CSE 332 Data Abstractions:

Dictionary ADT: Arrays, Lists and Trees

Kate Deibel
Summer 2012

Where We Are

Studying the absolutely essential ADTs of computer science and classic data structures for implementing them

ADTs so far:
- Stack: push, pop, isEmpty, …
- Queue: enqueue, dequeue, isEmpty, …
- Priority queue: insert, deleteMin, …

Next:
- Dictionary/Map: key-value pairs
- Set: just keys
- Grabbag: random selection

Dictionary sometimes goes by Map. It's easier to spell.

MEET THE DICTIONARY AND SET ADTS

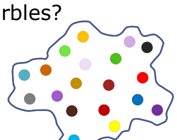
Dictionary and Set ADTs

The ADTs we have already discussed are mainly defined around actions:
- Stack: LIFO ordering
- Queue: FIFO ordering
- Priority Queue: ordering by priority

The Dictionary and Set ADTs are the same except they focus on data storage/retrieval:
- insert information into structure
- find information in structure
- remove information from structure

A Key Idea

If you put marbles into a sack of marbles, how do you get back your *original* marbles?



You only can do that if all marbles are somehow unique.

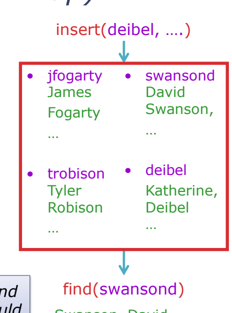
The Dictionary and Set ADTs insist that everything put inside of them must be unique (i.e., no duplicates).

This is achieved through keys.

The Dictionary (a.k.a. Map) ADT

Data:
- Set of (key, value) pairs
- keys are mapped to values
- keys must be comparable
- keys must be unique

Standard Operations:
- insert(key, value)
- find(key)
- delete(key)



Like with Priority Queues, we will tend to emphasize the keys, but you should not forget about the stored values

The Set ADT

Data:
- keys must be comparable
- keys must be unique

Standard Operations:
- insert(key)
- find(key)
- delete(key)

Comparing Set and Dictionary

Set and Dictionary are essentially the same
- Set has no values and only keys
- Dictionary's values are "just along for the ride"
- The same data structure ideas thus work for both dictionaries and sets
- We will thus focus on implementing dictionaries

But this may not hold if your Set ADT has other important mathematical set operations
- Examples: union, intersection, isSubset, etc.
- These are binary operators on sets
- There are better data structures for these

A Modest Few Uses

Any time you want to store information according to some key and then be able to retrieve it efficiently, a dictionary helps:

- Networks: router tables
- Operating systems: page tables
- Compilers: symbol tables
- Databases: dictionaries with other nice properties
- Search: inverted indexes, phone directories, …
- And many more

But wait…

No duplicate keys? Isn't this limiting? Duplicate data occurs all the time!?

Yes, but dictionaries can handle this:
- Complete duplicates are rare. Use a different field(s) for a better key
- Generate unique keys for each entry (this is how hashtables work)
- Depends on why you want duplicates

Example: Dictionary for Counting

One example where duplicates occur is calculating frequency of occurrences

To count the occurrences of words in a story:
- Each dictionary entry is keyed by the word
- The related value is the count
- When entering words into dictionary
  - Check if word is already there
  - If no, enter it with a value of 1
  - If yes, increment its value

Calling Noah Webster…
or at least a Civil War veteran in a British sanatorium…

IMPLEMENTING THE DICTIONARY

Some Simple Implementations

Arrays and linked lists are viable options, just not great particular good ones.

For a dictionary with n key/value pairs, the worst-case performances are:

	Insert	Find	Delete
Unsorted Array	O(1)	O(n)	O(n)
Unsorted Linked List	O(1)	O(n)	O(n)
Sorted Array	O(n)	O(log n)	O(n) ← *Again, the array shifting is costly*
Sorted Linked List	O(n)	O(n)	O(n)

Lazy Deletion in Sorted Arrays

10	12	24	30	41	42	44	45	50
✓	✗	✓	✓	✓	✓	✗	✓	✓

Instead of actually removing an item from the sorted array, just mark it as deleted using an extra array

Advantages:
- Delete is now as fast as find: O(log n)
- Can do removals later in batches
- If re-added soon thereafter, just unmark the deletion

