CSC 325 Adv Data Structures

Lecture 9

Ropes

Using StringBuilder is one way of dealing with a large number of strings. You could also use the very similar StringBuffer in Java. However, there is a data structure that could be better than either of those for concatenating, deleting, and overall managing large strings.

Your text editor (or word processor) has to deal with very large strings. You can think of your entire code file as one big string (with newlines in it). You already learned with the "Tries" lecture that manipulating immutable strings can be a very expensive task. You basically have to create a new string, copy over the old characters, and insert the new stuff. While StringBuilder and StringBuffer can help, you can also use a Rope!

The rope data structure allows you to store strings in an unusual way, that ends up being very fast to manipulate. I'm not going to go into a lot of depth on this one. If you look online, there isn't much on this data structure, however it is my understanding that manipulating very large strings is commonplace for text editors and ropes is how they do it! So it's worth a look at how this works.

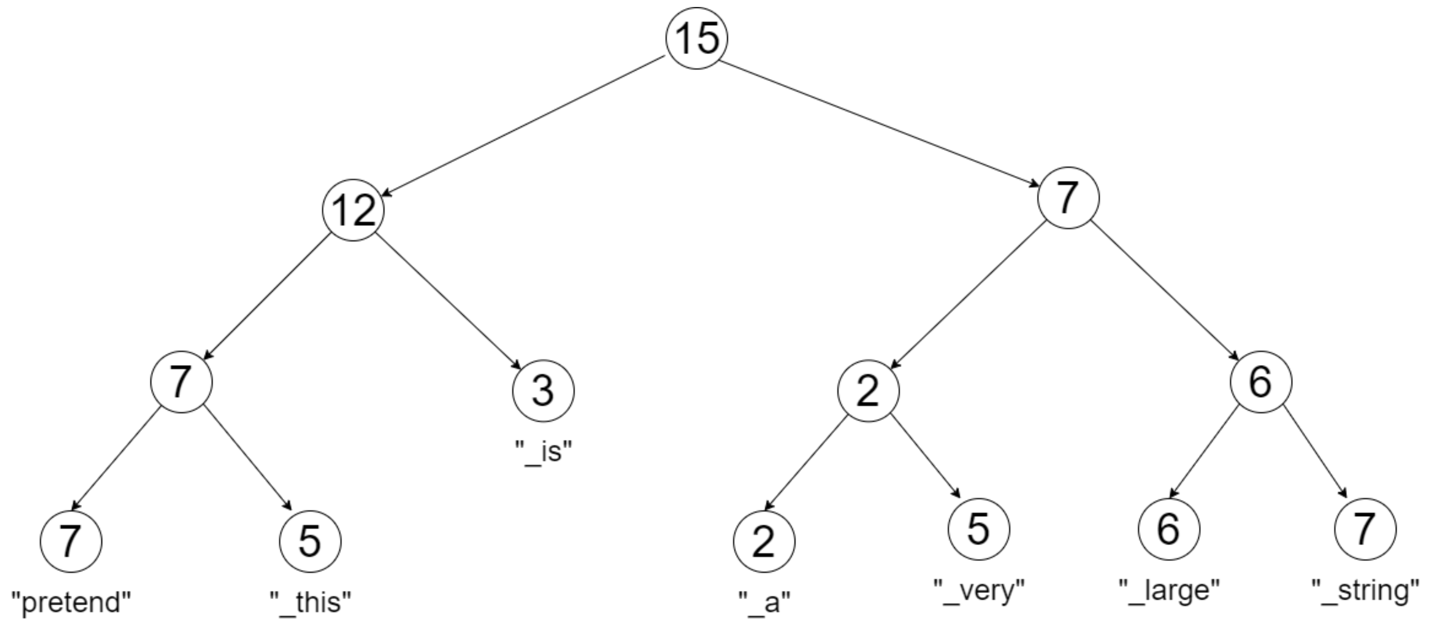
Imagine you have the following code:

```
String largeStr = "Pretend_this_is_a_very_large_string";
```

The normal way to store this is for you to use the "String" datatype built into Java in which case this is essentially an array of characters, filled with the characters of the string:

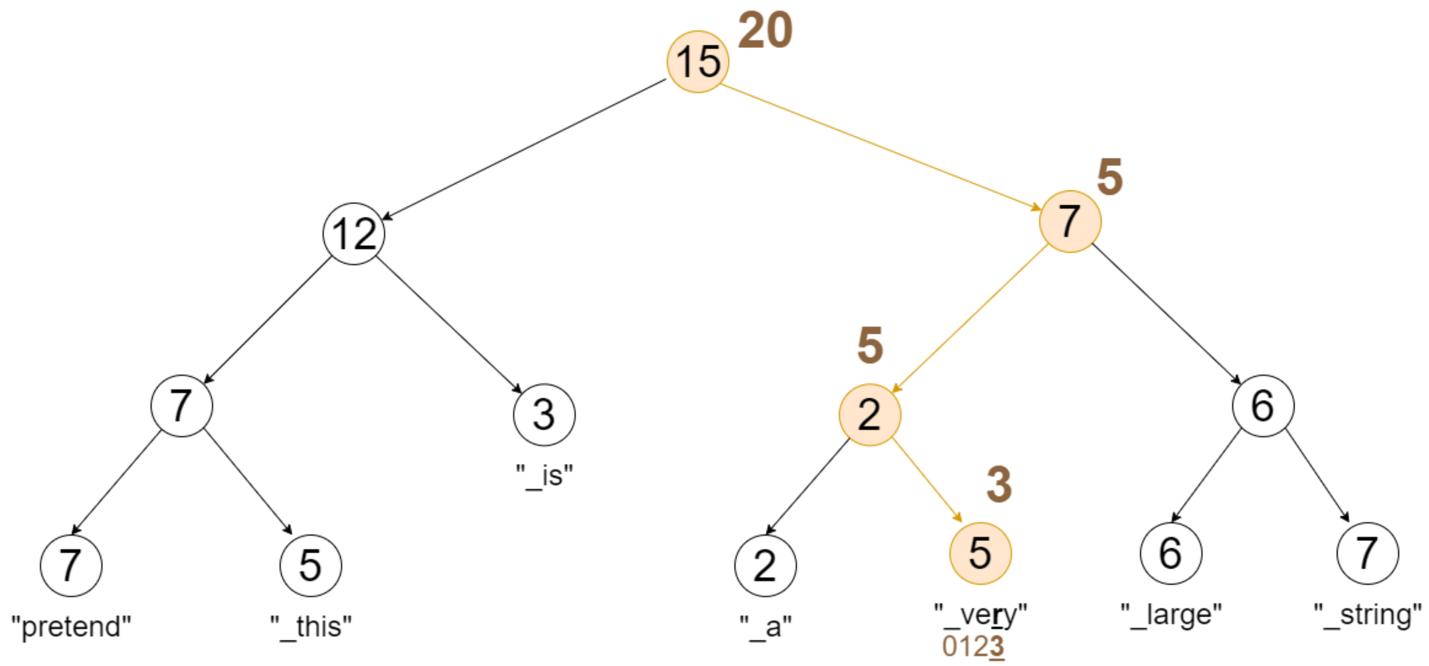
P	r	e	t	e	n	d	_	t	h	i	s	_	i	s	_	a	_	v	e	r	y	_	l	a	r	g	e	_	s	t	r	i	n	g

