15-122 : Principles of Imperative Computation

Summer 1 2012

Assignment 7: Ropes

(Programming Part)

Due: Friday, June 22, 2012 by 23:59

For the programming portion of this weeks homework, youll get to implement a ropes data structure - an alternative to strings. In particular, you will learn how to allocate and deallocate memory in C.
You should submit these files electronically by the due date. Detailed submission instructions can be found below.

1 Assignment: Ropes - An Alternative to Strings

For the programming portion of this week's homework, you'll get to implement a ropes data structure (defined later) that will allow you to start coding in C with special emphasis on explicit memory management. In particular, you will learn how to allocate and deallocate memory, C style.

You should submit your code electronically by 11:59 pm on the due date. Detailed submission instructions can be found below.

Starter code. Download the file `hw7.zip` from the course website.

Compiling and running. Compile your code using GCC. The following set of options will catch many common mistakes at compile-time:

```
gcc -Wall -Wextra -std=c99 -pedantic -Werror <files...>
```

For details on how we will compile your code, see the file `COMPILING.txt` included in the starter code. To enable assertion checking, ensure that `DEBUG` is defined using the `-DDEBUG` option. **Warning:** *You will lose credit if your code does not compile.*

To detect any invalid memory accesses or memory leaks, you can run your compiled binary through Valgrind:

```
valgrind ./a.out
```

Use the `-v` option for verbose output, or the `--leak-check=full` option to generate a more complete report of possible memory leaks. **Warning:** *You will lose credit if your code has invalid memory accesses or memory leaks.*

Submitting. Once you've completed some files, you can submit them by running

```
handin -a hw7 <file1> ... <fileN>
```

You can submit files as many times as you like and in any order. When we grade your assignment, we will consider the most recent version of each file submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

Annotations. Use the macros in `contracts.h` to write appropriate annotations for your code in a style similar to what we've been doing in C_0. Remember that writing these annotations *before* writing the code will help you understand the problem more clearly and save debugging time later. **Annotations are part of your score for the programming problems; you will not receive full credit if they are weak or missing.**

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on the course bboard (`academic.cs.15-122`) if you're unsure of what constitutes good style.

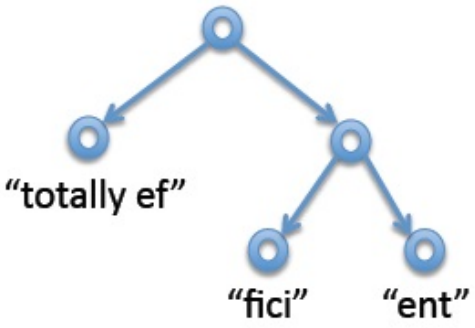

Figure 1: One possible rope representing the string "totally efficient".

1.1 Background

In C, strings are represented as arrays of characters. This allows constant-time access of a character at an arbitrary position, but it also has some disadvantages. In particular concatenating two strings (function `strcat`) is an expensive operation since we have to create a new character array and copy the two given strings into the new array character by character. A new data structure called `ropes` attempts to improve efficiency of concatenation by representing strings as a directed acyclic graph, where the leaves contain ordinary strings and the interior nodes represent concatenations. For example, the string "totally efficient" might look as shown in Figure 1 (among many other possibilities).

1.2 Ropes

The rope is a data structure proposed by researchers Boehm, Atkinson, and Plass in 1995. The summary of their paper states that,

> Programming languages generally provide a string or text type to allow manipulation of sequences of characters. This type is usually of crucial importance, since it is normally mentioned in most interfaces between system components. We claim that the traditional implementations of strings, and often the supported functionality, are not well suited to such general-purpose use. They should be confined to applications with specific, and unusual, performance requirements. We present ropes or heavyweight strings as an alternative that, in our experience leads to systems that are more robust, both in functionality and in performance.

The rope data structure is designed to support efficient, scalable, non-destructive operations on immutable strings. In particular, string concatenation using ropes can be done in constant time. Ropes gain much of their efficiency from clever use of sharing data, so the main subtlety in their implementation is tracking when it is safe to free associated memory.

