Abstract Data Types

Abstract Data Types

- An **abstract data type** is a mathematical set of data, along with operations defined on that kind of data
- Examples:
  - **int**: it is the set of integers (up to a certain magnitude), with operations +, -, /, *, %
  - **double**: it's the set of decimal numbers (up to a certain magnitude), with operations +, -, /, *

Data Structures

- A data structure is a user-defined abstract data type
- Examples:
  - **Complex numbers**: with operations +, -, /, *, *magnitude*, *angle*, etc.
  - **Stack**: with operations *push*, *pop*, *peek*, *isempty*
  - **Queue**: *enqueue, dequeue, isempty …*
  - **Binary Search Tree**: *insert*, *delete*, *search.*
  - **Heap**: *insert, min, delete-min*.

Data Structure Design

- Specification
  - A set of data
  - Specifications for a number of operations to be performed on the data
- Design
  - A lay-out organization of the data
  - Algorithms for the operations
- Goals of Design: **<u>fast</u>** operations

Implementation of a Data Structure

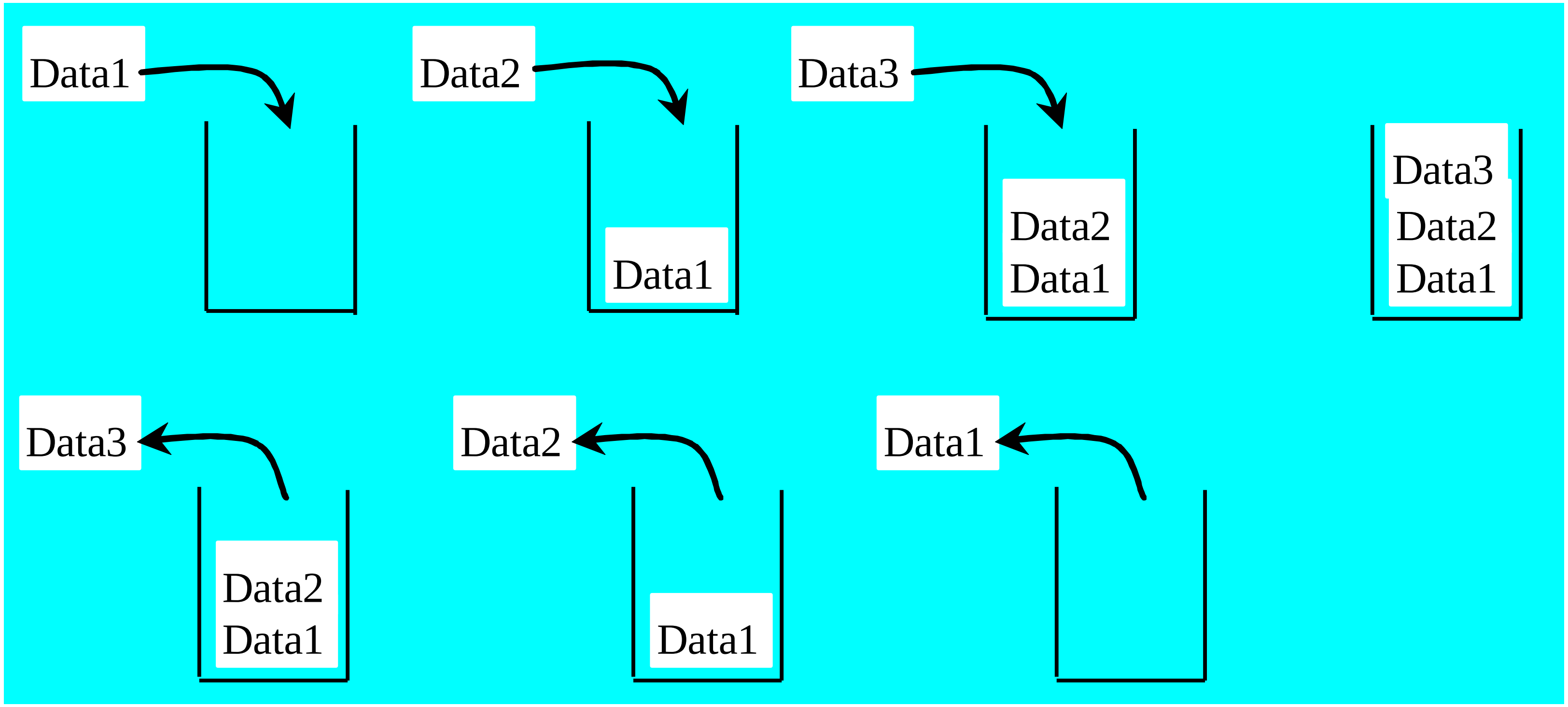
- Representation of the data using built-in data types of the programming language (such as int, double, char, strings, arrays, structs, classes, pointers, etc.)
- Language implementation (code) of the algorithms for the operations

