Hashing

By: Anthony Nguyen

1. Introduction

Lately, we've been learning data structures where items are put in pretty random locations, and without actually looking at the entire data structure, finding items in the data structure is pretty tedious. Wouldn't it be nice if you were just able to pin point your item straight away (or almost straight away) (or in technical terms O(1))? This is where hashing comes in.

A hash function is a function that takes the value of an item and then spits out an array index to store the item. For consistency, let's call this item a **key**. The array that holds all of these keys is called a **hash table**. As long as the array is big enough, functions like insert, find, and remove are really quick. Why?

1. For **insert**, all you have to do is run the hash function, get the index, and put the key in that index.
2. For **find**, all you have to do is run the hash function, get the index, and find out if it's there or not
3. For **remove**, you run the hash function, get the index, if it's there then take out the key

But then why doesn't everyone just use a hash function? There are some issues.

1. First off, because the size of an array is finite, only so much data can be stored in a single hash table.
2. Second, each spot in the array, or **bucket**, can only hold one key. This means that if a hash function spits out the same array index for two different items, we have to work around that. When there is a conflict for an array spot, we get what is called a **collision**.
3. Third, since array indexes are all integers, if you want to write a hash function for something like a string you have to figure out a way to smartly change your string into an integer.

Some hash functions are better than others. Good hash functions don't cause lots of collisions, use the space efficiently, and spread out the values so you don't get what's called **clustering**.

2. Example of Hash Functions

2.1 Division Method

If the values of the keys are all integers (or some of them at least), then an appropriate hash function to use would be the **division** hash function. The function looks like this:

$$h(x) = x \; Mod \; n$$

Where h(x) is the array index, x is the numeric value of the key, and n is some number that is pre-defined. The value for n is determined strategically by the programmer. Pick an *n* that will minimize the number of collisions.

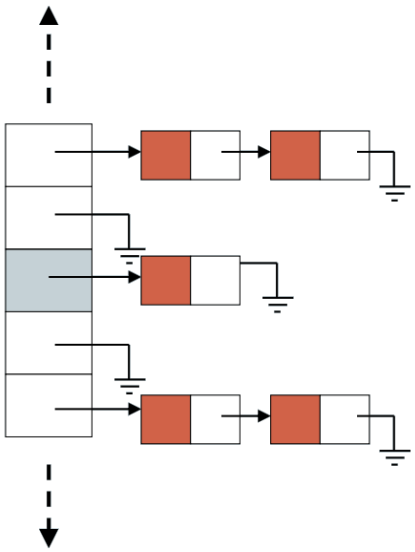
3. Getting Around Collisions

Collisions suck but there are ways to get around it. This is completely situational.

3.1 Separate Chaining

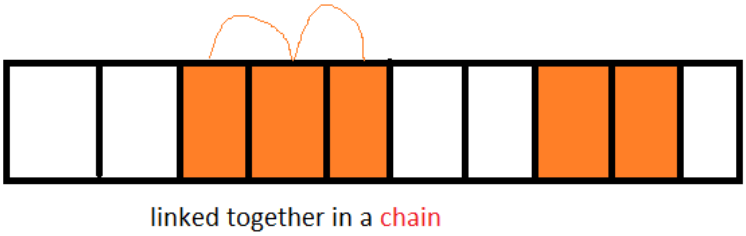
For this method, when an index in the array is filled and the hash function tells you to put it there, you simply prepend it onto the array index to create a chain on that index. This **eliminates all collisions**, and

is therefore the safest method, but means that you can't necessarily find the array index immediately. It also makes removal more difficult. The hash table looks like this:

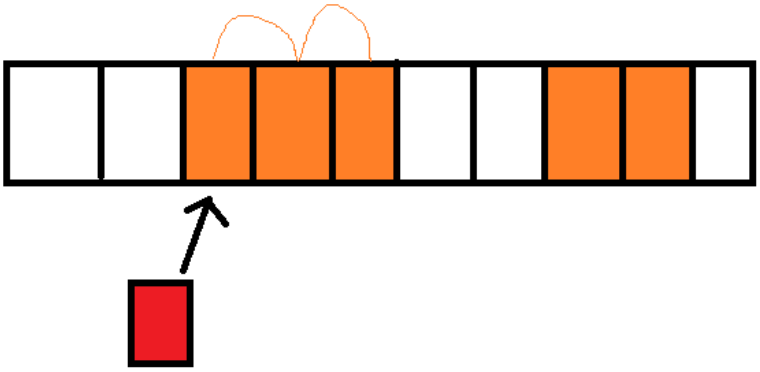


3.2 Regular Chained Scatter Table

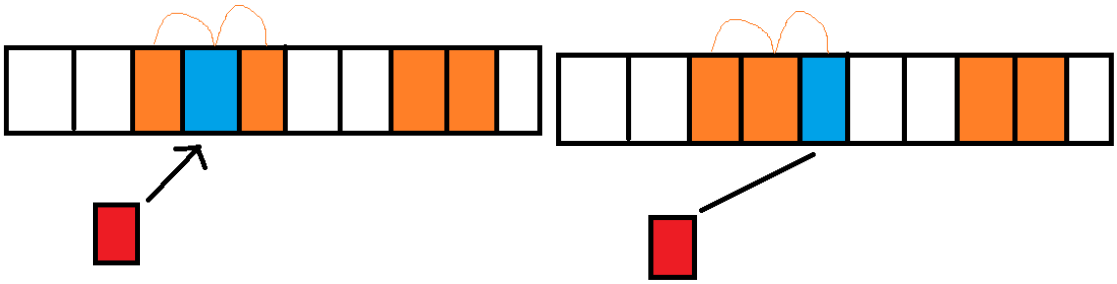
With a regular chained scatter table, when a collision occurs, the key will be inserted at some other index in the array. Each index in the array will contain a value node and a pointer node much like a linked list. This new location will be pointed to by the pointer from the old array location. Subsequent keys in an array are called **a chain**.



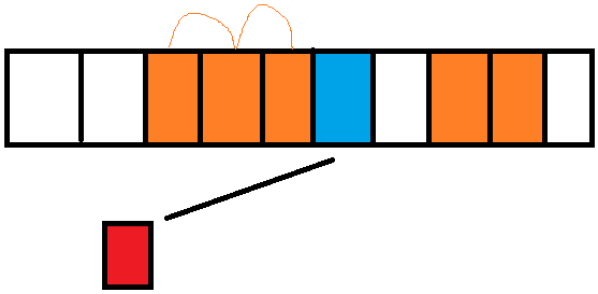
How do we go about finding this new location in the scatter table? Consider this example here.



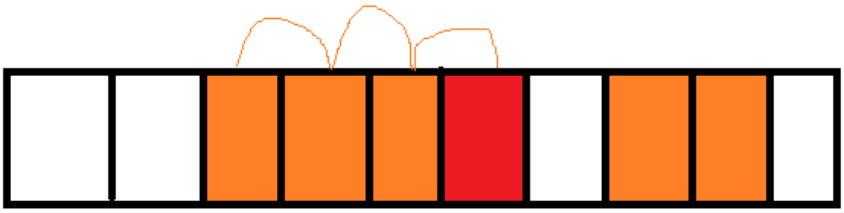
The hash function points to this pre-occupied space, and the red key wants to enter the hash table. What we do first is follow the pointers until we get to the end of the chain.



Once we have the end of the chain, we then do a linear search through the array to find an empty array index.



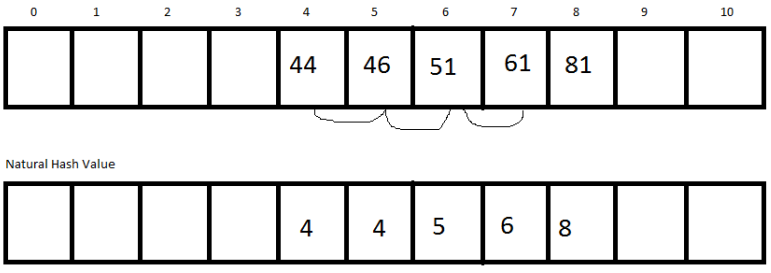
Once we have an empty area, we place the key into that location, and make the end of the chain point to that location.



For finding, you only have to traverse the chain to see if it's there.

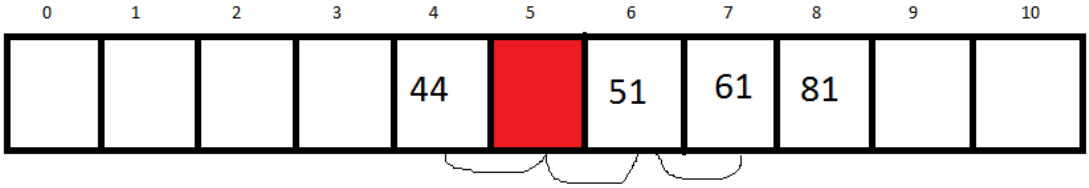
3.2.1 Withdrawing from a Regular Chained Scatter Table

