CSE 102 | **Programming Assignment**
W.A. Burkhard | **Due: February 21, 1997**

Linear Hashing and Indicies

You are to create an index class implemented with linear hashing in this exercise. We will be using a dictionary, found in `/usr/dict/words`, as the data. To expedite matters, we will provide the class `DataFile` which accesses data identifier objects. This class provides the data in pseudo-random order; each objext returned by the class contains a key value which is a string containing an word and a record identifier which is an offset within `/usr/dict/words`.

```
class Rec {
public:
    long int position ;
    char name [20] ;
}
```

The practical, real world senario we are trying to capture here is that there is real data that we can only read, and the index, we are building for our use, references the real data. The data identifier objects contain the data key (which in this exercise *is* the data) as well as the data reference. There are applications in which we would want to modify the real data; extending the results of this exercise to include data modification/addition would be only a small step.

We will provide the `public` interface for the `Index` class; it is to be used to create and access your linear hashing index data. The private part of this class is just a suggestion.

```
#include <stdio.h>
#include "Boolean.h"

class Index : {
        Bucket bucket ;         // private page copy
                // will need two for differing Primary and Overflow pages.
        unsigned split() ;
        double alphaMax , suf ;
        unsigned number , overflows ;
        unsigned level , next ;
        FILE * primary , * overflow ;  // stdio library
public:
        Index ( char * pathname , double aMax ) ;
        ~Index ( ) ;
        bool insert ( const Rec & rec ) ;
        bool fetch ( Rec & rec ) const ;
        void display ( ostream & os ) ;
        void measure ( ostream & os ) ;
} ;
```

The `Index` constructor opens an existing index or creates a new one if the file with pathname `name` does not exist. The `aMax` parameter is the storage utilization factor for the to-be-created index; if the index exists, this parameter is ignored otherwise it is stored in the structure. The destructor `~Index` closes files and releases all main memory structures used by the index. The `insert` method includes a data identifier object in the linear hashing structure; it returns true if successful and false otherwise. The `fetch` method attempts to access a data identifier object possessing the `name` identical to the string specified by `rec.name`. It returns true, with the data identifier object completed, if the record is present; otherwise, it returns false. The `display` method presents the records of each chain. The `measure` method presents the average successful and unsuccessful search lengths in terms of page accesses.

The second class `DataFile` accesses a file of pseudo-randomly ordered data identifier objects. This class is

```
#include <stdio.h>
#include "Boolean.h"

class DataFile {
    FILE * fptr ;
public:
    DataFile ( char * pathname ) ;
    void Reset ( void ) ;
    bool NextItem ( Rec & bool ) ;
} ;
```

The constructor `DataFile` creates an object that allows access to the data via the data identifier objects; its one string parameter is the pathname to the data identifier objects file. The `Reset` method moves the DataFile to the first identifier object in the file. The `NextItem` method sets its parameter to the next object in the data identifier object file. It returns true if another item is present and false otherwise. The data identifier objects file `data22` contains 20,000 data identifier object records.

Stream input/output library

Here are several commands that will be useful for handling files. You can use either the C++ streams library or the C stdio library for the exercise. I will provide a separate handout describing some of the stdio library functions.

```
fstream primary ;                    // creates a (null) stream named primary.

    primary.open( "path name" , ios::in|ios::out|ios::nocreate ) ;
    // associates file "path name" with primary stream.
    // the command succeeds only if the file does not exist.
    primary.fail()              // indicates whether an operation fails.
    primary.close( ) ;          // terminates association between file and stream.
                                // random access within a file.
    primary.seekp( loc ) ;      // next operation at position loc
    primary.tell( ) ;           // returns the current position within file.
                                // random access within a file.
    primary.flush( ) ;          // buffered data is moved immediately to destination.
    primary.read( bucket , sizeof(Bucket) ) ;
                                // reads sizeof(Bucket) bytes from primary into bucket.
    primary.write( bucket , sizeof(Bucket) ) ;
                                // writes sizeof(Bucket) bytes to primary from bucket.
```

A more precise syntax and semantics for these operations is given within iostream and fstream manual pages. The fstream.h and iostream.h include files also contain useful information. Unfortunately generally available discussion of these operations is non-existent to spare at best. We will provide some examples of these functions for your perusal as well.

