Assignment 3: Fun with hash tables

(for some definition of the word "fun")

Out: Friday, April 29
Due: Friday, May 7, noon

Important: change in late policy and extension policy

As discussed in class, because of technical limitations with Canvas, we can't implement the logarithmic penalty scale for late assignments. So starting with this assignment, we will be applying a **flat 30% penalty for all assignments marked late** by Canvas.

We will also be handling extensions differently now that Canvas understands what an extension is. So starting for this assignment, you should **register for extensions by emailing ian@northwestern.edu** and I will add you to the extension list on Canvas. As always, you must request an extension **at least 24 hours before the normal deadline**.

Overview

In this assignment, you will implement both chained and open-coded hash tables, as faster alternatives to the ListDictionary class you implemented in Exercise 2.

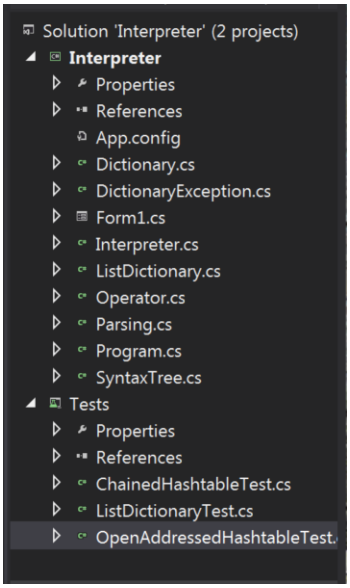
Nerdly grammar note: "hash table" is in fact two words. That means that, by most naming conventions, a class implementing a hash table should be called something like "HashTable" or "ChainedHashTable". However, in practice, many systems, including .NET, just say "Hashtable", and people even often write "hashtable" as one word in English. We will follow that convention in naming classes here.

Getting started

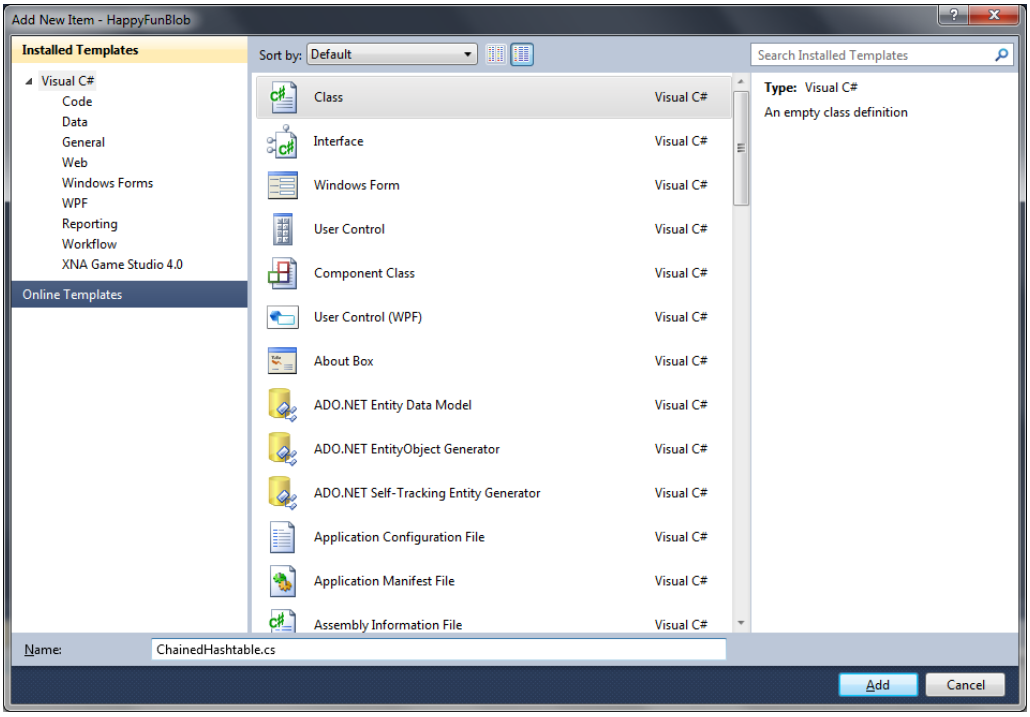
This is a modified version of assignment 2. So you should start by making a copy of your completed assignment 2. Then, replace the DictionaryException.cs file with the one provided with this assignment (otherwise you won't have the HashTableFullException class, which you'll need).

