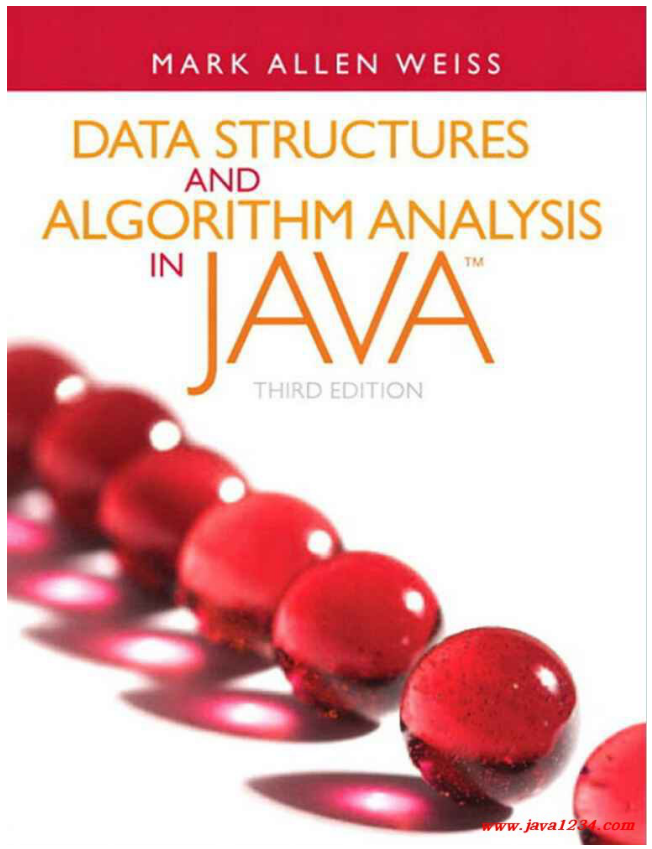
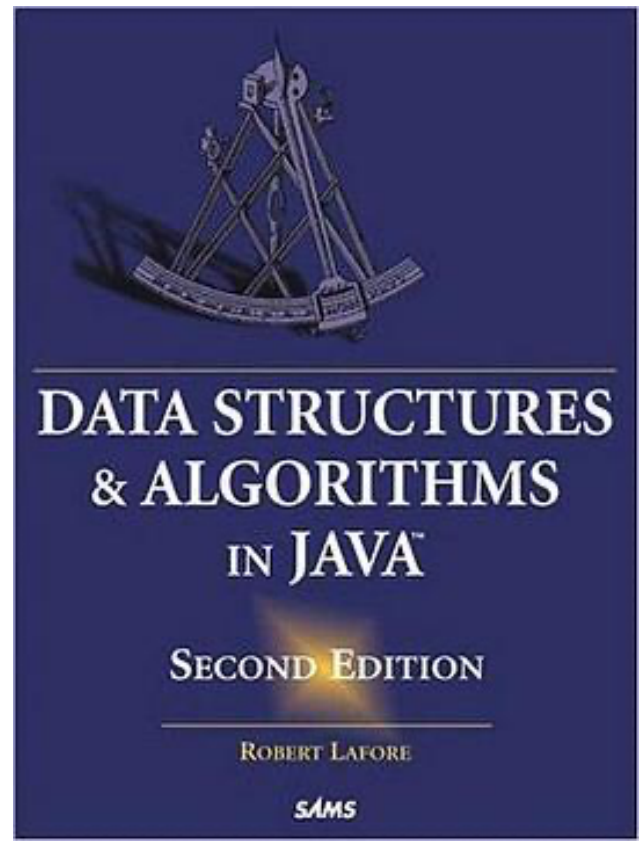
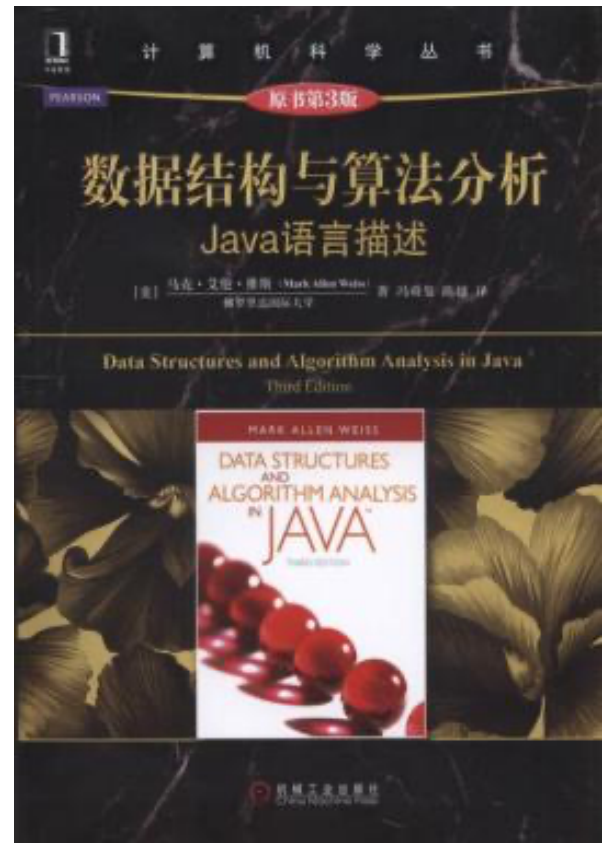
Topic 11 – Hash







Motivation

- In previous classes, we were able to make improvements in our search algorithms by taking advantage of information about where items are stored in the collection with respect to one another.
- For example, by knowing that a list was ordered, we could search in logarithmic time using a binary search.
- In this section we will attempt to go one step further by building a data structure that can be searched in $O(1)$ time.
- This concept is referred to as **hashing**.

- Please log in to your “MOODLE SYSTEM” to answer the homework questions.
  - **Note there are some problem on the “MOODLE SYSTEM” .**
  - **So, our online homework will assign later.**

Pre Homework for 8:20

- Suppose that n integers are stored at a Data Structure. What is the time complexity if we want to check an element is in it or not.
- Note use big O notation.
- (a) This Data Structure is an ordered array
- (b) This Data Structure is an unordered array
- (c) This Data Structure is a linked list.

Pre Homework for 10:20

- Suppose that n integers are stored at a Data Structure. What is the time complexity if we want to check an element is in it or not.
- Note use big O notation.
- (a) This Data Structure is a linked list.
- (b) This Data Structure is an unordered array
- (c) This Data Structure is an ordered array

- Before we continue to explain the course content, please watch the videos.
  - Hashes 1 Introduction_480p.mp4
  - Hashes 2 Hash Functions_480p.mp4
  - Hashes 3 Collisions_480p.mp4
- Note that those files have uploaded to our QQ group.

Collision Problem for 8:20

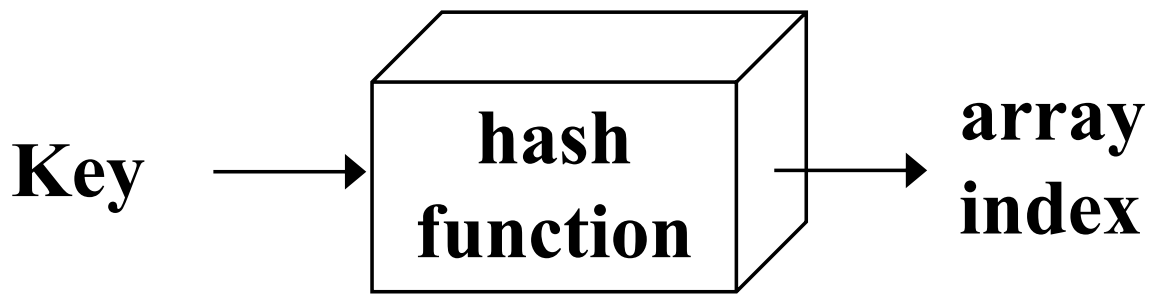
- Suppose that f is a hashing function for integers as follows.
- Which f has collision?
- (a) f(x) = x^2 + 5
- (b) f(x) = sin(x)
- (c) f(x) = x – 100
- (d) f(x) = x^3

Collision Problem for 10:20

- Suppose that f is a hashing function for integers as follows.
- Which f has collision?
- (a) f(x) = x^5
- (b) f(x) = 3x + 90
- (c) f(x) = 4x^2 + 3
- (d) f(x) = $\left\lceil \frac{x+1}{2} \right\rceil$

Hash function

- Hash function is a function that maps key into value (array index).
- We always suppose that the comput time of hash function is quickly. That is ***O*(1)**.



