Database Systems

Instructor: Hao-Hua Chu
Fall Semester, 2004

Assignment 4: Heap File Page Structure

Deadline: 17:00 , October 26 (Tuesday), 2004
This is a group assignment, and at most 2 people per group are allowed. (It means that one person group and two people group are allowed)

Cheating Policy: If you are caught cheating, your grade is 0.
Late policy: We will not accept any assignment submissions after deadline.
Demo: The demonstration will be announced on the course web page later.

Introduction

In this assignment, you will implement the page structure for the Heap File layer. You will be given some source code and some driver routines to test the code.

Preliminary Work

Begin by reading the description of Heap Files in section 9.5.1, and the description of page formats in section 9.6.2. A HeapFile is seen as a collection of records. Internally, records are stored on a collection of HFPage objects.

You will be implementing just the HFPage class, and not all of the HeapFile code. Read the description in the text of how variable length records can be stored on a slotted page, and follow this page organization.

Compiling Your Code and Running the Tests

Copy all the files from **web site** into your working directory. If you *make* the project, it will create an executable named *hfpage* . Right now, it does not work; you will need to fill in the bodies of the HFPage class methods. The methods are defined (empty) in file hfpage.C.

Sample output of a correct implementation is available in *sample_output*.

You need to write your code on a Linux operating system, because your code needs to be linked with several pre-compiled Linux libraries. We strongly recommend you to use a previous version GNU C compiler, not the version 3.0 or later. (If you could compile the program with these newer compilers, please tell us your solution.)

This assignment can't be compiled on some of our Linux workstation. After our testing, you could program your code on linux3, linux4, linux5, linux6, linux7, linux9, linux11, and linux12.

Design Overview and Implementation Details

Have a look at the file *hfpage.h* in **HFPage/include**. It contains the interfaces for the HFPage class. This class implements a "heap-file page" object. Note that the protected data members of the page are given to you. All you need to do is to implement the public member functions. You should put all your code into the file *hfpage.C*.

A note on the slot directory: In the description of the text, the slot directory is located at the end of the page, and grows toward the beginning of the page. This does mean, however, that you will need to write the code so the records themselves are placed beginning at the start of the page. Be very careful with your pointer arithmetic.

Also note that in order to add a record to a page, there has to be a room for the record itself in the data area, and also room for a new slot in the data area (unless there happens to be a pre-allocated slot that's empty).

Please follow the Minibase Error Protocol. An example file illustrating the use of the error protocol is available in **HFPage/src/ErrProc.sample**. It covers much of what you need to know about the protocol. You can look at new_error.h for more details. It is in **HFPage/include**.

The Methods to be Implemented

void HFPage::init(PageId pageNo):

This member function is used to initialize a new heap file page with page number pageNo. It should set the following data members to reasonable defaults: nextPage, PrevPage, slotCnt, curPage, freePtr, freeSpace. You will find the definitions of these data members in *hfpage.h*. The nextPage and prevPage data members are used for keeping track of pages in a HeapFile. A good default unknown value for a PageId is INVALID_PAGE, as defined in page.h. Note that freePtr is an offset into the data array, not a pointer.

PageId HFPage::getNextPage():
This member function should return the page id stored in the nextPage data member.

PageId HFPage::getPrevPage():
This member function should return the page id stored in the prevPage data member.

void HFPage::setNextPage(PageId pageNo):
This member function sets the nextPage data member.

void HFPage::setPrevPage(PageId pageNo):
This member function sets the prevPage data member.

Status HFPage::insertRecord(char* recPtr, int reclen, RID& rid):
This member function should add a new record to the page. It returns OK if everything went OK, and DONE if sufficient space does not exist on the page for the new record. If it returns OK, it should set rid to be the RID of the new record (otherwise it can leave rid untouched.) Please note in the parameter list **recPtr** is a char pointer and **RID&** denotes passed by reference. The Status enumerated type is defined in new_error.h if you're curious about it. You may want to look that file over and handle errors in a more informative manner than suggested here. The RID struct is defined to be:

```
Struct RID {

PageID pageNo;
int slotNo;

int operator == (const RID rid) const
{ return (pageNo == rid.pageNo) && (slotNo == rid.slotNo); };

int operator != (const RID rid) const
{ return (pageNo != rid.pageNo) || (slotNo != rid.slotNo); };

friend ostream& operator << (ostream& out, const struct RID rid); };
```

In C++, **struct** are aggregate data types built using elements of other types. The pageNo identifies a physical page number (something that the buffer manager and the DB layers understand) in the file. The slotNo specifies an entry in the slot array on the page.

Status HFPage::deleteRecord(const RID& rid):

This member function deletes the record with RID rid from the page and compact the hole created from the deleted record. Compacting the hole, in turn, requires that all the offsets (in the slot array) of all records after the hole be adjusted by the size of the hole, because you are moving these records to "fill" the hole. You should leave a "hole" in the slot array for the slot which pointed to the deleted record, if necessary, to make sure that the rids of the remaining records do not change. The slot array can be compacted only if the record corresponding to the last slot is being deleted. It returns OK if everything goes OK, or FAIL otherwise. (what could go wrong here?)

Status HFPage::exchangeRecord(const RID& firstrid, const RID& secondrid):

This member function exchanges two records with RID firstrid and secondrid from the page. While exchanging two records, you are suggested to exchange two contents with firstrid and secondrid and adjust the slot information if the lengths of these two records are different. It returns OK if everything goes OK, or FAIL otherwise.

Status HFPage::firstRecord(RID& firstRid):

This routine should set firstRid to be the rid of the "first" record on the page. The order in which you return records from a page is entirely up to you. If you find a first record, return OK, else return DONE.

Status HFPage::nextRecord(RID curRid, RID& nextRid):

Given a *valid* current RID, curRid, this member function stores the next RID on the page in the nextRid variable. Again, the order of your return records is up to you, but do make sure you return each record exactly once if someone calls nextRecord! Don't worry about changes to the page between successive calls (e.g. records inserted to or deleted from the page). If you find a next RID, return OK, else return DONE. In case of an error, return FAIL.

Status HFPage::getRecord(RID rid, char * recPtr, int& recLen):

Given a rid, this routine copies the associated record into the memory address *recPtr. You may assume that the memory pointed by *recPtr has been allocated by the caller. RecLen is set to the number of bytes that the record occupies. If all goes well, return OK, else return FAIL.

Status HFPage::returnRecord(RID rid, char*& recPtr, int& recLen):

This routine is very similar to HFPage::getRecord, except in this case you do not copy the record into a caller-provided pointer, but instead you set the caller's recPtr to point directly to the record on the page. Again, return either OK or FAIL.
DONE is a special code for non-errors that are nonetheless not "OK": it generally means "finished" or "not found." FAIL is for errors that happen outside the bounds of a subsystem.

int HFPage::available_space(void):

This routine should return the amount of space available for a new record that is left on the page. For instance, if all slots are full and there are 100 bytes of free space on the page, this method should return (100 - sizeof(slot_t)) bytes. This accounts for the fact that sizeof(slot_t) bytes must be reserved for a new slot and cannot be used by a new record.

bool HFPage::empty(void): Returns true if the page has no records in it, and false otherwise.

How to hand-in

Email the file “hfpage.C” to the yfhuang@csie.ntu.edu.tw before deadline and submit two student names and IDs in the file. The mail subject will be [DBMS]ASS4 – IDs. Please hand in a short report (1 page) to illustrate how your implementation works (each group hand in one report).