Hash Table Demonstration: Class and Method Documentation

Author: Dominic “Seth” Jones-Jackson (She/They)  
Initialization Date: December 6, 2020  
Finalization Date: December 9, 2020

# Abstract

A “hash table” is a data struct designed to pre-sort provided data in order to reduce the time complexity of searching for a particular datum. This is accomplished by using a “hash” method to determine where in an array a particular datum should go; then, when searching, the same hash method cam be used to establish where to look right away. This reduces overall time complexity of searches to almost O(1). (I say “almost” because sometimes a “collision” causes multiple data items to be put in the same index, and so a few extra steps are needed to find an exact item in those cases.)

For our demo, we’re using a list of contacts as the data in question, and we’re resolving collisions via simple linked lists in each index. We’re getting those contacts from phonenumbers.json via System.Text.JSON .NET Core’s namespace.

# Contact Class

A very basic class to associate contact names with contact methods.  
  
Properties:

* *string* Name
  + The name of the contact in question. This is the value used to “hash” a Contact (See “HashTable” for more information.)
* *string* Phone
  + The method of contacting this person. This is a string because, while we named it “Phone” to prevent clashing with the class name, it also allows for other types of contact methods such as e-mail.

Methods:

* *public* Contact(*string* name, *string* phone)
  + A “standard” constructor.
* *public* Contact()
  + A “default” constructor, that sets Name to “Optic” and Phone to "098-765-4321". This is mostly arbitrary – a default constructor is required for deserializing our JSON file.
* *public override sting* ToString()
  + Returns the following string:  
      
    "Name: (this.name)\n

Contact: (this.phone)\n";

# Node Class

A simple linked-list for resolving collisions. There’s a few efficiency methods that a normal linked list would have that we have not bothered with, as we’re not expecting to have more than a few Contacts in any given list made this way.  
  
Properties:

* *Contact* Value
  + The Contact object this Node is storing.
* *Node* Next
  + The next Node in the list.

Methods:

* *public* Node(*Contact* value)
  + The only constructor. “Next” is left null by default, indicating that this node is the last node in the list until a new one is added.
* *public void* Add(*Contact* value)
  + A recursive method that carries a value to the end of the list and appends it there.
* *public int* Count()
  + Another recursive method that returns the total number of Nodes.
* *public static Contact[]* Array(*Node* input)

Takes a linked list and returns an array of its contents.

# HashTable Class

A simple linked-list for resolving collisions. There’s a few efficiency methods that a normal linked list would have that we have not bothered with, as we’re not expecting to have more than a few Contacts in any given list made this way.  
  
Properties:

* *Node[]* Table
  + An array of linked lists, each of which contains some number (Or not) of Contacts.  
      
    We did it this way for two reasons. Firstly, the linked list allows us to resolve collisions by allowing an arbitrary number of Contacts in the same index. Secondly, the Node can more gracefully be left “null” if there isn’t a Contact in a given index.  
      
    Note: IT DOES NOT VALIDATE filenames as of now – you must provide the full path, and must be sure yourself that it is in fact a JSON file.

Methods:

* *public* HashTable(*int* size)
  + The “regular” constructor, not actually used in this demo. A starting “length” is necessary for the array, and typically it should be close to the number of expected Contacts to be sorted into it.
* *public* HashTable(*string* filename)
  + This constructor pulls in an array of Contacts from a JSON file with the provided filename. (The demo uses “*./phonenumbers.json*”, provided in both the Debug and Release versions.)
* *public void* Save(*string* filename)
  + Saves the table’s contents as an array of Contact objects in the indicated file.
* *public override string* ToString()

Returns a string of each index and the Contacts stored in those indices, including the “empty” indices.

* *public Contact[]* ToArray()

Returns an array of Contacts. Mainly used for the Save() method.

* *private int* Hash(*string* input)

The only private method. Casts each letter in the input as an integer, sums them, and takes the modules of the sum against the Table property’s length, to establish where in the array the input string should go.

* *public void* Add(*Contact* fox)

Takes the “hash” of the provided Contact’s Name property, and puts it in the appropriate index. (I don’t remember why I called the parameter “fox”, and I’m too scared to change it now.)

* *public Contact* Find(*string* input)

Takes the “hash” value of the input string, and checks the appropriate index for a Contact with a matching Name property. Throws an exception if there isn’t a matching Contact.