Algorithms

(Hash Tables)

Pramod Ganapathi
Department of Computer Science
State University of New York at Stony Brook

September 19, 2021

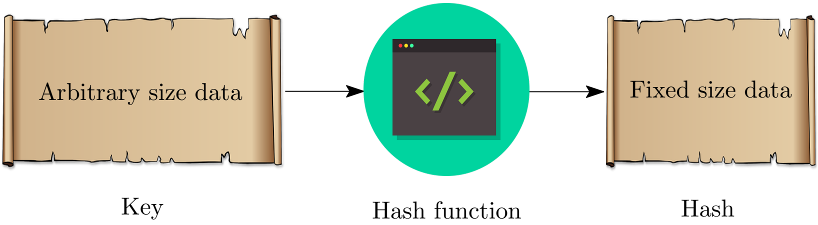
Contents

Hash Functions HOME

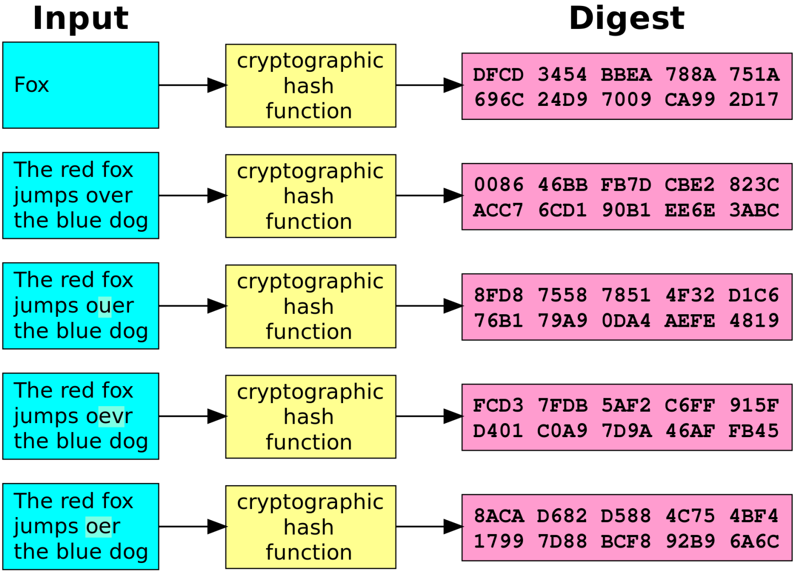
Hash function

Definition

- A hash function is a function that maps arbitrary size data to fixed size data.

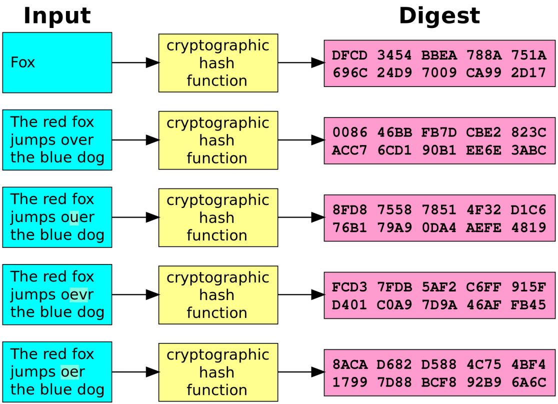


Cryptographic hash function



Source: Wikipedia

Properties of an ideal cryptographic hash function

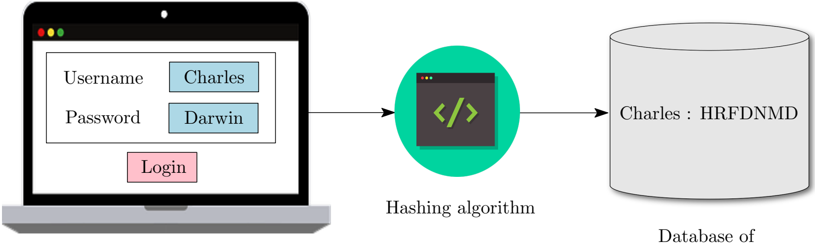


- Deterministic and fast
- Computing message from hash value is infeasible
- Computing two messages having same hash value is infeasible
- Tiny change in message must change the hash value drastically

Applications of hashing

- Web page search using URLs
- Password verification
- Symbol tables in compilers
- Filename-filepath linking in operating systems
- Plagiarism detection using Rabin-Karp string matching algorithm
- English dictionary search
- Used as part of the following concepts:
  - finding distinct elements
  - counting frequencies of items
  - finding duplicates
  - message digests
  - commitment
  - Bloom filters

