Dynamic Arrays and Linked Lists

Building Blocks

- An important concept in this course is that of the ***abstract data type*** (or ***ADT***). An ADT is simply a data type as viewed by the *user* of that data type (e.g. a programmer using it in their code).

- In particular, each ADT consists of a well-defined interface that consists mainly of the operations of a data type with which the user interacts, and these operations are specified in terms of their behavior (i.e. how each operation affects the data).

- And ADT is "abstract" because it is an implementation-independent view of the data type. In other words, when thinking about ADTs, we don't really care how they're implemented, e.g. how data is actually stored in memory or what algorithms will be used to implement the operations. We only care about the high-level behavior of each of the ADT's operations.

- We will explore several common ADTs in this course, including stacks, queues, trees, priority queues, graphs, etc.

- However, it will also be our goal in this course to *implement* many ADTs. At this point, we will need to start worrying about implementation details, including specifically how data is stored in memory, what algorithms are used to implement operations, etc. This view of a data type, from the *implementor's* perspective, is known as a data structure.
    - In other words, data structures serve as the building blocks of ADTs.

- Two of the most essential of these building blocks are the ***dynamic array*** and the ***linked list***. We'll be able to use these to build a number of different ADTs. Thus, these are the first data structures we'll explore here.

Dynamic arrays

- Arrays are a very useful data type. One of the main reasons arrays are so useful is the fact that they are stored in a contiguous block of memory (i.e. a collection of memory where the memory blocks have sequential/consecutive addresses).

- Because of this, arrays allow ***direct access*** (also called ***random access***) to the data they contain. That is, they allow each individual element of data to be accessed directly in the same amount of time it takes to access any other element of the data, regardless of how many individual elements are stored.

- For example, just think about a large array allocated in a C program, e.g.:

```
int* array = malloc(1000000 * sizeof(int));
```

- In this array, we can access any element simply by using its array subscript. Moreover, it will take the same amount of time to access any individual element, regardless of where in the array that element lives. For example, these two instructions will take the same amount of time:

```
array[0] = 0;
array[999999] = 0;
```

- One of the main drawbacks of a standard array (technically called a ***static array***) is the fact that they have a fixed size, and this size must be specified when the array is created (e.g. specified in the call to `malloc()` in a C program).

- This might be a problem if we don't know exactly how much data we'll want to store in the array. For example, if we end up needing to store more data than we initially allocated memory for, then we will need to allocate more memory.

- A dynamic array is a data structure that doesn't have a fixed capacity, like a static array. Instead a dynamic array has a variable size and can grow as needed as more elements are inserted into it.

- In other words, a dynamic array is an array that hides the details of managing the underlying memory storage (e.g. allocating more memory when needed) behind a simple interface.

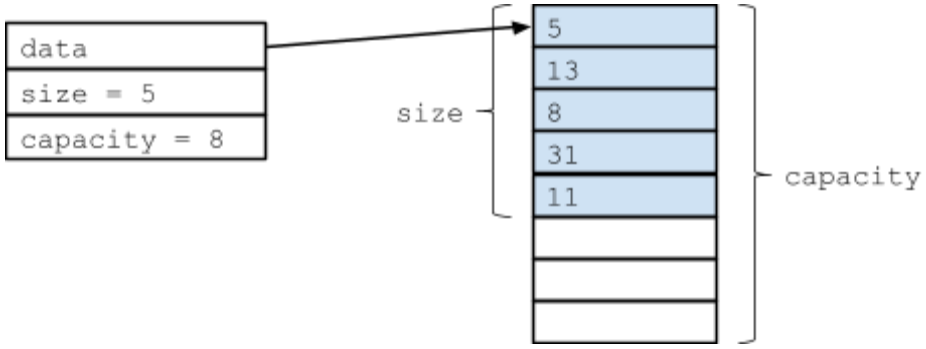
- Specifically, the following are the typical operations in a dynamic array's interface:
    - `get` – Gets the value of the element stored at a given index in the array
    - `set` – Sets/updates the value of the element stored at a given index in the array
    - `insert` – Inserts a new value into the array at a given index.

