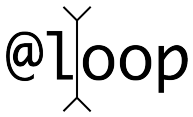
COMP 211: Principles of Imperative Computation, Fall 2014

Programming 8: Editor II

Due: Wednesday, November 19, 2014 by 11:59pm



For the programming portion of this week's homework, you will put the previous two weeks' assignments together into a better text editor.

1 Getting the code

The code handout for this assignment is available from the course web page. The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in.

To download and unzip the code, do the following:

1.1 Mac OS

1. On the assignments page of the course web site, right-click the Programming 8 `Code` link, and then click "Save Link as...", and choose to save it in `Documents/comp211/`.

2. Start a new terminal window and do the following:

   ```
   cd comp211
   tar -xzvkf hw8-handout.tgz
   ```

   This will make a directory called `comp211/hw8-handout` and extract all of the code for the assignment.

3. The file `hw8-handout/README.txt` contains a description of the various files, and instructions on running them.

1.2 Windows

1. On the assignments page of the course web site, right-click the Programming 8 `Code` link, and then click "Save Link as...", and choose to save it in your `comp211/`. To find this directory in your web browser, look in

   ```
   C:\cygwin\home\YOURNAME\comp211\
   ```

2. Start a new terminal window and do the following:

```
cd comp211
tar -xzkvf hw8-handout.tgz
```

This will make a directory called `comp211/hw8-handout` and extract all of the code for the assignment.

3. The file `hw8-handout/README.txt` contains a description of the various files, and instructions on running them.

2 With our powers combined

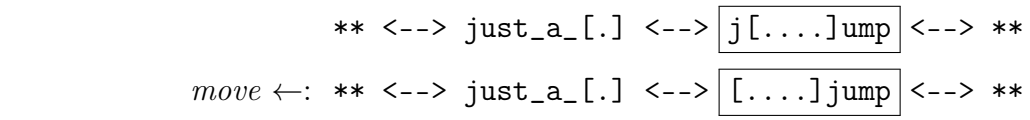
Implementing a text editor as just one gap buffer, as you did two weeks ago, is not particularly realistic, because it requires the entire file contents to be stored in a single, contiguous block of memory, which can be difficult to achieve for large files. Implementing a text buffer as a doubly-linked list of characters, as you did last week, also has its problems: representing a single ASCII character requires only 8 bits, but storing that character in a doubly linked list node would require an additional 64 bits—an amount of overhead that could be significant.

In this assignment, you will combine these two ideas, to get the best of both words. A text buffer will be represented by *a doubly linked list of gap buffers*. This way, each individual gap buffer can be small, avoiding the problem with allocating the whole document contiguously, but you can always add more space to the document by adding another link to the doubly-linked list. Moreover, the overhead of the doubly-linked list nodes is shared by all of the characters in the gap buffer—we'll use 16 characters, so there is only 4 bits of overhead per character instead of 64, but this can easily be tuned by making the gap buffers larger.

This assignment asks you to combine two data structures into a single program. This is a very important programming skill, because big programs are generally written by combining such pre-existing components. You are allowed to use the representation of gap buffers and doubly linked lists—you do **not** need to only use the functions defined in your `gapbuf.c0` and `dll_pt.c0` . **However, you should try to use the interface functions as much as possible.** In particular, you should never need to *modify* a DLL except by using the functions provided by `dll_pt.c0`, so if you find yourself manipulating DLLs directly, think about how to use one of these functions instead.

3 Text Buffers

The *text buffers* used by our text editor as a doubly-linked list of fixed-size gap buffers (each buffer is 16 characters long). The contents of a text buffer represented in this way is simply the concatenation of the contents of its requisite gap buffers, in order from the `start` to the `end`. The gap representing the text editor's cursor is the particular gap at the linked list's `point`. To move the cursor, we use a combination of gap buffer motion and doubly-linked list motion:



move ←: `** <--> [just_a[.]_] <--> [....]jump <--> **`

3.1 Text buffer invariants

There are a lot of invariants that we want to check in this representation. Two very simple ones are that our text buffers should be a valid linked lists (`is_dll_pt`), and each element in the linked list should be a gap buffer (`is_gapbuf`).

Another invariant that arises from this representation is that a text buffer must either be the empty text buffer consisting of a single empty gap buffer like so:

`START <--> [................] <--> END`

or else all the gap buffers must be non-empty. Additionally, all the gap buffers are themselves well-formed, and they all have the same size, and the size is 16. (We used gap buffers of size 8 for simplicity in some of the examples, but you must use 16 in your implementation.)

