

SCHOOL OF COMPUTING SCIENCE

Hash Table Demonstrator

Dragoş Miron - Najam Hussain - Lukasz Prelic - Daniel Budeanu - Jon Elliot

TABLE OF CONTENTS

Introduction to Hashing	3
Collision Handling	4
How to use the demonstrator	6

INTRODUCTION TO HASHING

Definition

A hash table consists of two components: a bucket array and a hash function. The hash function maps elements into the bucket array.

Hash Functions

Hash functions are algorithms that return hash codes used for elements to be entered into the bucket array. The ideal hash function generates different hash values for different elements, but this is not the case in real-life. Whenever a hash function returns the same hash code for two elements, a collision occurs. There are different ways of handling collisions (see COLLISIONS HANDLING chapter).

Uses

Hash functions are mostly used to generate fixed-length output data that acts as a shortened reference to the original data. This is useful when the output data is too large to use entirely.

One practical use is a data structure called a hash table where the data is stored associatively. This demonstrator will show how this data structure is created, how elements are inserted, removed or found and how collisions can be dealt with.

Another use is in cryptography, the science of encoding and safeguarding data. It is easy to generate hash values from input data and easy to verify that the data matches the hash, but hard to 'fake' a hash value to hide malicious data.

Performance of hash tables

Hash tables aim to have all their operations (insert, remove or find) in constant time. The worst case scenarios complexity depend on the manner in which the program deals with collisions (see COLLISIONS HANDLING chapter). The hash function also has an impact on the performance because if the number of collisions is minimised, the data structure will be more time efficient.

COLLISION HANDLING

There are lots of ways in which collisions are dealt with. This tutorial discusses two: linear probing and separate chaining.





Linear Probing

In case of linear probing, when the program tries to insert an element in a bucket that already contains another element, it will simply iterate over the buckets until it finds an empty one and insert the element there. In the worst case scenario, this can have an $O(n)$ complexity, where n represents the number of buckets. Also, when finding or removing an element that has been placed in a bucket which was found by linear probing the performance might drop to $O(n)$.

In the case of linear probing, the buckets can have three states: empty, deleted or full. When a bucket is empty, it means it doesn't and never contained an element. When a bucket is deleted, it means that currently it does not contain an element, but there has one that has been removed. The third state is when the bucket currently contains an element.

Inserting

When inserting an element in a bucket array and using separate chaining to deal with collisions there are 4 situations:

-  If the bucket is empty, the insertion takes place
-  If the bucket contains the element, the insertion fails because duplication is not allowed
-  If the bucket contains a different element, the program iterates over the buckets to find an empty or deleted bucket to insert the word in
-  If the bucket is in the deleted state, the program assigns a checker to iterate over the buckets until it finds an empty bucket or until it loops over, checking the full array. This is done because the checker might find the word in the array, in which case, insertion fails due to duplication restriction. Otherwise, insertion takes place in the deleted bucket.

✚ Removing and finding

When removing or finding an element, the program starts at the index indicated by the hash function, after which it iterates from that point onwards until it finds:

- ✚ an empty bucket in which case, the element does not exist, or
- ✚ the element, in which case the element is removed or found or
- ✚ a different element or a deleted bucket, in which case it continues to iterate

If the program iterated and looped over the same bucket, it means that the element is not in the hash table.