Hash Table

- A **hash table** is a collection of items which are stored in such a way as to make it easy to find them later.
- Each position of the hash table, often called a **slot**, can hold an item and is named by an integer value starting at 0.

Hash Table

- For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on.
- Initially, the hash table contains an entry - 1 so every slot is -1.
- Figure shows a hash table of size m =11.
- In other words, there are m slots in the table, named 0 through 10.

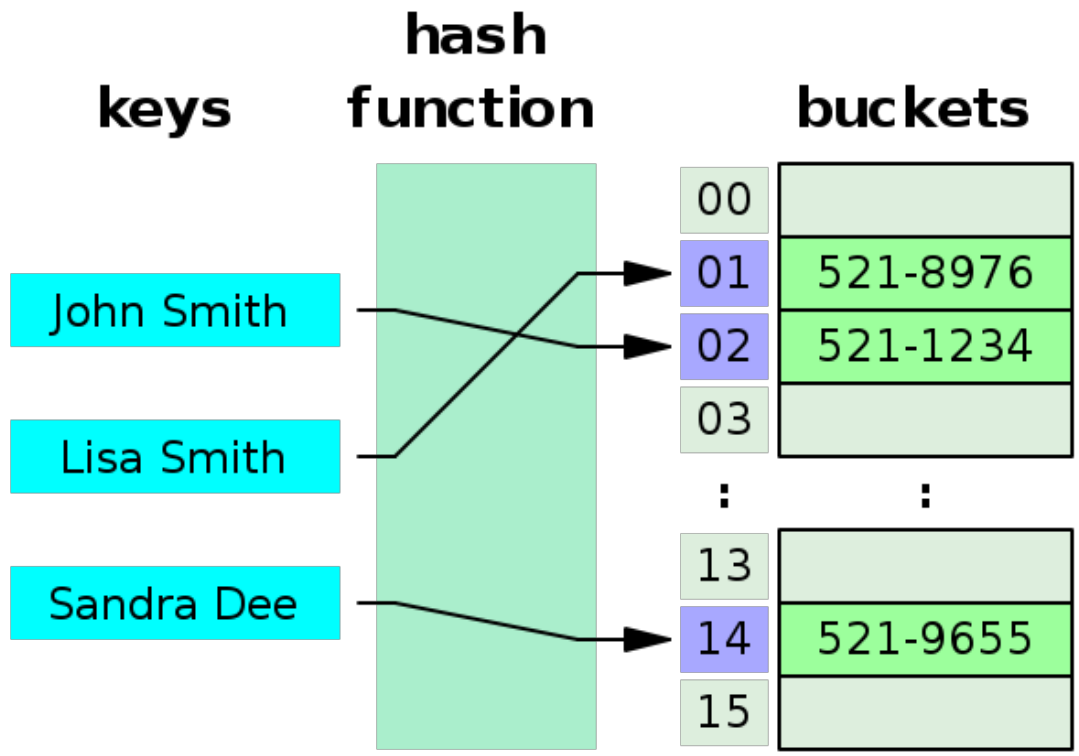
slot	0	1	2	3	4	5	6	7	8	9	10
entry	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Hash

- **Hashing** is a technique that is used to uniquely identify a specific object from a group of similar objects.
- Some examples of how hashing is used in our lives include:
    - In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
    - In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.
- In both these examples the students and books were hashed to a unique number.

Hash

- Assume that you have an object and you want to assign a key to it to make searching easy.
- To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values.
- However, in cases where the keys are large and cannot be used directly as an index, you should use ***hashing***.



Hash

- In hashing, large keys are converted into small keys by using **hash functions**.
- The values are then stored in a data structure called **hash table**.
- The idea of hashing is to distribute entries (key/value pairs) uniformly across an array.
- Each element is assigned a key (converted key).
- By using that key you can access the element in **O(1)** time.
- Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hash

- Hashing is implemented in two steps:
  1. An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.
  2. The element is stored in the hash table where it can be quickly retrieved using hashed key.

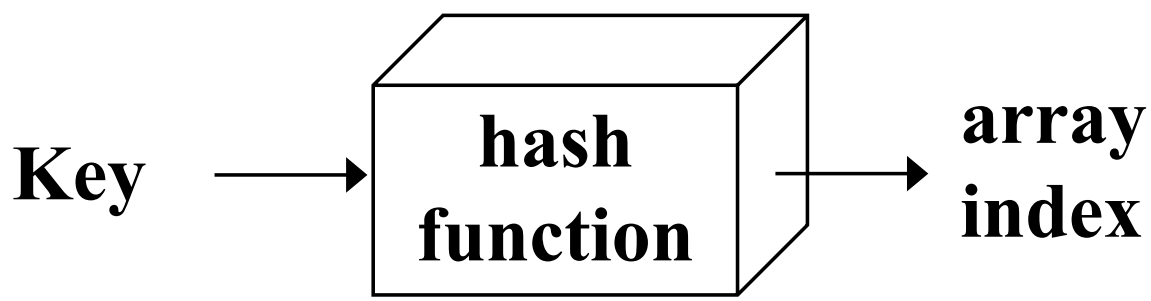
- hash = hashfunc(key)
- index = hash % array_size

- In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array_size − 1) by using the modulo operator (%).
  - hash = hashfunc(key)
  - index = hash % array_size

slot	0	1	2	3	4	5	6	7	8	9	10
entry	6	8	7	5	4	3	1	1	55	2	0

Hash function

- The mapping between an item and the slot where that item belongs in the hash table is called the **hash function**.
- The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and *m*-1.



Hash function

- Assume that we have the set of integer items 54, 26, 93, 17, 77, and 31.
- Our first hash function, sometimes referred to as the "remainder method," simply takes an item and divides it by the table size, returning the remainder as its hash value
  - h (key) = key % 11

- Table gives all of the hash values for our example items.

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

Hash function

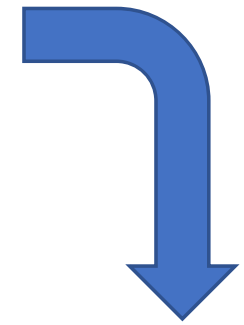
- h (key) = key % 11

- Table gives all of the hash values for our example items.
- Note that this remainder method (modulo arithmetic) will typically be present in some form in all hash functions, since the result must be in the range of slot names.

item	Hash Value	Note
54	10	10 = 54 % 11
26	4	4 = 26 % 11
93	5	5 = 93 % 11
17	6	6 = 17 % 11
77	0	0 = 77 % 11
31	9	9 = 31 % 11

Hash Table with Six Items

item	Hash Value	Note
54	10	10 = 54 % 11
26	4	4 = 26 % 11
93	5	5 = 93 % 11
17	6	6 = 17 % 11
77	0	0 = 77 % 11
31	9	9 = 31 % 11



- Hash Table

slot	0	1	2	3	4	5	6	7	8	9	10
entry	77	None	None	None	26	93	17	None	None	31	54

Hash Table with Six Items

- Once the hash values have been computed, we can insert each item into the hash table at the designated position as shown in the following.

slot	0	1	2	3	4	5	6	7	8	9	10
entry	77	None	None	None	26	93	17	None	None	31	54

- Note that 6 of the 11 slots are now occupied.
- This is referred to as the **load factor**, and is commonly denoted by $\lambda = \frac{number\ of\ items}{size\ of\ table}$.
- For this example, $\lambda = \frac{6}{11}$

Load Factor

- $\lambda = 0$
  - Empty
- $\lambda = 0.5$
  - a half data
- $\lambda = 1$
  - Our data is full

Hash Table with Six Items

- Now when we want to search for an item, we simply use the hash function to compute the slot name for the item and then check the hash table to see if it is present.
- This searching operation is $O(1)$, since a constant amount of time is required to compute the hash value and then index the hash table at that location.
- If everything is where it should be, we have found a constant time search algorithm.

Hash Table with Six Items

- You can probably already see that this technique is going to work only if each item maps to a unique location in the hash table.
- For example, if the item 44 had been the next item in our collection, it would have a hash value of 0 (44 % 11 = 0).
- Since 77 also had a hash value of 0, we would have a problem.
- According to the hash function, two or more items would need to be in the same slot.
- This is referred to as a **collision** (it may also be called a "clash").
- Clearly, collisions create a problem for the hashing technique.
- We will discuss them in detail later.

