Cmp Sci 187: Midterm Review

Based on Lecture Notes

What Did We Cover ?

- Basic Java (review)
- Software Design (Phone Directory)
- Correctness and Efficiency:

  Exceptions, Testing, Efficiency (Big-O)
- Inheritance and Class Hierarchies
- Lists and the **`Collection`** Interface

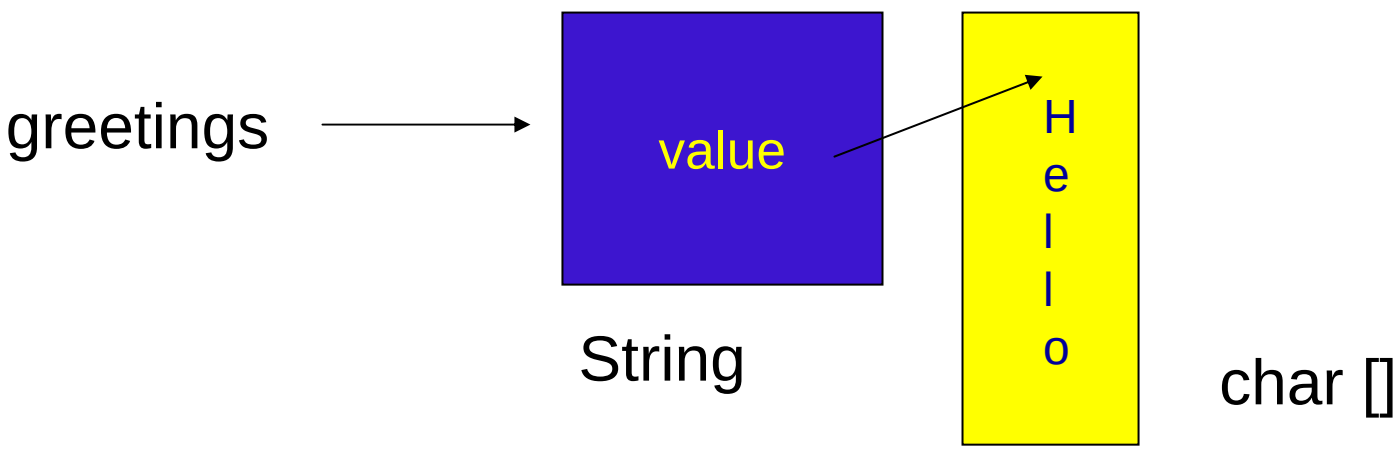
  Building Block for Fundamental Data Structures
- Stacks: Perhaps the Simplest Data Structure
- Queues: The Second Simplest

Classes and Objects

- The ***class*** is the unit of programming
- A Java program is a ***collection of classes***
- A class describes ***objects (instances)***
  - Describes their common characteristics: is a *blueprint*
  - Thus all the instances have these same characteristics
- These characteristics are:
  - ***Data fields*** for each object
  - ***Methods*** (operations) that do work on the objects

Methods: Referencing and Creating Objects

- You can **declare reference variables**
  - They reference objects of **specified types**
- Two reference variables can reference **the same object**
- The **new** operator creates an instance of a class
- A **constructor** executes when a new object is created
- Example: **`String greeting = "Hello";`**



Abstract Data Types, Interfaces

- A major goal of software engineering: *write reusable code*
- **Abstract data type** (ADT): data + methods
- A **Java interface** is a way to specify an ADT
  - Names, parameters, return types of methods
  - No indication of *how* achieved (procedural abstraction)
  - No representation (data abstraction)
- A class may ***implement*** an interface
  - Must provide bodies for all methods of the interface

Java 5

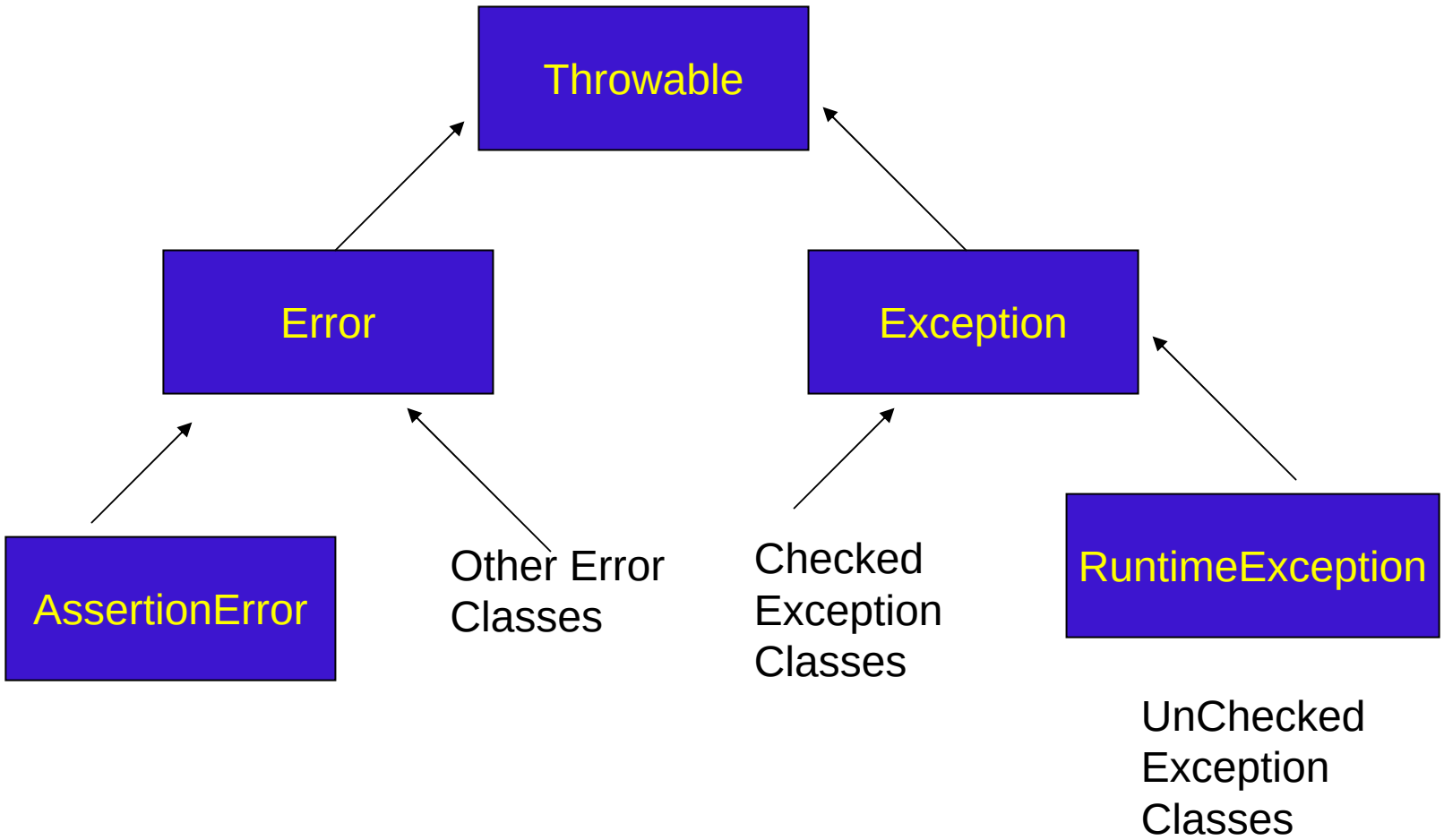
- Generics
  - **`ArrayList<String> = new ArrayList<String>();`**
- Inner Classes
  - Block Scoping, can make use of fields of outer class
  - Static nested class
- Auto (Un)Boxing
  - Primitive <-> wrapped object

