

Data Structures and Algorithms

Chapter 10

Maps

- *Map* is a data structure to efficiently store and retrieve values based on *search keys*.
- Map stores (*key*, *value*) pairs.
- Each (*key*, *value*) pair is called an *entry*.
- Keys are unique.
- Maps are also known as *associative arrays*.
- Applications:
 - (movie title, movie information)
 - (part number, part information)
 - (reservation number, reservation information)
 - (student id, student information)

Maps

Map ADT

- `size()`: Returns the number of entries in M .
- `isEmpty()`: Returns true if M is empty. Returns false, otherwise.
- `get(k)`: Returns the value v associated with the key k , if such entry exists. Returns null, otherwise.
- `put(k , v)`: If there is no entry in M with a key equal to k , then adds the entry (k, v) to M and returns null. Otherwise, replaces the existing value associated with the key k with v and returns the old value.

Maps

Map ADT

- `remove(k)`: Removes from M the entry with the key k and returns its value. If there is not entry in M with the key k , returns null.
- `keySet()`: Returns an iterable collection containing all keys in M .
- `values()`: Returns an iterable collection containing all values in M . If multiple keys map to the same value, then the value appears multiple times in the returned collection.
- `entrySet()`: Returns an iterable collection containing all $(key, value)$ entries in M .

Maps

Map ADT

- Map interface

```
1  public interface Map<K,V> {  
2      int size();  
3      boolean isEmpty();  
4      V get(K key);  
5      V put(K key, V value);  
6      V remove(K key);  
7      Iterable<K> keySet();  
8      Iterable<V> values();  
9      Iterable<Entry<K,V>> entrySet();  
10 }
```

- Note: *java.util.Map* interface provides more extensive set of operations than those defined above.

Maps

Map ADT

- Simple application example: Word Frequency
 - Counts frequency of each word in a text.
 - Create an empty map.
 - In the map, an entry is (word, frequency) pair.
 - Read one word at a time.
 - If the word is not in the map, insert it and set frequency = 1
 - If the word is already in the map, increment the frequency of the word.
- *WordCount.java* code.

Maps

Hash Tables

- *Hash table* is an efficient implementation of a map.
- Consider a map that stores n entries.
- Assume keys are integers in the range $[0, N - 1]$ and values are characters, usually $N \geq n$.
- We can design a lookup table of length N as follows, where keys are used as indexes:

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

Lookup table's capacity $N = 11$

Currently there are 4 entries: (1,D), (3,Z), (6,C), and (7,Q)

Maps

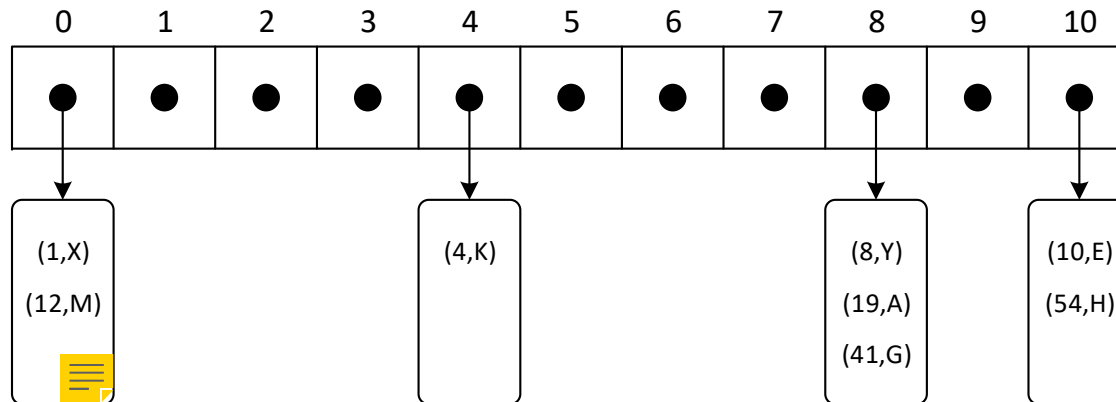
Hash Tables

- Issues:
 - The domain of keys may be much larger than the actual number of elements to be stored in the table, i.e., $N \gg n$. This is a waste of space.
 - Keys may not be integers. Then, they cannot be used as indexes in the table.
- Solution:
 - Use a *hash function* that maps keys to integers in the range $[0, N - 1]$, distributing keys relatively evenly.
 - N doesn't have to be very large (could be smaller).

