Lecture 9 – CSC 271 Data Structures

Hashing

1

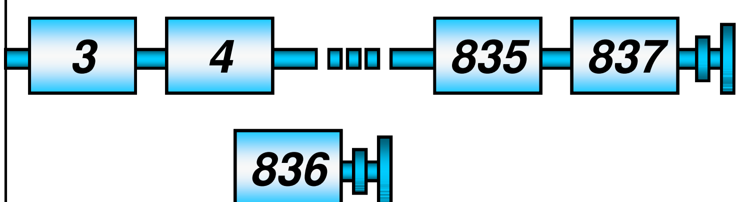
Desire

- We want to store *objects* in some *structure* and be able to *retrieve* them *extremely quickly*.
- The *number of items* to store *might be big*.

2

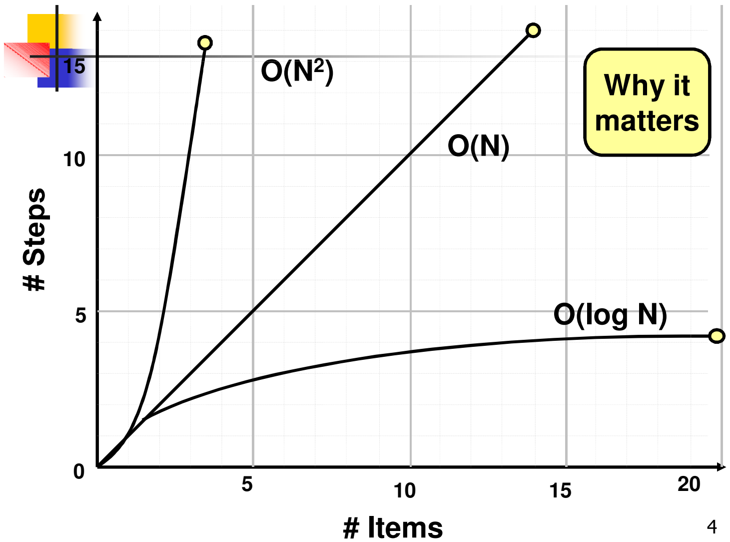
Hashing--Why?

Motivation: *Linked lists* work well enough for most applications, but provide *slow service for large data sets.*



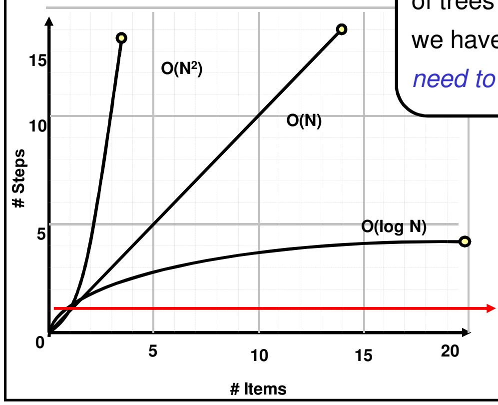
Ordered *insertion* takes too *long for large sets*.

3



4

Big Uh Oh

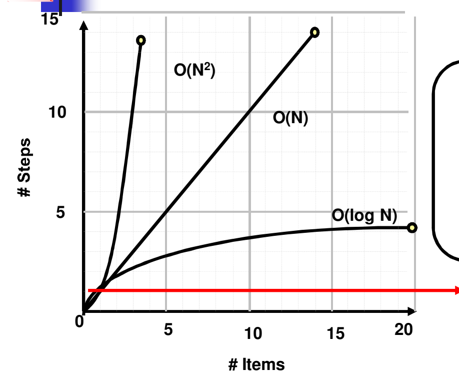


The ***O(logN)*** performance of trees is *good*, but what if we have *lots of data* and *need to be* really *fast?*

There's a way to *reduce access time* down to ***O(1),*** or nearly so

5

Sanity Check



A search time of ***O(1)?*** How is this *possible?*

Introducing....

6

Corned Beef Hash(ing)

A classic use for leftover corned beef. If you don't have enough leftover potatoes, you can use frozen hash brown potatoes in this dish.

2 tablespoons vegetable oil
1 onion, finely chopped
1 cup peeled, cubed, cooked potatoes
2 cups finely diced cooked corned beef
1/2 teaspoon thyme
salt and pepper to taste
dash Tabasco sauce
1/2 cup heavy cream
3 poached or fried eggs

Heat oil in a heavy skillet and sauté onions until tender. Add potatoes, meat, thyme, salt, pepper and Tabasco. Stir well and press mixture down with a spatula to form a large pancake. Pour cream over and press mixture down again.