Exceptions

- Categories of program errors
- Why you should catch exceptions
- The **`Exception`** hierarchy
  - Checked and unchecked exceptions
- The **`try-catch-finally`** sequence
- Throwing an exception:
  - What it means
  - How to do it

The Class Throwable

- **`Throwable`** is the superclass of all exceptions
- All exception classes inherit its methods



Efficiency

- Big-O notation
  - What it is
  - How to use it to analyze an algorithm’s efficiency

Efficiency Examples

```
for (int i = 1; i < n; i *= 2) {
  do something with x[i]
}
```

Sequence is 1, 2, 4, 8, ..., ~n.

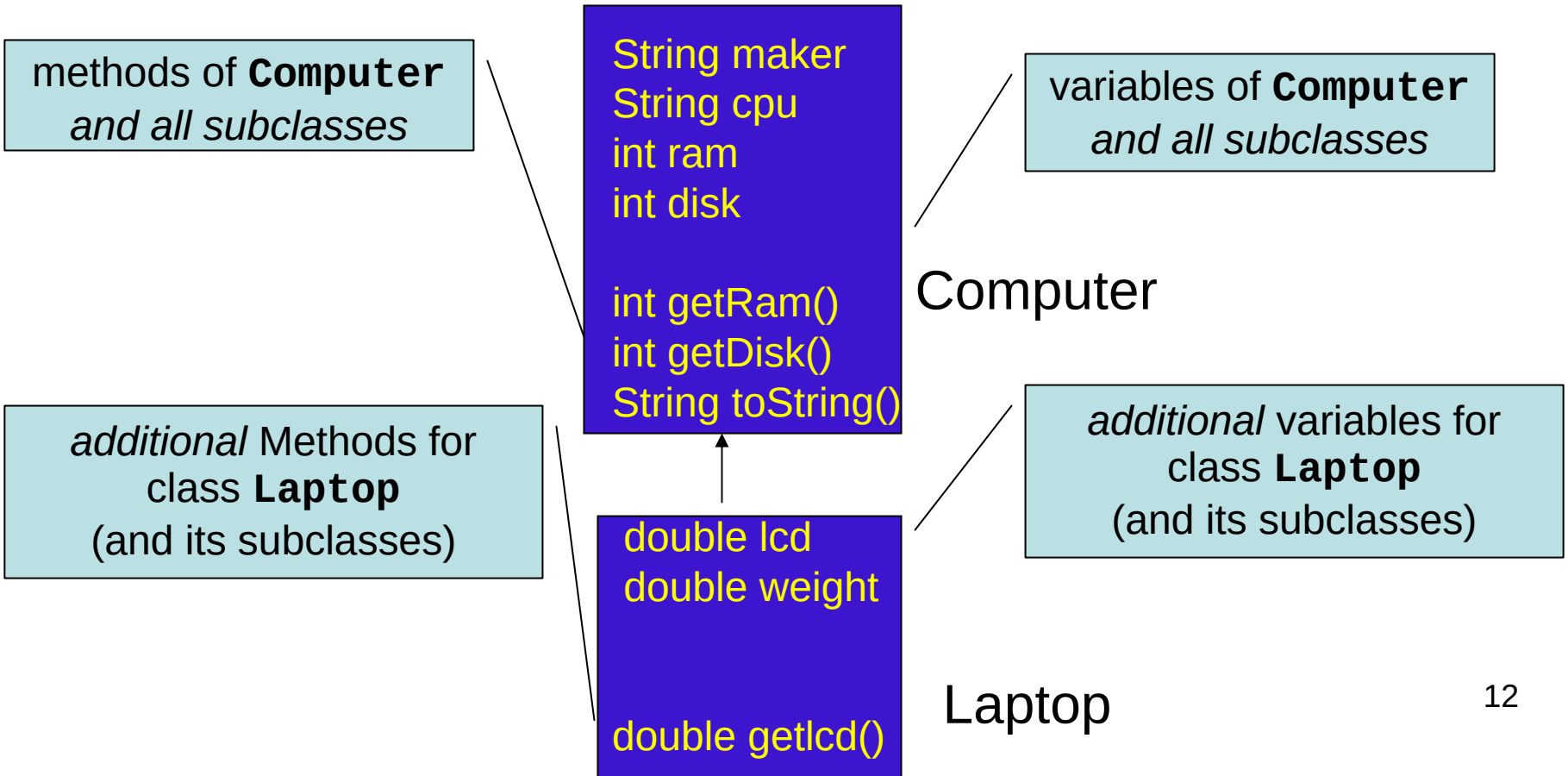
Number of iterations = $\log_2 n = \log n$.

