Data structures and Algorithms

Lecture 6: Hash tables

Dr. Julian Mestre
School of Computer Science

Some content is taken from material provided by the textbook publisher Wiley.

THE UNIVERSITY OF SYDNEY

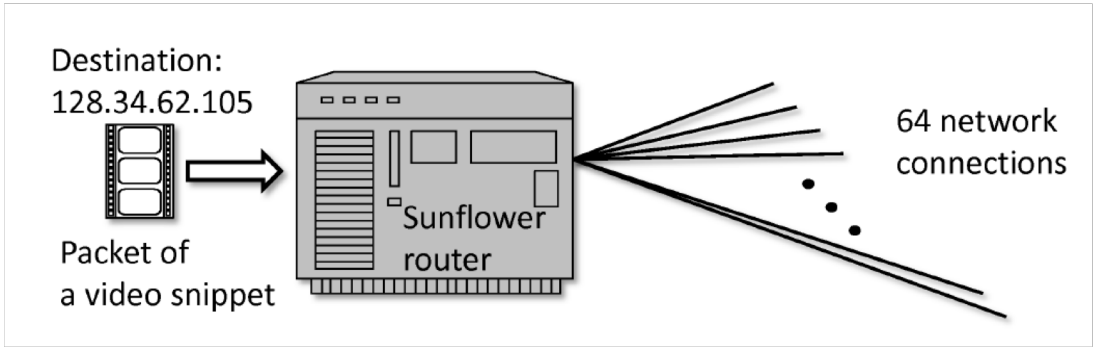
Application: Network Routers

Network routers process multiple streams of packets at high speed. To process a packet with destination k and data payload x, a router must determine which outgoing link to send the packet along

Such a system needs to support:

- destination-based lookups, i.e., get(k) operations that return the outgoing link for destination k
- updates to the routing table, i.e., put(k, c) operations, where c is the new outgoing link for destination k.

Ideally, we would like to achieve O(1) time for both operations.



Recall: Maps

A map models a searchable collection of key-value pairs (a.k.a., items or entries)

The main operations of a map are for searching, inserting, and deleting items

At most one item per key is allowed

Recall: Map Operations

- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size()
- is_empty()

Map Operations (extended)

- entries(): return an iterable collection of the entries in M
- keys(): return an iterable collection of the keys in M
- values(): return an iterable collection of the values in M

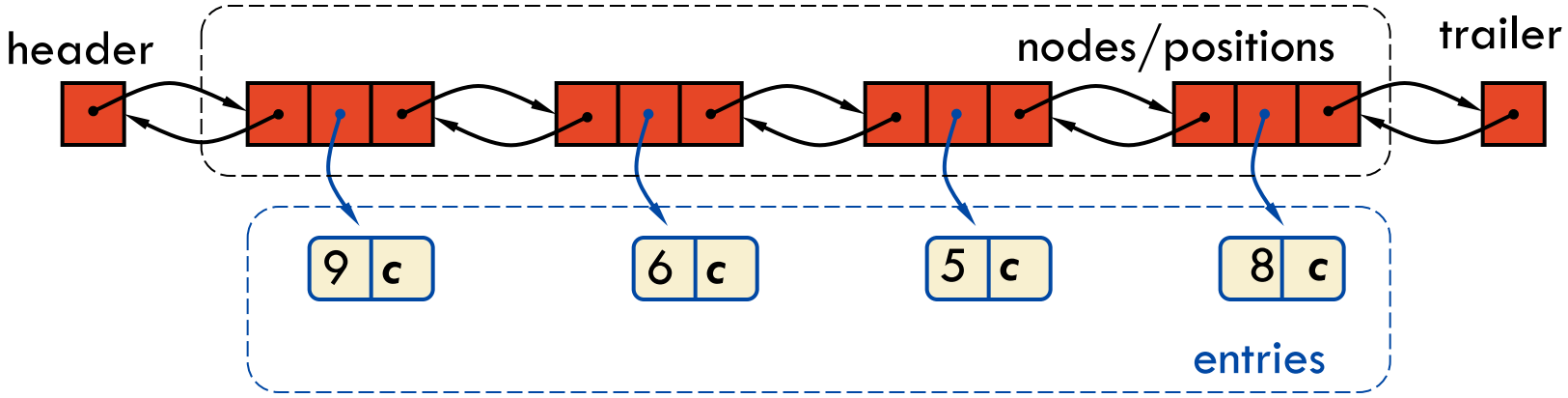
Example

Operation	Output	Map
		∅
is_empty()	**true**	∅
put(5,A)	**null**	(5,A)
put(7,B)	**null**	(5,A), (7,B)
put(2,C)	**null**	(5,A), (7,B), (2,C)
put(8,D)	**null**	(5,A), (7,B), (2,C), (8,D)
put(2,E)	C	(5,A), (7,B), (2,E), (8,D)
get(7)	B	(5,A), (7,B), (2,E), (8,D)
get(4)	**null**	(5,A), (7,B), (2,E), (8,D)
get(2)	E	(5,A), (7,B), (2,E), (8,D)
size()	4	(5,A), (7,B), (2,E), (8,D)
remove(5)	A	(7,B), (2,E), (8,D)
remove(2)	E	(7,B), (8,D)
get(2)	**null**	(7,B), (8,D)
is_empty()	**false**	(7,B), (8,D)

