

# CSC 172– Data Structures and Algorithms

Lecture #25

Spring 2018

Please put away all electronic devices

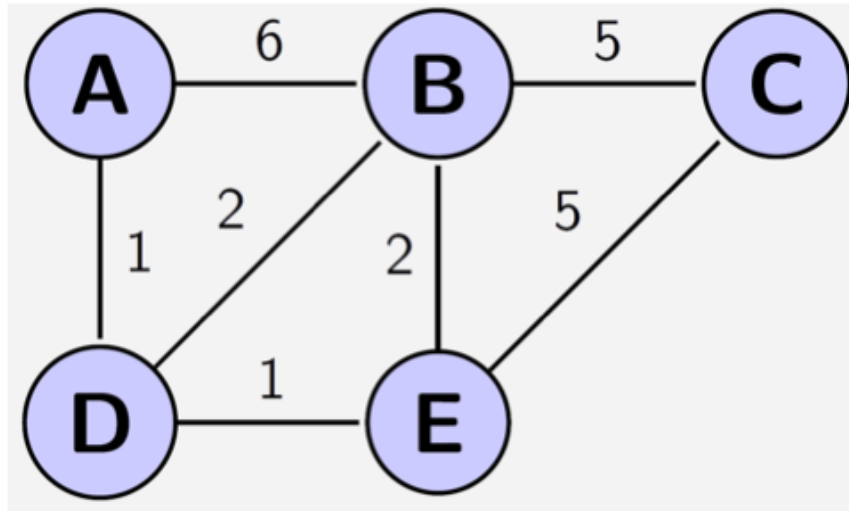


# Announcements

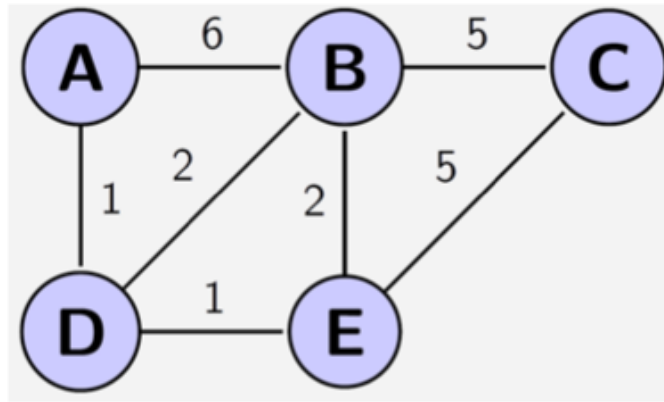
- This Thursday we will have a pre-Final Exam.
  - Will cover time consuming graph algorithms
    - Whatever we have covered last week
    - Worth 20 pts of the final Exam
    - Other 80 pts on Next Tuesday
    - Time: 20 minutes
    - Purpose: To give you extra time for other questions on the Final
    - This questions won't be repeated on the exam!

# Dijkstra's vs Prim's

Let us discuss how Dijkstra's and Prim's algorithms differ



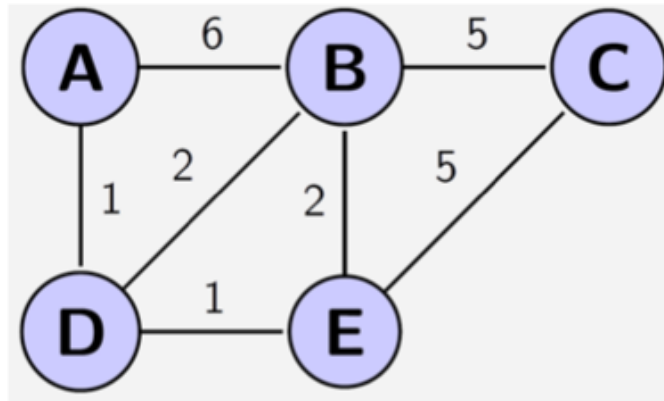
# Dijkstra's vs Prim's



	A	B	C	D	E
A	0,A	inf	inf	inf	inf

	A	B	C	D	E
A	0,A	inf	inf	inf	inf

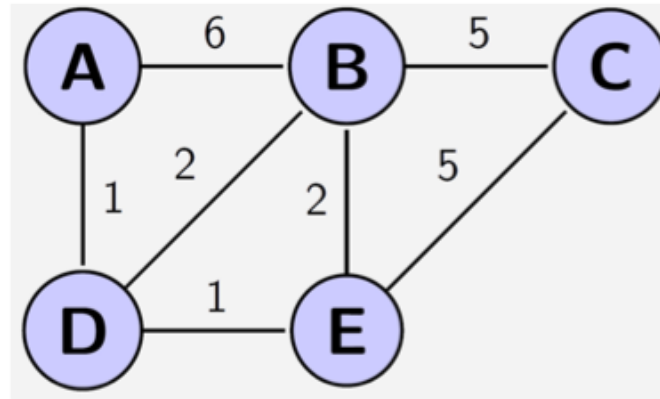
# Dijkstra's vs Prim's



	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf

	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf

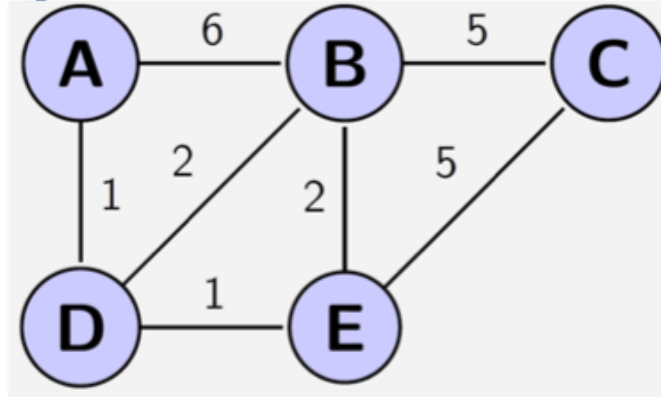
# Dijkstra's vs Prim's



	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf
D,A		3,D	inf		2,D

	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf
D,A		2,D	inf		1,D

# Dijkstra's vs Prim's

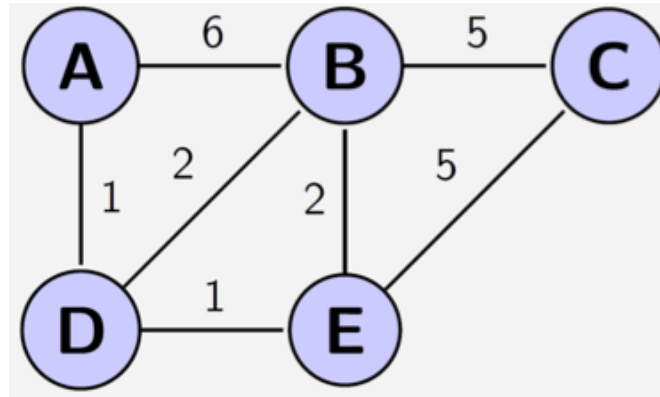


	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf
D,A		3,D	inf		2,D
E,D		3,D	7,E		

	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf
D,A		2,D	inf		1,D
E,D		2,D	5,E		



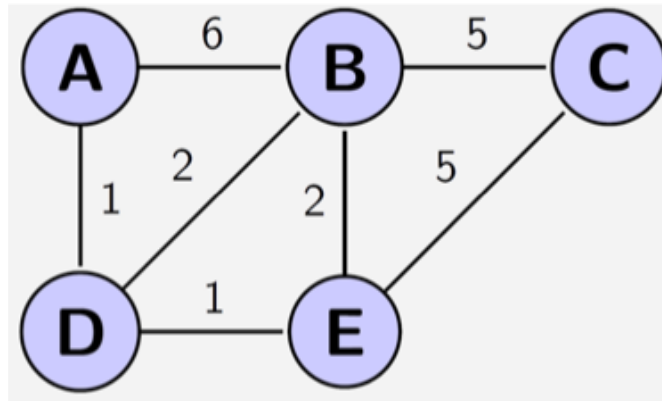
# Dijkstra's vs Prim's



	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf
D,A		3,D	inf		2,D
E,D		3,D	7,E		
B,D			7,E		

