Algorithms

(Hash Tables)

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

March 19, 2021

Contents

- (Sorted) Sets, Maps, Multisets, Multimaps
- Hash Tables
- Hash Functions
- Collisions
- Collision-Avoiding Techniques
  - Separate Chaining
  - Open Addressing

Dictionary ADTs

These ADTs are collection of items, where,
each item can be a key or a $(key, value)$ pair.

ADT	Item	Ordered?	Duplicates?	Implementation
Set	key	✗	✗	Hash table
Sorted set	key	✓	✗	Balanced tree
Multiset	key	✗	✓	Hash table
Sorted multiset	key	✓	✓	Balanced tree
Map	$(key, value)$	✗	✗	Hash table
Sorted map	$(key, value)$	✓	✗	Balanced tree
Multimap	$(key, value)$	✗	✓	Hash table
Sorted multimap	$(key, value)$	✓	✓	Balanced tree

Set ADT (java.util.Set interface)

Method	Functionality
`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.
`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 = unordered set; Map = unordered map.
  java.util.HashSet is an implementation of the set ADT.
  java.util.HashMap is an implementation of the map ADT.

Sorted set ADT (java.util.SortedSet interface)

Method	Functionality
`first()`	Returns the smallest element in S.
`last()`	Returns the largest element in S.
`ceiling(e)`	Returns the smallest element $\geq e$.
`floor(e)`	Returns the largest element $\leq e$.
`lower(e)`	Returns the largest element $< e$.
`higher(e)`	Returns the smallest element $> e$.
`subSet(e1,e2)`	Returns an iteration of all elements greater than or equal to $e1$, but strictly less than $e2$.
`pollFirst()`	Returns and removes the smallest element in S.
`pollLast()`	Returns and removes the largest element in S.

- java.util.TreeSet is an implementation of the sorted set ADT. java.util.TreeMap is an implementation of the sorted map ADT.

Multiset ADT

Method	Functionality
`add(e)`	Adds a single occurrences of e to the multiset.
`contains(e)`	Returns true if the multiset contains an element $= e$.
`count(e)`	Returns the number of occurrences of e in the multiset.
`remove(e)`	Removes a single occurrence of e from the multiset.
`remove(e, n)`	Removes n occurrences of e from the multiset.
`size()`	Returns the number of elements of the multiset (including duplicates).
`iterator()`	Returns an iteration of all elements of the multiset (repeating those with multiplicity greater than one).

- Java does not include any form of a multiset.
  Guava = Google Core Libraries for Java.
  Guava's Multiset is an implementation of the multiset ADT.
  Guava's Multimap is an implementation of the multimap ADT.
- Similarly, one can define sorted multiset ADT

Hash Tables

Hash tables

- A hash table is an efficient dictionary data structure to implement a set/multiset/map/multimap.
- A hash table performs put, remove, and get operations in constant expected time.
- Hashing is the implementation of hash tables.

Balanced search trees vs. Hash tables

- Balanced search tree $\Leftrightarrow$ sorted, Hash table $\Leftrightarrow$ unsorted
- Worst = worst-case, avg. = expected time (useful in practice)

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	$\mathcal{O}(n)$	✗	

Applications of hash tables

- 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

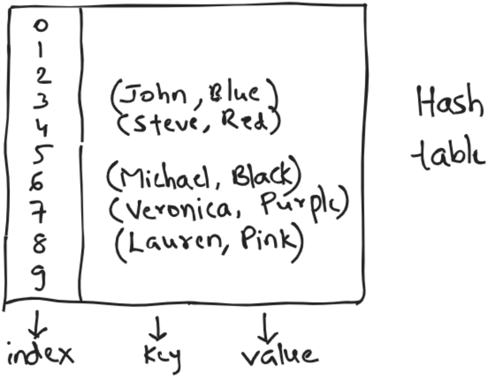
Map

- A map is a collection of key-value pairs (k, v), where, keys are unique.

