CS 2123-002 Data Structures

Instructor Dr. Turgay Korkmaz

Homework 7
Due date: check BB Learn
!!!! NO LATE HOMEWORK WILL BE ACCEPTED !!!
Total 50 points

(Double Linked List of Blocks, Abstract Data Type - Library)

Programming Exercise 13 from Chapter 9. For convenience, the question is copied at the end.

First get all the files under 09-Efficiency-and-ADTs from the class web page (follow the link "programs from the textbook" under online materials part). You also need to get the textbook's library (they are also available at the class web page).

To get all files at once, I also zipped each directory. After following the link above, go to click 00-zipped-files directory and first read README.txt whic explains how to get and use books library with books programs...If you don't want to use books library, then, you need to replace some of the functions like New(), GetLine() etc. with appropriate ones in the programs...

Then, **you are asked to provide a new implementation (say bufferdllb.c) as described in Programming Exercise 13.** Note that buffer.h will be the same. Finally compile editor.c with your new implementation (bufferdllb.c) by adding appropriate commands into Makefile. And thoroughly test your implementation….

As always, make sure you release (free) the dynamically allocated memories if you allocate any memory in your programs. So, before submitting your program, run it with `valgrind` to see if there is any memory leakage… Also if you need to debug your program, compile your programs with –g option and then run it with `gdb` and/or `ddd`.

What to return: !!!! NO LATE HOMEWORK WILL BE ACCEPTED !!!

1. Create a directory, say LASTNAME_hw7, and do all your work under that directory.
2. You will get different implementation of bufferADT library and client/driver program from the textbook, and use driver program with a new implementation of buffer ADT.
3. To compile the library and driver program, you must have a `Makefile` and use "`make`."
4. After compiling, run your program a few times with different input values and save the output (using script) into `output.txt` file. So you will have around 6-7 files in your LASTNAME_hw7 directory.
5. Go to parent directory of LASTNAME_hw7, and use

> `tar -cf LASTNAME_hw7.tar LASTNAME_hw7`

This will create a new file called LASTNAME_hw7.tar and it contains all of your files. So just submit this .tar file.

6. Go to WebCT (BB), and just submit LASTNAME_hw7.tar as **attachment** before the deadline. DO NOT submit other .h or .c files individually.

/* Don't forget to include comments about the problem, yourself and each major step in your program! */

You must submit your work using Blackboard Learn and respect the following rules:

1) All assignments must be submitted as either a zip or tar archive file unless it is a single pdf file.
2) Assignments must include all source code.
3) Assignments must include an output.txt file which demonstrates the final test output run by the student.
4) If your assignment does not run/compile, the output.txt file should include an explanation of what was accomplished, what the error message was that prevented the student from finishing the assignment and what the student BELIEVES to be the underlying cause of the error.

Programming Exercise 13 from Chapter 9:

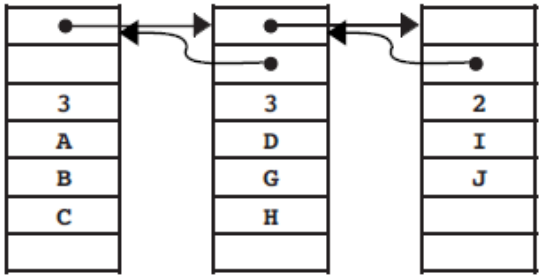
13. The biggest problem with using a doubly linked list to represent the editor buffer is that it is terribly inefficient in terms of space. With two pointers in each cell and only one character, pointers take up 89 percent of the storage, which is likely to represent an unacceptable level of overhead.

    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, since 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 decreases, but the time required to do an insertion increases. In the examples that follow, the block size is assumed to be four characters, although a larger block size would 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:

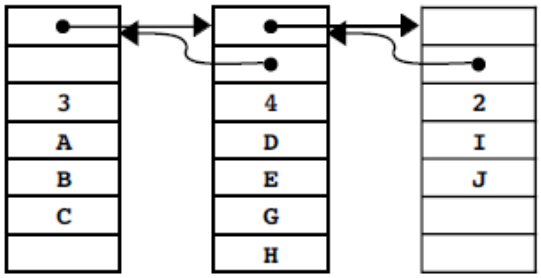


In this diagram, what you see are three blocks, each of which contains a pointer to the next block, a pointer to the previous block, the number of characters currently stored in the block, and four bytes of character storage. The actual definition of the `EditorBuffer` class and the contents of two link fields missing in this diagram (the first backward link and the last forward one) depend on your class representation, which is up to you to design. In particular, your buffer class must include some way to represent the cursor position, which presumably means that the private data members will include a pointer to the current block, as well as an index showing the current position within that block.

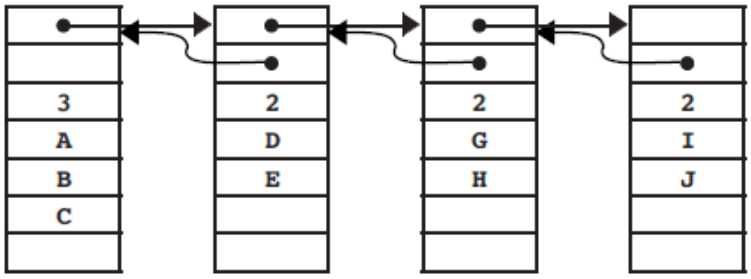
Assume for the moment that the cursor in the previous example follows the `D`, so the buffer contents are

`A B C D | G H I J`

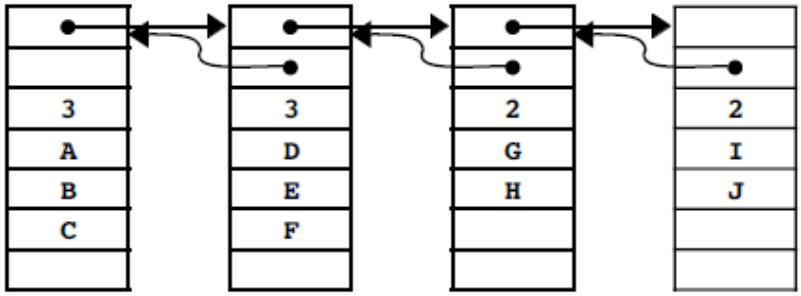
Suppose you want to insert the missing letters, `E` and `F`. Inserting the `E` is a relatively simple matter because the active block has only three characters, leaving room for an extra one. Inserting the `E` into the buffer requires shifting the `G` and the `H` toward the end of the block, but does 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 this point, it is a simple matter to insert the `F` in the second block:



Reimplement the `EditorBuffer` class so that the data representation of the buffer is a linked list of blocks, where each block can hold up to `MAX_BLOCK_SIZE` characters. In writing your implementation, you should be sure to remember the following points:

- You are responsible for designing the data structure for the `EditorBuffer` class. 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 should choose a strategy for representing the cursor that allows you to represent the possible states of a buffer in a consistent, understandable way. 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 a great deal 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 divide the 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.
- You should convince yourself that `~EditorBuffer` 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.
- You should explicitly document design decisions in your code.