Hashing

15-121
Data Structures

Ananda Gunawardena

Hashing

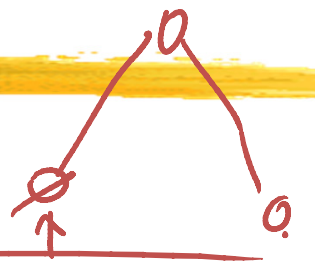
Why do we need hashing?

- Many applications deal with lots of data
  - Search engines and web pages
- There are myriad look ups.
- The look ups are time critical.
- Typical data structures like arrays and lists, may not be sufficient to handle efficient lookups
- In general: When look-ups need to occur in near constant time.

sorted O(log n)

Store, Find, update O(1)

Worst Case



	Store	Find	update
Array	O(1)	O(n)	O(n)
Sorted Arr	O(n)	O(log n)	O(n)
List	O(1)	O(n)	O(n)
Sorted List	O(n)	O(n)	O(n)
Stacks	~~O(1)~~ X	~~O(n)~~ X	X
Queues	X	X	X
BSTs	O(n)	O(n)	O(n)
AVL's	O(log n)	O(log n)	O(log n)

Why do we need hashing?

- Consider the internet(2002 data):
  - By the Internet Software Consortium survey at http://www.isc.org/ in 2001 there are 125,888,197 internet hosts, and the number is growing by 20% every six month!
  - Using the best possible binary search it takes on average 27 iterations to find an entry.
  - By an survey by NUA at http://www.nua.ie/ there are 513.41 million users world wide.

Why do we need hashing?

- We need something that can do better than a binary search, O(log N).
- We want, O(1).

Solution: **Hashing**

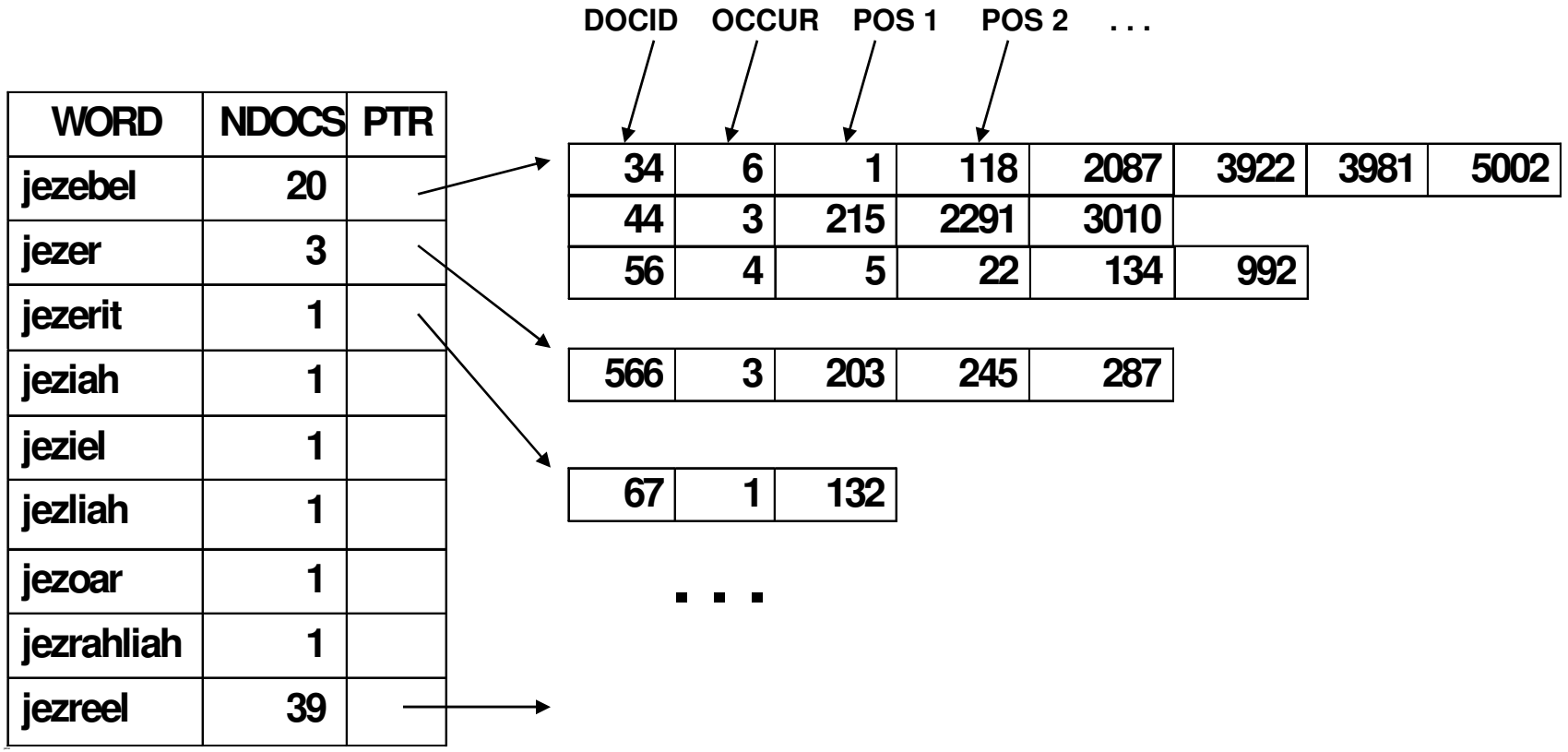
In fact hashing is used in:

