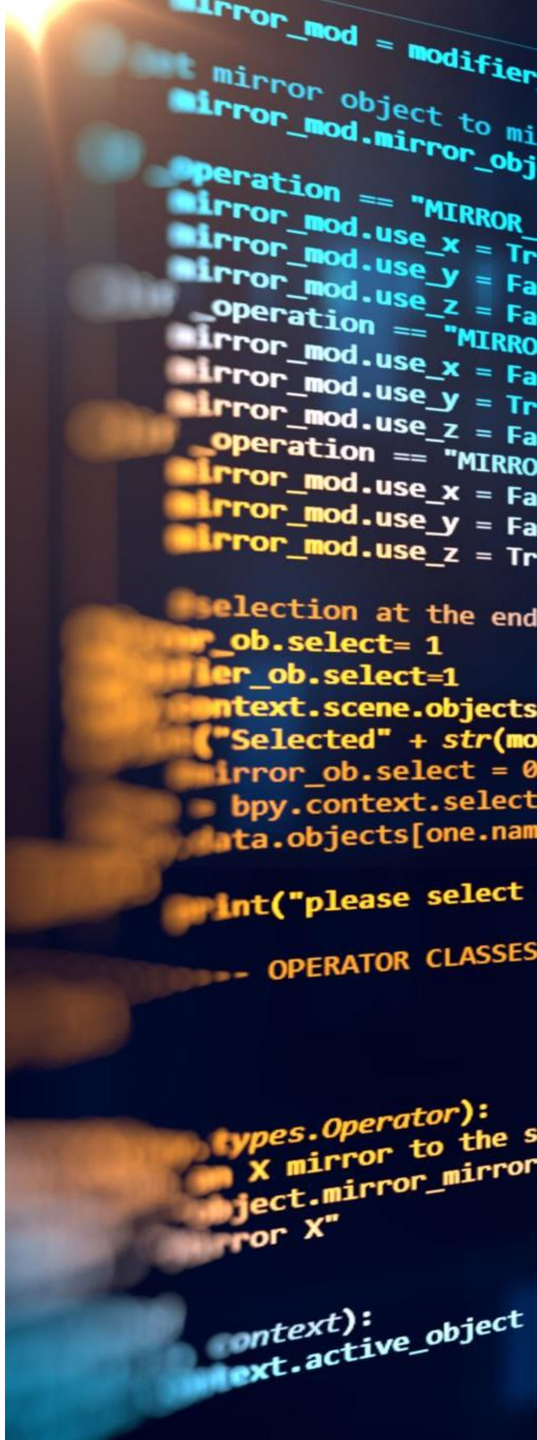
Please use the following QR code to check in and record your attendance.

CS 1037
Fundamentals of Computer Science II

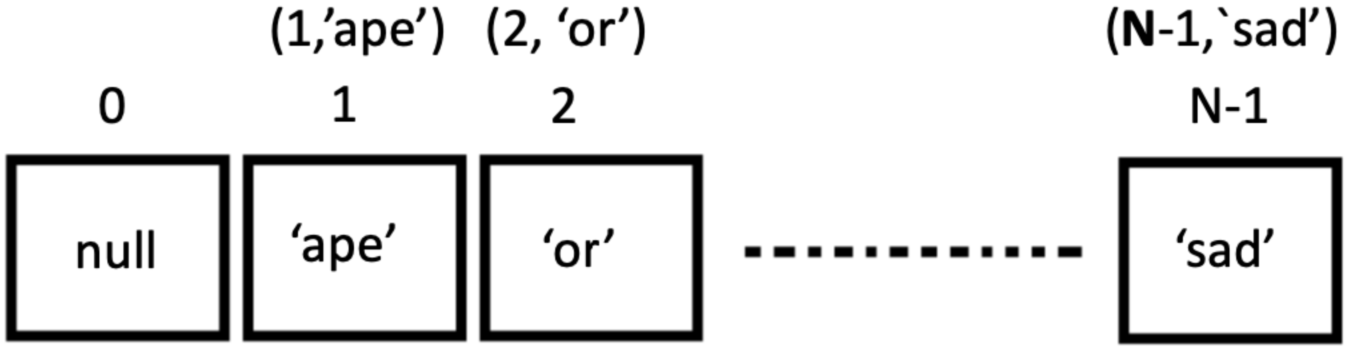
Hash Table

Ahmed Ibrahim



Key-Value Entries

- You have at most $\boldsymbol{N}$ entries $(\boldsymbol{k}, \boldsymbol{v})$
- Suppose keys k are unique integers between 0 to $N - 1$
- Create initially empty array A of size N
- Store (k, v) in A[k]
- Example



- Main operations (insert, find, remove) are O(1)
- We need O(N) space

Imagine that!

- What if we have 100 entries with integer keys, 0 to 1,000,000,000?
  - Do we still have O(1) insert(), delete(), find()?
  - We do not want 1,000,000,000 memory cells to store only 100 entries.
- What should we do?

The Concept of Hash Table

- Array of Fixed Size (***TableSize***)
- Each key is mapped into some number between 0 and (***TableSize - 1***). Mapping is done by something called the ***hash function***
- *The hash function ensures that two distinct keys are assigned to different cells.*
- Given the finite number of cells and an almost limitless supply of keys, a **hash function** is necessary to evenly distribute the keys among the cells!

0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	

Hash Tables and Hash Functions

- A hash table is a data structure that allows for efficient storage and retrieval of **key-value pairs** using a **hash function**.

- **Hash Tables**
  - **provide** fast data retrieval and insertion, typically in constant time, O(1), under ideal conditions.
  - **use** a hash function to transform input data (keys) into a fixed-size numerical value, determining where the data is stored in the table.

What if *keys* are NOT integers?

Key	Value
“Paul”	29
“Jane”	35
“Chloe”	88
“Alex”	18

Hashing Non-Integer Keys

- Array **A** of size **N** = 8
- Design function $h(k)$ that maps key $\boldsymbol{k}$ into integer range $0, 1, \dots, \boldsymbol{N} - 1$
- Entry with key k **is** stored at index $h(\boldsymbol{k})$ in the array **A**

How far ‘p’ from ‘a’ => 15 $h(k)$ => 15 mod 8 = 7

Hashing value

Distance = ASCII code of k−ASCII code of ’a’.

- ASCII code of `'p'` = 112.
- ASCII code of `'a'` = 97.
- Distance from `'p'` to `'a'`: $112 - 97 = 15$.

Alex 18	Jane 35	Chloe 88					Paul 29

Key	Value
“Paul”	29
“Jane”	35
“Chloe”	88
“Alex”	18

Component Sum Hash Code: String

- This method involves breaking a string into individual components (e.g., characters), converting them into numerical values (e.g., ASCII values), and summing them up to compute the hash code.
- Example:

  String: "post"

  ASCII values of characters: 'p' = 112, 'o' = 111,

  's' = 115, 't' = 116

  Hash code: h("post") = 112 + 111 + 115 + 116 = 454

- **Advantages**: Easy to implement, requires minimal computation, and works well for small, uniform-length strings.
- **Limitation**: High collision risk for anagrams (e.g., "stop" and "pots").

Source: https://junminlee3.medium.com/hash-tables-animations-that-will-make-you-understand-how-they-work-d1bcc850ba71

Hash Code: Polynomial Accumulation

- A more sophisticated method where a polynomial factor weights the position of each character in the string.
- Example:

  String: "post", constant c=33

  ASCII values of characters: 'p' = 112, 'o' = 111, 's' = 115, 't' = 116

  Hash code: $h(\text{"post"}) = (112 \cdot 33^0)+(111 \cdot 33^1)+(115 \cdot 33^2)+(116 \cdot 33^3)$

  $h(\text{"post"}) = 112+3663 + 125565 + 1334028 = 1469368$
- **Advantages**: The polynomial weighting ensures that even small changes in the string significantly alter the hash code. The position of characters impacts the hash, differentiating anagrams like "post" and "stop".

