

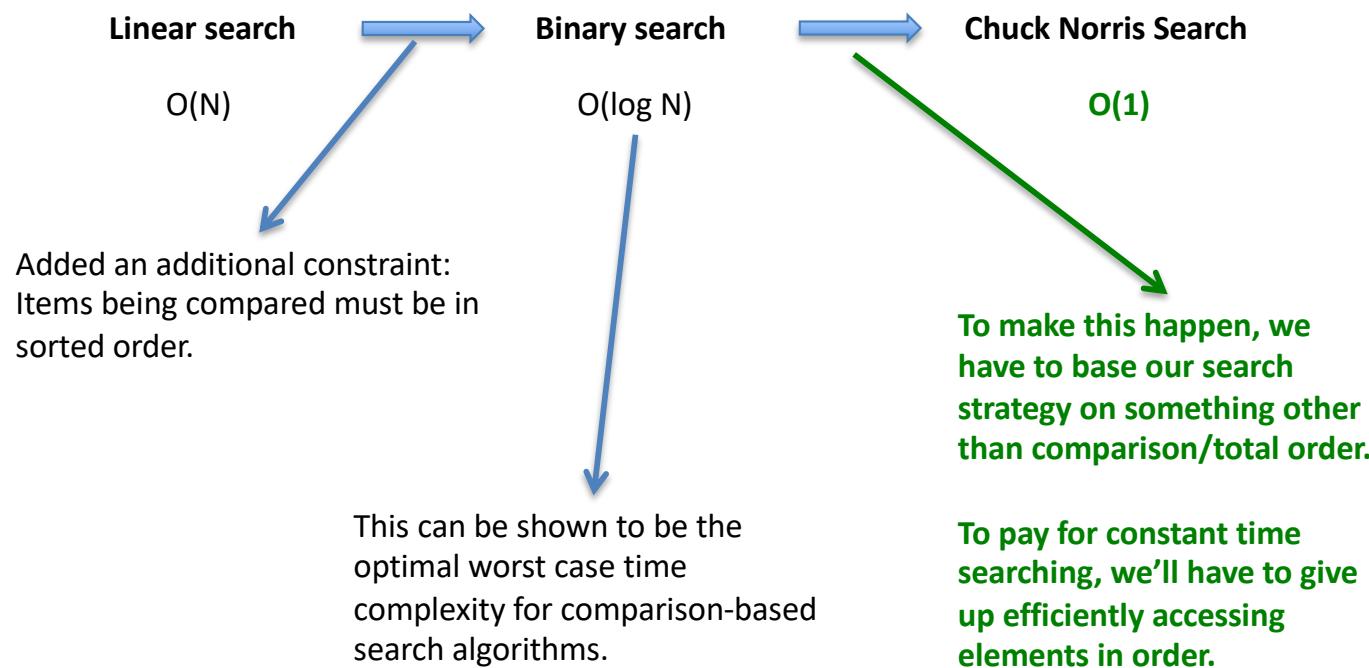


SAMUEL GINN
COLLEGE OF ENGINEERING

Hash Tables

Motivation: constant time searching

So far, the basic operation at the heart of our search strategies has been the *comparison* of one item to another, based on a total ordering of the elements.



Context: Map Collection

A **map** (or *associative array*, or *dictionary*) is collection composed of key-value pairs, where each possible key occurs at most once.

Example key-value pairs: (SS#, person info) (ISBN, book info) (phone#, address)
 (word, definition) (filename, location on disk)

Different data structures could be used to implement a map, but we can get a dramatic speed-up by considering a new type of data structure, one that's not based on comparison of elements with respect to total order.

Map method	Ordered List	Balanced Search Tree	Hash table
add(key, value)	O(N)	O(log N)	O(1)
remove(key)	O(N)	O(log N)	O(1)
search(key)	O(log N)	O(log N)	O(1)

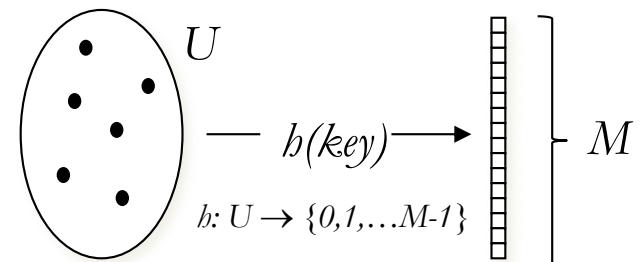
*This is average, not worst-case. But we will take steps to avoid the worst case, just as we did with quicksort.

Hash tables

A **hash table** is a data structure that implements the map behavior by using a mathematical function (the *hash function*) to associate keys with the location in a table where that key and its associated value are stored.

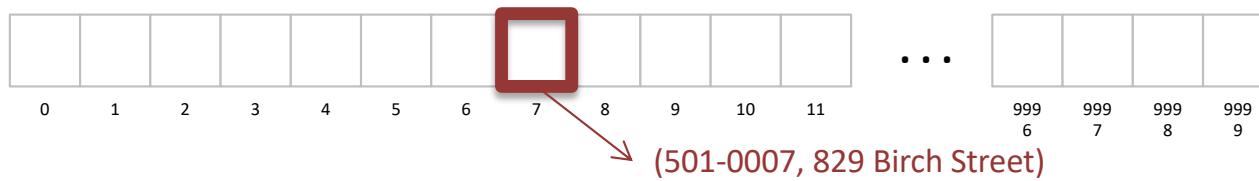
Example: (phone#, address) *Assume phone numbers in the 501 prefix of Auburn.*

