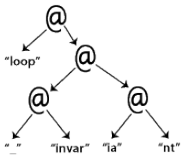
15-513: Introduction to Computer Systems, Summer 2019

Programming Homework 0: Cord Lab



Due: Saturday 1st June, 2019 by 9pm

For the 0th lab of 15-513, you will implement the data structure of cords, which provide constant-time string concatenation.

The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in.
For the fastest response, please use Piazza. Your posts will be private by default. Before asking a question, though, please read this handout in its entirety, and also look at the FAQ page. This lab involves a combination of many tools, each of which has its own quirks. They often produce error messages that are difficult to decipher. The FAQ covers many of these.

This lab is based on material from an introductory course. CMU Academic Integrity applies for this lab as well. You are not allowed to search online, ask for support from, or in any other way re-use code from previous iterations of 15-213/15-513/18-213 or other courses.

1 Logistics

- This is an individual project. All handins are electronic using the Autolab service.
- You should do all of your work in an Andrew directory, using either the shark machines or a Linux Andrew machine.

2 Logging in to Autolab

All 213/513 labs are being offered this term through a Web service developed by CMU students and faculty called *Autolab*. Before you can download your lab materials, you will need to update your Autolab account. Point your browser at the Autolab front page

```
https://autolab.andrew.cmu.edu
```

You will be asked to authenticate via Shibboleth. After you authenticate the first time, Autolab will prompt you to update your account information with a *nickname*. Your nickname is the external name that identifies you on the public scoreboards that Autolab maintains for each assignment, so pick something interesting! You can change your nickname as often as you like. Once you have updated your account information, click on "Save Changes" button, and then select the "Home" link to proceed to the main Autolab page.

© Carnegie Mellon University 2019

You must be enrolled to receive an Autolab account. If you added the class late, you might not be included in Autolab's list of valid students. In this case, you won't see the 513 course listed on your Autolab home page. If this happens, contact the staff and ask for an account. Accounts are updated every 24 hours.

3 Introduction to Cords

The most obvious implementation of a string is as an array of characters. However, this representation of strings is particularly inefficient at handling string concatenation. Running `strcat` in C on two strings of size n and m takes time in $O(n + m)$.

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

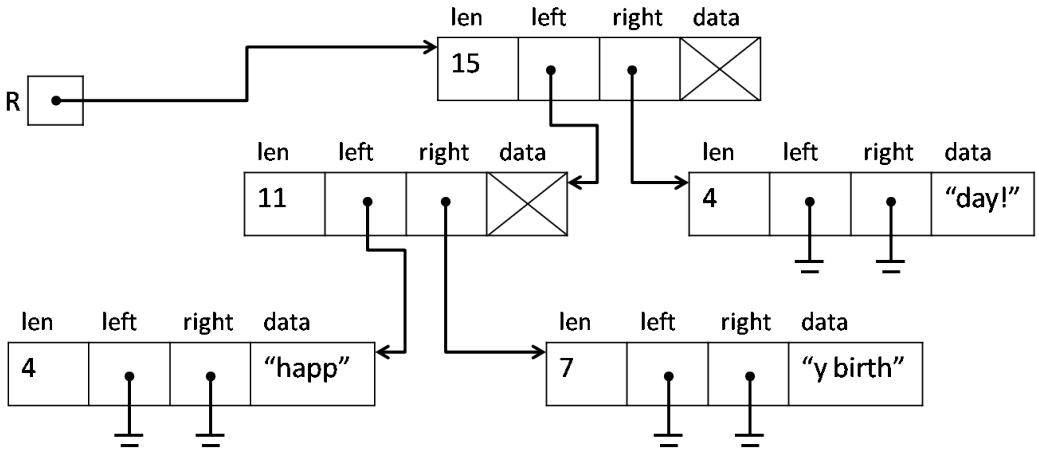
```
typedef struct cord_node cord;
struct cord_node {
  int len;
  cord* left;
  cord* right;
  char* data;
};
```

© Carnegie Mellon University 2019

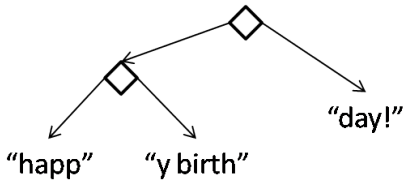
A valid cord must be either `NULL`, a leaf, or a non-leaf. More specifically:

- `NULL` is a valid cord. It represents the empty string.
- A cord 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 C string library function `strlen`).
- A cord is a non-leaf if it has non-`NULL` `left` *and* `right` fields, both of which are valid cords, 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 cords 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 cord. 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 (4 points)

In the file `cord.c`, write a data structure invariant **`bool`** `is_cord(cord* c)`. For full credit, you should ensure that your data structure invariant terminates on all inputs. HINT: If your circularity check requires more than 2-6 extra lines, you're doing it wrong.