Memory Address Hash Code

- In the context of C programming, a **memory address** hash code can be represented as an integer value derived from the memory address of a variable or object, typically using pointers.
- This approach involves taking the object's pointer (memory address) and using it directly as the hash value or applying a simple transformation.
- For instance, the memory address can be cast to an integer type to produce the hash code. This is often used to enable efficient access to objects.
    - Example: Address of a: 0x7ffeef24f8a4 => 2147483620 // Cast the 64-bit address to a 32-bit unsigned integer

```c
unsigned int hash(void *ptr) {
    return (unsigned int)(ptr); // Cast pointer to an integer type
}
```

Memory Address Hash Code (cont.)

- This approach is simple and efficient when the goal is to distinguish between objects based on their memory locations.
- Drawback: Objects with identical content but stored in different memory locations will produce different hash codes. For example:
  - `char str1[] = "Hello";`
  - `char str2[] = "Hello";`
  - `printf("Hash of str1: %u\n", hash(str1)); // Hash of str1: 134512345`
  - `printf("Hash of str2: %u\n", hash(str2)); // Hash of str2: 134512789`
- Although str1 and str2 have the same content, their memory addresses differ, resulting in different hash codes.

Basic Operations

Following are the basic primary operations of a hash table.

- Search – Searches an element in a hash table.
- Insert – inserts an element in a hash table.
- Delete – Deletes an element from a hash table.

Data Item:

```
struct DataItem {
    int data;
    int key;
};
```

Hash Method:

```
int hashCode(int key){
    return key % SIZE;
}
```

Search Operation

Time Complexity O(1)

```
// Search for a key in the hash table
DataItem* search(int key) {
int hashIndex = hashCode(key); // Compute the hash index

// Check if the slot at hashIndex is NULL
if (hashArray[hashIndex] == NULL) {
return NULL; // Key not found
}

// Check if the key matches
if (hashArray[hashIndex]->key == key) {
return hashArray[hashIndex]; // Key found
}

// If key doesn't match, return NULL (direct access assumes no
probing)
return NULL; // Key not found
}
```

Insert Operation

Time Complexity O(1)

```
// Insert a key-value pair into the hash table
void insert(int key, int data) {
// Allocate memory for the new DataItem
DataItem* item = (DataItem*)malloc(sizeof(DataItem));
item->data = data;
item->key = key;
int hashIndex = hashCode(key); // Compute the hash index

// Check if the slot at hashIndex is already occupied
if (hashArray[hashIndex] != NULL) {
printf("Error: Key %d maps to an occupied slot (collision).\n", key);
free(item); // Free the allocated memory to avoid a memory leak
return;}

// Place the item in the hash table
hashArray[hashIndex] = item;
printf("Inserted key %d with value %d at index %d\n", key, data,
hashIndex);}
```

Delete Operation

Time Complexity O(1)

```c
// Delete a key from the hash table (direct access)
void delete(int key) {
// Compute the hash index
int hashIndex = hashCode(key);

// Check if the slot at hashIndex contains the key
if (hashArray[hashIndex] != NULL && hashArray[hashIndex]->key ==
key) {
  free(hashArray[hashIndex]); // Free the memory
  hashArray[hashIndex] = NULL; // Mark the slot as empty
  printf("Deleted key %d from index %d\n", key, hashIndex);
} else {printf("Key %d not found. Cannot delete.\n", key);}
}
```

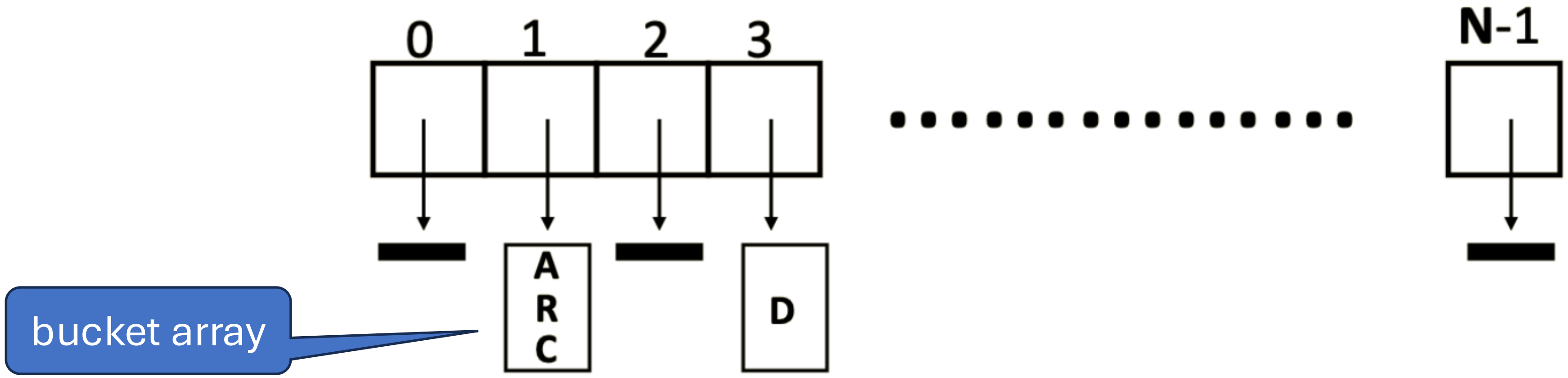
Collision

- Collisions occur when different elements are mapped to the same cell.
- Collision resolution strategies
  - **Separate Chaining –** Store colliding keys in a **linked list** at the same hash table index
  - **Open Addressing –** Store colliding keys elsewhere on the table

Source: https://junminlee3.medium.com/hash-tables-animations-that-will-make-you-understand-how-they-work-d1bcc850ba71

Collision Resolution by Chaining

- What if you still have N keys, which may not be unique? (1, A) (1, R) (1, C) (3, D)

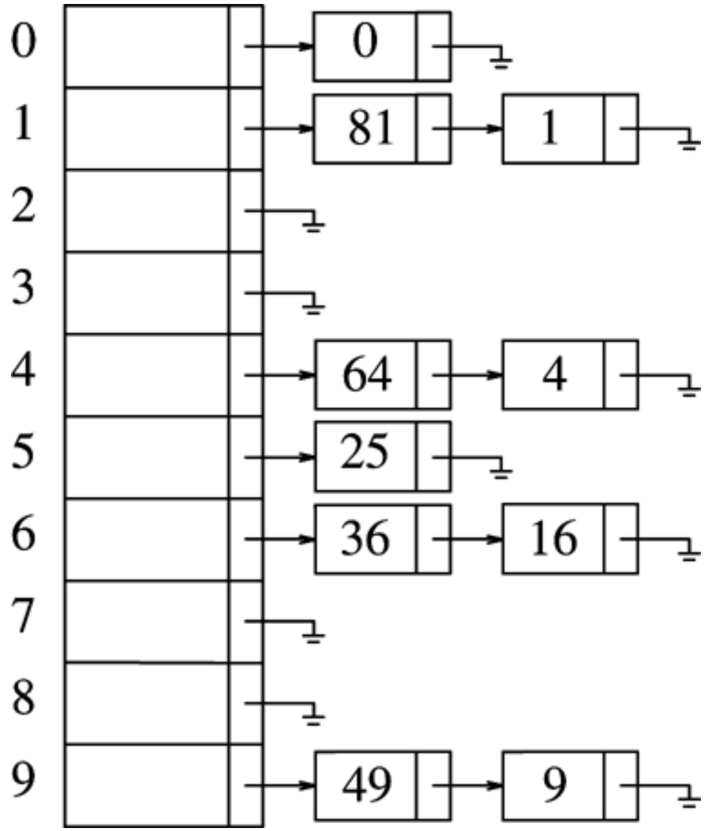


- A bucket array can be implemented as a **linked list.**
- Assume have at most a constant number of repeated keys methods find(), remove(), insert() are still O(1)

