CSCI 136
Data Structures & Advanced Programming

Lecture 11

Spring 2018

Profs Bill & Jon

Administrative Details

- Lab 4
  - A wrong version of `LinkedList.java` was posted on the course website.
  - If you downloaded the file, please delete it. If you have it open on your browser, please refresh your browser. (The honor code applies here.)
  - Your starter repo contains the correct version, so you don't have to do anything if you haven't checked out the file on the website
- Fill out the Google form by 10am

Last Time

- Singly Linked List Implementations
- Doubly Linked List
  - Lab 4: Dummy Nodes

Today

- Assertions
- SLL improvements?
    - Tail Pointer
    - Circularly Linked Lists
  - Doubly Linked Lists – with recursion!
- Search
  - Linear
  - Binary

Assertions

- Pre and post condition comments are useful to us as programmers, but they aren't enforced
- Assertions are a language feature that lets us test assumptions about our code
- Structure 5 has an `Assert` class
- Java language now has an `assert` keyword

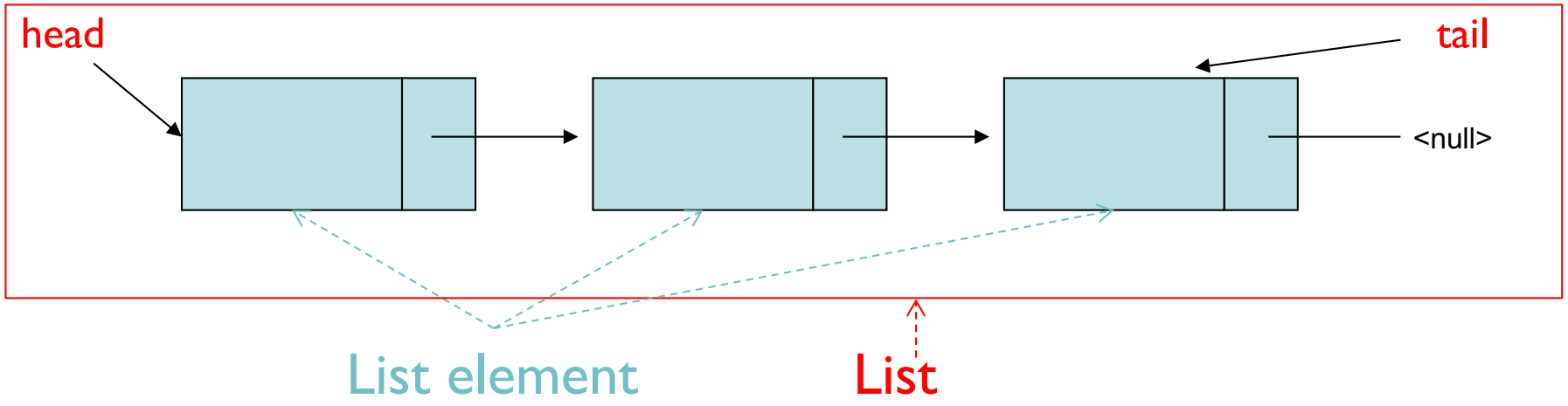
Assertions

- Structure5: static method that "just works"
  - `Assert.pre(<condition>, "Error Msg");`
- Java `assert`: must run code with `-ea` flag
  - `assert <condition>;`
  - `assert <condition> : "Error Msg";`

```
java -ea AssertionTest
```

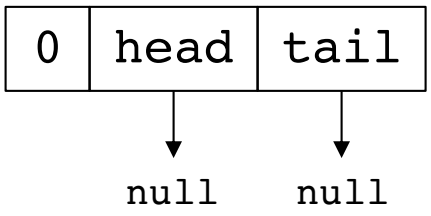
Linked List Improvements: tail

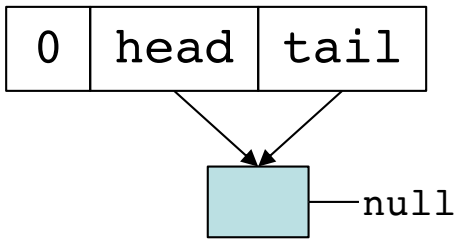
- Which of these `List` methods would be faster if the `SLL` class had a `SLLN tail` member variable?
  - `getLast()`
  - `addLast()`
  - `removeLast()`