A Simple List-Based Map

We can implement a map using an unsorted list

- Store the items of the map in a list S (based on a doubly-linked list), in arbitrary order



Performance of a List-Based Map

Performance:

- put takes O(1) time since we can insert the new item at the beginning or at the end of the sequence
- get and remove take O(n) time since in the worst case we must traverse the entire sequence to look for an item with the given key

The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rare (e.g., historical record of logins to a workstation)

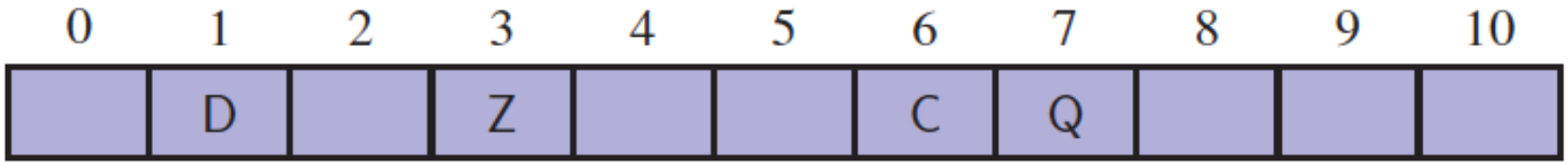
Simple Implementation with restricted keys

Maps support the abstraction of using keys as addresses to get items

Consider a restricted setting in which a map with n items with keys in a range from 0 to $N - 1$, for some $N \geq n$.

- Implement with an array of size N
- Key can be index so entries can be located directly
- $O(1)$ operations (get, put, remove)

Drawback is that usually $N \gg n$, e.g. StudentID is 9 digits, so a Map with StudentID key can be stored in array of 10,000,000,000 entries (way more than the number of students).



Evaluation of this structure

Really good worst-case runtime

Often, bad space utilization when key set is sparse in the space of possible keys, as in StudentID example

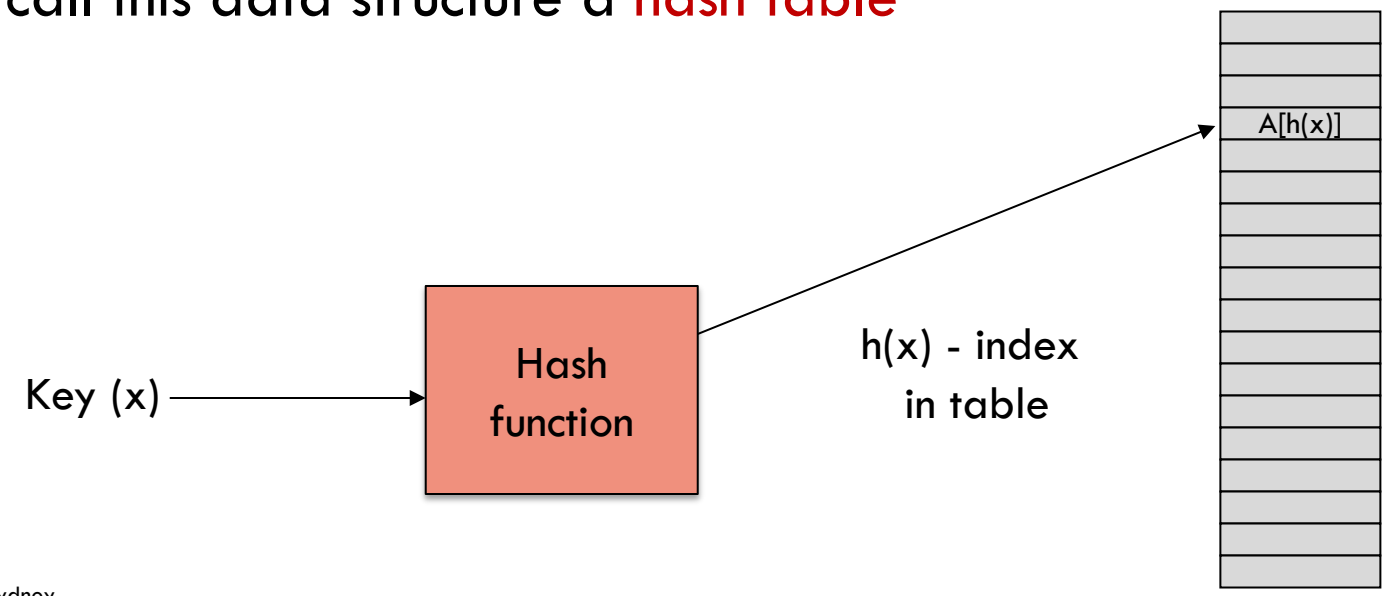
Unable to handle more general keys like strings

Hash Functions and Hash Tables

To get around these issues, we use a **hash function h** to map keys to corresponding indices in an array A.

