CSE 12 – Basic Data Structures and Object-Oriented Design
Lecture 16

Greg Miranda & Paul Cao, Winter 2021

This lecture is being recorded

Announcements

- Quiz 16 due Wednesday @ 8am
- Survey 6 due tonight @ 11:59pm
- PA6 due Wednesday @ 11:59pm
- Practice Map Problems – see schedule

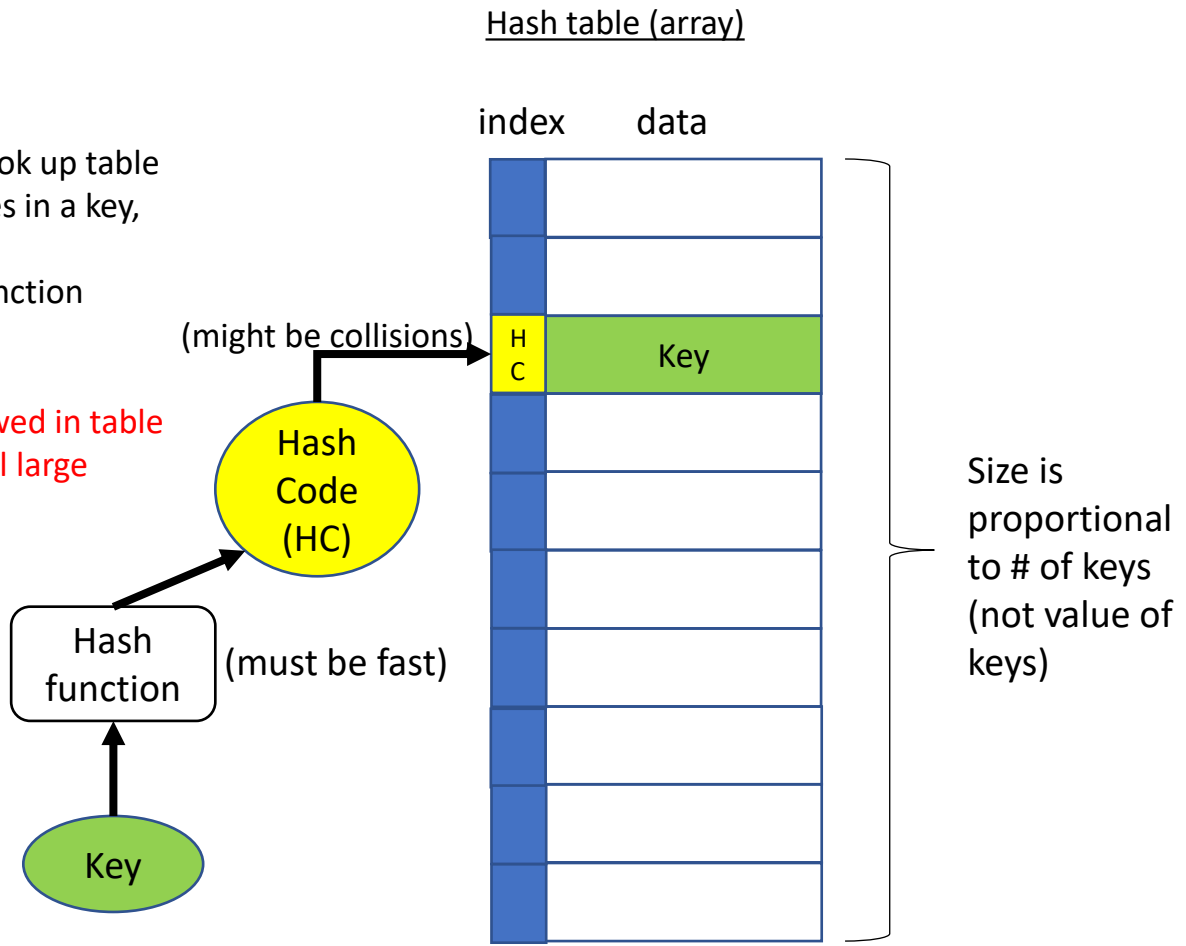
Topics

- Map and HashTable
- Questions on Lecture 16?

