CSI2131—File Management, Winter 2003, University of Ottawa

Assignment 3: Extendible Hashing Primary Index

Due: Friday April 4 at 12:00 noon, Weight: 10%.

Introduction

The purpose of this assignment is to give you a practical experience with indexes, specifically indexes using the extendible hashing technique. For the sake of simplicity, you will implement extendible hashing insertions but not deletions. You will be creating a primary index (extendible hashing index) for a datafile and using it for searching on the datafile.

The problem

In this assignment you are going to implement an index organized as an **extendible hashing table**. The index will be created for an existing datafile. The datafile (students.txt) is in the same format as the master file for assignment#2, though the records will appear in a different order, not sorted by student number.
The directory for the extendible hashing file is to be kept in main memory, while the buckets containing the index records (keys and their reference fields) will be kept in a file (EHbuckets.txt) (Note that the directory is assumed to fit in main memory, but the datafile and index bucket file are assumed to be too large to fit entirely in main memory. At any point in the program, you may have only one datarecord and only a couple of buckets stored in main memory.
The index records (which will be part of the buckets) are in the following format:

- student number (key used for the index, primary key): 8 characters (7 digits plus space)
- datarecord address (RRN): integer represented in text mode using 5 characters

Your assignment consists of two parts:

Part I:
This is a program that, given the datafile, builds the extendible hashing index.
This index must be built through successive insertions of index records into an originally empty extendible hashing directory. the index records are inserted in the order they appear in the datafile. While building the extendible hashing index, the directory is kept as an array in main memory, while the index buckets are kept in a file (`EHbuckets.txt`). During this succession of insertions you must print in a log file (`log.txt`) the contents of the whole directory after every 5 insertions, so the marker can check that you are doing your insertions correctly. The directory must be print in a compact (text) form in one line, for example: `4:2,1,3,0`
After you finish creating the index, the directory must be written to a file (`EHdirectory.txt`) in order to be used later. In this `EHdirectory.txt` file you may store the name of the file holding the buckets, the depth of the directory and the contents of each cell of the directory; you may choose the format.

Part II:
In part II, your program will allow for searches on the datafile using the extendible hashing index. The first thing your program must do is to open the directory (loading the directory file into an array and opening the bucket file) and the datafile.
Then you continuously prompt the user for a student number and search for the student number using the extendible hashing index; the process ends when the user gives student number -1. More precisely, in this search, given a student number, you must use the extendible hashing directory in order to discover the datarecord address (RRN) in the datafile and then via the RRN, show the datarecord to the user. You must handle the case in which the user enters a student number that is not in the database.

Hash Function:
You will use the following hash function: $h(s) = s^2 \bmod 19937$, where s is the student number.
An implementation which already gives the right directory index by grabbing from this hash function the proper number of bits in reverse order is posted on the file `hashing.txt`. You should use this function in your program.

Your programs

Both programs will use the following common classes, which you must create:

- Datafile class: used to manipulate the datafile, providing methods such as one that allows you to read a record by RRN, and any other method you may need to manipulate the the datafile.
- Directory class (friend of Bucket): used to hold the array for the directory of the extendible hashing index and provide public methods such as: insertion to the index and search in the index.
- Bucket class (friend of Directory): used to hold one bucket in main memory, providing methods to Directory class to read/write a bucket from/to a file, split a bucket in main memory creating another bucket in main memory, etc.

Advice: create and test each class separately and incrementally build methods and functionality to the classes.
Note: all files created should be TEXT files, so that the numbers are readable by the marker.

Part I: (70 marks)
The main program for part I will consist of the main function and perhaps some auxiliary functions to accomplish the tasks described for part I.

```
classes that must be created: Datafile, Directory, Bucket
program files:      mainI.cpp, Datafile.*, Directory.*, Bucket.*
                    (.* denotes both .h and .cpp files)
input files:        students.txt
output files:       EHdirectory.txt, EHbuckets.txt, log.txt
command line use:   mainI.exe students.txt EHdirectory.txt EHbuckets.txt log.txt
```

