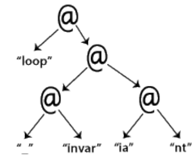
15-122: Principles of Imperative Computation, Fall 2015

Homework 7 Programming: Ropes



Due: Thursday, November 5 2015 by 22:00

For the programming portion of this week's homework, you will implement the data structure of ropes, which provide constant-time string concatenation.

The code handout for this assignment is at

`http://www.cs.cmu.edu/~fp/courses/15122-f15/assignments/ropes-handout.tgz`

The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in. There is a TEN (10) HANDIN LIMIT - you may only submit to Autolab ten times for this assignment without penalty; every additional handin will incur a half-point penalty.

For this assignment, the last thing the autograder will do is attempt to run your file rope-test.c0 against a few good data structures and many bad ones. Putting as many test cases as you can think of in this file will not only help you debug your code, but give you a chance to see how your testing stacks up against a large test suite. (Unless you're a highly diligent tester, it's likely you'll see many `TEST FAILED!` messages from this phase, but it doesn't affect your score, only your position on the scoreboard.)

1 Introduction to Ropes

The most obvious implementation of a string is as an array of characters. However, this representation of strings – which is also the way that the C0 compiler implements the `string` data type – is particularly inefficient at handling string concatenation. Running `string_join` in C0 on two strings of size n and m takes time in $O(n + m)$.

A *rope* is a tree-like data structure that provides a more efficient way of concatenating strings. A rope is a pointer to a `rope` data structure defined in C0 as follows:

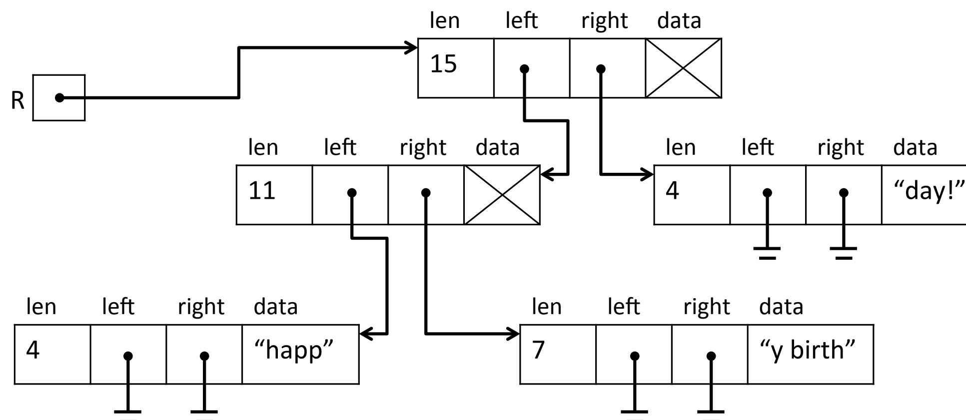
```
typedef struct rope_node rope;
struct rope_node {
  int len;
  rope* left;
  rope* right;
  string data;
};
```

A valid rope must be either `NULL`, a leaf, or a non-leaf. More specifically:

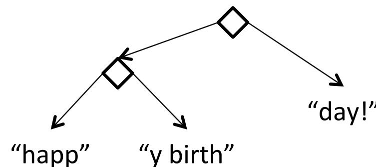
- `NULL` is a valid rope. It represents the empty string.

- A rope is a leaf if it is non-`NULL`, has a non-empty string `data` field, has `left` and `right` fields that are both `NULL`, and has a strictly positive `len` equal to the length of the string in the `data` field (according to the C0 string library function `string_length`).
- A rope is a non-leaf if it has non-`NULL` `left` and `right` fields, both of which are valid ropes, and if it has a `len` field equal to the sum of the `len` fields of its children. The `data` field of a non-leaf is unspecified. We'll call these non-leaves *concatenation nodes*.

This is one of many ropes that represents the 15-character string `"happy birthday!"`:



Note that where we indicate Xes in the `data` field, any contents would be allowed and we would still have a valid rope. We can also represent the same structure using a short-hand notation that illustrates the two different types of nodes, leaf nodes and concatenation nodes:



Task 1 (2 pts.) *In the file* `rope.c1`*, define the* `rope` *type as specified on the previous page and write a data structure invariant* `bool is_rope(rope* R)`*. For full credit, you should ensure that your data structure invariant terminates on* <u>*all*</u> *inputs. HINT: If your circularity check requires more than 2-6 extra lines, you're doing it wrong.*

