

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333



What is the worst-case time complexity of this `censor()` definition, if `lyrics` references a `DynamicArrayList` and $N = \text{lyrics.size}()$?

```
1  /** Replaces all instances of the given `word`
2   *   with "****" in these `lyrics`. */
3  static void censor(DynamicArrayList<String> lyrics,
4                    String word) {
5      while(lyrics.contains(word)) { O(N) iterations
6          int i = lyrics.indexOf(word); O(N) search
7          lyrics.set(i, "****");
8      }
9  } using get()/set() is faster - random access
```

$O(1)$ (A)

$O(N)$ (B)

$O(N^2)$ (C)

$O(N^3)$ (D)

next lecture: another fast implementation

Time Complexity of DynamicArrayList

DynamicArrayList

insert(int index, T elem): void

remove(int index): T

size(): int

get(int index): T

set(int index, T elem): void

contains(T elem): boolean

indexOf(T elem): int

delete(T elem): void

add(T elem): void

$\{O(N)$, need to shift elems when index=0

$O(1)$

$\{O(1)$ random access guarantee

$\{O(N)$ find() does linear search

$O(N)$ because of resize

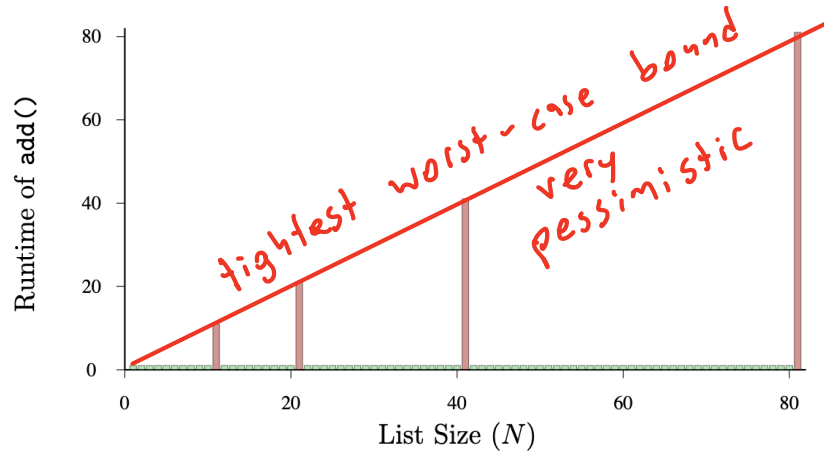
Amortized Time Complexity

Let's think a bit more about the runtime of `add()`.

- Usually, just write to one array cell, update size
 $O(1)$ operation

- Infrequently, resize and copy, $O(N)$ operation

We'd like a notion of the "typical" runtime
long-run average

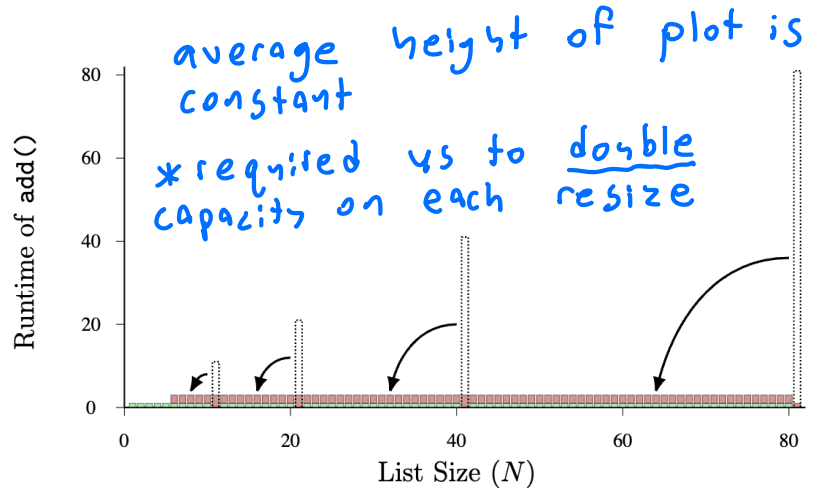


Amortized Time Complexity

= the total complexity of a *sequence* of method calls, divided by the number of calls.

