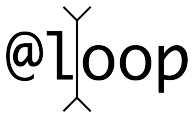
15-122: Principles of Imperative Computation, Spring 2024

Programming Homework 6: Text Buffers



Due: Thursday 7th March, 2024 by 9pm

For the programming portion of this week's homework, you'll use a doubly linked list to implement some of the basic features of a text editor.

Download the assignment handout from the course website or Autolab. 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 SIX (6) PENALTY-FREE HANDIN LIMIT. Every additional handin will incur a small (5%) penalty (even if using a late day). Your score for this assignment will be the score of your last Autolab submission.

Warning: In the course so far, we have previously considered interfaces (like pixels, stacks, queues, and dictionaries) that did not expose their internal representation to the user and that were tested as an opaque interface (black-box testing). The data structures and interfaces we implement for this assignment expose their internal representation to the client.

The expectation is that the client (who is sometimes the text editor, and sometimes you!) will mostly use this representation to read from the data structures, but they will usually not write to the data structures themselves. However, clients are allowed to manipulate the data structures, and so we expose the data structure invariants (like `is_tbuf`) as part of the interface.

This means that your data structure invariants will need to be *very good*, and we will test them very thoroughly. They need to permit anything permitted by the specification, and disallow anything that is not allowed by the specification. One thing we will *not* require you to do in this assignment is circularity checking: we will never test your data structures against a doubly linked list where you can follow `next` pointers forever without reaching `NULL` or one where you can follow `prev` pointers forever without reaching `NULL`.

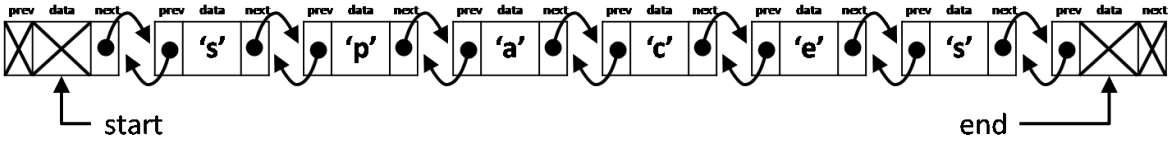
1 Doubly linked lists with cursors

We have seen singly-linked lists used to represent stacks and queues — sequences of nodes, each node containing some data and a pointer to the next node. The nodes in a *doubly-linked list* contain instead *two* pointers: one to the next element (`next`) and one to the *previous* element (`prev`). They have a `data` field just like those of a singly-linked list.

We will use doubly-linked lists to hold the contents of an editor. An *editable sequence* is represented in memory by a doubly-linked list and two pointers: one to the `start` of the sequence, one to the `end` of the sequence. We employ our usual trick of terminating the list (on both ends!) with "dummy" nodes whose contents we never inspect.

```
typedef struct dll_node dll;
struct dll_node {
    dll* prev;
    char data;
    dll* next;
};
```

The following doubly linked list contains the text "spaces":



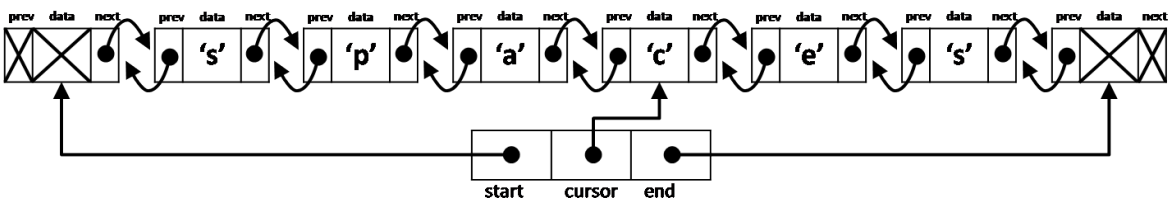
To use a doubly linked list as a text buffer, we need to add a *cursor*, which represents the position where edits can be made in the buffer. In an Emacs-like interface, the cursor frequently appears as a black rectangle, so that if the cursor was pointing to the node containing the character `'e'`, we'd see it displayed as