(501-1234, 102 Oak Street) (501-7834, 245 Elm Street) ... (501-0007, 829 Birch Street)



hash function: Take the last four digits of the phone number and use it for the array index.

$$h(501-1234) = 1234 \quad h(501-0563) = 563 \quad \textcolor{red}{h(501-0007) = 7}$$



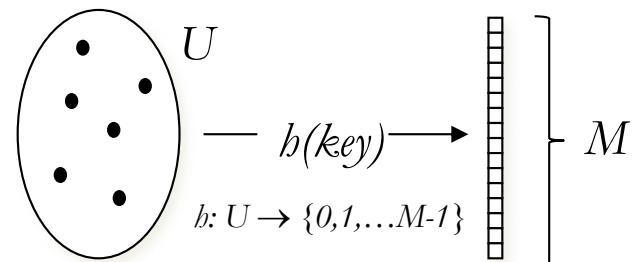
```
add(key, value) {  
    index = h(key);  
    table[index] = (key, value);  
}
```

```
search(key) {  
    index = h(key);  
    return table[index];  
}
```

```
remove(key) {  
    index = h(key);  
    table[index] = null;  
}
```

Hash tables

A **hash table** is a data structure that implements the map behavior by using a mathematical function (the *hash function*) to associate keys with the location in a table where that key and its associated value are stored.



```
add(key, value) {  
    index = h(key);  
    table[index] = (key, value);  
}
```

```
search(key) {  
    index = h(key);  
    return table[index];  
}
```

```
remove(key) {  
    index = h(key);  
    table[index] = null;  
}
```

This is simple and all three operations are $O(1)$ in the worst case, but **ONLY** if the hash function is **one-to-one** (if $h(x) = h(y)$ then $x = y$).

Hash tables

A **hash table** is a data structure that implements the map behavior by using a mathematical function (the *hash function*) to associate keys with the location in a table where that key and its associated value are stored.

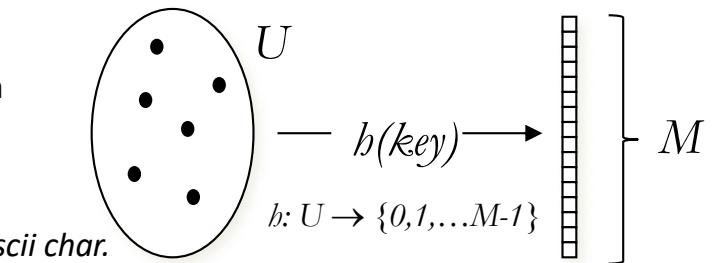
Example: (sessionid, session info) *Assume sessionid is a string of at most 80 ascii char.*

(5CpPi1g3FV, session info) (KBWocaE7mj, session info) ... (DsbAldOWGj, session info)

hash function: ??? *We'll have to come back to this one ...*

Some observations:

Since U is the set of all strings of at most 80 ascii char, $|U| = 128^{80}$.



Thus, $|U| > M$ and by the pigeonhole principle we know that some elements of the table will be assigned more than one sessionid.

We say that a **collision** occurs when $k_1 \neq k_2$ and $h(k_1) = h(k_2)$.

Collisions degrade hash table performance.

Hash tables

Most of the work in designing and implementing a hash table implementation goes into avoiding, mitigating, and resolving collisions and their effects.

We will discuss these ideas and organize the rest of the note set around the following elements of hash table implementations.

- Implementing hash functions
- Collision resolution strategies
- The add, remove, and search methods in context
- The uniform hashing assumption
- Performance analysis
- Generating and using hash codes

Hash functions, collisions

Hash functions

A good hash function must:

1. Be **deterministic**. (Always give the same index for the same key.)
2. Be **fast**. (Both asymptotically and literally.)
3. Provide a **uniform distribution** of keys over indexes. (Each table index is equally likely for each key.)

We'll think of the hash function working in two stages:

1. Convert the key into an int value called the hash code.
2. Map the hash code onto a legal index value.

For this course:

1. Use the hashCode method inherited from Object to compute hash codes.
2. Use modulus division (remainder operator % in Java) to map the hash code onto a legal index value.

```
public int h(Object key) {  
    int hashCode = key.hashCode();  
    int index = hashCode % table.length;  
    return index;  
}
```

The approach to hashing that we're using is called **modular hashing**.

In general, this is an example of the **division method**, with the alternative being **multiplicative hashing**.

Hashing example

Example: table length M = 11, $h(\text{hashcode}) = \text{hashcode} \% M$, hashcodes = {35, 22, 18, 94, 56}

22	56	35				94	18			
0	1	2	3	4	5	6	7	8	9	10

$$h(35) = 35 \% 11 = 2 \quad h(22) = 22 \% 11 = 0 \quad h(18) = 18 \% 11 = 7$$

$$h(94) = 94 \% 11 = 6 \quad h(56) = 56 \% 11 = 1$$

What would happen if an object with hashcode 12 were inserted?

$$h(12) = 12 \% 11 = 1$$

collision

Collisions

Example: table length M = 11, $h(\text{hashcode}) = \text{hashcode} \% M$, hashcodes = {35, 22, 18, 94, 56}

22	56	35				94	18			
0	1	2	3	4	5	6	7	8	9	10

$$h(12) = 12 \% 11 = 1$$

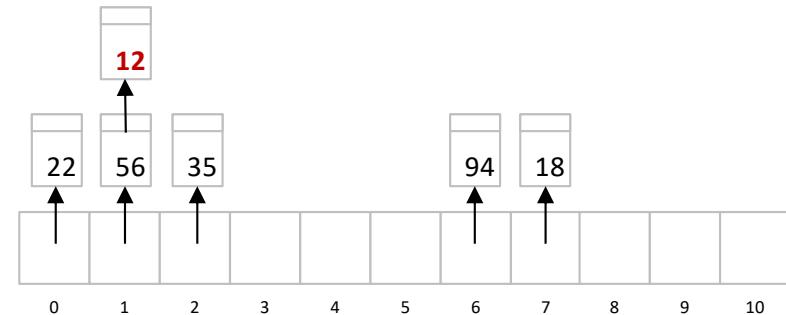
collision

How to handle this?

