1 Lower bound on runtime of comparison sorts

So far we've looked at several sorting algorithms – insertion sort, merge sort, heapsort, and quicksort. Merge sort and heapsort run in worst-case $O(n \log n)$ time, and quicksort runs in expected $O(n \log n)$ time. One wonders if there's something special about $O(n \log n)$ that causes no sorting algorithm to surpass it.

As a matter of fact, there is! We can prove that any comparison-based sorting algorithm must run in at least $\Omega(n \log n)$ time. By comparison-based, I mean that the algorithm makes all its decisions based on how the elements of the input array compare to each other, and not on their actual values.

One example of a comparison-based sorting algorithm is insertion sort. Recall that insertion sort builds a sorted array by looking at the next element and comparing it with each of the elements in the sorted array in order to determine its proper position.

Another example is merge sort. When we merge two arrays, we compare the first elements of each array and place the smaller of the two in the sorted array, and then we make other comparisons to populate the rest of the array.

In fact, all of the sorting algorithms we've seen so far are comparison sorts. But today we will cover counting sort, which relies on the values of elements in the array, and does not base all its decisions on comparisons. This algorithm runs in $O(n)$ time, in contrast to the comparison sorts that run in $\Omega(n \log n)$.

1.1 Decision trees

The basic idea is that sorting algorithms are being asked to do something rather complicated. Given an unsorted array, the algorithm must decide how to permute the array to produce sorted output. Because there are $n!$ possible orderings of an array with n elements, a comparison based sorting algorithm may be asked to do $n!$ different things. The decisions must be based on comparisons, which each provide 1 bit of information. To accommodate $n!$ possible outcomes, we need at least $\log(n!)$ bits of information, which as we will see, is roughly $n \log n$.

Another way to describe this uses decision trees:

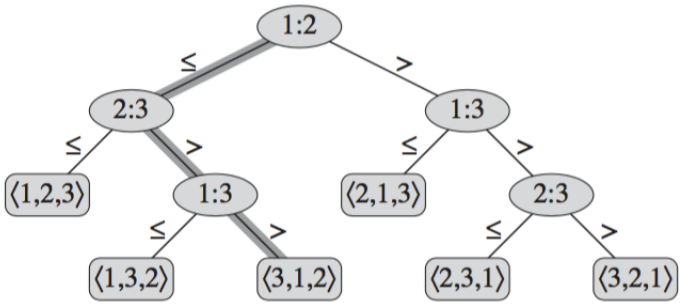

Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i:j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, and so the decision tree must have at least 6 leaves.

Any algorithm uses if statements to decide which lines of code to execute. In comparison sorts, the if statements correspond to comparisons between elements, and if element i is less than element j, the algorithm might do one thing, while if element i is greater than element j, the algorithm might do another thing.

We can track the progress of the sorting algorithm by looking at different branches of the decision tree. The tree in Figure 8.1 starts off by comparing the first and second elements. If the first element is smaller, it compares the second and third elements, while if the first element is bigger, it compares the first and third elements. If it took the left branch before and then discovers that the second element is less than the third element, then it knows the sorted order, and outputs $A[1] \leq A[2] \leq A[3]$.

If the comparisons had turned out differently, we would get other sorted orders – if the algorithm took the right branch every time, it would output $A[3] \leq A[2] \leq A[1]$.

1.2 Lower bound

Now in order for a comparison sort to be correct, the leaves of the decision tree must contain all possible permutations of the input array. This is because the input can be in any possible order, so the algorithm must be able to output all possible rankings of the input elements.

There are $n!$ permutations of the input array, so the decision tree must have at least $n!$ leaves. Therefore, the height of the decision tree must be $\Omega(\log(n!))$.

1.2.1 Why the height of a binary tree with m leaves is $\Omega(\log m)$

If a tree has height h, it must have at most 2^h leaves. This is because each leaf corresponds to a unique path from the root of the tree to the bottom of the tree. At each junction, there are at most two directions the path can take. Since there are at most h choices to be made, and each choice increases the number of possible paths by (at most) a factor of 2, there are at most 2^h paths to the bottom of the tree, so the tree has at most 2^h leaves.

Therefore, if the tree has m leaves, its height is at least $\log m$.