Cook for about 20 minutes, until the hash has a slight crust on the bottom. Flip it over. To do this easily, place a large dinner plate face down over hash and turn the skillet and plate over. Slide the hash from the plate back into the skillet to cook the over side.

Continue cooking for an addition 10 - 15 minutes. Slice hash into three wedges. Top each wedge with an egg and serve immediately.

Yield: 3 servings.

7

Hashing!

Naive Solution:

Imagine we had to create a *large table*, sized to the *range* of possible *social security numbers*.

```
Data[ ] myRecord = new Data[ 999999999 ];

/*                 123456789
 * NOTE: Here, we assume there are
 * approximately a billion social security
 * numbers
 */
```

Perhaps not the best?

8

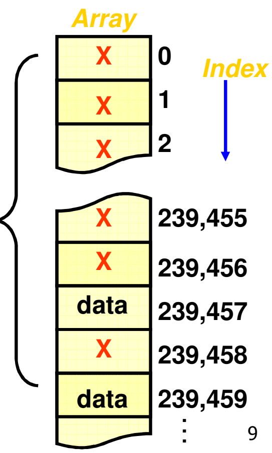
Example

Social Security numbers come in patterns of:

123-45-6578

There are *millions* of potentially unique numbers.

sparsely populated *array of objects* used to hold *~100* records



We might be tempted to use a *social security number* as an *"index value"* to some data set...

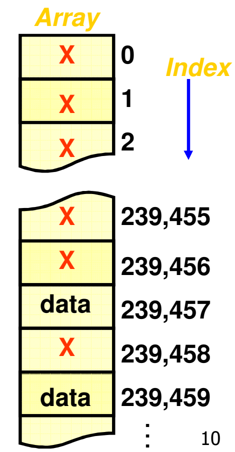
9

Example

If we only planned on holding a few *thousand records*, an *array* sized to nearly a *billion items* would be very wasteful.

Q: How can we combine the *speed of accessing* an array while still *efficiently* using available *memory* resources?

A: *Shrink* the *population range* values to fit the *array size*. Use a ***hash function***.



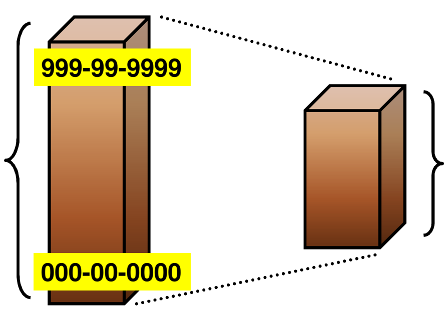
10

Hashing

Idea:

Shrink the *address space* to fit the *population size*.

range of *address space* (passed into a method)



population size (usually a fixed array size)

~ 100

11

Example

Instead of using the *social security number* as the *array index*,

```
StudentFile temp = studentRecords[iSocSecNum];
```

reduce the range of the number to something within the *size of the array:*

```
StudentFile temp =
      record[iSocSecNum % record.length];
```