- In OOP languages both the data representation and the operations are aggregated together into what is called **objects**
- The data type of such objects are called **classes**.
- Classes are blue prints, objects are instances.

Stack, Queue and List

Stacks

A stack can be viewed as a special type of list, where the elements are accessed, inserted, and deleted only from the end, called the top, of the stack.

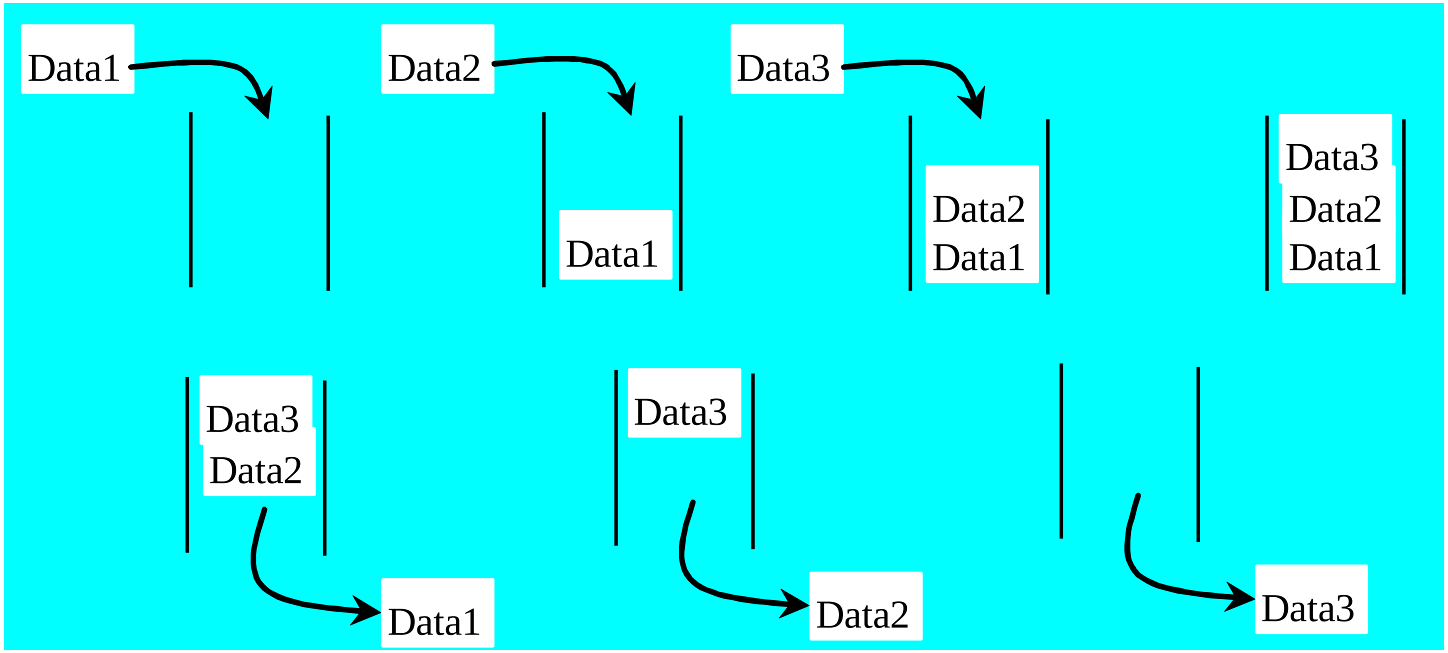


Stack

- A stack is maintained Last-In-First-Out
  (not unlike a stack of plates in a cafeteria)
- Standard operations
  - `isEmpty()`: returns `true` or `false`
  - `top()`: returns copy of value at top of stack (without removing it)
  - `push(v)`: adds a value v at the top of the stack
  - `pop()`: removes and returns value at top