- h is a mathematical function (always gives same answer for any particular x)
- h is fairly efficient to compute

We call this data structure a hash table



Hash Functions and Hash Tables

A hash function h maps keys of a given type to integers in a fixed interval $[0, N - 1]$.

- Example: $h(x) = x \bmod N$ is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x

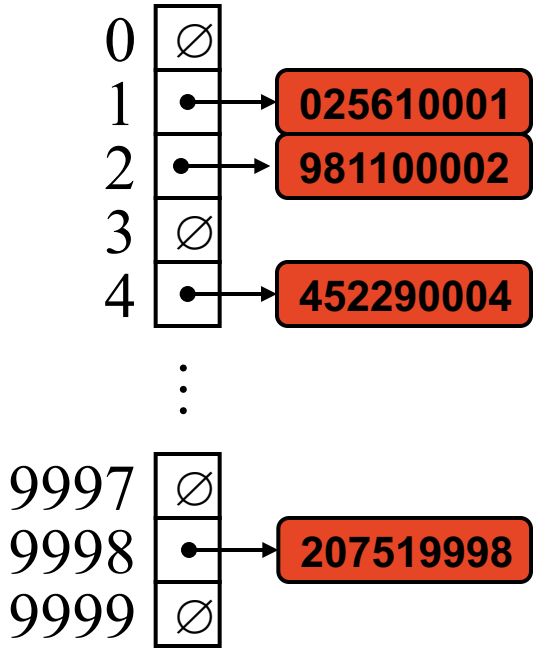
A hash table for a given key type K consists of

- Hash function $h: K \rightarrow [0, N-1]$
- Array (called table) of size N
- Ideally, item (x, o) is stored at $A[h(x)]$

Example

We design a hash table for a map storing entries as SIDs (student ids, a nine-digit positive integer).

Our hash table uses an array of size $N = 10{,}000$ and the hash function $h(x)$ = last four digits of x



Choice of Hash functions

Choosing a good hash function is not straightforward (to be discussed)

For our examples, we usually use very simple (and not good) choices that can be calculated by hand

- e.g. for unbounded integer key in array of size 11, we might use remainder mod 11 as hash function
- e.g. for String key, in array of size 10 we might do an example where h(S) = (position in alphabet of first character of S) mod 10, so h("Mary") = 3 since M is 13-th character in alphabet

Arithmetic modulo N

x mod N is mathematical notation for remainder

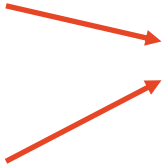
- If $x = c \cdot N + r$ with $0 <= r < N$ then $r = x \bmod N$
- Also $r = x - N \cdot \lfloor x/N \rfloor$
- So numbers wrap-around when working mod N
  - $35 \bmod 10 = 5$
  - $20 \bmod 10 = 0$
- Java/Python operator (x % N)

Hash Functions

Many types of keys to start from : integers, floating point numbers, strings, or arbitrary objects (for example a whole binary search tree)

A hash function h is usually the composition of two functions:

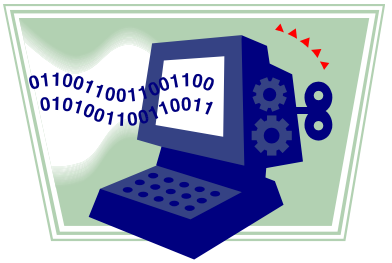
- Hash code:
  h_1 : keys $\rightarrow$ integers
- Compression function:
  h_2 : integers $\rightarrow [0, \boldsymbol{N} - 1]$



$h(x) = h_2(h_1(x))$

The goal of the hash function is to "disperse" the keys in an apparently random way. In general we want to avoid having many items being hashed to the same location.

Common Hash Codes



Designing a good hash code is a bit of an artform.

There two general approaches that one can take:

- view the key k as a tuple of integers $(x_1, x_2, \ldots, x_d)$ with each being an integer in the range [0, M-1] for some M
- view the key k as (possibly very large) nonnegative integer

Examples:

- strings, image
- IP address, account number

Summing components

Used for keys $k = (x_1, x_2, \ldots, x_d)$. There are many options:

- $h(k) = \sum_i x_i$
- $h(k) = \sum_i x_i \bmod p$ where p is a prime
- $h(k) = \bigoplus_i x_i \bmod p$

May cause problems because these hash code are invariant under permutations of the key tuple.

Example: “mate”, “meat”, “tame”, “team” all map to same code

Summing components

Used on keys $k = (x_1, x_2, \ldots, x_d)$. For a given value of a we define

$$h(k) = x_1\, a^{d-1} + x_2\, a^{d-2} + \ldots + x_{d-1}\, a + x_d$$

Now two permutations of the same tuple need to collide

Some observations:

- can be evaluated with Horner's algorithm in $O(d)$ time
- arithmetic ops usually done modulo a prime to avoid overflow
- value of a is chosen empirically to avoid collisions

Modular division

Used on keys k that are positive integers

$$h(k) = k \bmod N$$

for some primer number N

Fact: If keys are randomly uniformly distributed in $[0, M]$ where $M \gg N$ then the probability that two keys collide is $1/N$

Alas, keys are usually not random.