spaces

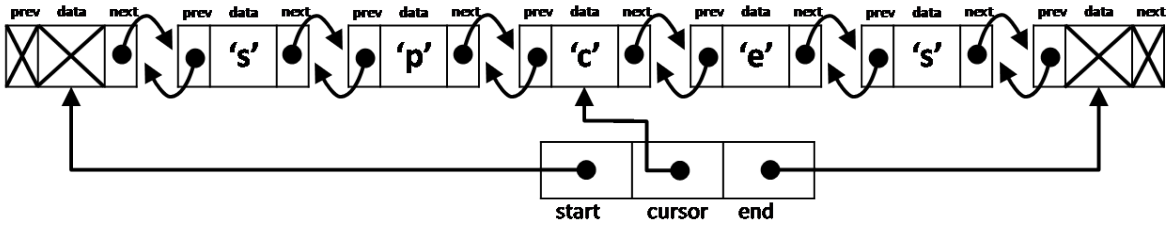
Pressing the left arrow key in a text editor moves the cursor one character backward (to the left).

spaces

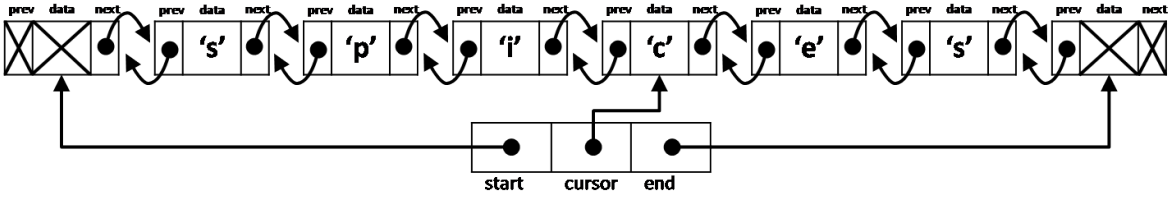
We can now draw the linked list corresponding to this text buffer along with its header, which contains `start`, `cursor`, and `end` fields:



Edits in a text buffer take place to the left of the cursor. If we press the "delete" key in the previous picture, it will delete the character to the left of the cursor:



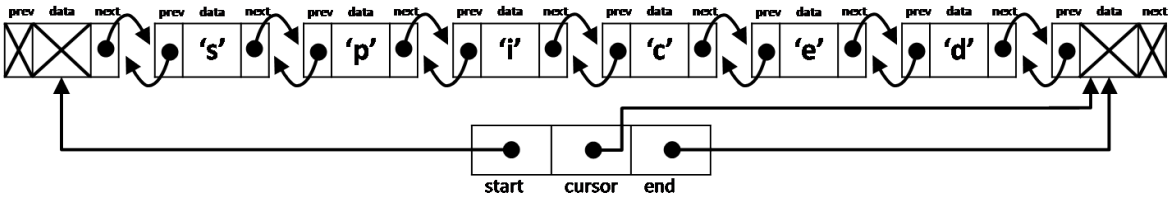
Insertions also happen to the left of the cursor. If we next typed the "i" key, that character would be entered in to the left of the cursor.



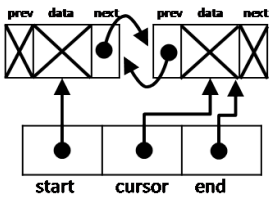
One consequence of this design is that, in order for additions and deletions to be made to the end of the buffer, the cursor needs to be able to go to the right of all the text. In other words, it must be possible for the `cursor` field to point to the same node that `end` points to. Starting from the buffer above, we can see what that looks like from the editor's point of view:

spi**c**es	(hit the right arrow key to move the cursor forward...)
spic**e**s	(hit the right arrow key to move the cursor forward...)
spice**s**	(hit the right arrow key to move the cursor forward...)
spices	(hit delete...)
spice	(type "d"...)
spiced	

As a doubly-linked list, this final buffer looks like this:



A new, empty text buffer `B` containing no text has a header and two nodes and the cursor at the `end`:



1.1 Data structure invariants

We use doubly-linked lists to implement *text buffers*, which have type `tbuf*`.

```
typedef struct tbuf_header tbuf;
struct tbuf_header {
    dll* start;
    dll* cursor;
    dll* end;
};
```

A valid text buffer is a non-`NULL` `tbuf*` with the following properties:

- the `next` links proceed from the `start` node to the `end` node, with the `cursor` node included in the segment from `start` (exclusive) to `end` (inclusive).
- the `prev` links mirror the `next` links.

These are our *linking invariants*.

