ADTs & the Java Collections Framework

Lecture 18
CS211 – Fall 2005

Announcements

- Prelim 1 regrade requests are due today!

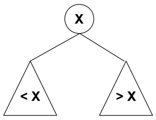
Recall: Useful ADTs

- Stack
  - Push/pop
  - **O(1) worst-case** time using linked list
- Queue
  - Put/get
  - **O(1) worst-case** time using linked list
- Priority Queue
  - Put/getMax
  - **O(log n) worst-case** time using heap (if max heap-size is known)
  - **O(log n) expected** time using heap + table-doubling
- Set
  - Insert/remove/query
  - **O(1) worst-case** time using bit vector (if universe is small)
  - **O(1) expected** time using hash-table + table-doubling
- Dictionary
  - Insert/remove/update/find
  - **O(1) expected** time using hash-table + table-doubling
  - **O(log n) worst-case** time using balanced tree

Dictionary Implementations

- Ordered Array
  - Better than unordered array because Binary Search can be used
- Unordered Linked-List
  - Ordering doesn't help
- Direct Address Table
  - Small universe ⇒ limited usage
- Hashtables
  - O(1) expected time for Dictionary operations

- Goal: Want guaranteed time-per-operation
- Idea: Use a Binary Search Tree (BST)
- BST Property:



Deleting from a BST

Cases:

- Delete a leaf
  - easy
- Delete a node with just one child
  - delete and replace with child
- Delete a node with two children
  - delete node's successor
  - write successor's data into node

- How do we find the successor?
- The successor always has at most one child. Why?
- Would work just as well using predecessor instead of successor

---

BST Performance

- Time for insert(), find(), update(), remove() is O(h) where h is the height of the tree
- How bad can h be?
- Operations are fast if tree is *balanced*
- How balanced is a random tree?
  - If items are inserted in random order then the expected height of a BST is O(log n) where n is the number of items
- If deletion is allowed
  - Tree is no longer random
  - Tree is likely to become unbalanced

Analysis Sketch for Random BST

- Only the number of items and their order is important
  - Can restrict our attention to BSTs containing items $\{1,\ldots, n\}$
- We assume that each item is equally likely to appear as the root
- Define $H(n) \equiv$ *expected* height of BST of size n
- If item i is the root then expected height is

  $1 + \max \{ H(i-1), H(n-i) \}$

  We average this over all possible i
- Can solve the resulting recurrence (by induction) to show

  $H(n) = O(\log n)$

Why use a BST instead of a Hashtable?

- If we use a *balanced* BST scheme then we achieve guaranteed *worst-case* time bound of O(log n) for typical Dictionary ops
- There are some operations that can be efficient on BSTs, but very inefficient on Hashtables

  report-elements-in-order

  getMin

  getMax

  select(k) // find the *k*-th element

  (maintain size of each subtree by using an additional *size* field in each node)
- Note that balanced BST schemes can be difficult to implement
  - But there are lots of reliable codes for these schemes available on the Web
  - Java includes a balanced BST scheme among its standard packages (java.util.TreeMap and java.util.TreeSet)

Java Collections Framework

Java Collections Framework

- *Collections*: holders that let you store and organize objects in useful ways for efficient access
- Since Java 1.2, the package java.util includes interfaces and classes for a *general collection framework*
- Goal: conciseness
  - A few concepts that are broadly useful
  - *Not* an exhaustive set of useful concepts
- Two types of concepts are provided
  - Interfaces (i.e., ADTs)
  - Implementations

JCF Interfaces and Classes

- Interfaces
  - Collection
  - Set (no duplicates)
  - SortedSet
  - List (duplicates OK)

  - Map (i.e., Dictionary)
  - SortedMap

  - Iterator
  - Iterable
  - ListIterator
- Classes
  - HashSet
  - TreeSet
  - ArrayList
  - LinkedList

  - HashMap
  - TreeMap

java.util.Collection<E> (an interface)

- public int size( );
  - Return number of elements in collection
- public boolean isEmpty( );
  - Return true iff collection holds no elements
- public boolean add (Object x);
  - Make sure the collection includes x; returns true if collection has changed (some collections allow duplicates, some don't)
- public boolean contains (Object x);
  - Returns true iff collection contains x (uses equals( ) method)
- public boolean remove (Object x);
  - Removes a single instance of x from the collection; returns true if collection has changed
- public Iterator<E> iterator ( );
  - Returns an Iterator that steps through elements of collection

java.util.Iterator<E> (an interface)

