Dictionaries
Hashtables and Hashsearch

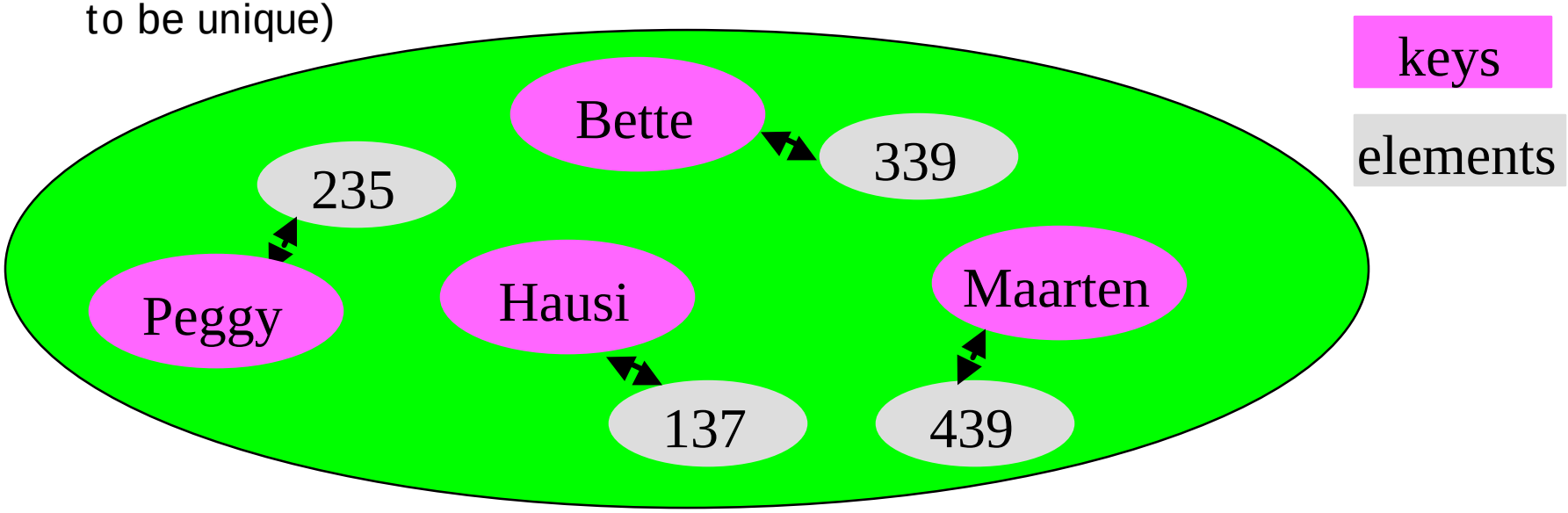
Reading Assignment
Chapter 8.1-8.3

Dictionaries….

- Many, many examples in our everyday life of dictionaries….
- The primary purpose is to look things up using some key…. The motivation being is that there is some additional information to the key that we would find useful
- E.g. account number in our bank, or a dictionary might hold a set of windows open in a graphical interface
- Like a priority queue, a dictionary if a container of key-element pairs – but a total order relation on the keys is always required for the priority queue, but not for the dictionary
- The simplest form of a dictionary only assumes that we can compare two elements to see if they are equal
- For an ordered dictionary, we will have additional methods defined
- Computer dictionaries are more powerful than paper dictionaries, why?
- So we need at least methods for inserting, removing and searching for elements using their keys

Dictionary

- A dictionary is an unordered container that contains key-element pairs
- The keys are unique, but the elements can be anything (e.g., don't have to be unique)



Search key	Associated element
Bette	339
Hausi	137
Peggy	235
Maarten	439

Dictionary ADT

- size(): returns the number of items in D. Output: Integer
- isEmpty(): Test whether D is empty. Output: Boolean
- elements(): Return the elements stored in D. Output: iterator of elements (objects)
- keys(): Return the keys stored in D. Output: iterator of keys (objects)
- findElement(k): if D contains an item with key == *k*, then return the element of that item, else return NO_SUCH_KEY. Output: Object
- findAllElements(k): Output: Iterator of elements with key *k*
- insertItem(k,e): Insert an Item with element *e* and key *k* into D.
- removeElement(k): Remove an item with key == k and return it. If no such element, return NO_SUCH_KEY Output: Object (element)
- removeAllElements(k): Remove from D the items with key == *k*. Output: iterator of elements.