Withdrawing from a chained scatter table is a little tricky. In a chain, if you remove an item from the middle, you don't want something called a **hole**. You also want to make sure that when you're switching values, you can still find easily find an item in the scatter table. Look at this example:

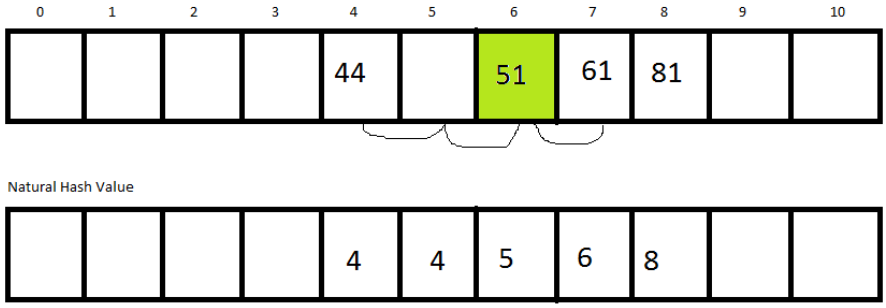


The hash function for the array is the first digit of the number. There's an array for the actual position of the value, and there's another array just for the natural hash value (if there wasn't a collision). If we wanted to remove a value in the middle (i.e. 46):

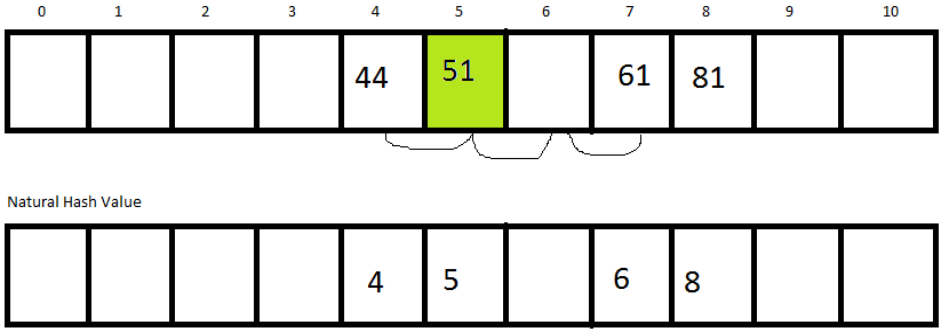
1. We first check to see if the item we're trying to remove is in the scatter table.
2. We check to see if it's at the end of the chain. If it is, we can literally just set the element equal to null and make element before it point to null. 46 is not at the end of the chain so we can't do that. This acts as almost a base case.
3. We erase the value at the position it's at.



4. This empty space creates a hole in the chain. We want to avoid this. To fix this, what we do is we take the **next element in the chain with a natural hash value GREATER THAN OR EQUAL TO the empty array index.** This is so that we can still find the value using the hash function! **Never put a value at a spot lower than its natural hash value.** We then use that value to fill the hole. In our case, the next element in the chain is 51 and its natural hash value is 5.



The array index we're inserting to has a value of 5. We can insert at that spot because 5=5.



5. We then do the same thing for the new hole. We do it until we reach the end of the chain, in which we are at step 2 and the entire procedure ends.

Sometimes, we are unable to find a value in the chain that fits the requirement, and we have to end the chain there and have a hole.

3.3 Open Addressing Methods

Open addressing is a method that does not use pointers but rather some algorithm that will bring you a different array index. It's by far the least expensive but causes more collisions than the linked list ones.

3.3.1 Linear Probing

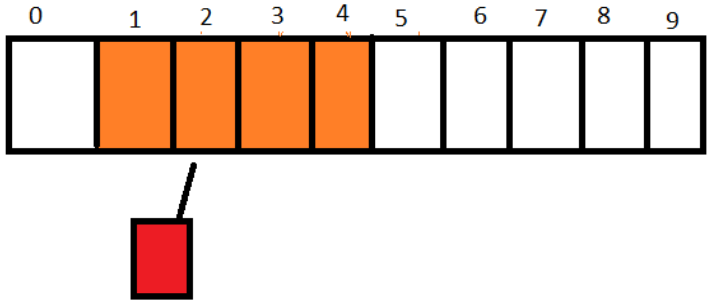
Linear probing involves running your hash function again if there is a collision and adding one to the result. This is basically a linear search for an open array index location. The function looks like this:

$$h(x,i) = (h(x) + i)\ mod\ M$$

This is where h(x,i) is the array index location, *i* is the number of times you've run the probing (minus one), h(x) is the original hash function, and *M* is the size of the array. The mod M is important because it makes sure that you don't go over the size of the array in your search.