- public boolean hasNext ( );
  - Returns true if the iteration has more elements
- public E next ( );
  - Returns the next element in the iteration
  - Throws NoSuchElementException if no next element
- public void remove ( );
  - The element most-recently returned by next( ) is removed from the collection
  - Throws IllegalStateException if next( ) not yet used or if remove( ) already called
  - Throws UnsupportedOperationException if remove( ) not supported

Additional Methods of Collection

- public Object [ ] toArray ( )
  - Returns a new array containing all the elements of this collection
- public <T> T[ ] toArray (T[ ] dest)
  - Returns an array containing all the elements of this collection; uses dest as that array if it can
- Bulk Operations:
  - public boolean containsAll (Collection c);
  - public boolean addAll (Collection c);
  - public boolean removeAll (Collection c);
  - public boolean retainAll (Collection c);
  - public void clear ( );

java.util.Set<E> (an interface)

- Set *extends* Collection
  - Set inherits *all* its methods from Collection
- A Set contains no duplicates
  - If you attempt to add( ) an element twice then the second add( ) will return false (i.e., the Set has not changed)
- Write a method that checks if a given word is within a Set of words
- Write a method that removes all words longer than 5 letters from a Set
- Write methods for the union and intersection of two Sets

Set Implementations

- java.util.HashSet<E> (a hashtable)
  - Constructors
    - public HashSet ( );
    - public HashSet (Collection c);
    - public HashSet (int initialCapacity);
    - public HashSet (int initialCapacity, float loadFactor);
- java.util.TreeSet (a balanced BST [red-black tree])
  - Constructors
    - public TreeSet ( );
    - public TreeSet (Collection c);
    - …

java.util.SortedSet<E> (an interface)

- SortedSet *extends* Set
- For a SortedSet, the iterator( ) returns the elements in sorted order
- Methods (in addition to those inherited from Set):
  - public E first ( );
    - Returns the first (lowest) object in this set
  - public E last ( );
    - Returns the last (highest) object in this set
  - public Comparator<? super E> comparator ( );
    - Returns the Comparator being used by this sorted set if there is one; returns null if the natural order is being used
  - …

java.lang.Comparable<T> (an interface)

public int compareTo (T x);

Returns a value (< 0), (= 0), or (> 0)

- (< 0) implies *this* is before x
- (= 0) implies *this*.equals(x) is true
- (> 0) implies *this* is after x

- Many classes implement Comparable
  - String, Double, Integer, Char, java.util.Date,…
  - If a class implements Comparable then that is considered to be the class's *natural ordering*

java.util.Comparator<T> (an interface)

public int compare (T x1, T x2);

Returns a value (< 0), (= 0), or (> 0)
- (< 0) implies x1 is before x2
- (= 0) implies x1.equals(x2) is true
- (> 0) implies x1 is after x2

- Can often use a Comparator when a class’s natural order is not the one you want
  - String.CASE_INSENSITIVE_ORDER is a predefined Comparator
  - java.util.Collections.reverseOrder( ) returns a Comparator that reverses the *natural order*

SortedSet Implementations

- java.util.TreeSet<E>
  - This is the only class that implements SortedSet
  - TreeSet’s constructors

    public TreeSet ( );
    public TreeSet (Collection<? extends E> c);
    …

- Write a method that prints out a SortedSet of words in order
- Write a method that prints out a Set of words in order

---

java.util.List<E> (an interface)

- List extends Collection
- Items in a list can be accessed via their index (position in list)
- The add( ) method always puts an item at the end of the list
- The iterator( ) returns the elements in list-order
- Methods (in addition to those inherited from Collection):
  - public E get (int index);
    - Returns the item at position index in the list
  - public E set (int index, E x);
    - Places x at position index, replacing previous item; returns the previous item
  - public void add (int index, E x);
    - Places x at position index, shifting items to make room
  - public E remove (int index);
    - Remove item at position index, shifting items to fill the space; returns the removed item
  - public int indexOf (Object x);
    - Return the index of the first item in the list that equals x (x.equals())
  - …

List Implementations

- java.util.ArrayList<E> (an array; expands via array-doubling)
  - Constructors

    public ArrayList ( );
    public ArrayList (int initialCapacity);
    public ArrayList (Collection<? extends E> c);
- java.util.LinkedList <E> (a doubly-linked list)
  - Constructors

    public LinkedList ( );
    public LinkedList (Collection<? extends E> c);
- Both include some additional useful methods specific to that class

