Linked Lists

Presented by Alexander Mak

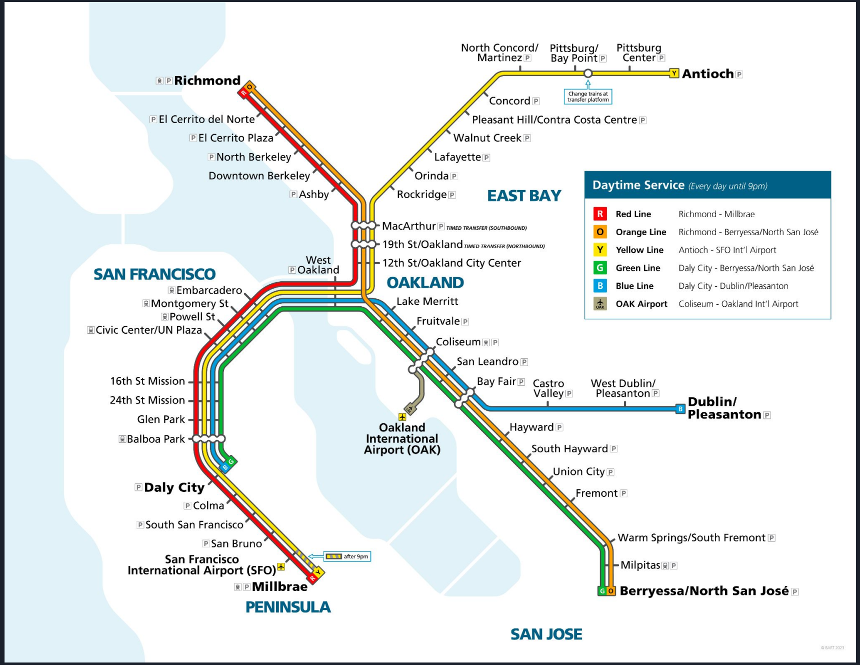
What is a linked list?

- Linear node-based data structure
- Each node contains:
  - Data
  - Reference (or pointer) to the next node
  - Optional: pointer to previous node (in case of doubly linked list)
- Forms a chain where each node points to the next one
- The beginning of a Linked List is known as the head
- The end of a Linked List is known as the tail
- Efficient for insertion and deletion operations
- Access times can be slower compared to arrays
- Unlike arrays, linked lists do not need to be contiguous in memory

Linked List Example: Train Network

Imagine a network of train lines. You can imagine that each line is its own linked list.

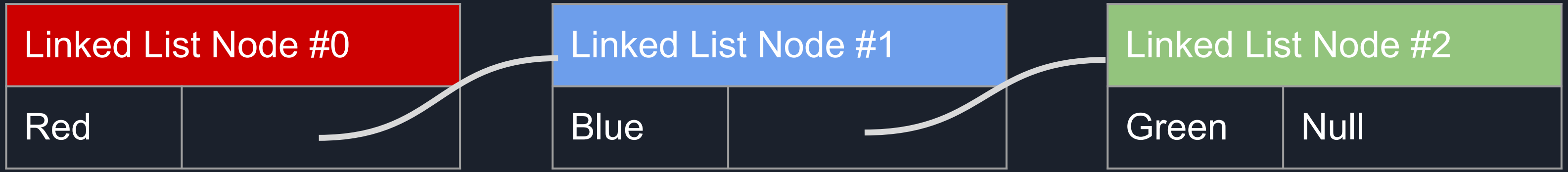
- Each stop is a node within the linked list.
- A node contains some information as well as a pointer to the next stop.
- A train only moves in one direction, similar to how a singly-linked list works
- If you are at a certain station, you can traverse in two different directions like a doubly linked list.



Presented by Alexander Mak

Singly Linked List

Linked List Node	
Value	Next Pointer



Linked List Node #0	
Red	

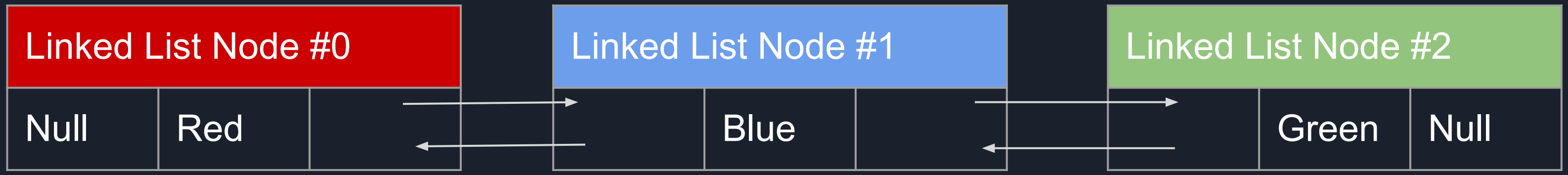
Linked List Node #1	
Blue	

Linked List Node #2	
Green	Null

- From ListNode0, we can access ListNode1 by using ListNode0.next
- We can also chain .next pointers.
  - From ListNode0, we can access ListNode2 by using ListNode0.next.next

