CS 240
Fall 2014

Mike Lam, Professor

Maps and Binary Search Trees

ADTs

- List
- Set
- Stack
- Queue
  - Deque
  - Priority Queue
- Tree
  - Binary Tree

New ADT: Map

- Map: key → value
- Unique keys, non-unique values
- Other names:
  - Dictionary (Python)
  - Associative array
- Many applications
  - Database: student ID# → student info
  - DNS: domain name → IP address
  - OS: process ID → process
  - Namespace: variable → value

Map ADT

`M[k]`	retrieve value for key
`M[k] = v`	modify value for key
`del M[k]`	remove key from map
`len(M)`	return # of keys
`iter(M)`	generate sequence of keys
`k in M`	return True if key has a mapping
`M.clear()`	remove all mappings
`M.keys()`	return a set of all keys
`M.values()`	return a set of all values
`M1 == M2`	return True if the maps have identical associations

Recall PA1

- Search engine DB: maps words → websites
- `index()` method
  - Crawl multiple websites for keywords
  - Add word → URL mapping for each keyword-site pair
  - This can be relatively slow
- `search()` method
  - Perform DB lookup
  - Returns websites associated with the given word
  - This needs to be FAST!

Map Implementations

- Store (key, value) tuples
- Variety of internal structures possible
- Important operations:
  - Insert
  - Lookup
  - Modify

Map Implementations

- Unsorted list
  - Insert: ***O(1)*** Lookup: ***O(n)*** Modify: ***O(n)***
- Sorted list
  - Insert: ***O(n)*** Lookup: ***O(log n)*** Modify: ***O(log n)***
- Skip list
  - Insert: ***O(log n)*** Lookup: ***O(log n)*** Modify: ***O(log n)***

Hashing

- Sneak peak: "hashing" is a particular kind of mapping: keys → buckets
  - Can be used to implement maps
    - Keep a bunch of buckets for data
    - Store and lookup items by their key using the hash mapping
  - If implemented properly, this can be VERY fast!
  - In fact, most operations are *O(1)* average time
  - This is what Python dictionaries use
- We will cover hashing in the last couple of weeks
  - But since we're already talking about trees...

Sorted Map ADT

- Nearly identical to regular Map ADT
  - Keys are sorted (not necessarily values!)
- Addition of ordering methods
  - `M.find_min()` and `M.find_max()`
  - `M.find_lt(k)`, `M.find_le(k)`, `M.find_gt(k)`, `M.find_ge(k)`
  - `M.find_range(start, stop)`
  - `iter(M)`
  - `reversed(M)`

Sorted Map Implementation

- Hashing does not work very well for sorted maps
- No inherent correlation between bucket ordering and key ordering
- In fact, the best hashing mechanisms distribute the keys in a uniformly random fashion across all buckets
- We will need another solution