Inheritance

- *Inheritance* and how it facilitates code reuse
- How does Java *find the "right" method* to execute?
  - (When more than one has the same name ...)
- Defining and using *abstract classes*
- Class **`Object`**: its methods and how to override them

A Superclass and a Subclass

- Consider two classes: **`Computer`** and **`Laptop`**
- A laptop is a *kind* of computer: therefore a subclass



Is-a Versus *Has-a* Relationships

- Confusing *has-a* and *is-a* leads to misusing inheritance
- Model a *has-a* relationship with an *attribute* (variable)

```
public class C { ... private B part; ...}
```

- Model an *is-a* relationship with inheritance
  - If every **C** is-a **B** then model **C** as a subclass of **B**
  - Show this: in **C** include **extends B**:

```
public class C extends B { ... }
```

Class **`Object`**

- **`Object`** is the root of the class hierarchy
  - Every *class* has **`Object`** as a superclass
- All classes inherit the methods of Object
  - But may override them

  - boolean equals(Object o)
  - String toString()
  - int hashCode()
  - Object clone()

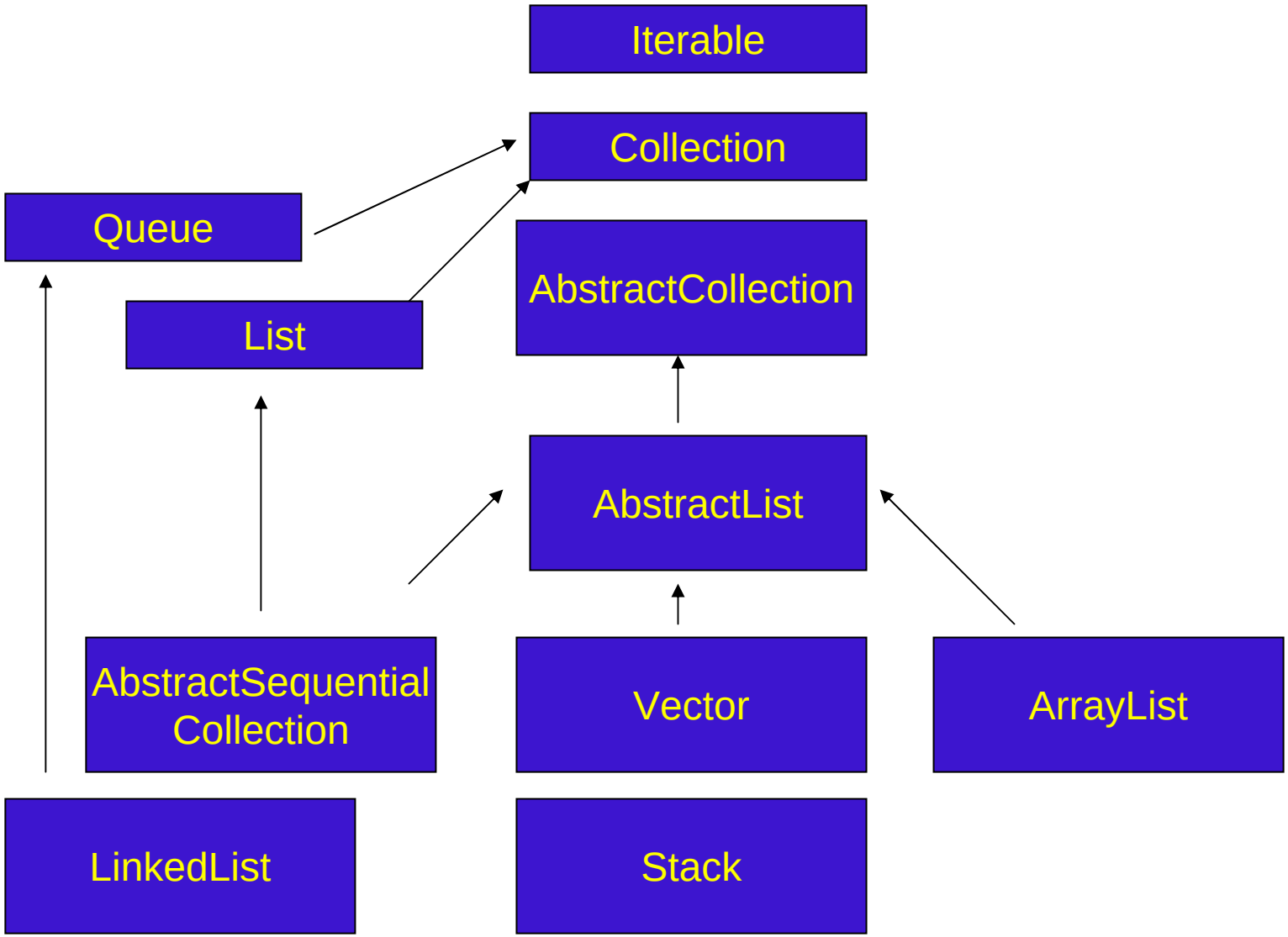
Inheriting from Interfaces vs Classes

- A class can *extend* 0 or 1 superclass
  - Called *single inheritance*
- An interface cannot extend a class at all
  - (Because it is not a class)
- A class or interface can *implement* 0 or more interfaces
  - Called *multiple inheritance*