1.2.2 Stirling's approximation

From this we know that the height of the decision tree is $\Omega(\log(n!))$. Stirling's approximation states that

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$$

We will not derive this in this class, but it is equation 3.18 in the textbook.

Taking logarithms, we get

$$\begin{aligned}\log(n!) &= \log\left[\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)\right]\\ &= \frac{1}{2}\log(2\pi n) + n\log(n/e) + \log(1+\Theta(1/n))\\ &\geq n\log n - n\log e\\ &= \Omega(n\log n)\end{aligned}$$

since the expression exceeds $\frac{1}{2}n\log n$ for sufficiently large n.

2 Counting sort

Now we will look at some non-comparison-based sorting algorithms.

Input: Counting sort is a sorting algorithm that relies on the values in the input array. Specifically, it assumes that the inputs are integers in the range 0 to k. (Counting sort may also be used on key-value pairs where the keys are integers in the range 0 to k and the values are other types of data, and we are sorting by key.)

Runtime: $\Theta(n+k)$.

Warning: If $k = O(n)$, the algorithm is also $O(n)$. But a single large value (say n^2) is enough to destroy the linear runtime of the algorithm. I am warning you of this because it's easy to think "counting sort is linear so we should use it all the time."

2.0.1 Stability

Counting sort is a **stable sort**. This means that if we are sorting by key, and two array elements have the same key, then they appear in the output in the same order that they appeared in the input.

Example: Suppose we're sorting this array using the numbers as keys:

```
[(2, "Carol"), (1, "David"), (2, "Alice"), (3, "Bob")]
```

Then a stable sorting algorithm would produce

```
[(1, "David"), (2, "Carol"), (2, "Alice"), (3, "Bob")]
```

It would not produce

```
[(1, "David"), (2, "Alice"), (2, "Carol"), (3, "Bob")]
```

2.1 Naive version of counting sort

This version of counting sort only works when the inputs are pure integers, and not when they are key-value pairs.

Note: In what follows, `Counts` will be a zero-indexed array. For whatever reason, the textbook decided to allow zero-indexed arrays in this one chapter.

1. Go through the array, counting how many of each element there are. Keep these tallies in the `Counts` array, where `Counts[i]` is the number of elements equal to `i`.
2. Use the `Counts` array to populate the output array. For example, if `Counts[1] = 5` and `Counts[2] = 7`, then first repeat `1` five times, and then repeat `2` seven times.

Example: Let `A = [2, 5, 3, 0, 2, 3, 0, 3]`. Then `Counts = [2, 0, 2, 3, 0, 1]`. `Counts` gets populated as follows:

```
[0, 0, 1, 0, 0, 0]
=> [0, 0, 1, 0, 0, 1]
=> [0, 0, 1, 1, 0, 1]
=> etc
```

Once we have `Counts`, we can repeat `0` two times, then repeat `1` zero times, then repeat `2` two times, etc, until we have `[0, 0, 2, 2, 3, 3, 3, 5]`.

2.1.1 Code

```
# Populate the Counts array
Create the array Counts[0..k]
for i = 0 to k:
    Counts[i] = 0
```

```
for j = 1 to A.length:
    Counts[A[j]] = Counts[A[j]] + 1

# Use the Counts array to populate the final output
idx = 1
for i = 0 to k:
    for j = 1 to Counts[i]:
        Answer[idx] = i
        idx = idx + 1
```

2.1.2 Runtime

Creating and initializing the `Counts` array takes $\Theta(k)$ time. Populating it with useful values takes $\Theta(n)$ time. Populating the output array takes $\Theta(k+n)$ time, because

1. The body of the inner loop runs n times total across the entire algorithm
2. The inner loop test has to run one extra time for each iteration of the outer loop, which costs $k+1$ extra operations. (The outer loop test also has to run $k+1$ times.)

The second point seems kind of silly, but suppose our array was like

`[0, 1000, 0, 1000, 0, 1000, 0, 1000]`

Then we would have to spend a substantial amount of time handling the outer loop, even though we only enter the inner loop twice.

Therefore, the entire algorithm takes $\Theta(k+n)$.

