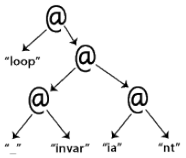
15-513: Introduction to Computer Systems, Summer 2023
Programming Homework 0: Cord Lab



Due: Tuesday 23rd May, 2023 by 11:59pm ET
Last handin: Friday 26th May, 2023 by 11:59pm ET

1 Overview

For the 0th lab of 15-513, you will implement the data structure of *cords*, which provide constant-time string concatenation. This lab will give you practice in the style of programming you will need to be able to do proficiently, especially for the later assignments in the class The material covered *should* all be review for you. Some of the skills tested are:

- Explicit memory management, as required in C.
- Creating and manipulating pointer-based data structures.
- Working with strings.
- Enhancing the performance of key operations by storing redundant information in data structures.
- Implementing robust code that operates correctly with invalid arguments, including NULL pointers.

If you have questions regarding the assignment, for the fastest response, please use Piazza. Your posts should be private by default. Before asking a question, though, please read this handout in its entirety, and also look at the FAQ page.

This is an individual project. You are not allowed to search online for, ask for support from, or re-use code from previous iterations of 15-213/15-513 or any other course. Before you begin, please take the time to review the course policy on academic integrity at http://www.cs.cmu.edu/~213/academicintegrity.html.

2 Logistics

- All handins are electronic using the Autolab service.
- You should do all of your work in an Andrew directory, using a Shark machine.

2.1 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 activate 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 the "Save Changes" button, and then select the "Home" link to proceed to the main Autolab page.

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 15-513 listed on your Autolab home page. If this happens, contact the staff and ask for an account. We update Autolab's list of students once every 24 hours, so please be patient.

2.2 Downloading the assignment

For all assignments in this class, you should do all your programming and testing using one of the Shark machines, which you can access via an SSH connection. The starter code and testing scripts that we give you are not guaranteed to work anywhere but the Shark machines. You can find a list of all the Shark machines at `http://www.cs.cmu.edu/~213/labmachines.html`. The Linux Bootcamp will cover using SSH and related topics in greater detail.

To begin working on this assignment, start by creating a directory for this course in a subdirectory of your AFS home directory that is accessible only by you. New Andrew accounts start out with a subdirectory with appropriate access control settings called `~/private`, so if you have that directory, these commands are sufficient:

```
shark$ mkdir ~/private/15513
shark$ cd ~/private/15513
```

If you don't have the `~/private` directory and don't know how to create a subdirectory that is accessible only by you, contact the Andrew helpdesk for instructions.

Once you are inside the 15513 directory, use the `autolab` command to download the starter code:

```
shark$ autolab download 15513-m23:cordlab
```

You may be prompted to authenticate; follow the instructions. This will create a deeper subdirectory named `cordlab` containing two files:

```
shark$ cd cordlab
shark$ ls
cordlab-handout.tar  cordlab.pdf
```

`cordlab.pdf` is another copy of this writeup; `cordlab-handout.tar` contains the files you need. Extract them with these commands:

```
shark$ tar --strip=1 -xf cordlab-handout.tar
shark$ rm cordlab-handout.tar
shark$ ls
check-format     cordlab.pdf   grade-iscord.o helper.mk       README
cord.c           grade-basic.o grade-rec.o    Makefile        test-cord.c
cord-interface.h grade-cordlab grade-sub.o    makesomecords.o xalloc.h
```


For this lab, you will only *need* to modify `cord.c`. You may find it useful to write tests for the API you are implementing; put these in `test-cord.c`. If you like, you may modify `.clang-format` so that "`make format`" applies your preferred style. Don't modify any other files! If you do, the autograder will ignore those modifications and your code probably won't compile.

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 $O(n + m)$.

