The Binary Search Tree ADT
Binary Search Tree

- A **binary search tree (BST)** is a binary tree with an **ordering** property of its elements, such that the data in any internal node is
  - **Greater than** the data in any node in its left subtree
  - **Less than** the data in any node in its right subtree
- **Note**: this definition does not allow duplicates; some definitions do, in which case we could say “**less than or equal to**”
Examples: are these Binary Search Trees?
Discussion

• Observations:
  • What is in the leftmost node?
  • What is in the rightmost node?
BST Operations

• A binary search tree is a special case of a binary tree
  • So, it has all the operations of a binary tree
• It also has *operations specific to a BST*:
  • *add* an element (requires that the BST property be maintained)
  • *remove* an element (requires that the BST property be maintained)
  • *remove the maximum* element
  • *remove the minimum* element
Searching in a BST

• Why is it called a binary *search* tree?
  • Data is stored in such a way, that it can be more *efficiently* found than in an ordinary binary tree
Searching in a BST

• Algorithm to search for an item in a BST

  • Compare data item to the root of the (sub)tree
  • If data item = data at root, found
  • If data item < data at root, go to the left; if there is no left child, data item is not in tree
  • If data item > data at root, go to the right; if there is no right child, data item is not in tree
Search Operation – a Recursive Algorithm

To search for a value $k$;

returns true if found or false if not found

If the tree is empty, return false.

If $k ==$ value at root

   return true: we’re done.

If $k <$ value at root

   return result from search for $k$ in the left subtree

Else

   return result from search for $k$ in the right subtree.
**Search Operation**

*Search for 13*: visited nodes are coloured yellow; return false when node containing 12 has no right child

*Search for 22*: return false when node containing 23 has no left child
BST Operations: add

• To add an item to a BST:
  • Follow the algorithm for searching, until there is no child
  • Insert at that point

• So, new node will be added as a leaf
• (We are assuming no duplicates allowed)
To insert 13:

Same nodes are visited as when *searching* for 13.

Instead of returning *false* when the node containing 12 has no right child, build the new node, attach it as the right child of the node containing 12, and return *true*. 
Add Operation – an Algorithm

To insert a value $k$ into a tree, returning true if successful and false if not

Build a new node for $k$.
If tree is empty
    add new node as root node, return true.
If $k ==$ value at root
    return false (no duplicates allowed).
If $k <$ value at root
    If root has no left child
        add new node as left child of root, return true
    Else insert $k$ into left subtree of root.
If $k >$ value at root
    If root has no right child
        add new node as right child of root, return true
    Else insert $k$ into the right subtree of root.
Example: Adding Elements to a BST

1: Add 26

26

15

2: Add 15

26

15

3: Add 38

26

15

38

31

4: Add 31

26

15

38

31

5: Add 7

26

15

38

7

31

5: Add 34

26

15

38

7

31

34
Binary Search Tree Traversals

• Consider the traversals of a binary search tree: preorder, inorder, postorder, level-order

• Try the traversals on the example on the next page
  • Is there anything special about the order of the data in the BST, for each traversal?

• Question: what if we wanted to visit the nodes in descending order?
Binary Search Tree Traversals

Try these traversals:

• preorder
• inorder
• postorder
• level-order
Binary Search Tree ADT

• A BST is just a binary tree with the ordering property imposed on all nodes in the tree.

• So, we can define the **BinarySearchTreeADT** interface as an extension of the **BinaryTreeADT** interface.
public interface BinarySearchTreeADT<T> extends BinaryTreeADT<T> {
    public void addElement(T element);
    public T removeElement(T targetElement);
    public void removeAllOccurrences(T targetElement);
    public T removeMin();
    public T removeMax();
    public T findMin();
    public T findMax();
}

The BinarySearchTreeADT interface
Implementing BSTs using Links

• See `LinkedBinarySearchTree.java`
  • Constructors: use `super()`
  • `addElement` method
    • *(does not implement our recursive algorithm of p.12; also, allows duplicates)*
  • note the use of `Comparable`: so that we can use `compareTo` method to know where to add the new node
• `removeMin` method
Implementing BSTs using Links

• The special thing about a Binary Search Tree is that finding a specific element is efficient!
  • So, LinkedBinarySearchTree has a find method that overrides the find method of the parent class LinkedBinaryTree
    • It only has to search the appropriate side of the tree
    • It uses a recursive helper method findAgain
  • Note that it does not have a contains method that overrides the contains of LinkedBinaryTree – why not?
    • It doesn’t need to, because contains just calls find
Using Binary Search Trees: Implementing Ordered Lists

• A BST can be used to provide efficient implementations of other collections!
• We will examine an implementation of an **Ordered List ADT** as a **binary search tree**
• Our implementation is called **BinarySearchTreeList.java**
  (naming convention same as before: this is a BST implementation of a List)
Using BST to Implement Ordered List

- BinarySearchTreeList implements `implements` OrderedListADT
  - Which extends ListADT
  - So it also implements ListADT
  - So, what operations do we need to implement?
    - add
    - removeFirst, removeLast, remove, first, last, contains, isEmpty, size, iterator, toString
  - But, for which operations do we actually need to write code? …
Using BST to Implement Ordered List

- **BinarySearchTreeList** extends our binary search tree class **LinkedBinarySearchTree**
  - Which extends **LinkedBinaryTree**
  - So, what operations have we *inherited*?
    - `addElement`, `removeElement`, `removeMin`, `removeMax`, `findMin`, `findMax`, `find`
    - `getRoot`, `isEmpty`, `size`, `contains`, `find`, `toString`, `iteratorInOrder`, `iteratorPreOrder`, `iteratorPostOrder`, `iteratorLevelOrder`
Discussion

• First, let’s consider some of the methods of the List ADT that we do not need to write code for:
  • contains method: we can just use the one from the LinkedBinaryTree class
  • What about the methods
    • isEmpty
    • size
    • toString
Discussion

• To implement the following methods of the `OrderedListADT`, we can call the appropriate methods of the `LinkedBinarySearchTree` class (fill in the missing ones)
  • `add` call `addElement`
  • `removeFirst` call `removeMin`
  • `removeLast`
  • `remove`
  • `first`
  • `last`
  • `iterator`