CSCI 136
Data Structures & Advanced Programming

Lecture 9

Spring 2018

Profs Bill & Jon

Administrative Details

- Lab 1
  - I apologize for not having it returned yet
  - Feedback will show up on GitHub as a “Pull Request”
  - PRs give you the option to view comments line-by-line, and respond to comments
  - New workflow this semester, so it is taking time to get the kinks worked out. It should be faster turnaround than printouts once it is working.

Last Time

- Revisited Vector Growth
  - Additive: $O(n^2)$
  - Multiplicative: $O(n)$
- Recursion
  - Base case
  - Recursive “leap of faith”
- Lab 3
  - Subset Sum
    - Helper method!
    - Big-O?

Today

- Induction
  - An important proof strategy
  - Closely tied to recursion
- List: A general-purpose interface
- Implementing Lists with linked structures
  - Singly Linked Lists
  - Circularly Linked Lists
  - Doubly Linked Lists

Mathematical Induction

- The mathematical cousin of recursion is induction
- Induction is a proof technique
- Reflects the structure of the natural numbers
- Use to simultaneously prove an infinite number of theorems!

Mathematical Induction

- Example: Prove that for every `n` ≥ 0

$$P_n : \sum_{i=0}^{n} i = \boxed{0 + 1 + \ldots + n = \frac{n(n+1)}{2}}$$

- Proof by induction mirrors recursion:
  - Base case:
    - P_n is true for n = 0
  - Inductive hypothesis:
    - If P_n is true for some `n`≥0, then P_{n+1} is true.
      - (Using a smaller version of the problem, we solve a larger version)

Mathematical Induction

$$P_n : \sum_{i=0}^{n} i = 0 + 1 + \ldots + n = \frac{n(n+1)}{2}$$

- Prove the base case: P_n is true for $n = 0$
  - Just check it! Substitute 0 into the equation.

$$0 = 0(1)/2$$

- Assume the inductive hypothesis: P_n is true for some $n \geq 0$
- Then use assumption to show that P_{n+1} is true.

Write out P_{n+1} and target equality

$$P_{n+1}: \underbrace{0 + 1 + \ldots + n}_{\text{This is } P_n!} + (n+1) = \frac{(n+1)\big((n+1)+1\big)}{2} = \frac{(n+1)(n+2)}{2}$$

$$\frac{n(n+1)}{2} + (n+1) = \frac{n(n+1) + 2(n+1)}{2} = \frac{n^2+3n+2}{2} = \frac{(n+1)(n+2)}{2}$$

- First equality holds by assumed truth of P_n!

What about Recursion?

- What does induction have to do with recursion?
  - Same form!
    - Base case
    - Inductive case that uses simpler form of problem
- We can prove things about recursive functions using induction.
- Example: factorial
  - Prove that fact(n) requires n multiplications

```
public static int fact(n) {
        if (n==0) return 1;
        return n * fact(n-1);
}
```

fact(n) requires n multiplications

- Prove that fact(n) requires n multiplications
  - Base case: n = 0 returns 1
    - 0 multiplications
  - Inductive Hypothesis: Assume true for all k<n, so fact(k) requires k multiplications.
  - Prove, from simpler cases, that the *n*th case holds
    - `fact(n)` performs `1` multiplication (`n*fact(n-1)`).
    - We know `fact(n-1)` requires `n-1` multiplications (by our I.H.)
    - `1+n-1 = n`
      - therefore `fact(n)` requires `n` multiplications.

Mathematical Induction

- Prove: $\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$

(Practice at home)

- Prove: $0^3 + 1^3 + \ldots + n^3 = (0 + 1 + \ldots + n)^2$

- Prove: `fib(n)` makes at least `fib(n)` calls to `fib()`

Counting fib() method calls

- Prove that `fib(n)` makes at least `fib(n)` calls to `fib()`
  - Base cases: n = 0: 1 call; n = 1; 1 call
  - Inductive Hypothesis: Assume that for some `n`≥2, fib`(n-1)` makes at least `fib(n-1)` calls to `fib()` and `fib(n-2)` makes at least `fib(n-2)` calls to `fib()`.
  - Claim: Then `fib(n)` makes at least `fib(n)` calls to `fib()`
    - 1 initial call: fib(n)
    - By induction: At least fib(n-1) calls for fib(n-1)
    - And as least fib(n-2) calls for fib(n-2)
    - Total: 1 + fib(n-1) + fib(n-2) > fib(n-1) + fib(n-2) = fib(n) calls
  - Note: Need two base cases!

