

# **CSC263H**

## **Data Structures and Analysis**

**Prof. Bahar Aameri & Prof. Marsha Chechik**

Winter 2024 – Week 5

## Dictionary ADT:

- **Objects:** A collection of **key-value pairs** (keys are *unique*).
- **Operations:**
  - *Search*( $D, k$ ): return  $x$  in  $D$  s.t.  $x.key = k$ , or NIL if no such  $x$  is in  $D$ .
  - *Insert*( $D, x$ ): insert  $x$  in  $D$ ; if some  $y$  in  $D$  has  $y.key$  equal to  $x.key$ , replace  $y$  by  $x$ .
  - *Delete*( $D, x$ ): remove  $x$  from  $D$ .

# Data Structures for Dictionaries: Hash Tables

**Problem 1:** Read a grade file, where grades are integers between 0 to 99. Keep track of number of occurrences of each grade.

**Fastest Solution:** Create an array  $T$  of size 100.  $T[i]$  stores the number of occurrences of grade  $i$ .

**Problem 2:** Read a data file, keep track of number of occurrences of each integer value (from 0 to  $2^{32} - 1$ ).

**Fastest Solution:** Create an array of size  $2^{32}$ , as above.

**Wasteful** use of memory, especially when data are files relatively small.

**Problem 3:** Read a text file, keep track of number of occurrences of each word.

Cannot use **keys as indices** anymore!

1. We need to be able to convert any type of key to an integer.
2. We need to map the universe of keys into a small number of slots.

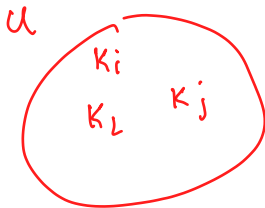
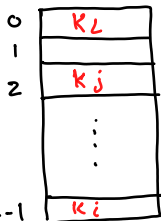
A **hash function** does both!

# Hash Table

- **Universe  $\mathcal{U}$ :** The set of all possible keys.
- **Hash Function:** A function from the set of all possible keys to integers between 0 and  $m - 1$ :  
 $h : \mathcal{U} \rightarrow \{0, 1, \dots, m - 1\}.$

**Hash Table:** A data structure containing an **array of length  $m$**  and a **hash function**  
 $h : \mathcal{U} \rightarrow \{0, 1, \dots, m - 1\}.$

- $h(k)$  maps a key  $k$  to one of the  $m$  positions in hash table  $T$ .  
That is,  $h(k)$  is the the **index** at which the key  $k$  is stored.
- Each array location called a slot or a bucket.



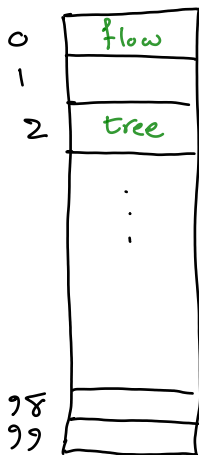
$$\begin{aligned}h(k_i) &= m-1 \\h(k_L) &= 0 \\h(k_j) &= 2\end{aligned}$$

# Hash Table: Collisions

- If  $m \geq |\mathcal{U}|$ , then there exists a hash function  $h$  which maps each key to a unique slot (no collisions).  
Such a function is called a **perfect hash function**.  
Typically the number of possible keys is **much bigger** than the number of array slots.
- If  $m < |\mathcal{U}|$ , then at least one **collision** occurs.

### Example:

Suppose  $\mathcal{U}$  is the set of all English words and  $m = 100$



$$h(\text{tree}) = 2$$

$$h(\text{flow}) = 0$$

$$h(\text{ten}) = 2$$

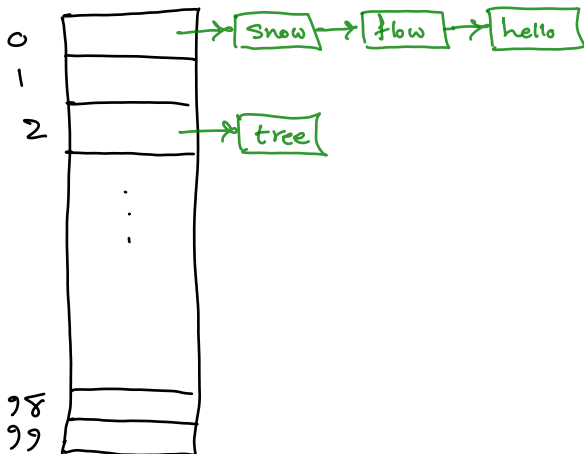
---

Handling Collisions:

- Chaining (Closed Addressing).
- Open Addressing.

