Dictionaries

Written by Rachel J. Morris, last updated Nov 13, 2018

About

Dictionaries are also known as **associative arrays**, **maps**, or **hash tables.** Dictionaries are a way we can store information in an array and use a mathematical function (a **hash function**) to find the index for any given data in the array.

Topics

1. Key-Value pairs

2. Hash function

3. Collision strategies
   - Linear Probing
   - Quadratic Probing
   - Double Hashing
   - Comparison

Key-value pairs

1. Key-value pairs

Dictionaries are often thought of as “key-value pairs”, where we have some data to store (this is the value), and we can look it up by its key (some searchable identifier).

Employee ID (Key)	Employee obj (Value)

`1024`	`Name: Amina`
	`Department: Dev`

`2048`	`Name: Benita`
	`Department: Sales`

`3650`	`Name: Claus`
	`Department: QA`

`1280`	`Name: Caiyun`
	`Department: QA`

Notes

1. Key-value pairs

The **key** for the data may be an integer, but it could also be any other data-type. Generally, the key is some unique identifier for the data.

Notes

1. Key-value pairs

Some examples:

- Word Counter – the key is a word (string), and the value is the amount of times that word has appeared in a document (integer).

Word	Count
“the”	4
“word”	3
“key”	2

- Organizer – The key is a date, and the value is an item due on that date.

Date	To do
2018-04-26	CS 211 project
2018-04-27	CS 250 project
2018-04-28	CS 235 exam

Notes

1. Key-value pairs

But we need some way to map a **key** to some **index** in an array. What position will the item be stored at?

This is where we use a **Hash Function**, a mathematical function to convert the key into an index.

Notes

Hash functions

2. Hash Functions

Let’s say we’re going to store employees in a Dictionary by their unique employee IDs. Employee IDs begin at 1000 (not 0 like the array index) and can be any 4 digit number. They’re not sequential.

We want to assign an array index to each employee ID so we can quickly search for an employee by their ID. For this, we need a hash function. For example, this could be one:

```
/*  Input: employee ID integer ... Output: array index */
int Hash( int employeeId )
{
    return employeeId % ARRAY_SIZE;
}
```

Notes

2. Hash Functions

When we’re writing a hash function, we will want to take into account how *likely* a collision will be. A general design rule is that making your **array size** a **prime number** helps reduce collisions.

You can read more about choosing a hash function here:
https://en.wikipedia.org/wiki/Hash_table#Choosing_a_hash_function

But for our project, we will just be using this simple hash:

```
int Hash( int key )
{
    return key % ARRAY_SIZE;
}
```

Notes

Collision Strategies

3. Collision strategies

In some cases, different **key** values may result in the same **index** value after being passed through the Hash Function. Because of this, we need a strategy to deal with these **collisions**.

Employee ID (Key)	Array Size	Hashed index
0	20	0
20	20	0 COLLISION
25	20	5
40	20	0 COLLISION

Notes

3. Collision strategies

Key: 0 Hash: 0 % 20 = 0 Index: 0

Linear Probing:
For a linear probe, all you do is move forward by +1 until you find the next available index.

Notes

Linear probing:
Move index forward by the same # each time.

3. Collision strategies

Key: 0	Hash: 0 % 20 = 0	Index: 0		
Key: 20	Hash: 20 % 20 = 0	Index: 0		COLLISION
		Linear:	0 + 1	
		Index:	1	

Linear Probing:
For a linear probe, all you do is move forward by +1 until you find the next available index.

Notes

Linear probing:
Move index forward by the same # each time.

3. Collision strategies

Key: 0	Hash: 0 % 20 = 0	Index: 0		
Key: 20	Hash: 20 % 20 = 0	Index: 0		**COLLISION**
		Linear:	0 + 1	
		Index:	1	
Key: 40	Hash: 40 % 20 = 0	Index:	0	**COLLISION**
		Linear:	0 + 1	
		Index:	1	**COLLISION**
		Linear:	1 + 1	
		Index:	2	

Linear Probing:
For a linear probe, all you do is move forward by +1 until you find the next available index.

Notes

Linear probing:
Move index forward by the same # each time.

3. Collision strategies

For Linear Probing, you can use any linear value; +1, +2, +3, +4, etc – we just move by the same amount each time.

Notes

Linear probing: Move index forward by the same # each time.

3. Collision strategies

Key: 0 Hash: 0 % 20 = 0 Index: 0

Quadratic Probing:
Move forward by the # of collisions squared each time.

Notes

Linear probing:
Move index forward by the same # each time.

Quadratic probing:
Move index by the # of collisions squared each time.

3. Collision strategies

Key: 0	Hash: 0 % 20 = 0	Index: 0		
Key: 20	Hash: 20 % 20 = 0	Index: 0		COLLISION
		Linear:	$0 + 1^2$	
		Index:	1	

Quadratic Probing:
Move forward by the # of collisions squared each time.

Notes

Linear probing:
Move index forward by the same # each time.

Quadratic probing:
Move index by the # of collisions squared each time.

