# Buffer Pools and File Processing Projects for an Undergraduate Data Structures Course

Clifford A. Shaffer
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

## ABSTRACT

This paper presents a family of programming projects appropriate to a sophomore-level data structures course, centered around the concept of a buffer pool serving as the access intermediary to a disk file. These projects provide a meaningful vehicle for practicing object-oriented design techniques and teach fundamental material on file processing and manipulating binary data. I begin with a concrete example, a heap stored on disk and mediated by a buffer pool. Several important intellectual concepts introduced by such a project are enumerated. Significant extensions and alternatives to the basic project are then described. I conclude with some observations on the role of file processing in modern CS curricula, and the significance of recent trends away from coverage of these topics.

## Categories and Subject Descriptors

D.2 [Software Engineering]: Design Tools and Techniques – *object-oriented design methods*; D.4 [Operating Systems]: Communications Management – *buffering*; E.5 [Data]: Files – *organization/structure*

## General Terms

Algorithms, design

## Keywords

Buffer pools, heaps, object-oriented design, file processing, data structures

## 1. INTRODUCTION

The sophomore or junior-level data structures course (referred to sometimes as "CS7" [9]) plays a central role in many computer science departments. In surveys of Virginia Tech computer science alumni, the data structures course stands out as one of the two most important to their professional careers (the other cited course is Operating Systems). Students returning from technical job interviews regularly report data structures as being the most important topic in technical interview questions for software development positions.

At Virginia Tech, this course [3] is normally taken in the student's second semester of the sophomore year, following traditional CS1/CS2 courses in the first year and an object-oriented design course at the beginning of the second. As is typical, our students also get some introduction to data structures topics in our second freshman-year (CS2) course. The primary learning goals of our sophomore data structures course are to gain proficiency with fundamental data structures and introductory algorithm analysis, and to acquire enough self sufficiency so that students can select a suitable data structure for a task or analyze the suitability of a data structure for a task.

Over the years, design issues (particularly related to object-oriented design) have become an increasingly important part of the data structures course as I have taught it. Despite the fact that our data structures course is preceded by a course whose stated purpose is to teach object-oriented design, the projects encountered in the data structures course are often the first assignments that make a student explore significant design choices. The reason for this is that the type of projects that we elect to use are (a) of sufficient size and complexity to give rise to an opportunity for significant interactions between objects, and (b) have by nature a collection of objects whose boundaries of concern are open to some interpretation.

In recent years, there has been a trend away from topics in lower division CS curricula that ground students in the fundamental workings of their computers. For example, where years ago separate courses in data structures and in file processing were standard in many CS departments, today the Joint Taskforce Computing Curriculum 2001 [5] does not include anything equivalent to the standard CS7 course, and true file processing topics primarily appear in an elective on implementing database systems. The basic background for file processing (primitive data representation, disk and file system architecture, memory hierarchy, caching and buffering) does appear in the CC 2001 body of knowledge, but is scattered among different content subcategories and much of the most fundamental material does not appear in any of the suggested required courses. Thus, it seems likely that in the future, students will receive even less instruction in these topics than they do now. Examining popular data structures textbooks of appropriate level for CS7 courses [4, 6, 7, 8, 10] shows that while some still maintain significant coverage of such topics, others do not give more than a cursory mention of them, and these are clearly inadequate in preparing students to write programs that either involve processing data files or any form of binary data.

In this paper, I show that there is no natural antagonism between training students in object-oriented design (which tends towards abstracting away physical details) and training students in file processing techniques (which requires some appreciation for the physical characteristics of computers). Within the context of a data structures course, file processing topics naturally force students to address separation of concerns between entities, which makes for interesting and significant exercises in object-oriented design. I present a family of projects that deal with many fundamental issues of basic file processing. These projects are extremely practical in the material that they convey to students who will later be professional programmers, they have much intrinsic interest to students, they address fundamental topics that should be at the heart of any data structures course, and they naturally support instruction in object-oriented design.

## 2. THE DISK-BASED HEAP PROJECT

In this section I present one of the simplest instances of a project using a buffer pool [7] to mediate access to a file structure. The file structure is a heap. This project appears deceptively simple, yet it still manages to introduce a number of important concepts that will be enumerated in the Section 3. Section 4 suggests some related, more difficult projects and the additional ideas that they introduce.

The most important entity in this project is the buffer pool. The buffer pool is an intrinsically important data structure, since it introduces a concrete implementation for the concept of caching. Caching is important due to its real-world relevance to computer performance. The buffer pool concept demonstrates the concept of a hierarchy of memory. In particular a relatively large, slow, cheap memory (in this case a disk file) is supported by a relatively small cache of fast, expensive memory (in this case the buffer pool in main memory).

