CS211FZ: Data Structures and Algorithms II

3- Hash Tables

LECTURER: CHRIS ROADKNIGHT

CHRIS.ROADKNIGHT@MU.IE

Introduction (1)

Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE. For example, a compiler that translates a programming language maintains a symbol table, in which the keys of elements are arbitrary character strings corresponding to identifiers in the language.

A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list - $\theta(n)$ time in the worst case - in practice, hashing performs extremely well.

Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$.

Hashing is everywhere

Hash tables are everywhere.

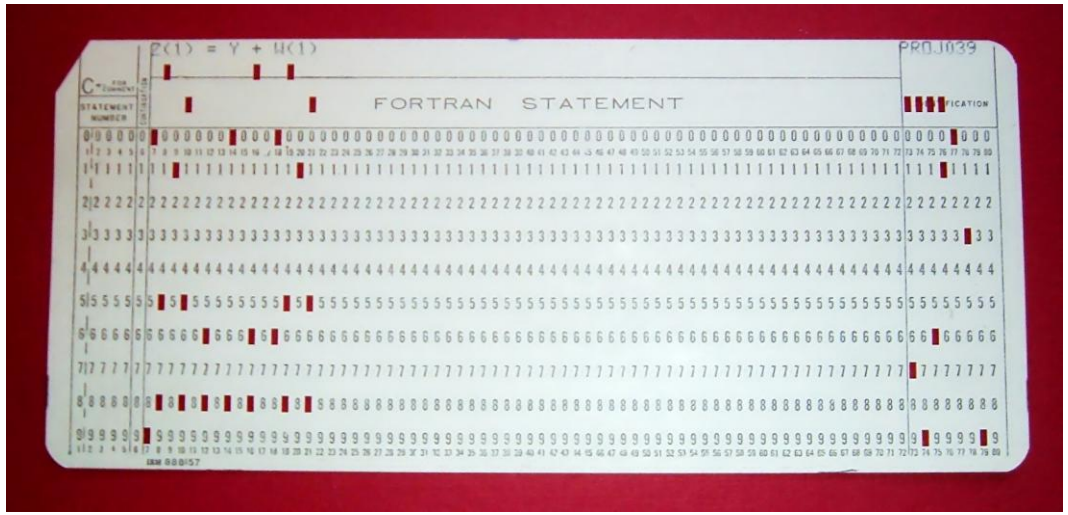
Python dictionaries, JavaScript objects, and database indexes are all hash tables underneath.

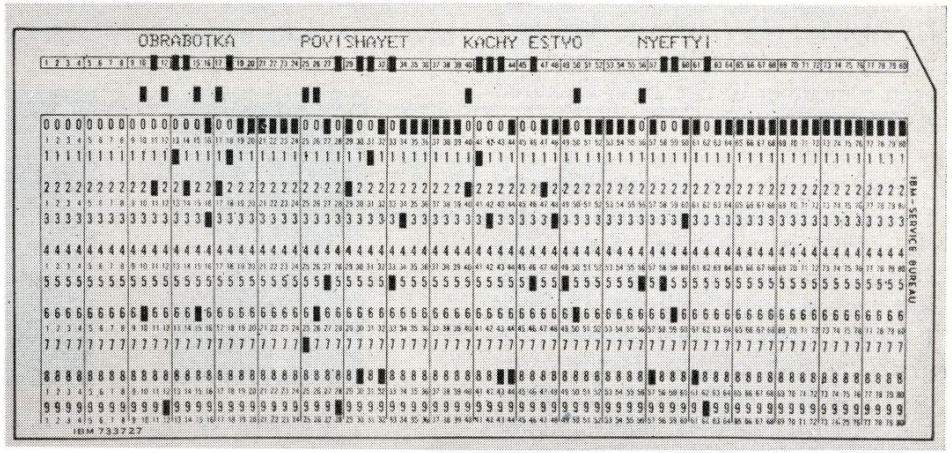
Google’s search engine uses them to map words to web pages.

Cryptocurrencies like Bitcoin rely on hash functions (like SHA-256) to secure transactions

Origin’s

The idea of hashing was first mentioned in the 1950s by IBM researcher Hans Peter Luhn to speed up searching through punched-card databases. It’s older than most programming languages we use today, proving good ideas stick around.





Compiler Symbol table usage

We put a symbol in the table along with its associated data

We access this when we need it

We delete it when no longer needed

But how do we know where it is in the table?

