Assignment #3: Editor Buffers

This assignment adapted from a similar one by Eric Roberts.

Due: Wed April 26th in class

In this assignment, you will complete the quest described in class: 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 to avoid the substantial space overhead imposed by cells containing two links for every character, this assignment also asks you to implement a hybrid strategy in which blocks consisting of several characters are linked together. Although the assignment is staged in 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, **you need not turn in code for the first part**.

We will give you all three of the buffer implementations we showed in class: one using an array, another using stacks, and the singly-linked list implementation. Once you have the two additional implementations you will build in Parts 1 and 2, in Part 3, you will run some tests and consider the strengths and weaknesses of the various versions. We provide all the client code that allows you to interactively test the ADTs and run performance trials, so your role will be to act as the ADT implementor only. You should make no changes at all to the main program **`editor.c`** or to the **`buffer.h`** interface for the assignment. 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.

The editor client

The editor client we provide is pretty much the same one described in the first two sections of Chapter 9 in the text. We added a few additional commands noted here:

'C'	Clear buffer
'G'	Print debugging information
'P'	Run performance trial

The clear command just empties the contents of the buffer so you can start editing afresh. The debugging option is provided as a hook for you. You can implement your own debugging function that will dump the internal state of the buffer in a way that is helpful to examine while executing a test. In particular, for the more complicated chunklist you will implement in Part 2, this can prove to be quite valuable to show you the current state of your structure at any given point during execution. The performance command runs a sequence of trials and reports on the time spent and memory used for a given buffer implementation.

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.

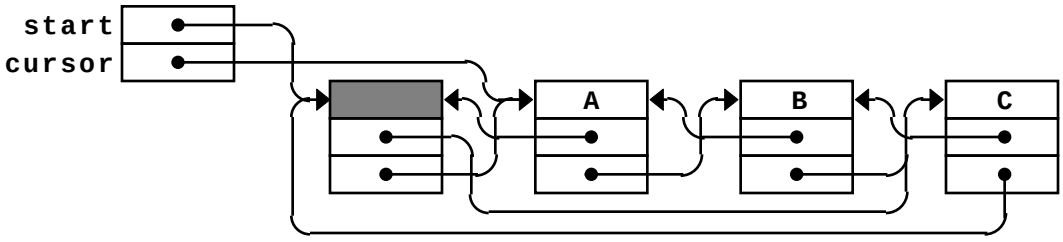
Chapter 9 outlines the final steps that are necessary to reach the goal of having all editor operations execute in $O(1)$ time. The basic strategy is to maintain a link pointer to the previous cell as well as one to the following cell, thereby forming a **doubly linked list,** which is discussed beginning 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 `next` 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 `bdoublelink.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.

Save a copy of your doubly-linked list. Although you won't hand in that code, you will need that implementation to run the performance trials in Part 3.

Part 2—Reimplement the editor using a "chunklist"

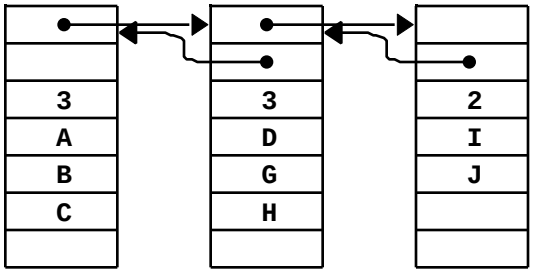
The main problem with the doubly linked list in Part 1 is that it is terribly inefficient in terms of space. With two pointers in each cell and only one character, pointers take up 89% of the storage, which is likely to represent an unacceptable level of overhead.

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.

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 *O*(1). As the block size gets larger, the storage overhead goes down but the time required to do 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:

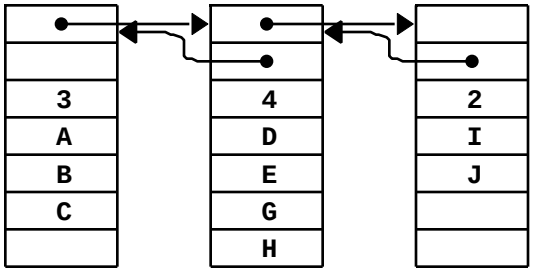


This diagram does not show the complete representation of the buffer type. In this diagram, what you see represented are three blocks, each of which contains a pointer to the next block, a pointer to the previous block, the number of characters currently stored in the block, and four bytes of character 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.

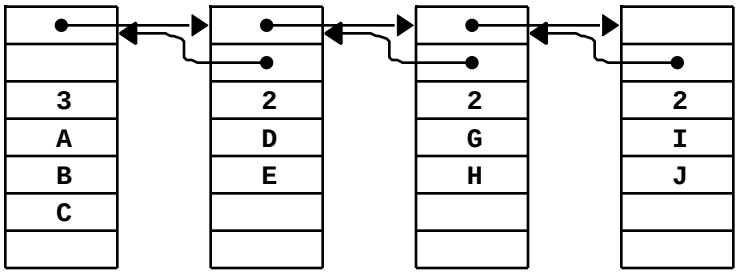
Assume for the moment that the cursor in the previous example follows the **D**, so that the buffer contents are

A B C D | G H I J

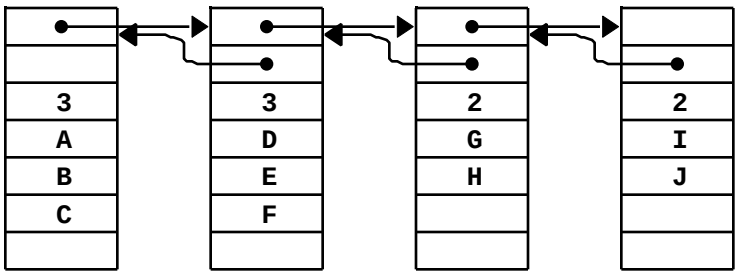
Suppose that you want to insert the missing letters, **E** and **F**. Inserting the **E** is a relatively simple matter because the active block has only three characters, leaving room for an extra one. Inserting the **E** into the buffer requires shifting the **G** and the **H** toward the end of the 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 implement **bchunklist.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. It's up to you to decide whether a dummy cell is worthwhile, but given these blocks are quite a bit bigger than the single char cells, it's worth considering forgoing that convenience and handling the special cases directly. Either approach is fine with us.
- 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.

