Maps and Hashing

Chapter 8

Chapter Objectives

- To understand the Java Map and Set interfaces and how to use them
- To learn about hash coding and its use to facilitate efficient search and retrieval
- To study two forms of hash tables—open addressing and chaining—and to understand their relative benefits and performance tradeoffs

Chapter Objectives (continued)

- To learn how to implement both hash table forms
- To be introduced to the implementation of Maps and Sets
- To see how two earlier applications can be more easily implemented using Map objects for data storage

The Set Abstraction

- A set is a collection that contains no duplicate elements
- Operations on sets include:
  - Testing for membership
  - Adding elements
  - Removing elements
  - Union
  - Intersection
  - Difference
  - Subset

The Set Interface and Methods

- Has required methods for testing set membership, testing for an empty set, determining set size, and creating an iterator over the set
- Two optional methods for adding an element and removing an element
- Constructors enforce the no duplicate members criterion
- Add method does not allow duplicate items to be inserted

Maps and the Map Interface

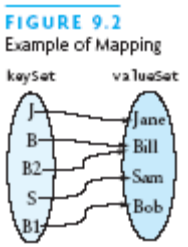


FIGURE 9.2
Example of Mapping

- The Map is related to the Set
- Mathematically, a Map is a set of ordered pairs whose elements are known as the key and the value
- Keys are required to be unique, but values are not necessarily unique
- You can think of each key as a “mapping” to a particular value
- A map can be used to enable efficient storage and retrieval of information in a table

Maps and the Map Interface (continued)

TABLE 9.2

Some `java.util.Map<K, V>` Methods

Method	Behavior
`V get(Object key)`	Returns the value associated with the specified key. Returns **`null`** if the key is not present.
`boolean isEmpty()`	Returns **`true`** if this map contains no key-value mappings.
`V put(K key, V value)`	Associates the specified `value` with the specified `key` in this map (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`.
`V remove(Object key)`	Removes the mapping for this `key` from this map if it is present (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`.
`int size()`	Returns the number of key-value mappings in this map.

Hash Tables

- The goal behind the hash table is to be able to access an entry based on its key value, not its location
- We want to be able to access an element directly through its key value, rather than having to determine its location first by searching for the key value in the array
- Using a hash table enables us to retrieve an item in constant time (on average) linear (worst case)

Hash Codes and Index Calculation

- The basis of hashing is to transform the item's key value into an integer value which will then be transformed into a table index

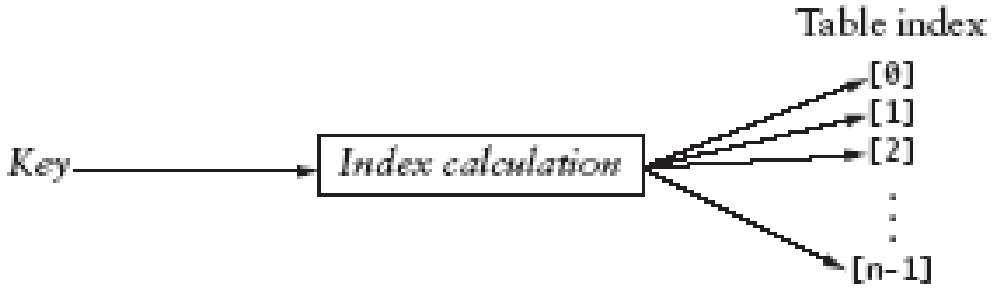


FIGURE 9.4
Index Calculation for a Key

Methods for Generating Hash Codes

- In most applications, the keys will consist of strings of letters or digits rather than a single character
- The number of possible key values is much larger than the table size
- Generating good hash codes is somewhat of an experimental process
- Besides a random distribution of its values, the hash function should be relatively simple and efficient to compute

Java HashCode Method

- For strings, simply summing the int values of all characters will return the same hash code for sign and sing
- The Java API algorithm accounts for position of the characters as well
- The String.hashCode() returns the integer calculated by the formula: $s_0 \times 31^{(n-1)} + s_1 \times 31^{(n-2)} + \ldots + s_{n-1}$

  where s_i is the *i*th character of the string, and *n* is the length of the string
- "Cat" will have a hash code of: 'C' x 31^2 + 'a' x 31 + 't'
- 31 is a prime number that generates relatively few collisons

Open Addressing

- Consider two ways to organize hash tables
  - Open addressing
  - Chaining
- Linear probing can be used to access an item in a hash table
  - If that element contains an item with a different key, increment the index by one
  - Keep incrementing until you find the key or a null entry

Open Addressing (continued)

Algorithm for Accessing an Item in a Hash Table

1. Compute the index by taking the item's `hashCode() % table.length`.
2. `if table[index]` is `null`
3. The item is not in the table.
4. `else if table[index]` is equal to the item
5. The item is in the table.