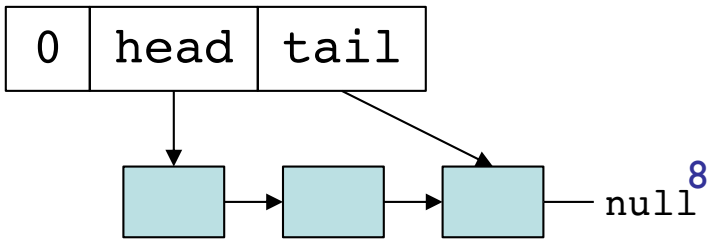


Linked List Improvements: tail

- After adding a `tail` to SLL:
  - `addLast()` and `getLast()` become O(1)
  - `removeLast()` is not improved. Why?
    - We need to know the `SLLN` *before* tail so we can reset tail
- Side effects
  - We now have three cases to consider in method implementations
    - Think about `add(int i, E o)`
      - empty list, `head==tail`, `head!=tail`







Linked List Improvements: CircularlyLinkedLists

- Use `next` reference of last element to reference `head` of list
- Replace `head` reference with `tail` reference
- Access head of list via `tail.next()`
- ALL operations on head are fast!
- `addLast()` is still fast
- Only modest additional complexity in implementation
- Can "cyclically reorder" list by changing `tail` node

- Question: What's a circularly linked list of size 1?

DoublyLinkedLists

- Nodes keep reference/links in **both** directions
- DLL keeps `head` and `tail` references
- `DoublyLinkedListNode` instance variables:
  - `DLLN<E> next;`
    `DLLN<E> prev;`
    `E value;`

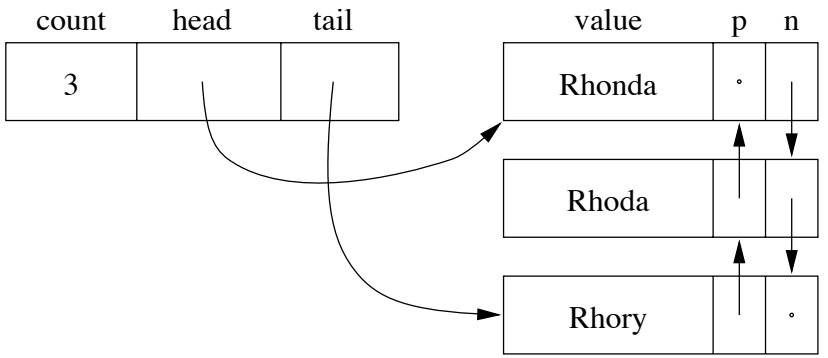


Figure 9.7,
Bailey pg. 202

DoublyLinkedLists

- Space overhead is proportional to number of elements
  - Still O(n) like SLL and Vector
- ALL operations on tail (including removeLast) are fast!
- Additional complexity in each list operation
  - Example: `add(E d, int index)`
    - Four cases to consider now:
      - empty list
      - add to front
      - add to tail
      - add in middle

```
public class DoublyLinkedNode<E> {
        protected E data;
        protected DoublyLinkedNode<E> nextElement;
        protected DoublyLinkedNode<E> previousElement;

// Constructor inserts new node between existing nodes
public DoublyLinkedNode(E v,
                DoublyLinkedNode<E> next,
                DoublyLinkedNode<E> previous)
{
        data = v;
        nextElement = next;
        if (nextElement != null)
                nextElement.previousElement = this;
        previousElement = previous;
        if (previousElement != null)
                previousElement.nextElement = this;
}
```

DoublyLinkedList Add Method

```
public void add(int i, E o) {
       if (i == 0) {
              addFirst(o);
       } else if (i == size()) {
               addLast(o);
       } else {
              // Find items before and after insert point
              DoublyLinkedNode<E> before = null;
              DoublyLinkedNode<E> after = head;
              // search for ith position
              while (i > 0) {
                 before = after;
                 after = after.next();
                 i--;
              }
       // before, after refer to items in slots i-1 and i
       // continued on next slide
```

