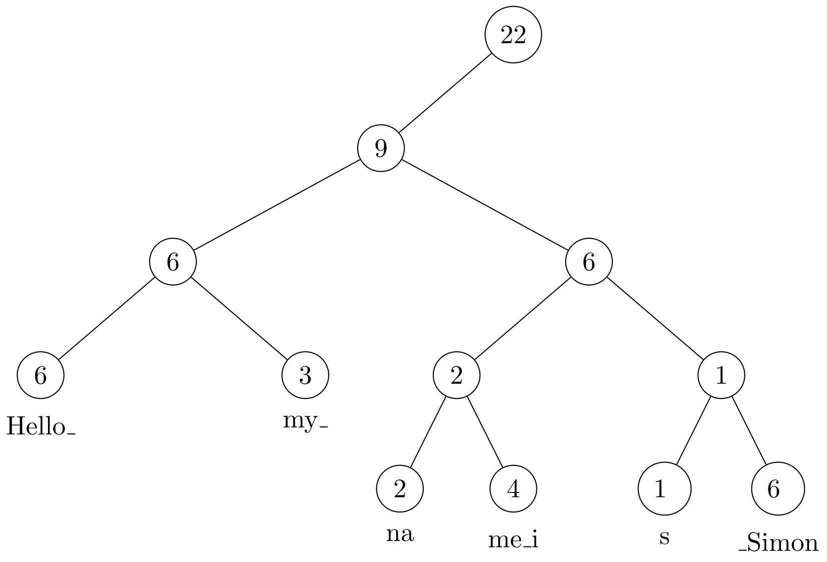

Figure 1: A Rope representing the string "Hello my name is Simon" (Adapted from an image by Meng Yao, licensed under CC BY-SA 3.0, via Wikimedia Commons). Each node is labelled with its weight.

0 The Rope Data Structure[1]

A Rope is a special type of binary tree that is used to represent a string. Insertions and deletions can be performed efficiently on Ropes, making them preferable to standard (e.g. C-style) strings when these operations occur frequently.

The data for the string represented by a Rope is stored in the leaf nodes: each leaf node contains a (standard) string *segment*, which makes up part of the represented string. Intuitively, the string represented by a Rope t, denoted $\mathit{repr}(t)$, is defined recursively as follows:

1. If t is a leaf node, then $\mathit{repr}(t)$ is the segment stored at that node.
2. If t has only a single child c, then $\mathit{repr}(t) = \mathit{repr}(c)$
3. Otherwise, $\mathit{repr}(t) = \mathit{repr}(l)$ `++` $\mathit{repr}(r)$.

Each node in a Rope also stores an integer *weight*. The weight of a leaf node is equal to the length of the segment it contains. The weight of a non-leaf node is equal to the length of the string represented (in the $\mathit{repr}(.)$ sense above) by its *left* child. Storing weights enables several operations on the Rope to be performed more efficiently: for example, it is possible to compute the length of the represented string via a single traversal down the right children of the tree. Fig. 1 visualises a Rope representing the string "Hello my name is Simon"[2] (we have used "␣" to represent spaces in the figure illustration).

For this challenge, you will write a Rope implementation and prove the correctness of various operations on Ropes with respect to the strings they represent. In your solutions, you can represent characters and strings of characters in any reasonable way (e.g. using integers to represent characters).

Tasks

(a) Implement a function `toStr(r:Rope): String` that returns the string represented by a Rope. The `toStr()` function should correspond to the recursive definition *repr* described above (in particular, it is inefficient: it traverses the entire tree, ignoring the integer weights).

(b) Prove that your function is memory-safe/crash free and always terminates.

[1]We warmly thank Zack Grannan for contributing the idea for this challenge.

[2]This example is taken from the Wikipedia page on Ropes (`https://en.wikipedia.org/wiki/Rope_(data_structure)`)

The `toStr()` function is inefficient, but the idea is to now implement several efficient functions (that avoid calling this function) and *specify* them in terms of this simpler function (if your verification tool supports these notions, `toStr()` calls should only be made in specifications / ghost code in the tasks below).

More Tasks

(c) Implement a recursively-defined function `strLen(r:Rope): int` that *efficiently* computes the length of the string represented by a Rope (i.e. using the `weight` field values). Prove that, if `result` represents the result of calling this function on a Rope `r`, the property `result == length(toStr(r))` where `length` represents a length function on `String` values.

(d) Implement a function `concat(left:Rope, right:Rope): Rope`, that joins two `Rope` instances into one by allocating a new node and making the two parameters its children. Specify and prove that applying `toStr` to the result of a call to `concat` gives a `String` equal to `toStr(left) ++ toStr(right)`.

(e) Implement a function `delete(r:Rope, i:int, len:int) : Rope`, that returns a `Rope` in which the `len` characters starting from position `i` have been removed from the represented string. The function may destructively update/reuse the passed `Rope` instance (or not) as you prefer. Specify and prove that your implementation has the correct effect on `String` represented by the resulting `Rope`.

(f) Prove that the implementations of these functions are memory-safe/crash free and always terminate.

As an example of the last operation, if (for some `Rope r`) `toStr(r) == "lizard"` then it's expected that `toStr(delete(r,1,2)) == "lard"`.

The `delete` operation can, for example, be implemented by adapting the following pseudocode:

```
def delete(tree, i, len):
   (left, remaining) = split(tree, i)
   (_, right) = split(remaining, len)
   return concat(left, right)
```

in which `split()` is a function that splits a `Rope` in two, e.g. via the following pseudo-code:

```
def split(tree, i):
   if isLeaf(tree):
       right = mkLeaf(tree.value[i..])
       tree.value = tree.value[..i]
       tree.weight = i
       return (tree, right)
   else if i < weight:
       (left, right) = split(tree.left, i)
       return (left, concat(right, tree.right))
   else if i > weight:
       (left, right) = split(tree.right, i - weight)
       return (concat(tree.left, left), right)
   else:
       return (tree.left, tree.right)
```

In the code above, `mkLeaf(s)` creates a new (leaf) Rope with value `s`. For a `String s` and in-bounds integer `i`, the notations `s[..i]` and `s[i..]` represent the strings in which (only) the first `i` elements have been kept/deleted, respectively. Your implementations need not precisely follow the pseudo-code above.

Extension The paper "Ropes: an Alternative to Strings" (*Software: Practice and Experience*, 1995) describes the following optimisation for the concatenation operation:

> In the general case, concatenation involves simply allocating a concatenation node containing two pointers to the two arguments. For performance reasons, it is desirable to deal with the common case in which the right argument is a short flat string specially. If both arguments are short leaves, we produce a flat rope (leaf) consisting of the concatenation. This greatly reduces space consumption and traversal times. If the left argument is a concatenation node whose right child is a short leaf, and the right argument is also a short leaf, then we concatenate the two leaves, and then concatenate the result to the left son of the left argument.

Even More Tasks

(g) Implement these two optimisations to the `concat` function (you might want to make a new version). You can use a function `isShort(s: String): bool` to represent the "shortness" criterion; if your tool supports it you don't need to give the function a body. In other words, the function can be *uninterpreted*: your verification can be agnostic as to what counts as "short" (that's up to the function).

(h) Implement a method `insert(r: Rope, i: int, toInsert: String)` that modifies the `Rope`, such that its new string representation includes the passed string `toInsert` inserted at position `i`.

(i) Prove that the implementations of these functions are memory-safe/crash free and always terminate.

As an example of the last operation, if (for some `Rope r`) `toStr(r) == "lard"` then it's expected that, after calling `insert(r,1,"iz")`, instead we have `toStr(r) = "lizard"`.