Random Linear Hash Function

Used on keys k that are positive integers

$$h(k) = ((a\,k + b) \bmod p) \bmod N$$

for some primer number p, and a and b are chosen uniformly at random from $[1, p-1]$ with $a \neq 0$

Fact: If the keys are in the range $[0, M]$ and $p > M$ then the probability that two keys collide is $1/N$

[See Theorem 6.3 in GT for a proof]

Universal hash functions

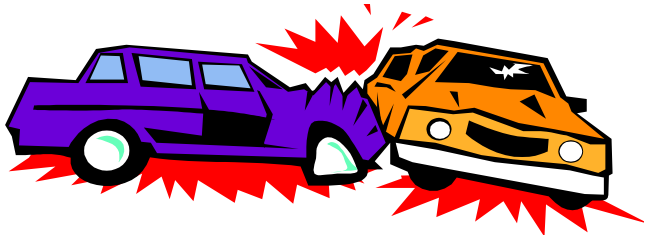
Suppose that [0, M] is the range of our keys and we need a has function with range in [0, N-1]

Let H be a family of such hash functions. We say that H is 2-universal if picking h uniformly at random (UAR) from H yields

$$Pr[h(i) = h(j)] <= 1/N$$

Fact: Let h be a function chosen UAR from a 2-universal family then the expected number of collision for a given key k in a set of n keys is n/N

Collision Handling



Collisions occur when two or more elements are hashed to the same location in our array

A good hash function makes collisions rare

However, when collision do happen we need to have a method for dealing with them:

- Separate chaining
- Linear probing
- Cuckoo hashing

Separate Chaining

Let each cell in the table point to a linked list holding the entries that map there

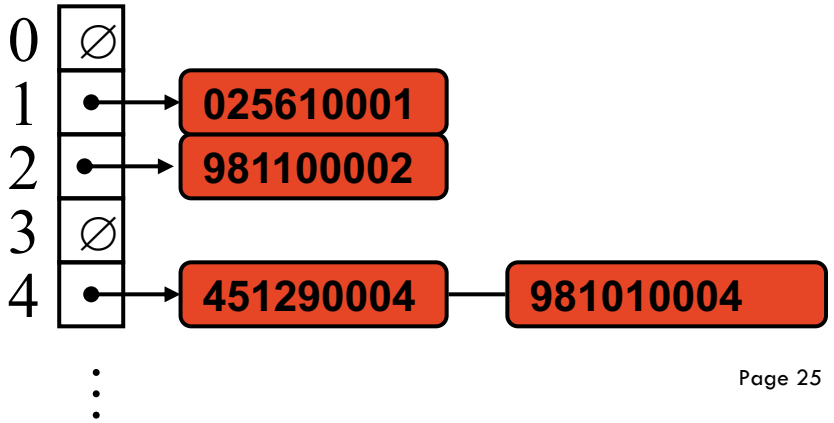
Get, put, and remove operations are delegated to the appropriate list

Separate chaining is simple, but requires additional memory outside the table

```
def get(k):
   return A[h(k)].get(k)

def put(k,v):
   A[h(k)].put(k,v)

def remove(k):
   return A[h(k)].remove(k)
```



Performance of Separate Chaining

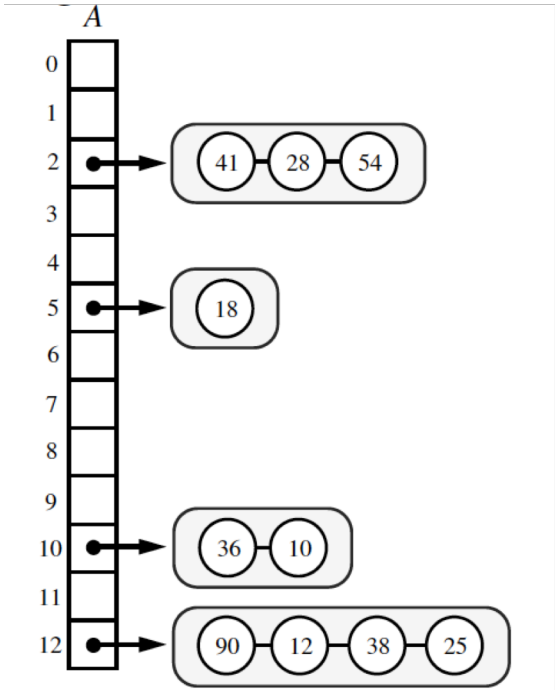
Assume that our hash function, maps n keys to independent uniform random values in the range [0,N−1].

Let X be a random variable representing the number of items that map to a bucket in the array A, then $E(X) = n/N$

The parameter n/N, is called the load factor of the hash table, usually written as α.

The expected time for hash table operations is $O(1+\alpha)$ when collisions are handled with separate chaining.

