15-122: Principles of Imperative Computation, Spring 2011
Assignment 4: Data Structure Design, Memory Layout, and Amortization

William Lovas (`wlovas@cs`) Bill Zorn

Out: Wednesday, June 1, 2011
Due: Monday, June 6, 2011
(Written part: before lecture,
Programming part: 11:59 pm)

1 Written: Potpourri (20 points)

The written portion of this week's homework will give you some practice working with amortized analysis, memory management, debugging, and recursion. You can either type up your solutions or write them *neatly* by hand, and you should submit your work in class on the due date just before lecture begins. Please remember to *staple* your written homework before submission.

1.1 Amortized Analysis

We can implement a queue using two stacks, an `in` stack and an `out` stack.

- An empty queue is one where both the `in` stack and the `out` stack are empty.
- To enqueue an element, simply push it onto the `in` stack.
- To dequeue an element, pop one from the `out` stack—but if the `out` stack is empty, first reverse the contents of the `in` stack onto the `out` stack.

Exercise 1 (6 pts)**.** Consider the run-time complexity of the `enq` and `deq` operations for such a queue.

(a) How many stack operations are needed, in the worst case, for a single `enq` or `deq` operation on a queue of size *n*?

(b) How many stack operations are needed, in the worst case, for a sequence of k `enq` or `deq` operations, starting from an empty queue? Use amortized analysis to justify your answer.

1.2 Memory Management

Constant factors can influence the memory efficiency of data structures and algorithms, even if the asymptotic complexity is the same. For example, a stack can be internally implemented in multiple ways without changing the interface functions you use to manipulate it. The implementation we saw in class uses linked lists:

```
struct list_stack {
  struct list_node* top;
  struct list_node* bottom;
};

struct list_node {
  elem data;
  struct list_node* next;
};
```

Another possible implementation uses unbounded arrays instead:

```
struct uba_stack {
    struct ubarray * U;
};

struct ubarray {
  int limit;         /* limit > 0 */
  int size;          /* 0 <= size && size <= limit */
  elem[] elems;      /* \length(elems) == limit */
};
```

Assume for the moment that both stack implementations store characters as their data elements: `typedef char elem` has been defined previously in the code. Recall that in C_0, a `char` is represented using 8 bits, a pointer is represented using 64 bits, and an `int` is represented using 32 bits. An array of characters of size n takes exactly $8n + 64$ bits—$8n$ for the characters and 32 bits each for two `int`s: one to record the length of the array and one to record the size of its elements. The size of a struct is the sum of the sizes of its elements.

Exercise 2 (4 pts)**.** Consider some program that uses a stack of characters. If the program can allocate only 1 MB (2^{20} bytes, or 8×2^{20} bits) of memory for this stack, what is the maximum number of characters that it can store using a `struct list_stack`? How about using a `struct uba_stack`? (Show your work.)

1.3 Debugging

The following helper function is intended to take a structure `cell` containing a stack of strings and print it out, along with some general identifying information about it. You may assume that the code compiles, that type `elem` is defined as `string`, and that stacks are implemented using linked lists as in lecture.

```
typedef struct wordcell * cell;
struct wordcell {
  int ID;
  int size;
  stack words;
};

bool is_cell(cell W) {
  if (W == NULL) return false;
  if (! (0 <= W->ID && W->ID < 1000)) return false;

  return (W->words == NULL && W->size == 0)
      || (is_stack(W->words) && stack_size(W->words) == W->size);
}

void printcell(cell W)
//@requires is_cell(W);
{
  print("Cell: "); printint(W->ID); print("\n");
  print("size: "); printint(W->size); print("\n");

  list place = W->words->top;

  while (place != W->words->bottom) {
    println(place->data);
    place = place->next;
  }
}
```

Exercise 3 (4 pts)**.**

(a) Devise a test case that causes `printcell` to segmentation fault. You may write out this case as a short `main()` function. Assume that the program is compiled with the `-d` flag, and that a contract failure is different from a segmentation fault. **Hint:** What are all the ways a segmentation fault can occur in a C_0 program?

