5.2

Linked Lists - Advanced Techniques

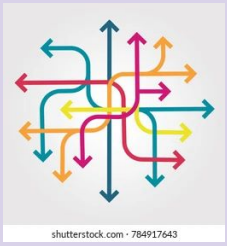
Lesson Plan

- [10] Icebreaker
- [25] Advanced Techniques

- [60] Practice

Icebreaker

Technique - Pointer Spaghetti*



*not an actual thing

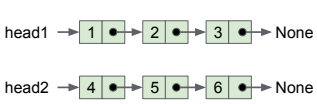
Pointer Spaghetti

So far we've mainly just traversed through a list with one, maybe two, pointers.

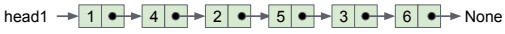
Sometimes we need more than that!

Problem - Zipper

Inputs:

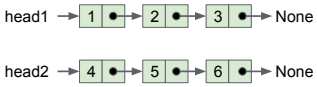


Output:

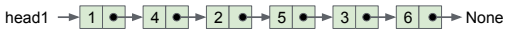


Problem - Zipper

Inputs:

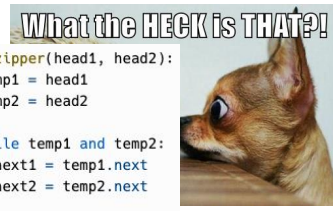


Output:

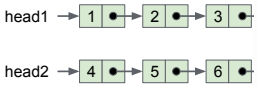



```
def zipper(head1, head2):
  temp1 = head1
  temp2 = head2

  while temp1 and temp2:
    next1 = temp1.next
    next2 = temp2.next

    temp1.next = temp2
    temp2.next = next1

    temp1 = next1
    temp2 = next2

  return head1
```

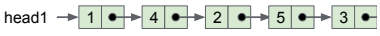
Problem - Zipper



Inputs:

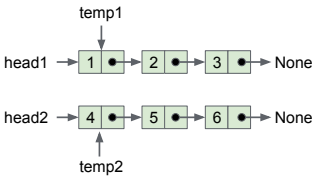


Output:



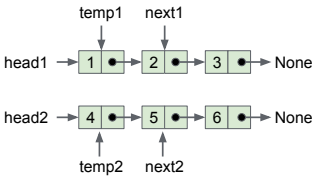

```
def zipper(head1, head2):
  temp1 = head1
  temp2 = head2

  while temp1 and temp2:
    next1 = temp1.next
    next2 = temp2.next

    temp1.next = temp2
    temp2.next = next1

    temp1 = next1
    temp2 = next2

  return head1
```

Problem - Zipper



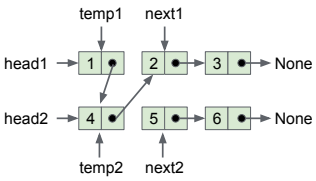

```
def zipper(head1, head2):
  temp1 = head1
  temp2 = head2

  while temp1 and temp2:
    next1 = temp1.next
    next2 = temp2.next

    temp1.next = temp2
    temp2.next = next1

    temp1 = next1
    temp2 = next2

  return head1
```

Problem - Zipper



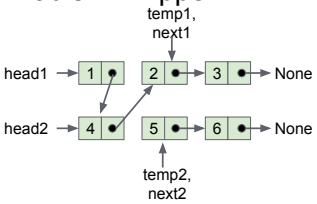

```
def zipper(head1, head2):
  temp1 = head1
  temp2 = head2

  while temp1 and temp2:
    next1 = temp1.next
    next2 = temp2.next

    temp1.next = temp2
    temp2.next = next1

    temp1 = next1
    temp2 = next2

  return head1
```

Problem - Zipper



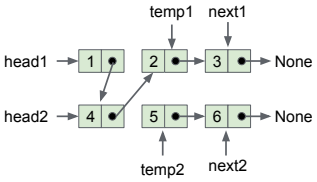

```
def zipper(head1, head2):
  temp1 = head1
  temp2 = head2

  while temp1 and temp2:
    next1 = temp1.next
    next2 = temp2.next

    temp1.next = temp2
    temp2.next = next1

    temp1 = next1
    temp2 = next2

  return head1
```