But the worst case time is O(n), which happens when all the items collide into a single chain



Open addressing using Linear Probing

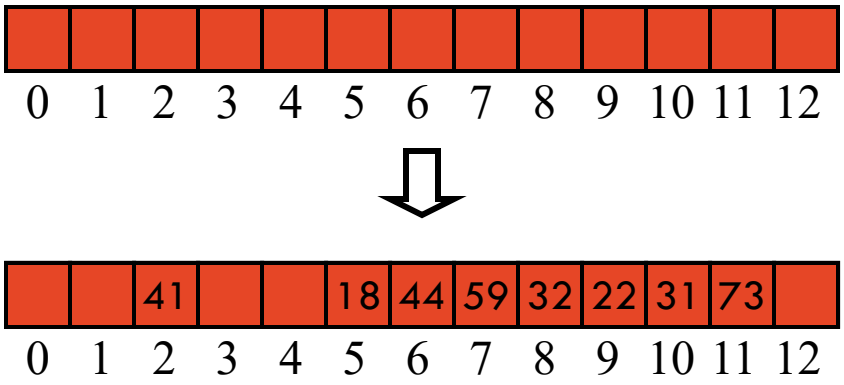
Open addressing: the colliding item is placed in a different cell of the table

Linear probing: handles collisions by placing the colliding item in the next (circularly) available cell

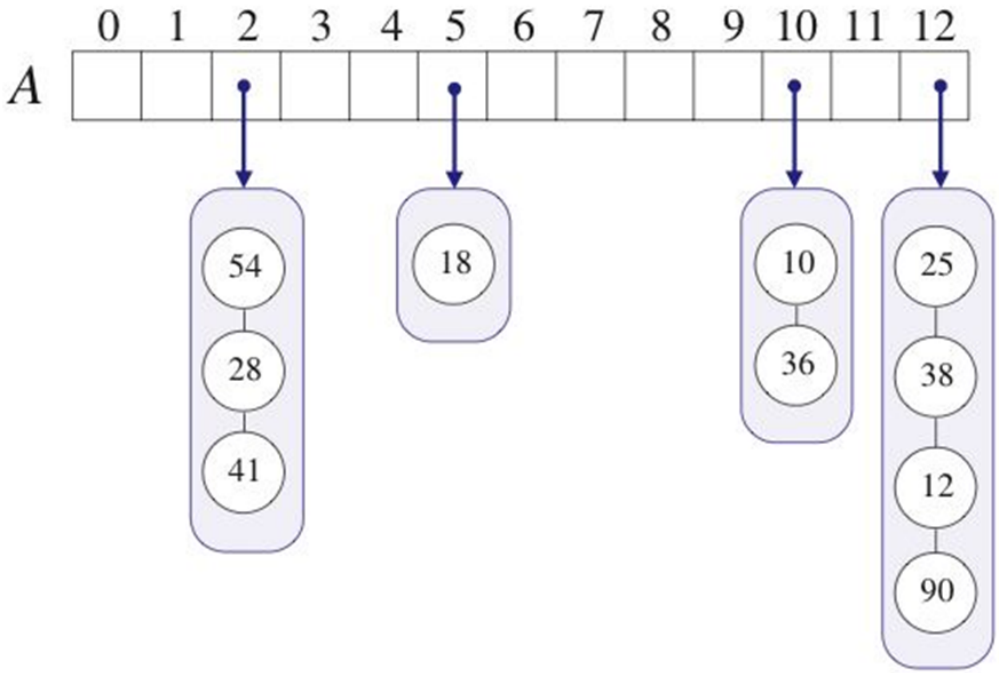
- Each table cell inspected is referred to as a probe
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

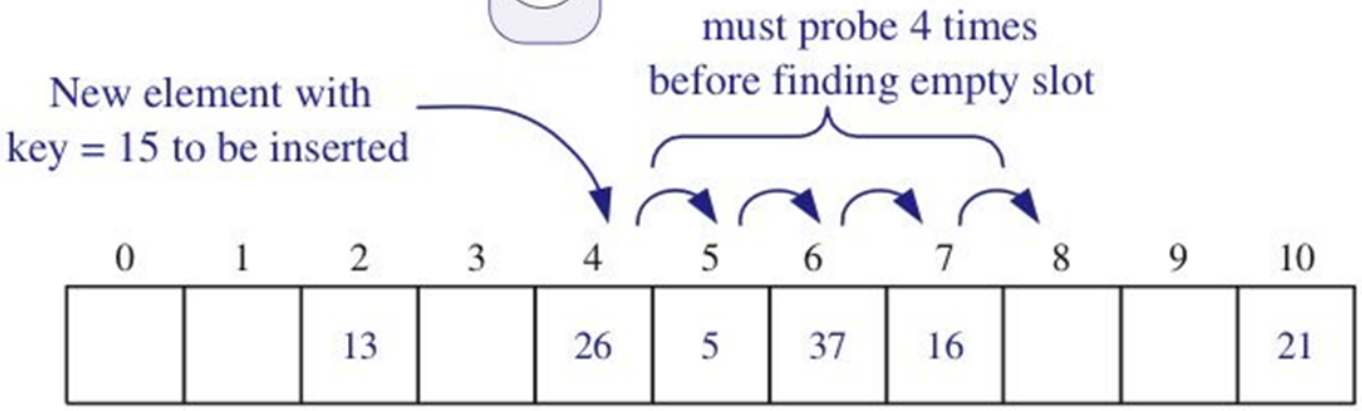
Example with h(x) = x mod 13.
Suppose we sequentially insert:

18 [5], 41 [2], 22 [9],
44 [5], 59 [7], 32 [6],
31 [5], 73 [11]

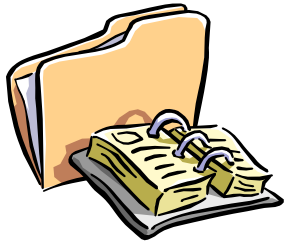


Chaining versus probing





Search with Linear Probing



How to implement get(*k*) in a hash table with linear probing:

- Start at cell *h*(*k*)
- Probe consecutive locations until one of the following occurs
  - An item with key *k* is found, or
  - An empty cell is found, or
  - *N* cells have been probed

```
def get(k):
  i ← h(k)
  p ← 0
  repeat
    c ← A[i]
    if c = ∅ then
      return null
     else if c.get_key()=k
      return c.get_value()
    else
      i ← (i + 1) mod N
      p ← p + 1
  until  p = N
  return null
```

Updates with Linear Probing

To handle insertions and deletions, we introduce a special object, called *DEFUNCT*, which replaces deleted elements

- get(*k*): must pass over cells with DEFUNCT and keep probing until the element is found, or until reaching an empty cell

- remove(*k*): search for the entry as in get(k). If found, replace it with the special item *DEFUNCT* and return element o

- put(*k*,o): search for the entry as in get(k), but we also remember the index j of the first cell we find that has DEFUNCT or empty.

  If we find key k, we replace the value there with o and return the previous value. If we don't find k, we store (*k*, o) in cell with index j

  Throw exception if table is full

Performance of Linear Probing

In the worst case, get, put, and remove take $O(n)$ time.

Fact: Assuming hash values are uniformly randomly distributed, expected number of probes for each get and put is $1/(1-\alpha)$ where $\alpha = n/N$ *is the load factor of the* hash table.

Thus, if the load factor is a constant < 1 then the expected running time for get and put operations is $O(1)$

In practice, hashing is very fast provided the load factor is not close to 100%, but removals complicate the implementation and degrade the performance.

Hash tables implementations

Recall that load factor of a hash table is defined as $\alpha = n/N$

Experiments and theory suggest that α should be kept not too high:

- Java uses chaining with $\alpha < 0.75$ and switches from a linked list to a binary search tree if bucket gets too large
- Python uses open addressing with $\alpha < 0.66$

When a hash table reaches its load factor, the table is replaced with a larger table (e.g., twice the size) and the elements are hashed over to the new table

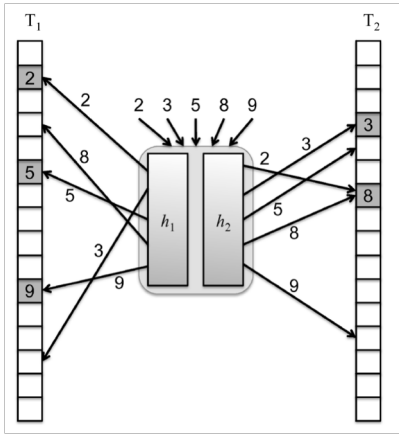
Cuckoo hashing

Main problem with the methods we've seen so far is that operations take O(n) time in the worst-case time.

Cuckoo hashing achieves worst-case O(1) time for lookups and removals, and expected O(1) time for insertions.

In practice Cuckoo hashing is 20-30% slower than linear probing but is still often used due to its worst case guarantees on lookups and its ability to handle removal more gracefully.

The Power of Two Choices



Use two lookup, T_1 and T_2, each of size N

Use two hash functions, h_1 and h_2, for T_1 and T_2 respectively

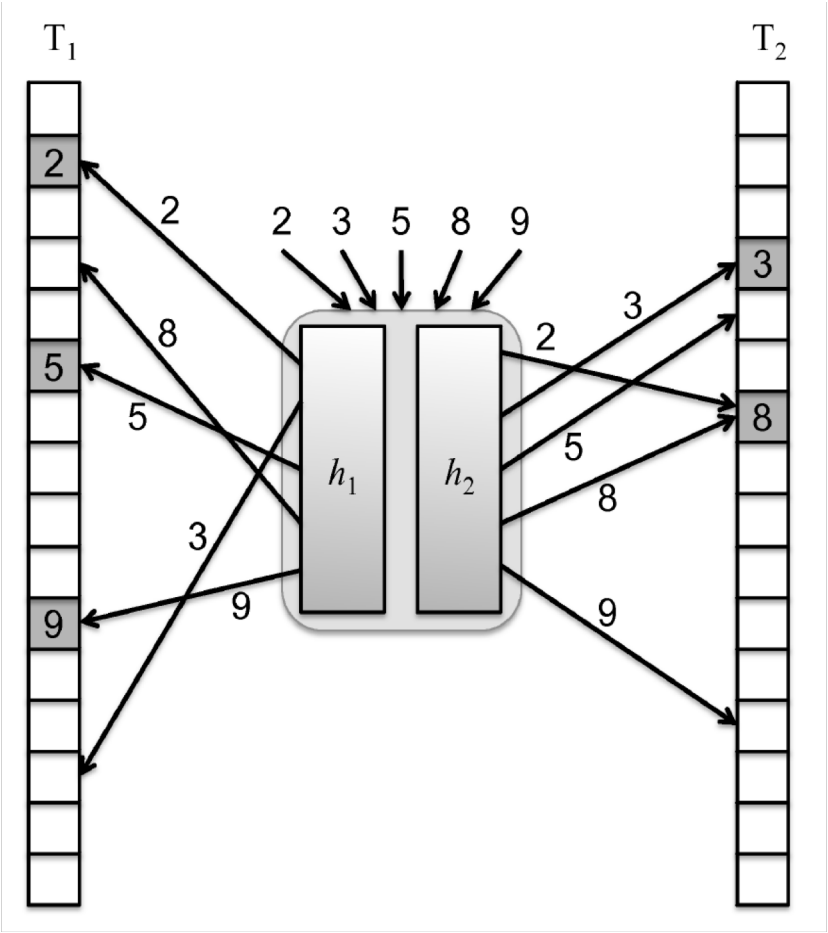
For an item with key k, there are only two possible places where we are allowed to store the item: $T_1[h_1(k)]$ or $T_2[h_2(k)]$