Creating and Using the Index

The file `main1.c` contains a program skeleton to make an index from the data identifier records. The file `main2.c` contains a more conventional driver program that will access the data records within `/usr/dict/words`.

Hints

Your first task is to implement a linear hashing index maintenance routine. I suggest that you first build the `Index insert` and `display` methods.

There is the question of sizes for the logical blocks: I suggest that you parameterize this (defines will work well here) so you can get things running with fairly small block sizes. You will be able to display small blocks easily. I suggest something like ten data identifier objects per primary block and five such objects in the overflow blocks.

1) Build the Index in small incremental steps. Get each working before going on to the next. Begin with the Index constructor and destructor; make sure the constructor can read file headers and properly initialize Index's state variables. Similarly, the destructor must be able to write the file headers as well. Your `Index::write` function should verify these work correctly. Then begin building a single chain without using the split operation. You will need the write function to verify that your "single" chain version properly contains the desired pairs. Finally, build and test your split function.

2) The management of freed overflow pages is necessary. You can choose several ways to do this. One approach is to keep a list of freed pages linked together on disk. Another approach is to keep an array of available freed pages in memory and then write this to disk, say at the end of the file, within the Index destructor.

3) You will need to include data with both the overflow and primary files regarding its state. The state contains sufficient information allowing the file to be opened and used at future. One very common approach to this is to include a header record at the beginning of each file.

Extra Credit

a. Use a Bloom filter to improve the search performance within Pages. Choose a signature width sufficiently large so the "false drop" rate is no greater than 0.1.

b. Implement the index with B+trees.

c. Enhance the index class to accommodate range query methods such as this.

```
rangeQuery { Rec & low , Rec & high , List & list } ;
```

The `List` is to return all the data references for records satisfying the rangeQuery.

Here is `main2.c` note, it uses only the standard input-output library.

```
/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 *
 *  main    demonstrates the presentation of the dictionary data
 *
 *
 * = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "DataFile.h"

int
main( int argc , char ** argv )
{
    FILE * jjj ;
    int i ;
    char array[24] ;
    Rec rec ;

    if ( argc <= 1 ) {
        fprintf ( stderr , "Usage: %s <data file>\n" , argv[0] ) ;
        exit ( 1 ) ;
    }
    DataFile * a = new DataFile(argv[1]) ;
    jjj = fopen( "/usr/dict/words" ,"r" ) ;

    i = 0 ;
    do {
        a->NextItem(rec) ;
        fseek(jjj, rec.position, 0 ) ;
        fgets(array,24,jjj) ;
        printf ( "%d\t%s\t%s" , rec.position , rec.array , array ) ;
        i++ ;
    } while ( i <= 20000 ) ;
    fclose(jjj) ;
}
```

Here is a skeleton `main1.c` program. The features regarding the interaction with the main program are presented.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

class Rec {
public:
    int position ;
    char array[20] ;
} ;

int
main( int argc , char ** argv )
{
    FILE * jjj , * kkk ;
    Index * p ;
    Rec rec ;
    DataFile * q ;
    double alpha ;

/* portion of the creation process */

    sscanf ( argv[3] , "%f" , &alpha ) ;
                                /* convert the string to a double value. */
    p = new Index ( argv[2] , alpha ) ;
    q = new DataFile ( argv[1] ) ;


/* portion of the insertion process*/

    q-> NextItem ( rec ) ;      /* the next item */
    p-> insert ( rec ) ;        /* the insertion method */

/* portion of the fetch process */

    char data[25] ;
    scanf ( "%s" , &data ) ;
    strcpy ( rec.name , data ) ;
    p->fetch ( rec ) ;
                                          /* now rec.position contains the
                                             data reference value          */
}
```