CS106B Handout #27
Fall 2000 October 27, 2000

Assignment #5: Editor Buffer

Due: Friday, November 3 at 5:00 p.m.

In this assignment, you will complete the quest we described in class this week: designing an implementation for **`buffer.h`** in which all six of the fundamental editing operations run in constant time. The basic technique that you will use in your implementation is the **doubly linked list**, which is described informally in Chapter 9 but not completely coded. Moreover, to give you a sense of how much memory is being consumed in overhead (for the double linked list, two pointers per character!) and how this can be avoided, this assignment also asks you to implement a hybrid strategy. This strategy, known sometimes as a **"chunk list"**, uses blocks consisting of several characters that are linked together. Although this portion of assignment is divided into parts to make sure that you understand the dynamics of doubly linked lists before moving on to the more complicated task of combining characters into blocks.

The supplied modules

The starter files consist of the following files:

`editor.c`	The outer editor application. Makes calls to **`buffer.h`** interface
`buffer.h`	The interface library for the buffer.
`listbuf.c`	A singly-linked list implementation of **`buffer.h`**
`Demo Editor`	A demo version of the program so you can see how it behaves

When you create your project, you will need to add `editor.c and listbuf.c` and to it. This should give you a program that *behaves* in the same way as the demo app.

You begin with a linked-list implementation (`listbuf.c`) given in Chapter 9. You will need to write your own `buffer.c` to replace the `listbuf.c` file for Part 1 of this assignment, and then you will create yet another version of `buffer.c` for Part 2. **You may not make any changes to the main program `editor.c`, and you should not change the `buffer.h` interface for Part 1 or Part 2**. If you've got some terrific new extension that you want to add to the editor in search of an extra-credit score, get the assignment working first and then extend the `buffer.h` interface to support the new features in your editor. **If you do extra credit that requires changing buffer.h or editor.c, you must submit a version that supplies only the required functionality**. Submit your extra credit as an additional set of files.

Part 1—Reimplement `buffer.h` using a doubly linked list

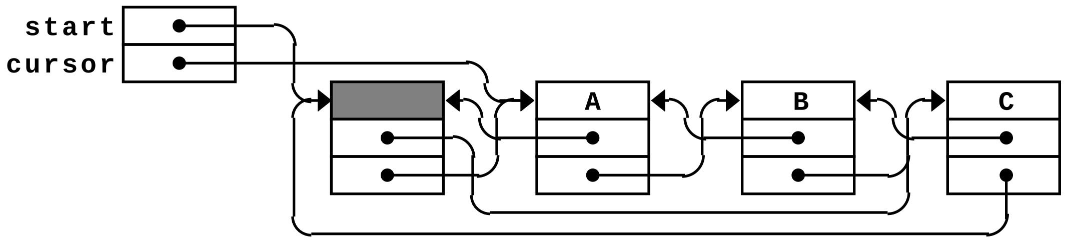
From the standpoint of computational complexity, the only remaining problem with the linked-list implementation of `buffer.h` is that various operations are slow in one direction but fast in the other. Deleting or moving forwards and moving to the beginning of the buffer are all constant time operations. Unfortunately, moving backwards and moving to the end of the buffer remain linear-time operations.

Chapter 9 outlines the final steps that are necessary to reach the goal of having all editor operations execute in constant time. The basic strategy is to maintain a link pointer to the previous cell as well as one to the following cell, thereby forming a **doubly linked list,** which is discussed beginning on page 407. Each cell in the doubly linked list must have two link fields: a `prev` field that points to the previous cell and a `next` field that points to the next one. It still makes sense to maintain a dummy cell in the buffer, and you can use the fact that the dummy cell contains both a `prev` and a `next` field to your advantage in the design. As in the singly linked list, the `next` field of the dummy cell points to the first character cell, but you can use the `prev` field of the dummy cell to point to the end of the buffer. Moreover, if you experiment with this design a little, you will discover that the insertion and deletion operations become more symmetrical if you make the `link` field of the last cell point back to the dummy cell instead of having it be `NULL`.

When implemented according to this design, the doubly linked representation of the buffer containing the text

```
A|B C
```

looks like this:



This structure provides all the information you need to reimplement the buffer operations in constant time.

The steps you need to complete Part 1 of the assignment are as follows:

1. Change the representation of the `cellT` type so that the structure includes the extra `prev` pointer described in the preceding paragraphs.
2. Implement a `buffer.c`, (replacing `listbuf.c`), to correspond to the new data structure representation.