We want insertion, deletion and retrieval all to take O(1)

Symbol table

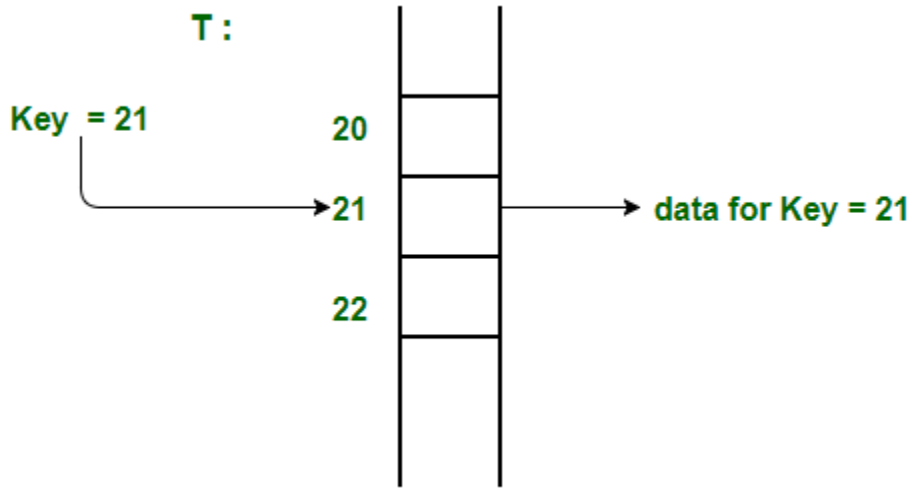
Name	Type	scope	Memory address	value	Additional Info
distance	variable	Global	0x1000	Uninitialized	Data type: float
pi	constant	Global	0x1004	3.14159	Data type: float, read-only
calculateArea	function	Global	0x1008	N/A	Return type: float
radius	parameter	Local	0x2000	0x1000	Data type: float

Direct-address tables

Direct-address tables

Direct addressing is a simple **data structure** that works well when the universe U of keys is reasonably small. Suppose that an application needs a dynamic set in which each element has a key drawn from the universe $U = \{0, 1, \ldots, m-1\}$, where m is not too large.

To represent the dynamic set, we use an array, or *direct-address table*, denoted by $T[0..m-1]$, in which each position, or *slot*, corresponds to a key in the universe U.



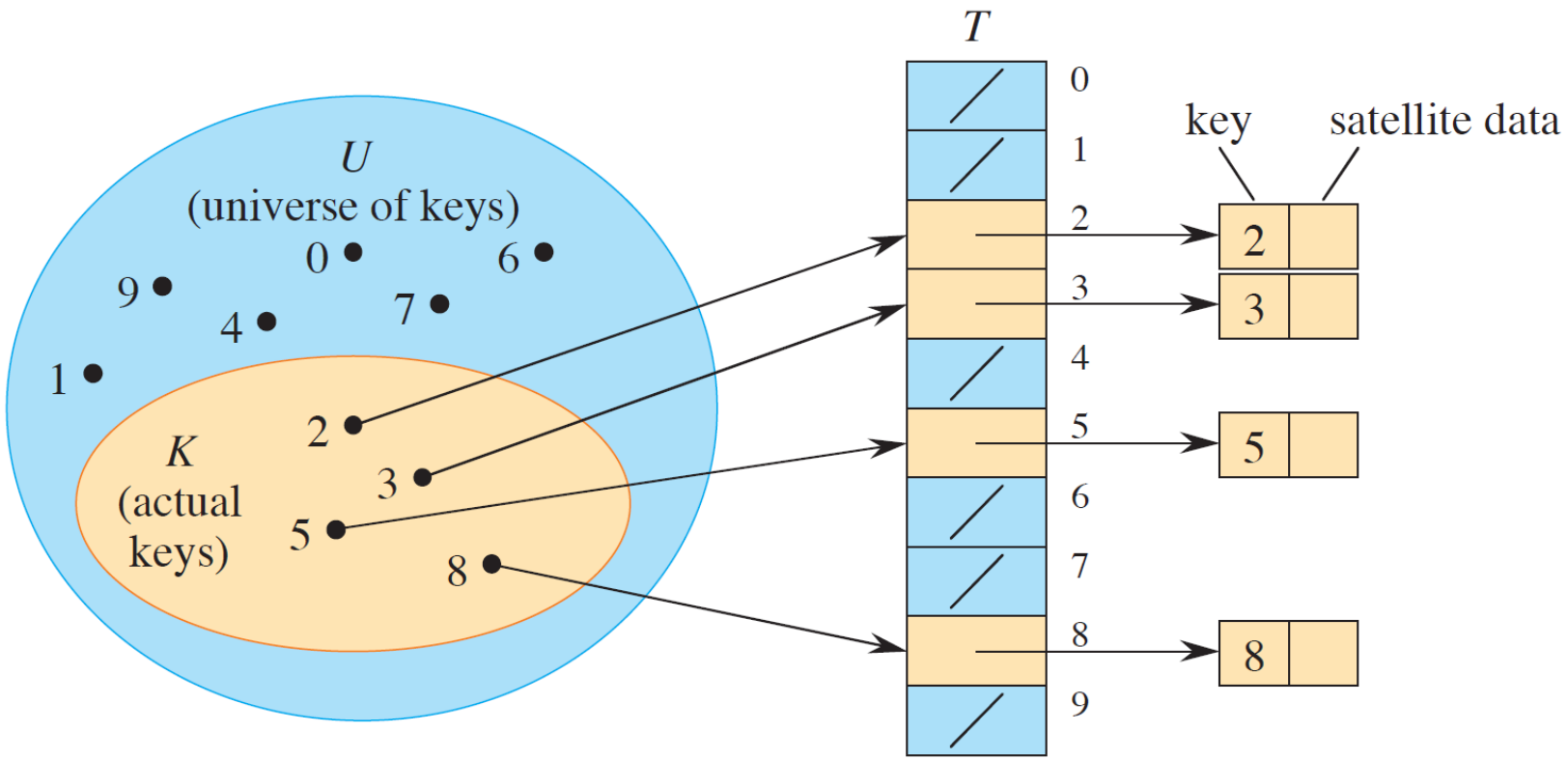
An example

Implementing a dynamic set by a direct-address table T.

Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index into the table.

The set $k = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements.

The other slots, in blue, contain NIL.



Implementation

```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] = NIL
```

Each of these operations take only $O(1)$ time.

Hash tables

The downside of direct addressing

If the universe U is large or infinite, storing a table T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. Furthermore, the set K of keys actually stored may be so small relative to U that most of the space allocated for T would be wasted.

Example:

Key = Age at the start of 2025, in seconds

Data = People in this room

1743225007 = Chris Roadknight

Xxxxxxxxxxx = A Student

2000+ million keys for 100 students

Hash Table

When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a directed-address table.

Specifically, the storage requirement reduces to $\theta(|K|)$ while maintaining the benefit that searching for an element in the hash table still requires only $O(1)$ time.

The catch is that this bound is for the *average-case time*, whereas for direct addressing it holds for the *worst-case time*.

What does this mean?

Basic Idea

With direct addressing, an element with key k is stored in slot k, but with hashing, we use a *hash function* h to compute the slot number from the key k, so that the element goes into slot $h(k)$. The hash function h maps the universe U of keys into the slots of a *hash table* $T[0:m-1]$:

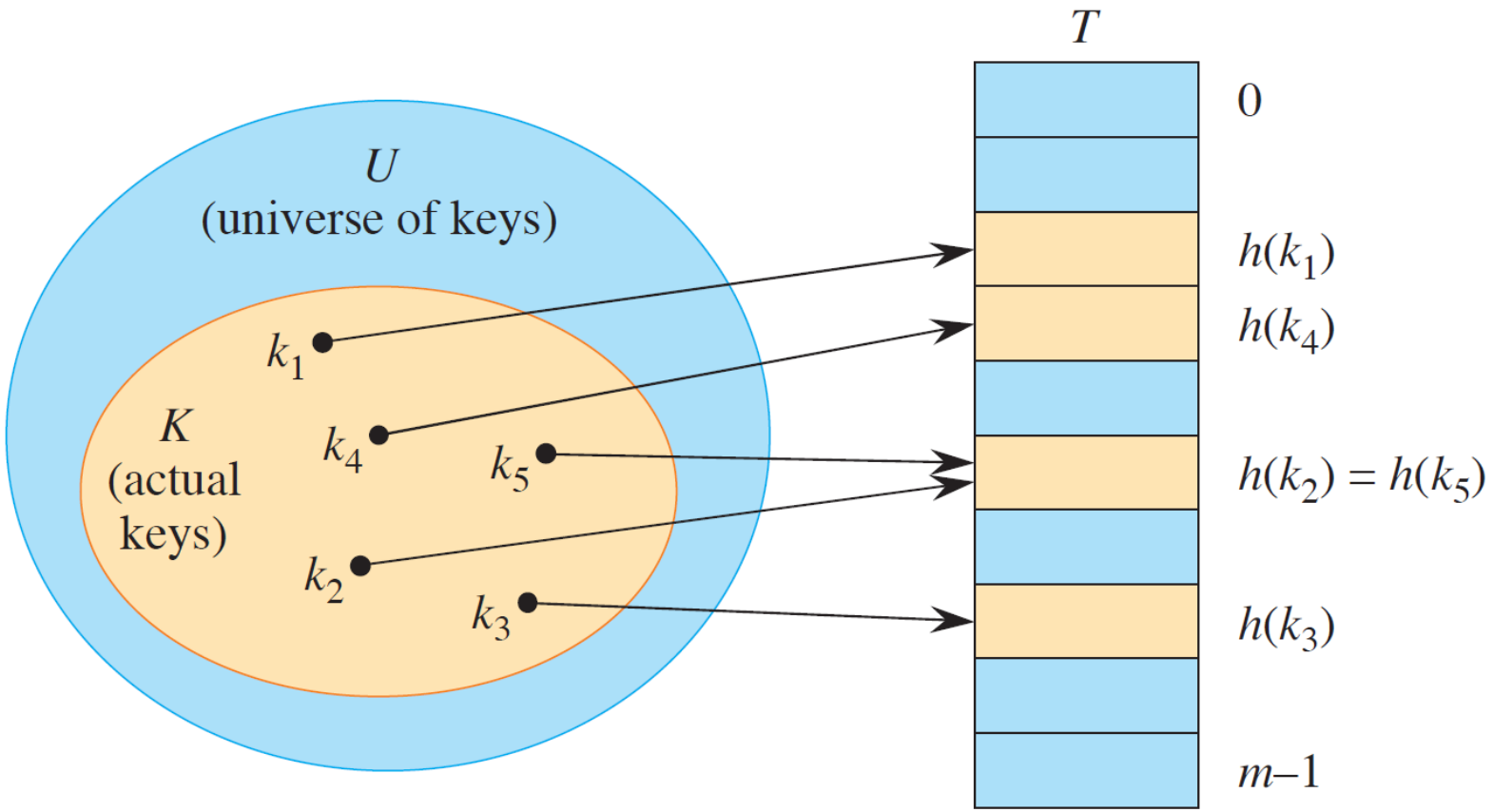
$$h: U \to \{0, 1, \ldots, m-1\},$$

where the size m of the hash table is typically much less than $|U|$. We say that an element with key k hashes to slot $h(k)$, and we also say that $h(k)$ is the hash value of key k.

