Poll Everywhere

PollEv.com/2110fa25   text 2110fa25 to 22333

What are the contents of `list` after these operations?

```
public static void main(String[] args) {
    CS2110List<String> list = new DynamicArrayList<>();
    list.add("apple");
    list.add("grape");
    list.insert(1, "banana");
    list.insert(0, "grape");
    list.set(2, "orange");
    list.remove(1);
    list.delete("grape");
}
```

See lecture demo code that visualizes all operations.

["orange", "grape"]

Lecture 13: Linked Data

CS 2110

October 7, 2025

Today's Learning Outcomes

53. Implement a generic class or method with one or more generic type parameters. Use generic classes in client code.

57. 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.

58. Draw object (or node) diagrams to visualize linked data structures. Implement methods on linked data structures.

DynamicArrayList Performance

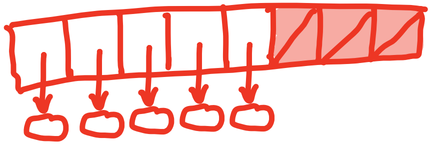
O(N) memory usage

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 single collection objects that "know about" all of their contents.

- Benefits (random access) + drawbacks (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 with Citations

(Centralized) (De-Centralized)

Nodes and Linked Chains

Smaller objects = "Nodes"

Each node:

- Carries small amount (1 element) of data