Password authentication



Password login

Hashing algorithm

Database of encrypted passwords

Hash Tables HOME

Hash table

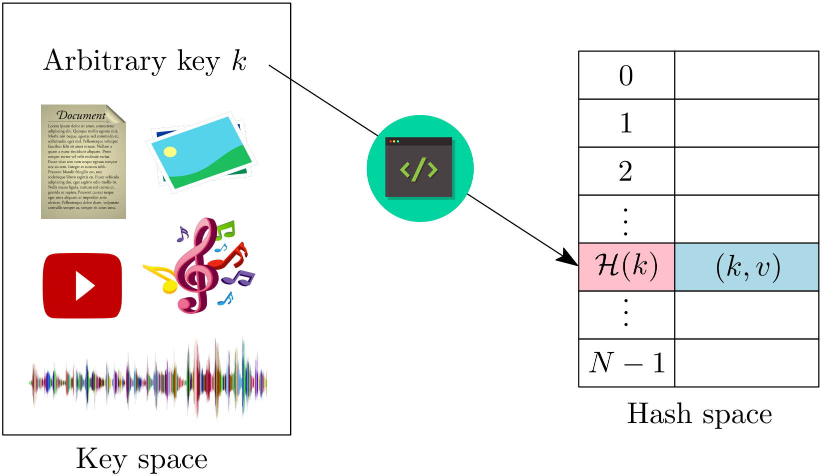
- A hash table is a data structure to implement dictionary ADT
- A hash table is an efficient implementation of a set/multiset/map/multimap.
- A hash table performs insert, delete, and search operations in constant expected time.

Balanced search trees vs. Hash tables

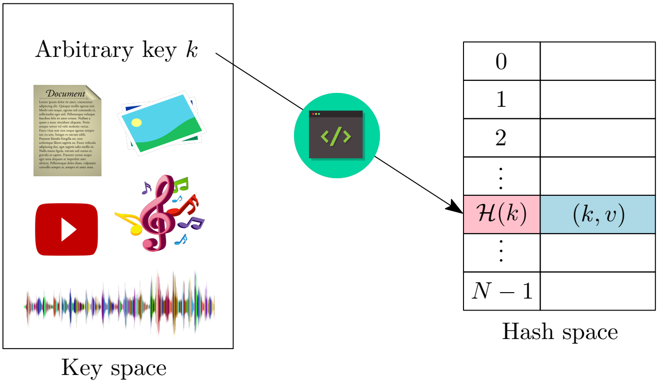
- Balanced search tree is ideal for sorted collection
- Hash table is ideal for unsorted collection

Operations		Balanced tree (worst)	Hash table (avg.)	Hash table (worst)
Sorting-unrelated operations	Insert	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
	Delete	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
	Search	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Sorting-related operations	Sort	$\mathcal{O}(n)$	✗	
	Minimum	$\mathcal{O}(\log n)$	✗	
	Maximum	$\mathcal{O}(\log n)$	✗	
	Predecessor	$\mathcal{O}(\log n)$	✗	
	Successor	$\mathcal{O}(\log n)$	✗	
	Range-Minimum	$\mathcal{O}(\log n)$	✗	
	Range-Maximum	$\mathcal{O}(\log n)$	✗	
	Range-Sum	$\mathcal{O}(n)$	✗	

Hash table


Arbitrary key k
Document
Key space
0
1
2
$\vdots$
$\mathcal{H}(k)$
(k, v)
$\vdots$
$N-1$
Hash space

Hash table



- A hash function is a mapping from arbitrary objects to the set of indices $[0, N-1]$.
- The key-value pair (k, v) in stored at $A[\mathcal{H}(k)]$ in the hash table.

Hash table

Questions

1. How can keys of arbitrary objects be mapped to array indices that are whole numbers?
2. How can an infinite number of keys be mapped to a finite number of indices?
3. What are the pros/cons of hash tables w.r.t. balanced trees?
4. What are the properties of an ideal hash function?
5. Is there one practical hash function that is best for all input?
6. What is the hash function used in Java?
7. Will there be collisions during hashing?
   If yes, how can we avoid collisions?
8. Is there a relation between table size and the number of items?
9. Why (key, value) pairs? Why not tuples?