Disadvantages:
- Extra space for the "is-it-deleted" flag
- Data structure full of deleted nodes wastes space
- find O(log m) time (m is data-structure size)
- May complicate other operations

Better Dictionary Data Structures

The next several lectures will dicuss implementing dictionaries with several different data structures

AVL trees
- Binary search trees with guaranteed balancing

Splay Trees
- BSTs that move recently accessed nodes to the root

B-Trees
- Another balanced tree but different and shallower

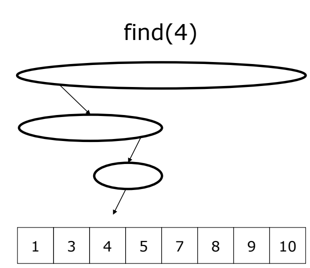
Hashtables
- Not tree-like at all

See a Pattern?

Why Trees?

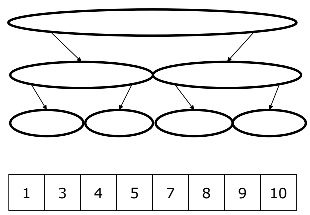
Trees offer speed ups because of their branching factors
- Binary Search Trees are structured forms of *binary search*

Binary Search



Binary Search Tree

Our goal is the performance of binary search in a tree representation



Why Trees?

Trees offer speed ups because of their branching factors

- Binary Search Trees are structured forms of *binary search*

Even a basic BST is fairly good

	Insert	Find	Delete
Worse-Case	O(n)	O(n)	O(n)
Average-Case	O(log n)	O(log n)	O(log n)

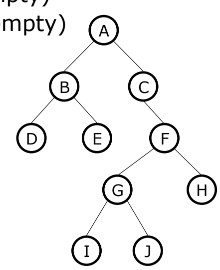
Cats like to climb trees… my Susie prefers boxes…

BINARY SEARCH TREES: A REVIEW

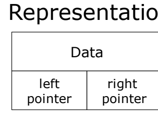
Binary Trees

A non-empty binary tree consists of a

- a root (with data)
- a left subtree (may be empty)
- a right subtree (may be empty)



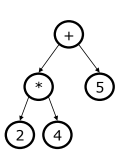
Representation:



- For a dictionary, data will include a key and a value

Tree Traversals

A traversal is a recursively defined order for visiting all the nodes of a binary tree



Pre-Order: root, left subtree, right subtree

\+ * 2 4 5

In-Order: left subtree, root, right subtree

2 * 4 + 5

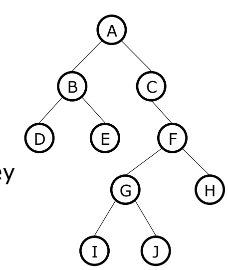
Post-Order: left subtree, right subtree, root

2 4 * 5 +

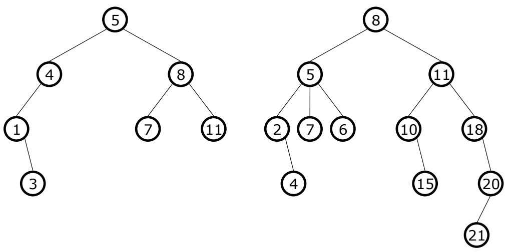
Binary Search Trees

BSTs are binary trees with the following added criteria:

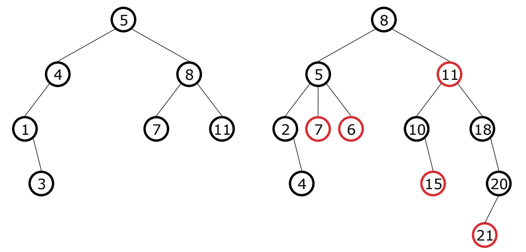
- Each node has a key for comparing nodes
- Keys in left subtree are smaller than node's key
- Keys in right subtree are larger than node's key



Are these BSTs?



Are these BSTs?



Calculating Height

What is the height of a BST with root r?