Now, suppose we have this scenario here, where the hash function spits out a value of 2 for some key that you want to insert.

$$h(x,0) = (2 + 0)\ mod\ 10 = 2$$

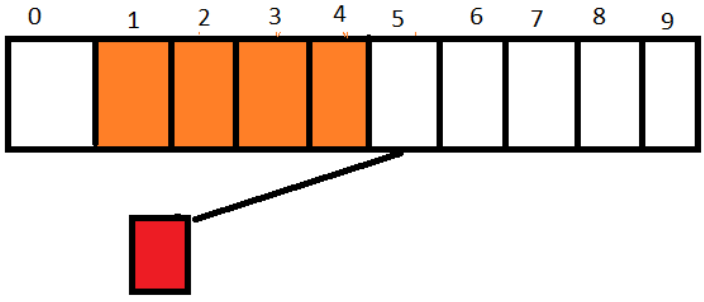


Well, there's a collision. What you do is run the hash function again and add one. You'll get three.

$$h(x,1) = (2 + 1)\ mod\ 10 = 3$$

That one's filled too. Well writing out all the steps is tedious so I'm just gonna go ahead and skip to *i=3*.

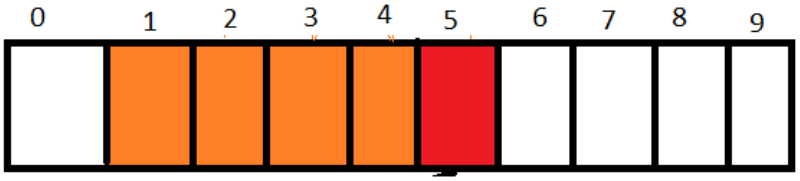
$$h(x,3) = (2 + 3)\ mod\ 10 = 5$$



No way, and empty index! Fill it in and you're good to go.

3.3.2 Quadratic Probing

Linear probing actually sucks because it takes really long if there's a chain, and if you have values that will give similar array indexes you'll get what's called a **cluster**. A cluster is when there are too many filled array indexes in succession. Hash functions are supposed to spread out keys in the array.



too many things in a row..

To fix this problem (well sort of), we use quadratic probing. Quadratic probing is very similar to linear probing except we change up the formula a little, so that we square the *i* value.

$$h(x,i) = (h(x) + i^2)\ mod\ M$$

So now, collisions won't be dealt with by putting the item next to the array index, it'll be put at some value farther away. I'm not going to visually show the example to this, you can figure it out. It's very similar to linear except you square your *i*.

What is wrong with quadratic probing? The problem is that it is not hard to repeat array indexes with the quadratic probing. Eventually, when the array gets more than half full, there is a good chance that the probing will go into an infinite loop and you will never find that needed empty array index. This is even worse if the value for M is not a prime number.

3.3.3 Double Hashing

In many cases, double hashing works the best. Double hashing takes one hash function, runs it, and if there's a collision, it will run another hash function multiplied by *i*.

$$h(x,i) = (h1(x) + (i * h2(x))\ mod\ M$$

4. Implementation in Code

Now we get to the fun stuff (but not really) and implement this in code. We're going to use open addressing to make this thing.

4.1 The Data Structure

```
1. typedef struct scatterTableType
2. {
3.         int M, N ;
4.         char **array ;
5. } scatterTableType ;
```

scatterTableType is the name of our data structure. It contains the members **M (size of the array), N (number of inserted elements in the array), and **array (the actual array of the scatter table).** Remember that *array* doesn't have to be a character, it can be any data type you want.

4.2 Properly Calling the Hash Function

The Open Addressing function

When we call the hash function in the main code, we're actually calling the **open addressing function** known as *h*. As input, this takes in the size of the array, the number of times (minus 1) it's been run *i,* and the value.

Calling the open addressing function somewhere else looks like this (**this is not the actual open addressing function**) (this is done in the **find function** which will be explained later):

We want the open addressing function to loop until we find an empty location or we find a duplicate value. We use a variable called *found* to keep track of this.

```
1.      do
2.      {
```

Every time the function runs, there is a collision, and we add one to the *i* counter.

```
3.          i++ ; //tracks number of collisions.
```

We then pass off the input variables that were stated before to the function.

```
4.          *index = h( val, scatterTable->M, i ) ; //hash value
```

Now there are some conditions to end the loop. If there is nothing at the hash function index, then it is empty, and we can end the loop condition.

```
5.          if (  scatterTable->array[*index] == NULL ) //exit case: found location NOT val
6.          {
7.              found = 1 ;
8.          }
```