Example

- Hash table T is a vector of lists
  - Only singly linked lists are needed if memory is tight
- Key k is stored in the list at T[$h(k)$]
- E.g. TableSize = 10
  - $h(k)$ = k mod 10

Insertion sequence = 0, 1, 4, 9, 16, 25, 36, 49, 64, 81



Insertion in Hash Table with Chaining Pseudocode

```
FUNCTION insert(HashTable, key):
    index = hashFunction(key, HashTable.size) // Compute hash
index
    newNode = CREATE HashEntry
    newNode.key = key
    newNode.next = NULL

    IF HashTable.table[index] = NULL THEN
        // No linked list exists at this index; start a new one
        HashTable.table[index] = newNode
    ELSE
        // Collision: Add newNode at the beginning of the list
        newNode.next = HashTable.table[index]
        HashTable.table[index] = newNode
    END IF
END FUNCTION
```

Load Factor λ

- The **load factor (λ)** is a measure that describes how **full a hash table** is.
- It is defined as: $\lambda = N / M$ Where:
    - N: The total number of elements stored in the hash table.
    - M: The total number of slots in the hash table.
- The average length of a chain is equal to the **load factor**
    - A smaller load factor indicates fewer collisions and better performance.
    - A larger load factor increases the likelihood of collisions, leading to longer chains.
- To maintain a smaller λ, the hash table should be re-sized (**rehash**) when it becomes **too full**.
- Keep the *TableSize* **prime** to ensure a good distribution

Example

- Imagine a hash table with M=10 (array slots) and N=9 elements.
- The load factor is: λ = N / M = 9 / 10 = 0.9
- We insert the following elements into the hash table= > Keys: 0, 31, 14, 25, 46, 49, 1, 4, 16
- With a hash function: $h(k)$ = k mod M (where M=10).
- **Collisions** may occur in buckets 0 and 6. We could consider resizing the table (rehash) to reduce λ and minimize collision.

Index	Keys Stored (Chaining)
0	0
1	1, 31
2	
3	
4	4, 14
5	25
6	46, 16
7	
8	
9	49

Example (cont.)

- Imagine a hash table with M=20 (array slots) and N=9 elements.
- The load factor is: λ = N / M = 9 / 20 = 0.45
- We insert the following elements into the hash table=> Keys: 0, 31, 14, 25, 46, 49, 1, 4, 16
- With a hash function: $h(k)$ = k mod M (where M=10).

Index	Keys
0	0
1	1
2	
3	
4	4
5	25
6	46
7	
8	
9	49

Index	Keys
10	
11	31
12	
13	
14	14
15	
16	16
17	
18	
19	

Table Resized

Drawbacks of Chaining

- Each bucket requires a pointer to a linked list or another dynamic structure. This increases **memory usage**, particularly when many collisions occur.
- As the load factor (ratio of elements to buckets) increases, the linked lists grow longer, resulting in slower search, insertion, and deletion operations (O(n) in the worst case)

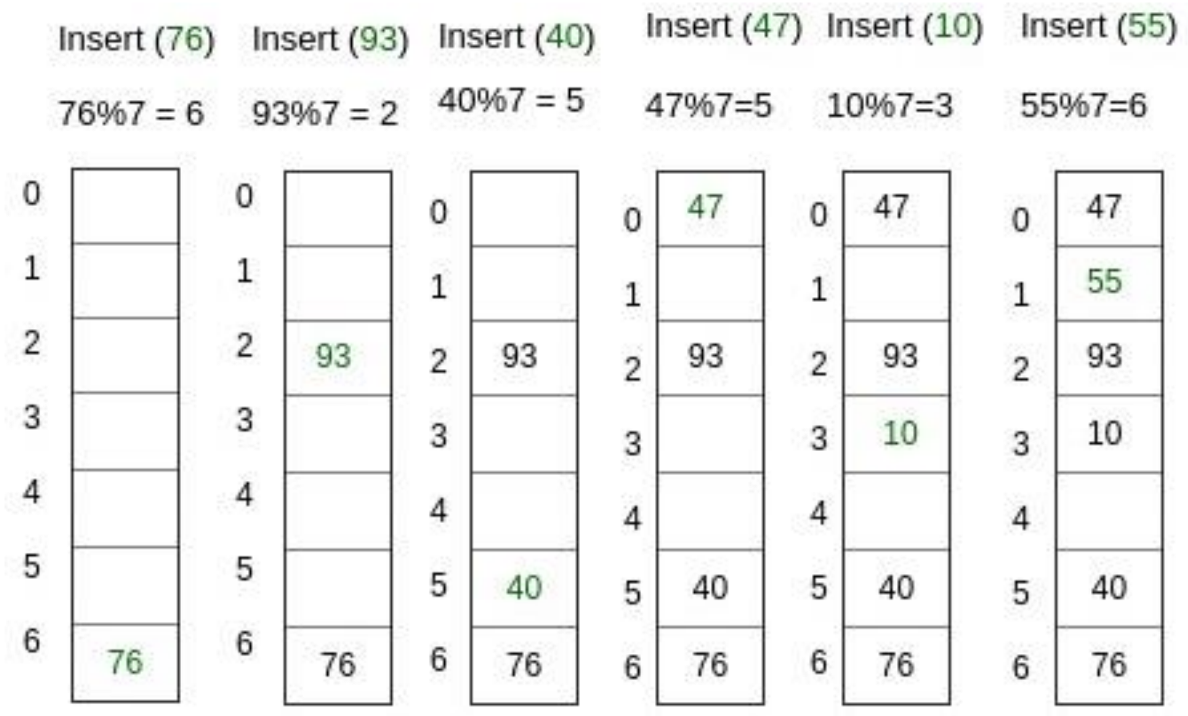
When to Use Chaining

- Works well when the hash table is relatively sparse, minimizing the number of collisions.
- Applications:
  - Database indexing (e.g., hash indexes).
  - Dictionary or symbol table implementations in programming languages.

Open Addressing

Open Addressing

- Open addressing resolves collisions by probing (searching) for the next available slot in the hash table.
- All entries are stored directly within the table, making it compact and self-contained.
- Handle collisions by placing the colliding item in the next (**circularly**) available table cell.



Linear Probing

Source: https://junminlee3.medium.com/hash-tables-animations-that-will-make-you-understand-how-they-work-d1bcc850ba71

Linear Probing

- Each table cell inspected is referred to as a *probe*.
- **Properties**
  - $\lambda \leq 1$
  - performance degrades with difficulty in finding the right spot
- **Probe sequence is**
  - h(k) mod size
  - h(k) + 1 mod size
  - h(k) + 2 mod size

- Time Complexity:
  - Best Case: O(1)
  - Worst Case: O(N). This happens when all elements have collided and we need to insert the last element by checking free space individually.

Formula: $h'(k, i) = (h(k) + i) \; mod \; N$

Advantages: Simple and efficient with a low load factor.