4 Implementing Cords

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

© Carnegie Mellon University 2019

```
// typedef _______ cord_t;
int cord_length(cord_t R);
cord_t cord_join(cord_t R, cord_t S)
  /*@requires cord_length(R) <= int_max() - cord_length(S); @*/ ;
char cord_charat(cord_t R, int i)
  /*@requires 0 <= i && i < cord_length(R); @*/ ;
cord_t cord_sub(cord_t R, int lo, int hi)
  /*@requires 0 <= lo && lo <= hi && hi <= cord_length(R); @*/ ;
```

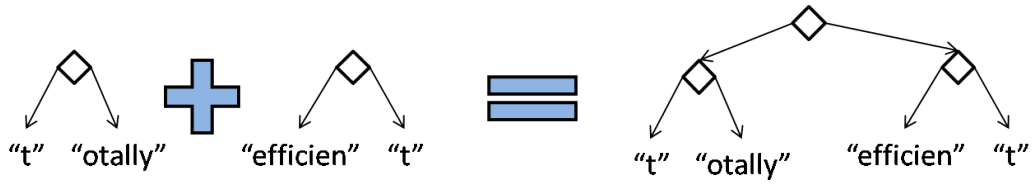
Functionally, these four functions should do the same thing as the similarly-named function in the C **string** library, `strlen`, `strcat`, character at, and substring. We'll also implement two functions for converting between C strings and our data type of cords.

```
cord_t cord_new(string s);
char* cord_tostring(cord_t R);
```

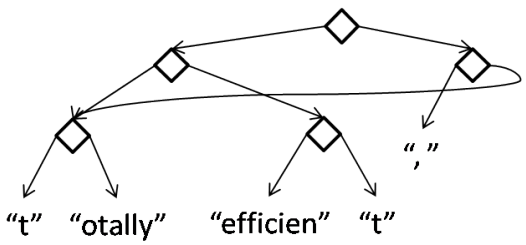
When we talk about the big-O behavior of cord operations, we assume for simplicity the cord's leaves contain strings that are smaller than some small constant, which means that all operations on C strings can be treated as constant-time operations.

Task 2 (5 points) **Constant time operations.**
In the file `cord.c`, implement the $O(1)$ functions `cord_new`, `cord_length`, and `cord_join`.

The `cord_new` function takes any string and returns a cord without any concatenation nodes. The `cord_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 cord library can continue using the cord representing `"totally"` even though the allocated memory for that cord is a part of the cord representing `"totallyefficient"`. This *structure sharing* between different cords means that, while cords 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 cord for `"totally"` above and `R2` is the cord for `"efficient"` above, then executing the expression `cord_join(cord_join(R1, R2), cord_join(cord_new(", "), R1))` will produce the following structure in memory:



© Carnegie Mellon University 2019

Structure sharing for cords only works because none of the cord interface functions allow us to modify cords 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 `cord_join`.

Task 3 (4 points) **Simple recursive operations.**
In the file `cord.c`, implement the recursive functions `cord_charat` and `cord_tostring`.

Your implementation of `cord_charat` should take, in the worst case, time proportional to the *height* of the cord. If we kept cords balanced, this would mean that `cord_charat` would take time in $O(\log n)$, where n is the length of the cord as reported by `cord_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 cords in any way.

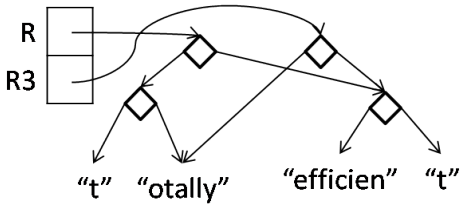
The `cord_tostring` function returns the string that a cord represents. There's a way to implement this function so that its running time is in $O(n)$, but this would be overkill. Just implement the most natural recursive solution possible, which uses `strcat`.

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

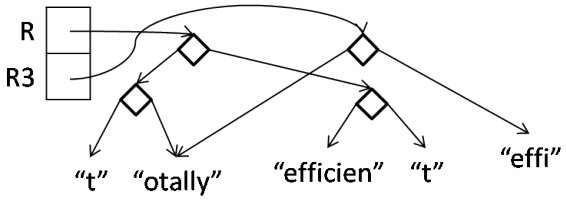
© Carnegie Mellon University 2019

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

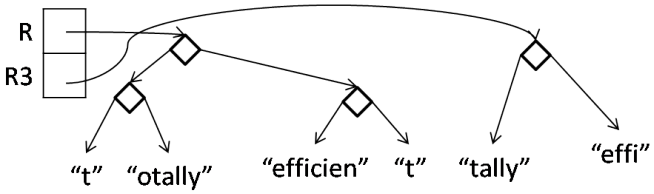
After running `cord_t R3 = cord_sub(R, 1, 16);`



After running `cord_t R3 = cord_sub(R, 1, 11);`



After running `cord_t R3 = cord_sub(R, 2, 11);`



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

Task 4 (7 points) In the file `cord.c`, implement the recursive function `cord_sub`. Without changing the structure of the original cord in any way, this function should minimize memory allocation by sharing as much of the original cord as possible.

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?

5 Freeing your data structure

Task 5 (0 points) Since we have allocated memory for our `cord` data structure, we should free any memory we allocated after we are done with the program. For this lab, though, you are not required to implement this feature.

However, you should definitely think about where and how you are calling `free`. You can use `valgrind` to check for memory leak.

© Carnegie Mellon University 2019