3. Collision strategies

Key: 0 Hash: 0 % 20 = 0 Index: 0

Key: 20 Hash: 20 % 20 = 0

Index: 0		COLLISION
Linear:	$0 + 1^2$	
Index:	1	

Key: 40 Hash: 40 % 20 = 0

Index:	0	COLLISION
Linear:	$0 + 1^2$	
Index:	1	COLLISION
Linear:	$0 + 2^2$	
Index:	4	

Quadratic Probing:
Move forward by the # of collisions squared each time.

Notes

Linear probing:
Move index forward by the same # each time.

Quadratic probing:
Move index by the # of collisions squared each time.

3. Collision strategies

Example implementation:

```
int quadratic_probing_insert(int *hashtable, int key, int *empty)
{
    int i, index;
    for (i = 0; i < SIZE; i++) {
        index = (key + i*i) % SIZE;
        if (empty[index]) {
            hashtable[index] = key;
            empty[index] = 0;
            return index;
        }
    }
    return -1;
}
```

From https://en.wikipedia.org/wiki/Quadratic_probing

Notes

Linear probing: Move index forward by the same # each time.

Quadratic probing: Move index by the # of collisions squared each time.

3. Collision strategies

With **Double Hashing**, we have a second hash function that we use when there is a collision. This can be implemented in a few different ways.

In our project we will use the two hash functions:

```
int HashTable<TK,TD>::HashFunction( int key )
{
    return key % TABLE_SIZE;
}
```

```
int HashTable<TK,TD>::HashFunction2( int key )
{
    return 7 - ( key % 7 );
}
```

Notes

Linear probing:
Move index forward by the same # each time.

Quadratic probing:
Move index by the # of collisions squared each time.

Double hashing:
A second hash function is used during collisions.

3. Collision strategies

We get some *originalIndex* via the HashFunction call, and if there is a collision we add the result of

```
HashFunction2( originalIndex )
```

onto the new index each time, essentially giving us

```
collisionCount * HashFunction2( originalIndex );
```

```
int HashTable<TK,TD>::HashFunction( int key )
{
    return key % TABLE_SIZE;
}
```

```
int HashTable<TK,TD>::HashFunction2( int key )
{
    return 7 - ( key % 7 );
}
```

Notes

Linear probing: Move index forward by the same # each time.

Quadratic probing: Move index by the # of collisions squared each time.

Double hashing: A second hash function is used during collisions.

3. Collision strategies

So some code we can use in the GetIndex function is…

```
int originalIndex = index;
int collisions = 0;

while ( collision … ) {
     collisions++;
     index = ( HashFunction(key) + collisions * HashFunction2(key) )
          % TABLE_SIZE;
}
```

See also https://en.wikipedia.org/wiki/Double_hashing

```
int HashTable<TK,TD>::HashFunction( int key )
{
    return key % TABLE_SIZE;
}
```

```
int HashTable<TK,TD>::HashFunction2( int key )
{
    return 7 - ( key % 7 );
}
```

Notes

Linear probing:
Move index forward by the same # each time.

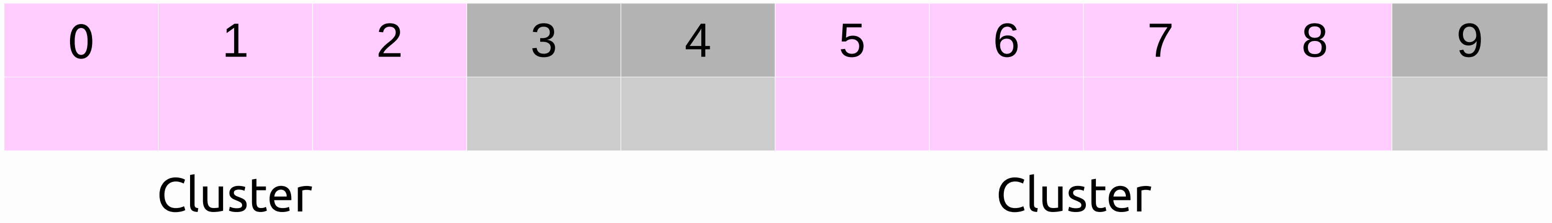
Quadratic probing:
Move index by the # of collisions squared each time.

Double hashing:
A second hash function is used during collisions.

3. Collision strategies

Comparison:

With **Linear Probing**, it is simple to implement but items will tend to clump together. Since we move over by the same amount each time, this **clustering** means it will take longer to find an empty spot each time there's a collision.



Notes

Linear probing:
Move index forward by the same # each time.

Quadratic probing:
Move index by the # of collisions squared each time.

Double hashing:
A second hash function is used during collisions.

3. Collision strategies

Comparison:

With **Quadratic Probing**, it is more resistant to clustering, though it still might occasionally occur.

With **Double Hashing**, the amount of spaces we move depends on the data itself, rather than some linear or quadratic function.

Notes

Linear probing: Move index forward by the same # each time.

Quadratic probing: Move index by the # of collisions squared each time.

Double hashing: A second hash function is used during collisions.

Conclusion

Hashes are used in more than just data structures, and hash tables are used in low-level operating system procedures as well.

Dictionaries are pretty common in many programming languages and can be very useful to use, so it is good to know how they work.