returns an *index* within the appropriate *range*

12

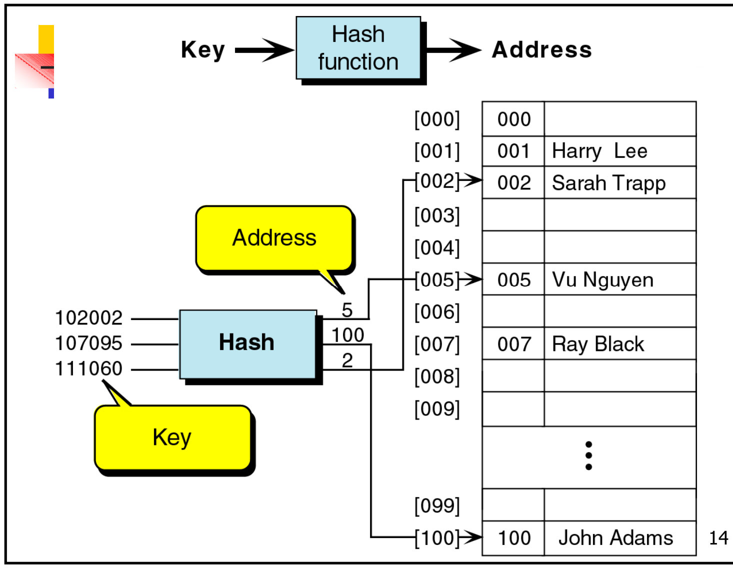
Recall

- Our friend the Mod Function %

```
x % y
```

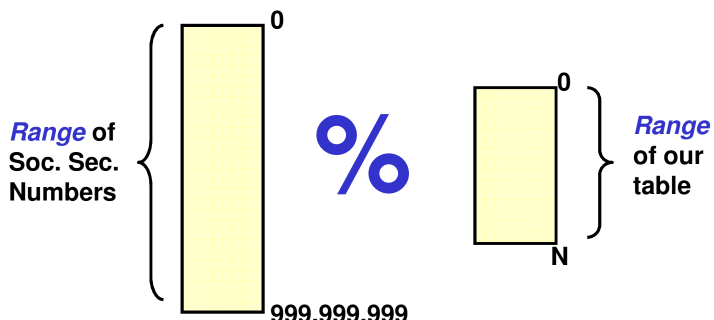
- will yield values between *0* and *y - 1*

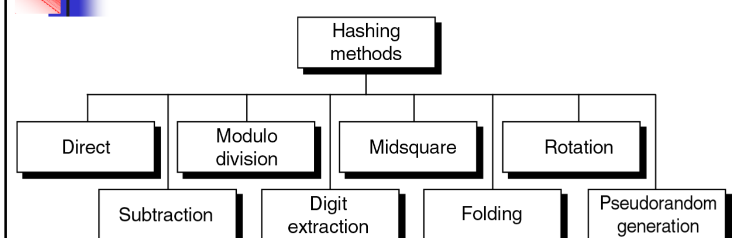
Note remarkable similarity to range of values for the *index* of an *array of size y*



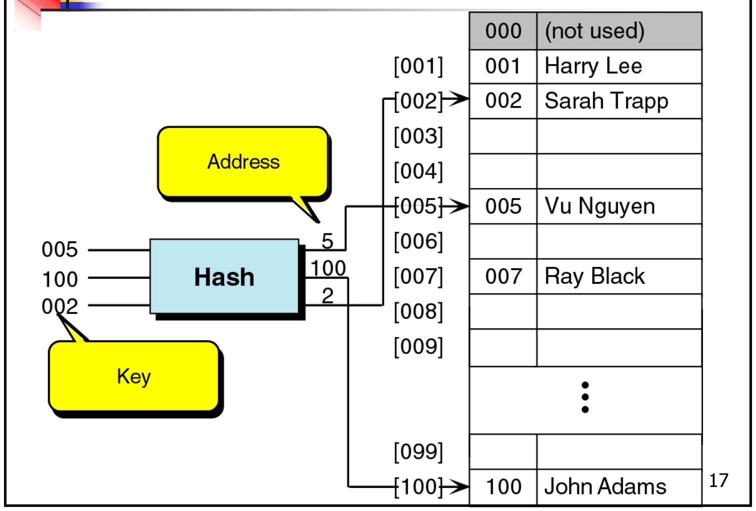
The Art of Hashing

Obviously, the *hash function is the key*. It *takes a large range* of values, and *shrinks* them to fit a *smaller address range.*



