PROGRAM 02

CS310
DILLES, JACOB
3/31/08

NOTE TO T.A.:

The structure of my program may differ significantly in some ways from the assignment. These internal enhancements were made with the express permission of Prof. Nordstrom. Externally the program is indistinguishable from the less efficient methodology. Please do not hesitate to contact me with any questions or concerns at:

JDILLES@GMU.EDU

Table Of Contents

Abstract

Program 02 demonstrates the relationship between a virtualized "physical" hardware layer and a direct access file implemented using hash-code bucketed sectors holding a number of data records, in this case information on mountain names, locations, and elevations. Interaction between each real "piece" is simulated by limiting component interfaces to their physical counterparts.

Note

My implementation of overflow sectors is substantially different then the method outlined in the assignment. These changes were specifically approved by Prof. Nordstrom on 2008-03-18. Reasoning and methodology for said modifications follow.

Introduction

There are four primary classes in Program 02, plus one additional for the terminal interface. The objective of this project is simulation of a physical hard disk; as such the class responsible is thusly named **Disk**. The classes **Sector** and **Record** are object-oriented data abstractions, and do not represent their physical counterparts of the same name. The class **DirectFile** is the only one that interacts with the concept of mountains, besides the **MountPrompt** class - which manipulates a **DirectFile**. A minor but important class is **Hash**, which is used by **DirectFile** to determine which bucket a record should fall in. Although it may appear to consist of a rather large code base of 1,242 lines (45,407 bytes) the majority of it is inline comments and compiled is only 20,620 bytes. The project took approximately 19 hours to complete, and has no external data structure dependencies.

Functionality

Starting from the user’s terminal, the first encounter is with **MountPrompt.** This simple, procedural class prompts the user with several options: help, open, insert, find, remove, and quit. Although all of these operations are self explanatory, at any time the “help” command will bring up sort explanations of each command. Command processing is implemented through simple string manipulation. When the user enters a line of text followed by a LF (\n) the *Main()* method breaks the line into two parts separated by the first space. The first part is switched as the command, while the second part is passed to the command (if not null) as the argument. Thus “find Everest” at the terminal would internally call *find(“Everest”)*. All sub-methods are named according to their function. NOTE: Due to my modified overflow implementation, I was able to easily implement a *Remove()* operation, although it was not called for in the assignment, I felt that it was an appropriate function of a file, thus was included. All terminal input is regex checked and validated as much as possible to prevent the user from passing invalid commands to the **DirectFile.**

At startup, **MountPrompt** creates a **Disk**, which represents a physical hard disk in that it’s operations are limited to reading and writing fixed sectors of data. In the case of our disk, there are 2000 sectors, each holding 512 **char**s of data. In Java, the **char** is a primitive type that represents a positive number from 0 to 65,535 – that is 16 bits of data. Thus each sector can hold 8KB or 8,192 bits, and making our disk a whopping 16MB.

To further the analogy, **Disk** reads to and writes from a disk buffer, although it is not used in the same way that one would with a physical disk (Java would more likely use a **InputStream** and **OutputStream**) Internally **Disk** holds each sector as a **char[]**, and addresses the sectors in an array of char arrays, or **char[][]. Disk** will unquestionably read from or write to any sector so long as it is in its valid range.

DirectFile makes read and write operations on **Disk.** NOTE: This is where my implementation begins to differ from the original approach. Rather then dealing with each record **char** by **char** in one big file, I chose to delegate responsibility in an object-oriented fashion, such that the physical sector is now represented by class **Sector,** and the physical record now class **Record.** This allows wonderful encapsulation in that atomically distinct records can be batched in a sector without the sectors knowledge of the internals of the record. Thus **DirectFile** will work with any record (See **interface Record**) so long as it meets the interface criteria:

A **Record** must have a key that is accessible
A **Record** must represent itself as a 60 char array
A **Record** must be able to reconstruct itself from said 60 char array

Any data **Record** stores is immaterial to **Sector,** so long as it returns a **char**[60] and when constructed as *new* ***Record****(char[])* it produces the same key. The data oriented constructor for Record is implementation dependent. In the mountainous configuration, Record stores it's data as plain text in the **char[]** as follows:

```
NAME:        0 - 27
COUNTRY:    27 - 54
ALTITUDE:   54 – 60
```

Sector is significantly more complex, but has only one constructor: *new* ***Sector****(char[512]).* Thus regardless of intention, you must read a sector off the disk to create a sector object. Complete details are documented inline with the class, but in summery a sector is composed of 11 "blocks" of data which have defined positions within the 512 characters. These are as follows:

CHESUM:	0 - 12
SECNUM:	12 - 22
REFNUM:	22 - 32
RECORD0:	32 - 92
RECORD1:	92 - 152
RECORD2:	152 - 273
RECORD3:	212 - 272
RECORD4:	272 - 332
RECORD5:	332 - 392
RECORD6:	392 - 452
RECORD7:	452 - 512