To modify, delete, add, or otherwise change this string (for example, adding more characters to the middle, or removing a word) will cause Java to build an entirely new string. Instead, let's put this text into a rope. I'll show you what that could look like and then we will discuss:



Indexing

First let's look at how we would perform the "charAt" function from Java. This function retrieves a character at the given index. Given an index, such as 20, how do we determine which character that is:



Finding a character at a given index is similar to finding a value in a BST. If the number is less, you go left. If the number is greater than or equal to, you go right. However, in a rope, if you go right, you subtract the node number from the number you are looking at. So, if I'm looking for the character at index 20, I first go to the root. Since the root has 15 and 20 is greater, I move right, but I also subtract the node value of 15. So now 20 becomes 5. Now I examine the right child node which contains 7. Since 5 is less than 7, I go left. I examine 2 and since 5 is greater than 2, I go right and subtract the node value of 2. So 5 becomes 3. Now that I'm at a leaf node, I examine the string held in that node. I find the character at index 3, which is "r". Is this correct? Let's check:

P	r	e	t	e	n	d	_	t	h	i	s	_	i	s	_	a	_	v	e	**r**	y	_	l	a	r	g	e	_	s	t	r	i	n	g
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	**20**	21	22	23	24	25	26	27	28	29	30	31	32	33	34

Yep, that's correct! Think of other numbers from 0 to 34 and double check that you can navigate to the correct character from the rope structure above.

Node Values

I went over indexing first so you can get a feel for what the numbers actually represent. Basically, the numbers in a leaf node tell you how many characters are in the string that the leaf node contains. The numbers in a parent node show you the number of total characters in the left subtree. For example, the parent node containing 6 (right part of the tree above), has the value 6 because the left subtree (which is just the leaf node containing "_large") has 6 characters. The parent of 6 which contains a value of 7, has this value because there are 7 total characters in its left subtree (2 characters from "_a" and 5 characters from "_very"). The reason why parent nodes contain the number of total characters in the left subtree is because if you go right while searching for an index, this is the number of characters you would be passing up. And so we subtract that number from the search index. Essentially if you are looking for index 8 and you know there are 5 characters you just passed up (because the node value was 5), then you are now looking for 3 characters past that. However, if you are looking for the character at index 4, then seeing a value of 5 would mean that the index 4 character is within that group, so go left. Each time you go right you are skipping several characters in the total string in your search for the given index. Each time you go left you are committing to searching that group of characters since your search index is within that group. This makes searching for a character at a given index pretty fast. However, just storing all characters in a single array makes searching for an index even faster, so why bother to do all this? What is the benefit?

Concatenating

Concatenating two strings is a costly operation. You need to build a new string, copy over contents of both given strings, and then destroy the given strings:

String 1

~~H~~	~~e~~	~~l~~	~~l~~	~~o~~

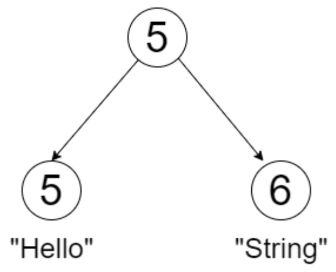
String 2

~~S~~	~~t~~	~~r~~	~~i~~	~~n~~	~~g~~

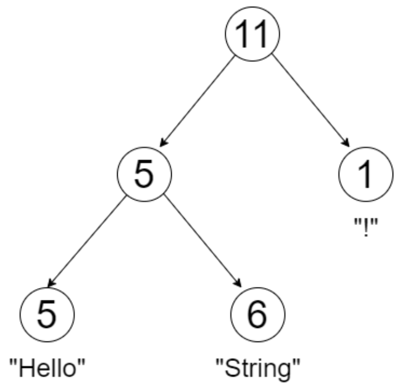
String 1 and 2

H	e	l	l	o	S	t	r	i	n	g

However, with a rope, we can preserve both string 1 and 2 and just combine them with a new root node:

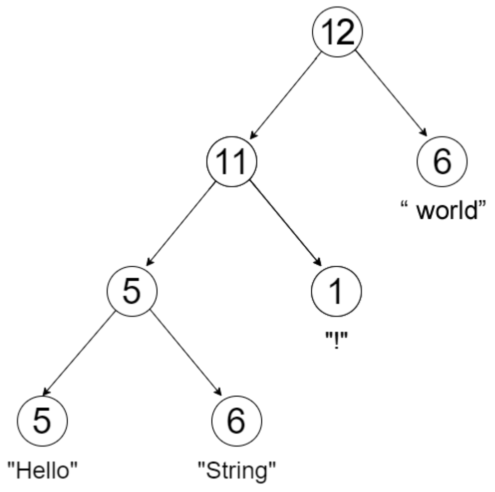


This is a fast operation since we preserve the node containing "Hello" and the node containing "String" and just add a new root node to combine them. Let's now concatenate the string "!":



If we ever want the entire string, we can simply do an in-order traversal of the rope to get out "HelloString!"