Web searches
Compilers

Spell checkers
passwords

Databases
Many others

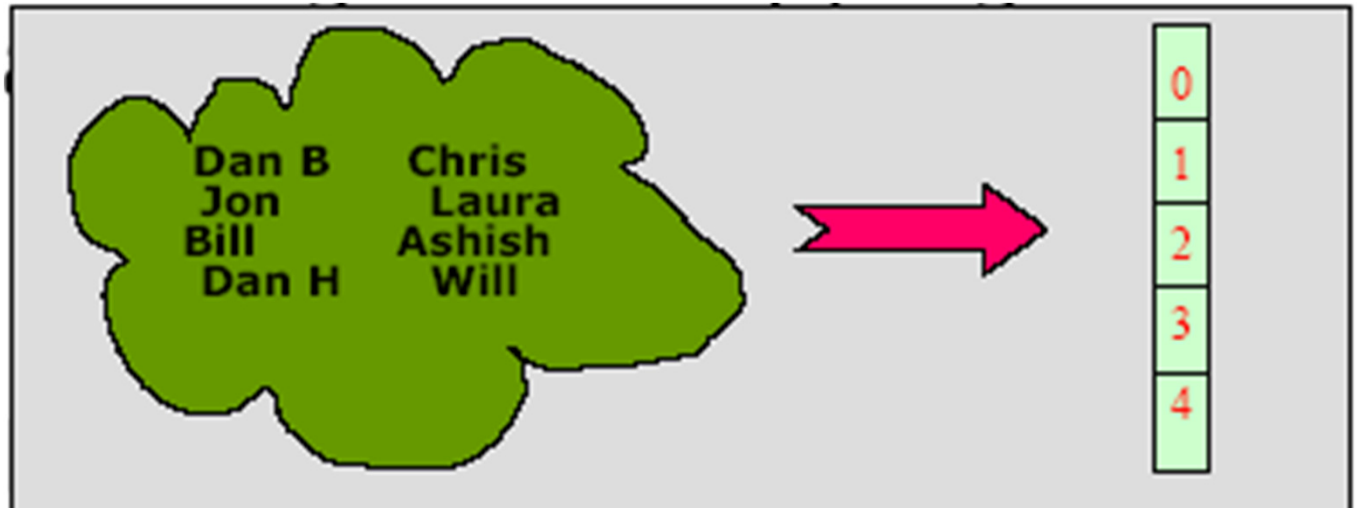
Building an index using HashMaps

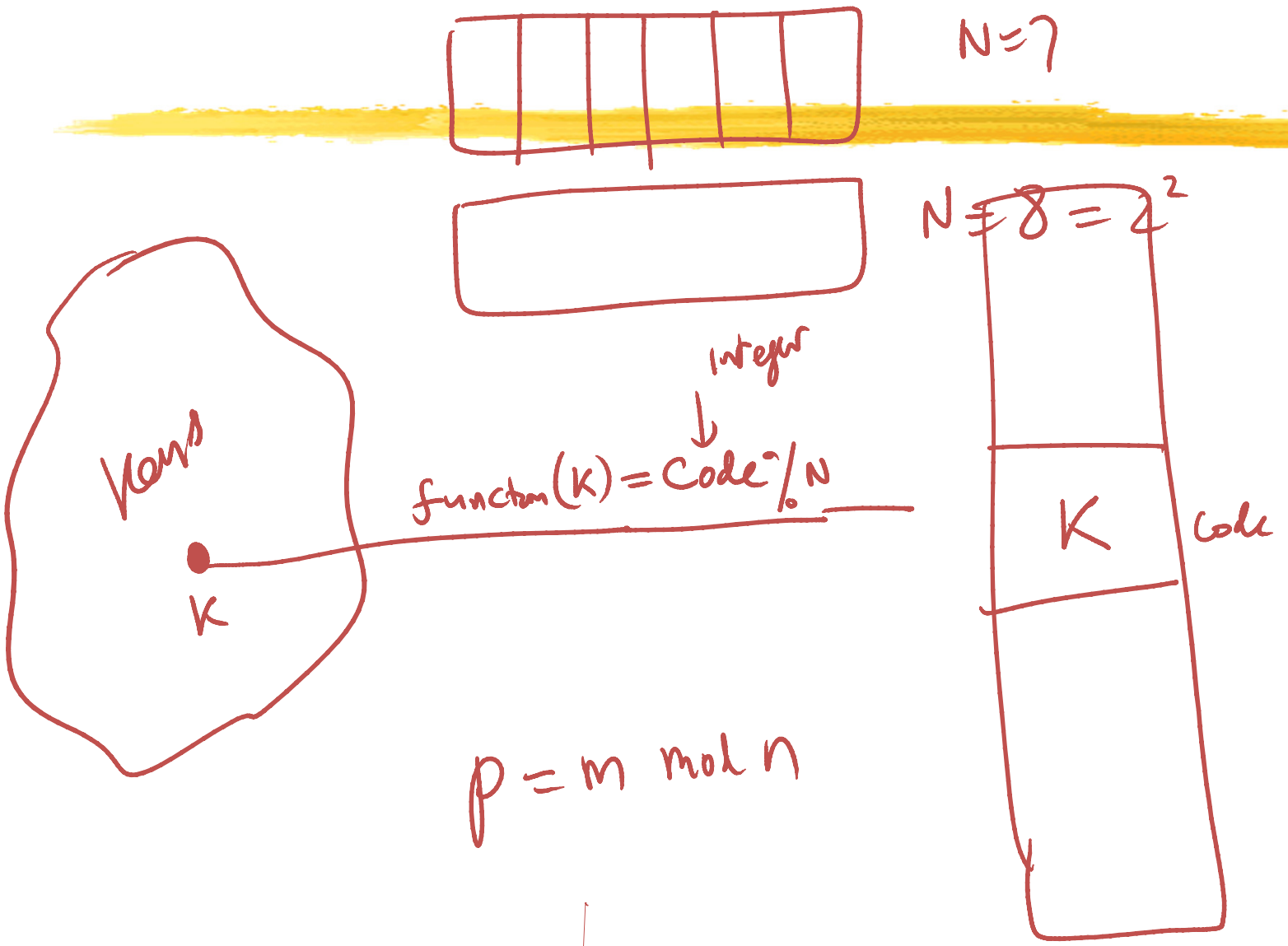


More on this in Graphs…

The concept

- Suppose we need to find a better way to maintain a table (Example: a Dictionary) that is easy to insert and search in O(1).