Be sure to test your program as thoroughly as you can. Even if the buffer display looks right, you may have internal problems. For example, make sure that you can move the cursor in both directions across parts of the buffer where you have recently made insertions and deletions. Also, be certain that your program correctly detects the beginning and end of the buffer and does not allow, for example, the `MoveForwardCommand` to move past the end of the buffer.

Part 2—Reimplement the editor using a linked list of blocks ("chunk list")

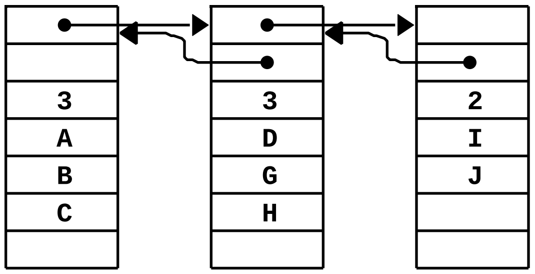
The main problem with the doubly linked list in Part 1 is that it is terribly inefficient in terms of space. With two pointers in each cell and only one character, pointers take up 89% of the storage, which is likely to represent an unacceptable level of overhead.

The best way around this problem is to combine the array and linked-list models so that the actual structure consists of a doubly linked list of units called **blocks,** where each block contains the following:

- The `prev` and `next` pointers required for the doubly linked list
- The number of characters currently stored in the block
- A fixed-size array capable of holding several characters rather than a single one.

This is the structure often called a "chunk list". By storing several data characters in each block, you reduce the storage overhead because the pointers take up a smaller fraction of the data. However, because the blocks are of a fixed maximum size, the problem of inserting a new character never requires shifting more than *k* characters, where *k* is the **block size** or maximum number of characters per block. Because the block size is a constant, the insertion operation remains constant. As the block size gets larger, the storage overhead goes down but the time required for an insertion goes up. In the examples shown below, the block size is assumed to be four characters, although a larger block size would probably make more sense in practice.

To get a better idea of how this new buffer representation works, consider how you would represent the character data in a block-based buffer. The characters in the buffer are stored in individual blocks, and each block is chained to the blocks that precede and follow it by link pointers. Because the blocks need not be full, there are many possible representations for the contents of a buffer depending on how many characters appear in each block. For example, the buffer containing the text `"ABCDGHIJ"` might be divided into three blocks, as follows:

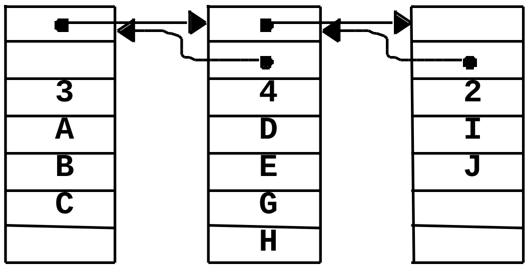


This diagram does not show the complete representation of the buffer type. In this diagram, what you see represented are three blocks, each of which contains a pointer to the next block, a pointer to the previous block, the number of characters currently stored in the block, and four bytes of character storage. The actual definition of the concrete buffer type and the contents of two link fields missing in this diagram (the first backward link and the last forward one) depend on your representation of the concrete structure of the buffer, which is up to you to design. In particular, the buffer structure you design must include some way to represent the cursor position, which presumably means that the actual `bufferCDT` structure definition must include a pointer to the current block as well as an index showing the current position within that block.

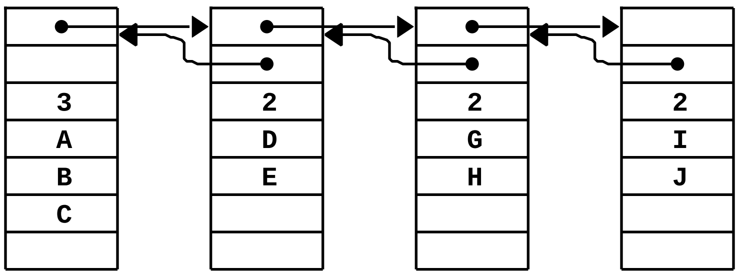
Assume for the moment that the cursor in the previous example follows the **`D`**, so that the buffer contents are