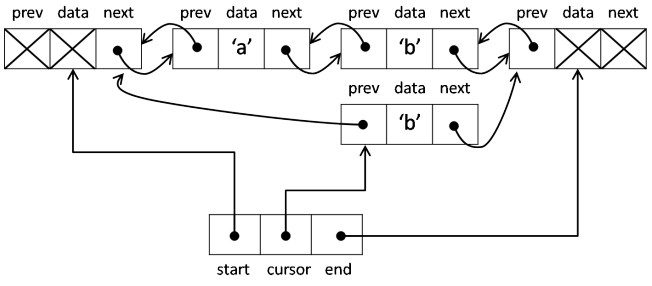
Task 1 (5 points) In the file `tbuf.c0`, implement the function

```
bool is_tbuf(tbuf* B)
```

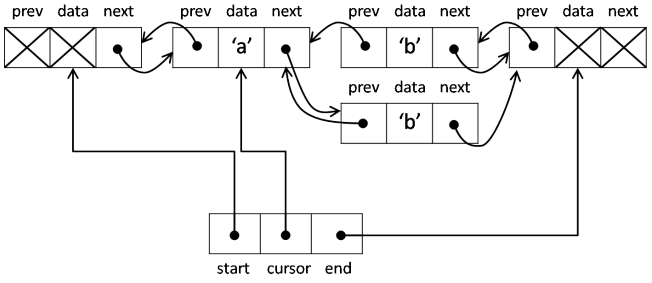
that formalizes the linking invariants on a doubly-linked list text with a cursor.

You may find that writing a helper function `bool is_dll_segment(dll* a, dll* b)` will help you implement `is_tbuf`. You are not required to check for circularity, but you may find it to be a useful exercise (it's actually easier for a doubly linked list than for singly-linked ones).

This task is not trivial. There are many ways for a doubly-linked list to be invalid, even without circularity. For instance, your `is_tbuf` function will be tested against structures with `NULL` pointers in various locations and against almost-correct doubly-linked lists:



Not a doubly-linked list (the `cursor` isn't on the path from `start` to `end`).



Not a doubly-linked list (the `prev` links don't mirror the `next` links).

You can test your `is_tbuf` by writing test cases in `test-is_tbuf.c0`. It is highly recommended to test extensively as the rest of the assignment will rely on `is_tbuf` as a data structure invariant and having a working `is_tbuf` will help with debugging for the rest of the assignment.

1.2 Manipulating doubly-linked lists

Task 2 (9 points) Implement the following utility functions on doubly-linked lists with cursors:

Function:	**Returns true iff...**
`bool tbuf_at_left(tbuf* B)`	...the cursor is as far left as possible
`bool tbuf_at_right(tbuf* B)`	...the cursor is as far right as possible

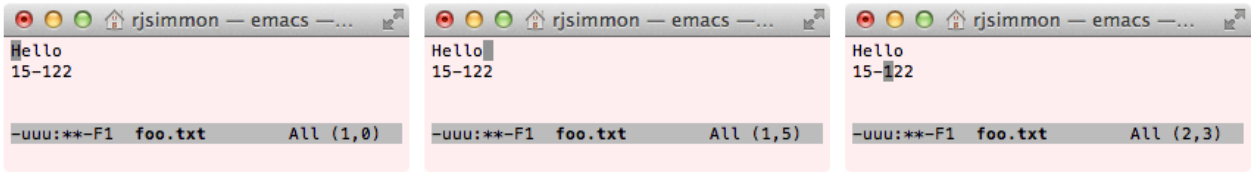
and the following interface functions for manipulating doubly-linked lists with cursors:

`tbuf* tbuf_new()`	Create a new and empty text buffer
`void tbuf_forward(tbuf* B)`	Move the cursor forward, to the right
`void tbuf_backward(tbuf* B)`	Move the cursor backward, to the left
`char tbuf_delete(tbuf* B)`	Remove the node to the cursor's left (should return the deleted `char`)
`void tbuf_insert(tbuf* B, char c)`	Insert `c` to the cursor's left

If an operation cannot be performed, a contract (precondition) should fail. These functions should require and preserve the data structure invariant you wrote above, and you should both document this fact and use it to help write the code.

1.3 Rows and columns

Up until now, nothing that we've written has been specific to doubly-linked lists of *characters*. One thing we care a great deal about in a text editor is which characters are newlines, because that is what lets us know our position in the document: the row and column. In Emacs (and the remaining tasks of this assignment), the first row is row 1, and the first column is column 0. In these Emacs buffers, you can see the (row, column) displayed in the lower right corner:



We can calculate the column of the cursor by working backwards until we find a newline, and we can calculate the row of the cursor by working backwards to the beginning of the buffer and counting the newlines. Note that in the middle example, the cursor is atop a cell containing a newline `'\n'`, but the cursor is at the end of row 1, not the beginning of row 2.