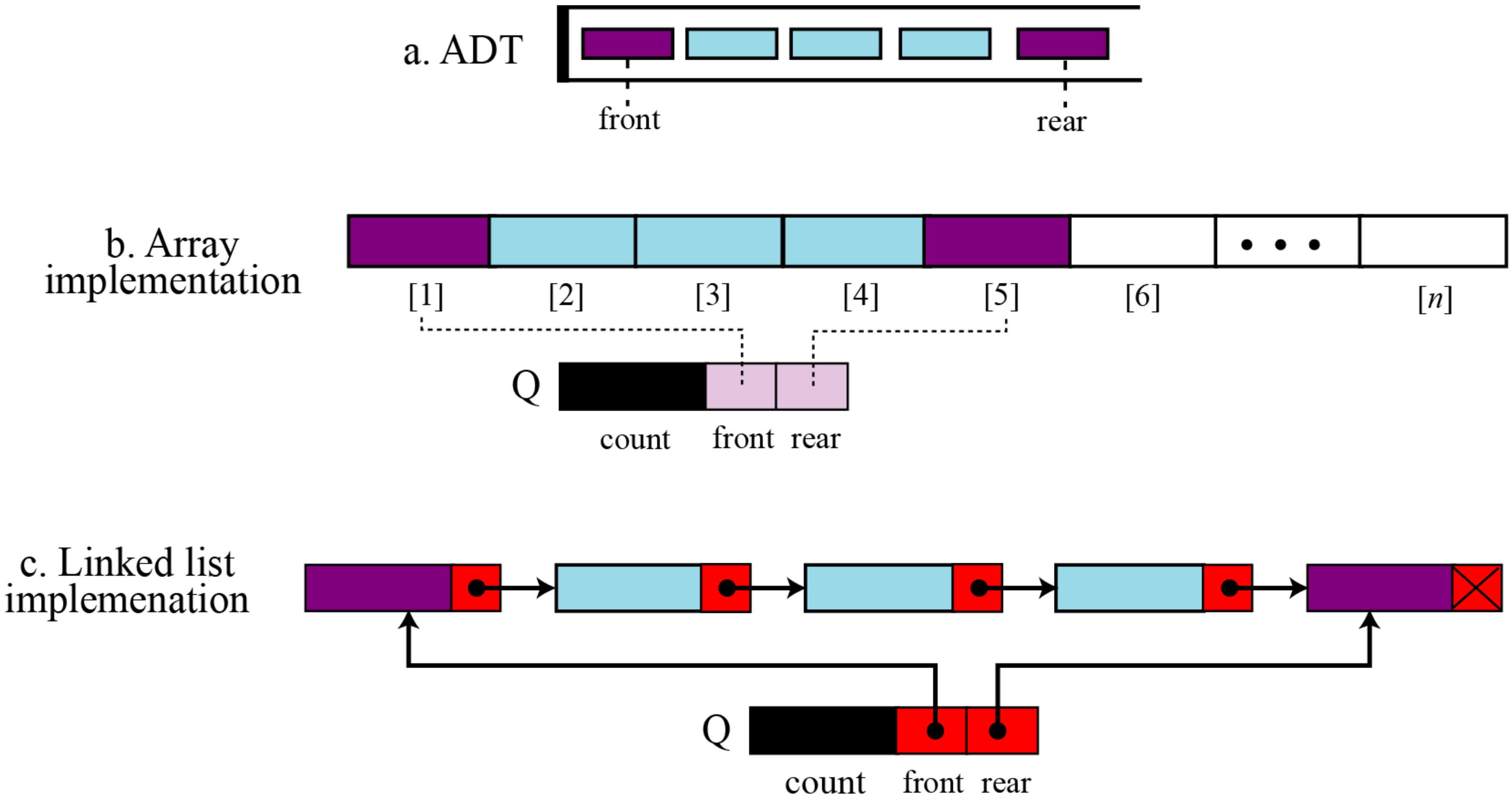
Queues

A queue represents a waiting list. A queue can be viewed as a special type of list, where the elements are inserted into the end (tail) of the queue, and are accessed and deleted from the beginning (head) of the queue.



