Binary Search Trees
Faster Searching

• The *BinaryTree* component family can be used to arrange the labels on binary tree nodes in a variety of useful ways.

• A common arrangement of labels, which supports searching that is much faster than linear search, is called a *binary search tree (BST)*.
BSTs Are Very General

• BSTs may be used to search for items of any type $T$ for which one has defined a total preorder, i.e., a binary relation on $T$ that is total, reflexive, and transitive
BSTs Are Very General

- BSTs may be used to search for items of any type $T$ for which one has defined a total preorder, i.e., a binary relation on $T$ that is total, reflexive, and transitive.

A binary relation on $T$ may be viewed as a set of ordered pairs of $T$, or as a boolean-valued function $R$ of two parameters of type $T$ that is true iff that pair is in the set.

total preorder, i.e., a binary relation on $T$ that is total, reflexive, and transitive
BSTs Are Very General

- BSTs may be used to search for items of any type \( T \) for which one has defined a total preorder, i.e., a binary relation \( R \) on \( T \) that is total, reflexive, and transitive.

The binary relation \( R \) is **total** whenever:

\[
\text{for all } x, y : T \quad (R(x, y) \text{ or } R(y, x))
\]
BSTs Are Very General

• BSTs may be used to search for items of any type \( T \) for which one has defined a total preorder, i.e., a binary relation on \( T \) that is total, reflexive, and transitive.

The binary relation \( R \) is reflexive whenever:

\[
\text{for all } x: T \ (R(x, x))
\]
BSTs Are Very General

• BSTs may be used to search for items of any type \( T \) for which one has defined a total preorder, i.e., a binary relation on \( T \) that is total, reflexive, and transitive.

The binary relation \( R \) is transitive whenever:

\[
\text{for all } x, y, z: T \quad (\text{if } R(x, y) \text{ and } R(y, z) \text{ then } R(x, z))
\]
Simplifications

• For simplicity in the following illustrations, we use only one kind of example:
  – $T = \text{integer}$
  – The ordering is $\leq$

• For simplicity (and because of how we will use BSTs), we assume that no two nodes in a BST have the same labels
Simplifications

• For simplicity in the following illustrations, we use only one kind of example:
  – $T = \text{integer}$
  – The ordering is $\leq$

• For simplicity (and because of how we will use BSTs), we assume that no two nodes in a BST have the same labels.

Both these simplifications are inessential: BSTs are not limited to these situations!
BST Arrangement Properties

• A binary tree is a BST whenever the arrangement of node labels satisfies these two properties:
  1. For every node in the tree, if its label is \( x \) and if \( y \) is a label in that node’s *left* subtree, then \( y < x \)
  2. For every node in the tree, if its label is \( x \) and if \( y \) is a label in that node’s *right* subtree, then \( y > x \)
The Big Picture
Every label $y$ in this tree satisfies $y < x$
Every label \( y \) in this tree satisfies \( y > x \)
And It’s So Everywhere
Every label $y$ in this tree satisfies $y < x$
And It’s So Everywhere

Every label \( y \) in this tree satisfies
\[
y > x
\]
Examples of BSTs
Non-Examples of BSTs
Non-Examples of BSTs

Property 1 is violated here.
Non-Examples of BSTs

Property 1 is violated here.
Non-Examples of BSTs

Property 2 is violated here.
Searching for $x$

• Suppose you are trying to find whether any node in a BST $t$ has the label $x$

• There are only two cases to consider:
  – $t$ is empty
  – $t$ is non-empty
Searching for $x$

• Suppose you are trying to find whether any node in a BST $t$ has the label $x$

• There are only two cases to consider:
  – $t$ is empty
  – $t$ is non-empty

Easy: Report $x$ is not in $t$. 
Searching for $x$
Searching for $X$

Does $x = r$?
If so, report that $x$ is in $t$.
If not ...
Is $x < r$?
If so, report the result of searching for $x$ in this tree.
If not ...
Then it must be the case that \( x > r \). Report the result of searching for \( x \) in this tree.
Why It’s Faster Than Linear Search

• You need to compare to the root of the tree, and then (only if the root is not what you’re searching for) search either the left or the right subtree—but not both
  – Compare to linear search, where you might have to look at all the items, which would be equivalent to searching both subtrees
Example: Searching for $5$
Does $5 = 8$?
No ...
Is $5 < 8$?
Yes, so report the result of searching for 5 in this tree.
Recursion

• Searching the left subtree at this point simply involves making a **recursive call** to the method that searches a BST

• Against our usual advice about recursion, let’s trace into that call and see what happens
  
  – Why? Because some people, e.g., interviewers, may expect you to understand BSTs without mentioning recursion/induction
Example: Searching for 5
Does $5 = 3$?
No ...
Searching for 5

Is $5 < 3$?
No ...
Then it must be the case that $5 > 3$. Report the result of searching for 5 in this tree.
It’s Another Recursive Call

- Let’s continue tracing into calls ...
Example: Searching for 5
Does $5 = 6$?
No ...
Is $5 < 6$?