In order to implement ropes, we start by defining a rope node. A rope node is defined as follows.

```
typedef struct rope_node* rope;
struct rope_node {
  size_t size;          /* size of the string in this rope */
  size_t position;      /* length of the left rope */
  struct rope *left;    /* pointer to left child, NULL for leaves */
  struct rope *right;   /* pointer to right child, NULL for leaves */
  char *data;           /* string data, NULL for interior nodes */
  size_t ref_count;     /* number of references to this rope */
};
```

Each rope node represents an immutable string. The type `size_t` stands for unsigned integers between `0` and `SIZE_MAX`, a macro defined in `stdint.h`. We use a reference counting scheme `ref_count` to track how many references to a node exist; when no references remain, the memory pointed to may be freed. The `size` field contains the length of the string; the `position` field tells us the starting index of the right rope or the size of the left rope. The `left` and `right` are pointers to rope nodes. If a string is not merged with another string, it remains as a single node with `data` field set to that string. For all interior nodes (not leaves), `data` field is set to NULL.

The following Figure 2 shows that the rope actually is a directed graph where each node has at most 2 outgoing edges and zero or more incoming edges. In a rope each internal node represents a concatenation of its children, and the leaves represented as contiguous arrays of characters.

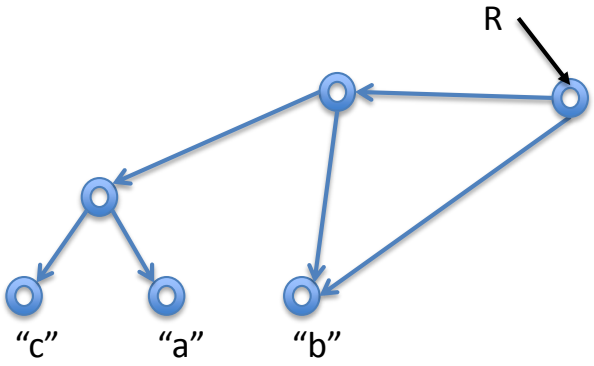

Figure 2: Rope is a directed acyclic graph.

Task 1 (4 pts) *Implement the following function*

```
bool is_rope(rope s);
```

that takes a rope and returns `true` *or* `false` *depending on whether the rope satisfies the rope invariants. One of the invariants is that a rope has no cycles. This function must be used to write contracts in your later code.*

Task 2 (3 pts) *Implement the following function*

```
rope rope_new(char *str);
```

that takes a pointer to a C string and returns a new rope representing it. The function should make a deep copy of its argument, this is essential for the correct implementation of the `rope_sub` *function.*

A newly constructed rope returned from the `rope_new(char *str)` has the following characteristics:

- the `size` field contains the length of the string, as computed by `strlen(str)`,
- the `position` field is set to 0,
- the `data` field is a copy of `str`, and
- the `ref_count` field is set to 1.

Task 3 (5 pts) *Implement the following function*

```
rope rope_join(rope str1, rope str2);
```

that takes two ropes and combines them to create a single rope by creating a new parent node to connect the ropes. Note that there is no explicit merging takes place between ropes, but merely connecting them in the right way. The function should "retain" a copy of the child ropes by incrementing their reference counts.

The diagram in Figure 3 shows how two individual ropes are merged to form one rope. The result of `rope_join(str1, str2)` is a new rope with the following characteristics:

- the `size` field contains the total length of the represented string,
- the `data` field is set to NULL,
- the `position` field contains the starting index of the right rope (or equivalently, the size of the left rope),
- the `left` and `right` fields are pointers to the child ropes,
- the `ref_count` field is set to 1, and
- the children's `ref_count` fields are increased by 1.

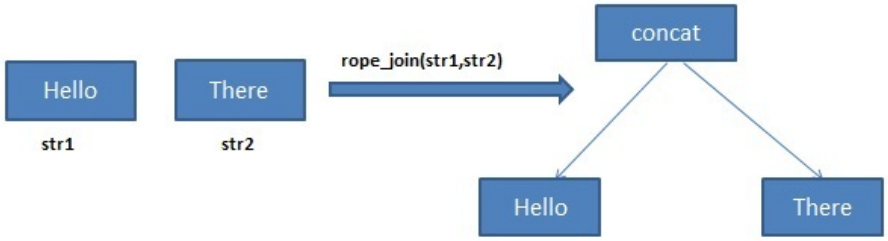

Figure 3: Concatenating two leaf ropes to form a new rope.

Task 4 (5 pts) *Implement the function*

```
void rope_free(rope str);
```

that frees `str` *if its* `ref_count` *is exactly 1. Then the function recursively traverses in a directed graph (a rope) decrementing reference counts for all reachable nodes. If the count becomes zero, the function frees the memory associated with that node. Be careful not to free any memory that might still be in use!*

In case when `str->ref_count > 1`*, the function does not do anything.*

Use `valgrind` *to test your code for invalid memory accesses and memory leaks.*

For the rope in Figure 2, calling `rope_free(R)` will result in recursive removing all nodes. On the other hand, calling `rope_free` on any other nodes won't release any memory. When you free a leaf don't forget to free data in that node.