   `else`
6. Continue to search the table by incrementing the index until either the item is found or a `null` entry is found.

Table Wraparound and Search Termination

- As you increment the table index, your table should wrap around as in a circular array
- Leads to the potential of an infinite loop
- How do you know when to stop searching if the table is full and you have not found the correct value?
  - Stop when the index value for the next probe is the same as the hash code value for the object
  - Ensure that the table is never full by increasing its size after an insertion if its occupancy rate exceeds a specified threshold

Hash Table Considerations

- You cannot traverse a hash table in a meaningful way as the sequence of stored values is arbitrary
- When an item is deleted, you cannot just set its table entry to null
  - Store a dummy value instead
  - Deleted items waste storage space and reduce search efficiency
- Use a prime number for the size of the table so as to reduce collisions
- A fuller table will result in increased collisions

Reducing Collisions Using Quadratic Probing

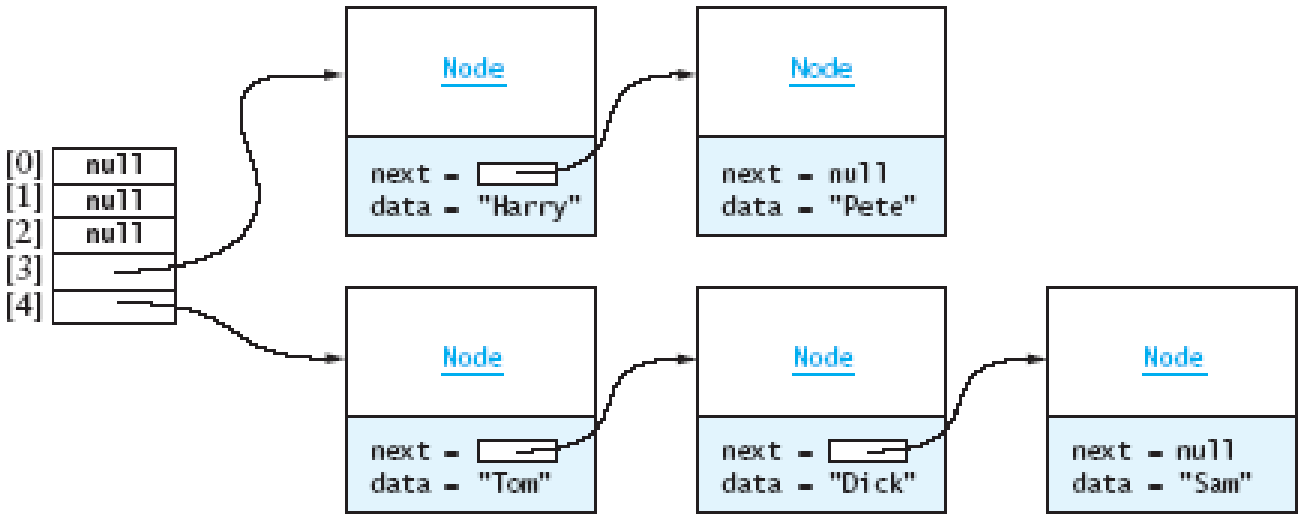
- Linear probing tends to form clusters of keys in the table, causing longer search chains
- Quadratic probing can reduce the effect of clustering
  - Increments form a quadratic series
  - Disadvantage is that the next index calculation is time consuming as it involves multiplication, addition, and modulo division
  - Not all table elements are examined when looking for an insertion index

Chaining

- Chaining is an alternative to open addressing
- Each table element references a linked list that contains all of the items that hash to the same table index
  - The linked list is often called a bucket
  - The approach sometimes called bucket hashing
- Only items that have the same value for their hash codes will be examined when looking for an object

Chaining (continued)

FIGURE 9.7
Example of Chaining



Performance of Hash Tables

- Load factor is the number of filled cells divided by the table size
- Load factor has the greatest effect on hash table performance
- The lower the load factor, the better the performance as there is a lesser chance of collision when a table is sparsely populated

