COMS W3134 Data Structure - Problem Set #3

Xiaoli Sun (xs2338) - xs2338@columbia.edu

March 28, 2020

Problem 1 (Weiss 4.6)

Let x:=the number of full nodes. y:=the number of leaves. Use induction:
Base case: when x=1. Since the tree is non-empty, the root has 1 leaf. Thus, x=0, y=1, x+1=y. Correct.
Hypothesis: suppose x+1=y is true for all x $\in [0, k]$
Induction: when $x_{new} = k + 1$, we can create this new tree based on an old tree where $x_{old} = k$. Convert a leaf to a full node in the old tree. Thus, y is deducted by 1 since the leaf is now a root. And y is increased by 2 since the full node we just created has 2 children.
$y_{new} = y_{old} - 1 + 2 = y_{old} + 1 = (x_{old} + 1) + 1 = x_{new} + 1$.
Thus, for $x_{new} = k + 1$ we have x+1=y.
Based on the induction, the number of full nodes plus one equal to the number of leaves in a nonempty tree.

Problem 6

We should check if the node has been marked as "deleted" before we perform each method.

```
public class BinarySearchTree<AnyType extends Comparable<? super AnyType>> {
    /**
     * Construct the tree.
     */
    public BinarySearchTree( )
    {
        root = null;
    }

    /**
     * Insert into the tree; duplicates are ignored.
     * @param x the item to insert.
     */
    public void insert( AnyType x )
    {
        root = insert( x, root );
    }

    /**
     * Remove from the tree. Nothing is done if x is not found.
     * @param x the item to remove.
     */
    public void remove( AnyType x )
    {
        root = remove( x, root );
    }

    /**
     * Find the smallest item in the tree.
     * @return smallest item or null if empty.
     */
    public AnyType findMin( )
    {
        if(isEmpty( ))
            throw new NullPointerException();
        AnyType result = findMin(root).element;
        return result;
    }
```

```
/**
 * Find the largest item in the tree.
 * @return the largest item of null if empty.
 */
public AnyType findMax( )
{
    if( isEmpty( ) )
        throw new NullPointerException();
    AnyType result =findMax(root).element;
    return result;
}

/**
 * Find an item in the tree.
 * @param x the item to search for.
 * @return true if not found.
 */
public boolean contains( AnyType x )
{
    return contains( x, root );
}

/**
 * Make the tree logically empty.
 */
public void makeEmpty( )
{
    root = null;
}

/**
 * Test if the tree is logically empty.
 * @return true if empty, false otherwise.
 */
public boolean isEmpty( )
{
    if(this.findMax(root)==null && this.findMin(root)==null) {
        return true;
    } else {
        return false;
    }
}
```

```
/**
 * Print the tree contents in sorted order.
 */
public void printTree( )
{
    if( isEmpty( ) )
        System.out.println( "Empty tree" );
    else
        printTree( root );
}

/**
 * Internal method to insert into a subtree.
 * @param x the item to insert.
 * @param t the node that roots the subtree.
 * @return the new root of the subtree.
 */
private BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )
{
    if( t == null )
        return new BinaryNode<>( x, null, null );

    int compareResult = x.compareTo( t.element );

    if( compareResult < 0 )
        t.left = insert( x, t.left );
    else if( compareResult > 0 )
        t.right = insert( x, t.right );
    else {
        //duplicate
        if(t.deleted==true) {
            t.deleted=false;
        } // if it has been deleted before, add back. else, do nothing
    }

    return t;
}

/**
 * Internal method to remove from a subtree.
```

```
     * @param x the item to remove.
     * @param t the node that roots the subtree.
     * @return the new root of the subtree.
     */
    private BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
    {
        if( t == null )
            return t;   // Item not found; do nothing

        int compareResult = x.compareTo( t.element );

        if( compareResult < 0 )
            t.left = remove( x, t.left );
        else if( compareResult > 0 )
            t.right = remove( x, t.right );
        else
            t.deleted=true; //lazy deletion
        return t;
    }



    /**
     * Internal method to find the smallest item in a subtree.
     * @param t the node that roots the subtree.
     * @return node containing the smallest item.
     */
    private BinaryNode<AnyType> findMin( BinaryNode<AnyType> t ) {
        //if t!=null
        if (t != null) {
            if(t.deleted==true) {
                if(t.left==null) {
                    if(t.right==null) {
                        return null;
                    } else {
                        return findMin(t.right);
                    }
                } else {
                    //t.left!=null
                    if(t.right==null) {
                        return findMin(t.left);
                    } else {
                        //t.right !=null
                        if(findMin(t.left)==null) {
                            return findMin(t.right);
                        } else {
                            return findMin(t.left);
