ECE368 Project #4

Due Friday, April 11, 2014, 11:59pm

Description

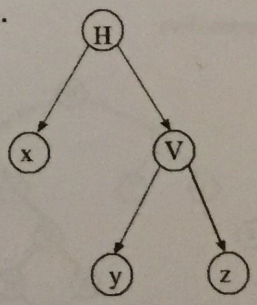
This project is to be completed on your own. In Project #3, you implemented a program involving tree traversal(s) to compute the packing of rectangles, represented by a binary tree. In that project, you assume that the given binary tree represents only one possible packing.

This project is a continuation of Project #3 in some sense. You will get a chance to earn some of the points you lost on the programming component, but not the report component, of Project #3. However, this project is also different from Project #3 in two ways. First, in Project #3, the connections among nodes are explicitly specified in the input file. In this project, you will be given the post-ordering of a strictly binary tree and you will have to develop an algorithm to construct the strictly binary tree. Second, in Project #3, you assume that the given binary tree represents only one possible packing. in this project, you will learn that for a given binary tree with n leaf nodes (i.e., n rectangles), it can simultaneously represents $2n-3$ possible packing solutions, one of which is the solution you computed in Project #3.

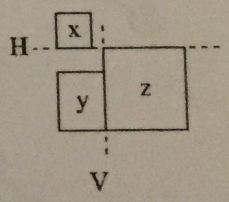
Let us consider the 3-rectangle example shown in Project #3 and redrawn here. Recall that the dimensions (width, height) of the three rectangles x, y, and z are $(3,3)$, $(4,5)$, and $(7,7)$. For the binary tree representation shown in (a), the post-order printing of the tree will give you the following:

```
(3.000000e+00,3.000000e+00)(4.000000e+00,5.000000e+00)(7.000000e+00,7.000000e+00)V
```

where each leaf node is represented by its dimensions in parentheses $(width, height)$, and each non-leaf node is represented by its cutline (`H` or `V`). The smallest room (shown in (b)) containing the three rectangles is of dimensions $(11,10)$.



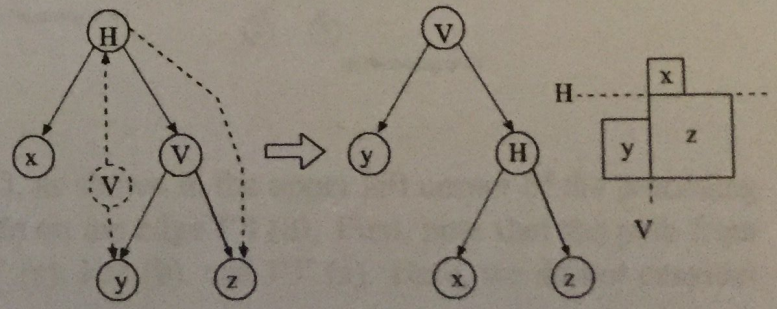
(a) A binary tree



(b) The corresponding packing

(c) shows another binary tree representation that can be obtained from the representation in (a). This representation in (c) is obtained by re-locating the root node of the tree in (a) on the edge Vy. We call the re-location of the root node as re-rooting. When we re-root Vy, y is kept as the left node of V, as in the original tree. The parent node of V would now be the right child node of V in the re-rooted representation, and the original right child node of V now becomes the right child node of H, the original parent node of V.

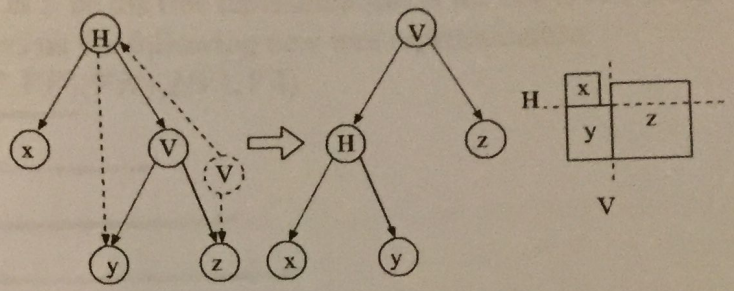
Essentially, we kept the Vy edge. Hence, we made the original parent node of V the new right child of V. As V is the



(c) Re-rooting at Vy

right child of the original parent node, we made the original right child of V the new right child of the original parent node.

(d) shows the binary tree representation obtained by re-rooting the edge Vz. Here, we kept the Vz edge. Therefore, we made the original parent node of V the new left child of V. As V is the right child of the original parent node, we made the original left child of V the new right child of the original parent node.

