

Introduction to Computation and Problem Solving

Class 31: Lab: Hashing

Prof. Steven R. Lerman
and
Dr. V. Judson Harward

Motivation

- Can we search in better than $O(\lg n)$ time?
- The operation of a computer memory does considerably better than this. A computer memory takes a key (the memory address) to insert or retrieve a data item (the memory word) in constant ($O(1)$) time.
- Access times for computer memory do not go up as the size of the computer memory or the proportion used increases.

Direct Addressing

- Computer memory access is a special case of a technique called *direct addressing* in which the key leads directly to the data item.
- Data storage in arrays is another example of direct addressing, where the array index plays the role of key.
- The problem with direct addressing schemes is that they require storage equal to the range of all possible keys rather than proportional to the number of items actually stored.

3

Direct Addressing Example

- Let's use the example of social security numbers.
- A direct addressing scheme to store income information on US tax payers would require a table of 1,000,000,000 entries since a social security number has 9 digits.
- It doesn't matter whether we expect to store data on 100 tax payers or 100,000,000.
- A direct addressing scheme will still require a table that can accommodate all 1 billion potential entries.

4

Hashing

- Hashing is a technique that provides speed comparable to direct addressing ($O(1)$) with far more manageable memory requirements ($O(n)$, where n is the number of entries actually stored in the table).
- Hashing uses a function to generate a pseudo-random *hash code* from the object key and then uses this *hash code* (~direct address) to index into the *hash table*.

5

Hashing Example

- Suppose that we want a small hash table with a capacity of 16 entries to store English words.
- Then we will need a hash function that will map English words to the integers 0, 1, ..., 15.
- We usually divide the task of creating a hash function into two parts:
 1. Map the key into an integer.
 2. Map the integer "randomly" or in a well distributed way to the range of integers ($\{0, \dots, m-1\}$, where m is the capacity or number of entries) that will be used to index into the hash table.

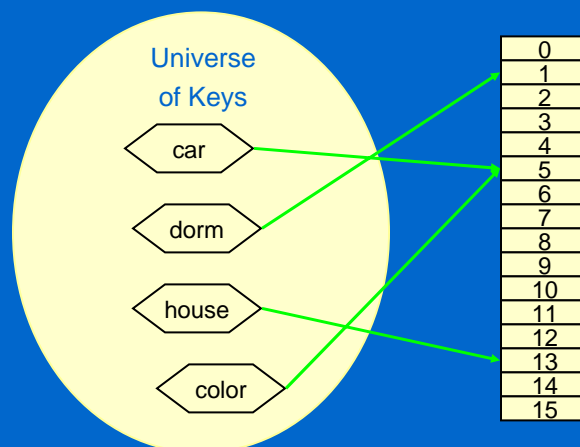
6

Hash Code Example

- As an example, consider the hash code that takes the numeric value of the first character of the word and adds it to the numeric value of the last character of the word (*step 1*), then takes the remainder *mod 16* (*step 2*).
- For instance, the numeric value of a "c" is 99 and of a "r" is 114. So, "car" would hash to $(99 + 114) \bmod 16 = 5$.

7

Hash Code Diagram



8

Collisions

- "car" and "color" hash to the same value using this hash function because they have the same first and last letter. Our hash function may not be as "random" as it should be.
- But if n (# of stored items) $>$ m (the number of storage slots), duplicate hash codes, otherwise known as *collisions*, are inevitable.
- In fact, even if $n < m$, collisions may still be likely as a consequence of *von Mises's* argument (also known as the birthday paradox: if there are 23 people in a room, the chance that at least two of them have the same birthday is greater than 50%).

9

Hashing Tasks

1. Designing an appropriate hash function to assign hash codes to keys in such a way that a non-random set of keys generates a well-balanced, "random" set of hash codes;
 If the hash codes aren't random, excessive collisions will "clump" the keys under the same direct address.
2. Coping with any collisions that arise after hashing.

10

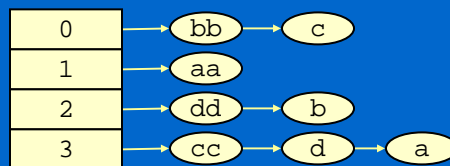
Chaining to Avoid Collisions

- *Chaining* is a simple and efficient approach to managing collisions.
- In a hash table employing chaining, the table entries, usually known as slots or buckets, don't contain the stored objects themselves, but rather linked lists of objects.
- Objects with colliding keys are inserted on the same list.
- Insertion, search, and deletion become 2 step processes:
 1. Use the hash function to select the correct slot.
 2. Perform the required operation on the linked list that is referenced at that slot.