Put the colliding element in an unused location in the table.

22	56	35			12	94	18			
0	1	2	3	4	5	6	7	8	9	10

Let the table elements store linked lists of colliding elements.



Collision resolution

The most intuitive time that a collision could occur is during insertion. That is, a new element being inserted hashes to an index that is already occupied by another (non-duplicate) element. For insertion, resolving the collision means answering the question: Where does the new element go?

Two broad categories of collision resolution:



These terms or similar can have opposite meanings!

- | | |
|--------------------------|---|
| Open addressing | Find an alternate to the home index <i>inside</i> the hash table (that is, find another index). Note that in this scheme each index contains a reference to an element, and at any point in time an index contains at most one element reference. |
| Closed addressing | Store all elements <i>outside</i> the hash table. That is, each index contains a reference to an auxiliary data structure that stores all the elements that hash to a given index. |

Note that this same situation happens during a search or a deletion. For example, say you're searching for a specific element but its home index contains something else. Does this mean that the table doesn't contain the element you're searching for? Not necessarily. For search (and deletion), resolving the collision means answering the question: Where could the target element be since something else occupies its home index?

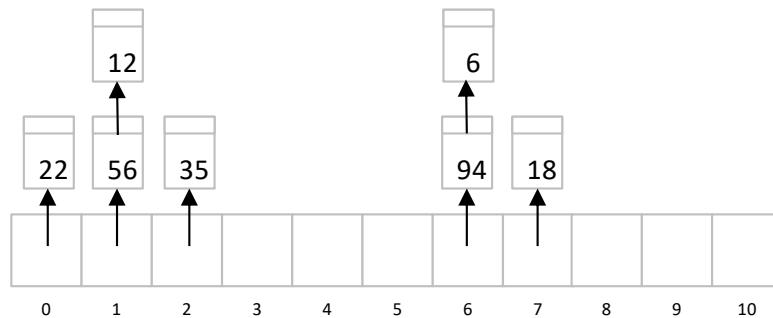
Closed addressing with chaining

Closed addressing – chaining

Store all elements outside the hash table in linked lists.

Each index in the table points to a linked list that stores all elements that hash to that index.

Example: M = 11, h(hashcode) = hashcode % M, hashcodes = {35, 22, 18, 94, 56, 12, 6}



$$h(35) = 35 \% 11 = 2 \quad h(22) = 22 \% 11 = 0 \quad h(18) = 18 \% 11 = 7$$

$$h(94) = 94 \% 11 = 6 \quad h(56) = 56 \% 11 = 1$$

$$h(12) = 12 \% 11 = 1 \quad \textcolor{red}{\text{collision}}$$

$$h(6) = 6 \% 11 = 6 \quad \textcolor{red}{\text{collision}}$$

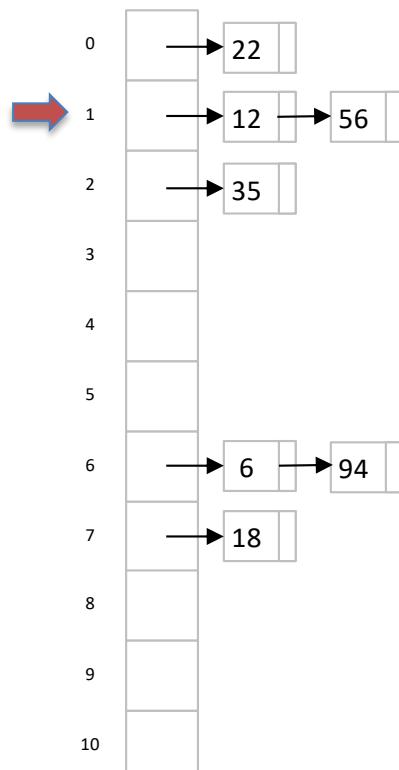
Closed addressing – chaining

Removing elements

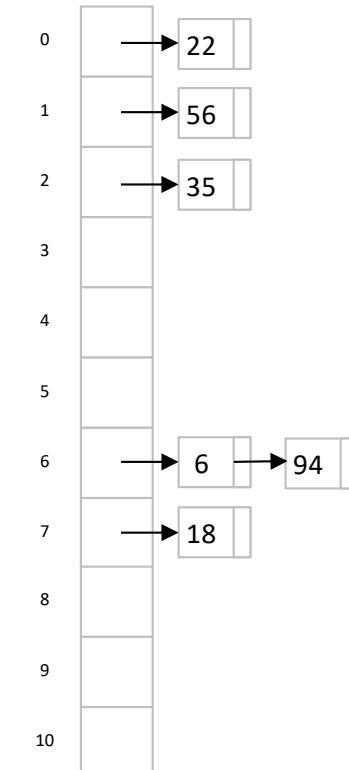
Trivial – just delete the element from the list.

Example: Delete the element with hashcode 12.

$$12 \% 11 = 1$$



Search the list in bucket 1 and delete the element with hashcode 12 if present.



Closed addressing – chaining

The performance of closed addressing with chaining is intuitively straightforward.

