CSCI-1200 Data Structures — Spring 2022
Homework 8 — Ropes

In this assignment we will be implementing a partial and **modified** version of a `Rope`. Due to our modifications, other sources may not use the same terminology or may describe implementation details that are not relevant to our HW8 implementation - in fact even Wikipedia will differ on some specifics. For this reason you should avoid looking at code or online resources about ropes, and cannot use someone else's code in this assignment. Lecture/lab code is okay to use though. You should read the entire assignment before beginning your work. You should also make sure you understand the basic concepts discussed at the end of Lecture 18.

The idea behind a rope is that normally, inserting into and erasing from the middle of a string (character array) is very inefficient, for the same reasons that these operations are slow on an STL vector. Instead, we can represent a string inside a binary tree where each node has a weight (a non-negative integer) and a value (zero or more characters). Any node that is not a leaf will have an empty string as its value - only the leaves hold pieces of the string. The weight of a leaf is the length of its value. The weight of any non-leaf is the number of characters in the leaves of its left subtree.

Our assignment notably does not implement *operator--* for rope iterators, nor does it implement *erase()* or *insert()*. Much like an STL `map` or `set`, ropes are best implemented with a balanced tree. However, since we have not discussed balanced trees and this assignment will be large enough without such details, we will ignore the balanced tree requirement. The README has a section where you will discuss performance and the impact of not having a balanced tree on performance.

Provided Files

We have provided several files which contain further instructions. `main.cpp` contains some simple tests in *BasicTests()* for various functions you will need to write, and the expected output is shown in `output_basic.txt`. Your own additional testing should be added to *StudentTests()*. We have also given you *Rope.h* which you can modify as you see fit, *Rope_student.cpp* which has the functions you need to write, and *Rope_provided.cpp* which you should not change at all. We will use our own copy of *Rope_provided.cpp* when testing on Submitty. You can add additional functions to the .h file and *Rope_student.cpp* - we will always use your version of these two files. *Pay particular attention to the comments in Rope_student.cpp*!

Hints

You may find it beneficial to borrow code from our partial implementation of the `ds_set` class and associated helper classes. Focus on writing the destructor first and then you can start working through *BasicTests()* step by step. Carefully consider whether it makes more sense to do a function iteratively or recursively. Keep in mind for *index()* you must write an iterative version. You may want to add some private helper functions to the `Rope` class. The most challenging function will be *split()*, and breaking the work up can make debugging easier. You might also want to call the provided print functions a lot when debugging to quickly visualize your rope.

Submission

Submit your *main.cpp*, *Rope.h*, *Rope_student.cpp*, and *README.txt*. Dr. Memory will be used on this assignment and you will be penalized for leaks and memory errors in your solution. If you get at least 3 points on test case 13 by the end of Wednesday, you can submit on Friday without being charged a late day. Please remember that all submissions are still due by the end of Saturday.

concat()

concat() adds the right-hand (rhs) rope to the left-hand (lhs) rope by creating a new root. In our case, this is written as `lhs.concat(rhs)`.

new root

left root

right root

9

6   2

6   3   2   4

Hello_   my_   na   me_i
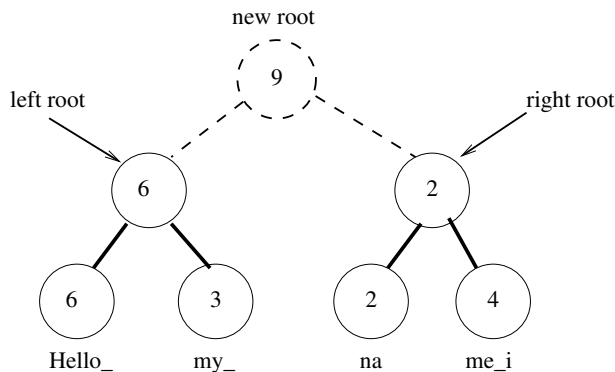
Starting with two Ropes, *left* and *right* we do the following: (Keep in mind we may need to copy the right subtree, look in the provided code for clues to the correct behavior for our case!)

1. Create a new node which will be the new root

2. Connect it to the root of the left rope and update weight

3. Connect it to the root of the right rope

4. Set the new node to be the root

index()

`index(i)` for Ropes works like `operator[i]` would for a string or vector, it returns the ith character, with the first character being at $i = 0$.

The blue arrows/text show the search and how i changes for $i = 9$.