≈ the average / expected performance

add() ing N elements to an empty Dynamic ArrayList requires $O(N)$ total work, so add() has $O(1)$ amortized runtime complexity.





Lecture 13: Linked Data

CS 2110

October 6, 2026

Today's Learning Outcomes

- 49. Implement a generic class or method with one or more generic type parameters. Use generic classes in client code.
- 52. Compare the performance of a List implemented with a dynamic array and a List implemented with a linked chain. Determine which is preferable for a given use case.
- 53. Draw node diagrams to visualize linked data structures. Implement methods on linked data structures.

DynamicArrayList Performance

$O(N)$ memory usage

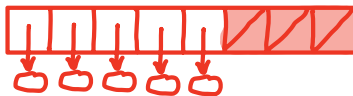
$O(1)$ `get()` and `set()` operations

- "Random access guarantee" of backing storage array

Amortized $O(1)$ `add()` at end of list

$O(N)$ memory copies / shifts to support `insert()` / `remove()` at arbitrary indices (e.g. at beginning of list)

- "Dense", centralized storage



Decentralized Storage

Arrays are "centralized" collection objects that "know about" all their contents.

Benefit: Random access guarantee

Drawback: Expensive modifications

Alternate approach: split storage across multiple objects that each have a "local view" of the collection

– Need a way to navigate between these objects to interact with all data

Analogy: Large Textbook vs. Research Articles
(Centralized) (De-Centralized)

Nodes and Linked Chains

The smaller objects in a decentralized data structure are called its **nodes**.

Each node:

- Stores one element of data
- Holds a reference (*links*) to the "next" node

```
class Node<T> {  
    /** The element in this node. */  
    T data;  
    /** The next node in the chain. */  
    Node<T> next;  
}
```

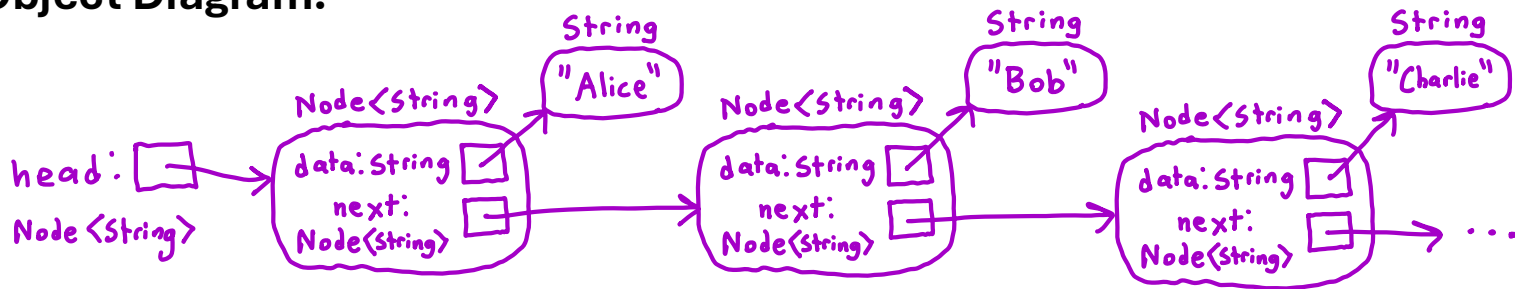
Linking multiple nodes together forms a "chain"

First Node in chain is called the "head"

Starting from the head, we can access any element by following enough links.

