Testing Sorting

cs206
April 14

Writing Comprehensive Tests

- By Software or Algorithm Analysis
  - Examine implementation (algorithm)
    - Draw a tree in which every "if" causes a branching
      - Yes left; No right
    - Consider what the program should do at every leaf
    - Figure out a test case to get to every leaf in tree
    - Problems with global / method variables
    - Work with only public methods
      - Handling side-effects (like printing)
- Check what program actually does against what it should do

Writing Comprehensive Tests — Pt 2

- Test -driven design
  - Work with abstract description of problem
  - Before any implementation write tests with bothin input and output
  - Any implementation must pass all tests.
  - If pass all tests, the the program does with it is supposed to do.

Test Example — Tree Insertion

```
public void insertAlt(final E element) {
        if (root==null) {
            root=new Node(element);
            size = 1;
        } else
            iInsertAlt(root, element);
    }
```

```
    private void iInsertAlt(final Node treepart, final E toBeAdded) {
        final int cmp = treepart.payload.compareTo(toBeAdded);
        if (cmp==0) return; // the item is in the tree
        if (cmp>0) {   // Mar 26 fixed wrong direction on comparison
            if (treepart.left==null) {
                size++;
                treepart.left=new Node(toBeAdded);
            } else {
                iInsertAlt(treepart.left, toBeAdded);
            }
        } else {// cmp>0
            if (treepart.right==null) {
                size++;
                treepart.right=new Node(toBeAdded);
            } else {
                iInsertAlt(treepart.right, toBeAdded);
            }}}
```

Testing Conclusions

- From Software / Algorithm Analysis
  - If the software itself is flawed, the tests may incorrectly indicate that the program is correct
  - Tests may be closely tied to implementation — so implementation change requires test change
- From Algorithm Analysis
  - Complete algorithm specs are hard to write

Abstract Classes

Halfway between interface and class

```java
public abstract class AbstractPriorityQueue <K extends Comparable<K>, V> {
    enum Ordering { ASCENDING, DESCENDING, MIN, MAX}
    protected Ordering order;
    protected class Entry<L extends Comparable<L>,W> {
        final L theK;
        final W theV;
        public Entry(L kk, W vv) {
            theK = kk;
            theV = vv;
        }
        protected int doCompare(Entry<L,W> e2) {
            switch (order) {
                case MIN:
                case ASCENDING:
                    return this.theK.compareTo(e2.theK);
                case MAX:
                case DESCENDING:
                default:
                return e2.theK.compareTo(this.theK);
            }
        }
        public String toString() {
            return "{"+theK+","+theV+"}";
        }
    }
    public abstract int size();
    public abstract boolean isEmpty();
    public abstract boolean offer(K k, V v);
    public abstract V poll();
    public abstract V peek();
}
```

- Some methods may be defined
- Other methods “abstract”
- Cannot be new’ed
    - ~~new AbstractPriorityQueue~~
- Extend like normal class, but must implement abstract methods

Priority Queue Sort

- Sorting using a priority queue
  1. Insert with a series of `insert` operations
  2. Remove in sorted order with a series of `poll` operations
- Efficiency depends on implementation and runtime of `insert` and `poll`

Selection Sort

- Selection-sort:
  - select the min/max and swap with 0
- priority queue is implemented with an unsorted sequence
- Time:
  - Add: O(n)
  - Remove: $O(n^2)$

Example

Phase 1 — Inserting

(a)	7	(7)	
(b)	4	(7,4)	
….			
(g)	()		(7,4,8,2,5,3,9)

Phase 2 — Polling

(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

Insertion Sort

- Insertion-sort:
  - insert/swap the element into the correct sorted position
- Priority queue where the priority queue is implemented with a sorted sequence
- Time:
  - Add:O(n^2)
  - Remove: O(n)

Example

Phase 1 — Inserting

(a)	7	(7)
(b)	4	(4,7)
(c)	8	(4,7,8)
(d)	2	(2,4,7,8)
(e)	5	(2,4,5,7,8)
(f)	3	(2,3,4,5,7,8)
(g)	9	(2,3,4,5,7,8,9)

Phase 2 — polling

(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

Heap Sort

- Heap-sort:
  - Insertion — no more than $\log_2(n)$ steps
  - Deletion — no more than $\log_2(n)$ steps
- priority queue is implemented with a heap

- Time:
  - Add: $O(\log_2(n))$
  - Remove: $O(\log_2(n))$

Example

Phase 1 — Inserting

(a)	7	(7)
(b)	4	(4,7)
(c)	8	(4,7,8)
(d)	2	(2,4,8,7)
(e)	5	(2,4,8,7,5)
(f)	3	(2,4,3,7,5,8)
(g)	9	(2,4,3,7,5,8,9)

Phase 2 — polling

(a)	(2)	(3,4,7,5,8,9)
(b)	(2,3)	(4,5,7,9,8)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

Mini Homework

14, 6, 18, 2, 13, 7, 8, 9, 3, 17, 5, 10, 11, 12, 15, 19, 16, 0, 1, 4

For the data above, count the number of primitive operations for each of insertion, selection and heap sorts using the priority queues discussed. A primitive operation is: comparison, move an item in an array / arraylist. Show the count.

Also, show the contents of the queue when it contains all of the above items . Show the array or arraylist.

You may assume that the key and value are identical.