University of Toronto

csc228S - File Structures and Data Management, Fall 2001

Midterm Test

Monday, October 22, 2001

Duration: 50 minutes

Aids allowed: None

Family Name: ______________________ Given names: ______________________

Student #: ______________________ Tutor: ______________________

- There are 10 pages, including this one. The test is out of 85 marks and the value of each question is provided; please use this information to manage your time effectively.
- Answer all questions directly on the exam in the space provided (I've give you lots). If you need more space, use the back of the exam sheet and indicate clearly where your answer is continued.

Part A:	______	/ 25
Part B:	______	/ 20
Part C:	______	/ 20
Part D:	______	/ 20
Total	______	/ 85

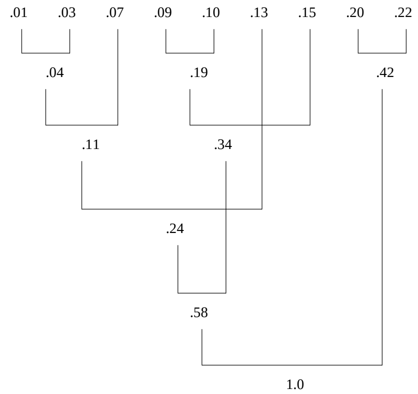
Part A [25 marks in total]

As part of your Napster enterprise, you've discovered a new genre of minimalist music, in which all songs contain only nine different notes. You figure that with a variable-length compression scheme, you might be able to compress your audio files and save space on your disk. You decide on a Huffman coding of the data, and after some analysis of a collection of songs, come up with the following probabilities of the set of nine symbols in the uncompressed files:

Symbol:	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9
Probability:	.01	.03	.07	.09	.10	.13	.15	.20	.22

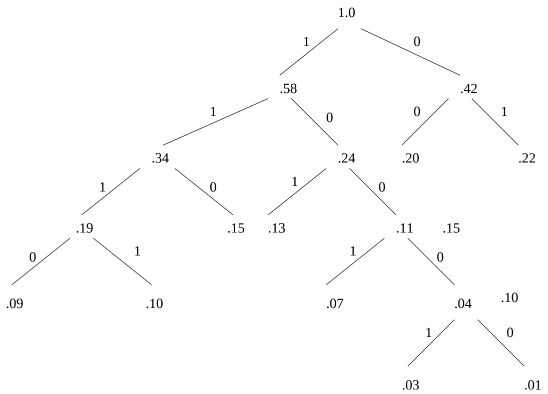
(a) (10 marks) Derive the Huffman source reduction tree. Show the probability at each node in the tree, including the root.

`ANSWER`



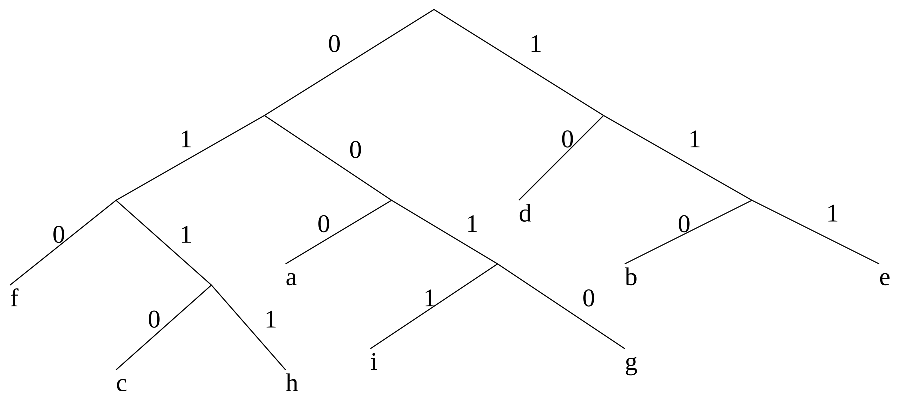
(b) (6 marks) Assign Huffman codes to each symbol. Begin by **annotating your tree in part (a)** by assigning a bit to each branch in the tree according to the policy that the branch to the larger probability is assigned a "1". Note that this will result in a **unique** code assignment. Next, using your annotated tree, compute the code assignments for each symbol and show them in a table.

`ANSWER`



a1	10000
a2	10001
a3	1001
a4	1110
a5	1111
a6	101
a7	110
a8	00
a9	01

(c) (2 marks) Consider the following hypothetical Huffman tree below (whose probabilities at both internal and leaf nodes have been removed for clarity).



