Data Structures
Logistics

• Midterm next Monday
  – Half of the people will be in Wu & Chen
  – The other half in DRLB A1
  – Will post past midterms tonight (expect similar types of questions)

• Review on Wednesday
  – Come with questions

• HW3 graded
  – HW4 will be graded by the end of the week
  – HW5 due Thursday next week (last homework on Python)
Why do we need data structures?

• Mostly, to store data in different ways
  – Depending on the application, may emphasize different functionality

• We’ve already seen some built-in Python structures
  – Lists make it easy to access elements by using their index
  – Dictionaries make it easy to map keys to values
  – Sets are best for storing elements without duplicates
  – Tuples store related things in one place

• Many other data structures exist in practice
  – Any CS major includes (at least one) an entire course on this topic
  – Will cover just some basic concepts today (also practice with classes/references)
Linked list

• One of the most widely used data structures
  – Ordered – similar to a standard Python list
  – Makes it very easy to add/remove elements in arbitrary positions

• Each element in the list is called a Node
  – The Node contains the actual data (e.g., city coordinates)
  – Also a pointer to the next element

• Many variants of linked lists exist
  – Today we’ll cover stacks and queues
Linked list implementation

• Contains a **head** and a **tail** node
  – First and last items in the list, respectively
  – Depending on the convention, these nodes may or may not contain actual data

```
12  ➔  99  ➔  37  ➔  null
```

• Each node contains a reference to the next node
• IDLE example: linkedList.py
Adding/Removing Elements

- Depending on application, three options
  1. Add to end, remove from the beginning
     - Queue
  2. Add to/remove from the beginning
     - Stack
  3. Add to/remove from other positions
     - Non-standard but possible
     - Might be slow
Queue

• Elements are added at the end and removed from the beginning
  – Also known as a first-in-first-out (FIFO) data structure

• Very fast to add/remove elements

• A possible application is a website server
  – Requests are handled in order of arrival

• IDLE example: queue.py
Stack

• Elements are added at the beginning and removed from the beginning
  – Also known as a **last-in-first-out (LIFO)** data structure
• Also very fast to add/remove
• A possible application is an “undo” implementation
  – Any change is “**pushed**” (i.e., added) to the stack
  – If you press “undo”, the latest change is “**popped**” (i.e., removed)
• IDLE example: stack.py
Doubly-linked lists

• Each node may point not only to next but also to previous
  – Implementation very similar to standard linked list
  – Each node will two pointers instead of one

• More powerful than singly-linked list
  – Can implement anything implemented by a singly-linked list
  – Requires more memory for extra pointers

• A possible application would be “undo” + “redo”
  – Upon “undo”, don’t remove the latest item but just move the pointer to the previous item
  – If “redo” later, can go back to “undo”-ed element
Pros and cons of linked lists

• Pros
  – Add/remove at beginning/end is very fast
  – Can grow arbitrarily (no need to reset length as with a standard list)
  – No need to set an initial length of the list

• Cons
  – Add/remove in the middle is slow (have to traverse the list to the item in question)
  – Searching is slow (may have to traverse the whole list)
  – Indexing is slow (have to traverse the list to the item in question)
Tree

• Another way of storing data
  – If we don’t want to store things chronologically (as in a list), we can store them based on some relation between the data
  – This will make it easier to search for elements

• Trees also have nodes
  – Instead of next/previous, a node has children
  – Each Node contains that data, plus references to all its children

• Many types of trees
  – Today we’ll cover binary trees
Tree implementation

- The top Node is known as a **root** node
  - The root may or may not contain data depending on the implementation
- The children of the root are roots of the **branches** of the tree

- IDLE example: tree.py
Binary Tree

- A **binary tree** is a tree where each node has at most two children
  - A **balanced binary tree** has the same number of nodes in each branch

- Binary trees are very useful structures
  - An obvious application is searching
    - Suppose all items in the left branch are smaller and all items in the right branch are bigger – then searching would be very fast
  - Implementation somewhat involved – need to restructure the tree when inserting new items
Pros and cons of binary trees

• Pros
  – Searching is fast (we always know which branch to follow)
  – Can grow arbitrarily big

• Cons
  – Add/delete is longer than with lists (need to restructure the tree)
    • Not super slow!
  – Depending on the sequence of inputs, the tree might not be naturally balanced (will have to restructure it to make it balanced)
  – If tree is not balanced, searching will be slower (in the worst case, each node has only one child, and the tree will look like a list)