Problem - Zipper



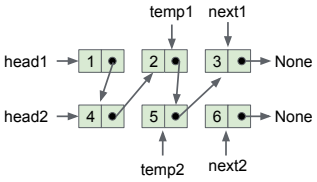

```
def zipper(head1, head2):
  temp1 = head1
  temp2 = head2

  while temp1 and temp2:
    next1 = temp1.next
    next2 = temp2.next

    temp1.next = temp2
    temp2.next = next1

    temp1 = next1
    temp2 = next2

  return head1
```

Problem - Zipper



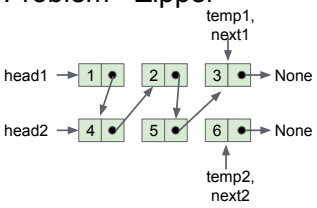

```
def zipper(head1, head2):
  temp1 = head1
  temp2 = head2

  while temp1 and temp2:
    next1 = temp1.next
    next2 = temp2.next

    temp1.next = temp2
    temp2.next = next1

    temp1 = next1
    temp2 = next2

  return head1
```

Problem - Zipper



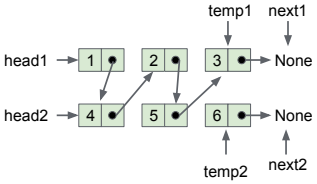

```
def zipper(head1, head2):
  temp1 = head1
  temp2 = head2

  while temp1 and temp2:
    next1 = temp1.next
    next2 = temp2.next

    temp1.next = temp2
    temp2.next = next1

    temp1 = next1
    temp2 = next2

  return head1
```

Problem - Zipper



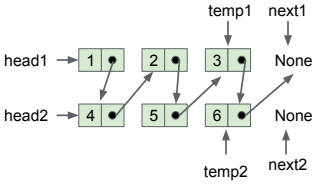

```
def zipper(head1, head2):
  temp1 = head1
  temp2 = head2

  while temp1 and temp2:
    next1 = temp1.next
    next2 = temp2.next

    temp1.next = temp2
    temp2.next = next1

    temp1 = next1
    temp2 = next2

  return head1
```

Problem - Zipper



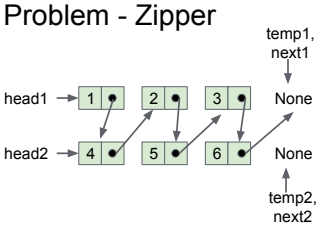

```
def zipper(head1, head2):
  temp1 = head1
  temp2 = head2

  while temp1 and temp2:
    next1 = temp1.next
    next2 = temp2.next

    temp1.next = temp2
    temp2.next = next1

    temp1 = next1
    temp2 = next2

  return head1
```

Problem - Zipper



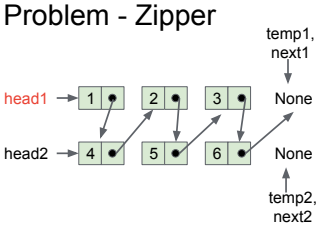

```
def zipper(head1, head2):
  temp1 = head1
  temp2 = head2

  while temp1 and temp2:
    next1 = temp1.next
    next2 = temp2.next

    temp1.next = temp2
    temp2.next = next1

    temp1 = next1
    temp2 = next2

  return head1
```

