Stacks, Queues, and Deques

Linear ADTs

- The first three ADTs we'll explore in this course each hold a linear collection of data, i.e. data in which the elements form a sequence. These ADTs are the ***stack***, the ***queue***, and the ***deque***.

- While stacks, queues, and deques are all linear ADTs, as we'll see, they each impose different constraints that control how elements are ordered within them. Thus, each of these ADTs is useful in different situations than the others.

Stacks

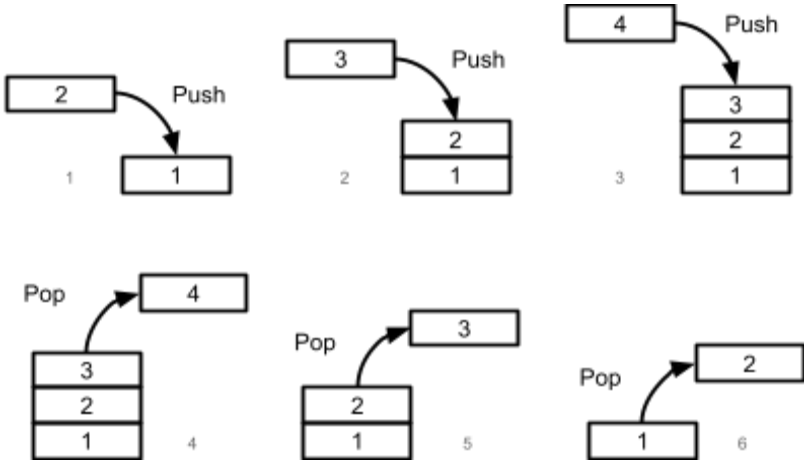
- A stack is a linear ADT that imposes a ***last in, first out*** (or ***LIFO***) order on elements. In other words, in a stack, the last element that was inserted must always be the next one that's removed.

- Because of the LIFO ordering it imposes, the stack ADT behaves much like a physical stack in the real world, e.g. a stack of books or a stack of dishes. In particular, in a physical stack in the real world, items may only be placed on top of the stack, and only the top item on the stack may be removed.

- Similarly, we can think about an instance of the stack ADT as having two ends: the top and bottom. Just like with a physical stack, new elements can only be inserted at the top of an instance of the stack ADT, and only the element at the top may be removed.

- The stack ADT supports two main operations:
    - ***Push*** – inserts an element on top of the stack.
    - ***Pop*** – removes the top element from the stack.

- Visually, a stack looks something like this:



- Here, the value 1 is already in the stack when the value 2 is pushed onto it. The values 3 and 4 are subsequently pushed onto the stack before three values are popped.

- Again, note that when a value is popped from the stack, it is always the top value (i.e. the most recently pushed) that is removed. This results in the characteristic behavior that values are popped from a stack in the *opposite* order they were pushed.
    - For example, above, values are pushed in the order 2, 3, 4, but when they are popped, they are removed in the opposite order: 4, 3, 2.

- You can probably think of several real-world applications of a stack. Some straightforward examples are a web browser's "back" history, or the "undo" operation in a text editor.

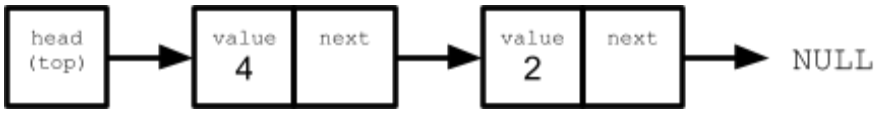
- A stack can be implemented using either a dynamic array or a linked list as the underlying data storage mechanism.

Implementing a stack using a linked list

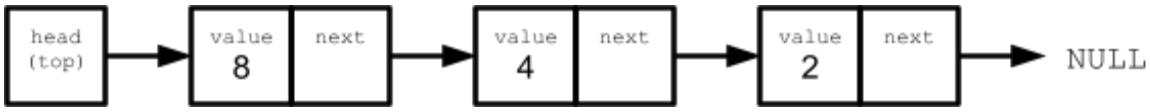
- Implementing a stack using a linked list is very straightforward. In particular, we can implement a stack using a singly-linked list, where the head of the list corresponds to the top of the stack.