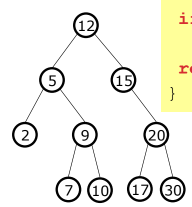
```
int treeHeight(Node root) {
  if(root == null)
    return -1;

  return 1 + max(treeHeight(root.left),
                 treeHeight(root.right));
}
```

Running time for tree with *n* nodes:
  O(*n*) – single pass over tree

How would you do this without recursion?
Stack of pending nodes, or use two queues

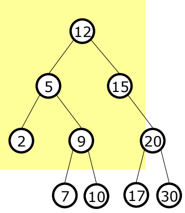
Find in BST, Recursive

```
Data find(Key key, Node root){
 if(root == null)
   return null;
 if(key < root.key)
   return find(key, root.left);
 if(key > root.key)
   return find(key, root.right);
 return root.data;
}
```



Find in BST, Iterative

```
Data find(Key key, Node root){
 while(root != null && root.key != key) {
  if(key < root.key)
    root = root.left;
  else(key > root.key)
    root = root.right;
 }
 if(root == null)
    return null;
 return root.data;
}
```



Performance of Find

We have already said it is worst-case O(n)

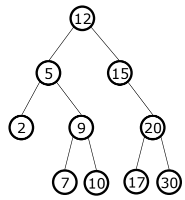
Average case is O(log n)

But if want to be exact, the time to find node x is actually Θ(depth of x in tree)

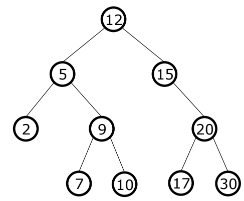
- If we can bound the depth of nodes, we automatically bound the time for find()

Other "Finding" Operations

- Find minimum node
- Find maximum node
- Find predecessor of a non-leaf
- Find successor of a non-leaf
- Find predecessor of a leaf
- Find successor of a leaf

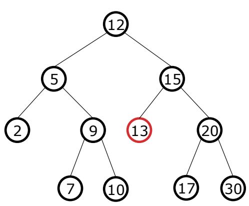


Insert in BST



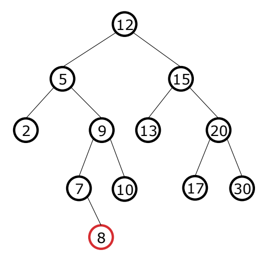

```
insert(13)
insert(8)
insert(31)
```


Insert in BST



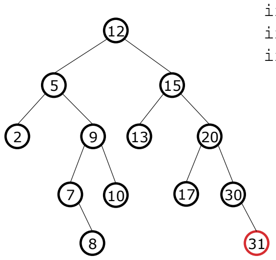

```
insert(13)
insert(8)
insert(31)
```


Insert in BST



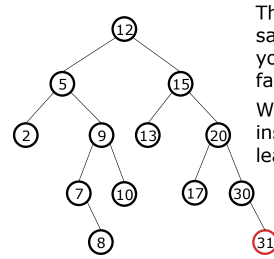

```
insert(13)
insert(8)
insert(31)
```


Insert in BST




```
insert(13)
insert(8)
insert(31)
```

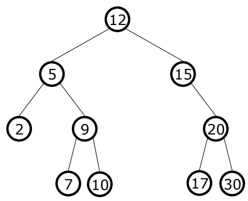

Insert in BST



The code for insert is the same as with find except you add a node when you fail to find it.

What makes it easy is that inserts only happen at the leaves.

Deletion in BST



Why might deletion be harder than insertion?

Deletion

Removing an item disrupts the tree structure

Basic idea:

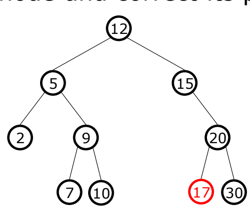
- find the node to be removed,
- Remove it
- Fix the tree so that it is still a BST

Three cases:

- node has no children (leaf)
- node has one child
- node has two children

Deletion – The Leaf Case

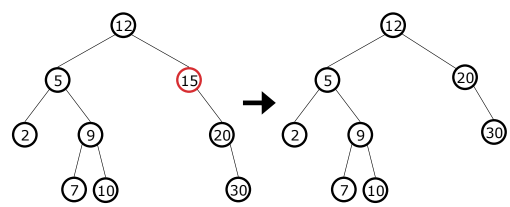
This is by far the easiest case… you just cut off the node and correct its parent



`delete(17)`

Deletion – The One Child Case

If there is only one child, we just pull up the child to take its parents place

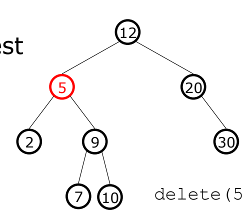


`delete(15)`

Deletion – The Two Child Case

Deleting a node with two children is the most difficult case. We need to replace the deleted node with another node.

