15-122 : Principles of Imperative Computation

Summer 1 2012

Assignment 4: Text Editor

(Programming Part)

Due: Monday, June 11, 2012 by 23:59

For the programming portion of this week's homework, you'll implement the core data structure for a text editor: the gap buffer. You will write three C_0 files corresponding to different tasks:

- `gap-buffer.c0` (described in Section 1.2),
- `text-buffer.c0` (described in Section 1.3)
- `text-editor.c0` (described in Section 1.4)

Be sure to *read the specifications carefully*—you'll thank yourself later!

You should submit these two files electronically by the due date. Detailed submission instructions can be found below.

1 Assignment: Text Editor (30 points)

Starter code. Download the file `hw4.zip` from the course website. When you unzip it, you will find the following files

`gap-buffer.c0`	Gap buffer data structure
`text-buffer.c0`	Doubly Linked List of gap buffers
`text-editor.c0`	Text editing functionalities
`hw4-main.c0`	The driver program for testing
`visuals.c0`	Visual representation (required by hw4-main.c0)
`expected.txt`	Expected output
`lovas-E0.c0`	E0 editor

Compiling and running. For this homework, use the `cc0` command as usual to compile your code. Don't forget to test your annotations by compiling with the `-d` switch to enable dynamic checking. **Warning:** *You will lose credit if your code does not compile.*

Submitting. Once you have completed some files, you can submit them by running the command

```
handin -a hw4 gap-buffer.c0 text-buffer.c0 text-editor.c0
```

The `handin` utility accepts a number of other switches you may find useful as well; try `handin -h` for more information.

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. Be sure to include appropriate `//@requires`, `//@ensures`, `//@assert`, and `//@loop_invariant` annotations in your program. You should write these as you are writing the code rather than after you're done: documenting your code as you go along will help you reason about what it should be doing, and thus help you write code that is both clearer and more correct. **Annotations are part of your score for the programming problems; you will not receive maximum credit if your annotations 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.

1.1 Gap Buffer Text Editor: Overview

In this assignment you will implement a simple text editor based on the *gap buffer* technique. A gap buffer is a generalization of an unbounded array: whereas an unbounded array allows for efficient insertion and deletion of elements from the end only, a gap buffer allows efficient insertion and deletion of elements from the middle. Appending items to the end of the array requires very little work, because this means simply placing it in the next unused index and increasing the size. However, there is no unused space in the middle. The only way to place a new item somewhere in the middle is to shift elements over to make room for the new item. A gap buffer attempts to overcome the overhead of shifting by placing the empty portion of the array in the middle. Hence the name "gap buffer" referring to the "gap" in the middle of the "buffer". The gap is not fixed to any one position at all. At any time it could be in the middle of the buffer or just at the beginning or anywhere in the buffer. We can immediately see the potential benefits of this approach. Moving the **cursor** in the text editor we are automatically moving a gap, always providing the unused portion of the array to be used for possible insertions. At worse case we have to move the gap from the beginning of the text file to the another end. But if subsequent operations are only a few indexes apart we will get a lot better performance comparing to a dynamic array. This is why it's said that the gap buffer technique increases performance when repetitive operations occurring at relatively close indexes. We claim without proof that the amortized cost of insertion into the gap buffer is constant.

Implementing a text editor as just one gap buffer is not particularly realistic. One large edit buffer requires the entire file contents to be stored in a single, contiguous block of memory, which can be difficult to allocate for large files. Instead, a more realistic strategy is to combine the gap buffer technique with a doubly linked list. The benefit of a linked list is that it allows the file to be split across several chunks of memory. Therefore, in this assignment we will represent a text editor as a doubly linked list where each node contains a fixed-size (16 characters) gap buffer. The contents of a text file represented in this way is simply the concatenation of the contents of each gap buffer in the linked list.

1.2 Gap Buffer

A gap buffer is an array of characters. The array is logically split into two segments of text - one at the beginning of the array, which grows upwards; and one at the end which grows downwards. The space between those two segments is called the *gap*. The gap is the **cursor** in the buffer. To move the cursor forwards, you just move the gap (assuming that the gap is not empty). To insert a character after the cursor, you place it at the beginning of the gap. To delete the character before the cursor, you just expand the gap.