An Example

Using a hash function h to map keys to hash-table slots.

Because keys k_2 and k_5 map to the same slot, they collide.



Collision (1)

There is one hitch, namely that two keys may hash to the same slot. We call this situation a collision. Fortunately, there are effective techniques for resolving the conflict created by collisions.

The ideal solution is to avoid collisions altogether. We might try to achieve this goal by choosing a suitable hash function h. One idea is to make h appear to be "random", thus avoiding collisions or at least minimizing their number.

No matter how clever your hash function is, collisions—where different keys land in the same spot—will happen. The "birthday problem" in probability shows that with just 23 items in a 365-slot table, there's a 50% chance of a collision

Collision (2)

The very term “to hash” evokes images of random mixing and chopping, captures the spirit of this approach. (Of course, a hash function h must be deterministic in that a given input k must always produce the same output $h(k)$.)

Because $|U| > m$, however, there must be at least two keys that have the same hash value and avoiding collisions altogether is impossible. Thus, although a well-designed, “random”-looking hash function can reduce the number of collisions, we still need a method for resolving the collisions that do occur.

In security, a "hash flood" attack, they send deliberately colliding keys to clog up a table, turning that (O(1)) speed into (O(n)). Web servers using hash tables for input processing—like versions of PHP or Python—got compromised by this in the early 2000s until we added randomisation to hash functions.