- When a value is pushed into a linked-list-based stack, it simply becomes the new head of the list. When a value is popped, the current head of the list is removed, and the next link becomes the new head.

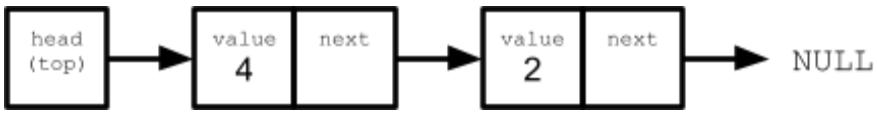
- For example, the list below represents a stack containing two values. In particular this list represents a stack with the value 4 at the top and the value 2 underneath:



- If the user ran the operation `push(8)` on this stack, the value 8 would become the new head of the underlying list, which would look like this:



- If the user then called the operation `pop()`, the value 8 would be removed as the head of the list, and the list would again look like this:



- If the user called `pop()` again, the value 4 would be removed from the list, and a subsequent call to `pop()` would result in the value 2 being removed and the list (and hence also the stack) being empty.

- Because the `push()` and `pop()` operations of a stack built on top of a linked list simply manipulate the head of the list, they are both *O(1)* operations (best-case, worst-case, and average).

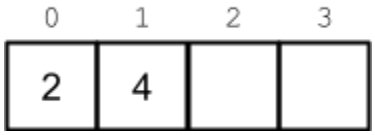
Implementing a stack using a dynamic array

- Implementing a stack using a dynamic array is also straightforward. When using a dynamic array as the underlying storage, it is easiest to use the *end* of the array to represent the head of the stack.

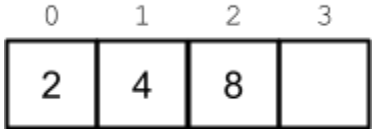
- In particular, when a new element is pushed onto a stack that's using a dynamic array for its underlying storage, that element is inserted at the end of the array. When an element is popped from the stack, the array's last element is removed.

- As elements are inserted into the stack's underlying dynamic array, the array is allowed to resize itself just as a normal dynamic array does.

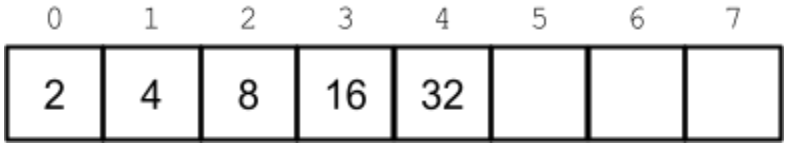
- For example, the array below represents the same original stack we looked at above when exploring linked list-based stacks. In particular, it contains two elements, with the value 4 at the top of the stack and the value 2 underneath:



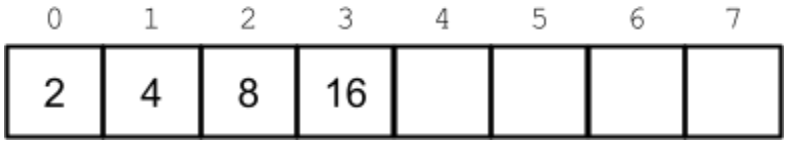
- If the user called the operation `push(8)` on this stack, the resulting underlying array would have the value 8 inserted at the end:



- If the user then again called `push(16)` and `push(32)` (in that order), those two values would each also be inserted at the end of the underlying array. The second of these operations (`push(32)`) would result in the array being resized, since it would not have enough capacity to hold the value 32:



- A call to `pop()` would result in the value 32 being removed from the end of the array:



- Further calls to `pop()` would first result in 16, then 8, then 4, and finally 2 being removed from the array.

- Again, because the `push()` and `pop()` operations of a stack built on top of a dynamic array simply insert and remove values to and from the end of the array, they are both *O(1)* operations (best-case, worst-case, and average for `pop()` and average-case for `push()`).