Visualizing a Linked Chain (of Strings)

Object Diagram:



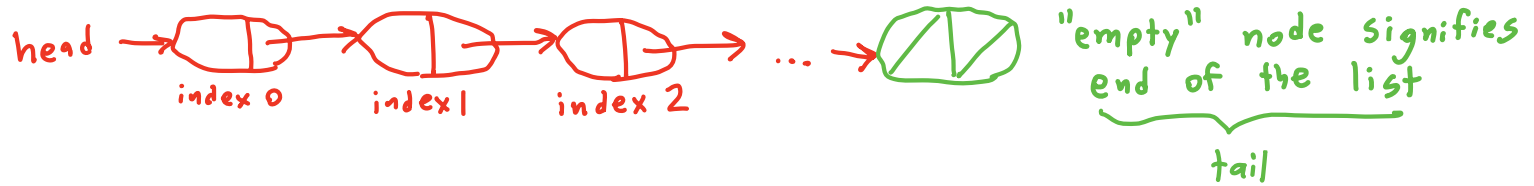
Node Diagram: Retain "link structure", abstract away other details



Linked Lists

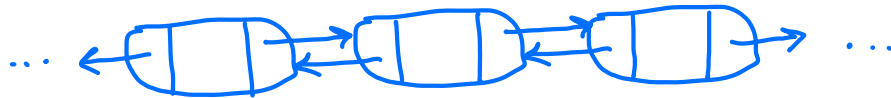
We can use a linked chain to implement the CS2110List interface.

We'll link the nodes in index order with the **head** node at index 0.



"SinglyLinked List" class, since each Node links to a single other Node

vs. "DoublyLinkedList" with backward pointers



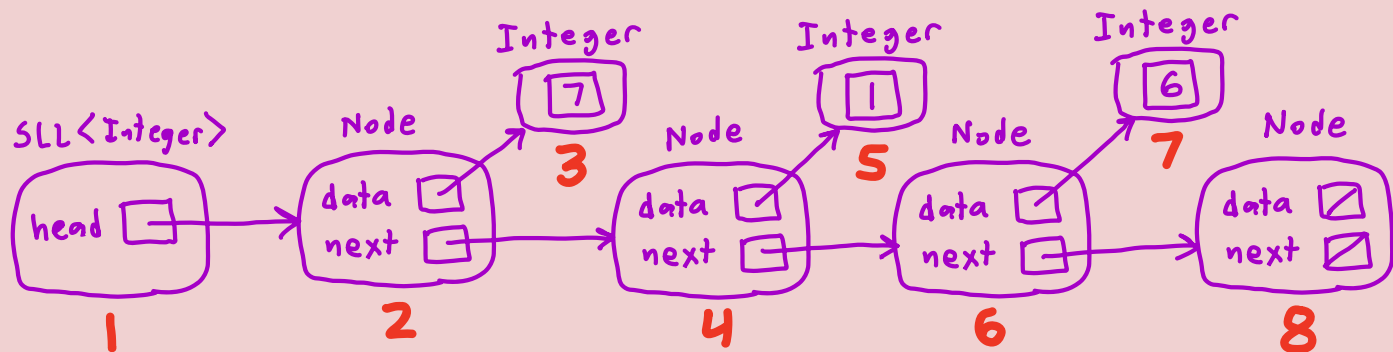
Poll Everywhere

PollEv.com/javabear

text javabear to 22333



How many heap objects are required to represent a `SinglyLinkedList<Integer>` object with size = 3?



4

(A)

7

(C)

6

(B)

8

(D)



Coding Demo: SinglyLinkedList State



Nested Classes

Sometimes, an auxiliary class is needed to help model the state of an object.

The client doesn't need to know/worry about this class, so we can **encapsulate** it from their view by nesting it inside of the main (outer) class.

