Binary Trees Lab

Lab 9

Generated by Doxygen 1.8.6

Wed Nov 4 2015 09:13:59

Contents

1 Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

2 Class Index

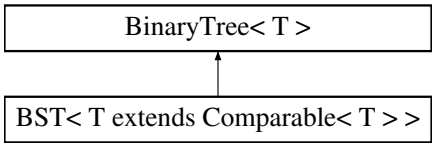
2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

3 Class Documentation

3.1 BinaryTree< T > Class Reference

Inheritance diagram for BinaryTree< T >:



Public Member Functions

- BinaryTree ()
- void add (T value)
- boolean isEmpty ()
- T getValue ()
- void setValue (T newValue)
- BinaryTree< T > getLeft ()
- void setLeftValue (T value)
- BinaryTree< T > getRight ()
- void setRightValue (T value)
- boolean isLeaf ()
- int getSize ()
- int height ()
- boolean equals (Object other)

Protected Member Functions

- boolean hasLeft ()
- boolean hasRight ()

3.1.1 Detailed Description

Common "base class" for any and all directed di-graphs (or, what we will be creating in this class). This class is extended by the Binary Search Tree class (qv), and therefore relies upon some of its properties (indirectly) and methods. Note that all of the methods belonging to this class address the *structure* of the tree, which is a di-graph, independently of what values might be stored at these locations. In other words: the operations defined by this class are common to all directed di-graphs, which means that they are appropriate to the BST (binary search tree) that you will be required to define using this as a base class.

Observe that the signatures for many of the methods on this class return *subtree*s and **never** `BTNode`s. This ensures that clients never know anything about the underlying representation of these objects or any of their descendants.

A note on exceptions

Implementors *should not rely* upon the exception handling mechanism in place of using the appropriate methods to ensure that elements in a Binary Tree are in an appropriate state before invoking accessors: in other words, you should check that a node is not `null` before calling a method such as `getValue()`.

Author

UMD CS Dept.

Parameters

<T>	

3.1.2 Constructor & Destructor Documentation

3.1.2.1 BinaryTree ( )

default ctor, required for inheritance mechanism..

3.1.3 Member Function Documentation

3.1.3.1 void add (T *value*)

Adding a value to a Binary Tree, in this case, means placing the value at the first available location in the tree. Note: this logic is somewhat arbitrary, unlike what we do with a Binary Search Tree, where the location of the new value to be added is uniquely determined by the ordering relation used to define the BST. Lacking any kind of an ordering relation, we default to this behavior. In a more sophisticated implementation, we might prefer a location that keeps the resulting tree "balanced." And this last point is a chief motivation for the study of balanced binary trees, which you will explore later in your time here.

Parameters

value	

3.1.3.2 boolean equals (Object *other*)

Overrides the Object equals method to return true only when this and the other tree match node by node, value by value. Note: Binary Trees, as specified in this implementation, do not allow NULL values. Nevertheless, the equals method method may be called with a `null` parameter `other`.

3.1.3.3 BinaryTree< T > getLeft ( )

Throws an unspecified exception if this tree is empty; otherwise, returns the left subtree attached to this tree. Note, the left subtree may be null.

Returns

3.1.3.4 BinaryTree< T > getRight ( )

Throws an unspecified exception if this tree is empty; otherwise returns the right subtree attached to this tree. Note, the right subtree may be null.

Returns

3.1.3.5 int getSize ( )

Returns the size of any binary tree, which is the number of nodes it contains.

Returns

3.1.3.6 T getValue ( )

Given any non-empty binary tree, returns the value at its root. Note: if the `tree` is empty, then a runtime exception is thrown.

Returns

3.1.3.7 boolean hasLeft ( ) [protected]

[Optional] A convenience method: returns true only when the left branch contains a non-empty tree.

Returns

3.1.3.8 boolean hasRight ( ) [protected]

[Optional method] As a convenience, returns true only when the right branch contains a non-empty tree.

Returns

3.1.3.9 int height ( )

The "height" of any binary tree is the number of edges in the longest path from its root to its deepest child (leaf) node. (You may use Java's `Math` library for part of this function's computation if you wish.)

Very important

Numerous "conventions" are available here. Some give the height of an empty tree to be 0, others -1. We will specify that the height of an empty tree is -1.

Returns

a non-negative integer (for non-empty trees, -1 in the case of an empty tree)

3.1.3.10 boolean isEmpty ( )

Returns true iff this tree is empty. Note: this is the only "safe" method that is provided for this purpose. Clients are expected to use this method rather than relying upon catching a runtime exception.