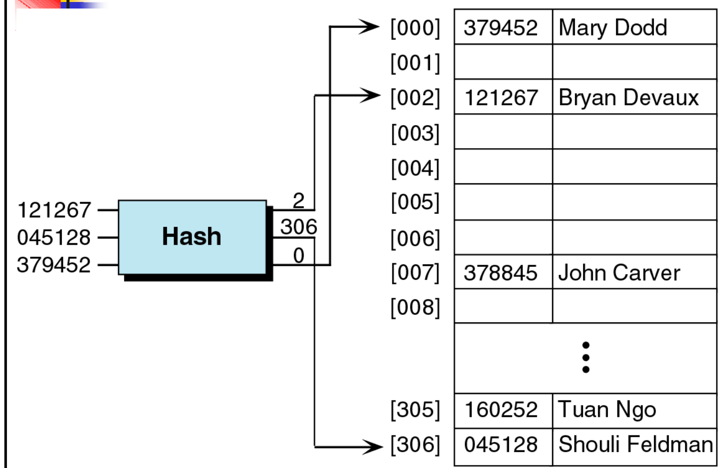


Direct



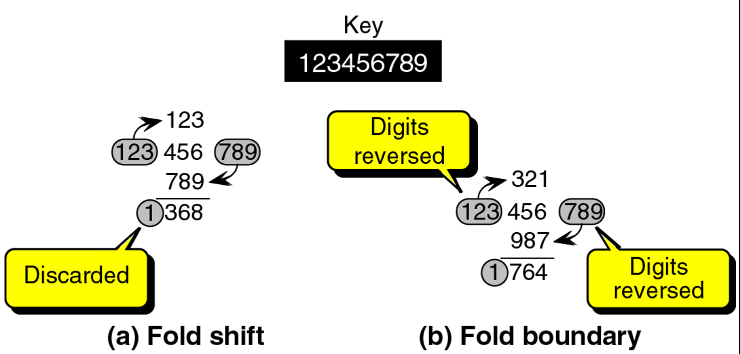
17

Modulo division



18

Folding



19

Rotation

Original Key	Rotation	Rotated Key
600101	600101	160010
600102	600102	260010
600103	600103	360010
600104	600104	460010
600105	600105	560010

20

A problem…

- We have an *array* of length ***100***
- We have about *50 students*
- We *hash* using: **`ssn % 100`**

- George P. Burdell
  - 123-45-6789
- George W. Bell
  - 321-54-7689

Collision!

Hash Functions: How To Design

The Perfect Hash Function:

- would be *very fast* (used for *all* data access)
- would return a *unique result* for *each key,*
  i.e., would result in *zero collisions*
- in general case, ***perfect hash doesn't exist***
  (we can create one for a *specific population*, but as soon as that population changes...)

Common Hash Functions:

- *Digit selection* : e.g., *last 4* digits of *phone* number
- *Division* : *modulo*
- *Character keys*: use *ASCII* num values for chars (e.g., *'R' is 82)*

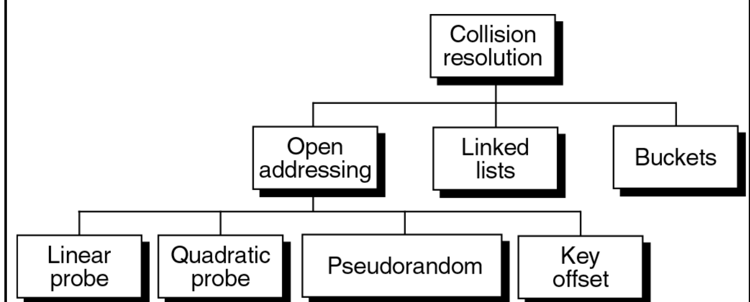
Cost of Hash

Two costs of hashing:

1. loss of *natural order*
   - side effect of desired *random* shrinking
   - lose any *ordering* of *original indices*
2. *collision* will occur
   - *no perfect hash function*
   - *when* **(not *"if"*)** *collision*, how to *handle* it?

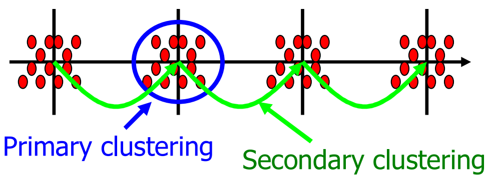
Collision Resolution strategies:

1. Multiple record *buckets*: small for each index, but . . .
2. *Open address* methods: look for next open address, but . . .
3. Coalesced *chaining*: use *cellar for overflow (~34..40%* of size)
4. *External chaining*: *linked list* at each location



Clustering

- Primary clustering
- Secondary clustering



Collision Resolution

Technique: *Multiple element buckets*

- ***Idea***: have *extra spaces* there for *overflow*
- if *population of 8,* and if *hash* function of ***mod 8***, then:

	1st hash	1st collision	2nd collision
0			
1			
2			
3			
4			
5			
6			
7			

Problems: using ***3N*** space; "what if *3rd collision* at any one locale?"