Maps

Hash Tables

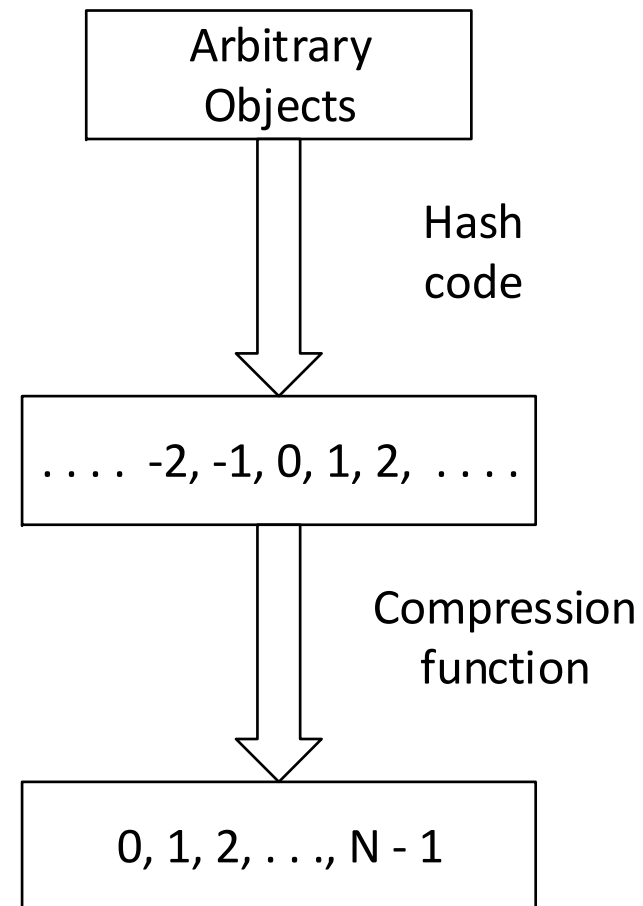
- Ideally a hash function distributes keys evenly across the table.
- In practice, some keys are mapped to the same location.
- One solution: each slot in the table keeps a *bucket* which stores a collection of entries. This table is called *bucket array*.



Maps

Hash Function

- Two step process:
 - *Hash code* maps keys of arbitrary object type to integers. The resulting integer is also called *hash code*.
 - *Compression function* maps the hash code to integers in the range $[0, N - 1]$



Maps

Hash Code

- Treat bit representation of base types as integers
- Polynomial hash code: used for strings or variable-length objects
- Cyclic-shift hash code: a variant of polynomial hash code
- Java has a default *hashCode()* function defined in the *Object* class, which returns a 32-bit integer of *int* type.
- When designing a *hashCode()* for a user-defined class, make sure: If *x.equals(y)*, *x.hashCode() = y.hashCode()*

Maps

Compression Function

- When two keys are mapped to the same hash table index, it is called *collision*.
- A good compression function must distribute hash codes (of keys) relatively uniformly across the hash table to minimize collisions.
- Will discuss two compression functions (compression functions are often called just *hash functions*):
 - *division* method
 - *MAD (multiply-add-and-divide)* method

Maps

Compression Function

- Division method: $i \bmod N$,
where i is an integer (such as a hash code) and N is the hash table size.
- *MAD* method: $[(ai + b) \bmod p] \bmod N$,
where N is hash table size, p is a prime number larger than N , and a and b are integers in $[0, p - 1]$, $a > 0$.

```
private int hashCode(K key) {  
    return (int) ((Math.abs(key.hashCode( )) * scale + shift)  
                  % prime) % capacity);  
}
```


Maps

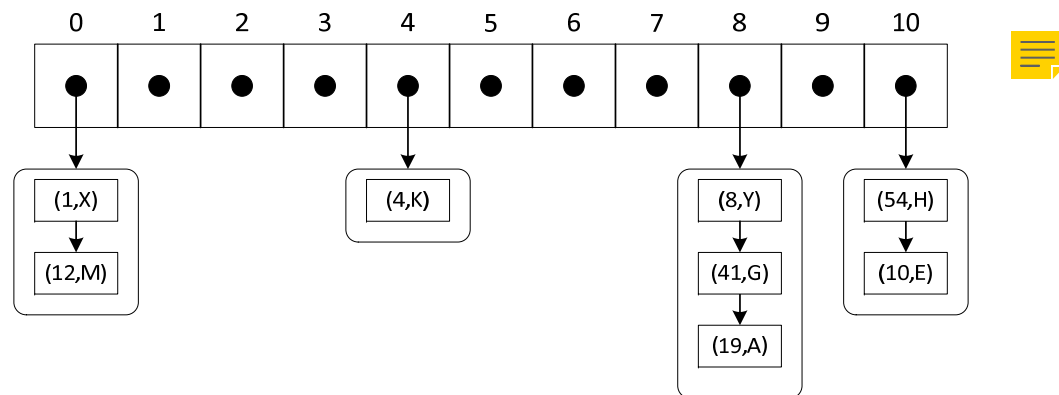
Compression Function

- *MAD* method is better (in terms of well distributing keys across the has table), but division method is more efficient.

