Linked List Testbench

The linked list testbench is entirely contained in testbench.h and testbench.cpp. As this testbench will be used to grade your linked list implementation, modifying either of these files without permission constitutes academic misconduct.

The testbench is initiated using the function `test_dlinkedlist()`. This function takes two parameters:

- const char* tracefile – the filename (on the SD card) of the trace file to run. Syntax of the tracefile is described below.
- const char* outfile – the filename (on the SD card) to save the output of the trace. If this parameter is NULL, output will be sent to the console only.

During development, it is recommended that you make your own tracefiles to use in testing your code. Doing so will allow you to build an automated testing suite that runs before the game to validate your linked list. If done properly, a comprehensive ***unit test*** suite can provide automated feedback during development to deter bugs and mistakes. The testbench gives you an opportunity to describe what behavior you expect from your code, and to determine automatically if it correctly follows that behavior.

This mbed testbench is entirely for your benefit. Grading for the linked list part of the project will be done using a different (but very similar!) unit test framework, so it is in your best interest to build unit tests for yourself ahead of time.

Notes on generic data types in C

The doubly linked list library is a generic linked list implementation, in that it takes a `void*` type for its data input. Void pointers in C can be used to hold any type of data – (almost) anything can be validly cast to a void pointer without loss of information. This pointer value is a single machine word in length – on the mbed, this is 16-bit word. Since a pointer is just a number interpreted as an address, it can be abused to hold plain integers too. For example, all of the below is valid:

```
// In this case, the void pointer is actually a pointer to a struct
typedef struct my_struct_t { … } my_struct;
my_struct* s = (my_struct*) malloc(sizeof(my_struct));
insertHead(dll, (void*)s);

// In this case, raw integer data is used.
int my_int = 42;
insertHead(dll, (void*)my_int);
```

This gives us some interested design decisions when it comes to implementing the linked list. Whether the data is structured or plain determines whether its memory needs to be freed explicitly upon deletion. In the case of structured data (the first case above), `free()` needs to be used when the node is deleted from the list in order to avoid leaking the `malloc`'d memory. For the integers, however, there is no additional data allocated – `free()` doesn't make sense in this context. In order to keep the list as generic as possible, the `shouldFree` flag is included as a second parameter to the `deleteForward()`, `deleteBackward()`, and `destroyList()` API calls. This flag indicates whether the implementation should use free() to release the data structure, or if it can safely skip doing so.

Trace file format

For simplicity, the testbench uses only 16-bit integer values in the list. This allows us to define a simple text format for expressing test cases in terms of their API operations. This format is like an assembly language for linked list operations. It describes exactly which linked list operations to perform and what values to use for those operations.

Each operation should be on its own line. The format is as follows, and corresponds directly with the linked list API:

```
insertHead       ih <num>
insertTail       it <num>
insertAfter      ia <num>
insertBefore     ib <num>
deleteBackward   db <expect>
deleteForward    df <expect>
getHead          h  <expect>
getTail          t  <expect>
getCurrent       c  <expect>
getNext          n  <expect>
getPrevious      p  <expect>
getSize          s  <expect>
```

`<num>` indicates the number to insert. Behavior is only well-defined for positive 16-bit integers! If you use anything else, the testbench probably won't work right. Note that the integer "0" is also special and shouldn't be inserted as data in the list. `NULL == 0`, so using 0 as a data value can leads to ambiguous results.

`<expect>` indicates the expected return value of the function. As before, this should only be a positive 16-bit integer. Zero can be used to detect failure (For example, if the current pointer is invalid, `getCurrent()` should return `NULL == 0`).

Additional operations

```
Check list          e <expect> <expect> ...
```

This command walks the list from head to tail using getHead() and getNext(), verifying each expected value along the way. It also checks that the list has the correct number of items as indicated in the command. **This will move the current pointer**.

```
Reset               r
```

This command calls destroyList() to delete the current list, and reinitializes a new list by calling create_dlinkedlist().

```
Comments            % The rest of this line is ignored
```

As in any programming language, our tracefiles allow comments using the format above.

```
insertAfter (expect fail)       iaf <num>
insertBefore (expect fail)      ibf <num>
```

These two commands are the same as their positive counterparts above, except that they will succeed if the underlying linked list command fails. This is useful for testing edge cases – see the example below for how to use these instructions properly.

Examples

Consider this example trace:

```
% Sample trace
ih  1
ih  2
ih  3
iaf 42          % Attempt to insert 42; current pointer is invalid
e   3 2 1
```

This trace begins by inserting new head node repeatedly, building up the list:

```
(1) -> (2, 1) -> (3, 2, 1)
```

Since no command to set the current pointer has ever been used, the current pointer is in its initial state - invalid. Therefore, an attempt to use `insertAfter(42)` in this scenario should fail. This is tested using `iaf 42`. Finally, the list is verified with the Check List command. All commands on this trace would succeed, and the final state of the list is `(3, 2, 1).`