        - This is different from `set` in that no existing array values are overwritten by `insert`. Instead, `insert` moves all subsequent elements down one spot in the array to make room for the inserted one.
        - Sometimes, dynamic array implementations limit insertion to a specific location in the array, e.g. only at the end.
    - `remove` – Removes an element at a given index from the array and moves all subsequent elements up one spot in the array to fill in the "hole" left by the removed one.
        - Sometimes, dynamic array implementations avoid moving elements up a spot by only allowing the last element to be removed.

- Dynamic array implementations typically work as follows:
    - A regular array of known capacity is maintained for underlying data storage by the dynamic array.
    - As elements are inserted into the dynamic array, they are simply stored in this underlying data storage array.
    - If an element is inserted into the dynamic array, and there isn't capacity for it in the underlying data storage array, the capacity of the underlying data storage array is doubled. Then the new element is inserted into this larger data storage array.

- To implement a dynamic array, we need to keep track of three things (e.g. in a C struct):
    - `data` – The underlying data storage array for the dynamic array.
    - `size` – The number of elements currently stored in the array.
    - `capacity` – The total number of elements the underlying data storage array has space for before it must be resized.

- Assuming for simplicity that we are storing integer values, a dynamic array in use would look something like this:



- In the diagram above, `data` is simply a pointer to dynamically allocated C array.

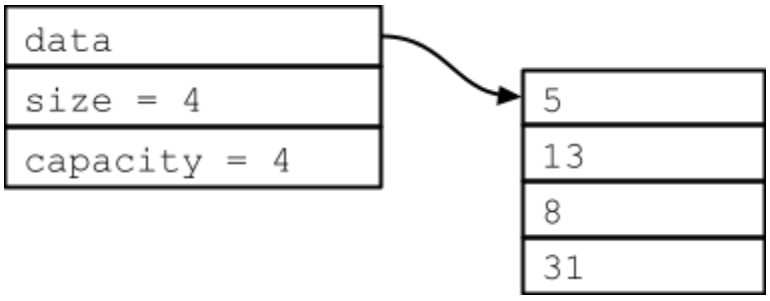
Inserting an element into a dynamic array

- Let's spend a minute exploring in more detail how insertion into a dynamic array works in practice.

- Most of the time, insertion into a dynamic array is a simple operation. Specifically, if the dynamic array's `size` is less than its `capacity`, then we know there's at least one free spot in the dynamic array's underlying data storage array (i.e. `data`). In this case, we can simply insert the new element into the available empty space.

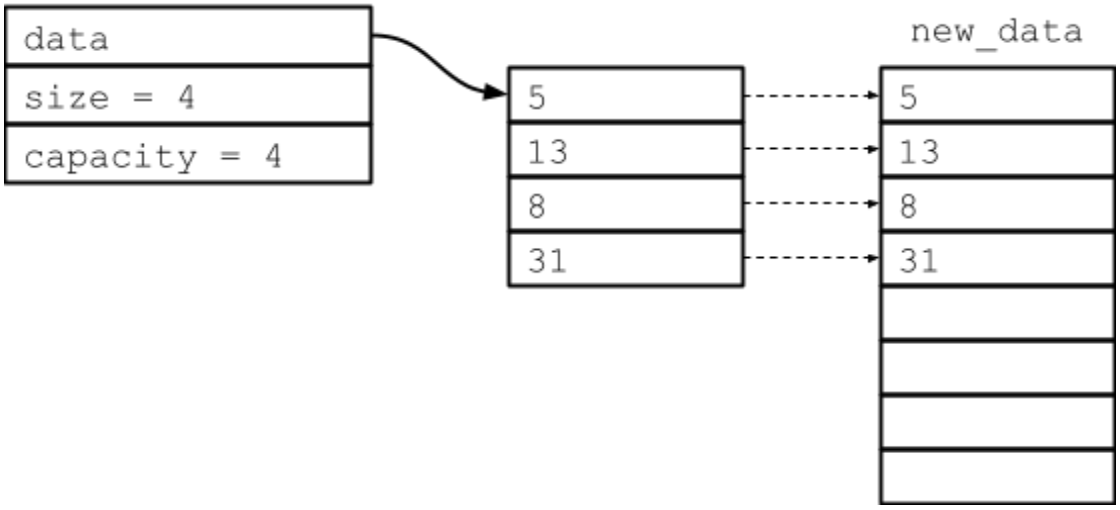
- A more challenging situation is when a user wants to insert into a dynamic array whose `size` is *equal* to its `capacity`. In this situation, there isn't any available space in which we can just insert the new element, so we need to increase the capacity of the underlying data storage array (i.e. `data`) in order to *make* space for the new element.