Buckets

[000]	Bucket 0	379452	Mary Dodd
[001]	Bucket 1	070918	Sarah Trapp
		166702	Harry Eagle
		367173	Ann Georgis
[002]	Bucket 2	121267	Bryan Devaux
		572556	Chris Walljasper
	⋮		
[307]	Bucket 307	045128	Shouli Feldman

Linear probe places here

Collision Resolution

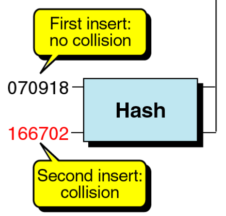
Technique: *Open address methods*

- ***Idea***: upon *collision*, look for an *empty spot*
- if *population of 8*, and if *hash function* of *mod 8*
- Assume *data* items arrived in the *order*: **W, X, Y, Z, A, B, C, D**

0	D hashes to 2	D belongs at 2, but C already there
1	W hashes to 1	W already at 1, so C to *next available* slot
2	C hashes to 1	
3	X hashes to 3	
4	Y hashes to 4	
5	Z hashes to 3	X already at 3, so Z to *next available* slot
6	A hashes to 6	
7	B hashes to 5	B belongs at 5, but Z already there

Problem: *Deteriorates* to an *unsorted list* (e.g., ***O(N)***)

Open Addressing – Linear Probe



[000]	379452	Mary Dodd	
[001]	070918	Sarah Trapp	
[002]	121267	Bryan Devaux	Probe 1
[003]	166702	Harry Eagle	Probe 2
[004]			
[005]			
[006]			
[007]	378845	John Carver	
[008]			
	⋮		
[305]	160252	Tuan Ngo	
[306]	045128	Shouli Feldman	

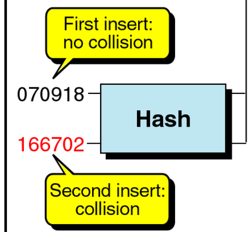
- **Advantages:**
  - *Simple*
  - Elements *near to home address*
- **Disadvantages:**
  - More *complex search* algorithms especially if *element deleted*

29

Open Addressing – Quadratic Probe

- Like linear probe, but *increment* is the *collision probe number squared:*

  $1^2, 2^2, 3^2, \ldots, n^2$



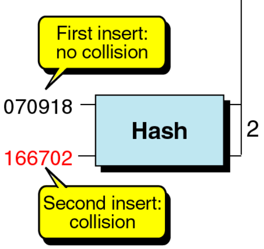
[000]	379452	Mary Dodd	
[001]	070918	Sarah Trapp	
[002]	121267	Bryan Devaux	Probe 1
[003]	166702	Harry Eagle	Probe 2
[004]			
[005]			
[006]			
[007]	378845	John Carver	
[008]			
	⋮		
[305]	160252	Tuan Ngo	
[306]	045128	Shouli Feldman	

30

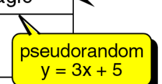
Open addressing – pseudorandom collision resolution

Double hashing – *address is rehashed*

- Use *collision address* in the *random number calculation (not the key – hash function)*



[000]	379452	Mary Dodd
[001]	070918	Sarah Trapp
[002]	121267	Bryan Devaux
[003]		
[004]		
[005]		
[006]		
[007]	378845	John Carver
[008]	166702	Harry Eagle
	⋮	
[305]	160252	Tuan Ngo
[306]	045128	Shouli Feldman



31

Open addressing – **Key offset**

- *Double hashing method – different collision path for different keys*
- New address calculated as a function of the old address and the key

Example:

Offset = (int)(key/list_size)

Address = ((offset + old address) % list_size)

- key = 166702 ; list_size = 307 ;
- **address using %:** address = 166702 % 307 = *1 → collision?*

Offset = int(166702/307) = 543

New address = ((543 + 1) % 307) = 237

237 collision? → repeat the process

Offset = int(166702/307) = 543

New address = ((543 + 237) % 307) = 166

32

Collision Resolution

Technique: Coalesced chaining:

- ***Idea***: have small *extra "cellar"* to handle collision
- if *population of 8*, and if *hash* function of ***mod 8***
- Assume *data items arrived* in the *order:* **W, X, Y, Z, A, B, C, D**

0		
1	W hashes to 1	9
2	D hashes to 2	
3	X hashes to 3	10
4	Y hashes to 4	
5	B hashes to 5	
6	A hashes to 6	
7		
8		
9	C hashes to 1	
10	Z hashes to 3	

Cellar

Works well **with cellar of** ***35 to 40%*** **of** ***N*** **if** ***good hash function;*** **cellar** ***can overflow*** **if need be**