Yes, so report the result of searching for $5$ in the (empty) left subtree.
The Recursion Stops Here

• Remember, we already noted that when searching for something in an empty tree, we can simply report it is not there
• No new recursive call results
How many nodes did the algorithm visit, and compare labels to 5? At worst, how many could it be?
Example: Searching for 5

What about in this tree?
Wait! How Can This Work?

• With the `BinaryTree` components, there are no methods to “move down the tree”
• This is why recursion is crucial
  – To search a subtree, you disassemble the original tree, search in one of the subtrees, and then (re)assemble it before returning the answer
Refined Searching for $x$
Does $x = r$?
If so, report that $x$ is in $t$.
If not ...
Disassemble $t$ into the root and its two subtrees.
Is $x < r$?
If so, remember the result of searching for $x$ in this tree.
If not ...
Then it must be the case that $x > r$. Remember the result of searching for $x$ in this tree.
Searching for $x$

Before returning the result of the search, (re)assemble $t$ from its parts.
Inserting $x$

• Suppose now you are trying to insert into a BST $t$ the label $x$ (which we assume to be not already in $t$; remember that there are no duplicate labels in $t$)

• There are only two cases to consider:
  – $t$ is empty
  – $t$ is non-empty
Inserting $x$

- Suppose now you are trying to insert into a BST $t$ the label $x$ (which we assume to be not already in $t$; remember that there are no duplicate labels in $t$)

- There are only two cases to consider:
  - $t$ is empty
  - $t$ is non-empty

Easy: Make $x$ the root of the updated $t$. 
Inserting $x$
There is no reason to ask whether \( x = r \); why?
Is $x < r$?
If so, insert $x$ into this tree.
If not ...
Then it must be the case that $x > r$. Insert $x$ into this tree.
Removing the Smallest

• Suppose now you are trying to remove from a BST $t$ the smallest label (assuming that $t$ is not empty)

• There is only one case to consider:
  – $t$ is non-empty
Removing the Smallest
Does the root have a non-empty left subtree?
If so, remove the smallest label from this tree.
If not, then \( r \) is the smallest label in \( t \). Make the right subtree the new value of \( t \), and return \( r \).
Removing $x$

- Suppose now you are trying to remove from a BST $t$ the label $x$ (which we assume to be in $t$)

- There is only one case to consider:
  - $t$ is non-empty
Removing $x$
Does $x = r$?
If so, ouch! (Later ...)
If not ...
Is \( x < r \)?
If so, remove \( x \) from this tree.
If not ...
Then it must be the case that $x > r$. Remove $x$ from this tree.
Back to the problematic case: $x = r$, i.e., we need to remove the root of $t$. What can we do?
The Idea

• To avoid restructuring the entire tree, we can move to the root some label from further down in the tree that would not violate the BST arrangement properties.

• There are two good possibilities for the label to be moved:
  – The next-smaller label than $x$
  – The next-larger label than $x$
Leverage Prior Work

• We already know how to remove the smallest label from a tree

• So, it is easier to implement this strategy:
  – Remove the smallest label from the right subtree (because this is the next-larger label after $x$ in the original tree $t$)
  – Make that label the new root of $t$
Leverage Prior Work

• We already know how to remove the smallest label from a tree.

• So, it is easier to implement this strategy:
  – Remove the smallest label from the right subtree (because this is the next-larger label after $x$ in the original tree $t$)
  – Make that label the new root of $t$

But first, what if the right subtree is empty?
If the right subtree is empty, then make the left subtree the new value of $t$. 
If the right subtree is not empty, then replace $x$ in the root with the smallest label from the right subtree.
Resources

• *Big Java (4th ed)*, Section 16.5
  – [https://library.ohio-state.edu/record=b9476806~S7](https://library.ohio-state.edu/record=b9476806~S7)