22

9

9   0

6   6

6   3   2   1

Hello_   my_   0

2   4   1   6

na   me_i   s   _Simon

[0]
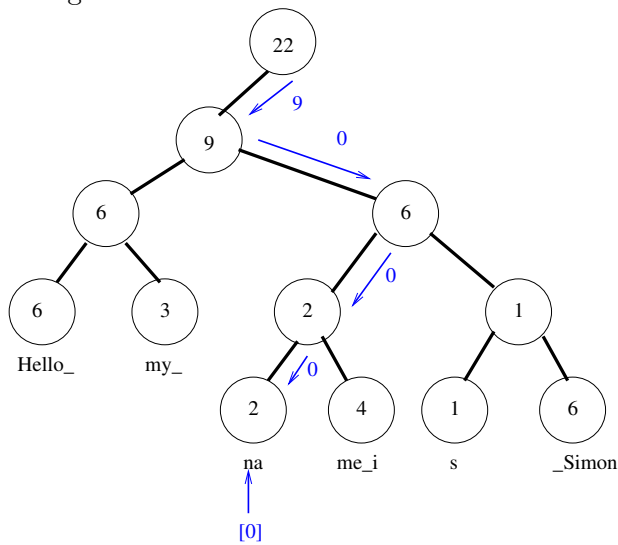
To find the character at index $i = 9$ (assuming we start at index 0):

1. Start at the root and decide to go left since the root's weight is 22 meaning the left subtree has indices 0-21

2. Next, decide to go right since the current node's weight is 9, meaning the left subtree has indices 0-8. Subtract the root's weight (9) from i, so now $i = 0$. (Anytime we go right we must subtract the weight of the current node. Think about why.)

3. Since $i = 0$ we will always go left. A search will continue making choices to go left or right until it hits a leaf. (You can consider why this is always correct.)

4. At the leaf, return the character at index i, in our case that's `"na"[0]` which is 'n'.

See next page for split() details
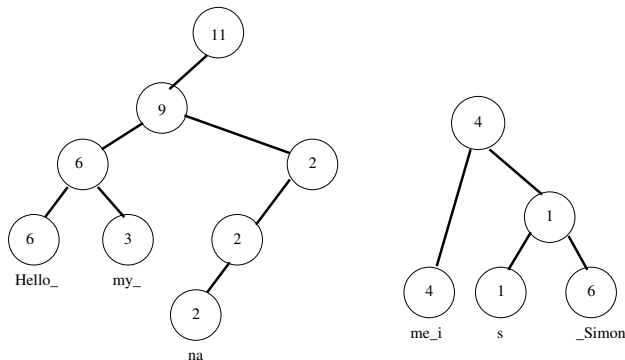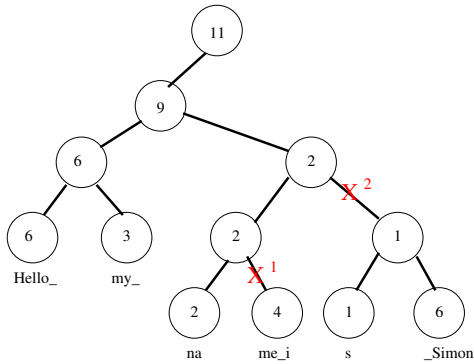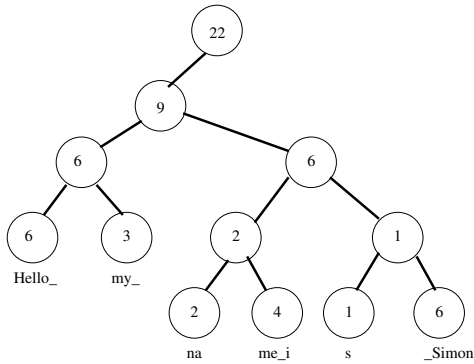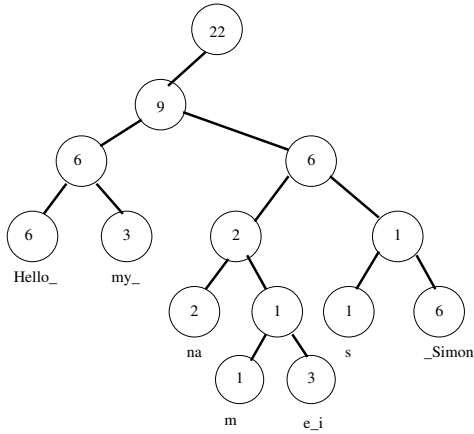
split()

To split, we first find the index we want to split at. Remember that for a split at index i, indices 0 through (i-1) stay in the left rope, and everything else goes to the right.

If i is in the middle of a node's value, then we have to cut the node into two pieces. For example, if we chose $i = 12$, then we would want the "e" in "me i" to be the first part of the new righthand rope. But this is in the middle of a leaf. To fix this, give the leaf two new children, a left child with "m", and a right child with "e i", and then update weights accordingly. We can then use the new "e i" leaf to start our actual split.

In the lower example, we consider $i = 11$ so that we do not have to deal with this.

1. First we find $i = 11$ which is the first letter in "me i". Following the rules of *index()*, i will have been adjusted to be 0. So the leaf will not need to be split.

2. Next, we disconnect the leaf from the original (lefthand) rope.

3. Going up the rope, for any node where the right subtree will completely be in the new (righthand) rope, we also disconnect the right child from the original rope.

4. Going from left to right we concatenate the root of each subtree that we disconnected, since they will all be small ropes. In this example we split the X^1 and then X^2, so we just *concat_no_copy(X^1, X^2)*. (This is different from the *concat()* described earlier in that we never want to copy the right-hand Rope, we always want to just adjust pointers when using it to help with *split()*!

5. Finally, update the weights of all nodes (unless you did this when you were disconnecting/concatenating).

3