Insert with Linear Probing Pseudocode

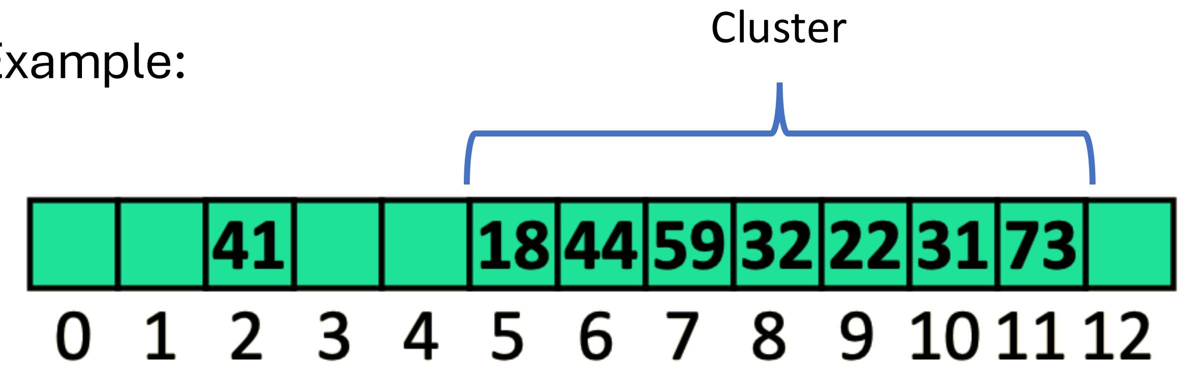
```
FUNCTION insert(HashTable, key, value):
    // Compute the initial hash index
    hashIndex = hashFunction(key, HashTable.size)
    probeCount = 0 // Initialize probe count
    WHILE probeCount < HashTable.size DO
        // Linear probing formula
        index = (hashIndex + probeCount) MOD HashTable.size
        IF HashTable[index].isEmpty THEN
            // Found an empty or deleted slot
            HashTable[index].key = key
            HashTable[index].value = value
            HashTable[index].isEmpty = FALSE
            PRINT "Inserted key", key, "at index", index
            RETURN
        END IF
        probeCount = probeCount + 1 // Move to the next slot
    END WHILE
    PRINT "Hash table is full. Cannot insert key", key
END FUNCTION
```

Find with Linear Probing Algorithm

1. Start at the index given by the hash function $h(k)$.
2. Probe consecutive locations in the hash table until one of the following conditions is met:
   1. The item with key k is found.
   2. An empty cell (NULL) is encountered, indicating the key is not present.
   3. The table has been fully traversed without finding the key.

Challenges with Linear Probing

- Entries tend to form clusters in contiguous regions of the hash table.
- Clustering increases the number of probes required for operations like find, insert, and remove.
- The more probes per operation, the slower the hash table's performance.
- Example:



What about remove?

- Solution – Replace deleted entry with special **marker** (**null**) to signal that an entry was deleted from that cell

- h(x) = x mod 13

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

- Remove(18), h(18) = 18 % 13 = 5

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

- Remove(31), h(31) = 31 % 13 = 5

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

- 31 is not found now!

Question!

- A hash table of size 7 uses linear probing for collision resolution. Initially, the table is empty. The following sequence of keys is inserted into the table: 76, 40, 48, 5, 55. The hash function is: $h(k) = k \mod 7$
- What will be the final state of the hash table? Select the correct option:

a) Keys: 48, 5, 55, Empty, Empty, 40, 76

b) Keys:55, 40, 48, 5, Empty, 76, Empty

c) Keys: 76, 40, 48, Empty, Empty, 5, 55

d) Keys: 48, 5, 76, 40, 55, Empty, Empty

00:01:59

Quadratic Probing

- Instead of probing linearly, it probes quadratically:

$$h'(k,i) = (h(k) + c_2 i^2) \mod N$$

where c_1 and c_2 are constants.

- **Advantages**: Reduces clustering compared to linear probing.
- **Disadvantages**: We may fail to find an open slot if the table size N is not a prime number.

- $f(i) = i^2$
- Probe sequence:
  - h(k) mod size
  - h(k) + 1 mod size
  - h(k) + 4 mod size
  - h(k) + 9 mod size

  ...

Quadratic Probing Example

	insert(76) $76\%7 = 6$	insert(40) $40\%7 = 5$	insert(48) $48\%7 = 6$	insert(5) $5\%7 = 5$	insert(55) $55\%7 = 6$
0			48	47	47
1					
2				5	5
3					55
4					
5		40	40	40	40
6	76	76	76	76	76
probes:	1	1	2	3	3

Insert using Quadratic Probing

```
FUNCTION insert(key, value)
    hashIndex = hashFunction(key)    // Compute initial hash index
    p = 0                            // Initialize probe count
    WHILE p < N DO                   // Limit the number of probes
        i = (hashIndex + p^2) MOD N  // Quadratic probing formula
        IF hashTable[i].isEmpty THEN
            hashTable[i].key = key       // Store the key
            hashTable[i].value = value   // Store the associated value
            hashTable[i].isEmpty = FALSE // Mark slot as occupied
            PRINT "Inserted key", key, "at index", i
            RETURN
        END IF
        p = p + 1  // Increment the probe count
    END WHILE
    PRINT "Hash table is full. Cannot insert key", key
END FUNCTION
```

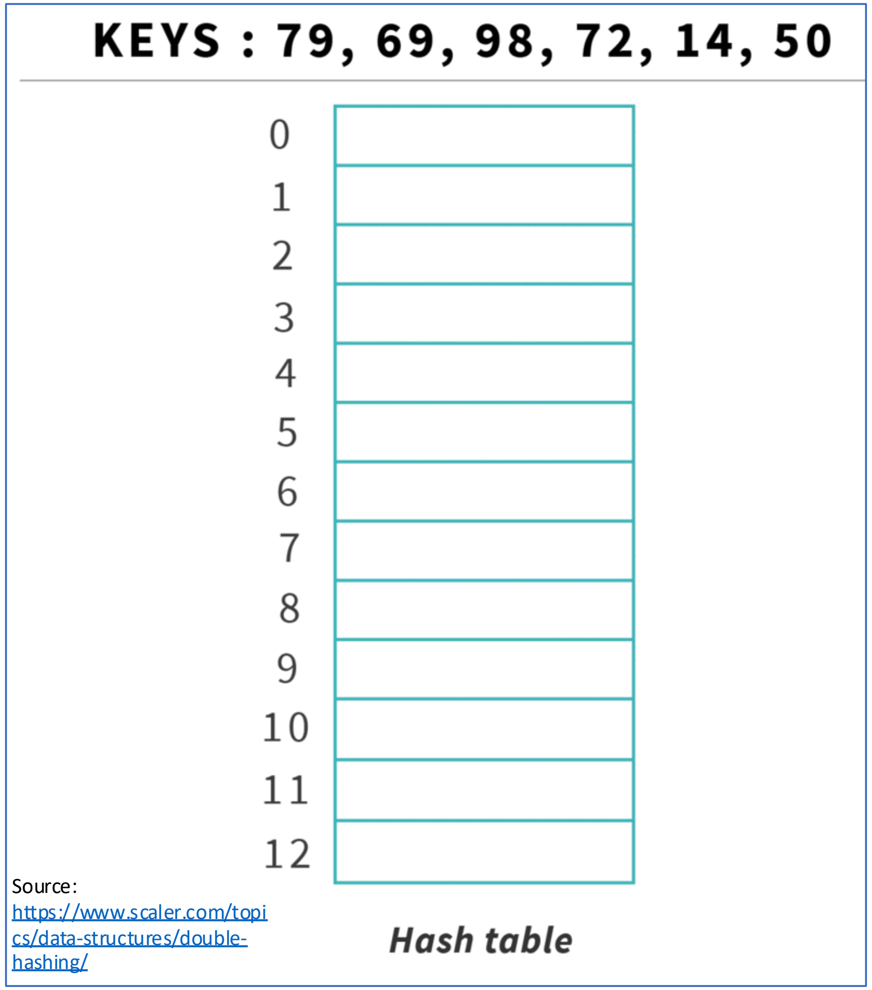
Open Addressing (Double Hashing)