                        }
                    }
                }
            } else {
                //t.delete==false
                if(t.left==null) {
                    return t;
                } else {
                    //t.left!=null
                    if(findMin(t.left)==null) {
                        return t;
                    } else {
                        return findMin(t.left);
                    }
                }

            }
        }
        //t==null
        return null;
    }

    /**
     * Internal method to find the largest item in a subtree.
     * @param t the node that roots the subtree.
     * @return node containing the largest item.
     */
    private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
    {
        if(t!=null) {
            if(t.deleted==true) {
                if(t.right==null) {
                    if(t.left==null) {
                        return null;
                    } else {
                        return findMax(t.left);
                    }
                } else {
                    if(t.left==null) {
                        return findMax(t.right);
                    } else {
                        //t.left!=null
                        if(findMax(t.right)==null) {
                            return findMax(t.left);
                        } else {
                            return findMax(t.right);
                        }
                    }
                }
            } else {
                if(t.right==null) {
                    return t;
                } else {
                    if(findMax(t.right)==null) {
                        return t;
                    } else {
                        return findMax(t.right);
                    }
                }
            }
        }
        //t==null
        return null;
    }

    /**
     * Internal method to find an item in a subtree.
     * @param x is item to search for.
     * @param t the node that roots the subtree.
     * @return node containing the matched item.
     */
    private boolean contains( AnyType x, BinaryNode<AnyType> t )
    {
        if( t == null )
            return false;

        int compareResult = x.compareTo( t.element );

        if( compareResult < 0 )
            return contains( x, t.left );
        else if( compareResult > 0 )
            return contains( x, t.right );
        else
        if(t.deleted==false){
            return true;    // Match
        } else {
            return false; //Match but has been deleted
        }

    }

    /**
     * Internal method to print a subtree in sorted order.
     * @param t the node that roots the subtree.
     */
    private void printTree( BinaryNode<AnyType> t )
    {
        if( t != null )
        {
            printTree( t.left );
            if(t.deleted==false) {
                System.out.println( t.element );
            }
            printTree( t.right );
        }
    }

    /**
     * Internal method to compute height of a subtree.
     * @param t the node that roots the subtree.
     */
    private int height( BinaryNode<AnyType> t )
    {
        if( t == null )
            return -1;
        else
            return 1 + Math.max( height( t.left ), height( t.right ) );
    }

    // Basic node stored in unbalanced binary search trees
    private static class BinaryNode<AnyType>
    {
        // Constructors
        BinaryNode( AnyType theElement )
        {
            deleted = false;
            element  = theElement;
            left     = null;
            right    = null;
        }

        BinaryNode( AnyType theElement, BinaryNode<AnyType> lt, BinaryNode<AnyType> rt
        {
            deleted = false;
            element  = theElement;
            left     = lt;
            right    = rt;
        }


        AnyType element;            // The data in the node
        BinaryNode<AnyType> left;   // Left child
        BinaryNode<AnyType> right;  // Right child
        boolean deleted;            // for lazy deletion, deleted=true if this node ha
    }


    /** The tree root. */
    private BinaryNode<AnyType> root;


    // Test program
    public static void main( String [ ] args )
    {
        BinarySearchTree<Integer> t = new BinarySearchTree<>( );
        final int NUMS = 400;
        final int GAP  =   37;

        System.out.println( "Checking... (no more output means success)" );

        for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
            t.insert( i );

        for( int i = 1; i < NUMS; i+= 2 )
            t.remove( i );
        if( NUMS < 40 )
            t.printTree( );
        if( t.findMin( ) != 2 || t.findMax( ) != NUMS - 2 )
            System.out.println( "FindMin or FindMax error!" );
        for( int i = 2; i < NUMS; i+=2 )
            if( !t.contains( i ) )
                System.out.println( "Find error1!" );

        for( int i = 1; i < NUMS; i+=2 )
        {
            if( t.contains( i ) )
                System.out.println( "Find error2!" );
        }

    }

}
```