Scheduling GIS 4: B+ Tree

Assignment:

For this project, you will implement a moderately large database of seminar records. Users will be able to insert and delete seminar records from the database, and search for seminars by ID or range of IDs. You will implement a B^+-tree, mediated by a buffer pool, to store the seminar records and support the search queries.

Input and Output

There will be two input parameters to the program. The first will be the name of a file that will be used to store the B^+-tree. The second parameter will be the number of buffers in the buffer pool.

Your program will also read a command file from standard input (`stdin`) and write responses to the commands to standard output (`stdout`). The input command file conains a series of commands (some with associated parameters, separated by spaces), one command for each line. No command line will require more than 120 characters. Commands are free format in that an arbitrary number of additional spaces may be interspersed between parameters, and blank lines may appear between commands. You do not need to check for syntax errors in the command lines.

Each input command should result in meaningful feedback in terms of an output message. Each input command should be echo'ed to the output.

Commands and their syntax are as follows.

insert *ID title date length cost*

A seminar record contains the following fields:

- ID: An integer that uniquely identifies the seminar.
- Title: A string, beginning with a letter, and may contain letters, underscores, and digits.
- Date/Time: A string in the format YYMMDDHHmm where YY is the last two digits of the year, MM is the month, DD is the date, HH is the hour (24-hour clock) and mm is the minutes.
- Length: An integer representing the seminar length in minutes.
- Cost: An integer representing the seminar cost in whole dollars.

It is an error to attempt to insert a seminar with the same ID as a seminar already in the database.

dump

The B^+-tree is printed out, one node printed on each line. (Note that these output lines could get rather long.) Internal nodes should print (1) the block number for the node, (2) the count of the number of children for that node, and (3) a series of key values and block numbers for the children as stored in the internal node's child array. Leaf nodes should print (1) the block number for the node, (2) the count of the number of records stored in that node, and (3) for each record, the key value and the length of that record.

search *flag ID* [*ID*]

The first parameter for the search routine is the debug print flag. This will take the form of either a '+' or a '-'. If the flag is '+', then each node visited during the search (both internal and leaf nodes) will be printed using the format described for the dump command, with one node per line. If the flag is '-' then these nodes will not be printed. Search may take two forms. It either has one integer parameter, or two. If it has only one parameter, then the seminar record (if any) with that ID is printed. If the search command has two parameters, than all records with IDs within that range will be printed out, one record per line. If there are two parameters, the first ID value will be less than the second.

delete *ID*

Remove the record (if any) which matches the ID.

Implementation:

Your database will be stored in a B^+-tree, indexed by ID value. This B^+-tree will reside in a disk file, mediated by a buffer pool using the LRU buffer replacement scheme. Blocks in the file will be 512 bytes long, and thus, nodes of the B^+-tree will also be 512 bytes long (each block of the file will correspond exactly to one node in the B^+-tree). You B^+-tree's internal nodes and leaf nodes should be implemented as separate subclasses derived from an abstract B^+-tree node class.

The bulk of the space in an internal node will be taken up by an array of key/pointer pairs. In this case, the "pointer" that you store will actually be the block number for the child node, since each B^+-tree node corresponds to a disk block. Internal nodes must also store additional information, such as a flag to indicate that it is an internal node, its block number, and how many children it currently has. You have some flexibility in exactly what is stored in the internal nodes, but since the key/pointer pairs should require 8 bytes each, and since the node is 512 bytes in length, your internal node should be storing an array of about 60 key/pointer pairs. This means that the B^+-tree has branching factor of around 60. Thus, every internal node (except possibly the root node) has between 30 and 60 children.

Seminar records are variable length. Leaf nodes of the B^+-tree store the actual seminar records. Thus, you will need to take the various fields of the seminar record and package them up as a series of bytes to be stored in some B^+-tree leaf node. When a record is found to match a search query, you will need to unpack this series of bytes to get the various fields back for printing out. Note that some fields in the seminar record are integers (and must be stored as such in the database) and others are ASCII (and must be stored as such). Be careful about how you deal with terminating ASCII strings. You could choose to store a string with a length field, or terminated by a null character.

Since the records are variable length, there is not a particular number of records that can be stored in a leaf node. Rather, the insertion operation would place a new record into a given leaf node when there are enough free bytes in that node to store it, and otherwise split the node so that the resulting two nodes are as close as possible to being balanced (and thus roughly half full).

Along with each seminar record stored in the leaf node, you should store the key value for that seminar, and a length value (in bytes) for the seminar record. The records stored in a leaf node should be in ascending order by key value (record ID). They should simply be arranged, in order, adjacent to each other in the node. Search within a leaf node will be done by sequentially moving through the records stored on that leaf node. When inserting a new record into the node, you will simply shift to the right any necessary records in order to make room.

When your program has completed reading all commands from the command file, before it terminates, it should flush to disk any dirty buffers and then close the database file. This will allow you and the GTAs to examine the database file to help with debugging and grading. Your program should also keep counters for the number of times you read a block from disk, and write a block to disk. These counts should be output at the end of your program.