The List Interface

```
interface List {
   size()
   isEmpty()
   contains(e)
   get(i)
   set(i, e)
   add(i, e)
   remove(i)
   addFirst(e)
   getLast()
   .
   .
   .
}
```

- It’s an interface…therefore it provides no implementation
- Can be used to describe many different types of lists
- `Vector` implements `List`
- Other implementations are possible…

Pros and Cons of Vectors

Pros

- Good general purpose list
- Dynamically Resizeable
- Fast access to elements
  - `vec.get(387425)` finds item 387425 in the same number of operations regardless of `vec`'s size

Cons

- Slow updates to front of list (why?)
- Hard to predict time for add (depends on internal array size)
- Potentially wasted space

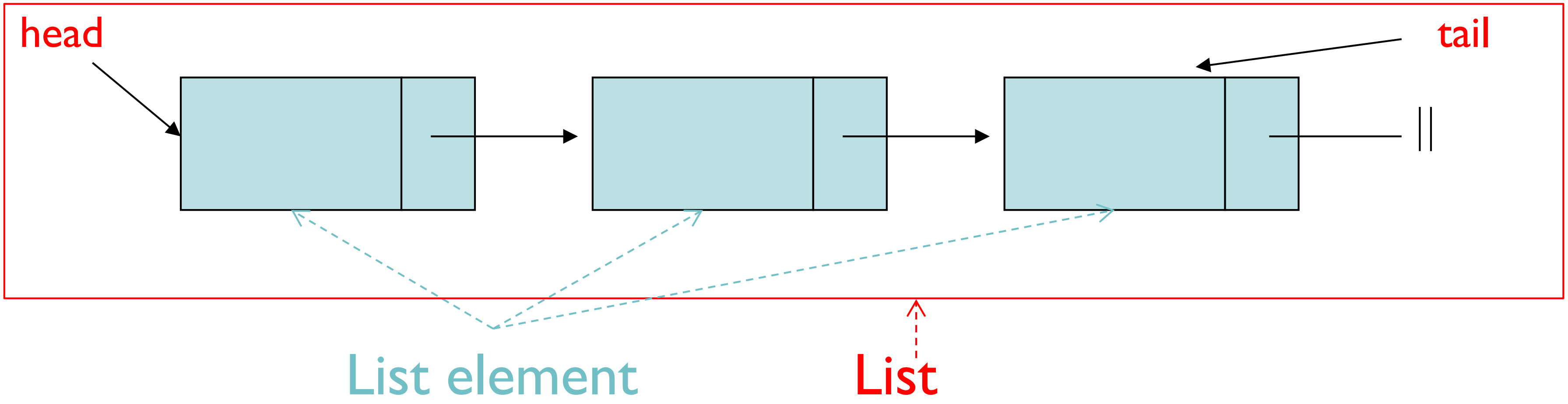
What if we didn't have to copy the array each time we grew `vec`?

List Implementations

- General concept for storing/organizing data
- `Vector` implements the `List` interface
- We'll now explore other `List` implementations
  - `SinglyLinkedList`
  - `CircularlyLinkedList`
  - `DoublyLinkedList`

Linked List Basics

- There are two key aspects of Lists
  - Elements of the list
    - Store data, point to the "next" element
  - The list itself
    - Includes `head` (sometimes `tail`) member variable
- Visualizing lists

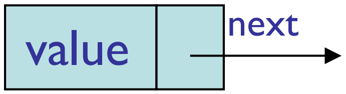


Linked List Basics

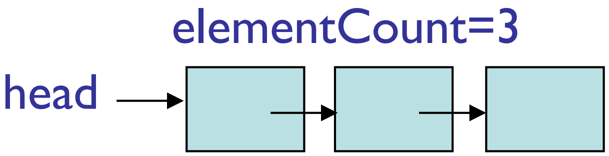
- List nodes are recursive data structures
- Each "node" has:
  - A data value
  - A next variable that identifies the next element in the list
  - Can also have "previous" that identifies the previous element ("doubly-linked" lists)
- What methods does the `Node` class need?

SinglyLinkedLists

- How would we implement `SinglyLinkedListNode`?
  - `SinglyLinkedListNode` = SLLN in my notes
  - SLLN = `Node` in the book (in Ch 9)



- How about `SinglyLinkedList`?
  - `SinglyLinkedList` = SLL in my notes



- What would the following look like?
  - `addFirst(E d)`
  - `getFirst()`?
  - `addLast(E d)`?   (more interesting)
  - `getLast()`?