15-122: Principles of Imperative Computation, Fall 2015

Programming homework 6: Text Editor

@loop

Checkpoint: Thursday, October 22, 2015 by 22:00
Due: Thursday, October 29, 2015 by 22:00

For this programming assignment, you'll implement the core data structures for a text editor. There are *two* deadlines: the first one is to make sure you've started, and will give you your first 10 points on the assignment, which must be earned at this time. At the second, final deadline, the autograder will re-run the tests from the first part of the assignment, but will not award points. *The checkpoint represents much less than half the assignment in terms of both lines of code and conceptual difficulty.*

The code handout for this assignment is at

`http://www.cs.cmu.edu/~fp/courses/15122-f15/assignments/editor-handout.tgz`

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 FIVE (5) PENALTY-FREE HANDIN LIMIT for the checkpoint, and another TEN (10) PENALTY-FREE HANDIN LIMIT for the second half of the assignment. Every additional handin will incur a very small (0.2 point) penalty. It is advisable to budget a few handins specifically to get feedback on your specification functions; having good specification functions will help you as you write the rest of your code.

Task 0 - Style (0 pts) With this assignment, we will again emphasize *programming style* more heavily. We will actually be looking at your code and evaluating it based on the criteria outlined at `http://www.cs.cmu.edu/~rjsimmon/15122-s15/etc/styleguide.pdf`. We will make comments on your code via Autolab, and will assign an overall passing or failing style grade. A failing style grade will be temporarily represented as a score of -30 on the second part of the assignment. This -30 will be reset to 0 once you:

1. fix the style issues,
2. see a member of the course staff during office hours, and
3. briefly discuss the style issues and how they were addressed.

We will evaluate your code for style in two ways. We will use `cc0` with the `-w` flag that gives style warnings – code that raises warnings with this flag is almost certain to fail style grading. Because the `-w` flag does not check for good variable names, appropriate comments, or appropriate use of helper functions, these issues will be checked by hand.

Task 1 (7 pts) *Seven points for this assignment will be given for contracts that are not checked by the autograder. Be sure to include the preconditions, postconditions, and where necessary, loop invariants to ensure the data structure invariants and the safety of your code. More details about contracts are included below.*

1 Overview: A text editor based on gap buffers

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: although an unbounded array allows for efficient insertion and deletion of elements from the end, a gap buffer allows for efficient insertion and deletion of elements at the middle.

Given an unbounded array that is only half filled, adding items after the last item currently in the array requires very little work, because this simply means 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 in the middle is to shift elements over to make room for the new item.

A gap buffer attempts to avoid the cost of shifting by placing the empty portion of the array somewhere within the array. Hence the name “gap buffer" referring to the “gap" within the “buffer". The gap is not fixed to any one position. At any time it could be at the halfway point of the buffer or just at the beginning or anywhere in the buffer. We can immediately see the potential benefits of this approach.

When the user moves the **cursor** in the text editor, the implementation automatically moves the gap, thereby providing the unused portion of the array to be used for possible insertions. In the worst case scenario we have to move the gap from the beginning of the text file to the end. But if subsequent operations are only a few indexes apart, we will get a lot better performance compared to using a dynamic array. This is why it is said that the gap buffer technique increases performance of 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 specific size of 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.

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, and they will usually not write to the data structures themselves. However, they are allowed to manipulate the data structures, and so we expose the data structure invariants (like `is_gapbuf`) 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`.

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

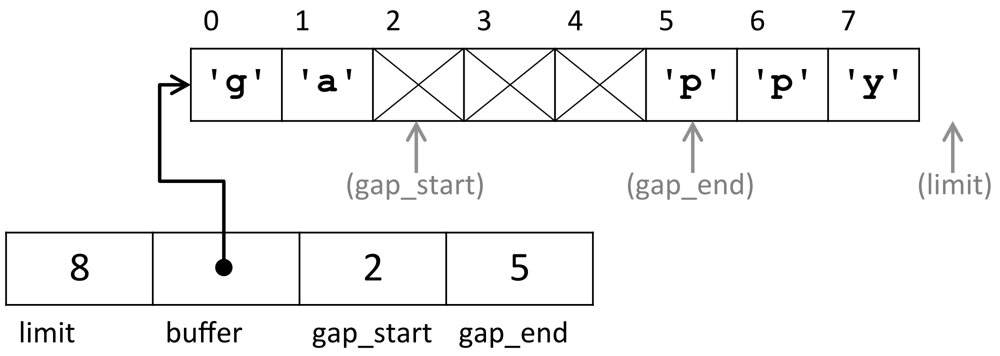

Figure 1: A gap buffer in memory.