Inheritance

- Java does ***not*** implement *multiple inheritance*
- Get some of the advantages of multiple inheritance:
  - *Interfaces*
  - *Delegation*
- Sample class hierarchy: *drawable shapes*

Collection Hierarchy

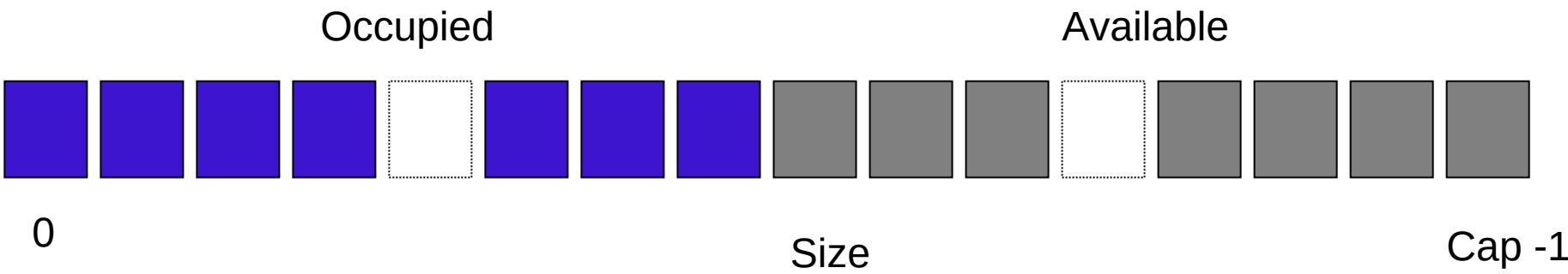


Lists (1)

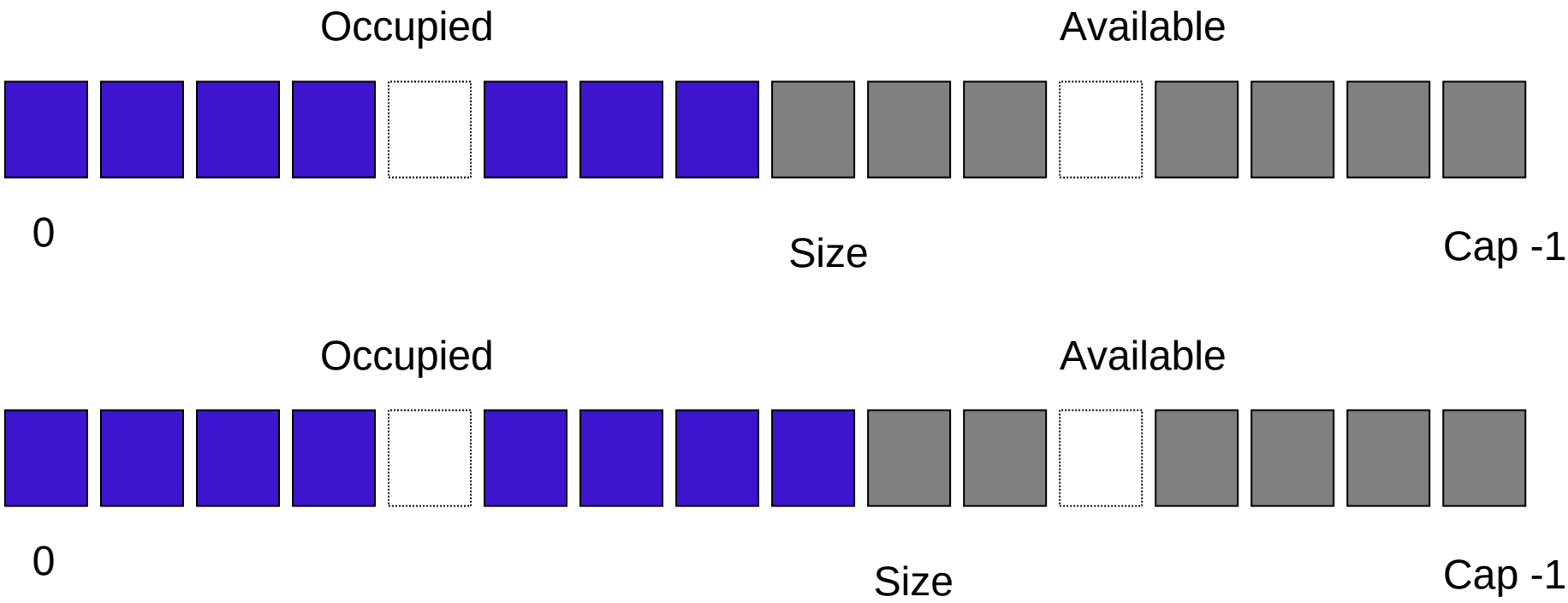
- The **`List`** interface
- Writing an array-based implementation of **`List`**
- Linked list data structures:
  - Singly-linked
  - Doubly-linked
  - Circular
- Implementing **`List`** with a linked list
- The **`Iterator`** interface
  - hasNext(), next(), remove()
- Implementing **`Iterator`** for a linked list

