

Hashing

Dr. Anirban Ghosh

School of Computing
University of North Florida



More on implementing maps

- Arrays and Lists? Too inefficient! Linear runtimes
- Plain BSTs? Too inefficient! Linear runtimes; $O(n)$ for search, insert, delete
- RB-trees? Good choice! Logarithmic runtimes are guaranteed; $O(\log n)$ for search, insert, delete

Can we do better in practice?

Answer

HASH TABLES

☞ With hash-tables, it is not possible to obtain a sorted sequence of the records in $O(n)$ time like BSTs by simply traversing it

Warm up (the simplest possible case)

INDEX/KEY	0	1	2	3	4	5	6	7	8	9	10
Value	<i>D</i>		<i>Z</i>			<i>C</i>	<i>Q</i>				

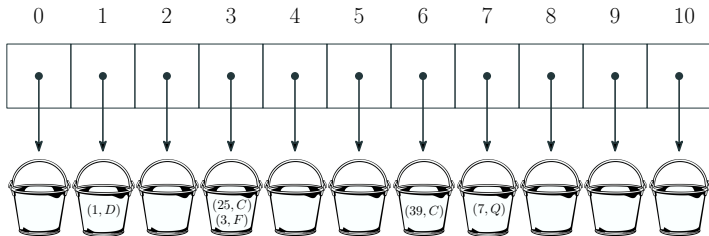
Assume that we have $n \leq 11$ records to maintain where the keys are in the range $[0, 10]$
Insertions, deletions, look-ups can be executed in $O(1)$ time each since an array of length $N = 11$ can be used for implementing the table

The situation

- Let n be the number of records stored and N be the number of possible keys
- What if N is really large, say in the order of millions and n much less than N ?
- **Example.** for integer keys, $N = 2^{31} - 1 = 2,147,483,647$; but, n in most cases is much less than $2^{31} - 1$. Are we still going to use an array of size 2,147,483,647? Probably not a good idea. Space wastage may be severe. Storing such an array will require $(2^{31} - 1) \times 1 \text{ byte} \approx 2.147 \text{ GB}$ of space (assuming 1 byte is enough to store a character)!

A space-efficient solution

Map the keys to the set of array indices using some function (a.k.a. hash function). A bucket (singly linked-list, for instance) is present at every index that can hold more than one records. Note that In this setup, we can store more than N records inside the table since we are using buckets.



These buckets have theoretically *infinite* capacities

Let $h(\text{key}) = \text{key} \bmod 11$

For any key k , $0 \leq h(k) \leq 10$

The record (k, v) is put in the bucket at index $h(k)$

What if the keys are not integers?

- 1 Convert the non-integer key to an integer using some function h_1 ; after applying h_1 , we get an integer $h_1(k)$; the function h_1 is known as the **hashcode**
- 2 Next, map $h_1(k)$ to an array index using another function h_2 known as the **compression function**

The record (k, v) maps to the index $h_2(h_1(k))$

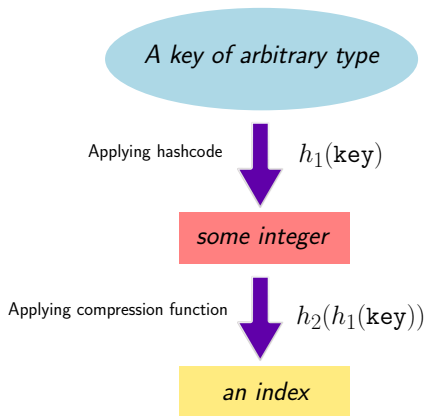
Example

$$h_1(\text{"Doctor Strange"}) = 1938383$$

$$h_2(1938383) = 1938383 \bmod N, \text{ where } N \text{ is the size of the array used}$$

👉 In the previous example, $N = 11$

The hashcode and the compression function



Hash function

$$h = h_2(h_1(k))$$

The array plus the hash function is called **hash-table**

The main idea behind hashing

It is *unlikely* that two different records (k_i, v_i) and (k_j, v_j) , where $k_i \neq k_j$ will map to the same bucket in the hash table when the size of the table N is sufficiently large and the hash function is chosen appropriately. Consequently, it is unlikely that the buckets will be crowded.

If the buckets are not crowded, *searches*, *insertions*, and *deletions* would run fast in practice.

