COMP 211: Principles of Imperative Computation, Fall 2014

Programming 7: Doubly Linked Lists

Due: Wednesday, November 12, 2014 by 11:59pm

For the programming portion of this week's homework, you'll implement another core data structure for a text editor: doubly linked lists

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 6 `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 hw7-handout.tgz
```

   This will make a directory called `Documents/comp211/hw7-handout` and extract all of the code for the assignment.

3. The file `hw7-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 6 `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 hw7-handout.tgz
```

This will make a directory called `comp211/hw7-handout` and extract all of the code for the assignment.

3. The file `hw7-handout/README.txt` contains a description of the various files, and instructions on running them.

2 A text editor based on doubly linked lists

Implementing a text editor as just one gap buffer, as you did last week, 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. (There is also a problem that the gap buffers you implemented last week have a fixed size, so you would need to know the maximum size of the document in advance—but we could fix that by using an unbounded array (see the reading for Tuesday)!)

In this assignment, you will investigate an alternative implementation based on doubly linked lists. This has its own 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. You will fix this next week, by *combining* your doubly linked list code with your gap buffer code to get the best of both worlds. But for this week we will concentrate on the doubly linked list part.

3 Doubly-Linked Lists

The lists we have seen so far are *singly-linked lists*: each node has a pointer to the node that follows it. 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`).

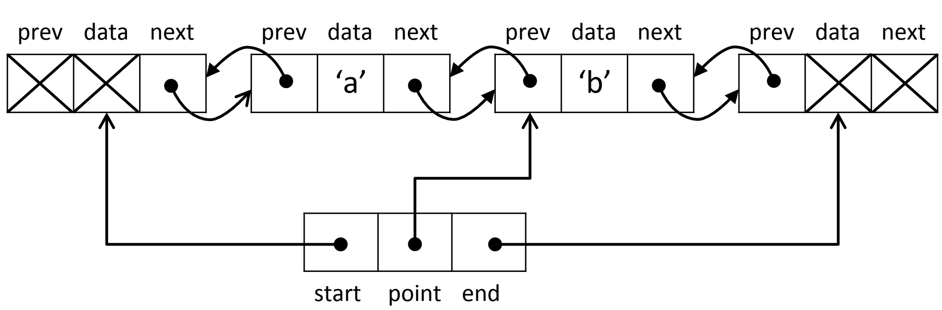

Figure 1: A pointed doubly linked list in memory.

A *pointed doubly linked list* 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 1). Like we did with queues in class, we terminate the list on both ends with "extra" 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;
};
```

Note that we define `dll` to be a synomym for `struct dll_node`, but `dll_pt` to be a synonym for a pointer type `struct dll_pt_header *`. This means we will generally use `dll*` (pointer to a DLL node) as the type of a variable, while we will generally use `dll_pt` (no star, because it is already a pointer type) as the type of a variable. Since this is your first assignment with recursive data structures, it will be helpful to visually emphasize the pointers to DLL nodes, which is why we have not defined `dll` to be a pointer type.

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
```

The doubly-linked list functions that you will implement can be parametrized by the type of the elements of the list; in the handout we will take the data elements to be C0 characters.

3.1 Invariants

Task 1 (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(struct dll_pt_header* B)
```

that formalizes the linking invariants on a doubly-linked list with a point. You are not required to check for circularity, but you may find it to be a useful exercise. The data values of `start` *and* `end` *will be ignored, and are drawn as X's in the diagrams, but you do not need to check that they are anything in particular.*

This task is probably the hardest one on the assignment. 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:

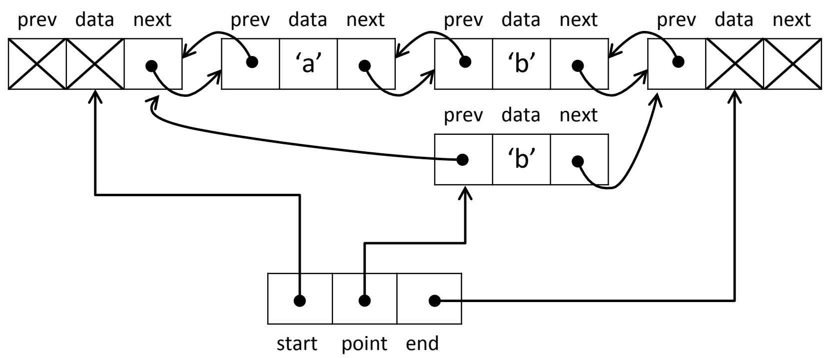

Figure 2: Not a doubly-linked list (the `point` isn't on the path from `start` to `end`).

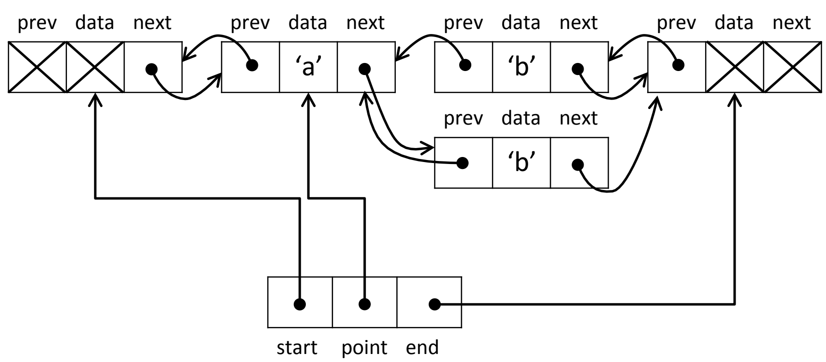

Figure 3: Not a doubly-linked list (the `prev` links don't mirror the `next` links).

You may find it helpful to implement a helper function