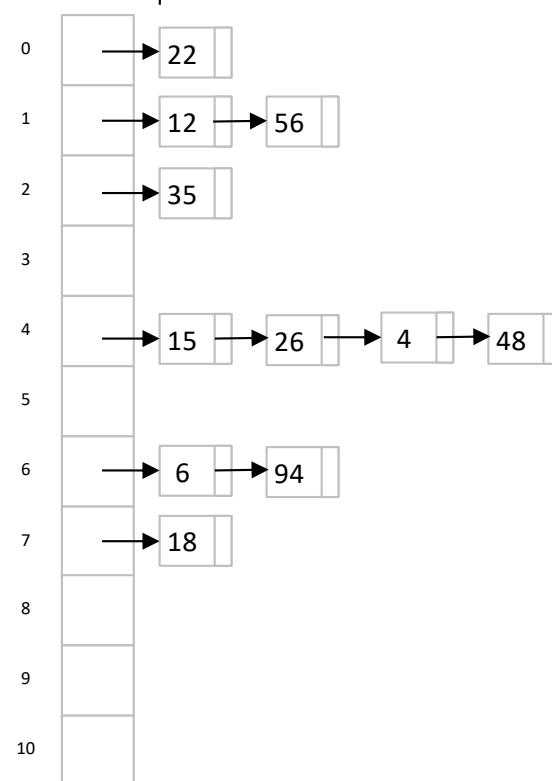
The time required for **add**, **remove**, and the **search** methods is **proportional to the length of the collision chains**.

How do we quantify these lengths?

$O(1)$?
 $O(N)$?
something else?

[more later ...]

Collisions cause these linked lists to grow, thus slowing down search, add, and remove.



Opening addressing

Open addressing – probing

Find an open (unused) index in the table. This process is called **probing**.

Probing repeatedly applies the following formula to resolve the collision:

```
index = (home + i*C) % M
```

where **home** is the index at which the collision occurred, **i** is the number of probe attempts made so far, **C** is a constant multiplier, and **M** is the number of indexes in the table (table.length).

Different choices for the constant multiplier C results in three distinct probing strategies:

Linear Probing (C = 1) $\text{index} = (\text{home} + \text{i}) \% \text{M}$

Quadratic Probing (C = i) $\text{index} = (\text{home} + \text{i}^2) \% \text{M}$

Double Hashing (C = h₂) $\text{index} = (\text{home} + \text{i} * \text{h}_2) \% \text{M}$
 $\text{h}_2(\text{hashcode}) = 1 + (\text{hashcode} \% (\text{M}-1))$

Open addressing – linear probing

Example: $M = 11$, $h(\text{hashcode}) = \text{hashcode} \% M$, hashcodes = {35, 22, 18, 94, 56, 12, 6}

22	56	35	12			94	18	6		
0	1	2	3	4	5	6	7	8	9	10

$$h(35) = 35 \% 11 = 2 \quad h(22) = 22 \% 11 = 0 \quad h(18) = 18 \% 11 = 7$$

$$h(94) = 94 \% 11 = 6 \quad h(56) = 56 \% 11 = 1$$

$$h(12) = 12 \% 11 = 1 \quad \textcolor{red}{\text{collision}}$$

Probe sequence:

$$\text{index} = (\text{home} + i) \% M$$

$$\text{index} = (1 + 1) \% 11 = 2$$

$$\text{index} = (1 + 2) \% 11 = 3$$

$$h(6) = 6 \% 11 = 6 \quad \textcolor{red}{\text{collision}}$$

Probe sequence:

$$\text{index} = (\text{home} + i) \% M$$

$$\text{index} = (6 + 1) \% 11 = 7$$

$$\text{index} = (6 + 2) \% 11 = 8$$

Open addressing – quadratic probing

Example: $M = 11$, $h(\text{hashcode}) = \text{hashcode} \% M$, hashcodes = {35, 22, 18, 94, 56, 12, 6}

22	56	35			12	94	18			6
0	1	2	3	4	5	6	7	8	9	10

$$h(35) = 35 \% 11 = 2 \quad h(22) = 22 \% 11 = 0 \quad h(18) = 18 \% 11 = 7$$

$$h(94) = 94 \% 11 = 6 \quad h(56) = 56 \% 11 = 1$$

$$h(12) = 12 \% 11 = 1 \quad \textcolor{red}{\text{collision}}$$

Probe sequence:

$$\text{index} = (\text{home} + i^2) \% M$$

$$\text{index} = (1 + 1^2) \% 11 = 2$$

$$\text{index} = (1 + 2^2) \% 11 = 5$$

$$h(6) = 6 \% 11 = 6 \quad \textcolor{red}{\text{collision}}$$

Probe sequence:

$$\text{index} = (\text{home} + i^2) \% M$$

$$\text{index} = (6 + 1^2) \% 11 = 7$$

$$\text{index} = (6 + 2^2) \% 11 = 10$$

Open addressing – double hashing

Example: $M = 11$, $h(\text{hashcode}) = \text{hashcode} \% M$, hashcodes = {35, 22, 18, 94, 56, 12, 6}