Queues

- A queue is a data structure that imposes a ***first in, first out*** (or ***FIFO***) ordering on the elements it stores. In other words, unlike with a stack, the first element to be removed from a queue is the first one that was placed into it. Before an element can be removed from a queue, it must wait for the removal of all other elements that were inserted into the queue before it.

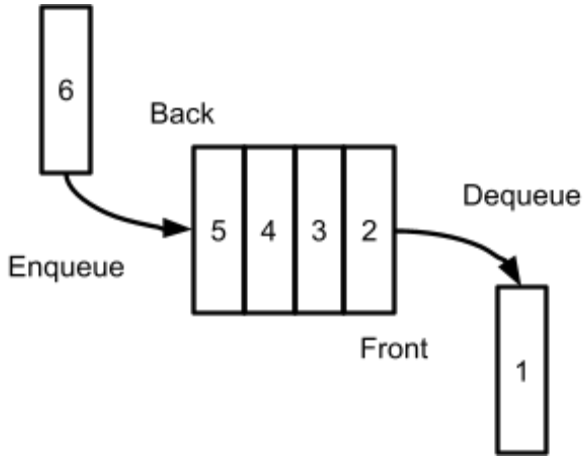
- Thus, a queue ADT works much like a line of people in the real, physical world, such as a line of people waiting to check out at the grocery store or a line of people waiting to ride a ride at an amusement park.
    - Indeed, a line like this is often called a queue in other countries.

- Specifically, in a real-world line of people, e.g. at an amusement park, everyone rides a ride in the order in which they entered the line for that ride, with the first person in the line being the first person who gets to ride, the second person in line the second who gets to ride, and so forth. Each person in the line has to wait to ride until all of the people who entered the line before them get to ride.

- This is exactly the way a queue ADT works. In particular, we can think of a queue ADT as having two ends: a front and a back. Every time a new element is inserted into the queue, it is inserted at the back, and every time an element is removed from the queue, it is removed from the front.

- The queue ADT supports two main operations:
    - ***Enqueue*** – inserts an element at the back of the queue.
    - ***Dequeue*** – removes the front element from the queue.

- We can visualize a queue like this:



- Unlike with a stack, elements are dequeued from a queue in the same order in which they are enqueued. For example, if the values 1, 2, and 3 are enqueued in that order, they will also be dequeued in the same order: 1, 2, 3.

- For this reason, queues also have many real-world applications, such as scheduling, I/O buffering, etc.

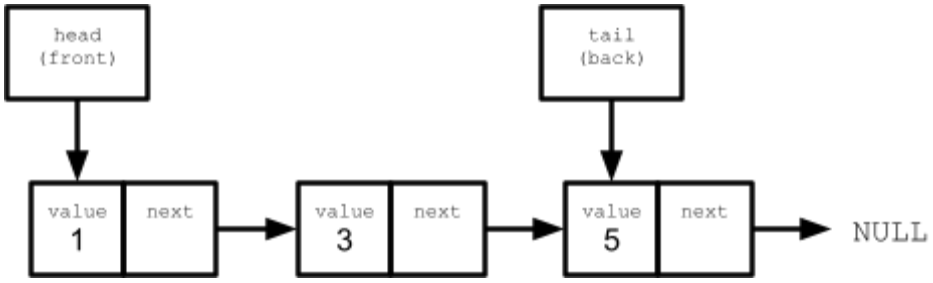
- Note also that, again unlike with a stack, where we work only with the top, we must be able to work with both ends of the queue. This will require us to implement a queue differently than we implement a stack. However, with some modifications, we can still use either a linked list or a dynamic array as the underlying storage mechanism for a queue.