`A B C D|G H I J`

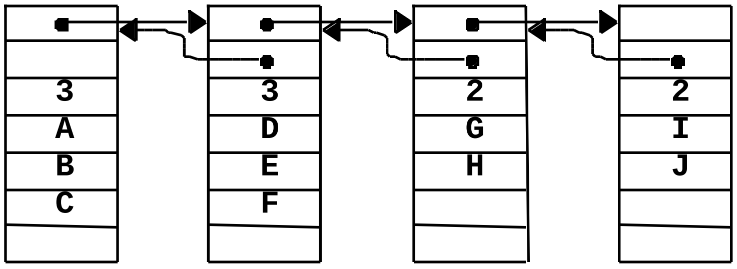
Suppose that you want to insert the missing letters, **`E`** and **`F`**. Inserting the **`E`** is a relatively simple matter because the active block has only three characters, leaving room for an extra one. Inserting the **`E`** into the buffer requires shifting the **`G`** and the **`H`** toward the end of the block, but does not require any changes to the pointers linking the blocks themselves. The configuration after inserting the **`E`** therefore looks like this:



If you now try to insert the missing **`F`**, however, the problem becomes more complicated. At this point, the current block is full. To make room for the **`F`**, you need to split the block into two pieces. A simple strategy is to split the block in two, putting half of the characters into each block. After splitting (but before inserting the **`F`**), the buffer looks like this:



From here, it is a simple matter to insert the **`F`** in the second block:



Your job in Part 2 of the assignment is to reimplement `buffer.c` so that a **`bufferADT`** is a linked list of blocks, where each block can hold up to `MaxCharsPerBlock` characters. The details of the data structure design are up to you.

In writing your implementation, you should be sure to remember the following points:

- It is your responsibility to design the data structure for the `bufferCDT` type. As a general hint, it helps to think hard about the design of your data structure before you start writing the code. Draw pictures. Figure out what the empty buffer looks like. Consider carefully how the data structures change as blocks are split. **You must draw a diagram of your structure in your code much like the one provided in listbuf.c.**
- Make sure that you understand how the cursor is represented in the buffer and try to adopt a strategy that provides as much consistency of structure as possible. To get a sense of whether your representation works well, make sure that you can answer basic questions about your representation. How does your buffer structure indicate that the cursor is at the beginning of the buffer? What about a cursor at the end? What about a cursor that falls between characters in different blocks? Is there a unique representation for such circumstances or is there ambiguity in your design. In general, it will help you a lot if you try to simplify your design and avoid introducing lots of special case handling.
- If you have to insert a character into a block that is full, you should adopt the strategy described above and divide the full block in half before making the insertion. This policy helps ensure that neither of the two resulting blocks starts out being filled, which might immediately require another split when the next character came along.
- If you delete the last character in a block, your program should free the storage associated with that block unless it is the only block in the buffer.
- **Your program should not orphan any memory.** Memory that is no longer in use must be freed in this assignment, and **this will now be a graded part of your assignment**.
- Make sure that `FreeBuffer` works, in the sense that all allocated memory is freed and that you never reference data in memory that has been returned to the heap.
- Your implementation should work for any reasonable block size (0, of course, is not reasonable, but almost anything else is). Part of the challenge of the assignment is to make sure your code works regardless of the block size without the mess of masses of special cases.
- Document your design choices in the code.

Extensions

The best extension would be to turn this into Microsoft Word™. In case you don't have quite enough time for that, we will suggest a few more practical ideas.

Some things to consider:

1. <u>Add a search command</u>. The user would specify a string to search for, and the editor would check to see if it is in the buffer, starting the search from the current cursor position. If the string is found, the cursor would be repositioned to the point just before the string. If the string is not present, the cursor does not move.

2. <u>Add a search and replace command</u>. This is similar to the above, but the user also specifies replacement text. If the search string is found, it is replaced. You will have to come up with a simple syntax for specifying the command.

3. <u>Add cut, copy, and paste commands</u>. This involves maintaining a separate "paste buffer". Since we don't have a good way to mark text, you might want to let the user specify the number of characters involved. For example, copying 5 characters would copy the five characters following the cursor into the paste buffer. Cutting 5 characters would copy them and then delete them from the text. The paste command would insert the characters in the paste buffer into the editor buffer at the current cursor position. How you store the paste buffer is up to you. A reasonable assumption might be that it would not be very long.

4. Thinking back to Boggle, that lexicon was pretty neat. Why not use it here to implement…a SPELL CHECKER! If you can come up with a way to find the words in your buffer, then it would be easy to run them through the lexicon. This would be a great extension—many thanks to former TA Rob Baesman for suggesting it!

Deliverables

As always, we would like you to submit any .c and .h files that you write. So, you should turn in your buffer.c file for both Parts 1 and 2.

If you extend the program, your must **submit separate files** (with different names, i.e. `extendededitor.c`, `extendedbuffer.c` and `extendedbuffer.h`) for the extended version, along **with a standard `buffer.c`** that supports the regular version.

This is a good time to remind you that you should **never** turn in the only electronic copy of a program. Always keep a backup on a floppy or your hard drive.