Note that if we continue to append strings, our rope will be quite lopsided. A rope doesn't have a mechanism to remain balanced, so this can happen. If we append the string " world" to our rope, we will get the following:

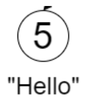


Building

To help understand how these things are built, let's consider the rope above. Let's say that Java converted String objects to ropes behind the scenes. What I mean is, let's pretend that when Java sees this code:

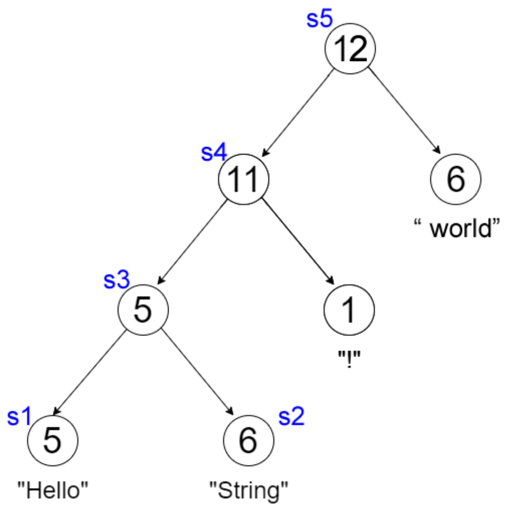
```
String s1 = "Hello";
```

It makes a rope like the following:



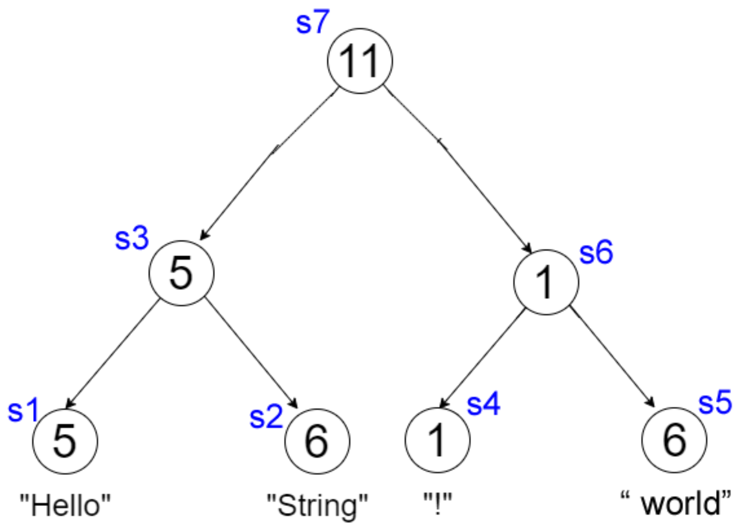
And then assigns s1 to this rope. This is a rope with only a single node, which is its root. If Java did this, then to get the full rope above, we would need to do the following:

```
String s1 = "Hello";
String s2 = "String";
String s3 = s1 + s2;
String s4 = s3 + "!";
String s5 = s4 + " world";
```



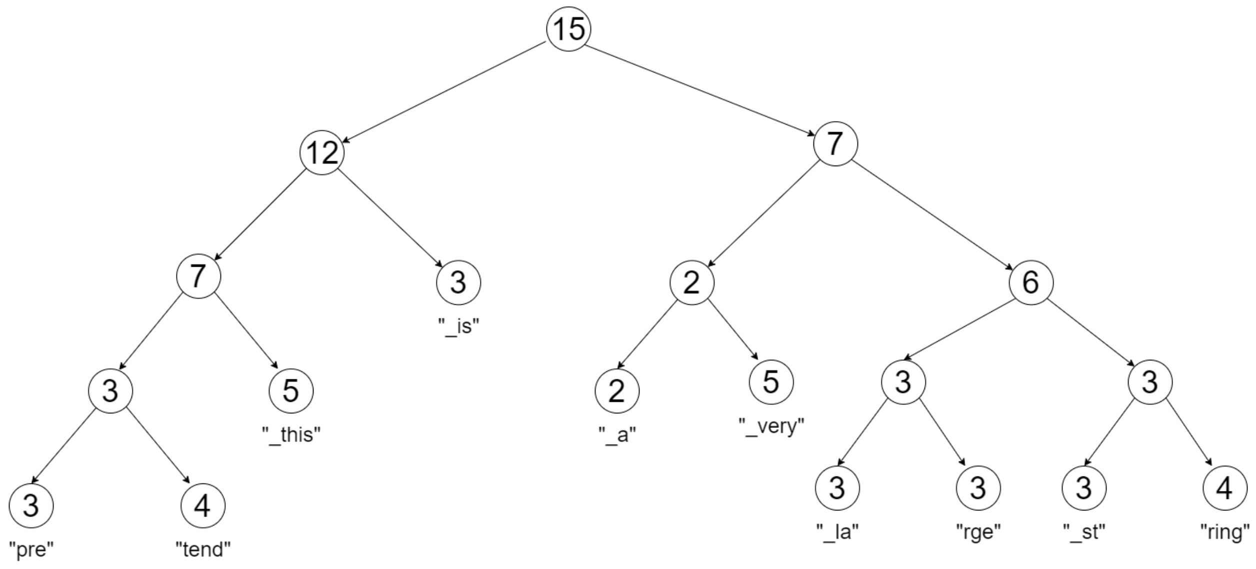
If we wanted the rope to be more balanced, we could construct it as follows:

```
String s1 = "Hello";
String s2 = "String";
String s3 = s1 + s2;

String s4 = "!";
String s5 = " world";
String s6 = s4 + s5;

String s7 = s3 + s6;
```



Chunking

Note that you don’t have to store entire words in the leaf nodes, you could also put in a size limit. So, if the developer wanted to store a really long word, this word could be chopped into pieces (which programmers normally refer to as “chunks”). For example, the original rope at the beginning of the lecture could have been done with a size limit of 5:



Note that we did NOT apply a max limit to the other ropes in this lecture, so only this rope has had the max limit applied.

An in-order traversal would still reveal the entire string correctly (i.e. "Pretend_this_is_a_very_large_string"), however now each leaf node holds a max of 5 characters. This makes for a bigger tree but gives more granular control over inserting characters into whichever position you like in the string.

