

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

$O(1)$

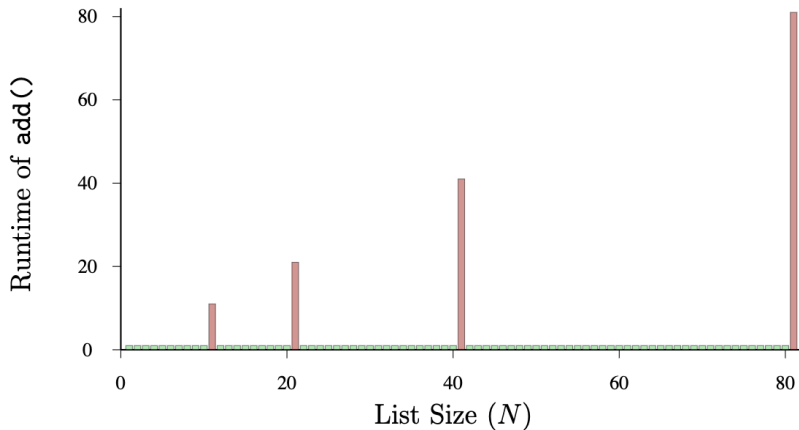
$\{O(1)$ random access guarantee

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

$O(N)$ because of resize

Amortized Time Complexity

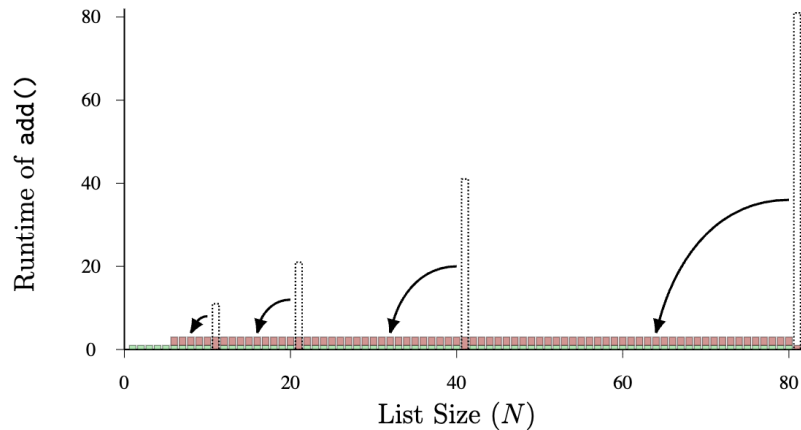
Let's think a bit more about the runtime of `add()`.



Amortized Time Complexity

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

≈ the average / expected performance





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

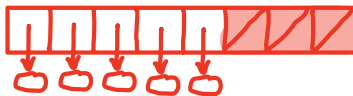
$O(1)$ `get()` and `set()` operations

- "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

Nodes and Linked Chains

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

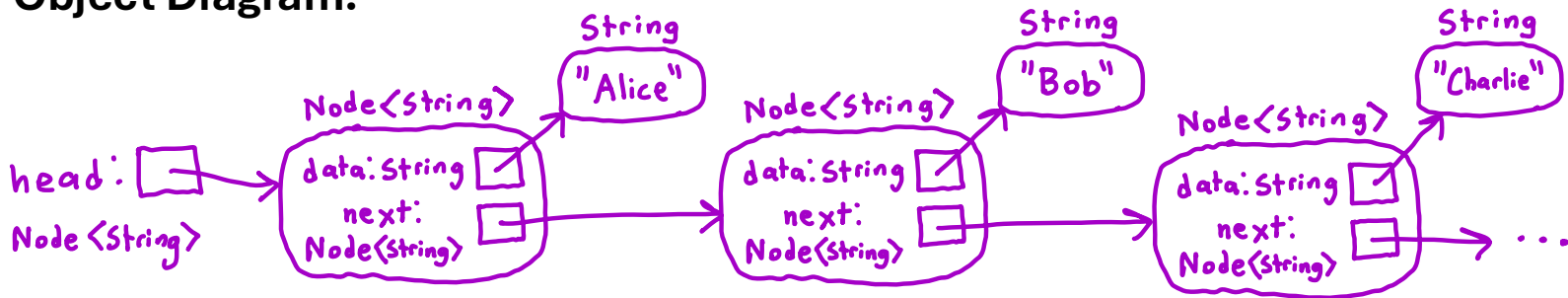
Each node:

- 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;  
}
```

Visualizing a Linked Chain (of Strings)

Object Diagram:



Node Diagram:

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.



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.



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

}
```



Coding Demo: Using `nodeAtIndex()`



Complexity Analysis

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`

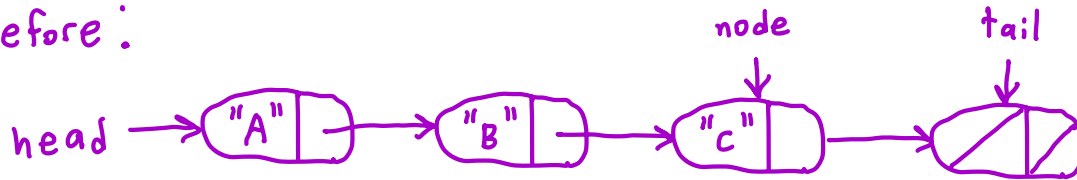
Adding Nodes: spliceIn()

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

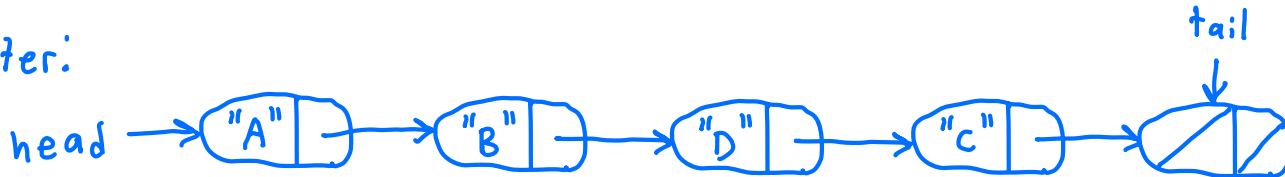
```
/** Adds the given `elem` just before this existing list `node`. */  
private void spliceIn(Node<T> node, T elem) { ... }
```

Strategy: Before/After Node Diagrams

Before:



After:





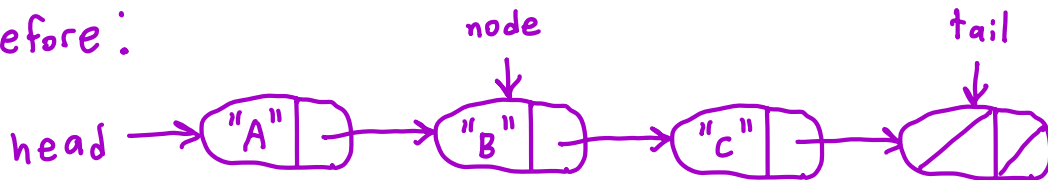
Coding Demo: `add()` / `insert()`



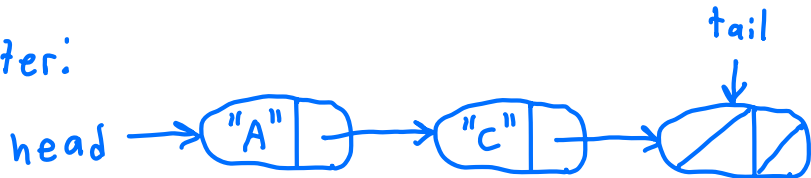
Removing Nodes: spliceOut()

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

Before:



After:





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