The **RECORD0 – RECORD7** are 60 **char** blocks which each holds a record. On instantiation, the **Sector** chops up these into 8 arrays and creates 8 **Record** objects. Because each **Record** (by interface spec.) has a **key**, the **Sector** class can easily determine if it has a **Record** with **key** X, simply by checking with each **Record** object for a match. Because it returns the object rather then a **char[]**, it is MUCH more flexible in that the record object can be smart, encode data however it wants, or do any special operation it wishes without any modification of the **Sector** object. For instance, in an implementation where only 7 bit ASCII data was stored in a record, a special **Record** class could hold a 136-character string in only 60 **char**s, achieving a much higher data density.

Since a sector is 512 chars and a record is 60, that lets us hold eight records per sector. This leaves 32 extra chars (6%) that would, as per assignment, go to waste. This is a shame, as 1MB of our 16MB disk would be unused. NOTE: The composition of Sector is the PRIMARY deviation from the assignment. Where as originally sector buckets that overflowed were to be dumped randomly in overflow Sectors, I have used part of the unused 32 char sector to hold an address which points to the next sector which holds records from the same bucket. I chose the address to be stored as a ten digit, plain text **int**, thus capping addressable sectors at 9,999,999,999. As each sector can hold 7,680 bits of data, that imposes a limit of approximately 69.84TB per **DirectFile.** This could be increased to 7.68x10^19 bytes by using ten digits of hexadecimal instead. For comparison, the modern NTFS file system has a maximum file size of 1.84x10^19 bytes.

If the **REFNUM** is "0", the Sector has not overflowed. Otherwise, for simplicity sake, the **REFNUM** is the integer value of the next sector of the same hash bucket on the **Disk.** The last two values are **SECNUM**, which simply holds the bucket number, and **CHKSUM** stores an **int** value. **CHKSUM** is a 12 char ADLER32 checksum of the data in the Sector. NOTE: This again was not part of the assignment, but there was all that extra room not being used... It is computed on the 500 chars (not including the sum space) after all the data has been formatted. Although the Adler algorithm cannot prevent malicious tampering, it does quite effectively detect data corruption, much more so then a typical CRC. When a new Sector object is created, the same algorithm is run, and warns the user of data corruption should **CHKSUM** not match the data in the sector.

DirectFile is very simple thanks to Sector and Record. The important value in **DirectFile** is *bucketsAllocated*. This is not only the number of sectors initially written to disk, but also the limit of the hash function (see below). The first step in writing a record is hashing it's key. In the case of the assignment, this will produce a number between 0 and 599 which represents the disk address of the FIRST sector holding data for that hash bucket. In practice it is slightly more complicated, **DirectFile** may offset the original hash buckets for one reason or another. However the offset is maintained in both reads and writes and does not affect the results. Next, a **Sector** object is created when said sector address is read off of the **Disk**. If there is room in the **Sector**, the record is added there. If the **Sector** is full, an overflow sector, adds the overflow address to the former **Sector**, and writes the data to the overflow **Sector**. Reading is similar, **DirectFile** recursively follows REFNUM pointers, creating **Sector** objects and polling them for their keys.

For example, hash bucket #323 is originally stored in sector #323. If a ninth record needs to be inserted in hash bucket #323, the next available sector on the disk gets allocated as an overflow – say #634. The value 0000002000 is written to sector #323 **REFNUM,** and the record is inserted into sector #634. This allows very efficient overflow allocation because the first overflow sector is accessed almost as fast as the original (only one additional read) and the next is still N time where N is records, not sectors. To access the newest record, the key is once again hashed into #323. The sector at that address is read into a **Sector** object, which the **DirectFile** asks if it has the desired key. It does not (since the record is not there) but it DOES have an overflow sector, so it simply recuses: checking the sector, following the overflow, etc. until the record is found or the **REFNUM** is 0.

NOTE: Writing a hash function was not part of the assignment. The **Hash** class was statically encapsulated purely for testing convenience. While the assignment recommended using *java.lang.String.hashCode()* as the base hash function, I decided that it would be interesting to evaluate the distribution of said function rather then accepting it outright. I wrote a tester class that could evaluate a hash function with a 27 char key (as per assignment) generated randomly to match anticipated input data:

```
[UPPER]{1} + [lower]{2,10} + [" ",", ","'s "," - "]{1} + [UPPER]{1} +
[lower]{2,10} (MAX27)
```

or read directly from a file. The output of my test program produced results in the format:

```
i=2940,max=599,min=0,range=599,usd=596,miss=0%,maxB=11,avrB=5,score=45%
i=2940,max=599,min=0,range=599,usd=593,miss=1%,maxB=11,avrB=5,score=45%
```