11

Chaining Illustration

keys = { a, b, c, d, aa, bb, cc, dd }



12

Load Factor and Performance

- The ratio of the number of items stored, n , to the number of table slots, m , n/m , is called the table's *load factor*.
- Because the linked lists referenced by the hash slots can accommodate an arbitrary number of elements, there is no limit on the capacity of a hash table that employs chaining.
- If the hash function employed does not distribute the keys well, the performance of the table will degrade.
- The worst case for a hash table as for a binary search tree is that of the linked list. This occurs when all the keys hash to the same slot.
- Given a good hash function, however, it can be proved that a hash table employing chaining with a load factor of L can perform the basic operations of insertion, search, and deletion in $O(1 + L)$ time.
- For efficiency, keep load factor ≤ 0.75

13

Hash Table Iterators

- Iterating over a hash table is also a two step process.
For every slot containing a linked list, hlist
For every item, i, in hlist
Return i
- Because the hash codes assign items to slots "randomly" and we are not ordering the linked lists, such an iteration has no order.
- Order and locality is one of the characteristics that one trades for speed of access in hash tables.

14

Iterator Efficiency

- The time to iterate over a table with few collisions is proportional to the capacity of the table or the number of slots, not the number of items stored.
- It will take roughly as long to iterate over a large capacity table with few items as one in which the number of items stored approaches the number of slots.
- Traditional applications of hash tables do not require ordered access or the ability to iterate.

15

Typical Hash Table Applications

- One of the classic uses of hash tables is to manage the symbol table for interpreters and compilers. Symbol names (e.g, variable names) are the key, and symbol data, e.g., type, location, etc, are contained in the object stored. Fast lookup is the main need.
- If an ordered listing of the symbol table is required (and these days it hardly ever is) it can be provided in a second pass.
- Other hash table uses typically involve managing a class of data by an arbitrary key such as social security number, account number, or ID number.

16

Hash Functions

- Effective hash functions are crucial for efficient hash table implementation.
- We often want to store keys that are anything but random. Think of storing people's names, last name first. In America there will be many keys that start off "Smith, ..." and many keys that end "..., John". We would like a hashing function for names to be as likely to distribute the many "Smith" entries in separate hash slots as it would be to separate "Smith, John" from say, "Harward, Judson".

17

hash₁ and *hash₂* functions

- Let's assume that we want to store objects from a universe U into a hash table with m slots.
- Formally, we require a hash function, $hash()$, that maps each object $k \in U$ to an integer h , $0 \leq h < m$, or less formally, $hash()$ must map each k to its slot.
- We have already mentioned that a hash function can often be considered as the composition of two functions: $hash_1: U \rightarrow I$ and $hash_2: I \rightarrow \{h \in I \mid 0 \leq h < m\}$, where I is the set of integers.

18

hashCode ()

- In an object-oriented language like Java, the first phase of hashing, the *hash*, function, is the responsibility of the key class, not the hash table class.
- The hash table will be storing entries as `Objects`. It does not know enough to generate a hash code from the `Object`, which could be a `String`, an `Integer`, or a custom object.
- Java acknowledges this via the `hashCode ()` method in `Object`. All Java classes implicitly or explicitly extend `Object`. And `Object` possesses a method `hashCode ()` that returns an `int`.
- Caution: the `hashCode ()` method can return a negative integer in Java; if we want a non-negative number, and we usually do, we have to take the absolute value of the `hashCode ()`.

19

What properties should a hash code have?

- Since we use hash codes to index into the hash table and find objects, an object's hash code must remain fixed over the course of a program.
- A true random number would not make a legal hash code since we would get a different value each time we hashed.
- If two objects are equivalent (but not necessarily identical), such as two copies of the same string, then the hash codes for the two objects should be the same, because they must retrieve the same object from the hash table.
- More formally, if `o1` and `o2` are `Objects` and `o1.equals(o2)`, then `o1.hashCode ()` should `==` `o2.hashCode ()`.
- If you override the `equals ()` in a class, you should also probably override the `hashCode ()` method.

20

How to Override the equals() Method

In Java, there is an idiom associated with overriding the equals() method. Assume we are overriding the method in a class called MyClass:

```
public boolean equals(Object o) {
    if ( o instanceof MyClass ) {
        MyClass other = (MyClass) o;
        // insert logical equality testing code
        // here; if everything is OK, return true
    }
    return false;
}
```

21

Hash Code Design

- There is more art than science in hashing, particularly in the design of *hash₁* functions.
- The ultimate test of a good hash code is that it distributes its keys in an appropriately "random" manner.
- There are a few good principles to follow:
 1. A hash code should depend on as much of the key as possible.
 2. A hash code should assume that it will be further manipulated to be adapted to a particular table size, the *hash₂* phase.