There is more we can uncover with ropes, such as how to insert values in the middle, how to delete characters and other operations. I think this is enough to give you a general idea of how this structure works, though if you are curious, I highly recommend looking into all the things you can do with ropes. As I said in the beginning, the knowledge out there is scarce on this data structure, so you may or may not find a lot of info on this. Nevertheless, ropes are a fun example of another data structure for string manipulation.

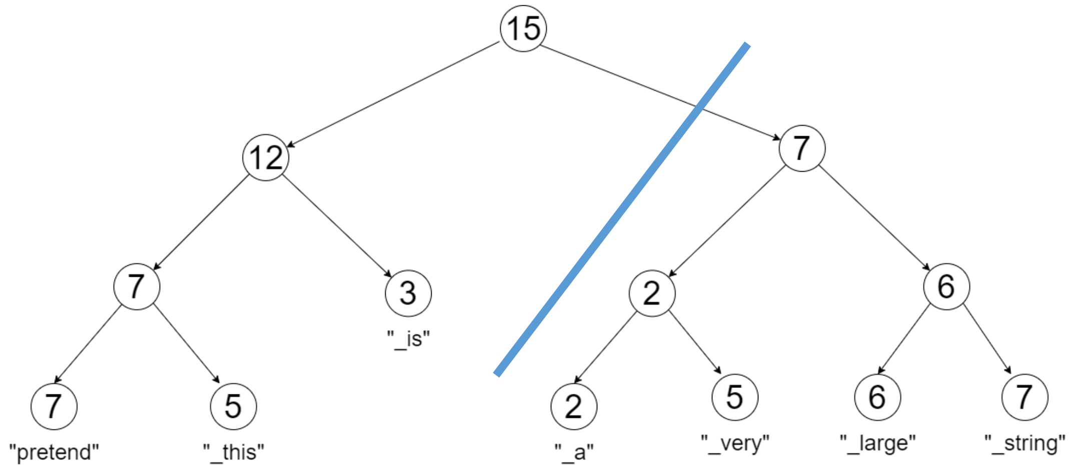
Inserting

If we want to insert text at the beginning or end of our rope, it's a simple concatenation. But what about inserting text somewhere in the middle? To accomplish this, we must split the rope into two ropes (a left and right rope), concatenate our text to the left rope, then concatenate the ropes back together. First we must locate the split point in our rope.

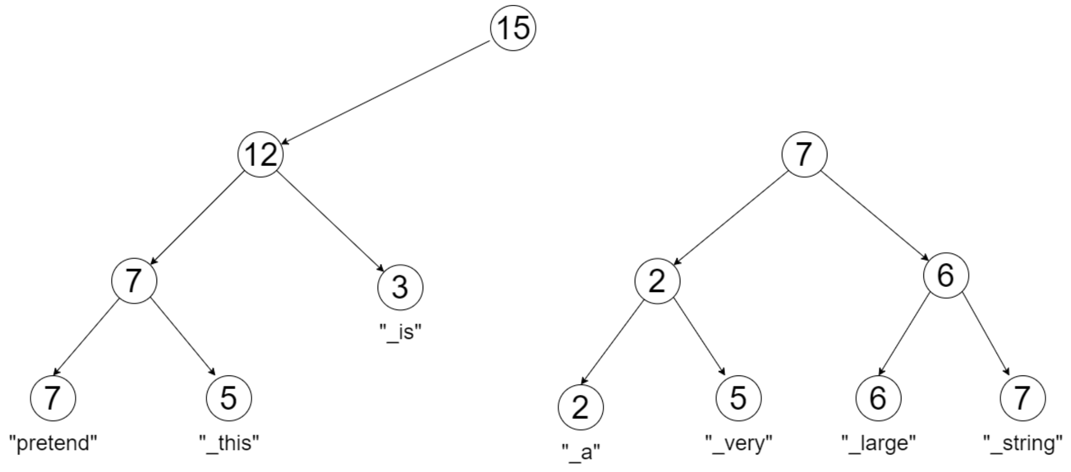
If our text is the text from before and our cursor is positioned somewhere in between:

```
Pretend_this_is|_a_very_large_string
```

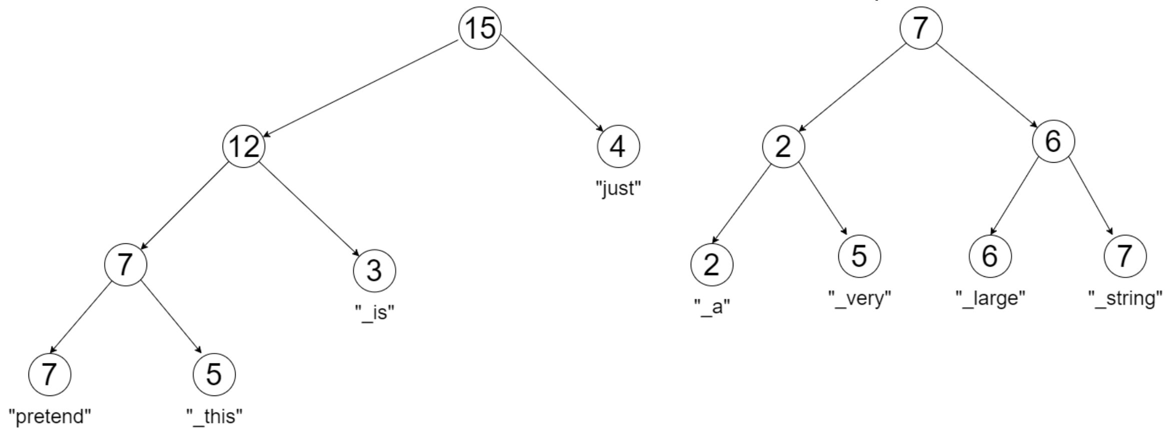
The cursor is right before index 15, so first we find index 15 and split the rope right before it:



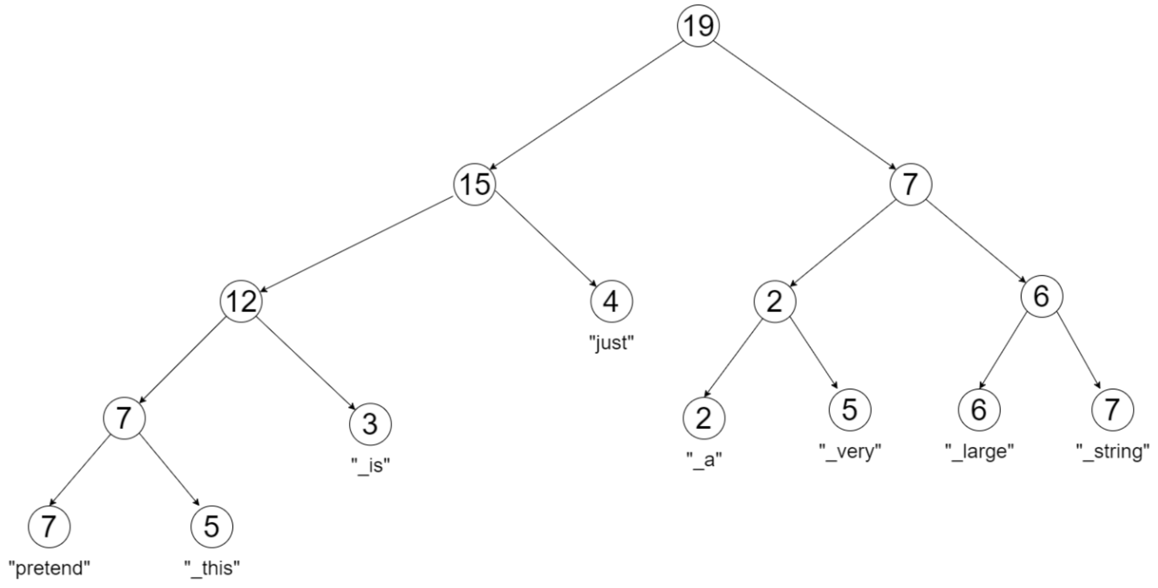
This gives us two ropes:



We can now concatenate the text we want to insert into the left rope:



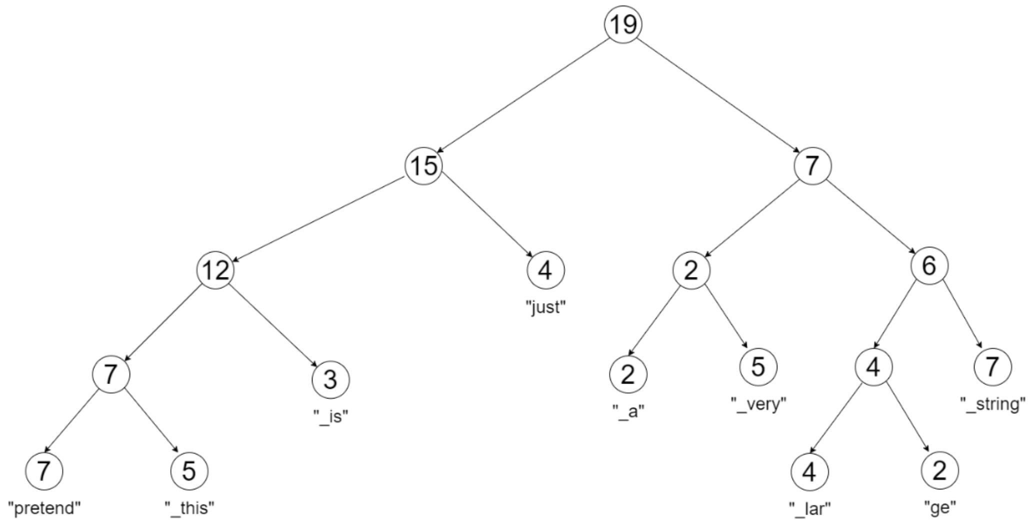
Then we simply concatenate both ropes back together:



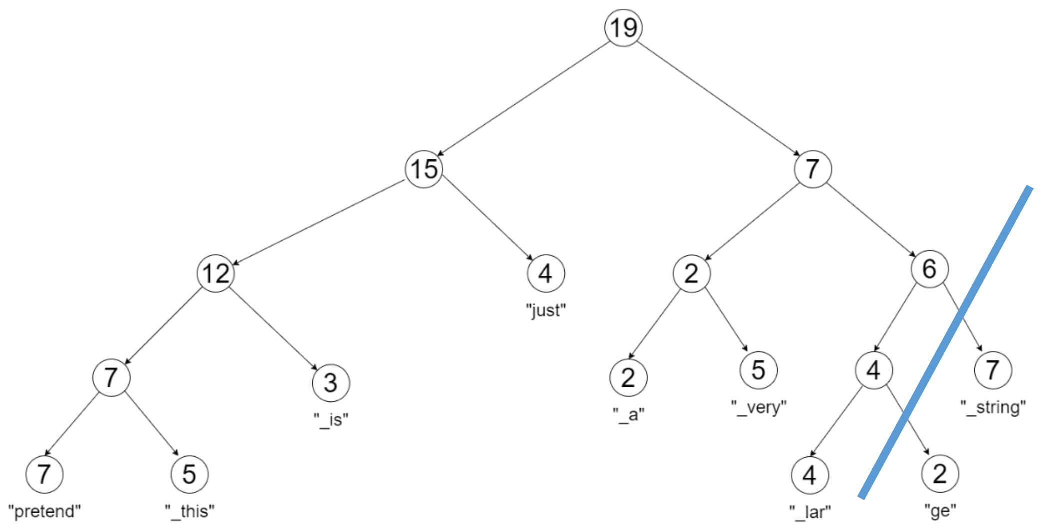
What about if the insertion point is in the middle of a leaf node string?

```
Pretend_this_is_a_very_lar|ge_string
```

