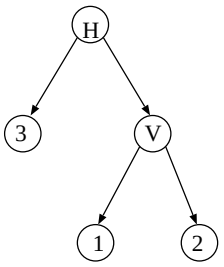
ECE368 Programming Assignment #3

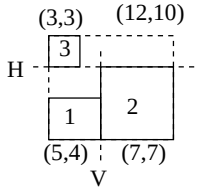
This assignment covers learning objective 1: An understanding of basic data structures, including stacks, queues, and trees; learning objective 5: An ability to design and implement appropriate data structures and algorithms for engineering applications.

In programming assignment #2 (PA2), you implemented a program involving tree traversal(s) to compute the "packing" of rectangles, represented by a strictly binary tree. In that assignment, you assume that the given binary tree represents only one possible packing. (As in PA2, we use the term packing loosely because we do not really pack the rectangular blocks tightly.)

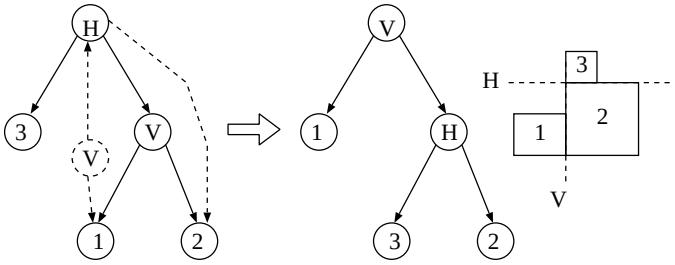
This assignment is a continuation of PA2 in some sense. However, this assignment is also different from PA2. In PA2, you assume that the given binary tree represents only one possible packing. In this assignment, you will learn that for a given binary tree with n leaf nodes (i.e., n rectangles), it can simultaneously represent $2n-3$ possible packing solutions, one of which is the solution you computed in PA2. The idea of using a single tree to represent multiple solutions has been used to more efficiently explore the solution space.



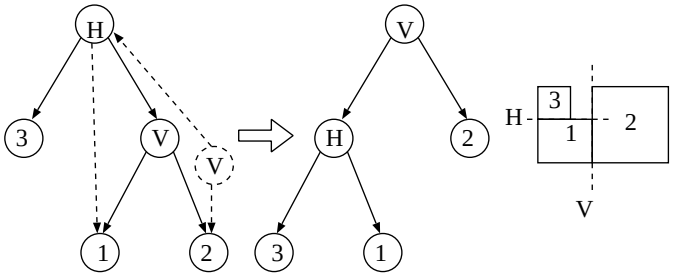
(a) A binary tree



(b) The corresponding packing



(c) Re-rooting at V1



(d) Re-rooting at V2

1 Re-rooting

Let us consider the 3-rectangle example shown in PA2 and redrawn here. Recall that the dimensions (width, height) of the three rectangles 1, 2, and 3 are $(4,5)$, $(7,7)$, and $(3,3)$, respectively. The smallest room (shown in (b)) containing the three rectangles is of dimensions $(12,10)$.

(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 $V1$. We call the re-location of the root node as re-rooting. When we re-root $V1$, 1 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.