- Holds a reference (i.e., links) to another Node, the "next" node

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

    /** The next node in the chain. */
    Node<T> next;
}
```

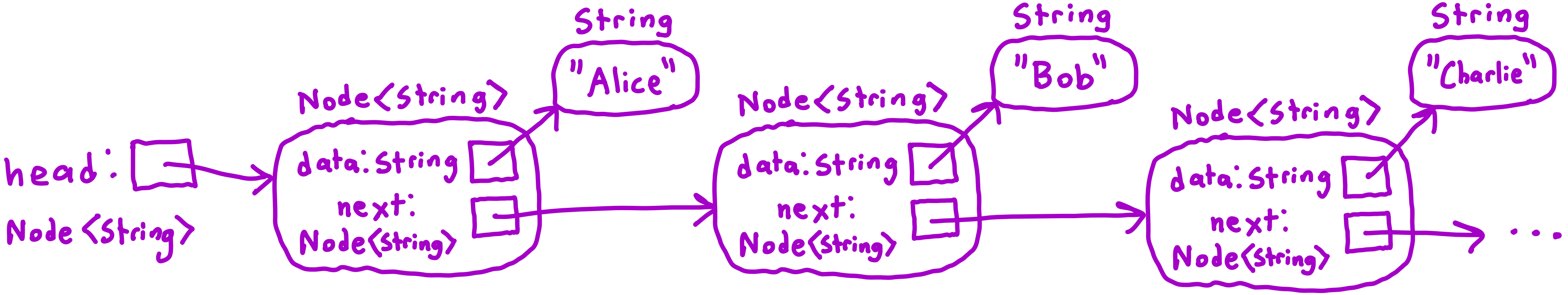
Linking multiple nodes together forms a "chain"

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:

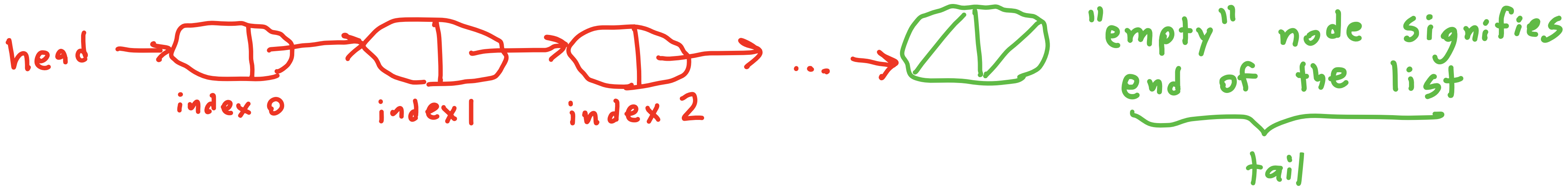


Node Diagram: Retain "link structure", abstract away other details

Linked Lists

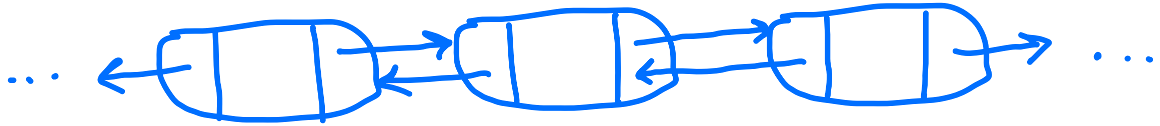
We can use a linked chain to implement the CS2110List interface

Nodes are linked in index order with head at index 0



"SinglyLinkedList" class, since each Node links to a single other Node

vs. "DoublyLinkedList" with backward pointers

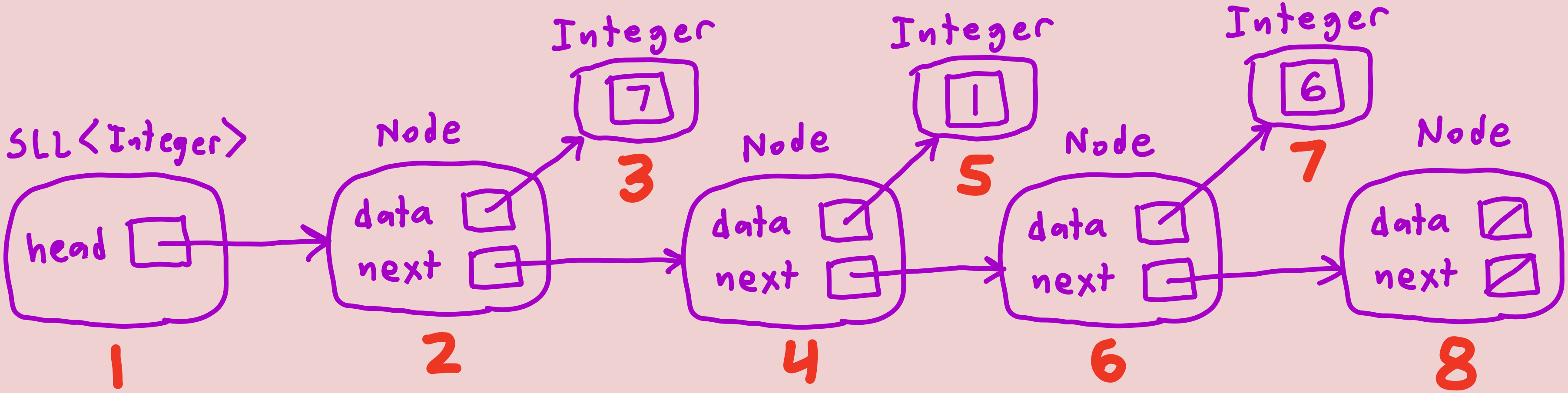


Focus of A6

Poll Everywhere

PollEv.com/2110fa25 text 2110fa25 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 main class.

Two "Flavors":

preferred ↙

static nested classes

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

non-static "inner" classes

have access to fields of "outer object"

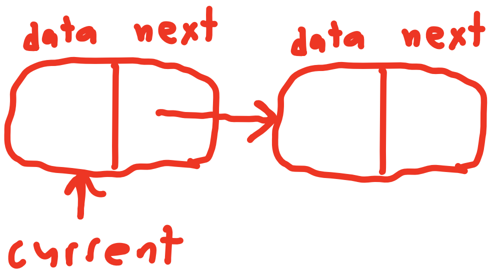
everything in nested class is visible to outer class

Coding Demo: Constructor / `assertInv()`

`nodeAtIndex()` Helper Method

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

