CSC C78F 1999
Assignment 3

due: Friday, November 12, 1999 at 4:10pm in Nick's office.

1. Let the procedure RB-INSERT-FIXUP(T,x) consist of the lines 3 through 18 of RB-INSERT(T,x) given on page 268. Then RB-INSERT(T,x) could be written as follows:

   ```
   TREE-INSERT(T,x)
   color[x] := RED
   RB-INSERT-FIXUP(T,x)
   ```

   Consider the following credit invariant for a red-black tree:
   Every black node with two black children has \$1 and
   every black node with two red children has \$2.

   Using this credit invariant, prove that the amortized costs of RB-INSERT-FIXUP and RB-DELETE-FIXUP are $O(1)$. Be sure that your solution addresses the following:

   (a) How many comparisons and assignments does \$1 pay for?

   (b) How much is charged for performing RB-INSERT-FIXUP?

   (c) How much is charged for performing RB-DELETE-FIXUP?

   (d) Explain why there is always sufficient money available to pay for the actual cost of these operations and ensure that the credit invariants remain true.

   BONUS: Explain how you would have derived the given credit invariant.

2. You are to implement a dictionary using perfect hashing with a two-level hash table, as described in class. Recall that in a two-level hash table, the top-level consists of a single hash table, T, and the lower level consists of n hash tables, $T_0, T_1, ..., T_{n-1}$, one for each position in table T. Each table has its own hash function. We use h to denote the hash function for table T, and h_k to denote the hash function for table T_k.

   Your program should have the following parameters:

   - p, the number of elements in the universe of keys, U.
     We assume that p is a prime number.
   - n, the number of keys to be stored in the dictionary, and the number of positions in the top-level table, T. We assume that n is much less than p.

You should represent the two-level hash table as a single array, A, as described in class. You may assume that the total space used by all the lower-level tables is at most $8n$. The first step is to declare an array, A, that is large enough.

The second step is to find a good hash function, h, for the top-level table. Recall that h is good if the lower-level tables don't take up too much space. In this case, the sum of their sizes must be no more than $8n$. To find a good h, you repeatedly pick hash functions at random until a good one is found. This is done with the algorithm below. In this algorithm, $h(x) = (a\,x \mod p) \mod n$ where a is an integer such that $1 \leq a \leq p-1$. To choose h at random, the algorithm simply chooses a value for a at random.

```
ALGORITHM 1:
  Choose a value for a at random.
  Repeat until h is good:
    Using h, hash all the keys in set S to table T,
      counting how many keys hash to each bucket.
    Let bk be the total count for bucket k.
    Sum the values of bk*bk for all k.
    If the sum is at most 4n
      then h is good, so store the value of a;
      else h is bad, so choose another value for a at random.
```

Having found a good hash function, h, we also know how many keys, b_k, hash to the k^{th} bucket. Thus, we know how much space each lower-level table requires. Specifically, table T_k should be of size $2b_k^2$.

The third step is to divide up array A into the various pieces of the two-level hash table. For instance, you must determine the starting position of each lower-level table, T_k, in the array.

The fourth step is to find a perfect hash function, h_k, for each of the lower-level tables, T_k. To do this, let S_k denote the set of all keys in S that hash to bucket k. You can easily search for a perfect hash function, h_k, with the algorithm below. Again, the idea is to repeatedly choose a hash function at random until a perfect one is found. In this algorithm, $h_k(x) = (a_k\,x \mod p) \mod 2b_k^2$ where a_k is an integer such that $1 \leq a_k \leq p-1$. To choose h_k at random, the algorithm simply chooses a value for a_k at random.

```
ALGORITHM 2:
  Choose a value for ak at random.
  Repeat until hk is perfect:
    Hash all the keys in set Sk to table Tk.
    If there are no collisions
      then hk is perfect, so store the value of ak;
```

```
else hk is not perfect, so choose another value for ak at random.
```

Observe that when Algorithm 2 terminates, all the keys in S have been hashed. The dictionary is now ready to be searched by users. For our purposes, a user simply wants to know whether a given key is in the dictionary.

