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: List interface/ADT
    ```java
    public interface List<E> {
        public void add(int index, E x);
        public boolean contains(Object o);
        public E get(int index);
        ...
    }
    ```

Sets

- ADT Set
  - Maintains a set of objects.
  - Operations:
    - void insert(Object element);
    - boolean contains(Object element);
    - void remove(Object element);
    - boolean isEmpty();
    - void clear();
- Where used:
  - Keep track of states that were visited already
  - Wide use within other algorithms
- Note: no duplicates allowed
  - A "set" with duplicates is sometimes called a multiset or bag

Queues

- ADT Queue
  - Maintains a queue of objects where objects are added to the end and extracted at the front.
  - 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

Priority Queues

- ADT PriorityQueue
  - Maintains a queue where objects are first sorted by priority, then by arrival time.
  - 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

Stacks

- ADT Stack
  - Maintains a collection of objects where objects are added and removed at the front.
  - Operations:
    - void push(Object element);
    - Object pop();
    - Object peek();
    - boolean isEmpty();
    - void clear();
- Where used:
  - Frame stack
  - Wide use within other algorithms
Dictionaries

- ADT Dictionary (aka Map)
  - Stores a collection of key-value pairs. Objects are accessed via the key.
  - 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

Array Implementation of Stack

```java
class ArrayStack implements Stack {
    private Object[] array; //Array that holds 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?

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
  - Hashtables

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; }
}
```

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

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

Queue Implementations

- Possible implementations
  - Linked List
    - head
    - last
  - Array with head always at A[0]
    - last
  - Array with wraparound
    - head
    - last
- Recall: 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

- Algorithm
  - 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 Array 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: Implement 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
  - Unsorted
    - insert O(1) O(n)
    - update O(n) O(log n)
    - find O(n) O(log n)
    - remove O(n) O(n)
  - Sorted
- 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 (e.g. all legal identifiers)
  - : U \rightarrow [0,...,m-1]
    where m = hash table size
- Usually |U| is much bigger than m, so collisions are possible (two elements with the same hash code)
- Hash function h should
  - be easy to compute
  - avoid collisions
  - have roughly equal probability for each table position

A Hashing Example

- Suppose each word below has the following hash-code
  - Jan 7
  - Feb 0
  - Mar 5
  - Apr 2
  - May 4
  - Jun 7
  - Jul 3
  - Aug 7
  - Sep 2
  - Oct 5
- 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)
- We count the expected number of probes (i.e. 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
  - \( = n/m = \lambda \)
- Expected number of probes for a successful search
  - \( = 1 + \lambda/2 = O(\lambda) \)
- 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 (i.e. copy into new table)
- 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></th>
</tr>
</thead>
<tbody>
<tr>
<td>Everything has just been copied</td>
<td>( n ) inserts</td>
</tr>
<tr>
<td>Half were copied in previous doubling</td>
<td>( n/2 ) inserts</td>
</tr>
<tr>
<td>Half of those were copied in doubling before previous one</td>
<td>( n/4 ) inserts</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Total work</td>
<td>( n + n/2 + n/4 + ... \leq 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

### Java Hash Functions

- Most Java classes implement the `hashCode()` method
  - `hashCode()` returns `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 for:
  - 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`
- Implementation
  - Use chaining
  - Initial (default) size = 101
  - Load factor = \( \lambda_0 = 0.75 \)
  - Uses table doubling (2*previous+1)

### Linear & Quadratic Probing

- These are techniques in which all data is stored directly within the hash table array
- Quadratic Probing
  - Similar to Linear Probing in that data is stored within the table
  - Probe at \( h(X) \), then at
    - \( h(X) + 1 \)
    - \( h(X) + 2 \)
    - ...  
    - \( h(X) + i^2 \)
  - Works well when
    - \( \lambda < 0.5 \)
    - Table size is prime
Universal Hashing

• 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 can come up with

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???

```
public int hashCode() {
    return 37 * name.hashCode() + 113 * type.hashCode() + 42;
}
```

```
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;
}
```

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