- Please watch the videos.
  - Hashes 4 Hash Functions for Strings_480p.mp4
  - Hashes 5 Compressing numbers to fit the size of the array_480p.mp4
  - Hashes 6 Make an integer positive_480p.mp4
- Note that those files have uploaded to our QQ group.

Hash function

- A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table.
- The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.
- To achieve a good hashing mechanism, it is important to have a good hash function with the following basic requirements:
  - Easy to compute
  - Uniform distribution
  - Less **collisions**

Hash function

- To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:
  - Easy to compute: It should be easy to compute and must not become an algorithm in itself.
  - Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.
  - Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.
    - Collision: $f(a) = f(b)$ for some $a \neq b$

Hash function

- **Note**:
  - Irrespective of how good a hash function is, collisions are bound to occur.
  - Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

Need for a good hash function

- Let us understand the need for a good hash function.
- Assume that you have to store strings in the hash table by using the hashing technique {“abcdef”, “bcdefa”, “cdefab” , “defabc” }.

Need for a good hash function

- To compute the index for storing the strings, use a hash function that states the following:
  - The index for a specific string will be equal to the sum of the ASCII values of the characters modulo 599.
  - As 599 is a prime number, it will reduce the possibility of indexing different strings (collisions).
  - It is recommended that you use **prime numbers** in case of modulo.
  - The ASCII values of a, b, c, d, e, and f are 97, 98, 99, 100, 101, and 102 respectively.
  - Since all the strings contain the same characters with different permutations, the sum will 599.
  - The hash function will compute the same index for all the strings and the strings will be stored in the hash table in the following format.
  - As the index of all the strings is the same, you can create a list on that index and insert all the strings in that list.

Need for a good hash function

- {“abcdef”, “bcdefa”, “cdefab” , “defabc” }
  - (a + b + c + d + e + f) = (97 + 98 + 99 + 100 + 101 + 102) = 597

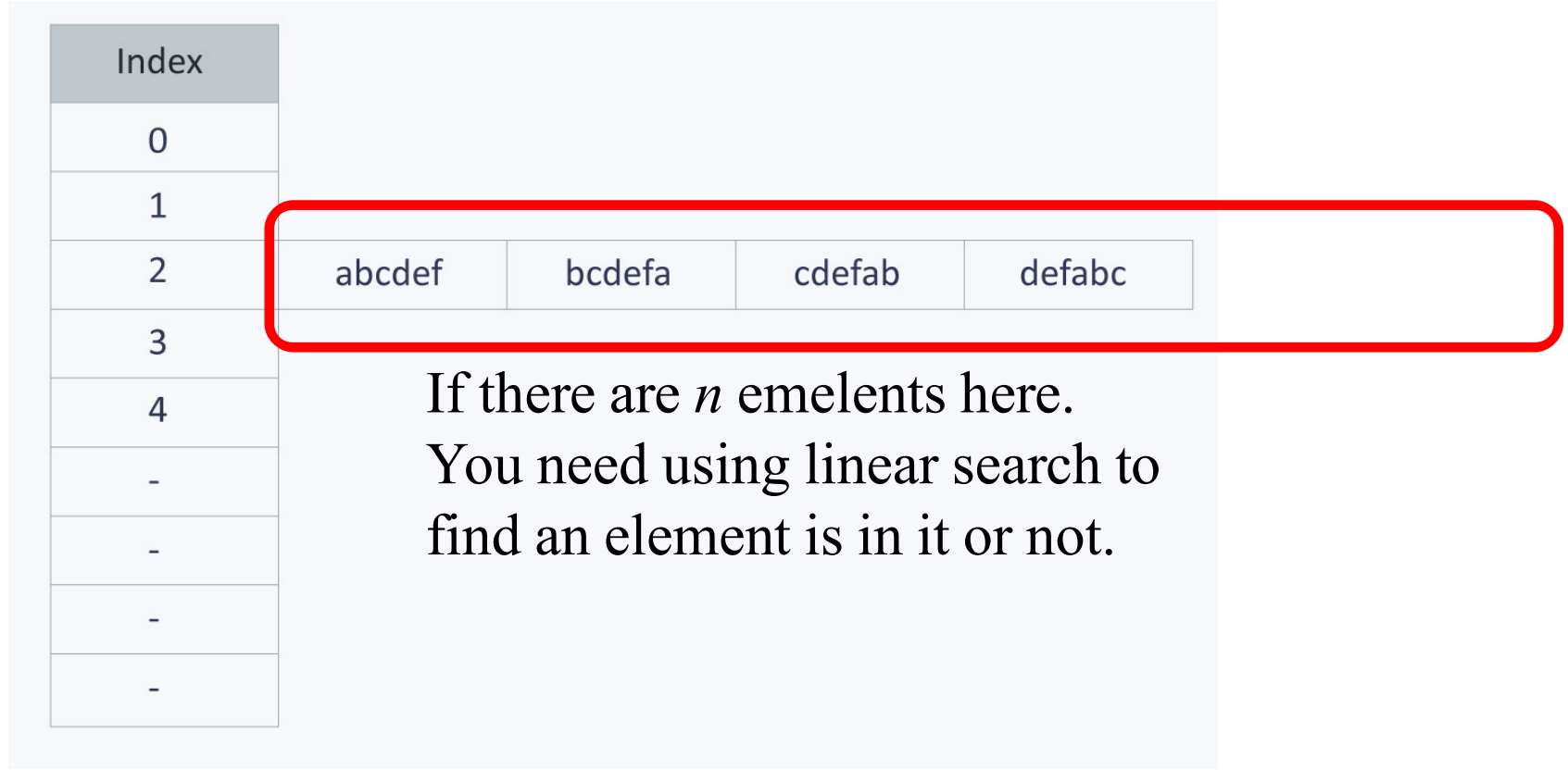
Hash Table

Here all strings are sorted at same index

Index				
0				
1				
2	abcdef	bcdefa	cdefab	defabc
3				
4				
-				
-				
-				
-				

Need for a good hash function

- {“abcdef”, “bcdefa”, “cdefab” , “defabc” }
  - (a + b + c + d + e + f) = (97 + 98 + 99 + 100 + 101 + 102) = 597
- Here, it will take **O(n)** time (where n is the number of strings) to access a specific string.
- This shows that the hash function is not a good hash function.



Need for a good hash function

- Let's try a different hash function. The index for a specific string will be equal to sum of ASCII values of characters multiplied by their respective order in the string after which it is modulo with 2069 (prime number).
  - "ASCII + ***position***"

String	Hash function	Index
abcdef	(97***1*** + 98***2*** + 99***3*** + *100*4 + 1015 + *102*6) % 2069	38
bcdefa	(98***1*** + 99***2*** + 100***3*** + *101*4 + 1025 + *97*6) % 2069	23
cdefab	(99***1*** + 100***2*** + 101***3*** + *102*4 + 975 + *98*6) % 2069	14
defabc	(100***1*** + 101***2*** + 102***3*** + *97*4 + 985 + *99*6) % 2069	11

Need for a good hash function

Hash Table

Here all strings are stored at different indices

Index	
0	
1	
-	
-	
-	
11	defabc
12	
13	
14	cdefab
-	
-	
-	
-	
23	bcdefa
-	
-	
-	
38	abcdef
-	
-	

perfect hash function

- Given a collection of items, a hash function that maps each item into a unique slot is referred to as a **perfect hash function**.
  - No collisions
- However, collision is hard to avoid.
- So, our goal is to create a hash function that minimizes the number of collisions, is easy to compute, and evenly distributes the items in the hash table.
- There are a number of common ways to extend the simple remainder method.
- We will consider a few of them here.

Some Hash fucntions

- **Division (**remainder method): h(key) = key % D
- **Mid-square** : square + choose some mid digits
- **Folding addition** :
  - **Shift**
  - **Boundary**
- **Digit analysis** :

Mid-Square Method

- Step 1. a seed value is taken and it is squared.
- Step 2. some digits from the middle are extracted as the new seed.
- Step 3. This process (Step 1 and 2) is repeated as many times as a key is required.
- Note. This technique can generate keys with high randomness if a big enough seed value is taken.