Hashing

- Let's modify our array-based look up table
- Need a hash-function h(x): takes in a key, returns an index in the array
- gold standard: random hash function

- In general, no null value is allowed in table
- Table size is fixed and in general large



hashCode

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

As much as is reasonably practical, the hashCode method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)

Returns:

`a hash code value for this object.`

See Also:

`equals(java.lang.Object), System.identityHashCode(java.lang.Object)`

Generates a number that can be used by a hash function (or simply as the hash value itself)

In general, what is the difference between the value returned by hashCode() and the index location where the item ends up in a particular hash table?

A. Nothing. The value returned by hashCode can be used directly as the index for the item in any hash table
B. The value returned by hashCode might be larger than the size of the hash table
C. The hashCode function might return the same value for two different objects, and indexes in hash tables must always be unique for different objects
D. The hashCode function might return different values for two objects that are considered equal (and for hash tables, two values that are considered equal must have the same hashcode/index value)

hashCode

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

As much as is reasonably practical, the hashCode method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)

Returns:

a hash code value for this object.

Hash functions

- A hash function maps an object or key to a position in an array (the hash table). What properties are required and/or desired from a good hash function?

Hash function

- To be useful, a hash function must be ***fast***
  - Its performance should not depend on the particular key.
  - Runs in "constant time" (more on this later…)

- A hash function must also be ***deterministic****:*
  - Given the *same value,* it must *always* return the *same array index.* (Otherwise, how would we find something we stored earlier?)

- A “good” hash function should also be **uniform**:
  - Each “slot” *i* in the array should be equally likely to be chosen as any other slot *j.*

Hashing Is Cool

- Probably most used and important data structure in CSE
- Probably in every system you've ever used
- Part of every modern programming language
- Remember it's derived from Dictionary ADT
  - Key, value pair (insert, delete, and search)
  - Does the key exist, if so, give me the item associated with that key
  - Assuming no two items have the same key

Collision Resolution Strategies

Collision resolution: Separate Chaining

- Insert the numbers below into a hash table of size M=11 using the hash function H(k)=k % M. Perform each of the following:

1. Separate chaining collision resolution without resizing What is the load factor?
2. Separate chaining collision resolution WITH resizing with the 8^{th} element is inserted (double M to 22). What is the final load factor?

{22, 32, 43, 11, 12, 35, 24, 7}

{22, 32, 43, 11, 12, 35, 24, 7}

Collision resolution: Separate Chaining

- The idea is to keep a list of all elements that hash to the same value.
  - The array elements are pointers to the first nodes of the lists.
  - A new item is inserted to the front of the list.
    - Why the front? O(1) insertion
- Advantages
  - Better space utilization for large items.
  - Simple collision handling: searching linked list.
  - Overflow: we can store more items than the hash table size.
  - Deletion is quick and easy: deletion from the linked list.
- Disadvantage
  - Complicated data structure. Array of link list.

Analysis of Separate Chaining

- Load factor definition
  - Ratio of number of elements (M) in a hash table to the hash TableSize.
    - Load factor = M/TableSize.
  - The average length of a list is also a load factor.

Table size as prime number

- Table size as a prime number helps minimize clustering
- Doubling the table size using prime numbers.
  - https://www.planetmath.org/goodhashtableprimes

Strings as keys

- Keys are not always integers.

- The classic example is a dictionary. If you want to put every word of an English-language dictionary, from *a* to *zyzzyva*
  - *(who knows what it is and how to pronounce it?*)

  into your computer's memory so they can be accessed quickly, a hash table is a good choice.

- A similar widely used application for hash tables is in computer-language compilers, which maintain a *symbol table* in a hash table.
  - The symbol table holds all the variable and function names made up by the programmer, along with the address where they can be found in memory.