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

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

that formalizes the gap buffer data structure invariants.

Gap 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 "p" and the "a" in "space"):

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

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

```
the spa[..]ce race
```

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

```
the sp[...]ce race
```

To insert a character (say, "i"), we write it into the left side of the gap (shrinking it by one):

```
the spi[..]ce 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!!
```

Note that in an emacs-like interface, where the cursor appears as a highlighted character in the buffer, the cursor will display on the character immediately following the gap. So following the examples above,

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

would display as:

```
the space race
```

while

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

would display as:

```
the space race█
```

In the above illustrations, we use dots to indicate spots in the gap buffer whose contents we don't care about. Those spots in the gap buffer don't need to contain the default character `'\0'` or the character `'.'` or anything else in particular.

Task 3 (3 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 4 (4 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*

See page 5 for details. If an operation cannot be performed (e.g., moving the gap backward when it's already at the left end), a contract (precondition) should fail.

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.

2.1 Testing

You can test your gap buffer implementation interactively by compiling and running the provided `gapbuf-test.c0`; you are encouraged to use this file as a starting point for writing your own unit tests.

```
 % cc0 -d -w -o gapbuf-test gapbuf.c0 gapbuf-test.c0
 % ./gapbuf-test
```

Try entering "`space race<<<<<<<<^p<<the >>^p>>>>>>>>!!<<<<<<<^^^^^great`" and seeing what happens. (It is okay to share strings like this on Piazza.)

After you have tested your gap buffer implementation, you can hand it in before the checkpoint deadline to earn your first ten points on the assignment. No further points will be awarded for these tasks after the checkpoint deadline passes.

3 Doubly-Linked Lists

Another data structure that will be used to represent an edit buffer is a *doubly-linked list with a point.* 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 as we will use them in this assignment 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`).

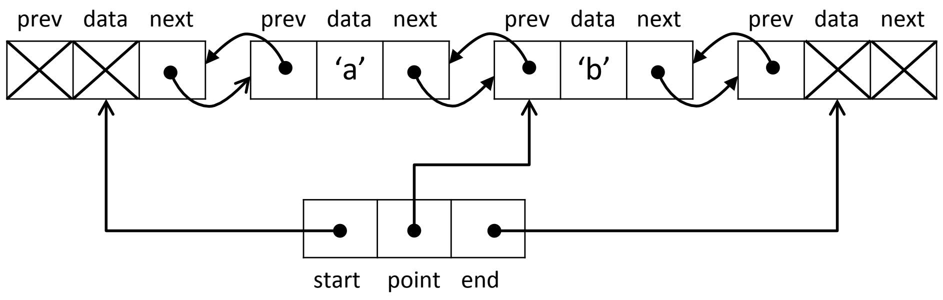

Figure 2: An editable sequence as a doubly-linked list in memory.

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 dll_node dll;
struct dll_node {
    elem data;
    dll* next;
    dll* prev;
};

typedef struct dll_pt_header dll_pt;
struct dll_pt_header {
    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' <-->` $\boxed{\texttt{'b'}}$ `<--> 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. (The picture above treats the data elements as C0 characters.)

Task 5 (5 pts) *A valid doubly-linked list has the following properties:*

- *the* `next` *links proceed from the* `start` *node to the* `end` *node, passing* `point` *node along the way*
- *the* `prev` *links mirror the* `next` *links*
- `point` *is distinct from both the* `start` *and the* `end` *nodes, i.e., the list is non-empty*

Implement the function

```
bool is_dll_pt(dll_pt* B)
```

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

You may find that writing a helper function `bool is_dll_segment(dll* a, dll* b)` *will help you implement* `is_dll_pt`*. 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_dll_pt` function will be tested against structures with `NULL` pointers in various locations and against almost-correct doubly-linked lists:

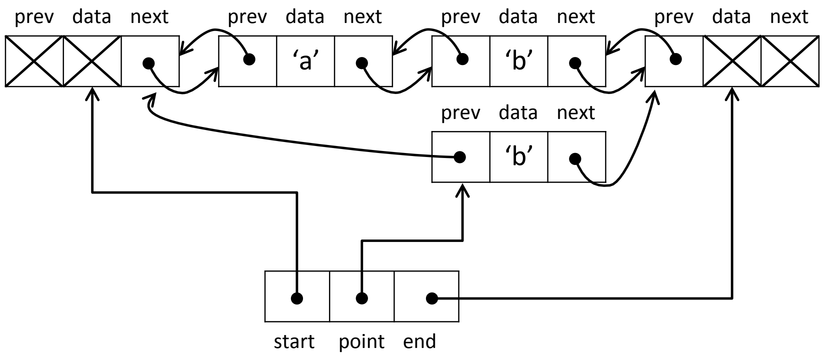

Figure 3: Not a doubly-linked list (the `point` isn't on the path from `start` to `end`).