22	56	35		12		94	18		6	
0	1	2	3	4	5	6	7	8	9	10

$$h(35) = 35 \% 11 = 2 \quad h(22) = 22 \% 11 = 0 \quad h(18) = 18 \% 11 = 7$$

$$h(94) = 94 \% 11 = 6 \quad h(56) = 56 \% 11 = 1$$

$$h(12) = 12 \% 11 = 1 \quad \textcolor{red}{\text{collision}}$$

Probe sequence:

$$\begin{aligned} \text{index} &= (\text{home} + i * h_2) \% M \\ h_2 &= 1 + (\text{hashcode} \% (M-1)) \end{aligned}$$

$$h_2 = 1 + 12 \% 10 = 3$$

$$\text{index} = (1 + 1 * 3) \% 11 = 4$$

$$h(6) = 6 \% 11 = 6 \quad \textcolor{red}{\text{collision}}$$

Probe sequence:

$$\begin{aligned} \text{index} &= (\text{home} + i * h_2) \% M \\ h_2 &= 1 + (\text{hashcode} \% (M-1)) \end{aligned}$$

$$h_2 = 1 + 6 \% 10 = 7$$

$$\text{index} = (6 + 1 * 7) \% 11 = 2$$

$$\text{index} = (6 + 2 * 7) \% 11 = 9$$

Open addressing – search, load factor, remove

Open addressing – search

1. Hash the key that is being searched for to find its home index.
2. If the home index is empty, return not found.
3. If the home index contains the key being searched for, return found.
4. If the home index contains anything else, follow the probe sequence.

Example: M = 11, linear probing

22	56	35	12			94	18	6		
0	1	2	3	4	5	6	7	8	9	10

Target = 21 $h(21) = 21 \% 11 = 10$ **not found** 1 look

Target = 16 $h(16) = 16 \% 11 = 5$ **not found** 1 look

Target = 35 $h(35) = 35 \% 11 = 2$ **found** 1 look

Target = 33 $h(33) = 33 \% 11 = 0$ **not found** 5 looks

Two more adds ...

22	56	35	12	77	88	94	18	6		
0	1	2	3	4	5	6	7	8	9	10

Target = 33 $h(33) = 33 \% 11 = 0$ **not found** 10 looks

Open addressing – load factor

As the table becomes more full collisions become more likely.

Load factor (λ) = the ratio of the number of elements in the table to the capacity of the table.

$$\lambda = N / M \quad \text{empty table: } \lambda = 0 \quad \text{full table: } \lambda = 1$$

Most hash tables maintain the load factor between 0.5 and 0.8. (java.util.HashMap 0.75)

Example:

$M = 11$, linear probing, $\lambda = 7 \div 11 = 0.64$, λ max threshold = 0.70

22	56	35	12			94	18	6		
0	1	2	3	4	5	6	7	8	9	10

add(33) resulting λ would be $= 8 \div 11 = 0.73$

Rehash, then insert:

$M = 23$, linear probing, $\lambda = 8 \div 23 = 0.35$, λ threshold = 0.70

		94				6				56	33	35	12				18				22	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Open addressing – removing elements

Open addressing – Linear Probing

We can't simply delete an element. Why?

Example: Delete the element with hashcode 79.

26	92	78	79	95	90	41		60			37	51
0	1	2	3	4	5	6	7	8	9	10	11	12



$79 \% 13 = 1$ Search the probe sequence for 79. Now what?

Option 1: Immediate deletion

Make index 3 empty, then move the elements in the remaining probe sequence closer to their home bucket (if possible) by using the new empty bucket.

26	92	78		95	90	41		60			37	51
0	1	2	3	4	5	6	7	8	9	10	11	12

$$95 \% 13 = 4$$

$$90 \% 13 = 12$$

$$41 \% 13 = 2$$

26	92	78	90	95	41			60			37	51
0	1	2	3	4	5	6	7	8	9	10	11	12

Open addressing – removing elements

Open addressing – Linear Probing

We can't simply delete an element. Why?

Example: Delete the element with hashcode 79.

26	92	78	79	95	90	41		60			37	51
0	1	2	3	4	5	6	7	8	9	10	11	12



$79 \% 13 = 1$ Search the probe sequence for 79. Now what?

Option 2: Lazy deletion

Don't make index 3 empty. Instead, mark it as "unoccupied but not empty" or replace 79 with a "sentinel key". These marked indexes are emptied either periodically or when the table is rehashed.

26	92	78	∅	95	90	41		60			37	51
0	1	2	3	4	5	6	7	8	9	10	11	12

Search terminates on empties, add terminates on either empty or unoccupied.

Open addressing – removing elements

Open addressing – Double Hashing Lazy deletion is our only practical option. Why?

Example: Delete the element with hashcode 79.

26	92		51	95		19	78	60	79		37	90
0	1	2	3	4	5	6	7	8	9	10	11	12



$79 \% 13 = 1$ Search the probe sequence for 79. Now what?

$$h_2 = 1 + (79 \% 12) = 8 \quad \text{probe } 1 = 9$$

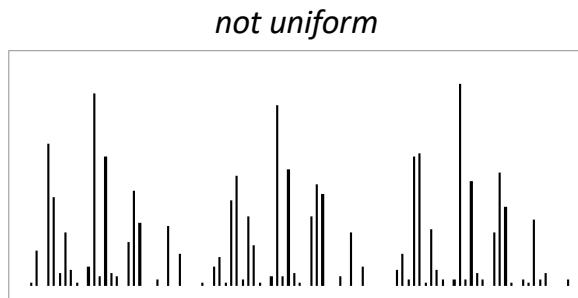
26	92		51	95		19	78	60	∅		37	90
0	1	2	3	4	5	6	7	8	9	10	11	12

Uniform hashing

Uniform hashing

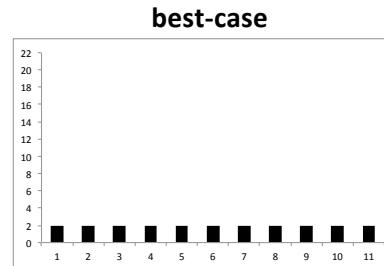
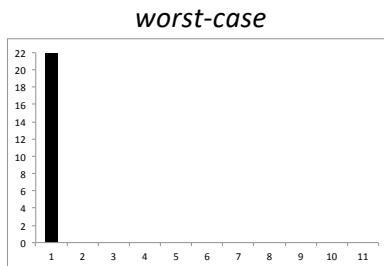
Simple uniform hashing assumption: For a hash table with M indexes, each key is equally likely to hash to an integer between 0 and $M-1$ inclusive.

