5.1

Linked Lists - Fundamentals

Lesson Plan

- [10] Icebreaker
- [20] Quick Select
- [25] Linked Lists

- [40] Practice

Icebreaker

Quick! Back to Sorting Land

Quick Select

Can we find the median element in (expected) O(n) time?

Yes.

Partition

[12, 19, 18, 26, 25, 6, 15]

[12, 6, 15, 19, 18, 26, 25]

```
lst = [30, 12, 19, 26, 25, 6, 15, 18]
pivot = lst[-1]
small_lst = []
big_lst = []

for i in range(len(lst) - 1):
  if lst[i] <= pivot:
    small_lst.append(lst[i])
  else:
    big_lst.append(lst[i])

result = []
result.extend(small_lst)
result.append(pivot)
result.extend(big_lst)
```

Quick Select

[12, 19, 18, 26, 25, 6, 15]

[12, 6, 15, 19, 18, 26, 25]

This element is exactly in its right place! So we know which side of it the median element fell on!

Quick Select

[12, 19, 18, 26, 25, 6, 15]

[12, 6, 15, 19, 18, 26, 25]

Partition again, but only with the elements that might possibly be the median!

Quick Select

[12, 19, 18, 26, 25, 6, 15]

[12, 6, 15, 19, 18, 26, 25]

[12, 6, 15, 19, 18, 25, 26]

Quick Select

[12, 19, 18, 26, 25, 6, 15]

[12, 6, 15, 19, 18, 26, 25]

[12, 6, 15, 19, 18, 25, 26] And again!

Quick Select

[12, 19, 18, 26, 25, 6, 15]

[12, 6, 15, 19, 18, 26, 25]

[12, 6, 15, 19, 18, 25, 26]

Quick Select

[12, 19, 18, 26, 25, 6, 15]

[12, 6, 15, 19, 18, 26, 25]

[12, 6, 15, 19, 18, 25, 26]

[12, 6, 15, 18, 19, 25, 26]

Whenever we choose a pivot, it ends up in exactly its place in the ordering.
We placed this pivot right in the middle, so it's the median element!
Note that the whole list isn't necessarily sorted!

Quick Select - Runtime

In the ideal case, we partition about half the elements, then about half of the remaining, then half of the remaining… and so on.

Partition is a linear operation.

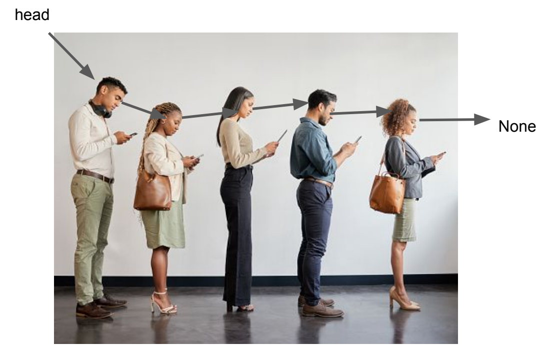
$n + n/2 + n/4 + \ldots + 1 < 2n$

So we can quick select in O(n)!

Quick Select - Implementation [repl.it]

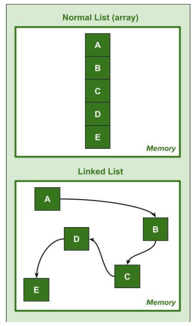
We understand this algorithm, but in vague terms. Let's try to implement it together!

Linked Lists!



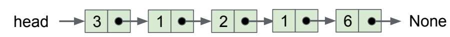
Linked Lists - Fundamentals

- A ***Linked List*** is an alternative *implementation* of the List Abstract Data Type.
- An item in a Linked List is commonly known as a ***Node***.
- Unlike array's where the items are held in a contiguous block of memory, linked lists nodes do not have to be.The nodes can be scattered! 🤯
- Each node contains information about where another node is located in memory.
    - This is called a ***Link***
- The chaining of nodes via ***Links*** is where the data structure derives its name.



