CSE 101
Introduction to Data Structures and Algorithms
Programming Assignment 5

In this assignment you will create a Dictionary ADT, whose keys are strings and whose values are ints, then use it to determine word frequencies in a text file. Begin by reading the handout DictionaryADT.pdf posted on the class website, to the end of the section on hash tables. Also read the relevant sections in the text CLRS (4th edition, sections 11.1-11.4). Your Dictionary will be based on a hash table in which collisions are resolved by open addressing. It will, as closely as is practicable, emulate the design of the dictionary data type in Python.

The top-level client in this project will be named WordFrequency.c. Any text file containing words and punctuation characters will be a valid input file for this client (as was the case for pa1). It will read each line of the input file, parse the words on that line by discarding delimiters (defined as in pa1), then convert each word to all lower case characters (unlike pa1 but like the example FileIO.c). WordFrequency.c will then look up the word in the Dictionary. If the word is not in the Dictionary, it will insert it with value 1. If the word is in the Dictionary, it will increment its associated value by 1. After this, WordFrequency.c will print the Dictionary to the output file in a format that reflects the insertion order of the words. Specifically, the output file will begin with a blank line. If there are n unique words in the input file (not distinguishing upper from lower case), then each of the the next n lines of output will cantain a (lower case) word, followed by a space, then a colon, then a space, then the frequency of the word. The output file will conclude with a single blank line. Words will be printed in the order they were inserted into the Dictionary, i.e. line 2 of the output will contain the first word inserted, and line $n + 1$ will contain the last word inserted.

Example 1

Input:
```
One, two + :)
two::One,,,three
three/two
Four
one, five=three
```

Output:
```

one : 3
two : 3
three : 3
four : 1
five : 1
```

A number of matched pairs of input-output files are included in the examples section of the webpage, along with a random input file generator.

Dictionary ADT Specifications

One could easily write the above client in Python by using the *dictionary* data type, then just iterate over the dictionary in its default order, while printing to the output file. As of Python 3.6, the dictionary type preserves the insertion order of its keys, even if lookups and deletions are mixed with insertions. Python does this by using *two* arrays to store the data associated with a dictionary object. One array is an open address table, as described in the above handout. It is maintained as a sparse array with a lot of open space, making the dictionary operations efficient. Its contents are not key-value pairs however. Instead, the table stores indices to a data array, which itself stores the key-value pairs in their order of insertion into the dictionary. This is a dense array, maintained in compact form to save space. Both arrays are independently expanded as needed to make room for the insertion of additional pairs into the dictionary.

Your Dictionary ADT for this project will simulate the design of the Python dictionary type described above. Note though that the similarity will be internal only, since the names of your Dictionary operations

will be very different from those in Python. Your Dictionary ADT will export the following types and operations.

```
// Exported types -------------------------------------------------------------
typedef const char* keyType;  // Points to const data
typedef int valType;

// Dictionary reference type
typedef struct DictionaryObj* Dictionary;


// Constructors-Destructors ---------------------------------------------------

// newDictionary()
// Constructs a new empty Dictionary.
Dictionary newDictionary(void);

// freeDictionary()
// Frees heap memory associated with *pD, and sets *pD to NULL.
void freeDictionary(Dictionary* pD);


// Access functions -----------------------------------------------------------

// size()
// Returns the number of key-value pairs in D.
int size(Dictionary D);

// contains()
// Returns true if D contains a pair with key k, returns false otherwise.
bool contains(Dictionary D, keyType k);

// getValue()
// Returns the value associated with key k.
// pre: contains(D, k)
valType getValue(Dictionary D, keyType k);


// Manipulation procedures ----------------------------------------------------

// clear()
// Sets D to the empty state.
void clear(Dictionary D);

// setValue()
// If D contains a pair with key==k, overwrites the associated value with v,
// otherwise inserts the new pair (k, v) into D.
void setValue(Dictionary D, keyType k, valType v);

// removeKey()
// Deletes the pair with key==k from D.
// pre: contains(D, k)
void removeKey(Dictionary D, keyType k);


// Other operations -----------------------------------------------------------

// copy()
// Returns a new Dictionary containing the same key-value pairs as D.
Dictionary copy(Dictionary D);
```