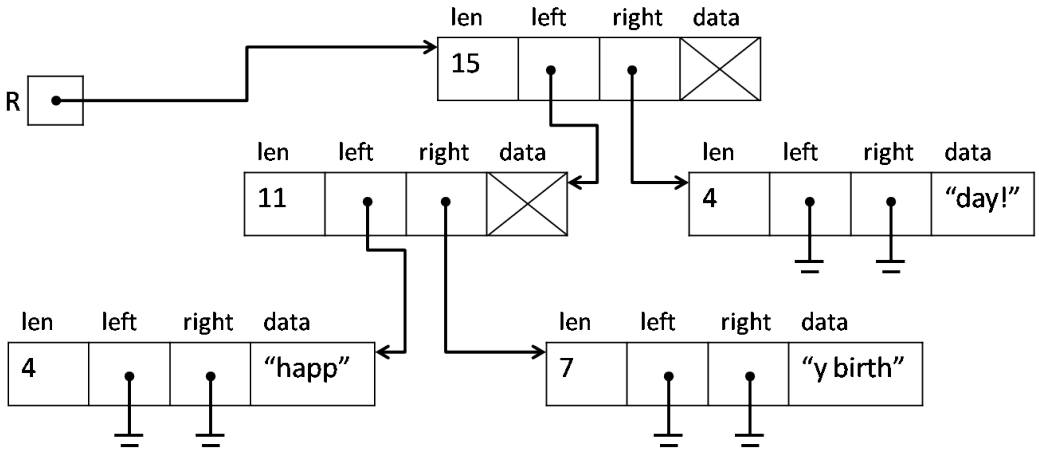
A *cord* is a tree-like data structure that provides a more efficient way of concatenating strings. A cord is a pointer to a `cord` data structure defined in C as follows:

```
typedef struct cord_node cord_t;
struct cord_node {
  size_t len;
  const cord_t *left;
  const cord_t *right;
  const char *data;
};
```

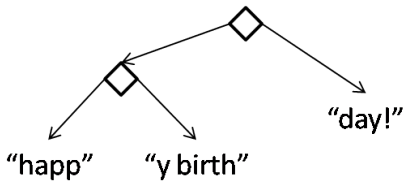
A valid cord must be either `NULL`, a leaf, or a non-leaf. More specifically:

- `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(const cord_t *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.

```
size_t cord_length(const cord_t *R);
const cord_t *cord_join(const cord_t *R, const cord_t *S);
char cord_charat(const cord_t *R, size_t i);
const cord_t *cord_sub(const cord_t *R, size_t lo, size_t hi);
```

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.

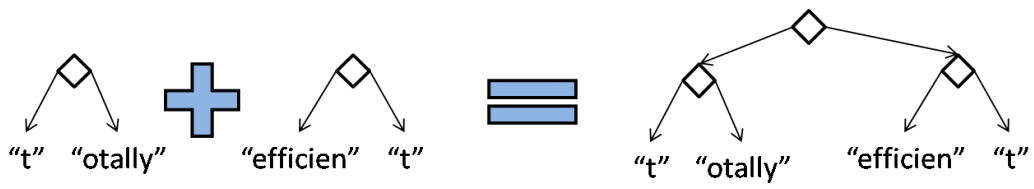
```
const cord_t *cord_new(const char *s);
char *cord_tostring(const cord_t *R);
```

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`.

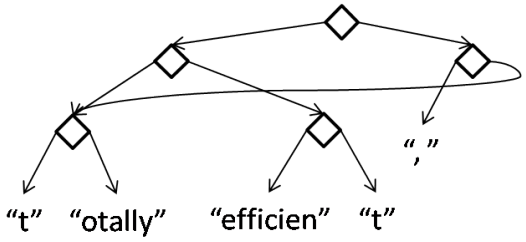
The `cord_new` function takes any string and returns a cord without any concatenation nodes. The `cord_join` function is able to work in constant time because, at most, it has to allocate a single concatenation node:



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:



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`.

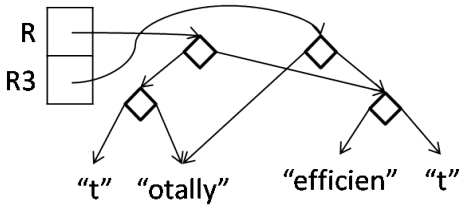
Your implementation of `cord_charat` should take, in the worst case, time proportional to the *height* of the cord. If we kept cords balanced, this would mean that `cord_charat` would take time in $O(\log n)$, where n is the length of the cord as reported by `cord_length`. We will not, however, implement 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`. Your implementation of `cord_tostring` should always return a pointer to an allocated string: think about what should happen if it receives NULL as an argument, for example.