Where **i**=number of trials, **max**=largest bucket#, **min**=smallest bucket#, **usd**=number of used buckets, **miss**=number of buckets not used, **maxB**=maximum keys hashed in any bucket, **avrB**=average number of records per bucket, and **score**=avrB/maxB% for easy analysis (the only thing that matters in distribution is score). As it turns out, the two hashes are very similar run on the actual data provided for the assignment. However throw some random data at it and there are a significant number of collisions. I wrote my own hash function designed specifically to deal with short length, printing letter only keys. It works by using a lookup table of 255 randomly generated 32-bit numbers. Thus for each input table[in] from 0-255 (ASCII char) there is an arbitrary, statistically random function that relates f(in) = out. The function, loosely based on a Universal Hash operates as follows. The hash is a **char** (0 - 65,535) number that starts at 0. Char was chosen over **int** or **short** to prevent rollovers from creating negative values. Each character in the input key is processed sequentially, first by looking its ASCII value up in the table. Note that the actual input value is never directly added to the hash code. Eight iterations of eight consecutive values taken from the table form a rolling window that performs a cascade of XOR operations on the hash value. The prime number shift XOR avalanche after the window loop further assists the function in providing a very well distributed hash code. After all characters have been processed, the value is modulus maximum value and returned.

File I/O

To read in the "mountains.txt" file provided with the assignment, **MountPrompt** creates a **BufferedReader** from an **InputStreamReader**, which is in turn created from a **FileInputStream** that is passed a **File** that the user specifies. Since the mountain-specific **Record** class has a constructor that deals directly with the "name#country#altitude" format, a **Record** object is created from each line. If there is invalid data **Record** throws a runtime exception that is caught in the try-catch, and simply alerts the user that the input file is invalid. This record is then added to **DirectFile**.

Also, all data handling classes (**Disk**, **DirectFile**, **Sector**, and **Record**) have dumping methods that can be used to print or write out a **char**-by-**char** copy of their contents. This proved to be very useful in debugging. Care must be taken when dumping the **Disk** class though; it takes around 10 seconds to write the entire disk to a file. Here is an example of a sector dump:

```
        Sector number: 323
        Overflow addy: 634
        Population:    8
        Checksum:      1571311202

        --Data Dump--
        ..1571311202......323.......634
        Clark Mountain............United States................8602
        Faschaunereck.............Austria......................8569
        Hochgall / Collalto.......Austria/Italy...............11272
        Manaslu...................Nepal.......................26758
        Peak E....................United States...............13230
        Red Lake Peak.............United States...............10063
        Snowy Peak................United States................3899
        Three Fingers.............United States................6870
        --End Dump--
```

From this you can see the char[] structure of all 512 bytes in the sector. Note the overflow address 634, which dumps to:

```
        Sector number: 634
        Overflow addy: 0
        Population:    1
        Checksum:      -1606874103

        --Data Dump--
        .-1606874103......323.........0
        White Princess............United States................9850
        ............................................................
        ............................................................
        ............................................................
        ............................................................
        ............................................................
        ............................................................
        ............................................................
        --End Dump—
```

Note here that the overflow address is set to 0, indicating that there are no more overflow sectors after this one.

Efficiency

A single set of unorganized overflow sectors is very inefficient for access since if a record is not found in the primary bucket sector, all overflow sectors must be scanned. This requires many disk reads (very expensive) and makes deletion very difficult because once the main sector is full no records may be removed from it. The above system requires only one read per eight records, regardless of how many overflow sectors exist. Moreover, deletions are easy and efficient; you may remove records at will so long as the overflow pointer remains set. If a great many deletions occur, a file can be compacted by reading all records from a given bucket into an array of **Record** objects, wiping the records from all Sectors from said bucket, then filling in **Record** objects starting in the original **Sector**. A simple population and overflow counter maintained for each bucket could aid in determining when compacting was necessary. After all sectors are compacted one may create a map of the overflow area, deleting any empty sectors, and moving those farthest away to the front. Since any overflow **Sector** is aware of its bucket number, this can be carried out with minimal disk I/O.

While vastly more efficient in access time, this system does use more space then a mass-overflow implementation. Specifically because each overflow sector is tied to a specific bucket, there are often overflow sectors that are not full, some having only one or two records. With the data provided, this translates to 37 overflow sectors holding 62 records. In a mass overflow system, said records would occupy only 8 sectors. It is a direct tradeoff between performance and space: a given overflow sector in my system is directly accessed, while all 8 sectors otherwise must be scanned on every overflow access. Though ultimately the entire purpose of a hash based direct file is performance over space; technically the entirety of the sample data could be stored on 368 (as opposed to 637) unordered sectors, but to find a given record would take 368 disk reads. My implementation of overflow records is an extension of this philosophy.

(2008-03)
JSD