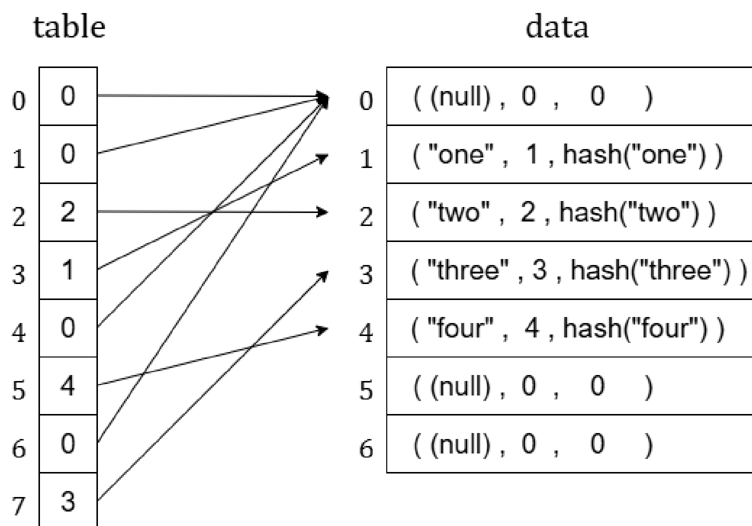
```
// equals()
// Returns true if A and B contain the same key-value pairs, and returns false
// otherwise.
bool equals(Dictionary A, Dictionary B);

// printDictionary()
// Prints a string representation of Dictionary D to the FILE pointer out. Each
// key-value pair is printed on its own line in the form "key : value". Pairs
// will appear in the order in which they were inserted into the Dictionary.
void printDictionary(FILE* out, Dictionary D);

// printDiagnostic()
// Prints a string representation of the internal state of Dictionary D to FILE
// out.  First D->data is printed with accompanying parameters, then D->table
// is printed with its accompanying parameters.
void printDiagnostic(FILE* out, Dictionary D);
```

As discussed above, there are two arrays to keep track of in this project. In order to reduce confusion, we will call the first (sparse) array **table**, since it is really a hash table in the usual sense, and refer to its indices as **slots**, conforming to common usage. We call the second (dense) array **data**, since it stores the actual Dictionary data, and refer to its indices as just **indices**.

Note though that the data array does not actually store pairs. Instead, it stores triples of the form $(\text{key}, \text{value}, \text{code})$, where $\text{code} = \text{hash}(\text{key})$ is the hash code corresponding to the string key. Observe that the hash function $h(k, i)$ for open addressing depends on the table length (see handout). Therefore every table expansion changes the hash function, and hence it entails a complete rearrangement of the table contents. For the sake of efficiency, the hash code is computed just once when the pair (key, value) is first inserted into the Dictionary, then saved, so it doesn't need to be recalculated on each table expansion.

Example 2

Let $\text{length}(table) = 8$, $\text{length}(data) = 7$ and suppose the pairs $(\text{"one"}, 1), (\text{"two"}, 2), (\text{"three"}, 3)$ and $(\text{"four"}, 4)$ are inserted, in order, into the Dictionary. The resulting table and data arrays are



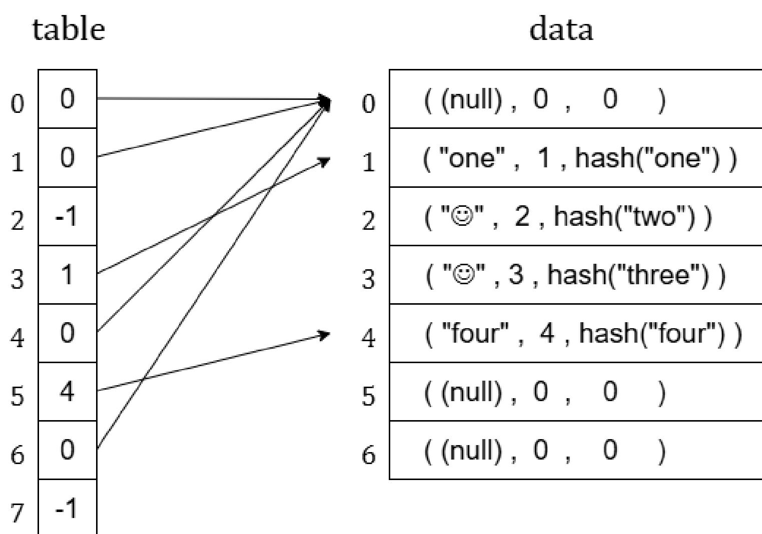
To see how this configuration comes about, one must calculate the hash codes for the keys "one", "two", "three", and "four", then calculate the probe sequence for each code. These calculations are performed by the example HashTest.c posted on the class webpage under /Examples/C/HashCode. Observe that the special table value representing EMPTY is 0, and so empty slots in table point to data[0]. Therefore 0 is