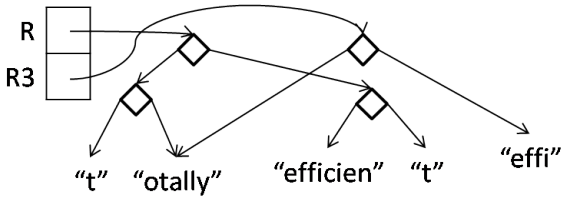
Sharing between cords gets more interesting once we start considering the `cord_sub` function. You are recommended to implement a function `string_sub(s,lo,hi)` which returns the segment of the string `s` from index `lo` (inclusive) to index `hi` (exclusive). The function `cord_sub` must do the same 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.

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

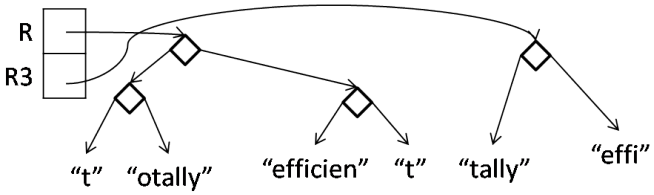
After running `const cord_t *R3 = cord_sub(R, 1, 16);`



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



After running `const 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 Memory allocation

5.1 Checking for errors

It is possible for the `malloc` and `calloc` functions to return `NULL` if they are unable to allocate memory, because the system has run out of memory. However, if this is not explicitly checked for, this can result in a null pointer dereference (which generally causes a segmentation fault in C).

A simple way to avoid this without cluttering your code is to use wrapper functions that exit the program whenever `malloc` or `calloc` fail. The corresponding `xmalloc` and `xcalloc` functions, provided in the `xalloc.h` file, do exactly this.

Proper error-checking is not required for this lab, but it will be graded in future labs as part of style grading, so you should keep it in mind. You should also keep in mind that exiting the program when `malloc` fails may not necessarily be the right thing to do in all circumstances.

5.2 Uninitialized memory

One important difference between `malloc` and `calloc` to keep in mind is that the former function does not initialize memory, which means that the contents of the memory it returns could be anything. As such, when you allocate a struct with `malloc`, you should always initialize all fields of the struct before using it.

In contrast, the `calloc` function will always zero-initialize memory, which you may find to be useful in certain situations. For more information about these functions, refer to the man pages by typing `man malloc` into the command line.

5.3 Freeing your data structure

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.

5.4 Valgrind

Valgrind is a tool that can detect various issues with the use of memory, such as:

1. Leaked memory (such as missing a call to `free`)
2. Out-of-bounds memory accesses (such as indexing past the end of an array)
3. Uninitialized memory usage (such as forgetting to initialize a local variable)
4. Incorrect calls to `malloc` or `free` (such as calling `free` twice)

Even though we won't be able to free memory, Valgrind can still be very useful for detecting other types of errors. You can run Valgrind on your `test-cord` program using the following command:

```
shark$ valgrind --leak-check=no ./test-cord
```


Note that we are passing the `--leak-check=no` flag for this lab only in order to suppress warnings about leaked memory.

6 Evaluation

6.1 Testing

To compile and test your code locally, run these commands:

```
shark$ make
shark$ ./grade-cordlab
```

The program '`grade-cordlab`' runs your tests (from `test-cord.c`) and also runs the same set of tests that the autograder will. Note that you have **not** been provided the source code for these additional tests! If a test fails, you will see an error message like this:

```
Test './grade-sub edge'... FAIL: killed by SIGSEGV
*** Hint: cord_sub edge cases
```

The hint tells you something about what might be wrong, and you can use `gdb` on the command shown ("`./grade-sub edge`" in this case) to investigate, but you will have to work out what the bug is for yourself.

6.2 Style

This lab will not be style graded. The score you receive on Autolab will be your final score. However, your code for all labs **must be formatted correctly to receive points on Autolab.** We require you to use the `clang-format` tool to format your code. To do this, run this command:

```
shark$ make format
```

You can modify the `.clang-format` file to reflect your preferred code style. More information is available in the style guideline at http://www.cs.cmu.edu/~213/codeStyle.html.

7 Handin

To submit your code to Autolab, run these commands:

```
shark$ make format
shark$ make submit
```

You must submit your code to Autolab to receive credit for this assignment! Passing the tests locally is not enough.

After running "`make submit`", always check the Autolab website (https://autolab.andrew.cmu.edu/) to make sure you have received the grade you expect. In future labs, the autograder may do additional or more stringent tests than what you can run locally.

You may upload your work as often as you like until the due date. The *most recent* upload is the one that will be graded!