Maps

Collision Handling

- When two keys are mapped to the same slot in the hash table, it is called *collision*.
- Will discuss two collision resolution approaches: *chaining* and *open addressing*. 
- Chaining: Each slot in the table keeps an unsorted list and all keys that are mapped to the same slot are kept in the list.



Maps

Chaining Method

- Advantage: Easy to implement
- Drawback:
 - Additional storage
 - In the worst case, all keys are stored in the same list, which increases running time.
- Running time
 - Load factor $\lambda = n / N$, which is expected size of a list.
 - Map operations run in $O(\lceil n / N \rceil)$ or $O(\lambda)$
 - If keys are well distributed, $O(\lambda) = O(1)$ and running time is $O(1)$.
 - In the worst case, $O(n)$.

Maps

Open Addressing

- All entries are stored in a hash table itself.
- No additional data structure and no additional storage space is needed.
- When adding a new key causes a collision, an alternative location in the table is found and the new element is stored in that location.
- Will briefly discuss three open addressing techniques – *linear probing*, *quadratic probing*, and *double hashing*.

Maps

Linear Probing

- Assume A is the array of a hash table.
- Inserting an entry (k, v) .
 - Hash function h is applied to key k , i.e., $j \leftarrow h(k)$. We say k is mapped to j .
 - If $A[j]$ is empty, then the entry is stored there, i.e., $A[j] \leftarrow (k, v)$.
 - If that slot is occupied, the next bucket $A[j+1]$ is *probed* to see whether it is available.
 - If it is empty, the entry is stored there. Otherwise, the next bucket, $A[j+2]$, is probed, and so on, until an empty slot is found or all slots have been probed.
 - The sequence of slots probed, called *probe sequence*, is determined by $A[(j+i) \bmod N]$, for $i = 0, 1, 2, \dots, N-1$.
 - i is called *probe number*.



Maps

Linear Probing

- Illustration: $N = 10$, $h = k \bmod N$, keys are added in the following order: 4, 12, 14, 24.

0	1	2	3	4	5	6	7	8	9
		12		4					

0	1	2	3	4	5	6	7	8	9
		12		4	14				

Diagram showing the insertion of key 14. An arrow points from the label '14' to index 4. A second arrow points from index 4 to index 5, indicating a probe sequence.

0	1	2	3	4	5	6	7	8	9
		12		4	14	24			

Diagram showing the insertion of key 24. An arrow points from the label '24' to index 4. A second arrow points from index 4 to index 5. A third arrow points from index 5 to index 6, indicating a probe sequence.

Maps

Linear Probing

- Searching an entry with key = k .
 - A key k is mapped to the array index j , i.e., $j \leftarrow h(k)$.
 - If $A[j]$ is empty, then conclude the entry is not in the hash table.
 - If that slot is occupied and it has the entry with k , then the entry is found.
 - If the slot is occupied and the key of the entry in the slot is not k , the next bucket, $A[j+1]$, is probed, and so on, until the entry is found or all slots have been probed.

Maps

Linear Probing

- Deleting an entry:
 - Assume initially all slots are empty.
 - Assume we want to remove an entry in $A[j]$.
 - We cannot simply remove the entry in $A[j]$.
 - Assume the current table is:

0	1	2	3	4	5	6	7	8	9
		12		4	14	24			

- And, we delete an entry with key = 14.