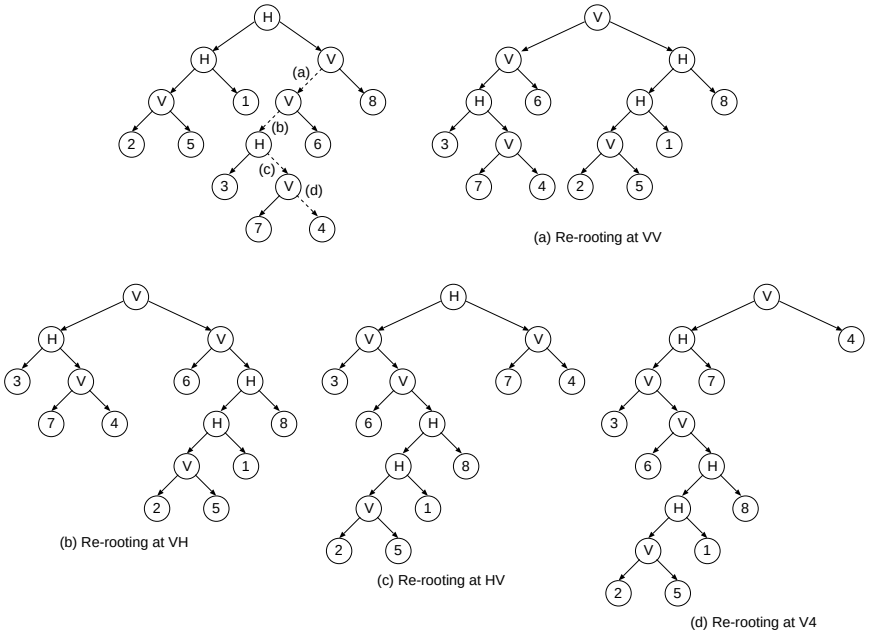
Essentially, we kept the $V1$ edge. Hence, we made the original parent node of V the new right child of V. As V is the right child of the original parent node, we made the original right child of V the new right child of the original parent node.

(d) shows the binary tree representation obtained by re-rooting the edge $V2$. Here, we kept the $V2$ 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.

The representation in (c) still requires a room of dimensions $(12, 10)$ to pack all rectangles. However, the representation in (d) requires a smaller room of dimensions $(12, 7)$. 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 PA2, 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. The following new tree representation corresponding to the operation of re-rooting at $V4$:

$$\texttt{Re-Root}(\texttt{Re-Root}(\texttt{Re-Root}(\underbrace{\texttt{Re-Root}(T,VV)}_{(a)},VH)_{(b)},HV)_{(c)},V4)_{(d)}$$

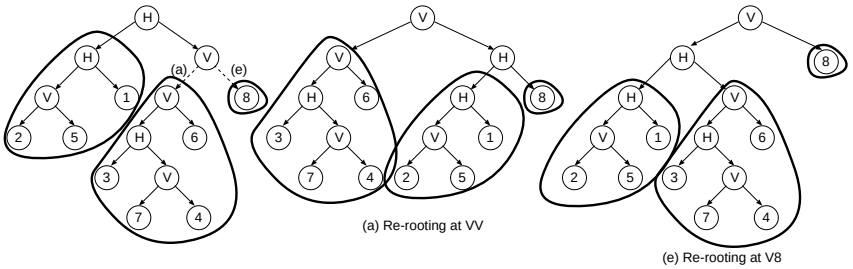
The binary tree in (a) is re-rooted to form the binary tree in (b), which is re-rooted to form the binary tree in (c), which is then re-rooted to form the binary tree in (d). In other words, to re-root at an edge, it is necessary to first re-root at the edges from the root node (except for the edge immediately after the root node) to the edge of interest.

Given a strictly binary tree of $n > 1$ 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 $2n-4$ representations that can be derived from re-rooting. 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. When there are $n = 2$ leaf nodes, there is only one representation available.

When there is $n = 1$ leaf node, there is only one representation available; no re-rooting is possible because there are no edges. (Of course, when the tree is empty, no re-rooting is possible.)

Since you may have already found the solution of the explicit representation of a given strictly binary tree, this assignment asks you to find the solutions of the representations that can be derived from re-rooting as defined earlier. Given a binary tree representation of $n \geq 1$ leaf nodes, you will write a program to find for each of the available re-rooted representations, the width and height of the smallest rectangular room to enclose all rectangles for that re-rooted representation.

Again, when $n = 1$ or 2, there are no re-rooted representations available. (Of course, when the tree is empty, no re-rooting is possible.) When $n > 2$, there are $2n-4$ re-rooted representations.



The key to this assignment 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.

Note that in the preceding figure, after re-rooting at VV, i.e., (a), edges in the left subtree of the re-rooted tree are valid for re-rooting. That is because all edges on the left (including VV) are from the original tree. The right "edge" at the root node, i.e., VH is not from the original tree. Therefore, all edges in the right subtree of the re-rooted tree are not valid for re-rooting.