2 Implementing Ropes

A full interface for ropes would presumably need to mimic the entire C0 string library. In this section, we'll just be implementing a limited subset of this library.

```
typedef _______ rope_t;
int rope_length(rope_t R);
rope_t rope_join(rope_t R, rope_t S)
  /*@requires rope_length(R) <= int_max() - rope_length(S); @*/ ;
char rope_charat(rope_t R, int i)
  /*@requires 0 <= i && i < rope_length(R); @*/ ;
rope_t rope_sub(rope_t R, int lo, int hi)
  /*@requires 0 <= lo && lo <= hi && hi <= rope_length(R); @*/ ;
```

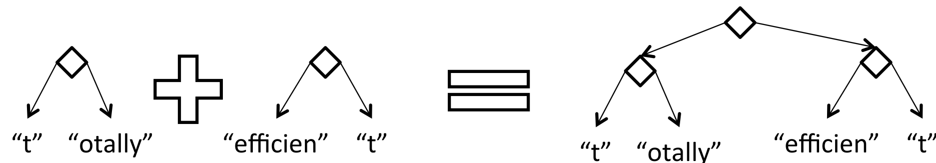
Functionally, these four functions should do the same thing as the similarly-named function in the C0 `string` library. We'll also implement two functions for converting between C0 strings and our data type of ropes.

```
rope_t rope_new(string s);
string rope_tostring(rope_t R);
```

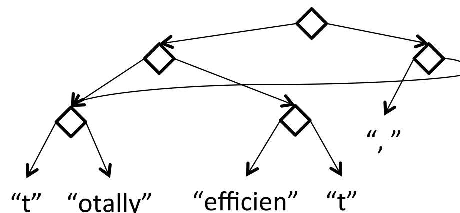
When we talk about the big-O behavior of rope operations, we assume for simplicity the rope's leaves contain strings that are smaller than some small constant, which means that all operations on C0 strings can be treated as constant-time operations.

Task 2 (3 pts.) *In the file* `rope.c1`*, implement the* $O(1)$ *functions* `rope_new`*,* `rope_length`*, and* `rope_join`*.*

The `rope_new` function takes any C0 string and returns a rope without any concatenation nodes. The `rope_join` function is able to work in constant time because, at most, it has to allocate a single concatenation node:



In the example above, the client of the rope library can continue using the rope representing `"totally"` even though the allocated memory for that rope is a part of the rope representing `"totallyefficient"`. This *structure sharing* between different ropes means that, while ropes are a data structure that we can treat like a tree, the memory representation may not actually be a tree. Here's another example: if `R1` is rope for `"totally"` above and `R2` is the rope for `"efficient"` above, then executing the expression `rope_join(rope_join(R1, R2), rope_join(rope_new(", "), R1))` will produce the following structure in memory:



Structure sharing for ropes only works because none of the rope interface functions allow us to modify ropes after they have been created. By sharing structure, we can make very very big strings without allocating much memory, and this is one reason it was necessary to add the precondition checking for overflow to `rope_join`.

Task 3 (2 pts.) *In the file* `rope.c1`*, implement the recursive functions* `rope_charat` *and* `rope_tostring`*.*

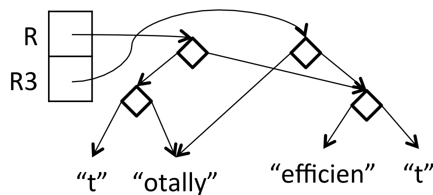
Your implementation of `rope_charat` should take, in the worst case, time proportional to the *height* of the rope. If we kept ropes balanced, this would mean that `rope_charat` would take time in $O(\log n)$, where n is the length of the rope as reported by `rope_length`. We will not, however, implement balancing in this assignment, and none of the code you write in this section should modify the structure of existing ropes in any way.

The `rope_tostring` function returns the string that a rope represents. There's a way to implement this function so that its running time is in $O(n)$, but this would be overkill. Just implement the most natural recursive solution possible, which uses `string_join`.

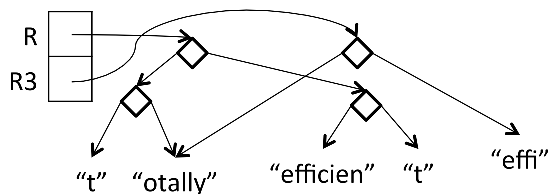
Sharing between ropes gets more interesting once we start considering the `rope_sub` function. The C0 library function `string_sub(s,lo,hi)` returns the segment of the string `s` from index `lo` (inclusive) to index `hi` (exclusive). The function `rope_sub` must do the same thing, without changing the structure of the original rope in any way, while also maximizing sharing between the old rope and the new rope and only allocating a new node when it is impossible to use the entire string represented by an existing rope.

Here are some examples, where we have `R` as the rope representing `"totallyefficient"` from the previous page.

