Data Structures
COMP242

Ala' Hasheesh
ahashesh@birzeit.edu

Hashing

Hashing (Properties)

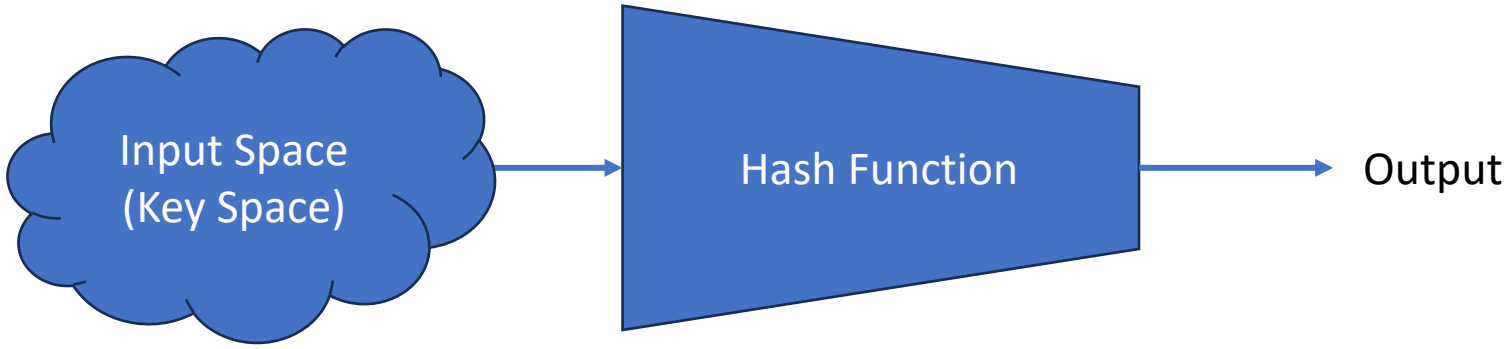
1. Constant time access O(1). Or almost constant depending on the implementation! Depending on the implementation this can grow to O(logn) or O(n)!

2. Hash Table: is an array of fixed size n (usually n is a prime).

3. General Idea is to map a key (i.e., our object) to an index and insert it into the table!

Hashing (Example)

Assume Table size is 7.

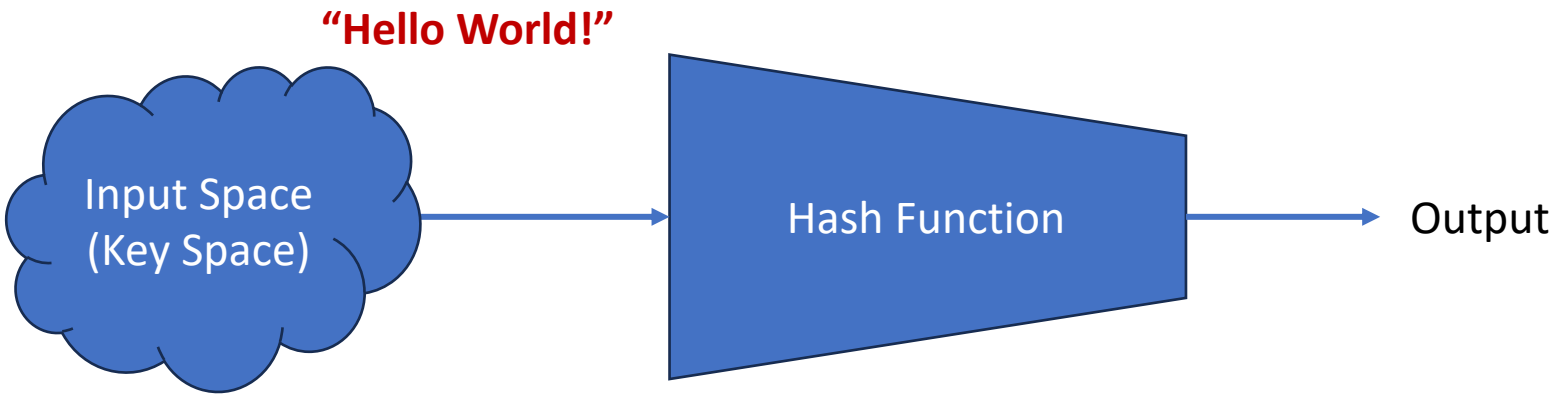


0	
1	
2	
3	
4	
5	
6	

Hash Table

Hashing (Example)

Assume Table size is 7.

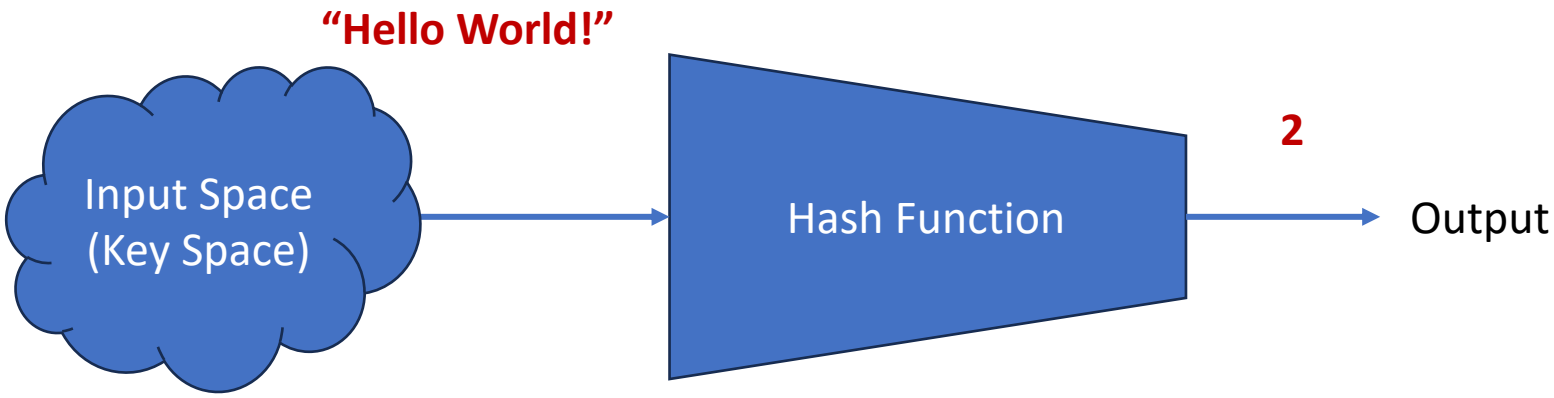


0	
1	
2	
3	
4	
5	
6	

Hash Table

Hashing (Example)

Assume Table size is 7.

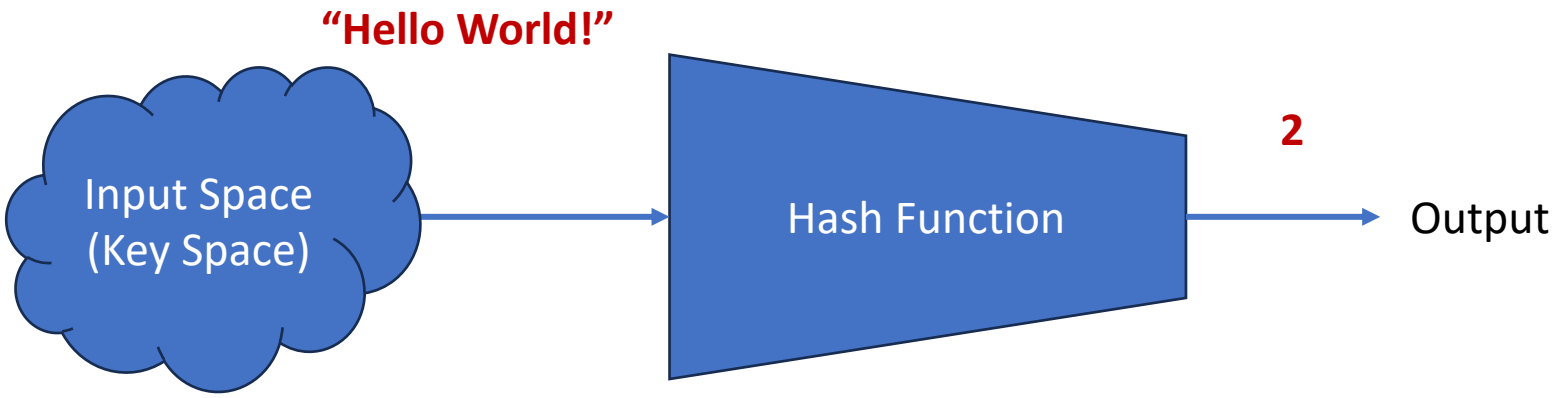


0	
1	
2	
3	
4	
5	
6	

Hash Table

Hashing (Example)

Assume Table size is 7.

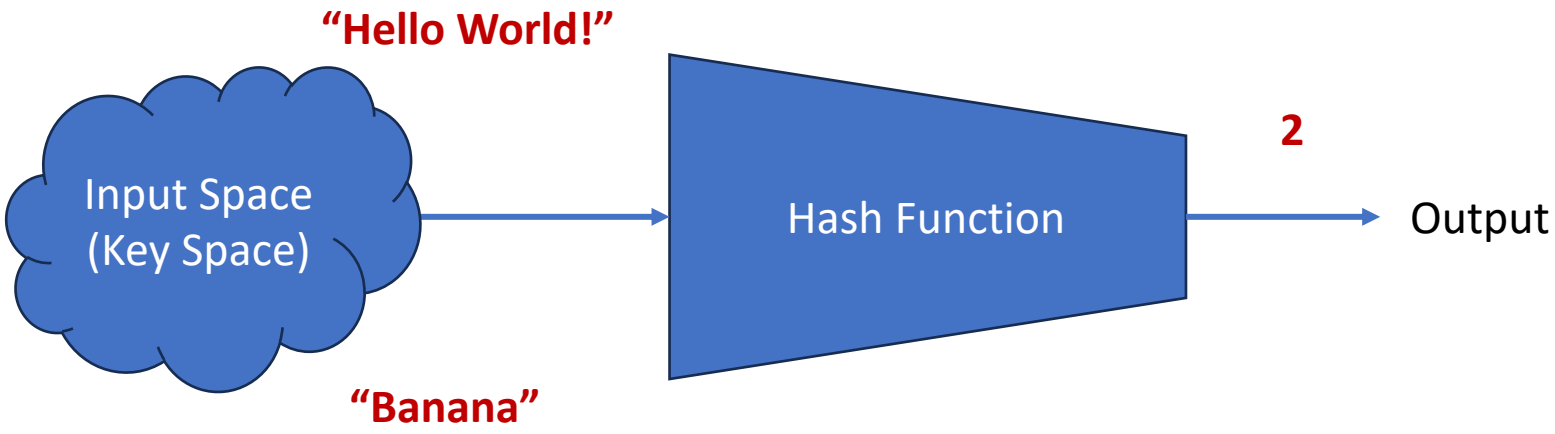


0	
1	
2	Hello World!
3	
4	
5	
6	

Hash Table

Hashing (Example)

Assume Table size is 7.

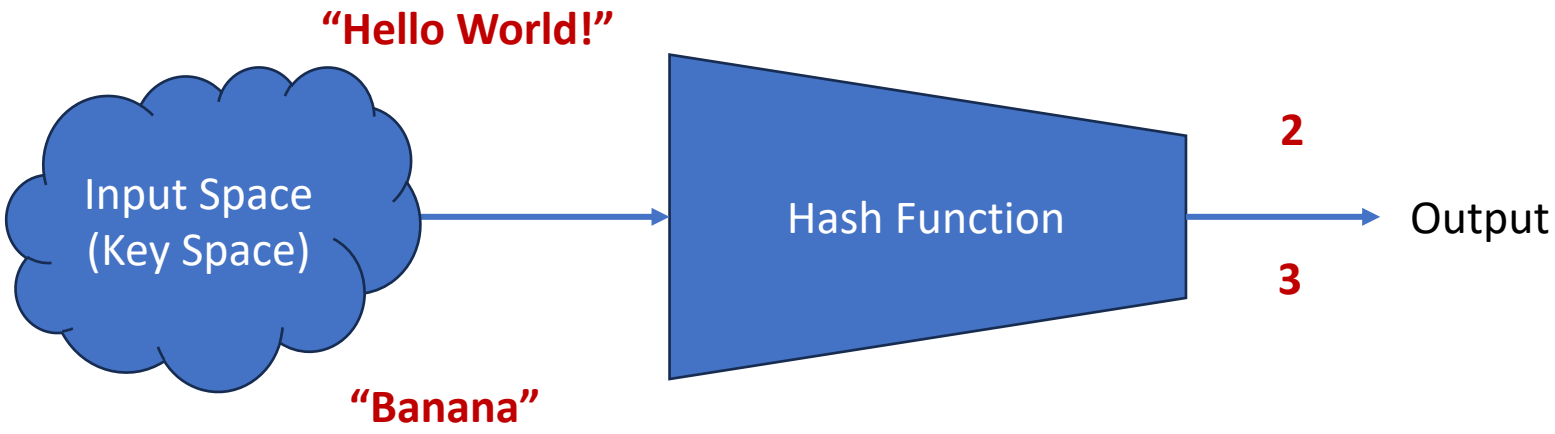


0	
1	
2	Hello World!
3	
4	
5	
6	

Hash Table

Hashing (Example)

Assume Table size is 7.