Dictionary Interface

```
public interface Dictionary {
  public void insert(Object key, Object element);
  public Object member(Object key);
  public Object delete(Object key);
  public Enumeration keys();
  public Enumeration elements();
  boolean isEmpty();
  int size();
}
```

- To implement a generic interface, we also need to compare the search key against the keys in the dictionary
- Provide an **`equals()`** routine and a **`Comparable interface`**

```
int equals(Object x)
interface Comparable {…} --- optional, only if the dictionary
                                          is ordered.
```

See: http://java.sun.com/j2se/1.4.1/docs/api/java/lang/Comparable.html

Abstract Class java.util.Dictionary

- The Dictionary class is the abstract parent of any class, such as Hashtable, which maps keys to elements
- Any non-null object can be used as a key and as an element
- As a rule, the equals method should be used by implementations of this class to decide if two keys are the same.

```
public abstract class java.util.Dictionary {
  abstract void put(Object k, Object v); // inserts key & value
  abstract Object get(Object k); // returns value for this key
  abstract Object remove(Object k); // removes key & its element
  abstract Enumeration elements(); // returns all elements
  abstract Enumeration keys(); // returns enumeration of all keys
  boolean isEmpty(); // returns true if dictionary is empty
  abstract int size(); // returns number of keys in dictionary
}
```

This abstract class is now obsolete….

Interface -- java.util.Map

- Keys must be unique in the implementing classes of Map
- The method equals will be used to determine if two keys are equal – therefore we must hard-code the equals method within the key class

Method Summary

boolean **containsKey**(Object key) -- Returns true if this map contains a mapping for the specified key.

boolean **containsValue**(Object value) -- Returns true if this map maps one or more keys to the specified value.

Object **get**(Object key) -- Returns the value to which this map maps the specified key.

boolean **isEmpty**() -- Returns true if this map contains no key-value mappings.

Set**keySet**() -- Returns a set view of the keys contained in this map.

Object **remove**(Object key) -- Removes the mapping for this key from this map if it is present (optional operation).

int **size**() Returns the number of key-value mappings in this map.

Interface -- java.util.Map (2)

- Sentinel value? How is this done for get(k)?

- Returns null if the map contains no mapping for this key.
- But a return value of null does not *necessarily* indicate that the map contains no mapping for the key; it's also possible that the map explicitly maps the key to null.
- The `Boolean containsKey(Object key)` operation may be used to distinguish these two cases…

Implementing Dictionaries.....

- We will look at:
  - Log Files (see Section 8.2 in the book) and
  - Hash Tables (see Section 8.3 in the book)

Log Files

- We could implement a dictionary using an unordered vector, list or general sequence to store key-element pairs
- Often referred to as a log file or audit file
- If the log file is
- Each new pair is added to the end of the log file (O(1) cost to add a new pair)
- A find could require O(n) operations in the worst case
- How much space is required?
- What are some suitable applications for this implementation?

- But for applications that have roughly the same number of finds as insertions – this implementation would not be suitable!

Hash Tables

- In a hash table, we consider the key of an element its address
- Worst case for finding an element in a hash table can still be O(n) – but in the expected case it could be as good as O(1)… (we will come back to this….)

- Two major parts of a Hash Table:
  - Bucket Arrays (where we put stuff)
  - Hash Functions (how we know where to put and find stuff)

Bucket Arrays

- A *bucket* is a container for each key-element pair
- A *bucket array* for a hash table is an Array A of size N, where each cell of A is thought of as a "bucket"
- An element *e* with key *k* is simply inserted into the bucket A[*k*]
- Any bucket cells associated with keys not present in the dictionary hold the special NO_SUCH_KEY object

Bucket Arrays -- 2

- Problems with this?
- Well, if keys are not unique, then >1 element may be mapped to the same bucket causing a *collision…* (we will come back to this….)