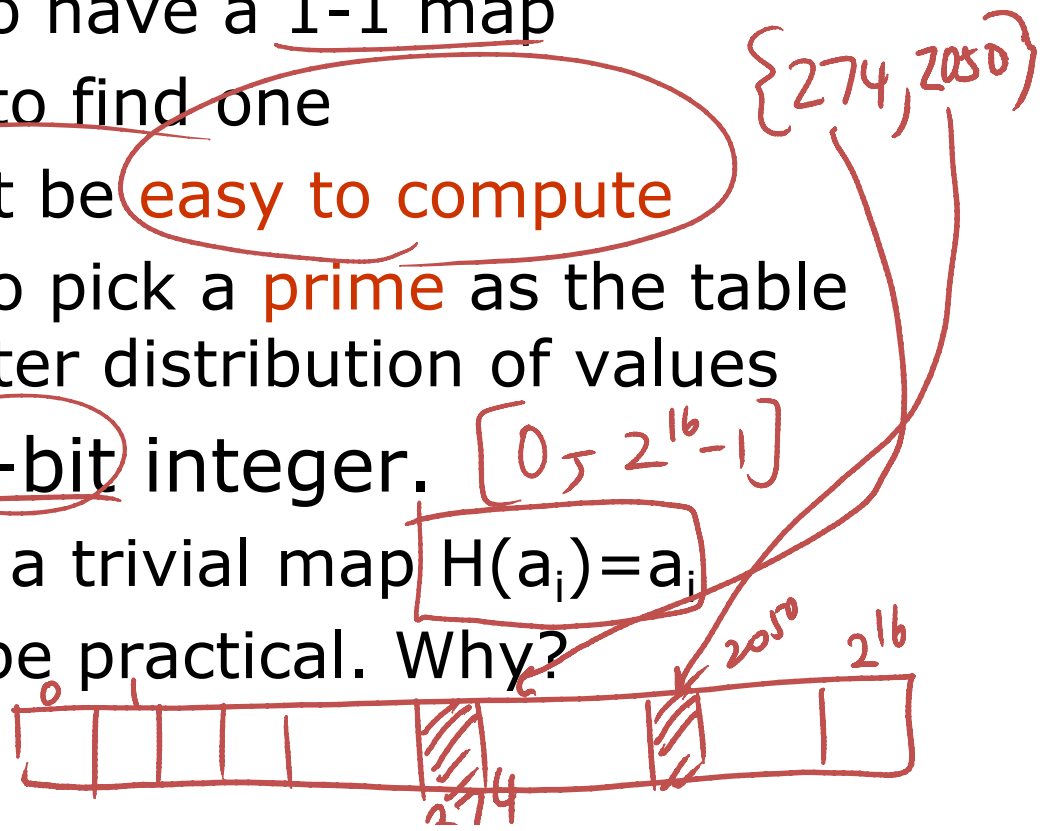




$p = m \bmod n$

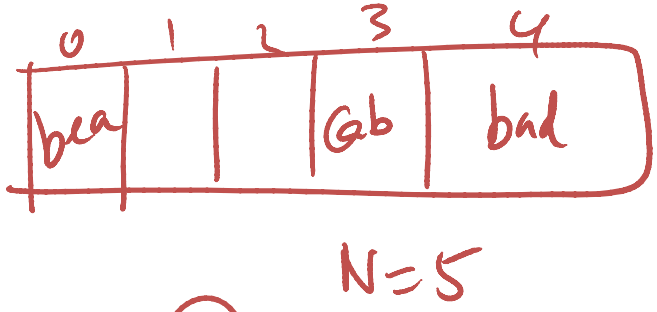
Big Idea in Hashing

- Let $S=\{a_1, a_2, \ldots a_m\}$ be a set of objects that we need to map into a table of size N.
  - Find a function such that $H: S \rightarrow [1 \ldots n]$
  - Ideally we'd like to have a 1-1 map
  - But it is not easy to find one
  - Also function must be easy to compute
  - It is a good idea to pick a prime as the table size to have a better distribution of values
- Assume a_i is a 16-bit integer.
  - Of course there is a trivial map $H(a_i)=a_i$
  - But this may not be practical. Why?



Finding a hash Function

- Assume that N = 5 and the values we need to insert are: cab, bea, bad etc.
- Let a=0, b=1, c=2, etc
- Define H such that
  - H[data] = (∑ characters) Mod N
- H[cab] = (2+0+1) Mod 5 = 3
- H[bea] = (1+4+0) Mod 5 = 0
- H[bad] = (1+0+3) Mod 5 = 4



Collisions

- What if the values we need to insert are “abc”, “cba”, “bca” etc…
  - They all map to the same location based on our map H (obviously H is not a good hash map)
- This is called “Collision”
- When collisions occur, we need to “handle” them
- Collisions can be reduced with a selection of a good hash function

Choosing a Hash Function

to reduce/avoid Collisions

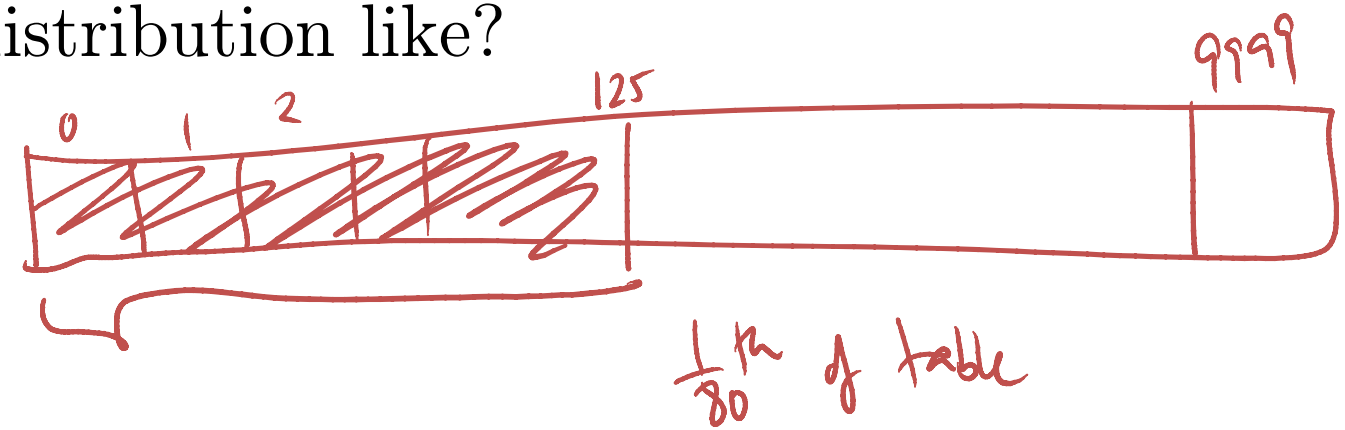
- A good hash function must
  - Be easy to compute
  - Avoid collisions
- How do we find a good hash function?
- A bad hash function
  - Let S be a string and $H(S) = \sum_{i=0}^{n-1} S_i$ where S_i is the i^{th} character of S
  - Why is this bad?

