STANDARD ADTS

Lecture 17
CS2110 – Spring 2013
Abstract Data Types (ADTs)

- A method for achieving abstraction for data structures and algorithms
- ADT = model + operations
- Describes what each operation does, but not how it does it
- An ADT is independent of its implementation

- In Java, an interface corresponds well to an ADT
  - The interface describes the operations, but says nothing at all about how they are implemented
- Example: Stack interface/ADT

```java
public interface Stack {
    public void push(Object x);
    public Object pop();
    public Object peek();
    public boolean isEmpty();
    public void clear();
}
```
Queues & Priority Queues

- **ADT Queue**
  - Operations:
    - void add(Object x);
    - Object poll();
    - Object peek();
    - boolean isEmpty();
    - void clear();
  - Where used:
    - Simple job scheduler (e.g., print queue)
    - Wide use within other algorithms

- **ADT PriorityQueue**
  - Operations:
    - void insert(Object x);
    - Object getMax();
    - Object peekAtMax();
    - boolean isEmpty();
    - void clear();
  - Where used:
    - Job scheduler for OS
    - Event-driven simulation
    - Can be used for sorting
    - Wide use within other algorithms

A (basic) queue is “first in, first out”. A priority queue ranks objects: getMax() returns the “largest” according to the comparator interface.
Sets

- **ADT Set**
  - Operations:
    ```java
    void insert(Object element);
    boolean contains(Object element);
    void remove(Object element);
    boolean isEmpty();
    void clear();
    for(Object o: mySet) { ... }
    ```

- **Where used:**
  - Wide use within other algorithms

- **Note:** no duplicates allowed
  - A “set” with duplicates is sometimes called a *multiset* or *bag*

  *A set makes no promises about ordering, but you can still iterate over it.*
Dictionaries

- ADT Dictionary (aka Map)
  - Operations:
    - void insert(Object key, Object value);
    - void update(Object key, Object value);
    - Object find(Object key);
    - void remove(Object key);
    - boolean isEmpty();
    - void clear();

- Think of: key = word; value = definition

- Where used:
  - Symbol tables
  - Wide use within other algorithms

A HashMap is a particular implementation of the Map interface
These are *implementation* “building blocks” that are often used to build more-complicated data structures

- Arrays
- Linked Lists
  - Singly linked
  - Doubly linked
- Binary Trees
- Graphs
  - Adjacency matrix
  - Adjacency list
Given that we want to support some interface, the designer still faces a choice:

- What will be the best way to implement this interface for my expected type of use?
- Choice of implementation can reflect many considerations.

**Major factors we think about**

- Speed for typical use case
- Storage space required
Array Implementation of Stack

```java
class ArrayStack implements Stack {

    private Object[] array; //Array that holds the Stack
    private int index = 0; //First empty slot in Stack

    public ArrayStack(int maxSize) {
        array = new Object[maxSize];
    }

    public void push(Object x) { array[index++] = x; }
    public Object pop() { return array[--index]; }
    public Object peek() { return array[index - 1]; }
    public boolean isEmpty() { return index == 0; }
    public void clear() { index = 0; }
}
```

Question: What can go wrong?

.... What if maxSize is too small?
Linked List Implementation of Stack

class ListStack implements Stack {
    private Node head = null;  // Head of list that
                               // holds the Stack

    public void push(Object x) { head = new Node(x, head); }
    public Object pop() {
        Node temp = head;
        head = head.next;
        return temp.data;
    }
    public Object peek() { return head.data; }
    public boolean isEmpty() { return head == null; }
    public void clear() { head = null; }
}

O(1) worst-case time for each operation (but constant is larger)

Note that array implementation can overflow, but the linked list version cannot
Possible implementations

- **Recall:** operations are `add`, `poll`, `peek`, ...

  - **For linked-list**
    - All operations are $O(1)$

  - **For array with head at A[0]**
    - `poll()` becomes expensive
    - Other ops are $O(1)$
    - Can overflow

  - **For array with wraparound**
    - All operations are $O(1)$
    - Can overflow
A Queue From 2 Stacks

- Add pushes onto stack A
- Poll pops from stack B
- If B is empty, move all elements from stack A to stack B
- Some individual operations are costly, but still $O(1)$ time per operations over the long run
Dealing with Overflow

- For array implementations of stacks and queues, use *table doubling*
- Check for overflow with each insert op
- If table will overflow,
  - Allocate a new table twice the size
  - Copy everything over
- The operations that cause overflow are expensive, but still constant time per operation over the long run (proof later)
Goal: Design a Dictionary (aka Map)