Encoding of information

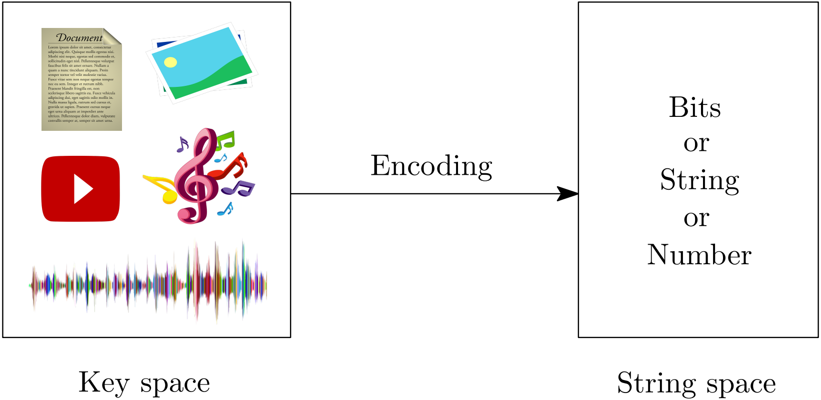
Problem

- How can keys of arbitrary objects be mapped to array indices that are whole numbers?

Encoding of information

Problem

- How can keys of arbitrary objects be mapped to array indices that are whole numbers?



Two stages of hash function

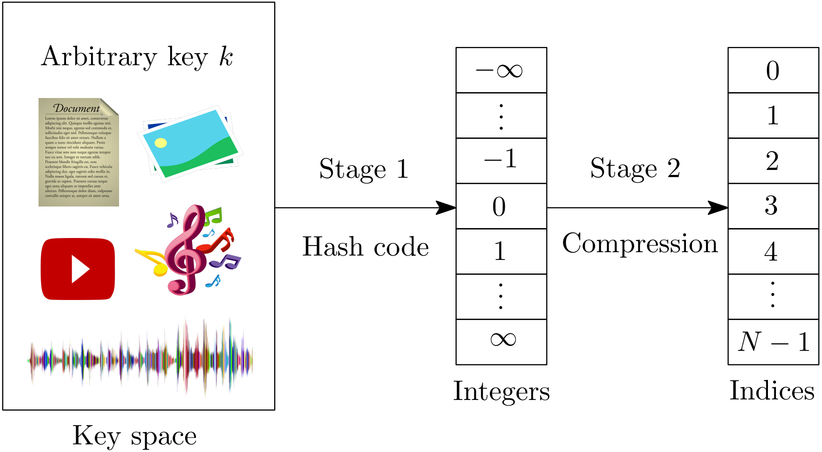
Problem

- How can an infinite number of keys be mapped to a finite number of indices?

Two stages of hash function

Problem

- How can an infinite number of keys be mapped to a finite number of indices?



Stage 1 of hash function: Hash code

- Consider bits as integer.
  Hashcode(byte | short | char) = 32-bit int ▷ upscaling
  Hashcode(float) = 32-bit int ▷ change representation
  Hashcode(double) = 32-bit int ▷ downscaling
  Hashcode$(x_0, x_1, \ldots, x_{n-1}) = x_0 + x_1 + \cdots + x_{n-1}$ ▷ sum
  Hashcode$(x_0, x_1, \ldots, x_{n-1}) = x_0 \oplus x_1 \oplus \cdots \oplus x_{n-1}$ ▷ xor
- Polynomial hash codes.
  Hashcode$(x_0, x_1, \ldots, x_{n-1}) =$
  $\boxed{x_0 a^{n-1} + x_1 a^{n-2} + \cdots + x_{n-2} a + x_{n-1}}$ ▷ polynomial
- Cyclic-shift hash codes.
  Hashcode$_k(x)$ =Rotate$(x, k$ bits) ▷ cyclic-shift
  e.g.: Hashcode$_2(111000) = 100011$

Stage 2 of hash function: Compression function

A good compression function minimizes the number of collisions for a given set of distinct hash codes.

- Division method.

  $\boxed{\text{Compression}(i) = i \ \% \ N}$ $\triangleright$ remainder

  $N \geq 1$ is the size of the bucket array.

  Often, N being prime "spreads out" the distribution of primes.

  Ex. 1: Insert codes $\{200, 205, \ldots, 600\}$ into N-sized array.

  Which is better: $N = 100$ or $N = 101$?

  Ex. 2: Insert multiple codes $\{aN + b\}$ into N-sized array.

  Which is better: $N =$ prime or $N =$ non-prime?
- Multiply-Add-and-Divide (MAD) method.

  $\boxed{\text{Compression}(i) = ((ai + b) \ \% \ p) \ \% \ N}$ $\triangleright$ remainder

  $N \geq 1$ is the size of the bucket array.

  p is a prime number larger than N.