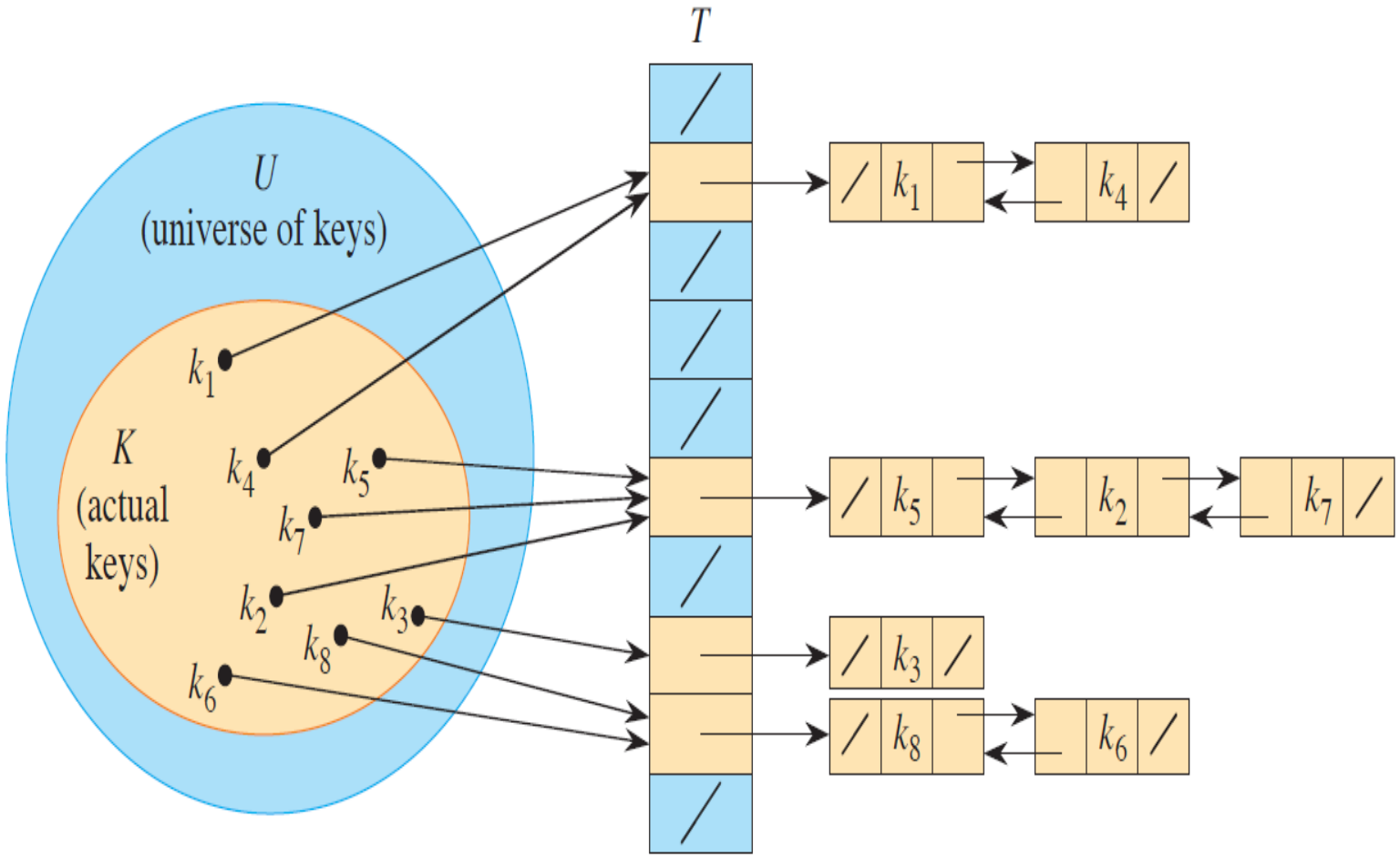
Collision resolution by chaining

Each nonempty hash-table slot $T[j]$ points to a linked list of all the keys whose hash value is j.

For example, $h(k_1) = h(k_4)$

and $h(k_5) = h(k_2) = h(k_7)$.

The list can be either singly or doubly linked. We show it as doubly linked because deletion may be faster that way when the deletion procedure knows which list element (not just which key) is to be deleted.



Implementation

When collisions are resolved by chaining, the dictionary operations are straightforward to implement.

We use the linked-list procedures from CS210FZ.

The worst-case running time for insertion is $O(1)$.

We can delete an element in $O(1)$ time if the lists are doubly linked

```
CHAINED-HASH-INSERT(T, x)
1   LIST-PREPEND(T[h(x.key)], x)

CHAINED-HASH-SEARCH(T, k)
1   return LIST-SEARCH(T[h(k)], k)

CHAINED-HASH-DELETE(T, x)
1   LIST-DELETE(T[h(x.key)], x)
```

Analysis of hashing with chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given a hash table T with m slots that stores n elements, we define the load factor $\propto$ for T as n/m, that is, the average number of elements stored in a chain.

Our analysis will be in terms of $\propto$, which can be less than, equal to, or greater than 1.

The worst-case

The worst-case behaviour of hashing with chaining is terrible: all n keys hash to the same slot, creating a list of length n.

The worst-case time for searching is thus

$$\Theta(n) + \textit{the time to compute the hash function}$$

no better than if we used one linked list for all elements.

The average-case

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average. What would be a bad hashing function?

For now, we assume that any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to. We call this *the assumption of simple uniform hashing*.

For $j = 0, 1, \ldots, m-1$, let us denote the length of the list $T[j]$ by n_j, so that

$n = n_0 + n_1 + \ldots + n_{m-1}$, and the expected value of n_j is

$E[n_j] = \propto = n/m.$

Hash Functions

Hash Functions

What makes a good hash function?

A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the *m* slots, independently of where any other key has hashed to.

Have no way to check this condition (since we rarely know the probability distribution from which the keys are drawn).

The keys might not be drawn independently.

In practice, we can employ heuristic techniques to create a hash function that preforms well. Qualitative information about the distribution of keys may be useful in this design process. For example, consider a compilers' symbol table.

Hash Functions

A good approach derives the hash value in a way that we expect to be independent of any patterns that might exist in the data. For example, the "division method" computes the hash value as the remainder when the key is divided by a special prime number.

We note that some applications of hash function might require stronger properties than are provided by simple uniform hashing. For example, we might want keys that are "close" in some sense to yield hash values that are far apart.

Interpreting keys as natural numbers

Most hash functions assume that the universe of keys is the set $N = \{0, 1, 2, \ldots\}$ of natural numbers. Thus, if the keys are natural numbers, we find a way to interpret them as natural numbers.

For example, we can interpret a character string as an integer expressed in suitable radix notation.

For example, we might interpret identifier pt as the pair of decimal integers $(112, 116)$, since $p = 112$ and $t = 116$ in the **ASCII** character set; then, express as a radix-128 integer, pt becomes $(112 \times 128) + 116 = 14452$.

The division method

In the **division method** for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

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

For example, if the hash table size $m = 12$ and the key is $k = 100$, then $h(k) = 4$. It requires only a single division operation, hashing by division is quite fast.

When using the division method, we usually avoid certain values of m. For example, m should not be a power of 2, since if $m = 2^p$ then $h(k)$ is just the p lowest-order bits of k. Unless we know that all low-order p−bit patterns are equally likely, we are better off designing the hash function to depend on all the bits of the key.

The division method

A prime not too close to an exact power of 2 is often a good choice for m. For example, suppose we wish to allocate a hash table, with collisions resolved by chaining, to hold roughly $n = 2000$ character strings, where a character has 8 bits.

We don't mind examining an average of 3 elements in an unsuccessful search*, and so we allocate a hash table of size $m = 701$. We could choose $m = 701$ because it is a prime near $2000/3$ but not near any power of 2.

Treating each key k as an integer, our hash function would be

$$h(k) = k \bmod 701.$$

*checking a key and not finding it in the chain

The multiplication method

The **multiplication method** for creating hash functions operations in two steps. First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. Then, we multiply this value by m and take the floor of the result. In short, the hash function is:

$$h(k) = \lfloor m\ (kA\ mod\ 1) \rfloor$$

Where "$kA\ mod\ 1$" means the fractional part of kA, that is $kA - \lfloor kA \rfloor$. An advantage of the multiplication method is that the value of *m* is not critical.

