

Lab 2

September 27, 2005

Due Date: October 4 (One Week)

1 Background (courtesy of Dr. Jeremy Buhler)

The goal of this lab is to implement hashing as part of a tool for finding matching substrings in two text strings or files. In particular, we'll look at the application for comparing genomic DNA sequences. The approach to biosequence comparison that we'll use here is an important part of such well-known tools as FASTA (Pearson & Lipman 1988) and BLAST (Altschul et al. 1990, 1997).

A DNA sequence is a string of characters, called *bases*, from the alphabet $\{a, c, g, t\}$. Genomic DNA encodes a large collection of *features*, including:

- *genes* – the instructions for building proteins;
- *regulatory sites* – sequence markers recognized by cellular machinery that can increase or decrease the rate at which a given gene is used to make protein;
- *repeats* – junk left behind by *transposable elements*, pieces of DNA that can autonomously copy themselves and move around in the genome. Transposable elements proliferate, then die, leaving behind many inactive copies of themselves in the genome as repeats.

All DNA is subject to *mutations* that alter its sequence over time. However, functional sequence features like genes and regulatory sites are more resistant to mutation than DNA that doesn't code for anything (because natural selection usually kills off organisms with too many mutations in these features). We can therefore find these features by comparing DNA sequences from two different organisms and seeing which parts of the sequences have remained similar to each other since their lineages split from their last common ancestor.

Abstractly, we are given two long strings s_1 and s_2 of characters, and we want to find short substrings (parts of the features) common to s_1 and s_2 . In general, the common substrings might not be exactly the same because even functional sequences mutate over time; however, we often find that s_1 and s_2 still exhibit exactly matching substrings of at least 10 to 15 characters in the neighborhood of their shared features. By detecting these exact substring matches, we can find the most likely locations of the shared features in s_1 and s_2 and can then apply more sensitive but more expensive similarity search algorithms only at those locations.

Naively, we could find k -mers (i.e. substrings of length k) common to s_1 and s_2 by comparing every k -mer from one sequence to every k -mer from the other. Such an approach would take time $\Theta(|s_1| \cdot |s_2|)$, which is unacceptable because interesting DNA sequences range from thousands to billions of characters in length (your own genome is about three billion characters long). Fortunately, there is a much better approach.

Call s_1 the **corpus string** and s_2 the **pattern string**, and assume we are searching for common substrings of length k . We first construct a table T of every k -mer in the pattern string, remembering all offsets in the pattern where it occurs. Then, for each k -mer in the corpus string, we check whether it occurs in the table T ; if so, we have found a match between pattern and corpus. If T supports constant-time insertions and lookups (e.g. if it is a **hash table**), we can process the entire pattern and corpus in time $\Theta(|s_1| + |s_2| + M)$ where M is the number of common substrings actually found. In general, this time cost is much lower than the cost of the naive algorithm. Fast substring matching based on hashing therefore forms the core of many of today's high-speed biosequence matching algorithms.

To make things slightly more interesting, we will also allow the user to specify a **mask string** that contains “uninteresting” DNA. For example, if we’re looking for matching genes, we might not be interested in any common substrings that are part of known repeats. Any k -mer appearing in the mask string is removed from the table before searching for matches in the corpus.

2 Your Assignment

Part 1: Implement a BucketMapping using Open Addressing (45 points)

We have provided you with code that implements most of a sequence matching tool. However, there’s an important piece missing: a `BucketMapping` class supports a mapping from a key (which is a string) to a set (or *bucket*) of data elements that are associated with that key. Your implementation *must* be based upon hashing with collisions resolution by open addressing with double hashing. When an new key is inserted into the mapping, you are required to reuse the first slot marked as deleted (if any) that occurs on the probe sequence before the first empty slot is reached.

The `BucketMapping` class must support the following six public methods¹

`BucketMapping(int capacity)`: this constructor creates a new, empty mapping that can accommodate `capacity` keys while maintaining a load factor of 1/2. (In reality, the constructor would optionally take a desired load factor, but for the purposes of this lab, you can just use a load factor of 1/2.)

`void put(String key, Object data)`: Associates the given data item with the specified key. If the key is already in use, then the data is added to the existing bucket. Otherwise, a new bucket should be created that will contain the single data item. *NOTE: For the purposes of this lab, you can assume that the capacity provided to the constructor is adequate and do not need to include any code to resize the table. However, it would be good to throw an exception if the hash table is full (as opposed to having an infinite loop).*

`boolean containsKey(String key)`: Returns true if the given key is contained in the mapping. Otherwise, false is returned.

`Iterator get(String key)`: Returns an iterator for the bucket of data items associated with the specified key. It should return null if the key is not in use.

`void remove(String key)`: Removes the given key from the mapping.

`int size()`: Returns the number of keys currently held in the mapping. (A key is counted only once regardless of how many items are in the bucket associated with it.)

To help you (and us) check that your implementation is working correctly, a simply test program `Lab2Tester` is also provided. When providing the output from this driver you are to print (from within your `BucketMapping` class) the sequence of slots probed AND the slot where the item is placed when it is inserted into the mapping.

It is your job to be sure that your code provides the functionality described correctly using hashing with open addressing. The TAs will be checking this in your code and deductions will be made if there are errors even if you obtained the correct output for the test data. So do your own testing in addition to the test provided for you.

