CSE 12
The Map Abstract Data Type

• The Map ADT
• Implementations of the Map ADT
• Hashing and Hash Tables
• Collisions and Collision Resolution Strategies
The Map ADT

- The ADT's we have talked about so far are container structures intended to hold data of a certain type.
- The ADT’s operations deal with objects of that type, sometimes called *keys*: adding them, removing them, checking if the data structure contains them, iterating over them, etc.
- The Map ADT has a slightly different emphasis: it is intended to hold *pairs* of data of certain types.
- Map operations deal with <key, value> pairs: adding a pair, removing a pair given its key, returning the value of a pair given its key, iterating over the keys or values or pairs, etc.
- Map is also sometimes known as: Table, Dictionary, Associative Memory.
Examples of Map Applications

• dictionary – a set of words, each one associated with its definition: \(<word,definition>\) pairs
• book index – collection of key words, each one associated with the page numbers on which the word appears: \(<keyword,list-of-pages>\) pairs
• symbol table – a compiler data structure associating identifiers with declaration information: \(<identifier,declaration-information>\) pairs

...Can you think of others?

A **map** is a collection type that stores \(<key, value>\) pairs. A value associated with a key is retrieved by supplying the map with the key.
Map Description, Properties & Attributes

Description
A map stores <key, value> pairs. Given a key, a map provides the value associated with that key. The types of the key and value are such that it must be possible to test for equality among keys and among values.

Properties
1. Duplicate keys are not allowed.
2. A key may be associated with only one value.
3. A value may be associated with more than one key.
4. Keys can be compared to one another for equality; similarly for values.
5. Null keys and values are not allowed.

Attributes
size: The number of <key, value> pairs in this map.
Map Operations

Map()
pre-condition: none
responsibilities: constructor—create an empty map
post-condition: size is set to 0
returns: nothing

put( KeyType key, ValueType value )
pre-condition: key and value are not null
key can be compared for equality to other keys in this map
value can be compared for equality to other values in this map
responsibilities: puts the <key, value> pair into the map. If key already exists
in this map, its value is replaced with the new value
post-condition: size is increased by one if key was not already in this map
returns: null if key was not already in this map, the old value
associated with key otherwise
exception: if key or value is null or cannot be compared to keys/values in this map
Map Operations

get( KeyType key )
pre-condition: key is not null and can be compared for equality to other keys in this map
responsibilities: gets the value associated with key in this map
post-condition: the map is unchanged
returns: null if key was not found in this map, the value associated with key otherwise
exception: if key is null or cannot be compared for equality to other keys in this map

remove( KeyType key )
pre-condition: key is not null and can be compared for equality to other keys in this map
responsibilities: remove the value from this map associated with key
post-condition: size is decreased by one if key was found in this map
returns: null if key was not found in this map, the value associated with key otherwise
exception: if key is null or cannot be compared for equality to other keys in this map
containsValue( ValueType value )
pre-condition: value is not null and can be compared for equality to other values in this map
responsibilities: determines if this map contains an entry containing value
post-condition: the map is unchanged
returns: true if value was found in this map, false otherwise
exception: if value is null or cannot be compared for equality to other values in this map

containsKey( KeyType key )
pre-condition: key is not null and can be compared to other keys in this map
responsibilities: determines if this map contains an entry with the given key
post-condition: the map is unchanged
returns: true if key was found in this map, false otherwise
exception: if key is null or cannot be compared for equality to other keys in this map
Map Operations

values()

pre-condition: none
responsibilities: provides a Collection view of all the values contained in this map. The returned Collection supports element removal, but not element addition. A removal made to the Collection is reflected in this map and vice versa
post-condition: the map is unchanged
returns: a Collection providing a view of the values from this map

A Collection view is a different way to “view” the values stored in the map
A Test Plan for Map

- As always: use predicate and accessor methods to verify state changes caused by mutators

- Some examples:
  - put a ⟨key, value⟩ pair in an empty map; given the key, should get the value back
  - put a ⟨key, value⟩ pair in an empty map, then put in another pair using the same key but a different value; the original value should be returned. size should be 1
  - put a ⟨key, value⟩ pair in an empty map, then put in another pair using the same value but a different key. The map should regard these as separate entries, so size should be 2. contains() should verify that the map contains the two keys and the value, and get() on the two keys should return equal values
Implementing the Map ADT

• As you know, any precisely defined ADT can be implemented in many different ways

• For example, could use a linked list to implement Map
  – Define a class Entry<K,V> that has 2 instance variables: one of type K, one of type V
  – Linked list nodes will contain data of that type:
    ```java
    public class LinkedNode<Entry<K,V>>
    ```

• But now put() in a linked list of length \( n \) takes time \( O(n) \), and implementing get() also takes time \( O(n) \) in the worst case

• We would like to do better than that! Consider using a hash table to implement Map
Implementations of the Map interface in the JCF
Hashing and Hash Tables - Motivation