Returns

3.1.3.11 boolean isLeaf ( )

Returns true iff this tree (or subtree) is non-empty and has neither a left nor a right branch.

Returns

3.1.3.12 void setLeftValue (T *value*)

Replaces value of left child with `value`.

Parameters

value	

3.1.3.13 void setRightValue (T *value*)

Replaces value of right child with `value`/

Parameters

value	

3.1.3.14 void setValue (T *newValue*)

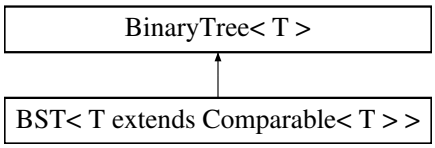
Replaces the "value" stored on this tree (its root) with `newValue`.

Parameters

newValue	

3.2 BST< T extends Comparable< T > > Class Reference

Inheritance diagram for BST< T extends Comparable< T > >:



Public Member Functions

- BST ()
- boolean contains (T ele)
- void add (T value)
- BST< T > remove (T value)
- void printPreOrder ()
- Iterator< T > iterator ()

Static Public Member Functions

- static< TextendsComparable< T > >
  boolean isBST (BinaryTree< T > tree)

Additional Inherited Members

3.2.1 Detailed Description

This version of the classic Binary Search Tree is similar to the commonly found textbook implementations. Our purpose in creating the lab is to have you practice implementing many commonly-referenced "binary tree" and "binary search tree" algorithms. Note the distinction: not every Binary Tree is a Binary Search Tree. Hence, some operations belong to **all** trees (i.e., are shared by all directed di-graphs), such as counting the number of nodes contained in the tree, obtaining a left or right subtree, determining if a tree is a leaf, determining if two trees are `equal` and other operations. Other operations, however, only make sense in the setting of a Binary Search Tree, such as getting the maximum or minimum elements, and *optimized* searching because in the case of BSTs, some ordering relation has been specified and this guides the structure of the tree itself.

Differences/Exceptional Requirements

You will first implement a generic `BinaryTree` class that is parameterized on general types (objects). This class must provide a handful of "structural" operations, such as retrieving left and right branches, retrieving values from the root (topmost node of a tree), and a handful of predicates—including an override of the `equals` method.

Your BST must *extend* the `BinaryTree` class, and must, in addition, restrict the values that may reside within the BST to homogeneous Comparable types. The construction algorithm should place values that are less than or equal to the root to the left, and those greater to the right. Duplications will be permitted.

Finally, your implementation should provide an `Iterator` that allows users to iterate through the nodes of a BST *in "sort" order*, i.e., the order in which nodes values' should be presented by the Iterator must correspond to their *natural ordering* as determined by the `compareTo` method.

Author

UMD CS Department

Parameters

<T>	

3.2.2 Constructor & Destructor Documentation

3.2.2.1 BST ( )

Default ctor(): depending upon how one initialized the internal tree, this method's body is likely empty.

3.2.3 Member Function Documentation

3.2.3.1 void add (T *value*)

Inserts `value` in its correct position in the tree, as determined by its natural ordering.

Parameters

value	

Returns

3.2.3.2 boolean contains (T *ele*)

This method returns true only when `ele` appears somewhere in the tree.

Parameters

ele	

Returns

3.2.3.3 static <TextendsComparable<T> boolean isBST (BinaryTree< T > *tree*) [static]

Accepts any `Binary Tree` and returns `true` iff it satisfies the definition of a binary search tree: (1) it is empty; (2) it is a leaf; or (3) its left and right subtrees (left and right values) are less than or equal or greater than the value at the immediate root (the immediate) parent, respectively.

Parameters

tree	

Returns

3.2.3.4 Iterator< T > iterator ( )

Returns an Iterator<T>, that enumerates the values in the underlying tree **in order**. (You might find it easier to create an internal list by performing an in-order traversal of the tree, adding to the internal list as you go and going from there!)

3.2.3.5 void printPreOrder ( )

Untested, but valuable method ... you should consider writing this method as well as a printInOrder() method, which might help you in constructing your Iterator.

3.2.3.6 BST<T> remove (T *value*)

Remove **one (the first in sort order)** occurrence of `value` from the tree. Do this by recursively walking the tree, splicing out nodes whose values matches `value`.

Parameters

value	

Returns

Index