Implementing a queue using a linked list

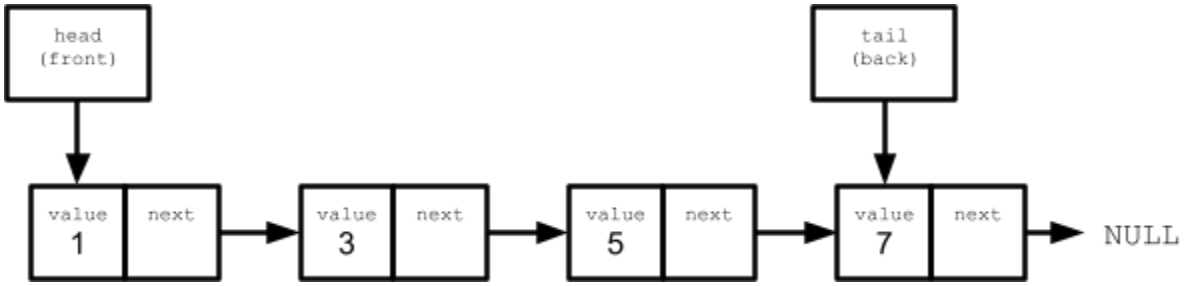
- We can still implement a queue using a singly-linked list. However, because we need to be able to work with both ends of a queue, enqueuing onto the back of the queue and dequeuing from the front, we must keep track of both the head of the list and the tail of the list.

- Specifically, in a linked list-based implementation of a queue, enqueued values are inserted at the tail of the list, while values are dequeued from the head.

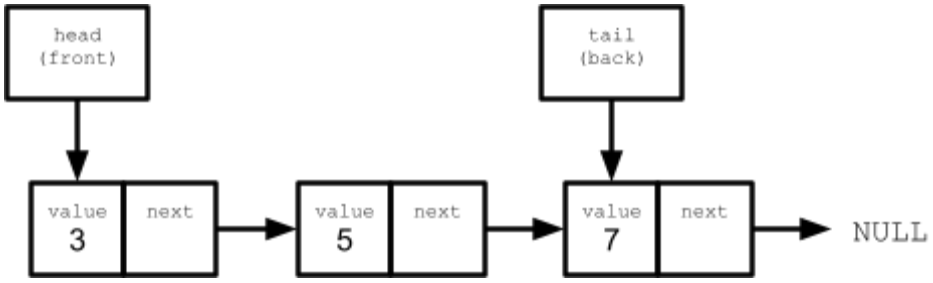
- For example, the linked list below represents a queue containing three values: 1 at the front of the queue, followed by 3, with 5 at the back of the queue:



- If the user called `enqueue(7)` on this queue, the value 7 would be inserted at the tail of the underlying list:



- If the user then called `dequeue()` on this queue, the value 1 would be removed from the head of the underlying list:



- Additional calls to `dequeue()` would first result in the value 3, then 5, and finally 7 being removed from the list.

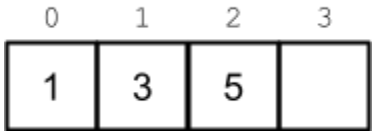
- As with a linked list-based stack, operations on a linked list-based queue (i.e. `enqueue()` and `dequeue()`) have *O(1)* runtime complexity (best-case, worst-case, and average), since they involve only the constant-time operations of allocating a single link and updating pointers.

Implementing a queue using a dynamic array

- As when using a linked list, implementing a queue using a dynamic array as the underlying data storage mechanism requires that we make slight modifications that allow us to easily work with both the front and back of the queue.

- When implementing a queue using a dynamic array, the front of the queue will correspond to the front of the array, and the end of the queue will correspond to the end of the array.