Task 3 (2 points) Implement the following functions on doubly-linked lists with cursors:

Function:	
`int tbuf_row(tbuf* B)`	Return the row of the cursor
`int tbuf_col(tbuf* B)`	Return the column of the cursor

These functions should not modify the text buffer. Furthermore, they should be as efficient as possible, only scanning necessary portions of the doubly-linked list.

1.4 Testing

You can test your doubly-linked-list implementation interactively by compiling and running the provided `tbuf-test.c0`, which treats elements as C0 characters as in the illustration above. You do not need to modify this file (nor `test-main.c0`), but you are encouraged to use it as a starting point for writing your own unit tests.

```
% cc0 -d -w -o tbuf-test tbuf.c0 tbuf-test.c0 test-main.c0
% ./tbuf-test
Visualizing an initially-empty text buffer.
The '<' character mimics going backwards (left arrow key)
The '>' character mimics going forwards (right arrow key)
The '^' character mimics deletion (delete key)
The '@' character mimics a newline (enter key)
All other characters just insert that character

Give initial input (empty line quits):
```

Suggestion: try entering "`steady^<<<<^>>^>>^@<<@^^`" the initial input.

2 Editor

One desirable feature of a text editor is the ability to report the current row and column. (You can display this information in Emacs by typing "`M-x line-number-mode`" and "`M-x column-number-mode`".) In this section, we'll introduce an `editor` type which keeps track of both the cursor and the row and column position.

Remember: You've already implemented your code for doubly-linked lists with cursors, and hopefully you've tested that implementation extensively. Avoid re-implementing features in this part of the assignment! Whenever possible, reuse the functions you've already implemented.

2.1 Data structure invariants

An editor keeps track of both the text buffer and **`int`** fields storing the row and column.

```
typedef struct editor_header editor;
struct editor_header {
    tbuf* buffer;
    int row;
    int col;
};
```

A valid editor data structure is non-`NULL`, has a valid text buffer, and has recorded `row` and `col` fields that match the information returned by the `tbuf_row` and the `tbuf_col` functions.

Task 4 (3 points) In the file `editor.c0`, implement the function

```
bool is_editor(editor* E)
```

that formalizes the data structure invariants on an editor.

2.2 Manipulating text in the editor

By tracking the row and column in the data structure, we can report this information to the user without ever having to recalculate the row using `tbuf_row`. It's good to avoid this, because `tbuf_row` can be expensive to run. We will sometimes, but only rarely, need to recalculate the column by calling `tbuf_col`. Any single row is usually relatively short (80 columns maximum, if you're using good style), so this should be fast.

Specifically, we only need to recalculate the column by calling `tbuf_col` when we delete a newline character or move left from the beginning of one line to the end of the previous line.

Task 5 (6 points) Efficiently implement the following interface functions for manipulating editors:

`editor* editor_new()`	Create a new and empty editor
`void editor_forward(editor* E)`	Move the cursor forward, to the right
`void editor_backward(editor* E)`	Move the cursor backward, to the left
`void editor_delete(editor* E)`	Remove the node to the cursor's left
`void editor_insert(editor* E, char c)`	Insert `c` to the cursor's left

These functions directly respond to a user's input. That means that if an operation cannot be performed (e.g., pressing the "left" key to move the cursor backward with `editor_backward` when it's already at the left end), the function should leave the text buffer **unchanged** instead of raising an error or assertion violation.

2.3 Testing

The same testing program you used for testing your `tbuf` implementation can also be used for this part of the assignment:

```
% cc0 -d -w -o editor-test tbuf.c0 editor.c0 editor-test.c0 test-main.c0
% ./editor-test
```

You can additionally test your code with a visual text editor designed by legendary former TA and instructor William Lovas:

```
% cc0 -d -w -o E0 tbuf.c0 editor.c0 lovas-E0.c0
% ./E0
```

Task 6 (bonus) In `editor.c0`, implement the functions `editor_up` and `editor_down`. They will enable you to use the up and down arrow keys to move around the `E0` text editor more effectively.

The up and down keys should take you up or down a row unless you're on the first or last row (respectively) already. If possible, this operation should leave you on the same column as before. This is not possible when the row you've landed on has too few columns. In that case, you should leave the cursor at the rightmost end of the row.