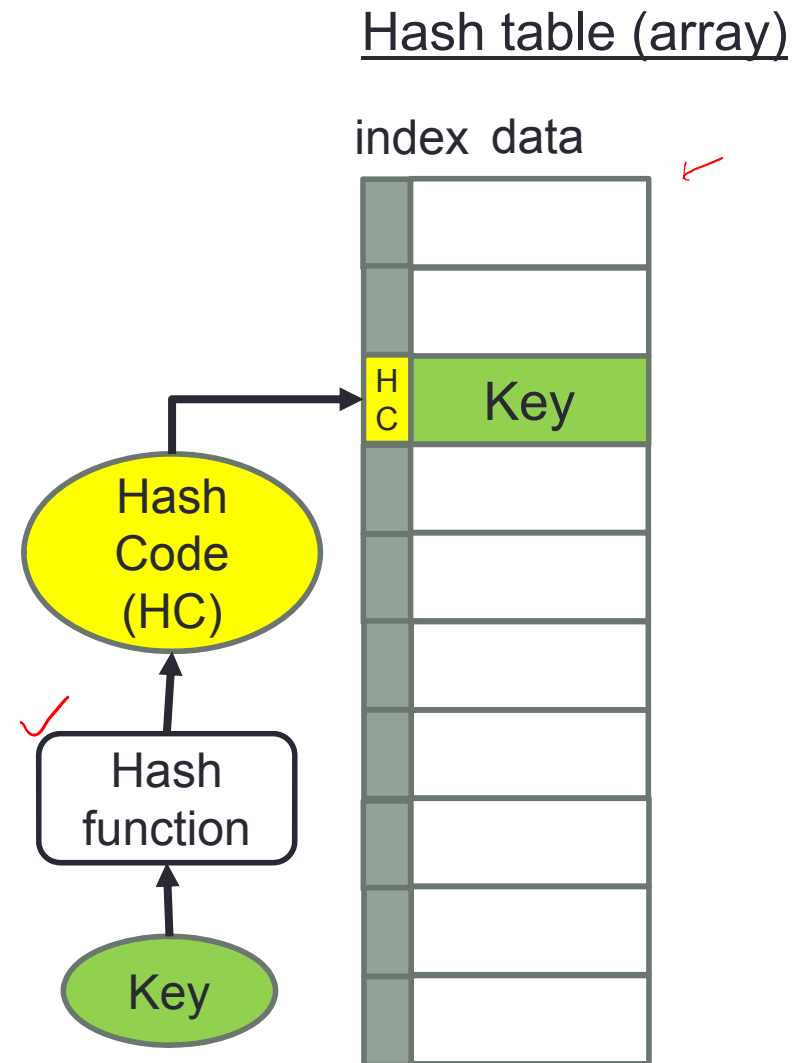




# CSE 100: HASH TABLES

# Making hashing work

- Important issues in implementing hashing are:
  - Deciding on the **hash function**
  - Deciding on the **size of the hash table**
  - Deciding on the **collision resolution strategy**
- With a good hashtable design,  $O(1)$  average-case insert and find operation time costs can be achieved, with  $O(1)$  space cost per key stored
- This makes hashtables a very useful and very commonly used data structure



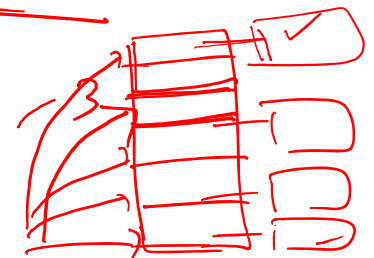
# Simple (and effective?) hash function for integers: $H(K) = K \bmod M$

- When is the function  $H(K) = K \bmod M$  where  $K$  is the key and  $M$  is the table size a good hash function for integer keys?

When  $M$  is prime OR When values of  $K$  are evenly distributed

$i = 0 \dots 10$   
 $\uparrow$   
 $k = 2i$

|   |            |          |
|---|------------|----------|
| 0 | $H(K) = 0$ |          |
| 2 | $2 = 2$    | $(0, 2)$ |
| 4 | $0 = 4$    |          |
| 6 | $2 = 1$    |          |
| 8 | $0 = 3$    |          |



Because you can never count on evenly distributed keys, always use prime-size table with this hash function

# Hash table size

- By "size" of the hash table we mean how many slots or buckets it has
- Choice of hash table size depends in part on choice of hash function, and collision resolution strategy
- But a good general "rule of thumb" is:
  - The hash table should be an array with length about 1.3 times the maximum number of keys that will actually be in the table, and
  - Size of hash table array should be a prime number (especially with the simple hash function we looked at)
- So, let  $M$  = the next prime larger than 1.3 times the number of keys you will want to store in the table, and create the table as an array of length  $M$
- (If you underestimate the number of keys, you may have to create a larger table and rehash the entries when it gets too full; if you overestimate the number of keys, you will be wasting some space)

# Hash functions for strings

- It is common to want to use string-valued keys in hash tables
- What is a good hash function for strings?
- The basic approach is to use the characters in the string to compute an integer, and then take the integer mod the size of the table
- How to compute an integer from a string?
  - You could just take the last two 16-bit chars (or last four 8-bit characters) of the string and form a 32-bit int 
  - But then all strings ending in the same 2 (or 4) chars would hash to the same location; this could be very bad
  - It would be better to have the hash function depend on *all* the chars in the string
- There is no recognized single "best" hash function for strings. Let's look at some possible ones

# String hash function #1

- This hash function adds up the integer values of the chars in the string (*then need to take the result mod the size of the table*):

```
int hash(std::string const & key) {  
    int hashVal = 0;  
    len=key.length();  
    for(int i=0; i<len; i++) {  
        hashVal += key[i];  
    }  
    return (hashVal; mod M)  
}
```



What is wrong with this hash function **for storing words of 8-characters?**

- A. Nothing
- B. It is too complex (takes too long) to compute
- C. It will lead to collisions between words with similar endings
- D.** It will not distribute keys well in a large table
- E. It will never distribute keys well in any table

# String hash function #1

- This hash function adds up the integer values of the chars in the string (then need to take the result mod the size of the table):

```
int hash(std::string const & key) {  
    int hashVal = 0, len = key.length();  
    for(int i=0; i<len; i++) {  
        hashVal += key[i];  
    }  
    return hashVal;  
}
```

- This function is simple to compute, but it often doesn't work very well in practice:
- Suppose the keys are strings of 8 ASCII capital letters and spaces
- There are  $27^8$  possible keys; however, ASCII codes for these characters are in the range 65-95, and so the sums of 8 char values will be in the range 520 - 760
- In a large table ( $M > 1000$ ), only a small fraction of the slots would ever be mapped to by this hash function! For a small table ( $M < 100$ ), it may be okay ↵

## String hash function #2: Java code

- java.lang.String's hashCode() method uses a polynomial with  $x=31$  (though it goes through the String's chars in reverse order), and uses Horner's rule to compute it:

```
class String implements java.io.Serializable, Comparable {  
    /** The value is used for character storage. */  
    private char value[];  
    /** The offset is the first index of the storage that is used. */  
    private int offset;  
    /** The count is the number of characters in the String. */  
    private int count;  
    public int hashCode() {  
        int h = 0;  
        int off = offset;  
        char val[] = value;  
        int len = count;  
  
        for (int i = 0; i < len; i++)  
            h = 31*h + val[off++];  
  
        return h;  
    }  
}
```

$$H(s) = \sum_{i=0}^n (s.charAt(i) * x^i)$$

$(x=31)$

# Collisions

- Since the hash function is  $O(1)$ , a hash table has the potential for very fast find performance (the best possible!), but...
- ... since the hash function is mapping from a large set (the set of all possible keys) to a smaller set (the set of hash table locations) there is the possibility of *collisions*: two different keys wanting to be at the same table location

# Collision resolution strategies

- Unless we are doing "perfect hashing" we have to have a collision resolution strategy, to deal with collisions in the table.
- The strategy has to permit find, insert, and delete operations that work correctly!
- Collision resolution strategies we will look at are:
  - Linear probing (from your reading)
  - Random hashing
  - Separate chaining (from your reading)

