CSC3100 Tutorial 10
Hashing

Jingjing Qian
jingjingqian@link.cuhk.edu.cn

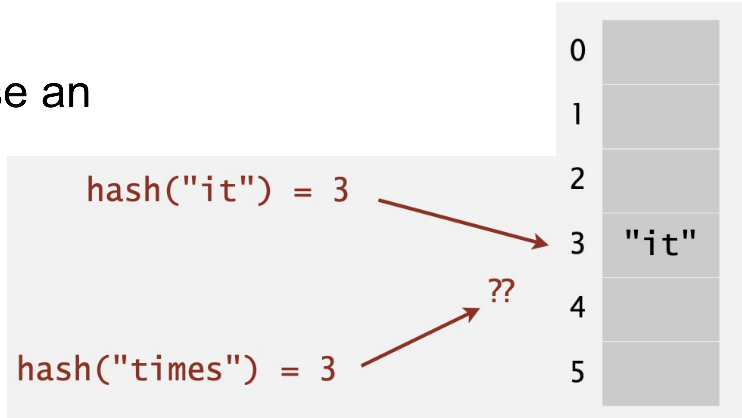
Review

	Insert	Search
ordered array (keys are not indexes)	O(N)	O(logN)
ordered linked list	O(N)	O(N)
unordered array (keys are not indexes)	O(N)	O(N)
unordered linked list	O(1)	O(N)
binary search tree	O(logN)	O(logN)

Searching takes at least O(log n) time.
We can do better with hashing.

Review

- The key idea of hashing is using a function h to map a large universe U to a small range $\{0, 1, 2, \cdots, m-1\}$. Then we can use an array A of size m to store.
- $A[i]$ is called a **slot**. The function h is called **hash function**.



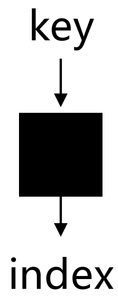
- We should try to ensure that for $x \neq y$, $h(x) \neq h(y)$. When two different inputs passed to the hash function produce the same hash value, for $x \neq y$, $h(x) = h(y)$, **Hash collisions** occur.
- We should find some good hash function to reduce collisions. And we should use some technics to ensure that if collision happens, we can also get the correct answer.

- **Load factor** is defined as

$$\alpha = \frac{|U|}{m} = \frac{\#\, elements}{\#\, slots}$$

Hash function

- Hash function: method for computing hash table index from key.



- Idealistic goal. Scramble the keys uniformly to produce a table index.
  - Efficiently computable.
  - Each table index equally likely for each key.

Ex. Phone numbers.
- Bad: first three digits.
- Better: last three digits.

Hash function

- For numeric keys,
  - When U is small, we can use an injection (injective function) to map U to $\{0, 1, 2, . . . , m - 1\}$, Then there are no collisions. In each slot, we can also store the element itself. Given $U = \{a, a + 1, a + 2, \cdots , b\}$ and $b - a$ is small, we can use the hash function

$$h(x) = x - a.$$

  - When U is large, for example, $U = \{0, 1, 2, \cdots , 2^{32} - 1\}$, we can not use an injection because it will make m large. If U is a set of integer, we can use division hashing:

Key mod TableSize m

where we usually choose m a prime number not close to the power of 2 or 10

If $m = 10^p$ or 2^p, then $h(k)$ only uses the lowest-order p digits of the key value k. **Unless it is known that all low-order p bit patterns are equally likely, it is better to make the hash function depend on all the bits of the key.**

8237643 mod 1000=643

Hash function

- Sometimes, U is not a set of integer.

  Given U = {string of length 3} and the charset is ASCII, we can use the hash function

  $$h\ (s) = s[0] + s[1] \times 128 + s[2] \times 128^2$$

  It means s can be treated as an integer of base 128.

Collision resolution - Chaining

- **Chaining:**
  - The idea is to make each slot of the hash table point to a linked list of records that have the same hash function value.
  - When collision happens, we store multiple elements in the linked list.
  - Chaining is simple but requires additional memory outside the table. Load factor α can be larger than 1.
  - Less sensitive to hash functions.

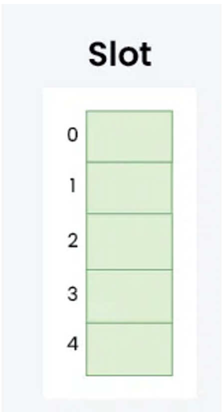
- **Add:**

Ex.

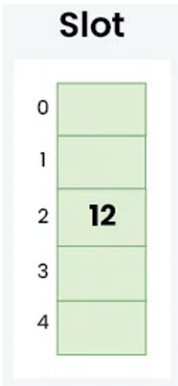
Hash function = key % 5, Elements = 12, 22, 15 and 25.