- Assuming keys are unique – searches, insertions and removals take how long?
- But how much space does it use?
- N is not necessarily related to *n*, the number of items in the dictionary

- ….. So we need a good mapping from our keys to integers in the range [0,N-1]

Hash Functions

- The hash function *h* maps each key *k* in our dictionary to an integer in the range [0, N-1] – where N is the capacity of the bucket array
- Then we can use *h* (*k*) as an index into the bucket array instead of key *k* – item (*k*,*e*) is stored in A[*h* (*k*)]
- A hash function is *good* if –
  - Collisions are minimized as much as possible
  - The evaluation of the hashing function is fast and easy to compute

Hash Functions -- 2

- The hashing function consists of two actions –
  - Mapping the key *k* to an integer – which we call the *hash code*
  - And then mapping the hash code to an integer within the rage of indices of the bucket array – we call this the *compression map*

Hash Codes…

- Take our arbitrary key k in our dictionary and assign it an integer value – does not have to be in the range [0,N-1] or even be a positive integer
- But the set of hash codes should be as unique as possible to avoid collisions
- Note the hash codes should be the same for the keys that are the same (to enable equality testing)

Generating Hash Codes

- The Object class in Java has a default hashCode() method, but we usually need to override it (could just a representation of the object's location in memory)
- For data types that have the same number as bits as an integer, simply map all the bits to an integer representation
  - For byte, short, int and char – cast to int
  - For float, convert to an integer by calling Float.floatToIntBits(x)
- For longer types, we can take the low-order bits and sum all the high order bits and add it to the low-order bits:

```
int hashCode(long i)
          { return (int)((i > > 32) + (int)i);}
```

String Hash Function: An Example

- Let
  - *s* be a key of type **String**
  - sum be the sum of the ordinal values of all the characters in s
  - N be the hashtable size
- Then the hashtable index *k* is
  - *k* = sum % N
    - where **%** is the modulo operator.
    - Thus, ***k*** is in the range **0** to **N-1**
- Example

  s = "ABC"

  N = 59 (prime number)

  sum = ord('A') + ord('B') + ord('C')

  = 60 + 61 + 62 = 183

  k = **sum** % **N** = 183 % 59 = 6

Generating Hash Codes -- 2

- *Polynomial hash codes* may be more suitable for characters or other multiple-length objects that are tuples
  - Suppose we just used the previous summation method for strings – but comon strings like “pots”, “stop” and “spot” would collide…
  - So a better approach is to take into the consideration the locaiton of each of the letters….
  - See p. 345 for more details on how to generate polynomial hash codes