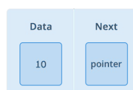
LinkedList vs. Python List

```
       0    1    2    3    4
lst = [3,   1,   2,   1,   6]
```



Linked Lists - Variations

A Node is generally composed of two sub-parts:



- Data: The value of node, this can be anything (int, string, list, custom object, etc…).
- Link: The address or addresses to another node in the chain

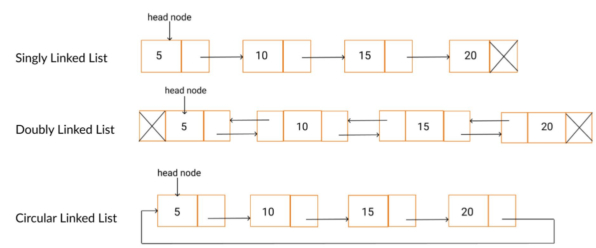
Nodes in a linked list can have various ways of linking to each other. Based on these differences, we give a naming prefix to a Linked List to distinguish what type of linking the nodes have to each other. Here are some common list types below.

1. ***Singly Linked List*** - Most common type of linked list. Each node in the list contains the address to the ***Next*** node in the list. Singly Linked List allow for only one way of traversal.
2. ***Doubly Linked List*** - Also known as a Two-way list, each node contains the addresses to the Next node and the ***Previous*** node in the list. This allows for both forward and backwards traversal
3. ***Circularly Linked List*** - The ***Tail*** (last) node in the list, points back to the ***Head*** (first) node in the list

For this course, our focus will be on Singly Linked List

Linked Lists - Variations

Here's a visual representation of each variant.



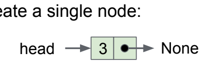
ListNode - code

For a Singly Linked List the node can be defined as follows:

```python
class ListNode:

  def __init__(self, val, next=None):
    self.val = val
    self.next = next
```

How to create nodes

Create a single node:




```python
head = ListNode(3)
# or
head = ListNode(3, None)
```

ListNode - code

Create a list like:


```python
head = ListNode(2, None)
head = ListNode(1, head)
head = ListNode(3, head)
# or
head = ListNode(3, ListNode(1, ListNode(2)))
```

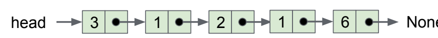
Linked List Techniques - Traversal

Traversal

How do we iterate through every element of a LinkedList if we only have access to the head?

```python
def sum(lst):
  total = 0
  for x in lst:
    total += x
  return total
```

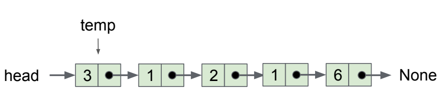
Traversal



How do we iterate through every element of a LinkedList if we only have access to the head?

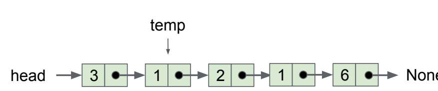
```python
def sum(head):
  total = 0





  return total
```

Traversal



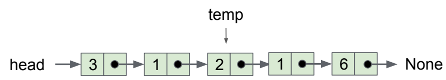
total = 3

Traversal



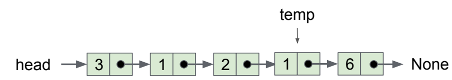
total = 4

Traversal



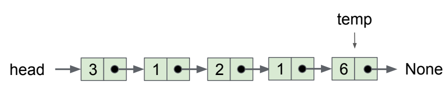
total = 6

Traversal



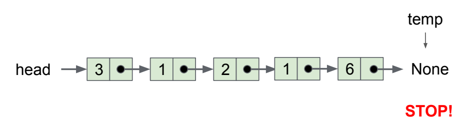
total = 7

Traversal



total = 13

Traversal

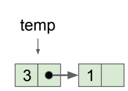


total = 13

Linked List Techniques - Adding Nodes

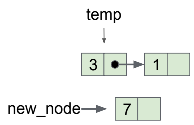
Adding

How do we add a new node after temp?