22

String Class hashCode ()

- The Java String class overrides Object() and hence must override hashCode().
- Java's internal representation of a String is an array characters:

```
// character storage
private char value[];
// offset is the index of the first position used
private int offset;
// count is the number of characters in the String
private int count;
// Cache the hash code for the string
private int hash = 0;
```

23

String Class hashCode (), 2

```
public int hashCode() {
    int h = hash;
    if (h == 0) {
        int off = offset;
        char val[] = value;
        int len = count;

        for (int i = 0; i < len; i++)
            h = 31*h + val[off++];
        hash = h;
    }
    return h;
}
```

24

The $hash_2$ Function

- Once the `hashCode()` method returns an `int`, we must still distribute it, the $hash_2$ role, into one of the m slots, h , $0 \leq h < m$. The simplest way to do this is to take the absolute value of the modulus of the hash code divided by the table size, m :

```
k = Math.abs( o.hashCode() % m );
```

- This method may not distribute the keys well, however, depending on the size of m . In particular, if m is a power of 2, 2^p , then this $hash_2$ will simply extract the low order p bits of the input hash.
- If you can rely on the randomness of the input $hash_1$, then this is probably adequate. If you can't, it is advisable to use a more elaborate scheme by performing an additional hash using the input hash as key.

25

Integer Hashing

A good method to hash an integer (including our hash codes) multiplies the integer by a number, A , $0 < A < 1$, extracts the fractional part, multiplies by the number of table slots, m , and truncates to an integer. In Java, if n is the integer to be rehashed, this becomes

```
private int hashCode( int n ) {  
    double t = Math.abs( n ) * A;  
    return ( (int) ( ( t - (int)t ) * m ) );  
}
```

Unintuitively, certain values of A seem to work much better than others. The literature suggests that the reciprocal of the golden ratio, $(\text{sqrt}(5.0) - 1.0) / 2.0$ works particularly well.

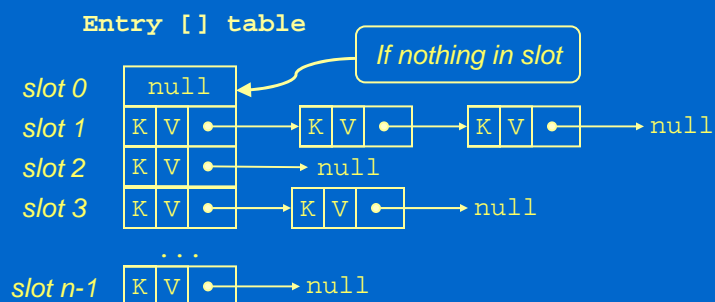
26

Hash Table Implementation

- We are going to implement our hash table (`HashMap`) as a `Map` with keys and values.
- A `HashMap` is a `Map`, not an `OrderedMap`, since it is unordered. `Map` has no `firstKey()` and `lastKey()` methods.
- We are using *singly linked lists* to resolve collisions.
- Since our singly linked list implementation from Lecture 26 is not a map and does not accommodate keys and values, we have embedded a reduced implementation of a singly linked list map in the `HashMap` class itself.

27

Sample Hashtable with Chaining



28

HashMap Members

```
public class HashMap<K, V>
    implements Map<K, V>
{
    private int length = 0;
    // heads of chains for slots
    private Entry [] table = null;
    private static final double GOLDEN =
        (Math.sqrt(5.0) - 1.0)/2.0;
    public static final int DEFAULT_SLOTS = 64;

    public HashMap( int slots ) {
        table = new Entry[ slots ];
        clear();
    }
}
```

29

static Inner Class Entry

```
private static class Entry<K, V> {
    final K key;
    V value;
    Entry<K, V> next;

    Entry( K k, V v, Entry<K, V> n )
    { key = k; value = v; next = n; }

    Entry( K k, V v )
    { key = k; value = v; next = null; }

    Entry( K k )
    { key = k; value = null; next = null; }
}
```

30

clear() Method

```
public void clear()
{
    length = 0;
    for ( int i = 0; i < table.length; i++ )
        table[ i ] = null;
}
```

31

put() Method

```
public V put( K k, V v )
{
    if ( k == null )
        throw new IllegalArgumentException(
            "Null key now allowed" );

    int idx = index( k.hashCode() );
    Entry<K, V> current = table[ idx ];
```

32

put() Method, 2

```
// If key exists in map, then exit with current
// pointing to it; else current will == null
while ( current != null )
{
    if ( current.key.equals( k ) )
        break;
    current = current.next;
}
```