Mid-Square Method

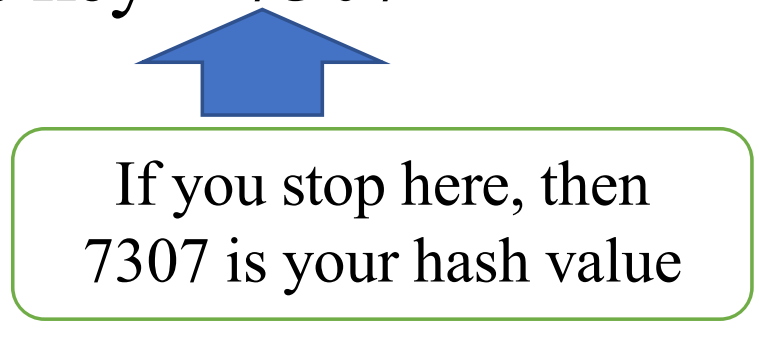
- Example.
  - Suppose a 4-digit key is taken .key = 4765
  - Hence, square of key is = 4765 * 4765 = 22705225
  - Now, from this 8-digit number, any four digits are extracted (Say, the middle four).
  - So, the new key value becomes key = 7052
  - Now, square of this new key is = 7052 * 7052 = 49730704
  - Again, the same set of 4-digits is extracted.
  - So, the new key value becomes key = 7307

  .

  .

  .

  - This process is repeated as many times as a hash value is required.



Exercise for 14:00

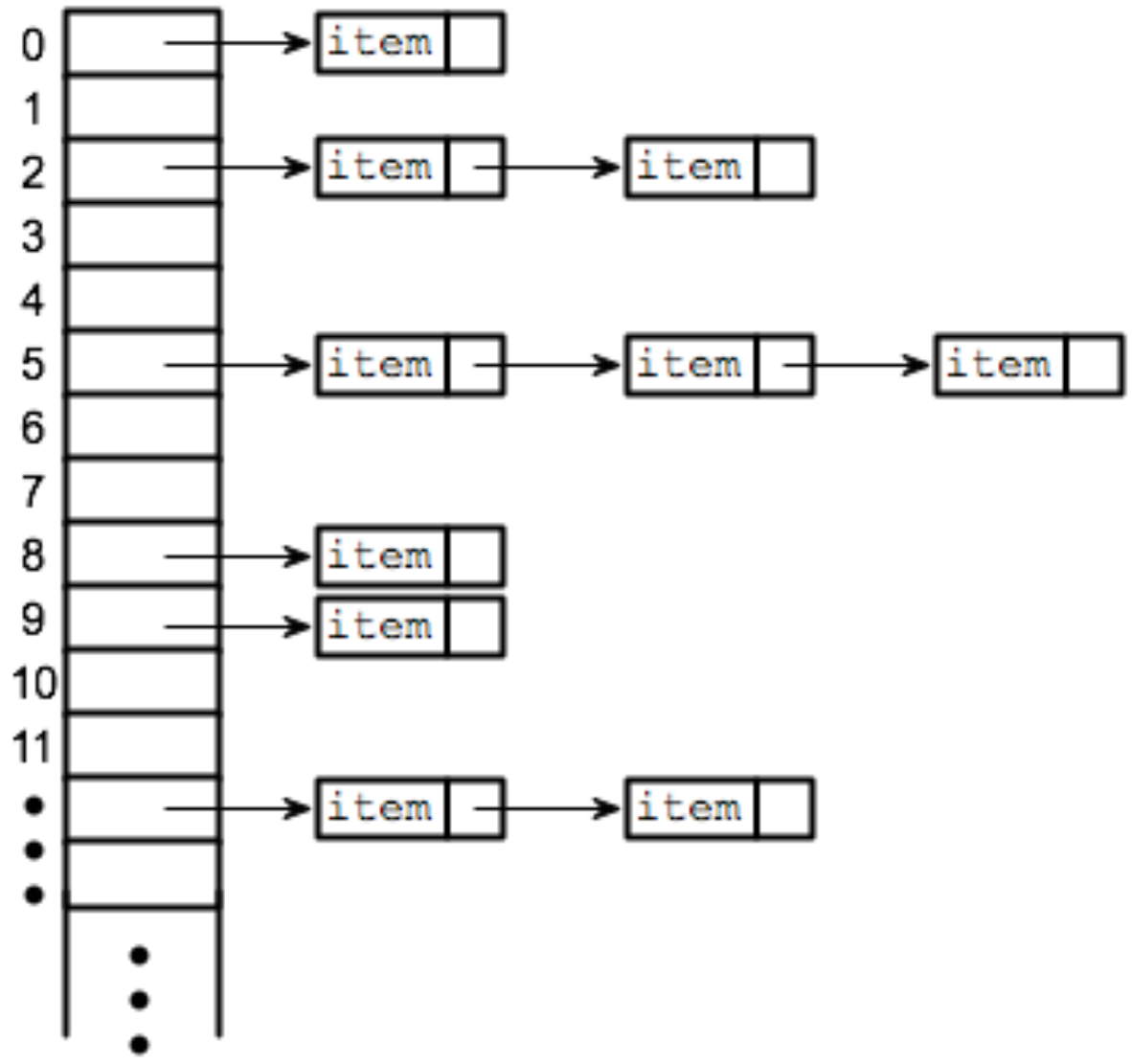
- Suppose that our hash function is hash(key) = key % table_size.
- In a hash table of size 13 which index positions would the following two keys map to?
- 27, 130
- (a) 1, 10
- (b) 13, 0
- (c) 1, 0
- (d) 2, 3

Collision resolution techniques

- Separate chaining (open hashing)
- linear probing
- quadratic probing
- double hashing

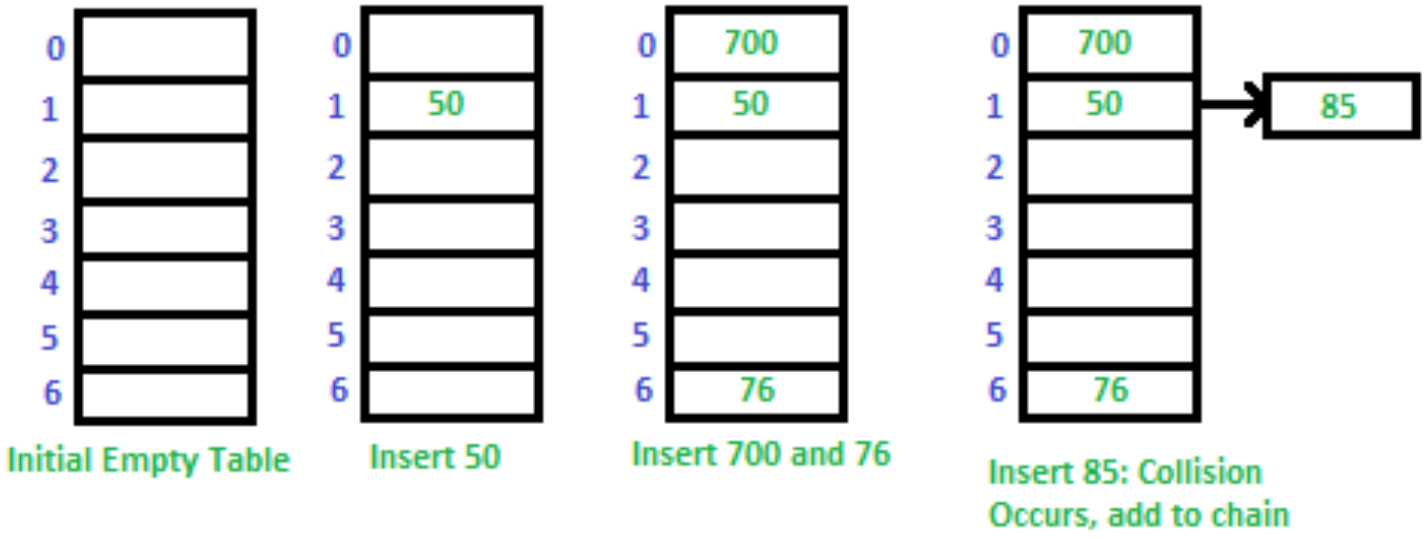
Separate chaining (open hashing)

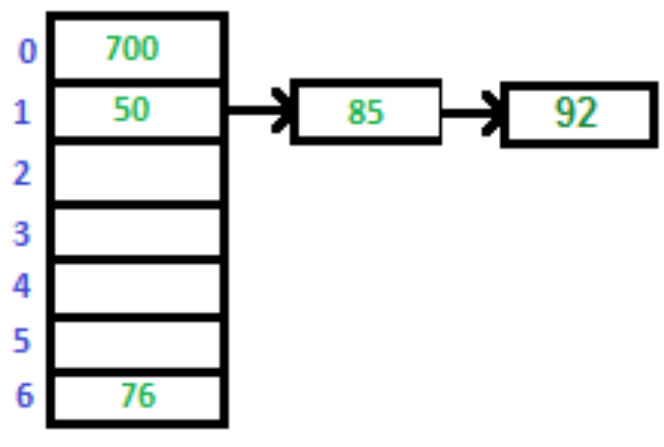
- linked lists
- In separate chaining, each element of the hash table is a linked list.
- To store an element in the hash table you must insert it into a specific linked list.
- If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.