Adding files

You should start by adding two new files to your project. Start Visual Studio by clicking the Interpreter.sln solution file as usual. Then, in the "solution explorer" (the list of files on the right of the window), find the Interpreter project:

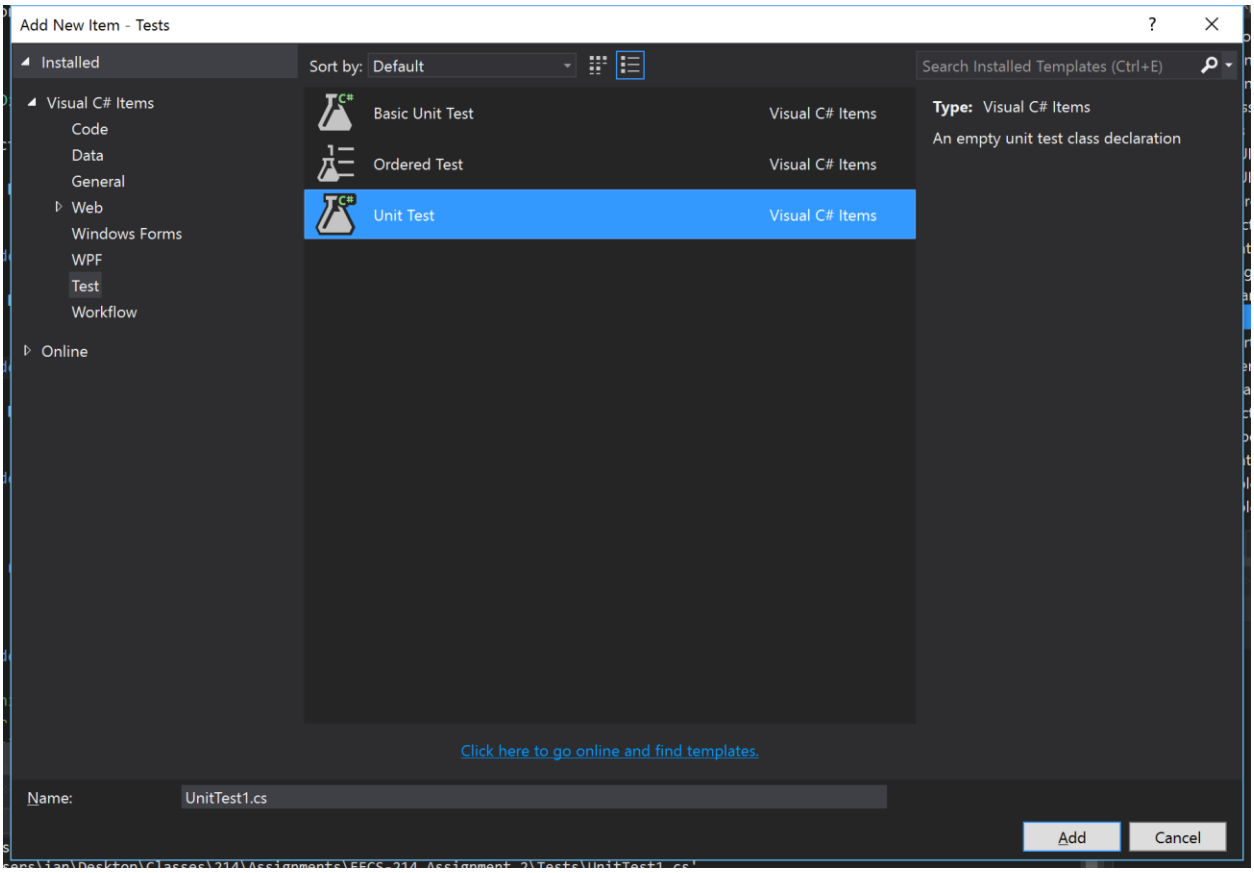


Note that this is not the whole solution (which is also called Interpreter), but the "project" for the interpreter application. It appears after "Solution items" in the solution explorer, but before "Tests" (the other project). Right click on the Interpreter project, and choose **Add**, then **New Item**. It will pop up a dialog box to let you choose the kind of file to add. Click where it says "Class" at the top (i.e. you want to add a file that holds the code for a class) and then enter **ChainedHashtable.cs** at the bottom as the name of the file and press **Add**:



This will add a new source file named ChainedHashtable.cs. Now repeat the steps above to add a second file called **OpenAddressedHashtable.cs**.

You'll also want to write unit tests for these new classes, and so you'll need files for those. You make those in the same way, except you right click on the **Tests** project rather than the Interpreter project. Again, you select Add and then New item, but then under type of item, choose Unit test:



And remember of course to fill in the file name (e.g. ChainedHashtableTests.cs or OpenAddressedHashtableTests.cs) at the bottom.

Important note

These problem sets are machine graded by taking the .cs files we expect you to have worked on (in this case, ChainedHashtable.cs, and OpenAddressedHashtable.cs), and copying them into our own version of the Visual Studio solution, that contains the necessary tests for grading. That process will completely fail, and **you will get a 0 grade**, if you:

- Add extra files we didn't ask you to
- Misspell the names of the files (although you don't have to worry about the names the files with your unit tests, since we won't be grading those)
- Misspell the names of the classes or methods
- Change the arguments or return types of the methods
- Use a different capitalization convention for the file names, classes, or methods.

That is, if you call your class ChainedHashTable rather than ChainedHashtable, the compiler will reject your code and we won't even be able to run any of the unit tests on it. So be sure to double-check the relevant spelling issues, and don't change the interfaces (method names, arguments, types, etc.) of the code we handed out!

Creating the new classes

Your new Hashtable classes will be just like your ListDictionary class, in that it will be a subclass of the existing, abstract class Dictionary. The Dictionary class defines the two familiar Store and Lookup methods, plus the Count property:

- public abstract void Store(string *name*, object *value*)
  Adds *name* to the dictionary with the value *value*. If name already appears in the dictionary, its entry should be modified to have the new value *value*. If (in the case of open addressing), the hash table is full, the store method should throw the HashtableFullException.

- public abstract object Lookup(string *name*)
  If the dictionary contains the key name, then it returns the value associated with it. If they key is not found in the dictionary, then it should throw a DictionaryKeyNotFoundException, by executing the following code:

  ```
  throw new DictionaryKeyNotFoundException(key);
  ```

  where `key` is the name it failed to find in the dictionary.

- public abstract int Count { get; }
  This is a property with just a get method. It should return the number of items stored in the dictionary.

You will make two classes, ChainedHashtable, and OpenAddressedHashtable. You should put the code for them in the files with their respective names, and you should declare each class to be **public** (that's necessary for the testing code to access it) and to be subclasses of **Dictionary**. And they should each declare a public constructor that takes the size (number of buckets) of the hash table as an argument. So their class declarations should start out with:

```
public class ChainedHashtable : Dictionary
{
        public ChainedHashtable(int size)
        {
```

in the case of the ChainedHashtable. The OpenAddressedHashtable should look the same (but with "OpenAddressedHashtable" substituted for "ChainedHashtable", of course).

Stubbing out the methods adding the methods

Start by writing **stubs** for all the methods in the classes. To make a stub, just write the declaration for the method as usual, but for the body of the method (i.e. the code inside that does the work), just write:

```
throw new NotImplementedException();
```

This should be pretty familiar to you from previous assignments where we wrote the stubs for you. So the stub for the constructor looks like:

```
public ChainedHashtable(int size) {
        throw new NotImplementedException();
}
```

This will satisfy the compiler enough to let things compile without you having to write all the code for all the methods at once.

For the Count property, the stub is a little different. Remember that properties are just methods with no arguments that you call as if they were fields. So if the user makes a hash table, h, and then says "h.Count", they're really calling a method even though the call is written as if Count was a field.

