Hashing

Dr. Anirban Ghosh

School of Computing
University of North Florida

- Arrays and Lists? Too inefficient! Linear runtimes
- Plain BSTs? Too inefficient! Linear runtimes
- RB-trees? Good choice! Logarithmic runtimes are guaranteed

Can we do better?

Answer

Theoretically NO but in practice YES!

HASH TABLES

Warm up (the simplest possible case)

Index/Key	0	1	2	3	4	5	6	7	8	9	10
Value	D		Z			C	Q				

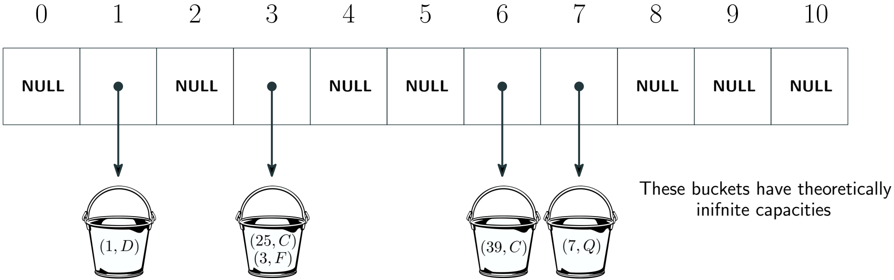
Assume that we have $n = 11$ records to work with and the keys are in the range $[0, 10]$

Insertions, deletions, look-ups can be executed in $O(1)$ time each

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 is 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 $(2^{31} - 1) \times 4$ bytes ≈ 8.7 GB of space!

Map the keys to the set of array indices using some function (a.k.a. hash function). Every index can hold more than one records (a bucket of records).



$$h(\textbf{key}) = \textbf{key} \bmod 11$$

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 $h_2(h_1(k))$

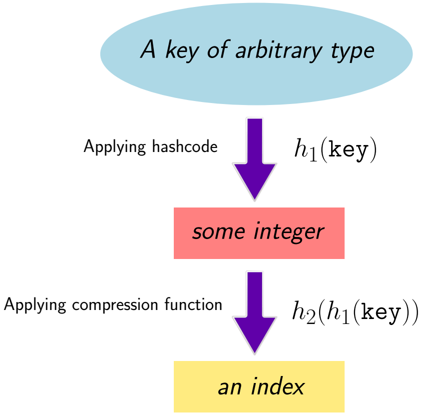
Example

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

$h_2(1938383) = 1938383 \bmod p$, where p is the size of the array

☞ In the previous example, $p = 11$

The hashcode and the compression function



Hash function

$h = h_2(h_1(k))$

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

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!
- If two objects are equal according to the optional `equals` method of the class, then calling the `hashCode` method on each of the two objects must produce the same integer result.

Illustration

Java's hashcode for Strings

Let $s = s_0 s_1 \ldots s_{n-1}$, where every s_i is a character

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

```java
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
```