Adding

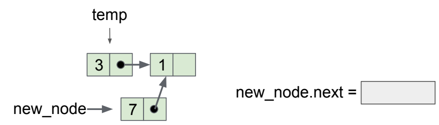
How do we add a new node after temp?



1. Create the new node

Adding

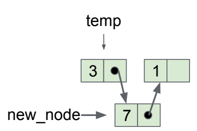
How do we add a new node after temp?



1. Create the new node
2. Make its ‘next’ point to the next node

Adding

How do we add a new node after temp?

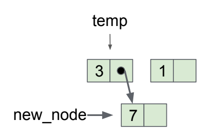


temp.next = new_node

1. Create the new node
2. Make its ‘next’ point to the next node
3. Have the original node’s ‘next’ point to the new node

Adding

How do we add a new node after temp?



temp.next = new_node

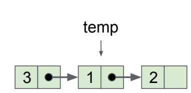
Why can’t we do step 3 before step 2?

1. Create the new node
2. Make its ‘next’ point to the next node
3. Have the original node’s ‘next’ point to the new node

Linked List Techniques - Removing Nodes

Removing

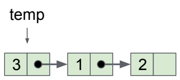
How would you remove temp from this list?



It’s impossible from here!

Removing

How would you remove temp from this list?



1. Find the node *before* the one you want to remove

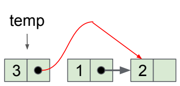
Removing

How would you remove temp from this list?

1. Find the node *before* the one you want to remove
2. Have its ‘next’ point at the one *after* the node you want to remove

Removing

How would you remove temp from this list?



temp.next = temp.next.next

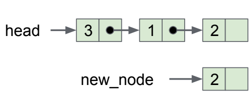
1. Find the node *before* the one you want to remove
2. Have its ‘next’ point at the one *after* the node you want to remove

The removed node will be left dangling. That’s okay!

Example Problem - Append

Append

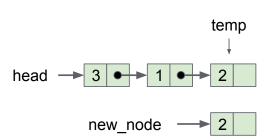
Create a new node with some value and add it to the end of the list



1. Create your new node

Append

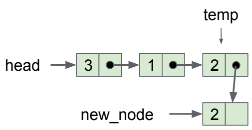
Create a new node with some value and add it to the end of the list



1. Create your new node
2. Iterate to the end of the list with a temp variable

Append

Create a new node with some value and add it to the end of the list



1. Create your new node
2. Find the end of the list with a temp variable
3. Set temp’s ‘next’ to point to new_node

Tangent Time! - Structuring a Solution

What should we be returning?

When a function modifies a list, always return the head of the list

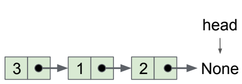
Can you find the issues?

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

  while head:
    head = head.next

  head.next = new_node

  return head
```

Can you find the issues?

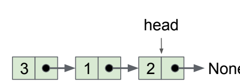
```python
def append(head, x):
  new_node = ListNode(x)

  while head:
    head = head.next

  head.next = new_node

  return head
```

We’re advancing head too far!



Can you find the issues?

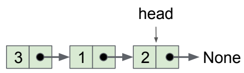
```python
def append(head, x):
  new_node = ListNode(x)

  while head.next:
    head = head.next

  head.next = new_node

  return head
```



Can you find the issues?

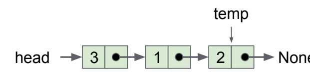
```python
def append(head, x):
  new_node = ListNode(x)

  while head.next:
    head = head.next

  head.next = new_node

  return head
```

We're not returning the actual head anymore!



Can you find the issues?

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

  temp = head
  while temp.next:
    temp = temp.next

  temp.next = new_node

  return head
```



Can you find the issues?