Similarly, after re-rooting at $H8$, i.e., (e), only the right subtree of the re-rooted tree is valid for re-rooting. Since there are no edges in the right subtree, the re-rooting is complete for that subtree. The left subtree of the re-rooted tree is not valid for re-rooting.

2 Deliverables

In this assignment, you are to develop your own include files and source files, and these files should be compiled into an executable as follows:

```
gcc -O3 -std=c99 -Wall -Wshadow -Wvla -pedantic *.c -o pa3
```

If you supply a `Makefile` in your submission, we will use the command following command to generate the executable `pa3`.

```
make pa3
```

The executable `pa3` would be invoked as follows:

```
./pa3 in_file out_file1 out_file2 out_file3 out_file4
```

The executable loads the binary tree from `in_file` and produces four output files `out_file1`, `out_file2`, `out_file3`, and `out_file4`.

Among these four output files, the first three output files will account for the base score of 100 points, and the last output file will account for the bonus score of 50 points.

You should model your `main` function after the `main` function in PA2. Of course, you should make suitable modifications to account for the differences in PA2 and PA3.

The input file `in_file` is of the same format as the first output file of PA2. In other words, the input file is obtained by performing a post-order traversal of the strictly binary tree. Therefore, given an input file that contains the post-order traversal of a strictly binary tree, you have to re-construct the corresponding strictly binary tree. (In PA2, you have to re-construct the corresponding strictly binary tree when you are given its pre-order traversal.)

The first two output files are of the same format as the input file of PA2, a pre-order traversal of a strictly binary tree representing a packing.

Please refer to the description file of PA2 for a description of the formats of `in_file`, `out_file1`, and `out_file2`. Again, the format of `in_file` of PA3 is the same as the format of `out_file1` of PA2. The format of `out_file1` and `out_file2` of PA3 is the same as the format of `in_file` of PA2.

2.1 First output file

The strictly binary tree of the packing for the first output file `out_file1` is obtained as follows: Starting from the root node, you alternately visit the left and right edges down the tree until you come to a leaf node. The first output file should contain the strictly binary tree of the packing that is obtained by re-rooting at the last edge of this path.

Of course, to re-root at this last edge, it is necessary to perform re-rooting at corresponding edges along the way (except for the left edge of the root node). As a result, for the 3-rectangle example shown earlier, the corresponding re-rooted strictly binary tree is the same as the given strictly binary tree, as no re-rooting can be performed. On the other hand, for the 8-rectangle example, the given strictly binary tree should be re-rooted at the edge $H1$. The pre-order printing of the re-rooted strictly binary tree for the these two examples are in `3lr.pr` and `8lr.pr`.

2.2 Second output file

The strictly binary tree of the packing for the second output file `out_file2` is obtained as follows: Starting from the root node, you alternately visit the right and left edges down the tree until you come to a leaf node. The second output file should contain the strictly binary tree of the packing that is obtained by re-rooting at the last edge of this path.

Again, to re-root at this last edge, it is necessary to perform re-rooting at corresponding edges along the way (except for the right edge of the root node). As a result, for the 3-rectangle example shown earlier, the corresponding re-rooted strictly binary tree is obtained by re-rooting at the edge $V1$. For the 8-rectangle example, the given strictly binary tree should be re-rooted at the edge $V6$. Of course, to obtain this, we have to re-root at edge VV and then $V6$. The pre-order printing of the re-rooted strictly binary tree for the these two examples are in `3rl.pr` and `8rl.pr`.

2.3 Third output file

We now provide the details of the output file `out_file3`. `argv[4]` `out_file3` contains the name of the file that `pa3` would use to store the dimensions of the smallest rectangular room that encloses all rectangular blocks for each re-rooted representation.

The file is divided into lines, and each line corresponds to a node in the given strictly binary tree, and the nodes are printed in a pre-order traversal fashion.

Recall that the re-rooting is defined based on an edge in the given strictly binary tree. Therefore, each node, except the root node, can uniquely identify the edge that connects the node to its parent node in the given strictly binary tree.