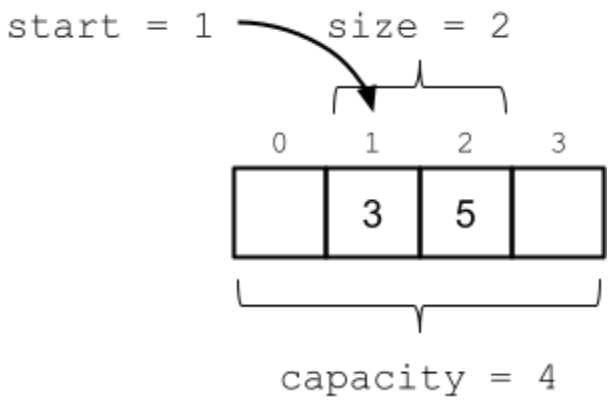
- For example, the following array represents a queue containing three values: 1 at the front of the queue, followed by 3, with 5 at the back of the queue:



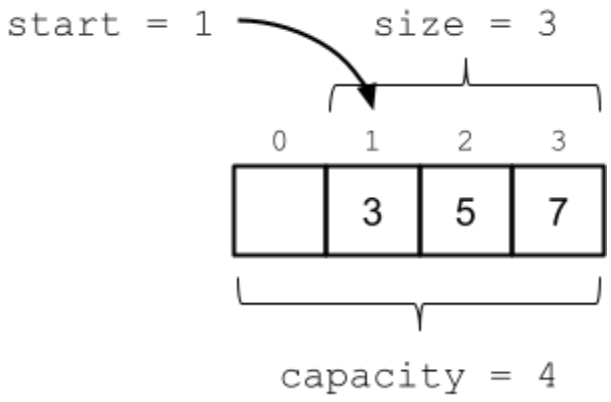
- This is straightforward so far. If we want to enqueue a new value here, we can simply insert it at the end of the array. However, what will happen if we want to *dequeue* a value?

- If we remove the first value from the underlying array when we dequeue, we will presumably have to move each remaining value forward one index in the array, so the front element of the queue again lives at the front of the array. However, this would mean that each dequeue operation will have *O(n)* runtime complexity, which is undesirable.

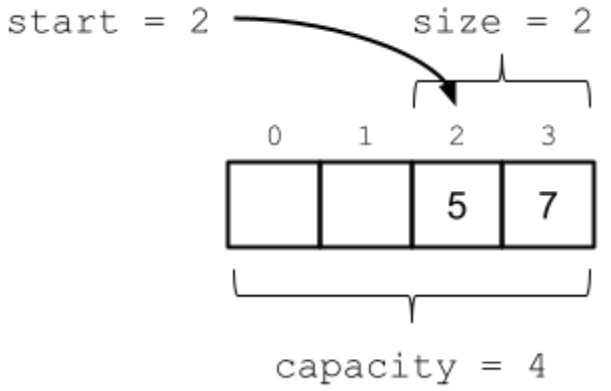
- Instead, what we will do is allow the index corresponding to the front of the queue to “float” back into the middle of the array. To make this work, we’ll add an additional field to the dynamic array to keep track of the start of the data, e.g.:



- Now, if the user runs the operation `enqueue(7)`, we will still place 7 at the end of the array:

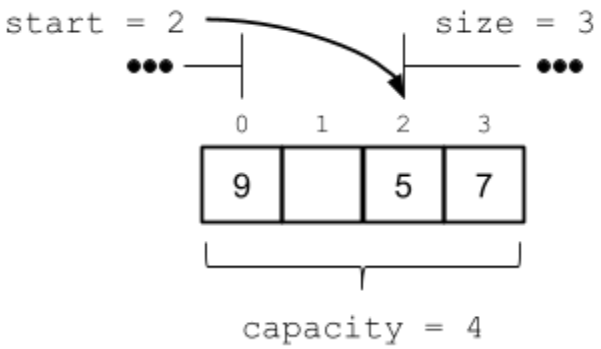


- And, if the user calls `dequeue()` again, we would again simply update the index of the start of the array (and decrement the size):



- What do we do now, though, if the user calls `enqueue(9)`? Do we resize the array, since we've written up to its end? Doing this could result in a fairly inefficient use of memory.

- Instead, we'll allow the values to *wrap around* back to the beginning of the array. In particular, a call to `enqueue(9)` would result in an underlying array that looked like this:



- An array like this, in which we allow data to wrap around from the back to the front, is known as a ***circular buffer***.

- A question you might have now, with these modifications, is how we will know which index in the array corresponds to the back of the queue.