- **Problem**: Indexed access in an array is fast $O(1)$, but only if you know the index! Not knowing a key’s index in advance, you have to search for the key
  - searching is $O(\log_2 n)$ if array elements are comparable and the array is sorted; otherwise $O(n)$

- **Idea**: What if we could use the key itself to determine its index in the array?
  - Use a fast $O(1)$ *hash function* which takes as input a key, and returns the array index for that key
  - If this works, then we will have overall search time cost $O(1)$, which is very good
Hashing and Hash Tables

- A **hash table** is an indexed collection (usually implemented using an array)

- Each indexed location in the hash table is called a **bucket** and can hold one (and possibly more) entries; each entry holds a key-value pair

- There is a **hash function** that takes a key, and returns an index that is used to index a bucket in the hash table.
Hashing and Hash Tables - Complications

- The Good: If the hash function returns a different index for each key, then basic hash table operations (insertion, removal, and retrieval) will have time cost O(1)

- The Bad: To guarantee a unique bucket for each key, the table would have to have as many buckets as there are possible keys
  - Since the set of possible keys may be quite large (think of English words, or people's names, or 32-bit integers, etc.), this is not practical in general:
  - The table would be huge, and (since the number of actual keys stored in the table is typically much smaller than the number of possible keys) would waste a lot of space
Design the hash table to have \( m \) buckets, with \( m \) only slightly larger than the number of actual keys in the table (much smaller than the number of possible keys), so space is used efficiently.

Allow the hash function to hash more than one key to the same bucket.

More than one key hashing to the same bucket is called a collision.

This raises another complication: now we need a strategy for dealing with collisions.

Two classes of strategies: open addressing, closed addressing.
Collision Resolution: open addressing

- **open addressing**
  - also known as “closed hashing”
  - each bucket can contain only a single entry
  - when a collision occurs, we must find a bucket at another index in the table to store the entry
  - we “open up the addressing”, and allow the entry to be stored in a bucket *other than* the one to which it originally hashed
  - example: linear probing, which uses a simple linear search for an empty bucket
Collision Resolution: closed addressing

• **closed addressing**
  – an entry *must* be stored at in the bucket to which it originally hashed
  – So we allow the bucket capacity to be greater than 1; thus each bucket is itself a collection (hopefully small)
  – example: separate chaining, which uses a simple singly-linked list as the collection
Hashing examples

• In the following examples, keys are Strings, and the hash function hash(String s) is defined as:

  *the position in alphabet of first letter in s*

  – (this is not really a very good hash function for Strings… it's just for these examples)

• We will look at:
  – open addressing and closed addressing strategies
  – the basic insert, find, and remove algorithms
Open Addressing: Linear Probing

• A collision at bucket indexed $i$ means another entry is already there, and a different bucket must be found for the new entry.

  – Idea: perform a linear search for an empty bucket

  – Try the bucket at $(i + 1)$; if that is occupied, try $(i + 2)$ and so on, wrapping around to index 0 when the table end is reached

  – This will always find an empty bucket, if there is one

  – This strategy is called linear probing
Linear Probing - inserting

(a) Inserting “Bill” – there is no collision
(b) Inserting “Boris” – there is a collision at position 1
(c) Inserting “Bing” – there are collisions at positions 1 and 2
(d) Inserting “Carol” then “Dora” grows the cluster

*clustering* – keys occupying adjacent locations in the table. During an insert, a key that hashes to any location in a cluster will grow the cluster; the cluster acts like a linked list.
Linear Probing - searching

- Hash to key’s hash position, then do a linear search until either the key is found (success) or an empty bucket is found (failure)

(a) Finding “Bing” requires three probes before succeeding

(b) Finding “Betsy” requires six probes before failing
Linear Probing - removing

- Remember how a search is done?
- Clustering complicates things if a remove operation just deletes an entry, since it can cause a gap in entries that were forced to be adjacent due to collisions when inserting

(a) With “Bing” deleted, there is a gap at position 3, so a search probe beginning in position 2 or 3 fails when it encounters the gap

(b) With a bridge in the place “Bing” occupied, the probe correctly advances to position 4
Closed Addressing: Separate Chaining

• With Open Addressing we allowed an entry to be stored in a bucket other than the one to which it hashes.

• With Closed Addressing we require that an entry only be stored at the bucket to which it hashes.

• This in turn requires that buckets be allowed to store more than one entry.

• How to do that?...
Separate Chaining – insert, find, delete

- Solution: each bucket in the table is a pointer to a data structure which holds key,value pairs
- Then insert, find, delete is implemented as:
  - Apply hash function to key to determine bucket
  - Perform insert, find, or delete on the data structure at that bucket
- When these data structures are singly-linked lists, this strategy is called separate chaining
Next time

• Hash table time costs
• Hash functions
• The Map<K,V> interface and implementations

Reading:  : Gray, Ch 12