- **Operations**
  - void insert(key, value)
  - void update(key, value)
  - Object find(key)
  - void remove(key)
  - boolean isEmpty()
  - void clear()

**Array implementation:** Using an array of (key,value) pairs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unsorted</th>
<th>Sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>update</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>find</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>remove</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

n is the number of items currently held in the dictionary
Idea: compute an array index via a *hash function* $h$

- $U$ is the universe of keys
- $h: U \rightarrow [0, \ldots, m-1]$ where $m = \text{hash table size}$
- Usually $|U|$ is much bigger than $m$, so *collisions* are possible (two elements with the same hash code)

- $h$ should
  - be easy to compute
  - avoid collisions
  - have roughly equal probability for each table position

Typical situation:
$U = \text{all legal identifiers}$

Typical hash function:
$h$ converts each letter to a number, then compute a function of these numbers

Best hash functions are highly random
This is connected to cryptography
We’ll return to this in a few minutes
A Hashing Example

- Suppose each word below has the following hashCode

<table>
<thead>
<tr>
<th>Word</th>
<th>Hash Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>jan</td>
<td>7</td>
</tr>
<tr>
<td>feb</td>
<td>0</td>
</tr>
<tr>
<td>mar</td>
<td>5</td>
</tr>
<tr>
<td>apr</td>
<td>2</td>
</tr>
<tr>
<td>may</td>
<td>4</td>
</tr>
<tr>
<td>jun</td>
<td>7</td>
</tr>
<tr>
<td>jul</td>
<td>3</td>
</tr>
<tr>
<td>aug</td>
<td>7</td>
</tr>
<tr>
<td>sep</td>
<td>2</td>
</tr>
<tr>
<td>oct</td>
<td>5</td>
</tr>
</tbody>
</table>

- How do we resolve collisions?
  - use chaining: each table position is the head of a list
  - for any particular problem, this might work terribly

- In practice, using a good hash function, we can assume each position is equally likely
Analysis for Hashing with Chaining

- Analyzed in terms of load factor $\lambda = \frac{n}{m} = \frac{\text{(items in table)}}{\text{(table size)}}$
- We count the expected number of probes (key comparisons)
- Goal: Determine expected number of probes for an unsuccessful search
  - Expected number of probes for an unsuccessful search = average number of items per table position = $\frac{n}{m} = \lambda$
  - Expected number of probes for a successful search = $1 + \lambda = O(\lambda)$
- Worst case is $O(n)$
Table Doubling

- We know each operation takes time $O(\lambda)$ where $\lambda = n/m$

- So it gets worse as $n$ gets large relative to $m$

- **Table Doubling:**
  - Set a bound for $\lambda$ (call it $\lambda_0$)
  - Whenever $\lambda$ reaches this bound:
    - Create a new table twice as big
    - Then rehash all the data
  - As before, operations *usually* take time $O(1)$
    - But sometimes we copy the whole table
Analysis of Table Doubling

Suppose we reach a state with n items in a table of size m and that we have just completed a table doubling

<table>
<thead>
<tr>
<th>Copying Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Everything has just been copied</td>
</tr>
<tr>
<td>n inserts</td>
</tr>
<tr>
<td>Half were copied previously</td>
</tr>
<tr>
<td>n/2 inserts</td>
</tr>
<tr>
<td>Half of those were copied previously</td>
</tr>
<tr>
<td>n/4 inserts</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Total work</td>
</tr>
<tr>
<td>n + n/2 + n/4 + … = 2n</td>
</tr>
</tbody>
</table>
Analysis of Table Doubling, Cont’d

- Total number of insert operations needed to reach current table = copying work + initial insertions of items = 2n + n = 3n inserts

- Each insert takes expected time $O(\lambda_0)$ or $O(1)$, so total expected time to build entire table is $O(n)$

- Thus, expected time per operation is $O(1)$

- Disadvantages of table doubling:
  - Worst-case insertion time of $O(n)$ is definitely achieved (but rarely)
  - Thus, not appropriate for time critical operations
Concept: “hash” codes

- Definition: a hash code is the output of a function that takes some input and maps it to a pseudo-random number (a hash)
  - Input could be a big object like a string or an Animal or some other complex thing
  - Same input always gives same out
  - Idea is that hashCode for distinct objects will have a very low likelihood of collisions
- Used to create index data structures for finding an object given its hash code
Java Hash Functions

- Most Java classes implement the `hashCode()` method
- `hashCode()` returns an int
- Java’s `HashMap` class uses 
  \[ h(X) = X.hashCode() \mod m \]
- `h(X)` in detail:
  - `int hash = X.hashCode();`
  - `int index = (hash & 0x7FFFFFFF) \% m;`