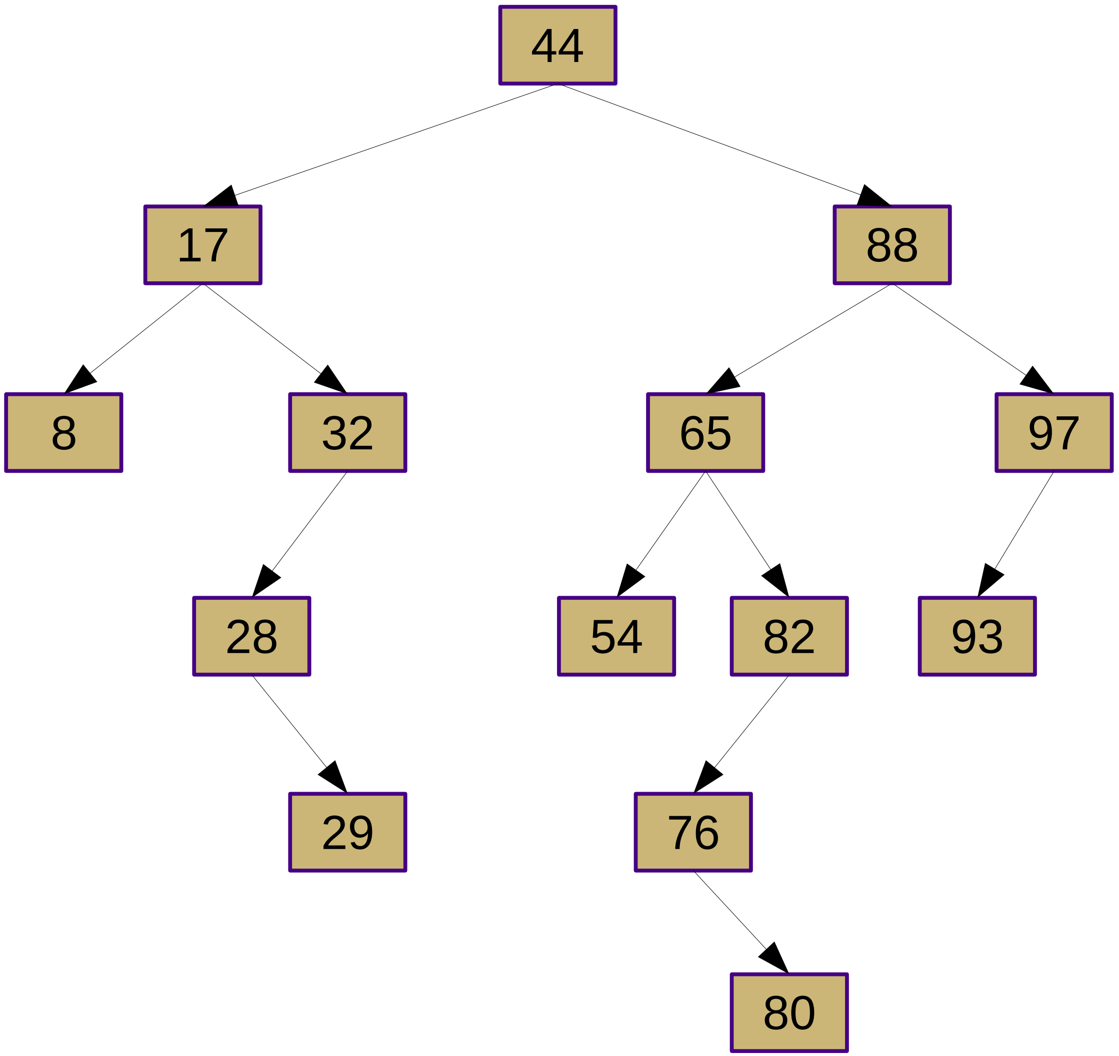
Sorted Map Implementation

- Intuition: use binary trees to bound the number of keys we have to examine during lookups
- Two goals:
  - We want to bound the tree to roughly *O(log n)* levels
    - This implies restrictions on the structure of the tree
    - Heaps accomplish this by restricting the tree to be complete
  - We also want a stronger ordering than the heap-order property
    - This implies restrictions on the content of the tree
    - It also makes it far more difficult to maintain completeness

Binary Search Tree

- Binary Search Tree (BST)
  - Each tree node stores a key-value pair *(k,v)* and two child node references
  - All keys in the left subtree are less than *k*
  - All keys in the right subtree are greater than *k*
  - Often we will ignore the values
    - They are irrelevant to BST implementation details

Binary Search Tree



Binary Search Tree

- Iterate over all keys
  - Inorder recursive tree traversal
    1. Recurse on left subtree
    2. Visit current key
    3. Recurse on right subtree
- Finding min/max key
  - Follow left/right child references exclusively
- Finding predecessor/successor keys
  - Requires more complex traversal
  - Could be a descendant or an ancestor

Binary Search Tree

- Searching for a particular key
  - `def search(k)`
    - `if k == self.key:`
      - `return self`
    - `elif k < self.key and self.left is not None:`
      - `return left.search(k)`
    - `elif k > self.key and self.right is not None:`
      - `return right.search(k)`
    - `else:`
      - `return None        # not found`