# Linear probing: inserting a key

- When inserting a key  $K$  in a table of size  $M$ , with hash function  $H(K)$ 
  1. Set  $\text{indx} = H(K)$
  2. If table location  $\text{indx}$  already contains the key, no need to insert it. Done!
  3. Else if table location  $\text{indx}$  is empty, insert key there. Done!
  4. Else collision. Set  $\text{indx} = (\text{indx} + 1) \bmod M$ .
  5. If  $\text{indx} == H(K)$ , table is full! (Throw an exception, or enlarge table.) Else go to 2.

$M = 7$ ,  $H(K) = K \bmod M = 7$

insert these keys 701 (1), 145 (5), 217 (0), 19 (5), 13 (6), 749 (0)

in this table, using linear probing:

|        |     |     |    |     |   |     |    |
|--------|-----|-----|----|-----|---|-----|----|
|        | 217 | 701 | 13 | 749 |   | 145 | 19 |
| index: | 0   | 1   | 2  | 3   | 4 | 5   | 6  |

# Linear probing: searching for a key

- If keys are inserted in the table using linear probing, linear probing will find them!
- When searching for a key  $K$  in a table of size  $N$ , with hash function  $H(K)$  :
  1. Set  $\text{indx} = H(K)$
  2. If table location  $\text{indx}$  contains the key, return FOUND.
  3. Else if table location  $\text{indx}$  is empty, return NOT FOUND.
  4. Else set  $\text{indx} = (\text{indx} + 1) \bmod M$ .
  5. If  $\text{indx} == H(K)$ , return NOT FOUND. Else go to 2.
- Question: How to delete a key from a table that is using linear probing?
  - Could you do "lazy deletion", and just mark the deleted key's slot as empty? Why or why not?

# Random hashing

- Random hashing avoids clustering by making the probe sequence depend on the key
- With random hashing, the probe sequence is generated by the output of a pseudorandom number generator seeded by the key (possibly together with another seed component that is the same for every key, but is different for different tables)
- The insert algorithm for random hashing is then:
  1. Create RNG seeded with K. Set  $\text{indx} = \text{RNG.next()} \bmod M$ .
  2. If table location  $\text{indx}$  already contains the key, no need to insert it. Done!
  3. Else if table location  $\text{indx}$  is empty, insert key there. Done!
  4. Else collision. Set  $\text{indx} = \text{RNG.next()} \bmod M$ .
  5. If all  $M$  locations have been probed, give up. Else, go to 2.
- Random hashing is easy to analyze, but because of the "expense" of random number generation, it is not often used. There is another method called double hashing that works just as well.

$$H(u) = (u)$$



# Resolving Collisions: Double hashing

- A sequence of possible positions to insert an element are produced using two hash functions
- $h_1(x)$ : to determine the position to insert in the array,  $h_2(x)$ : the offset from that position

701 (1, 2), 145 (5,4), 217 (0,3), 19 (5, 3), 13 (6,2), 749 (0,2)

in this table:

|     |     |     |  |    |     |    |
|-----|-----|-----|--|----|-----|----|
| 217 | 701 | 749 |  | 19 | 145 | 13 |
|-----|-----|-----|--|----|-----|----|

index:                      0                      1                      2                      3                      4                      5                      6

$(5 + 3) \text{ mod } 7$

# Open addressing vs. separate chaining

- Linear probing and random hashing are appropriate if the keys are kept as entries in the hashtable itself...
  - doing that is called "open addressing" ←
  - it is also called "closed hashing" ←
- Another idea: Entries in the hashtable are just pointers to the head of a linked list ("chain"); elements of the linked list contain the keys... ←
  - this is called "separate chaining"
  - it is also called "open hashing"
- Collision resolution becomes easy with separate chaining: just insert a key in its linked list if it is not already there
  - (It is possible to use fancier data structures than linked lists for this; but linked lists work very well in the average case)