- https://en.wikipedia.org/wiki/Symbol_table

Converting words to numbers

- Let's say we want to store **a 50,000**-word English-language dictionary in main memory

Question: Use the length of a word as a hash function:

A: It is a good choice

B: It is not uniform

C: It is not fast

D: It is not deterministic

Converting words to numbers (50,000-words)

- Another idea: Use ASCII code, in which *a* is 97, *b* is 98, and so on, up to 122 for *z.* Then **add** the numbers.
- Let's say *a* is 1, *b* is 2, *c* is 3, and so on up to 26 for *z.* We'll also say a blank is 0, so we have 27 characters. (assume no capitals).
  - *I subtracted 96 to make the math easier.*

- "cat"= 3 + 1 + 20 = 24.

A: It is a good choice

B: It is not uniform

C: It is not fast

D: It is not deterministic

Converting words to numbers (multiplying powers)

Idea: Hash each word into a unique location.

How? Number analogy:

$7{,}546 = 7*1000 + 5*100 + 4*10 + 6*1$

(or using powers of 10):

$7{,}546 = 7*10^3 + 5*10^2 + 4*10^1 + 6*10^0$

Why base 10? Because there are **10** possible digits.

Converting words to numbers (multiplying powers)

- Why base 10? Because there are 10 possible digits.

- If we are apply the same idea to strings, what is our base?

A: 10

B: 27 (26 letters and the space)

C: Can’t apply the method

Converting words to numbers (multiplying powers)

- We will use base 27 because we have 26 letters and 1 space.
- “cat”: c = 3, a = 1, t = 20.

- What is the corresponding conversion?

$H(cat) = 3*27^2 + 1*27^1 + 20*27^0$

Converting words to numbers (multiplying powers)

- The largest 10-letter word (let's assume 10 is the max length of the word), *zzzzzzzzzz,* translates into …

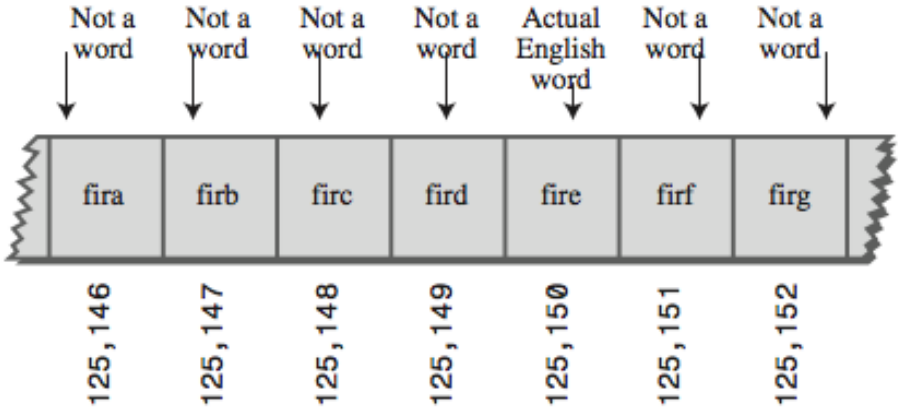
- ?

Converting words to numbers (multiplying powers)

- The largest 10-letter word (let's assume 10 is the max length of the word), *zzzzzzzzzz,* translates into …

- $26*27^9 + 26*27^8 + 26*27^7 + 26*27^6 + 26*27^5 + 26*27^4 + 26*27^3 + 26*27^2 + 26*27^1 + 26*27^0$ ←HUGE NUMBER.

- Another problem:



Conclusion

- Our first scheme—adding the numbers—generated too few indices.
- This latest scheme—adding the numbers times powers of 27—generates too many.
- Hash Tables have large but reasonable size. We will use the second approach with some changes.
- $\texttt{key = 3*27}^{2}\texttt{ + 1*27}^{1}\texttt{ + 20*27}^{0}$
- `index = (key) % tableSize;`

Hash Strings: “cat”

$hashVal = 3*27^2 + 1*27^1 + 20*27^0$

```
public static int hashFunc1(String key) {
  int hashVal = 0;
  int pow27 = 1;      // 1, 27, 27*27, etc

  for (int j=key.length()-1; j>=0; j--) { // right to left
    int letter = key.charAt(j) - 96;      // char code
    hashVal += pow27*letter;              // times the power of 27
    pow27 *= 27;                          // Increase the power
  }

  return hashVal % tableSize;      // mod by table size
}
```