To implement a gap buffer there are a couple bits of information that we need to keep track of. A gap buffer is represented in memory by an array of elements stored along with its size (limit) and two integers representing the beginning (inclusive, gap_start) and end (exclusive, gap_end) of the gap (see Figure 1).

```
typedef struct gap_buffer* gapbuf;
struct gap_buffer {
    int limit;          /* limit > 0 */
    char[] buffer;      /* \length(buffer) == limit */
    int gap_start;      /* 0 <= gap_start        */
    int gap_end;        /* gap_start <= gap_end <= limit */
};
```

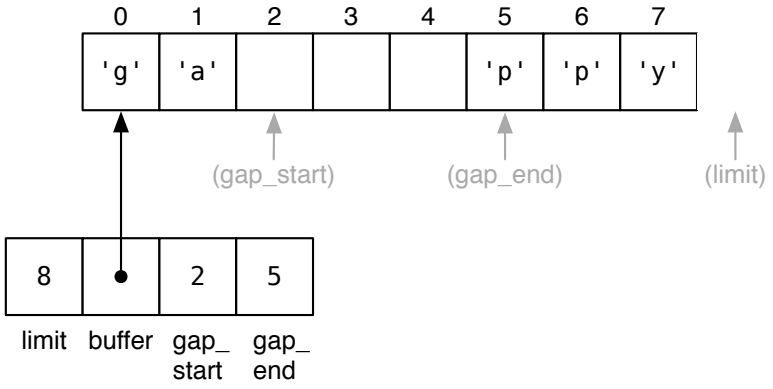

Figure 1: A gap buffer in memory.

Task 1 (2 pts) *Implement the function* ***is_gapbuf*** *as specified below:*

```
bool is_gapbuf(gapbuf G)
```

that formalizes the gap buffer data structure invariants. A valid gap buffer is non-NULL, has a strictly positive limit which correctly describes the size of its array, and has a gap start and gap end which are valid for the array.

Text buffers may allow a variety of editing operations to be performed on them; for the purposes of this assignment, we'll consider only four operations: move forward a character, move backward a character, insert a character, and delete a character. As an example, below is a diagram of a gap buffer which is an array of characters with a gap in the middle (situated between the "s" and the "p" in "space"):

```
the s[...]pace race
```

To move the gap (the cursor in the text editor) forward, we copy a character across it:

```
the sp[...]ace race
```

To delete a character (before the cursor), we simply expand the gap:

```
the s[....]ace race
```

To insert a character (say, 'p'), we write it into the gap (shrinking it by one):

```
the sp[...]ace race
```

The gap can be at the left end of the buffer,

```
[...]the space race
```

or at the right end of the buffer,

```
the space race[...]
```

and a buffer can be empty,

```
[................]
```

or it can be full (this depends on the buffer size (limit))

```
the space ra[]ce
```

Task 2 (4 pts) *Implement the following utility functions on gap buffers:*

Function:	***Returns true iff...***
`bool gapbuf_empty(gapbuf G)`	*the gap buffer is empty*
`bool gapbuf_full(gapbuf G)`	*the gap buffer is full*
`bool gapbuf_at_left(gapbuf G)`	*the gap is at the left end of the buffer*
`bool gapbuf_at_right(gapbuf G)`	*the gap is at the right end of the buffer*

Task 3 (5 pts) *Implement the following interface functions for manipulating gap buffers:*

`gapbuf gapbuf_new(int limit)`	*Create a new gapbuf of size limit*
`void gapbuf_forward(gapbuf G)`	*Move the gap forward, to the right*
`void gapbuf_backward(gapbuf G)`	*Move the gap backward, to the left*
`void gapbuf_insert(gapbuf G, char c)`	*Insert the character c before the gap*
`void gapbuf_delete(gapbuf G)`	*Delete the character before the gap*

If an operation cannot be performed (e.g., moving the gap backward when it's already at the left end), it should leave the gap buffer unchanged.

All functions should require and ensure the data structure invariants. Furthermore, the gap buffer returned by `gapbuf_new` should be empty. Use these facts to help you write your code, and document them with appropriate assertions.

1.3 Doubly-Linked Lists

Another data structure that will be used to represent an edit buffer is a *doubly-linked list*. 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 of a doubly-linked list contain a data field just like those of a singly-linked list, but in contrast, the doubly-linked nodes contain *two* pointers: one to the next element (next) and one to the *previous* (prev).

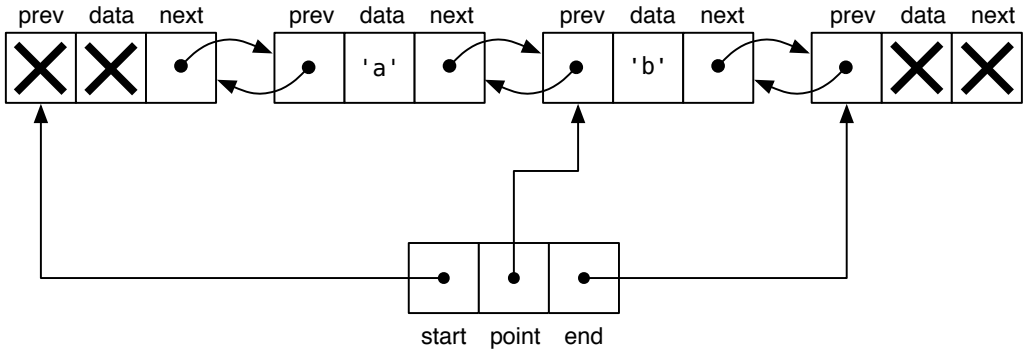

Figure 2: An editable sequence as a doubly-linked list in memory.

