CSE 12 – Basic Data Structures

Prof. Christine Alvarado
[Slides borrowed/adapted from slides by Cynthia Lee, Rakesh Vama, & Roshni Chandrashekhar]
Announcements

1. HW8
   1. Released next Wednesday
   2. Due Tuesday week 10 (2 week assignment)
   3. Worth 400 points.
   4. It will be LONG. If you wait to start you will not succeed and you will significantly drop your grade in the class!

2. HW8 is PARTNER ENCOURAGED
   1. You must sign up via the Google form no later than next FRIDAY at 5pm
   2. Once you sign up you are “married” to your partner. Divorces must be done in person by me or Dr. P
Today’s topics

- BST and their implementations
- BST running times
BST Contains: Let’s write it!

// Return true if toFind is in the BST rooted at root, // false otherwise
boolean contains( BSTNode root, E toFind ) {
    if ( root == null ) return false;
    if ( _________________ )
        return contains( _________________, toFind );
    else if ( _________________ )
        return contains( _________________, toFind );
    else
        return _________________;
}

Recursive step and base case 2: How do you know which way to go? Fill in the blanks above. Hint: use compareTo.
boolean contains( BSTNode root, E toFind ) {
    if ( root == null ) return false;
    if ( toFind.compareTo( root.getElement() ) < 0 )
        return ______1_______________________;
    else if (toFind.compareTo( root.getElement() ) > 0 )
        return ______2_______________________;
    else
        return ______3_______________________;
}

What goes in blank 1?
A. false
B. true
C. contains( root.getLeftChild(), toFind )
D. contains( root.getRightChild(), toFind )
E. contains( root, toFind )
// Return true if toFind is in the BST rooted at root, // false otherwise
boolean contains( BSTNode root, E toFind ) {
    if ( root == null ) return false;
    if ( toFind.compareTo( root.getElement() ) < 0 )
        return contains( root.getLeftChild(), toFind );
    else if ( toFind.compareTo( root.getElement() ) > 0 )
        return _______2______________________;
    else
        return _______3______________________;

What goes in blank 2?
A. false
B. true
C. contains( root.getLeftChild(), toFind )
D. contains( root.getRightChild(), toFind )
E. contains( root, toFind )
// Return true if toFind is in the BST rooted at root, // false otherwise
boolean contains( BSTNode root, E toFind ) {
    if ( root == null ) return false;
    if ( toFind.compareTo( root.getElement() ) < 0 )
        return contains( root.getLeftChild(), toFind );
    else if (toFind.compareTo( root.getElement() ) > 0 )
        return contains( root.getRightChild(), toFind );
    else
        return ________3______________________;
}

What goes in blank 3?
A. false
B. true
C. contains( root.getLeftChild(), toFind )
D. contains( root.getRightChild(), toFind )
E. contains( root, toFind )
BST Contains: Let’s write it!

// Return true if toFind is in the BST rooted at root, 
// false otherwise

boolean contains( BSTNode root, E toFind ) {
    if ( root == null ) return false;
    if ( toFind.compareTo( root.getElement() ) < 0 )
        return contains( root.getLeftChild(), toFind );
    else if (toFind.compareTo( root.getElement() ) > 0 )
        return contains( root.getRightChild(), toFind );
    else
        return false;
}
contains() – a recursive method

contains( node, target ) =

- false, if node is null  \textit{base case – failure}
- true, if node's element is equal to the target  \textit{base case – success}
- contains( node's left child, target ), if target < node's element  \textit{recursive case – left}
- contains( node's right child, target ), if target > node's element  \textit{recursive case – right}

where:
- node – the root of the subtree to search (initially the root of the tree)
- target – the element for which we are searching

Call and return paths for successful and unsuccessful contains() calls
void add( E toAdd ) {
    if ( this.root == null )
        this.root = new BSTNode( toAdd );
    else
        addToExistingTree( root, toAdd );
}

void addToExistingTree( BSTNode root, E toAdd )
{
    ...
}
BST Add: Slightly different from your reading

```java
void addToExistingTree( BSTNode root, E toAdd )
{
    int value = toAdd.compareTo( root.getElement() );
    if ( value < 0 ) {
        if ( value.getLeftChild() == null ) {
            BSTNode n = new BSTNode( toAdd );
            root.setLeftChild( n );
            n.setParent( root );
        } else {
            addToExistingTree( root.getLeftChild(), toAdd );
        }
    } else if ( value > 0 ) {
        // Repeat for other side
    }
}
```
You should also be familiar with BST delete. We might ask you questions about it or ask you to write some part of it on the exam (not the whole thing from scratch, though). See your reading.
What is the BEST CASE cost for doing find() in BST?

A. O(1)
B. O(log n)
C. O(n)
D. O(n log n)
E. O(n^2)
What is the worst case cost for doing `find()` in BST?

A. O(1)
B. O(log n)
C. O(n)
D. O(n log n)
E. O(n^2)
What is the WORST CASE cost for doing find() in BST if the BST is full?

A. O(1)
B. O(log n)
C. O(n)
D. O(n log n)
E. O(n^2)
What is the WORST CASE cost for doing find() in BST if the BST is full?

$O(\log n)$

It takes $\Theta(h)$ operations to insert into a BST, where $h$ is the tree’s height.

But how do we relate $h$ to $n$?
Relating $h$ (height) and $n$ (#nodes)

How many nodes are on level $L$ in a completely filled binary search tree?

A. 2
B. $L$
C. $2L$
D. $2^L$
Relating $h$ (height) and $n$ (#nodes)

How many nodes are in a completely filled BST?

A. $n = \sum_{L=0}^{h-1} 2^L$
B. $n = 2^L$
C. $n = \sum_{h=0}^{n} 2^h$
D. $n = 2^{h+L} - 1$
Relating $h$ (height) and $n$ (#nodes)

Remember, we’re trying to find $h = f(n)$, so we need to solve for $h$... but how?

$$n = \sum_{L=0}^{h-1} 2^L$$

$$2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{h-1} = 1 + 2 + 4 + 8 + \cdots + 2^{h-1}$$
Relating $h$ (height) and $n$ (#nodes)

Remember, we’re trying to find $h = f(n)$, so we need to solve for $h$... but how?

\[ n = \sum_{L=0}^{h-1} 2^L = 2^h - 1 \]

\[ \log(n + 1) = \log(2^h) = h \]
The #1 issue to remember with BSTs is that they are great when balanced (O(log n) operations), and horrible when unbalanced (O(n) operations).

Balance depends on order of insert of elements.

Over the years, people have devised many ways of making sure BSTs stay balanced no matter what order the elements are inserted.

We won’t talk about these ways here, but stay tuned for CSE 100…
Is it bigger than a breadbox?

- no
  - Do you eat it with eggs?
    - no: a mouse
    - yes: Spam

- yes
  - Is it worth a lot of money?
    - no: a bag of trash
    - yes: Does it know Java?
      - no: a house
      - yes: a computer scientist
Let's say I want to add “an airplane” and the question “Does it fly” to the tree as the left child of Does it know Java? Draw the new tree. Which nodes do you need references to?