Abstract Datatypes and their Implementation

Some Data Structures
- Elementary Data Structures
  - Arrays, Lists, Trees
- Search Structures
  - Binary Search Trees, Hashtables
- Sequence Structures
  - Stacks, Queues, Priority Queues, Heaps, Extensible Arrays (Java Vectors)
- Graphs

Choosing a Data Structure
Issues:
- What operations do I need to perform on the data?
  - Insertion, deletion, searching, reset to initial state?
- How efficient do the operations need to be?
- Are there any additional constraints on the operations or on the data structure?
  - Can there be duplicates?
  - When extracting elements, does order matter?
- Is there a known upper bound on the amount of data? Or can it grow unboundedly large?

First Things First
- What operations do you need to perform?
- in Java, these are usually specified by an interface (e.g. Iterator, Collection, Set)
- independent of the implementation
- avoid overspecification!

Abstract Datatypes (ADTs)
- A collection of abstract operations and constraints specified independently of the implementation
- Examples: bag, priority queue, dictionary

Two Examples
```java
interface Searchable<E> {
  void insert(E obj);
  void delete(E obj); //remove all objects equal to obj
  boolean search(E obj);
}

interface Bag<E> {
  void put(E obj);
  E get(); //extract some object
  boolean isEmpty();
}
```
One ADT, Many Implementations

interface Bag<E> {
    void put(E obj);
    E get(); //extract some object
    boolean isEmpty();
}

class Stack<E> extends java.util.Stack<E>
    implements Bag<E> {
    public boolean isEmpty() { return empty(); }
    public E get() { return pop(); }
    public void put(E obj) { push(obj); }
}

Searching

interface Searchable<E> {
    void insert(E obj);
    void delete(E obj); //remove all objects equal to obj
    boolean search(E obj);
}

Searching -- Arrays vs Lists

• Arrays
  – Advantage: Random access, fast searching -- O(log n) if sorted
  – Disadvantage: fixed size, insertion & deletion are linear if sorted

• Lists
  – Advantage: Extensible, insertion & deletion are constant time
  – Disadvantage: No random access, searching is linear (even if sorted)

Extensible Arrays (Vectors)

• A good compromise
  – random access, but extensible
  – reallocates if add would cause array bound to be exceeded

public class Vector<E> {
    boolean add(E o);
    void add(int index, E element);
    boolean addAll(Collection<? extends E> c);
    boolean contains(Object elem);
    E elementAt(int index);
    Enumeration<E> elements();
    int indexOf(Object elem);
    int lastIndexOf(Object elem);
    boolean remove(Object o);
    int size();
}
Hashing

• An excellent solution if duplicates not allowed
  – In practice, constant time insert, delete, search
• Based on a hash function that converts data to an index into a large array of lists
  – unlikely that two randomly chosen data items would hash to the same value (this is called a collision)
  – usually implemented in native code -- extremely fast

Java HashSet

```java
public class HashSet<E> {
    boolean add(E o);
    void clear();
    Object clone();
    boolean contains(Object o);
    boolean isEmpty();
    Iterator<E> iterator();
    boolean remove(Object o);
    int size();
}
```

Hashing

Data structure consists of an array of lists

• Insertion:
  – Hash data to get array index
  – Append data to a list at that index

• Search:
  – Hash data to get array index
  – Look for data by walking down list at that index

• Deletion:
  – Hash data to get array index
  – Walk down list at that index and remove data

Performance

Affected by many factors:

• Size of array relative to number of data items
  – Consider limit where there is only 1 bucket
  – as bad as simple linked lists!

• Quality of hash function
  – Good hash functions do not lead to clustering of data → low collision rate

Examples of Hash Functions

`int → {0,1,...,99}`

• Bad:
  – constant functions: hash(x) = 7
  – two most significant digits: hash(379988) = 37

• Better:
  – two least significant digits: hash(379988) = 88
  – sum of digit pairs mod 100: hash(379988) = 37*99+88 (mod 100) = 24
Universal Hashing

- Parametrized family of numeric functions
  - e.g., \( f_{abc}(x) = ax^2 + bx + c \pmod{100} \)
- Pick \( a, b, c \) at random!
- Works as well or better than hand-crafted hash functions in most cases!
- Disadvantage: no persistence

Test of Hash Function

- Multiplicative hash function
  - size of hashtable = 1024
  - key \( k \) is in range 0..32677
  - hash function \( h(k) = (((32768*0.6125423371*k)%32768)%1024) \)

Testing a Hash Function

class HashTest {
  public static void main(String[] args) {
    int[] histogram = new int[1024];
    for (int i = 0; i < 32768; i++) {
      int bucket = (((int)((32768*0.6125423*i)%32768)%1024);
      histogram[bucket]++;
    }
    // print histogram
    System.out.println("Histogram:");
    for (int i = 0; i < 1024; i++) {
      System.out.print(i + " " + histogram[i] + "   ");
      if (i%10 == 0) System.out.println();
    }
  }
}

Testing a Hash Function

Distribution of keys among buckets

- Number of keys = 32768
- Number of buckets = 1024
- Average number of keys/bucket = 32
- Number of keys in each bucket was always in range 29-34
  - Conclusion: this is a good hash function

Hashing Objects

So far, we have stored only integers in hash tables. In general, we want to store objects.
- Give each object an int hash code. Java method: hashCode()
- Contract for hashCode() method:
  - Whenever it is invoked in the same object, it must return the same result
  - Two objects that are equal must have the same hash code
  - Two objects that are not equal should return different hash codes, but are not required to do so

Observations

- Hashing is popular in practice because code is easy to write and maintain and performance is typically excellent
- Performance depends on two key factors:
  - load factor \( \lambda = \) number of entries/size of array
  - choice of hash function
  - if \( \lambda \leq 3/4 \) and hash function is chosen well, get expected \( O(1) \) complexity for all operations
- Our version is called hashing with separate lists or chained hashing – used in Java Collections
- Other methods such as open-address hashing
Dictionaries

• In many applications, we want a more general search structure that stores (key, value) pairs
  – Given a key, find the associated value
• Examples:
  – language dictionaries: key is word, value is meaning
  – telephone directory: key is name, value is telephone number
  – grade sheet for CS211: key is netID, value is grade
• This type of ADT is called a dictionary

Java Hashtables

```java
class HashTest {
    static Hashtable<String, Integer> h = new Hashtable<String, Integer>();
    static {
        h.put("two", new Integer(2));
        h.put("three", new Integer(3));
        h.put("five", new Integer(5));
        h.put("seven", new Integer(7));
    }
    public static void main(String[] args) {
        System.out.println(h.get("three"));
        Enumeration e = h.elements();
        while (e.hasMoreElements()) {
            System.out.print(e.nextElement());
        }
    }
}
```

Dictionaries

```java
public abstract class Dictionary<K, V> {
    abstract Enumeration<V> elements();
    abstract V get(Object key);
    abstract boolean isEmpty();
    abstract Enumeration<K> keys();
    abstract V put(K key, V value);
    abstract V remove(Object key);
    abstract int size();
}
public class Hashtable<K, V> extends Dictionary<K, V> {
    ...
}
```

Next Time

• Priority Queues
• Heaps
• Graphs