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

  temp = head
  while temp.next:
    temp = temp.next

  temp.next = new_node

  return head
```

For an empty list, this throws an error!

Can you find the issues?

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

  if not head:
    return new_node

  temp = head
  while temp.next:
    temp = temp.next

  temp.next = new_node

  return head
```

Structuring a Solution

- When modifying a list, return the head
- Use temp variables to iterate through a list, leaving the head where it belongs
- Never modify existing node's .val values, unless explicitly instructed to
- Watch for edge cases!
  - What if the input is empty?
  - Is your function different when the target is the first/last element, or if it's somewhere in between?
  - Does the head of the list need to change?

Example Problem - Pop

Remove the last element from a LinkedList

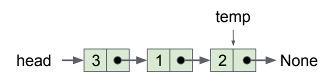
Pop - Can you find the issues?

```python
def pop(head):
  temp = head

  while temp.next:
    temp = temp.next

  temp.next = None
  return head
```

Pop - Can you find the issues?

```python
def pop(head):
  temp = head

  while temp.next:
    temp = temp.next

  temp.next = None
  return head
```

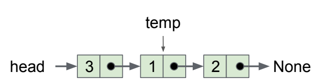
We're iterating too far!



We can't remove this node now!

Pop - Can you find the issues?

```python
def pop(head):
  temp = head

  while temp.next.next:
    temp = temp.next

  temp.next = None
  return head
```



Pop - Can you find the issues?

```python
def pop(head):
  temp = head

  while temp.next.next:
    temp = temp.next

  temp.next = None
  return head
```

This throws an error for an empty list!

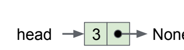
Pop - Can you find the issues?

```python
def pop(head):
  temp = head

  if not head:
    raise Exception('Cannot remove from an empty list!')

  while temp.next.next:
    temp = temp.next

  temp.next = None
  return head
```

Pop - Can you find the issues?

```python
def pop(head):
  temp = head

  if not head:
    raise Exception('Cannot remove from an empty list!')

  while temp.next.next:
    temp = temp.next

  temp.next = None
  return head
```

This *still* throws an error for lists of size 1!



Pop - Can you find the issues?

```python
def pop(head):
  temp = head

  if not head:
    raise Exception('Cannot remove from an empty list!')

  if not head.next:
    return None

  while temp.next.next:
    temp = temp.next

  temp.next = None
  return head
```

Linked List Techniques - Multiple Runners

Multiple Runners

The runner technique means that you iterate through the linked list with two pointers simultaneously, with one ahead of the other.

The ***fast node*** might be ahead by a fixed amount, or it might be hopping multiple nodes for each one node that the ***slow node*** iterates through.

Let's consider the following problem:

> Given a non-empty, singly linked list with head node `head`, return a middle node of linked list.
>
> If there are two middle nodes, return the second middle node.

Multiple Passes

We could solve this using the following logic.

1. If the list is empty do nothing!
2. Get the length of the list, use that to deduce the midpoint
3. Iterate from 0 to midpoint, and return the node

This solution works!

We can do it another way for practice!

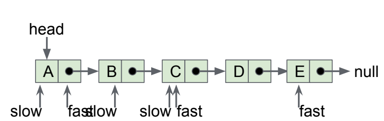
```python
def middleNode(head):
    if head == None or head.next == None:
        return head
    count = 0
    temp = head
    while temp != None:
        temp = temp.next
        count+=1
    mid = count//2
    temp = head
    while temp != None and mid > 0:
        temp = temp.next
        mid -=1
    return temp
```

One Pass w/ Multiple Runners

Let's utilize the Runner Technique and use the following logic.

1. If the list is empty, do nothing!
2. Let's declare a slow and fast pointer both starting at head
3. For every iteration, move the slow pointer by one and the fast by two
4. When the fast pointer reaches the finish line, our slow pointer should have arrived at the middle node!
5. Return the slow pointer

```python
def middleNode(head):
    if head == None or head.next == None:
        return head
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```



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