2.2 Full version of counting sort

The real version of counting sort uses a similar idea, but

1. It also has to work on key-value pairs.
2. We want it to be stable.

2.2.1 Idea

1. Once we populate the `Counts` array (denoted `C` in the pseudocode), we find the cumulative sum of the array. For instance, if `Counts = [2, 0, 2, 3, 0, 1]`, then we create `CumSum = [2, 2, 4, 7, 7, 8]`.
2. Note that if `CumSum[key] = 7` and `CumSum[key - 1] = 4`, then the fifth, sixth, and seventh elements in the sorted array should be equal to `key`. In this case, our array is `A = [2, 5, 3, 0, 2, 3, 0, 3]`, so the fifth, sixth, and seventh elements should be `3`, as predicted by `CumSum`.

3. Now we iterate backwards through the keys of the input array. When we see a key `A[j].key`, we place that key-value pair in the location specified by `CumSum`, and decrease the value of `CumSum[A[j].key]`.
4. The reason we decrease the value of `CumSum[A[j].key]` is because if we see another entry with the same key, we want to place that entry immediately left of the previous entry (note that we are iterating through the entries backwards).
5. Note that this guarantees that our algorithm is a stable sort.
6. Note that the values in `CumSum` don't get decreased to zero – they only get decreased exactly the number of times that the key is seen in the array. At the end of the algorithm, `CumSum` is garbage.

2.2.2 Algorithm

Here is the pseudocode, written to account for the presence of key-value pairs. The original algorithm in CLRS (which is also printed below) is only written for the case of an array of integers.

```
# Populate the counts array
let Counts[0..k] be a new array
for i = 0 to k:
    Counts[i] = 0
for j = 1 to A.length:
    Counts[A[j].key] = Counts[A[j].key] + 1

# Find the cumulative sum
let CumSum[0..k] be another new array, of all zeroes
CumSum[0] = Counts[0]
for i = 1 to k:
    CumSum[i] = Counts[i] + CumSum[i - 1]
for j = A.length down to 1:
    Answer[CumSum[A[j].key]] = A[j]
    CumSum[A[j].key] = CumSum[A[j].key] - 1
```

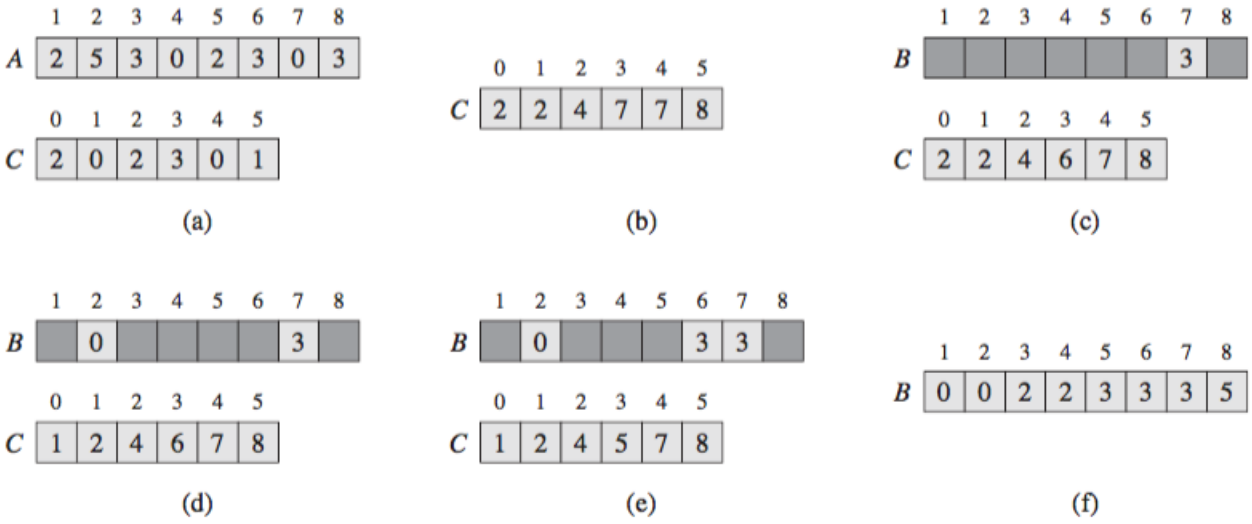

Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

2.2.3 Runtime

The first and third for loop take $\Theta(k)$ time, and the other two for loops take $\Theta(n)$ time. So overall the algorithm takes $\Theta(k + n)$ time. When $k = O(n)$, this beats the comparison sort lower bound of $\Omega(n \log n)$, because it is not a comparison sort. Counting sort uses the values of the elements, instead of relying on comparisons between them.