	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf
D,A		2,D	inf		1,D
E,D		2,D	5,E		
B,D			5,E		

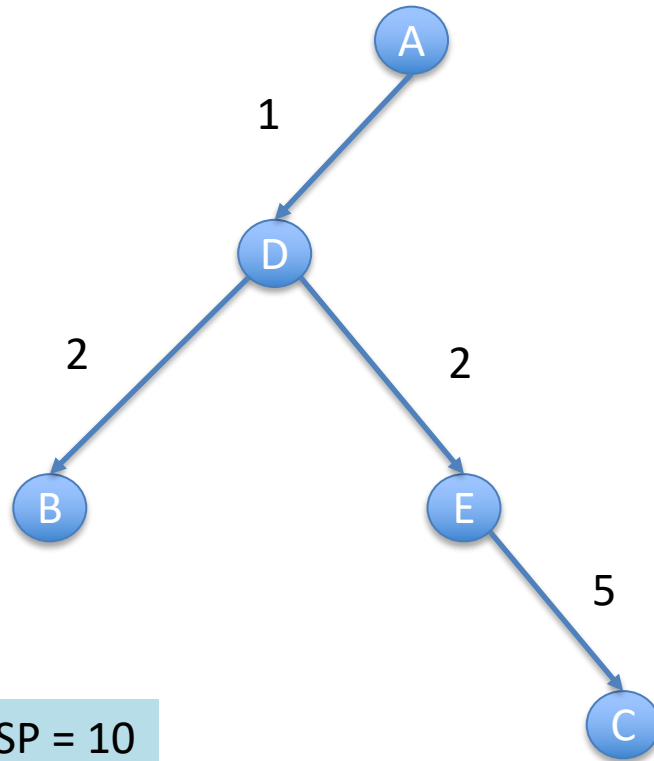
# Dijkstra's vs Prim's



	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf
D,A		3,D	inf		2,D
E,D		3,D	7,E		
B,D			7,E		
C,E					

	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf
D,A		2,D	inf		1,D
E,D		2,D	5,E		
B,D			5,E		
C,E					

# Final Outcome: Prim's and Dijkstra's



They are the same!

But, it's just a coincidence. Most of the time, it will not be the case

Cost of MSP = 10

Shortest Path from A to C ?

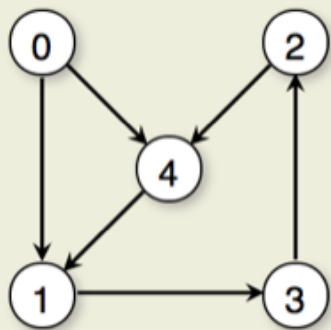
Shortest Path from B to C ?

# Dijkstra's vs Prim's

- $A, B = 5$
- $B, C = 5$
- $A, C = 8$

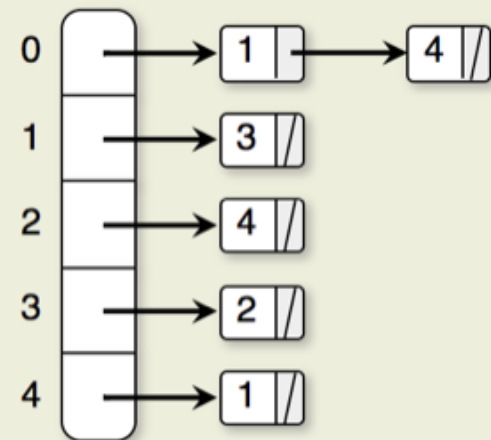
An Example where Dijkstra's (A to B and C) and Prim's produce different output.

# **ANALYSIS OF DIJKSTRA'S ALGORITHM**



	0	1	2	3	4
0		1			1
1				1	
2					1
3			1		
4		1			

Adajceny Matrix



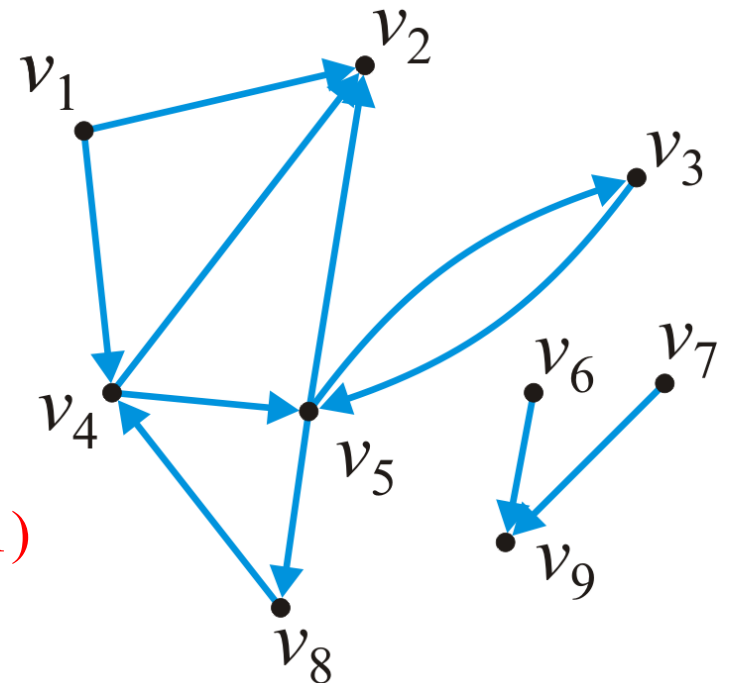
Adajceny List

# Adjacency matrix

The matrix entry  $(j, k)$  is set to true if there is an edge  $(v_j, v_k)$

	1	2	3	4	5	6	7	8	9
1		T		T					
2									
3					T				
4		T			T				
5		T	T					T	
6									T
7									T
8				T					
9									

- Requires  $\Theta(|V|^2)$  memory
- Determining if  $v_j$  is adjacent to  $v_k$  is  $O(1)$
- Finding all neighbors of  $v_j$  is  $\Theta(|V|)$

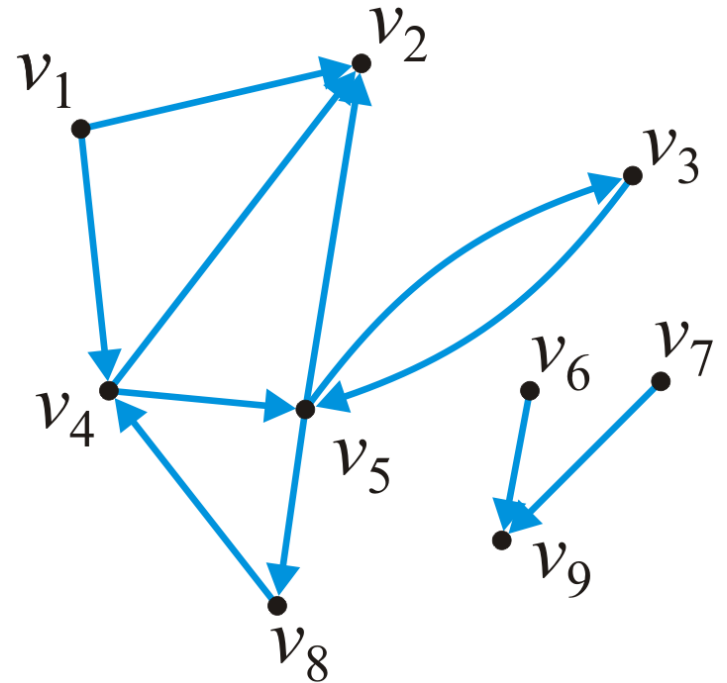


# Adjacency list

Most efficient for algorithms is an adjacency list

- Each vertex is associated with a list of its neighbors

```
1  • → 2 → 4
2  •
3  • → 5
4  • → 2 → 5
5  • → 2 → 3 → 8
6  • → 9
7  • → 9
8  • → 4
9  •
```



