Department of Computing & Information Systems
Trent University

COIS 2020H: Data Structures and Algorithms

Assignment 2: Hash Tables

Due date: Monday, November 7, 2016 at midnight
No assignment can be accepted after midnight on November 7, 2016
Hints:
Find a partner as quickly as possible
Start early

Some students in our class asked to see an implementation of the hash table class … which got me thinking. Why not implement the hash table as an assignment?

So, for this assignment, the primary goal is to design, implement, and test a generic class called HashTable<TKey, TValue>. The HashTable class is supported by a private class called Entry which stores the key, item, and status of a particular entry of the hash table. The HashTable class also includes the following constructor and methods. You are welcome to define additional properties and methods.

```
public enum TStatus {EMPTY, FULL, DELETED}

public class HashTable<TKey, TValue>
{
        private class Entry
        {
                public TKey Key          { get; set }
                public TValue Item       { get; set }
                public TStatus Status    { get; set }
                …
        }

        private Entry[ ] table;          // array of entries
        private int size;                // capacity of the hash table
        private int count;               // number of entries in the hash table
        private int scheme:              // 1 for linear, 2 for quadratic

        // Note for many methods below, the GetHashCode of TKey is needed

        public HashTable(int size, int scheme)          { … }
        // Creates an empty hash table of size using the resolution scheme 1 for linear and 2 for quadratic

        public void Add (TKey key, TValue item)         { … }
        // Adds an item with key to the hash table (keys must be unique)
```

```
    public bool Remove (TKey key)                  { … }
    // Removes the item with key from the hash table and returns true if done; false otherwise

    public bool ContainsKey (TKey key)             { … }
    // Returns true if the key is used in the hash table; false otherwise

    public TValue Retrieve (TKey key)              { … }
    // Returns the item with key if found; default(TValue) otherwise

    private int Linear (int i)
    // Uses linear probing to return the next available (EMPTY/DELETED) index in the table

    private int Quadratic (int i)
    // Uses quadratic probing to return the next available (EMPTY/DELETED) index in the table

    …
}
```

In the case of linear or quadratic probing, once a table is 72% or 50% FULL, respectively, the size of the table is doubled (in the case of linear probing) or doubled to the next prime number (in the case of quadratic probing). So, additional methods are required to "double" the table size and to determine when a number is prime.

To test the implementation of your hash table, define two additional classes called Coordinate and City which will serve as TKey and TValue, respectively. Because the Coordinate class overrides GetHashCode (from the Object class), it is also required to override the Equals method as well. Again, you can define additional properties and methods (such as ToString).

```
public class Coordinate                                              // used as the key
{
    private int longitude;                                           // from 0 … 99
    private int latitude;                                            // from 0 … 99

    public Coordinate (int longitude, int latitude)    { … }
    public override bool Equals (Object obj)           { … }         // from the Object class
    public override int GetHashCode ( )                { … }         // from the Object class
}
```

```
public class City
{
    private string name;
    private int population;

    public City (string name, int population)          { … }
    …
}
```

Testing

To gain confidence that the individual parts of your program are working properly, test incrementally (a bit at a time) and assemble the results of your testing for the grader. Consider the following testing steps:

1) Test the Coordinate and City classes first, particularly the Equals and GetHashCode methods
2) Test the Entry class as a public class using Coordinate and City
3) Test the HashTable class by testing each individual method (private and public), again using Coordinate and City

Empirical Results

Now that everything has been tested, create two hash tables, one using linear probing and another using quadratic probing. Randomly generate 1000 instances of City and insert each instance into both tables. For convenience, all cities can have the same name and population. However, keys (i.e. coordinates) must be unique. Coordinates from 0 to 99 can be generated using r.Next(100) as in Assignment 1. Finally, keep track of the total number of collisions for both resolution schemes.

Repeat this experiment 20 times with 1000 cities each and average out the number of collisions over all trials for each scheme. Record your results.

Assume that the initial table size is lucky 13.

What to Submit

As usual, submit the source and executable files as well as the test and empirical results both digitally and in hard copy (except for the executable).

Grading Scheme

Coordinate class	12%
City class	4%
Entry class	4%
HashTable class	50%
Testing	12%
Empirical results	9%
Inline documentation	9%