Hashcodes

- Based on the type of keys we are using, one can design various kinds of hash-codes. Desired properties of hash codes:

☞ If two keys k_i, k_j are different, then the two corresponding outputs of hashcode should be different

$$k_i \neq k_j \implies h_1(k_i) \neq h_1(k_j)$$

☞ Should be very fast to compute

- In Java, the **Object** class (super-class of every Java class) defines the **hashCode()** method using the object's memory address
- This means the **hashCode()** method can be invoked on any object using the dot operator!
- If two objects are equal according to the optional **equals** method of a certain class, then separately invoking the **hashCode()** method on each of the two objects must produce the same integer result.

Illustration

Java's hashCode for Strings

Let $s = s_0s_1 \dots s_{n-1}$, where every s_i is a character

$$h_1(s) = (\text{ASCII}(s_0) \times 31^{n-1}) + (\text{ASCII}(s_1) \times 31^{n-2}) + \dots + (\text{ASCII}(s_{n-1}) \times 31^0)$$

```
public class HashCodeDemo {  
    public static void main(String[] args) {  
        String s1 = "UNF is FUN";  
        String s2 = "FUN is UNF";  
        String s3 = "UNF iss FUN";  
  
        System.out.print(s1.hashCode() + " ");  
        System.out.print(s2.hashCode() + " ");  
        System.out.print(s3.hashCode());  
    }  
}
```

Output

120001564 63052472 -499412747

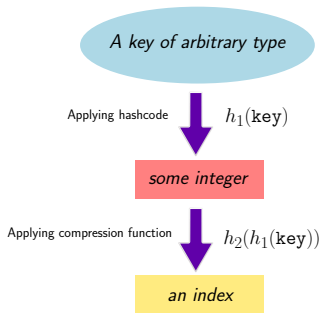
- The built-in Java classes such as String, Integer, Double, etc. redefine this function; see Java's documentation to see the precise mathematical functions

```
Double d1 = 101.98;  
System.out.print(d1.hashCode() + " ");  
  
d1 = 101.981;  
System.out.print(d1.hashCode() + " ");  
  
d1 = -101.981;  
System.out.print(d1.hashCode() + " ");
```

Output

```
296942503 -195025192 1952458456
```

Compression function



The popular compression function

$$h_2(x) = x \bmod N$$

where N is the size of the table.

👉 Make sure that $x \geq 0$, otherwise, $h_2(x) < 0$, making it useless for array indexing!

What is a collision?

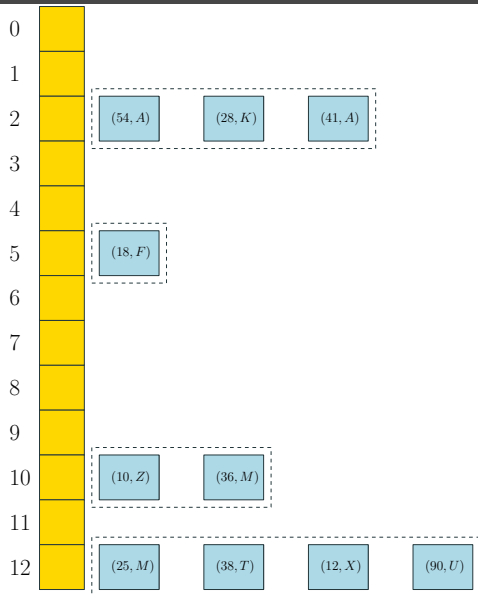
We say that two records (k_i, v_i) and (k_j, v_j) are said to have **collided** if $h(k_i) = h(k_j)$. This means both got mapped to the same array index.

What to do if multiple records map to the same index?

- Choice 1: **Separate chaining** (uses buckets, very popular); number of records stored can be much more than the size of the array
- Choice 2: **Open addressing** (does not use buckets, less popular); number of records stored cannot be more than the size of the array

Separate Chaining

Separate chaining: use a container (linked-list/ArrayList) at every index



Hash-function used in this example

$$h(x) = x \bmod 13$$

Load-factor of hash-tables

$$\lambda = \frac{\text{the number of records present in the map } (n)}{\text{size of the array } (N)}$$

☞ $\lambda < 1$ is always desired. This implies no container is probably overcrowded, resulting in fast operation speeds. In Java's implementation of hash-tables, by default, load-factors are never allowed to exceed **0.75**. If the load-factor exceeds 0.75, the size of the array is increased and all the records present in the map are re-inserted (every one of them). We are allowed to choose a different value for the load-factor.