- Requires  $\Theta(|V| + |E|)$  memory
- On average:
  - Determining if  $v_j$  is adjacent to  $v_k$  is  $O\left(\frac{|E|}{|V|}\right)$
  - Finding all neighbors of  $v_j$  is  $\Theta\left(\frac{|E|}{|V|}\right)$



– On average:

- Determining if  $v_j$  is adjacent to  $v_k$  is  $O\left(\frac{|E|}{|V|}\right)$

- checking this  $|V|$  number of times:  $O(|E|)$ ;

# Implementation and analysis

The initialization requires  $\Theta(|V|)$  memory and run time

We iterate  $|V| - 1$  times, each time finding next closest vertex to the source

- Iterating through the table requires is  $\Theta(|V|)$  time
- Each time we find a vertex, we must check all of its neighbors
- With an **adjacency matrix**, the run time is  $\Theta(|V|(|V| + |V|)) = \Theta(|V|^2)$
- With an **adjacency list**, the run time is  $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$  as  $|E| = O(|V|^2)$

Can we do better?

- Recall, we only need the closest vertex
- How about a **priority queue**?

# Pseudocode for Dijkstra's Algorithm (Using PQ)

```
void dijkstra(s) {
    queue = new PriorityQueue<Vertex>();
    for (each vertex v) {
        v.dist = infinity; // can use Integer.MAX_VALUE
        queue.enqueue(v);
        v.pred = null;
    }
    s.dist = 0;

    while (!queue.isEmpty()) {
        u = queue.extractMin();
        for (each vertex v adjacent to u)
            relax(u, v);
    }
}
```

```
void relax(u, v) {
    if (u.dist + w(u,v) < v.dist) {
        v.dist = u.dist + w(u,v);
        v.pred = u;
    }
}
```

# Recall (Adjacency List)

– On average:

- Determining if  $v_j$  is adjacent to  $v_k$  is  $O\left(\frac{|E|}{|V|}\right)$

- checking this  $|V|$  number of times:  $O(|E|)$ ;

# Let's find the runtime

```
while (!queue.isEmpty()) {  
    u = queue.extractMin();  
    for (each vertex v adjacent to u)  
        relax(u, v);  
}
```

```
while (!queue.isEmpty()) { //  $O(|V|)$   
    u = queue.extractMin(); //  $O(\log |V|)$ 
```

```
while (!queue.isEmpty()) {  
    for (each vertex v adjacent to u) {
```

How many times?:

We are traversing the whole adjacency list.

So,  $O(|E|)$

```
relax(u, v); Runtime:  $O(\log |V|)$ 
```

So: total runtime =  $O(|V|\log |V| + |E|\log |V|) = O(|E|\log |V|)$

# Implementation and analysis

The initialization still requires  $\Theta(|V|)$  memory and run time

- The priority queue will also require  $O(|V|)$  memory
- We must use an adjacency list, not an adjacency matrix

We iterate  $|V| - 1$  times, each time finding the *closest* vertex to the source

- Place the distances into a priority queue
- The size of the priority queue is  $O(|V|)$
- Thus, the work required for this is  $O(|V| \log(|V|))$

Is this all the work that is necessary?

- Recall that each edge visited may result in updating the priority queue
- Thus, the work required for this is  $O(|E| \log(|V|))$

Thus, the total run time is  $O(|V| \ln(|V|) + |E| \log(|V|)) = O(|E| \log(|V|))$

# Once Again: Dijkstra's to Prim's

```
void dijkstra(s) {
    queue = new PriorityQueue<Vertex>();
    for (each vertex v) {
        v.dist = infinity; // can use Integer.MAX_VALUE
        queue.enqueue(v);
        v.pred = null;
    }
    s.dist = 0;

    while (!queue.isEmpty()) {
        u = queue.extractMin();
        for (each vertex v adjacent to u)
            relax(u, v);
    }
}
```

```
void relax(u, v) {
    if (u.dist + w(u,v) < v.dist) {
        v.dist = u.dist + w(u,v);
        v.pred = u;
    }
}
```

Source: <http://www.cs.dartmouth.edu/~thc/cs10/lectures/0509/0509.html>

# Now Prim's

```
void prims(s) {
    queue = new PriorityQueue<Vertex>();
    for (each vertex v) {
        v.dist = infinity; // can use Integer.MAX_VALUE
        queue.enqueue(v);
        v.pred = null;
    }
    s.dist = 0;

    while (!queue.isEmpty()) {
        u = queue.extractMin();
        for (each vertex v adjacent to u)
            relax(u, v);
    }
}
```

```
void relax(u, v) {
    if (w(u,v) < v.dist) {
        v.dist = w(u,v);
        v.pred = u;
    }
}
```



# Summary

Dijkstra's algorithm and Prim's:

- It solves the problem in  $O(|E| \log(|V|))$  time using Priority Queue
- The algorithm can be further improved to run in time  $O(|V| \log(|V|) + |E|)$  --- (using Fibonacci Heap; We won't cover it)



**HASHMAP**

# Motivations

- Balanced search trees
  - Store (key, value)-pairs
  - $O(\log n)$ -time search, insert, delete, max, min
  - Relatively complex implementations
- Can we improve running times for basic operations?

# Example

- Given a set of (key, value)-pairs
  - Keys are in the set  $\{0, 1, 2, 3, \dots, n-1\}$
  - Values could be anything
- Store them in an array (*direct access table*)

0	1	2	3	...	...	...	...	n-2	n-1
$v_0$	NULL	$v_2$	NULL	...	...	...	...	$v_{n-2}$	$v_{n-1}$

- Search/insert/delete takes  $O(1)$ -time

I wish I could get such a good runtime



# What are the drawbacks?

- Keys are not always integers.
- Unlikely see such perfect key set in practice
  - (Key, Value) = (English Word, Meaning)
  - (Key, Value) = (URL, IP address)
- Thus there'll be lots of NULL entries
  - Wastes lots of space because  $n \gg \# \text{ pairs}$
  - Say keys are 8-byte integers,  $n = 2^{256} - 1$

# **HASH MAP**

# Map (for storing key-value pair)

- **Wanted:** data structure for an online dictionary
- **With what we know so far, what are the choices?**
  - Randomize the keys and insert one by one in a **normal BST**
  - Use **Balanced Search Tree**
  - Sort the entries on the keys, use **binary search**
  - Use **TreeMap** (which uses Red-Black balanced binary tree as the container)
- **Runtime**
  - Search still takes  $O(\log n)$ -time
  - Can we do better?

Yes,  $O(1)$  ....





