NC STATE UNIVERSITY

CSC 541
Assignment 2

In-Memory Indexing with Availability Lists

Introduction

The goals of this assignment are three-fold:

1. To investigate the use of field delimiters and record sizes for field and record organization.
2. To build and maintain an in-memory primary key index to improve search efficiency.
3. To use an in-memory availability list to support the reallocation of space for records that are deleted.

Student File

During this assignment you will build and maintain a simple file of student records. Each record will have four fields: `SID` (student ID), `last` (last name), `first`, (first name) and `major` (program of study). Fields within a record will be variable-length, and will be delimited by the `|` character. For example

```
712412913|Ford|Rob|Phi
```

represents a student with an `SID` of 712412913, a `last` of Ford, a `first` of Rob, and a `major` of Phi (Rob Ford is minoring in Ethics).

`SID` is the primary key for a student record. This means every individual student record will have a unique `SID`.

Records will be variable-length, and will be stored one after another in a binary data file. Each record will be preceded by an integer that defines the size of its corresponding record.

Note. Read the above description of the record size carefully! It is stored as binary data in a manner similar to how integer data were stored and retrieved in Assignment 1. It is not appropriate to store the size as an ASCII string. For example, if you wanted to read a record at file offset `off` in a file referenced through a `FILE` stream `fp`, it would be done as:

```
#include <stdio.h>

char *buf;      /* Buffer to hold student record */
FILE *fp;       /* Input/output stream */
long  rec_off;  /* Record offset */
int   rec_siz;  /* Record size, in characters */

/*  If student.db doesn't exist, create it, otherwise read
 *  its first record
 */

if ( ( fp = fopen( "student.db", "r+b" ) ) == NULL ) {
  fp = fopen( "student.db", "w+b" );
```

```
} else {
  rec_off = 0;
  fseek( fp, rec_off, SEEK_SET );
  fread( &rec_siz, sizeof( int ), 1, fp );

  buf = malloc( rec_siz );
  fread( buf, 1, rec_siz, fp );
}
```

Writing a new record uses a similar, reverse procedure. First, convert the record's body into a character buffer with fields delimited by `|`. Next, seek to the appropriate position in the student file and write an integer representing the buffer's size in bytes, in binary, using `fwrite()`. Finally, write the buffer to the file with `fwrite()`.

Program Execution

Your program will be named *assn_2* and it will run from the command line. Two command line arguments will be specified: an availability list order, and the name of the student file.

```
assn_2 avail-list-order studentfile-name
```

Your program must support three different availability list ordering strategies.

1. `--first-fit` Holes in the availability list will be saved in the order they are added to the list.
2. `--best-fit` Holes in the availability list will be sorted in ascending order of hole size.
3. `--worst-fit` Holes in the availability availability list will be sorted in descending order of hole size.

For example, executing your program as follows

```
assn_2 --best-fit student.db
```

would order holes in the availability list in ascending order of hole size, and would use `student.db` to store records.

Note. If you are asked open an existing student file, you can assume the availability list order specified on the command line matches the order that was used when the student file was first created.

In-Memory Primary Key Index

In order to improve search efficiency, a primary key index will be maintained in memory as a reference to each record stored in the file. For our records, `SID` will act as a primary key. This means each entry in your index will have a structure similar to:

```
typedef struct {
  int key;   /* Record's key */
  long off;  /* Record's offset in file */
} index_S;
```

Index entries should be stored in a collection that supports direct access and dynamic expansion. One good choice is a dynamically-expandable array. Index entries must be sorted in ascending key order, with the smallest key is at the front of the index and the largest key is at the end. This will allow you to use a binary search to find a target key in the index.

The index will not be re-built every time the student file is re-opened. Instead, it will be maintained in a permanent form on-disk. As your program exits, you will write your index to disk, saving its contents in an index file. When you re-open the student file, you will load the corresponding index file, immediately reconstructing your primary key index. The index will have exactly the same state as before, and will be ready to use to access records in the student file.

You can use any format you want to store each key–offset pair the index file. The simplest approach is to read and write the entire structure as binary data using `fread()` and `fwrite()`, for example

```
#include <stdio.h>

typedef struct {
```

```
  int key;   /* Record's key */
  long off;  /* Record's offset in file */
} index_S;

FILE    *out;         /* Output file stream */
index_S  prim[ 50 ];  /* Primary key index */

out = fopen( "index.bin", "wb" );
fwrite( prim, sizeof( index_S ), 50, out );
fclose( out );
```

Note. To simplify your program, you can assume the primary key index will never need to store more than 10,000 key–offset pairs.

In-Memory Availability List

When records are deleted from the file, rather than closing the hole that forms (an expensive operation), we will simply record the size and the offset of the hole in an in-memory availability list. Each entry in your availability list will have a structure similar to:

```
typedef struct {
  int siz;   /* Hole's size */
  long off;  /* Hole's offset in file */
} avail_S;
```

Note. To simplify your program, you can assume the availability list will never need to store more than 10,000 size–offset pairs.

The availability list will not be re-built every time the student file is re-opened. Instead, similar to the primary index, it will be maintained in a permanent form on-disk. As your program exits, you will write your availability list to disk, saving its contents in an availability list file. When you re-open the student file, you will load the corresponding availability list file, immediately reconstructing your availability list.

As noted above, you can assume a consistent availability list order for a given student file. In other words, if you are asked to open an existing student file, the availability list order specified on the command line will always match the order that was being used when the availability list was written to disk.

When new records are added, we will search the availability list for a hole that can hold the new record. If no hole exists, the record is appended to the end of the student file. The order of the entries in the availability list is defined by the availability order specified on the command line when your program is run.

First Fit

If your program sees the availability order `--first-fit`, it will order entries in the availability list in first in–first out order. New holes are appended to the end of the availability list. When a new record is added, a first-fit strategy is used to search from the front of the availability list until a hole is found that is large enough to hold the new record.

If the hole is larger than the new record, the left-over fragment is saved as a new hole at the end of the availability list.

Best Fit

If your program sees the availability order `--best-fit`, it will order entries in the availability list sorted in ascending order of hole size. New holes are inserted into the availability list in the proper sorted position. If multiple holes have the same size, the entries for these holes should be sorted in ascending order of hole offset.

When a new record is added, a best-fit strategy is used to search from the front of the availability list until a hole is found that is large enough to hold the new record. Because of how the availability list is ordered, this is the smallest hole that can hold the new record.

If the hole is larger than the new record, the left-over fragment is saved as a new hole at its sorted position in the availability list.

Hint. Use C's `qsort()` function to sort the availability list.

Worst Fit

If your program sees the availability order `--worst-fit`, it will order entries in the availability list sorted in descending order of hole size. New holes are inserted into the availability list in the proper sorted position. If multiple holes have the same size, the entries for these holes should be sorted in ascending order of hole offset.

When a new record is added, a worst-fit strategy is used to examine the first entry in the availability list to see if it is large enough to hold the new record. Because of how the availability list is ordered, this is the largest hole that can hold the new record.

If the hole is larger than the new record, the left-over fragment is saved as a new hole at its sorted position in the availability list.

Hint. Use C's `qsort()` function to sort the availability list.

User Interface

The user will communicate with your program through a set of commands typed at the keyboard. Your program must support four simple commands:

- `add key rec`
  Adds a new record `rec` with an `SID` of `key` to the student file. The format of `rec` is a `|`-delimited set of fields (exactly as described in Student File section above), for example

  ```
  add 712412913 712412913|Ford|Rob|Phi
  ```

  adds a new record with an `SID` of 712412913, a `last` of Ford, a `first` of Rob, and a `major` of Phi.
- `find key`
  Finds the record with `SID` of `key` in the student file, if it exists. The record should be printed in `|`-delimited format, (exactly as described in Student File section above), for example

  ```
  find 712412913
  ```

  should print on-screen

  ```
  712412913|Ford|Rob|Phi
  ```

- `del key`
  Delete the record with *SID* of `key` from the student file, if it exists.
- `end`
  End the program, close the student file, and write the index and availability lists to the corresponding index and availability files.

Add

To add a new record to the student file

1. Binary search the index for an entry with a key value equal to the new `rec`'s `SID`. If such an entry exists, then `rec` has the same primary key as a record already in the student file. Write

   ```
   Record with SID=key exists
   ```

   on-screen, and ignore the add request, since this is not allowed. If the user wants to update an already-existing record, they must first delete it, then re-add it.
2. Search the availability list for a hole that can hold hold `rec` plus the record size integer that precedes it.

   If a hole is found, remove it from the availability list, and write `rec`'s size and body to the hole's offset. If the hole is bigger than `rec` plus its record size integer, there will be a fragment left at the end of the hole. Add the fragment back to the availability list as a new, smaller hole.

   If no appropriately-sized hole exist in the availability list, append `rec`'s size and body to the end of the student file.
3. Regardless of where `rec` is written, a new entry must be added to the index holding `rec`'s key and offset, maintaining the index in key-sorted order.

Find

To find a record, binary search the index for an entry with a key value of `key`. If an entry is found, use it's offset to locate and read the record, then print the record on-screen.

If no index entry with the given key exists, write

```
No record with SID=key exists
```

on-screen.

Del

To delete a record, binary search the index for an entry with a key value of `key`.

If an entry is found, use the entry's offset to locate and read the size of the record. Since the record is being deleted, it will form a hole in the student file. Add a new entry to the availability list containing the new hole's location and size. Remember, the size of the hole is the size of the record being deleted, plus the size of the integer preceding the record. Finally, remove the entry for the deleted record from the index.

If no index entry with the given key exists, print

```
No record with SID=key exists
```

on-screen.

End

This command ends the program by closing the student file, and writing the index and availability lists to their corresponding index and availability list files.

Programming Environment

All programs must be written in C, and compiled to run on the `remote.eos.ncsu.edu` Linux server. Any ssh client can be used to access your Unity account and AFS storage space on this machine.

Writing Results

When your program ends, you must print the contents of your index and availability lists. For the index entries, print a line containing the text `Index:` followed by one line for each key–offset pair, using the following format.

```
printf( "key=%d: offset=%ld\n", index[i].key, index[i].off );
```

Next, for the availability list entries, print a line containing the text `Availability:` followed by one line for each size–offset pair, using the following format.

```
printf( "size=%d: offset=%ld\n", avail[i].siz, avail[i].off );
```

Finally, you must determine the number of holes `hole_n` in your availability list, and the total amount of space `hole_siz` occupied by all the holes (*i.e.*, the sum of the sizes of each hole). These two values should be printed using the following format.

```
printf( "Number of holes: %d\n", hole_n );
printf( "Hole space: %d\n", hole_siz );
```

This will allow you to compare the efficiency of different availability list orderings to see whether they offer better or worse performance, in terms of the number of holes they create, and the amount of space they waste within the student file.

Your assignment will be run automatically, and the output it produces will be compared to known, correct output using `diff`. Because of this, **your output must conform to the above requirements exactly**. If it doesn't, `diff` will report your output as incorrect, and it will be marked accordingly.

Supplemental Material

In order to help you test your program, we provide two input and two output files. The input files contain commands for your program. You can use file redirection to pass them in as though their contents were typed at the keyboard.

- `input-01.txt`, an input file of commands applied to an initially empty student file, and
- `input-02.txt`, an input file of commands applied to the student file produced by `input-01.txt`.

The output files show what your program should print after each input file is processed.

- `output-01-first.txt`, the output your program should produce after it processes `input-01.txt` using `--first-fit`,
- `output-02-first.txt`, the output your program should produce after it processes `input-02.txt` using `--first-fit`,

- `output-01-best.txt`, the output your program should produce after it processes `input-01.txt` using `--best-fit`,
- `output-02-best.txt`, the output your program should produce after it processes `input-02.txt` using `--best-fit`,
- `output-01-worst.txt`, the output your program should produce after it processes `input-01.txt` using `--worst-fit`, and
- `output-02-worst.txt`, the output your program should produce after it processes `input-02.txt` using `--worst-fit`.

For example, to test your program using `--best-fit`, you would issue the following commands:

```
% rm student.db
% assn_2 --best-fit student.db < input-01.txt > my-output-01-best.txt
% assn_2 --best-fit student.db < input-02.txt > my-output-02-best.txt
```

Note: As shown in the example above, you start a "new" student database by removing any existing student file. If your program sees the student file doesn't exist, it can assume that the index and availability files shouldn't exist, either. You can handle this assumption in any way you want. One simple approach would be to open the index and availability files in `w+b` mode, which enables reading and writing, and automatically discards any existing files with the same names.

You can use `diff` to compare output from your program to our output files. If your program is running properly and your output is formatted correctly, your program should produce output identical to what is in these files.

Please remember, the files we're providing here are meant to serve as examples only. Apart from holding valid commands, and the previous guarantees of a limit of 10,000 key–offset and 10,000 size–offset pairs, **you cannot make any assumptions** about the content of the input files we will use to test your program.

Hand-In Requirements

Use Moodle (the online assignment submission software) to submit the following files:

- `assn_2`, a Linux executable of your finished assignment, and
- all associated source code files (these can be called anything you want).

There are four important requirements that your assignment must satisfy.

1. Your executable file must be named exactly as shown above. The program will be run and marked electronically using a script file, so using a different name means the executable will not be found, and subsequently will not be marked.
2. Your program must be compiled to run on `remote.eos.ncsu.edu`. If we cannot run your program, we will not be able to mark it, and we will be forced to assign you a grade of 0.
3. Your program must produce output that exactly matches the format described in the Writing Results section of this assignment. If it doesn't, it will not pass our automatic comparison to known, correct output.
4. You must submit your source code with your executable prior to the submission deadline. If you do not submit your source code, we cannot MOSS it to check for code similarity. Because of this, any assignment that does not include source code will be assigned a grade of 0.

Updated 07-Feb-18