Chapter 1

Introduction to algorithm design

n/a

Chapter 2

Algorithm analysis

Notes

The dominance pecking order:

$$n! \gg c^n \gg n^3 \gg n^2 \gg n^{1+\epsilon} \gg n \log n \gg n \gg \sqrt{n} \gg$$
$$\log^2 n \gg \log n \gg \log n / \log\log n \gg \log\log n \gg \alpha(n) \gg 1$$

Solutions

2-10

(a) $f(n) = (n^2 - n)/2,\ g(n) = 6n.$

Is $f(n) = O(g(n))$? If so, there is c such that $f(n) \leq cg(n)$ for sufficiently large n.

$$\frac{1}{2}\left(n^2 - n\right) \leq 6n \ \rightarrow\ n^2 - n \leq 12n \ \rightarrow\ n(n-1) \leq 12n$$

Suppose there is such a c, then

$$n(n-1) \leq 12cn \ \rightarrow\ n - 1 \leq 12c$$

Clearly we can always find n such that this inequality won't hold, so $f(n) \neq O(g(n))$.

Is $g(n) = O(f(n))$? If so, there is c such that $g(n) \leq cf(n)$ for sufficiently large n.

$$6n \leq \frac{1}{2}\left(n^2 - n\right) \ \rightarrow\ 12n \leq n^2 - n = n(n-1) \ \rightarrow\ 12 \leq n - 1 \ \rightarrow\ 13 \leq n.$$

So with $c = 1$ the inequality will hold for $n_0 \geq 13$, and $g(n) = O(f(n))$.

(b) $f(n) = n + 2\sqrt{n},\ g(n) = n^2$.

$f(n) = O(g(n)) \Leftrightarrow f(n) \le cg(n)$ for sufficiently large n.

$$\begin{gathered} n + 2\sqrt{n} \le cn^2,\ \text{with } c = 1, \\ n + 2\sqrt{n} \le 2n \text{ for } n > 4, \\ 2n \le n^2 \text{ so } f(n) = O(g(n)). \end{gathered}$$

$g(n) = O(f(n)) \Leftrightarrow g(n) \le cf(n)$ for sufficiently large n. But this asks to find c such that $n^2 \le c\,(n + 2\sqrt{n})$; since ultimately $n^2 \gg n$, $g(n) \ne O(f(n))$.

(c) $f(n) = n\log n,\ g(n) = n\sqrt{n}$.

$$\begin{aligned} f(n) = O(g(n)) \Leftrightarrow n\log n &\le cn\sqrt{n},\ \text{with } c = 1, \\ \rightarrow\ \log n &\le \sqrt{n/2}, \end{aligned}$$

since $\sqrt{n} \gg \log n,\ f(n) = O(g(n))$.

By the same argument, $g(n) \ne O(f(n))$.

(d) $f(n) = n + \log n,\ g(n) = \sqrt{n}\ \rightarrow\ n + \log n \le c\sqrt{n}$, and since $n \gg \sqrt{n}$, any constant factor will be dominated by the linear term, so $f(n) \ne O(g(n))$. Conversely and by the same argument, $g(n) = O(f(n))$.

(e) $f(n) = 2\left(\log n\right)^2,\ g(n) = \log n + 1$. Note that $2\left(\log n\right)^2 = 2\log^2 n$, and $\log^2 n \gg \log n$, so $g(n) = O(f(n))$ and $f(n) \ne O(g(n))$.

(f) $f(n) = 4n\log n + n,\ g(n) = \left(n^2 - n\right)/2$. We know that $n\log n \gg n$, so we can consider just this term from $f(n)$. But ultimately the quadratic term in $g(n)$ dominates so $f(n) = O(g(n))$.

2-11

(a) $f(n) = 3n^2,\ g(n) = n^2$.

With $c = 3,\ f(n) \le 3g(n)$ so $f(n) = O(g(n))$.

$f(n) = \Omega(n) \Leftrightarrow cg(n) \le f(n)$ for sufficiently large n. For $c = 1$ the inequality holds, so $f(n) = \Omega(g(n))$ and $f(n) = \Theta(g(n))$.

(b) $f(n) = 2n^4 - 3n^2 + 7,\ g(n) = n^5$.

$n^5 \gg n^4$ so $f(n) = O(g(n))$ and $f(n) \ne \Omega(g(n))$.

(c) $f(n) = \log n,\ g(n) = \log n + \frac{1}{n}$.

$\lim_{n\to\infty} \frac{1}{n} = 0$, so as $n \to \infty$, $f(n) - g(n) = 0$. So no function dominates the other. Thus, $f(n) = \Theta(g(n))$.