Queues

- Queue Manipulation Operations
  - `isEmpty()`: returns `true` or `false`
  - `first()`: returns copy of value at front
  - `add(v)`: adds a new value at rear of queue Enqueue
  - `remove()`: removes, returns value at front Dequeue



Queue implementation

Implementing Stacks and Queues

- Using an Arraylist to implement Stack
- Use a Linked list to implement Queue

Since the insertion and deletion operations on a stack are made only at the end of the stack, using an array list to implement a stack is more efficient than a linked list.

Since deletions are made at the beginning of the list, it is more efficient to implement a queue using a linked list than an array list.

Design of the Stack and Queue Classes

There are two ways to design the stack and queue classes:

- Using inheritance: You can declare the stack class by extending the array list class, and the queue class by extending the linked list class.

  – Using composition: You can declare an array list as a data field in the stack class, and a linked list as a data field in the queue class.

Composition is Better

Both designs are fine, but using composition is better because it enables you to declare a complete new stack class and queue class without inheriting the unnecessary and inappropriate methods from the array list and linked list.

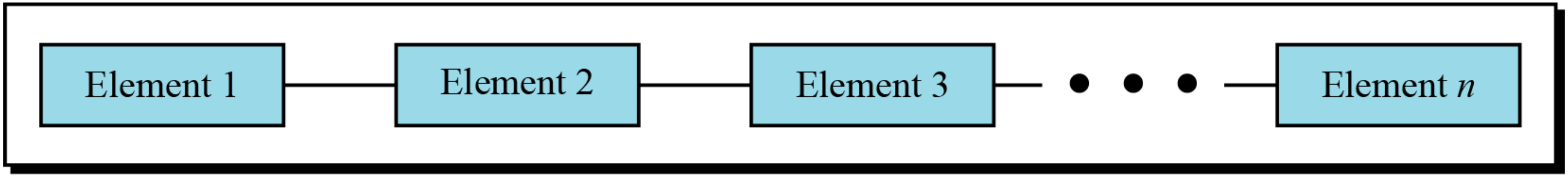
MyStack and MyQueue

MyStack	MyStack
-list: MyArrayList	
+isEmpty(): boolean	Returns true if this stack is empty.
+getSize(): int	Returns the number of elements in this stack.
+peek(): Object	Returns the top element in this stack.
+pop(): Object	Returns and removes the top element in this stack.
+push(o: Object): Object	Adds a new element to the top of this stack.
+search(o: Object): int	Returns the position of the specified element in this stack.

MyQueue	MyQueue
-list: MyLinkedList	
+enqueue(element: Object): void	Adds an element to this queue.
+dequeue(): Object	Removes an element from this queue.
+getSize(): int	Returns the number of elements from this queue.

GENERAL LINEAR LISTS

- Stacks and queues defined in the two previous sections are restricted linear lists.
- A general linear list is a list in which operations, such as insertion and deletion, can be done anywhere in the list—at the beginning, in the middle or at the end. Figure shows a general linear list.



General linear list

Operations on general linear lists

Six common operations: *list*, *insert*, *delete*, *retrieve*, *traverse* and *empty*.

The *list* operation

The list operation creates an empty list. The following shows the format:

list (listName)

List features

- **ORDERING**: maintains order elements were added (new elements are added to the end by default)
- **DUPLICATES**: yes (allowed)
- **OPERATIONS**: add element to end of list, insert element at given index, clear all elements, search for element, get element at given index, remove element at given index, get size
  - some of these operations are inefficient! (seen later)

- list manages its own size; user of the list does not need to worry about overfilling it

Java's `List` interface

- Java also has an interface `java.util.List` to represent a list of objects:
  (a partial list)

`public void add(int index, Object o)`
Inserts the specified element at the specified position in this list.