33

put() Method, 3

```
if ( current == null ) { // Did we find it
    // No, insert a new item at head of chain
    length++;
    // New head points to old head
    table[ idx ] = new Entry( k, v, table[ idx ] );
    return null;
} else { // Yes, we found it
    // Replace value and return old one
    V ret = current.value;
    current.value = v;
    return ret;
}
```

34

index() Method

```
// Rehash to guarantee good hash distribution
private int index( int hcode )
{
    double t = Math.abs( hcode ) * GOLDEN;
    return ((int) ((t - (int)t) * table.length ));
}
```

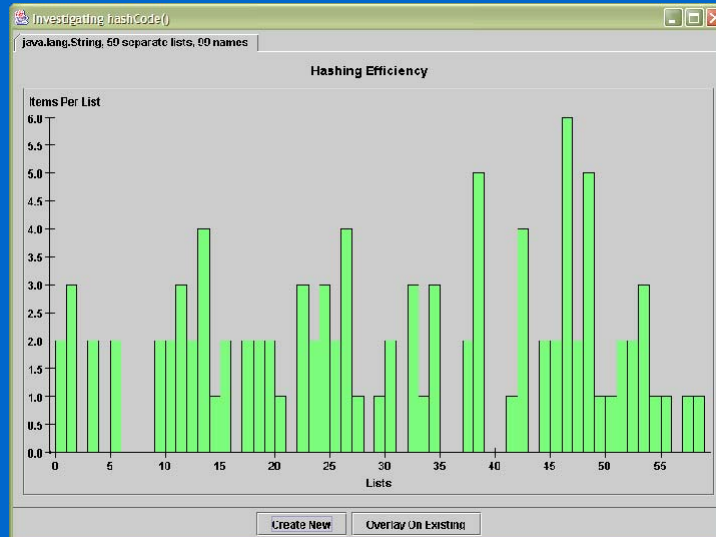
35

get() Method

```
public V get( K k ) {
    if ( k == null )
        throw new IllegalArgumentException(
            "Null key now allowed" );
    Entry<K, V> current = table[ index( k.hashCode() ) ];
    // check this slot for a match;
    // if slot is empty, return immediately
    while ( current != null ) {
        if ( current.key.equals( k ) )
            return current.value;
        current = current.next;
    }
    return null;
}
```

36

Using the HashMain Application to Explore Hash Codes



37

HashMain Introduction

HashMain.java:

- Allows a user to select any Java class, C, with a constructor that takes a single `String` argument,
- Creates 108 instances of the class C, using the names of the students in 1.00 last year and the `String` constructor of the class,
- Inserts the resulting instances into a `HashMap` and creates a histogram representing the distribution of the objects in the `HashMap`.
- By examining this histogram, we can gain insight into the efficacy of the `hashCode` method defined in the class C.

38

How to Experiment with HashMain, 1

- Download from the class web site the zip file, `Lecture31.zip`, which contains: `ResultViewer.java`, `Name.java`, `Name1.java`, `Name2.java`, `MapIterator.java`, `Map.java`, `HashMap.java`, `HashMain.java`, `FirstLastName.java`, `jas.jar`, and `names.txt`.
- Extract them to a new directory.
- Create a new project and associate it with the directory into which you just saved these files.
- Right-click on the project you have just created and select "Properties". Click the "Java Build Path" item in the list to the left, and then select the "Libraries" tab to the right. Click the "Add JARs" button and navigate to select `jas.jar` in the directory to which you saved the unzipped files. Click OK.
- Compile the project.

39

How to Experiment with HashMain, 2

- Create a histogram for `java.lang.String` by executing `HashMain` and clicking on the "Create New" button.
- What is the largest number of collisions for any single list?
- What is the smallest number?
- How many lists are empty?
- Click the overlay button and enter `Name1`. Which class appears to have the better `hashCode()` implementation, `String` or `Name1`? Why?

40

How to Experiment with HashMain, 3

- Now try the `Name2` and `Name` classes. Check the source code of the `Name`, `Name1`, and `Name2` classes. Why is the hash distribution of the `Name` class so good? Note that it inherits its `hashCode()` method from `Object`.
- Execute the main method in `Name1` by executing `Name1` (instead of `HashMain`). Do the `equals()` and `hashCode()` methods behave the way you would expect? Try the same with the `Name` class. What does this say about the excellent `hashCode` distribution of the `Name` class.
- Finally, modify `Name1` by trying to improve the `hashCode()` method. Can you come up with a version that has a better distribution than `Name2`?