(b) Change `printcell` so that this case does not cause a segmentation fault. The specification function `is_cell` is correct—do not change it! **Hint:** Your change should not affect more than a few lines.

1.4 Recursion

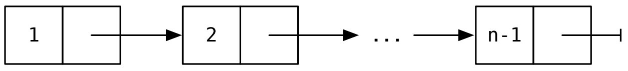
Exercise 4 (6 pts)**.** Recursive functions can be used to construct data structures of recursive type, like linked lists. For the following exercise, you may assume a function `int list_length(struct list_node* L)` which has the property that `list_length(NULL) == 0` and a `list_cons` function which adds one element to the front of a linked list.

```
struct list_node* list_cons(elem x, struct list_node* xs)
//@ensures list_length(\result) == list_length(xs) + 1;
{
    struct list_node* res = alloc(struct list_node);
    res->data = x; res->next = xs;
    return res;
}
```

(a) Write a recursive function matching the following specification:

```
struct list_node* from(int start, int end)
  //@requires start <= end;
  //@ensures list_length(\result) == end - start;
  ;
```

The result of a call like `from(1, n)` should be a list containing the numbers from 1 to $n - 1$ in order, terminated by a `NULL` pointer:



(For this problem, the type `elem` is defined to be `int`.)

(b) Prove that if the pre-condition holds, the returned result satisfies the post-condition. When reasoning inside the body of a function f, if you *know* that the pre-condition of a recursive call to f holds before the call, you may *assume* the post-condition of that call holds after the call.

2 Programming: Implementing a Text Editor (30 points)

In the programming portion of this week's assignment, you'll practice a bit of data structure design by implementing the core data structure for a text editor: the edit buffer. Using contracts to guide your coding is a major theme, so be sure to *read the specifications carefully*—you'll thank yourself later!

You should submit your code electronically by 11:59 pm on the due date. Detailed submission instructions can be found below.

Starter code. Download the file `hw4-starter.zip` from the course website. It contains definitions of the edit buffer data structures (`gapbuf.c0` and `tbuf.c0`) and some code for visualizing them (`visuals.c0`), which you may find useful for testing your implementation.

Compiling and running. Compile and test your code using either `cc0` or `coin`. Don't forget to include the `-d` switch to enable dynamic checking of contracts!

For details on how we will compile your code, see the file `COMPILING.txt` included in the starter code. **Warning:** *You will lose credit if your code does not compile.*

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

```
    handin -a hw4 <file1>.c0 ... <fileN>.c0
```

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.

The submission verifier framework for this assignment will check *only* that your code compiles: you are responsible for testing your code thoroughly!

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.

2.1 Text Buffer Primer

A *text buffer* is an abstract data type for representing the contents of a text editor's edit buffer. In this assignment, you will explore several possible representations of a text buffer, eventually settling on one that strikes a nice balance between space efficiency and time efficiency.

A text buffer may be visualized as a sequence of characters with a distinguished *cursor* position. For instance, we might write "`ab|cdef`" for a text buffer containing the characters *a* through *f* with the cursor between the *b* and the *c*. Text buffers may allow a variety of editing operations to be performed on them; for the purposes of this assignment, we'll consider text buffers that implement four operations: move forward a character, move backward a character, insert a character, and delete a character.

```
typedef struct tbuf * tbuf;

void forward_char(tbuf B);
void backward_char(tbuf B);
void insert_char(tbuf B, char c);
void delete_char(tbuf B);
```

If `B` refers to a text buffer representing `ab|cdef`, the effect of each of these operations can be illustrated by the following examples:

operation	**`B` after operation**
`forward_char(B)`	`abc|def`
`backward_char(B)`	`a|bcdef`
`insert_char(B, 'X')`	`abX|cdef`
`delete_char(B)`	`a|cdef`

Your goal by the end of the assignment is to implement these operations based on a representation of a text buffer as a *doubly-linked list* of *gap buffers*.