The syntax for defining a new property is:

```
Type Name
{
        get
        {
                … your code …
        }
}
```

In this case, the Count property is public and also overriding the abstract declaration from the parent class, so you'd say:

```
public override Count
{
        get
        {
                … your code …
        }
}
```

You should include stubs for the constructor, Lookup(), Store(), and Count for both classes. Again, mark them **override**s of the parent class and **public**.

Writing the class implementations

Now you should fill in the implementation for each class, including the methods that you have stubbed out. Write each class to use the appropriate form of collision resolution (chaining for ChainedHashtable, open addressing for OpenAddressedHashtable).

Computing a hash on a string

Before you can make the hash table, you'll need a hash function. Since the hash tables in this assignment use strings as their keys, you only need to worry about hashing strings, not other kinds of objects. For this assignment, you can use the **sum of all the characters** in the string as your initial hash (see the hash table lecture). It's not a great hash function, but it's good enough for this assignment.

To map the initial hash into to a hash table bucket, you should use the **multiplication method** discussed in the lectures and the book. Use Knuth's value, $(\sqrt{5}-1)/2$, for the A coefficient. When doing this, remember that:

- The square root function is called **Math.Sqrt** in C#. So $\sqrt{5}$ is **Math.Sqrt(5)**.
- The mod function is typed as a **%** sign in C, C++, and C#, so $x \bmod 1.0$ would be x%1.0.
- The floor function, $\lfloor x \rfloor$, which rounds downward to the nearest integer, can be computed just by casting to an int.[1]

Implementing open addressing

For open addressing, you should use linear probing. Like the hash function, it's not what you'd use for an industrial strength system, but it's fine for this assignment.

Open addressing is easier to implement if you use two separate arrays for the keys and the values, rather than making a new class that colds a key/value pair and making one array of pairs.

Throwing exceptions

The Store() method for OpenAddressedHashtable will need to throw the HashtableFullException if the program attempts to store a new key into a hash table that's already full. You can do this just by saying:

```
throw new HashtableFullException();
```

As with the ListDictionary, the Lookup() method needs to throw the DictionaryKeyNotFoundException when the program tries to lookup a key that isn't present in the hash table. Again, you can do this just by creating an exception using new and then throwing it. The only difference is that the constructor for DictionaryNotFoundException takes the key that wasn't found as an argument:

```
throw new DictionaryKeyNotFoundException(key);
```

where *key* is the argument that was passed to Lookup().

---

[1] This technically only gives the right answer when the number you're rounding is positive. But our hash function is always positive, so we're safe.

Defining the Count property

Remember that the count property should just return the number of items in the hash table. Most likely, you'll implement it by adding a separate field that holds the "real" count, modify the Store() method to update it when appropriate, and then have the count property return the value of the field. So then why not make it a field? Because for other types of Dictionaries, it might be computed in some other, more complicated manner. So to allow for that, the Dictionary class declares it to be a property with a get method. And it declares it abstract (i.e. virtual) so that every particular subclass of Dictionary is free to have its own method.

Note that the number of items shouldn't increase if you store a new value for a key that was already in the dictionary. That is, if you store the 2 as the value of "x", but "x" was already in the dictionary, you haven't increased the number of items in the dictionary; you've just changed the value associated with an existing item. Your testing code can check for this by first storing a value under a key in the dictionary, then storing a different value, and then checking to make sure the Count didn't increase.

Testing

As before, you should create test cases to test your hash tables. You can likely take a lot of your test cases from you ListDictionary tests and simply alter them to work with your hash table classes. However, you may also want to add tests to check things like whether your open addressed tables notice when they fill.

In addition, you can try out your hash tables by modifying the file GUI.cs to use your hash tables rather than your list dictionary. Just open the file and find the line:

```
private readonly Dictionary environment = new ListDictionary();
```

and modify it so that makes a ChainedHashtable or OpenAddressedHashtable instead of a ListDictionary.

Turning it in

Before turning in your assignment, you should double-check the spelling and capitalization of the file names, classes, and methods you created. Again, if you turn in code with the wrong names, the grader will fail completely, and you will receive a 0.

As before, make a ZIP file of your project and upload it to canvas.