Lecture 8: Linked Lists

Recap

- We've seen lots of algorithms that operate on arrays
- We've seen 2 structures based on an arrays (ArrayList, BinarySearchSet)
- Arrays have some nice properties, and some frustrating ones

Nice features of arrays

- “Random Access” is constant time
- To get element i, we compute (address of the first element) + i*(size of each element)
- There is special hardware for doing this, so it's really fast, and clearly a constant time operation

Drawbacks of arrays

- All elements are stored sequentially. That means adding/removing anywhere except the end is really hard
- Arrays are fixed size, so if we need to grow a data structure, we need to allocate and copy to a new array
- To avoid allocation/copying, we usually overallocate and potentially waste space

Space

- Can we store elements of a collection nonconsequtively?
- How do we keep track of them?
- Pointers!

Linked Data Structures

- “Nodes” in our data structure store data as well as pointers to other nodes
- We'll store pointers to at least one node (possibly more) to get “into” our data structure. We'll see an example in a bit
- Typically the lifetime of nodes is longer than a single function call, so almost all data structures are on the heap
- The set of relationships we can express with linked data structures is much richer than we can describe with arrays (just prev/next)

Simplest linked data structure: singly linked list

(AKA Forward List)

- Our Nodes will store a piece of data and one link which we call the “next” pointer
- To get “into” the list, we need to store a pointer to the first element which we call the “head” pointer
- That's the whole data structure

Iteration (looping through the list):

```
for node = head; node ≠ null; node = node.next:
    use node.data somehow
```

Prepend:

```
newNode = new Node(data, head)
head = newNode
```

Splice (insert in the middle):

```
newNode = new Node(data, currentNode.next)
node.next = newNode
```

Analysis

- Iteration: O(N)
- All others are O(1) because they are “local” operations and only touch 1 or 2 nodes **if we know the node we're modifying**

Removal?

- Slightly tricky. We need to modify the PREVIOUS node! Easy once we know it
- `prevNode.next = prevNode.next.next`

List data structure

- Usually we store head and size member variables
- How would we compute size if we just stored head?
- null indicates the end of the list

Downsides

- If we need to find a node, we have an O(N) operation because we can't “jump around” the list. We must go through the list in order
- This makes deleting slighlty tricky, since we can't go backwards
- It also makes operations at the end of the list (like append) expensive

Upgrade: Doubly linked list

- Small change: add a “prev” pointer to our nodes
- Sometimes it's useful to add a “tail” pointer to the list to jump right to the end
- More pointers need to be updated when add/remove elements
- Extra memory cost of another pointer is usually worth it unless you're SURE you never need to go backward

The real downsides

- Despite having O(1) complexity for some operations we might need (add/delete from somewhere besides the end), it is often faster to use array based data structures
- CPU caches are designed to read data from memory in order
- Linked lists do not do this!
- Each time we follow a pointer can take 100s of CPU cycles, and we have to just wait!

The real upsides

- We don't have to do “amortized analysis”. Every O(1) operation is really O(1)
- Our allocations don't have to be big. Allocating a big contiguous array can be difficult/impossible on a system with limited memory. LL's just require allocating small nodes
- Pointers to node data don't change as the list changes. Sometimes an important property! If we use a resizeable array, elements move around in memory when the array grows
- There's a good chance you'll get a question about one in an interview, so understanding them will help you get a job

Implementation trick: header

- Consider implementing `add(index, value)` for a LL class
- If the index is 0, we need to modify the head member, otherwise we need to iterate and modify one of the nodes
- For remove, if we remove the first element we need to update the head, otherwise we modify one of the nodes
- There's a few places where we'd need to special case modifying “head” which is annoying and possibly error prone (we need to remember to test those cases)
- What's special about head? It's not in a node!

Header node

- We can get rid of special cases by making a “header” node which is an actual node, but which doesn't store data
- To access the first bit of data in our list, we access `header.next.data`
- Now even if we modify the front of our list, we don't need any special case logic!
- Similarly, we can add a “trailer” node to the end to avoid special cases when modifying our “tail”
- Basically, if we get any “normal tests” to pass, that probably means we got the whole implementation correct!