Separate chaining (open hashing)

- Let us consider a simple hash function as "**key mod 7**" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



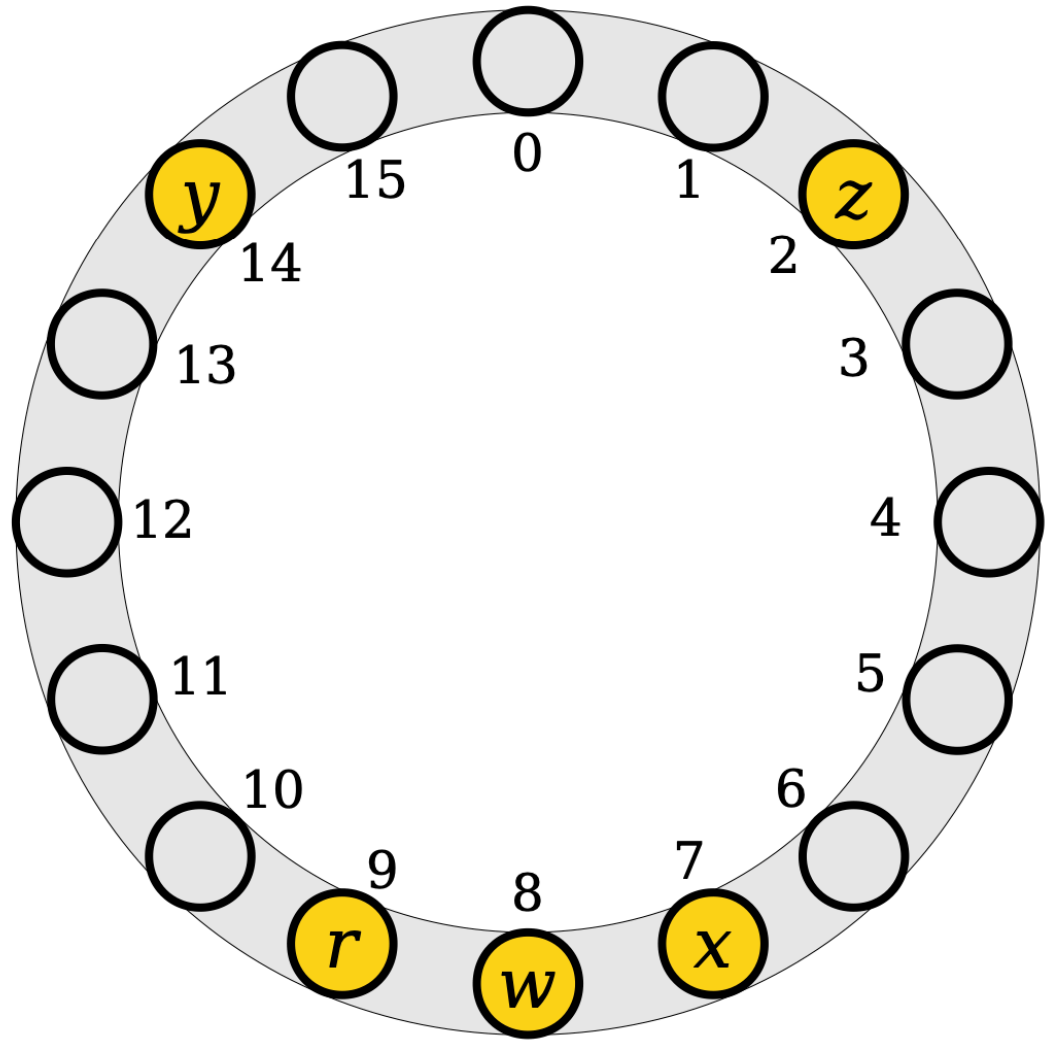


Separate chaining (open hashing)

- **Advantages:**
  - 1) Simple to implement.
  - 2) **Hash table never fills up**, we can always add more elements to the chain.
  - 3) Less sensitive to the hash function or load factors.
  - 4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.
- **Disadvantages:**
  - 1) Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
  - 2) Wastage of Space (Some Parts of hash table are never used)
  - 3) If the chain becomes long, then search time can become **O(n)** in the worst case.
  - 4) Uses extra space for links.

Linear probing (open addressing or closed hashing)

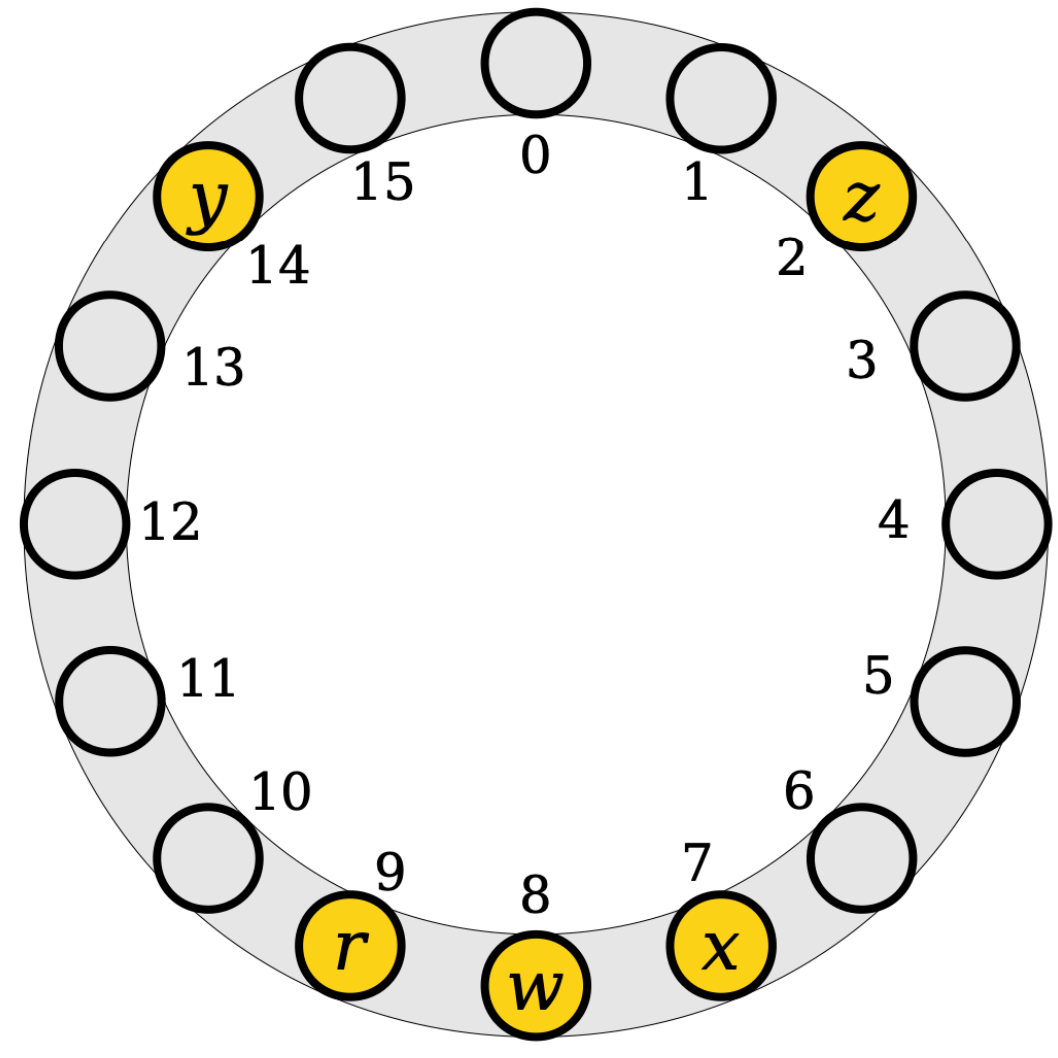
- ***Open addressing:*** Allow elements to "leak out" from their preferred position and spill over into other positions.