- In this situation, where `size` is equal to its `capacity`, insertion into the dynamic array requires several steps. We'll walk through an example to illustrate all of them. Imagine that before the `insert` operation was called, our dynamic array looked like this (again, assuming for simplicity that we're only storing integers):

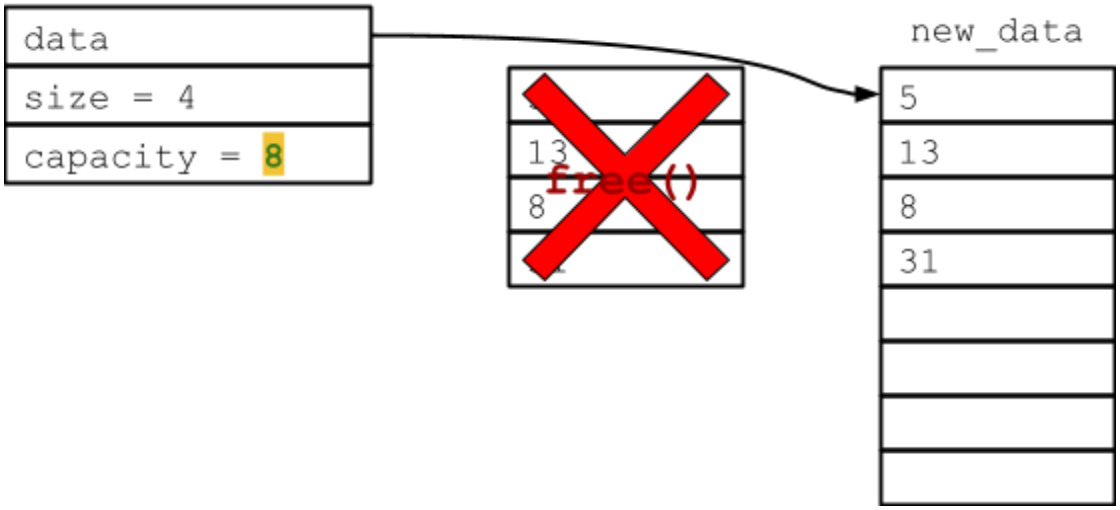


- Let's now imagine that the user calls the `insert` operation to insert the value 16 at the end of the array. Our first step here will be to allocate a new array that has twice the capacity of the current underlying data storage array. For the moment, this new array will not be part of the dynamic array structure itself. In other words, the underlying data storage array (i.e. `data`) will not change quite yet.