Chaining

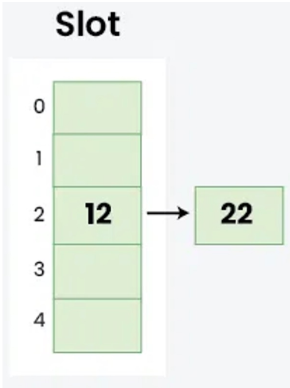
- **Add:** Ex. Hash function = key % 5,
  Elements = 12, 22, 15 and 25



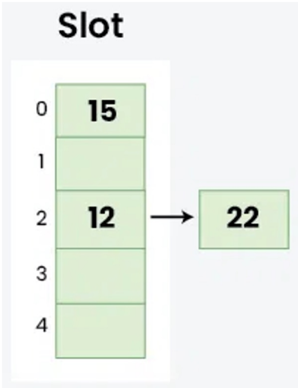
Step 1:
Empty hash table with range of hash values from 0 to 4 according to the hash function provided. Each entry storing a linked list.



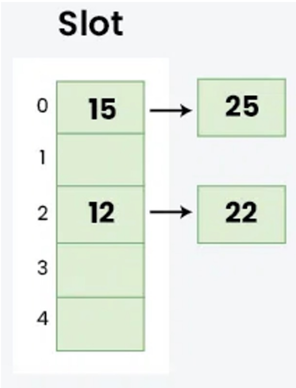
Step 2:
12 % 5 = 2.
Insert 12 to the linked list at **slot2.**



Step 3:
22 % 5 = 2
Insert 22 to the linked list at **slot2.**



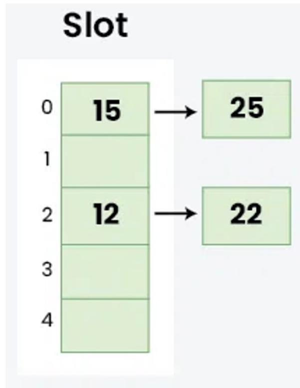
Step 4:
15 % 5 = 0
Insert 15 to the linked list at **slot0.**



Step 5:
25 % 5 = 0
Insert 25 to the linked list at **slot0.**

Chaining

- **Search** for a record with key k
  - Retrieve the linked list according to $h(k)$
  - Search the linked list.



- **Delete** record with key k
  - Retrieve the linked list according to $h(k)$
  - Delete node in the linked list.

Collision resolution - Open-addressing

- In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. So the load factor always $\alpha \leq 1$.
- When the load factor α is greater than a threshold (0.75 in practice), we choose a larger m' ($2m$ in practice) to construct a new hash table. Then the elements in the old hash table are added into the new hash table one by one.

- When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table:
  - We extend the hash function as
  $$h'(x, i) : U \times \{0, 1, \cdots, m-1\} \to \{0, 1, \cdots, m-1\}.$$
  - When we add an element x into the hash table, we get a probe sequence
  $$[h'(x, 0), h'(x, 1), h'(x, 2), \cdots, h'(x, m-1)].$$

Open-addressing

Add

We sequentially check whether the slot is empty.
If so, we place x in that slot. Otherwise, we continue checking.

$\textsc{Open-Addressing-Hash-Table-Add}(x, value)$

```
for pos ∈ [h′(x, 0), h′(x, 1), h′(x, 2), ⋯, h′(x, m − 1)]
        if A[pos] = NIL
                A[pos] = x ↦ value
```

Open-addressing

- **Hash function h'**

  - **Linear Probing**
    If in case the location that we get is already occupied, then we check for the next location.
    $$h'(x, i) = (h(x) + i) \bmod m.$$

  - **Quadratic Probing**
    Take the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.
    $$h'(x, i) = (h(x) + i^2) \bmod m.$$

  - **Double hashing**
    Make use of two hash function
    $$h'(x, i) = (h_1(x) + ih_2(x)) \bmod m.$$
    Note that gcd $(h_2(x), m) = 1$ for all x.

Open-addressing

Add: Take Quadratic Probing for example.

Ex.

Table Size = 7, hash function as Hash(x) = x % 7,
collision resolution strategy to be f(i) = i^2 . Insert = 22, 30, and 50

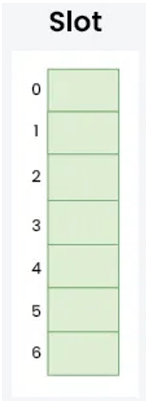
- **Quadratic Probing**
  Take the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

$$h'(x, i) = (h(x) + i^2) \bmod m.$$

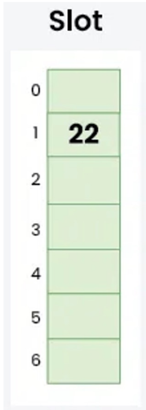
Open-addressing

Add: Ex.

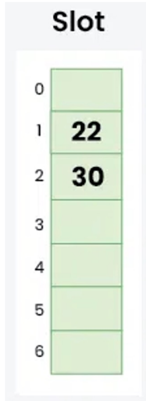
Table Size = 7, hash function as Hash(x) = $x \% 7$,
collision resolution strategy to be f(i) = i^2 . Insert = 22, 30, and 50



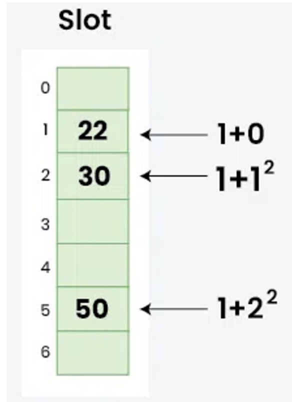
Step 1:
Empty hash table with range of hash values from 0 to 6 according to the Table Size provided.