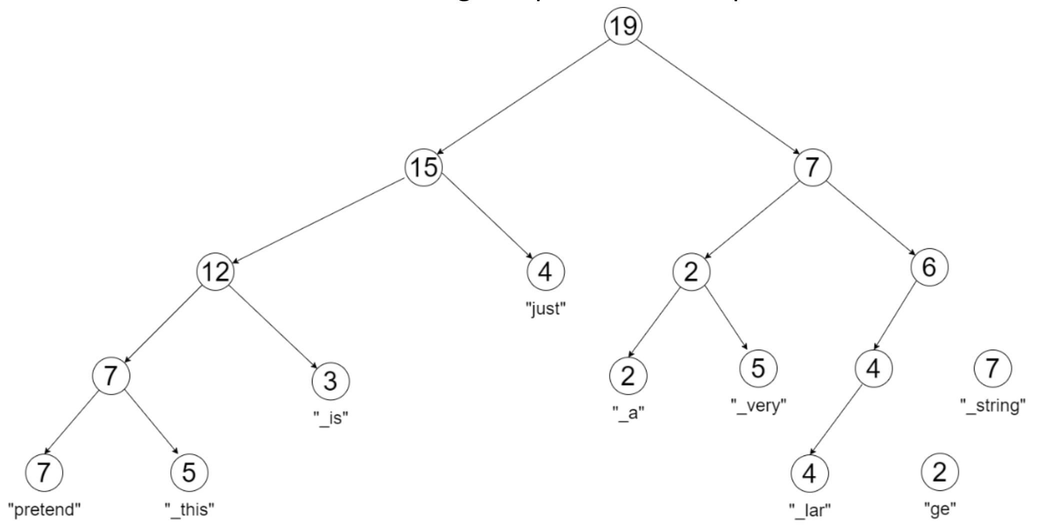
In this case we first must split the node containing "_large":



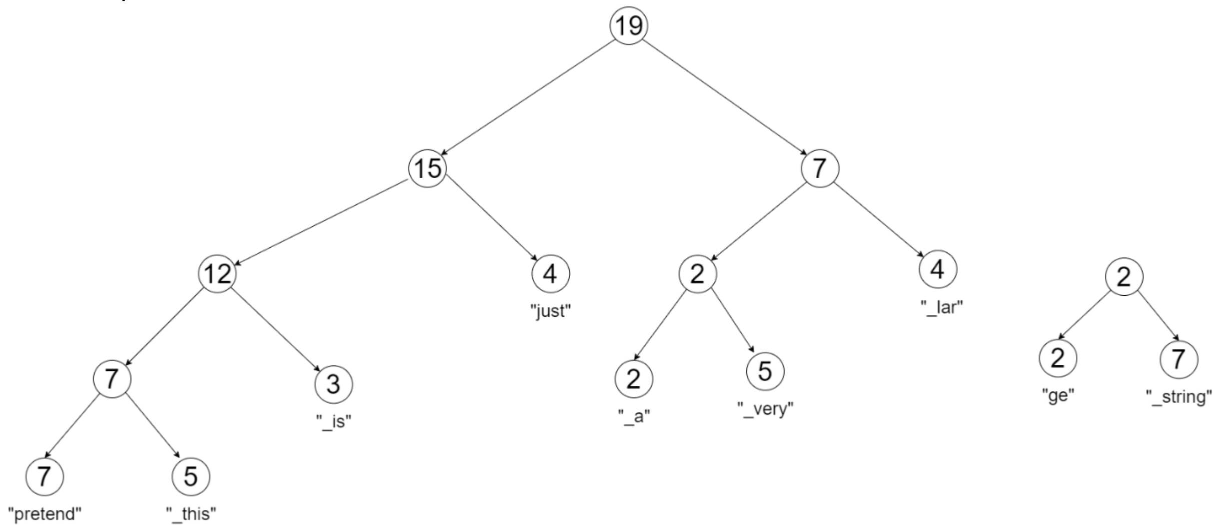
Then we can split the tree right before the "ge"



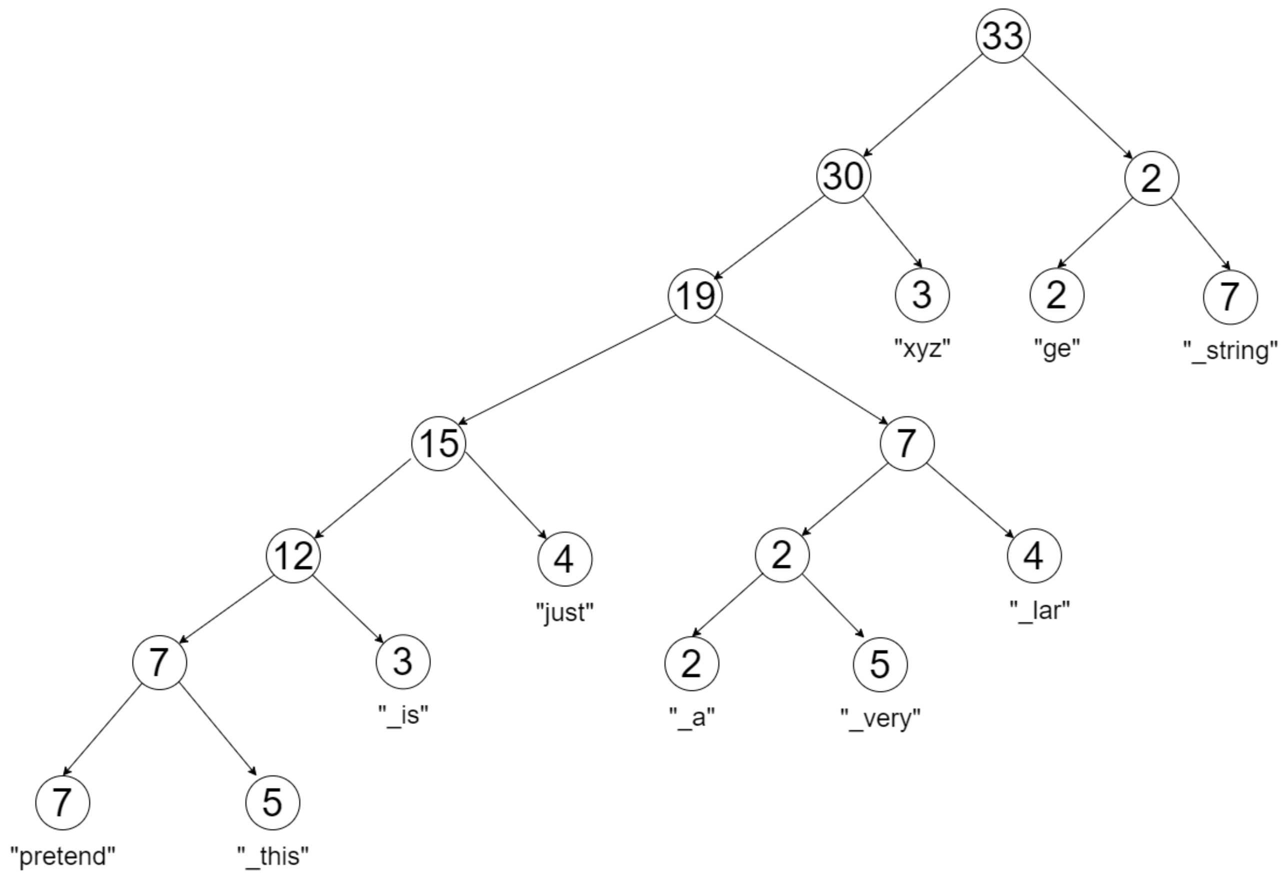
In this case we arrive at a broken right rope and a left rope that needs some fixing up:



When this happens, we need to mend together the pieces to form a rope on the right and then rebalance to fix the rope on the left:



Now we can concatenate our inserted text to the left rope and join the ropes back together:

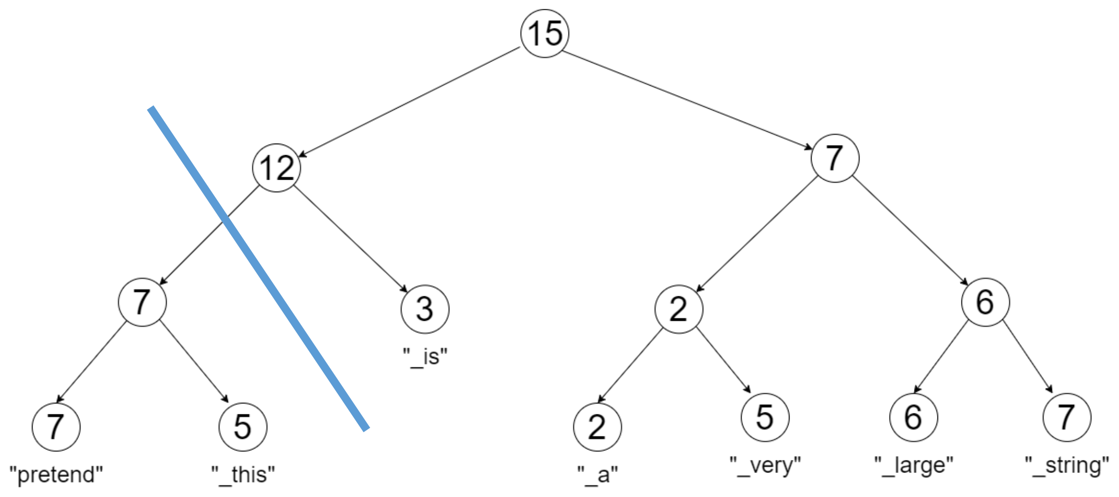


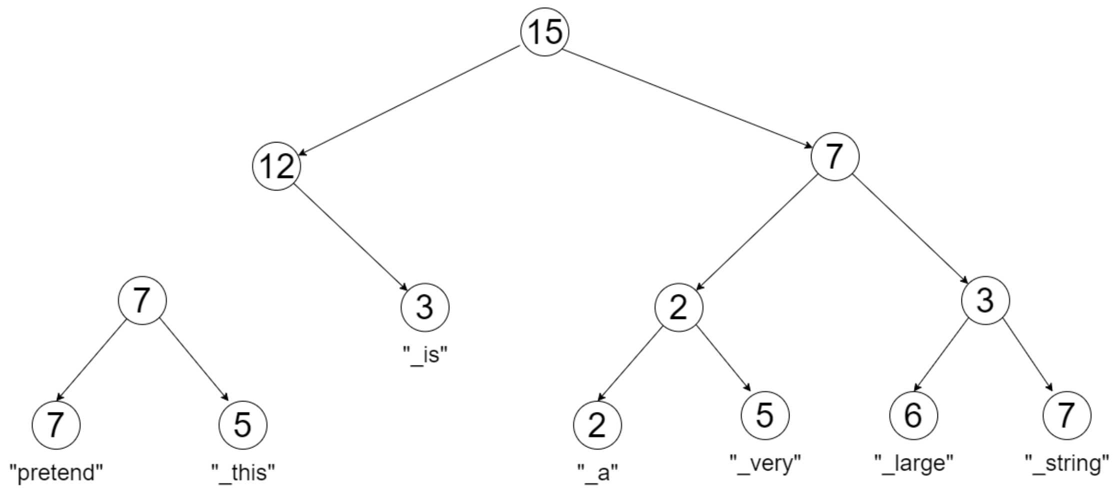
Deleting

Once you understand the concept of splitting the tree, deleting becomes straightforward. For example, to delete the following part of our text:

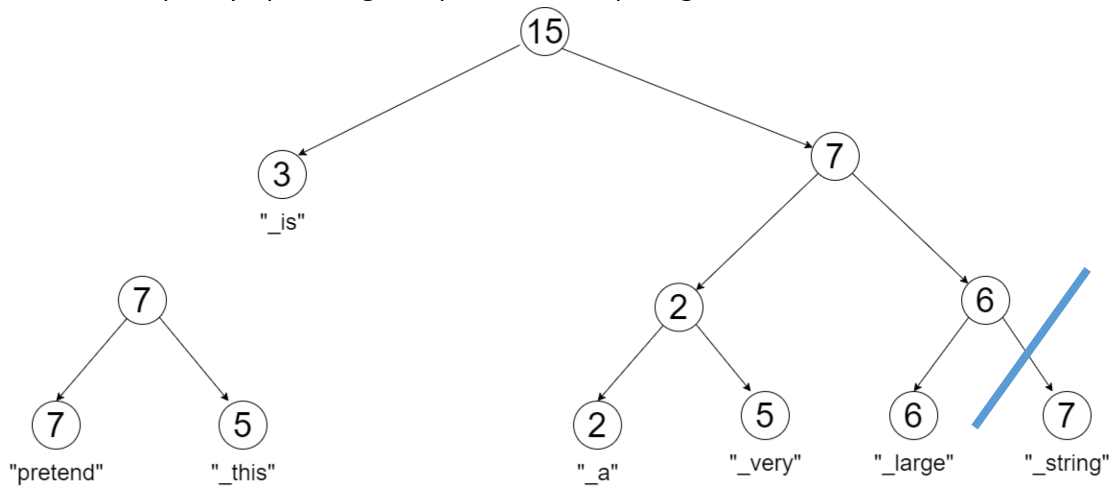
`Pretend_this` **|** `_is_a_very_large` **|** `_string`