Show the resulting sequence of codes in the compressed file for the following input strings:

1. adcbgf
2. eehcid

ANSWER

```
0001001101100010010
1111110111011000 1110
```

(d) (2 marks) Using the Huffman tree in part (c), decode each of the following compressed sequences:

1. 00000011111100110110
2. 0101001010010110111

ANSWER

```
aaeeic
fdfdfbe
```

(e) (5 marks) Consider an audio file containing 1,000,000 symbols consisting of equal numbers of only two symbols, one 3 bits long, while the other is 5 bits long. If your disk has 50 bytes/sector and 400 sectors/track, then how many platters would be required to hold the compressed file such that reading the entire file requires only one seek? You may assume that when the file is compressed, codes are not aligned to byte boundaries, i.e., codes may span byte boundaries or multiple codes (if they are small) may reside in a single byte.

```
ANSWER

- 1,000,000 symbols means 500,000 of each
- that's 500,000 x 5 bits + 500,000 x 3 bits = 500,000 bytes
- 500,000 bytes @ 50 bytes/sector = 10,000 sectors
- 10,000 sectors @ 400 sectors/track = 25 tracks
- in order for there to be only one seek, the entire compressed
  file needs to reside on one cylinder. Since each platter has
  two surfaces, we need 25 surfaces = 13 platters (one surface left over)
```

Part B [20 marks in total]

Following up on your Napster work in Assignments 1 and 2, you wish to develop a file structure for manipulating album records. Specifically, you want to support the query of album records by artist. Thus, the prompt, "Neil Young", would return all albums recorded by Neil Young. In the following questions, you will be asked to design appropriate file structures for indexing. Note that indexes must be kept in files.

(a) (6 marks) If album records are variable length and there are no primary keys, describe an effective indexing file structure.

```
ANSWER

If there are no primary keys, then an artist index must point directly
to a set of album records. In this case, since the records are
variable length, the pointers would be byte offsets. To implement
this, you would need to first create a file containing a set of
fixed-length records. Each of these fixed-length records contains a
fixed-length artist name and a fixed-length byte offset to a second
file. This second file can be designed in two ways (you only need to
provide one method to get the marks). It may contain variable length
records, each containing a collection of byte offsets to the original
album file. Alternatively, it may contain fixed length records, each
containing a byte offset and a pointer (RRN) to another byte offset
record, effectively forming a linked list. This second option is
preferrable, especially when new records (offsets) are added. The
first file is sorted by artist name and allows efficient search of an
index through direct access. That gets you to the second file, which
actually holds the results of the index operation, i.e., byte offsets
to the album records that you want. Since the album records are
variable width, you can't use RRN's (as you did in Assignment #2) but
rather byte offsets.
```

(b) (4 marks) If each album record is assigned a primary key, describe an effective indexing file structure.

```
ANSWER

This is similar to the file structure above with a couple of
changes. In the ''second'' file, all byte offsets are replaced by
primary keys. This now means we need a third file of fixed-length
records mapping primary keys to album file byte offsets. This third
file should be sorted by primary key.
```

(c) (4 marks) Given your design in (b), if an album record is deleted, what effect would this have on your indexing file structure?

ANSWER

If an album record is deleted, its corresponding primary key entry in the ''third'' file is flagged. No changes are made to the secondary index files, as deleted album records are ''screened'' in the primary key index.

(d) (6 marks) Given your design in (b), if a new album record is added, what effect would this have on your indexing file structure?

ANSWER

If a new album is added, we need to define a new primary key. This key would need to be inserted into the primary key index file and the file would need to be resorted. If the artist has been encountered before, there should be an entry for it in the secondary key index. How to add the primary key depends on your design (again, you only need to provide one method). If you chose a variable-length record containing all primary keys for a given artist, then you have the problem of trying to add something to a record that may not have room to accommodate it. In that case, you would have to create a new record that's big enough, copy the contents of the old record plus the new key to the new record, and update the secondary key index artist entry to point to this new record. The old record is added to the available record list, and a suitable placement strategy would be used to manage the available space. The better solution is offered by the fixed record length approach, in which a new record is simply added to the end of the file, with a simple pointer update for the previous ''end of list''. If the artist is new, a new index entry would need to be created in the first file, and the file would need to be resorted. This new entry would point to a new record in the second file (regardless of whether the second file is fixed-length or variable-length) containing the new primary key.