- Linear Probing places an item in the first available cell in a series
- Double hashing uses the secondary hash function $\boldsymbol{h'(k)}$ and places the item in the first available cell in the series:

$$(\mathbf{h(k)} + \mathbf{p} \cdot \mathbf{h'(k)}) \bmod \mathbf{N} \qquad \text{for } \mathbf{p} = 0,\ 1, \ldots, \mathbf{N} - 1$$

  - Must have $0 < h'(k) < N$
  - N need to be **prime** to allow probing of all cells
  - linear probing is a special case of double hashing with
    - $h'(k) = 1$ for all k
- Double hashing spreads entries more evenly through hash array

Open Addressing (Double Hashing)



Source: https://www.scaler.com/topics/data-structures/double-hashing/

Insert the keys **79, 69, 98, 72, 14, 50** into the Hash Table of size N = 13

- h(k) = k mod 13
- h'(k) = 1 + (k mod 11)

79 mod 13 = 1
69 mod 13 = 4
98 mod 13 = 7
72 mod 13 = 7

h_{new} = [h(72)+p* h'(72)] % 13
= [7 + 1 * (1 + 72 % 11)] % 13
= 1

h_{new} = [h(72)+ p* h'(72)] % 13
= [7 + 2 * (1 + 72 % 11)] % 13
= 8

Key	Index
79	1
69	4
98	7
71	8
14	5
50	11

$$Load\ Factor\ \propto = \frac{6}{13} < 0.50$$

Deletion in Open Addressing

- Handling Deletion
  1. Add an **isDeleted** flag to mark slots as deleted, distinguishing them from truly empty slots.
  2. Treat deleted slots as **occupied** during a search to avoid breaking the probing sequence.
  3. Allow insertion into deleted slots if no other empty slot is available.
  4. Periodically **rehash** the table to clean up deleted slots and optimize performance.
- This ensures consistency for linear probing, quadratic probing, and double hashing.

Open Addressing Performance

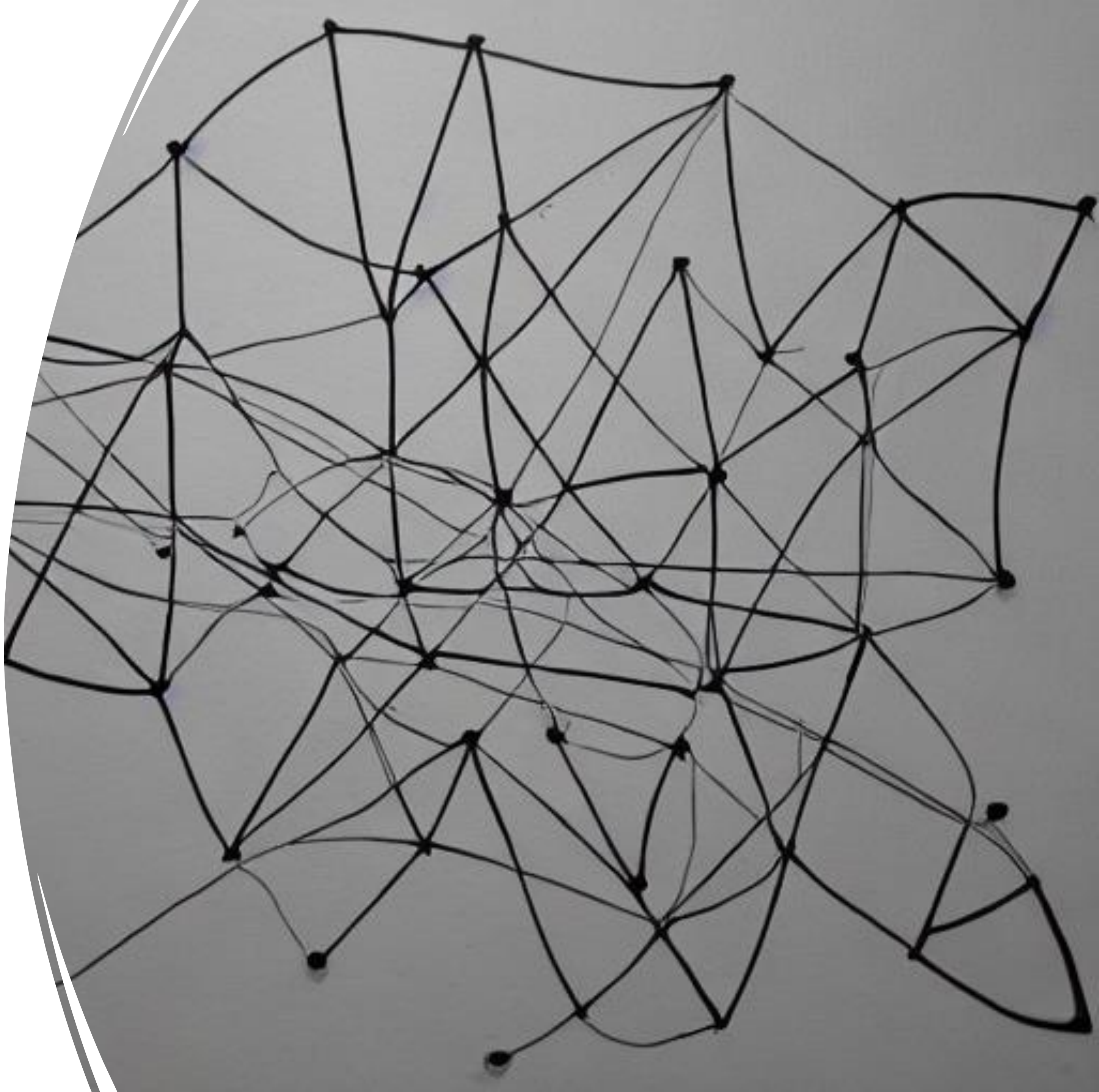
- Worst case: find(), insert() and remove() are O(n)
  - the worst case occurs when all inserted keys collide
- Load factor ∝ = n/N affects performance,

  N = $size\ of\ the\ array$

Chaining vs. Open Addressing

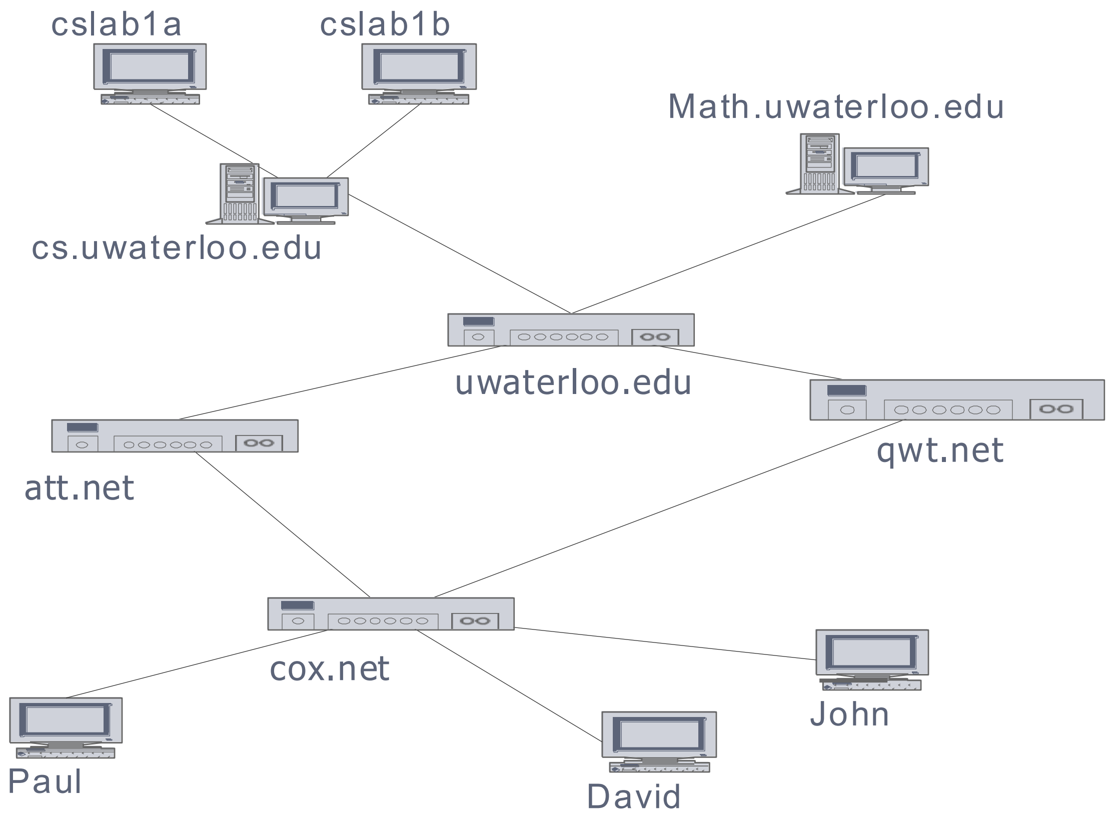
- Open addressing saves **space** over chaining.
- Chaining is usually **faster** (depending on the load **factor** of the bucket array) than the open addressing.
- Thus, if memory space is not a major issue, use chaining; otherwise, use open addressing.

