Data Structures and Algorithms
Ordered Lists and Binary Trees

Chris Brooks

Department of Computer Science
University of San Francisco
6-0: Ordered Lists

6 We will often want to have a list in which all the elements are ordered.

6 As we add or remove elements, order is maintained.
   △ List of names in alphabetical order
   △ List of files from smallest to largest
   △ List of records from earliest to most recent
Operations:

- Insert an element in the list
- Check if an element is in the list
- Remove an element from the list
- Print out the contents of the list, in order

How to implement this? Array or linked list?
Using an Ordered Array – Running times:

Check
Insert
Remove
Print
6-3: Implementing Ordered Lists

Using an Ordered Array – Running times:

- Check: \( \Theta(\lg n) \)
- Insert: \( \Theta(n) \)
- Remove: \( \Theta(n) \)
- Print: \( \Theta(n) \)
6-4: Implementing Ordered List

Using an Unordered Array — Running times:

Check
Insert
Remove
Print
Using an *Unordered* Array – Running times:

- Check: $\Theta(n)$
- Insert: $\Theta(1)$
- Remove: $\Theta(n)$
- Print: $\Theta(n \log n)$

(Given a fast sorting algorithm)

Keeping the array ordered makes insertion require $\Theta(n)$ time.
Implementing Ordered List

Using an Ordered Linked List – Running times:

Check
Insert
Remove
Print
### 6-7: Implementing Ordered List

Using an Ordered Linked List – Running times:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Insert</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Remove</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Print</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

With an unordered list, insertion requires $\Theta(1)$ time.
6-8: The Best of Both Worlds

- Linked Lists – Insert fast / Find slow
- Arrays – Find fast / Insert slow
- The only way to examine nth element in a linked list is to traverse (n-1) other elements

4 -> 8 -> 12 -> 15 -> 22 -> 25 -> 28

- If we could leap to the middle of the list ...
6-9: The Best of Both Worlds
Move the initial pointer to the middle of the list:

We’ve cut our search time in half! Have we changed the $\Theta()$ running time?
6-11: The Best of Both Worlds

Move the initial pointer to the middle of the list:

We’ve cut our search time in half! Have we changed the $\Theta()$ running time?

Repeat the process!
6-12: The Best of Both Worlds
6-13: The Best of Both Worlds

Grab the first element of the list:

4 8 12 15 22 25 28

Give it a good shake -
Binary Trees are Recursive Data Structures

- Base Case: Empty Tree
- Recursive Case: Node, consisting of:
  - Left Child (Tree)
  - Right Child (Tree)
  - Data
The following are all Binary Trees (Though not Binary Search Trees)
6-16: Tree Terminology

- Parent / Child
- Leaf node
- Root node
- Edge (between nodes)
- Path
- Ancestor / Descendant
- Depth of a node \( n \)
  - Length of path from root to \( n \)
- Height of a tree
  - (Depth of deepest node) + 1
6-17: Full Binary Trees

- Each node has 0 or 2 children
- Full Binary Trees

- Not Full Binary Trees
6-18: Complete Binary Trees

- Can be build by starting at the root, and filling the tree by levels from left to right
- Complete Binary Trees

Not Complete Binary Trees
6-19: Binary Search Trees

6 Binary Trees

6 For each node n, (value stored at node n) > (value stored in left subtree)

6 For each node n, (value stored at node n) < (value stored in right subtree)
6-20: Example Binary Search Trees

A
  B
    D
  D

D
  B
    C
  A

D
  B
    F
      A
      C
      G
Binary Search Trees are recursive data structures, so most operations on them will be recursive as well.

Recall how to write a recursive algorithm ...
6-23: Writing a Recursive Algorithm

1. Determine a small version of the problem, which can be solved immediately. This is the base case.
2. Determine how to make the problem smaller.
3. Once the problem has been made smaller, we can assume that the function that we are writing will work correctly on the smaller problem (Recursive Leap of Faith).
   △ Determine how to use the solution to the smaller problem to solve the larger problem.
6-24: Finding an Element in a BST

First, the Base Case – when is it easy to determine if an element is stored in a Binary Search Tree?
First, the Base Case – when is it easy to determine if an element is stored in a Binary Search Tree?

- If the tree is empty, then the element can’t be there
- If the element is stored at the root, then the element is there
6-26: Finding an Element in a BST

6. Next, the Recursive Case – how do we make the problem smaller?
6-27: Finding an Element in a BST

Next, the Recursive Case – how do we make the problem smaller?