E.g. h(9) = [25*(9*0.5 mod 1)]= [25*(4.5 mod1)]= 25*0.5=**12**

Ascii String conversion

Chris = 67 +72+82+73+83 = 278

Ian = 73+65+78 = 2016

Paul = ……

Universal hashing

If a malicious adversary chooses the keys to be hashed by some fixed hash function, then the adversary can choose n keys that all hash to the same slot, yielding an average retrieval time of $\Theta(n)$. Any fixed hash function is vulnerable to such terrible worst-case behaviour.

The only effective way to improve the situation is to choose the hash function randomly in a way that is independent of the keys that are actually going to be stored. This approach, called **universal hashing**, can yield provably good performance on average, no matter which keys the adversary chooses.

Universal hashing

In universal hashing, at the beginning of execution we select the hash function at random from a carefully designed class of functions.

This is because we randomly select the hash function, the algorithm can behave differently on each execution, even for the same input, guaranteeing good average-case performance for any input.

Randomly selected hash function on random hash function (truly random isn't reversible)

Open Addressing

Open Addressing

In **open addressing**, all elements occupy the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL. When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table.

No list and no elements are stored **outside** the table, unlike in chaining. Thus, in open addressing, the hash table can "fill up" so that no further insertions can be made: one consequence is that the load factor α can never exceed 1.

Open Addressing

The advantage of open addressing is that it avoids pointers altogether. Instead of following pointers, we compute the sequence of slots to be examined. The extra memory freed by not storing pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collisions and faster retrieval.

To perform insertion using open addressing, we successively examine, or **probe**, the hash table until we find an empty slot in which to put the key. Instead of being fixed in the order $0, 1, \ldots, m-1$, the sequence of positions probed **depends on the key being inserted.**

Hash Function

To determine which slots to probe, we extend the hash function to include the probe number as a second input. Thus, the hash function becomes

$$h : U \times \{0, 1, \ldots, m - 1\} \rightarrow \{0, 1, \ldots, m - 1\}$$

With open addressing, we require that for every key k, the **probe sequence**

$$(h(k, 0), h(k, 1), \ldots, h(k, m - 1))$$

be a permutation of $(0, 1, \ldots, m - 1)$, so that every hash-table position is eventually considered as a slot for a new key as the table fills up.

Implementation

The HASH-INSERT procedure takes as input a hash table T and a key k that is assumed to be not already present in the hash table.

It either returns the slot number where it stores key k or flags an error because the hash table is already full.

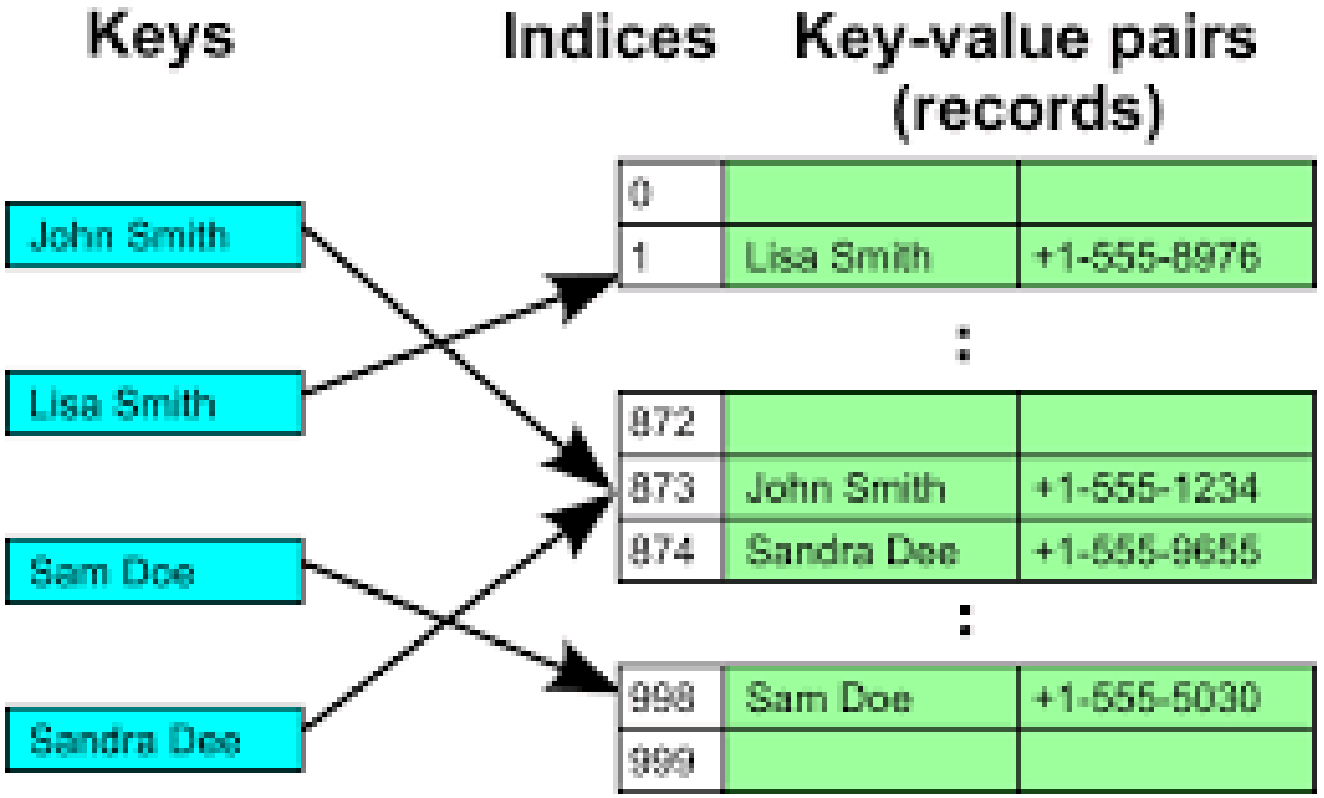
```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3      q = h(k, i)
4      if T[q] == NIL
5          T[q] = k
6          return q
7      else i = i + 1
8  until i == m
9  error "hash table overflow"
```