Problem - Zipper



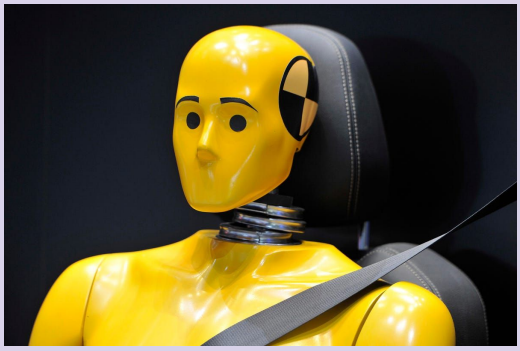

```
def zipper(head1, head2):
  temp1 = head1
  temp2 = head2

  while temp1 and temp2:
    next1 = temp1.next
    next2 = temp2.next

    temp1.next = temp2
    temp2.next = next1

    temp1 = next1
    temp2 = next2

  return head1
```

Problem - Zipper




```
def zipper(head1, head2):
  temp1 = head1
  temp2 = head2

  while temp1 and temp2:
    next1 = temp1.next
    next2 = temp2.next

    temp1.next = temp2
    temp2.next = next1

    temp1 = next1
    temp2 = next2

  return head1
```

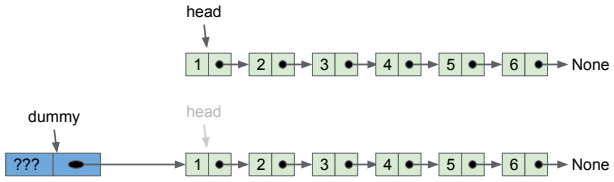
Technique - Dummy Node



Dummy Node

A node that isn't actually part of the list. We don't care about its value and should never return it.

It gives us a useful reference, and sometimes eliminates edge cases because the list is never truly "empty".



Append

```
4  def append(head, x):
5    new_node = ListNode(x)
6
7    if not head:
8      return new_node
9
10   temp = head
11   while temp.next:
12     temp = temp.next
13
14   temp.next = new_node
15
16   return head
```

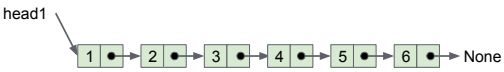
Append


```
4  def append(head, x):
5    new_node = ListNode(x)
6
7    if not head:
8      return new_node
9
10   temp = head
11   while temp.next:
12     temp = temp.next
13
14   temp.next = new_node
15
16   return head
```

```
def append(head):
  new_node = ListNode(x)

  dummy = ListNode("dummy", head)

  temp = dummy
  while(temp.next):
    temp = temp.next

  temp.next = new_node

  return
```

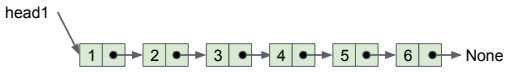
Even / Odd

Split this list into 2 - one with even numbers, one with odd numbers



Even / Odd

Split this list into 2 - one with even numbers, one with odd numbers



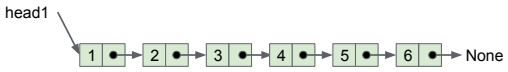

```
evens_head = None
odds_head = None
```

evens_head →

odds_head →

Even / Odd

Split this list into 2 - one with even numbers, one with odd numbers




```
evens_head = None
odds_head = None
```

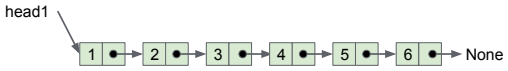
evens_head →

odds_head →

An edge case occurs for the first element we add to each even/odd head!

Even / Odd

Split this list into 2 - one with even numbers, one with odd numbers




```
evens_head = None
odds_head = None
```

evens_head →

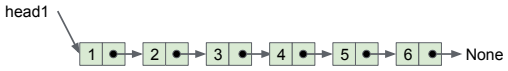
odds_head →

An edge case occurs for the first element we add to each even/odd head!

```
if temp.val % 2 == 0:
  if not evens_head:
    evens_head = temp
  else:
    # append it
```

Even / Odd

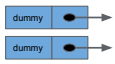
Split this list into 2 - one with even numbers, one with odd numbers




```
evens_dummy = ListNode("dummy")
odds_dummy = ListNode("dummy")
```

evens_dummy →

odds_dummy →




```
if temp.val % 2 == 0:
  # append it!
```

You'll get to practice a problem like this with Partition today!