Part C [20 marks in total]

You're asked to design a file structure for a set of records, each pertaining to a movie that was made during the year 2000. The parameters of your design are as follows:

1. The records in the file will be sorted by date.
2. There will be no additions or deletions of records in the file once it's created, as there were a finite number of movies made in 2000.
3. You want to be able to search the file by date, therefore direct access is necessary.
4. Therefore the records are fixed-length.
5. The fields in your fixed-length records are variable-length.
6. Depending on the field contents, all fields may not fit on a single record.
7. No record information is to be lost.
8. Fields are never empty.
9. The file structure is restricted to one file.

(20 points) Propose a record file structure that maximizes space efficiency and supports the above parameters. How is a new record added?

```
ANSWER:

(Note that my solution is quite verbose, and includes some details
that are not required for full marks. Alternative, acceptable designs
will also get you the marks.)

Within a fixed-length record, we will first add a single-byte field
that indicates whether or not the information fits on a single
record. If not, then the byte is set, and a byte offset field
immediately follows. This byte offset will point to a location in the
file where the data in the record will continue. Note that this byte
offset points to a position in an ``overflow'' set of records placed
after the last record in the sorted sequence.

Following the single-byte field (and possibly the offset field, in the
case that the byte is set) come the variable-length fields. If a field
can be accommodated in the remaining space, it is added to the
record. Otherwise, the field occupies the remaining space in the
record with the remainder written to a position in the overflow record
area specified by the head pointer (stored at the beginning of the
file). Since space is a premium, field delimeters should be used to
manage the variable-length fields, although byte count is
acceptable. Keyword-value is not space efficient. Since fields are
never empty, we don't need a record delimeter
```

The head pointer is initialized to the first byte of the first record following the sorted sequence. The first time a record ''overflows'', a new record is added to the end of the file. After the overflow information is written out to the new record (or records, if more than one is required), the head pointer will either point to the next free byte in one of the overflow records, or to the first byte in a new, unused record (actually, such a record could be created only when needed by checking to see if the byte offset modulo the record length is zero - this check would work in the case of the first overflow record as well). Note that overflow information for a record is stored in contiguous overflow records if more than one overflow record is required. Therefore, no byte offset is required to move from one overflow record to another.

Part D [20 marks in total]

In Assignment 2, you are required to cosequentially process two intermediate files, each representing the output of a component of a query. One file contains songs of a given artist, while the other file contains songs of a particular genre. Your program will perform a cosequential match function, selecting those records common to both files. We also looked at the cosequential match in detail in class.

If the compound query were extended to allow any number of conjuncts, then you would need to perform a k-way match across k intermediate files, i.e., output the record if it's common to all k files. In the space below, provide a cosequential algorithm that performs a k-way match. Let your function be called "k-way-match" and have it take 2 arguments (NumFiles, FileArray). Assume that you have at your disposal a file class with three methods. Method "Next" updates some private member variable to point to the next record in the file or eof; it returns a value of 0 if the file is empty and 1 otherwise. Method "Curr" returns the primary key of the record pointed to by the private member variable; you can assume that the files are sorted in increasing order by this primary key. Finally, method "Output" outputs the record pointed to by the private member variable. **IMPORTANT:** You can also assume the existence of a function, "GetIndexOfMaxCurr", which takes NumFile and FileArray as arguments, and returns the index of the file whose "Curr" is maximum. Please write simple pseudocode, not C++ code. And assume that the above methods/functions are provided - don't write pseudocode for them. You don't need to provide any declarations, open/close files, etc. - simply show the essential steps in the algorithm. BTW, my solution is 15 lines.

ANSWER:

```
k-way-match (num_files, file_array)
no_files_empty = 1;
for i = 1 to num_files do
   no_files_empty = no_files_empty && Next(file_array[i])
while (no_files_empty)
   match = 1
   max_index = GetIndexOfMaxCurr(num_files, file_array)
   for i = 1 to num_files do
      if Curr(file_array[i]) < Curr(file_array[max_index])
         no_files_empty = Next(file_array[i])
         match = 0
   if match == 1
      Output(file_array[1])
      for i = 1 to num_files do
         no_files_empty = no_files_empty && Next(file_aray[i])
```

```
Note: alternatively, you could replace the contents of the for loop
with the following:

      while Curr(file_array[i]) < Curr(file_array[max_index])
         no_files_empty = Next(file_array[i])
      if Curr(file_array[i]) > Curr(file_array[max_index])
         match = 0
```