

CS231 Fall 2013 Lecture 21

Stephanie R Taylor

October 23, 2013

1 Administrative Topics

- Nothing special to note today.
- If I am correct, this is the first time I have had no administrative topics. But that is worth noting.
- So the administrative topic today is that today is special because we have no administrative topics.

2 Quick sort

The strategy of a quick sort is to recursively split the list into smaller pieces, one with all the “smaller” items and one with all the “larger” items, sort each list, then concatenate them together to form one long list. The algorithm is

```
define quicksort( array )  
    [small_elems , big_elems] = partition( array )  
    quicksort( small_elems )  
    quicksort( big_elems )  
    array = [small_elems , big_elems]
```

To decide which items are “smaller” and which are “larger”, each call chooses a “pivot” value. This pivot value marks the threshold between small and large.

One major advantage of quick sort over merge sort is that it can be done “in place”. The merge sort requires twice as much space (one to store the array, and a temporary array for doing the merging). For large data, this matters. To do the small-large partitioning in place, we identify pairs of elements that are in the wrong “halves” and swap their values.

2.1 The algorithm

Here is the code:

```
// sorts a[start] to a[end] in place
private static void quickSort(int [] a, int start, int end) {
    // partition the list
    int pivot = a[(start+end)/2];
    int i = start;
    int j = end;
    do {
        while (a[i] < pivot)
            i++; // skip over values already in the right half of the list
        while (pivot < a[j])
            j--; // skip over values already in the right half of the list
        if (i <= j) { //swap a[i] and a[j]
            // We have found a pair that are out of place, if we swap them, then
            // both will be in the correct "halves" of the list
            swapIntegers(a, i, j);
            i++;
            j--;
        }
    } while (i <= j); // once the indices cross,
    // we know we have examined every item in the list

    // do the recursive sorting
    if (start < j)
        quickSort(a, start, j);
    if (i < end)
        quickSort(a, i, end);
}
```

2.2 Analysis

Visualize the calls in a tree, like we did for merge sort.

Suppose we have an array of length n .

At the top level, of the tree, we examine each value once, and swap some of them. That is work proportional to n .

At the next level down, we have 2 calls to sort lists of length approximately $n/2$. So there are two calls, each doing work proportional to $n/2$.

At the next level down, we have approximately 4 calls to sort lists of length approximately $n/4$. So there are four calls, each doing work proportional to $n/4$.

And so on...

So each each level, we are doing work of $\Theta(n)$. Just like in merge sort, there will be approximately $\log n$ levels, which means the entire algorithm is $\Theta(n \log n)$.

2.3 Caveat

The sizes of the two sublists may not be even! At each stage, we pick a “pivot” point that we use as our threshold for determining which elements are “big” and which are “small”. There are multiple strategies for choosing the pivot value. One common method is random, another is simply choosing the value the is currently at the midpoint of the array we are sorting.

It could be the case, that we end up choosing the smallest value as the pivot. If we do that at each level, we will be splitting the list into a list of length 1 and a list of length $n - 1$. That would necessitate n levels and would cause the method to be $\Theta(n^2)$. Statistically, this isn’t likely. We don’t usually run into a pathological case like this. So we do on our analysis on the “average” case, and that is the $\Theta(n \log n)$ we talked about above.

For more on quick sort, see Wikipedia: <http://en.wikipedia.org/wiki/Quicksort>

3 Stacks and Queues

All data structures we have seen so far are kinds of collections (Array, ArrayList, LinkedList). The elements of a collection are not necessarily supposed to be ordered (from the user's perspective), but all the implementations we have seen are ordered somehow, although the ordering may not reflect anything useful (e.g. is the order of the links in a web page really important?). Sometimes order is less important. Sometimes other things are more important in a collection.

Consider a web browser with Back and Forward buttons. How can the browser implement this? It must save the previous URLs in order to be able to jump back. What kind of collection should it use to store the URLs? Should it be ordered? [yes] What other properties are necessary?

3.1 Stacks

A (pure) stack is a data structure with the following set of allowable operations:

1. create an empty stack
2. check whether the stack is empty
3. find the size of the stack (number of items in it)
4. push (add something to the top of the stack)
5. pop (remove the top item from the stack),
6. peek or top (look at the top of the stack without removing it).

LIFO discipline: last in, first out.

Called stacks because of similarity to real-world stack in which adding and removing things in the middle is hard.