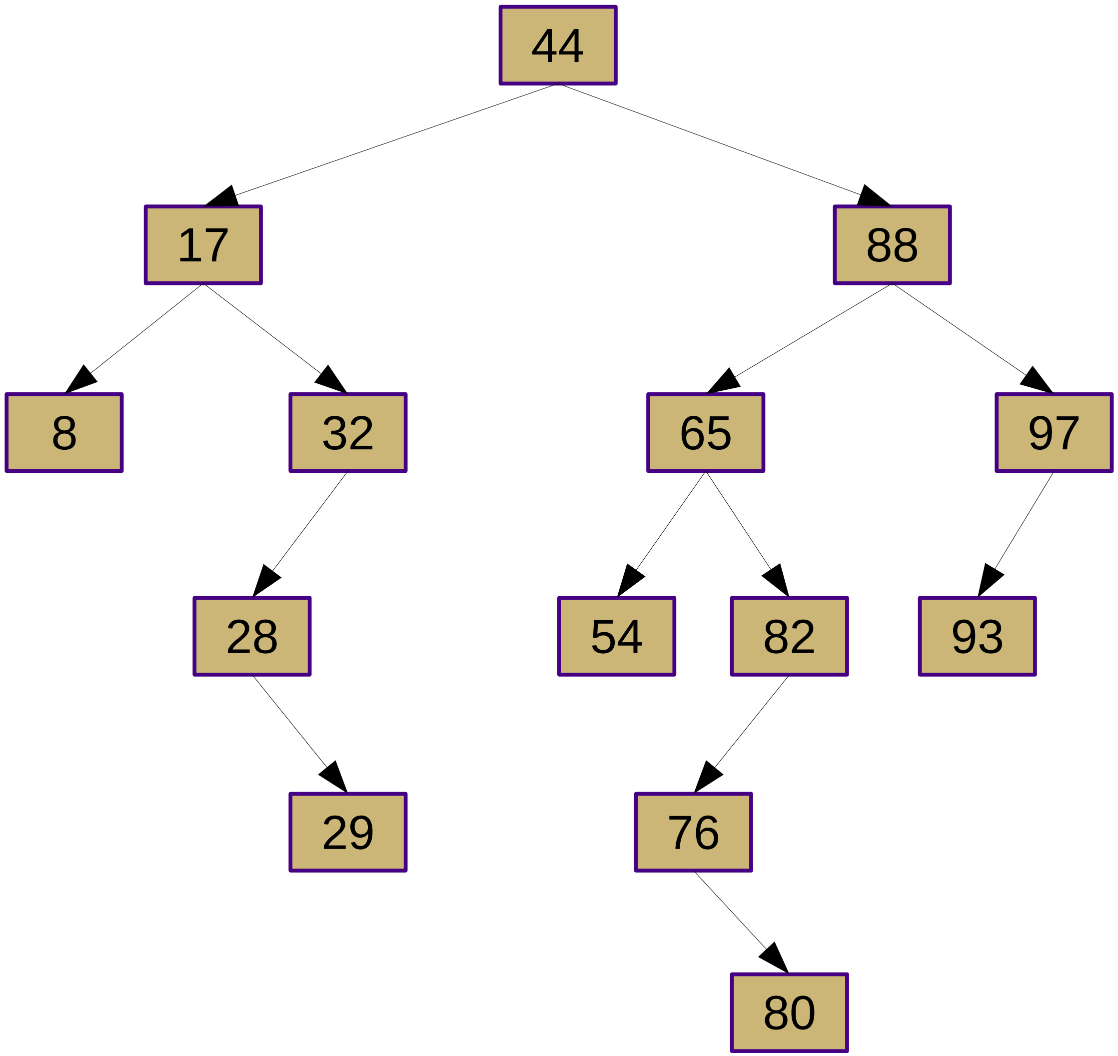
Binary Search Tree

- Insertion of new key
  - `def insert(k, v):`
    - `if k == self.key:`
      - `self.value = v`
    - `elif k < self.key:`
      - `if self.left is None:`
        - `self.left = _Node(key, value)`
      - `else:`
        - `self.left.insert(k, v)`
    - `else: # if k > self.key:`
      - `if self.right is None:`
        - `self.right = _Node(key, value)`
      - `else:`
        - `self.right.insert(k, v)`

Binary Search Tree

- Deletion of a key
  - Find the correct node (p)
  - If p has no children:
    - Remove p
  - If p has one child (c):
    - Replace p with c
  - If p has two children:
    - Find the predecessor (r) of p
      - Since p has two children, the predecessor will be in the left subtree
      - Predecessor's key is greater than any key in the left subtree and less than any key in the right subtree
      - Thus, r will NOT have a right child
    - Replace p's key with r's key
    - Remove r and replace with its left child (if it had one)

Binary Search Tree



Binary Search Tree

- What is the minimum key?
- What is the maximum key?
- What is the predecessor of 88?
- What is the predecessor of 82?
- What is the predecessor of 76?
- What is the predecessor of 29?
- What is the predecessor of 44?
- What is the successor of 17?
- What is the successor of 29?

Binary Search Tree

- Where should new key 5 go?
- Where should new key 68 go?
- Where should new key 100 go?
- What will the tree look like after removing 29?
- What will the tree look like after removing 28?
- What will the tree look like after removing 82?
- What will the tree look like after removing 88?
- What will the tree look like after removing 65?
- What will the tree look like after removing 44?

BST Analysis

- Most worst-case running times are *O(h)*
  - Where *h* is the height of the binary tree
  - This makes restraining the tree's height very important to being efficient
  - For mostly-random insertions and deletions, $h \approx \log n$
  - For other situations, we need to use a more "balanced" binary tree implementation
- Running time of find_range is *O(s+h)*
  - Where s is the number of items returned
- Running time of iterators is *O(n)*
  - Has to visit every key