1 Treaps

- Treaps are a simple way to do insertions and deletions to keep a binary search tree balanced.

- Each TreapNode stores a data item, a left node link and a right node link as in a Binary Search Tree. We add one extra field labeled `priority`.

- The `priority` is used to determine at what level the inserted node resides. The trick is to maintain a heap order in the tree which means that every node has a higher priority than its parent node. Equivalently, the parent node has a lower priority than its children.

- By maintaining this heap priority order, nodes are randomly pushed higher or lower in the tree, negating any effects of an unbalanced tree - each insertion or deletion reorders the tree by priority, but still keeping the Binary Search Tree order we require.

- To do an insertion, we insert into the Treap as in a Binary Search Tree. Once we find the insertion spot, we then adjust the tree to fulfill the heap priority protocol, rotation a node up the tree until its priority is less than its parent.

- To do a deletion, we find the node to be deleted and assume its priority is a maximum (lowest level in the tree). This allows us to rotate the node down the tree until it becomes a leaf. Once we reach the node being a leaf, we can simply delete it.

- The code that follows is a slight modification of the Treap.java code in the Weiss solutions: [http://users.cis.fiu.edu/~weiss/dsaajava3/code/](http://users.cis.fiu.edu/~weiss/dsaajava3/code/)

- This version creates a Treap with number_of_nodes (from the command line argument), prints the treap, and then asks for a node to be deleted and prints the Treap again.
// Treap class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x ) --> Insert x
// void remove( x ) --> Remove x
// boolean contains( x ) --> Return true if x is found
// Comparable findMin( ) --> Return smallest item
// Comparable findMax( ) --> Return largest item
// boolean isEmpty( ) --> Return true if empty; else false
// void makeEmpty( ) --> Remove all items
// void printTree( ) --> Print tree in sorted order
// ******************ERRORS******************************
// Throws UnderflowException as appropriate

/**
 * Implements a treap.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 *
 * modified by Peter Allen: this version creates a treap with number_of_nodes (from command
 * line argument), printe the treap, and then asks for nodes to be deleted and prints the treap again
 */

import java.io.*;
import java.util.*;

public class Treap<AnyType extends Comparable<? super AnyType>>
{
    /**
     * Construct the treap.
     */
    public Treap()
    {
        nullNode = new TreapNode<>(null);
        nullNode.left = nullNode.right = nullNode;
        nullNode.priority = Integer.MAX_VALUE;
        root = nullNode;
    }

    /**
     * Insert into the tree. Does nothing if x is already present.
     * @param x the item to insert.
     */
    public void insert( AnyType x )
    {
        root = insert( x, root );
    }

    /**
     * Remove from the tree. Does nothing 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 the smallest item, or throw UnderflowException if empty.
     */
    public AnyType findMin()
if ( isEmpty( ) ) {
    System.out.println("underflow!");
    System.exit(0);
}

TreapNode<AnyType> ptr = root;
while ( ptr.left != nullNode )
    ptr = ptr.left;
return ptr.element;

/**
 * Find the largest item in the tree.
 * @return the largest item, or throw UnderflowException if empty.
 */
public AnyType findMax( )
{
    if ( isEmpty( ) ) {
        System.out.println("underflow!");
        System.exit(0);
    }
    TreapNode<AnyType> ptr = root;
    while ( ptr.right != nullNode )
        ptr = ptr.right;
    return ptr.element;
}

/**
 * Find an item in the tree.
 * @param x the item to search for.
 * @return true if x is found.
 */
public boolean contains( AnyType x )
{
    TreapNode<AnyType> current = root;
    nullNode.element = x;
    for ( ; ; )
    {
        int compareResult = x.compareTo( current.element );
        if ( compareResult < 0 )
            current = current.left;
        else if ( compareResult > 0 )
            current = current.right;
        else
            return current != nullNode;
    }
}

/**
 * Make the tree logically empty.
 */
public void makeEmpty( )
{
    root = nullNode;
}
/**
 * Test if the tree is logically empty.
 * @return true if empty, false otherwise.
 */
public boolean isEmpty()
{
    return root == nullNode;
}

/**
 * Print the tree contents in sorted order.
 */
public void printTree()
{
    if( isEmpty( ) )
        System.out.println( "Empty tree" );
    else
        printTree( root, 0 );
}

/**
 * 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 TreapNode<AnyType> insert( AnyType x, TreapNode<AnyType> t )
{
    if( t == nullNode )
        return new TreapNode<>( x, nullNode, nullNode );
    int compareResult = x.compareTo( t.element );
    if( compareResult < 0 )
    {
        t.left = insert( x, t.left );
        if( t.left.priority < t.priority )
            t = rotateWithLeftChild( t );
    }
    else if( compareResult > 0 )
    {
        t.right = insert( x, t.right );
        if( t.right.priority < t.priority )
            t = rotateWithRightChild( t );
    }
    // Otherwise, it’s a duplicate; 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 TreapNode<AnyType> remove( AnyType x, TreapNode<AnyType> t )
{
    if( t != nullNode )
    {
        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 );
        }
        // Otherwise, it’s a duplicate; do nothing
        return t;
    }
}
t.right = remove( x, t.right );
else
{
    // Match found
    if( t.left.priority < t.right.priority )
        t = rotateWithLeftChild( t );
    else
        t = rotateWithRightChild( t );
    if( t != nullNode ) // Continue on down
        t = remove( x, t );
    else
        t.left = nullNode; // At a leaf
}
return t;

/**
 * Internal method to print a subtree in sorted order.
 * @param t the node that roots the tree.
 */
private void printTree( TreapNode<AnyType> t, int level )
{
if( t != t.right )
{
    printTree( t.right, level + 5 );
    for (int j =0; j<level;j++) System.out.print(" ");
    System.out.println( t.element.toString( ) );
    printTree( t.left, level + 5 );
}
}

/**
 * Rotate binary tree node with left child.
 */
private TreapNode<AnyType> rotateWithLeftChild( TreapNode<AnyType> k2 )
{
    TreapNode<AnyType> k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    return k1;
}

/**
 * Rotate binary tree node with right child.
 */
private TreapNode<AnyType> rotateWithRightChild( TreapNode<AnyType> k1 )
{
    TreapNode<AnyType> k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    return k2;
}

private static class TreapNode<AnyType>
{
    // Constructors
    TreapNode( AnyType theElement )
    {
        this( theElement, null, null );
    }

    TreapNode( AnyType theElement, TreapNode<AnyType> lt, TreapNode<AnyType> rt )
    {
element = theElement;
left = lt;
right = rt;
priority = generator.nextInt(1000);
}

// Friendly data; accessible by other package routines
AnyType element; // The data in the node
TreapNode<AnyType> left; // Left child
TreapNode<AnyType> right; // Right child
int priority; // Priority
private static Random generator = new Random();
}

private TreapNode<AnyType> root;
private TreapNode<AnyType> nullNode;

// Test program
public static void main( String [] args )
{
    if (args.length!=1) {
        System.out.println("usage: java Treap num_of_nodes");
        System.exit(0);
    }
    int NUMS = Integer.parseInt(args[0]);
    Treap<Integer> t = new Treap<>(
    for( int i = 0; i<NUMS; i++ )
        t.insert( i );
    System.out.println("Inserts complete");
    t.printTree( );
    Scanner scan = new Scanner(System.in);
    System.out.println("enter node to deleted");
    int x=scan.nextInt();
    while(x>=0){
        t.remove(x);
        System.out.println("tree with " + x + " removed");
        t.printTree();
        System.out.println("enter node to deleted");
        x=scan.nextInt();
    }
}
}