$S_0 S_1 S_2 \ldots S_{n-1}$

Any permutation of S_i's will create the same value (i.e. Collision)

Choosing a Hash Function?

- Question
  - Think of hashing 10000, 5-letter words into a table of size 10000 using the map H defined as follows.
  - $H(a_0a_1a_2a_3a_4) = \Sigma\ a_i$ (i=0,1....4) $0 \leq \leq 25 \times 5$
  - If we use H, what would be the key distribution like?



Choosing a Hash Function

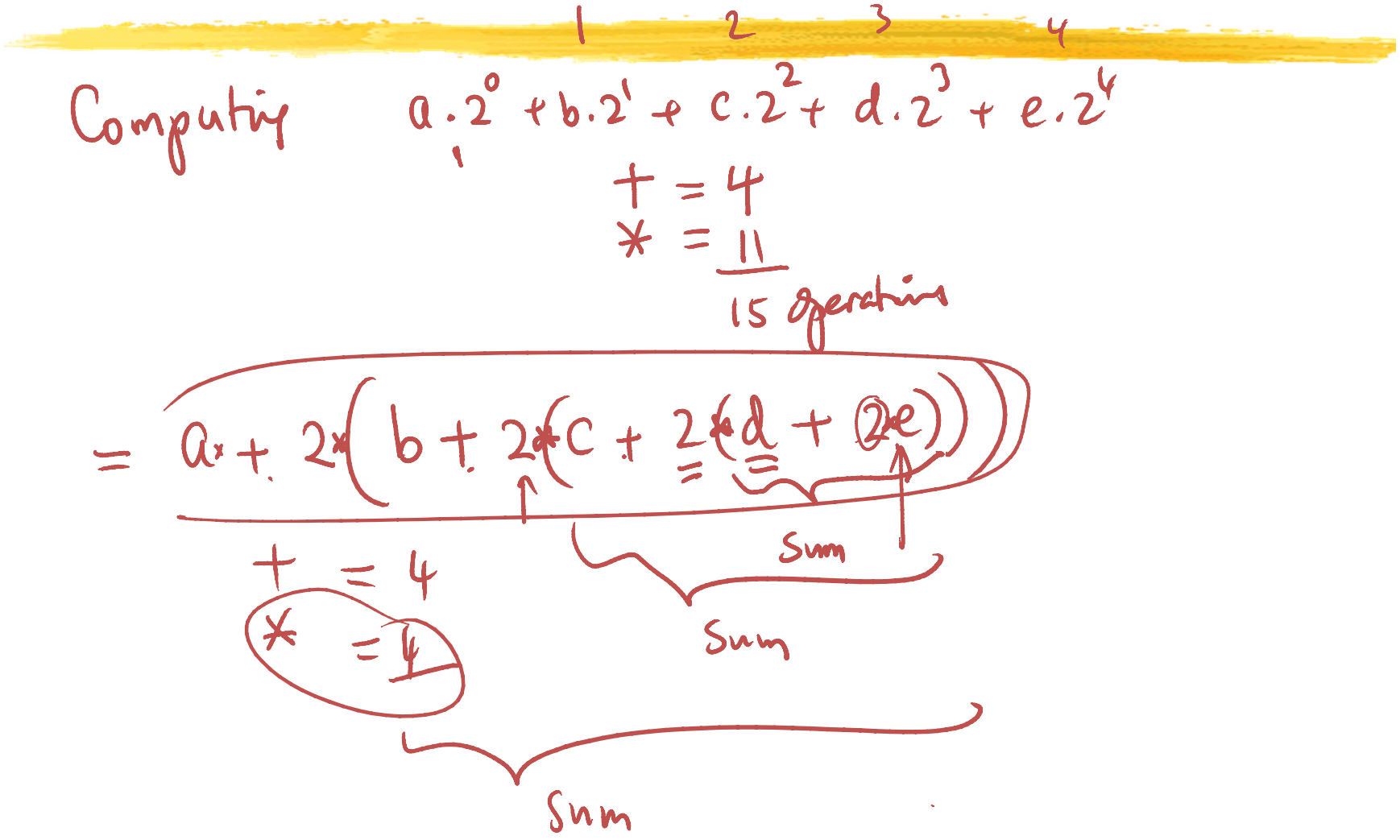
- Suppose we need to hash a set of strings $S = \{S_i\}$ to a table of size N
- $H(S_i) = (\Sigma\ S_i[j].d^j)$ mod N, where $S_i[j]$ is the j^{th} character of string S_i
  - How expensive is to compute this function?
    - cost with direct calculation

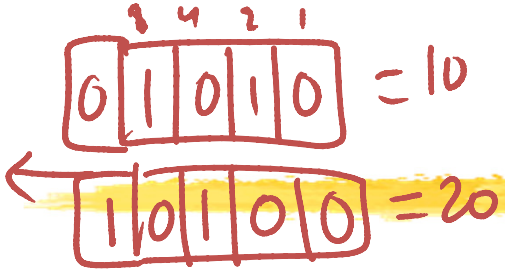
a=0, b=1, c=2

$H(abc) = a \cdot 2^0 + b \cdot 2^1 + c \cdot 2^2 = 10 \ \%5 = 0$

$H(bac) = b \cdot 2^0 + a \cdot 2^1 + c \cdot 2^2 = 9 \ \%5 = 4$

    - Is it always possible to do direct calculation?
  - Is there a cheaper way to calculate this? Hint: use Horners Rule.