---

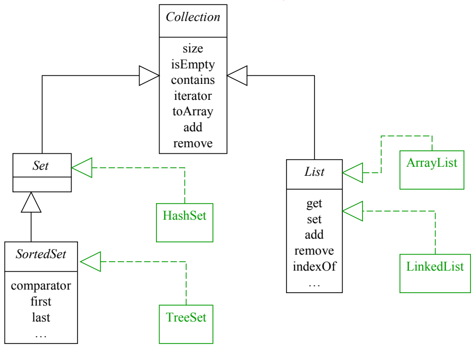
Efficiency Depends on Implementation

- Object x = list.get(k);
  - O(1) time for ArrayList
  - O(k) time for LinkedList
- list.remove(0);
  - O(n) time for ArrayList
  - O(1) time for LinkedList
- If (set.contains(x))…
  - O(1) expected time for HashSet
  - O(log n) for TreeSet

- Write a Stack class
- Write a Queue class
- Write a PriorityQueue class that works on Comparable objects

---

Summary



java.util.Map<K,V> (an interface)

- Map does *not* extend Collection
- A Map contains key/value pairs instead of individual elements
- Methods
  - public V put (K key, V value);
    - Associates value with key in the map; returns the old value associated with key or null if the key did not previously appear in the map
  - public V get (Object key);
    - Returns the object to which this key is mapped or null if there is no such key
  - public boolean containsKey (Object key);
    - True iff Map contains a pair using the given key
  - public boolean containsValue (Object value);
    - True iff there is at least one pair with this value
  - public V remove (Object key);
    - Removes any mapping for the key; returns old value associated with key if there was one (null otherwise)

More Map Methods

- Other methods
  - public int size ( );
    - Return the number of key/value pairs in the Map
  - public boolean isEmpty ( );
    - True iff Map holds no pairs
- Bulk methods
  - public void putAll (Map<? extends K, ? extends V> otherMap);
    - Puts all the mappings from otherMap into this map
  - public void clear ( );
    - Removes all mappings
- Sets/Collections derived from a Map
  - public Set<K> keySet ( );
    - Returns a Set whose elements are the keys of this map
  - public Collection<V> values ( );
    - Returns a Collection whose elements are all the values of this map

java.util.SortedMap<K,V> (an interface)

- Extends the Map contract: requires that keys are sorted
- The iterators for keySet( ), values( ), and entrySet( ) all return items in order of the keys

- Methods (in addition to those inherited from Map):
  - public Comparator<? super K> comparator ( );
    - Returns the comparator used to compare keys for this map; null is returned if the natural order is being used
  - public K firstKey ( );
    - Returns the first (lowest value) key in this map
  - public K lastKey ( );
    - Returns the last (highest value) key in this map
  - …

Set and SortedSet Implementations

- java.util.HashMap (a class; implements Map)
  - Constructors
    public HashMap ( );
    public HashMap (Map<? extends K, ? extends V> map);
    public HashMap (int initialCapacity);
    public HashMap (int initialCapacity, float loadFactor);
- java.util.TreeMap (a class; implements SortedMap)
  - Constructors
    public TreeMap ( );
    public TreeMap (Map<? extends K, ? extends V> map);
    public TreeMap (Comparator<? super K> comp);
    …

Efficiency & Some Comments

- Both TreeMap and HashMap are meant to be accessed via keys
  - get, put, containsKey, remove are all fast
    - O(1) expected time for HashMap
    - O(log n) worst-case time for TreeMap
  - containsValue is slow
    - O(n) for both HashMap and TreeMap

- Both HashSet and TreeSet are actually implemented by building a HashMap and a TreeMap, respectively

- Given a Map that maps student ID number to student name, print out a list of students sorted by ID number and another list sorted by name (assume no duplicate names)

The java.util.Arrays Utility Class

- Provides useful static methods for dealing with arrays
  - sort
    - Mostly uses QuickSort
    - Uses MergeSort for Object[ ] (it's *stable*)
  - binarySearch
  - equals
  - fill
- These methods are overloaded to work with
  - arrays of each primitive type
  - arrays of Objects

- Methods sort and binarySearch can use the natural order or there is a version of each that can use a Comparator