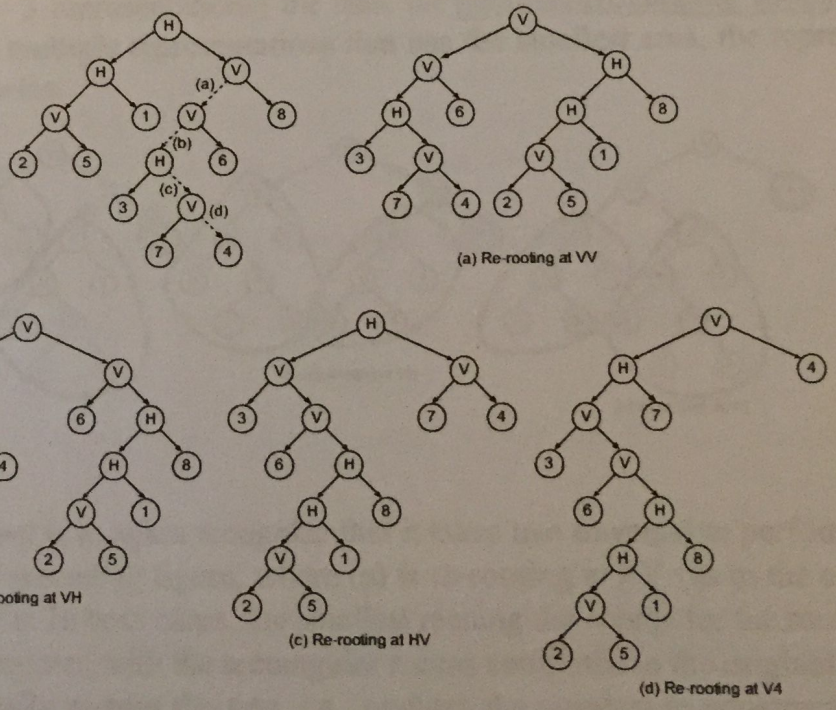


(d) Re-rooting at Vz

The representation in (c) still requires a room of dimensions $(11, 10)$ to pack all rectangles. However, the representation in (d) requires a smaller room of dimensions $(11, 8)$. In other words, the representation is (d) is more optimal than the representations in (a) and (c).

Note that while this re-rooting operation may look similar, it is actually *different* from the rotation operations that are used to balance the height of a binary search tree.

The preceding example demonstrates how you may re-root a strictly binary tree representation at edges that are separated from the original root node by just one edge. Now, we shall show you how to re-root at edges that are farther away from the original root node.



Consider the second example in Project #3, as shown in the upper left corner of the preceding figure. We want to re-position at the root node on the edge $V4$ (d). First, note that the path from the $V4$ to the root node includes the edge HV (c), VH (b), and VV (a). Here, we do not consider the right branch of the root node.

Let Re-Root(T,e) denote the new tree obtained by re-rooting at edge e of a given tree T, where e is one edge away from the root node of T. Let T be the tree representation in the upper left corner of the preceding figure. Re-rooting at $V4$ gives us the following new tree representation:

Re-Root(Re-Root(Re-Root(Re-Root(T,VV),VH),HV),$V4$)

(a)

(b)

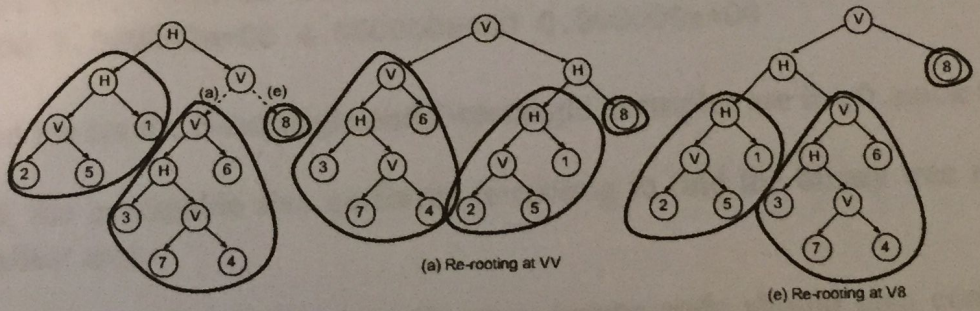
(c)

(d)

The binary tree in (a) is re-rooted to form the binary tree in (b), which is re-rooted to form the binary tree in (c), which is then re-rooted to form the binary tree in (d).

Given a strictly binary tree of n leaf nodes (i.e., rectangles), there are $2n-1$ nodes altogether. Therefore, there are $2n-2$ edges altogether. Among these edges, we do not re-root the two edges right below the root nodes (as the re-rooting operation cannot be applied when there is no parent node). Therefore, other than the given representation, there are $2n-4$ representation. Altogether, there are $2n-3$ strictly binary tree representations for a given strictly binary tree representation. Therefore, there are 3 representations in the first example and 13 representations in the second example.