1 2 3 4
Computing $a \cdot 2^0 + b \cdot 2^1 + c \cdot 2^2 + d \cdot 2^3 + e \cdot 2^4$
+ = 4
* = 11
15 operations
= a + 2 * (b + 2 * (c + 2 * (d + 2 * e)))
+ = 4
* = 4
Sum
Sum
Sum

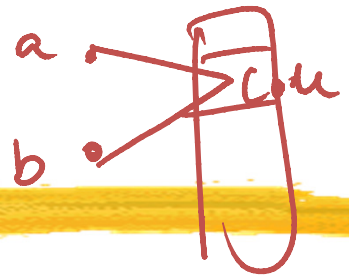


Code

```
public static int hash(String key, int n){
    int value = 0;
    for (int i=0; i<key.length(); i++)
        value = (value*128+ key.charAt(i))%n;
    return value;
}
```

- What does this function return if "guna" is hashed into a table of size 101?
- What is the complexity of code in terms of string length?
- What are some of the problems with this function?

Collisions



- Hash functions can be *many-to-1*
  - They can map different search keys to the same hash key.

    `hash1(`a`) == 9 == hash1(`w`)`

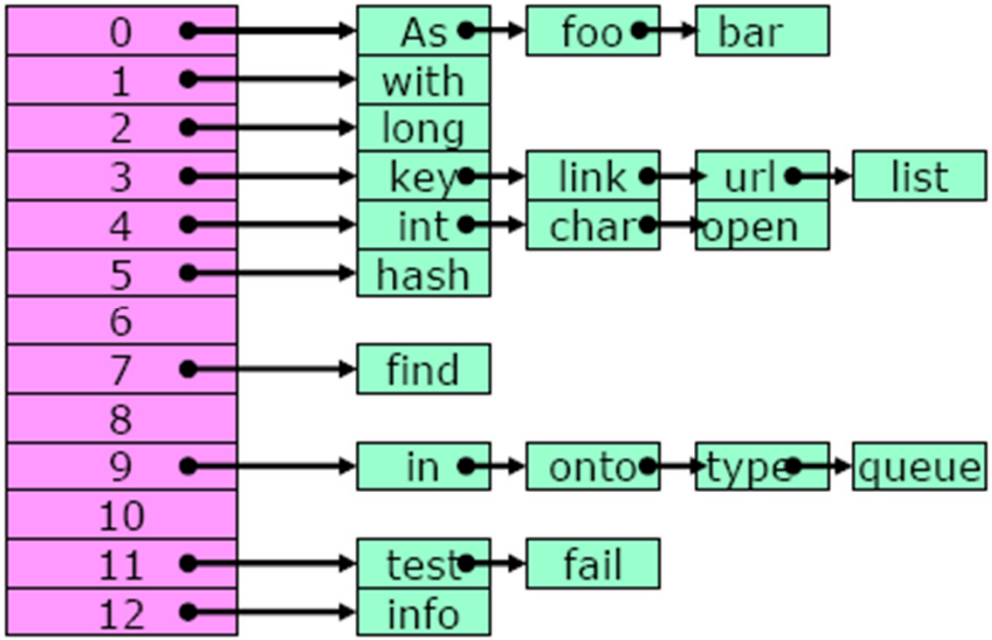
- Must compare the search key with the record found
  - If the match fails, there is a **collision**

Collision Resolving strategies

- Separate chaining
- Open addressing
  - Linear Probing
  - Quadratic Probing
  - Double Probing
  - *Etc.*

Separate Chaining

- Collisions can be resolved by creating a list of keys that map to the same value



Separate Chaining

- Use an array of linked lists
  - LinkedList[ ] Table;
  - Table = new LinkedList(N), where N is the table size
- Define Load Factor of Table as
  - λ = number of keys/size of the table

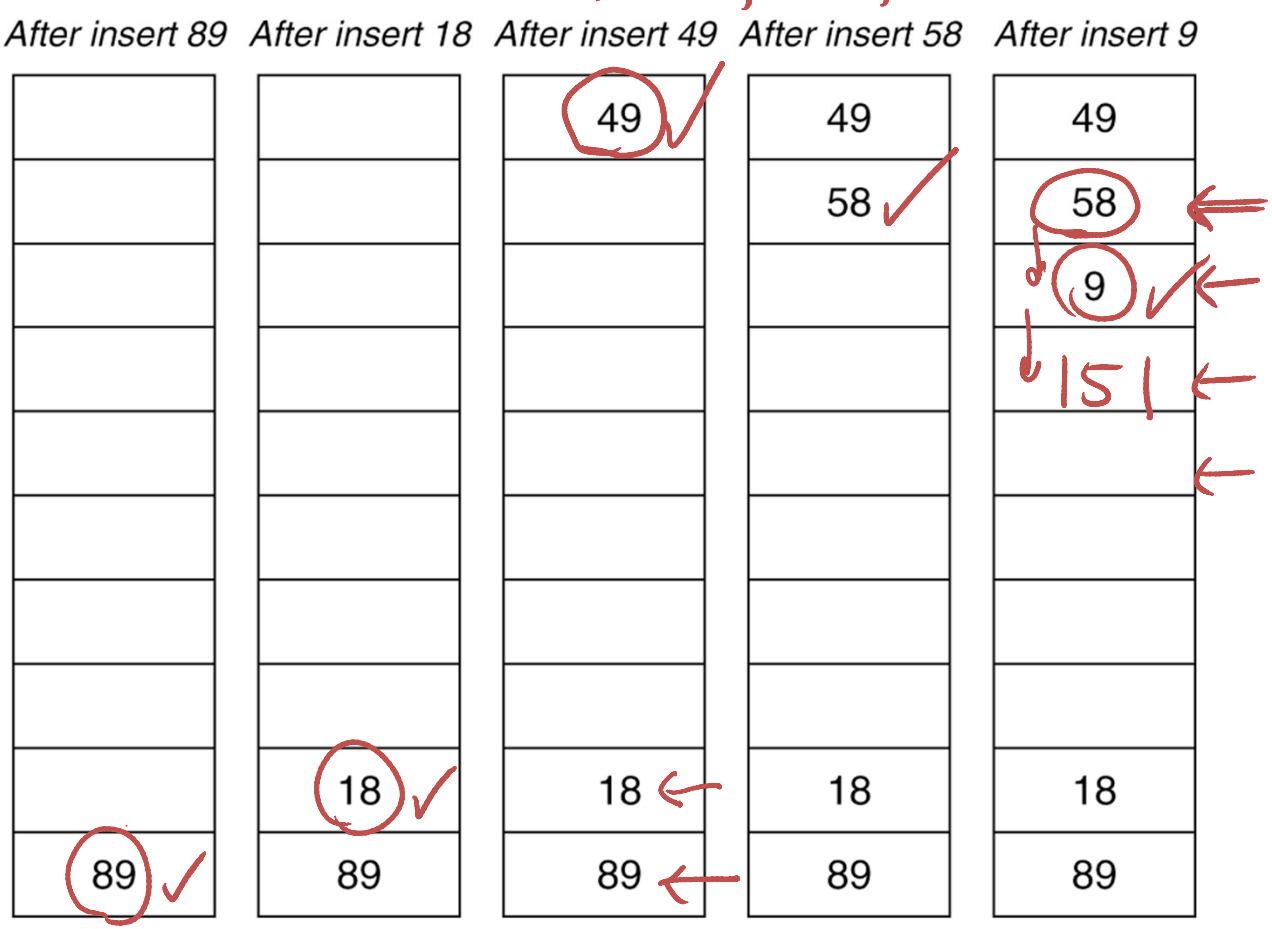
    (λ can be more than 1)
- Still need a good hash function to distribute keys evenly
  - For search and updates