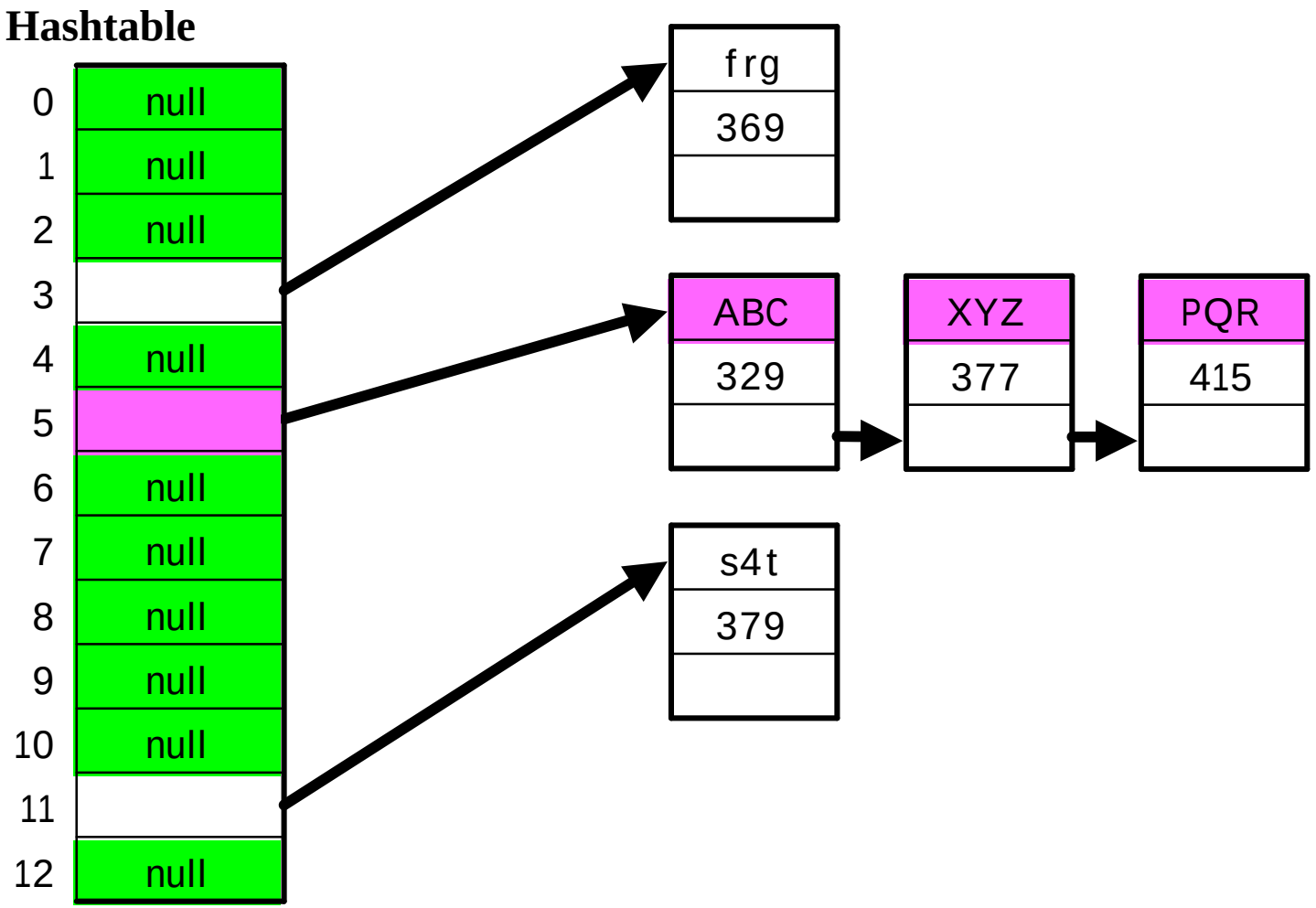
Compression Maps

- Note, we have generated some integer for each key but we still need to map onto the range [0, N-1]
- The simplest compression map is :
  - $h(k) = |k| \bmod N$
  - If N is a prime number then the hash function will help evenly spread out the the distribution of the hashed values and reduce collisions
- When two keys map to the same index, we have a *hash collision*
- When a collision occurs, a *collision resolution algorithm* is used to establish the locations of the colliding keys
- Designing good hash functions is an art!

Collision Resolution

- If two keys map to the same hashtable index, we have a collision
- Two approaches to resolve collisions
  - **Separate chaining**
    - Store all elements which map to the same location in a linked list
  - **Open addressing or rehash**
    - When more than one elements map to the same location check other cells of the hashtable whether they are free in a given order
    - Linear probing
      - inspect k+1, k+2, k+3, ...
    - Quadratic probing
      - inspect k+1, k+4, k+9, k+16, k+25, k+36, ...