0	
1	
2	Hello World!
3	Banana
4	
5	
6	

Hash Table

Hashing (Definitions)

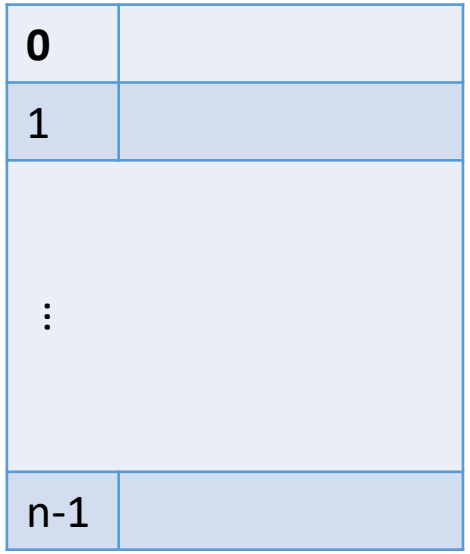
1. Constant time access O(1). Or almost constant depending on the implementation! Depending on the implementation this can grow to O(logn) or O(n)!

2. **Hash Table**: is an array of fixed size n (usually n is a prime).

3. General Idea is to map a key to an index and insert it into the table!

4. **Hash Function**: is a special function than maps our element to another value. In a hash table, hash function will map the input into a value in the range [0 – n). From 0 to (n - 1)

5. Hash Function in general maps values from one range (big) to another (small). One way to bound its values into [0 - n-1] is to use mod!



Hash Table

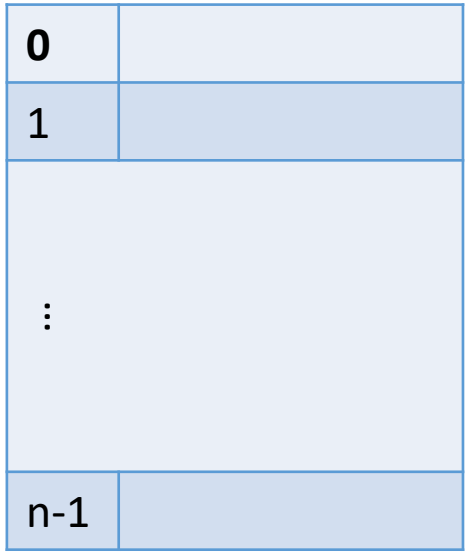
x % n will always produce values in the range [0 - n-1]

Hashing (Hash Function)

1. Don’t use the element directly (even if it’s an integer).

2. Map the element using the hash function into the range [0 - n-1]

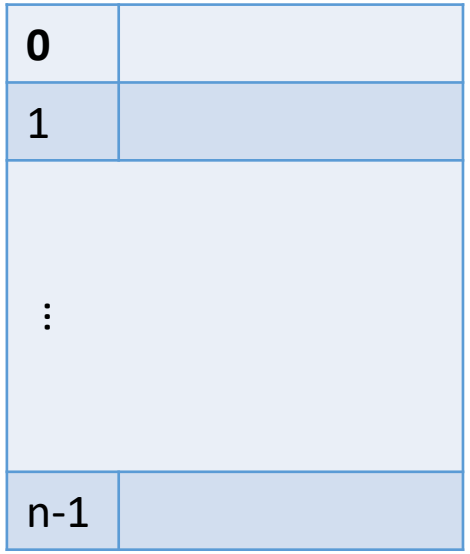
3. In other words, given an array of size **n**, we use a hash function **h(k)** to map Input **x** into some index in the array!



Hash Table

Hashing (Hash Function)

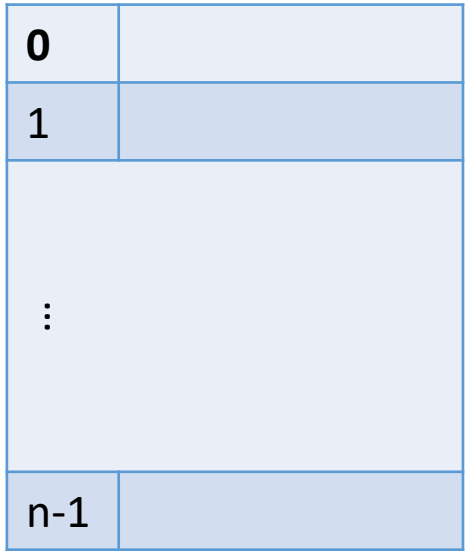
1. Don’t use the element directly (even if it’s an integer).
2. Map the element using the hash function into the range [0 - n-1]
3. In other words, given an array of size **n**, we use a hash function **h(k)** to map Input **x** into some index in the array!
4. Mapping is not unique and different inputs can map to the same index!



Hash Table

Hashing (Hash Function)

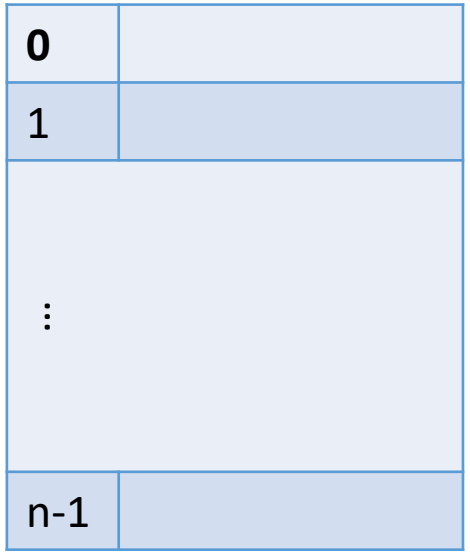
1. Don’t use the element directly (even if it’s an integer).
2. Map the element using the hash function into the range [0 - n-1]
3. In other words, given an array of size **n**, we use a hash function **h(k)** to map Input **x** into some index in the array!
4. Mapping is not unique and different inputs can map to the same index!
5. Good hash functions produce unique output (most of the time!).



Hash Table

Hashing (Hash Function)

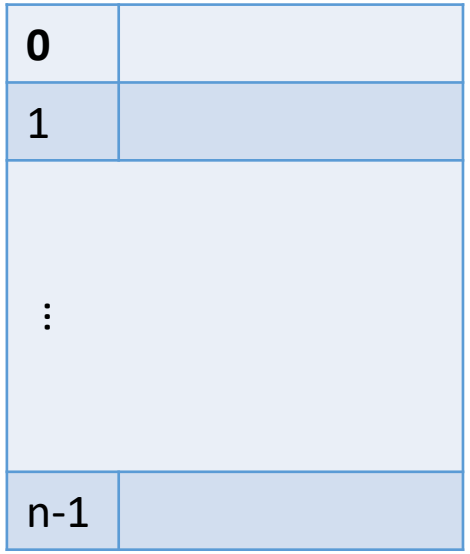
1. Don’t use the element directly (even if it’s an integer).
2. Map the element using the hash function into the range [0 - n-1]
3. In other words, given an array of size **n**, we use a hash function **h(k)** to map Input **x** into some index in the array!
4. Mapping is not unique and different inputs can map to the same index!
5. Good hash functions produce unique output (most of the time!).
6. When two inputs are mapped into the same output (same index) we call that a **Collision**.



Hash Table

Hashing

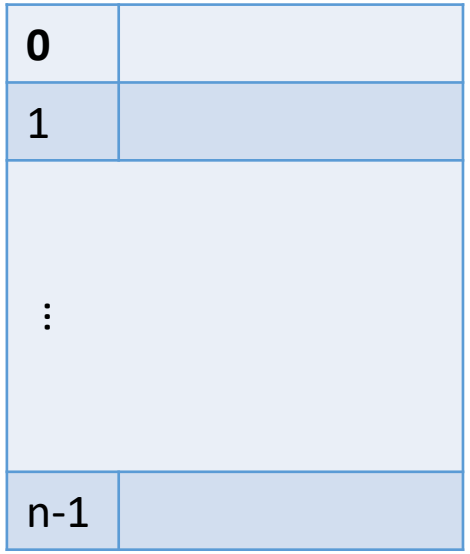
- Hash an input (any input of any type) and produce an index as an output!



Hash Table

Hashing

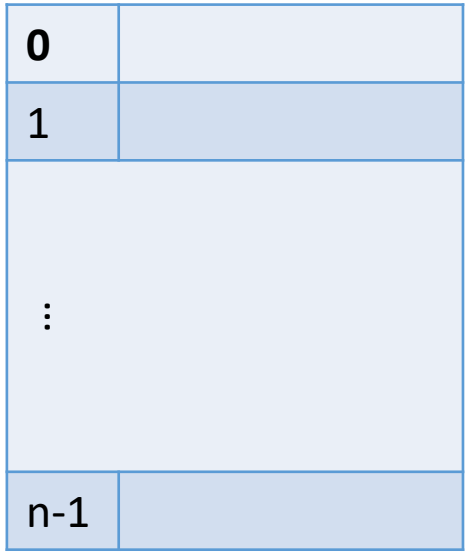
- Hash an input (any input of any type) and produce an index as an output!
- If the hash function is good with unique output this will give **O(1)** access time. Which is a great improvement over **O(logn)**.



Hash Table

Hashing

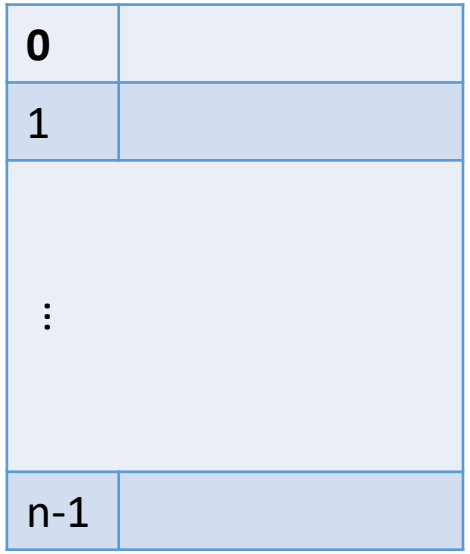
- Hash an input (any input of any type) and produce an index as an output!
- If the hash function is good with unique output this will give **O(1)** access time. Which is a great improvement over **O(logn)**.
- The main challenge is to find a good hashing function and storage strategy.



Hash Table

Hashing

- Hash an input (any input of any type) and produce an index as an output!
- If the hash function is good with unique output this will give **O(1)** access time. Which is a great improvement over **O(logn)**.
- The main challenge is to find a good hashing function and storage strategy.
  - Space (We want to utilize the array to its fullest)
  - Collisions and uniqueness of the output.
  - Running time of the hash function itself!



Hash Table

Example

- Key Space (i.e., Input Space) is **integers**.
- Table Size (i.e., **n**) is **10**.
- Hash Function is the module operator (%).

$$h(x) = x \,\%\, n = x \,\%\, 10$$

- We want to insert the following numbers:

15, 17, 229, 11, 320, 353

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Table

Example

Insert 15, 17, 229, 11, 320, 353

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Table

Example

Insert 15, 17, 229, 11, 320, 353

$h(15) = 15 \,\%\, 10 = 5$

0	
1	
2	
3	
4	
5	15
6	
7	
8	
9	

Hash Table

Example

Insert 15, 17, 229, 11, 320, 353

$h(17) = 17 \,\%\, 10 = 7$

0	
1	
2	
3	
4	
5	15
6	
7	17
8	
9	

Hash Table

Example

Insert 15, 17, 229, 11, 320, 353

$$h(229) = 229 \,\%\, 10 = 9$$

0	
1	
2	
3	
4	
5	15
6	
7	17
8	
9	229

Hash Table

Example

Insert 15, 17, 229, 11, 320, 353

$$h(11) = 11 \,\%\, 10 = 1$$

0	
1	11
2	
3	
4	
5	15
6	
7	17
8	
9	229

Hash Table

Example

Insert 15, 17, 229, 11, 320, 353

$h(320) = 320 \,\%\, 10 = 0$

0	**320**
1	11
2	
3	
4	
5	15
6	
7	17
8	
9	229

Hash Table

Example

Insert 15, 17, 229, 11, 320, 353

$h(353) = 353 \,\%\, 10 = 3$

0	**320**
1	11
2	
3	353
4	
5	15
6	
7	17
8	
9	229

Hash Table

Example

Insert 15, 17, 229, 11, 320, 353

$$h(353) = 353 \,\%\, 10 = 3$$

What will happen if we try to insert (320, 110, 10, 2000)

0	**320**
1	11
2	
3	353
4	
5	15
6	
7	17
8	
9	229

Hash Table

Example

Insert 15, 17, 229, 11, 320, 353

$$h(353) = 353 \,\%\, 10 = 3$$

0	**320**
1	11
2	
3	353
4	
5	15
6	
7	17
8	
9	229

Hash Table

What will happen if we try to insert (320, 110, 10, 2000)

All of them will map to 0! (**Collisions**)

Hashing - Collisions

- Collisions happen when multiple keys (i.e., items are mapped to the same value)

  $h(x) = x \,\%\, 10$
  $h(10) = 10 \,\%\, 10 = 0$
  $h(320) = 320 \,\%\, 10 = 0$
  $h(1700) = 1700 \,\%\, 10 = 0$

- The main issue is that we can't store multiple items in the same cell (i.e., **bucket**)
- Our choice of the hash function is what determines the number of collisions.
- Choosing a **bad hash** function will give lots of collisions.
- Even **good hash** functions will produce some collisions.
- We need to handle collisions even if our hash function is almost perfect (very rare collisions).

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Table

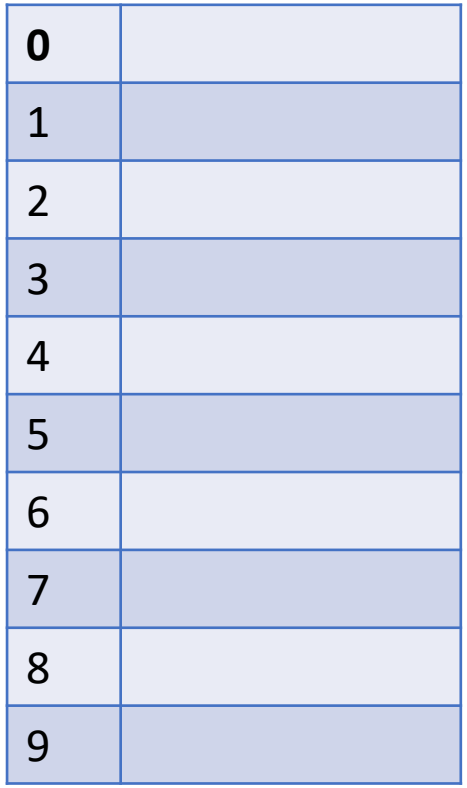
Hashing - Collisions

We have two main ways to deal with collisions:

1. **Open Hashing** (**Separate Chaining**): Each **cell**/**bucket** in the table is a linked list.

2. **Closed Hashing** (**Open Addressing**): We recompute the hash value according to a rule.

    1. **Linear Probing**
    2. **Quadratic Probing**
    3. **Double Hashing**



Hash Table

Hashing - Open Hashing (Separate Chaining)

- Keep a list of keys/items that map into the same value! (Array of linked lists)
  - Buckets point to the first element.
  - New elements are inserted to the end of the list.

- **Advantages (Pros)**
  - Better space utilization for large number of items.
  - Simple collision handling - Just append or search the list.
  - We can store more items than the hash table size (overflow).

- **Disadvantages (Cons)**
  - If we have a bad hash function (lots of collisions) the worst cast running time.
    Would increase to O(n). In other words, almost all elements will be in a single bucket!
  - Do we use single or double linked lists?
  - Bad Cache performance (due to implementation of linked list).
  - Difficulty in rehashing (We will talk about this later!)

Hashing - Open Hashing (Separate Chaining)

- Key Space (i.e., Input Space) is **integers**.
- Table Size (i.e., **n**) is **10**.
- Hash Function is the module operator (%).

$$h(x) = x \,\%\, n = x \,\%\, 10$$

- We want to insert the following numbers:

20, 15, 11, 35, 12, 40, 64, 4, 81, 9

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Table

Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

$h(20) = 20 \,\%\, 10 = 0$

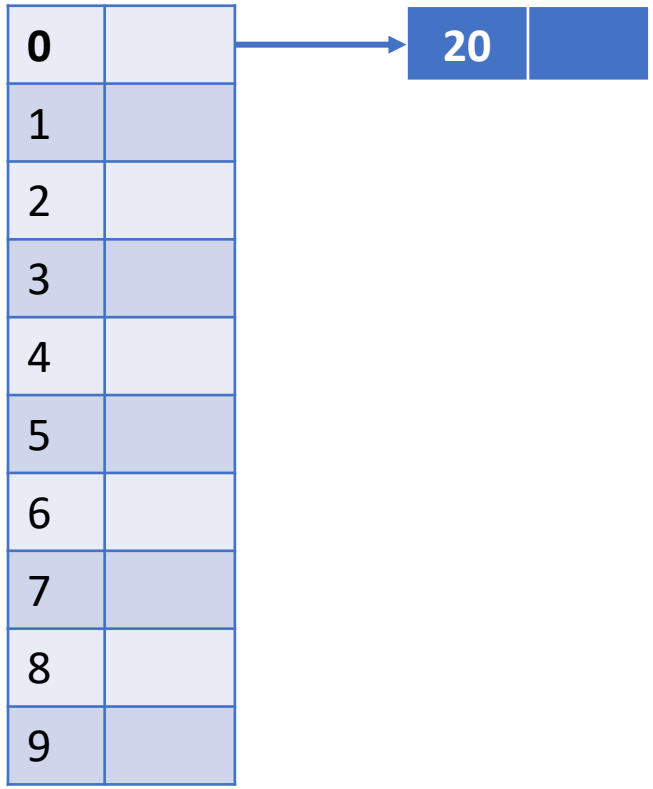
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Table

Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

$h(20) = 20 \,\%\, 10 = 0$

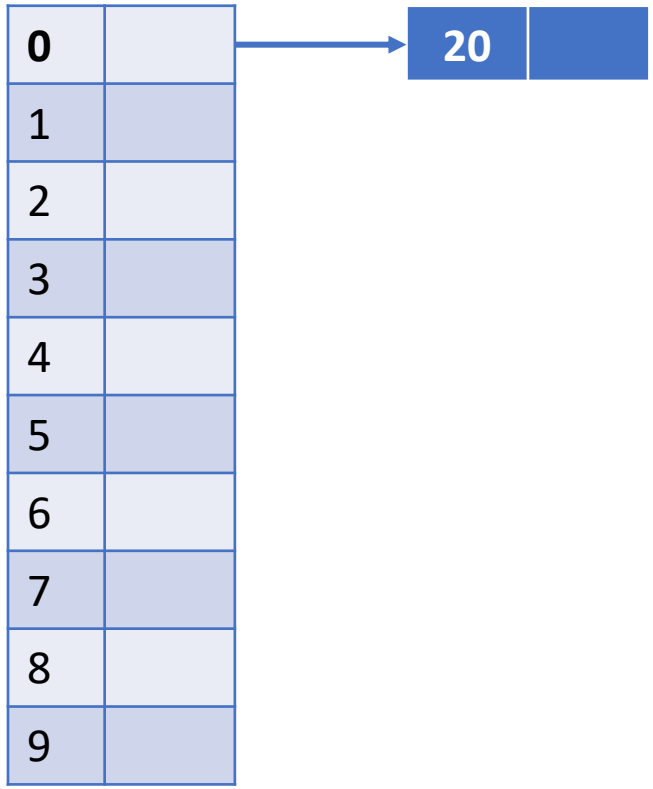


Hash Table

Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

$h(15) = 15 \,\%\, 10 = 5$

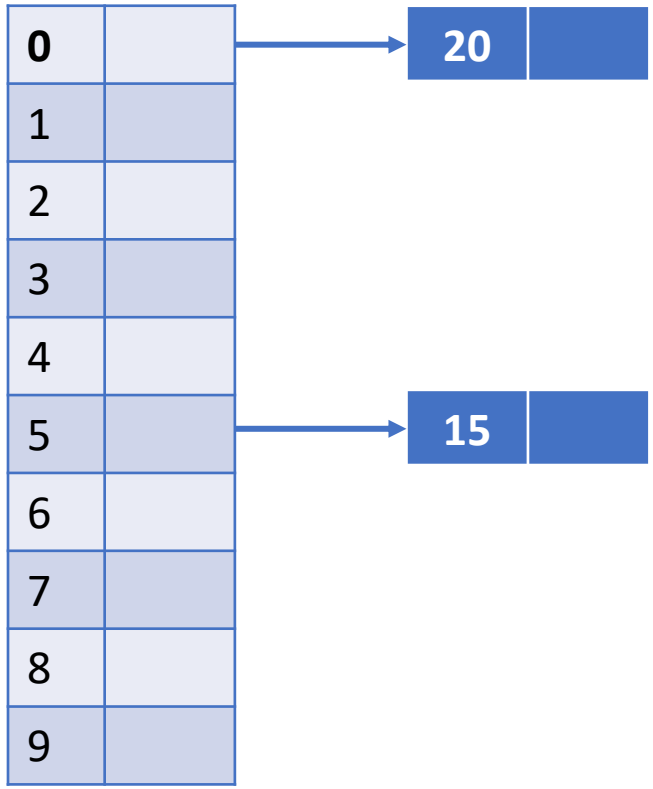


Hash Table

Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

$h(15) = 15 \,\%\, 10 = 5$

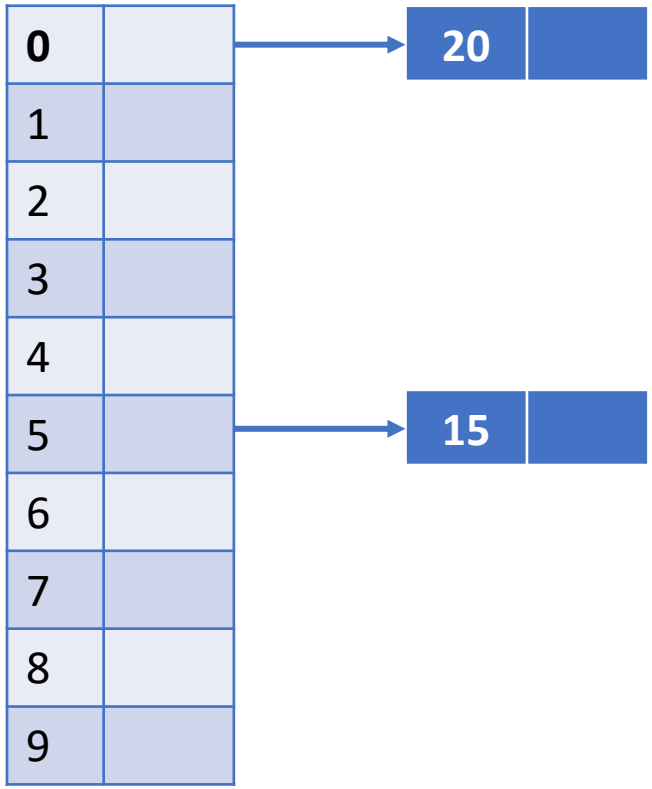


Hash Table

Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

$h(11) = 11 \,\%\, 10 = 1$

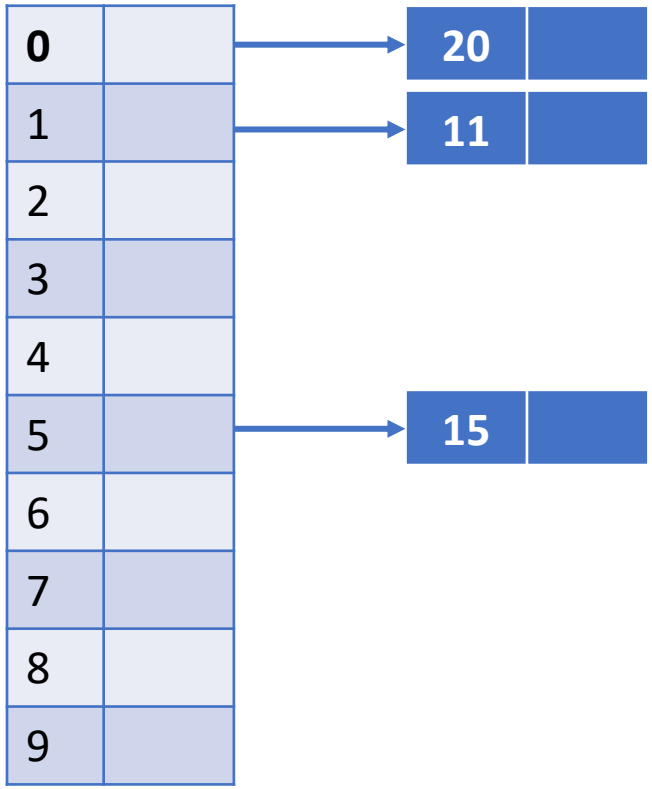


Hash Table

Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

$h(11) = 11 \,\%\, 10 = 1$

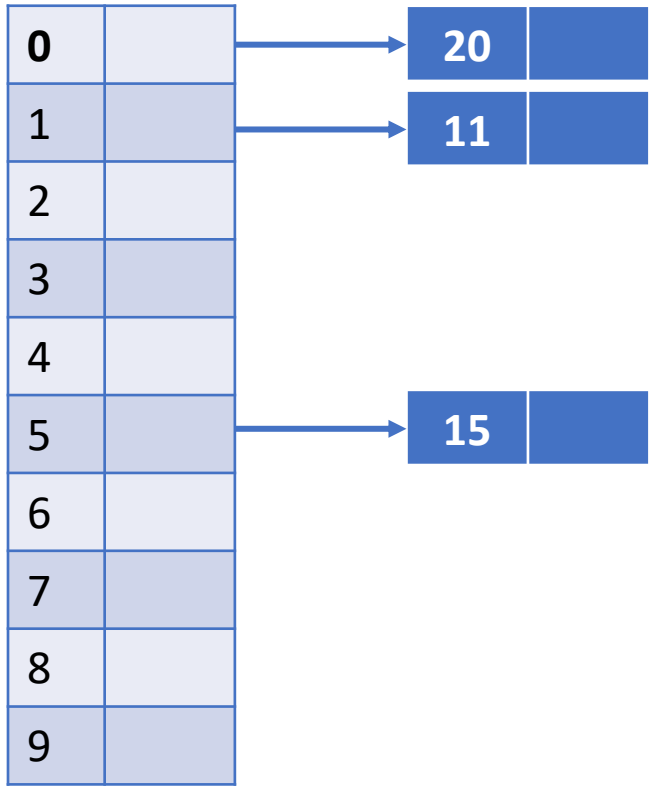


Hash Table

Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

$h(35) = 35 \,\%\, 10 = 5$

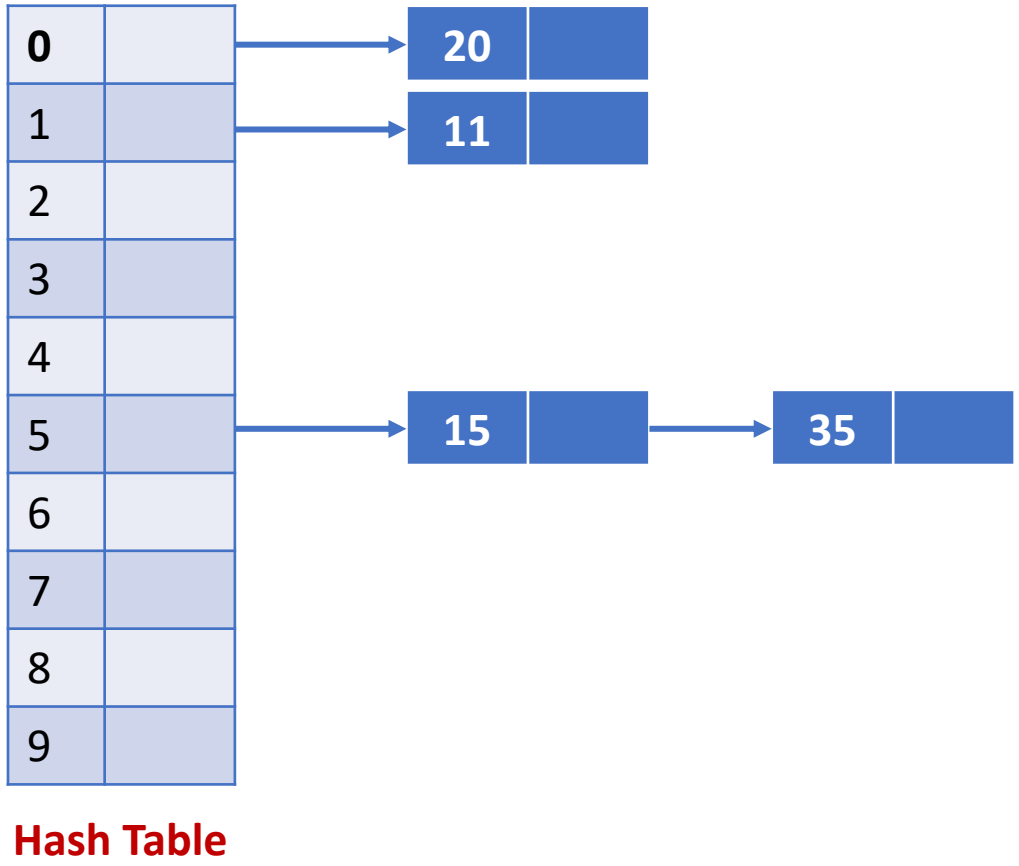


Hash Table

Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

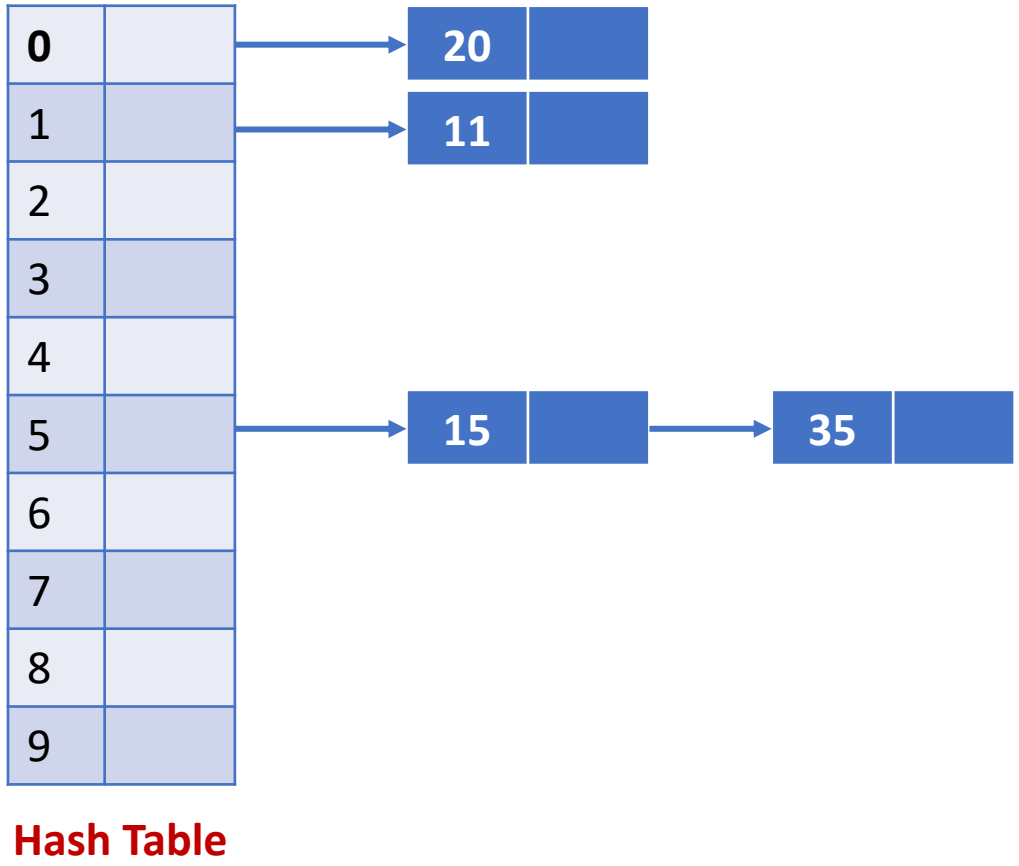
$h(35) = 35 \% 10 = 5$



Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

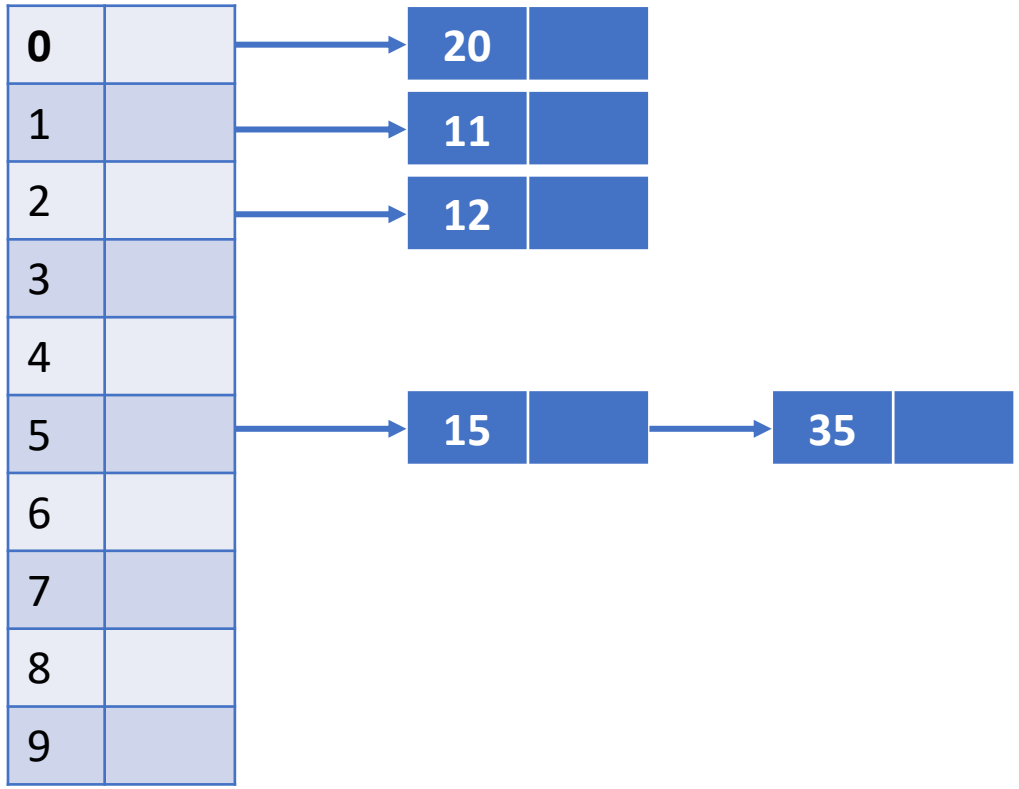
$h(12) = 12 \,\%\, 10 = 2$



Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

$h(12) = 12 \,\%\, 10 = 2$

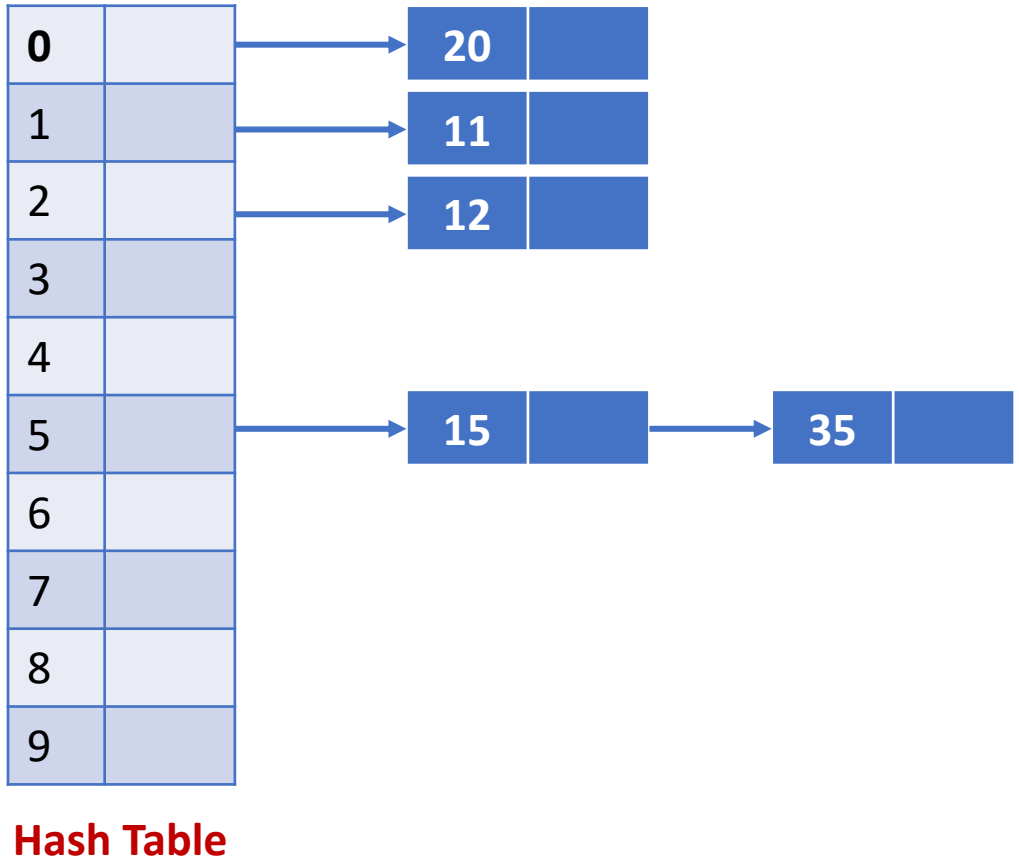


Hash Table

Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

$h(40) = 40 \,\%\, 10 = 0$

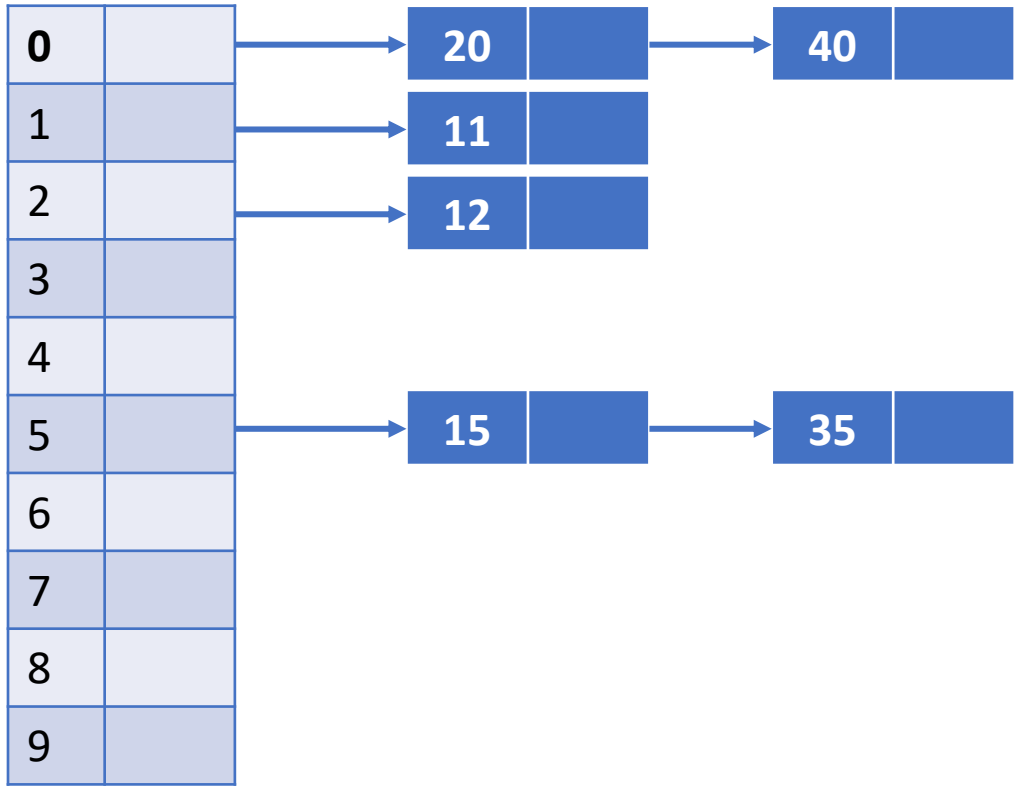


Hash Table

Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

$h(40) = 40 \,\%\, 10 = 0$

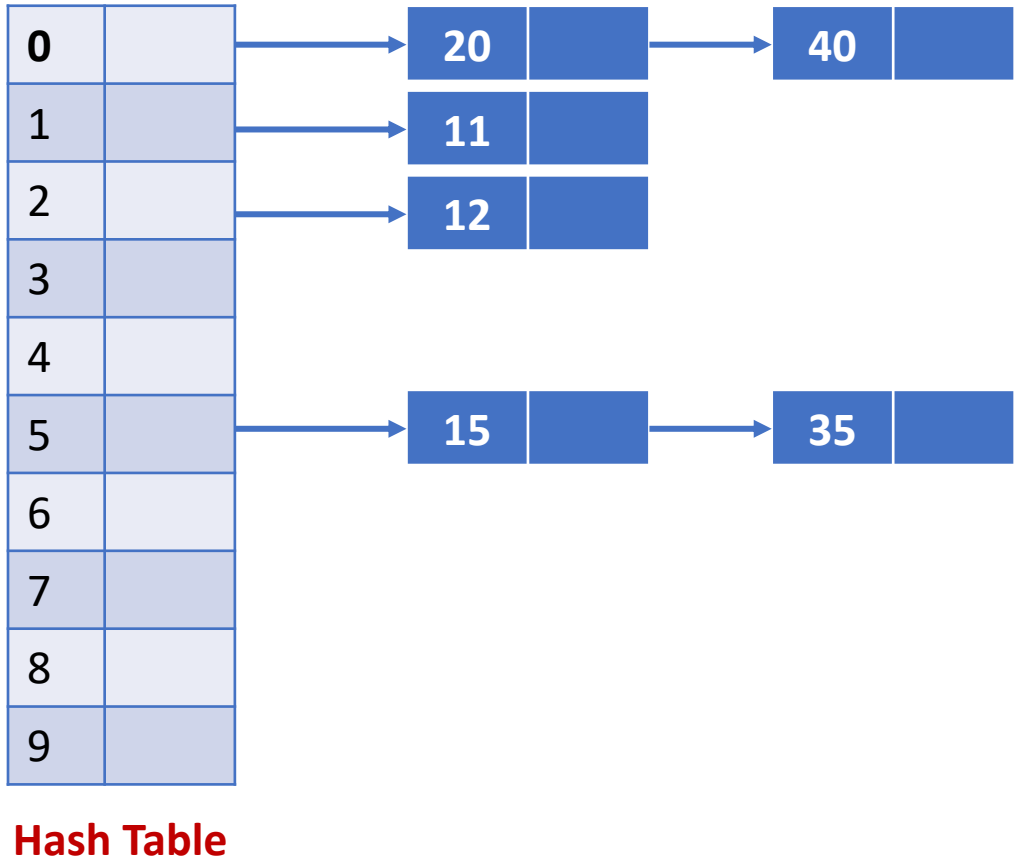


Hash Table

Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

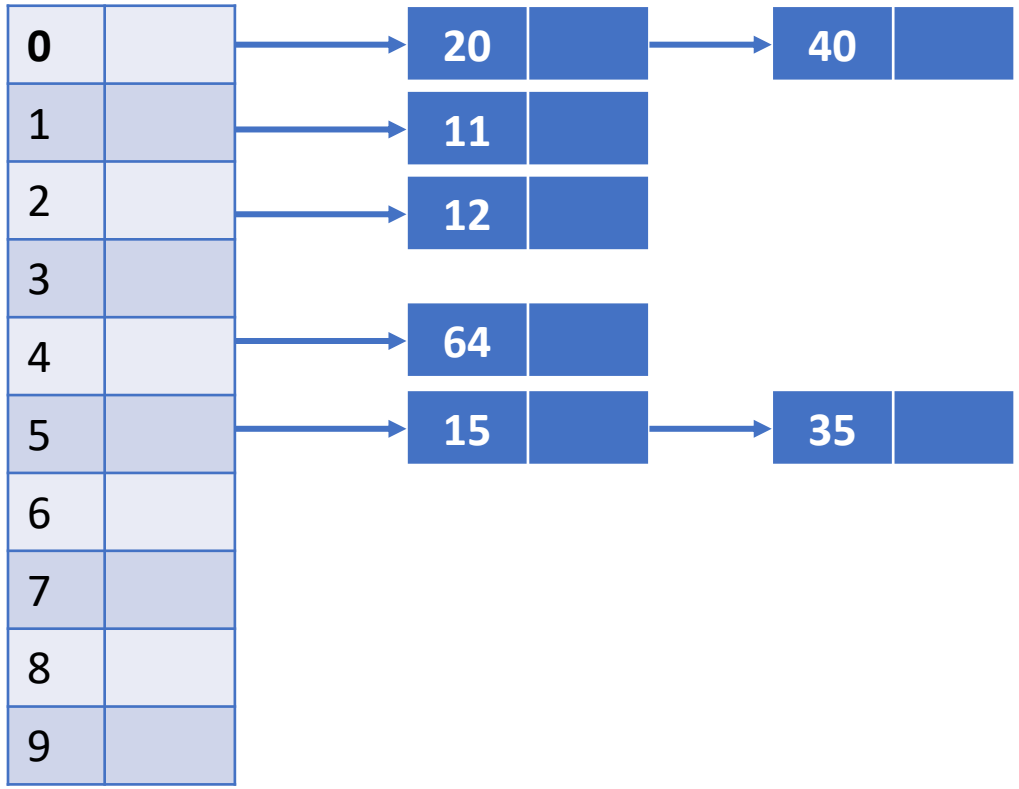
$h(64) = 64 \,\%\, 10 = 4$



Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

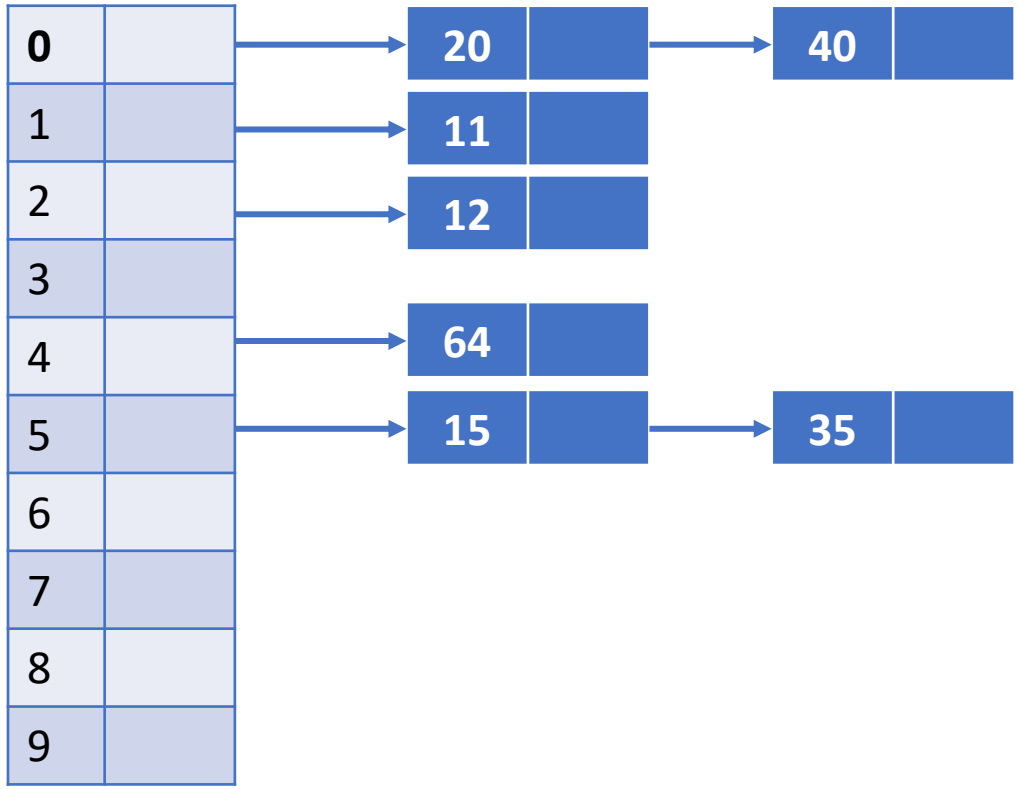
$h(64) = 64 \,\%\, 10 = 4$



Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

$h(4) = 4 \,\%\, 10 = 4$

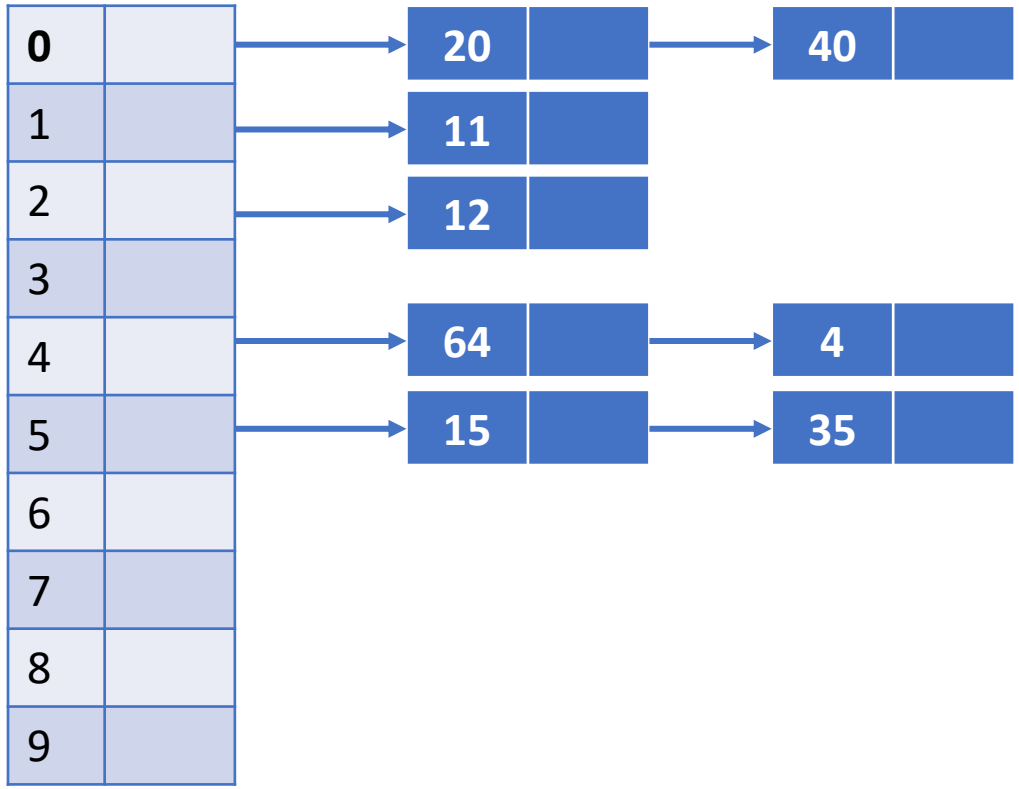


Hash Table

Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

$h(4) = 4 \,\%\, 10 = 4$

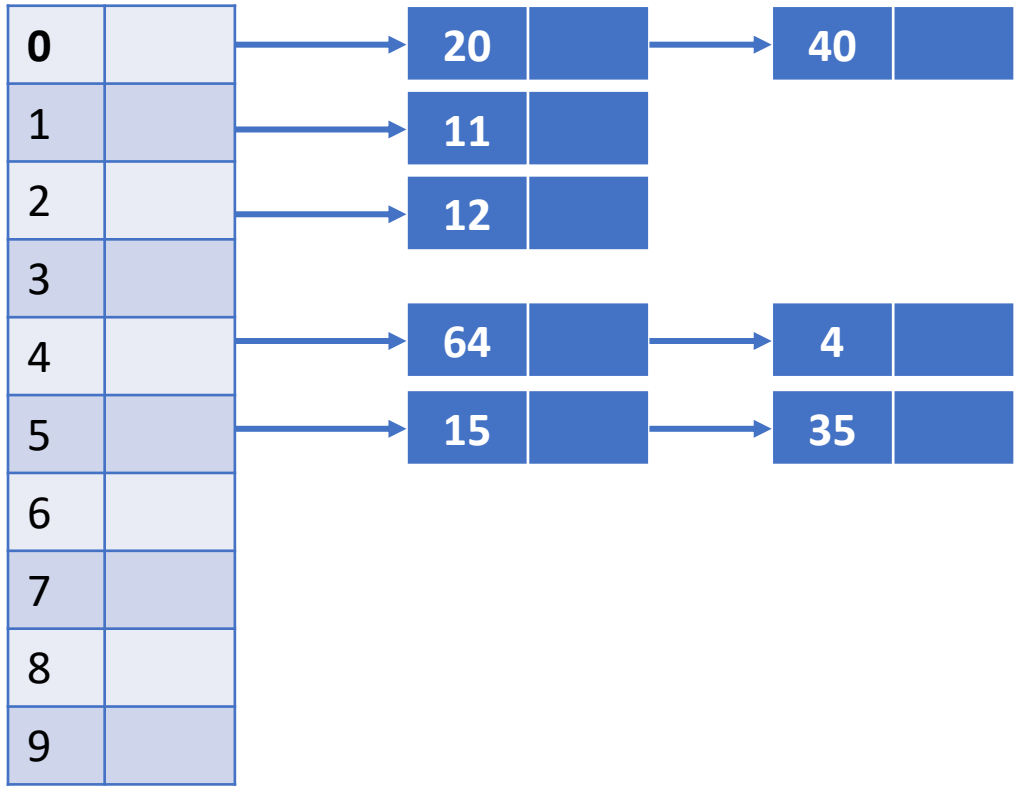


Hash Table

Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

$h(81) = 81 \,\%\, 10 = 1$

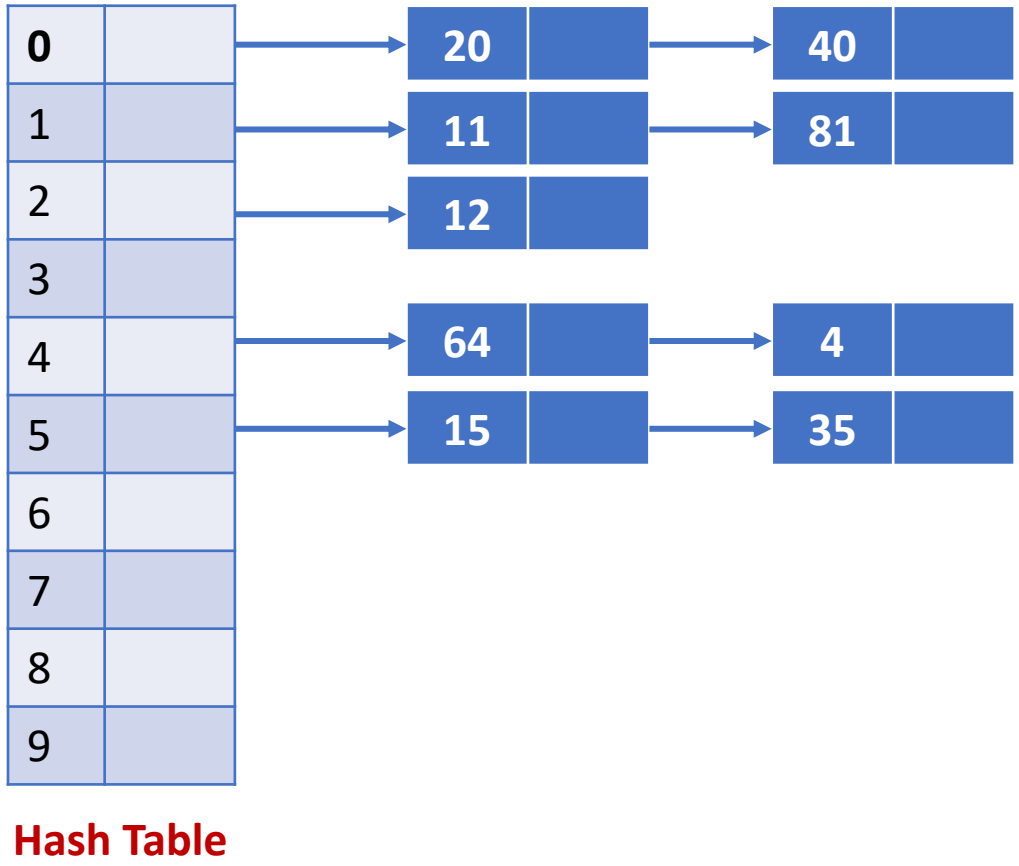


Hash Table

Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

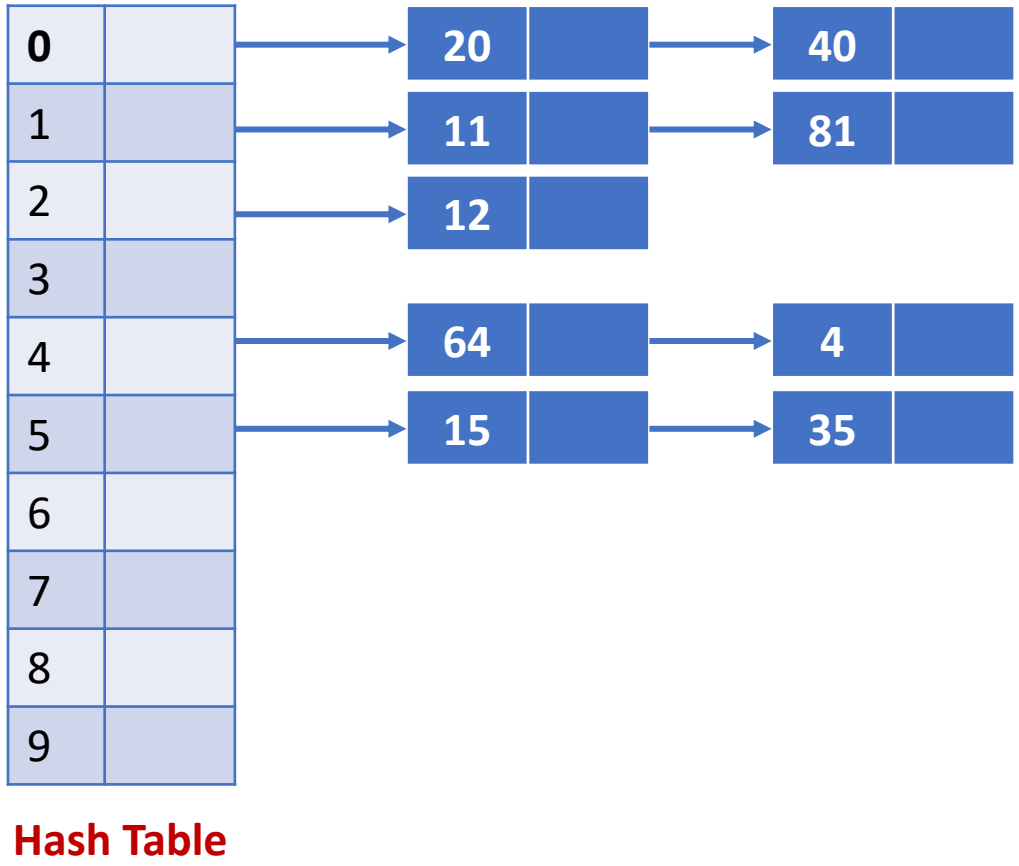
$h(81) = 81 \,\%\, 10 = 1$



Hashing - Open Hashing (Separate Chaining)

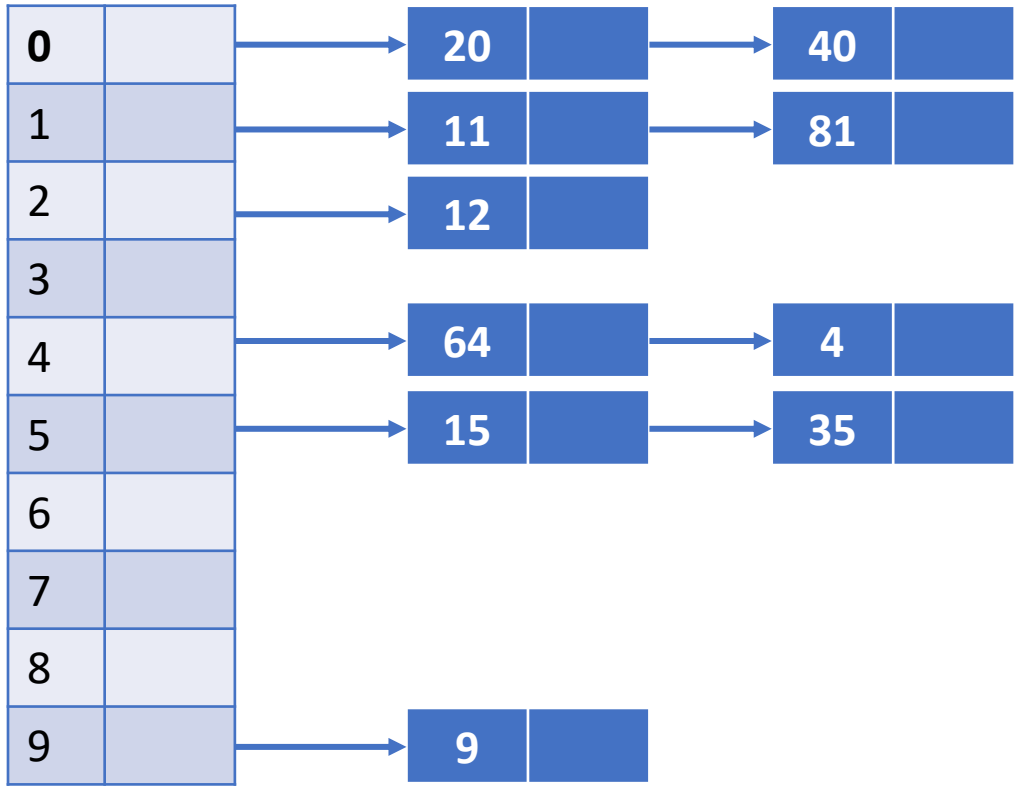
Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9

$h(9) = 9 \,\%\, 10 = 9$



Hashing - Open Hashing (Separate Chaining)

Insert 20, 15, 11, 35, 12, 40, 64, 4, 81, 9



Hash Table

Hashing - Closed Hashing (Open Addressing)

We will explain three common strategies:

1. Linear Probing
2. Quadratic Probing
3. Double hashing

Hashing - Closed Hashing (Open Addressing) Linear Probing

We resolve conflicts by sequentially scanning an array (**with wraparound**) until we find an empty cell.

Example: Assume hash function is given by

$$\mathrm{h}(x) = x \,\%\, 10$$

Then when a collision occurs, we apply the following

$$\mathrm{h}'(x) = \big(h(x) + f(i)\big)\% \, n, \qquad i \geq 0$$

Usually in linear probing, we use

$$\mathrm{f}(i) = i, \qquad i \geq 0$$

Hashing - Closed Hashing (Open Addressing) Linear Probing

$\mathrm{h}^0(x) = (h(x) + 0)\%\, n$

$\mathrm{h}^1(x) = (h(x) + 1)\%\, n$

$\mathrm{h}^2(x) = (h(x) + 2)\%\, n$

…

$\mathrm{h}^i(x) = (h(x) + i)\%\, n$

Hashing - Closed Hashing (Open Addressing) Linear Probing

- Key Space (i.e., Input Space) is **integers**.
- Table Size (i.e., **n**) is **10**.
- Hash Function is the module operator (%).

$$h(x) = x \,\%\, n = x \,\%\, 10$$

- We want to insert the following numbers:

89, 18, 49, 58, 69

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$h(89) = 89 \,\%\, 10 = 9$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$h(89) = 89 \,\%\, 10 = 9$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$h(18) = 18 \,\%\, 10 = 8$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$h(18) = 18 \,\%\, 10 = 8$

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$h(49) = 49 \,\%\, 10 = 9$

Collision!

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$$h(49) = 49 \,\%\, 10 = 9$$

Collision!

$$\Rightarrow (h(49) + 1)\%10 = (9 + 1)\%\, 10 = 10 \,\%\, 10 = 0$$

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$$h(49) = 49 \,\%\, 10 = 9$$

Collision!

$$\Rightarrow (h(49) + 1)\%10 = (9 + 1)\%\, 10 = 10 \,\%\, 10 = 0$$

0	**49**
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$$h(58) = 58 \,\%\, 10 = 8$$

Collision!

0	**49**
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$$h(58) = 58 \,\%\, 10 = 8$$

Collision!

$$\Rightarrow (h(58) + 1) \,\%\, 10 = (8 + 1)\% \, 10 = 9 \,\%\, 10 = 9$$ ✖

0	**49**
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$$h(58) = 58 \,\%\, 10 = 8$$

Collision!

$$\Rightarrow (h(58) + 1) \,\%\, 10 = (8 + 1)\% \, 10 = 9 \,\%\, 10 = 9$$ ✖

$$\Rightarrow (h(58) + 2) \,\%\, 10 = (8 + 2)\% \, 10 = 10 \,\%\, 10 = 0$$ ✖

0	**49**
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$$h(58) = 58\ \%\ 10 = 8$$

Collision!

$$\Rightarrow (h(58) + 1)\ \%\ 10 = (8 + 1)\%\ 10 = 9\ \%\ 10 = 9$$ ✖

$$\Rightarrow (h(58) + 2)\ \%\ 10 = (8 + 2)\%\ 10 = 10\ \%\ 10 = 0$$ ✖

$$\Rightarrow (h(58) + 3)\ \%\ 10 = (8 + 3)\%\ 10 = 11\ \%\ 10 = 1$$

0	**49**
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$$h(58) = 58\ \%\ 10 = 8$$

Collision!

$$\Rightarrow (h(58) + 1)\ \%\ 10 = (8 + 1)\%\ 10 = 9\ \%\ 10 = 9$$ ✖

$$\Rightarrow (h(58) + 2)\ \%\ 10 = (8 + 2)\%\ 10 = 10\ \%\ 10 = 0$$ ✖

$$\Rightarrow (h(58) + 3)\ \%\ 10 = (8 + 3)\%\ 10 = 11\ \%\ 10 = 1$$

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

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$$h(69) = 69 \,\%\, 10 = 9$$

Collision!

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

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$$h(69) = 69 \,\%\, 10 = 9$$

Collision!

$$\Rightarrow (h(69) + 1) \,\%\, 10 = (9 + 1)\% \, 10 = 10 \,\%\, 10 = 0$$ ✖

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

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$$h(69) = 69\ \%\ 10 = 9$$

Collision!

$$\Rightarrow (h(69) + 1)\ \%\ 10 = (9 + 1)\%\ 10 = 10\ \%\ 10 = 0$$ ✖
$$\Rightarrow (h(69) + 2)\ \%\ 10 = (9 + 2)\%\ 10 = 11\ \%\ 10 = 1$$ ✖

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

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$$h(69) = 69 \,\%\, 10 = 9$$

Collision!

$$\Rightarrow (h(69) + 1) \,\%\, 10 = (9 + 1)\% \, 10 = 10 \,\%\, 10 = 0$$ ✖
$$\Rightarrow (h(69) + 2) \,\%\, 10 = (9 + 2)\% \, 10 = 11 \,\%\, 10 = 1$$ ✖
$$\Rightarrow (h(69) + 3) \,\%\, 10 = (9 + 3)\% \, 10 = 12 \,\%\, 10 = 2$$

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

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69

$$h(69) = 69 \,\%\, 10 = 9$$

Collision!

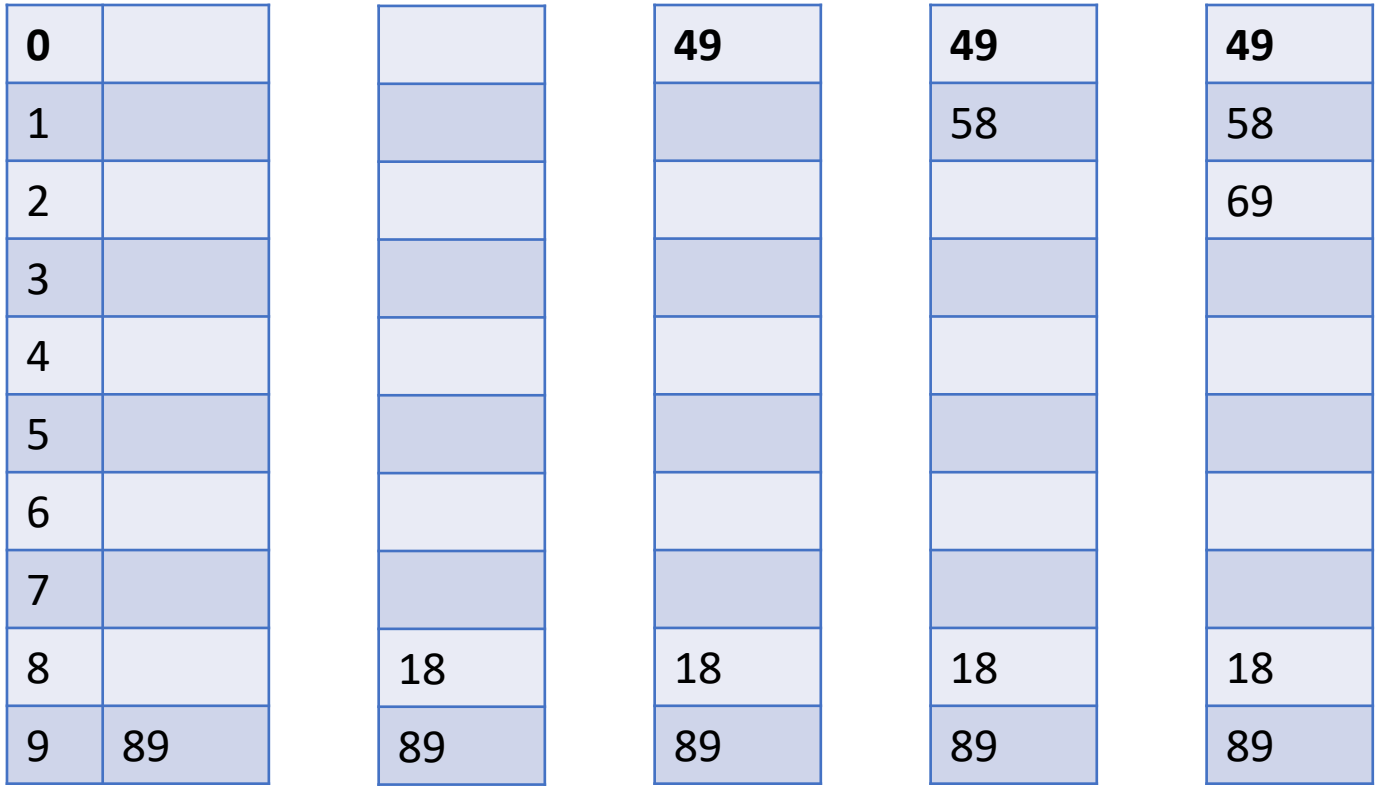
$$\Rightarrow (h(69) + 1) \,\%\, 10 = (9 + 1)\% \, 10 = 10 \,\%\, 10 = 0$$ ✖
$$\Rightarrow (h(69) + 2) \,\%\, 10 = (9 + 2)\% \, 10 = 11 \,\%\, 10 = 1$$ ✖
$$\Rightarrow (h(69) + 3) \,\%\, 10 = (9 + 3)\% \, 10 = 12 \,\%\, 10 = 2$$

0	**49**
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Linear Probing

Insert 89, 18, 49, 58, 69



Hashing - Closed Hashing (Open Addressing) Linear Probing

Advantages (Pros)

1. Simplicity: Very simple to implement.
2. Very Efficient for cache use (Items are near each other)
3. We don’t need additional memory.
4. Guaranteed to find a spot.
5. We can mark cells as deleted and reuse them!

Hashing - Closed Hashing (Open Addressing) Linear Probing

Disadvantages (Cons)

1. Clustering: Data is stored after each other (bad for search performance).
2. Not good for large amount of data.
3. Data is not distributed uniformly to buckets.
4. Delete performance and operation can take a hit.
5. If the hash function is bad it can result in even worse running time!

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

We resolve conflicts by sequentially scanning an array (**with wraparound**) until we find an empty cell.

Example: Assume hash function is given by

$$\mathrm{h}(x) = x \,\%\, 10$$

Then when a collision occurs, we apply the following

$$\mathrm{h}'(x) = (h(x) + i^2)\% \, n, \qquad i \geq 0$$

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

$\mathrm{h}^0(x) = (h(x) + 0^2)\%\, n$

$\mathrm{h}^1(x) = (h(x) + 1^2)\%\, n$

$\mathrm{h}^2(x) = (h(x) + 2^2)\%\, n$

…

$\mathrm{h}^i(x) = (h(x) + i^2)\%\, n$

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

- Key Space (i.e., Input Space) is **integers**.
- Table Size (i.e., **n**) is **10**.
- Hash Function is the module operator (%).

$$h(x) = x \,\%\, n = x \,\%\, 10$$

- We want to insert the following numbers:

89, 18, 49, 58, 69

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 89, 18, 49, 58, 69

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 89, 18, 49, 58, 69

$h(89) = 89 \,\%\, 10 = 9$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 89, 18, 49, 58, 69

$h(89) = 89 \,\%\, 10 = 9$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 89, 18, 49, 58, 69

$h(18) = 18 \,\%\, 10 = 8$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 89, 18, 49, 58, 69

$h(18) = 18 \,\%\, 10 = 8$

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 89, 18, 49, 58, 69

$h(49) = 49 \,\%\, 10 = 9$

Collision!

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 89, 18, 49, 58, 69

$$h(49) = 49\ \%\ 10 = 9$$

Collision!

$$\Rightarrow (h(49) + 1^2)\%10 = (9 + 1)\%\ 10 = 10\ \%\ 10 = 0$$

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 89, 18, 49, 58, 69

$$h(49) = 49 \,\%\, 10 = 9$$

Collision!

$$\Rightarrow (h(49) + 1^2)\%10 = (9 + 1)\%\, 10 = 10 \,\%\, 10 = 0$$

0	**49**
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 89, 18, 49, 58, 69

$$h(58) = 58 \,\%\, 10 = 8$$

Collision!

0	**49**
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 89, 18, 49, 58, 69

$$h(58) = 58 \,\%\, 10 = 8$$

Collision!

$$\Rightarrow (h(58) + 1^2) \,\%\, 10 = (8 + 1) \%\, 10 = 9 \,\%\, 10 = 9$$ ✖

0	**49**
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 89, 18, 49, 58, 69

$$h(58) = 58\ \%\ 10 = 8$$

Collision!

$$\Rightarrow (h(58) + 1^2)\ \%\ 10 = (8 + 1)\%\ 10 = 9\ \%\ 10 = 9$$ ✖

$$\Rightarrow (h(58) + 2^2)\ \%\ 10 = (8 + 4)\%\ 10 = 12\ \%\ 10 = 2$$

0	**49**
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 89, 18, 49, 58, 69

$$h(58) = 58\ \%\ 10 = 8$$

Collision!

$$\Rightarrow (h(58) + 1^2)\ \%\ 10 = (8 + 1)\%\ 10 = 9\ \%\ 10 = 9$$ ✖

$$\Rightarrow (h(58) + 2^2)\ \%\ 10 = (8 + 4)\%\ 10 = 12\ \%\ 10 = 2$$

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

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 89, 18, 49, 58, 69

$$h(69) = 69 \,\%\, 10 = 9$$

Collision!

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

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 89, 18, 49, 58, 69

$$h(69) = 69\ \%\ 10 = 9$$

Collision!

$$\Rightarrow (h(69) + 1^2)\ \%\ 10 = (9 + 1)\%\ 10 = 10\ \%\ 10 = 0$$ ✖

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

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 89, 18, 49, 58, 69

$$h(69) = 69\ \%\ 10 = 9$$

Collision!

$$\Rightarrow (h(69) + 1^2)\ \%\ 10 = (9 + 1)\%\ 10 = 10\ \%\ 10 = 0$$
$$\Rightarrow (h(69) + 2^2)\ \%\ 10 = (9 + 4)\%\ 10 = 13\ \%\ 10 = 3$$

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

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 89, 18, 49, 58, 69

$$h(69) = 69 \,\%\, 10 = 9$$

Collision!

$$\Rightarrow (h(69) + 1^2) \,\%\, 10 = (9 + 1)\% \, 10 = 10 \,\%\, 10 = 0$$
$$\Rightarrow (h(69) + 2^2) \,\%\, 10 = (9 + 4)\% \, 10 = 13 \,\%\, 10 = 3$$

0	**49**
1	
2	58
3	69
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Rule

If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

Table Size = the first prime > 2 * n

$$h(x) = x \,\%\, TableSize$$

Proof

Refer to the book

0	**49**
1	
2	58
3	69
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Counter Example

- Key Space (i.e., Input Space) is **integers**.
- Table Size (i.e., **n**) is **16**.
- Hash Function is the module operator (%).

$$h(x) = x \,\%\, n = x \,\%\, 16$$

- We want to insert the following numbers:

0, 1, 4, 9, 16

Hashing - Closed Hashing (Open Addressing) Quadratic Probing

Insert 0, 1, 4, 9, 16

- $h(0) = 0 \,\%\, 16 = 0$
- $h(1) = 1 \,\%\, 16 = 1$
- $h(4) = 4 \,\%\, 16 = 4$
- $h(9) = 9 \,\%\, 16 = 9$
- $h(16) = 16 \,\%\, 16 = 0$
  - $h(16) = (0 + 1) \,\%\, 16 = 1$
  - $h(16) = (0 + 2^2) \,\%\, 16 = 4$
  - $h(16) = (0 + 3^2) \,\%\, 16 = 9$
  - $h(16) = (0 + 4^2) \,\%\, 16 = 0$
  - $h(16) = (0 + 5^2) \,\%\, 16 = 9$
  - $h(16) = (0 + 6^2) \,\%\, 16 = 4$

Hashing - Closed Hashing (Open Addressing) Double Hashing

We resolve conflicts by sequentially scanning an array (**with wraparound**) until we find an empty cell.

Example: Assume hash functions are given by

$$h_1(x) = x \,\%\, 10$$
$$h_2(x) = 7 - (x \,\%\, 7)$$

Then when a collision occurs, we apply the following

$$\mathrm{h}'(x) = (h_1(x) + i * h_2\,(x))\%\, n, \qquad i \geq 0$$

Good choice for h_2 is to choose a prime R < Table Size

$$h_2(x) = R - (x \,\%\, R)$$

Hashing - Closed Hashing (Open Addressing) Double Hashing

$\mathrm{h}^0(x) = (h_1(x) + 0 * h_2(x))\%\, n$

$\mathrm{h}^1(x) = (h_1(x) + 1 * h_2(x))\%\, n$

$\mathrm{h}^2(x) = (h_1(x) + 2 * h_2(x))\%\, n$

…

$\mathrm{h}^i(x) = (h_1(x) + i * h_2(x))\%\, n$

Hashing - Closed Hashing (Open Addressing) Double Hashing

- Key Space (i.e., Input Space) is **integers**.
- Table Size (i.e., **n**) is **10**.
- Hash Functions are:

$$h_1(x) = x \,\%\, n = x \,\%\, 10$$

$$h_2(x) = 7 - (x \,\%\, 7)$$

- We want to insert the following numbers:

89, 18, 49, 58, 69

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Table

Hashing - Closed Hashing (Open Addressing) Double Hashing

Insert 89, 18, 49, 58, 69

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Table

Hashing - Closed Hashing (Open Addressing) Double Hashing

Insert 89, 18, 49, 58, 69

$h(89) = 89 \,\%\, 10 = 9$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Table

Hashing - Closed Hashing (Open Addressing) Double Hashing

Insert 89, 18, 49, 58, 69

$h(89) = 89 \,\%\, 10 = 9$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Double Hashing

Insert 89, 18, 49, 58, 69

$h(18) = 18 \,\%\, 10 = 8$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Double Hashing

Insert 89, 18, 49, 58, 69

$h(18) = 18 \,\%\, 10 = 8$

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Double Hashing

Insert 89, 18, 49, 58, 69

$h(49) = 49 \,\%\, 10 = 9$

Collision!

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Double Hashing

Insert 89, 18, 49, 58, 69

$$h(49) = 49 \,\%\, 10 = 9$$

Collision!

$$\Rightarrow (h_1(49) + 1 * h_2(49))\%10 = (9 + (7 - 49 \,\%\, 7))\%\, 10 = (9 + 7) \,\%\, 10 = 6$$

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Hash Table

Hashing - Closed Hashing (Open Addressing) Double Hashing

Insert 89, 18, 49, 58, 69

$$h(49) = 49 \,\%\, 10 = 9$$

Collision!

$$\Rightarrow (h_1(49) + 1 * h_2(49))\%10 = (9 + (7 - 49 \,\%\, 7))\%\, 10 = (9 + 7) \,\%\, 10 = 6$$

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

Hash Table

Hashing - Closed Hashing (Open Addressing) Double Hashing

Insert 89, 18, 49, 58, 69

$$h(58) = 58 \,\%\, 10 = 8$$

Collision!

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

Hash Table

Hashing - Closed Hashing (Open Addressing) Double Hashing

Insert 89, 18, 49, 58, 69

$$h(58) = 58\ \%\ 10 = 8$$

Collision!

$$\Rightarrow (h_1(58) + 1 * h_2(58))\%10 = (8 + (7 - 58\ \%\ 7))\%\ 10 = (8 + 5)\ \%\ 10 = 3$$

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

Hash Table

Hashing - Closed Hashing (Open Addressing) Double Hashing

Insert 89, 18, 49, 58, 69

$$h(58) = 58\ \%\ 10 = 8$$

Collision!

$$\Rightarrow (h_1(58) + 1 * h_2(58))\%10 = (8 + (7 - 58\ \%\ 7))\%\ 10 = (8 + 5)\ \%\ 10 = 3$$

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

Hash Table

Hashing - Closed Hashing (Open Addressing) Double Hashing

Insert 89, 18, 49, 58, 69

$$h(69) = 69 \,\%\, 10 = 9$$

Collision!

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

Hash Table

Hashing - Closed Hashing (Open Addressing) Double Hashing

Insert 89, 18, 49, 58, 69

$$h(69) = 69\ \%\ 10 = 9$$

Collision!

$$\Rightarrow (h_1(69) + 1 * h_2(69))\%10 = (9 + (7 - 69\ \%\ 7))\%\ 10 = (9 + 1)\ \%\ 10 = 0$$

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

Hash Table

Hashing - Closed Hashing (Open Addressing) Double Hashing

Insert 89, 18, 49, 58, 69

$$h(69) = 69\ \%\ 10 = 9$$

Collision!

$$\Rightarrow (h_1(69) + 1 * h_2(69))\%10 = (9 + (7 - 69\ \%\ 7))\%\ 10 = (9 + 1)\ \%\ 10 = 0$$

0	**69**
1	
2	
3	58
4	
5	
6	49
7	
8	18
9	89

Hash Table

Hashing - Load Factor

Let N = number of items we want to store in the table

$$Load\ Factor\ (\lambda) = \frac{\text{N}}{\text{Table Size}}$$

Example

Table Size = 50, N = 5 then λ = 0.1

Hashing - Rehashing

Rehashing is the process of transferring items into a bigger table (i.e., array) when load factor (λ) exceeds a particular value (usually half the size). In other words, we build/create another table and transfer items to it!

A good choice for the size of the new table is:

$$New\ Size = First\ Prime > 2\ * Old\ Size$$

Hashing - Rehashing

Rehashing is the process of transferring items into a bigger table (i.e., array) when load factor (λ) exceeds a particular value (usually half the size). In other words, we build/create another table and transfer items to it!

A good choice for the size of the new table is:

$$New\ Size = First\ Prime > 2\ * Old\ Size$$

When load factor (λ) gets large, number of collisions increases (number of probes increase), so inserting new items/searching for items takes longer time. Moreover, inserting might fail!

Hashing - Rehashing

Rehashing is the process of transferring items into a bigger table (i.e., array) when load factor (λ) exceeds a particular value (usually half the size). In other words, we build/create another table and transfer items to it!

A good choice for the size of the new table is:

$$New\ Size = First\ Prime > 2\ * Old\ Size$$

When load factor (λ) gets large, number of collisions increases (number of probes increase), so inserting new items/searching for items takes longer time. Moreover, inserting might fail!

We usually rehash when:

- First insert fails.
- Table is half full (λ = 0.5), in other words, when load factor is 50%
- When load factor is 75%

Hashing - Rehashing

Assume we are using Linear Probing with $h(x) = x \ \% \ 5$

0	**4**
1	
2	2
3	
4	9

Insert 9, 4, 2

Hashing - Rehashing

Assume we are using Linear Probing with $h(x) = x \ \% \ 5$

0	**4**
1	
2	2
3	
4	9

$$New\ Size = First\ Prime > 2 * Old\ Size$$

$$New\ Size = First\ Prime > 2 * 5$$

$$New\ Size = First\ Prime > 10$$

$$New\ Size = 11$$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Insert 9, 4, 2

Hashing - Rehashing

Assume we are using Linear Probing with $h(x) = x \ \% \ 5$

0	**4**
1	
2	2
3	
4	9

Insert 9, 4, 2

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

Hashing - Key Space

What if input is String and not integers?

Hashing - Key Space

What if input is String and not integers?

1. $h(x) = \left(\sum_{i=0}^{n-1} ASCII(x_i)\right)\%N$

   tea, **eat** and all **anagrams** will cause collisions

2. $h(x) = \left(\sum_{i=0}^{2} ASCII(x_i) * 27^i\right)\%N$

   tea, **tea**m and all words with the same 3 first character's cause collisions

3. $h(x) = \left(\sum_{i=0}^{n-1} ASCII(x_i) * 32^i\right)\%N$

Hashing - Key Space

```
int h1(String s) {
    int answer = 0;
    for (int i = 0; i < s.length(); i++) {
        answer += s.charAt(i);
    }

    return answer;
}

int h2(String s) {
    int answer = 0;
    for (int i = 0; i <= 2 && i < s.length(); i++) {
        answer += s.charAt(i) * (int)Math.pow(27, i);
    }

    return answer;
}
```

```
int h3(String s) {
    int answer = 0;
    for (int i = 0; i < s.length(); i++) {
        answer += s.charAt(i) * (int)Math.pow(32, i);
    }

    return answer;
}
```

Hashing - Key Space

How to make 32^i better and why we choose it?

Hashing - Key Space

How to make 32^i better and why we choose it?

$y = x * 2^n = x \ll n$ (Shift left)

$y = \frac{x}{2^n} = x \gg n$ (Shift Right)

$32^i = 2^{5i} = 2 \ll 5i$ (This is very quick!)

$$h(x) = \left(\sum_{i=0}^{n-1} ASCII(x_i) * 32^i \right) \% N$$

Hashing - Key Space

Assume we have “Milk”

$$h(x) = \left(\sum_{i=0}^{n-1} ASCII(x_i) * 32^i \right) \%N$$

$$h(x) = ASCII(M) * 32^0 + ASCII(i) * 32^1 + ASCII(l) * 32^2 + ASCII(k) * 32^3$$

Hashing - Key Space

Assume we have “Milk”

$$h(x) = \left(\sum_{i=0}^{n-1} ASCII(x_i) * 32^i\right) \%N$$

$$h(x) = ASCII(M) * 32^0 + ASCII(i) * 32^1 + ASCII(l) * 32^2 + ASCII(k) * 32^3$$

Horner’s Rule

$$ax^5 + bx^4 + cx^3 + dx^2 + ex + f$$

$$(((\big((ax + b)x + c\big)x + d)x + e)x + f$$

Hashing - Key Space

Assume we have “Milk”

$$h(x) = \left(\sum_{i=0}^{n-1} ASCII(x_i) * 32^i \right) \%N$$

```
int hash(String s) {
    int answer = 0;
    for (int i = 0; i < s.length(); i++) {
        answer = (answer * 32) + s.charAt(i);
    }

    return answer;
}
```

Hashing - Key Space

Assume we have “Milk”

$$h(x) = \left(\sum_{i=0}^{n-1} ASCII(x_i) * 32^i \right) \%N$$

```
int hash(String s) {
    int answer = 0;
    for (int i = 0; i < s.length(); i++) {
        answer = (answer << 5) + s.charAt(i);
    }

    return answer;
}
```

Hashing - Key Space

Any Questions?