Applying Hash Functions

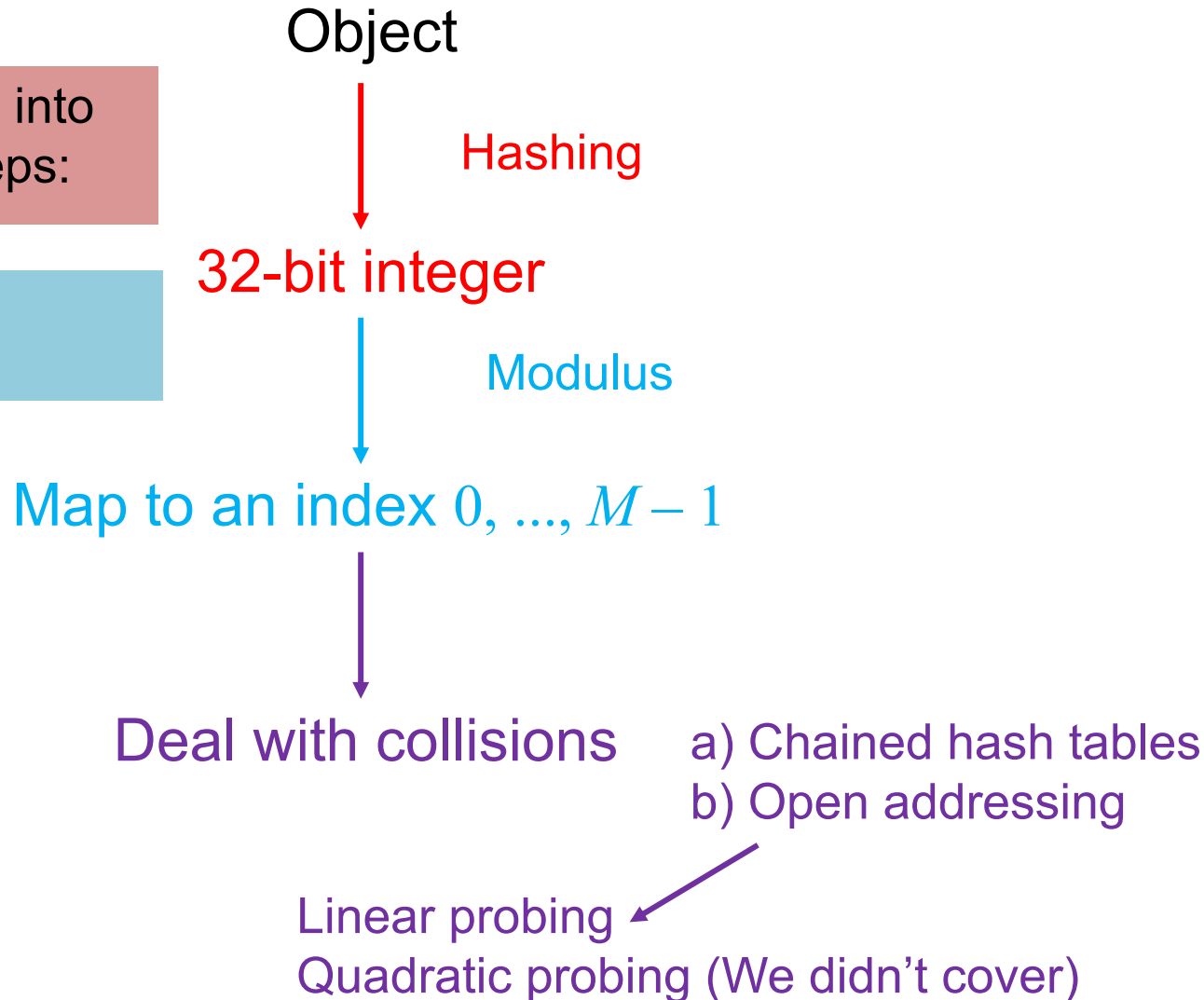
**HASHING**

# The hash process

9.1.4

We will break the process into three **independent** steps:

We will try to get each of these down to  $O(1)$



# Hash of an object

- What we want:
  - Hash an object into a 32-bit integer
  - **How to achieve this:**
    - Write your own hashCode
    - Java provides hashCode for free

# Java hashCode()

```
public static void main(String[] args) {  
    String str1 = "Hello World";  
    String str2 = "Hello CSC 172";  
  
    System.out.println(str1.hashCode());  
    System.out.println(str2.hashCode());  
}
```

-862545276  
-1394859631

# Java String hashCode

```
public int hashCode() {  
    int h = hash;  
    if (h == 0 && value.length > 0) {  
        char val[] = value;  
  
        for (int i = 0; i < value.length; i++) {  
            h = 31 * h + val[i];  
        }  
        hash = h;  
    }  
    return h;  
}
```

Returns a hash code for this string.  
The hash code for a String object is  
computed as

$$s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \dots + s[n-1]$$

# Properties

Necessary properties of such a hash function  $h$  are:

- Should be fast: ideally  **$O(1)$**
- The hash value must be *deterministic*
  - It must always return the same 32-bit integer each time
- **Equal objects hash to equal values**
  - $x = y \Rightarrow h(x) = h(y)$
- If two objects are randomly chosen, there should be only a **one-in- $2^{32}$**  chance that they have the same hash value
  - We call it **collision**

We would love:

Two different objects hash to two different values

But not possible. So, aim for the best

# Collision is simply unavoidable

- Pigeonhole principle
  - $K+1$  pigeons,  $K$  holes  $\rightarrow$  at least one hole with  $\geq 2$  pigeons
- There are many more objects in the universe than  $2^{32}$ 
  - Object set = set of strings of length  $\leq 30$  characters
  - Object set = set of possible URLs
  - Object set = set of possible file names in a CD-ROM

Done with First level of Hashing. Next?

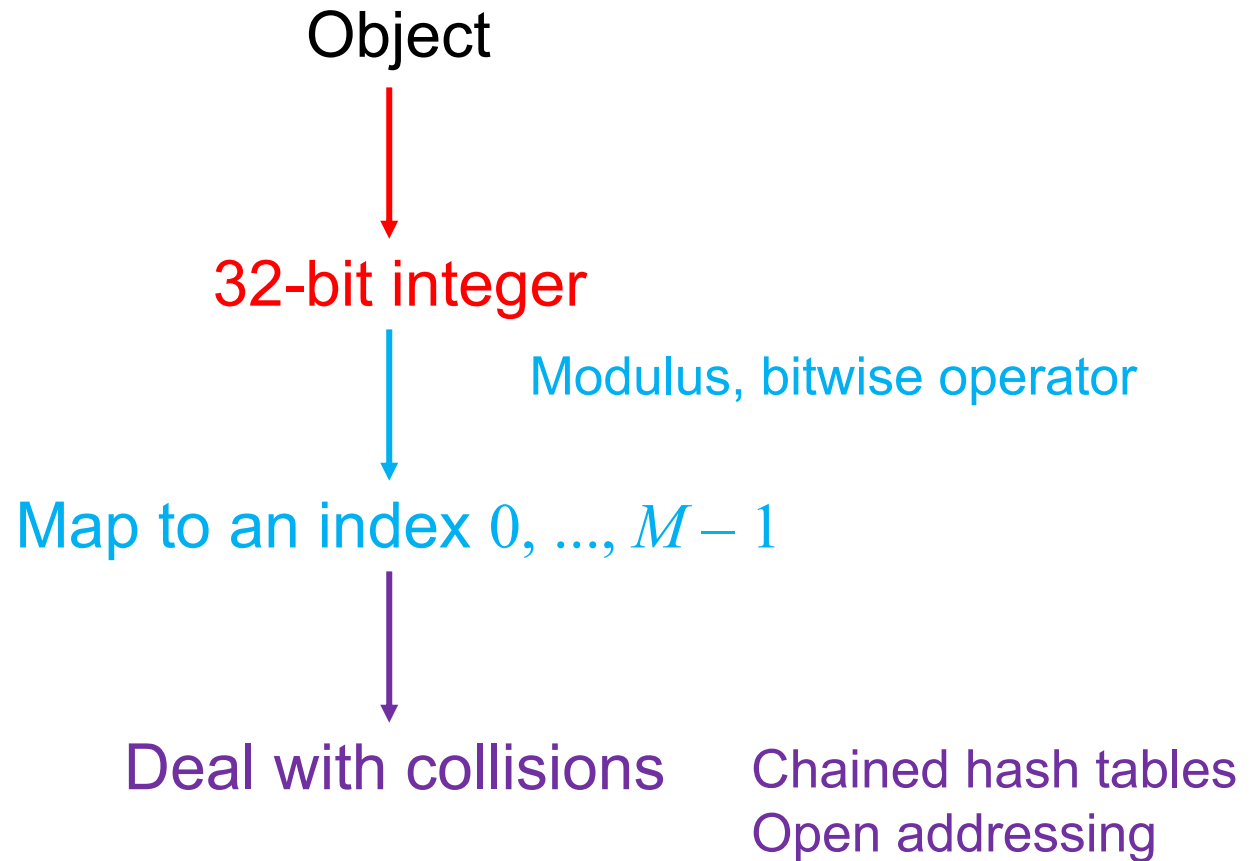
Another level of Hashing (or Mapping)