Implementing an **`ArrayList`** Class

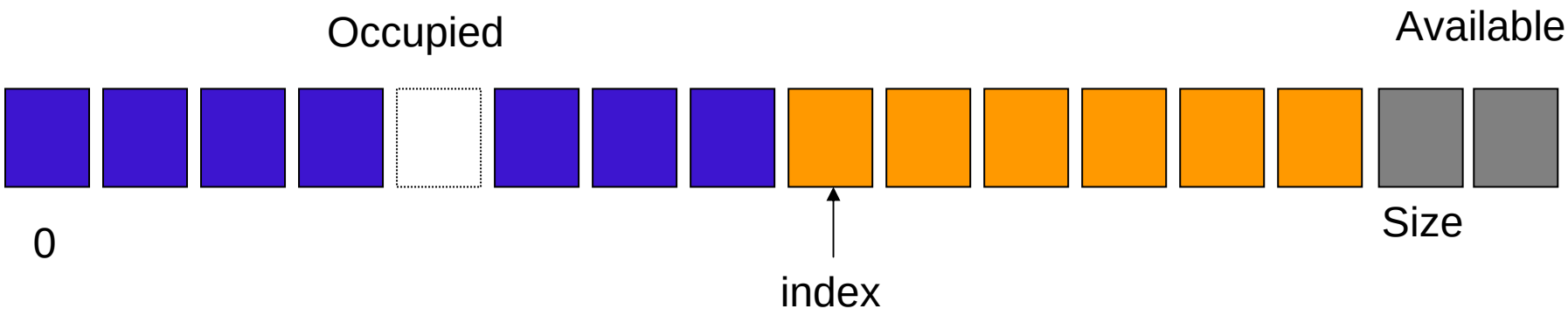
- **`KWArrayList`**: simple implementation of **`ArrayList`**
  - Physical size of array indicated by data field **`capacity`**
  - Number of data items indicated by the data field **`size`**