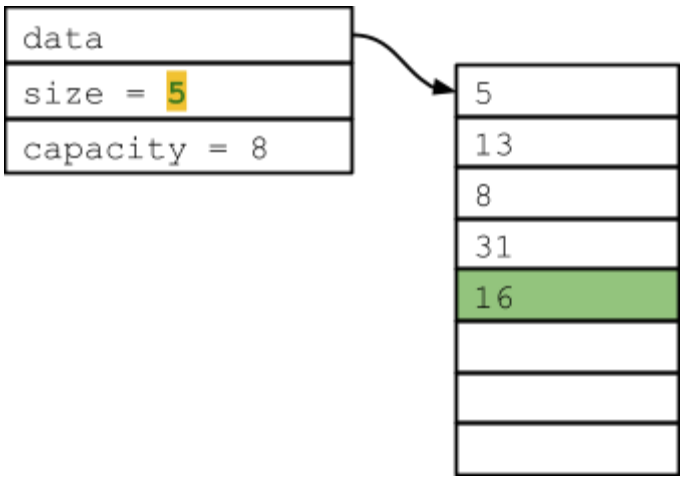
- Once the new, bigger array (let's call it `new_data`) is allocated, we'll copy all of the values from the current data storage array into the corresponding locations in the new array (i.e. copying from `data` to `new_data`):



- Once the stored values are copied over to the new array, we actually no longer need the old data storage array, so we can free it and update the dynamic array to use the new array (i.e. `new_data`) as its underlying data storage array (updating the dynamic array's capacity accordingly):



- Finally, with the additional capacity we've gained by doubling the size of the underlying data storage array, we have space to insert the new value. We can do that now, increasing the dynamic array's size by 1 to reflect the new stored value:



- This functionality enables us to handle all possible situations in which the `insert` operation is used: either the underlying data storage array has capacity for the element being inserted or it doesn't.

- We'll formally analyze the *performance* of the dynamic array a little later in the course. For now, though, try to think about what some advantages of the dynamic array are and what some of its disadvantages are.

Linked lists

- Linked lists (like dynamic arrays) are a ***linear*** data structure. In other words, data in a linked list (like data in a dynamic array) forms a linear sequence, with the individual data elements placed one after the other within the data structure.

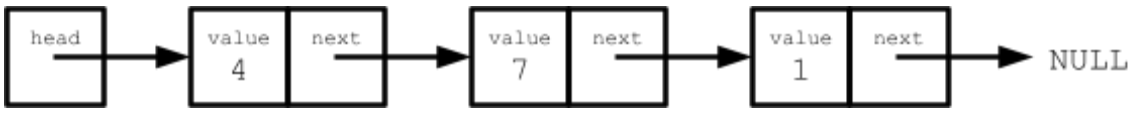
- A linked list differs from a dynamic array, however, in that each individual value in a linked list is stored in its own small structure called a ***link***, and these individual links are chained together into a sequence by having each one "point" to the next (and sometimes the previous) link in the list.

- In other words, each link in a linked list stores exactly one value and (at a minimum) points to the next link in the list. Thus, a simple link structure needs at least two fields:
    - `value` – This is where the value associated with the link is stored.
    - `next` – This points to the next link in the list (or to `NULL`, if there is no next link).

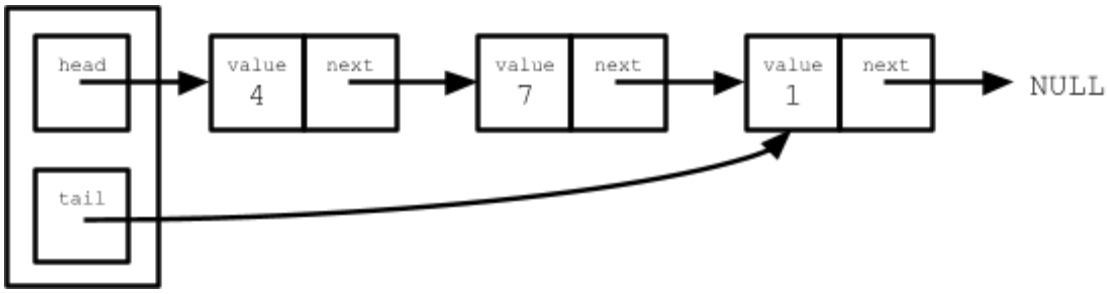
- A linked list in which each link points only to the next link in the list is known as a ***singly-linked list***.