# Hash Table: Chaining

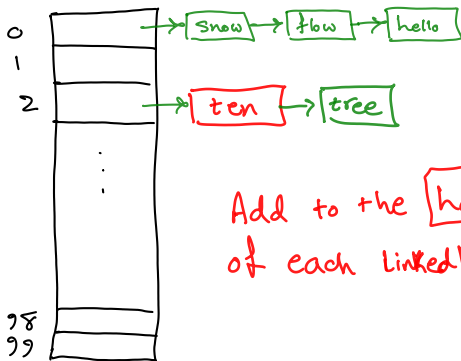
**Chaining:** Each **bucket** in the array points to a **linked list** of key-value pairs.



### HashInsert( $k, v$ ):

1. Compute  $h(k)$ . Set  $i = h(k)$ .
2. Search the linked list stored at  $T[i]$  to check whether an element with key  $k$  already exists.
3. If so, replace the existing value with  $v$ .  
If not, insert a new node to the head of the list.

T:



HashInsert( $ten, Nil$ )

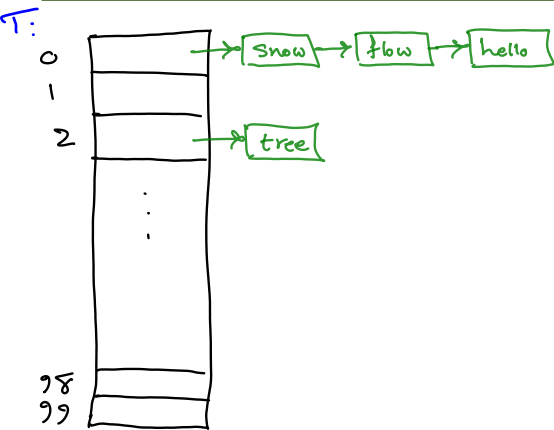
$$h(ten) = 2$$

Add to the **head**  
of each linkedlist



### HashSearch( $k$ ):

1. Compute  $h(k)$ . Set  $i = h(k)$ .
2. Access index  $i$  in the table.
3. Search the linked list stored at  $T[i]$ .



Hashsearch(flow):

$$h(\text{flow}) = 0 \rightarrow T[0] \checkmark$$

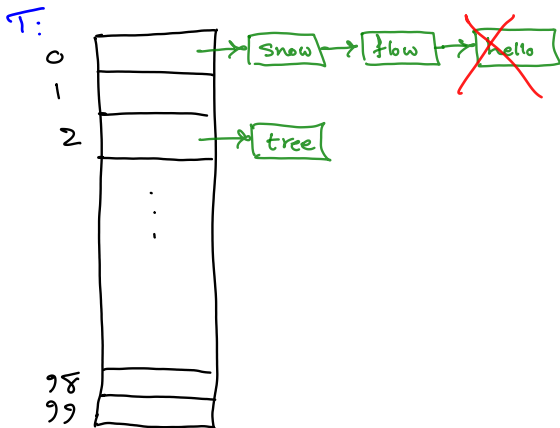
---

HashSearch(hey)

$$h(\text{hey}) = 2 \rightarrow T[2] \times$$

### HashDelete( $k$ ):

1. Compute  $h(k)$ . Set  $i = h(k)$ .
2. Search the linked list stored at  $T[i]$ .
3. If an element with key  $k$  is found, delete it from the list.



HashDelete(hello)

$h(\text{hello}) = 0 \rightarrow T[0]$

## Chaining: Worst-Case Running Time

Worst Case for Search: All elements in the table are hashed into the same bucket

$n$  is the total number of elements stored in the hashtable

Worst-case Running Time:

- HashSearch:  $\Theta(n)$
- HashInsert:  $\Theta(n)$
- HashDelete:  $\Theta(n)$

We assume that the hash value  $h(k)$  is computed in constant time.  
That is, the run time for computing  $h(k)$  is in  $\Theta(1)$ .

**Practical consideration:** Hash tables work well in real-world applications. That is because:

- The worst case almost never happens.
- Average case performance is really efficient.

# Chaining: Search Average-Case Running Time

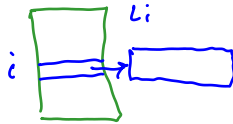
## Analyzing Average-Case Running Time – Review

1. Define a random variable  $t_n(x)$  denoting the number of steps executed by the algorithm on an input  $x$  with size  $n$ .
2. Compute  $\mathbb{E}[t_n]$ .

