Abstract Data Types (ADTs)

- A method for achieving abstraction for data structures and algorithms
- 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

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:
    - void insert(Object element);
    - boolean contains(Object element);
    - void remove(Object element);
    - boolean isEmpty();
    - void clear();
  - Where used:
    - Symbol tables
    - 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.

Data Structure Building Blocks

- 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
From interface to implementation

- 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

```java
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;
    }
}
```

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

Queue Implementations

- Possible implementations
  - Recalling: operations are add, poll, peek...
    - For linked list
      - All operations are O(1)
    - For array with head at A[0]
      - poll takes time O(n)
      - 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 Insert</th>
<th>Sorted Insert</th>
<th>Unsorted Update</th>
<th>Sorted Update</th>
<th>Unsorted Find</th>
<th>Sorted Find</th>
<th>Unsorted Remove</th>
<th>Sorted Remove</th>
<th>Unsorted isEmpty</th>
<th>Sorted isEmpty</th>
</tr>
</thead>
<tbody>
<tr>
<td>void insert</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>void update</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Object find</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>void remove</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>False</td>
<td>True</td>
<td>O(n)</td>
<td>O(n)</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>False</td>
<td>True</td>
<td>O(n)</td>
<td>O(n)</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>void clear()</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>False</td>
<td>True</td>
<td>O(n)</td>
<td>O(n)</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

n is the number of items currently held in the dictionary

Hashing

- Idea: compute an array index via a hash function \( h \)
  - \( U \) is the universe of keys
  - \( h: U \rightarrow \{0, \ldots, m-1\} \)

Typical situation:
- \( U \) = 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 hash code

<table>
<thead>
<tr>
<th>Hash Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>jan 7</td>
<td></td>
</tr>
<tr>
<td>mar 5</td>
<td></td>
</tr>
<tr>
<td>apr 2</td>
<td></td>
</tr>
<tr>
<td>may 4</td>
<td></td>
</tr>
<tr>
<td>jun 7</td>
<td></td>
</tr>
<tr>
<td>jul 3</td>
<td></td>
</tr>
<tr>
<td>aug 7</td>
<td></td>
</tr>
<tr>
<td>sep 2</td>
<td></td>
</tr>
<tr>
<td>oct 5</td>
<td></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 \( \lambda = n/m \)
  - (items in table)/(table size)
  - Expected number of probes for an unsuccessful search = average number of items per table position = \( n/m = \lambda \)
  - Expected number of probes for a successful search = \( 1 + \lambda \) = \( O(\lambda) \)

- We count the expected number of probes (key comparisons)
  - Goal: Determine expected number of probes for an unsuccessful search:
    - 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>
<th>Table Doubling</th>
</tr>
</thead>
<tbody>
<tr>
<td>n inserts</td>
<td>n + n/2 + n/4 + \ldots = 2n</td>
</tr>
<tr>
<td>n/2 inserts</td>
<td></td>
</tr>
<tr>
<td>n/4 inserts</td>
<td></td>
</tr>
<tr>
<td>\ldots</td>
<td></td>
</tr>
<tr>
<td>Total work</td>
<td></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 $O(\lambda)$ 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:
  - `int`: 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 = 0.75
  - Uses table doubling ($2^{*}previous+1$)

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 $h(X)+1$, $h(X)+2$, ..., $h(X)+i$, $h(X)+i^2$
  - 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 $i < 0.5$
  - 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
- Hash tables
  - \( O(1) \) 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

hashCode() and equals()

- 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 Object.equals(), you had better also override Object.hashCode()

- But how???

```java
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;
    }
}
```
Professional quality hash codes?

- 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!