Performance of Hash Tables (continued)

TABLE 9.4
Number of Probes for Different Values of Load Factor (L)

L	Number of Probes with Linear Probing	Number of Probes with Chaining
0	1.00	1.00
0.25	1.17	1.13
0.5	1.50	1.25
0.75	2.50	1.38
0.85	3.83	1.43
0.9	5.50	1.45
0.95	10.50	1.48

Implementing a Hash Table

TABLE 9.5
Interface KWHashMap<K, V>

Method	Behavior
`V get(Object key)`	Returns the value associated with the specified key. Returns **null** if the key is not present.
`boolean isEmpty()`	Returns **true** if this table contains no key-value mappings.
`V put(K key, V value)`	Associates the specified value with the specified key. Returns the previous value associated with the specified key, or **null** if there was no mapping for the key.
`V remove(Object key)`	Removes the mapping for this key from this table if it is present (optional operation). Returns the previous value associated with the specified key, or **null** if there was no mapping.
`int size()`	Returns the size of the table.

TABLE 9.6
The Inner Class Entry<K, V>

Data Field	Attribute
`private K key`	The key.
`private V value`	The value.
Constructor	**Behavior**
`public Entry(K key, V value)`	Constructs an Entry with the given values.

TABLE 9.6 (continued)

Method	Behavior
`public K getKey()`	Retrieves the key.
`public V getValue()`	Retrieves the value.
`public V setValue(V val)`	Sets the value.

Implementing a Hash Table (continued)

TABLE 9.7
Data Fields for Class `HashtableOpen<K, V>`

Data Field	Attribute
`private Entry<K, V>[] table`	The hash table array.
`private static final int START_CAPACITY`	The initial capacity.
`private double LOAD_THRESHOLD`	The maximum load factor.
`private int numKeys`	The number of keys in the table excluding keys that were deleted.
`private int numDeletes`	The number of deleted keys.
`private final Entry<K, V> DELETED`	A special object to indicate that an entry has been deleted.

TABLE 9.8
Private Methods for Class `HashtableOpen`

Method	Behavior
`private int find(Object key)`	Returns the index of the specified key if present in the table; otherwise, returns the index of the first available slot.
`private void rehash()`	Doubles the capacity of the table and permanently removes deleted items.

TABLE 9.9
Data Fields for Class `HashtableChain<K, V>`

Data Field	Attribute
`private LinkedList<Entry<K, V>>[] table`	A table of references to linked lists of `Entry<K, V>` objects.
`private int numKeys`	The number of keys (entries) in the table.
`private static final int CAPACITY`	The size of the table.
`private static final int LOAD_THRESHOLD`	The maximum load factor.

Implementation Considerations for Maps and Sets

- Class Object implements methods hashCode and equals, so every class can access these methods unless it overrides them
- Object.equals compares two objects based on their addresses, not their contents
- Object.hashCode calculates an object's hash code based on its address, not its contents
- Java recommends that if you override the equals method, then you should also override the hashCode method

Implementing HashSetOpen

TABLE 9.10
Corresponding Map and Set Methods

Map Method	Set Method
`Object get(Object key)`	`boolean contains(Object key)`
`Object put(Object key, Object value)`	`boolean add(Object key)`
`Object remove(Object key)`	`boolean remove(Object key)`

Implementing the Java Map and Set Interfaces

- The Java API uses a hash table to implement both the Map and Set interfaces
- The task of implementing the two interfaces is simplified by the inclusion of abstract classes AbstractMap and AbstractSet in the Collection hierarchy

Chapter Review

- The Set interface describes an abstract data type that supports the same operations as a mathematical set
- The Map interface describes an abstract data type that enables a user to access information corresponding to a specified key
- A hash table uses hashing to transform an item's key into a table index so that insertions, retrievals, and deletions can be performed in expected O(1) time
- A collision occurs when two keys map to the same table index
- In open addressing, linear probing is often used to resolve collisions

Chapter Review (continued)

- The best way to avoid collisions is to keep the table load factor relatively low by rehashing when the load factor reaches a value such as 0.75
- In open addressing, you can’t remove an element from the table when you delete it, but you must mark it as deleted
- A set view of a hash table can be obtained through method entrySet
- Two Java API implementations of the Map (Set) interface are HashMap (HashSet) and TreeMap (TreeSet)