- Let  $t_{m,n}(k)$  denote the number of steps executed by *HashSearch* to find  $k$  in a hash table containing  $m$  buckets and storing  $n$  elements.
- **Simple Uniform Hashing Assumption (SUHA):** Any key equally likely to hash to any bucket.

$$\mathbb{E}[t_{m,n}(k)] = 1 + \text{expected running time of searching for } k \text{ in } L_i$$

$$i = \underbrace{h(k)}_{\Theta(1)}$$



# Chaining: Search Average-Case Running Time

Expected Run Time in an **Unsuccessful** Search (under SUHA):

$$\mathbb{E}[t_{m,n}(k)] = 1 + \text{expected length of } L_i = 1 + \frac{n}{m}$$

- Probability that key  $k$  is hashed to bucket  $i$  (i.e., the probability of  $h(k) = i$ ):  
There are  $m$  candidate position for  $k$ ,  $k$  is equally likely to hash to any of them:

$$\Pr[h(k) = i] = \frac{1}{m}$$

- The expected length of the linked list stored at a bucket  $i$ :

$$\mathbb{E}[\text{len}_i] = \frac{n}{m}$$

where  $\text{len}_i$  denotes length of the linked list stored in bucket  $i$ .

## Chaining: Search Average-Case Running Time

**Load Factor** of a hash table  $T$ : The ratio of the number of keys, denoted by  $n$ , stored in  $T$  to the number of buckets of  $T$ , denoted by  $m$ .

$$\alpha = \frac{n}{m}$$

$$E[t_{m,n}(k)] = 1 + \alpha \\ \in \Theta(1 + \alpha)$$

If  $\frac{n}{m} \in \Theta(1)$ , then the average-case run time is also in  $\Theta(1)$

# Chaining: Search Average-Case Running Time

## Expected Run Time in a **Successful** Search (under SUHA):

That is  $k$  is a key that exists in the hash table.

Let  $k_1, k_2, k_3, \dots, k_n$  be the *order of insertion* into the hash table.

$k$  could be  $k_1$ , or  $k_2$ , or  $k_3$ , or ..., or  $k_n$ .

The probability that  $k$  is  $k_i$  ( $1 \leq i \leq n$ ) is:  $\frac{1}{n}$

So the expected number of steps to find  $k$  is the sum over:

the probability that  $k$  is  $k_i$ , times the number of steps required to find  $k_i$

$$\begin{aligned}\mathbb{E}[t_{m,n}(k)] &= \frac{1}{n} \times S_1 + \frac{1}{n} \times S_2 + \frac{1}{n} \times S_3 + \dots + \frac{1}{n} \times S_n \\ &= \frac{1}{n} \sum_{i=1}^n S_i\end{aligned}$$

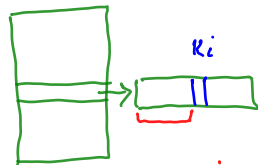
$S_i$  denotes the expected number of steps to find  $k_i$ .

$S_i$  = expected number of steps to find  $k_i$

= number of elements examined during search for  $k_i$

= 1 + number of elements **before**  $k_i$  in the linked list stored at  $h(k_i)$

= 1 + number of keys that hash **same as**  $k_i$  and are inserted **after**  $k_i$ .



b.c. we insert  
at the front

Let  $X_i$  be the expected number of keys that hash samely as  $k_i$ .  
Define **indicator random variables** for  $X_i$ :

$$X_{i,j} = \begin{cases} 1 & \text{if } h(k_i) = h(k_j) \\ 0 & \text{otherwise.} \end{cases}$$

### Indicator Random Variables – Reminder

Recall that indicator random variables are  $X_1, X_2, \dots, X_m$  s.t.:

- $X = X_1 + X_2 + \dots + X_m$ ;
- Each  $X_i$  has only two possible values: 0 or 1.

Then  $\mathbb{E}[X]$  is computed as follows:

$$\begin{aligned} \mathbb{E}[X] &= \mathbb{E}[X_1 + \dots + X_m] \\ &= \mathbb{E}[X_1] + \dots + \mathbb{E}[X_m] && \text{(by linearity of expectation)} \\ &= Pr[X_1 = 1] + \dots + Pr[X_m = 1] \end{aligned}$$

where the last equality holds because for each  $X_i$ :

$$\mathbb{E}[X_i] = 0 \times Pr[X_i = 0] + 1 \times Pr[X_i = 1] = Pr[X_i = 1].$$



# Chaining: Search Average-Case Running Time


Let  $X_i$  be the expected number of keys that hash same as  $k_i$ .