What node is the best to replace 5 with?



`delete(5)`

Deletion – The Two Child Case

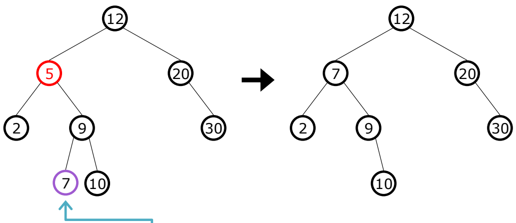
Idea: Replace the deleted node with a value guaranteed to be between the node's two child subtrees

Options are

- successor from right subtree: findMin(node.right)
- predecessor from left subtree: findMax(node.left)
- These are the easy cases of predecessor/successor

Either option is fine as both are guaranteed to exist in this case

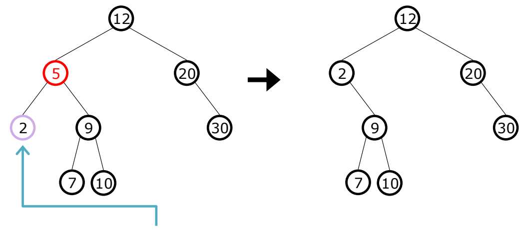
Delete Using Successor



findMin(right sub tree) → 7

```
delete(5)
```


Delete Using Predecessor



findMax(left sub tree) → 2

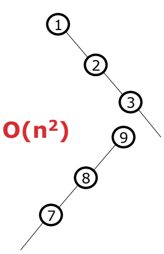
```
delete(5)
```


BuildTree for BST

We had buildHeap, so let's consider buildTree

Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty tree

- If inserted in given order, what is the tree?
- What big-O runtime for this kind of sorted input? **$O(n^2)$**
- Is inserting in the reverse order any better?



BuildTree for BST (take 2)

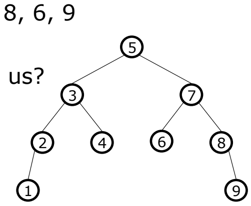
What if we rearrange the keys?

- median first, then left median, right median, etc. → 5, 3, 7, 2, 1, 4, 8, 6, 9

What tree does that give us?

What big-O runtime?

O(n log n)



Give up on BuildTree

The median trick will guarantee a O(n log n) build time, but it is not worth the effort.

Why?

- Subsequent inserts and deletes will eventually transform the carefully balanced tree into the dreaded list
- Then everything will have the O(n) performance of a linked list

Achieving a Balanced BST (part 1)

For a BST with n nodes inserted in arbitrary order

- Average height is O(log n) – see text
- Worst case height is O(n)
- Simple cases, such as pre-sorted, lead to worst-case scenario
- Inserts and removes can and will destroy the balance

Achieving a Balanced BST (part 2)

Shallower trees give better performance

- This happens when the tree's height is O(log n) ← like a perfect or complete tree

Solution: Require a **Balance Condition** that

1. ensures depth is always $O(\log n)$
2. is easy to maintain

Doing so will take some careful data structure implementation… Monday's topic

Time to put your learning into practice…

DATA STRUCTURE SCENARIOS

About Scenarios

We will try to use lecture time to get some experience in manipulating data structures

- We will do these in small groups then share them with the class
- We will shake up the groups from time to time to get different experiences

For any data structure scenario problem:

- Make any assumptions you need to
- There are no “right” answers for any of these questions

GrabBag

A GrabBag is used use for choosing a random element from a collection. GrabBags are useful for simulating random draws without repetition, like drawing cards from a deck or numbers in a bingo game.

GrabBag Operations:

- Insert(item e): e is inserted into the grabbag
- Grab(): if not empty, return a random element
- Size(): return how many items are in the grabbag
- List(): return a list of all items in the grabbag

In groups:

- Describe how you would implement a GrabBag.
- Discuss the time complexities of each of the operations.
- How complex are calls to random number generators?

Improving Linked Lists

For reasons beyond your control, you have to work with a very large linked list. You will be doing many finds, inserts, and deletes. Although you cannot stop using a linked list, you are allowed to modify the linked structure to improve performance.

What can you do?