Stack:

Add (insert at the head)

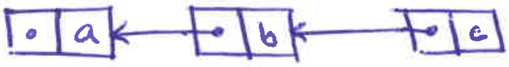
- Store the new entry and the address to current head.
- Update length by adding 1
- Update head address to new entry.

Remove (at the head)

- Update length by subtracting 1
- Update head by ~~following~~ with the address stored in the ~~first~~ head.

Both operations run in constant time.

Queue:



Add: (insert at the tail)

Same as stack, except tail is updated not head.

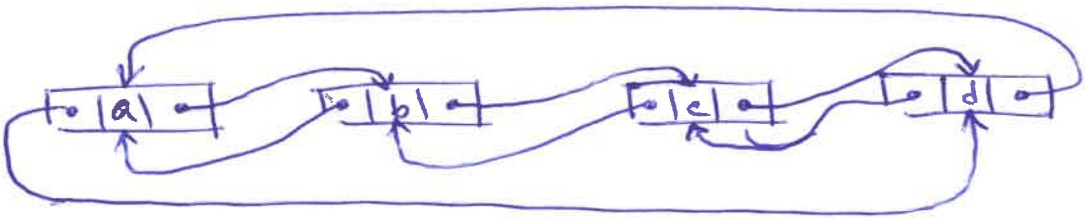
Remove:

Same as stack.

Both operations run in constant time.

What about getting the i^{th} entry? This is not constant time. The algorithm needs start at the head and traverse the whole list until the i^{th} entry. Thus, ~~getting~~ get(i) is $O(n)$.

~~There is~~ We could add more links. A doubly linked list is similar to a linked list but <u>two</u> addresses are stored (instead of just one)



Now there is no reason to specify head and tail. Just one is needed.

Does the complexity of the Queue or Stack operations change for dLLists? No!

Does the complexity for get(i) change? No!

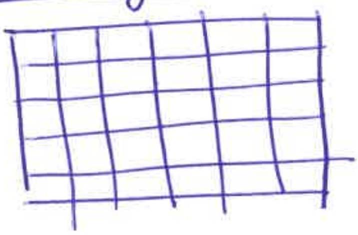
$$O\left(\frac{n}{2}\right) = O(n).$$

Arrays

Objects are stored in a prescribed location in memory often, but not necessarily, in a contiguous way.

<u>Visualization</u>

<u>Memory</u>



Arrays

An array, unlike a linked list, stores all of its contents in a contiguous block of memory. (Space requirements might force discontinguoities, but we'll ignore this.) An array would keep track of:

- head : address of first entry
- length : number of entries in the array.
- entry_size: number of bytes for each entry.

At creation of the array, the program would reserve $length \cdot entry\_size$ bytes of memory.

Linked lists allow for more dynamic use cases:

- can add to or remove from head/tail in constant time
- data need not be homogeneous.

Thus, arrays are more static.

Adding an entry at the start or end requires ~~defining~~ constructing a new array and copying all previous entries. For example: suppose our array variable is a. Then

```
Add_at_head(a, x):
    a_new = new array of length a.length + 1.
    a_new[0] = x
    for i in range(a.length):
        a_new[i+1] = a[i].
    return a_new
```

Thus, adding an element at the head requires $O(n)$ copies. (Remember n is the size of input, which is essentially the length of a.)

Adding at the end is similar. Try this yourself! Inserting <u>anywhere</u> in an array is an $O(n)$ operation. This is true of linked ~~or~~ lists as well, except at the beginning or end of the linked list.

<u>Reading</u> or getting in a linked list is also an $O(n)$ operation — again except at the head or tail. For arrays, this is an $O(1)$ operation — <u>always</u>!

~~Get(a, i):~~

Note: addresses in memory are integers but written in hexadecimal, usually, so it might not be totally obvious when looking at it.

Get(a, i):

```
    address = a.head
    return data at(address + i * a.entry_size)
```

Getting the entries of an array uses basic maths. Maybe now, we can see why it is very convenient to start indices at 0 rather than 1.

Arrays also encode both Stack and Queue data types in a probably more straight-forward way than linked-lists.

Let's compare the basic operations:

operation	linked list	array
get(i)	O(n)	O(1)
set(i, x)	O(n)	O(1)
add_head(x)	O(1)	O(n)
remove_head(x)	O(1)	O(n)
add_tail(x)	O(1)	O(n)
remove_tail(x)	O(1)	O(n)
add(i, x)	O(n)	O(n)
remove(i)	O(n)	O(n)