Key	Value
User ID	User record
Employee ID	Employee record
Student ID	Student record
Patient ID	Patient record
Profile ID	Person details
Order ID	Order details
Transaction ID	Transaction details
URL	Web page
Full file name	File

Hash tables

- A hash table is an efficient implementation of a set or map, i.e., insert, delete, and search operations take constant expected time.
- Example: Suppose we store (name, favorite color) pairs
  We place the key-value pairs in the cells of the hash table array



Hash tables

Questions

- Why do we need to think in terms of (key, value) pairs?
  Why not k-tuples?
- How are keys of arbitrary objects mapped to array indices which are whole numbers?
- A hash table is a data structure of finite size. How can an infinite number of keys be mapped to a finite number of indices?
- Can there be collisions during mapping?
  That is, isn't there a nonzero chance that different keys get mapped to the same index?
- Is there a relation between the table size N and the number of elements n?

Hash Functions

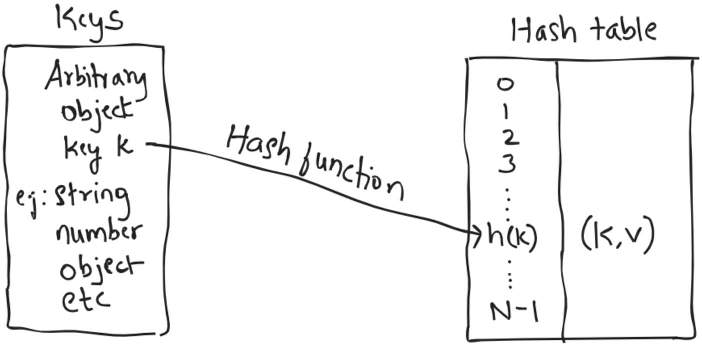
Hash functions

Questions

1. How are keys of arbitrary objects mapped to array indices which are whole numbers?
2. A hash table is a data structure of finite size. How can an infinite number of keys be mapped to a finite number of indices?

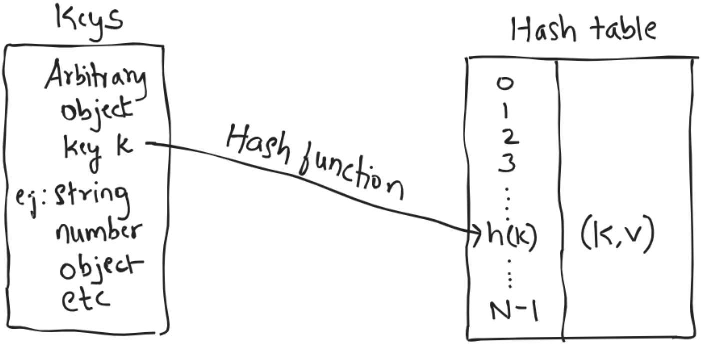
Solution

- Answer: Hash function



Hash functions

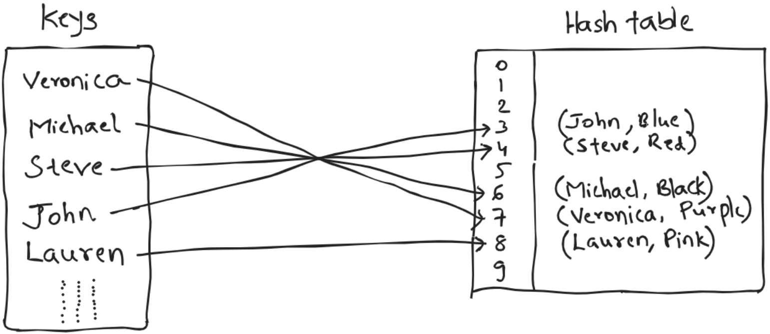
Solution (continued)



- A hash function is a mapping from arbitrary objects to the set of indices $[0, N-1]$.
- A hash function stores key-value pair (k, v) in array $A[h(k)]$.
- A hash function is good when it is easy to compute, fast to compute, and leads to few collisions.