`public Object get(int index)`
Returns the element at the specified position in this list.

`public int indexOf(Object o)`
Returns the index in this list of the first occurrence of the specified element, or `-1` if the list does not contain it.

`List` interface, cont'd.

`public int lastIndexOf(Object o)`

Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain it.

`public Object remove(int index)`

Removes the object at the specified position in this list.

`public Object set(int index, Object o)`

Replaces the element at the specified position in this list with the specified element.

- Notice that the methods added to `Collection` by `List` all deal with indexes; a list has indexes while a general collection may not

Some list questions

- all of the list operations on the previous slide could be performed using an array instead!

- open question: What are some reasons why we might want to use a list class, rather than an array, to store our data?

- thought question: How might a List be implemented, under the hood?

- why do all the List methods use type `Object`?

List Iterations

A particularly slow idiom

```
// print every element of linked list
for (int i = 0; i < list.size(); i++) {
    Object element = list.get(i);
    System.out.println(i + ": " + 
  element);
}
```

- This code executes an O(n) operation (`get`) every time through a loop that runs n times!
  - Its runtime is O(n^2), which is much worse than O(n)
  - this code will take prohibitively long to run for large data sizes

The problem of position

- The code on the previous slide is wasteful because it throws away the position each time
  - every call to `get` has to re-traverse the list!
- it would be much better if we could somehow keep the list in place at each index as we looped through it

- Java uses special objects to represent a position of a collection as it's being examined...
  - these objects are called "iterators"

List Iterator

- ListIterator type
- Gives access to elements inside a linked list
- Encapsulates a position anywhere inside the linked list
- Protects the linked list while giving access