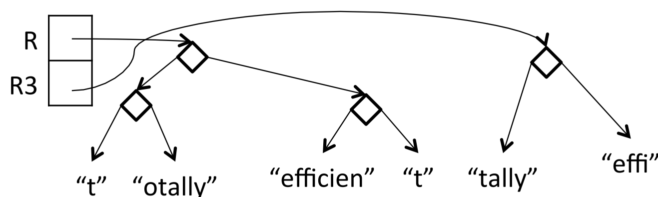
After running `rope_t R3 = rope_sub(R, 1, 16);`



After running `rope_t R3 = rope_sub(R, 1, 11);`



After running `rope_t R3 = rope_sub(R, 2, 11);`



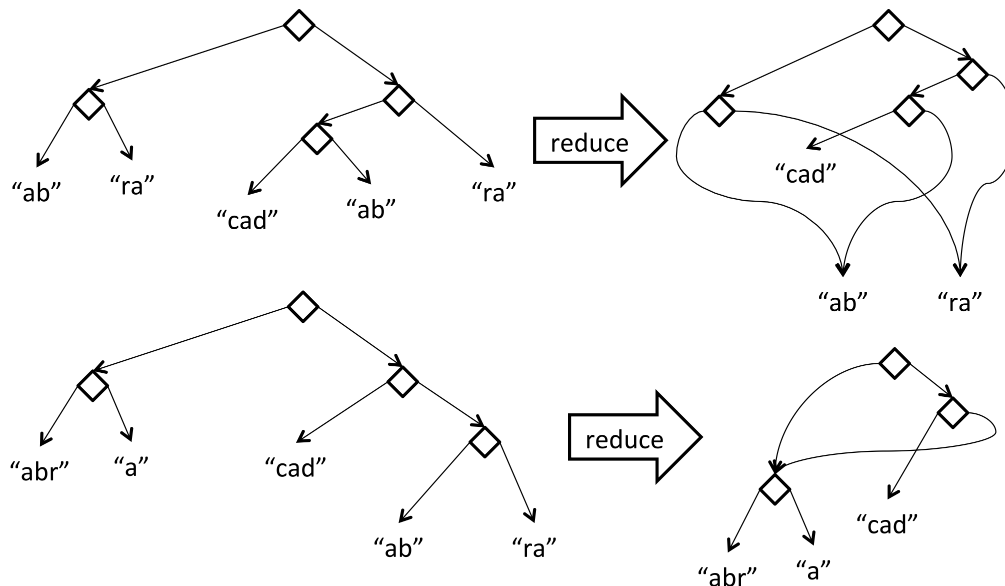
Running `rope_sub(R,0,1)` and `rope_sub(R,7,16)` should not cause *any* new memory to be allocated, because these substrings are captured by subtrees of the original rope. Running `rope_sub(R,2,3)` must return a newly-allocated leaf node containing the string `"t"`.

Task 4 (5 pts.) *In the file* `rope.c1`*, implement the recursive function* `rope_sub`*. Without changing the structure of the original rope in any way, this function should minimize memory allocation by sharing as much of the original rope as possible.*

One hint: in your recursive function, try to first identify all the cases where it is possible to return immediately without any new allocation. What cases are left?

3 Reducing Memory Usage

The `rope_join` and `rope_sub` functions use sharing to reduce memory usage, but we specified that they should not change the structure of existing ropes. For this task, you will write a memory-reduction procedure that changes the structure of existing ropes to conserve memory without changing the strings that those ropes represent.



Your memory-reduction procedure will use a hash set. When given a rope, you should look up that rope in the hash set to see if an equivalent rope (one representing the same string) already exists in the set. If so, your memory-reduction procedure can just return that already-stored rope in place of the rope you were given.

If your memory-reduction procedure doesn't find the rope already in the hash set, it should recursively call itself, first on the left sub-rope, and then on the right sub-rope. Then, without allocating any additional memory, you can replace the original left and right sub-ropes with the results of calling the memory-reduction procedure on them. (It's always okay to replace a rope with another rope that represents the same string.) Now you have a rope with two sharing-maximized sub-ropes; this new rope should be added to the hash set for future use.

Task 5 (3 pts.) *In the file* `rope.c1`*, implement the function* `rope_reduce` *using a helper function. This function takes an array* `A` *of ropes, allocates a hash set, and runs the memory-reduction procedure on each of the ropes in the array. Ropes and sub-ropes stored in lower indices should remain in the hash set and be re-used when processing ropes and sub-ropes stored in higher indices.*

More examples of what `rope_reduce` does can be found on the next page. You'll need to write functions to initialize the client interface of hash sets. The simplest way to write the `elem_hash_fn` and `elem_equal_fn` functions involves repeatedly using `rope_charat`. That implementation will work fine for this assignment, but it is possible to create a more efficient version.

More examples of how we expect `rope_reduce` to work: