

- **Double Hashing**

- **Idea:** When a collision occurs, increment the index (mod tablesize), just as in linear probing. However, now do not automatically choose 1 as the increment value
  - > Instead use a second, different hash function ( $h2(x)$ ) to determine the **increment**
- **This way keys that hash to the same location will likely not have the same increment**
  - >  $h1(x1) == h1(x2)$  with  $x1 \neq x2$  is bad luck (assuming a good hash function)
  - > However, ALSO having  $h2(x1) == h2(x2)$  is REALLY bad luck, and should occur even less frequently
  - > It also allows for a collided key to move (mostly – depending on  $h2(x)$ ) **anywhere in the table**
- See example on next slide



Index	Value	Probes
0		
1		1
2		
3		1
4		1
5		1
6		1
7		
8		2
9		
10		2

## Double Hashing Example

*Compare to Slide 18 of  
Lecture 5*

14	$h(x) = 3$	
17	$h(x) = 6$	
25	$h(x) = 3$	$h_2(x) = 5$
37	$h(x) = 4$	
34	$h(x) = 1$	
16	$h(x) = 5$	
26	$h(x) = 4$	$h_2(x) = 6$

$$h(x) = x \bmod 11$$

$$h_2(x) = (x \bmod 7) + 1$$



## Double Hashing

- Note that we still get collisions with DH
  - And even multiple collisions in one operation
  - In this case we iterate just as we do with LP, using the DH increment multiple times
- However, because  $h_2(x)$  varies for different keys, it allows us to spread the data throughout the table, **even after an initial collision**
- But we must be careful to ensure that double hashing always "works"
  - Make sure increment is  $> 0$ 
    - > Note the +1 in our  $h_2(x)$ :  $h_2(x) = (x \bmod 7) + 1$
    - > Our mod operator can result in 0, which is fine for an absolute address, but not for an increment!



## Double Hashing

- Make sure **no index is tried twice before all are tried once**
  - > Why? Think about this?
  - > Consider table to right and assume:
    - >  $h(Z) = 3$  and  $h_2(Z) = 2$
  - > What would happen when we search the table?
  - > How can we fix this?
    - > Make M a prime number
- Note that these were not issues for linear probing, since the increment is clearly  $> 0$  and if our increment is 1 we will clearly try all indices once before trying any twice

Index	Value
0	
1	V
2	
3	W
4	
5	X
6	
7	Y



## Collision Resolution

- As  $\alpha$  increases, double hashing shows a definite improvement over linear probing
  - Discuss
- However, as  $\alpha \rightarrow 1$  (or as  $N \rightarrow M$ ), **both schemes degrade to Theta(N)** performance
  - Since there are only M locations in the table, as it fills there become fewer empty locations remaining
  - Multiple collisions will occur even with double hashing
  - This is especially true for **inserts** and **unsuccessful finds**
    - > Both of these continue **until an empty location is found**, and few of these exist
    - > Thus it could take close to M probes before the collision is resolved
    - > Since the table is almost full  $\text{Theta}(M) = \text{Theta}(N)$



### ► Open Addressing Issues

- We have just seen that **performance degrades as  $N$  approaches  $M$** 
  - Typically for open addressing we want to keep the table partially empty
    - > For linear probing,  $\alpha = 1/2$  is a good rule of thumb
    - > For double hashing, we can go a bit higher (3/4 or more)
  - How can we do this?
    - > Monitor the logical size (number of entries) vs. physical size (array length) to calculate  $\alpha$
    - > **Resize the array and rehash all of the values** when  $\alpha$  gets past the threshold
    - > Rehashing all of the data seems like a LOT of work!
    - > Is this better than leaving it as is?
      - > We will discuss



## Open Addressing Issues

- What about **delete**?
  - Why is this a problem?
  - Consider the LP table on the right and assume  $H(Z) == 2$  but it was placed in index 4 due to a collision
  - Search for Z would try 2, 3, 4, finding Z at location 4
  - Now delete(Y) and search for Z again
    - > Search would stop at index 3 with not found even though Z is present
  - Deleting Y broke the chain
- How can we fix this?

Index	Value
0	
1	W
2	X
3	Y
4	Z
5	
6	
7	



## Open Addressing Issues

- One solution (see p. 471 of text)
  - > Rehash all keys from deleted key to end of cluster
  - > Note that in this case Z still hashes to 2 and will move to position 3 and once again be within the chain
  - > Will this be a lot of work?
    - > Discuss
- Will not work with double hashing though – **why?**
  - What can we do with double hashing?
    - > Discuss

Index	Value
0	
1	W
2	X
3	Y
4	Z
5	
6	
7	





## Open Addressing Issues

- Can we use hashing without delete?
  - > Yes, in some cases (ex: compiler using language keywords)
  - > We build a hash table, use it for searches, and then throw it away entirely
    - > We never delete individual items



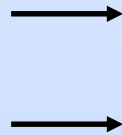
- Closed Addressing

- ▶ Recall that in this scheme, each location in the hash table represents a **collection of data**
  - If we have a collision we resolve it within the collection, without changing hash addresses
- ▶ Most common form is **separate chaining**
  - Use a simple **linked-list** at each location in the table
    - Look at example
      - > Using the same data that we previously used for linear probing and separate chaining
    - **Discuss placement of nodes in chain**



## Separate Chaining

Index	Value
0	
1	→
2	
3	→
4	→
5	→
6	→
7	
8	
9	
10	



14	$h(x) = 3$
17	$h(x) = 6$
25	$h(x) = 3$
37	$h(x) = 4$
34	$h(x) = 1$
16	$h(x) = 5$
26	$h(x) = 4$

$$h(x) = x \bmod 11$$



### ► Performance of separate chaining?

- Performance is dependent upon **chain length**
- Clearly a **not found** search must traverse entire chain
  - Chain length is determined by the load factor,  $\alpha$ 
    - > Ave chain length = (total # of nodes)/(M)
    - > But (total # of nodes) == N so
    - > Ave chain length ==  $N/M = \alpha$
  - As long as  $\alpha$  is a small constant, performance is still  $\Theta(1)$ 
    - > Ex:  $N = 150, M = 100 \rightarrow \alpha = 1.5$ 
      - > This is still clearly  $\Theta(1)$
      - > Note also that N can now be greater than M
    - > More **graceful degradation** than open addressing schemes



## Separate Chaining

- However, if  $N \gg M$ , then it can still degrade to  $\Theta(N)$  performance
  - > Ex:  $N = 1000, M = 10 \rightarrow \alpha = 100$
  - > Thus we **may still need to resize the array** when  $\alpha$  gets too big
- A **poor hash function** can also degrade this into  $\Theta(N)$ 
  - > Think about what will happen in this case
  - > Discuss
- Can we develop a closed addressing scheme that can mitigate the damage caused by a poor hash function?
  - Think about this!



### ► What if we used "better" collections at each index?

- Sorted array?
  - Space overhead if we make it large and copying overhead if we need to resize it
  - Inserts require shifting
- BST?
  - Could work
    - > Now a poor hash function would lead to a large tree at one index – **still  $\Theta(\log N)$**  as long as tree is relatively balanced
- But is it worth it?
  - Not really – separate chaining is simpler (less overhead) and we want a good hash function anyway
  - In this case we should **fix the hash function**

