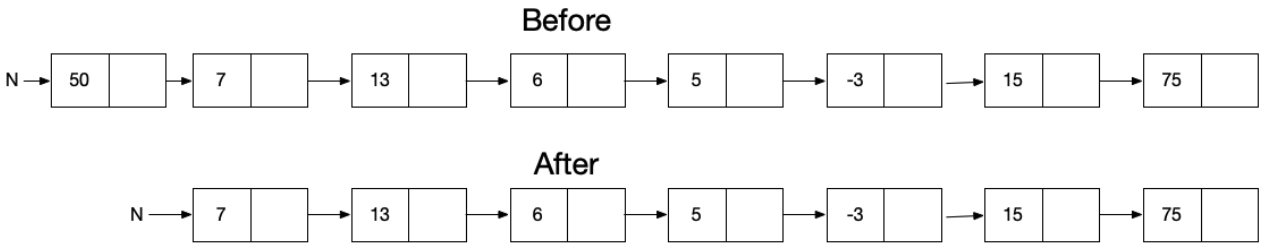

Figure 12. - The list N after removing the head value.

Lastly let's see how to destructively remove a value from the end of a list, `f::N⟶N`.

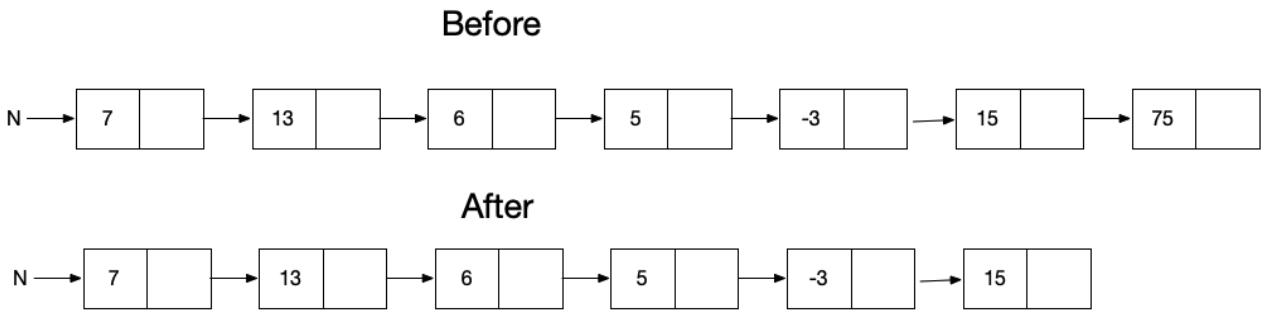

Figure 13. - N is the result of removing its last node and freeing the memory for that node.

Here, once again, the memory for the removed node is freed for reuse. Also, what was previously the next-to-last node in N is modified to refer to nothing. That way the end of the list is known. Not doing this is a common bug that can lead to crashes and security exploits.

Wrap Up

So there you have it. These are some of the most common behaviors with lists. There are others, for example sorting and inserting or removing something from the middle of a list, but those behaviours follow the same sort of things you've seen here in this reading. There is a lot that you've been shown here. It will probably take a good amount of time with you playing around with sketches of lists to more completely understand the implications of how functional and non-functional lists are treated in various languages.