The task is to implement a heap data structure on disk. The idea is that the heap is too large to fit in main memory (although for practical reasons this is not actually true for most tests of the project). Recall that a heap is most typically implemented as an array. For this project, the array is stored in a file on disk, and the array must not be directly accessed by the heap's client. Instead, the heap may only be accessed via calls to a buffer pool object. The array is broken into physical blocks or pages of some predetermined size, typically equal to the disk sector size or a multiple of the disk sector size. (Recall that a disk sector is the basic unit of I/O permitted at the physical level, though the programming language or operating system might hide this fact from programmers via its own buffering scheme). Each page corresponds to the amount of data to be stored in a single buffer of the buffer pool. Accesses to elements in the heap are mapped to pages of the file, which the buffer pool will read into memory as necessary. Typically in my own formulations of the assignment [3], the buffer pool will be managed using a standard Least Recently Used (LRU) buffer replacement scheme.

The physical implementation of the buffer pool might use an array of buffer pointers, or a linked list of buffers. While I normally give no preference to either (there is no practical performance difference), variations on the project could ask students to experiment with the alternatives, or with variations on the LRU buffer replacement policy.

## 3. KEY PEDAGOGICAL ISSUES

This section identifies the key pedagogical issues relevant to the disk-based heap project described in the previous section.

**Binary Data and Random Access:** The heap typically stores binary data (perhaps a simple integer), and thus the file containing the heap is a binary file. For nearly all students, any previous file I/O experience will strictly be stream (i.e., sequential) processing of ASCII files. Thus, this project introduces students to basic binary file I/O (which always take place in this project as reads or writes of disk blocks corresponding to the buffer pool pages). To a limited extent, students will see the use of (for example) integer variables stored as integers on disk rather than as their ASCII representation. However, this simple version of the project does not bring out this particular concept so strongly as projects described later. Students will also see an example of file processing that is not simply streaming (sequential) in nature. Random access to the heap is required.

**Independent Entities:** The buffer pool is independent in a significant sense both from its client and from the file that stores the heap. This project strongly motivates OOP concepts, in that the buffer pool really cannot help but act as an independent agent standing between the client and the file structure. Most previous projects seen by these students, while they might incorporate OOP, typically could just as well be done procedurally. Here, the buffer pool can be viewed as a cooperating agent with the file and the heap's client.

**Design Choices:** Design choices are important to this project, most notably as they relate to the buffer pool interface. When implementing buffer pools, there are two basic approaches that can be taken regarding the transfer of information between the client of the buffer pool and the buffer pool class itself. The first approach is to pass "messages" between the two. In this case, the client provides space for the message and a value for the amount of data requested. Disadvantages of this approach are that the size of the message needs to be known by the client, and there can be considerable overhead in copying the messages. The alternative approach is to have the buffer pool provide to the client a direct pointer to a buffer that contains the necessary information. In this approach, the client is made aware that the storage space is divided into blocks of a given size, where each block is the size of a buffer. The client requests specific blocks from the buffer pool, with a pointer to the buffer holding the requested block being returned to the client. The client may then read from or write to this space. If the client writes to the space, the buffer pool must be informed of this fact. The reason is that, when a given block is to be removed from the buffer pool, the contents of that block must be written to the file if it has been modified. If the block has not been modified, then it is unnecessary to write it out. A further problem with the second approach is the risk of stale pointers. When the client is given a pointer to some buffer space at time **T1**, that pointer does indeed refer to the desired data at that time. If further requests are made to the buffer pool, it is possible that the data in any given buffer will be removed and replaced with new data. If the client at a later time **T2** then accesses the data referred to by the pointer given at time **T1**, it is possible that the data are no longer valid because the buffer contents have been replaced in the meantime. Thus the pointer into the buffer pool's memory has become "stale." To guarantee that a pointer is not stale, it should not be used if intervening requests to the buffer pool have taken place.

**Logical vs. Physical Existence:** There does not exist any single physical locus for the heap. The binary file is merely backing storage for the buffer pool, it is not necessarily the heap in its entirety. At most times, the heap does not physically exist in any one location. Rather, it is merely a logical construct spread across the totality of the buffer pool and the file. This is quite an intellectual hurdle for many students.