- There is also a method for viewing an array as a List:

  static List asList (Object[ ] a);
  - Note that the resulting List is *backed by* the array (i.e., changes in the array are reflected in the List and vice versa)

Unmodifiable Collections

- Dangerous version:

```
public final String suits[ ] = { “Clubs”, “Diamonds”, “Hearts”, “Spades” };
```

- The final modifier means that suits always refers to the same array, but the array’s elements can be changed
  - suits[0] = “Leisure”;

- Safe version (it would be better really to use an Enum):

```
private final String theSuits[ ] = { “Clubs”, “Diamonds”, “Hearts”, “Spades” };
public final List suits = Collections.unmodifiableList(Arrays.asList(theSuits));
```

- The Collections class provides *unmodifiable wrappers*; any methods that would modify the collection throw an UnsupportedOperationException
  - unmodifiableCollection, unmodifiableSet, unmodifiableSortedSet, unmodifiableList
  - unmodifiableMap, unmodifiableSortedMap

The java.util.Collections Utilities

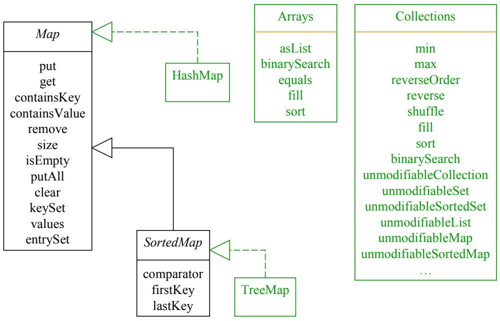
```
public static Object min (Collection c);
public static Object min (Collection c, Comparator comp);
public static Object max (Collection c);
public static Object max (Collection c, Comparator comp);

public static Comparator reverseOrder ( );          // Reverse of natural order

public static void reverse (List list);             // Reverse the list
public static void shuffle (List list);             // Randomly shuffle the list
public static void fill (List list, Object x);      // List is filled with x’s

public static void sort (List list);                // Sort using natural order
public static void sort (List list, Comparator comp);
public static void binarySearch (List list, Object key);
public static void binarySearch (List list, Object key, Comparator comp);
…
```

Summary



Additional JCF Interfaces & Classes

- java.util.Queue<E>
  - An interface
  - Has peek( ) op
  - Implemented by
    - LinkedList
    - PriorityQueue
- Legacy classes
  - java.util.Hashtable
  - java.util.Vector
  - java.util.Stack
- java.util.PriorityQueue<E>
  - A class
  - Heap-based PQ using table-doubling
  - Ordering is based on *natural order* or on a Comparator
    - To use a Comparator, it must be specified in the constructor
  - Implements Queue

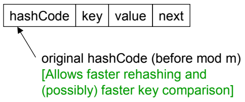
Odds & Ends

Hash Tables in Java

java.util.HashMap
java.util.HashSet
java.util.Hashtable (legacy)

- Use chaining
- Initial (default) size = 101
- Load factor = λ_0 = 0.75
- Uses table doubling (2*previous+1)

A node in each *chain* looks like this:



Hashing Application: Spell Checking

- We want to create a “spelling dictionary” containing 10,000 words
  - A spelling query should be fast
  - Should return true iff word is contained in dictionary
- Basic idea:
  - Use a Hashtable consisting only of bits (say 100K bytes or about 800,000 bits)
  - Compute a hash value for each word and turn on the corresponding bit in the table
  - What’s the probability of a false positive? (It’s too high!)
  - Fix: Use more hash functions

Linear & Quadratic Probing

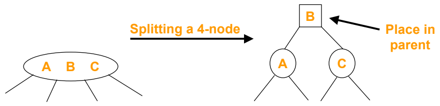
- These are techniques in which all data is stored directly within the hash table array
- Linear Probing
  - Probe at h(X), then at
    - h(X) + 1
    - h(X) + 2
    - …
    - h(X) + i
  - Leads to *primary clustering*
    - Long sequences of filled cells
- Quadratic Probing
  - Similar to Linear Probing in that data is stored within the table
  - Probe at h(X), then at
    - h(X)+1
    - h(X)+4
    - h(X)+9
    - …
    - $h(X)+ i^2$
  - Works well when
    - $\lambda < 0.5$
    - table size is *prime*

Hash Table Pitfalls

- Good hash function is *required*
- Watch the load factor (λ), especially for Linear & Quadratic Probing

