AC21008 Multi-paradigm Programming and Data Structures Assignment 3 - Chained Hash Tables in C++

Due date - Wednesday 14th November, 10:00pm

Part 1 - Chained Hash Table

Your task is to implement a chained hash table in C++. To get you started, skeleton `HashTable.h` and `HashNode.h` files are provided. You must implement the methods declared in the skeleton classes and you may implement additional methods.

Your code must handle failures (such as inability to allocate memory, duplicate hash keys, out of range, etc.) gracefully by throwing an appropriate exception.

Aside from `HashTable.h` and `HashNode.h`, your code may only include the following libraries:

- `<iostream>`
- `<vector>`
- `<list>`

Part 2 - Computing Discrete Logarithms with a Monte Carlo Algorithm

Your task is to implement a tool `mc_dlog` written in C++ that given the numbers g, a, n, outputs with high probability the correct solution x to the equation $g^x = a \mod n$.

You must use your hash table from Part 1 to solve this problem. You may use the `<sstream>`, `<random>` and `<ctime>` standard library in addition to the libraries listed above.

The Algorithm, Step 1: Order of g

If you compute powers of g modulo an integer n, the sequence will repeat itself after a number of steps. The *order* of g is the smallest number of steps after which the sequence repeats. That is, the smallest positive integer k for which $g^x = g^{x+k}$ (for all x).

We must therefore first compute the *order* of the basis g.

To find the order, intialize a hash table called `Ord`. In the following, `h()` refers to your hash function in the `HashTable` class.

Repeat the following steps $\sqrt{n}$ times.

1. Pick a random number r between 0 and $n-1$
2. If $y = g^r \mod n$ is already a key in table `Ord`

   Then: Return `r-Ord[h(y)]` or `Ord[h(y)]-r` whichever is greater than 0. If neither is greater than 0, then continue with step 1.

   Else: Add the key `y` to `Ord` with value `r`. (That is, assign `r` to `Ord[h(y)]`.)

If after $\sqrt{n}$ attempts you fail to find a duplicate key, assume that the order of g is $n-1$.

The order that you find will sometimes be a multiple of the actual order of g. You can improve your algorithm by repeating the above steps and choosing random numbers that are smaller than the order you have previously found.

The Algorithm, Step 2: Finding the discrete logarithm

To compute the discrete logarithm of a to the basis g modulo n, you will use two hash tables, table `A` and table `B`. `h()` again refers to the hash function in your `HashTable` implementation.

Repeat the following $\sqrt{n}$ times:

1. Pick a random number r between 0 and $n-1$.
2. If $y = a \cdot g^r \mod n$ is a key in table `B`

   Then: You are done, output `B[h(y)] - r`.

   Else: Store the value r under the key y in table `A`.
3. Pick a random number r between 0 and $n-1$.
4. If $y = g^r$ is a key in table `A`

   Then: You are done, output `r - A[h(y)]`.

   Else: Store the value r under the key y in table `B`.

Putting the two algorithms together

The result you get from your algorithm in Step 2 needs to be reduced *modulo* the order of g that you computed in Step 1. Use `%` for the modulo operator. If the result after reduction with `%` is a negative number, then add the order of `g` to it.

Both your algorithms may fail to find a result sometimes, that's fine. If your algorithm in step 1 fails to find a result, you may be computing with the wrong order. If your algorithm in step 2 fails to find a result, your program will produce no output.

If you execute your program 10 times, then it should – on average – produce a correct result more than once.

What to implement

The functionality that your `mc_dlog` tool must provide is:

- Calling `./mc_dlog g a n`, where the command line parameters `g`, `n`, and `a` are positive integers, outputs the discrete logarithm of `a` to the basis `g` modulo `n` to stdout (i.e., using `cout`) with some probability grater than 0. It may fail to produce a result or produce an incorrect result with some probability. In both cases the main function's return value must be 0.

  Note: Your program should not output anything other than an integer, unless there is an error.

- If no, too few, or incorrect arguments are given, the program must exit with a non-zero return value and an error message.

What you should submit

You must submit three individual files, i.e., `HashTable.h`, `HashNode.h`, and `mc_dlog.cpp`. We will only look at those three files and *no* other files may be submitted.

Marking Scheme

The letter mark awarded for this assignment is determined as follows:

BF:

- No `HashTable.h` or no `HashNode.h` file has been submitted.
- You have not implemented **your own hash table** based on the provided skeletal `HashTable.h` and `HashNode.h` files.
- The implementation is not an extension of the provided skeletal classes.
- The implementation is a different data structure instead of (a part of) a chained hash table.
- The implementation uses libraries other than the ones permitted in the specification above.

CF: `HashTable.h` and `HashNode.h` are in the repository but one or more of the following issues are found:

- The tester file `test0.cpp` cannot be compiled or does not result in an executable program with "g++ -Wall -Wextra -pedantic test0.cpp"
- The compiled tester file `test0.cpp` crashes.

M3: The code crashes or fails on `test1.cpp`
M2: The code crashes or fails on `test2.cpp`
M1: The code crashes or fails on `test3.cpp`

If all of the preceding tests succeed you will earn `D3` or higher. Additional tests (not provided) will be run on your submitted files (including `mc_dlog.cpp`). For each test that your submitted files pass, you will be awarded the next higher letter mark. For example, if your files do not pass any additional tests, you will get a `D3`. If your files pass 3 additional tests you will earn a `C3`. If your files pass 13 tests, you will earn an `A1`.

The additional test files are not provided, but they concern the following aspects of your code:

- 1 test: Fails if there are compiler warnings when your HashTable code is compiled with the switches "-Wall -Wextra -pedantic" on any of the files `test0.cpp`, through `test3.cpp`.
- 2 tests concern special cases, exceptions, and error handling (out of memory conditions, hash collisions) of your `HashTable.h` and `HashNode.h` implementations.
- 1 test concerns the quality of your hash function (in terms of uniformity of its output, preventing collisions) in `HashTable.h`
- 1 test concerns the efficiency of your code in `HashTable.h`.

To earn marks for the following tests concerning the `mc_dlog.cpp` file, you must have a functioning `HashTable.h` and `HashNode.h` implementation as determined by the tests up to the letter mark `D3`.

- 2 tests concern the basic functionality of mc_dlog.cpp: Your `mc_dlog.cpp` program implements the algorithm specified above and provides the correct (and not hard-coded) output with high probability for small numbers (test A), and for very large numbers (test B).
- 2 tests check whether the smallest solution is found with high probability.
- 1 test checks for compiler warnings when compiling `mc_dlog.cpp` with `-Wall -Wextra -pedantic`.
- 2 tests: Exceptions and error handling (out of memory conditions, command line parameter errors) of your `mc_dlog.cpp` implementation.
- 1 test concerns the efficiency of your `mc_dlog.cpp` code.