Example: Same data, different hash functions



Keys = first 1000 unique words in Charles Dickens' *A Tale of Two Cities*. $M = 96$

Example: $N = 22$ $M = 11$

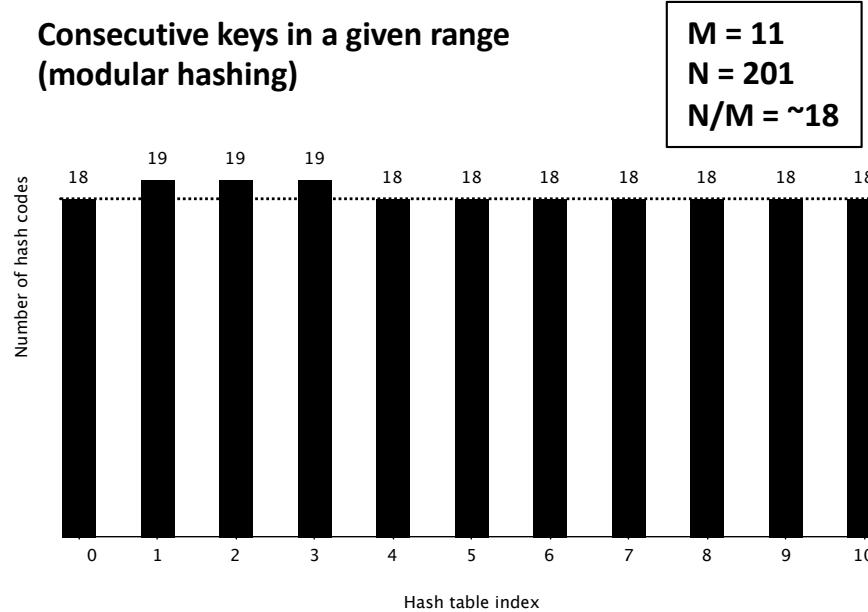


Uniform hashing will ensure that each index in the table has, *to within a small constant factor*, N/M elements.

Uniform hashing

Working definition For a hash table with M indexes and N keys, each index has to within a small constant factor N/M keys hashed to it.

Example:



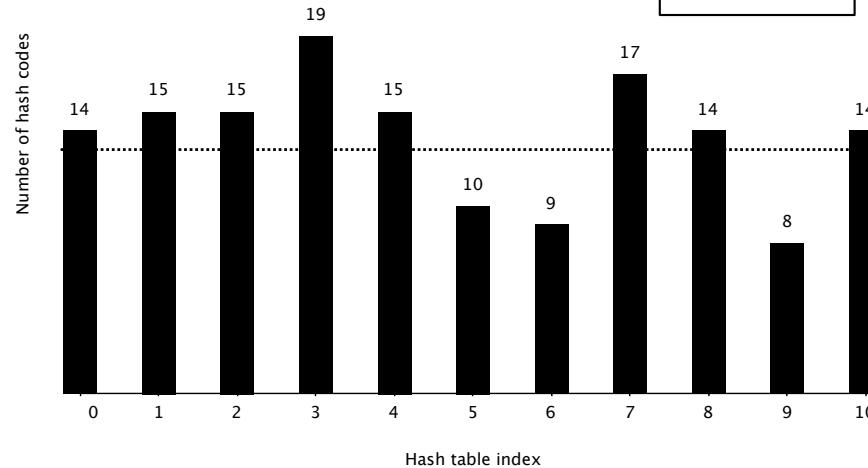
Uniform hashing

Working definition For a hash table with M indexes and N keys, each index has to within a small constant factor N/M keys hashed to it.

Example:

**Random keys in a given range > N
(modular hashing)**

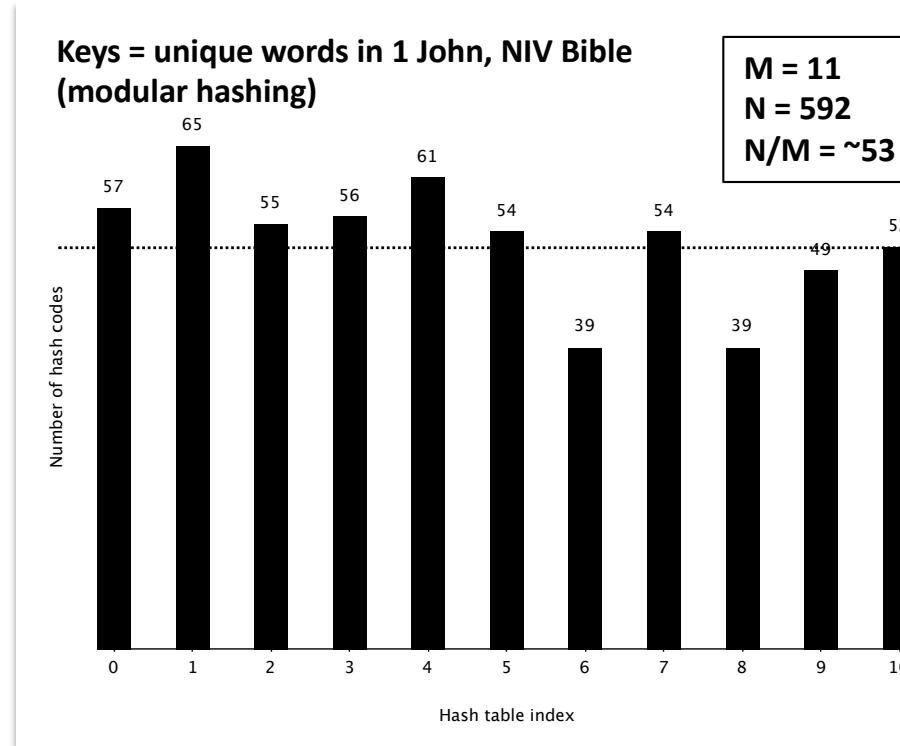
**$M = 11$
 $N = 150$
 $N/M = \sim 13$**