**Pointers:** The concept of a "pointer" has changed, and to some extent is more fully exposed to the student than has likely ever occurred to them before. This will hopefully make them confront some of their typical assumptions. For example, being sloppy about the use of "null" pointers can catch up to students here, since the most naive implementation for a null pointer is to use a value (zero) that in fact is a legal position within the binary file. However, the use of file locations in place of pointers is fairly weak in this variation project, since all accesses are probably viewed in terms of array indices instead of pointers. See Section 4 on project extensions for scenarios that better bring out the concept of byte position in place of pointers.

**Multiple Views of Data:** Computer memory is mutable. The buffer pool sees the data and a buffer as a collection of bytes accessed using void or char pointers (if implemented in **C**++). In contrast, the the client sees its own object type (an array of heap elements). A key part of the buffer pool interface design is the necessary information hiding to make the buffer pool a generic container that has no knowledge of the client's data type.

**Abstraction in Design:** Abstract thinking is required for success in several places. First, the heap data structure is inherently a good example of physical versus logical representations. [7, 1] On the one hand, the most likely physical implementation for the heap is as an array, while on the other hand its logical representation and the view it provides via its ADT is a tree. The factors listed above about how the heap structure is never physically instantiated in any one physical location also forces the student toward more abstract thinking, relying on viewing the program as cooperating ADTs rather than viewing the program in its entirety at the physical level. For many students, this could be the first time they must view a program at this abstract level if they hope to succeed. Thus, this program helps to push students away from using programming-language oriented thinking, and toward programming-free thinking. [1]

## 4. FURTHER PROJECT IDEAS

The heap-based project described in Section 2 (or using a file that essentially stores any other form of a large array) represents a relatively easy version of the general problem of using a buffer pool to mediate a file structure. Most importantly, there are no significant memory management issues, since (1) the heap elements are of fixed size and (2) the heap grows and shrinks naturally from the end. Thus, there are no gaps within the file to track; only the size of the heap needs to be remembered.

In my own course, I typically give four projects during a 15-week semester. The disk-based heap project is typically the easiest of the four, and is used as an introduction to file processing. I normally give it as the third project, with the fourth building additional file processing techniques, as described below. I normally give students about two weeks to complete it, which nearly all can do easily. The total project code is probably about five pages long.

Many variations on this project are possible, most of them being more difficult to implement. In some semesters, I've begun with a project that required students to implement some key data structure in main memory in the "normal" way (perhaps a tree structure), then had students implement a buffer pool package in a second project, and finally had students use the buffer pool to mediate a disk-based version of the original data structure in a capstone project.

One simple step up in difficulty from the heap would be a tree structure with fixed-size nodes. For example, a binary search tree (BST) could be implemented where all nodes store the same-sized data element such as an integer. When nodes are deleted from the BST, their position in the file must be remembered somehow or else a "storage leak" will have occurred. A simple solution is to implement a freelist. One of the "pointers" of the BST nodes can be used to connect together the freelist. Nodes are simply added to the freelist when removed from the BST, and new nodes inserted to the BST can be taken from the freelist if any are available (otherwise the file is expanded to "make" more nodes). Of course, the buffer pool serves as a mediator for the freelist just as it does for the rest of the program.

The project's difficulty can be increased by making tree nodes be of variable size. Now instead of a simple freelist, a full-blown memory manager [7] must be implemented. Variations such as first-fit vs. worst fit can be explored, or buddy-method implementations. Many choices are possible. Fortunately, the complexities of the memory manager can be divorced from the difficulty of the tree structure itself. Students might have implemented (for example) a BST in memory, then a buffer pool, and finally the disk-based BST.

The greatest difficulty can be achieved by combining an intrinsically difficult data structure, such as a B-tree, with the memory manager and buffer pool. This is by far the most difficult project that I've ever given to an undergraduate class. Typically only about two thirds of my students are capable of completing a basic disk-based B-tree project. Fortunately, it is possible for students to successfully demonstrate completion of subsets of the full project, such as an implementation without a completely functional memory manager.

## 5. THE ROLE OF FILE PROCESSING

Most students now complete their undergraduate CS program without ever encountering the most basic concepts of file processing. Over the past ten years or more, I've occasionally polled the students taking my graduate-level algorithms class. Not counting those who came through our own program, typically three quarters or more profess to have never written a program using binary file I/O. This observation holds for both foreign and US students.

In theory, there is no reason to discuss file processing, since the effects are "merely" constant-factor differences, and not changes in asymptotic behavior. However, since the cost to access data on disk is approximately a million times greater than to access data in memory, such differences result in great practical effects on real programs, and on how disk-bound programs should properly be implemented. In fact, the obvious time costs associated with file-based programming provide one of the few clear demonstrations of efficiency principles that students will encounter in their undergraduate programming career (in other words, their own programs run too slow if they don't do it right).