Hash functions

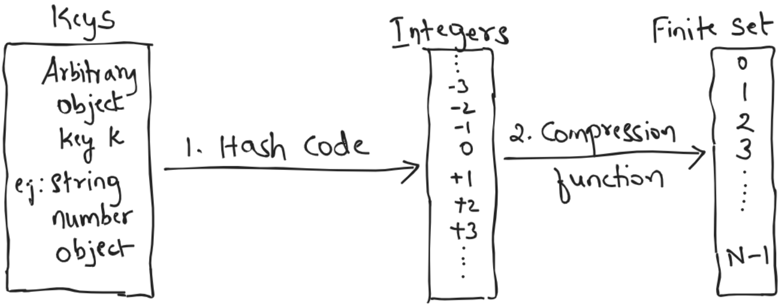
Solution (continued)



- A hash function is a mapping from arbitrary objects to the set of indices $[0, N-1]$.
- A hash function stores key-value pair (k, v) in array $A[h(k)]$.
- A hash function is good when it is easy to compute, fast to compute, and leads to few collisions.

Hash functions

Solution (continued)



- For modularity, assume a hash function consists of two stages:
  1. Hash code
  2. Compression function

  Advantage: The hash code portion of the computation is independent of a specific hash table size

Hash codes

- 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$_2(111000) = 100011$

Compression functions

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.

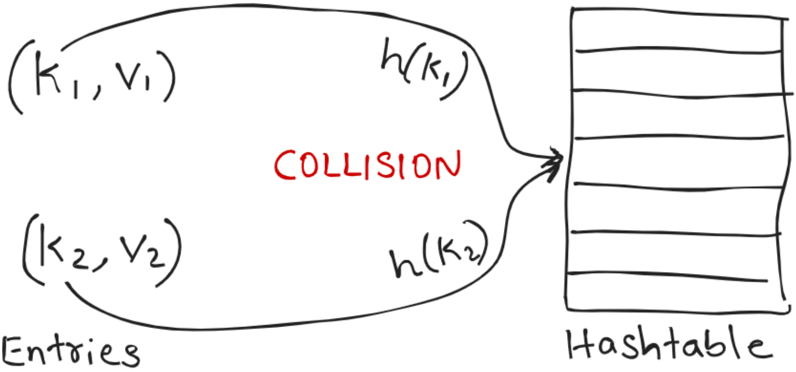
  p is a prime number larger than N.

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

  Usually eliminates repeated patterns in the set of hash codes.

Collisions

Suppose you want to insert two entries (k_1, v_1) and (k_2, v_2) into a hashtable such that $h(k_1) = h(k_2)$. This is called collision as you cannot insert both the entries at the same location.
So, we need to handle collisions.



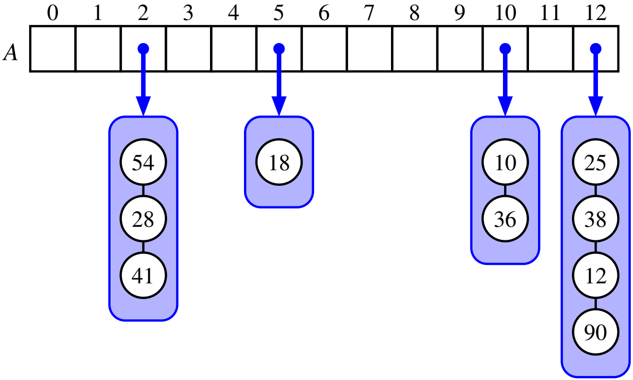
Collision-handling schemes

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. $A[hash].\textsc{AddLast}((key, value))$ $\quad \triangleright$ $A[hash]$ is a linked list

$\textsc{Get}(key)$

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

$\textsc{Remove}(key)$

1. $hash \leftarrow \textsc{Hash}(key)$
2. **return** $A[hash].\textsc{Remove}(key)$ $\quad \triangleright$ 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	$f(i) = \textsc{Random}(i, \textsc{Hash}(key))$

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?									

Linear probing: Remove

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

Remove Key	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	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.

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 $value$

Separate chaining, Open addressing: 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.