Sorting, Stacks, and Assignment 3

CS187 Data Structures
February 17, 2010
Chris Vigorito
Java Sorting Options

- Regular arrays: `Arrays.sort()`
- Collections (e.g., `ArrayList`): `Collections.sort()`
- Both use "natural ordering" unless you specify a specific `Comparator` object
Natural Ordering

• For most classes in API, “natural ordering” is common sense ordering
• Obvious for numbers
• Lexicographically (alphabetically) for Strings
• For other defined classes, need to look up in API
Comparable

- Interface that specifies natural ordering of a class
- Just one method: `compareTo(<T> o)`
- Compares object with method parameter
- Returns -1, 0, or 1 if object is less than, equal to, or greater than method parameter
Arrays

- Overloaded static sort method for each primitive type (`int[]`, `double[]`, etc.)

- **Example:** `Arrays.sort(myArray);`

- Sorts array in ascending order using natural ordering

- Same for an array of Objects (must implement `Comparable`)

Wednesday, February 17, 2010
Collections

- `sort` method for any collection that implements `List`

- **Example:** `Collections.sort(myList);`

- Sorts array in ascending order using natural ordering (list type must implement `Comparable`)
Comparator

- Used to define another ordering on an object type
- Allows you to sort based on multiple different features
- Interface with method `compare(<T> o1, <T> o2)` that must be implemented
- Method returns -1, 0, or 1 if `o1` is smaller, equal to, or bigger than `o2`
Using Comparator

• Both Arrays and Collections class have sort methods that let you specify a Comparator

• Comparator must be of appropriate type (can’t cause ClassCastException)

• All sort methods and Comparator use generics

• Example: Collections.sort(myList, myComparator)
public class LengthComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    } // compare()
} // LengthComparator

class Test {
    public static void main(String[] args) {
        ArrayList<String> myList = new ArrayList<String>();
        // populate list
        Collections.sort(myList); // sorts myList lexicographically
        Collections.sort(myList, new LengthComparator());
        // sorts myList by String length
    } // main()
} // Test
Summary

• If using primitive type arrays, use Arrays.sort to sort them

• If using an array of Object, use Arrays.sort, but object type must implement Comparable

• If using a collection, use Collections.sort - collection type must implement Comparable

• Create a Comparator when you want to sort according to a different ordering than implemented by Comparable
Stacks

• Java implementation of a stack is given in `java.util.Stack`

• **Extends** `Vector`, **which is an implementation of an abstract list, similar to ArrayList**

• Provides the standard push and pop methods, plus some other useful non-standard ones

• Uses generics like `ArrayList`
Stack

• `void push(<T> o)` pushes `o` onto top of stack

• `<T> pop()` pops top element off the stack and returns it

• `<T> peek()` returns object on top of stack, but leaves it on the stack

• **Both pop and peek throw `EmptyStackException` if stack is empty**
Stack

- boolean isEmpty() returns true if stack is empty
- int search(Object o) looks for o in the stack and returns its distance from the top of the stack (top of stack has distance 1)
- int size() returns number of elements in stack
Stack

- Also have access to all Vector methods, like `get`, `set`, `indexOf`, `iterator`, etc.

- Sort of defeats the purpose of representing as a stack, but allows you to access inner elements when necessary (like in Assignment 3)

- Stack iterator iterates over stack elements starting from `bottom`

- Like with `ArrayList`, can’t change list while iterating (can’t push or pop) - not a good idea anyway
import java.util.Stack;

public class StackTest {
    public static void main(String[] args) {
        Stack<Integer> myStack = new Stack<Integer>();
        myStack.push(2);
        myStack.push(4);
        myStack.push(6);
        System.out.println(myStack); // prints [2 4 6]
        System.out.println(myStack.peek()); // prints 6
        myStack.pop();
        System.out.println(myStack.pop()); // prints 4
    }
}

Wednesday, February 17, 2010
Assignment 3

• Solving the N-Queens problem
• How do you place N queens on an NxN chessboard so that no two queen are attacking?
• Attacking means same column, row, or diagonal
• Usually many distinct solutions - you need to find all of them for a given N
N Queens

- Lots of algorithms for finding solutions
- You will use simple backtracking
  - Has a straightforward recursive implementation, but you must implement with a stack directly
  - Like simulating an N-nested for loop
Backtracking

- For each row, place a queen in first valid position, then move on to next row
- If there is no valid position, then backtrack - return to previous row and try next position
- If you successfully place a queen in the last row, then a solution has been found - print and try next position
Implementing with a Stack

- Stack will keep track of current position in each row
- Each element stores column position, while depth in stack represent row position
- When you place a queen, you push the column position on the stack
- When you need to backtrack, you pop and continue from next position
- If stack size is ever N, then solution found
Example (N=4)
Assignment 3

- Starter code provided
- `printSolution` method prints out a solution given a stack
- Your job is to write `solve` method, which finds all solutions and calls `printSolution` when appropriate
- Don’t change `main`, but you are allowed to add methods
Assignment 3

- Probably a good idea to write a method to check placement validity
- Need to access elements of stack to check if a placement is valid
- Don’t use push or pop
- Use iterators - recall they start from bottom
- Possible to check a given position in just two lines - don’t need a lot of code