This restriction, simplifies lookup dramatically, while still allowing worst-case expected $O(1)$ running time for put and remove.

An Example of Cuckoo Hashing

Each key in the set S = {2, 3, 5, 8, 9} has two possible locations it can go, one in the table T_1 and one in the table T_2.

Note that 2 and 8 collide in T_2, but that is okay, since there is no collision for 2 in its alternative location in T_1.



Pseudo-code for get and remove

```
def get(k):
    if T1[h1(k)] ≠ null and T1[h1(k)].key = k then
        return T1[h1(k)].value
    if T2[h2(k)] ≠ null and T2[h2(k)].key = k then
        return T2[h2(k)]
    return null
```

```
def remove(k):
    temp ← null
    if T1[h1(k)] ≠ null and T1[h1(k)].key = k then
        temp ← T1[h1(k)].value
        T1[h1(k)] = null
    if T2[h2(k)] ≠ null and T2[h2(k)].key = k then
        temp ← T2[h2(k)].value
        T2[h2(k)] = null

    return temp
```

Both are simple and run in O(1) time

High level idea behind put

If a collision occurs in the insertion operation in the cuckoo hashing scheme, then we evict the previous item in that cell and insert the new one in its place.

This forces the evicted item to go to its alternate location in the other table and be inserted there, which may repeat the eviction process with another item, and so on.

Eventually, we either find an empty cell and stop or we repeat a previous eviction, which indicates an eviction cycle.

If we discover an eviction cycle, then we bail out or rehash all the items into larger tables

Intuition for the Name

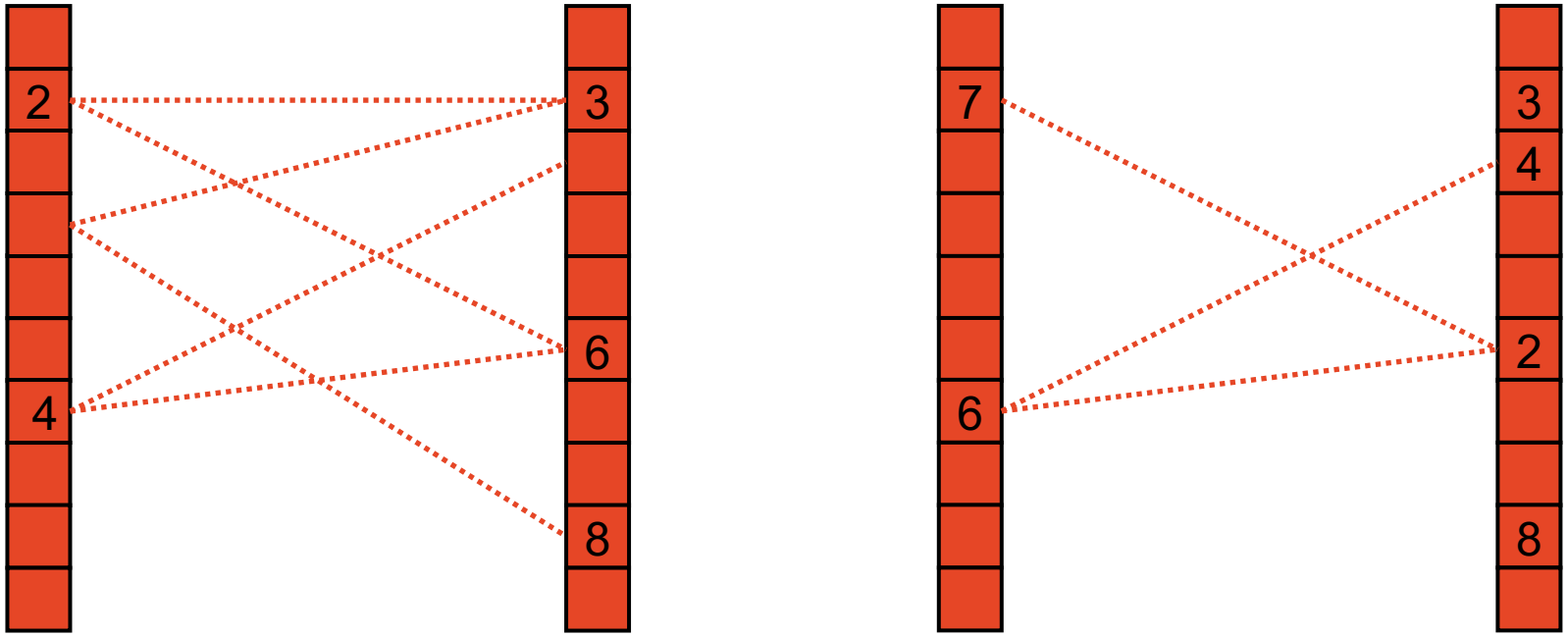
The name “cuckoo hashing” comes from the way the put(k, v) operation is performed in this scheme, because it mimics the breeding habits of the Common Cuckoo bird.