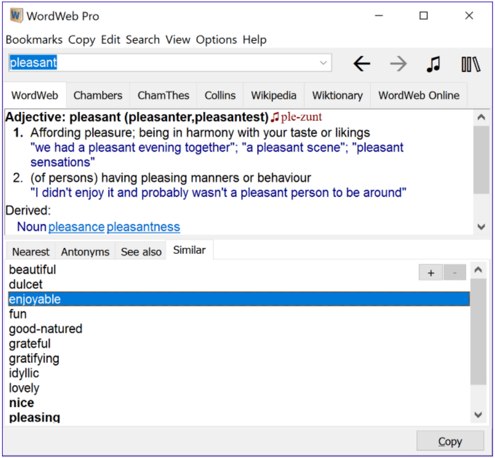
  a, b are random integers from the range $[0, p - 1]$ with $a > 0$.

  Usually eliminates repeated patterns in the set of hash codes.

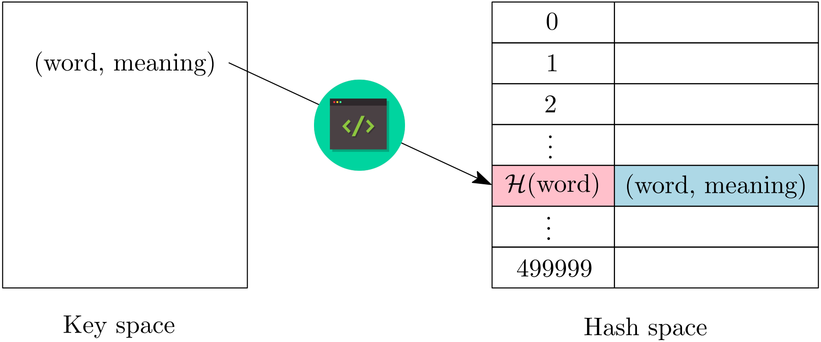
English dictionary

Problem

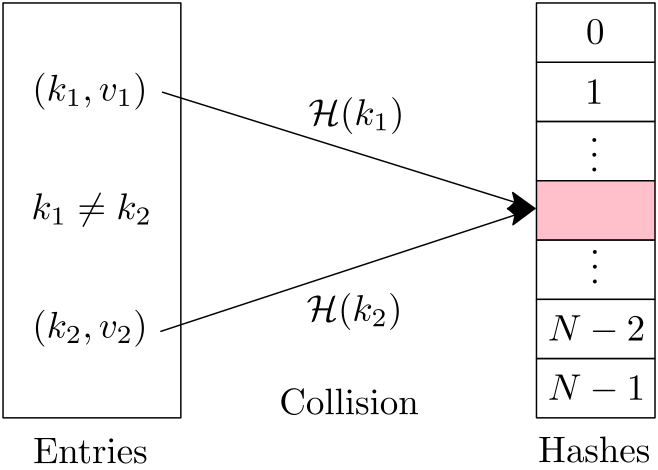
- How do you implement the English dictionary search such that searching for a word takes $\mathcal{O}(1)$ time on an average?



Hash table of English dictionary



Collisions


(k_1, v_1)
$k_1 \neq k_2$
(k_2, v_2)
$\mathcal{H}(k_1)$
$\mathcal{H}(k_2)$
Collision
0
1
$\vdots$
$\vdots$
$N-2$
$N-1$
Entries
Hashes

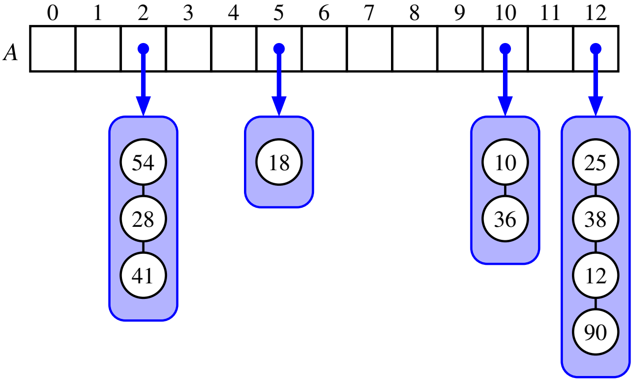
Collision-handling schemes

There are two major collision-handling schemes or collision-resolution strategies.

Collision-handling scheme	Features
Separate chaining	Extra space (for secondary data structures) Simpler implementation
Open addressing	No extra space More complicated implementation

Separate chaining

- Have each bucket $A[j]$ store its own secondary container.
- We use secondary data structures (e.g. array list, linked list, balanced search trees, etc) for each bucket.



