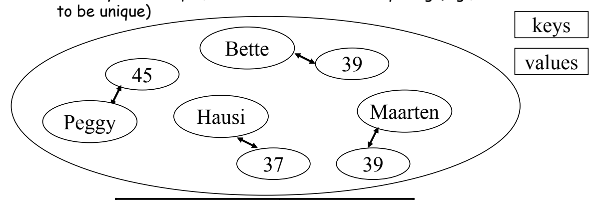
Dictionaries
Hashtables and Hashsearch

Reading Assignment
Chapter 8.1-8.3

Dictionary

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



Search key	Associated value
Bette	39
Hausi	37
Peggy	45
Maarten	39

Dictionary Interface

```
public interface Dictionary {
  public void insert(Object key, Object value);
  public Object member(Object key);
  public Object delete(Object key);
  public Enumeration keys();
  public Enumeration values();
  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 {…}
```


Implementation Strategies

	insert()	delete()	member()
Linear list (linked list, array, vector)	O(1)	O(n)	O(n)
Balanced binary search tree	O(log n)	O(log n)	O(log n)
Hashtable	O(1)	O(1)	O(1)

Abstract Class java.util.Dictionary

- The Dictionary class is the abstract parent of any class, such as Hashtable, which maps keys to values
- Any non-null object can be used as a key and as a value
- 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 value
  abstract Enumeration elements(); // returns all values
  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
}
```


Class java.util.Hashtable

```
public class Hashtable extends Dictionary
       implements Cloneable, Serializable  {
  Hashtable(); // creates empty hashtable with default capacity
  Hashtable(int cap); // hastable with capacity cap
  void clear(); // clears hashtable
  Object clone(); // Creates a shallow copy of this hashtable.
  boolean contains(Object val); // tests if a key maps to val
  boolean containsKey(Object key);//tests if val is a key
  Enumeration keys(); // returns an enumeration of the keys
  Enumeration elements(); returns an enumeration of the values
  Object get(Object key) ; // returns the value for this key
  void put(Object key, Object val); // maps the key to its value
  Object remove(Object key);  //removes the key and its value
  void rehash(); // increase the capacity of the hashtable
  boolean isEmpty(); // tests if this hashtable is empty
  int size(); //returns the number of keys in this hashtable
  String toString(); // returns a string of this hashtable
}
```


Using java.util.Hashtable Class

```
public static void main(String[] args) {
  Hashtable ht = new Hashtable();
  ht.put("Bette", new Integer(39));
  ht.put("Hausi", new Integer(37));
  ht.put("Peggy", new Integer(45));
  ht.put("Maarten", new Integer(39));

  Integer n = (Integer)ht.get("Hausi");
  if (n != null) {
    System.out.println("Hausi's value = " + n);
  }
}
```


How does hashing work?

- How can we find an element in a hashtable in constant time O(1)?
- Given a key, we compute an index into the hashtable using a *hash function* of a *hash code*

key → hashFunction() → index

Search key	Associated value
Bette	39
Hausi	37
Peggy	45
Maarten	39

Hashtable

Hash Functions

- A *hash function* or *hash code* maps keys to indeces
  - It should map keys uniformly across all possible indeces
  - It should be fast to compute
  - It should be applicable to all objects
- Hashtable size
  - The hashtable size should be a prime number
  - The hashtable size should **not** be a power of two
- 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
- In some cases when we know all of the key values in advance we can construct a perfect hash function that maps each key to a different index (i.e., with no collisions)
- Designing good hash functions is an art

String Hash Function: An Example

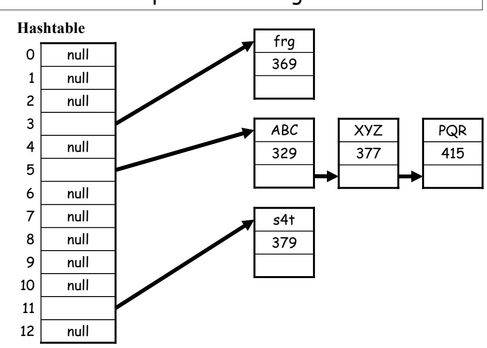
- Let
  - **s** be a key of type **String**
  - **sum** be the sum of the ordinal values of all the characters in **s**
  - **N** be the hashtable size
- Then the hashtable index **k** is
  - **k = sum % N**
    - where **%** is the modulo operator.
    - Thus, **k** is in the range 0 to **N-1**
- Example

```
s = "ABC"
N = 59 (prime number)
sum = ord('A') + ord('B') + ord('C')
    = 60 + 61 + 62 = 183
k = sum % N = 183 % 59 = 6
```


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



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)
- 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

0	empty
1	empty
2	deleted
3	frg
4	empty
5	ABC
6	XYZ
7	PQR
8	empty
9	deleted
10	empty
11	s4t
12	empty

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)

0	empty
1	empty
2	deleted
3	frg
4	empty
5	ABC
6	XYZ
7	empty
8	empty
9	PQR
10	empty
11	s4t
12	empty

Implementation of a Hashtable

- The following hashtable implementation has the following features:
  - ➢ Implements a hash function for Strings
  - ➢ Uses open addressing and in particular linear probing
  - ➢ The Hashtable is an array of HashEntries (Associations)
  - ➢ A HashEntry can assume three states
    - empty, valid, deleted
  - ➢ This implementation does not grow the hashtable if it gets too full (i.e., it ignores the load factor of the Hashtable)
  - ➢ The size of the HashTable should be a prime number so that the modulo operator, used in the hashFunction and in the collision resolution algorithms, distributes the indices over the entire index space more evenly; in particular, don’t use a hashtable size which is a power of two

Class HashEntry