A List Iterator

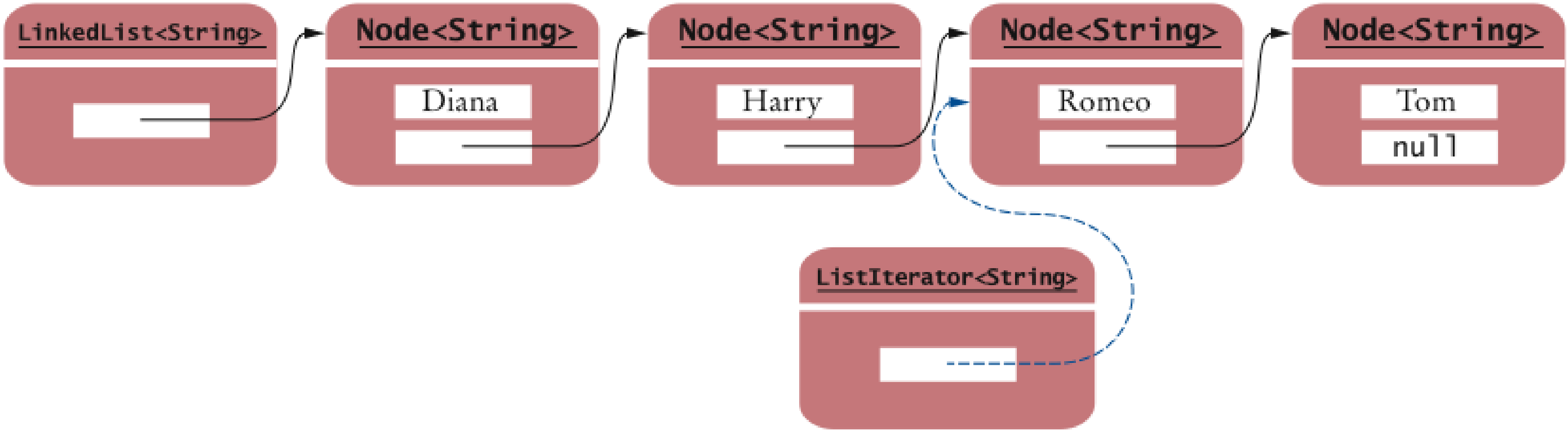


Figure 2 A List Iterator

A Conceptual View of the List Iterator

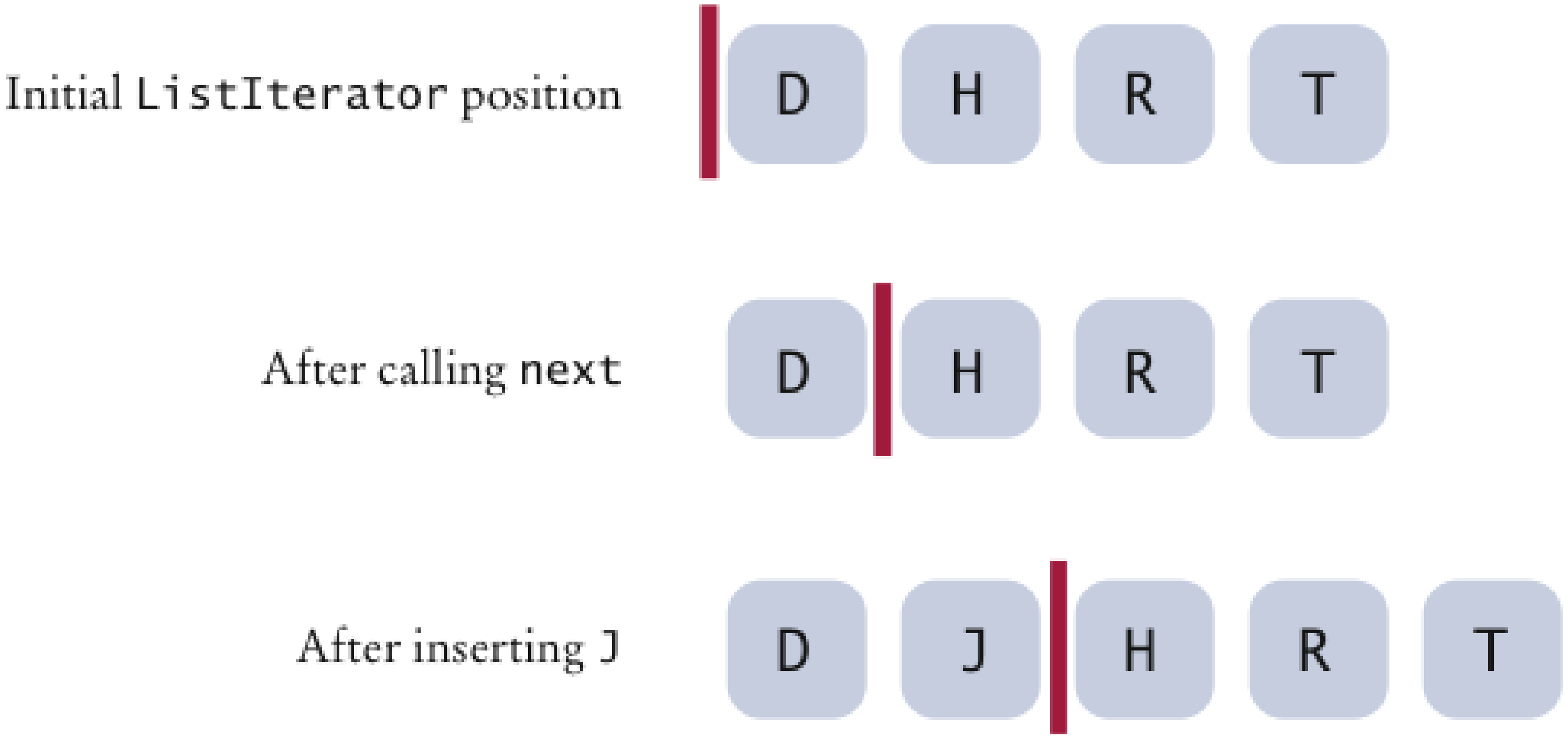


Figure 3 A Conceptual View of the List Iterator

List Iterator

- Think of an iterator as pointing between two elements
  - *Analogy: Like the cursor in a word processor points between two characters*
- The listIterator method of the LinkedList class gets a list iterator

```
LinkedList<String> employeeNames = ...;
ListIterator<String> iterator =
   employeeNames.listIterator();
```

List Iterator

- Initially, the iterator points before the first element
- The next method moves the iterator:

```
iterator.next();
```

