CS/ENGRD 2110
Object-Oriented Programming and Data Structures

Spring 2011
Thorsten Joachims

Lecture 16: Standard ADTs

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

```
public interface List<E> {
        public void add(int index, E x);
        public boolean contains(Object o);
        public E get(int index);
        …
}
```

Sets

- ADT Set
  - Operations:
    - **`void insert(Object element);`**
    - **`boolean contains(Object element);`**
    - **`void remove(Object element);`**
    - **`boolean isEmpty();`**
    - **`void clear();`**
- Where used:
  - Peg Solver (memoize boards 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 & 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

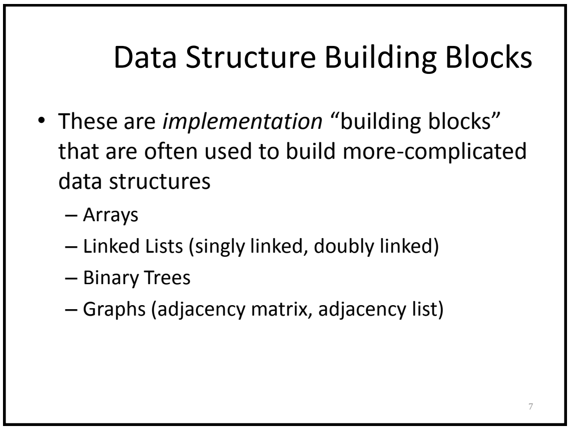
- ADT PriorityQueue
  - Operations:
    - void insert(Object x);
    - Object getMax();
    - Object peekAtMax();
    - boolean isEmpty();
    - void clear();
- Where used:
  - Job scheduler for OS
  - Event-driven simulation
  - Can be used for sorting
  - Wide use within other algorithms

Stacks

- ADT Stack
  - Operations:
    - **`void push(Object element);`**
    - **`Object pop();`**
    - **`Object peek();`**
    - **`boolean isEmpty();`**
    - **`void clear();`**
- Where used:
  - Frame stack
  - Wide use within other algorithms

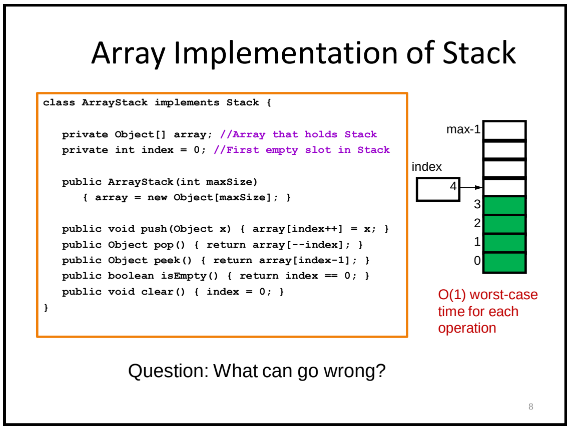
Dictionaries

- ADT Dictionary (aka Map)
  - 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



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)

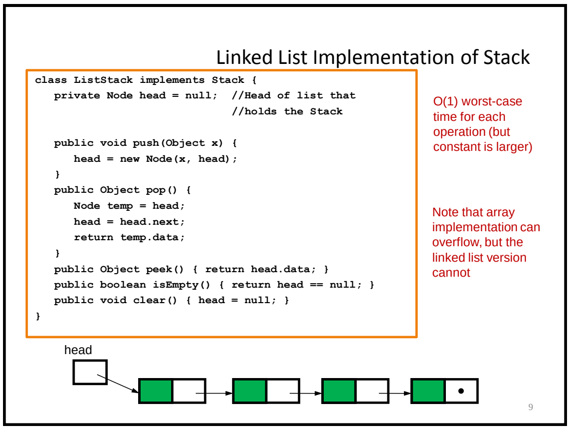


Array Implementation of Stack

```
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; }
}
```

O(1) worst-case time for each operation

Question: What can go wrong?

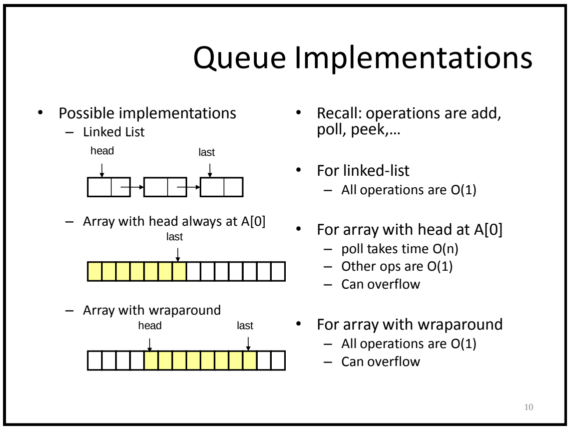


Linked List Implementation of Stack

```
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



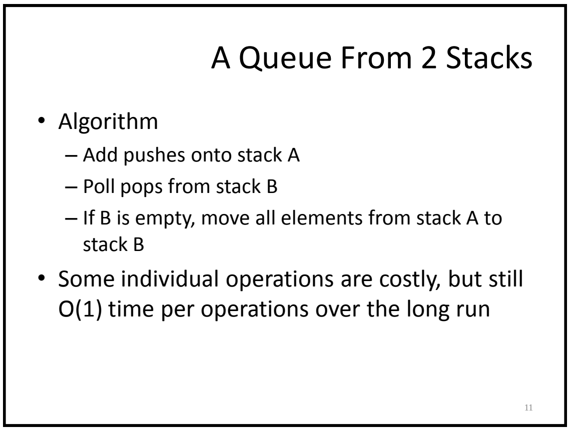
Queue Implementations

- Possible implementations
  - Linked List

  - Array with head always at A[0]

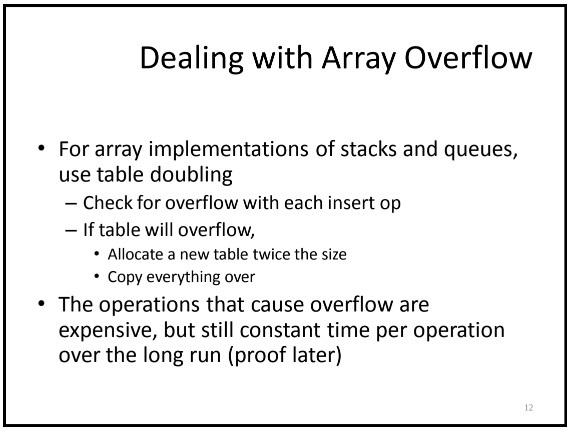
  - Array with wraparound

- 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	Sorted
– insert	O(1)	O(n)
– update	O(n)	O(log n)
– find	O(n)	O(log n)
– remove	O(n)	O(n)

- 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)
  - h: U → [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 λ = 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 = λ
- Expected number of probes for a successful search

  = 1 + λ/2 = O(λ)
- Worst case is O(n)

Table Doubling

- We know each operation takes time O(λ) where λ=n/m
- So it gets worse as n gets large relative to m
- Table Doubling:
  - Set a bound for λ (call it λ_0)
  - Whenever λ 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

	Copying Work
Everything has just been copied	n inserts
Half were copied in previous doubling	n/2 inserts
Half of those were copied in doubling before previous one	n/4 inserts
…	…
Total work	n + n/2 + n/4 + … = 2n

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(λ_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 = λ_0 = 0.75
  - Uses table doubling (2*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
  - $\lambda < 0.5$
  - 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???

`hashCode()` and **`equals()`**

```
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;
   }
}
```

`hashCode()` and **`equals()`**

```
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;
   }
}
```

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