Assignment 3 - Hashing

COL106, 2019

1 Errata

1. 28th Aug: Defined class `NotFoundException` in the interface file
2. 29th Aug: Changed `bool` to `boolean` as return type of `contains`
3. 30th Aug: Added a note on disallowed imports.
4. 31st Aug: Updated return type of hash functions to `long`.
5. 1st Sep: Updated definitions of `djb2` and `sdbm` hash functions to use `Math.abs()` to handle negative hash values.
6. 1st Sep: Updated definition of `sdbm` to ensure that h_2 is never 0, which causes double hashing to go to an infinite loop.
7. 1st Sep: Added a note clarifying that hash table size T will be prime for double hashing.

2 Introduction

Suppose you are building a database of IITD student records. A student record contains a wealth of information about a student - her first name, last name, hostel, department, CGPA etc. The database is heavily loaded each year during a) the admission period when all the fresher students need to be added to the system, b) the grading period when the CGPAs of all the students need to be updated, and c) right after the convocation period when the graduating batch needs to be deleted from the system.

Since hash tables offer amortized expected constant time costs for insert, update and delete operations, they form a natural data structure choice for the above problem. In this assignment, you will implement the above database using a hash table keyed on the tuple (first name, last name), assuming it is unique to a student. The value corresponding to a key is an object containing all the other details about a student.

Each hash table implementation must deal with the problem of collisions. For this assignment, you need to compare two techniques for collision-resolution: 1) double-hashing, and 2) separate chaining using binary search trees. You need to implement both approaches and understand the impact of various design choices on the overall space and time complexity of all supported operations.

In the double-hashing approach, the hash table is visualised as an array of a pre-determined size based on an estimated maximum size of the dataset. If the estimated maximum size of the dataset is N then $T = 1.5N$ is a reasonably good hash table size. To insert an object with a given key, the index for the key is calculated using a hash function h_1. If the location corresponding to the resulting index is empty, the object is inserted at that location. Otherwise, i.e. in case of a collision, a fresh index is calculated using a different hash function h_2. This process is repeated until an empty location is found. More precisely, at the i-th repetition of this process, the hash of a key k is given by $h(k,i) = (h_1(k) + i.h_2(k)) \bmod T$.

In the separate chaining approach also, the hash table is visualised as an array of a pre-determined size, but this size is independent of the maximum size of the dataset, and is usually much smaller. To insert an object, the index corresponding to the key is first calculated using the hash function h_1. The resulting location contains a pointer to a different data structure, in this case the root of a binary search tree (BST), where all elements with the same index are stored. Since a BST requires that its nodes be comparable to each other, we need to define an order among the objects being stored in the BST. The key (first name, last name) is not comparable, since an object may be “bigger” than another by the first name but “smaller” by the last name. Therefore, we define the key for the BST to be the student’s first name, which is comparable by the alphabetical order.

3 Implementation

At a high-level, you need to implement the following interface.

```
class NotFoundException extends Exception {}

public interface MyHashTable_<K, T> {
   // Insert new object with given key
   public int insert(K key, T obj);

   // Update object for given key
   public int update(K key, T obj);

   // Delete object for given key
   public int delete(K key);

   // Does an object with this key exist?
   public boolean contains(K key);

   // Return the object with given key
   public T get(K key) throws NotFoundException;

   // ”Address” of object with given key (explained below)
   public String address(K key) throws NotFoundException;
}
```

You need to supply two implementations for the above interface - one using the double hashing approach, and another using the separate chaining approach.

The `insert`/`update`/`delete` functions return a count of the number of steps required to perform that operation. For the double hashing approach, this should return the number of times a new hash *h* is calculated. For the separate chaining approach, this should return the number of nodes touched in the BST while performing the given operation.

The `address` function returns the address of the object with the given key. In the double hashing approach, it needs to return the string representation of the index where the object for the given key was finally inserted. In the separate chaining approach, it needs to return a string of the form *index-bstseq*, where *index* represents the index of the key in the hash table, and *bstseq* represents the sequence of *L*/*R* steps taken on the binary search tree of all objects whose index was the same. For example, suppose a student named `Arun Garg` is the first (in alphabetical order of the first name) among the group of 6 students with index 5, then `address(Pair("Arun", "Garg"))` should return ”5-LLL”. As another example, if a student named `"Naveen Kumar"` is the only student with index 2, then `address(Pair("Naveen", "Kumar"))` should return ”2-”.