Uses for stacks:

1. processing nested structure, like matching parentheses or palindromes or HTML tags.
2. in daily activities, people are often in the middle of something when something else comes along that must be dealt with first, e.g., work on computer, talk to visitor, answer the phone, deal with emergency or play game, write report, give lecture, answer question, emergency (pipe starts dripping acid)
3. run-time stack. The RT stack keeps track of which methods have invoked which other methods (it is a stack of frames or activation records) and so it behaves like a stack in that stack frames are pushed on and popped off. But note that elements of the RT stack can be accessed by methods other than push and pop.
4. reversing the cars in a train.

Interface:

```
public interface Stack {
    public int size();
    public boolean isEmpty();
    public Object top();
    public void push(Object o);
    public Object pop();
}
```

Questions:

- Should we be allowed to put null onto a stack of objects?
- Can we add the same object twice to the stack?
- Can a stack become full?
- Which methods can be implemented as convenience methods? [peek and isEmpty]
- What should happen if we pop or peek an empty stack?

We could do nothing or we could throw an Exception.

Exceptions are a built-in class that is used for exiting methods when an error occurs. E.g., NullPointerException. If an error occurs in a

method so that it can't continue, it can "throw" an exception to the method that called it. Think of it as the second method throwing back to the caller the excuse for why it can't do what was expected of it. `FullStackException` and `EmptyStackException` are just subclasses of `Exception` defined by the textbook authors.

3.2 Queues

One thing stacks are not good for: Storing a line of people waiting to get tickets, where the top of the stack is the next person to get a ticket. When a new person comes along, is it fair to push them on the top of the stack and so let them be next to get tickets?

Like a stack, queue can contain any object. The order of removal for a queue is the order of insertion (no priorities, no comparability requirement). A stack is LIFO and a queue is FIFO. The basic operations for queues are otherwise identical to those for stacks, except also for the standard names of `enqueue()` and `dequeue()` instead of `push` and `pop`.

Interface:

```
public interface Queue {
    public int size();
    public boolean isEmpty();
    public Object front() throws EmptyQueueException;
    public void enqueue(Object o);
    public Object dequeue() throws EmptyQueueException;
}
```

Applications of queues

- OS letting various processes run on one processor.
- simulation of real-world queues (Queueing theory) (standing in line)

3.3 Implementations of Stacks and Queues

Stack:

1. built-in `Stack` class in `java.util` package

2. Linked list – nice because adding to the beginning and removing from the beginning are easy
3. Array – use a index for the top of the Stack (next empty one or last filled one?) Copy when full.

Queue:

1. No built-in class exactly like a Queue, but some that are very close
2. Linked List – not as much fun, unless you keep a tail pointer (which is what we are doing in the project)
3. Array – Use it as a circular buffer.

3.4 Implementation of Stacks

I want to go over the details of implementations of stacks

1. using arrays, and
2. using lists.

```
public class ArrayStack implements Stack {
    private int count;
    private Object[] data;

    public ArrayStack() {
        count = 0;
        data = new Object[10];
    }

    public boolean isEmpty() {
        return (count == 0);
    }

    public int size() {
        return count;
    }

    public void push(Object obj) {
```

```

        if( count == data.length ) {
            // expand the array and copy over into it
            ....
        }
        data[count] = obj;
        count++;
    }

    public Object pop() throws EmptyStackException {
        if (count == 0) throw new EmptyStackException();
        count--;
        return data[count];
    }

    public Object peek() throws EmptyStackException {
        if (count == 0) throw new EmptyStackException();
        return data[count - 1];
    }
}

```

For the list implementation, we need a `ListNode` that has an `Object` as cargo:

```

public class ListNode {
    Object data;
    ListNode next;

    ListNode(Object d, ListNode n) {
        this.data = d;
        this.next = n;
    }
}

```

The instance variable here is called `topNode` and it just point to the element on the top of the stack:

```

public class NodeStack implements Stack {
    private Node topNode;

    public Stack() {
        topNode = null;
    }
}

```

The constructor creates an empty stack by initializing the instance variable to null. So naturally we can test for an empty stack as so...

```

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

```

```
}
```

Now, for putting things on and taking things off the list, we could have used the list methods we developed in the last week:

push is the same as add:

```
public void push(Object obj) {  
    topNode = new Node (obj, topNode);  
}
```

pop is the same as removeFirstNode, except that we have to play some games with a temporary variable:

```
public Object pop() {  
    if (topNode == null) throw new EmptyStackException();  
    Object ret = topNode.data;  
    topNode = topNode.next;  
    return ret;  
}
```

peek is a no-brainer.

```
public Object peek() {  
    if (topNode == null) throw new EmptyStackException();  
    return topNode.item;  
}
```