The other aspect of "file processing" topics that is of key importance is the practical aspect of working with binary data. Topics related to representation of integers versus ASCII data are part of the Computing Curriculum 2001, and are commonly taught at the freshman level. However, my experience has been that a large majority of students are unable to relate this information to actual programming, that is, they cannot make any practical use of the concepts. A large majority of students come to my sophomore-level data structures class unable to grasp the concept that UNIX text files are slightly (but significantly) different from MS Windows text files. Nor do they truly understand that an `int` variable represents an integer in a form that is radically different from its ASCII (print) representation, even if they can quote back 1's and 2's complement representations on demand. It is only once students implement a simple program that reads in an ASCII integer value (from a command line parameter or ASCII data file), stores it in an `int` variable (i.e., in binary format), writes that value in a file on disk, reads it back, and then prints it out again (thus recovering its ASCII representation), that they will grasp these issues. All computer science majors and minors should be required to do this fundamental exercise as part of some programming assignment during their lower division instruction.

One of the most striking results from a class doing any variation of the buffer pool project is the enormous range in running time among the students' implementations. It is well known that there is high variance in the run-time performance of programs produced by different programmers even when they are at the same level of professional experience [2]. It appears that this performance variation is even greater than usual for students being introduced to file processing. There can easily be an order of magnitude or more difference in the time required between the fastest and slowest programs in a typical class. Unlike typical memory-based projects, the performance difference caused by a poor implementation is great enough for students to notice in typical program testing and debugging. For many students, it is the first time that they have an intrinsic motivation to improve the efficiency of their own program, if only to get the debugging time down to something manageable.

At the same time as instruction in file processing techniques has clearly been declining, there also appears to be a trend in the programming tools used by programmers such that efficient file processing becomes more difficult. This is related to the trend toward greater abstraction in programming languages and the thinking of programmers. The ability to handle abstraction is of fundamental importance, and one of the things that distinguishes computer scientists from hack programmers. And refocusing programming languages toward higher-order tasks can hope to lead to greater programmer productivity by taking away as much mechanical burden as possible. However, it is important that our programming tools and languages achieve this abstraction in ways that do not seriously compromise program efficiency.

A good illustration of the potential pitfalls to the combination of greater abstraction in programming tools combined with less understanding of the underlying computer by programmers appears with the Java programming language's support for file I/O. In general I find that Java is much easier to use than **C**++, in part because it abstracts away the right things. However, in the case of the Java file I/O classes the particular implementation used, combined with a lack of understanding of the most basic file processing principles, leads to inefficient programs.

Java file I/O makes heavy use of the wrapper design pattern. The idea is that there is a basic file class with primitive functionality, and additional functionality can be provided by composing this basic file class with others. Typically, there will be several layers of classes wrapped within classes to achieve the desired functionality and interface. Unfortunately, the most fundamental buffering capability is not provided unless the right class is included at some point in the composition. Thus, the default is no buffering for a simple I/O stream. Simple buffering is automatic in **C**++ and most other programming languages. Programmers who have never had instruction in the fundamentals of file processing will not realize that the use of buffering is nearly always desired when doing basic sequential file I/O, but the default is not to have buffering in Java. The result is that typical programmers, if untrained in the basics of binary file I/O, will write grossly inefficient programs in Java whenever they do any file processing.

## 6. REFERENCES

[1] D. Aharoni. Cogito, ergo sum! cognitive processes of students dealing with data structures. In *Proceedings of SIGCSE 2000*, pages 26–30, Mar. 2000.

[2] F. Brooks, Jr. *The Mythical Man-Month*. Addison Wesley Longman, 1975.

[3] Department of Computer Science, Virginia Tech. CS2604 data structures and file processing website. Available at `courses.cs.vt.edu/~cs2604`, 2003.

[4] M. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 1998.

[5] J. T. on Computing Curricula. Computing curricula 2001 computer science final report, Dec. 2001.

[6] B. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. John Wiley & Sons, 2000.

[7] C. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis*. Prentice Hall, second edition, 2001.

[8] T. Standish. *Data Structures in Java*. Addison Wesley Longman, 1998.

[9] A. Tucker, B. Barnes, R. Aiken, K. Barker, K. Bruce, J. Cain, S. Conry, G. Engel, R. Epstein, D. Lidtke, M. Mulder, J. Rogers, E. Spafford, and A. Turner. *Computing Curricula '91*. Association for Computing Machinery and The Computer Society of the Institute of Electrical and Electronics Engineers, 1991.

[10] M. Weiss. *Data Structures & Algorithm Analysis in C++*. Addison Wesley Longman, 1999.