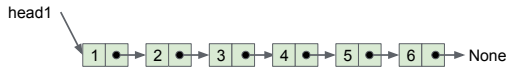
Maintaining a head and a tail

Head + Tail

If we have access to the last element in the list, appending is O(1)

If we don't, appending is O(n) - we have to iterate to the end to find it

Head + Tail - Even / Odd



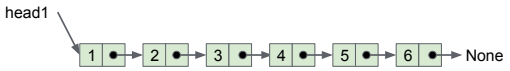

```
evens_dummy = ListNode("dummy")
odds_head = ListNode("dummy")
```


```
if temp.val % 2 == 0:
  # append it!
```

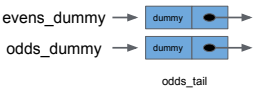
This will be O(n) if we don't keep track of the last element!

If we keep track of the last element we added, it's just O(1)!

Head + Tail - Even / Odd




```
evens_dummy = ListNode("dummy")
odds_dummy = ListNode("dummy")
```



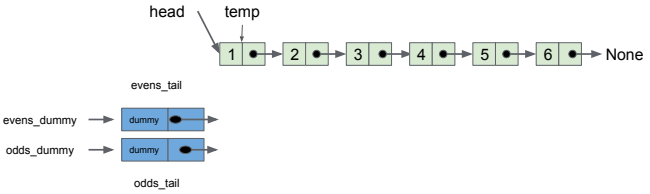

```
evens_tail = evens_dummy
odds_tail = odds_dummy
```

```
if temp.val % 2 == 0:
  evens_tail.next = temp
  evens_tail = temp
```

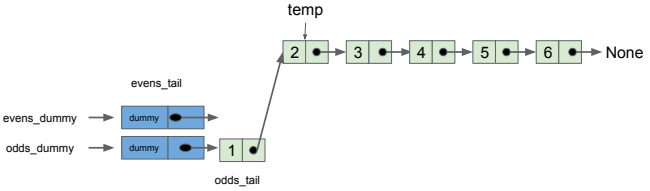
A Warning!

When we move nodes around like this, we aren't necessarily fixing the 'next' pointer at each step.

A Warning!

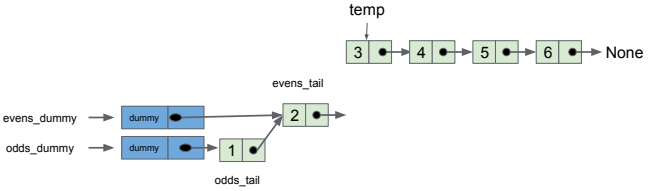


A Warning!



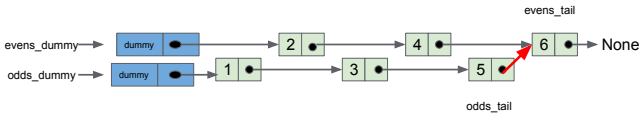
This is okay because we'll overwrite it when we add the 3

A Warning!



This is okay because we'll overwrite it when we add the 3

A Warning!



evens_tail.next = None
odds_tail.next = None

Using Auxiliary Data Structures

The data structure that would be optimal here.

The data structure I thought of first.

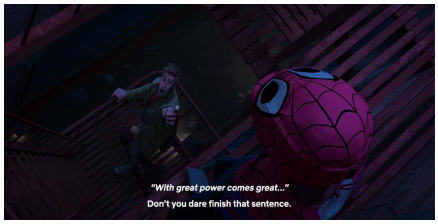
Using Auxiliary Data Structures

SOMETIMES it is easiest to use an auxiliary data structure.

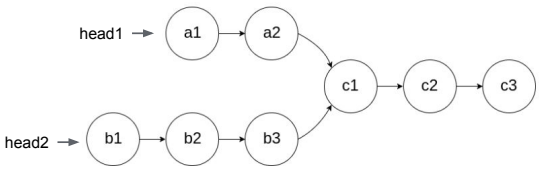
Using Auxiliary Data Structures

These problems can *almost always* be solved without auxiliary data structures, be very mindful of your extra space complexity.