Step 2:
22 % 7 = 1,
slot1 is empty.
Insert 22 to **slot1.**



Step 3:
30 % 7 = 2,
slot2 is empty.
Insert 30 to **slot2.**



Step 4:
50 % 7 = 1. **Slot1** is occupied.
1 + 1^2 = 2. **Slot2** is occupied.
1 + 2^2 = 5. **Slot5** is empty.
Insert 50 to **slot5.**

Open-addressing

- **Search** for a record with key x

  Sequentially check the slot in probe sequence $[h'(x, 0), h'(x, 1), h'(x, 2) \ldots h'(x, m\text{-}1)]$.
  - If all slots are empty, no record with key x in the hash table.
  - If we find the the slot with key equal to x, we find the desired element.

```
OPEN-ADDRESSING-HASH-TABLE-FIND(x)
for pos ∈ [h′(x, 0), h′(x, 1), h′(x, 2), ⋯, h′(x, m − 1)]
    if A[pos] = NIL
        return NIL
    elseif A[pos].key = x
        return A[pos].value
```

Open-addressing

- **Delete:** We cannot mark the deleted one as empty:

Assume that there exist $x_0 \neq x_1 \neq x_2$ such that the probe sequence of x_0, x_1, x_2 are equal. We add x_0, x_1, x_2 to the hash table sequentially:

x_0 in the slot of $h'(x_0, 0)$.
x_1 in the slot of $h'(x_0, 1)$.
x_2 in the slot of $h'(x_0, 2)$.

If we delete x_1 from the hash table, x_2 can not be found. Because after finding that $A[h'(x_2, 1)]$ is empty, we will not look for $A[h'(x_2, 2)]$ anymore.

```
OPEN-ADDRESSING-HASH-TABLE-FIND(x)
for pos ∈ [h'(x, 0), h'(x, 1), h'(x, 2), ⋯, h'(x, m − 1)]
    if A[pos] = NIL
        return NIL
    elseif A[pos].key = x
        return A[pos].value
```

Open-addressing

We should mark the deletion differently.

```
OPEN-ADDRESSING-HASH-TABLE-2-DELETE(x)
for pos ∈ [h′(x, 0), h′(x, 1), h′(x, 2), ⋯, h′(x, m − 1)]
    if A[pos] = NIL
        return
    elseif A[pos] = DELETED
        continue
    elseif A[pos].key = x
        return A[pos] = DELETED
```

Open-addressing

Adding and **finding** should also be modified.

```
OPEN-ADDRESSING-HASH-TABLE-2-ADD(x)
for pos ∈ [h′(x, 0), h′(x, 1), h′(x, 2), ⋯, h′(x, m − 1)]
    if A[pos] = NIL or A[pos] = DELETED
        A[pos] = x ↦ value
```

```
OPEN-ADDRESSING-HASH-TABLE-2-FIND(x)
for pos ∈ [h′(x, 0), h′(x, 1), h′(x, 2), ⋯, h′(x, m − 1)]
    if A[pos] = NIL
        return
    elseif A[pos] = DELETED
        continue
    elseif A[pos].key = x
        return A[pos].value
```

Application: an example

LeetCode P1: Two sum

Given an array of integers nums and an integer target, return *indices of the two numbers such that they add up to target*.
You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.
You can return the answer in any order.
Only one valid answer exists.

Example 1:
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].

Example 2:
Input: nums = [3,2,4], target = 6
Output: [1,2]

Example 3:
Input: nums = [3,3], target = 6
Output: [0,1]

Solution 1: Brute Force

Time complexity: $O(n^2)$
Space complexity: O(1)

```python
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        n = len(nums)
        for i in range(n - 1):
            for j in range(i + 1, n):
                if nums[i] + nums[j] == target:
                    return [i, j]
        return []  # No solution found
```

Solution 2: Hash

We can think it's O(1)

- Two-pass: Store elements and their indices in a hash table.

- One-pass: Iterate through the array once, and for each element, check if the target minus the current element exists in the hash table. If it does, we have found a valid pair of numbers. If not, we add the current element to the hash table.

Time complexity: O(n)
Space complexity: O(n)

```python
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        numMap = {}
        n = len(nums)

        # Build the hash table
        for i in range(n):
            numMap[nums[i]] = i

        # Find the complement
        for i in range(n):
            complement = target - nums[i]
            if complement in numMap and numMap[complement] != i:
                return [i, numMap[complement]]

        return []  # No solution found
```

Two-pass

```python
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        numMap = {}
        n = len(nums)

        for i in range(n):
            complement = target - nums[i]
            if complement in numMap:
                return [numMap[complement], i]
            numMap[nums[i]] = i

        return []  # No solution found
```

One-pass

Thanks!