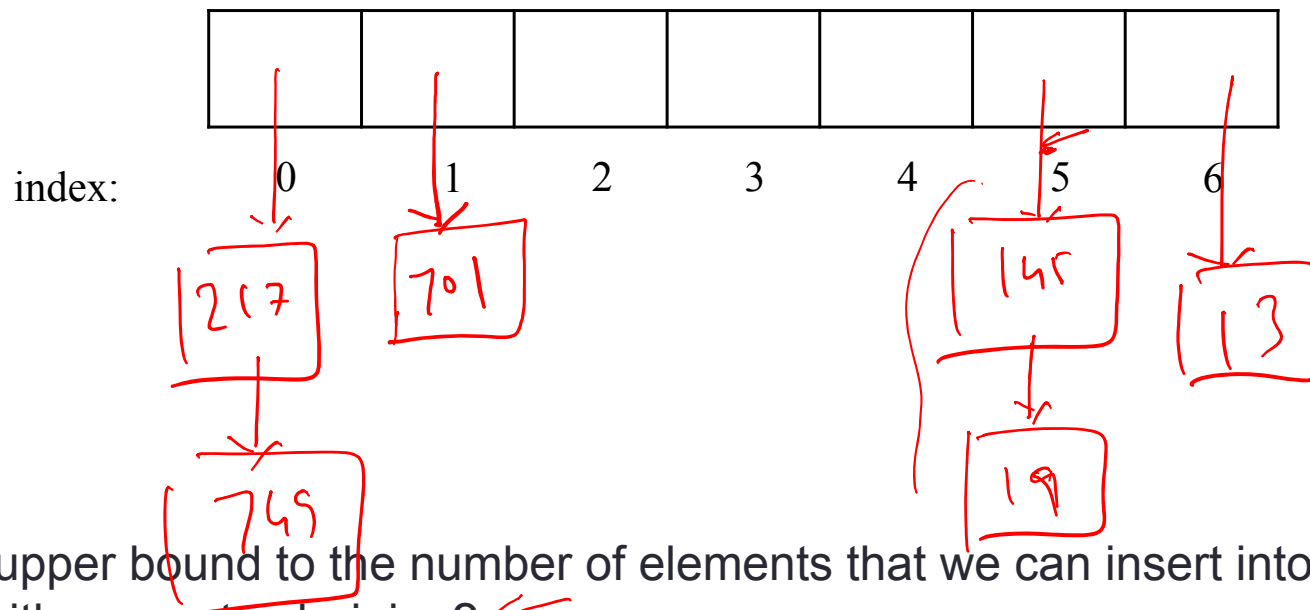
# Resolving Collisions: Separate Chaining

- using the hash function  $H(K) = K \bmod M$ , insert these integer keys:

701 (1), 145 (5), 217 (0), 19 (5), 13 (6), 749 (0)



in this table:



Is there an upper bound to the number of elements that we can insert into a hashtable with separate chaining? ←

- A. Yes, because of space constraints in the array
- B. No, but inserting too many elements can affect the run time of insert
- C. No, but inserting too many elements can affect the run time of find ←
- D. Both B and C**

# Analysis of open-addressing hashing

- What is the *worst-case* time to find a single element in a hash table with  $N$  elements in it?

- A.  $O(1)$
- B.  $O(\log(N))$
- ☒ C.  $O(N)$
- D.  $O(N\log(N))$
- E.  $O(N^2)$

M

# Analysis of open-addressing hashing

- A useful parameter when analyzing hash table Find or Insert performance is the *load factor*