- We could simply keep track of the index corresponding to the end of the array, much like we're already keeping track of the index corresponding to the start.

- However, a more elegant solution is to compute a mapping between the array's ***logical indices*** (i.e. the indices relative to the start of the data, `start`) and its ***physical indices*** (i.e. the indices relative to the start of the *physical array*).

- This mapping can be computed using a simple formula that calculates the physical index corresponding to a specific logical index by simply offsetting by the start index:

```
physical = start + logical;
```

- Because we're using a circular buffer, we need to account for logical indices wrapping around from the back of the array to the front. We can do this by adding the following check to our formula:

```
if (physical >= capacity) {
  physical -= capacity;
}
```

- This entire computation, including accounting for indices wrapping from back to front, can actually be accomplished in a single line using the modulo operator:

```
physical = (start + logical) % capacity;
```

- We can see this formula working in the example array depicted above:
    - We'd expect logical index 0 (i.e. the first element in the array) to correspond to physical index 2. Indeed, plugging the appropriate values into the computation above, we can see that the physical index corresponding to logical index 0 is (2 + 0) % 4 = 2.
    - As expected, logical index 1 is computed as (2 + 1) % 4 = 3.
    - We can see wrapping behavior working as expected as well, since the physical index corresponding to logical index 3 is correctly computed to be (2 + 2) % 4 = 4 % 4 = 0.

- Using this formula, we can always compute the index at which the next element will be inserted by computing the physical index for logical index equal to `size`.
    - This works the same way as when inserting at the end of an array where the data starts at physical index 0. In this case, the first open spot in the array corresponds to `array[size]`.

Dynamic array resizing for the queue implementation

- It's worth spending a minute to explore specifically how dynamic array resizing works in the implementation described above.

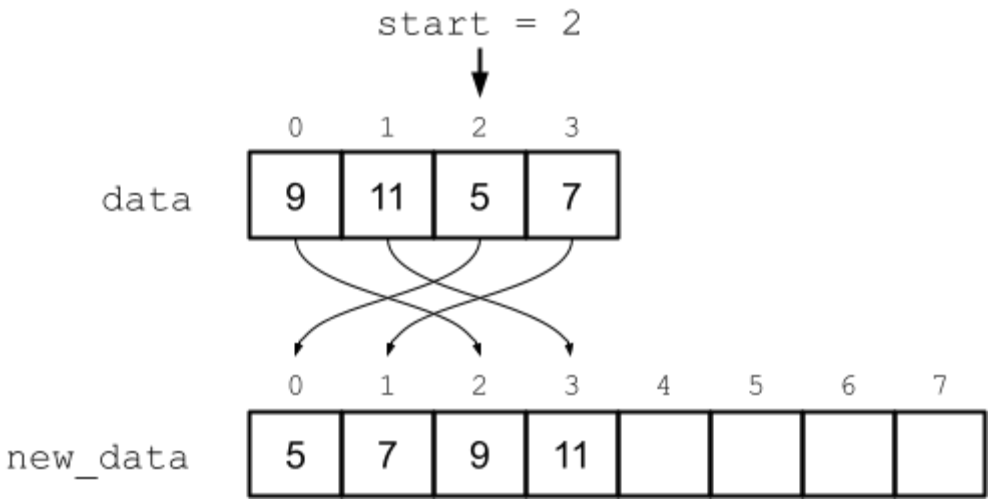
- As with the typical dynamic array implementation we'll always resize our queue's dynamic array when its `size` equals its `capacity`.
    - Note that because the *front* of our queue's underlying physical array can now contain empty spaces, a resize will not necessarily be needed when we've simply written data to the end of that physical array.

- When we resize our queue's underlying physical array, we will take the opportunity to ***reindex*** the array so that logical index 0 once again corresponds to physical index 0.

- To do this, we loop through the *logical indices* from 0 through `size - 1`, and copy the element at each *logical* index in the old array to the equivalent *physical* index in the new, larger array.