We will need to split our rope at the starting index and again at the ending index. This will give us 3 pieces. We then remove the middle piece and concatenate the leftmost and rightmost pieces. So, we make our first split:

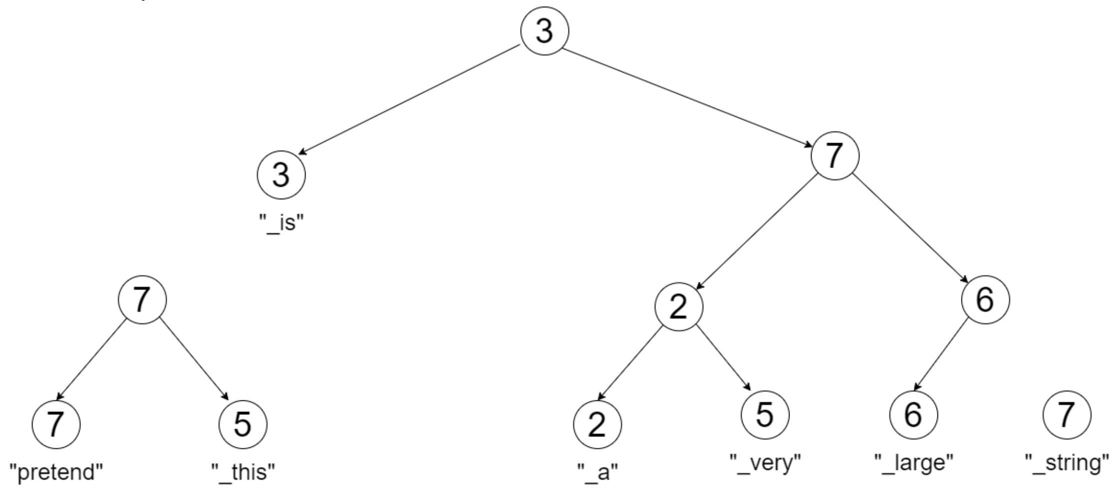




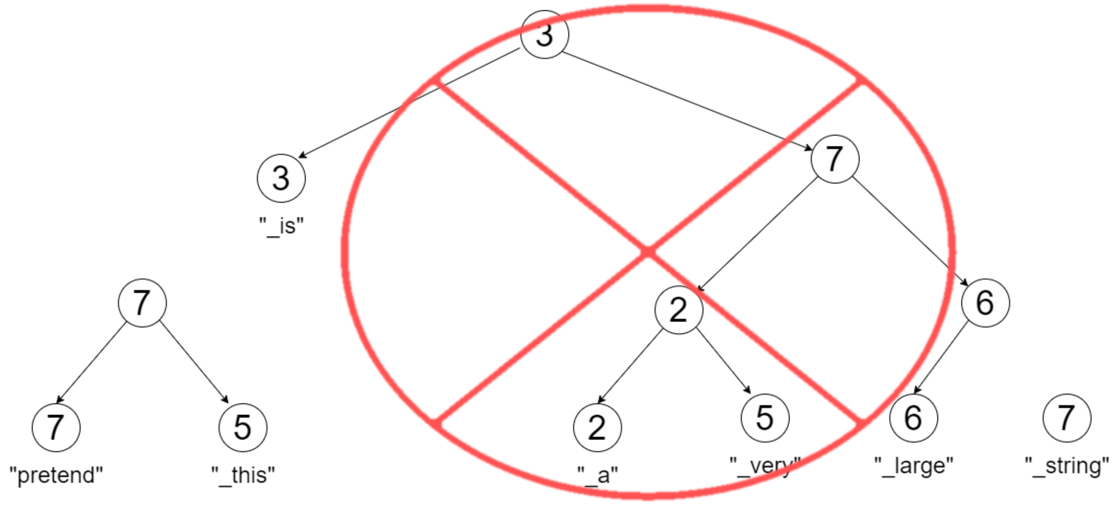
We need to pretty up our right rope and then split again:

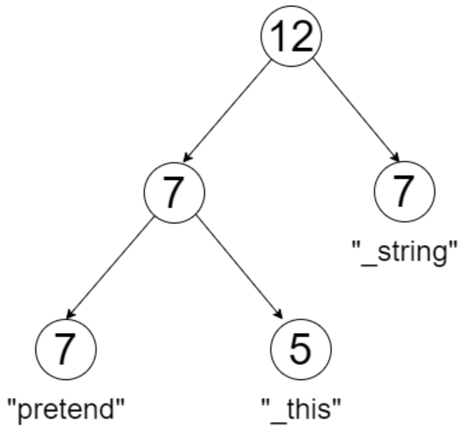


After the split:



Now we just remove the middle rope and concatenate the left and right:





Rebalancing

It turns out that ropes have a self-balancing operation. In general, ropes remain a balanced binary tree and we can perform operations to ensure this. This gives us logarithmic time for most operations. We will learn about self-balancing trees in a later lecture. Trees like Red-Black and AVL trees have rotations that allow them to restore their balanced property. Ropes have similar rotations. I'll leave the discussion on rotations alone for now as we will cover that in detail when we cover self-balancing trees.

Efficiency

Compared to a normal string, here is how ropes perform:

Operation	Rope	String
Indexing	`O(lg n)`	`O(1)`
Concatenation	`O(lg n)`	`O(n)`
Insert	`O(lg n)`	`O(n)`
Delete	`O(lg n)`	`O(n)`