2.2 Gap Buffers (`gapbuf.c0`)

One typical data structure used to implement an edit buffer is a *gap buffer*. A gap buffer is a generalization of an unbounded array: whereas an unbounded array allows for efficient addition and deletion of elements from the end only, a gap buffer allows efficient addition and deletion of elements from a *gap* somewhere in the middle. In addition, the gap buffer offers efficient operations to move the gap to the left and to the right.

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).

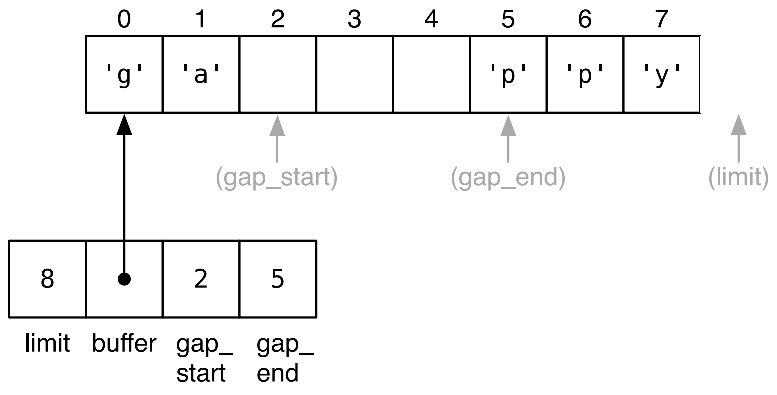


Figure 1: A gap buffer in memory.

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

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.

Task 1 (3 pts)**.** Implement a specification function `bool is_gapbuf(gapbuf G)` formalizing the gap buffer data structure invariants.

For the case of a gap buffer representing an edit buffer, the elements are characters, and we can visualize the gap buffer as a string of characters with a gap in the middle:

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

The above represents a gap buffer of size 16 containing the string "the space race", with the gap situated between the "s" and the "p" in "space". To delete a character, we simply expand the gap:

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

To move the gap, we copy a character across it:

```
the p[...]ace race
```

To insert a character, we write it into the gap, shrinking it by one:

```
the pe[..]ace race
```

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

```
[...]mineshaft gap
```

or at the right end of the buffer,

```
minecraft gap[...]
```

and a buffer can be empty,

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

or it can be full,

```
mind the ga[]p sir
```

Task 2 (2 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.

2.3 Doubly-Linked Lists (`tbuf.c0`)

Another data structure that could 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—but such a structure cannot provide constant time insertions or deletion in the middle of the list. 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).

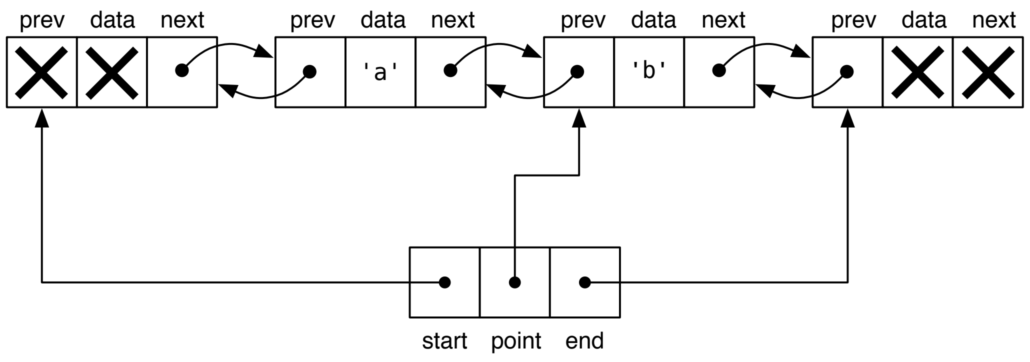


Figure 2: An editable sequence as a doubly-linked list in memory.