Of course, in practice, a dictionary usually stores additional information along with the keys. For instance, the key might be a student number, and the dictionary would store this along with the student's name and address. Typically, a user would provide a key, and the dictionary would retrieve the other information. However, to keep the assignment simple, we will pretend that there is no additional information, and that the user simply wants to know whether a given key is in the dictionary.

To determine whether key x is in the dictionary, you first compute $k = h(x)$. Position k in the top-level table, T, points to a lower-level table, T_k. Then you compute $j = h_k(x)$. Finally, x is in the dictionary if and only if it is stored in position j of table T_k. Observe that this takes only constant time.

As part of the grading, your program will be run on various inputs. This will include specific values for the parameters p and n, a set S of keys to appear in the hash table, and a sequence of keys, $x_1, x_2, \ldots, x_r$, to search for in your dictionary. To simplify the grading, your program should be able to print out the contents of your two-level hash table. In addition, all program input should be read from files, and all program output should be written to files. The format of these files is described below.

You may write your program in C, Java or Turing. You may also develop your programs on any computer that you like, however we must be able to run them on the Unix machines here at U of T. To avoid last minute surprises, we recommend that you regularly test your programs on the Unix machines here. If you leave this to the last minute, you run the serious risk of missing the deadline, since you may find that your program runs at home, but crashes on the machines here. If you miss the deadline for this reason, then your assignment will be counted as late, and no grace will be given.

INPUT FILES: The input to your program should be stored on four files, as follows:

- `parameters.dat`, which stores three integers. These are the values of the parameters p, n and r, in that order.
- `insert.dat`, which stores the set, S, of keys to be inserted into the dictionary. There are n keys in this set.
- `search.dat`, which stores a sequence of keys to search for in the dictionary. There are r keys in this set.

- `random.dat`, which stores a sequence of random numbers, to be used by your program in Algorithms 1 and 2, above.

Each input file stores a sequence of non-negative integers, with *one integer per line*. Your program should read these numbers *in the order given*; that is, the first number in the file should be read first, the second number should be read second, etc. A set of sample files for you to test your program on is available on the course web page. We will test your program on a different set of files.

OUTPUT FILES: The output produced by your program should be stored on two files, as follows:

- `dictionary.dat`, which stores the array, `A`, that holds your two-level hash table. Each array entry should be stored on a separate line. In addition, the string **** should be printed on a separate line between tables, *e.g.*, between tables T and T_1, between tables T_1 and T_2, etc. *Do not print any array entries beyond the end of table* T_n, since we will count the number of lines in your file to see if your 2-level hash table has the right size.
- `results.dat`, which stores the results of searching for the keys in `search.dat`, with one search result per line. Each line of the file should contain either the string "found" or the string "not found". These results should appear in the same order as the keys in `search.dat`. That is, the i^{th} line in `results.dat` should be the result of searching for the i^{th} key in `search.dat`.

PROGRAM STRUCTURE: Your program should carry out the following steps, in the order given:

(a) Read the parameters in the file `parameters.dat`.

(b) Read the keys in the file `insert.dat`.

(c) Construct the two-level hash table using these keys, reading random numbers from the file `random.dat` as they are needed. The numbers should be used in the order given; *i.e.*, the i^{th} number in the file should be the i^{th} random number used by your program.

(d) Print the array, `A`, to the file `dictionary.dat`, as described above.

(e) Read each of the keys in the file `search.dat`.

(f) Search for each of these keys in the dictionary. The result of each search is "found" or "not found".

(g) Output the result of these searches to the file `results.dat`, as described above.

Each of these steps should be clearly documented in your program, so that it is easy for the grader to find them. The last three steps may be interleaved; that is, you may have a loop that on each iteration, does the following: (i) reads a key from file `search.dat`, (ii) searches for it in the dictionary, and (iii) prints the result to the file `results.dat`.

WHAT TO HAND IN:

Using the cdf `submit` command, submit three files: (*i*) a well-documented listing of your program, (*ii*) the file `dictionary.dat` produced by your program, and (*iii*) the file `results.dat` produced by your program. Submit all three files using a single `submit` command. Note that the program you submit should consist of a *single file*, and it should be executable on the Unix machines here at U of T.

In addition, you should hand in (as hard copy) an analysis of the total space used by your resulting data structure in the worst case. (This should be expressed as a function of n, with an explicit constant for the leading term.)