considered a dummy index to data, and the first pair inserted into the Dictionary will be placed in data[1]. Subsequent pairs will be added to data in order, so that the data array maintains the Dictionary's insertion order. The result of function printDictionary() on the above example is:

one : 1
two : 2
three : 3
four : 4

Both the table and data arrays need to have their own special values for EMPTY and DELETED. For the table array, 0 represents EMPTY and -1 represents DELETED. For the data array, the value NULL represents EMPTY (printed above as `(null)`), and a string consisting of a single non-printable character (ascii code 255) represents DELETED. When a value is deleted from the data array, the key field in the triple is overwritten with this special value, while the value and code fields are unchanged. Thus, function printDictionary() can just iterate over the data array, starting at index 1, skip any DELETED elements, and halt when EMPTY is encountered.

Continuing with the above example, after deleting the keys "two" and "three" the table and data arrays have become



and the output of printDictionary() is:

one : 1
four : 4

These and other Dictionary manipulations are illustrated in the example DictionaryClient.c, posted under /Examples/pa5/ on the class webpage. The file DictionaryClientOut, included with the examples, contains the output of DictionaryClient.c and should be studied carefully.

You will implement the following policies as to when and how the table array should be expanded. Function newDictionary() will allocate table to be an array of length 8 whose contents are of type `int64_t` (found in the library `stdint.h`). DictionaryObj should contain a field of type `size_t` to keep track of the table size, as well as a field of type `double` to keep track of the load factor. The threshold for table expansion

will be when the load factor becomes larger than 0.67. Each expansion will increase the size of the table by a factor of 4, until its size is above 50,000. At that point, remaining expansions will increase its size by a factor of 2. You should initialize the following constants in Dictionary.c to facilitate these rules.

```
// Constants for table array (sparse, indices to data array)
const size_t TableInitialSize = 8;
const double TableLoadFactorThreshold = 0.67;
const int TableExpandFactor1 = 4;
const int TableExpandFactor2 = 2;
const int TableSizeThreshold = 50000;
const int TableEmpty = 0;
const int TableDeleted = -1;
```

In addition to expansion, the data array will also be compacted at certain times, to eliminate the "holes" resulting from deletions. You will implement the following policies as to when and how the data array should be expanded and compacted. First, your Dictionary.c must define a private type representing a triple (key, value, code). Here we will refer to that type as `element`, though you may give it any name you wish. The constructor newDictionary() will initialize data to be an array of length 1 whose contents are of type `element`. DictionaryObj should contain a field of type `uint64_t` to keep track of the next index at which an insertion into data will take place, which should be initialized to 1 by the constructor. Every time an insertion takes place, if the next-insertion field is equal to the length of the array (as it will be upon the first insertion), the data array will be expanded by a factor of 1.5 (and rounded up to the next integer). DictionaryObj should contain fields of type `uint64_t` to keep track of the number of pairs in the dictionary, the number of deletions that have taken place (i.e. the number of "holes" in the data array), and the size of the data array itself. Also, a field of type `double` should be included to keep track of the density of the data array, defined as

$$\text{density of data array} = \frac{\#\text{pairs}}{\#\text{pairs} + \#\text{deletions}}$$

This density should be initialized to 1, and when it falls below the threshold 0.8, the data array becomes eligible for compactification. However, since compacting the data array means eliminating the "holes", and therefore changing the positions of its elements, this operation entails a re-hashing of the entire table array. Therefore, compactification will wait until the next table expansion. Thus the table expansion operation must first check the data array and compactify if it is eligible, then expand and re-hash the table array. Initialize the following constants in Dictionary.c to facilitate these rules for the data array.

```
// Constants for data array (dense, dictionary data)
const size_t DataInitialSize = 1;
const double DataExpandFactor = 1.5;
const double DataDensityThreshold = 0.8;
const keyType DataEmpty = NULL;
const char dummy[] = {255, 0};       // array is const
const keyType DataDeleted = dummy;   // pointer is const
```

Recall that table expansion is accompanied by a complete rearrangement of the table contents, while data expansion leaves all elements in place. Thus the data expansion operation can be efficiently implemented using function `realloc()` from the `stdlib.h` library, which automatically copies data to the newly allocated array, and deallocates the old array (see C/FileIO/FileIO.c and C/Stack/Array/Stack.c). The table expansion operation however must be performed manually by copying all values into the new array in their proper locations (see C/Queue/Array/Queue.c under /Examples on the class webpage.)

