package homework\_5\_csc310;

public class Homework\_5\_CSC310

{

public static void main(String[] args)

{

//problem 1

hanoi tower\_1 = new hanoi(2);

tower\_1.solve();

System.out.println("\n\n\n");

hanoi tower\_2 = new hanoi(3);

tower\_2.solve();

System.out.println("\n\n\n");

hanoi tower\_3 = new hanoi(4);

tower\_3.solve();

System.out.println("\n\n\n");

//problem 2

String[] w = new String[6];

w[0] = "1";

w[1] = null;

w[2] = "2";

w[3] = null;

w[4] = null;

w[5] = "3";

//w = [1, null, 2, null, null, 3]

traverse tr = new traverse(w);

String[] y = tr.inorder();

tr.print\_inorder();

String[] z = tr.preorder();

tr.print\_preorder();

String[] x = new String[7];

for (int i = 0; i < 7; i++)

{

x[i] = Integer.toString(i);

}

//x = [0, 1, 2, 3, 4, 5, 6]

tr = new traverse(x);

y = tr.inorder();

tr.print\_inorder();

z = tr.preorder();

tr.print\_preorder();

for (int i = 0; i < 7; i++)

{

if (i == 3 || i == 6)

{

x[i] = null;

}

else

{

x[i] = Integer.toString(i);

}

}

//x = [0, 1, 2, null, 4, 5, null]

tr = new traverse(x);

y = tr.inorder();

tr.print\_inorder();

z = tr.preorder();

tr.print\_preorder();

}

}

class hanoi

{

//I created a class so that I didn't have to manually write a stack class.

//I simply copied and pasted min\_stack from hw3 for simplicity.

min\_stack a, b, c; //note: min\_stack is bounded in size, bounding our size at 20

int size; //This holds the number of rings

int step; //This keeps track of the number of steps done in solve()

hanoi(int n) //This initializes the stacks and size, then populates a with n rings

{

a = new min\_stack();

b = new min\_stack();

c = new min\_stack();

size = n;

for(int i = 0; i < n; i++)

{

a.push(n - i);

}

}

void solve() //Call this to solve a hanoi object - helper method

{

//initialize step

step = 0;

//output the initial state of all stacks

System.out.println("Initial for n = " + size + ":");

output\_towers();

//Then call the actual method

solve(size, a, b, c);

}

//This is where the actual solving happens.

//num holds the number of rings, init holds the stack that you are moving

//rings from, goal holds the stack that you are moving rings to, and aux

//holds the other stack.

private void solve(int num, min\_stack init, min\_stack aux, min\_stack goal)

{

//base case: number of steps to solve tower of hanoi - 2^n - 1

//if n = 0, then 2^0 - 1 = 1 - 1 = 0.

//So 0 is our base case

if (num > 0)

{

//First, take all the rings on top of the num ring and move them to

//the aux stack.

solve(num - 1, init, goal, aux);

//Second, move the num ring to the goal stack.

move\_ring(init, goal);

//Last, take the rings you just moved to the aux stack and place them

//on top of the num ring in the goal stack.

solve(num - 1, aux, init, goal);

}

}

//This method moves a ring and outputs a lot of information on how it is

//being moved. The arguments are the starting stack and the ending stack.

void move\_ring(min\_stack s, min\_stack e)

{

//these just hold the names of stacks s and e.

char from, to;

//assign name to from

if (s == a)

{

from = 'a';

}

else if (s == b)

{

from = 'b';

}

else

{

from = 'c';

}

//assign name to to

if (e == a)

{

to = 'a';

}

else if (e == b)

{

to = 'b';

}

else

{

to = 'c';

}

//Output the step number (step), which ring is being moved (s.top()), and

//the starting and ending stacks names (from and to respectively)

System.out.println("Step " + (++step) + ": Move ring " + s.top() + " from peg "

+ from + " to peg " + to + ".");

//Actually move the ring from s to e.

e.push(s.pop());

//Then output all the stacks to show what happened.

output\_towers();

}

//This could have been included in move\_ring(), but I wrote it separately to

//check my code at any point. This just outputs all the stacks.

void output\_towers()

{

System.out.print("Peg a: ");

a.print\_stack();

System.out.print("\nPeg b: ");

b.print\_stack();

System.out.print("\nPeg c: ");

c.print\_stack();

System.out.println("\n");

}

}

class traverse //So this class takes care of all tree traversal - including nodes

{

String[] tree; //This stores the tree in the form of an array.

String[] inorder\_traverse; //This stores the inorder traverse in an array.