(d) $f(n) = 2^{k \log n},\ g(n) = n^k$.

$$\begin{aligned} f(n) = O(g(n)) &\Leftrightarrow f(n) \leq cg(n) \\ &\rightarrow\ 2^{k \log n} \leq cn^k;\ \text{taking logarithms,} \\ &\rightarrow\ \log\left(2^{k \log n}\right) \leq \log\left(cn^k\right) = \log c + \log n^k \\ &\rightarrow\ k \log n \log 2 \leq \log c + k \log n. \end{aligned}$$

Ignoring constant terms and multiplicative constants, we are left with $\log n \leq \log n$, so $f(n) = \Theta(g(n))$.

(e) $f(n) = 2^n,\ g(n) = 2^{2n}$.

$2^n \leq c2^{2n}$ clearly holds for $c = 1$, so $f(n) = O(g(n))$.

$c2^{2n} \leq 2^n$? Well, $2^{2n} = 2^2 \cdot 2^n = 4 \cdot 2^n$, so $4c2^n \leq 2^n$ is satisfied with $c = 1/4$. So $f(n) = \Omega(g(n))$ and finally, $f(n) = \Theta(g(n))$.

2-12 $n^3 - 3n^2 - n + 1 = \Theta\big(n^3\big)$.

Note that $0 \leq 3n^2 + n \rightarrow n^3 \leq n^3 + 3n^2 + n \rightarrow n^3 - 3n^2 - n \leq n^3$. Thus $f(n) = O\left(n^3\right)$.

Now $cn^3 \leq n^3 - 3n^2 - n + 1$. Consider $c = 1/2$, then

$$\begin{aligned} n^3/2 &\leq n^3 - 3n^2 - n + 1 \\ -n^3/2 &\leq -3n^2 - n + 1 \\ -n^3 &\leq -6n^2 - 2n + 2 \\ n^3 &\geq 6n^2 + 2n - 2. \end{aligned}$$

This holds for $n_0 \geq 7$, so $f(n) = \Omega\big(n^3\big)$ and finally $f(n) = \Theta\big(n^3\big)$.

2-13 $f(n) = n^2 = O\left(2^n\right) \Leftrightarrow f(n) \leq c2^n$, after some n_0. For $c = 1$,

$$\begin{aligned} n^2 &\leq 2^n \\ \log(n^2) &\leq \log(2^n) \\ 2 \log n &\leq n \log 2 \\ \log n &\leq kn,\ k = \frac{\log 2}{2} \end{aligned}$$

Since $\log n \ll n$ this inequality holds for large enough n, and $n^2 = O(2^n)$.

2-14 $\Theta\big(n^2\big) = \Theta\big(n^2 + 1\big)$? This is to say whether both classes are the same. We can show this by assuming we have $f(n) = \Theta\big(n^2\big)$, and proving that $f(n) = \Theta\big(n^2 + 1\big)$, and likewise with the other assumption.

First, $f(n) = \Theta\big(n^2\big) \Rightarrow f(n) = O\big(n^2\big)$ and $f(n) = \Omega\big(n^2\big)$. Is $f(n) = O\big(n^2 + 1\big)$? This would mean $f(n) \leq c\left(n^2 + 1\right)$, for $n > n_0$. Since $f(n) =$

$O(n^2)$ there is c_0 such that $f(n) \le c_0 n^2$, but $c_0 n^2 \le c_0 \left(n^2+1\right)$ so with the same c_0 we see that $f(n) = O\left(n^2+1\right)$.

Now we want to show that $f(n) = \Omega\left(n^2+1\right)$, that is, $c\left(n^2+1\right) \le f(n)$ for $n > n_0$. Since $f(n) = \Omega\left(n^2\right)$ there are c_1, n_1 such that $c_1 n^2 \le f(n)$, for $n > n_1$. In particular this holds for $n+1 > n_1$, so

$$c_1 n^2 < c_1 (n+1)^2 \le f(n), \quad n > n_1.$$

But note that $n^2 + 1 \le (n+1)^2$, so for $n_0 = n_1 + 1$ above, we have that $c_1\left(n^2+1\right) \le c_1(n+1)^2 \le f(n)$ for $n > n_0$. So $f(n) = \Omega\left(n^2+1\right)$ and thus $f(n) = \Theta\left(n^2+1\right)$.

Now we need to show that $f(n) = \Theta\left(n^2+1\right) \Rightarrow f(n) = \Theta\left(n^2\right)$.

$f(n) = \Omega\left(n^2\right) \Leftrightarrow cn^2 \le f(n)$ for $n > n_0$. We know that $f(n) = \Omega\left(n^2+1\right)$ so there is a c_1 such that $c_1\left(n^2+1\right) \le f(n)$ for $n > n_1$; since $n^2 < n^2+1$, by letting $c = c_1$ we see that $c_1 n^2 < c_1\left(n^2+1\right) \le f(n)$, so $f(n) = \Omega\left(n^2\right)$.