An editable sequence is represented in memory by a doubly-linked list and three pointers: one to the start of the sequence, one to the end of the sequence, and one to the distinguished point node where updates may take place (see Figure 2). We employ our usual trick of terminating the list with "dummy" nodes whose contents we never inspect.

```
typedef struct list_node* dll;
struct list_node {
    elem data;
    dll next;
    dll prev;
};

typedef struct text_buffer* tbuf;
struct text_buffer{
    dll start;
    dll point;
    dll end;
};
```

We can visualize a doubly-linked list as the sequence of its data elements with terminator nodes at both ends and one distinguished element, called point:

```
START <--> 'a' <--> ['b'] <--> END
```

For now, we do not concern ourselves with the type of the data elements: basic doubly-linked list functions are agnostic to it anyway.

Task 4 (3 pts) *Implement the function* ***`is_linked`*** *as specified below:*

```
bool is_linked(tbuf B)
```

that formalizes the linking invariants on a doubly-linked list text buffer. The key invariant in a well-formed doubly-linked list is that the next *links proceed from the* start *node to the* end *node, passing* point *node along the way, and that the* prev *links mirror the* next *links. Additionally, we require that the* point *be always present in the list and be a distinct node from both the* start *and the* end *nodes, i.e., that the list be non-empty. You are not required to check for circularity, but you may find it to be a useful exercise.*

Task 5 (5 pts) *Implement the following utility functions on doubly-linked text buffers:*

Function:	***Returns true iff...***
`bool tbuf_at_left(tbuf B)`	*the point is at the far left end*
`bool tbuf_at_right(tbuf B)`	*the point is at the far right end*

and the following interface functions for manipulating doubly-linked text buffers:

`void tbuf_forward(tbuf B)`	*Move the point forward, to the right*
`void tbuf_backward(tbuf B)`	*Move the point backward, to the left*
`void tbuf_delete_point(tbuf B)`	*Remove the point node from the list*

As above, if an operation cannot be performed, it should leave the text buffer unchanged. When deleting the point, the new point may be either to the right or to the left of the old one.

These functions should require and preserve the linking invariant you wrote above, and you should both document this fact and use it to help write the code. Be especially careful when implementing deletion! Note, we cannot delete the point if it is the only non-terminator node.

1.4 Putting It Together

Now we are putting two pieces together. We will implement a 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. To move the cursor, we use a combination of gap buffer motion and doubly-linked list motion:

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

An invariant that arises from this representation is that a text buffer must be linked and either be the empty text buffer:

```
                  ** <--> [................] <--> **
```

or all the gap buffers must be non-empty. Additionally, all the gap buffers are themselves well-formed, and they all have the same size (you should use 16 in your implementation).

Task 6 (3 pts) *Implement the following specification functions on text buffers:*

Function:	***Returns true iff...***
`bool tbuf_empty(tbuf B)`	*the text buffer is empty*
`bool is_tbuf(tbuf B)`	*the text buffer satisfies all invariants*

and a text buffer constructor:

`tbuf tbuf_new()`	*constructs a new, empty text buffer*

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[...]]<--> **`

To split a full gap buffer, we have to copy each half of the character data into one of two 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]`

We can then link the new gap buffers into the doubly-linked list, taking care to preserve the text buffer invariants.

Task 7 (4 pts) *Implement a function* `split_point(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.*

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

```
** <--> deletio[.] <--> [n[........]] <--> **
```

delete: `** <--> [deletio[.]] <--> **`

Task 8 (4 pts) *Implement the following interface functions for manipulating text buffers:*

`void forward_char(tbuf B)`	*Move the point forward, to the right*
`void backward_char(tbuf B)`	*Move the point backward, to the left*
`void insert_char(char c, tbuf B)`	*Insert the character c before the point*
`void delete_char(tbuf B)`	*Delete the character before the point*

If an operation cannot be performed (e.g., moving the point backward when it's already at the left end), it should leave the text buffer unchanged.

1.5 Testing

For the resting purposes we provided `hw4-main.c0` test driver which prints a visual representation of the internal data of a text buffer. The expected output is stored in `expected.c0`. Using this driver you can test either your complete implementation or each function independently. The testing is based on printing functionalities implemented in `visuals.c0`.

After you've completed your text buffer implementation and tested it thoroughly, you can try it out interactively by compiling against `lovas-E0.c0` - a minimalist text editor front-end written by William Lovas.

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