Note the required function `printDiagnostic()` is not a true ADT operation, but only exists for the sake of testing and evaluating your Dictionary ADT by giving a direct view into its internal state. It should print the contents of the data array followed by the parameters numPairs, numDeleted, dataSize, dataNextIndex and dataDensity. It should then print a blank line, followed by the contents of the table array, then the parameters tableSize and tableLoadFactor. The exact format of this output can be inferred by studying the program DictionaryClient.c and its output DictionClientOut.

The following helper functions will aid in implementing the above rules and policies. The first two are *required*, and should be copied verbatim into Dictionary.c. First is $\text{hash}(k)$, which converts a string into a hash code. The return type `codeType` is an alias for `uint64_t`, and should be defined in Dictionary.c. Second is $\text{probe}(\text{code}, \text{tbl_size}, i)$, which returns the ith term in the probe sequence for code. Function $\text{probe}()$ is essentially the hash function for open addressing $h(k, i)$ defined in Example 2 from the Dictionary ADT handout.

```
// Required Helper functions ---------------------------------------------------

// hash()
// Returns the hash code for key k.
codeType hash(keyType k) {
   codeType result = 0x2096DB9D4D43C94E;
   // binary: 0010000010010110110110111001110101001101010000111100100101001110
   codeType nextChar = *k;               // get first char and convert to codeType
   while(nextChar) {                     // while not at end of k
      result ^= nextChar;                //    result = result (exor) nextChar
      result = (result<<5)|(result>>59); //    left rotate result by 5 bits
      nextChar = *(++k);                 //    get next char and convert to codeType
   }
   return result;
}

// probe()
// Returns the ith term in the probe sequence for code.
size_t probe(codeType code, size_t tbl_size, size_t i){
   codeType h1 = code & (tbl_size-1);             // code % tbl_size
   codeType h2 = 2*(code & (tbl_size/2 - 1)) + 1;  // 2*(code % tbl_size/2) + 1
   return (h1 + i*h2) & (tbl_size-1);             // (h1 + i*h2) % tbl_size
}



// Optional Helper functions ---------------------------------------------------

// findSlot()
// Steps through the probe sequence for code=hash(k). If the Dictionary contains
// key k, returns the slot in D->table containing the index of key k in D->data.
// Otherwise, returns the first slot in D->table that contains either the special
// value TableEmpty or the special value TableDeleted.
uint64_t findSlot(Dictionary D, keyType k, codeType code);

// insertIndex()
// Steps through the probe sequence for code and inserts idx into the first
// slot in array T (of length m) at which either the special value TableEmpty
// or the special value TableDeleted are found.
void insertIndex(int64_t* T, size_t m, size_t idx, codeType code);
```

```
// expandTable()
// Expands length of array D->table. If tableSize<TableSizeThreshold, increases
// size by TableExpandFactor1, otherwise increases size by TableExpandFactor2.
// Re-inserts indices of D->data into D->table.
void expandTable(Dictionary D);

// expandData()
// Expands length of array D->data, leaving contents unchanged.
void expandData(Dictionary D);

// compactData()
// Removes the deleted elements from array D->data, making its entries contiguious,
// and sets D->dataDensity equal to 1.
void compactData(Dictionary D);
```

What to Turn In

The program called DictionaryClient.c, included in Examples/pa5, should be considered a weak test of your Dictionary ADT. It illustrates, among other things, the computations that go into Example 2 above. As usual you should submit a program called DictionaryTest.c that contains your own tests of your Dictionary ADT. Also included are 5 matched pairs of input-output files for the client WordFrequency.c, along with a random input file generator written in Python (basically identical to that for pa1). Submit a README.md file and a Makefile that creates the executable WordFrequency. A Makefile is included in Examples/pa5 which you may alter. Thus six files will be submitted in all.

WordFrequency.c	written by you
Dictionary.c	" " "
Dictionary.h	" " "
DictionaryTest.c	" " "
README.md	" " "
Makefile	provided, you may change

Push these files to the pa5 folder in your git repository by the due date. As always, start early and ask questions.