- To insert an element *x*, compute *h*(*x*) and try to place *x* there.
- If it's full, keep moving through the array, wrapping around at the end, until a free spot is found.
  - (h(key) + i) % table_size

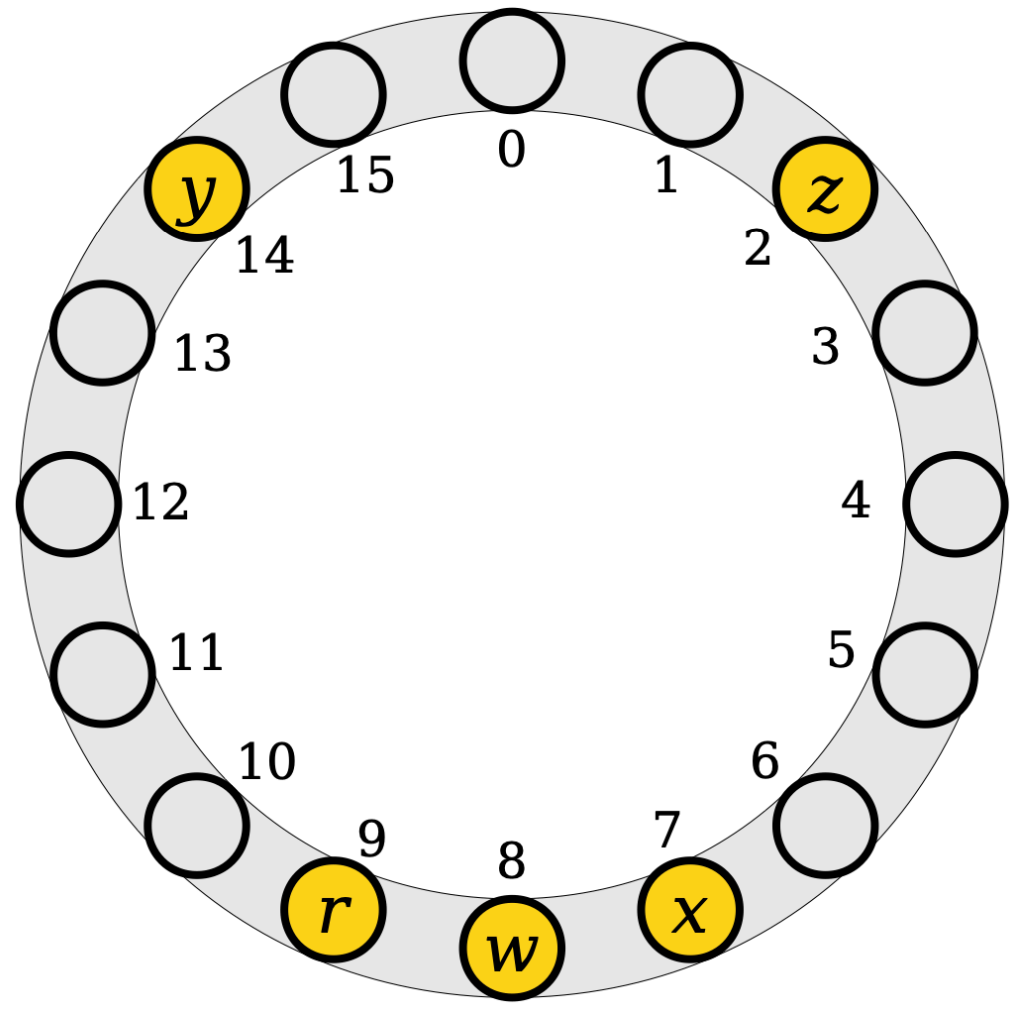
Linear probing (open addressing or closed hashing)

- Deletions are a bit trickier than in chained hashing.
- We cannot just do a search and remove the element where we find it.
- Why?



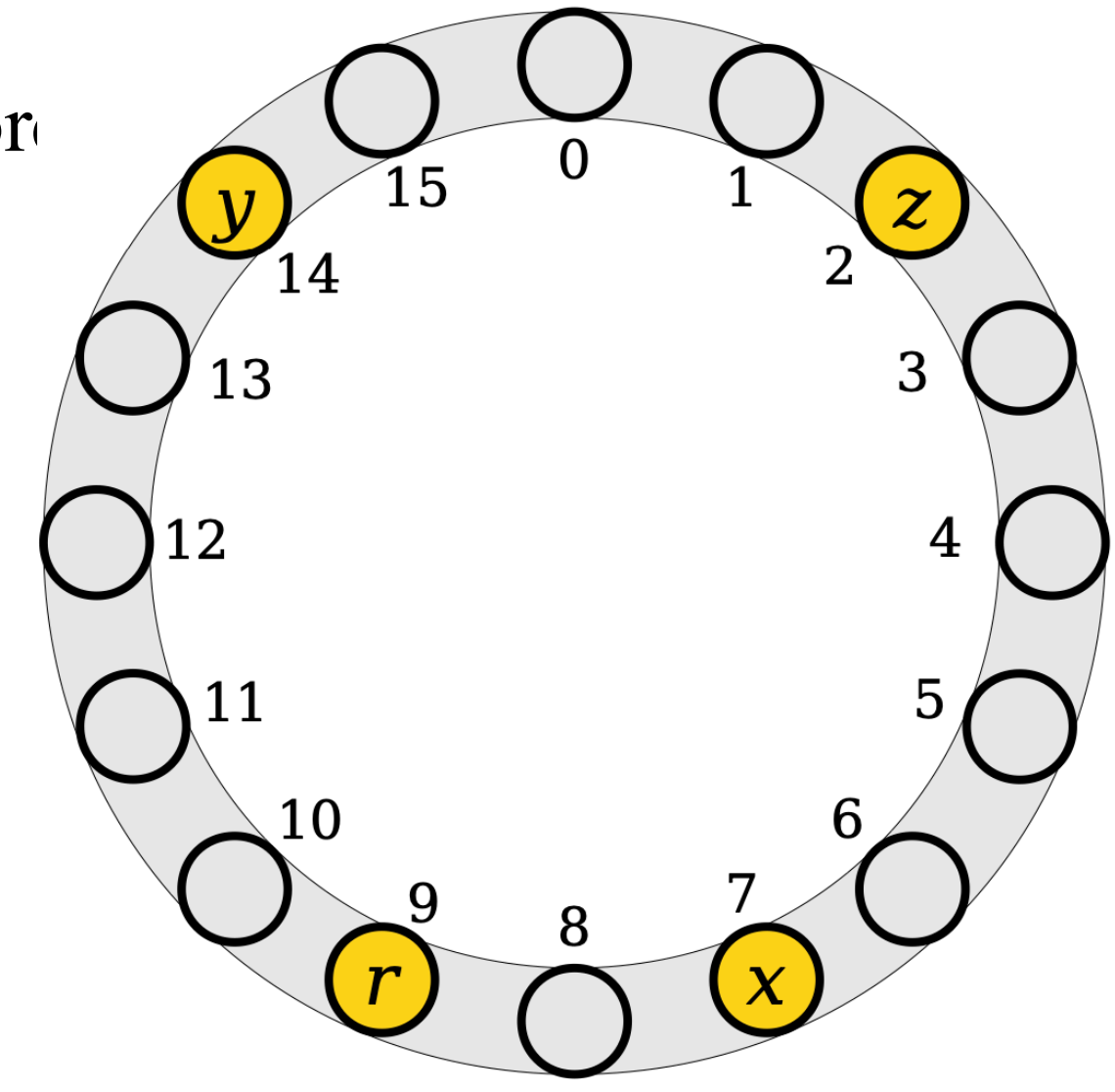
Linear probing (open addressing or closed hashing)

- Deletions are a bit trickier than in chained hashing.
- We cannot just do a search and remove the element where we find it.
- Why?
  - Suppose we insert x, w, r in order.
    - h(x) = h(w) = h(r) = 7