Much faster

**NEXT STEP**



# The hash process



# Properties

Necessary properties of this mapping function  $h_M$  are:

- a. Must be fast:  $O(1)$
- b. The hash value must be *deterministic*
  - Given  $n$  and  $M$ ,  $h_M(n)$  must always return the same value
- c. If two objects are randomly chosen, there should be only a **one-in- $M$**  chance that they have the same value from 0 to  $M - 1$

# Modulus operator

Easiest method: return the value modulus  $M$

```
int hash(int n, int M ) {  
    return n % M;  
}
```

Unfortunately, calculating the modulus (or remainder) is expensive

- We can simplify this if  $M = 2^m$
- We can use logic operations

# How Java Does it in HashMap?

- 

Length = Length of the array = m

```
int indexFor(int h, int length)
{
    return h & (length-1);
}
```



Not required for exam.  
You need to be familiar with 'bitwise AND'  
operator to appreciate this.

# Java HashMap get Method

```
public V  get(Object key) {  
    if (key == null)  
        return getForNullKey();  
    int hash = hash(key.hashCode());  
    for (Entry<K,V> e = table[indexFor(hash, table.length)];  
        e != null;  
        e = e.next) {  
        Object k;  
        if (e.hash == hash && ((k = e.key) == key || key.equals(k)))  
            return e.value;  
    }  
    return null;  
}
```

Not required for exam. But, go through it to appreciate how Java has implemented hashmap (uses chaining (covered later))

# Java HashMap put Method

```
public V put(K key, V value) {  
    if (key == null)  
        return putForNullKey(value);  
    int hash = hash(key.hashCode());  
    int i = indexFor(hash, table.length);  
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {  
        Object k;  
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {  
            V oldValue = e.value;  
            e.value = value;  
            e.recordAccess(this);  
            return oldValue;  
        }  
    }  
    modCount++;  
    addEntry(hash, key, value, i);  
    return null;  
}
```

Not required for exam. But, go through it to appreciate how Java has implemented hashmap

Source: <http://greppcode.com/file/repository.greppcode.com/java/root/jdk/openjdk/6-b14/java/util/HashMap.java#HashMap.get%28java.lang.Object%29>

# HashMap addEntry

```
void ↴ addEntry(int hash, K key, V value, int bucketIndex) {  
    Entry<K,V> e = table[bucketIndex];  
    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);  
    if (size++ >= threshold)  
        resize(2 * table.length);  
}
```

Not required for exam. But, go through it to appreciate how Java has implemented hashmap

Chaining

Open addressing

**NEXT STEP:  
COLLISION RESOLUTION**



# Background

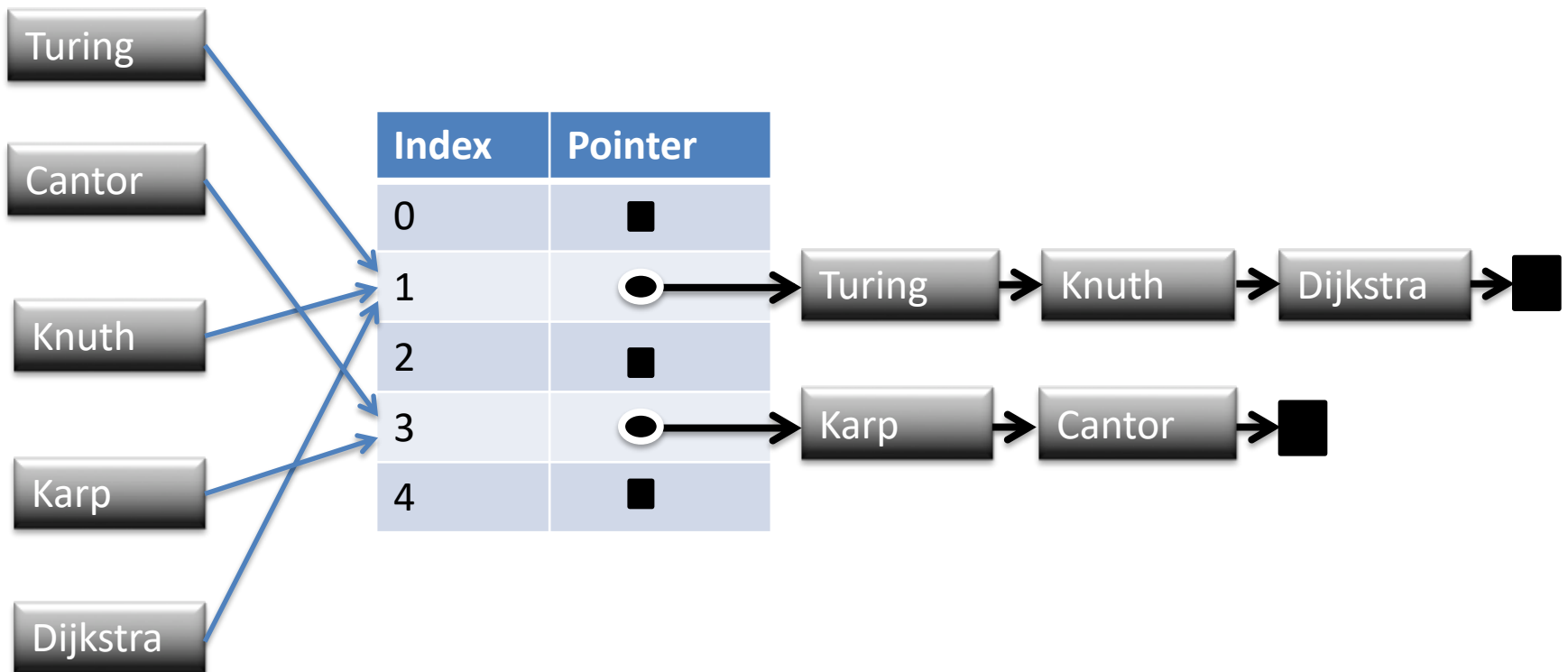
First, a review:

- We want to store objects in an array of size  $M$
- We want to quickly calculate the bin where we want to store the object
  - We came up with hash functions—hopefully  $\Theta(1)$
  - Perfect hash functions (no collisions) are difficult to design
- We will look at some schemes for dealing with collisions

# Chaining and Open Addressing

- Chaining:
  - Each index is associated with a linked list:
- Open Addressing:
  - Simply move down the table from the 'home' index until a free slot is found
    - How far at a time?
      - Depends on the algorithm

# Chaining



# Performance

- Under *simple uniform hashing assumption*
  - i.e. Each object hashed into a bucket with probability  $1/m$ , uniformly and independent from other objects
- Expected search time  $\Theta(1)$
- Worst-case search time  $\Omega(n)$  – though very unlikely
- Using universal hashing, expected time for  $n$  operations is  $\Omega(n)$