There are therefore at most three nodes that each does not correspond to an edge that could be re-rooted. The root node does not have a parent edge. If the given strictly binary tree has $n \geq 2$ leaf nodes, the left child node or right child node also does not correspond to an edge that could be re-rooted. If such a node is a leaf node, which is a rectangular block, we print to the output file with the format

`"%d\n",`

where the `int` is the label of the rectangular block.

If it is a non-leaf node, we print a character (followed by a newline character):

`"%c\n"`.

The character is either `'V'` or `'H'`, representing either a vertical cutline or a horizontal cutline, respectively.

For the other lines in the output file, each of them corresponds to an edge (connecting the node to its parent node) that could represent a re-rooted topology. If the node is a leaf node, we print to the output file with the format

`"%d(%d,%d)\n"`,

where the first `int` is the label of the rectangular block, the second `int` and the third `int` are respectively the width and height of the smallest rectangular room enclosing all rectangular blocks for this re-rooted representation.

If the node is a non-leaf node, we print to the output file with the format

`"%c(%d,%d)\n"`,

where the first `char` is either `'V'` or `'H'` representing the cutline of the non-leaf node, and the two `int`'s are respectively the width and height of the smallest rectangular room enclosing all rectangular blocks for this re-rooted representation.

For the 3-rectangle example, the following is the expected third output file (`3.rdim`):

```
H
3
V
1(12,10)
2(12,7)
```

Note that the first three lines correspond to the root node, left of root node, and the right of root node, all of which do not have re-rooted representations.

For the 8-rectangle example, the following is the expected third output file (`8.rdim`):

```
H
H
V(11,15)
2(12,14)
5(14,15)
1(11,15)
V
V(13,11)
H(13,11)
3(13,14)
V(12,16)
7(13,16)
```

```
4(15,13)
6(13,11)
8(11,15)
```

The first, second, and the seventh lines correspond to the root node, left of root node, and right of root node, all of which do not have re-rooted representations.

2.4 Fourth output file

The fourth output file has the same format as the first and second output files. It contains the pre-order traversal of a strictly binary tree representing a packing.

Recall that when $n = 1$ or 2, there are no re-rooted representations available. (Of course, when the tree is empty, no re-rooting is possible.) When $n > 2$, there are $2n-4$ re-rooted representations.

When no re-rooting is possible, the fourth output file should contain the pre-order traversal of the given strictly binary tree.

Now, consider the case where re-rooted representations are available. Let A_0 be the area of the smallest rectangular room enclosing the given strictly binary tree. We index the $2n-4$ re-rooted representations 1 through $2n-4$, according to the order in which they appear in the third output file. For the i-th re-rooted representation, $1 \leq i \leq 2n-4$, let A_i be the area of the smallest rectangular room enclosing the re-rooted representation.

For the 3-rectangle example, $A_0 = 12 \times 10 = 120$, $A_1 = 12 \times 10 = 120$, and $A_2 = 12 \times 7 = 84$.

For the 8-rectangle example, $A_0 = 11 \times 15 = 165$, $A_1 = 11 \times 15 = 165$, $A_2 = 12 \times 14 = 168$, $A_3 = 14 \times 15 = 210$, $A_4 = 11 \times 15 = 165$, $A_5 = 13 \times 11 = 143$, $A_6 = 13 \times 11 = 143$, $A_7 = 13 \times 14 = 182$, $A_8 = 12 \times 16 = 192$, $A_9 = 13 \times 16 = 208$, $A_{10} = 15 \times 13 = 195$, $A_{11} = 13 \times 11 = 143$, and $A_{12} = 11 \times 15 = 165$.

If $A_0 \leq A_i$, for all $1 \leq i \leq 2n-4$, the fourth output file should contain the pre-order traversal of the given strictly binary tree. Otherwise, let $A_{\min} = \min_{1 \leq i \leq 2n-4} A_i$ and $i_{\min}$ be the smallest index such that $A_{\min} = A_{i_{\min}}$. The fourth output file should contain the pre-order traversal of the $i_{\min}$-th re-rooted representation.