$$\alpha = N/M$$

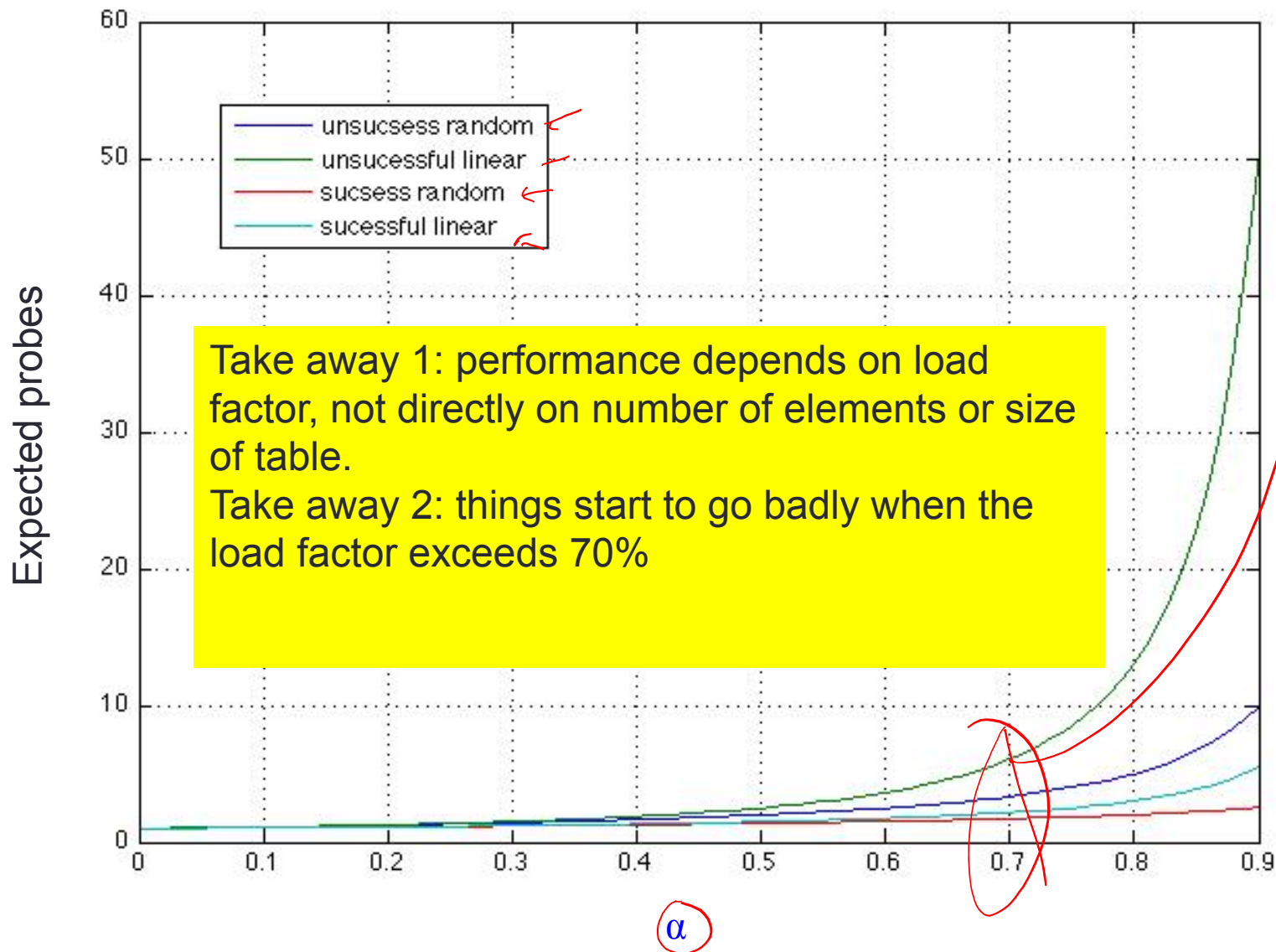

**where** M = size of the table

N = number of keys that have been inserted in the table

- The load factor is a measure of how full the table is
- Given a load factor  $\alpha$ , we would like to know the time costs, in the best, average, and worst case of
  - new-key insert and unsuccessful find (these are the same)
  - successful find

We will not derive these, but rather look at a graph that plots the relationship between  $\alpha$  and expected number of probes.

# Dependence of average performance on load



# Average case costs with separate chaining

open hashing

What you need to know:

- Separate chaining performance also depends only on load factor, and not the number of elements or size of table
- In practice it performs extremely well, even with relatively high loads