# Open Addressing

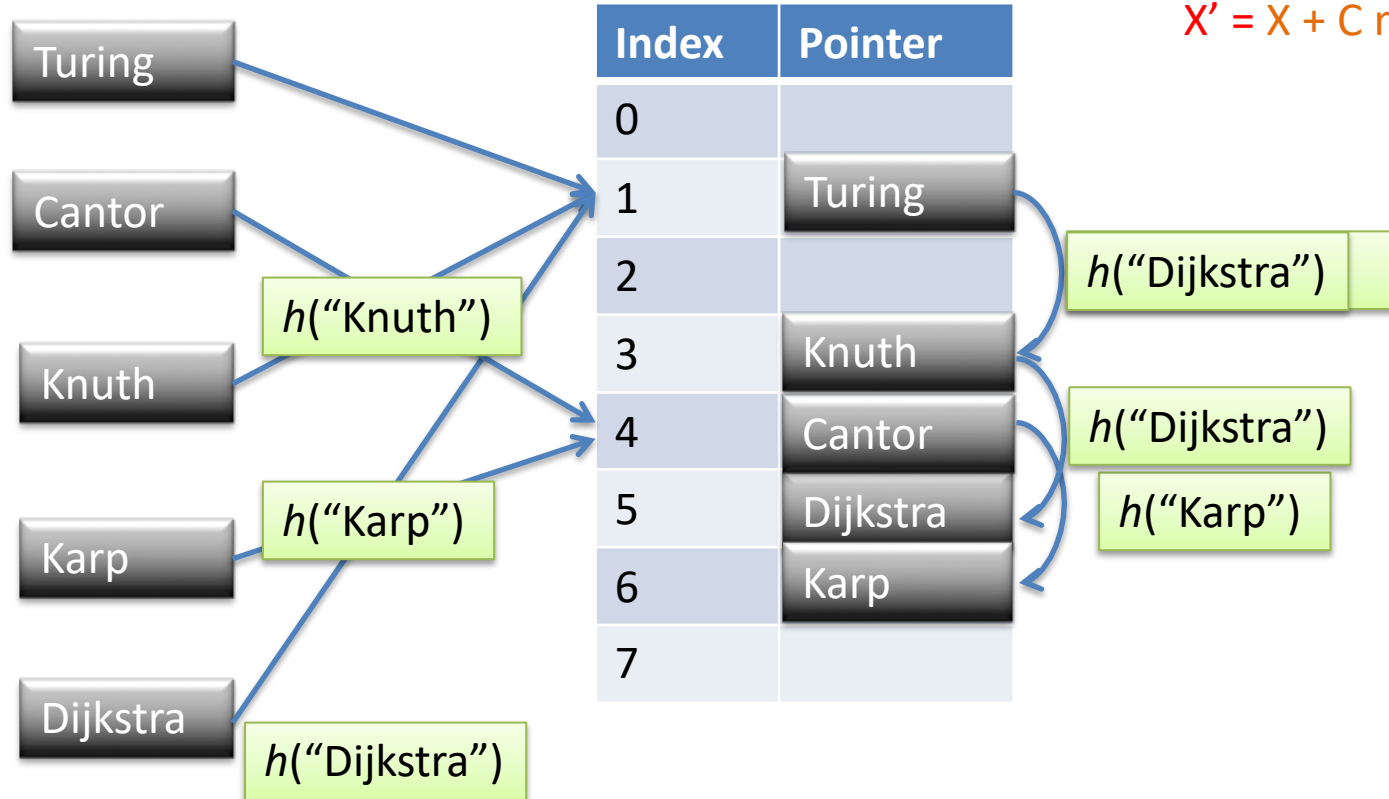
- Store all entries in the hash table itself, no pointer to the “outside”
  - For Linear Probing:
    - Calculate the final index  $X$  for the key
    - If the current location  $X$  is full,
      - try the next location  $X' = X + c \bmod M$
      - Continue until you find a free space
- Advantage
  - Less space waste
  - Perhaps good cache usage
- Disadvantage
  - More complex collision resolution
  - Slower operations

# Open Addressing

For this example:

$C = 2; M = 8$

$X' = X + C \bmod M$



# Conclusion

- HashMap:
  - A great data structure that can be used to:
    - Search
    - Insert
    - Delete
    - In  $O(1)$  time.
  - Not good for Range-Query
  - Not good for finding Max or Min
  - Next, one example for you showing how the whole process.
    - For collision resolution, chaining is performed.

# Example

As an example, let's store **hostnames** and allow a fast look-up of the **corresponding IP address**

- The **key-value** pairs contains
  - (hostname, IP address)
  - *E.g.*, ("optima1", 129.97.94.57)
- Hash Function Used:
  - **ASCII code of the first character of the host name**
    - (A terrible choice for Hashcode!!!)
- Final Array Size, **M = 8**
  - That is: all the hostnames need to be mapped to values between **0....7**



# Example

Further explanation:

- Final Array Size,  $M = 8$

- That is: all the hostnames need to be mapped to values between  $0....7$

- How to achieve this:

the final hash value (index of the array) of a **Hostname** will be the **last 3 bits** of the first character in the **Hostname**

Example:

The hash of "optimal" is based on "o"

ASCII Code for "o" is 01101**111**

The last 3 bits of o is 111.

111 represents **7**

So, ("optimal", 129.97.94.57) should be stored at index **7** of the array.

(Note: We will get the same result by performing mod 8 operation)

# Example

The following is a list of the binary representation of each letter:

– "a" is 1 and it cycles from there...

a	01100001	n	01101110
b	01100010	o	01101111
c	01100011	p	01110000
d	01100100	q	01110001
e	01100101	r	01110010
f	01100110	s	01110011
g	01100111	t	01110100
h	01101000	u	01110101
i	01101001	v	01110110
j	01101010	w	01110111
k	01101011	x	01111000
l	01101100	y	01111001
m	01101101	z	01111010

# Example

Our **HashCode**:

```
int hash(String str) {  
    int firstHash = str.charAt(0);  
    return firstHash;  
}
```

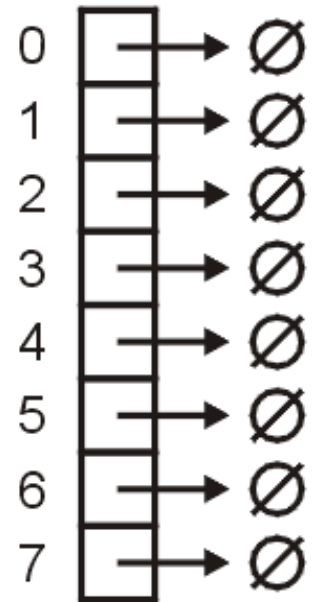
Our **Second (or Final) hash function (indexOf)** is:

```
int indexOf( int firstHash ) {  
    return firstHash % 8;  
}
```

OR

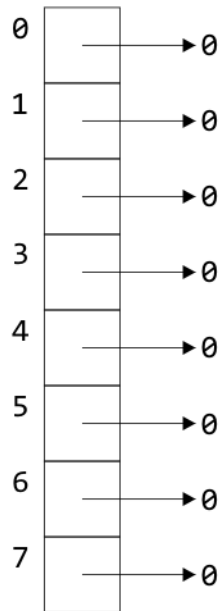
```
int indexOf( int firstHash ) {  
    return firstHash & 7;  
}
```

Both IndexOf method performs the exact same operation



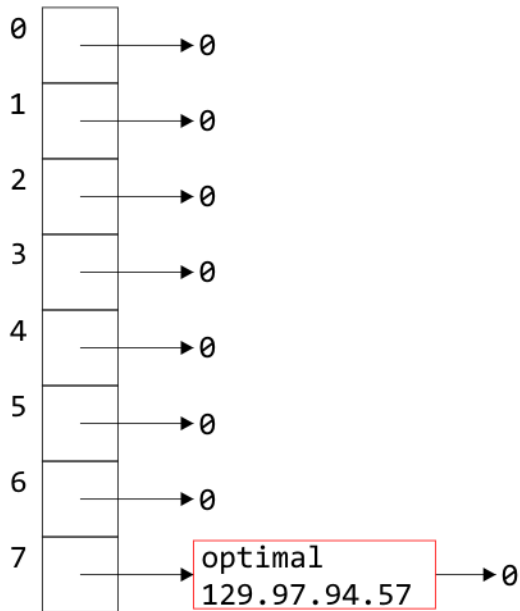
# Example

Starting with an array of 8 empty linked lists



# Example

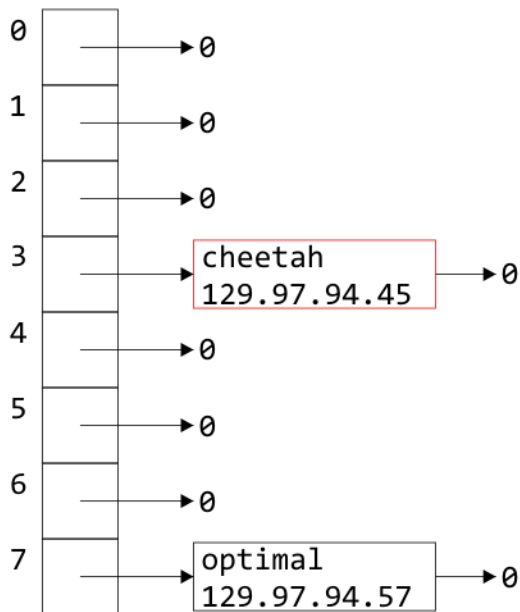
The pair ("optimal", 129.97.94.57) is entered into bin  
01101**111** = 7