Separate Chaining



Load Factor

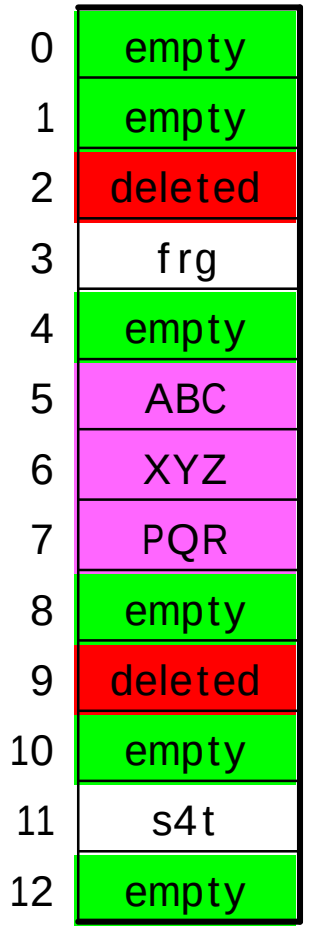
- We expect each bucket to be of size ⌈n/N⌉ -- this parameter is called the *load factor*
- The load factor should be bounded by 1, therefore the expected time for the standard dictionary operations is O(1) -- provided that n is O(N)

Open Addressing

- The disadvantage of the separate chaining method is that it requires the implementation of yet another data structure to hold the items in the log files – it also uses extra space
- There is another approach where we just store at most one element per bucket!
- It requires that n <= N
- Two approaches:
  - Linear probing
  - Quadratic probing

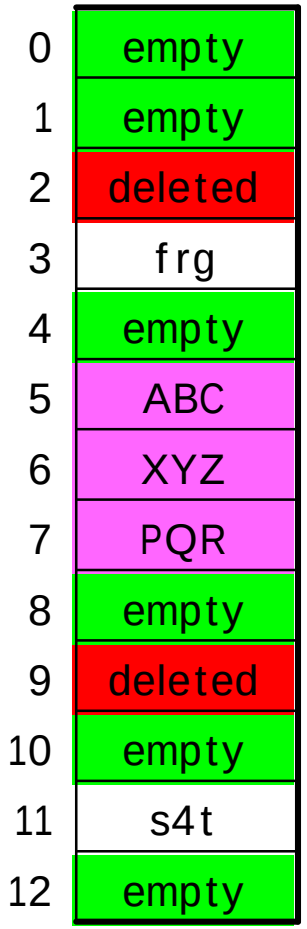
Linear Probing

- Linear probing is an open addressing algorithm
- Locations are checked from the hash location **k** to the end of the table and the element is placed in the first empty slot
  - If the bottom of the table is reached, checking "wraps around" to the start of the table (i.e., modulo hashtable size)



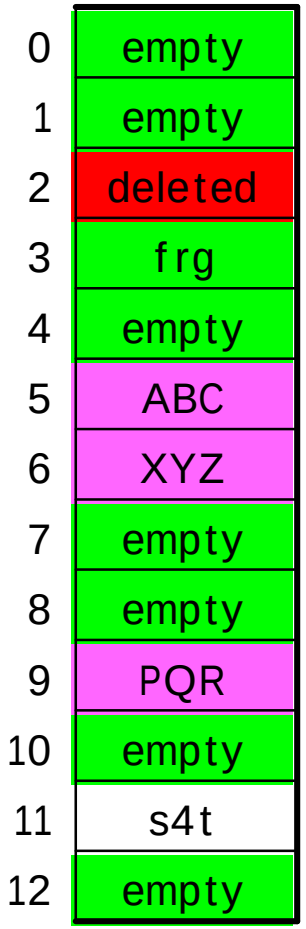
Linear Probing -- 2

- Collision resolution factors into **`member(), insert(), delete()`**
- Thus, if linear probing is used, these routines must continue down the table until a match or empty location is found
- Even though the hashtable size is a prime number (i.e., 13), this is probably not an appropriate size; the size should be at least 30% larger than the maximum number of elements ever to be stored in the table



Quadratic Probing

- Quadratic probing is another open addressing algorithm
- Locations are checked from the hash location to the end of the table and the element is placed in the first computed empty slot
  - Instead of probing consecutive location, we probe the 1st, 4th, 9th, 16th, etc. ? this is called quadratic probing
  - If the bottom of the table is reached, checking "wraps around" to the start of the table (i.e., modulo hashtable size)



Animating collision resolution approaches

- http://www.engin.umd.umich.edu/CIS/course.des/cis350/hashing/WEB/HashApplet.htm