Linear Probing

- The idea:
  - Table remains a simple array of size N
  - On **insert(x)**, compute f(x) mod N, if the cell is full, find another by sequentially searching for the next available slot
    - Go to f(x)+1, f(x)+2 etc..
  - On **find(x)**, compute f(x) mod N, if the cell doesn't match, look elsewhere.
  - Linear probing function can be given by
    - h(x, i) = (f(x) + i) mod N (i=1,2,….)

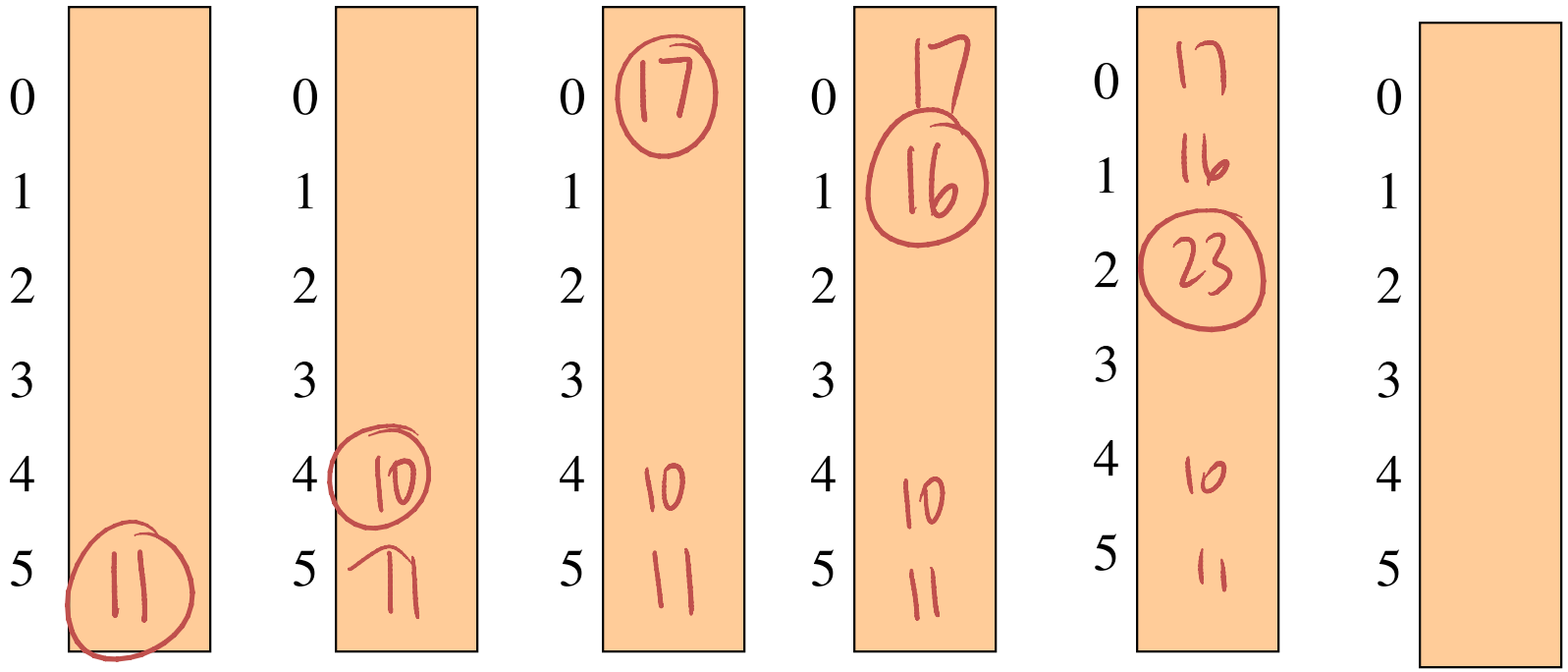
Figure 20.4
Linear probing hash table after each insertion