```java
/** 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

nodeAtIndex(i) runs in O(i) time, O(1) per link traversal
- O(1) space complexity (O(i) if recursive)

get() / 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, return false at tail.

indexOf(T elem)

Idea: Follow links from head, comparing data field of each node with elem. Keep track of indices during traversal.

Both O(N) operations – same as DynamicArrayList – linear search

Exercise: (on your own)
- code these up
- write loop invariants

Adding Nodes: `spliceIn()`

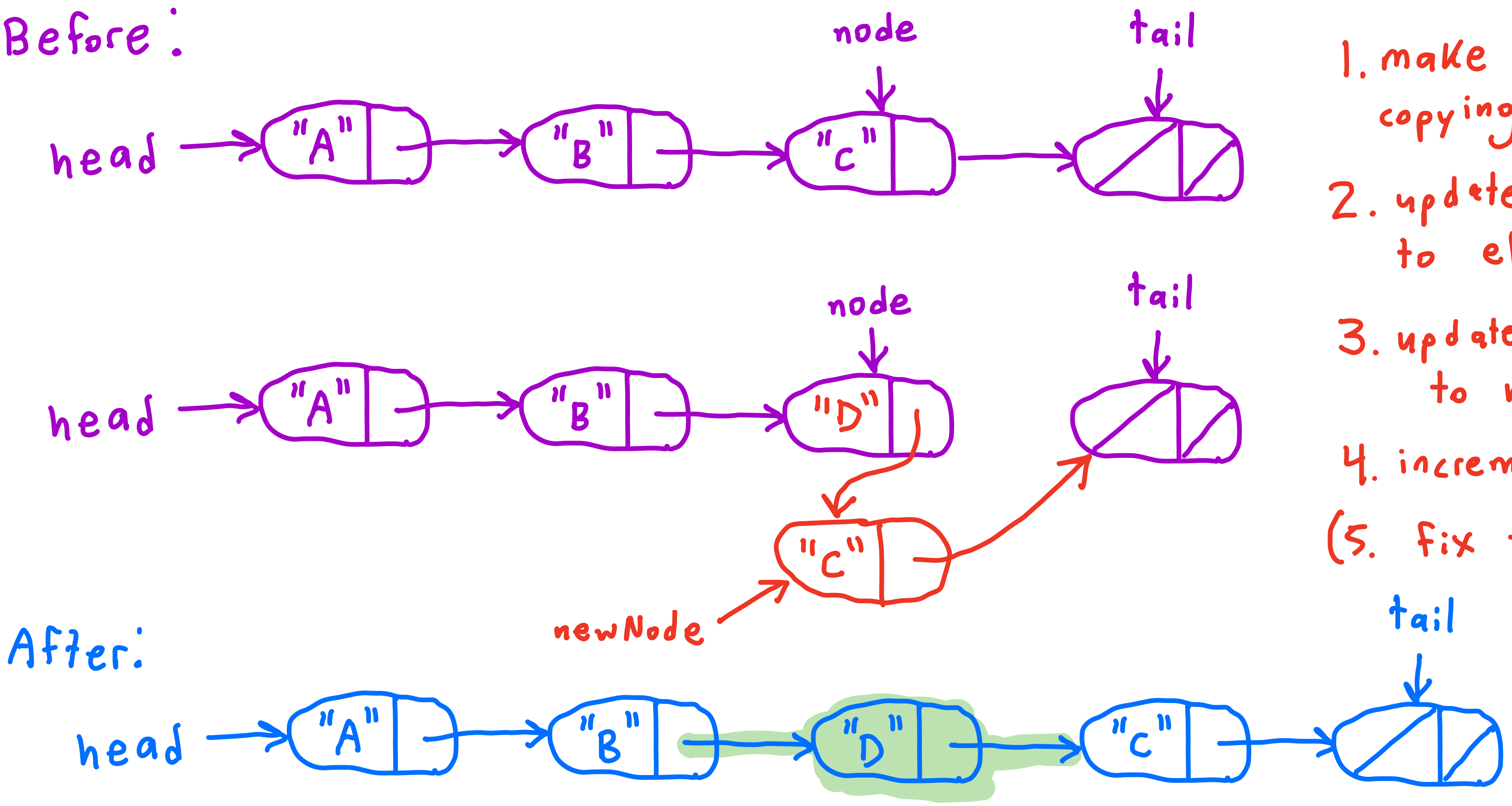
```
/** Adds the given `elem` just before this existing list `node`. */
private void spliceIn(Node<T> node, T elem) { ... }
```

Modifying (adding/removing elements from) a linked chain amounts to rewiring its links.

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



1. make new Node copying node
2. update node.data to elem
3. update node.next to newNode
4. increment size
(5. fix tail)

Poll Everywhere

PollEv.com/2110fa25 text `2110fa25` to `22333`

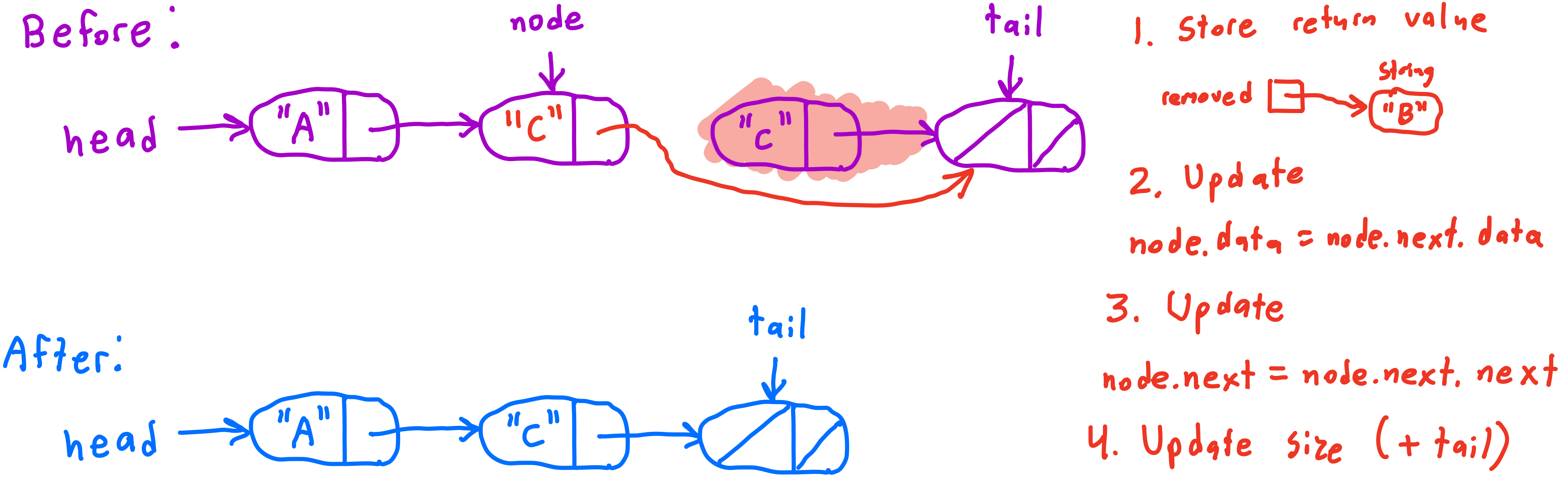
What are the time complexities of adding elements to the beginning / end of a `SinglyLinkedList`?

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) { ... }
```



Poll Everywhere

PollEv.com/2110fa25 text `2110fa25` 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 { all space "actively used" / pointers take up extra space }

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