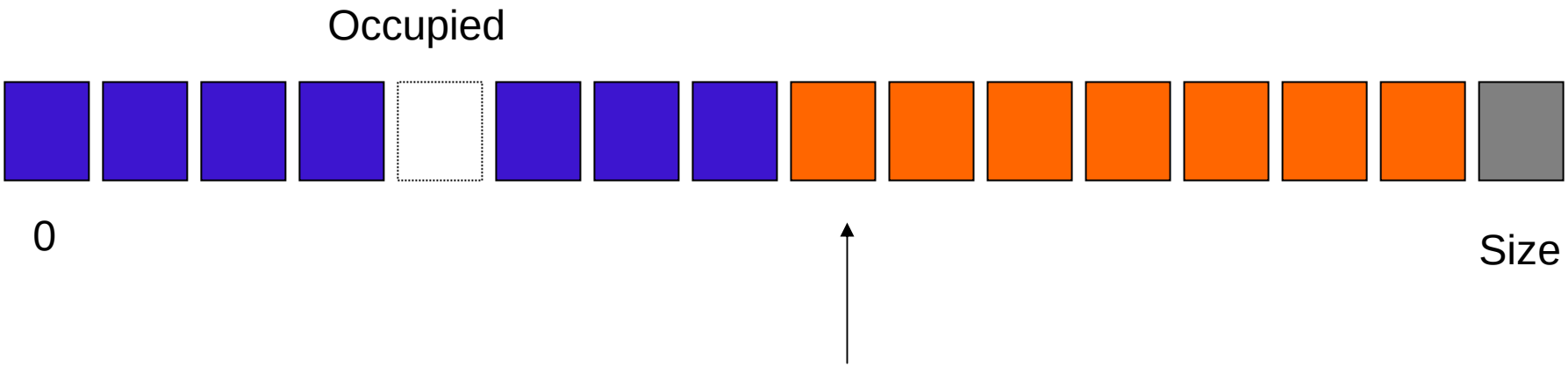
Implementing **`ArrayList.add(E)`**



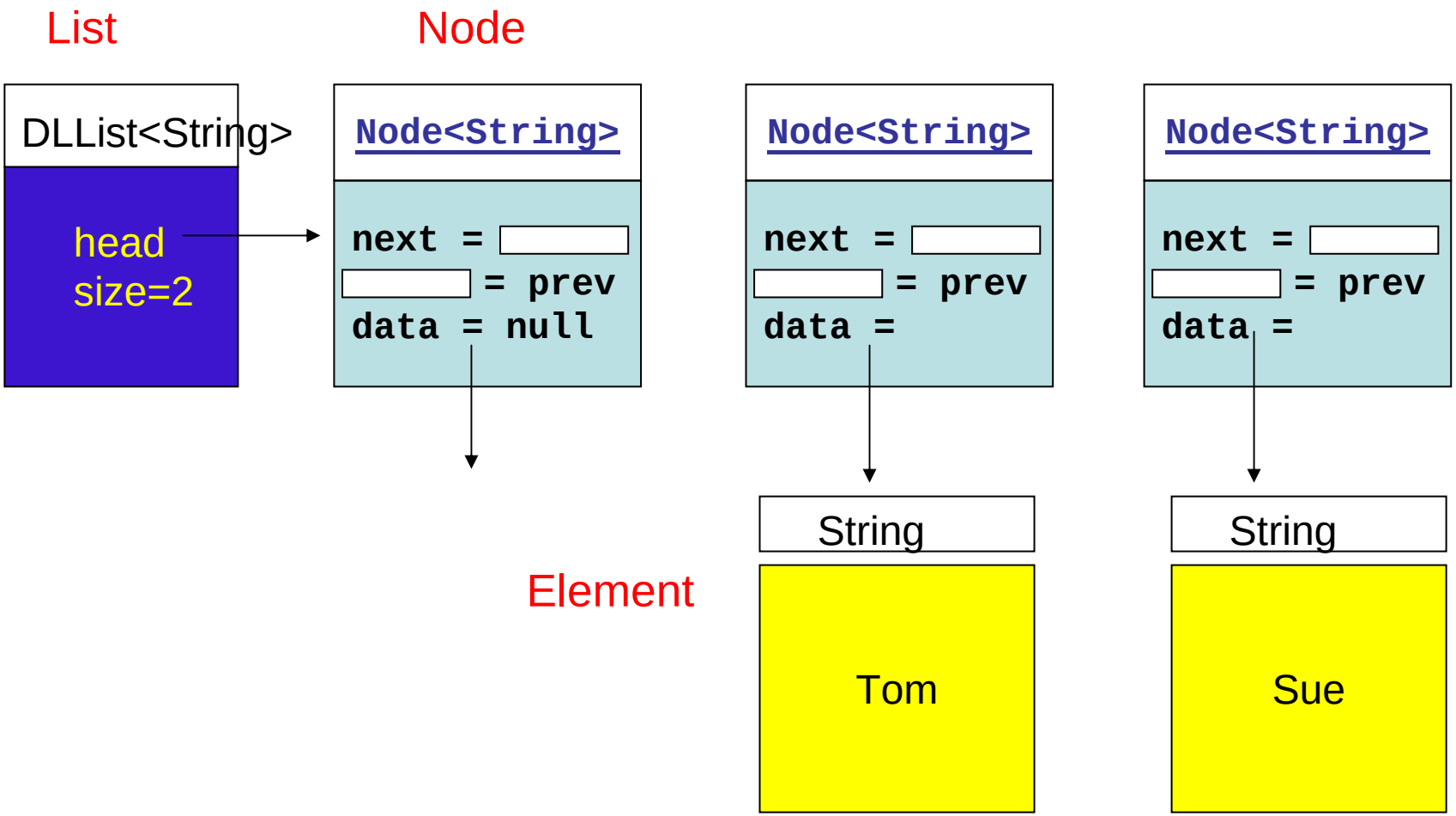
Implementing **`ArrayList.add(int,E)`**



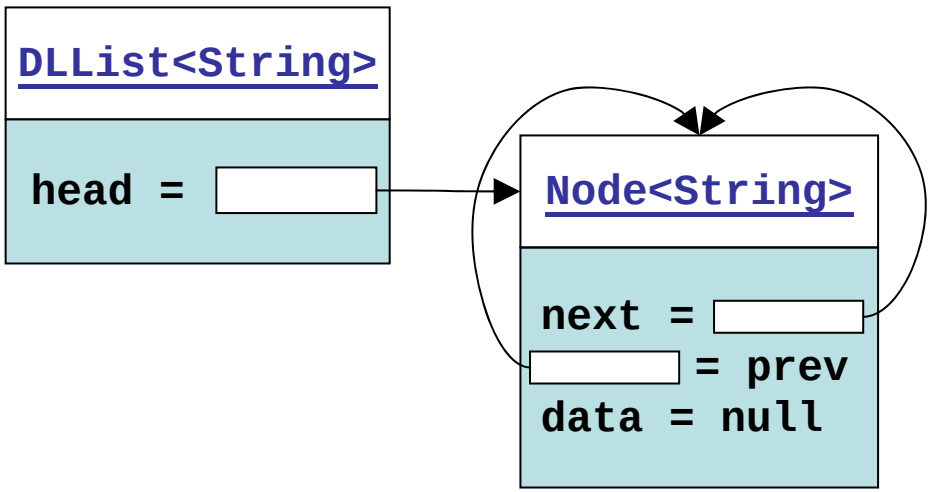
Implementing **`ArrayList.remove(int)`**