Given a binary tree representation of n leaf nodes, you will write a program to find the representation (out of $2n-3$ representations) the uses the smallest rectangular room to enclose all rectangles. If there are multiple representations that use the smallest area, the representation that has the smallest width wins.



The key to this project is to again recognize that it takes tree traversal to perform the computation. Take a look at the preceding figure, where (a) is re-rooting at VV (as in the earlier example) and (e) is re-rooting at $V8$. In both cases, the smallest rectangular rooms for the root node V and its child node H can be computed with the rectangular rooms computed in the original tree on the left. It is not necessary to really re-root the tree, i.e., updates the pointers to construct different trees. What is again important is to figure out the necessary information to pass along as you traverse the tree.

Deliverables:

In this project, you are to develop your own include file `reroot.h` and source file `reroot.c`, which can be compiled with the following command:

```
gcc -Werror -Wbad-function-cast -Wall -Wshadow -O3 reroot.c -o proj4
```

All declarations and definition of data structures and functions must reside in the include file or source file. The executable `proj4` would be invoked as follows:

```
proj4 input_file output_file
```

As in Project #3, the executable loads the binary tree from `input_file`, performs packing, and saves the packing into `output_file`. *Note that the packing in the* ***`output_file`*** *should correspond to the given binary tree. In other words, this part is in a sense Project #3.* However, the formats of `input_file` and `output_file` are different from those in Project #3 as follows:

Input file format: The input file contains a single text line that is the output of a post-order printing of the nodes in a strictly binary tree. Each leaf node is printed in the form of (*width*, *height*) where both *width* and *height* are of type `double`. Each non-leaf node is represented by its cut-line, which is an ASCII character `'H'` or `'V'`.

Output file format: The output file contains a rectangle per line. Each rectangle is represented by its width, height, x-coordinate and y-coordinate, all of which of type `double`. The rectangles are ordered according to the post-order traversal of the tree. For the 3-rectangle example, the output is:

```
3.000000e+00 3.000000e+00 0.000000e+00 7.000000e+00
4.000000e+00 5.000000e+00 0.000000e+00 0.000000e+00
7.000000e+00 7.000000e+00 4.000000e+00 0.000000e+00
```

The output file for the 3-rectangle and 8-rectangle examples are in `r0.npck` and `r6.npck`.

Moreover, the executable also performs re-rooting to find the binary tree representation that gives the smallest area.

On top of the functions in Project #3, your source code should also contain the following function:

Perform re-rooting

Perform re-rooting on the binary tree you have loaded in, using data type double for your computation of dimensions. At the end of the analysis, the program should report the following:

```
Preorder:

Inorder:

Postorder:
```

```
Width:
Height:

X-coordinate:
Y-coordinate:

Elapsed time:

Best width:
Best height:

Elapsed time for re-rooting:
```

The printing of the preorder, inorder, and postorder should follow the format of the input file. The `Width` and `Height` should be the width and height of the smallest rectangular room containing all rectangles based on the packing specified by the input binary tree. The `X-coordinate` and `Y-coordinate` should be those of the *first triangle* in the post-order traversal. The elapsed time corresponds to the time it takes to compute one single packing solution.

For `Best width` and `Best height`, you should report the width and height of the smallest room that encloses the packing specified by the binary tree (out of the possible $2n-3$ representations for a given binary tree). The elapsed time in this part of the screen output refers to the user time required to compute the smallest room to pack all rectangles.

Both elapsed times do not include the time it takes to load the binary tree representation and to save the packing. The screen output of the 3-rectangle and 8-rectangle examples are available as `r0.log.proj4` and `r6.log.proj4`.

Grading:

The project requires the submission (through Blackboard) of the C-code (source and include files) and a report detailing the results. *Your report should focus only on the re-rooting part, i.e., the part not overlapping with Project #3.* There should be a table listing the run-times, and the outputs of the optimal packing (width, height) obtained from running your code on some sample input files. In your report, also comment on the run-time and space complexity of your re-rooting algorithm. Also comment on the data structures and algorithm used to construct a strictly binary tree from the given input, and the space and time complexity of your tree construction algorithm. Your report should not be longer than 1 page and should be in plain text format or pdf.

Getting started:

The sample input files in the new format are provided as a zip file. The output files and screen output for two of the examples (`r0` and `r6`) are also provided. Copy over the files from the Blackboard website. Check out the Blackboard website for any updates to these instructions.