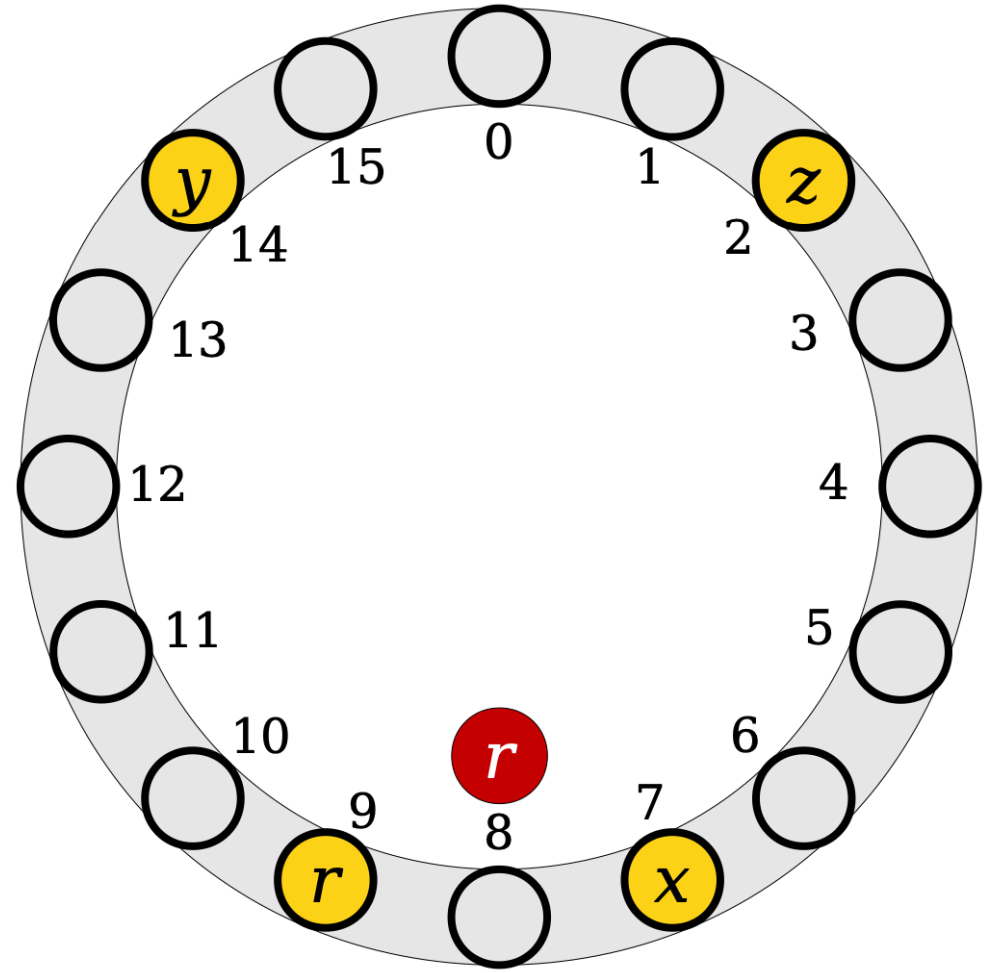
Linear probing (open addressing or closed hashing)

- Deletions are a bit trickier than in chained hashing.
- We cannot just do a search and remove the element where we find it.
- Why?
  - Suppose we insert x, w, r in or
    - h(x) = h(w) = h(r) = 7
- Then we remove w.
- Problem: We can not find r.



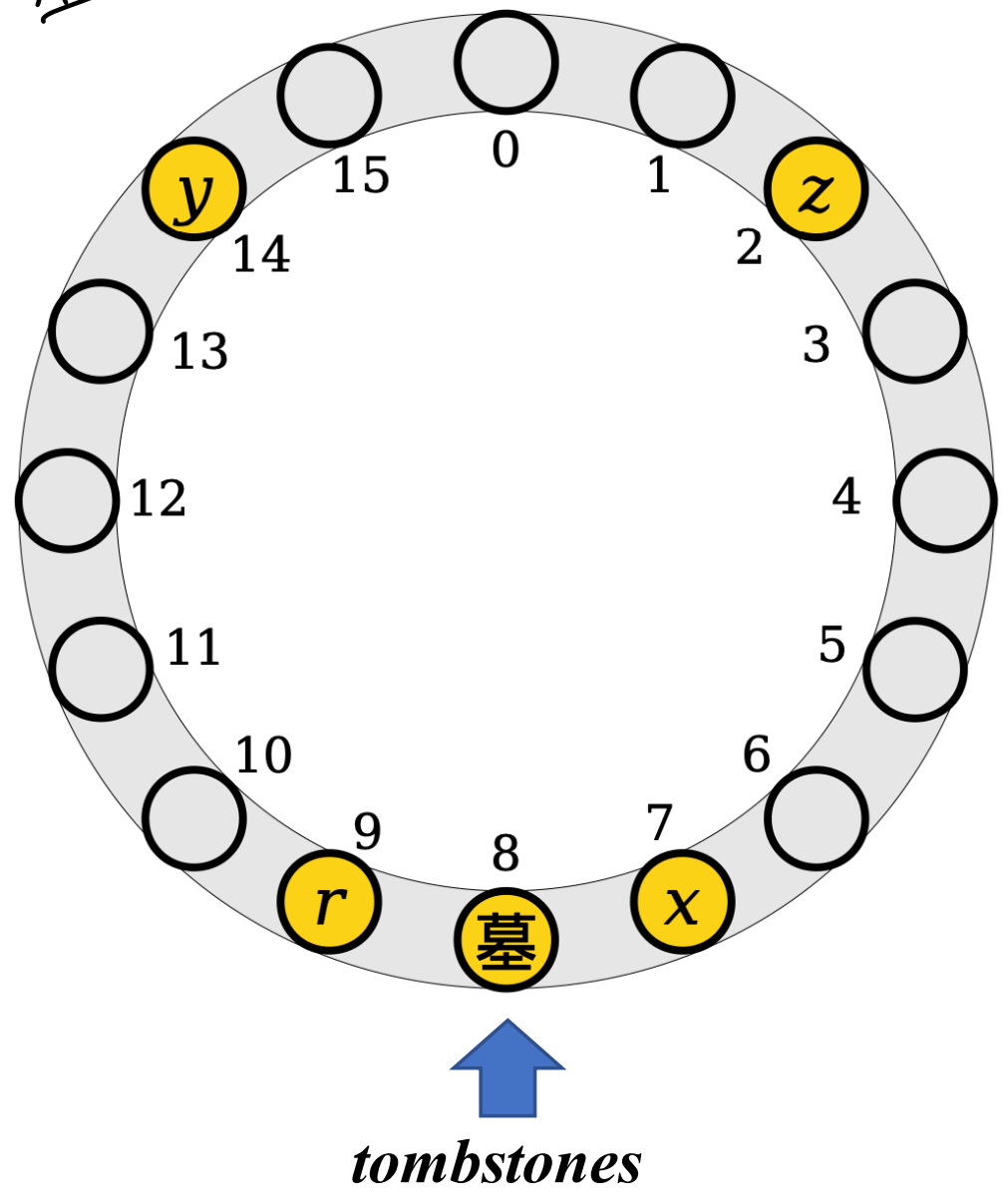
Linear probing (open addressing or closed hashing)

- Deletions are a bit trickier than in chained hashing.
- We cannot just do a search and remove the element where we find it.
- Why?
  - Suppose we insert x, w, r in order.
    - h(x) = h(w) = h(r) = 7
- Then we remove w.
- Problem: We can not find r.



Linear probing (open addressing or closed hashing)

- Deletions are often implemented using ***tombstones***.
- When removing an element, mark that the cell is empty and was previously occupied.
- When doing a lookup, don't stop at a tombstone.
- Instead, keep the search going.
- You need to watch out for wraparounds.
- When inserting, feel free to replace any tombstone you encounter.



Linear probing (open addressing or closed hashing)

- In practice, linear probing is one of the fastest general purpose hashing strategies available.
- ***Low memory overhead:*** just need an array and a hash function.
- ***Excellent locality:*** when collisions occur, we only search in adjacent locations in the array.
- ***Great cache performance:*** a combination of the above two factors.

Linear probing (open addressing or closed hashing)

当n大时，线性probing不大行了。

- Weakness:
  - Linear probing exhibits severe performance degradations when the load factor gets high.
  - The number of collisions tends to grow as a function of the number of existing collisions.
  - This is called ***primary clustering***

Quadratic Probing

- similar to Linear probing
- The difference is that if you were to try to insert into a space that is filled you would first check $1^2 = 1$ element away then $2^2 = 4$ elements away, then $3^2 = 9$ elements away then $4^2 = 16$ elements away and so on.
  - $(h(key) + i^2)$ % table_size
- If filled…

	Linear Probing	**Quadratic Probing**	
Frist	h(key)	h(key)	
Second	h(key) + 1	h(key) + 1^2	
Third	h(key) + 2	h(key) + 2^2	
Fourth	h(key) + 3	h(key) + 3^2	
Fifth	h(key) + 4	h(key) + 4^2	

Quadratic Probing

- Example. Table size is 16. First 5 pieces of data that all hash to index 2
  - First piece goes to index 2.
  - Second piece goes to 3 = ((2 + 1)%16
  - Third piece goes to 6 = ((2+4)%16
  - Fourth piece goes to 11 = ((2+9)%16
  - Fifth piece dosen’t get inserted because (2+16)%16==2 which is full so we end up back where we started and we haven’t searched all empty spots.