```
private class HashEntry { // local to class Hashtable
   public String key;
   public Object val;
   public boolean deleted;

   public HashEntry(String key, Object val) {
      this(key, val, false);
   }
   public HashEntry(String key, Object val, boolean deleted) {
      this.key = key;
      this.val = val;
      this.deleted = deleted;
}
```

- **Empty entry**: ht entry == **null**
- **Valid entry**: ht entry != **null** && deleted == **false**
- **Deleted entry**: ht entry != **null** && deleted == **true**

Implementation of class Hashtable

```
public class Hashtable implements Dictionary {
   private HashEntry data[];
   private int htSize;
   private int size;
   private static final int defaultCapacity = 997;

   public Hashtable(int initialCapacity) {
      data = new HashEntry[initialCapacity];
      size = 0;
      htSize = initialCapacity;
   }
   public Hashtable() { this(997); }

   public int size() { return size; }
   public int htSize() { return htSize; }
   public boolean isEmpty() { return size == 0; }
}
```


A hashfunction for strings

```
public int hashFunction(String key) {
   int x = 0;
   for (int k=0; k<key.length(); k++) {
      x = 37 * x + key.charAt(k);
      // 37 shifts the bit pattern to make
      // room for the new bit pattern
   }
   x = x % htSize;

   // for loop could generate overflow
   // and hence produce a negative value
   if (x < 0) x = x + htSize;
   return x;
}
```


Insert function

```
public void insert(String key, Object value) {
   if (key == null) return;
   int index = hashFunction(key);
   HashEntry x = data[index];
   if (x == null || x.deleted)  { // entry is empty
      data[index] = new HashEntry(key, value, false);
      size++;
      return;
   } else { // collision
      int k = (index + 1) % htSize;
      while (true) { // linear probing
         x = data[k];
         if (x == null || x.deleted)  { // entry is empty
            data[index] = new HashEntry(key, value, false);
            size++;
            return;
         } else {
            k = (k + 1 ) % htSize;
         }
      }
   }
}
```


Member function

```
public Object member(String key) {
   if (key == null) return null;
   int index = hashFunction(key);
   HashEntry x = data[index];
   if (x == null) { // entry is empty, not found
      return null;
   } else if (!x.deleted && x.key.equals(key)) {
      // found key
      return x.val;
   } else { // collision
      int k = (index + 1) % htSize;
      while (true) {
         // linear probing
         x = data[k];
         if (x == null) { // entry is empty
            return null;
         } else if (x.key.equals(key)) { // found key
            return x.val;
         } else {
            k = (k + 1) % htSize;
         }
      }
   }
}
```

Delete function

- For separate chaining, delete element in the linked list
- For open addressing, mark element as deleted in the hashtable since there might be elements following the deleted element in the linear or quadratic probing chain
- In the implementation below, we use open addressing and hence mark the element as being deleted

Delete function

```
public Object delete(String key) {
   if (key == null) return null;
   int index = hashFunction(key);
   HashEntry x = data[index];
   if (x == null) { // entry is empty, not found
      return null;
   } else if (!x.deleted && x.key.equals(key)) { // found key
      x.deleted = true; size--;
      return x.val;
   } else { // collision
      int k = (index + 1) % htSize;
      while (true) { // linear probing
         x = data[k];
         if (x == null) { // entry is empty
            return null;
         } else if (x.key.equals(key)) { // found key
            x.deleted = true; size--;
            return x.val;
         } else {
            k = (k + 1) % htSize;
         }
      }
   }
}
```


Analysis of Hashtable Access

- If the number of collisions is small, searching, inserting, and deleting elements in a hash table takes O(1) time
- To reduce the number of collisions, in addition to using a good hash function, we should make sure that the table does not get too full
- The load factor of a hash table is the ratio of occupied slots to total slots
- For best results, the load factor should not be above 0.6
- If it gets higher, we should extend the hash table and re-hash all of its elements

Summary

- Dictionary
  - Member, insert, delete; associations: keys, values
- Hashtable
  - Array of hash entries
  - Hash function
    - Compute index from key by 'hashing' the key
    - Distribute indeces over entire index space: 0..htSize
  - Collisions
    - Different keys map to the same index
    - Open addressing: linear, quadratic probing
    - Separate chaining
  - Hashtable implementation
    - Hashtable size should be a prime number
    - 3-state hashtable entry (empty, valid, deleted)
  - Time complexity
    - Member, insert, delete take O(1) time (i.e., constant time)

Fundamental Results of Computer Science

Sorting (Heapsort)	**O(n log n)**
Searching (Hashsearch)	**O(1)**
Selection (finding k^{th} largest element)	**O(n)**