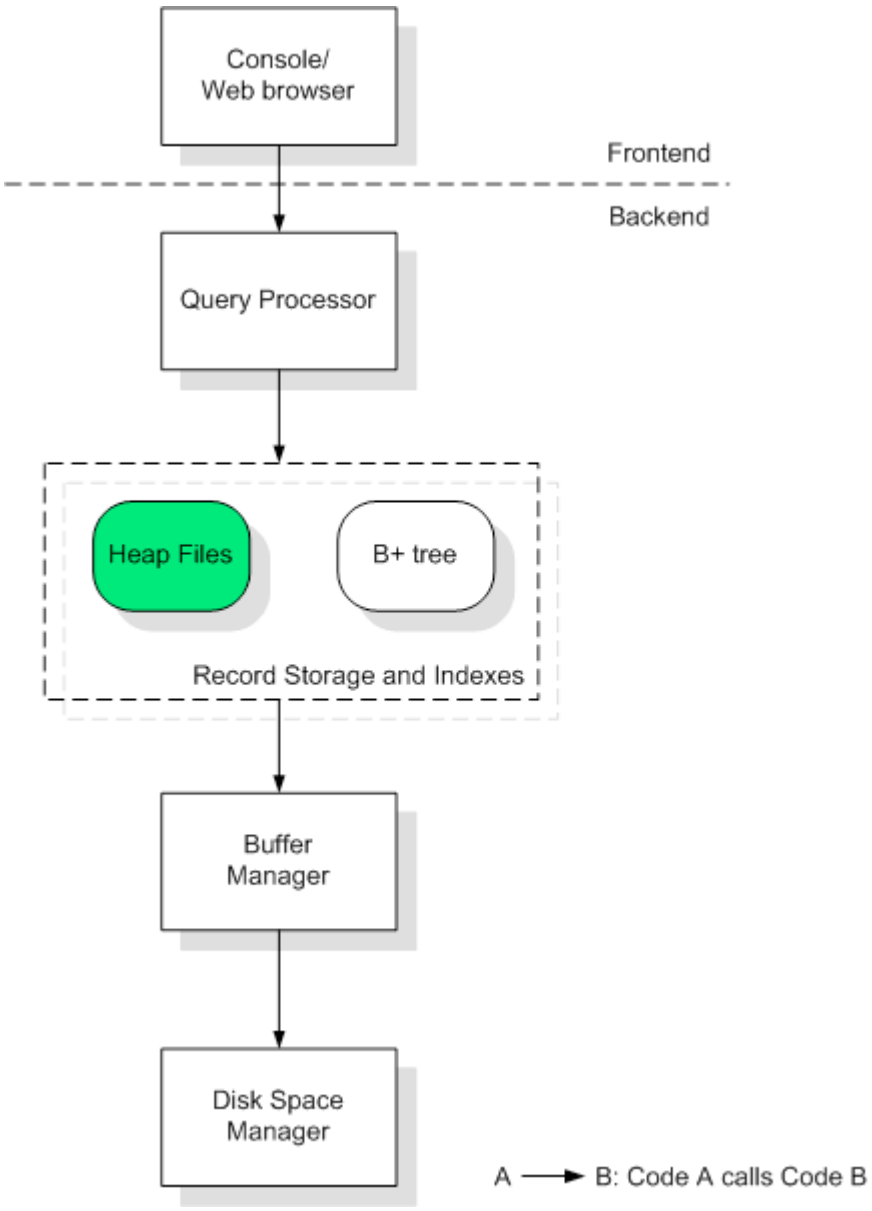
Project 1: HeapPage

Introduction



In this assignment, you will implement the page structure for the Heap File layer for storing variable length records (the component in green in the above figure). You will be given libraries for the lower layers (e.g. Buffer Manager and Disk Space Manager), and routines to test the code.

Preliminary Work

Begin by reading the description of heap files in Section 9.5, and the description of page formats in section 9.6 of the textbook (Gehrke book). A HeapFile is seen as a collection of records. Internally, records are stored on a collection of HeapPage objects.

You will be implementing just the HeapPage class, a sub-component of HeapFile, and not the entire HeapFile component. Read the description in the textbook of how variable length records can be stored on a slotted page, and follow this page organization. The file spacemgr/heappage.cpp contains skeleton implementations that you will complete in this practical.

Compiling Your Code and Running the Tests

Assuming your work directory is ~/, copy the file HeapPage.tgz with the skeleton code into your working directory and unzip it using the command tar -xvzf HeapPage.tgz. This will result in a folder ~/HeapPage/ contain the source code and CMake build files. Create a directory ~/HeapPage-bin/ , change to this directory and run cmake ~/HeapPage/ in order to create the makefiles for the project. Now run make to compile the source code. This should finish without errors, but potentially with a few warnings that should not bother you. (To re-compile the source code after having modified parts of the code, it suffices to run make; cmake does not need to be run again.) Run the resulting executable file ./minibase-heappage. It executes the test cases defined in spacemgr/heaptest.cpp which – not surprisingly – fail since a substantial part of the code in spacemgr/heappage.cpp is missing: Your task is to write additional code to pass these tests! The tests are: (1) test insertion and scan of fixed-size records, (2) test deletion of fixed-size records, (3) test updates of fixed-size records, (4) test the use of temporary heap files and variable-length records, and (5) test some error conditions.

Design Overview and Implementation Details

The header files (containing class and data structure definitions) of the entire minibase system can be found in ~/HeapPage/include/. Have a look at the file heappage.h. It contains the interfaces for the HeapPage 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 spacemgr/HeapPage.cpp. For this project, you should not modify any other file!