- In other words, when we resize, the value at *logical* index 0 in the old array gets copied to *physical* index 0 in the new array, the value at *logical* index 1 in the old array, gets copied to *physical* index 1 in the new array, and so forth.

- Visually, this would look something like this:



- Because, as we've seen, dynamic array insertion has *average O(1)* runtime complexity, even accounting for resizing, a queue's operations `enqueue()` and `dequeue()` will still have *O(1)* average runtime complexity when using a dynamic array as the underlying data storage.

- However, because the dynamic array may need to be resized on any given insertion, a queue's `enqueue()` operation will have *O(n)* worst-case runtime complexity when the queue is built on top of a dynamic array.

Deques

- A deque (pronounced “deck”), or double-ended queue, is a linear ADT that supports insertion and removal at both ends.

- Specifically, a deque supports these four primary operations:
    - ***Add to front***
    - ***Add to back***
    - ***Remove from front***
    - ***Remove from back***

- Given the flexibility offered by a deque, we could, if we wanted, build either a stack or a queue on top of a deque without additional modification.

- In addition, deques have real-world applications, such as in multi-processor job scheduling.

- Implementing a deque using a dynamic array for its underlying data storage can be done easily with a straightforward modification to the dynamic array-based queue implementation described above. Figuring out the details of the required modification is left as an exercise for the reader.
    - When implemented correctly, a dynamic array-based deque will support insertion at front or back with *O(1)* average runtime complexity and removal from front or back with *O(1)* worst-case runtime complexity.

- We’ll explore below how to use a linked list to implement a deque.

Implementing a deque using a linked list

- Implementing a deque using a linked list will require some modifications over and above the ones we made to implement a queue on top of a linked list.

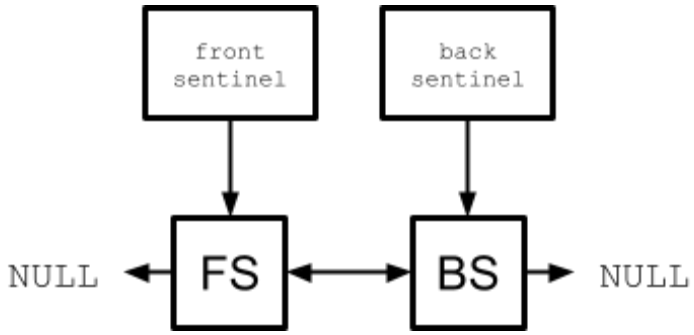
- Because a deque needs to support removal from both the front and back, we’ll need to use a doubly-linked list (i.e. one where each link points to both the next link and the previous link in the list) to implement a deque. This will, in particular, allow us to remove an element from the back of the deque and find the link that will become the *new* back.

- The next modification we'll make is not strictly required to implement a deque on top of a linked list, but, as we'll see, it will greatly simplify that implementation.

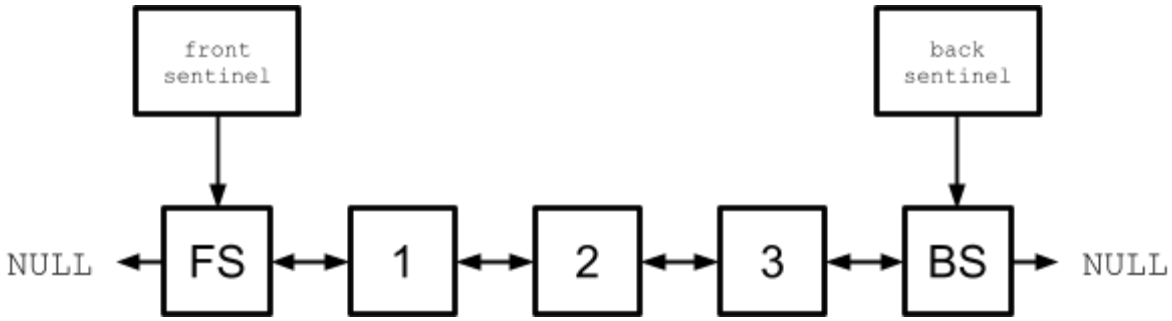
- Specifically, instead of maintaining pointers directly to the links at the head and tail of the list, we'll use a front and back sentinel in the list we use to implement our deque.