Part II: (30 marks)
The main program for part II will consist of the main function and perhaps some auxiliary functions to accomplish the tasks described for part II.

```
program files:      mainII.cpp, Datafile.*, Directory.*, Bucket.*
                    (.* denotes both .h and .cpp files)
input files:        EHdirectory.txt, EHbuckets.txt, students.txt
output files:       none (user uses console to see search results)
command line use:   mainI.exe students.txt EHdirectory.txt
```

How to hand in your assignment

Submit a zipped directory, as per the standards in the previous assignments, containing the following files:

- `mainI.cpp`, `mainII.cpp`, `Datafile.h`, `Datafile.cpp`, `Directory.h`, `Directory.cpp`, `Bucket.h`, `Bucket.cpp`.
- `report.txt` (maximum 1 page): a report (textfile written using, say, notepad) containing some high level description of the main components of your program. Include a section on "Functionality and known bugs" explaining all the parts of your program that are working. You must say whether you have a full implementation for part I only or for both part I and II. You should include any known bugs.

Your program must meet the specifications given in this handout and in any further clarification given later on the web page.

Hints to help you organize your classes

You may use, modify and add to the following suggested classes (Directory and Bucket). You may consult the algorithms and implementations from the textbook. You are free to design your class Datafile the way you wish. If you need extra classes, please place it in one of the files indicated (in C++ it is possible to place it together with another class, for instance).

```
// Directory.h  (You may modify, remove and/or add members)
// This class is a simplification of the one given in the
// textbook "File Structures" by Folk, Zoellick and Riccardi.
// Directory class contains directory and file information
// for the extendible hashing index
#ifndef DIRECTORY_H
#define DIRECTORY_H
#include "Bucket.h"

class Directory
{
public:

Directory(int maxBucketKeys);
~Directory();

int Create(char *dirFileName, char *bucketFileName);
   // create an empty extendible hashing index

int Open(char *dirFileName);
   // open an existing extendible hashing index

int Close();
   // close an existing extendible hashing index

int Insert (long key, int datafileRecAddr);
   // insert the pair (key,datafileRecAddr) into the index
   // may use "char *key" or "string & key" instead of "long key"

int Search (long key); // may use "char *key" or "string & key"
   // return the datafileRecAddr for the given key

void Print (ostream & outfile);
   // print the directory as a compact line to outfile; file is already open

protected:

int Depth;
int NumCells; // = 2^Depth
int * BucketAdrr;
    // array of bucket addresses to be allocated
fstream DirectoryFile; // logical name for directory file
fstream BucketFile; // logical name for bucket file

Bucket  *currentBucket; // pointer to current bucket
```

```
int DoubleSize(); // doubles the size of the directory

int InsertBucket (int bucketAddr, int first, int last);
   // change directory cells to refer to this bucket
int Find (long key); // return bucket address for key

int StoreBucket (Bucket & bucket);
    // update or append bucket in file
int LoadBucket (Bucket & bucket, int bucketAddr);
    // load bucket from file
int MaxBucketKeys;
    // for this assignment this will be =5 (initialize in constructor)

// the hash function may be included in this class, since it
// will be needed for Find, Search, Insert, etc.
friend class Bucket; // Bucket has access to its protected members
};
#endif
```

```
// Bucket.h   (You may modify, remove and/or add members)
// This class is a simplification of the one given in the
// textbook "File Structures" by Folk, Zoellick and Riccardi.
// Bucket class holds a bucket in main memory
#ifndef BUCKET_H
#define BUCKET_H
#include "Directory.h"

class Bucket
{
protected:
// there are no public members,
// access to Bucket members is only through class Directory

Bucket ( Directory *dir, int maxKeys);

int Insert (long key, int dataRecAddr); // may use "char *key" or "string & key"

void Split(); // split the bucket and redistribute keys

int NewRange (int & newStart, int & newEnd);
    // calculate the range in directory of new (split) bucket

int Redistribute (Bucket & newBucket);
    // redistribute keys between this and new bucket

int Depth; // depth of the bucket
// number of bits used in common by keys of this bucket

Directory *Dir; // pointer to the directory that contains the bucket

int BucketAddr; //address of this bucket in file

// include here variable to hold the various (maxKeys) index records in this bucket
friend class Directory; // Directory has access to its protected members
};
#endif
```