- Both the left and right subtrees are smaller versions of the problem. Which one do we use?
Next, the Recursive Case – how do we make the problem smaller?

Both the left and right subtrees are smaller versions of the problem. Which one do we use?

If the element we are trying to find is $< \text{ the element stored at the root}$, use the left subtree. Otherwise, use the right subtree.
Next, the Recursive Case – how do we make the problem smaller?

△ Both the left and right subtrees are smaller versions of the problem. Which one do we use?

△ If the element we are trying to find is \(<\) the element stored at the root, use the left subtree. Otherwise, use the right subtree.

How do we use the solution to the subproblem to solve the original problem?
Next, the Recursive Case – how do we make the problem smaller?

- Both the left and right subtrees are smaller versions of the problem. Which one do we use?
- If the element we are trying to find is \(<\) the element stored at the root, use the left subtree. Otherwise, use the right subtree.

How do we use the solution to the subproblem to solve the original problem?

- The solution to the subproblem \(\text{is the solution to the original problem (this is not always the case in recursive algorithms)}\)
6-31: Finding an Element in a BST

To find an element $e$ in a Binary Search Tree $T$:

1. If $T$ is empty, then $e$ is not in $T$
2. If the root of $T$ contains $e$, then $e$ is in $T$
3. If $e <$ the element stored in the root of $T$:
   - Look for $e$ in the left subtree of $T$
   - Otherwise
     - Look for $e$ in the right subtree of $T$
6-32: Finding an Element in a BST

```java
boolean find(Node tree, Comparable elem) {
    if (tree == null)
        return false;
    if (elem.compareTo(tree.element()) == 0)
        return true;
    if (elem.compareTo(tree) < 0)
        return find(tree.left(), elem);
    else
        return find(tree.right(), elem);
}
```
To print out all elements in a BST:

- Print all elements in the left subtree, in order
- Print out the element at the root of the tree
- Print all elements in the right subtree, in order
To print out all element in a BST:

6. Print all elements in the left subtree, in order
6. Print out the element at the root of the tree
6. Print all elements in the right subtree, in order
△ Each subproblem is a smaller version of the original problem – we can assume that a recursive call will work!
What is the base case for printing out a Binary Search Tree – what is an easy tree to print out?
6-36: *Printing out a BST*

- What is the base case for printing out a Binary Search Tree – what is an easy tree to print out?
- An empty tree is extremely easy to print out – do nothing!
- Code for printing a BST ...
void print(Node tree) {
    if (tree != null) {
        print(tree.left());
        System.out.println(tree.element());
        print(tree.right());
    }
}
Examples
6-39: Tree Traversals

6 PREORDER Traversal
   ▲ Do operation on root of the tree
   ▲ Traverse left subtree
   ▲ Traverse right subtree

6 INORDER Traversal
   ▲ Traverse left subtree
   ▲ Do operation on root of the tree
   ▲ Traverse right subtree

6 POSTORDER Traversal
   ▲ Traverse left subtree
   ▲ Traverse right subtree
   ▲ Do operation on root of the tree
6-40: PREORDER Examples
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:

- 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?
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
Comparable minimum(Node tree) {
    if (tree == null)
        return null;
    if (tree.left() == null)
        return tree.element();
    else
        return minimum(tree.left());
}
Iterative Version

```java
Comparable minimum(Node tree) {
    if (tree == null)
        return null;
    while (tree.left() != null)
        tree = tree.left();
    return tree.element();
}
```
6 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 \( e < \) value stored at the root, and insert the element into the right subtree if \( e > \) value stored at the root
6-56: 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, Comparable elem) {
    if (tree == null) {
        return new Node(elem);
    }
    if (elem.compareTo(tree.element()) < 0) {
        tree.setLeft(insert(tree.left(), elem));
        return tree;
    } else {
        tree.setRight(insert(tree.right(), elem));
        return tree;
    }
}
Deleting From a BST

6 Removing a leaf:
   △ Remove element immediately

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

6 Removing a node with two children:
   △ Replace node with largest element in left subtree, or the smallest element in the right subtree
We have been storing “Comparable” elements in BSTs.

Book uses “key()” method – elements stored in BSTs must implement a key() method, which returns an integer.

We can combine the two methods:
- Each element stored in the tree has a key() method
- key() method returns Comparable class