A note on the slot directory: In the description in the textbook, the slot directory is located at the end of the page, and grows toward the beginning. This has confused students in the past, since it means that negative offsets into the slot directory have to be used, so the current definition of HeapPage has the slot directory at the beginning of the page, after a few fixed member fields, and growing toward the end. This does mean, however, that you will need to write the code so the records themselves are placed beginning at the end 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 enough 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).

The Methods to be Implemented

void HeapPage::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**, **numOfSlots**, **pid**, **fillPtr**, **freeSpace**. You will find the definitions of these data members in *heappage.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 **fillPtr** is an offset into the data array, not a pointer.

PageID HeapPage::GetNextPage(): This member function should return the page id stored in the **nextPage** data member.

PageID HeapPage::GetPrevPage(): This member function should return the page id stored in the **prevPage** data member.

void HeapPage::SetNextPage(PageID pageNo): This member function sets the **nextPage** data member.

void HeapPage::SetPrevPage(PageID pageNo): This member function sets the **prevPage** data member.

Status HeapPage::InsertRecord(char* recPtr, int reclen, RecordID& RecordID): 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 **RecordID** to be the **RID** of the new record (otherwise it can leave **RecordID** untouched.) Please note in the parameter list **recPtr** is a char pointer and **RecordID&** 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 over that file and handle errors in a more informative manner than suggested here. A quick summary: **OK** means the operation is successful. **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.

The **RID** struct is defined to be:

```
Struct RecordID {

   PageID pageNo;
   int slotNo;
```

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

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

    friend ostream& operator << (ostream& out, const struct RecordID rid);

};
```

In C++, **structs** are aggregate data types built using elements of other types. The **pageNo** field 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 HeapPage::DeleteRecord(const RecordID& rid): This member function deletes the record with **RecordID** rid from the page, without compacting the created hole between records. You should also 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? e.g. Is the input **rid** valid? Is the corresponding slot valid?)

Status HeapPage::FirstRecord(RecordID& firstRid): This routine should set **firstRid** to be the **RecordID** of the first valid record on the page. If you find a first record, return **OK**, else return **DONE**.

Status HeapPage::NextRecord(RecordID curRid, RecordID& nextRid): Given a valid current **RecordID**, **curRid**, this member function stores the next valid **RecordID** on the page in the **nextRid** variable. If you find a next **RID**, return **OK**, else return **DONE**. In case of an error, return **FAIL**.

Status HeapPage::GetRecord(RecordID rid, char* recPtr, int& recLen): Given a **RecordID**, 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 HeapPage::ReturnRecord(RecordID rid, char*& recPtr, int& recLen): This routine is very similar to **HeapPage::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**.

int HeapPage::AvailableSpace(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)) bytes. This accounts for the fact that **sizeof(Slot)** bytes must be reserved for a new slot and cannot be used by a new record.

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

int HeapPage::GetNumOfRecords(): Returns the number of records currently stored in the page.

Optional Extra Task (for S+ mark)

Status HeapPage::DeleteRecord(const RecordID& rid): Read in the book about compacting heap pages. Improve the **DeleteRecord** function to compact the hole created between neighboring records. Compacting the hole requires that all the offsets (in the slot array) of all records between the hole and the free space offset (**fillPtr**) be adjusted by the size of the hole, because you are moving these records to "fill" the hole. You should still leave a "hole" in the slot array for the slot which pointed to the deleted record.

void HeapPage::CompactSlotDir(): Also implement the method **CompactSlotDir**. It compacts the slot directory (if possible) in order to maximise the available (contiguous) free space in the page. (NB: compacting the slot directory changes the slot IDs for different records, so any external references to records held in this page will be broken. We assume that this is OK.)

There is no test case defined to test this functionality. Come up with a strategy to test whether the **CompactSlotDir()** method works as expected. You do not have to implement this test but are expected to discuss your thoughts with the demonstrator.

What to Turn In

You are required to turn in *exactly the same set of files* as the set of files given to you at the beginning of the project: no more files, and no fewer files. These files should be zipped up into a file named *HeapPage-<FirstName-LastName>.zip* or .tgz (using the command tar -cvzf HeapPage-First-Last.tgz ~/HeapPage). These files should be organized in the same directory structure as well. Again, your task in this project is to only modify *spacemgr/heappage.cpp*. **Upload your solution (ie zip/tgz file) via the course website.** In addition to this submission, you will have to present your solution to the demonstrator in one of the sessions.

Deadline: end of your practical session in Week 4.

Make sure you start early! Good luck!

FAQ of Assignment 1

Q: In h*eappage.h*, the slot array has a length of only 1. How to use it? Shall we allocate additional memory if we would like to access **slot[1]**, **slot[2]** and so on?

A: You can assume the **slot[]** member field is an unbounded array. Therefore you can access **slot[1]**, **slot[2]** and so on directly, without having to dynamically allocate any additional memory for **slot[].** In C++ it is guaranteed that **slot[2]** will be located right after **slot[1]** in a consecutive region.

Q: To implement **FirstRecord** and **NextRecord**, how do we define the notion of "next"?

A: You get to define your own notion of "next". For instance, it can be defined by the ordering of records pointed to by the slot arrays. In that case, for example, the record pointed to by **slot[2]** is the "next" record of the record pointed to by **slot[1]**, assuming both slots are valid. However, if **slot[2]** is not valid, the next record of the one pointed to by **slot[1]** will be the one pointed to by some **slot[k]** where **k** is the smallest integer greater than 1 but less than number of slots such that **slot[k]** is valid.

Q: There are five test cases provided for **heappage/heapfile** in *heaptest.cpp*. Does the code need to pass the test cases in any ordering?

A: No. If your code passes the test cases in the ordering of case 1, 2, 3, 4, 5, that is good enough.