- next throws a NoSuchElementException if you are already past the end of the list
- hasNext returns true if there is a next element:

```
if (iterator.hasNext())
   iterator.next();
```

List Iterator

- The next method returns the element that the iterator is passing:

```
while iterator.hasNext()
{
   String name = iterator.next();
   //Do something with name
}
```

List Iterator

- Shorthand for loop:

```
for (String name : employeeNames)
{
  // Do something with name
}
```

Behind the scenes, the for loop uses an iterator to visit all list elements

List Iterator

- LinkedList is a *doubly linked list*
  - *Class stores two links:*
    - *One to the next element, and*
    - *One to the previous element*
- To move the list position backwards, use:
  - *hasPrevious*
  - *previous*

Adding and Removing from a LinkedList

- The add method:
  - *Adds an object after the iterator*
  - *Moves the iterator position past the new element:*

```
iterator.add("Juliet");
```

Adding and Removing from a LinkedList

- The remove method
  - *Removes and*
  - *Returns the object that was returned by the last call to next or previous*

```
//Remove all names that fulfill a certain condition
while (iterator.hasNext())
{
   String name = iterator.next();
   if (name fulfills condition)
      iterator.remove();  }
```

Adding and Removing from a LinkedList

- Be careful when calling remove:
  - *It can be called **only once** after calling next or previous:*

```
iterator.next();
iterator.next();
iterator.remove();
iterator.remove();
// Error: You cannot call remove twice.
```

  - ***You cannot call it immediately after a call to add:***

```
iter.add("Fred");
iter.remove(); // Error: Can only call remove after
               // calling next or previous
```

  - *If you call it improperly, it throws an IllegalStateException*

Methods of the `ListIterator` Interface

Table 2 Methods of the ListIterator Interface

`String s = iter.next();`	Assume that `iter` points to the beginning of the list `[Sally]` before calling `next`. After the call, `s` is `"Sally"` and the iterator points to the end.
`iter.hasNext()`	Returns `false` because the iterator is at the end of the collection.

`if (iter.hasPrevious())`
`{`
`   s = iter.previous();`
`}`
	`hasPrevious` returns `true` because the iterator is not at the beginning of the list.

`iter.add("Diana");`	Adds an element before the iterator position. The list is now `[Diana, Sally]`.
`iter.next();` `iter.remove();`	`remove` removes the last element returned by `next` or `previous`. The list is again `[Diana]`.

Sample Program

- `ListTester` is a sample program that
  - *Inserts strings into a list*
  - *Iterates through the list, adding and removing elements*
  - *Prints the list*

ListTester.java

```
import java.util.LinkedList;
import java.util.ListIterator;

/**
   A program that tests the LinkedList class
*/
public class ListTester
{
   public static void main(String[] args)
   {
      LinkedList<String> staff = new LinkedList<String>();
      staff.addLast("Diana");
      staff.addLast("Harry");
      staff.addLast("Romeo");
      staff.addLast("Tom");

      // | in the comments indicates the iterator position

      ListIterator<String> iterator = staff.listIterator(); // |DHRT
      iterator.next(); // D|HRT
      iterator.next(); // DH|RT

```

Continued

ListTester.java (cont.)

```
      // Add more elements after second element

      iterator.add("Juliet"); // DHJ|RT
      iterator.add("Nina"); // DHJN|RT

      iterator.next(); // DHJNR|T

      // Remove last traversed element

      iterator.remove(); // DHJN|T

      // Print all elements

      for (String name : staff)
         System.out.print(name + " ");
      System.out.println();
      System.out.println("Expected: Diana Harry Juliet Nina Tom");
   }
}
```

Continued

ListTester.java (cont.)

Program Run:

```
Diana Harry Juliet Nina Tom
Expected: Diana Harry Juliet Nina Tom
```

Self Check

Do linked lists take more storage space than arrays of the same size?

> **Answer:** Yes, for two reasons. You need to store the node references, and each node is a separate object. (There is a fixed overhead to store each object in the virtual machine.)

Self Check

Why don’t we need iterators with arrays?

Answer: An integer index can be used to access any array location.