3 Elementary data structures

3.1 Stacks and queues

A stack is like a stack of plates. You can "push" elements onto the stack (which is like putting plates on top of the stack), or you can "pop" elements off the stack (which is like taking plates off the top of a stack). Stacks follow a "last-in, first-out" policy, which means that if you pop an element off the stack, it's always the last element that you pushed onto the stack.

A queue is like a queue of people, like the line at the ice cream shop. You can insert elements into a queue (which is like a person entering at the back of the line), and you can remove elements from the queue (by helping the person at the front of the line). Queues follow a "first-in, first-out" policy, which means that if you remove an element from the queue, it will always be the element that has been in the queue for the longest time.

3.1.1 Implementation of a stack

A stack S (with at most n elements) can be implemented as an array of size n. The array has a pointer $S.top$ to the top of the stack, which gets incremented when elements are pushed onto the stack, and decremented when elements are "removed" from the stack.

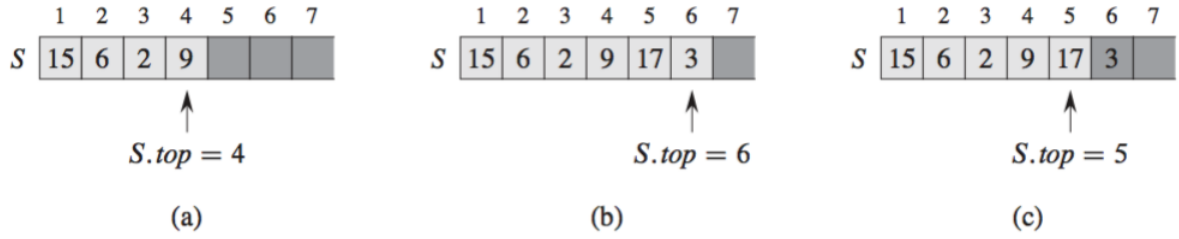

Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH$(S, 17)$ and PUSH$(S, 3)$. **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

Here is the pseudocode for the Push and Pop operations. Each of these operations takes $O(1)$ time.

```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else return FALSE

PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x

POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

3.1.2 Implementation of a queue

A queue Q of at most $n-1$ elements can also be implemented as an array of size n. Queues are a little trickier because we have to keep track of both the head and the tail.

We can implement the queue in a circular fashion, where the queue begins at some place in the array, and then "wraps around" to the first position in the array when there are too many elements. As elements are pushed onto the queue, the $Q.tail$ pointer gets incremented, and as elements are popped from the queue, the $Q.head$ pointer gets incremented. (These pointers get moved to the other side of the array when they go "off the edge.") As a result, the queue "travels" around the array in a circle.

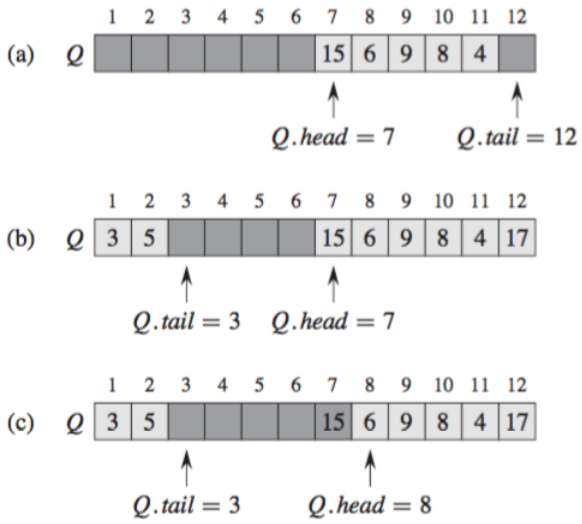

Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, and ENQUEUE$(Q, 5)$. **(c)** The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3       Q.tail = 1
4  else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3       Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

3.2 Linked lists

A linked list L is a data structure full of objects where each object (x) carries a pointer to the next one ($x.next$). (In a doubly linked list, each object also carries a pointer to the previous one, $x.prev$.) The head of the list ($L.head$) is the first element of the list, and $L.head.prev = NIL$.

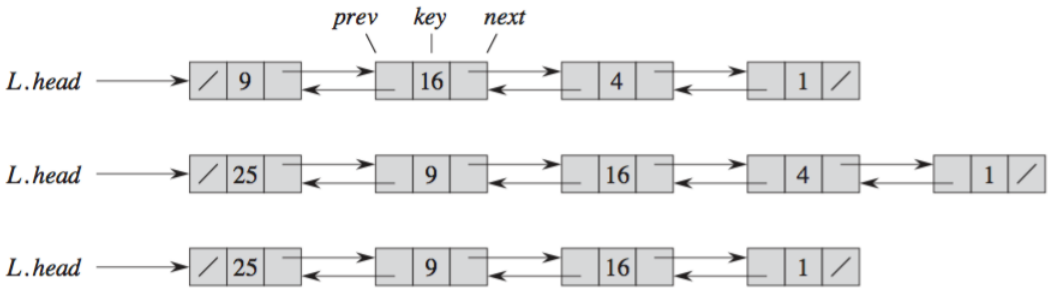

Figure 10.3 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. **(b)** Following the execution of LIST-INSERT(L, x), where $x.key = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. **(c)** The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

To search for an element in a linked list, we simply traverse the list until we find the element. This takes $\Theta(n)$ in the worst case, since we may have to search the entire list.

To insert an element into a linked list, we place the element at the head of the list, and update the "head" pointer. This takes $O(1)$ time.

To delete an element from the linked list, we update the pointers in the linked list so that the element's predecessor now points directly to the element's successor, thus "splicing" the element out of the linked list.

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5      x.next.prev = x.prev
```