Linked List

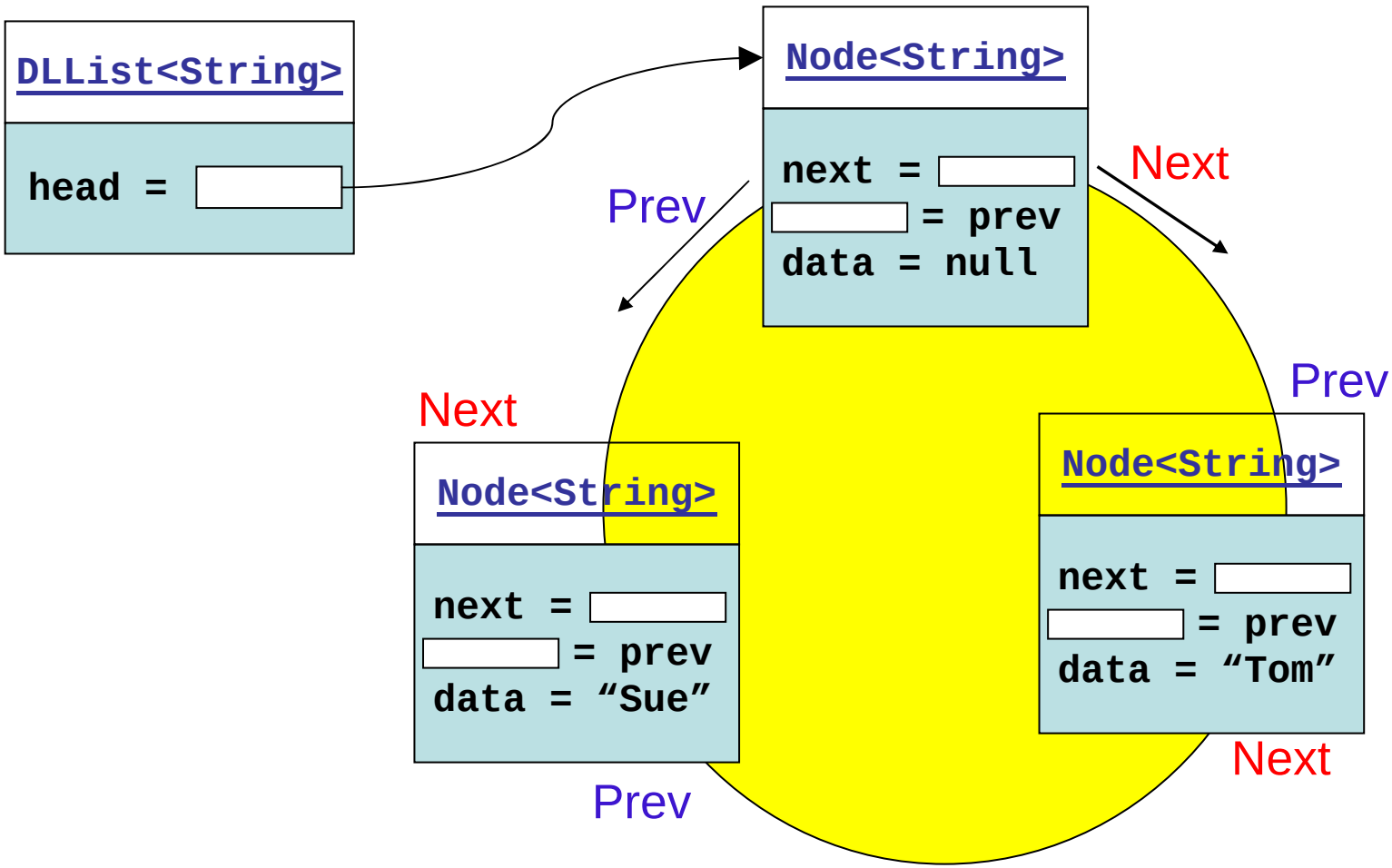

List
Node
DLList<String>
head
size=2
Node<String>
next =
= prev
data = null
Node<String>
next =
= prev
data =
Node<String>
next =
= prev
data =
String
String
Element
Tom
Sue

Implementing **`DLList`** With a “Dummy” Node

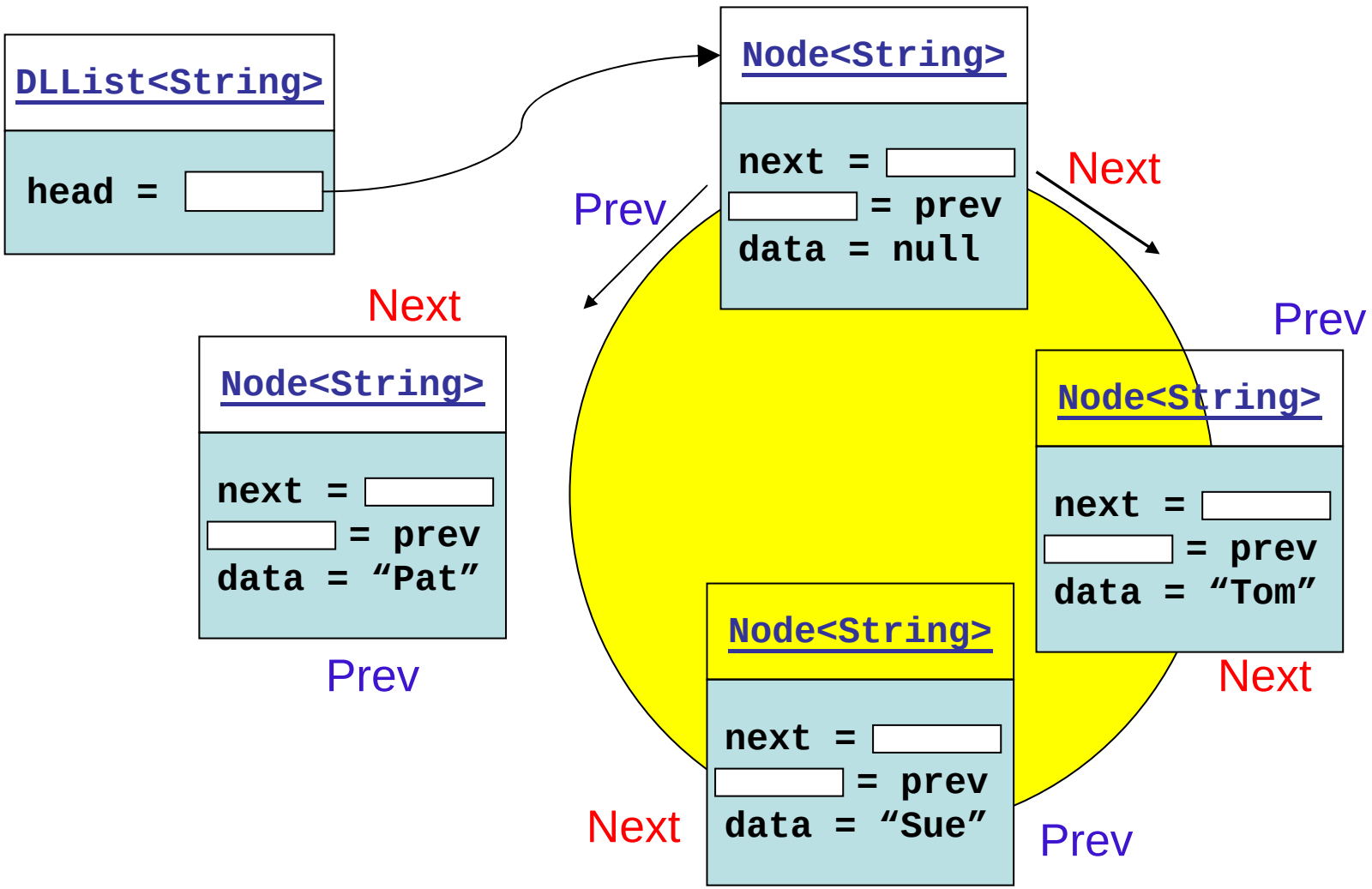


- The “dummy” node is always present
- Eliminates null pointer cases
  - Even for an empty list
- Effect is to simplify the code
- Helps for singly-linked and non-circular too

