Data Structures and Algorithms
Binary Search Trees

Chris Brooks

Department of Computer Science
University of San Francisco
To find the minimal element in a BST:

- **Base Case:** When is it easy to find the smallest element in a BST?
- **Recursive Case:** How can we make the problem smaller? How can we use the solution to the smaller problem to solve the original problem?
To find the minimal element in a BST:

Base Case:

- When is it easy to find the smallest element in a BST?
To find the minimal element in a BST:
Base Case:

- When is it easy to find the smallest element in a BST?
  - When the left subtree is empty, then the element stored at the root is the smallest element in the tree.
To find the minimal element in a BST:

Recursive Case:

- How can we make the problem smaller?
To find the minimal element in a BST:

**Recursive Case:**

1. How can we make the problem smaller?
   - Both the left and right subtrees are smaller versions of the same problem

2. How can we use the solution to a smaller problem to solve the original problem?
To find the minimal element in a BST:

Recursive Case:

- How can we make the problem smaller?
  - Both the left and right subtrees are smaller versions of the same problem

- How can we use the solution to a smaller problem to solve the original problem?
  - The smallest element in the left subtree is the smallest element in the tree
Object minimum(Node tree) {
    if (tree == null)
        return null;
    if (tree.left() == null)
        return tree.element();
    else
        return minimum(tree.left());
}
Iterative Version

Object minimum(Node tree) {
    if (tree == null)
        return null;
    while (tree.left() != null)
        tree = tree.left();
    return tree.element();
}
What is the base case – an easy tree to insert an element into?
What is the base case – an easy tree to insert an element into?

- An empty tree
- Create a new tree, containing the element $e$
Recursive Case: How do we make the problem smaller?
Recursive Case: How do we make the problem smaller?

△ The left and right subtrees are smaller versions of the same problem.
△ How do we use these smaller versions of the problem?
Recursive Case: How do we make the problem smaller?

- The left and right subtrees are smaller versions of the same problem.
- Insert the element into the left subtree if $e < \text{value stored at the root}$, and insert the element into the right subtree if $e > \text{value stored at the root}$. 

8-12: Inserting $e$ into BST
8-13: Inserting $e$ into BST $T$

6 Base case – $T$ is empty:
   △ Create a new tree, containing the element $e$

6 Recursive Case:
   △ If $e$ is less than the element at the root of $T$, insert $e$
     into left subtree
   △ If $e$ is greater than the element at the root of $T$, insert $e$
     into the right subtree
Tree manipulation functions return trees.

Insert method takes as input the old tree and the element to insert, and returns the new tree, with the element inserted.

- Old value (pre-insertion) of tree will be destroyed.

To insert an element $e$ into a tree $T$:

- $T = \text{insert}(T, e)$;
Node insert(Node tree, Object elem) {
    if (tree == null) {
        return new Node(elem);
    }
    if (elem < tree.element()) {
        tree.setLeft(insert(tree.left(), elem));
        return tree;
    } else {
        tree.setRight(insert(tree.right(), elem));
        return tree;
    }
}
One wrinkle to think about - we’re assuming that we can compare two elements.

Depending upon the type of data being stored, we may need to write a compare function.

More generic approach - write a Comparable interface, and make sure our elements conform to this interface.

In languages with overloading, such as C++, Python or Lisp, we could overload <, >, and =.
Removing a leaf:
  △ Remove element immediately

Removing a node with one child:
  △ Just like removing from a linked list
  △ Make parent point to child

Removing a node with two children:
  △ Replace node with largest element in left subtree, or
    the smallest element in the right subtree
6 Key: we need to find the right element to replace the deleted element.
6 We can find the minimal element in the right subtree
   △ Why will this work?
6 Or the maximal element in the left subtree
   △ Why will this work?
8-19: Balanced vs Unbalanced trees

How long does it take to find the minimal element?
- In terms of the depth of the tree?
- In terms of the number of nodes in the tree?
  - If the tree is balanced?
  - If the tree is unbalanced?

The order in which nodes are inserted will affect whether the tree is balanced.

Later in the semester, we’ll look at how to rebalance a tree efficiently.
8-20: Implementing a BST

A tree is just a Node with 0 or more children.

```java
class Node {
    private Node left;
    private Node right;
    private Object element;

    Node() { }
    Node(Object elem) {
        element = elem;
    }
    Node(Object elem, Node l, Node r) {
        element = elem;
        left = l;
        right = r;
    }
}
```
Node left() {
    return left;
}

void setLeft(Node l) {
    left = l;
}

Node right() {
    return right;
}

void setRight(Node r) {
    right = r;
}

Object element() {
    return element;
}

void setElement(Object e) {
    element = e;
}
Returns the height of the tree

\((\text{Length of the path to the deepest leaf}) + 1\)

Height = 5

Height = 6
int height(Node tree) {
    if (tree == null)
        return 0;
    return 1 + MAX(height(tree.left()),
                    height(tree.right()));
}
8-24: Tree Operations – NumNodes

6. Returns the number of nodes in a tree

Number of Nodes = 8
Number of Nodes = 6
int numNodes(Node tree) {
    if (tree == null)
        return 0;
    return 1 + numNodes(tree.left(),
                        tree.right());
8-26: Tree Operations – NumLeaves

Returns the number of leaves in a tree

Number of Leaves = 4
Number of Leaves = 1
int numLeaves(Node tree) {
    if (tree == null)
        return 0;
    if ((tree.left() == null) &&
        (tree.right() == null))
        return 1;
    return numLeaves(tree.left()) +
        numLeaves(tree.right());
}
How to write code to compute the value of this expression?
int value(Node tree) {
    if (tree.left() == null && tree.right() == null)
        return ((Integer) tree.element()).intValue();
    int left = value(tree.left());
    int right = value(tree.right());
    char op = ((Character) tree.element()).charValue();
    switch (op) {
        case '+':
            return left + right;
        case '*':
            return left * right;
        ...
    }
}
8-30: BST Implementation Details

6 Use BSTs to implement Ordered List ADT

6 Operations
   ▲ Insert
   ▲ Find
   ▲ Remove
   ▲ Print in Order

6 The specification (interface) should not specify an implementation
   ▲ Allow several different implementations of the same interface
6 BST functions require the root of the tree be sent in as a parameter

6 Ordered list functions should *not* contain implementation details!

6 What should we do?
BST functions require the root of the tree be sent in as a parameter.

Ordered list functions should *not* contain implementation details!

What should we do?
- Private variable, holds root of the tree
- Private recursive methods, require root as an argument
- Public methods call private methods, passing in private root