Not efficient yet

- Aside from the character conversion, there are **two** multiplications and an **addition** inside the loop

- **Horner's Method.**

$$var4*n^4 + var3*n^3 + var2*n^2 + var1*n^1 + var0*n^0$$

can be written as

$$(((var4*n + var3)*n + var2)*n + var1)*n + var0$$

Another version (Horner's method)

```
public static int hashFunc2(String key) {
  int hashVal = key.charAt(0);
  int pow27 = 1;       // 1, 27, 27*27, etc

  for (int j=1; j<key.length(); j++) {    // left to right
    int letter = key.charAt(j) - 96;      // char code
    hashVal = hashVal*27 + letter;        // multiply and add
  }

  return hashVal % tableSize;       // mod by table size
}
```

There's still at least one problem for long strings...

$var4*n^4 + var3*n^3 + var2*n^2 + var1*n^1 + var0*n^0$

can be written as

$(((var4*n + var3)*n + var2)*n + var1)*n + var0$

Final improvement

```
public static int hashFunc_final(String key) {
  int hashVal = key.charAt(0);
  int pow27 = 1;      // 1, 27, 27*27, etc

  for (int j=1; j<key.length(); j++) {    // left to right
    int letter = key.charAt(j) - 96;      // char code
    hashVal = (hashVal*27 + letter) % tableSize; // multiply and add
  }

  return hashVal;
}
```

Even faster

- Various bit- manipulation tricks can be played as well, such as

  using a base of 32 (or a larger power of 2) instead of 27, so that multiplication can be effected using the shift operator (>>),

Folding: another idea

- For larger strings: (length over 7)
  - Group more than one digit. (2, 3, 4).
  - Add them.
  - % tableSize.

Folding Example

Let's start by converting the string's characters into numbers. ASCII is a good candidate for this operation:

J	a	v	a		l	a	n	g	u	a	g	e

74	97	118	97	32	108	97	110	103	117	97	103	101

Now, we arrange the numbers we just obtained into groups of some size. Generally, we choose the group size value based on the size of our array which is 10^5. Since the numbers, in which we transformed the characters into, contain from two to three digits, without loss of generality, we may set the group size to two:

74	97	118	97	32	108	97	110	103	117	97	103	101	

The next step is to concatenate the numbers in each group as if they were strings and find their sum:

7497	11897	32108	97110	103117	97103	101

$$7497 + 11897 + 32108 + 97110 + 103117 + 97103 + 101 = 348933$$

Now we must make the final step. Let's check whether the number *348933* may serve as an index of our array of size 10^5. Naturally, it exceeds the maximum allowed value *99999*. We may easily overcome this problem by applying the modulo operator in order to find the final result:

```
348933 % 10000 = 48933
```

More ideas

- A few word documents with different hash function ideas. There are a few implementations in C++ so you can convert to Java. (credit goes to Geoff Kuenning)

  Statistics:

  http://programmers.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed

Hash Table – draw the picture (Separate Chaining)

```
int getIndex(String k) {
    return k.length;
}
```

# of buckets – 6

(i.e. the size of the array)

```
set("Smith", 1);
set("Maria", 2);
set("Christine", 3);
set("Brown", 4);
set("Julia", 5);
set("Garcia", 6);
set("Miller", 7);
set("Davis", 8);
set("Wesley", 9);
set("Martinez", 10);
```

Hash Table – draw the picture (Separate Chaining)

```
int getIndex(String k) {
    return k.length;
}
```

# of buckets – 4
(i.e. the size of the array)

expandCapacity() called in set()
LoadFactor – 0.75

```
set("Smith", 1);
set("Maria", 2);
set("Christine", 3);
set("Brown", 4);
set("Julia", 5);
set("Garcia", 6);
set("Miller", 7);
set("Davis", 8);
set("Wesley", 9);
set("Martinez", 10);
```

Questions on Lecture 16?