Course Recap

Announcements

- Final is on Saturday, May 6
  - 7-10pm in Science Center 101
  - Study guide is posted
- Review sessions next week
  - Wed, 1pm in Sci Cen 181 (me)
  - Thu, 2:30pm in Sci Cen 183 (Prof. Newhall)
  - Fri, TBD (Jeff)

Today’s Plan

- Review Linked Lists
- Talk about final exam
- Big ideas
- Course evaluations

Linked list recap

- Each item in a linked list corresponds to a node
- A node has two parts, corresponding to the two instance variables for our `Node` class:
  - `self.item` / `self.data`: the value in the list
  - `self.next`: the next `Node` in the list or `None` to signify the end of a list

Linked list recap

- If we have access to the first node of a linked list, we can access all the other nodes by following the `self.next` links

- To make implementing certain methods faster and more convenient, our `LinkedList` class also keeps track of the last node and the number of nodes:

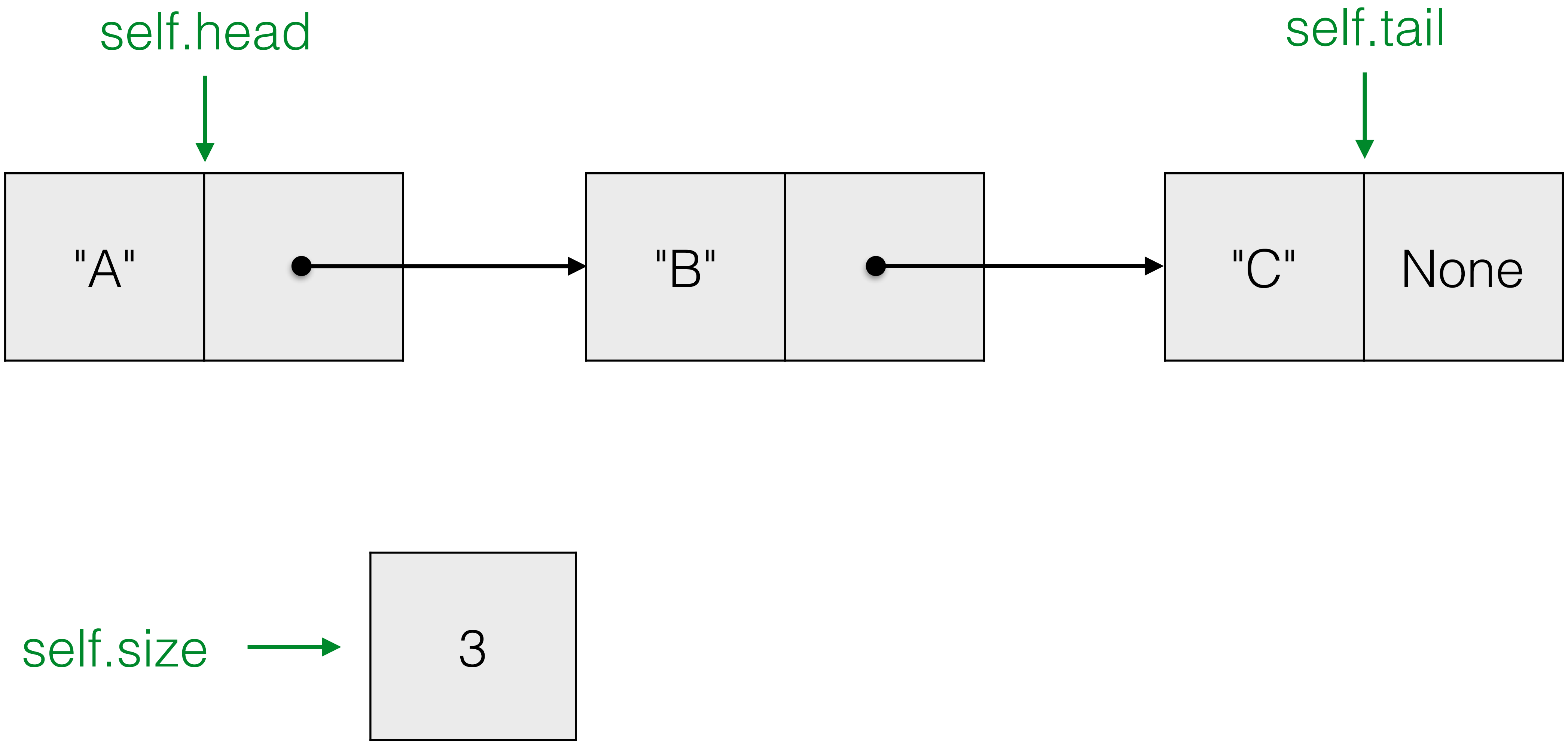
  - `self.head`: the first `Node` in the list