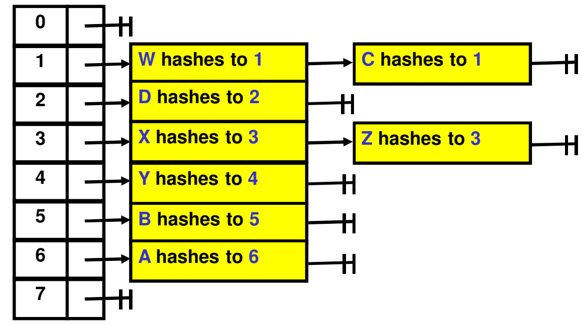
Cellar bottom is now *8*

33

Collision Resolution

Technique: External chaining:

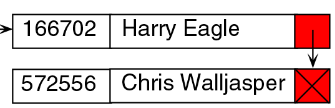
- ***Idea***: have *pointers* to all items at given *hash,* handle *collision* as *normal event*.
- if *population of 8*, and if *hash* function of *mod 8*
- Assume data *items arrived* in the *order:* **W, X, Y, Z, A, B, C, D**



34

Linked Lists

[000]	379452	Mary Dodd
[001]	070918	Sarah Trapp
[002]	121267	Bryan Devaux
[003]		
[004]		
[005]		
[006]		
[007]		
[008]		
	⋮	
[305]	160252	Tuan Ngo
[306]	045128	Shouli Feldman



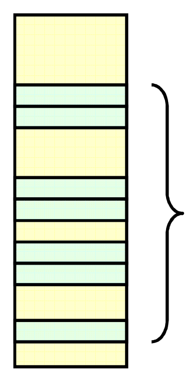
35

Load Factor

We can *measure how "full" our table* has become with a ***"load factor".***

A *load factor* is merely the *ratio of full spots to total spots.* It gives us a *measure of table utilization*.

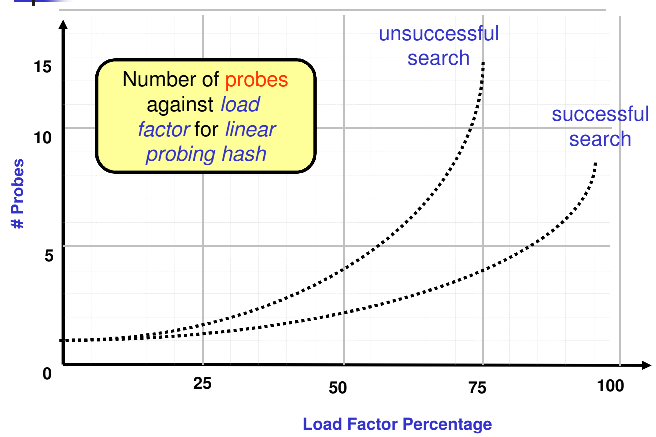
This gives us a way of *estimating the chance of a collision*



Approx. *40% utilized*

36

What Good is a "Load Factor"?



Probe?

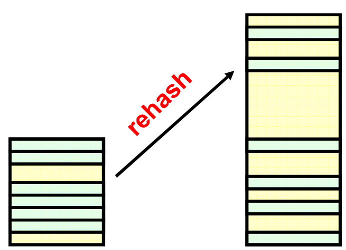
- Is this lecture sponsored by
- No, not exactly.
- A *"probe"* refers to an *attempt to find the target*.

Rehashing

Performance charts suggest that as our *load factor increases*, the *number of probes increases*.

At some point, it may be worth the trouble to *grow the table size, and rehash*

Make a *new table,* and *rehash* each entry into the new table

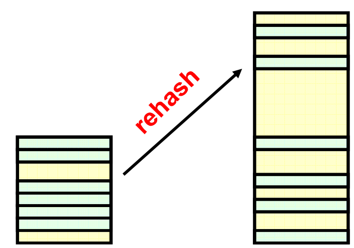


Rehashing

Question:

Why can't we just reuse the old hash values in our new, larger table?

Make sure you can answer such a question.



Better Hashing

The *key to efficient hashing* is the *hash function*. This is fairly *easy* if the *data* hold a *uniformly distributed number.*

But how can we *efficiently* convert a *name* into a *key number?*

Experimenting with this problem will expose some issues in hashing.

41

Hashing Names

Version # 1

```
public int getHash (String strName) {

    int hash = 0;

    for (int i =0; i < strName.length(); i++) {
        hash +=  (int) strName.charAt(i);
    }

   hash %= tableSize;

   return hash;
}
```

This works, but what are its limitations?