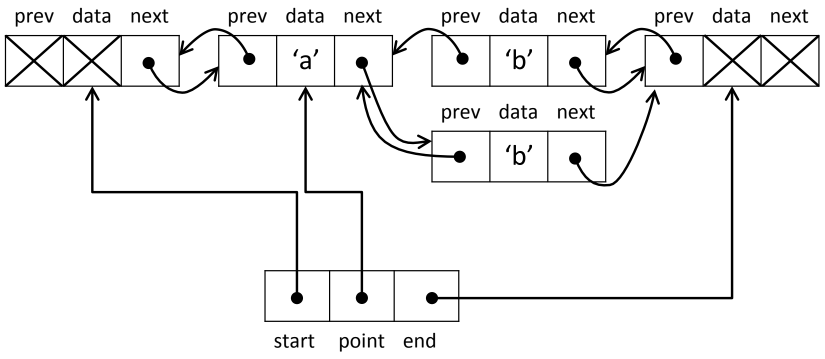

Figure 4: Not a doubly-linked list (the `prev` links don't mirror the `next` links).

Task 6 (4 pts) *Implement the following utility functions on doubly-linked lists with points:*

Function:	***Returns true iff...***
`bool dll_pt_at_left(dll_pt* B)`	*. . . the point is at the far left end*
`bool dll_pt_at_right(dll_pt* B)`	*. . . the point is at the far right end*

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

`void dll_pt_forward(dll_pt* B)`	*Move the point forward, to the right*
`void dll_pt_backward(dll_pt* B)`	*Move the point backward, to the left*
`void dll_pt_delete(dll_pt* B)`	*Remove the point node from the list*

As above, if an operation cannot be performed, a contract (precondition) should fail. 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 data structure 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.

3.1 Testing

You can test your doubly-linked-list implementation interactively by compiling and running the provided `dll_pt-test.c0`, which treats elements as C0 characters as in the illustration above. You are encouraged to use this file as a starting point for writing your own unit tests.

```
% cc0 -d -w -o dll_pt-test elem-char.c0 dll_pt.c0 dll_pt-test.c0
% ./dll_pt-test
```

Try entering "`steady`" as the input word and then "`^<<<<^>>^`" as the series of actions and seeing what happens.

If you write your own test code, make sure that you put either `gapbuf.c0` (which declares the type `elem` to be `gapbuf`) or `elem-char.c0` (which declares the type `elem` to be `char`) on the command line before `dll_pt.c0`. If you try to compile your `dll_pt.c0` file without first defining what `elem` is, you will probably get an error from `cc0` that looks something like `"expected a type name, found identifier 'elem'"`, because the C0 compiler assumes `"elem"` is an identifier unless you have already used a `typedef` to explain to C0 that it is really a type name. (Note: `dll_pt-test.c0` assumes `elem` is `char`.)

4 Putting It Together

Now we will implement 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:

$$\texttt{** <--> jus[.....] <--> }\boxed{\texttt{t[..]\_jump}}\texttt{ <--> **}$$

$$\textit{move} \leftarrow\text{:}\ \texttt{** <--> jus[.....] <--> }\boxed{\texttt{[..]t\_jump}}\texttt{ <--> **}$$

$$\textit{move} \leftarrow\text{:}\ \texttt{** <--> }\boxed{\texttt{ju[.....]s}}\texttt{ <--> [..]t\_jump <--> **}$$

4.1 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 list (`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:

$$\texttt{START <--> }\boxed{\texttt{[................]}}\texttt{ <--> END}$$

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:

$$\texttt{** <--> well\_i[..] <--> sn't\_[...] <--> }\boxed{\texttt{this\_[..]l}}\texttt{ <--> [.....]ong <--> **}$$

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:

$$\texttt{** <--> well\_i[..] <--> sn't\_[...] <--> }\boxed{\texttt{this\_[..]l}}\texttt{ <--> ong[.....] <--> **}$$

then, as we move to the right,

$$\texttt{** <--> well\_i[..] <--> sn't\_[...] <--> }\boxed{\texttt{this\_l[..]}}\texttt{ <--> ong[.....] <--> **}$$

$$\texttt{** <--> well\_i[..] <--> sn't\_[...] <--> this\_l[..] <--> }\boxed{\texttt{ong[.....]}}\texttt{ <--> **}$$

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

Task 7 (4 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).

