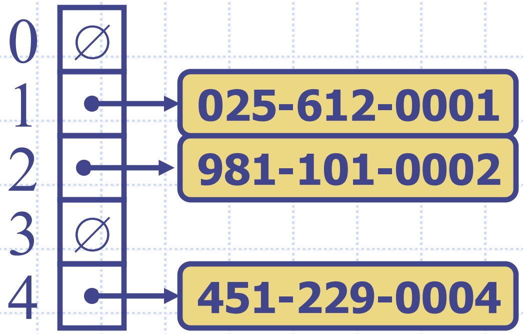
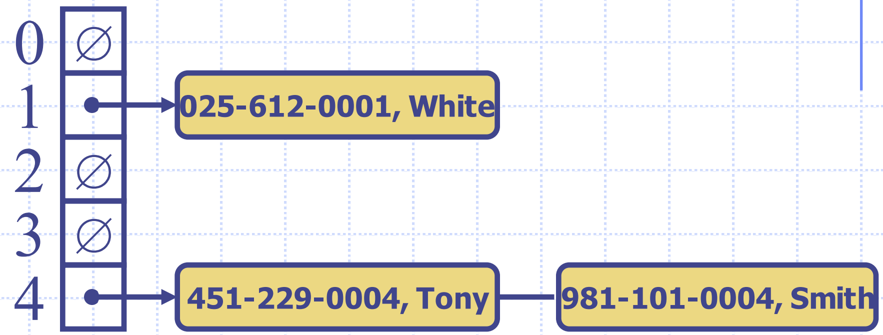
Hash Tables 4



Collision Handling Review: 1 Separate Chaining

- Collisions occur when different elements are mapped to the same cell
- Separate Chaining: let each cell in the table point to a linked list of entries that map there



- Separate chaining is simple, but requires additional memory outside the table

Second Approach for Handling Collisions --- Open Addressing

- Open addressing: the colliding item is placed in a different cell in **the table (The container)**
- **We have two different schemes in Open Addressing,**
  - 1)Linear Probing
  - 2)Double Hashing

Second Approach for Handling Collisions --- Open Addressing

- 1)Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell,
  - When probing reaches the end of the array, it wraps around to the beginning until it finds an available spot.
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes.

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

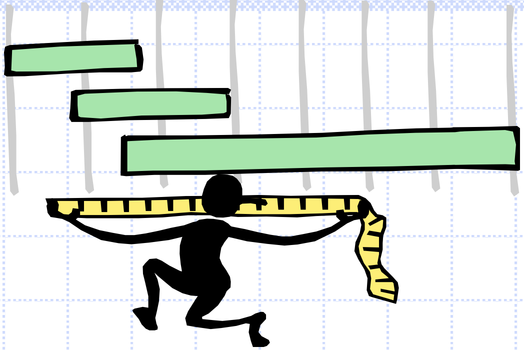
Second Approach for Handling Collisions --- Open Addressing

- 2) Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available **cell of the series,**

  $(i + jd(k)) \bmod N$, for $j = 0, 1, \ldots, N-1$

- **Where i** is the hash value of first hash function ***h().***
- The secondary hash function $d(k)$ cannot have zero values.
- The table size N must be a prime to allow probing of all the cells.
- E.g. If h(k) yields a value i = 10, and d(k) yields a value 17, and N = 101, Then the double hashing scheme will try these sequence of locations to find the first available spot,
  - {10, 27, 44, 61, 78, 95, 11, ...}
  - j= {0 , 1, 2, 3, 4, 5, 6,.......}

Performance of Hashing



- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the map collide, which nearly impossibly happens in real-world.
- The load factor $\alpha = n/N$ affects the performance of a hash table, where n is the actual number records in the table.
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is: $1 / (1 - \alpha)$
- The expected running time of all the hash table ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100% (better <70%)
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

When Hash Table is Useful

- Items in hash table usually do not have order,
  - Like a bag of items.
  - If you need to maintain order for a collection, hash table may not suitable.
  - But in Java, TreeMap, behaving like a hash table, maintains an order for all keys in a TreeMap.