  - `self.tail`: the last `Node` in the list

  - `self.size`: the number of `Node`s/values in the list (an int)

Linked list structure

- We may ask you to draw the structure of a linked list on the final:



Linked list methods

- Be able to recognize and understand code that implements various methods for the `LinkedList` class.

- Be able to implement methods for the `LinkedList` class.

- This includes inserting and removing, especially at the head or tail and methods that do a traversal.

    - Traversal relies on an accumulator variable, sometimes named `current`

- How to call linked list constructor, call methods

Linked list run times

	Python list	Linked list
Insert at beginning	O(n)	**O(1)**
Insert at end	O(n)*	**O(1)**
Remove from beginning	O(n)	**O(1)**
Remove from end	**O(1)**	O(n)
Get item at index	**O(1)**	O(n)

Final exam

- 3 hours, closed book

- What’s on it: linked lists, classes, recursion, searching, sorting, functions, loops, conditionals, types, operators, expressions, string formatting, getting and validating input, and more…

- What’s not on it: graphics, file i/o, vim, unix

- Good to know: top-down design

Be able to:

- Compute expressions and identify their types
- Write a complete, multi-function program
- Trace a multi-function program, showing output and drawing the stack
- Write a class and/or methods for a class; write code that uses a class
- Write recursive functions; draw stack diagram for a recursive function

Be able to:

- Identify the run-time of an algorithm:
    - *$O(1)$*, *$O(\log n)$*, *$O(n)$*, *$O(n \log n)$*, or *$O(n^2)$*
- Show steps in searching and sorting algorithms, know run times of searching and sorting algorithms.
- Write and understand methods for a linked list, write code that uses a linked list, identify run time of linked list methods, draw linked list structure.
- Find and fix bugs in code

Big ideas of the course

- Binary representation of data
    - With n bits we can represent 2^n things
    - e.g. 8 bits or 1 byte to represent a number between 0 and 255 -> 3 bytes for an RGB pixel
    - We can reuse the same bits to store any other type: bool, int, float, string, list, object, etc.

Big ideas

- The four parts of a program:
  - Getting user input: `raw_input`, reading files, keypresses, mouse clicks, and more…
  - Computation: everything from simple arithmetic to complex algorithms, moving data around, using data structures
  - Producing output: `print`, writing files, graphics, animation, and more…
  - Repetition: main loop of program

Big ideas

- Top-down design
  - Write main() first, delegating tasks to functions that don’t yet exist
  - It’s good to start with a plan, understanding that you may have to change the plan (in CS and in life)
- Incremental development
  - Write each function one at a time, testing and debugging as you go
- Testing and debugging
  - Assume you’ll make some mistakes on the first attempt; have strategies to find and correct these mistakes.
  - Programming requires humility.

Big ideas

- Abstraction and interfaces:
    - Computers are incredibly complex—if we had to understand the entire machine and all its software to get anything done, nothing would ever get done.
    - Interfaces abstract away some of this complexity, allowing us to harness the power of the computer without needing to understand every detail.
        - e.g. functions, classes, unix shell
    - We can layer abstractions on top of each other: use an existing interface in creating a new one.
    - This facilitates collaboration.

After CS 21

- More CS courses:
  - CS 31: how your computer works, executes a program, programming mostly done in C
  - CS 35: follow-up to CS 21, learn to implement and analyze more data structures, object-oriented programming in C++
  - Upper-level courses: graphics, artificial intelligence, machine learning, natural language processing, theory of computation, programming languages, software engineering, operating systems, and more…
- Learn more about Python/programming on your own
- Write a program of your own design.

Use coding for good

- Programming is a powerful skill.
- Like any powerful skill it can be used for good or for evil.
- Use it for good. If you put programs out into the world, think about the impact they will have.

Thank you!

- Thanks to Zoe, Nhung, and Rye.
- Thanks to you for all your hard work.
- Thanks for filling out the course evaluation (it’s very helpful for us)

Enjoy summer break!