- Importantly, a linked list always contains exactly as many links as it has stored values, and links are allocated and freed as values are added and removed, respectively, from the list.

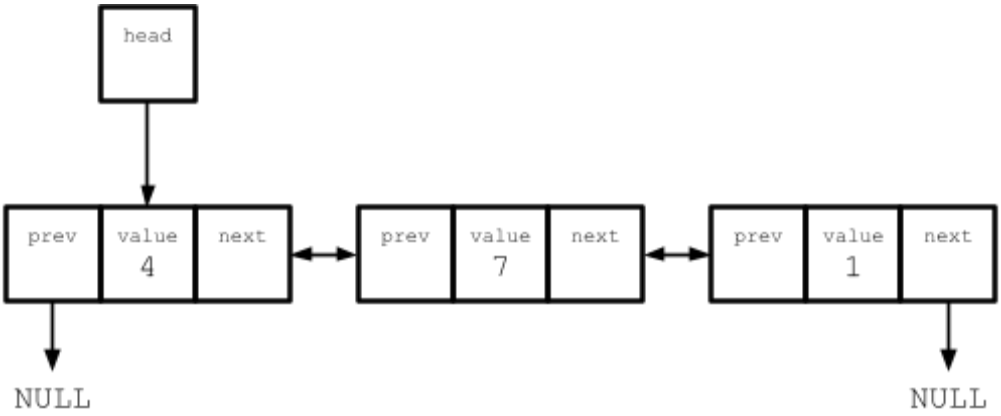
- The simplest form of linked list keeps track only of the first element in the list, which is known as the ***head*** of the list. For example, here's what a singly-linked list would look like if we were keeping track of just the head (e.g. in a pointer called `head`):



- There are several variations on this simple singly-linked list. One of these involves keeping track of both the head of the list and the ***tail***, or last element. This allows us to have easy access to both ends of the list. It would look something like this:

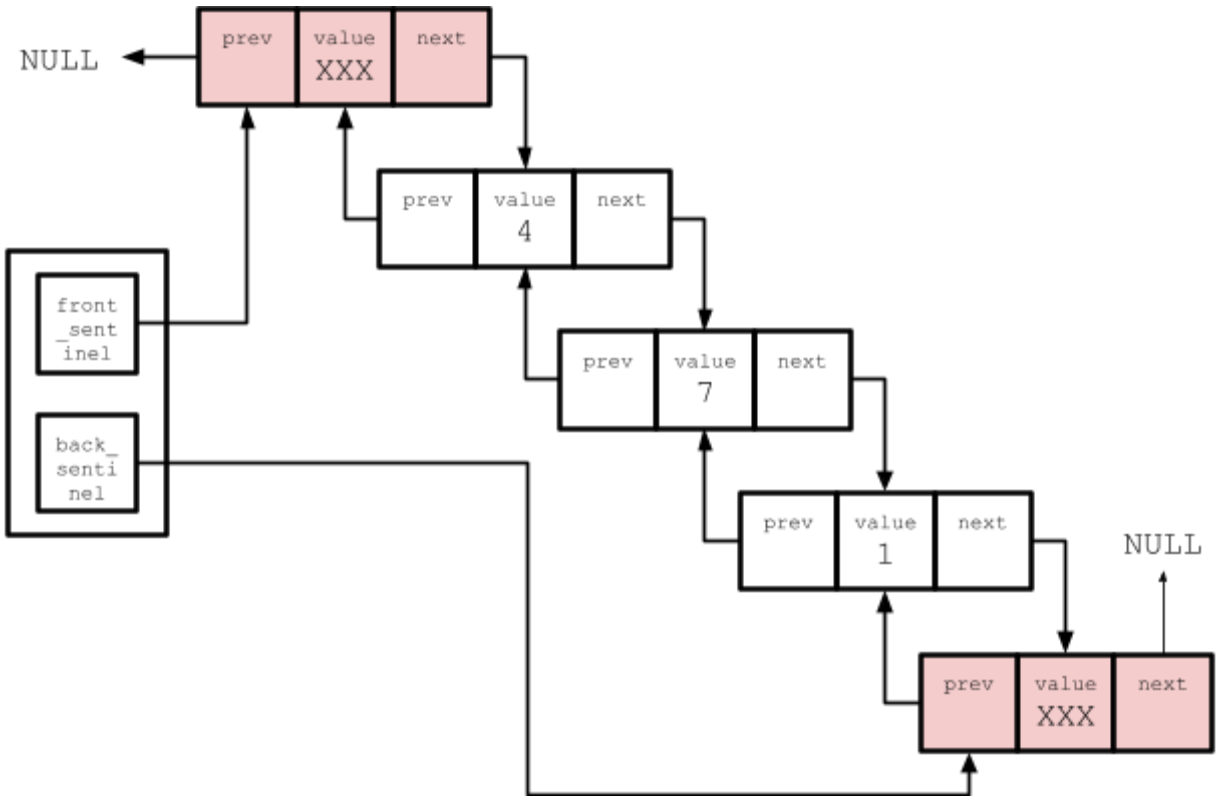


- Another variety of linked list is what's known as a ***doubly-linked list***. In a double-linked list, each link keeps track of both the next link *and* the previous link in the list:



- In a doubly-linked list, it's easy to move both forward and backward in the list, whereas in a singly-linked list, it's really only possible to move forward from one link to the next one.

- A final variant of the linked list adds ***sentinels***, which are simply special links/values that are used to designate the end of the list. Sentinels can be used to designate either the front of the list (i.e. the head) or the back of the list (i.e. the tail), or both. Here's an example of a doubly-linked list using both front and back sentinels:



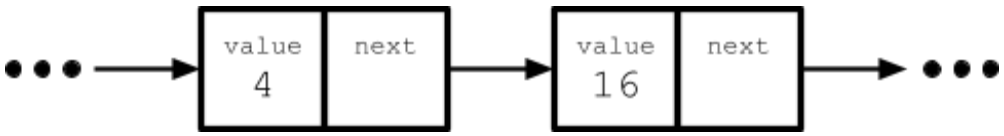
- As we'll see a bit later in the course, using sentinels can simplify some operations within the list by eliminating the need to check for special cases. In particular, a list using sentinels never contains zero links, since the sentinel links are permanently stored in the list. This means, for example, that in a list using sentinels, we never have to check whether the head is NULL.

Inserting an element into a linked list

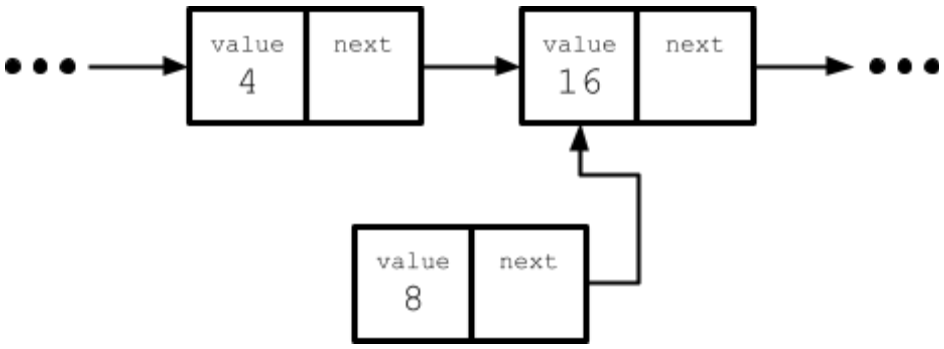
- Regardless of the variety of linked list we're working with, inserting a new element works pretty much the same, with only a few minor tweaks needed to account for the different varieties of list.