But just in case this saves you in an interview some day…

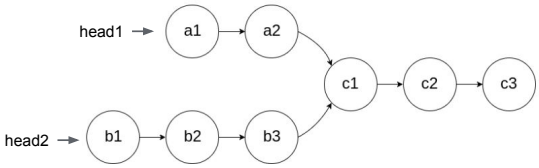


Using Auxiliary Data Structures - Intersection



Return the node where these two lists intersect

Using Auxiliary Data Structures - Intersection

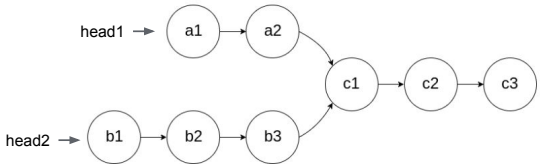


Return the node where these two lists intersect

seen_nodes:

1. Create an empty set seen_nodes

Using Auxiliary Data Structures - Intersection



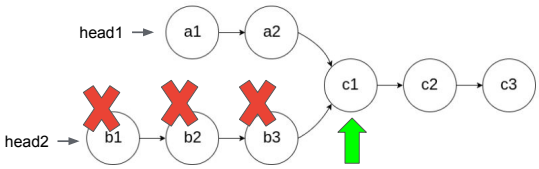
Return the node where these two lists intersect

seen_nodes: a1, a2, c1, c2, c3

1. Create an empty set seen_nodes
2. Iterate through head1, adding each node to seen_nodes

User-defined types (like ListNode) are hashable!

Using Auxiliary Data Structures - Intersection

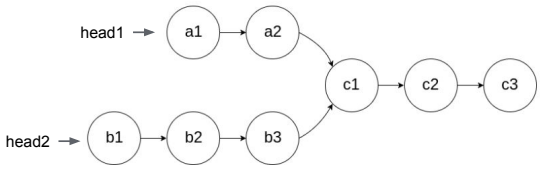


Return the node where these two lists intersect

seen_nodes: a1, a2, c1, c2, c3

1. Create an empty set seen_nodes
2. Iterate through head1, adding each node to seen_nodes
3. Iterate through head2, checking each node against seen_nodes. The first match is the intersection!

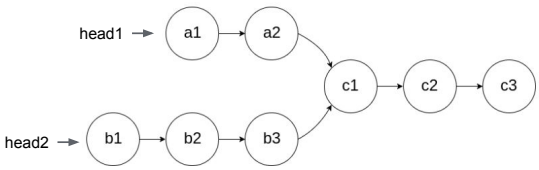
Using Auxiliary Data Structures - Intersection



How would we solve this problem *without* an auxiliary data structure?

1. Find the length of head1 (5)

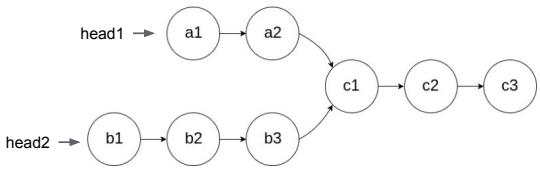
Using Auxiliary Data Structures - Intersection



How would we solve this problem *without* an auxiliary data structure?

1. Find the length of head1 (5)
2. Find the length of head2 (6)

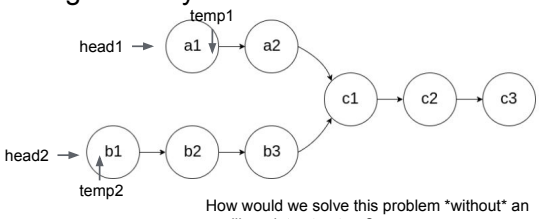
Using Auxiliary Data Structures - Intersection



How would we solve this problem *without* an auxiliary data structure?