Two "Flavors":

non-static "inner" classes

have access to fields of
"outer object"

✓ preferred
static nested classes

don't have access to fields
of "outer object"

everything in nested class is visible to outer class



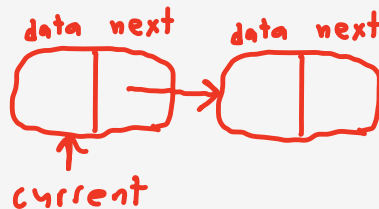
Coding Demo: Initialization



nodeAtIndex() Helper Method

Working with your neighbor, complete the definition of this method.

```
/** Returns a reference to the node at the given `index` (counting from 0) in
 * this linked list. Requires that `0 <= index <= size`. */
private Node<T> nodeAtIndex(int index) {
    assert 0 <= index && index <= size; // defensive programming
    Node<T> current = head;    int i = 0;
    while (i < index) { // loop inv: current = ith node in list
        current = current.next;    i++;
    }
    return current;
}
```





Coding Demo: Using `nodeAtIndex()`



Complexity Analysis

$\text{nodeAtIndex}(i)$ runs in $O(i)$ time, $O(1)$ per link traversal

- $O(1)$ space complexity ($O(i)$ if recursive)

$\text{get}()$ / $\text{set}()$ are $O(i)$, worst-case $O(N)$, operations
for linked lists

worse than dynamic arrays, de-centralization makes
navigation trickier since we lose random access guarantee

Other "Scanning" Methods

`contains(T elem)`

Idea: Follow links from head, comparing data field of each node with `elem`. Early return `true` if we find a match. Return `false` if we reach the tail.

`indexOf(T elem)`

Idea: Follow links from head, comparing data field of each node with `elem`. Have a local variable keeping track of indices during the traversal, and return its value when a match is found.

Exercise (on your own): Code these up and document their loop invariants

Both $O(N)$ operations due to linear search, same as `DynamicArrayList`

Adding Nodes: spliceIn()

Modifying linked chains (adding/removing elems) amounts to "re-wiring" their links.

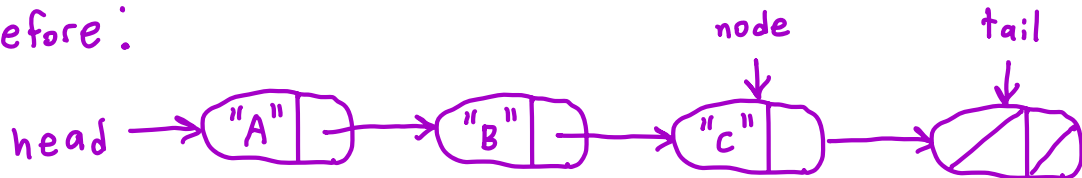
```
/** Adds the given `elem` just before this existing list `node`. */  
private void spliceIn(Node<T> node, T elem) { ... }
```

To improve upon performance of Dynamic Array List, we want these re-wirings to be "local" operations (once we locate node where they take place)

⇒ $O(1)$ runtime of spliceIn()

Strategy: Before/After Node Diagrams

Before:

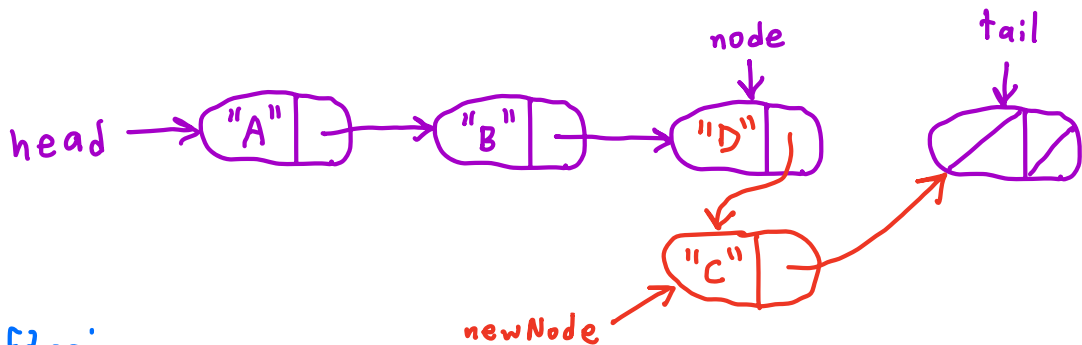


1. make new Node
copying node

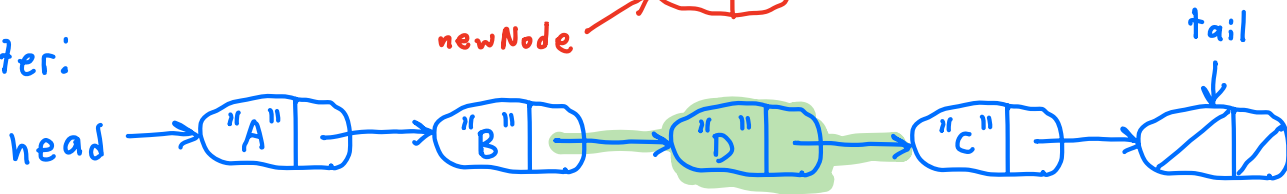
2. update node.data
to elem

3. update node.next
to newNode

4. increment size
(5. fix tail)



After:



Poll Everywhere

PollEv.com/javabear

text **javabear** to 22333



What are the time complexities of adding elements to the beginning / end of a SinglyLinkedList?

Beginning: $O(1)$

End: $O(1)$

(A)

Beginning: $O(1)$

End: $O(N)$

(B)

Beginning: $O(N)$

End: $O(1)$

(C)

Beginning: $O(N)$

End: $O(N)$

(D)



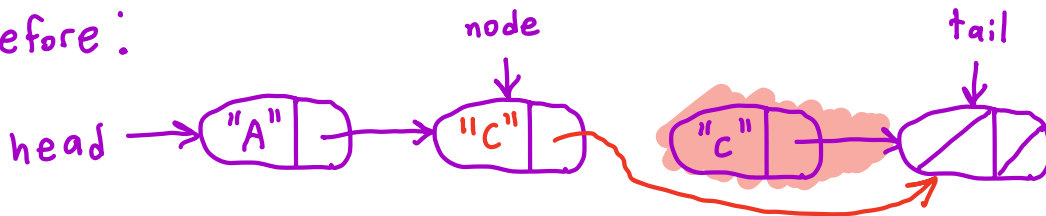
Coding Demo: `add()` / `insert()`



Removing Nodes: spliceOut()

```
/** Removes the given `node` from this list and returns its `data` field.  
 * Requires that `node != tail`. */  
private T spliceOut(Node<T> node) { ... }
```

Before:



1. Store return value



2. Update

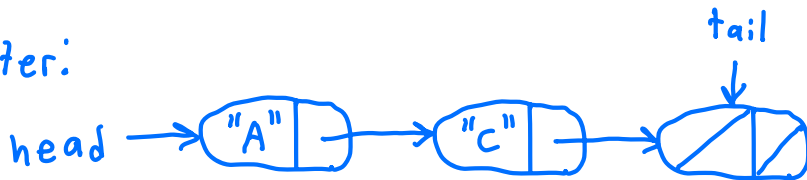
$\text{node.data} = \text{node.next.data}$

3. Update

$\text{node.next} = \text{node.next.next}$

4. Update size (+ tail)

After:



Poll Everywhere

PollEv.com/javabear

text **javabear** to 22333



What are the time complexities of removing elements from the beginning / end of a SinglyLinkedList?

Beginning: $O(1)$

End: $O(1)$

(A)

Beginning: $O(1)$

End: $O(N)$

(B)

Beginning: $O(N)$

End: $O(1)$

(C)

Beginning: $O(N)$

End: $O(N)$

(D)



Coding Demo: `remove()` / `delete()`



SinglyLinkedList Summary

$O(N)$ memory usage $\begin{cases} \text{all space "actively used"} \\ \text{pointers take up extra space} \end{cases}$

$O(1)$ re-wiring operations (given reference to location) for addition/removal

- $O(1)$ worst-case `add()`/`remove()` at beginning, `add()` at end
- No global resizing / memory shifting

$O(N)$ linear searching

$O(N)$ element access by index (`get()` / `set()`)

Linked Lists good for frequent updates at ends, bad for queries in middle