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```



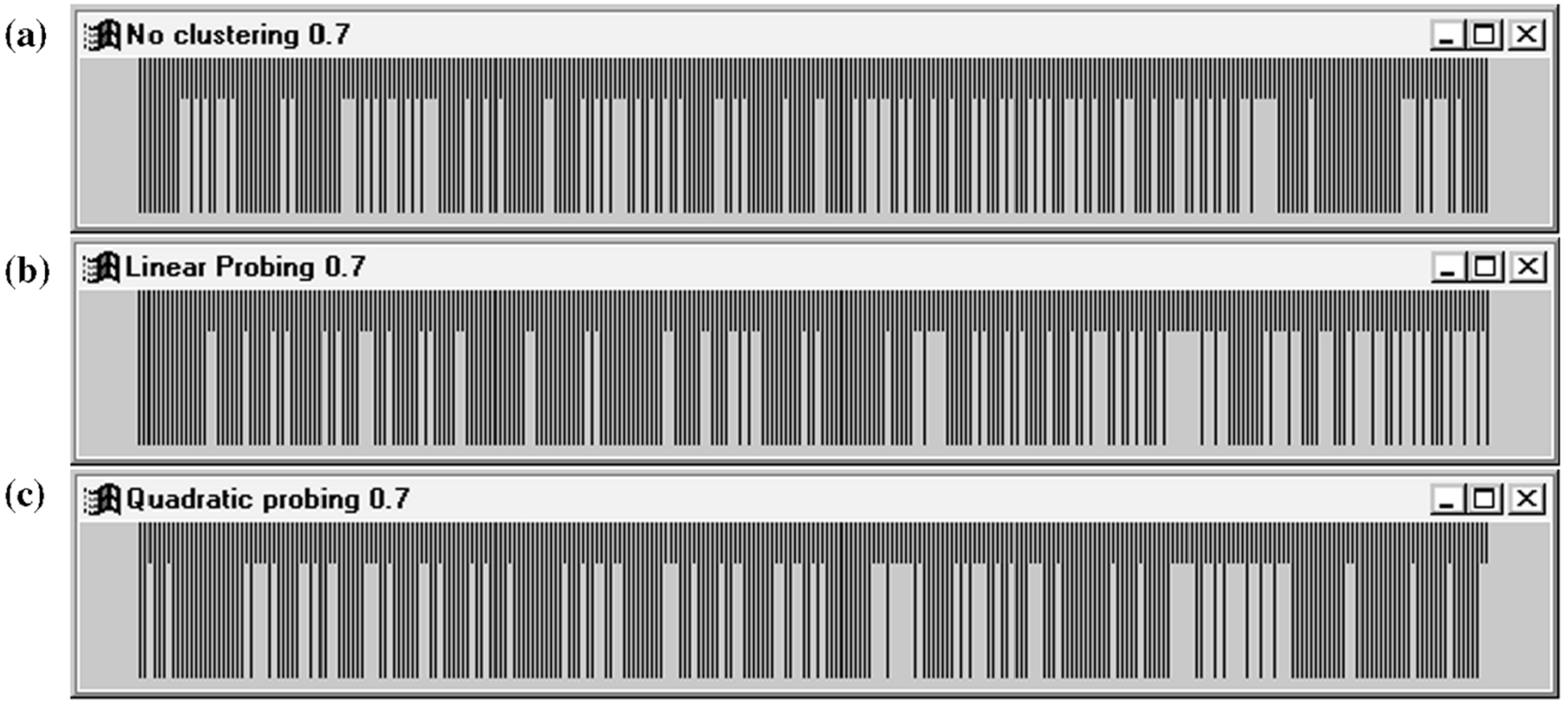
Linear Probing Example

- Consider H(key) = key Mod 6 (assume N=6)
- H(11)=5, H(10)=4, H(17)=5, H(16)=4,H(23)=5
- Draw the Hash table



Clustering Problem

- Clustering is a significant problem in linear probing. Why?
- Illustration of primary clustering in linear probing (b) versus no clustering (a) and the less significant secondary clustering in quadratic probing (c). Long lines represent occupied cells, and the load factor is 0.7.

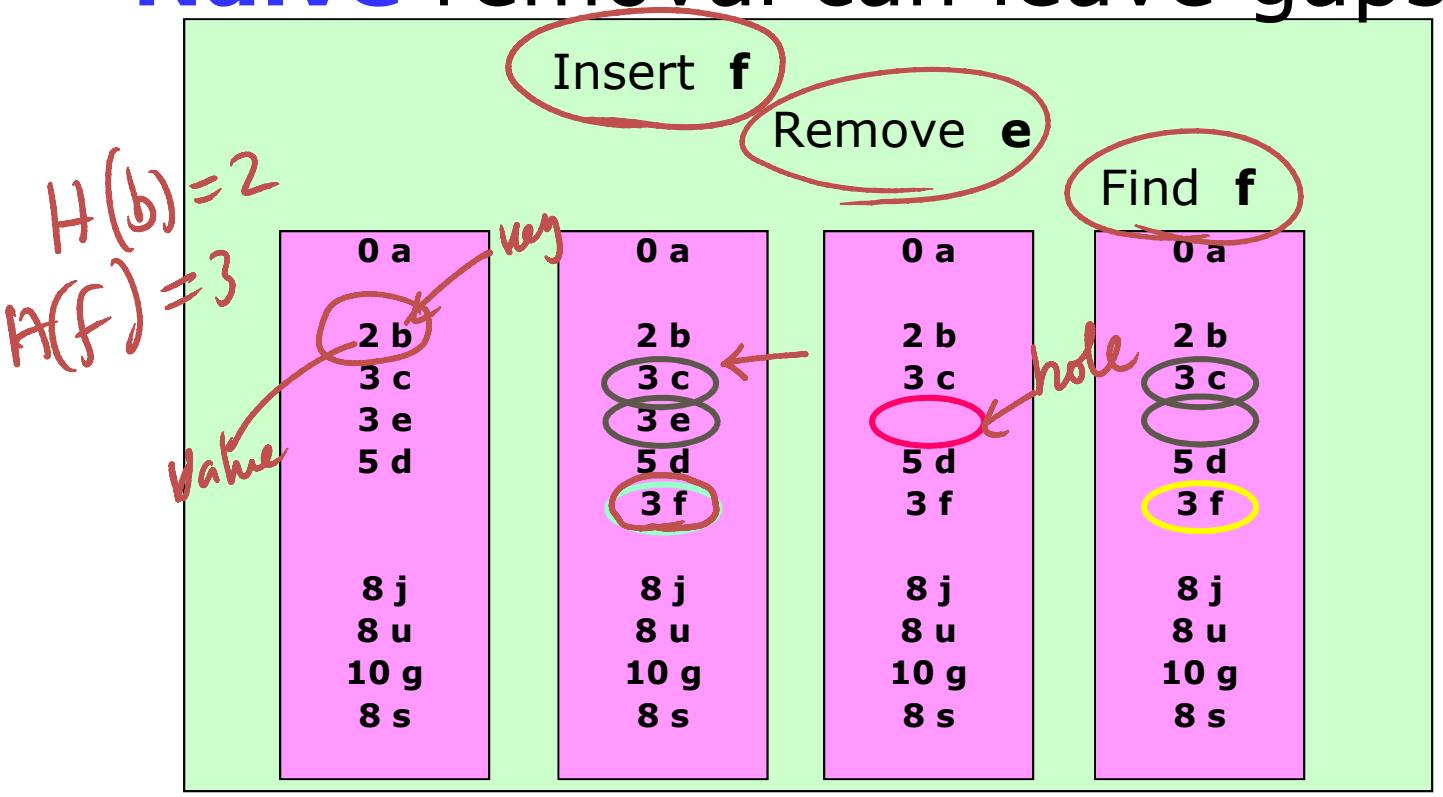


Linear Probing

- How about deleting items from Hash table?
  - Item in a hash table connects to others in the table(eg: BST).
  - Deleting items will affect finding the others
  - "Lazy Delete" – Just mark the items as inactive rather than removing it.