- We can always insert a new element at any point in the list, either between two existing links, at the head of the list, or at the tail of the list. We'll step through an example here where we insert a new link between two existing links, pointing out the ways insertion would differ if we were inserting at the head or tail.

- Let's start by imagining we have a singly-linked list containing (at least) the following two links (again, assuming for simplicity that we're just storing integers):

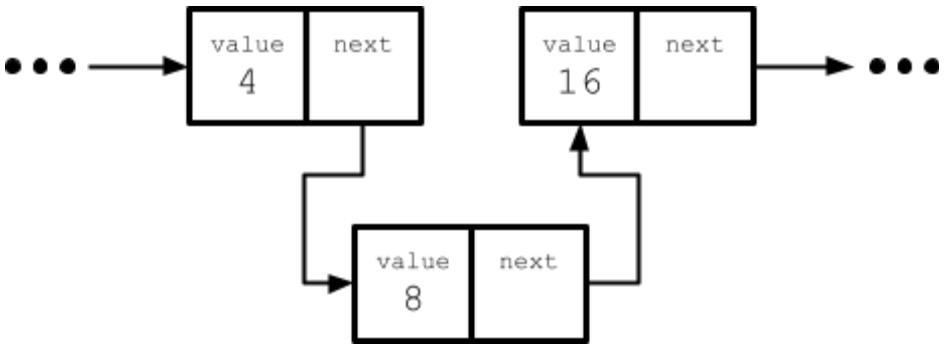


- Let's now imagine that we want to insert the value 8 between these links. Our first step would be to allocate a new link and set its `value` and `next` fields appropriately. We want the new link to live *between* the above two existing links, so its `next` field will point to the link containing the value 16:



- Note here that if we were inserting the value 8 at the end of the list (i.e. at the tail), the link containing 4 would have a `next` field that pointed to NULL, so our new link containing 8 would also have its `next` field set to point to NULL. In other words, regardless of where a new link is being inserted, it's `next` field will always be set to point to the same place as the `next` field of the link *after which* the new one is being inserted.

- Currently, our new link, containing the value 8, is not yet fully inserted into the list. In particular, if we were stepping through the list, we couldn't actually *get to* the new link from any other. To fully insert the new link, we have to update the `next` field of the link before it to point to the new link. In our example here, this means we'll have to update the link containing 4 to point to our new link containing 8 (instead of the one containing 16, which it used to point to):



- Again, note that if we were inserting a new link at the head of the list, we'd update the entire list's `head` pointer to point to the new link, since there would be no other link before the new one.

- Now the new value is fully inserted into the list, as we can both reach it from a previous link and reach the next link from it.

- If we were working with a doubly-linked list, we'd also have to appropriately update the `prev` fields of the new link and the one after it.

Removing an element from a linked list

- Removing an element in a linked list takes essentially the opposite steps as inserting a new one. Again, let's walk through an example to see how this works.

- Imagine we're working with a singly-linked list that contains (at least) the following links:

- Now let's imagine that we want to remove the link containing the value 8. Our first step would be to update the `next` field of the *previous* link to point *around* the one being removed. In our example, this means we'd update the `next` field of the link containing the value 4 to point to the link containing the value 16:

- If there was no link before the one being removed here (e.g. the link containing 8 was the head of the list), we'd simply update the list's `head` pointer to point around the link being removed. Similarly, if we were removing the link at the end of the list (i.e. the tail), we would update the previous link's `next` field to point to NULL.

- At this point, the link that we're removing is effectively taken out of the list, since we can't reach it from any other link. All we need to do now is free that link:

- Note that we didn't even need to make any modifications to the link being removed (including updating its `next` field), since freeing it wipes out the values of its fields.

- Again, if we were working within a doubly-linked list, we'd have to update the `prev` field of the link after the one being removed.

- Just like with the dynamic array, we'll more formally analyze the *performance* of the linked list a little later in the course. Again, though, spend a minute now to think about what some of the advantages and disadvantages of the linked list might be.