Example Balancing Scheme: 234-Trees

- Nodes have 2, 3, or 4 children (and contain 1, 2, or 3 keys, respectively)
- All leaves are at the same level
- Basic rule for insertion: We hate 4-nodes
  - Split a 4-node whenever you find one while coming down the tree
  - Note: this requires that parent is not a 4-node
- Delete is harder than insert
  - For delete, we hate 2-nodes
  - As in BSTs, cannot delete from a nonleaf so we use same BST trick: delete successor and recopy its data



234-Tree Analysis

- Time for insert or get is proportional to tree’s height
- How big is tree’s height *h*?
- Let *n* be the number of nodes in a tree of height *h*
  - n is large if all nodes are 4-nodes
  - n is small if all nodes are 2-nodes
- Can use this to show
  $h = O(\log n)$

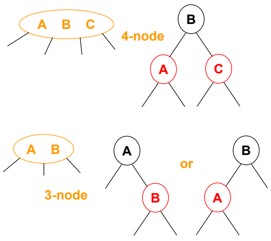
Analysis of tree height:

- Let *N* be the number of nodes, *n* be the number of items, and *h* be the height
- Define *h* so that a tree consisting of a single node is height 0
- It’s easy to see $1+2+4+\ldots+2^h \leq N \leq 1+4+16+\ldots+4^h$
- It’s also easy to see $N \leq n \leq 3N$
- Using the above, we have $n \geq 1+2+4+\ldots+2^h = 2^{h+1}-1$
- Rewriting, we have $h \leq \log(n+1) - 1$ or $h = O(\log n)$
- Thus, Dictionary operations on 234-trees take time O(log n) in the worst case

234-Tree Implementation

- Can implement all nodes as 4-nodes
  - Wasted space
- Can allow various node sizes
  - Requires recopying of data whenever a node changes size
- Can use BST nodes to emulate 2-, 3-, or 4-nodes

Using BSTs to Emulate 234-Trees



- A 2-node can be represented with a standard BST node
- A 4-node can be represented with three BST nodes
- A 3-node can be represented with two BST nodes (in two different ways)

Red-Black Trees

- We need a way to tell when an emulated 234-node starts and ends
- We mark the nodes
  - Black: “root” of 234-node
  - Red: belongs to parent
  - Requires one bit per node
- 234-tree rules become rules for *rotations* and color changes in red-black trees
- Result:
  - one black node per 234-node
  - Number of black nodes on path from root to leaf is same as height of 234-tree
  - All paths from root to leaf have same number of black nodes
  - On any path: at most one red node per black node
  - Thus tree height for red-black tree is O(log n)

Balanced Tree Schemes

- AVL trees [1962]
  - named for initials of Russian creators
  - uses rotations to ensure heights of child trees differ by at most 1
- 23-Trees [Hopcroft 1970]
  - similar to 234-tree, but repairs have to move back up the tree
- B-Trees [Bayer & McCreight 1972]
- Red-Black Trees [Bayer 1972]
  - not the original name
- Red-black convention & relation to 234-trees [Guibas & Stolfi 1978]
- Splay Trees [Sleator & Tarjan 1983]
- Skip Lists [Pugh 1990]
  - developed at Cornell

Selecting a Dictionary Scheme

- Use an unordered array for small sets (< 20 or so)
- Use a Hash Table if possible
  - Cannot efficiently do some ops that are easy with BSTs
  - Running times are expected rather than worst-case
- Use an ordered array if few changes after initialization
- B-Trees are best for large data sets, external storage
  - Widely used within data base software
- Otherwise, Red-Black Trees are current scheme of choice
- Skip Lists are supposed to be easier to implement
  - But shouldn’t have to implement—use existing code
- Splay trees are useful if some items are accessed more often than others
  - But if you know which items are most-commonly accessed, use a separate data structure

---

Selecting a Priority Queue Scheme

- Use an unordered array for small sets (< 20 or so)
- Use a sorted array or sorted linked list if few insertions are expected
- Use an array of linked lists if there are few priorities
  - Each linked list is a queue of equal-priority items
  - Very easy to implement
- Otherwise, use a Heap if you can
- Heap + Hashtable
  - Allow *change-priority* operation to be done in O(log n) expected time
- Balanced tree schemes
  - Useful and practical
- There are a number of alternate implementations that allow additional operations
  - Skew heaps
  - Pairing heaps
  - Fibonacci heaps
  - …