Quadratic Probing

- Note.
  - In order to guarantee that your quadratic probes will hit every single available spots eventually, your table size must meet be a prime number

Exercise for 14:00

- Concern a Hash Set with n = 5 buckets and the hash function **h(k) = 3k + 2 % table_size**
- Insert the values 1, 66, 31, and 15 in that order.
- 1. If the Hash Set uses Chaining, which bucket contains 15?
  - a. 0, b. 1, c. 2, d. 3, e. 4
- 2. If the Hash Set uses Open Addressing with Linear Probing, which bucket contains 15?
  - a. 0, b. 1, c. 2, d. 3, e. 4
- 3. If the Hash Set uses Open Addressing with Quadratic Probing, which bucket contains 15?
  - a. 0, b. 1, c. 2, d. 3, e. 4

Exercise for 15:40

- Concern a Hash Set with n = 5 buckets and the hash function **h(k) = 3k + 2 % table_size**
- Insert the values 1, 66, 31, and 15 in that order.
- 1. If the Hash Set uses Chaining, which bucket contains 31?
  - a. 0, b. 1, c. 2, d. 3, e. 4
- 2. If the Hash Set uses Open Addressing with Linear Probing, which bucket contains 31?
  - a. 0, b. 1, c. 2, d. 3, e. 4
- 3. If the Hash Set uses Open Addressing with Quadratic Probing, which bucket contains 31?
  - a. 0, b. 1, c. 2, d. 3 , e. 4

Double Hashing

- Double Hashing is works on a similar idea to linear and quadratic probing.
- The difference here is that instead of choosing next opening, a second hash function is used to determine the location of the next spot.
- For example, given hash function hash1 and hash2 and key. We do the following:
  - Check location hash1(key). If it is empty, put record in it.
  - If it is not empty calculate hash2(key).
  - check if hash1(key)+hash2(key) is open, if it is, put it in repeat with hash1(key)+2*hash2(key), hash1(key)*+3hash2(key) and so on, until an opening is found.
- (h1(key) + i * h2(key)) % table_size

Double Hashing

- Check location hash1(key). If it is empty, put record in it.
- If it is not empty calculate hash2(key).
- check if hash1(key)+hash2(key) is open, if it is, put it in
- repeat with hash1(key)+2*hash2(key), hash1(key)+3*hash2(key) and so on, until an opening is found.

- (h1(key) + i * h2(key)) % table_size

	Linear Probing	**Quadratic Probing**	**Double Hashing**
Frist	h(key)	h(key)	h1(key)
Second	h(key) + 1	h(key) + 1^2	h1(key) + h2(key)
Third	h(key) + 2	h(key) + 2^2	h1(key) + 2*h2(key)
Fourth	h(key) + 3	h(key) + 3^2	h1(key) + 3*h2(key)
Fifth	h(key) + 4	h(key) + 4^2	h1(key) + 4*h2(key)
	…	…	…

Rehashing

- Like ArrayLists, we have to guess the number of elements we need to insert into a hash table
- Whatever our collision policy is, the hash table becomes inefficient when load factor is too high.
- To alleviate load, **rehash**:
  - create larger table, scan current table, insert items into new table using new hash function

When to Rehash

- We can rehash as soon as load factor > 1/2
  - For quadratic probing, insert may fail if load factor > 1/2
- Or, we can rehash only when insert fails
- Heuristically choose a load factor threshold, rehash when threshold breached

Rehash Example

- Current Table:
  - quad. probing with h(x) = (x mod 7) 8, 0, 25, 17, 7

slot	0	1	2	3	4	5	6	7
value	**0**	**8**	**7**	**17**	**25**			

- New table
  - h(x) = (x mod 17)

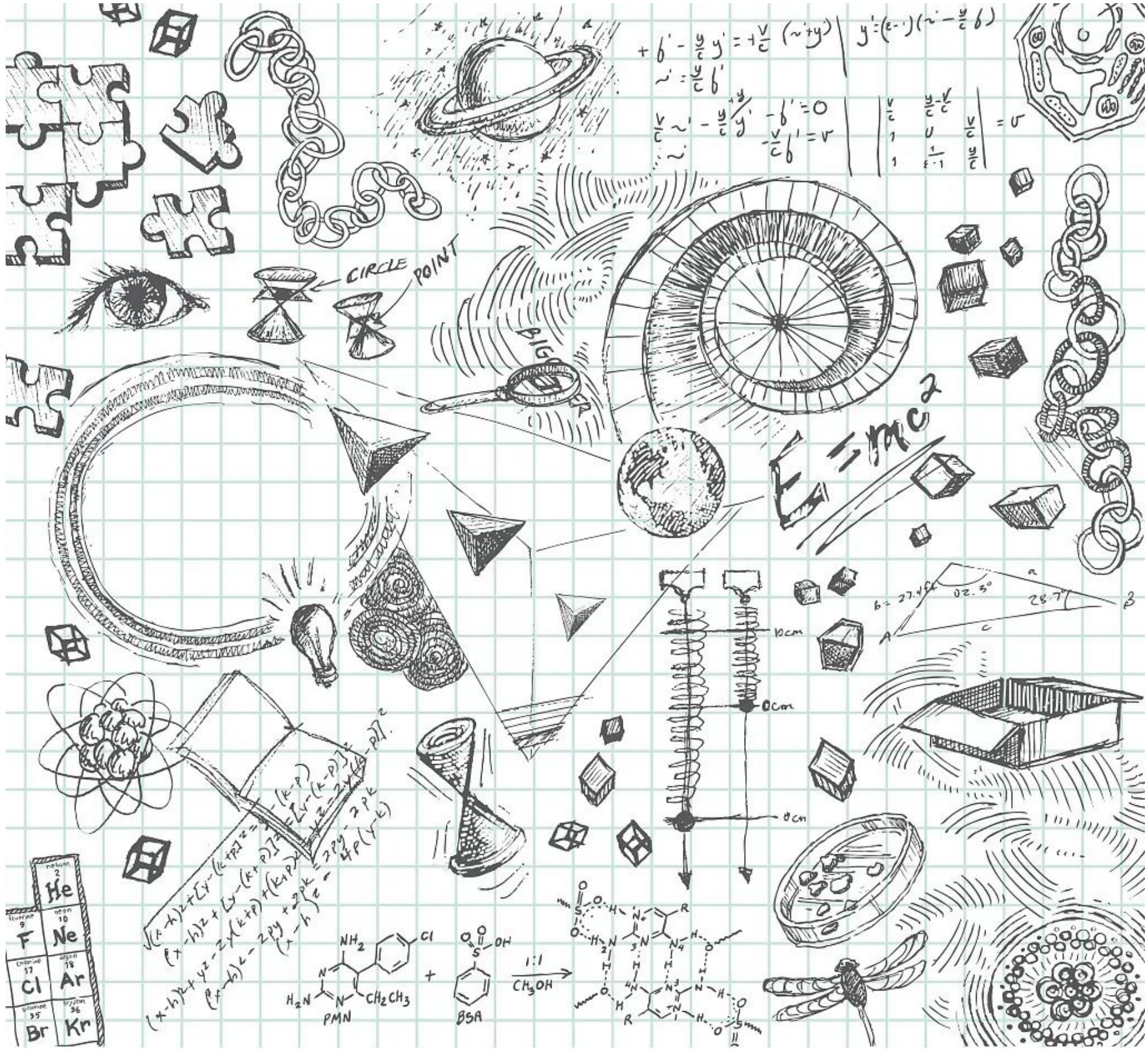
slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
value	**0**	**17**						**7**	**8**	**25**								

Rehash Cost

- No profound algorithm: re-insert each item Linear time
- If you rehash, inserting N items costs

$$O(1)*N + O(N) = O(N)$$

- Insert still costs O(1) amortized


CIRCLE
POINT
BIG
E=mc2
10cm
0cm
8cm
PMN
BSA
1:1
CH3OH