- Pay extra attention to the `DisplayBuffer` function. This isn't a particularly interesting or complicated operation, but if your display function doesn't work correctly, it truly complicates things. If you just inserted a new character and the displayed contents of the buffer now seem wrong, did inserting create the problem or is a buggy display misleading to you? Also, don't be shy about taking advantage of the `PrintDebuggingInfo` hook to dump the internal contents of your data structure in a raw format. This might help you identify if the display is the culprit or show what part of your structure has become corrupted or inconsistent.
- 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 comes 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.
- You'll probably need to be a bit more organized than usual in your testing effort to ferret out any lurking bugs. It may help to make a list of different cases that need to be exercised so you can be sure you have nailed them all (inserting at the end of a block, beginning of a block, into a full block, at the end of the buffer, deleting the first character in a block, the last, the middle, the only remaining char, inserting/deleting after moving the cursor from one block to another, etc.).
- Your program should not orphan any memory. Memory that is no longer in used must be freed in this assignment. As always, we recommend getting the entire program working without freeing first, and then go back and add freeing.
- 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.
- Document your design choices in the code.

Part 3—Run performance trials and report on your findings

As we have seen, there are many different data structures we could choose to represent and manipulate an editor buffer, each involving tradeoffs between amount of memory required, speed of movement and editing, difficulty of writing and maintaining the code, and so on.

For each of the implementations (the three we provided and the two additional ones you wrote), use the performance trial option available in the editor client to observe and record the performance of each implementation in the worksheet. Complete the thought questions at the end of the worksheet to explore issues of time/space tradeoffs and summarize the results of your experiments.

Possible extensions

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

1. *Add a search command*. 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. *Add a search and replace command.* 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. *Add cut, copy, and paste commands.* 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!

Accessing Files

On the class web site, there are two folders of starter files: one for Mac CodeWarrior and one for Visual C++. Each folder contains these files:

editor.c	Source file which implements the editor client.
performance.h	Interface file for performance module.
performance.c	Source file which implements performance functions.
buffer.h	Interface file for the bufferADT.
barray.c	Array implementation of the bufferADT.
bstack.c	Stack implementation of the bufferADT.
bsinglelink.c	Singly-linked list implementation of the bufferADT.

To get started, create your own starter project and add **`editor.c`** and **`performance.c`** to it, as well as the desired buffer implementation file.

Deliverables

At the beginning of Wednesday's lecture, turn a manila envelope containing:

- a printout of your code for the chunklist implementation (you do not need to hand in your double-linked list version)
- the completed worksheet from the next page
- answers to the thought questions that follow the worksheet
- a floppy disk containing your entire project.

Everything should be clearly marked with your name, CS106B and your section leader's name! This is a good time to remind you that you should **never** turn in the only electronic copy of a program. Keep a backup in a safe place!

"Breathe deeply. I don't think it really helps, but at least you won't suffocate." — Via Amy McDonald

Summarize your implementation results in this worksheet. Fast operations may not even register as taking any time at all given the coarse granularity of the system clock, but slower operations will start to count as you increase the buffer size. Run the performance trial on 3 different sizes (say 1000, 2000, and 5000 -- you may have to use larger numbers to register the time on faster computers). Record the time reported for the given operations and memory used. Analyze your algorithms and determine the Big O analysis of the worst case performance of each operation as a function of N, the number of characters in the buffer.

	Array	**Stack**	**Single-link**	**Double-link**	**Chunklist**
Memory Used 1000					
Memory Used 2000					
Memory Used 5000					
Jump to start 1000					
Jump to start 2000					
Jump to start 5000					
Jump to start Big-O					
Jump to end 1000					
Jump to end 2000					
Jump to end 5000					
Jump to end Big-O					
Move forward 1000					
Move forward 2000					
Move forward 5000					
Move forward Big-O					
Move backward 1000					
Move backward 2000					
Move backward 5000					
Move backward Big-O					
Insert 1000					
Insert 2000					
Insert 5000					
Insert Big-O					
Delete 1000					
Delete 2000					
Delete 5000					
Delete Big-O					

Thought Questions (to be answered and handed in with assignment)

Take the time to answer the following thought questions about your experiment. We're not looking for long involved essays here, just a chance for you to show us that you have thought about the issues involved. A sentence or two would be just fine.

1) Do the observed timed performances match your big O analysis—do functions reported to work in constant time stay fairly constant when the queue size grows? Do linear functions double with the input size? Can you explain any big discrepancies? Do all functions that have the same complexity take the same time (a trick question, are they even supposed to)?

2) What are the strengths and weaknesses of the array? the stack? the various lists?

3) Which implementation seems the most appealing if your goal were sheer speed? What if your goal were to use as little space as possible? What if your goal were to get the code written and debugged in the shortest amount of time?

4) Repeat the performance trials for the chunklist adjusting MaxCharsPerBlock to smaller and larger numbers. What does this tell you about the relationship between chunk size and memory use and speed? What appears to be a fairly optimal range for the chunksize if the buffer holds 1000 chars? What about 5000?

5) Are there other hybrid or completely unique implementations of an editor buffer that you might propose?