Doubly Linked List

Linked List Node		
Prev Pointer	Value	Next Pointer

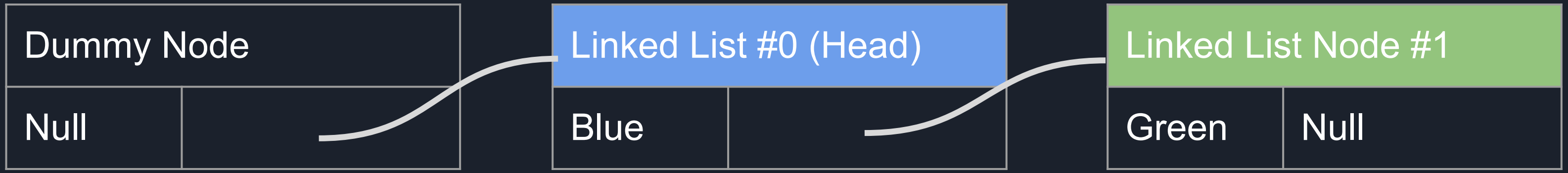


Linked List Node #0		
Null	Red	

Linked List Node #1		
	Blue	

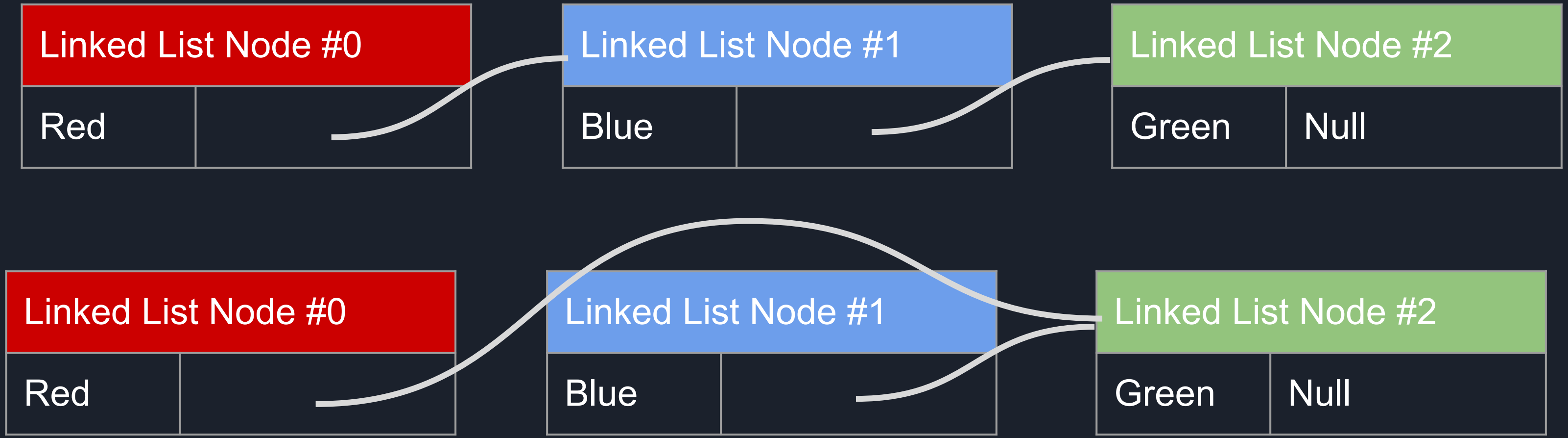
Linked List Node #2		
	Green	Null

Dummy Nodes



- When constructing linked lists for problems, it can often be helpful to create “dummy nodes”
- These dummy nodes will help in edge cases where the head node is modified
- The dummy node is not actually part of the list. It is simply a pointer to keep track of where the list starts