Task 5 (3 pts) *Implement the function*

```
char rope_charat(rope str, size_t idx);
```

that takes a rope and an index and returns the character at the index. Note that character can be in the left or right subtree.

Task 6 (5 pts) *Implement the function*

```
char* rope_to_chararray(rope str);
```

that takes a rope and returns a `'\0'`*-terminated array of characters (i.e., a C string). You must explicitly allocate associated memory for the* `char*` *returned as the client is not expected to do any memory allocation. (But by the golden rule, the client will be responsible for deallocating that memory!)*

Task 7 (3 pts) *Implement the function*

```
int rope_compare(rope str1, rope str2);
```

that takes two ropes and returns 1, 0, or -1 if the first rope is lexicographically greater than, equal to, or less than the second, respectively. (This behavior is similar to the C0 `string_compare` *function and the C standard library's* `strcmp` *function.)*

Task 8 (7 pts) *Implement the function*

```
rope rope_sub(rope str, int i, int j);
```

that takes a rope and two integers and returns a rope representing the substring between the first index (inclusive) and the second index (exclusive). You must explicitly allocate associated memory for the returned rope (for all its nodes) as the client is not expected to do any memory allocation. However the client will then be responsible for freeing the rope later using `rope_free`*.*

The following Figure 4 and Figure 5 demonstrate the function `rope_sub`.

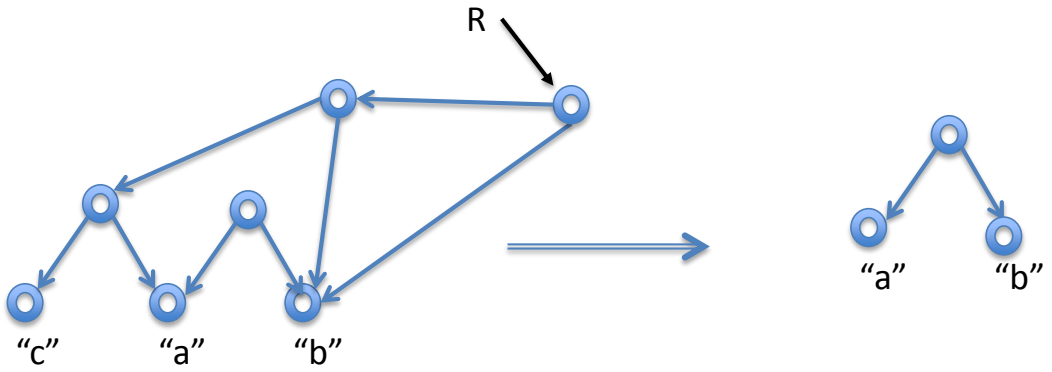

Figure 4: rope_sub(R, 1, 3), where R represents a rope for "cabb"

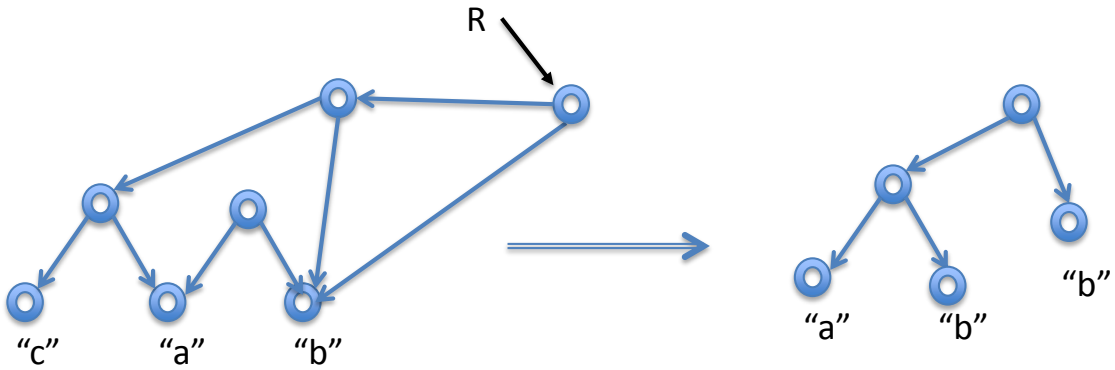

Figure 5: rope_sub(R, 1, 4), where R represents a rope for "cabb"

Don't forget to include all annotations using the macros in `contracts.h`, and be sure to test your code using `valgrind` for memory leaks!