Maps

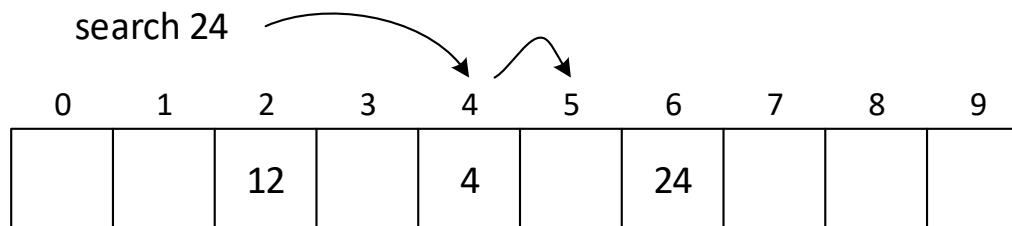
Linear Probing

- After deleting entry with key = 14

0	1	2	3	4	5	6	7	8	9
		12		4		24			

- Search entry with key = 24

24 is mapped to A[4]; occupied; A[5] is probed; empty; conclude entry with key = 24 is not in the table => this is wrong.



Maps

Linear Probing

- Solution: Put a “special object” or a “*defunct*” object in the slot from which an entry is deleted.
- For example, place ϕ in the slot when an entry is removed.
- After removing entry with key = 14

0	1	2	3	4	5	6	7	8	9
		12		4	ϕ	24			

- When inserting, the slot with ϕ is considered empty.
- When searching and entry with key = k , the slot with ϕ is considered having an entry with a key $\neq k$.

Maps

Linear Probing

- Linear probing tends to create *primary clustering*.
- A cluster is a contiguous occupied slots.
- Once a cluster is formed, it tends to grow, which is called *primary clustering*.

Maps

Quadratic Probing

- Uses a quadratic function to determine the next slot to probe.
- Example: Probe sequence is determined by $A[(h(k) + f(i)) \bmod N]$, for $i = 0, 1, 2, \dots, N - 1$, where $f(i) = i^2$
- Assume that we are inserting a key 24 and it is mapped to $A[4]$, and that it is occupied. Then, the probe sequence is:
 - $A[(4 + 1^2) \bmod 10] = A[5],$
 - $A[(4 + 2^2) \bmod 10] = A[8],$
 - $A[(4 + 3^2) \bmod 10] = A[3],$
 - \dots

Maps

Quadratic Probing

- Quadratic hashing does not have primary clustering.
- But, it still has clustering problem, which is called secondary clustering.
- There are quadratic probing methods that use different quadratic functions.

Maps

Double Hashing

- Does not cause serious clustering problem.
- Uses two hash functions.
- Probe sequence is determined by
 $A[(h(k) + i \cdot h'(k)) \bmod N]$, for $i = 0, 1, 2, \dots, N - 1$
- One common secondary hash function h' is:
 $h'(k) = q - (k \bmod q)$, for some prime number $q < N$, N is prime
- Another common h' is:
 $h'(k) = 1 + (k \bmod N')$, where N' is slightly smaller than N , N is prime

Maps

Double Hashing

- Example (of the second h')

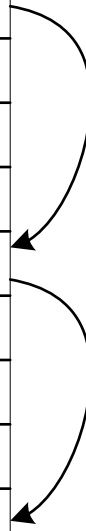
$$h(k) = k \bmod 13$$

$$h'(k) = 1 + (k \bmod 11)$$

$$h(k, i) = (h(k) + i * h'(k)) \bmod N$$

- Inserting $k = 14$, $h(k) = 1$, $h'(k) = 4$
- $h(14) = 1$, occupied
- $i = 1$: $1 + 4 = 5$, occupied
- $i = 2$: $1 + 8 = 9$, empty, store 14 here

0	
1	79
2	
3	
4	69
5	98
6	
7	59
8	
9	14
10	
11	37
12	



Maps

Load Factor and Efficiency

- Load factor is defined as $\lambda = n / N$
- A larger value of λ means there is higher probability of collisions.
- So, a smaller λ is better.
- With chaining method, λ could be greater than 1.
- With open addressing, $\lambda \leq 1$.
- Performance of chaining method:
 - A theoretical analysis shows that the average number of entries that need to be probed for a successful search is approximately $1 + \frac{\lambda}{2}$.

Maps

Load Factor and Efficiency

- Performance of chaining method (continued):
 - Let C be the average number of entries that need to be probed for a successful search.

λ	C
0.5	1.25
0.7	1.35
1.0	1.5
2.0	2

- Java uses chaining method and λ is set to 0.75 or less by default.

Maps

Load Factor and Efficiency

- Performance of double hashing:
 - The average number of slots that need to be probed for a successful search is approximately $\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$
 - Let C be the average number of slots that need to be probed for a successful search.

λ	C
0.3	1.19
0.5	1.39
0.7	1.72
0.9	2.56

References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, “Data Structures and Algorithms in Java,” Sixth Edition, Wiley, 2014.