```
bool links_ok(struct dll_pt_header* B)
```

that first checks that the `prev` and `next` links are non-NULL and mirror each other, and then to write a separate check that the point invariants are satisfied.

Testing To save you from having to contract many examples, we have provided a test file with a bunch of tests on valid and invalid doubly linked lists. To compile and run this test file, do

```
$ cc0 -d -o is_dll_pt-test elem-char.c0 dll_pt.c0 is_dll_pt-test.c0
$ ./is_dll_pt-test
```

You will then see messages like

```
TEST FAILED: is_dll_pt should return false when list has NULLs 1
```

if any of the tests fail. If the message itself is not enough of a hint, you can try reading the code in `is_dll_pt-test.c0` to see what example your code does not work on.

3.2 Operations

Task 2 (2 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*

Note that the invariants assert that the point is never equal to *start, so* `dll_pt_at_left` *means that the point is the node just to the right of the start, and similarly for* `dll_pt_at_right`.

Task 3 (3 pts) *Implement the following interface functions for manipulating doubly-linked lists with points:*

`dll_pt dll_pt_new(elem pointe)`	*Creates a new dll*
`void dll_pt_insert_before(dll_pt B, elem newel );`	*Inserts before the point*
`void dll_pt_insert_after(dll_pt B, elem newel );`	*Inserts after the point*

`dll_pt_new` *should create a new doubly-linked list with one element, where* `pointe` *is the element for its single node, which is also the point.*

`dll_pt_insert_before` *should create a new node whose element data is* `newel` *and insert it into the doubly-linked list just before the point node (i.e. in the direction of the start node). The point should still refer to the same node at the end as it did at the beginning.*

`dll_pt_insert_after` *should create a new node whose element data is* `newel` *and insert it into the doubly-linked list just after the point node (i.e. in the direction of the end node). The point should still refer to the same node at the end as it did at the beginning.*

Task 4 (3 pts) *Implement 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*

`dll_pt_delete` *must move the point either to the node immediately to the left of the point, or to the node immediately to the right of the point; when both are an option, you can pick which one to move it to.*

Task 5 (2 pts) *Give good pre- and post-conditions for all of these functions:*

- *All functions should require and ensure that the dynamically linked lists are valid.*
- *You should give sufficiently strong pre-conditions that the operation can always be performed, and you should document any special properties of the result in the post-condition. Unlike for gap buffers, insertion should never fail. However, we cannot delete the point if it is the only non-terminator node. Moreover, we can only move the point forward/backward if there is something to move it to.*

3.3 Testing

You can test your doubly-linked-list implementation interactively by compiling and running the provided `dll_pt-test.c0`.

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

Like for gap buffers last week, you can interactively insert, delete, and move the point:

```
Visualizing DLL with elem as char.
Initial buffer: START <--> _'A'_ <--> END

Now you can edit the buffer:
The '<' character moves the point backwards
The '>' character moves the point forwards
The '^' character deletes
A capital letter A-Z inserts BEFORE the point
A lowercase letter a-z inserts AFTER the point

Give actions (empty line quits): ABC<<abc
     START <--> _'A'_ <--> END
bef: START <--> 'A' <--> _'A'_ <--> END
bef: START <--> 'A' <--> 'B' <--> _'A'_ <--> END
bef: START <--> 'A' <--> 'B' <--> 'C' <--> _'A'_ <--> END
<= : START <--> 'A' <--> 'B' <--> _'C'_ <--> 'A' <--> END
<= : START <--> 'A' <--> _'B'_ <--> 'C' <--> 'A' <--> END
aft: START <--> 'A' <--> _'B'_ <--> 'a' <--> 'C' <--> 'A' <--> END
aft: START <--> 'A' <--> _'B'_ <--> 'b' <--> 'a' <--> 'C' <--> 'A' <--> END
aft: START <--> 'A' <--> _'B'_ <--> 'c' <--> 'b' <--> 'a' <--> 'C' <--> 'A' <--> END
```

So this will test `insert_before`, `insert_after`, `delete`, `forward`, `backward`, and `new` (which is used to make the initial DLL). It will also test `at_left` and `at_right`, since they should be used in specifications for these functions.

To test `at_left` and `at_right` independently, you can add tests for them to the function `test_dll` in `function-tests.c0`. This function constructs the examples from the handout directly. You can load this file in coin and run it as follows:

```
$ coin -d elem-char.c0 dll_pt.c0 function-tests.c0
```

Like last week, you will need to add do-nothing implementations for all of the functions before you can compile the tester, and you will need to implement `new` and one of the `insert` functions before you can test the others.