Graphs



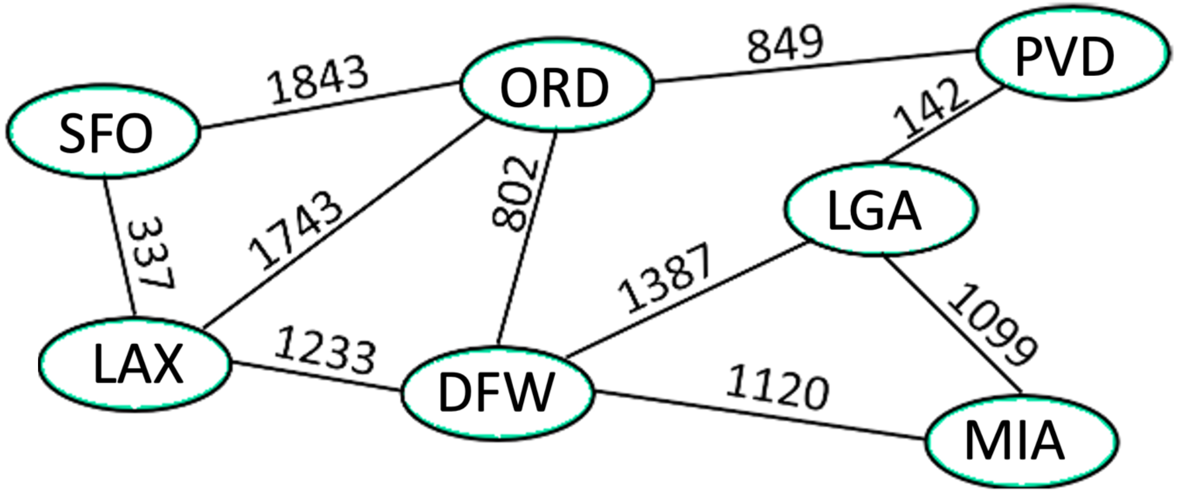
Motivation

- A graph is a natural representation for a special type of data:
- Computer networks
  - Local area network
  - Internet
  - Web



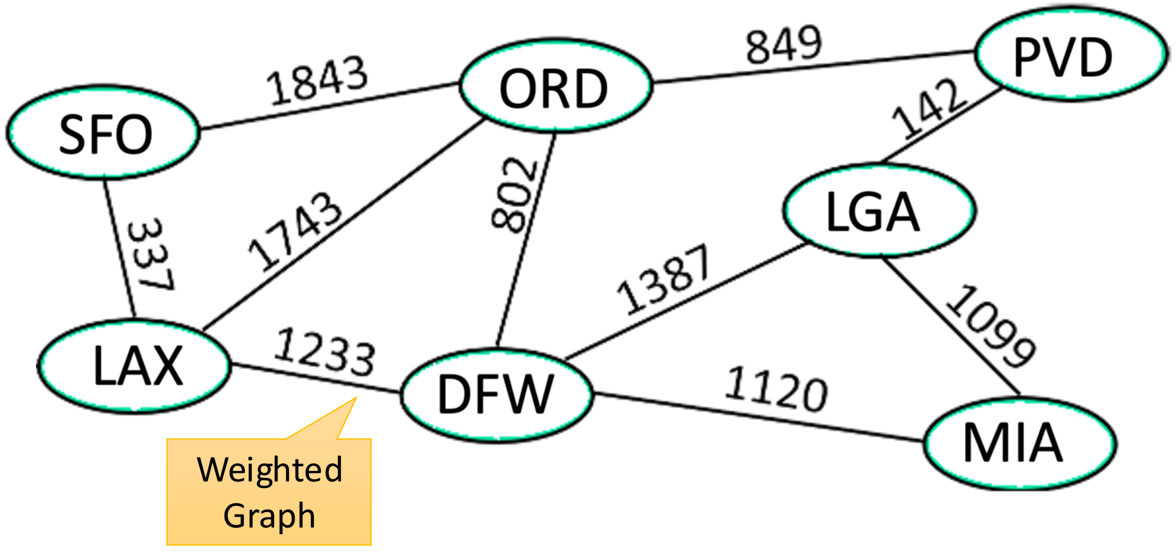
Motivation

- A graph is a natural representation for a special type of data:
- City map (Transportation Networks):
  - Each city is represented by a node
  - Can label each node with a three-letter airport code
  - Two cities with a direct flight between them are connected by an edge
  - You can label the edge with the mileage of the route, time to fly, etc.



Motivation

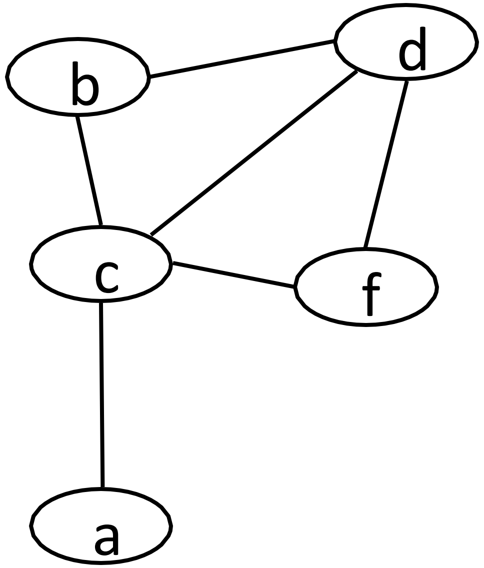
- You can answer many interesting questions using graphs
  - Can we reach one city from another city?
  - What is the route with a minimum number of connections between 2 cities?
  - What is the minimum mileage route between 2 cities?
- Many interesting questions can be answered efficiently using graphs.



