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
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 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

2: Add 15

3: Add 38

4: Add 31

5: Add 7

5: Add 34
BST Operations: Remove

• **Case 1**: value to be removed is in a *leaf node*
  • Node can be removed and the tree needs no further rearrangement

• **Case 2**: value to be removed is in an *interior node*
  • Why can’t we just change the link from its parent node to a successor node?
  • *We can* replace the node with its *inorder* predecessor (or successor)
    • Complex, and we will not implement this
Example: Removing BST Elements

1: Initial tree

```
26
  /   \
15   38
 / \
7   31
 / \
9   59
```

2: Remove 70

```
26
  /   \
15   38
 / \
7   31
 / \
9   34
```

```
26
  /   \
15   38
 / \
7   31
 / \
9   34
```
3: Remove 7

4: Remove 38
5: Remove 26

6: Remove 31
BST Operations: Remove Minimum

• Recall that *leftmost node* contains the minimum element
• Three cases:
  1) *root has no left child* (so, root is minimum)
     • its right child becomes the root
  2) *leftmost node is a leaf*
     • set its parent’s left child to null
  3) *leftmost node is internal*
     • the right child of the node to be removed becomes the parent’s left child
Example: Removing Minimum BST Element

1: Initial tree

26

15

38

7

18

33

59

9

2: Remove minimum

26

15

38

9

18

33

59

Case 3: internal node removed
3: Remove minimum

Case 2: leaf node removed

4: Remove minimum

Case 3: internal node removed
5: Remove minimum

26

38

33  59

Case 2: leaf node removed

6: Remove minimum

38

33  59

Case 1: root node removed
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
The `BinarySearchTreeADT` interface
UML Description of BinarySearchTreeADT

<<interface>>
BinarySearchTreeADT

addElement( )
removeElement( )
removeAllOccurrences( )
removeMin( )
removeMax( )
findMin( )
findMax( )

<<interface>>
BinaryTreeADT

getRoot()
toString()
isEmpty( )
size( )
contains( )
find( )
iteratorInOrder( )
iteratorPreOrder( )
iteratorPostOrder( )
iteratorLevelOrder( )
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
    • essentially implements our algorithm of p. 18
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 `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
  • removeFirst
  • removeLast
  • remove
  • first
  • last
  • iterator
Balanced Trees

- **Our definition**: a *balanced tree* has the property that, for any node in the tree, the height of its left and right subtrees can *differ by at most 1*
  - Note that conventionally the height of an empty subtree is *-1*
Balanced Trees

Which of these trees is a balanced tree?
Analysis of BST Implementation

• We will now compare the linked list implementation of an ordered list with its BST implementation, making the following important assumptions:
  • The BST is a *balanced* tree
  • The maximum level of any node is $\log_2(n)$, where $n$ is the number of elements stored in the tree
## Analysis of Ordered List Implementations: Linked List vs. Balanced BST

<table>
<thead>
<tr>
<th>Operation</th>
<th>LinkedList</th>
<th>BinarySearchTreeTreeList</th>
</tr>
</thead>
<tbody>
<tr>
<td>removeFirst</td>
<td>O(1)</td>
<td>O(log₂n)</td>
</tr>
<tr>
<td>removeLast</td>
<td>O(n)</td>
<td>O(log₂n)</td>
</tr>
<tr>
<td>remove</td>
<td>O(n)</td>
<td>O(log₂n) *</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*but may cause tree to become unbalanced</td>
</tr>
<tr>
<td>first</td>
<td>O(1)</td>
<td>O(log₂n)</td>
</tr>
<tr>
<td>last</td>
<td>O(n)</td>
<td>O(log₂n)</td>
</tr>
<tr>
<td>contains</td>
<td>O(n)</td>
<td>O(log₂n)</td>
</tr>
<tr>
<td>isEmpty</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>size</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>add</td>
<td>O(n)</td>
<td>O(log₂n) *</td>
</tr>
</tbody>
</table>
Discussion

• Why is our balance assumption so important?
  • Look at what happens if we insert the following numbers in this order without rebalancing the tree:
    
    3  5  9  12  18  20
Degenerate Binary Trees

• The resulting tree is called a \textit{degenerate} binary tree
  • Note that it looks more like a linked list than a tree!
  • But it is actually less efficient than a linked list (\textit{Why?})
Degenerate Binary Trees

• Degenerate BSTs are far less efficient than balanced BSTs
  • Consider the worst case time complexity for the \texttt{add} operation:
    • $O(n)$ for degenerate tree
    • $O(\log_2{n})$ for balanced tree
Balancing Binary Trees

• There are many approaches to balancing binary trees
  • But they will not be discussed in this course …