Define **indicator random variables** for  $X_i$ :

$$X_{i,j} = \begin{cases} 1 & \text{if } h(k_i) = h(k_j) \\ 0 & \text{otherwise.} \end{cases}$$

Example:

4 keys are hashed same as  $k_1 \rightarrow X_1 = 4$



The diagram shows four keys labeled  $k_2$ ,  $k_{10}$ ,  $k_{15}$ , and  $k_{20}$  in red. Red arrows point from each of these keys towards a common point centered above the position of  $k_1$  in the text above, illustrating that these four keys hash to the same bucket as  $k_1$ .

$$X_{1,2} = 1$$

$$X_{1,10} = 1$$

$$X_{1,15} = 1$$

$$X_{1,20} = 1$$

$$X_{i,j} = \begin{cases} 1 & \text{if } h(k_i) = h(k_j) \\ 0 & \text{otherwise.} \end{cases}$$

$$\mathbb{E}[X_{i,j}] = \Pr[X_{i,j}] = \frac{1}{m}$$

$S_i = 1 + \text{number of keys that are hashed same as } k_i \text{ and are inserted after } k_i$

$$= 1 + \sum_{j=i+1}^n \mathbb{E}[X_{i,j}]$$

$$= 1 + \sum_{j=i+1}^n \frac{1}{m}$$

$$\mathbb{E}[t_{m,n}(k)] = \frac{1}{n} \sum_{i=1}^n S_i = \frac{1}{n} \sum_{i=1}^n \left( 1 + \sum_{j=i+1}^n \frac{1}{m} \right)$$

$$= 1 + \frac{n}{2m} - \frac{1}{m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$$

$$\in \Theta(1 + \alpha)$$



### Average-case Running Time:

- *HashSearch*:  $\Theta(1 + \alpha)$
- *HashInsert*:  $\Theta(1 + \alpha)$
- *HashDelete*:  $\Theta(1 + \alpha)$

If the number of hash-table slots is **at least proportional** to the number of elements in the table, then  $\alpha \in \Theta(1)$ .

That is, all dictionary operations can be implemented with **constant** average run time.

# Hash Table: Open Addressing

**Open Addressing:** Store all items directly in  $T$  (no chaining).

If a collision occurred, look for another free spot in some **systematic manner** called **probing**.

**Implication:** The number of keys stored in the hash table cannot exceed the length of the array.

T:

0	
1	
2	tree
3	hey
4	hello
5	
6	

Insert "tree", where

$$h(\text{tree}) = 2$$

Insert "hey", where

$$h(\text{hey}) = 3$$

Insert "hello", where

$$h(\text{hello}) = 2$$

→ try  $T[2]$  x

try  $T[3]$  x

try  $T[4]$  ✓

**Probe Sequence:** Sequence of buckets to try.

**Search:** Follow the same probing approach used for insertion.  
Search returns *None* when encounters the first bucket that stores *None*.

**Implication:** Searching for an item requires examining more than just one spot.

T:

0	
1	
2	tree
3	three
4	snow
5	hey
6	

Search for "hey", where

$h(\text{hey}) = 3 \rightarrow \text{Try } T[3] \text{ X}$   
 $\rightarrow \text{Try } T[4] \text{ X}$   
 $\rightarrow \text{Try } T[5] \checkmark$

---

Search for "hello", where

$h(\text{hello}) = 4 \rightarrow \text{Try } T[4] \text{ X}$   
 $\rightarrow \text{Try } T[5] \text{ X}$

$\rightarrow \text{Try } T[6] \rightarrow \text{empty so return Fail}$

**Linear Probing:** Examine a linear sequence of slots.

Example probe sequence:  $h(k)$ ,  $h(k) + 1$ ,  $h(k) + 2$ ,  $h(k) + 3$ , ...

**Probe sequence:**  $(h(k) + i) \bmod m$ , for an integer  $i \geq 0$ .

**Problem:** Contiguous blocks of occupied locations (clusters) are created, causing further insertions of keys into any of these locations to take a long time.

# Open Addressing: Quadratic Probing

**Quadratic Probing:** Examine a non-linear sequence of slots.

Example probe sequence:  $h(k)$ ,  $h(k) + 2$ ,  $h(k) + 6$ ,  $h(k) + 12$ , ...