4.2 Manipulating text buffers

Task 8 (2 pts) *Implement a text buffer constructor:*

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

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:

$$\texttt{** <--> }\boxed{\texttt{splitend[]}}\texttt{ <--> **}$$

$$\textit{insert 's'}\text{: }\texttt{** <--> spli[....] <-->}\boxed{\texttt{tends[...]}}\texttt{<--> **}$$

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: $\boxed{\texttt{abc[]defghABCDEFGH}}$

splits into: $\boxed{\texttt{abc[........]defgh}}$
$\texttt{[........]ABCDEFGH}$

full buffer: $\boxed{\texttt{stuvwxyzSTUV[]WXYZ}}$

splits into: $\texttt{stuvwxyz[........]}$
$\boxed{\texttt{STUV[........]WXYZ}}$

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

Task 9 (4 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. If you find it helpful to do so, you can extend the interface to gap buffers or doubly-linked lists with points as part of your solution.*

To delete from the buffer we use the gap buffer's `gapbuf_delete` function. When one of the buffers becomes empty, we delete it unless it is the only gap buffer left:

$$\texttt{START <--> deletio[.] <-->}\boxed{\texttt{n[........]}}\texttt{<--> END}$$

$$\textit{delete}\text{: }\texttt{START <--> }\boxed{\texttt{deletio[.]}}\texttt{ <--> END}$$

Task 10 (4 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_insert(tbuf* B, char c)`	*Insert the character c before the cursor*
`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 assertion violation.*

4.3 Testing

You can test your text buffer implementation interactively by compiling and running the provided `tbuf-test.c0`, which works much like `gapbuf_test.c0`. You are encouraged to use this file as a starting point for writing your own unit tests. The `README.txt` has some suggested inputs.

```
% cc0 -d -w -o tbuf-test gapbuf.c0 dll_pt.c0 tbuf.c0 tbuf-test.c0
% ./tbuf-test
```

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.

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

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

5 Appendix: List of functions to implement

To help you keep track of the large number of functions you're writing, here's a list. Of course you may (should) define other helper functions and extend these interfaces as appropriate.

```
gapbuf.c0 - Gap buffers
  bool is_gapbuf(gapbuf* G);

  bool gapbuf_empty(gapbuf* G);    /* Returns true if the buffer is empty */
  bool gapbuf_full(gapbuf* G);     /* Returns true if the buffer is full  */
  bool gapbuf_at_left(gapbuf* G);  /* Returns true if the gap is at the   */
                                   /*   left end of the buffer            */
  bool gapbuf_at_right(gapbuf* G); /* Returns true if the gap is at the   */
                                   /*   right end of the buffer           */

  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 char c before the gap  */
  void gapbuf_delete(gapbuf* G);   /* Delete the char before the gap      */

dll_pt.c0 - Doubly-linked lists with a point
  bool is_dll_pt(dll_pt* B);

  bool dll_pt_at_left(dll_pt* B);  /* Returns true if the point is first  */
                                   /*   first (non-terminal) node         */
  bool dll_pt_at_right(dll_pt* B); /* Returns true if the point is last   */
                                   /*   last (non-terminal) node          */

  void dll_pt_forward(dll_pt* B);  /* Moves the point forward (right)     */
  void dll_pt_backward(dll_pt* B); /* Moves the point backward (left)     */
  void dll_pt_delete(dll_pt* B);   /* Remove the current point            */

tbuf.c0 - Text buffers
  bool is_tbuf(tbuf* B);

  tbuf* tbuf_new();                /* Creates an empty text buffer        */
  void tbuf_split_pt(tbuf* B);     /* Splits a full point into two nodes  */
  void tbuf_forward(tbuf* B);      /* Move the cursor forward/right by 1  */
  void tbuf_backward(tbuf* B);     /* Move the cursor backward/left by 1  */
  void tbuf_insert(tbuf* B, char c); /* Insert the char before the cursor */
  void tbuf_delete(tbuf* B);       /* Delete the char before the cursor   */
```