42

Hashing Names

```
public int getHash (String strName) {

    int hash = 0;

    for (int i =0; i < strName.length(); i++) {
        hash +=  (int) strName.charAt(i);
    }

   hash %= tableSize;

   return hash;
}
```

For *large tables*, this hash function *does not distribute the keys very well.*

GIVEN: Most *names* are *8 or fewer characters* long. Most names have characters up to *ASCII 127.*

So, on average, our *hash function returns numbers up to 1,016*. If the *table size* is a *large prime number*, we will *never* distribute *keys* to the *upper portion of the table*.

As a result, we will tend to have *more collisions* on the *lower part of the table.*

43

Hashing Names

Version # 2

```
public int getHash (String strName) {

    int hash = 0;

    hash = (int)
         strName.charAt(0) +
         27 * (int) strName.charAt(1) +
        729 * (int) strName.charAt(2);

   hash %= tableSize;

   return hash;
}
```

Strategy: *only* examine *first three characters*

This works, but what are its *limitations?*

Given: *27* is the *number of characters* in the alphabet, plus the space character. *729* is *27 ^2*.

44

Hashing (cont'd)

```
public int getHash (String strName) {

    int hash = 0;

    hash = (int) strName.charAt(0) +
         27 * (int) strName.charAt(1) +
         729 * (int) strName.charAt(2);

   hash %= tableSize;

   return hash;
}
```

There are now *26^3* (or *17,576)* combinations of letters. This *should distribute evenly over a large table.*

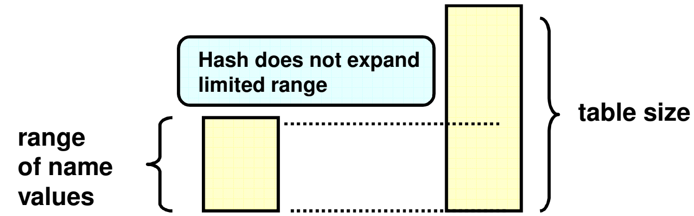
BUT: English does *not uniformly distribute letters* in words. There are in fact only *2,851 combinations of three letter sequences* in English. So once again, we *under utilize* the table. (Only about a *quarter* is *actually hashed*.)

45

Inductive Analysis

What happened in our *two previous examples?*

They worked, but what caused them to be *inefficient?*



The problem was a mismatch of *address space* and *table size*.
If the *table size exceeds the address range*, an *under utilization* occurs.

46

Improved Hash Function

```
public int getHash (String strName) {

    int hash = 0;

    for (int i=0; i< strName.length(); i++)
        hash = 27 * hash + (int) strName.charAt(i);

   hash %= tableSize;

   if (hash < 0 )
         hash += tableSize;

   return hash;
 }
```

Potential for *wrap-around*

Side note: for the mathematically inclined, this applies what is known as ***"Horner's rule"***

47

Why Is This a Better Hash?

```
public int getHash (String strName) {

    int hash = 0;

    for (int i=0;
         i< strName.length(); i++)
            hash = 27 * hash +
             (int) strName.charAt(i);

   hash %= tableSize;

   if (hash < 0 )
         hash += tableSize;

   return hash;
}
```

Still subject to quirks of the English language, but not sensitive to three-letter combinations.

Uses a *polynomial expansion to generate a large input value*, so the hash will likely use the *entire table*, even for *large tables.*

Addresses possible roll-over

48

- $14*32^3 + 15*32^2 + 20*32^1 + 5*32^0$
- ((14*32 + 15)*32 + 20)*32 + 5
- Horner's rule minimizes the number of computations

Hard Lessons about Hashing

Your *hash function* must be *carefully selected*.

It *varies with* your *data*. You have to study your input, and *base your hash on the properties of the input data*.

Your *range of input* should be *larger than your table size* (else your hashing will *under utilize the table).*

Watch out for *tables sized to a large prime number:* if that number lies *close to a power of 2, it can be trouble*.

Summary of Hash Tables

- Purpose: Allows extremely fast lookup given a key value. Reduce the address space of the information to the table space of the hash table.
- Hash function: the reduction function
- Collision: hash(a) == hash(b), but a!=b
- Collision resolution strategies
  - Multiple element buckets still risk collisions
  - Open addressing quickly deteriorates to unordered list
  - Chaining is most general solution

Test Yourself

In the context of a hashtable, what is the *address space?*

What is a *hashing function?*

Should a *hashing function return values equal to, greater than or less than the table size?* Why?

What *data structure* (seen in previous slides) might we use to *implement a hash table?*