DoublyLinkedList Add Method

```
            // Note: Still in "else" block!
            // before, after refer to items in slots i-1 and i

            // create new value to insert in correct position.
            // Use DLN constructor that takes parameters
            // to set its next and previous instance variables
            DoublyLinkedNode<E> current =
                  new DoublyLinkedNode<E>(o,after,before);

            count++; // adjust size

            // make after and before value point to new value
            before.setNext(current);
            after.setPrevious(current);
            // Note: These lines aren't needed---why?
      }
}
```

Lab 4: Any Questions?

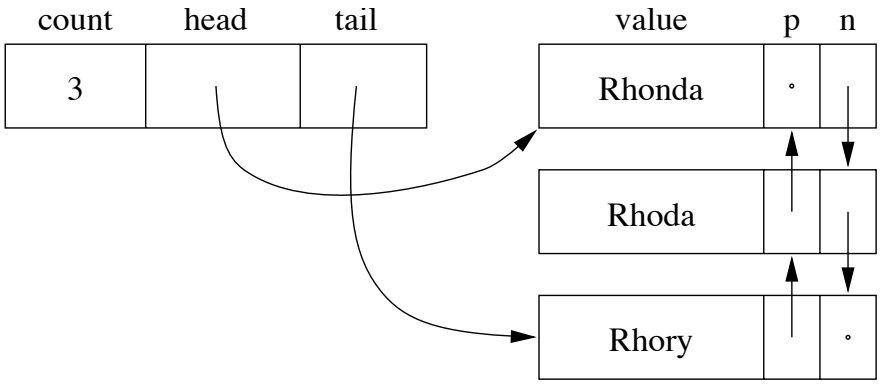
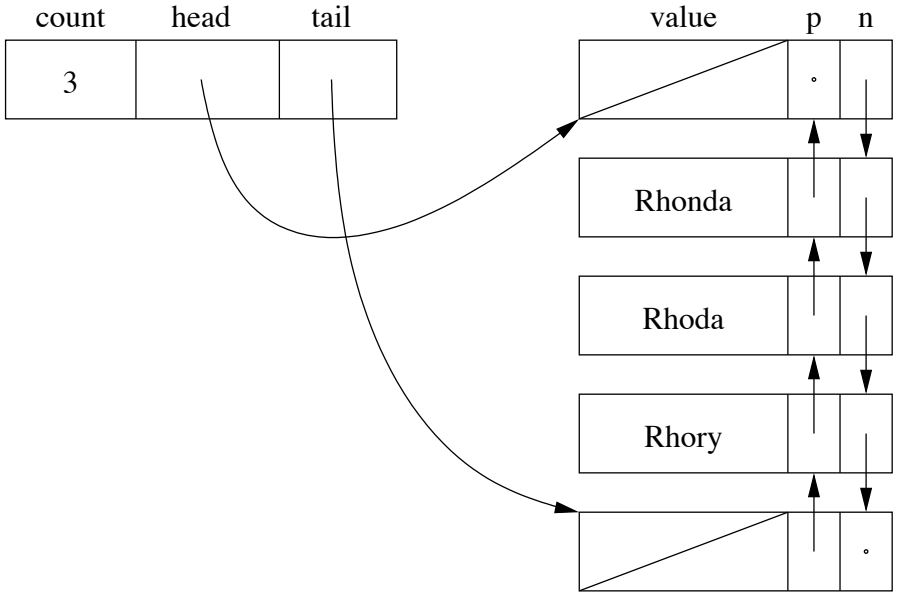


Figure 9.7,
Bailey pg. 202



Bailey pg. 215

Lists and Recursion

- Let's implement DLL's `int indexOf(E value)` recursively

  Questions:

  - What is the base case?
  - How do we call `indexOf(E value)` on a smaller version of the list?
    - Nodes are recursive; List interface hides implementation details

  - Prove by induction that `recIndexOf()` is O(n) recursive calls in the worst case