1. Find the length of head1 (5)
2. Find the length of head2 (6)
3. Calculate the *diff* (6 - 5 = 1)

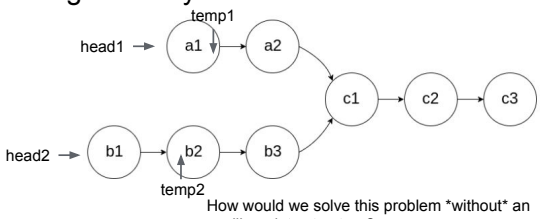
Using Auxiliary Data Structures - Intersection



How would we solve this problem *without* an auxiliary data structure?

1. Find the length of head1 (5)
2. Find the length of head2 (6)
3. Calculate the ***diff*** (6 - 5 = 1)
4. Set up temp pointers for each list

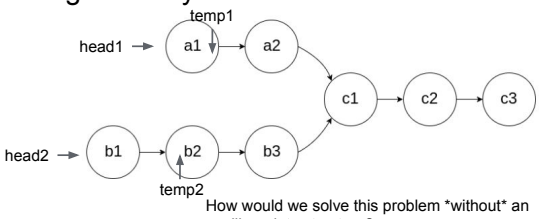
Using Auxiliary Data Structures - Intersection



How would we solve this problem *without* an auxiliary data structure?

1. Find the length of head1 (5)
2. Find the length of head2 (6)
3. Calculate the ***diff*** (6 - 5 = 1)
4. Set up temp pointers for each list
5. Advance the pointer of the longer list (2) by ***diff*** steps

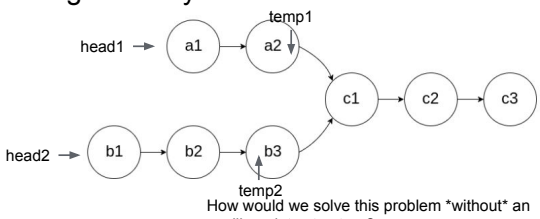
Using Auxiliary Data Structures - Intersection



How would we solve this problem *without* an auxiliary data structure?

1. Find the length of head1 (5)
2. Find the length of head2 (6)
3. Calculate the ***diff*** (6 - 5 = 1)
4. Set up temp pointers for each list
5. Advance the pointer of the longer list (2) by ***diff*** steps
6. Advance both pointers until they meet!

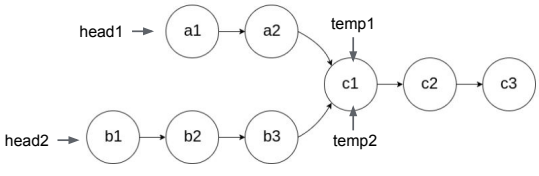
Using Auxiliary Data Structures - Intersection



How would we solve this problem *without* an auxiliary data structure?

1. Find the length of head1 (5)
2. Find the length of head2 (6)
3. Calculate the ***diff*** (6 - 5 = 1)
4. Set up temp pointers for each list
5. Advance the pointer of the longer list (2) by ***diff*** steps
6. Advance both pointers until they meet!

Using Auxiliary Data Structures - Intersection



How would we solve this problem *without* an auxiliary data structure?

1. Find the length of head1 (5)
2. Find the length of head2 (6)
3. Calculate the ***diff*** (6 - 5 = 1)
4. Set up temp pointers for each list
5. Advance the pointer of the longer list (2) by ***diff*** steps
6. Advance both pointers until they meet!

Practice Problems

You will be working in teams of 3 or 4. Use the table below to figure out what your role is. [repl.it]

Role	Responsibilities	Assignment Criteria
Captain	Share screen, write code, keep track of time, ensure all team members participate	Person who has been assigned this role the least number of times
Tester	Plays devil's advocate, design test cases, determine algorithm complexity (time and space)	Person who has been assigned this role the least number of times
Presenter	Explain solution to the class, present the team's algorithm design decisions, state solution's complexity (time and space), share one thing the team learned from the problem	Person who has been assigned this role the least number of times

If there are ties, get creative and come up with a way to break them (i.e., sort yourselves by last name, distance to Google Austin, etc.)
If there are 4 members in your team, you should have two Testers