# Example

Similarly, as "c" hashes to 3

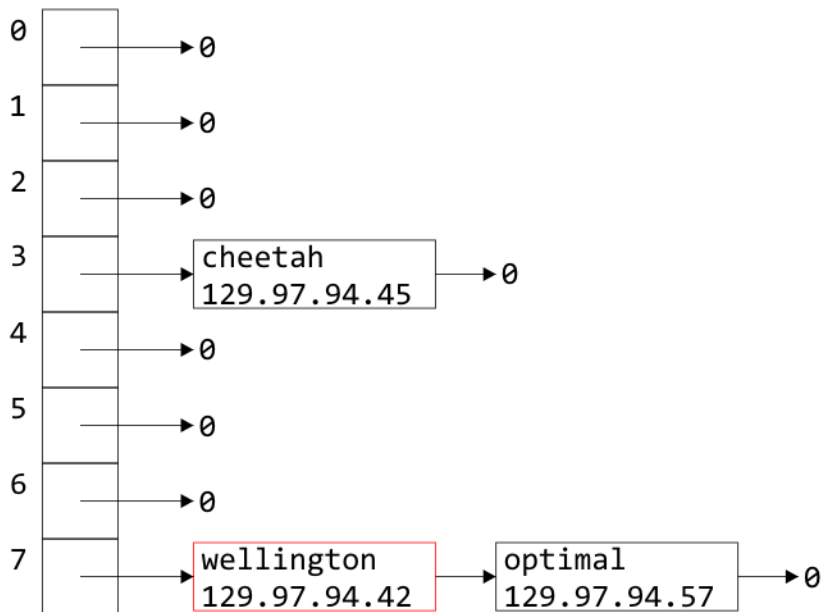
- The pair ("cheetah", 129.97.94.45) is entered into bin 3



# Example

The "w" in Wellington also hashes to 7

– ("wellington", 129.97.94.42) is entered into bin 7



# Example

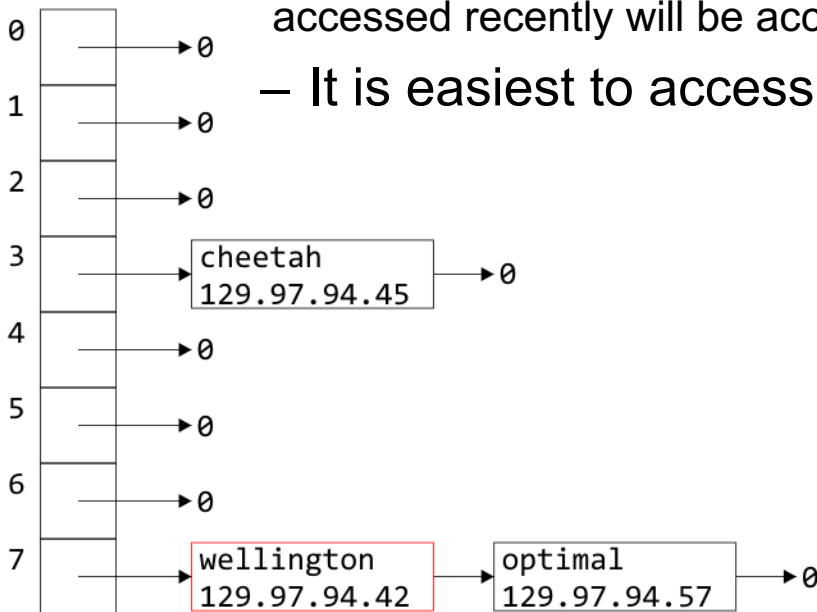
Why did I add at the beginning of the list?

- Do I have a choice?

- A good heuristic is

“unless you know otherwise, data which has been accessed recently will be accessed again in the near future”

- It is easiest to access data at the front of a linked list

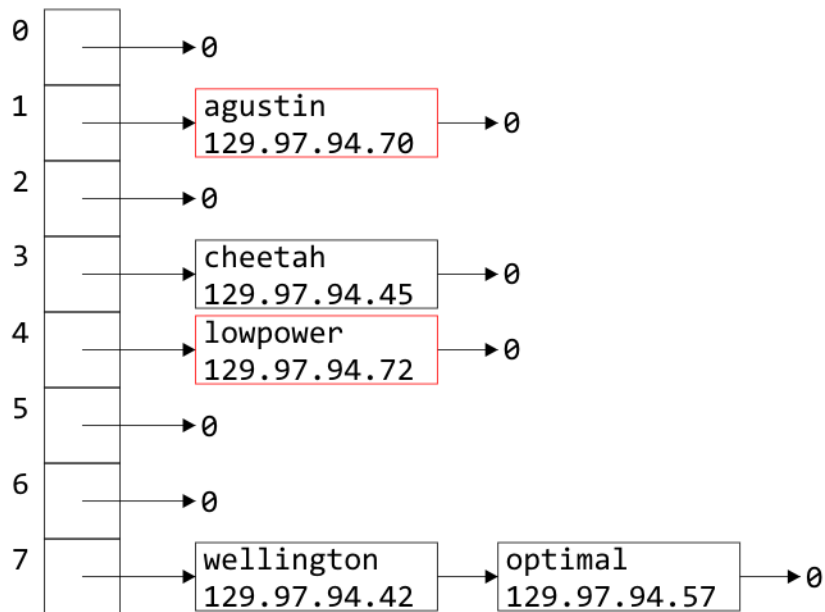


Heuristics include rules of thumb, educated guesses, and intuition



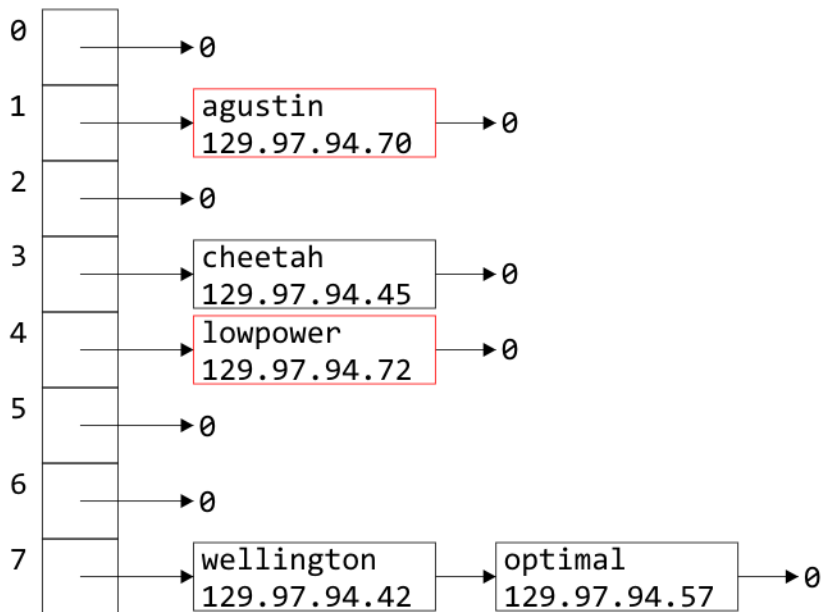
# Example

Similarly we can insert the host names "agustin" and "lowpower"