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

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

We can visualize a well-formed doubly-linked list as the sequence of its data elements with terminator nodes at the end and one distinguished element.

$$\texttt{** <--> 'a' <-->} \boxed{\texttt{'b'}} \texttt{ <--> **}$$

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 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 a distinct node from both the start and the end nodes, i.e., that the list be non-empty.[1]

Task 4 (3 pts)**.** Write a specification function `bool is_linked(tbuf B)` formalizing the linking invariants on a doubly-linked list text buffer. (You are not required to check for circularity, but you may find it useful to do so.)

Task 5 (3 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!)

2.4 Putting It Together (`tbuf.c0`)

While either gap buffers or doubly-linked lists of characters could be used to represent an edit buffer—and your implementations could easily be extended to implement the general edit buffer interface—neither strategy is 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 for large files, and furthermore, although all operations are amortized constant time, long pauses could still hamper interactivity. A doubly-linked list of single characters, while offering truly constant-time operations and allowing the file to be split across several chunks of memory, involves far too much space overhead at two pointers per character.

Instead, a more realistic strategy is to combine the two ideas by implementing a doubly-linked list of fixed-size gap buffers. 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. When a gap buffer is full, we can split it in two:

[1] And of course, for any of these to hold, several pointers must be non-`NULL`!

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

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

and when one becomes empty, we can delete it:

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

delete: $\texttt{** <--> }\boxed{\texttt{deletio[.]}}\texttt{ <--> **}$

To move, we use a combination of gap buffer motion and doubly-linked list motion:

$$\texttt{** <--> just\_a\_[.] <--> }\boxed{\texttt{j[....]ump}}\texttt{ <--> **}$$

move ←: $\texttt{** <--> just\_a\_[.] <--> }\boxed{\texttt{[....]jump}}\texttt{ <--> **}$

move ←: $\texttt{** <--> }\boxed{\texttt{just\_a[.]\_}}\texttt{ <--> [....]jump <--> **}$

An invariant that arises from this representation is that a text buffer is always *aligned*: every gap buffer *before* the point is non-empty with its gap to the *right*, and every gap buffer *after* the point is non-empty with its gap to the *left*.

$$\texttt{** <--> lawfu[...] <--> l\_evil\_c[] <--> }\boxed{\texttt{hao[....]t}}\texttt{ <--> [.]ic\_good <--> **}$$

Additionally, all the gap buffers are themselves well-formed, and they all have the same size (8 characters in the diagrams here, but you should use 16 for your implementation).

Task 6 (3 pts)**.** Implement a specification function `bool is_aligned(tbuf B)` formalizing the alignment invariant. Ensure that each gap buffer has size 16.

For a text buffer to be fully well-formed, it must either be the empty text buffer:

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

or it must be properly linked and aligned and have a non-empty point.

Task 7 (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

(**Aside:** At this point, you might consider re-declaring the `tbuf_delete_point` function with a stronger contract: you can now give it a much more precise specification, and doing so will help you later! Think about what further invariants it preserves beyond those specified earlier.)

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        full buffer: stuvwxyzSTUV[]WXYZ

splits into: abc[........]defgh        splits into: stuvwxyz[........]
             [........]ABCDEFGH                     STUV[........]WXYZ
```

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

Task 8 (4 pts)**.** Implement a function `tbuf_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.

Task 9 (4 pts)**.** Implement the following interface functions for manipulating text buffers:

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

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.

After you've completed your text buffer implementation and tested it thoroughly, you can try it out interactively by compiling against `e0.c0` from the starter code, a comically minimalist text editor front-end called E_0. Enjoy the hard-won fruits of your careful programming labor!

2.5 Judges' Prize: Extending the Editor

Extend the E_0 editor implementation with some interesting features. A few suggestions, to pique your imagination: a better display algorithm, a better splitting algorithm, line motion, more editing commands, copy and paste—be creative! Feel free to extend the data structures in any way necessary to support your changes effectively. Submit your modified implementation as files named `judges-*.c0` and include a `judges-README` file explaining your work. The coolest extension—both in terms of the interactive editing experience and the supporting data structures and algorithms—will receive a prize!