Graphs: Formal Definition

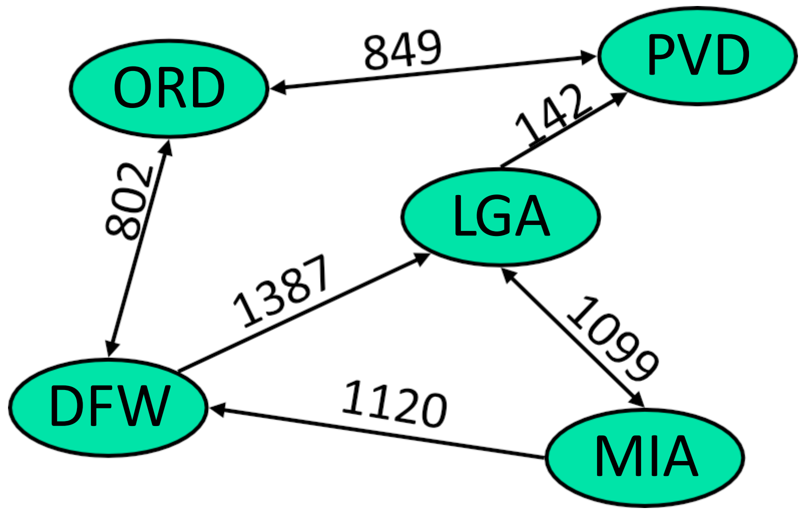
- A graph is a pair (***V, E***), where
  - V is a collection of **nodes** or **vertices**
  - E is a collection of pairs of vertices called **edges**
- In this example
  - ***V*** ={a,b,c,d,f}
  - ***E*** ={(a,c),(b,c),(c,f),(b,d),(d,f), (c,d)}

Pairs

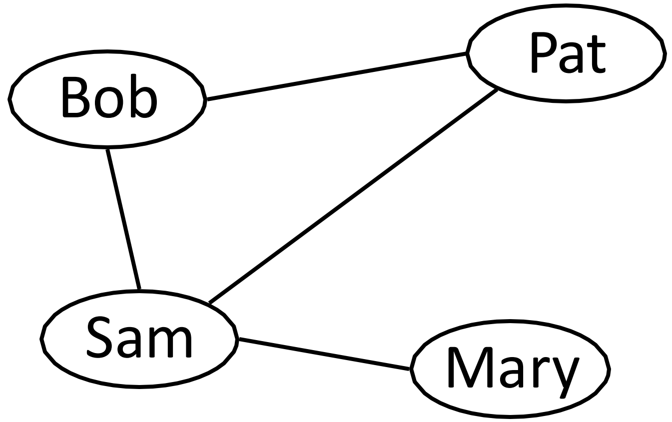


Graph Types

- Directed graph (Digraph)
  - All the edges are directed
  - e.g., flight route network (map)

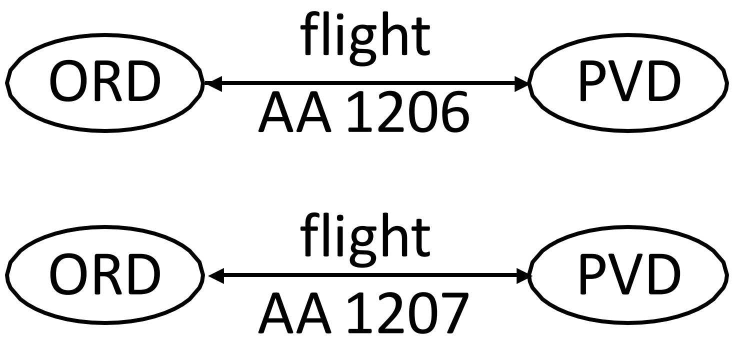


- Undirected graph
  - All edges are undirected
  - e.g.,"friends" network



Directed edge (cont.)

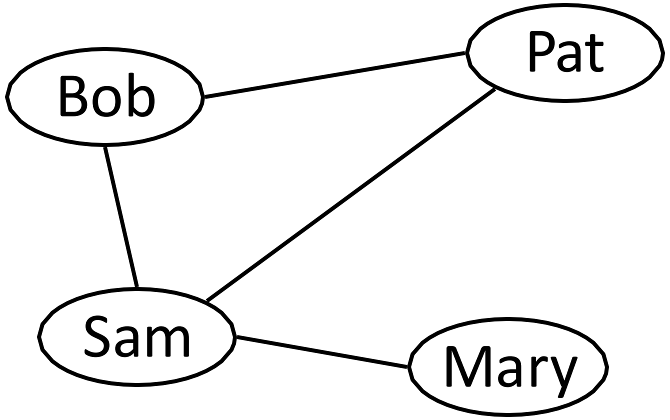
- Ordered pair of vertices (***u***,***v***)
  - First vertex ***u*** is the **origin**
  - Second vertex ***v*** is the **destination**
  - e.g., a flight
- (***v***,***u***) and (***u***,***v***) are two **different edges**



flight route network

Undirected Graph (cont.)

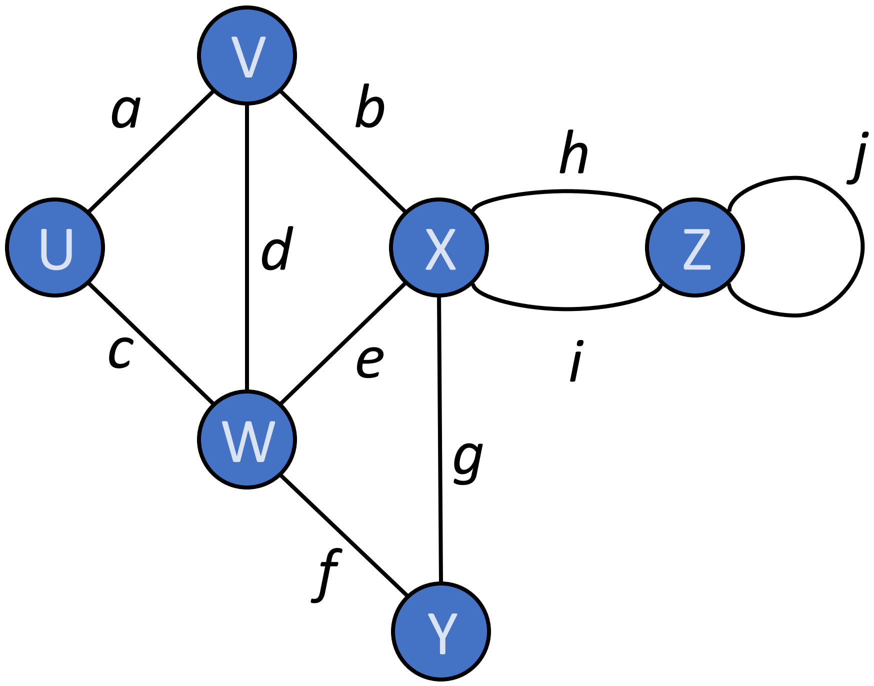
- Unordered pair of vertices (***u***,***v***)
  - e.g., a network of friends
- If ***Sam*** is a friend of ***Bob***, then ***Bob*** is also a friend of ***Sam***
- (***u***,***v***) and is (***v***,***u***) the **same edge**



A “friends” network

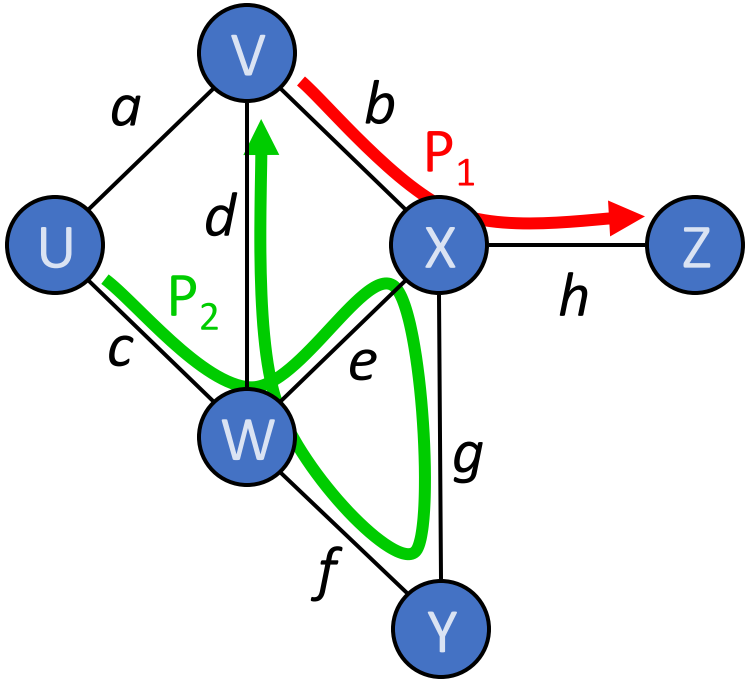
Graph Terminology

- **Endpoints** (or end vertices) of an edge
  - U and V are the *endpoints* of *a*
- **Edges** incident on a vertex
  - *a*, *d*, and *b* are *incident* on V
- **Adjacent** vertices
  - U and V are *adjacent*
- **Degree** of a vertex
  - X has *degree* 5
- **Parallel** (multiple) edges
  - *h* and *i* are *parallel edges*
- **Self-loop**
  - j is *a self-loop*



