CS234
MODULE 6 – LINKED LISTS

• What’s a linked list?
• What’s the point?
• A third thing probably

Updated: 2018-05-16
Linked Lists: What are they?

A linked list is a data structure.
Linked Lists: More Detail

Oh, you want more?

Let’s break the name down:

List: A collection of values stored in order (aka a sequence)

Linked: Connected together
Linked Lists: Picture Form

Each value goes in a node. We establish the order by linking the nodes together.

(In Python this will be a reference to the next node in the list, or None)

The list needs a link to the first node.
A Better Picture

That’s how I’ve been drawing memory so far, but it’s a mess with all the boxes and names and such.
Best Picture

To save myself even more on boxes, I’ll just draw the “head” field and not worry about the Object itself.
Racket’s List

A Racket List is a Linked List

You make a node with cons, you select its fields with first and rest.

Languages don’t always give you a linked list primitive.
Python’s List

No, it’s a **dynamic array**! See last week’s slides!
Pros and Cons?

CONS, get it???

Advantages:
• Append is easy, and no amortized analyses needed!
• Inserting after a given node is easy (if you already have a reference to that node)

Disadvantages:
• my_ll[idx] is $O(idx)$, not $O(1)$
• Memory usage
Why Use Lists?

Linked Lists are easy to make. If a language gives you
• Dynamically allocated Objects or Structs
• References

That’s all you need to make a list!
• or a tree
• or a graph
Why Else?

Linked Lists are easy to update.

Unlike dynamic arrays which give us $O(1)$ append *amortized*, we have achieved additions with a firm $O(1)$ complexity.

It’s also easy to add and remove nodes in the middle (if you have a reference)

With an array you need to move values around.
Traversing a List

If you have a reference called “node” you can get to the next node in the linked list using the node.next field

E.g. node = node.next
Traversing a List

If you have a reference called “node” you can get to the next node in the linked list using the node.next field

E.g. node = node.next
Traversing a List

If you have a reference called “node” you can get to the next node in the linked list using the node.next field

E.g. node = node.next
Traversing a List

If you have a reference called “node” you can get to the next node in the linked list using the node.next field

E.g. node = node.next
Traversing a List

If you have a reference called “node” you can get to the next node in the linked list using the node.next field

E.g. node = node.next
Traversing a List

If you have a reference called “node” you can get to the next node in the linked list using the $\text{node.next}$ field

E.g. $\text{node} = \text{node.next}$
Traversing a List

If you have a reference called “node” you can get to the next node in the linked list using the `node.next` field.

E.g. `node = node.next`

![Diagram showing traversal of a linked list](image)
Finding my_list[i]

```python
node ← head
while i ≠ 0:
    i ← i – 1
    node ← node.next
return node.item
```

Analysis: i iterations, each iteration is O(1). Total complexity: O(i) * O(1) = O(i)

Worst case input: i ∈ O(n)
Adding to the Front

my_list.add_front(2)

head ← Node(2, head)

O(1) time complexity. We did it!
Inserting a Node

It’s pretty simple to splice a node in after a node that you have a reference to. Let’s say you want to add “13” after node
Inserting a Node

It’s pretty simple to splice a node in after a node that you have a reference to. Let’s say you want to add “13” after node

![Diagram of a linked list with a node inserted after 15]
Deleting a Node

It’s also pretty easy to splice a node out again to delete it.

\[\text{node.next} \leftarrow \text{node.next.next}\]
Deleting a Node

It’s also pretty easy to splice a node out again to delete it.

```java
node.next ← node.next.next
```
Deleting from the Front (a Special Case)

head ← head.next
Deleting from the Front (a Special Case)

head ← head.next
Be Careful With References

If you aren’t careful with references, you can have awkward situations! You can easily have multiple references to the same object.

(In fact our algorithms wouldn’t really work if we weren’t allowed!)
Reference Issue #1: Shared Nodes
Shared Nodes

Can happen if you’re creating operations that work with several lists.

Can happen if you’re thinking Function but writing Imperative

Problem: Mutating a node might change both lists or only one, but it’s not consistent or predictable!
Reference Issue #2: Cycles
Cycles

Can happen if you make a mistake manipulating links in a list.

Can also be deliberate (there are reasons to do this).

If you’re not aware cycles could be present, traversals turn into infinite loops!
Lets Reimplement the Bag!

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>Array (sorted)</th>
<th>Linked List</th>
<th>Linked List (sorted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>O(1)*</td>
<td>O(n)</td>
<td>O(1)</td>
<td>???</td>
</tr>
<tr>
<td>Remove</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>???</td>
</tr>
<tr>
<td>Contains</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>???</td>
</tr>
</tbody>
</table>

* amortized

Conclusions?

Linked List seems slightly better than the array.
Finding a value in a sorted Linked List

Binary Search won’t work.

It relies on item_at being $O(1)$. Accessing the middle element of a linked list is not $O(1)$, it is $O(n)$.

Binary search is $O(n \log n)$. That’s worse than just checking every node in order
Filling in the Blanks

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>Array (sorted)</th>
<th>Linked List</th>
<th>Linked List (sorted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>O(1)*</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Remove</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Contains</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

Oh. So Sorted Linked Lists are bad..?
• As far as Big-O goes, yeah.
• Not if you want your values in order!