Lazy Delete

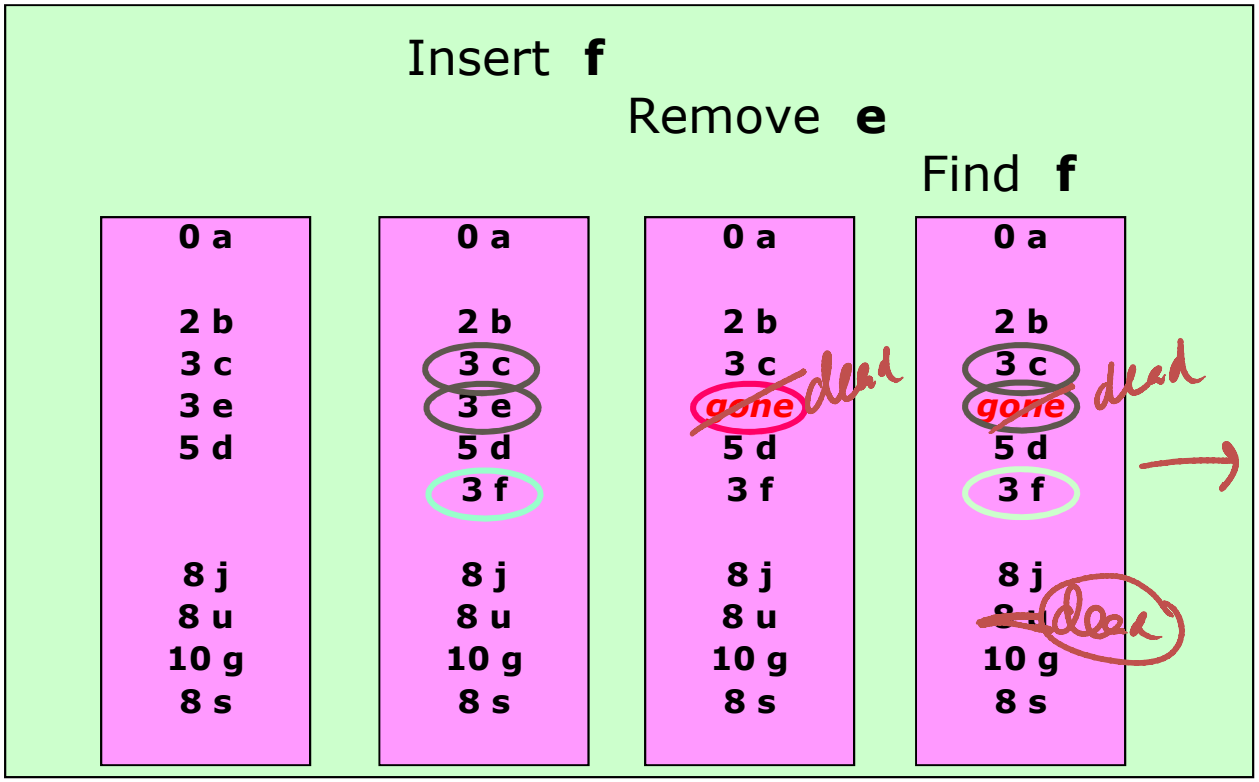
- **Naïve** removal can leave gaps!



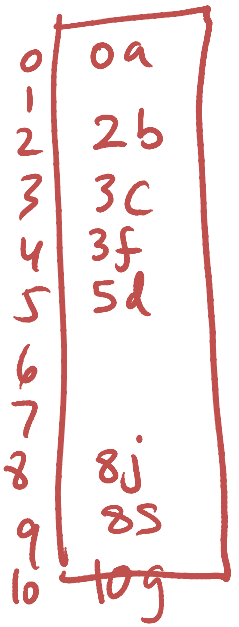
"3 f" means *search key* f and *hash key* 3

Lazy Delete

- **Clever** removal



O(n)



"3 f" means *search key* f and *hash key* 3

Load Factor (open addressing)

- **definition:** The *load factor* λ of a probing hash table is the fraction of the table that is full. The load factor ranges from 0 (empty) to 1 (completely full).
- It is better to keep the load factor under 0.7
- Double the table size and rehash if load factor gets high
- Cost of Hash function f(x) must be minimized
- When collisions occur, linear probing can always find an empty cell
  - But clustering can be a problem

Quadratic Probing

Quadratic probing

- Another open addressing method
- Resolve collisions by examining certain cells (1,4,9,…) away from the original probe point
- Collision policy:
  - Define $h_0(k)$, $h_1(k)$, $h_2(k)$, $h_3(k)$, …
    where $h_i(k) = (\text{hash}(k) + i^2) \bmod \text{size}$
- Caveat:
  - May not find a vacant cell!
    - Table must be less than half full ($\lambda < ½$)
  - (Linear probing always finds a cell.)

Quadratic probing

- Another issue
  - Suppose the table size is 16.
  - Probe offsets that will be tried:

```
 1      mod 16  =  1
 4      mod 16  =  4
 9      mod 16  =  9
16      mod 16  =  0
25      mod 16  =  9
36      mod 16  =  4
49      mod 16  =  1
64      mod 16  =  0
81      mod 16  =  1
```

only four different values!

Figure 20.6

A quadratic probing hash table after each insertion (note that the table size was poorly chosen because it is not a prime number).

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89