Separate chaining (via arraylist/linkedlist)

$\textsc{Put}((key, value))$

1. $hash \leftarrow \textsc{Hash}(key)$
2. $A[hash].\textsc{AddLast}((key, value))$ ▷ $A[hash]$ is a linked list

$\textsc{Get}(key)$

1. $hash \leftarrow \textsc{Hash}(key)$
2. **return** $A[hash].\textsc{Get}(key)$ ▷ returns value

$\textsc{Remove}(key)$

1. $hash \leftarrow \textsc{Hash}(key)$
2. **return** $A[hash].\textsc{Remove}(key)$ ▷ returns removed value

Open addressing

- All entries are stored in the bucket array itself.
- Strict requirement: Load factor must be at most 1.
- Useful in applications where there are space constraints, e.g.: smartphones and other small devices.
- Iteratively search the bucket $\boxed{A[(\textsc{Hash}(key) + f(i)) \ \% \ N]}$ for $i = 0, 1, 2, 3, \ldots$ until finding an empty bucket.

Scheme	Function
Linear probing	$f(i) = i$
Quadratic probing	$f(i) = i^2$
Double hashing	$f(i) = i \cdot \textsc{Hash2}(key)$
	e.g. $\textsc{Hash2}(key) = p - (key \ \% \ p)$ for prime $p < N$.
	Here, N should be a prime number.
Random generator	$f(i) = \textsc{Random}(i, \textsc{Hash}(key))$

Open addressing: Linear probing: Put

- Suppose $\textsc{Hash}(key) = key \ \% \ 10$

Put			Array									
Key	→	Hash	0	1	2	3	4	5	6	7	8	9
18	→	8									18	
41	→	1		41							18	
22	→	2		41	22						18	
32	→	2		41	22						18	
(2 probes)				41	22	32					18	
98	→	8		41	22	32					18	
(2 probes)				41	22	32					18	98
58	→	8		41	22	32					18	98
				41	22	32					18	98
(3 probes)			58	41	22	32					18	98
78	→	8	How many probes are required to insert 78?									

Open addressing: Linear probing: Remove

- Suppose $\textsc{Hash}(key) = key \ \% \ 10$

Remove Key	Array 0	1	2	3	4	5	6	7	8	9
–	58	41	22	32	78	19			18	98
58	58	41	22	32	78	19			18	98
		41	22	32	78	19			18	98
19		41	22	32	78	19			18	98

Hence, we cannot simply remove a found entry.

Remove Key	Array 0	1	2	3	4	5	6	7	8	9
–	58	41	22	32	78	19			18	98
58	58	41	22	32	78	19			18	98
	58	41	22	32	78	19			18	98
19	58	41	22	32	78	19			18	98
	58	41	22	32	78	19			18	98

Replace the deleted entry with the defunct object.

Open addressing: Linear probing

$\textsc{Put}((key, value))$

$hash \leftarrow \textsc{Hash}(key)$; $i \leftarrow 0$
while $(hash + i) \,\%\, N \neq null$ **and** $i < N$ **do** $i \leftarrow i + 1$
if $i = N$ **then throw** Bucket array is full
else $A[(hash + i) \,\%\, N] \leftarrow (key, value)$

$\textsc{Get}(key)$

$hash \leftarrow \textsc{Hash}(key)$; $i \leftarrow 0$
while $(hash + i) \,\%\, N \neq null$ **and** $i < N$ **do**
  $index \leftarrow (hash + i) \,\%\, N$
  if $A[index].key = key$ **then return** $A[index].value$
  $i \leftarrow i + 1$
return $null$

$\textsc{Remove}(key)$

$index \leftarrow \textsc{FindSlotForRemoval}(key)$
if $index < 0$ **then return** $null$
$value \leftarrow A[index].value$; $A[index] \leftarrow defunct$; $n \leftarrow n - 1$
return $value$

Complexity

- Suppose $N =$ bucket array size and $n =$ number of entries.
- Ratio $\lambda = n/N$ is called the load factor of the hash table.
- If $\lambda > 1$, rehash. Make sure $\lambda < 1$.
- Assuming good hash function, expected size of bucket is $\mathcal{O}(\lceil\lambda\rceil)$.
- Separate chaining: Maintain $\lambda < 0.75$
  Open addressing: Maintain $\lambda < 0.5$
- Assuming good hash function and $\lambda \in \mathcal{O}(1)$,
  complexity of put, get, and remove is $\mathcal{O}(1)$ expected time.

Applications HOME

Symbol tables in compilers

```
double foo(int count)
{
  double sum = 0.0;
  for (int i = 1; i <= count; i++)
    sum += i;
  return sum;
}
```

Symbol	Type	Scope
foo	function, double	global
count	int	function parameter
sum	double	block local
i	int	for-loop statement