UNIVERSITY OF WISCONSIN–MADISON
COMPUTER SCIENCES DEPARTMENT
CS 564: DATABASE MANAGEMENT SYSTEMS

Assignment 2: HFPage
Due: Wednesday, October 9, 1996, 5 p.m.

Instructors: Jeff Naughton and Raghu Ramakrishnan

# 1 Introduction

In this assignment, you will implement the page structure for the Heap file layer. You will be given libraries for the lower layers (Buffer Manager and Disk Space Manager), code for the HeapFile layer, and some driver routines to test that code.

*You will carry out this assignment in teams of two, with the same partner as before. If you want to change partners, you must let us know first.*

*Please follow the Minibase error protocol, as in the previous assignment.*

# 2 Classes to Familiarize Yourself With First

There are two main classes with which you should familiarize yourself: HeapFile and HFPage. A HeapFile is seen as a collection of records. Internally, records are stored on a collection of HFPage objects. Your task will be to implement the member functions for the HFPage class.

## 2.1 Available Documentation

You should begin by reading the chapter on Disks and Files to get an overview of the Heapfile/HFPage layer and buffer management. In addition, HTML documentation is available for Minibase, at "http://www.cs.wisc.edu/coral/minibase/minibase.html."

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.

# 3 Compiling Your Code and Running the Tests

Copy all the file from *cs564-1/fall96/proj2/src/Makefile* to your own local working directory.

Do 'make setup'. This will copy the necessary .C and .h files to your directory.

Do 'make'. This will compile the .C files in your directory, and create a program called hfpage. Run this command, and you should see output like the contents of SampleRun.txt found in the proj2 directory.

When you ran make acording to the instructions above, you linked the code in your directory with a library called libwith.a. This library contains a compiled version of a solution to the assignment. Your challenge will be to make the code in your directory work when you replace libwith.a with a library called libwithout.a. To do this, you first have to change your Makefile:

Open Makefile in your favorite editor. On the line that begins LFLAGS=, change the flag -lwith to -lwithout, then save the file in your editor.

Run 'mv hfpage solution'. Now the program solution will give you the behavior you're shooting for, if you need to recheck it later.

Now, if you rerun make, you will see that the methods for the HFPage class are not defined. Your job will be to implement these methods.

*Makefile:* The Makefile is provided for you to compile your project. Study this carefully before you make changes. You will notice that in order to compile your code, you will have to link in the libraries, either libwith.a in the solution version or libwithout.a the one your working with. You do not need to copy this to your directory. The Makefile automatically compiles your files and links in this library. *You should use the gnu version of 'make', which is in `/usr/psup/bin`.*

# 4 Design Overview and Implementation Details

Have a look at the file hfpage.h. 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 should need to do is implement the public member functions. You should put all your code into the file hfpage.C.

A note on the slot directory: The first entry of the slot directory (i.e. HFPage::slot[0]) is right at the end of the data area of the page. If you want to add more slots to the page (and you will), you will have to put them in the data area of the page, which means two things:

1. New slots will be at negative offsets in the slot directory array, i.e. the next slot after 0 will be found at HFPage::slot[-1], the one after that at HFPage::slot[-2], etc.

2. In order to add a record to a page, there has to be 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 pre-allocated slot that's empty).

## 4.1 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. (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 -1. Note that freePtr is an *offset* into the data array, not a pointer.

**void HFPage::dumpPage():** A debugging utility you can use if you wish to print out the contents of a page. This member function is not required for the assignment, but it's probably a good idea to do it for your own debugging use.

**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():** This member function sets the nextPage data member.

**void HFPage::setPrevPage():** 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.)

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:

```
typedef 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;};
};
```

**Status HFPage::deleteRecord(const RID& rid):** This member function deletes the record with RID rid from the page. It returns OK if everything goes OK, or FAIL otherwise (what could go wrong here?) This routine must compact the remaining records on the page into a contiguous area of memory, and make sure that the slot directory remains correct. 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.

**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 current RID, curRid, this member function stores the next RID on the page in the nextRid variable. Again, the order your return records is up to you, but do make sure you return each record exactly once if someone continually calls nextRecord! If you find a next RID, return OK, else return DONE.

**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 to *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.

**Status HFPage::returnOffset(RID rid, int& offset):** Given a RID, this routine returns the offset into HFPage::data where the associated record is stored.

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

**void HFPage::compact_slot_dir():** Remember that the slot directory can only be compacted by removing a consecutive sequence of empty slots that are at the *very end* of the slot directory — otherwise, the rids of remaining records might be changed!

## 5 What to Turn In, and When

You should turn in copies of your code together with copies of the output produced by running the tests provided by the TAs. The assignment is due at 5PM on October 9, 1996. The solution will be made public after that time, and solutions turned in after that time will not receive any credit. So be sure to turn in whatever you have, even if it is not working fully, at that time.

*I emphasize that late submissions will not receive any credit. Computers—and life!—being what they are, expect everything to take longer than you expect, even taking this expectation into account. So start early, and plan on getting things done well before the due date. Nothing short of a nuclear explosion (in the CS building, not the South Pacific) constitutes a valid reason for an extension.*