- What `hashCode()` returns:
  - Integer:
    - uses the int value
  - Float:
    - converts to a bit representation and treats it as an int
  - Short Strings:
    - `37*previous + value of next character`
  - Long Strings:
    - sample of 8 characters; `39*previous + next value`
hashCode() Requirements

- Contract for `hashCode()` method:
  - Whenever it is invoked in the same object, it must return the same result.
  - Two objects that are equal (in the sense of `.equals(...)`) must have the same hash code.
  - Two objects that are not equal should return different hash codes, but are not required to do so (i.e., collisions are allowed).
Hashtables in Java

- `java.util.HashMap`
- `java.util.HashSet`
- `java.util.Hashtable`

- Use chaining

- Initial (default) size = 101

- Load factor = \( \frac{1}{0} = 0.75 \)

- Uses table doubling (\( 2 \times \text{previous} + 1 \))

  - A node in each chain looks like this:

    ```
    hashCode | key | value | next
    ------------
    original hashCode (before mod m)
    Allows faster rehashing and (possibly) faster key comparison
    ```
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
    - \( \lfloor \frac{i}{n} \rfloor < 0.5 \)
    - Table size is prime
Universal Hashing

- In doubt, choose a hash function at random from a large parameterized family of hash functions (e.g., $h(x) = ax + b$, where $a$ and $b$ are chosen at random)
- With high probability, it will be just as good as any custom-designed hash function you dream up
Dictionary Implementations

- **Ordered Array**
  - Better than unordered array because Binary Search can be used

- **Unordered Linked List**
  - Ordering doesn’t help

- **Hashtables**
  - $O(1)$ expected time for Dictionary operations
Aside: Comparators

- When implementing a comparator interface you normally must
  - Override `compareTo()` method
  - Override `hashCode()`
  - Override `equals()`

- Easy to forget and if you make that mistake your code will be very buggy
We mentioned that the hash codes of two equal objects must be equal — this is necessary for hashtable-based data structures such as HashMap and HashSet to work correctly.

In Java, this means if you override \texttt{Object.equals()}, you had better also override \texttt{Object.hashCode()}

But how???
class Identifier {
    String name;
    String type;

    public boolean equals(Object obj) {
        if (obj == null) return false;
        Identifier id;
        try {
            id = (Identifier)obj;
        } catch (ClassCastException cce) {
            return false;
        }
        return name.equals(id.name) && type.equals(id.type);
    }
}
class Identifier {
    String name;
    String type;

    public boolean equals(Object obj) {
        if (obj == null) return false;
        Identifier id;
        try {
            id = (Identifier)obj;
        } catch (ClassCastException cce) {
            return false;
        }
        return name.equals(id.name) && type.equals(id.type);
    }

    public int hashCode() {
        return 37 * name.hashCode() + 113 * type.hashCode() + 42;
    }
}
class TreeNode {
    TreeNode left, right;
    String datum;

    public boolean equals(Object obj) {
        if (obj == null || !(obj instanceof TreeNode)) return false;
        TreeNode t = (TreeNode)obj;
        boolean lEq = (left != null)?
            left.equals(t.left) : t.left == null;
        boolean rEq = (right != null)?
            right.equals(t.right) : t.right == null;
        return datum.equals(t.datum) && lEq && rEq;
    }
}
class TreeNode {
    TreeNode left, right;
    String datum;

    public boolean equals(Object obj) {
        if (obj == null || !(obj instanceof TreeNode)) return false;
        TreeNode t = (TreeNode)obj;
        boolean lEq = (left != null)? left.equals(t.left) : t.left == null;
        boolean rEq = (right != null)? right.equals(t.right) : t.right == null;
        return datum.equals(t.datum) && lEq && rEq;
    }

    public int hashCode() {
        int lHC = (left != null)? left.hashCode() : 298;
        int rHC = (right != null)? right.hashCode() : 377;
        return 37 * datum.hashCode() + 611 * lHC - 43 * rHC;
    }
}
For large objects we often compute an MD5 hash

- MD5 is the fifth of a series of standard “message digest” functions
- They are fast to compute (like an XOR over the bytes of the object)
- But they also use a cryptographic key: without the key you can’t guess what the MD5 hashcode will be
  - For example key could be a random number you pick when your program is launched
  - Or it could be a password

- With a password key, an MD5 hash is a “proof of authenticity”
  - If object is tampered with, the hashcode will reveal it!