☞ In the beginning, when the table is empty, the number of buckets is set to **16** in Java's implementation of hash-tables.

put(k, v): inserts the record (k, v) into the map

- 1 $i = h(k);$
- 2 Let L be the container present at index i ;
- 3 Check L to see if a record is already present in L having key k ;
- 4 If such a record is present in L , the new record (k, v) cannot be inserted;
- 5 Otherwise, insert the record into L ;
- 6 Compute the current load factor λ ;
- 7 If λ exceeds **0.75**, execute **rehash()**;

Takes $O(n^2)$ time in the worst-case (when rehashing is needed). If rehashing is not required, it takes $O(n)$ time in the worst-case and $O(1)$ time on average.

☞ After rehashing, $\lambda \leq 0.75$

rehash()

rehash(): rehashes the map (an internal method)

- 1 Let N be the size of the current array A ;
- 2 Create a new array of empty containers A' whose size is $2N$;
- 3 Insert every record currently present in the map into the appropriate container in A' using the new compression function $h_2(x) = x \bmod 2N$ (before rehashing, it was $h_2(x) = x \bmod N$) but using the old hashcode h_1 (since the type of keys did not change). This step requires n `put(k, v)` operations.

☞ Rehashing runs in $O(n^2)$ time in the worst case where n is the number of records currently present in the map

Rehashing illustration

Bucket 0: [96, Jim] -> [112, Peter]
Bucket 1:
Bucket 2:
Bucket 3: [99, Jack]
Bucket 4: [36, Rose]
Bucket 5:
Bucket 6: [22, Charles]
Bucket 7: [23, Alice]
Bucket 8: [40, Bob]
Bucket 9: [41, Matthew]
Bucket 10: [10, Tom]
Bucket 11: [11, Dorothy]
Bucket 12: [92, Eric]
Bucket 13:
Bucket 14: [62, Donald]
Bucket 15:

Bucket 0: [96, Jim]
Bucket 1:
Bucket 2:
Bucket 3: [99, Jack]
Bucket 4: [36, Rose]
Bucket 5:
Bucket 6:
Bucket 7:
Bucket 8: [40, Bob]
Bucket 9: [41, Matthew]
Bucket 10: [10, Tom]
Bucket 11: [11, Dorothy]
Buckets 12 -to 15:
Bucket 16: [112, Peter]
Buckets 17 to 21:
Bucket 22: [22, Charles]
Bucket 23: [23, Alice]
Buckets 24 to 27:
Bucket 28: [92, Eric]
Bucket 29:
Bucket 30: [62, Donald]
Bucket 31: [31, Toby]

$$\text{Load-factor } (\lambda) = \frac{n}{N} = \frac{12}{16} = 0.75$$

After inserting [31, Toby], $\lambda = \frac{n}{N} = \frac{13}{16} = 0.8125 > 0.75$;
rehashing is required; new $\lambda = \frac{n}{N} = \frac{13}{32} \approx 0.40625 \leq 0.75$

`get(k)`: returns the value part of the record whose key is k

- 1 $i = h(k)$;
- 2 Let L be the container present at index i ;
- 3 Check L to see if a record is already present in L having key k ;
- 4 If such a record is present in L , return its value part;
- 5 Otherwise, return **null**;

Takes $O(n)$ time in the worst-case (when most records map to the same bucket)
but in practice takes $O(1)$ time on average

`remove(k)`

`remove(k)`: removes the record having key k , if present

- 1 $i = h(k)$;
- 2 Let L be the container present at index i ;
- 3 Check L to see if a record is present in L having key k ;
- 4 If it is present in L , remove and return the value part of the record (k, v) ;
- 5 Otherwise, return **null**;

Takes $O(n)$ time in the worst-case (when most records map to the same bucket)
but in practice takes $O(1)$ time on average

See the class `HashMapSeparateChaining`

In our implementation, the containers
are **singly linked-lists**

Worst-case scenario of separate chaining

- In the worst case, most of the records can map to the same index!
- Thus, insertion, deletion, and searching take $O(n)$ time in the worst-case if lists are used and $O(\log n)$ time if RB-trees are used
- In such cases, we do not get any advantage out of hashing
- However, if we are using good hash function, these extreme situations will almost never happen and we get super-speedy performance in the real-world