Reassigning Nodes



Linked List Node #0

Red

Linked List Node #1

Blue

Linked List Node #2

Green | Null

Linked List Node #0

Red

Linked List Node #1

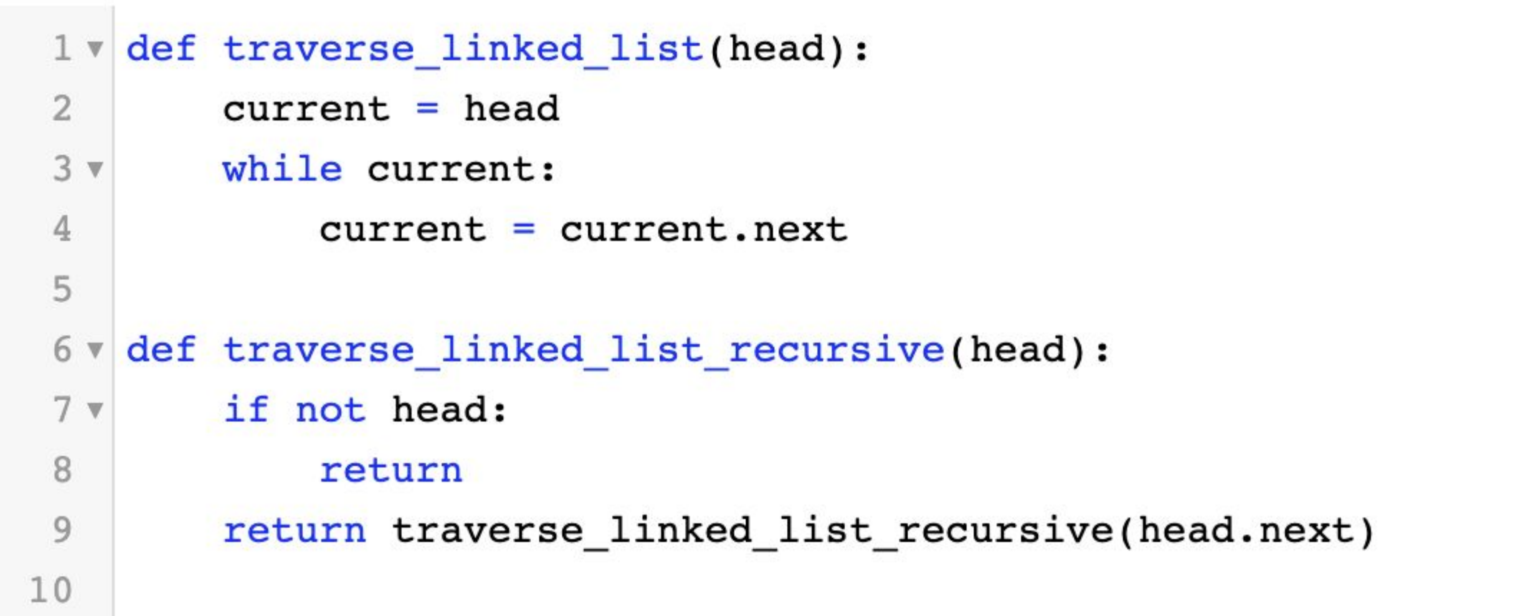
Blue

Linked List Node #2

Green | Null

- When “deleting” nodes from a list, one can simply reassign the next node
- In the above example, we can see that we removed Node #1 from the list by reassigning ListNode0.next to ListNode2

Traversing a Linked List



```python
def traverse_linked_list(head):
    current = head
    while current:
        current = current.next

def traverse_linked_list_recursive(head):
    if not head:
        return
    return traverse_linked_list_recursive(head.next)

```

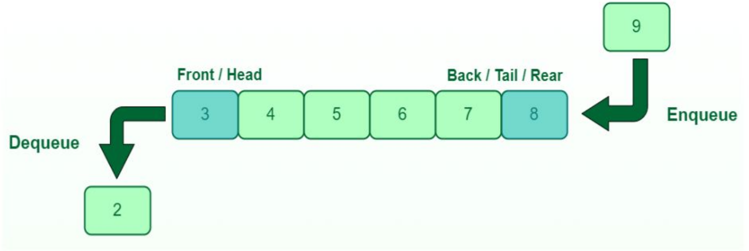
Comparing Runtimes with Arrays

Operations	Arrays	Singly Linked List	Doubly Linked List
Read/ Write ith element	O(1)	O(n) **	O(n) **
Insert / Remove End	O(1)	O(n) **	O(1)
Insert/ Remove Middle	O(n)	O(n) **	O(n) **
Insert/ Remove Beginning	O(n)	O(1)	O(1)

** These operations for a linked list are O(n) assuming you must traverse the list to access the node. If you can somehow create references to these nodes with something like a hashmap, it is possible to achieve O(1).

Queues

Operations	Time Complexity
Enqueue	O(1)
Dequeue	O(1)



Queue Data Structure

- FIFO (first in, first out)
- Because we can add/remove nodes at the beginning and end of a Doubly Linked List in O(1) time, it is actually more efficient to create a queue this way versus an array.
- In python, we can use deque from collections library

- Compared to stacks, there aren’t a lot of problems where queues shine on their own. However, they are extremely important when we need to implement breadth-first search algorithms in trees and graphs later.

Questions?

Let’s practice!

https://github.com/Dijkstra-LLC/dsa_live_pro/tree/main/W05D01/classwork

Presented by Alexander Mak