Announcements

- Quiz Thursday!
- Topics:
  - Searching, sorting, asymptotic complexity (Lectures 11 & 12)
  - ADTs and their implementations – Stacks, Queues, Priority Queues, Sets, Dictionaries, Arrays, Lists, Hashstables (Lecture 16) (today!)

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<T> {
      public void push(T x);
      public T pop();
      public T peek();
      public boolean isEmpty();
      public void clear();
  }
  ```

Queues & Priority Queues

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

- ADT PriorityQueue
  - Contains objects of type T extends Comparable<T>
  - Operations:
    ```java
    void insert(T x);
    T getMax();
    T peekAtMax();
    boolean isEmpty();
    void clear();
    ```
  - Where used:
    - Job scheduler for OS
    - Event-driven simulation
    - Can be used for sorting
    - Wide use within other algorithms

Sets

- ADT Set
  - Operations:
    ```java
    void insert(T element);
    boolean contains(T element);
    void remove(T element);
    boolean isEmpty();
    void clear();
    ```
  - Where used:
    - Wide use within other algorithms
  - Note: no duplicates allowed
  - A "set" with duplicates is sometimes called a multiset or bag

Dictionaries

- ADT Dictionary (aka Map)
  - Like Java interface Map<K,V>
  - Operations:
    ```java
    void insert(K key, V value);
    void update(K key, V value);
    V find(K key);
    void remove(K key);
    boolean isEmpty();
    void clear();
    ```
  - Think of: key = word; value = definition
  - Where used:
    - Symbol tables
    - Wide use within other algorithms
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

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?

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;
  }
}
```

Linked List Implementation of Stack

```java
class LinkedList implements Stack<T> {
  private Node head = null; //Head of list that holds the Stack

  public void push(T x) {
    head = new Node(x, head);
  }

  public T pop() {
    Node temp = head;
    head = head.next;
    return temp.data;
  }

  public T peek() {
    return head.data;
  }

  public boolean isEmpty() {
    return head == null;
  }

  public void clear() {
    head = null;
  }
}
```

Queue Implementations

- Possible implementations
  - Linked List
    - For linked list
      - all operations are O(1)
  - Array with head at A[0] (poll becomes expensive) (can overflow)
    - For array with wraparound
      - all operations are O(1)
      - can overflow
  - Array with head always at A[0] (poll becomes expensive) (can overflow)
    - For array with wraparound
      - all operations are O(1)
      - can overflow

A Queue From 2 Stacks

- push pushes onto stack A
- pop 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 operation 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

- Insert: $O(1)$
- Update: $O(n)$
- Find: $O(n)$
- Remove: $O(1)$
- Clear: $O(n)$

Hashing

- Idea: compute an array index via a hash function $h$
- $U$ is the universe of keys
- $h: U \rightarrow \{0, \ldots, m-1\}$ where $m$ = hash table size
- Usually $|U|$ is much bigger than $m$, so collisions are possible

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

Typical situation: $U$ = all legal identifiers

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

Java HashSet, HashMap

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
  - Nov 4
  - Dec 1

- 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 $\lambda = \frac{n}{m}$
  - $n$ = items in table
  - $m$ = (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 = $1 + \frac{\lambda}{2} = O(1)$
  - Worst case is $O(n)$

Table Doubling

- We know each operation takes time $O(\lambda)$ where $\lambda = \frac{n}{m}$
  - 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
    - Rehash all the data into the new table
  - Typical value for $\lambda_0$ is 0.75

- 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 we have just completed a table doubling

<table>
<thead>
<tr>
<th>Copying Work</th>
<th>Number of Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Everything has just been copied</td>
<td>$n$ inserts</td>
</tr>
<tr>
<td>Half were copied previously</td>
<td>$n/2$ inserts</td>
</tr>
<tr>
<td>Half of those were copied previously</td>
<td>$n/4$ inserts</td>
</tr>
<tr>
<td>Total work</td>
<td>$n + n/2 + n/4 + \ldots + 2n$</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(λ0) 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 method `int hashCode()`
- Java’s `HashMap` class uses `h(X) = X.hashCode() mod m`
- `h(X)` in detail:
  
  ```java
  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)

Hashables in Java

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

- Use chaining
- Initial (default) size = 101
- Load factor = λ0 = 0.75
- Uses table doubling
  
  ```java
  2 * previous + 1
  ```

- A node in each chain looks like:

```
<table>
<thead>
<tr>
<th>hashCode</th>
<th>key</th>
<th>value</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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
  - Works well when
    - i > 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???

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;
    }
}

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