Another important invariant is *alignment*. It is easiest to observe on larger cases:

`** <--> well_i[..] <--> sn't_[...] <--> [this_[..]l] <--> [.....]ong <--> **`

Notice that for all gap buffers to the left of the point, the gap is on the right. Similarly, for all gap buffers to the right of the point, the gap is on the left. We call this invariant alignment. If alignment fails to hold, we will have a very hard time moving the point between gap buffers. If we had this text buffer instead:

`** <--> well_i[..] <--> sn't_[...] <--> [this_[..]l] <--> ong[.....] <--> **`

then, as we move to the right,

```
** <--> well_i[..] <--> sn't_[...] <--> [this_l[..]] <--> ong[.....] <--> **
** <--> well_i[..] <--> sn't_[...] <--> this_l[..] <--> [ong[.....]] <--> **
```

we find that the cursor suddenly jumps to the end of the buffer, skipping over "ong" entirely.

Task 1 (6 pts) *A valid text buffer satisfies all the invariants described above: it is a valid doubly-linked list containing valid size-16 gap buffers, it is aligned, and it consists of either one empty gap buffer or one or more non-empty gap buffers. Implement the function*

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

that formalizes the text buffer data structure invariants.

Hint: you may find it easier to decompose `is_tbuf` into multiple functions (such as one that checks alignment and one that checks that the one-empty-or-all-nonempty property). In particular, if you define a function `bool empty_tbuf(dll_pt B)` as part of this task, you will find it helpful below.

You should write a function

```
void test_is_tbuf()
```

that creates some valid text buffers and some invalid text buffers and tests `is_tbuf` on them. You should use the `dll_pt` functions to create and manipulate the doubly-linked list.

3.2 Creating text buffers

Task 2 (2 pts) *Implement the following text buffer constructor:*

`tbuf tbuf_new()` *Construct a new, empty text buffer with a gap-buffer-size of 16*

3.3 Split and Insert

Recall that in order to be aligned, a text buffer must satisfy: all gap buffers to the left of ("before") the point must have their gaps on the right, and all gap buffers to the right of ("after") the point must have their gaps on the left. Alignment specifies nothing about the properties of the point itself.

To insert into the buffer (of the point node), we have to check if the buffer is full or not. When a gap buffer is full, we split the point node into two nodes. The data in the buffer will be split as well:

```
                ** <--> [splitend[]] <--> **
insert 's': ** <--> spli[....] <--> [tends[...]] <--> **
```

One way to implement this is as follows: first, create two new gap buffers, and copy each half of the character data into the new gap buffers, taking special note of where the new gaps should end up. The following diagrams may help you visualize the intended result:

```
full buffer: [abc[]defghABCDEFGH]

splits into: [abc[........]defgh]
              [........]ABCDEFGH
```

```
full buffer: [stuvwxyzSTUV[]WXYZ]

splits into: stuvwxyz[........]
             [STUV[........]WXYZ]
```

Then create two new nodes in the list, on either side of the point, with these new gap buffers in them, and delete the point.

It is also possible to implement this by only adding one new node, and modifying the gap buffer of the point node in place.

Task 3 (6 pts) *Implement a function* `tbuf_split_pt(tbuf B)` *which takes a valid text buffer whose point is full and turns it into a valid text buffer whose point is not full.*

Then implement

`void tbuf_insert(tbuf B, char c)` *Insert the character c before the cursor*

3.4 Delete and moving

To delete from the buffer we use the gap buffer's `gapbuf_delete` function and when one the buffer becomes empty, we delete it:

```
                START <--> deletio[.] <--> [n[........]] <--> END
delete: START <--> [deletio[.]] <--> END
```

Task 4 (6 pts) *Implement the following interface functions for manipulating text buffers:*

`void tbuf_forward(tbuf B)`	*Move the cursor forward, to the right*
`void tbuf_backward(tbuf B)`	*Move the cursor backward, to the left*
`void tbuf_delete(tbuf B)`	*Delete the character before the cursor*

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* `tbuf_backward` *when it's already at the left end), the function should leave the text buffer* **unchanged** *instead of raising an error or having a requires violation. This is different than in past weeks, when the anlogous functions were supposed to have pre-conditions that ensured they would always work.*

3.5 Testing

You can test your implementation by compiling and running the provided `tbuf-test.c0` test driver which prints a visual representation of the internal data of a text buffer.

```
% cc0 -d -o tbuf-test gapbuf.c0 dll_pt.c0 tbuf.c0 visuals.c0 tbuf-test.c0
% ./tbuf-test
     START <--> _[................]_ <--> END
'a': START <--> _a[...............]_ <--> END
'b': START <--> _ab[..............]_ <--> END
...
```

This runs a fixed series of operations. The expected output is stored in `expected.txt`.

You can also test interactively by compiling `E0`, a minimalist text editor front-end.

```
% cc0 -d -o E0 gapbuf.c0 dll_pt.c0 tbuf.c0 lovas-E0.c0
% ./E0
```

Enjoy the hard-won fruits of your careful programming labors!