Uniform hashing

Working definition For a hash table with M indexes and N keys, each index has to within a small constant factor N/M keys hashed to it.

Example:

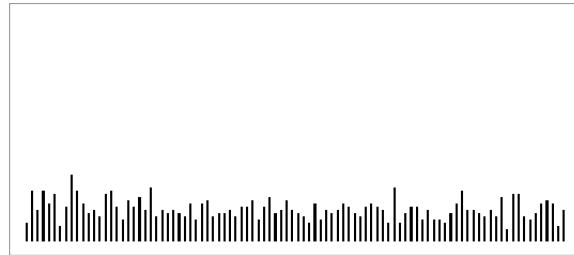
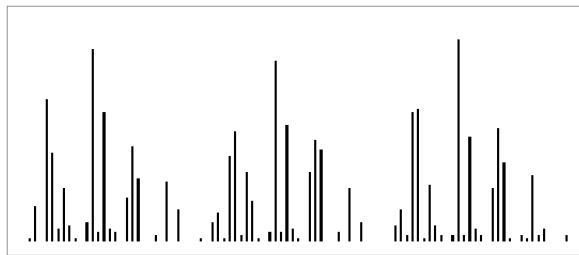


Uniform hashing

Being able to attain $O(1)$ time complexity depends on the simple uniform hashing assumption.

Uniform hashing ensures that the probability of a collision between two distinct keys in a hash table of size M is $1/M$.

Uniform hashing ensures that the size of each “collision chain” is approximately N/M .



Performance analysis

Performance analysis

Closed addressing – Separate chaining

Because of uniform hashing, we know that each list has to within a small constant factor λ elements.

Thus, add, remove, and search are $O(N/M)$.

By managing λ appropriately, we can ensure that no search will require more than some constant amount of comparisons.

For example, in a separate-chaining table with 1000 elements and 200 indexes ($\lambda = 5$), it is highly unlikely that any search will take much more than 5 comparisons.

Specifically, the probability that some list in the table has as many as 10 elements is ~ 0.00098 .

Performance analysis

Open addressing – Linear Probing

$$\sim \frac{1}{2} \left(1 + \frac{1}{1 - \lambda} \right) \quad \sim \frac{1}{2} \left(1 + \frac{1}{(1 - \lambda)^2} \right)$$

search hit search miss / insert

λ	1/2	2/3	3/4	9/10
hit	1.5	2.0	3.0	5.5
miss/insert	2.5	5.0	8.5	55.5

Open addressing – Double Hashing

$$\frac{1}{\lambda} \ln \left(\frac{1}{1 - \lambda} \right) \quad \frac{1}{1 - \lambda}$$

search hit search miss / insert

λ	1/2	2/3	3/4	9/10
hit	1.4	1.6	1.8	2.6
miss/insert	1.5	2.0	3.0	5.5

To guarantee an average cost of no more than a constant t on any search:

$$\lambda < 1 - \frac{1}{\sqrt{t}} \quad \text{for linear probing}$$

$$\lambda < 1 - \frac{1}{t} \quad \text{for double hashing}$$

How empty must we keep a hash table to guarantee no more than 5 probes for any search?

~45% empty for linear probing ($\lambda < 0.55$)
~20% empty for double hashing ($\lambda < 0.80$)

Double hashing provides the same performance in less space.

Computing hashcodes

Hashcode to legal index

```
public int h(Object key) {  
    int hashCode = key.hashCode();  
    int index = hashCode % table.length;  
    return index;  
}
```



hashCode can be negative and thus illegal for an index.

hashCode() returns an int value.

[-2,147,483,648 ... 2,147,483,647]

```
add(key, value) {  
    index = hash(key);  
    hashTable[index] = (key, value);  
}
```

```
public int h(Object key) {  
    int hashCode = key.hashCode();  
    int index = Math.abs(hashCode) % table.length;  
    return index;  
}
```

Math.abs(Integer.MIN_VALUE) = -2,147,483,648

"polygenelubricants".hashCode() = -2,147,483,648



We can't rely on Math.abs() ensuring only positive values.



Math.abs() can return a negative value!

```
public static int abs(int a) {  
    return (a < 0) ? -a : a;  
}
```

But this is a 1-in-4billion value. Do we really have to worry about it??

Find another way to ensure positive values, even in this special case.

Bit masking ints

Java's int data type is a 32-bit signed [two's complement](#) integer.

The left-most bit is the sign bit (1 = negative, 0 = positive).

Negation in two's complement notation is accomplished by inverting all the bits in the number being negated, then adding one.

