Simple indexed record program

You will write a program, which will manipulate a set of fixed-length records on file by accessing them though an index data structure in memory. Normally, when the program begins it will read the index file from the disk. The index file will contain the byte offsets to the records, and the records will be maintained ordered in the index. Adding a record will append the record to the file, and insert the index entry into the index data structure.

Deleting a record requires the removal of the index entry from the data structure. For now, we will not reclaim this space; therefore the data record may be left in the data file. If **add** is called the program should insert into the index, as well as appending data to the data file.

A simple record may be made to include the following fields:

ID Number:	integer	4 bytes
First Name:	C++ array	15 bytes
Last Name:	C++ array	15 bytes
Profession:	C++ array	16 bytes
Total bytes:		50 bytes

You should create a class with these fields. The class should include a constructor, an overloaded *friend* output operator << ( ), as well as a binary read and binary write methods (similar to *fread*() and *fwrite*() I showed in the example in class).

You really need to write two programs. Program #1 will read from a text file some initial records. For example the file might look like the following:

```
12345
John
Anderson
Engineer
14356
Ellen
James
Technician
. . .
```

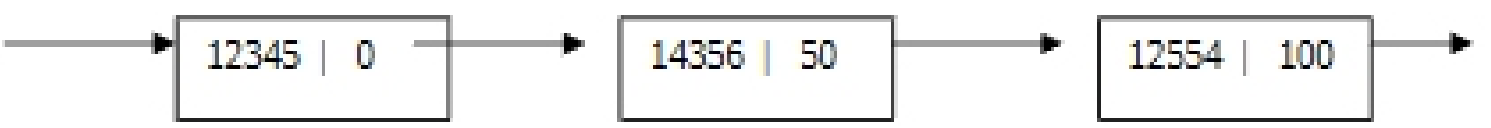
This program will open a binary file to be written. It will read four lines of the text file into four variables, then construct an object with the variables as parameters, and invoke the *fwrite* ( ) method to write the object as a record into the binary file. The program will continue to read from the text file and write to the binary file as long as there are records

in the file. As the program is writing the records, it must at the same time write the index file. The index file will keep the key and offset address for each record written. The index file may be a simple text file. For example, if we use the text sample I gave above, the index file might look like the following:

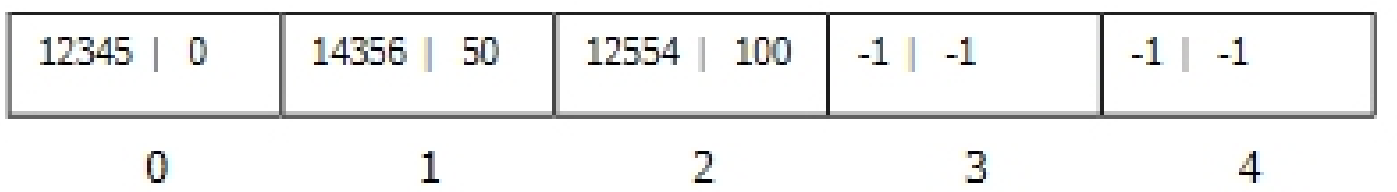
12345
0
14356
50
12554
100

12345 is the first key and 0 is the offset in the file to find that record. 14356 is the second key, and it is 50 bytes from the beginning of the file, etc.

Program #2 (the real program) will start by reading the index file, and creating a data structure to hold it. We need to create a new class, called **IndexEntry** which will have two data members, a key and an offset – both can be integer values. We can make the index be a linked list of nodes containing IndexEntries or it could also be an array of such entries. Below we see a possible linked list implementation



An array implementation could look like the following.



When the user needs to delete a record, the linked list version might simply, delete the IndexEntry node from the list, and then delete the memory allocation. Any subsequent search would no longer find the entry in the index. In the array implementation, we need to *pre-allocate* the memory, so unused entries would have –1 value to set them apart. An insert would simply use the first node with a –1 entry value. Deletes are more complex, because if we delete, say 14356, then we have a –1 node in the middle of valid nodes. A possible solution would be to have a data member with the number of valid entries in the index. Then a delete could simply copy the *last* entry over the one to be deleted. The last entry would then have a –1 placed into it to invalidate it. For example, if we delete 14356 from the array-implemented index, we would have a structure that looks like the following:

12345 | 0	12554 | 100	-1 | -1	-1 | -1	-1 | -1
0	1	2	3	4

In this way, all the valid index entries are always contiguous.

You are to write a simple menu-driven program to allow a user to display a record by key, delete a record, add a record, modify a record, display all records, and quit. To display a record by key, the user types in the key, and the program searches the index for that entry. If it is not found, a message reports that. If it is found, then the program goes to the correct offset (stored in the index entry) using a seek operation, then does an *fread* ( ) to load the object from the file. The object can de displayed using the overloaded << ( ) operator. Deleting and adding records also requires a search. In deleting, if search shows that the key is not in the index, the record does not exist. For adding, if a record exists, then it cannot be added. If it can be added, the user is prompted for all the fields, and the record is appended into the binary data file and an index entry is made for it.

For efficient searching, you may keep the index in key order. This is not required for this project. When the user quits, the program needs to write the index data structure to disk by overwriting the existing index file. In this way, when the program is restarted, the new index file will correctly reflect what exists in the binary data file.