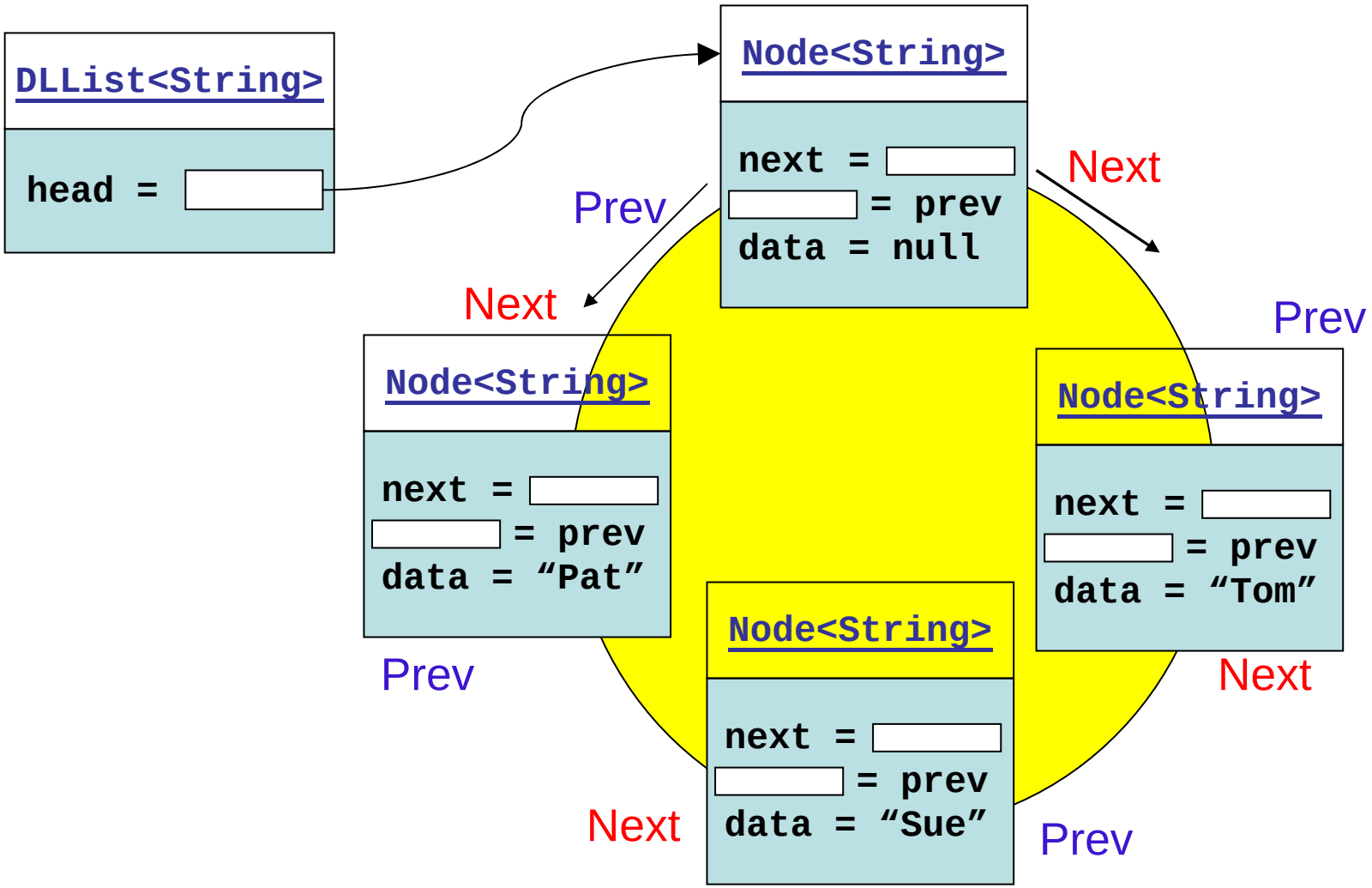
Implementing **DLList** Circularly



`DLList` Insertion



DLList Removal



Stacks

- The **`Stack<E>`** data type and its four methods:
  - **`push(E)`**, **`pop()`**, **`peek()`**, and **`empty()`**
- How the Java libraries implement **`Stack`**
- How to implement **`Stack`** using:
  - An array
  - A linked list
- Using **`Stack`** in applications
  - Finding palindromes
  - Testing for balanced (properly nested) parentheses
  - Evaluating arithmetic expressions

PostFix Form

1 + 2 * 3 + 4

Input	Stack	Output	
			//1
1		1	//2
+	+		//3
2		2	//4
*	*		//5
3		3	//6
+	===	* +	//7
	+		//8
4		4	//9
		+	//10

Evaluate Postfix

1 2 3 * + 4 +

Input	Stack	
		//1
1	1	//2
2	2 1	//3
3	3 2 1	//4
*	6 1	//5
+	7	//6
4	4 7	//7
+	11	//8
11		//9

Queue (1)

- Representing a waiting line, i.e., *queue*
- *FIFO*
- The methods of the **`Queue`** interface:
  `offer`, **`remove`**, **`poll`**, **`peek`**, and **`element`**
- Implement the **`Queue`** interface:
  - Singly-linked list
  - Circular array (a.k.a., circular buffer)
  - Doubly-linked list

Queue (2)

- Applications of queues:
  - Simulating physical systems with waiting lines ...
  - Using Queues and random number generators