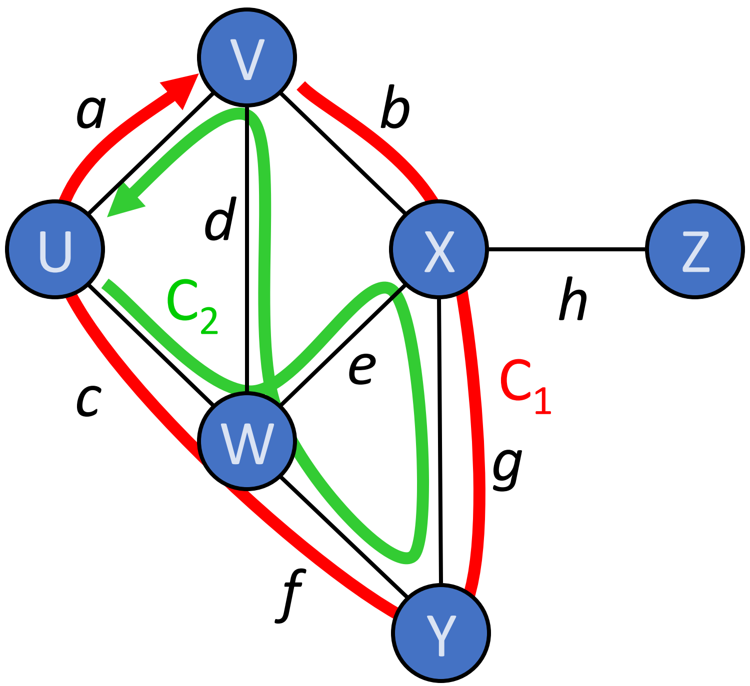
Graph Terminology (cont.)

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
- Simple path
  - A path such that all its vertices and edges are **distinct**
- Examples
  - **P_1=(V,b,X,h,Z)** is a simple path
  - **P_2=(U,c,W,e,X,g,Y,f,W,d,V)** is a path that is not simple



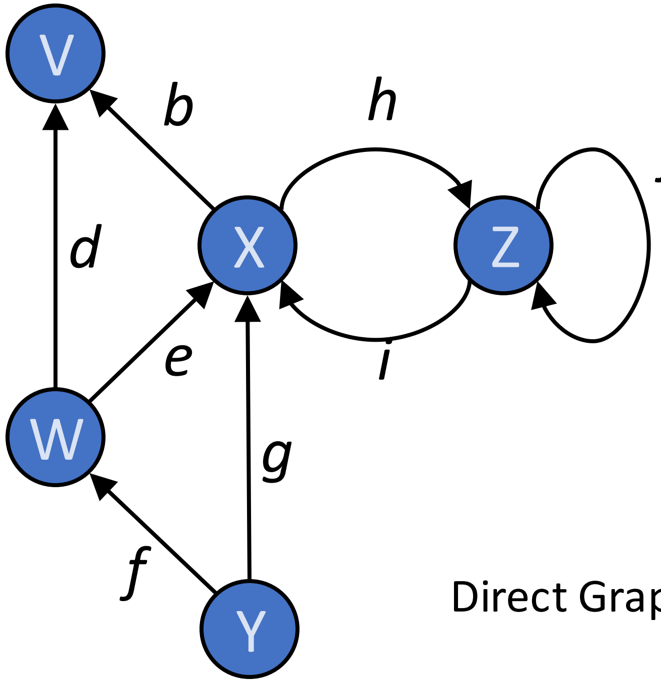
Graph Terminology (cont.)

- Cycle
  - circular sequence of alternating vertices and edges
- Simple cycle
  - cycle such that all its vertices and edges are **distinct**
- Examples
  - **C_1=(V,b,X,g,Y,f,W,c,U,a,V)** is a simple cycle
  - **C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,U)** is a cycle that is not simple



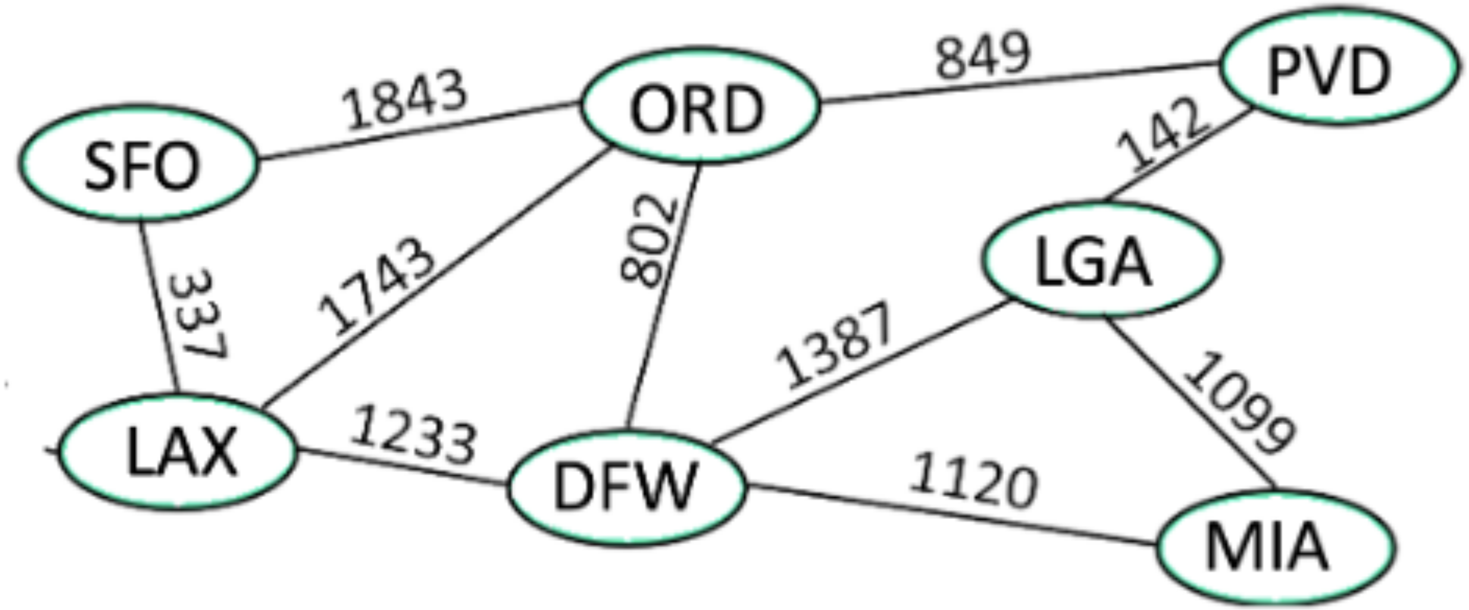
Graph Terminology (cont.)

- Outgoing edges of a vertex
  - *h* and *b* are the *outgoing edges* of X
- Incoming edges of a vertex
  - *e*, *g*, and *i* are *incoming edges* of X
- In-degree of a vertex
  - X has *in-degree* 3
- Out-degree of a vertex
  - X has *out-degree* 2



Direct Graph

Graph Properties




Thank you