Your `BucketMapping` will need some internal methods that implement its hash functions. I have provided three of these methods: `int hashCode(String s)` maps a String `s` to an integer value that

¹For those using C++ there will be a few minor differences in the interface that are described at the end of this section.

you can pass to the hash functions. I've also provided the primary hash function, `int hash(int hashKey)`, and the secondary hash function, `int stepHash(int hashKey)`. Do not change any of these provided functions.

In order for the secondary hash function to reach all slots (which is a property you want), the hash table size is required to be a power of 2. Let x be the desired hash table size based on the requested capacity. The size of the hash table you should use is $2^{\lceil \log_2 x \rceil}$ which is the smallest power of 2 that is greater than or equal to x .

Naturally, you may not use the Java `HashSet` or `HashMap` classes (or any comparable STL classes) to implement your dictionary! However, you *may* use the Java `ArrayList` (or whatever other library data structure you would like) to maintain the bucket. Note that the `Iterator` that must be returned by `get`, is supported by the Java `ArrayList`, so you can use its `iterator()` method. You are also welcome to look at the code provided in the CD that came with your textbook, but to make the changes needed for this assignment, you will need to understand open addressing. You are expected to implement the provided methods in an efficient and clean way versus using “round-about” ways because it enables you to use provided code as is versus modifying it.

The interface above is for the Java implementation. For the C++ version, things are just slightly different. The constructor also takes the substring match length to store in an instance variable, `matchLength` so that you can pass raw character arrays instead of String objects as keys. A `Record` class is provided to be used to maintain the buckets. The provided sequence matching tool, directly iterates through the `Vector` of integers held within the `Record` as opposed to using an iterator. Thus the return type for `get` is a `Record`. In order to facilitate the ability for the matching algorithm to free the storage used by the bucket, the return type for `remove` is also a `Record`. For the `put` method the second argument is an integer (as opposed to an `Object`). Finally, a C++ `equals` method is provided that returns true if and only if the first `matchLength` characters of two provided character arrays are equal.

Part 2: Empirically Study Number of Probes (5 points)

Each time you examine an entry in the hash table within the `contains` method, that is called a *probe*. This includes the table entry that ends the search (i.e. the one with the key if is in the hash table or an empty slot if it is not there). To measure how the performance of your hash table varies with the load factor, you are to modify your implementation to compute the average number of probes used by the marked `get` method call made at the end of `FastMatch`. You can modify the driver and/or your code to do this. See the below section on “What to Submit” for details on what you will need to submit.

Command Syntax and Reading From Files

The provided Java driver program has the following command syntax:

```
java FastMatch <transcript file> <corpus file> <pattern file> [ < mask file> ]
```

The corpus, pattern, and mask files contain their respective sequences, while the transcript file stores a record of the program's output. Note that the mask file is an optional argument. If no command line arguments are provided, the user will be asked to input which data set (1,2,3 or 4) should be used. The output will also be saved to a transcript file. More detail is included in the comments in `FastMatch.java`.

The C++ version of the driver omits the transcript file argument since it is easy to save the output using the Unix `transcript` utility.

What to Submit

Please staple the following items and write your name on each page. You should have a cover sheet (with all portions filled in including your name and signature) stapled to the front.

1. Turn in your `BucketMapping` class. If you modify the `Record` classes in any way, turn that in along with the code for any new classes created. (Include any new/revised header files for those using C++). There should be no need to submit any of the other provided code. Just summarize any changes you made (if any).
2. If your implementation is buggy and you were unable to fix the bugs, please tell us what you think is wrong and give us a test case that shows the error. You'll lose fewer points for a wrong answer if you indicate that you know it's wrong (and, if possible, why it's wrong).
3. Turn in the output from `Lab2Tester` where you show the sequence of probes made by both `put` and `remove`, and also output the final slot where the item is placed by `put`. Be sure to check your output is correct and note any problems. To help you check that everything works, here are the values of the hash functions.

	car	bat	cup	box
hash	4	4	1	1
stepHash	3	5	5	1

4. On the course web page (under labs) are four sets of DNA sequences on which to run your program; two of these sets include the correct outputs within the provided README file. Provide the output of `FastMatch` on all four test sets using the following:

	Data Set 1	Data Set 2	Data Set 3	Data Set 4
match length	3	15	18	65

You'll find more information about the Data sets in the provided README file. *Be sure to remove any print lines from within your `BucketMapping` that output the probe sequence. The only output in these tests runs should be those from `FastMatch`.*

5. If you did Part II, you should report the average number of probes made by all calls to the marked call to the `get` method (near the end of `FastMatch`) when using Data Set 2 with a match length of 15. Gather data for hash table sizes of 16384, 32768, 65536, 131072, and 262144. Report the load factor and average number of probes for each of the five hash table sizes. In computing the load factor you should divide the number of slots either in use or marked as deleted by the table size. To do this you may need to modify your `BucketMapping` to keep track of the number of slots used or flagged as deleted. (Just provide us with one copy of your code with the extra lines clearly marked.) Finally, briefly answer the following questions: How does the average number of collisions you found empirically compare to the expected number of probes derived in the text? If there are differences, do you best to explain why.