If the value stored at the hash function array index is found to be equal to the one being inserted, it will also end the loop condition, because then it will indicate the value is found. It will also assign a value of 1 to a variable called *inTable* which tells the program that the value is already in the scatter table.

```
9.          else if ( !strcmp( scatterTable->array[*index], val ) ) //exit case: found val
10.         {
11.             found = 1 ;
12.             inTable = 1 ;
13.         }
14.     } while ( !found ) ;
```

Now that we know how it works, we can write our real open address function.

```
1.  int h(char *val,int M,int i)
2.  {
```

We declare an integer variable to hold the index and return the index found from the hashing.

```
3.          int hVal  ;
```

We then pick our option for open addressing. **PICK ONE**. **f** is the first hash function, **i** is the number of times we've run this minus one, and **h2** is the second hash function for double hashing (if necessary).

```
1.      hVal = (f(val)%M + i*h2(val,M))%M ; //double hash
2.      hVal = (f(val) + i*i)%M ; //quadratic
3.      hVal = (f(val) + i)%M ; //linear
```

Finally, we return our hash value to the array.

```
4.      return( hVal ) ;
5.  }
```

4.3 Find

The find function is used by the main task as well as by many of the other functions including insert. The find function, as input, takes in the value we want to find, the scatter table, the index variable by reference (refresher: taking it by reference means that anything that updates in the function updates in the entire program), and the number of collisions variable by reference.

```
1.  int find(
2.          char *val,
3.          scatterTableType *scatterTable,
4.          int *index,
5.          int *nCollisions )
6.  {
```

We then declare a variable for the number of times the hash function has run (make it equal to -1 because at each iteration we want to add 1 and we want to start at 0), a **found** variable used to end the while loop, and an **inTable** variable to keep track of whether or not the value is found.

```
7.  int i=-1, found=0, inTable = 0 ;
```

We then set the number of collisions to be equal to zero at first.

```
8.      *nCollisions = 0 ;
```

We then run our check to make sure the scatter table isn't empty or full (i.e. size of the table is greater than number of elements in the table).

```
9.      if ( scatterTable->M > scatterTable->N ) //not empty
10.     {
```

We then run our do loop that was explained earlier in the section about **Open Addressing (4.2).**

```
11.                 //loop until null value found or match found within index
12.         do
13.         {
14.             i++ ; //tracks number of collisions.
15.             *index = h( val, scatterTable->M, i ) ; //hash value
16.             if (  scatterTable->array[*index] == NULL ) //exit case: found location NOT val
17.             {
18.                 found = 1 ;
19.             }
20.             else if ( !strcmp( scatterTable->array[*index], val ) ) //exit case: found val
21.             {
22.                 found = 1 ;
23.                 inTable = 1 ;
24.             }
25.         } while ( !found ) ;
```

The number of collisions is then set equal to i, which is the number of times we ran the loop minus 1 (to account for the first running).

```
26.         *nCollisions = i ;//How many indexes did we need to check until we found?
27.     }
```

Finally, we return the **inTable** variable to tell the other function if the find was successful or not.

```
28.     return( inTable ) ;
29. }
```

4.4 Insertion

Insertion takes, as inputs, the value we want to insert and the scatter table. It returns the number of collisions so we use an integer function.

```
1.  int insert( char *val, scatterTableType *scatterTable )
2.  {
```

We then declare variables to hold the index as well as the number of collisions.

```
3.          int index, nCollisions = 0 ;
```

We then check to see if the size of the scatter table is greater than the number of elements already inside the scatter table. This is to ensure the table isn't full and that the table isn't empty. This is the first part of the condition statement.

The second part of the condition statement is a little tricky. Our find function, returns a 1 or a 0 for **inTable** indicating if the find was successful or not. If the find was successful, then we can't insert, and we skip everything. If it isn't, then there is nothing stored at that index (remember index is passed by reference so the index found in the function comes back). This is why the condition is **!find**, because if the find function returns zero, then we're golden and we can keep going.

```
4.          if ( ( scatterTable->N < scatterTable->M )
5.                  && (!find( val, scatterTable, &index, &nCollisions ) ) )
6.          {
```

Here, it memory is allocated to the array index based on the value length plus one.

```
7.                  scatterTable->array[index] = (char *) malloc(strlen(val)+1) ;
```

It then copies the value using **strcpy** (built in) to the array index. The counter for the number of elements in the array (N) is increased by 1 to compensate for this.

```
8.                  strcpy( scatterTable->array[index], val ) ;
9.                  scatterTable->N++ ;
10.         }
```

Finally, we return the number of collisions.

```
11.         return( nCollisions ) ;
12. }
```