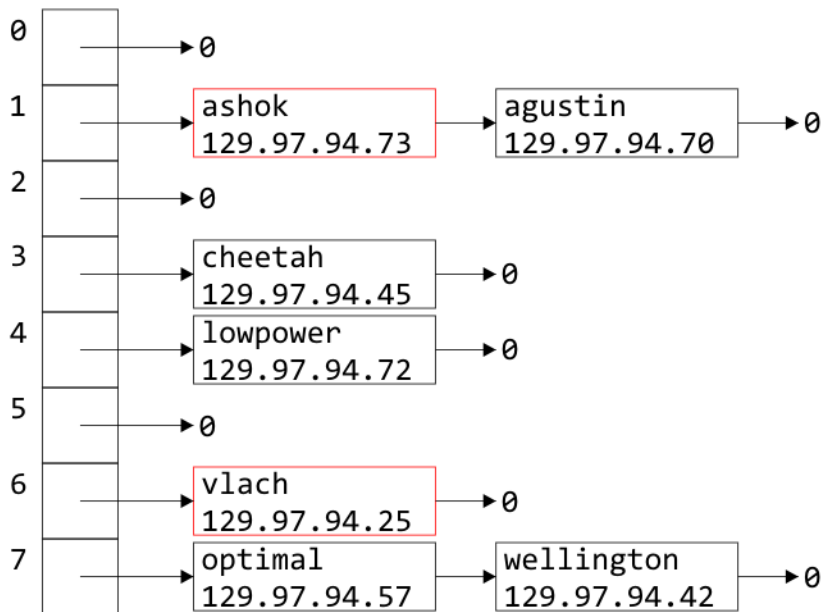
# Example

If we now wanted the IP address for "optimal", we would simply hash "optimal" to 7, walk through the linked list, and access 129.97.94.57 when we access the node containing the relevant string



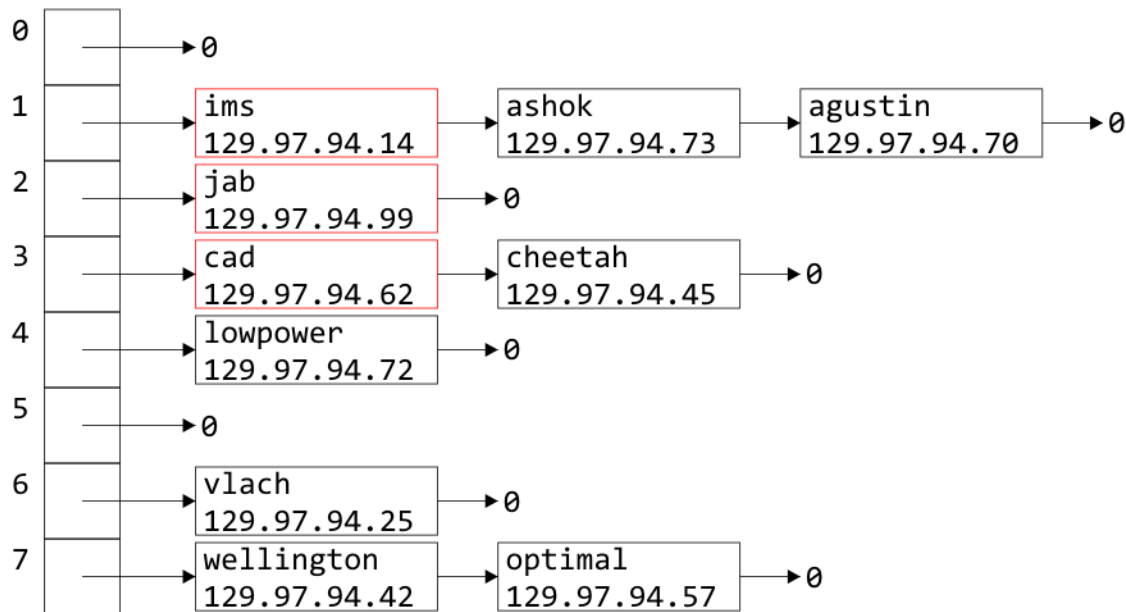
# Example

Similarly, "ashok" and "vlach" are entered into bin 7



# Example

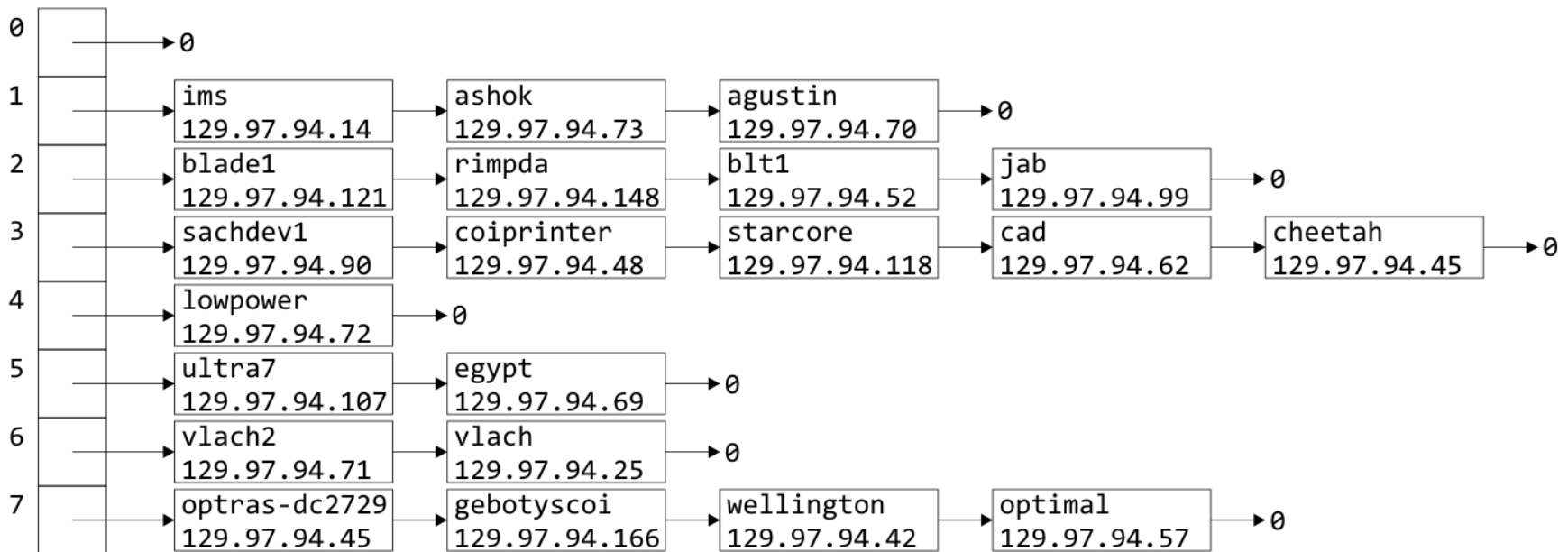
Inserting "ims", "jab", and "cad" doesn't even out the bins



# Example

Indeed, after 21 insertions, the linked lists are becoming rather long

- We were looking for  $O(1)$  access time, but accessing something in a linked list with  $k$  objects is  $O(k)$
- But, as long as size of the array is sufficiently larger than the number of entries, we are ok (Provided we have a good hash function to begin with)



# Acknowledgement

- A lot of these slides are taken from ECE 250 *Algorithms and Data Structures* course (University of Waterloo) offered by Prof. Douglas W. Harder.
- Special Thanks to Prof. Hung Ngo for his slides.