- Recall that a sentinel is a special link that is never removed from the list and is used like a "bookend" to demarcate the end of the list. Importantly, a sentinel does not actually store a value, though it is still represented using the same link structure used to represent ordinary links.

- Again, the sentinel links are never removed from the list. Thus, we know a list that uses both front and back sentinels is empty when it contains no links *other than* the sentinels (note that the `prev` and `next` pointers are not explicitly depicted here within the links but are instead indicated via bidirectional arrows):



- Real values are inserted into the list in links that live *between* the sentinels. Here's an example of a doubly-linked list with a few values between the front and back sentinel:



- In fact, this list represents a deque that contains the value 1 at the front and the value 3 at the back, with the value 2 between them.

- To add an element to the front of a deque built on top of a linked list like this one, we simply have to insert a new link *after* the front sentinel (but before the link immediately following the front sentinel). To add an element to the back of such a deque, we simply have to insert a new link *before* the back sentinel (but, again, after the link immediately prior to the back sentinel).

- Similarly removing a value from the front of a deque implemented on top of a linked list like the one above would entail removing the link immediately after the front sentinel, and removing a value from the back would entail removing the link immediately before the back sentinel.

- So why do we use sentinels like this when implementing a deque on top of a linked list?

- Remember, a deque must support *four* primary operations: adding to both the front and back and removing from both the front and back. Without using sentinels, each of these four operations would have to be implemented differently.
    - For example, when adding to the front in a list without sentinels, we would have to explicitly update the list's `head` pointer upon each insertion, while on the other hand, adding to the back would require explicitly updating the list's `tail` pointer.

- However, when we use a list with front and back sentinels, we eliminate the need to implement different functionality in the deque's insertion operations and different functionality in its removal operations. In other words, both of the insertion operations (add to front and add to back) can use the exact same mechanics when we use sentinels, as can both of the removal operations.

- Specifically, imagine that our list implementation provides a function called `add_before()` that inserts a new link with a given value *before a specified link already in the list*, e.g.:

```
void add_before(void* value, struct link* next) {
  struct link* new_link = malloc(sizeof(struct link));
  new_link->value = value;
  new_link->prev = next->prev;
  next->prev->next = new_link;
  new_link->next = next;
  next->prev = new_link;
}
```

- This function specifically does the following things:
    - Allocates a new link structure and inserts the new value into it.
    - Updates pointers to insert the new link *after* the one that previously came before `next` (i.e. `next->prev`).
    - Updates pointers to insert the new link *before* `next`.

- Because our list uses sentinels, this `add_before()` function can then be used to implement the add to front operation, which will always insert the new value *before* whatever link currently comes after the front sentinel (which, in the case of an empty list, will be the back sentinel):

```
void add_to_front(void* value) {
  add_before(value, front_sentinel->next);
}
```

- Similarly, we can use the same `add_before()` function to implement the add to back operation, which will always insert the new value before the back sentinel:

```
void add_to_back(void* value) {
  add_before(value, back_sentinel);
}
```

- In the same way, if we assume that our list provides a `remove_link()` function that removes a given link from the list, we can implement both of our removal operations using this function as follows:
    - The remove from front operation removes the link after the front sentinel: `remove_link(front_sentinel->next)`.
    - The remove from back operation removes the link before the back sentinel: `remove_link(back_sentinel->prev)`.

    - Neither operation does anything if the list is empty, e.g. if `front_sentinel->next == back_sentinel`.

- In this way, we've greatly simplified our implementation, factoring four total operations down to only two functions: `add_before()` and `remove_link()`.

- This, in turn, will make our code much easier to understand and maintain. This is a very important factor to consider when implementing data structures!

- In addition, because both of these functions do only constant-time work, all four of the primary deque operations will have *O(1)* runtime complexity (best-case, worst-case, and average) using this implementation.