The Common Cuckoo is a brood parasite—it lays its egg in the nest of another bird after first evicting an egg out of that nest.

Catesby, Cockoo of Carolina. The bird that is. 18th Century color illustration. Early American bird print. Catesby. Scan of 2 d images in the public domain believed to be free to use without restriction in the US.

Example Eviction Sequence

put(7) generated an eviction sequence of length 3:



Pseudo-code for put

```
def put(k, v):
    # try to fit item into T1
    if T1[h1(k)] ≠ null and T1[h1(k)].key = k then
        T1[h1(k)] ← (k, v)
        return
    # try to fit item into T2
    if T2[h2(k)] ≠ null and T2[h2(k)].key = k then
    T2[h2(k)] ← (k, v)
        return
    # start eviction sequence
    i←1
    repeat
        if Ti[hi(k)] = null then
            Ti[hi(k)] ← (k, v)
            return
        temp ← Ti[hi(k)]
        Ti[hi(k)] ← (k, v)
        (k, v) ← temp
        i ← 1 if i=2 else 2
    until a cycle occurs
    rehash elements
```

How to detect eviction cycles

Use a counter to keep track of the number of evictions. If we iterate enough times we are guaranteed to have a cycle.

Keep an additional flag for each entry. Every time we evict an entry, we flag it. After a successful put, we need to unflag the entries flagged.

The details of these strategies are not complicated and are left as an exercise for the tutorials.

Performance of Cuckoo Hashing

One can show that "long eviction sequences" happen with very low probability.

Fact: Assuming hash values are uniformly randomly distributed, expected work of n put operations is $O(n)$ provided $N > 2n$

Fact: Cuckoo hashing achieves worst-case $O(1)$ time for lookups and removals

Another ADT: Set

A set is an unordered collection of elements, without duplicates that typically supports efficient membership tests.

Elements of a set are like keys of a map, but without any auxiliary values.

Set ADT

add(e): Adds the element e to S (if not already present).

remove(e): Removes the element e from S (if it is present).

contains(e): Returns whether e is an element of S.

iterator( ): Returns an iterator of the elements of S.

There is also support for the traditional mathematical set operations of ***union***, ***intersection***, and ***subtraction*** of two sets S and T:

$$\begin{aligned} S \cup T &= \{e:\ e \text{ is in } S \text{ or } e \text{ is in } T\}, \\ S \cap T &= \{e:\ e \text{ is in } S \text{ and } e \text{ is in } T\}, \\ S - T &= \{e:\ e \text{ is in } S \text{ and } e \text{ is not in } T\}. \end{aligned}$$

addAll(T): Updates S to also include all elements of set T, effectively replacing S by $S \cup T$.

retainAll(T): Updates S so that it only keeps those elements that are also elements of set T, effectively replacing S by $S \cap T$.

removeAll(T): Updates S by removing any of its elements that also occur in set T, effectively replacing S by $S - T$.

Set implemented via Map

- Use a Map to store the keys, and ignore the value.
- Allows contains(k) to be answered by get(k)
- Similarly for add and remove
- Using HashMap for Map, gives main Set operations that usually can be performed in O(1) time.

MultiSet

- Like a Set, but allows duplicates
  - also called a Bag
  - operation count(e) says how many occurrences of e in collection
  - remove(e) removes ONE occurrence (provided e is in the collection already)
- Implement by Map where the element is the key, and the associated value is the number of occurrences

Practice vs Theory

In practice hash tables implementation are usually fast and people use them **as though** put, get, and remove take O(1) time

The analyses we covered in lecture assume uniformly random hash values, which are not possible to implement in practice. Removing these assumptions is an active area of research well beyond the scope of this class

In theory we do not know of an implementation of hash tables that can perform put, get, and remove O(1) time in the worst case

Theory of Hashing

There is rich Theory of hashing beyond the basic hashing schemes we covered in class:

- Quadratic probing
- Double hashing
- Perfect hashing
- Universal hash families that are k-wise independent
- Cuckoo hashing with a stash
- Pseudorandom generators
- Cryptographic hash functions
- etc