Decimal	Binary	
Integer.MAX_VALUE	0111 1111 1111 1111 1111 1111 1111 1111	2,147,483,647
15	0000 0000 0000 0000 0000 0000 0000 1111	
1	0000 0000 0000 0000 0000 0000 0000 0001	
0	0000 0000 0000 0000 0000 0000 0000 0000	
-1	1111 1111 1111 1111 1111 1111 1111 1111	
-15	1111 1111 1111 1111 1111 1111 1111 0001	
-2,147,483,647	1000 0000 0000 0000 0000 0000 0000 0001	
Integer.MIN_VALUE	1000 0000 0000 0000 0000 0000 0000 0000	-2,147,483,648

Hashcode to legal index

To ensure positive hashCode values, we will mask the sign bit using the bitwise AND operator (&) and treat the int like a 31-bit unsigned integer.

1010	<i>Mask the left-most bit, leave everything else unchanged.</i>
& 0111	
-----	hashCode & Integer.MAX_VALUE
0010	

```
public int h(Object key) {
    int hashCode = key.hashCode();
    int index = (hashCode & Integer.MAX_VALUE) % table.length;
    return index;
}
```

Correct! Will always be positive.

You often see Integer.MAX_VALUE expressed as a hexadecimal literal:

```
public int h(Object key) {
    int hashCode = key.hashCode();
    int index = (hashCode & 0x7fffffff) % table.length;
    return index;
}
```

Overriding hashCode()

The hashCode() contract from Object:

```
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 JavaTM programming language.)

Overriding hashCode()

A few Java wrappers ...

```
public final class Integer ... {  
    private final int value;  
  
    public int hashCode() {  
        return value;  
    }  
}  
  
> Integer i = new Integer(32);  
> i.hashCode()  
32
```

```
public final class Double ... {  
    private final double value;  
  
    public int hashCode() {  
        long bits = doubleToLongBits(value);  
        return (int)(bits ^ (bits >>> 32));  
    }  
}  
  
> Double d = new Double(3.14159)  
> d.hashCode()  
-1340954729
```

```
public final class Boolean ... {  
    private final boolean value;  
  
    public int hashCode() {  
        return value ? 1231 : 1237;  
    }  
}  
  
> Boolean t = new Boolean(true)  
> Boolean f = new Boolean(false)  
> t.hashCode()  
1231  
> f.hashCode()  
1237
```

A double in Java is a double-precision 64-bit IEEE 754 floating point. So, the hashCode method:

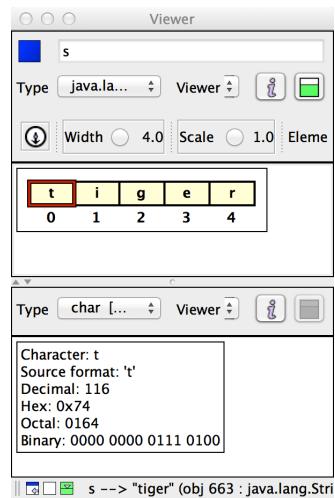
1. Convert to IEEE 754 floating-point “double format” bit layout.
2. xor the most significant 32 bits with the least significant 32 bits
3. Return the xor’d 32 bits as an int.

Overriding hashCode()

Strings are a common choice for keys and they offer a good pattern for us to think about hashing other composite types.

Strings are composed of individual char values, and a char in Java is represented as a 16-bit Unicode character. Java interprets a char as a numeric value from 0 to 65, 535 ($2^{16} - 1$).

```
public final class String ... {  
    private final char[] s;  
    ...  
}  
> String s = new String("tiger")
```



To compute a String's hash code, we could sum up the Unicode decimal values of its constituent characters.

$$\begin{array}{ccccccccc} t & i & g & e & r \\ 116 & + & 105 & + & 103 & + & 101 & + & 114 \\ & & & & & & & & = 539 \end{array}$$

```
public int hashCode(String s) {  
    int hash = 0;  
    for (int i = 0; i < s.length(); i++) {  
        hash = hash + s.charAt(i);  
    }  
    return hash;  
}
```

Overriding hashCode()

```
public int hashCode(String s) {  
    int hash = 0;  
    for (int i = 0; i < s.length(); i++) {  
        hash = hash + s.charAt(i);  
    }  
    return hash;  
}
```

hashCode("top") = 339
hashCode("pot") = 339

Make the order of the characters matter with a “place value” – choose a radix > 1 to compute the hash code.

Example: Radix = 11

$$\begin{array}{ccc} t & o & p \\ \text{---} & \text{---} & \text{---} \\ 11^2 \cdot 116 + 11^1 \cdot 111 + 11^0 \cdot 112 & = & 15369 \\ \\ p & o & t \\ \text{---} & \text{---} & \text{---} \\ 11^2 \cdot 112 + 11^1 \cdot 111 + 11^0 \cdot 116 & = & 14889 \end{array}$$

```
public int hashCode(String s) {  
    int hash = 0;  
    for (int i = 0; i < s.length(); i++) {  
        hash = 11*hash + s.charAt(i);  
    }  
    return hash;  
}
```

hashCode("top") = 15369
hashCode("pot") = 14889

String in Java ...

```
public final class String ... {  
    private final char[] s;  
  
    public int hashCode() {  
        int hash = 0;  
        for (int i = 0; i < length(); i++) {  
            hash = (31*hash) + s[i];  
        }  
        return hash;  
    }  
}  
> String s = new String("Hello")  
> s.hashCode()  
99162322
```

Overriding hashCode()

A decent recipe: Use the **Horner's Rule** pattern.

```
public class Book implements Comparable {
    private String author = new String("no title");
    private String title = new String("none");
    private int pages = 0;

    public int hashCode() {
        int hash = 17;
        hash = 31*hash + author.hashCode();
        hash = 31*hash + title.hashCode();
        hash = 31*hash + pages;
        return hash;
    }
}
```

17 and 31 are arbitrary primes.