String[] preorder\_traverse; //This stores the preorder traverse in an array.

int iter\_i = 0; //This iterates through tree for the inorder traverse

int iter\_p = 0; //This iterates through tree for the preorder traverse

traverse(String[] x)

{

tree = fix\_notation(x); //Assign the tree to the corresponding array

inorder\_traverse = new String[tree.length]; //initialize size for the other arrays

preorder\_traverse = new String[tree.length];

}

String[] inorder() //Helper method for the tree traversal

{

return inorder(tree);

}

private String[] inorder(String[] x) //Here is the actual method

{

//base case: if tree is of length 1, add the element to inorder\_traverse

if (x.length == 1)

{

inorder\_traverse[iter\_i++] = x[0];

}

//Otherwise,

else

{

//Traverse the left subtree.

inorder(left\_subtree(x));

//Add the root to inorder\_traverse

inorder\_traverse[iter\_i++] = x[0];

//Traverse the right subtree.

inorder(right\_subtree(x));

}

return inorder\_traverse;

}

//preorder() is very similar to inorder()

String[] preorder() //helper method

{

return preorder(tree);

}

String[] preorder(String[] x)

{

//base case: tree is of length 1, so add to preorder\_traverse.

if (x.length == 1)

{

preorder\_traverse[iter\_p++] = x[0];

}

//otherwise,

else

{

//add the root to preorder\_traverse.

preorder\_traverse[iter\_p++] = x[0];

//traverse the left subtree

preorder(left\_subtree(x));

//traverse the right subtree

preorder(right\_subtree(x));

}

return preorder\_traverse;

}

//This method just changes the notation slightly to fit the class better

//essentially, it changes the input tree to size 2^n - 1 by filling extra

//space with null.

static String[] fix\_notation(String[] x)

{

int count = 1; //determines size of n for 2^n - 1

String[] result; //holds fixed tree

//count size that the array is supposed to be

while (Math.pow(2, count) - 1 < x.length)

{

count++;

}

//size of result

int new\_len = (int)Math.pow(2, count) - 1;

//if x is already in the correct notation, return it

if (new\_len == x.length)

{

return x;

}

//otherwise, initialize result to new\_len

result = new String[new\_len];

//copy x into result

for(int i = 0; i < x.length; i++)

{

result[i] = x[i];

}

//fill the rest of result with null

for (int i = x.length; i < new\_len; i++)

{

result[i] = null;

}

return result;

}

//This will take a tree as an argument and return the left subtree as an array

//This is kind of complicated to find how to skip the exact number of spaces

static String[] left\_subtree(String[] x)

{

int count = 1; //This counts how many elements were added to result

int skip = 1; //When count = skip, I skip this number of elements of x

int r\_i = 0; //This iterates through result

x = fix\_notation(x); //Fix the notation of x before doing anything else

//Then this holds the left subtree. It will always be this size, as

//x.length - 1 takes out the root, and half of the rest is the right subtree

String[] result = new String[(x.length - 1) / 2];

//iterate through x

for (int i = 1; i < x.length; i++)

{

//assign x[i] to result.

result[r\_i++] = x[i];

//When count = skip,

if (count == skip)

{

//skip this many places

i += skip;

//double size of skip (this is the pattern of how many we skip)

skip \*= 2;

//reinitialize count at 1

count = 1;

}

//Otherwise

else

{

//increment count

count++;

}

}

return result;

}

//I slightly modified left\_subtree() to get right\_subtree().

static String[] right\_subtree(String[] x)

{

//data members are the same as left\_subtree()

int count = 1;

int skip = 1;

int r\_i = 0;

x = fix\_notation(x);

String[] result = new String[(x.length - 1) / 2];

//iterate through x

for (int i = 1; i < x.length; i++)

{

//When count = skip,

if (count == skip)

{

i += skip - 1;

skip \*= 2;

count = 1;

}

else

{

result[r\_i++] = x[i];

count++;

}

}

return result;

}

void print\_inorder()

{

System.out.print("[ ");

for (int i = 0; i < inorder\_traverse.length; i++)

{

if(inorder\_traverse[i] != null)

{

System.out.print(inorder\_traverse[i] + " ");

}

}

System.out.println("]");

}

void print\_preorder()

{

System.out.print("[ ");

for (int i = 0; i < preorder\_traverse.length; i++)

{

if(preorder\_traverse[i] != null)

{

System.out.print(preorder\_traverse[i] + " ");

}

}

System.out.println("]");

}

}