Implementation

The procedure HASH-SEARCH takes as input a hash table T and a key k, returning q if it finds that slot q contains key k, or NIL if key k is not present in table T.

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3      q = h(k, i)
4      if T[q] == k
5          return q
6      i = i + 1
7  until T[q] == NIL or i == m
8  return NIL
```

Open Addressing Example



Deletion

Deletion from an open-address hash table is difficult. When we delete a key from slot i, we cannot simply mark that slot as empty by storing NIL in it. If we did, we might be unable to retrieve any key k during whose insertion we had probed slot i and found it occupied.

We can solve this problem by marking the slot, sorting it in the special value DELETED instead of NIL. We then need to modify the procedure HASH-INSERT to treat such a slot as if it were empty so that we can insert a new key there. We do not need to modify HASH-SEARCH, since it will pass over DELETED values while searching.

Tombstones

A tombstone acts as a marker, indicating that a slot was once occupied but is now available for reuse.

How it Works:

Search: When searching, if a tombstone is encountered, the search continues along the probe sequence until either the element is found or an empty slot is reached.

Insertion: When inserting, if a tombstone is encountered, the new element can be placed in that slot.

Example:

Imagine a hash table using linear probing. If you insert elements with keys 1, 2, and 3, then delete 2, the slot for 2 would be marked as a tombstone. If you then search for 2, the search would continue past the tombstone until it finds the empty slot (or the element). If you insert a new element with key 4, it would be placed in the slot that was previously occupied by 2 (now marked as a tombstone).

Tombstones

A tombstone acts as a marker, indicating that a slot was once occupied but is now available for reuse.

How it Works:

Search: When searching, if a tombstone is encountered, the search continues along the probe sequence until either the element is found or an empty slot is reached.

Insertion: When inserting, if a tombstone is encountered, the new element can be placed in that slot.

Example:

Imagine a hash table using linear probing. If you insert elements with keys 1, 2, and 3, then delete 2, the slot for 2 would be marked as a tombstone. If you then search for 2, the search would continue past the tombstone until it finds the empty slot (or the element). If you insert a new element with key 4, it would be placed in the slot that was previously occupied by 2 (now marked as a tombstone).

Computing the probe sequences

Three techniques are commonly used to compute the probe sequences required for open addressing:

- Linear probing
- Quadratic probing
- Double hashing

These techniques all guarantee that $h(k,0)$, $h(k,1)$, …, $h(k,m-1)$ is a permutation of $0, 1, \ldots, m-1$ for each key k.

Linear Probing (1)

Given an ordinary hash function h^j: $U \to \{0, 1, \ldots, m-1\}$, which we refer to as an **auxiliary hash function**, the method of **linear probing** uses the hash function

$$h(k, i) = (h^j(k) + i\ )\ mod\ m$$

for $i = 0, 1, \ldots, m-1$. Given key k, we first probe $T[h^j(k)]$, i.e., the slot given by the auxiliary hash function. We next probe slot $T[h^j(k) + 1]$, and so on up to slot $T[m-1]$. Then we wrap around to slots $T[0], T\ [1]$, … until we finally probe slot $T[h^j(k) - 1]$. Because the initial probe determines the entire probe sequence, there are only m distinct probe sequences.

Linear Probing (2)

Linear probing is easy to implement, but it suffers from a problem known as **primary clustering**.

Long runs of occupied slots build up, increasing the average search time. Clusters arise because an empty slot preceded by i full slots gets filled next with probability $(i + 1)/m$.