Finally, the parametric type `K` needs to be a `Pair<String, String>` representing the pair (first name, last name) and `T` needs to be a student record, which is a class that implements the interface:

```
public interface Student_ {
   public String fname();      // Return student’s first name
   public String lname();      // Return student’s last name
   public String hostel();     // Return student’s hostel name
   public String department(); // Return student’s department name
   public String cgpa();       // Return student’s cgpa
}
```

You may assume that the CGPA is not calculated but is entered directly by the user when instantiating an object of this class.

3.1 Hash functions

The performance of a hash table crucially depends on the choice of the hash function(s). Ideally, we would like to have hash functions that provide a uniform distribution of hash values to avoid unnecessary collisions but in practice we will be using hash functions that are fast to compute but possibly not perfectly uniform. (Note that hash tables don't need many security properties provided by the slower cryptographic hash functions like md5/sha256 etc.) We will be using `djb2` and `sdbm` hashing algorithms [1] on the concatenation of the student's first name and last name for hash functions h_1 and h_2 respectively.

```
import java.lang.Math;
public static long djb2(String str, int hashtableSize) {
    long hash = 5381;
    for (int i = 0; i < str.length(); i++) {
        hash = ((hash << 5) + hash) + str.charAt(i);
    }
    return Math.abs(hash) % hashtableSize;
}
```

```
import java.lang.Math;
public static long sdbm(String str, int hashtableSize) {
    long hash = 0;
    for (int i = 0; i < str.length(); i++) {
        hash = str.charAt(i) + (hash << 6) + (hash << 16) - hash;
    }
    return Math.abs(hash) % (hashtableSize - 1) + 1;
}
```

You can assume that for the double hashing approach, the size of the hash table T is going to be a prime number greater than the size of the data set, typically around $1.5N$ to $2N$. This ensures that double hashing approach covers all indices of the hash table (why?).

4 Submission format instructions

As always, you need to create all your `.java` files in a directory named `src`, compress this directory to zip format and rename the zip file in the format `entrynumber_assignment3.zip`. For example, if your entry number is 2012CSZ8019, the zip file should be named `2012CSZ8019_assignment3.zip`. Then you need to convert this zip file to base64 format as follows and submit the `.b64` file on Moodle.

```
base64 entrynumber_assignment3.zip > entrynumber_assignment3.zip.b64
```

Inside the `src` directory, at the minimum you need to have a `README` and a file named `assignment3.java`. In the `README`, you need to report the time complexities of various operations for both the implementations. You should also report any interesting findings based on your experiments with the two implementations.

The `assignment3.java` file should take as command line arguments the hash table size T, the hashing approach (`DH` for double hashing, and `SCBST` for separate chaining with binary search trees) and an input file. A sample invocation looks like this:

```
cd src
javac *.java
java Assignment3 12 SCBST path/to/input.txt
```

The input file contains a new command in each line in the format `commandname [commandargs]`. A sample input file could look like this:

```
insert Ram Singh Nilgiri CS 6.5
insert Lallan Singh Nilgiri CE 6.0
insert Shyam Singh Satpura EE 8.2
update Lallan Singh Nilgiri CE 5.0
delete Ram Singh
contains Ram Singh
get Shyam Singh
```

```
address Shyam Singh
address Ram Singh
```

The corresponding output needs to be printed on the console, the result of each query in a separate line. Suppose we are running for `SCBST` and the index of `Lallan Singh` turns out to be the same as that of `Ram Singh`, but is different from that of `Shyam Singh`. Thus, for the above input file, the output could be:

```
1
2
1
2
2
F
Shyam Singh Satpura EE 8.2
1-
E
```

Here, `F` signifies false (`T` signifies true) and `E` signifies that an error happened (because the requested record does not exist).

Disallowed imports: You are not allowed to import anything from `java.util.*`. In particular, you are not allowed to import hash table, binary search tree or any other data structure implementation from anywhere.

References

[1] *Hash Functions*. http://www.cse.yorku.ca/~oz/hash.html