For the 3-rectangle example, $A_{\min} = 84$ and $i_{\min} = 3$. The fourth output file (`3.opt`) should be as follows (see the bottom-most re-rooted representation in Page 1):

```
V
H
3(3,3)
1(5,4)
2(7,7)
```

For the 8-rectangle example, $A_{\min} = 143$ and $i_{\min} = 5$. The smallest rooms of the 5-th, the 6-th, and the 11-th re-rooted representations have the same area of 143, and the smallest index is 5. The fourth output file (`8.opt`) should be the pre-order traversal of the 5th re-rooted representation as follows:

```
V
V
H
3(3,3)
V
7(1,2)
4(3,5)
6(5,3)
H
H
V
2(1,3)
5(3,2)
1(2,4)
8(2,4)
```

See the top-right re-rooted representation in Page 2.

3 Submission

The assignment requires the submission (through Brightspace) of a zip file called `pa3.zip` that contains the source code (`.c` and `.h` files). You can create your zip file as follows:

```
zip pa3.zip *.c *.h
```

Your zip file should not contain a folder.

You may also include a `Makefile` in the zip file. In that case, you can create your zip as follows:

```
zip pa3.zip *.c *.h Makefile
```

4 Grading

The grade depends on the correctness of your program and the efficiency of your program.

The first output file accounts for 25 points, and the second output file accounts for 25 points, and the third output file accounts for 50 points of the entire grade. Any output files that do not follow the formats specified in this assignment will be considered to be wrong. These three output files account for the base score of 100 points.

The fourth output file accounts for the bonus score of 50 points.

It is important that your program can accept any legitimate filenames as input or output files. Even if you cannot produce all output files correctly, you should still write the main function such that it produces as many correct output files as possible. If you do not the algorithm to produce

the necessary information required to generate an output file, you should leave the output file as an empty file or not create the output file at all.

Your `main` function should be written to accept an input file and 4 output files, regardless of whether you are producing the fourth output file.

It is important all the files that have been opened are closed and all the memory that have been allocated are freed before the program exits. Any memory leaks or errors will result in 50% penalty.

5 What you are given

We provide the post-order traversals of the 3-rectangle and 8-rectangle examples in `3.po` and `8.po`, respectively. We also provide the four output files of the 3-rectangle (`3lr.pr`, `3rl.pr`, `3.rdim`, and `3.opt`, respectively) and 8-rectangle examples (`8lr.pr`, `8rl.pr`, `8.rdim`, and `8.opt`) respectively.

We also provide the post-order traversals of 100-rectangle, 500-rectangle, and 1000-rectangle examples in `100.po`, `500.po`, and `1K.po`, respectively.

6 Some hints

6.1 Calculating the area

While the width or height of the smallest room of a strictly binary tree can be stored in an `int`, the area may overflow an `int`. Please also note that multiplying two `int`s may result in overflow even if you store the result in a suitable data type. In other words, if we have the following code fragment:

```
int a, b;
...  // some assignments to a and b made
long c = a * b;
```

The variable `c` may not store the correct product of `a` and `b` because the multiplication is performed using `int`, which may cause overflow/underflow to occur.

6.2 Recursive function

Unlike PA2 where we would use test cases that may cause a recursive function to have a stack overflow problem, we will not do that in this assignment. It is most straightforward to use recursive functions (or recursive helper functions) to produce all the output files for this assignment.

While you may have to modify the tree structure in the function, you probably want to restore the tree to the original structure before you return from the function. The following recursive function may serve as a template.

```
recursive_function(tree T, other parameters)
{
   return if base case;
   ...
   // not base case
   T' = modify T;
   apply recursive_function(T', other parameters);
   restore to T;
   T'' = modify T;
   apply recursive_function(T'', other parameters);
   restore to T;
   ...
   return;
}
```

Note that it is not necessary for a recursive implementation to have a function that calls itself. I could have two functions `A` and `B`. `A` does not call itself, but calls `B`. Similarly, `B` does not call itself, but calls `A`. Together, they constitute a recursive implementation.