Long runs of occupied slots tend to get longer, and the average search time increases.

Quadratic Probing

Quadratic probing is a collision resolution technique used in hash tables when two keys hash to the same index. Instead of searching for the next available slot linearly (like in linear probing), quadratic probing uses a quadratic function to determine the next slot to check. Typically, it follows a pattern like i^2, where i is the number of attempts made to resolve the collision.

For example, if the initial hash index is h(k), and a collision occurs, quadratic probing checks $h(k) + 1^2$, then $h(k) + 2^2$, $h(k) + 3^2$, and so on, until an empty slot is found or the table is deemed full.

It works best when the hash table is less than half full and the table size is a prime number, ensuring all slots can eventually be probed.

Double Hashing

Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. **Double hashing** uses a hash function of the form:

$$h(k,i) = (h_1(k) + ih_2(k)) \, mod \, m,$$

where both h_1 and h_2 are auxiliary hash functions. The initial probe goes to position $T[h_1(k)]$; successive probe positions are offset from previous positions by the amount $h_2(k)$, modulo m. Thus, unlike the case of linear or quadratic probing, the probe sequence here depends in two ways on the key k, since the initial probe position, the offset, or both, may vary.

An Example

Insertion by double hashing. The hash table has size 13 with

$h_1(k) = k \; mod \; 13$

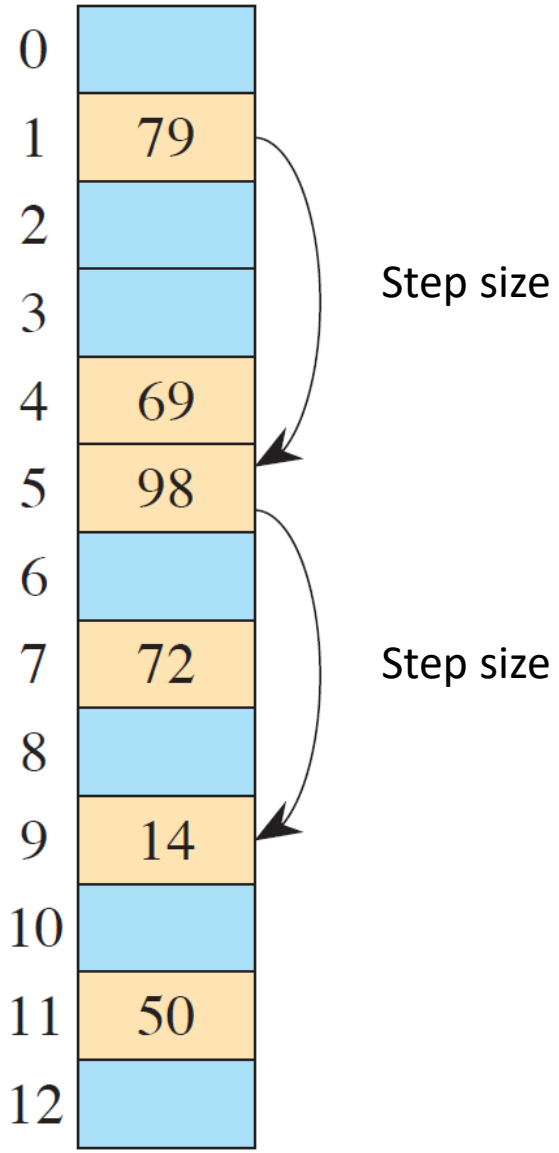
and

$h_2(k) = 1 + (k \; mod \; 11).$

Since $14 = 1(mod \; 13)$ and

$14 = 3 \; (mod \; 11),$

the key 14 goes into empty slot 9, after slots 1 and 5 are examined and found to be occupied.



Dynamic resizing

Once an open addressed hash table get too small for it’s keys it is doubled in size and rehashed

The load factor of a hash table, defined as the ratio of the number of stored elements to the table's total capacity (n/m), is a key indicator of when to dynamically resize it. The ideal threshold depends on the collision resolution method and performance goals, but a common rule of thumb is to resize when the load factor reaches around 0.7 (70%).

Resizing

Performance Trade-off: As the load factor increases, collisions become more frequent, degrading the hash table's performance from O(1) average-case time for lookups, inserts, and deletes to something closer to O(n) in the worst case due to clustering or long probe sequences.

Resizing at 0.7 keeps operations efficient while avoiding unnecessary memory waste or overfrequent resizing

Collision Resolution Specifics

Open Addressing (e.g., Quadratic Probing, Double Hashing): In open addressing, where all elements reside in the table itself, performance degrades faster as the load factor approaches 1 (full table). Studies and practical implementations suggest 0.5 to 0.7 as a sweet spot. Beyond 0.7, the number of probes increases sharply, especially for quadratic probing, which struggles with secondary clustering. Double hashing can tolerate slightly higher load factors (closer to 0.8) due to better distribution.

Chaining (Separate Chaining): With chaining, where collisions are resolved by linked lists at each slot, the table can handle load factors above 1 since slots can store multiple elements. However, resizing at 0.7 or 1 is still common to keep list lengths short (ideally O(1) rather than O(n)).

Resizing

Lower Thresholds: If memory is abundant and latency is critical (e.g., real-time systems), you might resize at 0.5 to minimize collisions further. Conversely, if memory is tight, you might push closer to 0.9, accepting slower performance.

Shrinking: If the load factor drops too low (e.g., below 0.25 or 0.1 after deletions), you might halve the table size to save space, though this is less common as it risks thrashing (repeated resizing).