3.2.1 Sentinels

One way to simplify the code is placing a "sentinel" element $L.nil$ between the head and tail of the list, and making the list circular. (Now we no longer need a head pointer, and the head and tail of the list are no longer special cases in the Insert and Delete operation.)

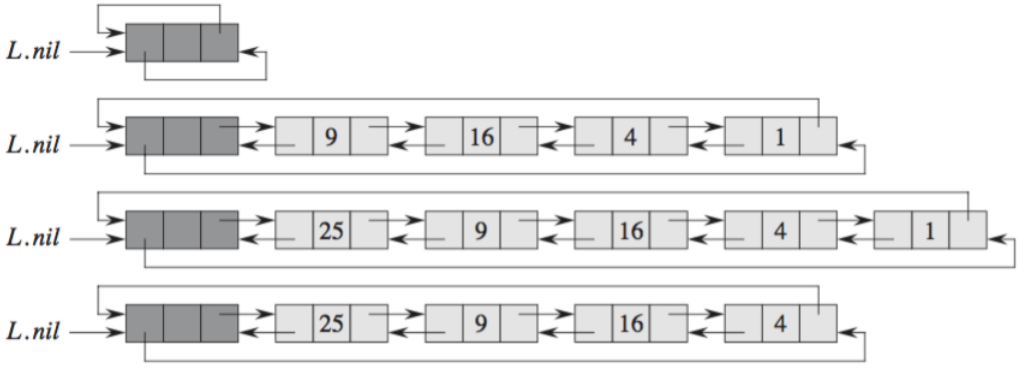

Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel $L.nil$ appears between the head and tail. The attribute $L.head$ is no longer needed, since we can access the head of the list by $L.nil.next$. **(a)** An empty list. **(b)** The linked list from Figure 10.3(a), with key 9 at the head and key 1 at the tail. **(c)** The list after executing LIST-INSERT$'(L, x)$, where $x.key = 25$. The new object becomes the head of the list. **(d)** The list after deleting the object with key 1. The new tail is the object with key 4.

```
LIST-DELETE'(L, x)
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

```
LIST-INSERT'(L, x)
1  x.next = L.nil.next
2  L.nil.next.prev = x
3  L.nil.next = x
4  x.prev = L.nil
```

```
LIST-SEARCH'(L, k)
1  x = L.nil.next
2  while x ≠ L.nil and x.key ≠ k
3      x = x.next
4  return x
```

Okay, we didn't simplify the code *that* much, but the concept of sentinels is going to be a lot more useful when we start looking at more complicated data structures, such as red-black trees.