**Probe sequence:**  $(h(k) + c_1 \times i + c_2 \times i^2) \bmod m$ , for an integer  $i \geq 0$ .

$$c_1, c_2 = 1$$



Insert "hello", where  $i=0$

$$h(\text{hello}) = 2 \rightarrow T[2] \times$$

$i=1$

$$h(\text{hello}) + 1 \times 1 + 1 \times 1 = 4 \rightarrow T[4] \times$$

$\begin{matrix} \swarrow & \downarrow & \downarrow & \downarrow \\ c_1 & i & c_2 & i^2 \end{matrix}$

$$(2 + 1 \times 2 + 1 \times 4) \bmod 7 = 1$$



**Problems:**

- Collisions still cause a milder form of clustering.
- Need to be careful with the values of  $c_1$  and  $c_2$ , as it could jump in such a way that some of the buckets are never reachable.

# Open Addressing: Double Hashing

**Double Hashing:** Use a second hash function to generate step values that depends on the key.

Example probe sequence:  $h_1(k)$ ,  $h_1(k) + h_2(k)$ ,  $h_1(k) + 2h_2(k)$ ,  $h_1(k) + 3h_2(k)$ , ...

**Probe sequence:**  $(h_1(k) + i \times h_2(k)) \bmod m$ , for an integer  $i \geq 0$ .

**T:**

0	
1	hello
2	tree
3	
4	snow
5	flow
6	

Insert "hello", where

Try  $T[2]$   $\times$   $\uparrow$   $h_1(\text{hello}) = 2$  and  $h_2(\text{hello}) = 3$

$i \leftarrow$

$h_1(\text{hello}) + 1 \times h_2(\text{hello}) = 2 + 1 \times 3 = 5$

Try  $T[5]$   $\times$   $\uparrow$   $5$

$i \leftarrow$

$(h_1(\text{hello}) + 2 \times h_2(\text{hello})) \bmod 7 =$

$(2 + 2 \times 3) \bmod 7 = 1 \rightarrow \text{Try } T[1] \checkmark$

Under **Simple Uniform Hashing Assumption** (SUHA):

- Average-case number of probes in an **unsuccessful** search:  $\frac{1}{1-\alpha}$ .
- Average-case number of probes in an **successful** search:  $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ .

(Proof in Section 11.4 of CLRS (optional)).

In **practice** open addressing works best when  $\alpha < 0.5$ .

# Hash Functions

A **good** hash function  $h$  should:

- ensure simple uniform hashing.
- $h(x)$  should depends on every part/bit of  $x$  (even for complex objects).
- should spreads out values.
- should be possible to be computed in constant time (i.e., in  $\Theta(1)$ ).

In practice, it is difficult to get all of these properties, but there are some **heuristics** that work well.

**Important Note:** When answering questions in assignments and tests of this course, you can assume that a good hash function exists, unless the question explicitly says otherwise.

## Intuition: Interpreting keys as integers

- Every object stored in a computer can be represented by a bit-string (string of 0's and '1s).
- The bit-string can be considered as base 2 representation of a non-negative integer.
- That is, any type of key can be converted to an integer.
- The integer value, however, is usually very larger than the number of buckets.

So a hash function needs to map larger integers to a small set of integers  $\{0, 1, \dots, m-1\}$  ( $0, 1, \dots, m-1$  represent indexes of the buckets).

# Implementing Hash Functions: Division Method

**Division method:**  $h(k) = k \bmod m$

**Pitfall:** Sensitive to the value of  $m$ .

Example 1:  $k \bmod 10$  depends only on last decimal digit of  $k$ .

Example 2:  $k \bmod 8$  depends only on last 3 bits of  $k$ .

Example 3:  $k \bmod 2^p$  depends only on last  $p$  bits of  $k$ .

**Implication:** Keys are not spread out.

**Good choice for  $m$ :** A prime not too close to an exact power of 2.

That is, the size of the hash table should be a prime.

**Pitfall:** Constrains the table size.

## Multiplication Method:

1. Multiply  $k$  by a real constant  $0 < A < 1$ . → mess-up  $k$  by multiplying  $A$
2. Let  $x$  be the fractional part of  $k \times A$  (note that  $0 < x < 1$ ). → take the fractional part of the mess
3.  $h(k) = \lfloor m \times x \rfloor$ . → multiply  $m$  to make sure the result is between 0 and  $m - 1$ .

- The optimal choice for  $A$  depends on the characteristics of the data being hashed.
- Tends to evenly distribute the hash values, because of the mess-up.
- Not sensitive to the value of  $m$  (unlike division method).

- Exercises in Chapter 4 of the course notes.
- Problems 11.1-1, 11.2-1, 11.2-2 in CLRS.
- Optional Readings: CLRS Sections 11.1, 11.2, 11.3 (except 11.3.3)