CS234

MODULE 6 – LINKED LISTS

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

Updated: 2018-05-24
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

![Diagram of a linked list with nodes 4, 8, 15, 16, 23, 42, and a reference to the first node labeled 'head' and another node labeled 'node' connected to a node labeled 4.](image-url)
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`
Finding my_list[i]

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

![Diagram showing linked list with node insertion](image-url)
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
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.

\[ \text{node.next} \leftarrow \text{node.next.next} \]
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!
### Let's 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!
More Update Operations

Adding to the front of a linked list is $O(1)$ (see slide 20)

What about adding to the back? Also $O(1)$ if we have a tail reference
Adding to the Back

It’s easy to insert into a list if you have a reference.

To add to the tail of the list, we need to have a reference to the tail. Not a problem if we’re using a wrapper Class.
Adding to the Back

Function add_back(v):
    Tail.next ← Node(v,None)
    Tail ← Tail.next
Removing from the Back

Removing from the Back is $O(n)$, if you have a singly linked list. So, let's not! Presenting, the doubly linked list:

Class DNode:

"""Fields: item, next, prev"

def __init__(self, i, n, p):
    self.item = i
    self.next = n
    self.prev = p
Doubly Linked Lists

None will represent “no next” as well as “no previous”
Removing from the Back

We need to:

tail ← tail.prev
tail.next ← None
Removing from the Back

We need to:

tail ← tail.prev
tail.next ← None
Temporaries and You

If you want to assign two expressions to two variables, but both expressions depend on the original values:
Temporary

To swap x and y

```
    temp = x
    x = y
    y = temp
```
Python’s Multiple Assignment

In Python, temporaries can be **automatic**

A one line swap of $x$ and $y$ in Python:

$$x, y = y, x$$

Neat!
Doubly Linked Lists and Waste

We’re now wasting twice as much memory, since each node has two references instead of 1.

Therefore: Use a doubly linked list only when necessary.

2 is a constant, so the waste is still $O(n)$. 
Sorting a Linked List

All of your favorite sorting algorithms work on linked lists.

Merge sort is great, actually!

On an Array (like Python’s list) it requires* O(n) memory

With a linked list you can do it in O(1) memory.

*there’s a way to do it in O(1) memory but it’s complicated
Merge Sorting a Linked List

The reason we don’t need additional storage is that it’s easy to merge without copying.

function merge(node1, node2):
    head = anode = smaller of node1, node2
    advance smaller of node1, node2
    while node1, node2 not empty:
        anode.next = smaller of node1, node2
        anode = anode.next
        advance smaller of node1, node2
Merging Two Linked Lists

7 < 8, so that’s what will be selected first
Merging Two Linked Lists

8 < 18, so node1 will be appended next, after p.
Merging Two Linked Lists

15 < 18 so it will be node1 again (p.next is already node1 so no changes are made)
Merging Two Linked Lists

18 < 77, node2 is appended next
Merging Two Linked Lists

42 < 77, node2 is appended next (p.next already equals node2)
Merging Two Linked Lists

node2 is done, so node1 goes next
node1 and node2 are both None, so the merge is completed
Why Linked Lists, Again?

An instructor who will remain nameless asked me “Why are you showing them linked lists in Python???”

A: Adding / Removing from the middle is easy (if you have a reference).

Linked lists are super cool when combined with another data structure.

Python’s dict is a Hash Table (q.v.)
In Python 3.6 this was augmented with a linked list to allow the dictionary to store the keys in order. Still O(1)!
Put a Pin in That

Obviously to combine a linked list with another data structure, we need to talk about those other data structures! So…until then, we’ll leave it at that.
Searching (again)

We couldn’t do binary search because we didn’t have a link to the middle node. What if we did?

Well, we still couldn’t, since we don’t have links to the $\frac{1}{4}$ and $\frac{3}{4}$ points! What if we did?

Well, we also need references to the $\frac{1}{8}$, $\frac{3}{8}$, $\frac{5}{8}$, and $\frac{7}{8}$ points? What if we did?

If we did all the way to fractions over $n$, we’d have a binary search tree!
Searching (randomly)

Besides a BST, there’s also something called a “skip list”.

The idea is this: Make multiple linked lists that share some nodes.

The top list contains the elements at index $n/2$.
The next list contains indices $n/4$, $n/2$, $3n/4$
...
The bottom list contains all $n$ elements
Skip List: A Picture

- Part of a length 16 skip list
Skip Lists

A Skip List lets us perform a binary search: Why?

We start in the top list. If we reach a point where node.value < X, but node.next == None or node.next.value > X, we stay at this node but descend a level.

Number of nodes we check per layer: 2
Number of layers: \( \log n \)

Search Cost: \( O(\log n) \)
Skip Lists: Problem

Updating the layers when you insert or remove is $O(n)$

Solution: Coin Toss

When adding a node to the skip list, toss a coin. Each time it comes up “heads” add it to another layer.

All nodes are in the bottom layer. $\sim \frac{1}{2}$ are in the next layer, $\sim \frac{1}{4}$ in the next. $\sim 1$ are in the $(\log n)^{th}$ layer.
Skip Lists, Expected Case

This is a sort of “average case” analysis. We’re assuming worst case input for the values, but expected (average) case for the coin tosses. Worst case coin tosses will make a regular linked list.

(Actually worst case coin tosses will make infinite loops, so let’s assume we stop tossing at log n tosses)

The odds of worst case tosses are $O\left(1 / 2^n\right)$ or “basically 0%”
Questions?

```c
prev->next = toDelete->next;
delete toDelete;

// if only forgetting were
// this easy for me.
```

assert "It's going to be okay.";