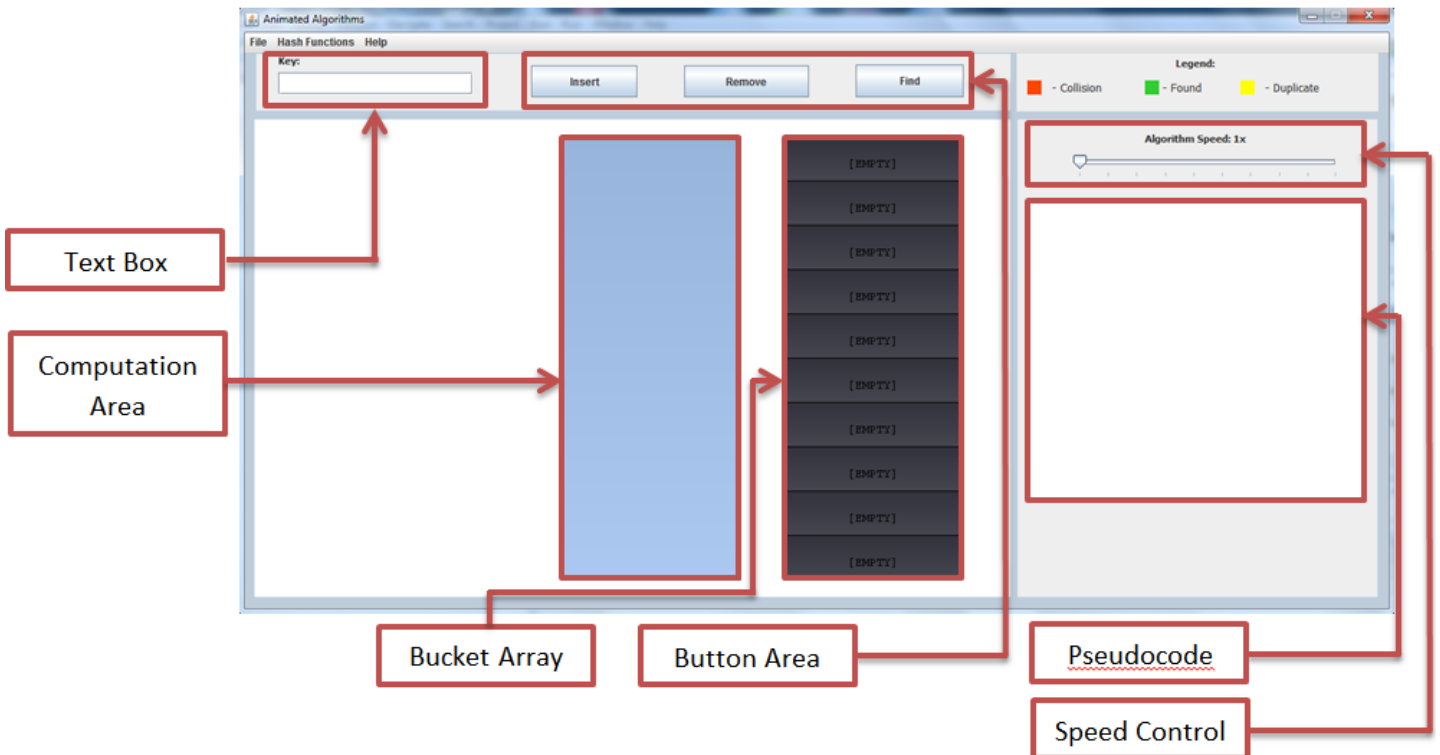
Separate Chaining

In the case of separate chaining, each bucket is a small map, for example a linked list. When the program wants to insert, remove or find an element, it calculates the hash code for it and then it accesses the indicated bucket. If the bucket contains that element, insert fails, but remove or find succeed. If the bucket doesn't contain the word or if it is empty, insertion succeeds while remove and find fail.

The worst case scenario complexity of inserting, removing or finding an element in this data structure depends of the data structure used for each bucket list. If a linked list is used, the operations have a worst case scenario complexity of $O(n)$, where n is the number of elements in the table.

HOW TO USE THE DEMONSTRATOR

To run the demonstrator simply double click on one of the executable .jar files named HashTableLP.jar or HashTableSP.jar and the following screen will appear:



To interact with the demonstrator, you first select one of the hashing functions from the Hash Functions menu. AddHash is selected by default. After, you can type in a word in the text box and press one of the three buttons in the button area. The animation will take place in the Computation Area and in the Bucket Array while the Pseudocode area will show the steps the algorithms is going through. The speed of the animation can be modified using the speed control slider.

Demo:

Firstly, run HashTableLP.jar with selected AddHash from the Hash Functions menu. Then:

1. Insert asd, then ads, sda, sad, dsa and das. This will show you how collisions are handled.
2. Now delete ads and sad. This shows you how the program iterates to find the elements.
3. Now try to delete sad again. Notice how the program checks until in finds an empty bucket. To demonstrate why it is not enough to stop at a deleted bucket, do step 4.

4. Delete sda. Notice that if the iterator would have stopped at the first deleted bucket, it would not have found the element.
5. Insert sda. Notice the checker iterating until it finds an empty bucket. This is done to avoid collisions. In this case, insertion will be permitted. Do step 6 to see another example.
6. Insert das. In this case the checker finds the word and does not allow insertion.
7. Insert jar, jra, ajr and arj. Now insert sad. See how the checker loops over and meets a checked value. In this case, even if it did not encounter a duplicate or an empty bucket, it still grants permission to insert.
8. Now insert ads. The capacity is now full. Try to insert anything.
9. Play with it. Try to insert, remove and find elements. If you want to reset the buckets to empty, select any hashing algorithm from the menu.

Now run HashTableSC.jar. A similar screen will appear. The difference is that the buckets can now support more than just one element. So when two elements have the same hash code, they are both inserted in the same bucket. The number on the buckets represents the number of items in that bucket. This demonstrator has a limit of 10 elements per bucket. To see the elements from a bucket, hover the mouse over that bucket and they will appear on the left of the Computation Area.

Note: An empty bucket will be grey. A bucket containing one or few elements will be green. As the number of elements of a bucket gets close to the limit, the bucket will shift color to red.

Demo:

1. Import animals.txt contained in this pack. (File -> Import)
2. Insert duplicate elements to notice that their insertion is denied. Insert sms to notice that insertion is denied because the bucket is full.
3. Try to find or to remove existent and inexistent elements.

Congratulations, now you should be able to understand Hash Tables.