Java's HashMap vs TreeMap: insertion time comparison

n	<code>java.util.HashMap</code>	<code>java.util.TreeMap</code>
10	1	1
100	1	1
1000	1	3
10000	1	7
100000	17	19
1000000	97	167
10000000	815	2889

👉 Times are reported in milliseconds

👉 In practice, HashMaps are more efficient in practice than TreeMaps

Open Addressing

Open addressing

- Use an array of size N
- At every index, we can store at most one record (no use of infinite capacity buckets)
- Let us look at such a popular technique known as **LINEAR PROBING**: if the spot is already occupied by some other record (collision), we consider the next available spot in the array by wrapping around

Example

0	1	2	3	4	5	6	7	8	9	10

Insert (13, A)

0	1	2	3	4	5	6	7	8	9	10
		(13, A)								

$$13 \bmod 11 = 2$$

Insert $(26, T)$

0	1	2	3	4	5	6	7	8	9	10
		$(13, A)$		$(26, T)$						

$$26 \bmod 11 = 4$$

Insert (5, Z)

0	1	2	3	4	5	6	7	8	9	10
		(13, A)		(26, T)	(5, Z)					

$$5 \bmod 11 = 5$$

Insert $(37, G)$; collision!

0	1	2	3	4	5	6	7	8	9	10
		(13, A)		(26, T)	(5, Z)	(37, G)				

$37 \bmod 11 = 4$, linear probe and then put the record at index 6

Insert (16, A); collision!

0	1	2	3	4	5	6	7	8	9	10
		(13, A)		(26, T)	(5, Z)	(37, G)	(16, A)			

$16 \bmod 11 = 5$, linear probe and then put the record at index 7

Insert $(21, F)$

0	1	2	3	4	5	6	7	8	9	10
		(13, A)		(26, T)	(5, Z)	(37, G)	(16, A)			(21, F)

$$21 \bmod 11 = 10$$

Insert $(43, Q)$; collision!

0	1	2	3	4	5	6	7	8	9	10
$(43, Q)$		$(13, A)$		$(26, T)$	$(5, Z)$	$(37, G)$	$(16, A)$			$(21, F)$

$43 \bmod 11 = 10$, linear probe by wrapping around and then put the record at index 0

Operations

In the following, **DEFUNCT** is a special record (`null, null`) whose key and value are set to `null`

- 1 **Insertion.** Use the hash function on the key to obtain the target index; if the spot is taken by some other record, use wrap-around linear probing to find the next available spot; cells containing **DEFUNCT** are considered to be empty
- 2 **Searching.** Use the hash function on the key to obtain the target index; if the desired record is not present at the index, do a wrap-around linear probe to search the record. If an empty spot is encountered, the desired record is not present. However, skip over the cells containing **DEFUNCT**. In this case, cells containing **DEFUNCT** are not considered to be empty.
- 3 **Deletion.** Use the hash function on the key to obtain the target index; if the desired record is not present at the index, use wrap-around linear probing to locate the record. However, skip over the cells containing **DEFUNCT**. In this case, cells containing **DEFUNCT** are not considered to be empty. If located, delete the record from the cell and put a special symbol **DEFUNCT** in its place (see the example next to see why)

All these three operations takes $O(n)$ time in the worst-case

Why DEFUNCT is necessary?

0	1	2	3	4	5	6	7	8	9	10
(43, Q)		(13, A)		(26, T)	(5, Z)		(16, A)			(21, F)

(37, G)
deleted

0	1	2	3	4	5	6	7	8	9	10
(43, Q)		(13, A)		(26, T)	(5, Z)	DEFUNCT	(16, A)			(21, F)

(37, G)
deleted

Search for (16, A) will fail if we store null at index 6. For this reason, putting a special marker **DEFUNCT** will help the search algorithm to find (16, A) by skipping over index 6.

Observations

- If we are using open addressing, the number of records that can be present in the array cannot exceed the array size since we are not using chaining
- But it is not the case in separate chaining
- Separate chaining consumes more space to maintain the containers at every index

Chapter 15 from

<https://opensa-server.cs.vt.edu/ODSA/Books/Everything/html/index.html>