$f(n) = O\left(n^2\right) \Leftrightarrow f(n) \le cn^2$ for $n > n_0$. Since $f(n) = O\left(n^2+1\right)$ we know there are c_1 and n_1 such that $f(n) \le c_1\left(n^2+1\right)$ for $n \ge n_1$; in particular, $f(n) \le c_1\left(n_1^2+1\right)$. Then let $c = c_1\left(n_1^2+1\right)$, then $cn^2 > c_1\left(n_1^2+1\right) \ge f(n)$, for $n > n_1$. Thus $f(n) = O\left(n^2\right)$ and $f(n) = \Theta\left(n^2\right)$.

This shows that if a function is $\Theta\left(n^2\right)$ then it must be $\Theta\left(n^2+1\right)$ and viceversa—that is, both classes are the same.

2-17

a) $f(n) = n^2 + n + 1$, $g(n) = 2n^3$. We want to find $c > 0$ such that $f(n) \le cg(n)$ for $n > 1$. $f(2) = 7$; $g(2) = 16 \to c = 1$.

b) $f(n) = n\sqrt{n}$, $g(n) = n^2$. $n\sqrt{n} < n^2 \to 2n^2 > n\sqrt{n} + n^2 \to c = 2$.

c) $f(n) = n^2 - n + 1$, $g(n) = n^2/2$. $n^2 - n + 1 < \frac{c}{2}n^2 \to$ if $n = 2$, $f(2) = 3$, and $g(2) = 2$. For $c = 2$, $n^2 - n + 1 < n^2$.$(f(2) = 3, 2g(2) = 4)$.

2-18 Let $f_1(n) = O(g_1(n))$, $f_2(n) = O(g_2(n))$. Show that $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.

Proof. There are c_1, n_1 such that $f_1(n) \le c_1 g_1(n)$ for $n > n_1$, and c_2, n_2 such that $f_2(n) \le c_2 g_2(n)$ for $n > n_2$. Let $c = \max(c_1, c_2)$, $n_0 = \max(n_1, n_2)$. Then $f_1(n) < cg_1(n)$, and $f_2(n) < cg_2(n)$, which implies that $f_1(n) + f_2(n) < cg_1(n) + cg_2(n) = c(g_1(n) + g_2(n))$ for $n > n_0$. □

2-19 Let $f_1(n) = \Omega(g_1(n))$, $f_2(n) = \Omega(g_2(n))$. Then there are c_1, n_1 such that $f_1(n) \ge c_1 g_1(n)$ for $n > n_1$, and c_2, n_2 such that $f_2(n) \ge c_2 g_2(n)$ for $n > n_2$.

Let $n_0 = \max(n_1, n_2)$ and let $c_0 = \min(c_1, c_2)$. Then $cg_1(n) \leq c_1 g_1(n)$ and also $cg_2(n) \leq c_2 g_2(n)$. These inequalities imply that

$$cg_1(n) \leq f_1(n), \ n > n_0,$$
$$cg_2(n) \leq f_2(n), \ n > n_0.$$

Thus $c(g_1(n) + g_2(n)) \leq f_1(n) + f_2(n), \ n > n_0$, and $f_1(n) + f_2(n) = \Omega(g_1(n) + g_2(n))$.

2-20 Let $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. Then there are c_1, n_1, c_2, n_2 such that $f_1(n) \leq c_1 g_1(n)$ for $n > n_1$, and $f_2(n) \leq c_2 g_2(n)$ for $n > n_2$. Let $c = \max(c_1, c_2)$ and $n_0 = \max(n_1, n_2)$. Then $f_1(n) \leq cg_1(n)$ and $f_2(n) \leq cg_2(n)$ for $n > n_0$, which implies that $f_1(n) \cdot f_2(n) \leq c(g_1(n) \cdot g_2(n))$. Thus $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.

2-21 We are to prove that $p(n) = a_k n^k + \ldots + a_0 = O\big(n^k\big)$, for $k \geq 0$ and arbitrary real coefficients a_i. This is to say that we can find a $c > 0$ and n_0 such that $p(n) \leq cn^k$, for all $n > n_0$.

Proof. We know that after some n, $a_k n^k > p(n) - a_k n^k$, that is to say, the leading order term will come to dominate. To see this, note that

$$\lim_{n \to \infty} \frac{n^m}{n^q} = 0 \Leftrightarrow q > m,$$

is to say that $n^q \gg n^m$.

Now take $c = \max(a_0, \ldots, a_k)$. Then clearly $cn^k + \ldots + c \geq a_k n^k + \ldots + a_0 = p(n)$. But because of the same argument as above, there is some n_0 after which $cn^k > cn^{k-1} + \ldots + c$. Then it follows that $p(n) \leq cn^k$ for $n > n_0$, which means $p(n) = O\big(n^k\big)$. $\square$

2-22 Let $a, b \in \mathbb{R}$, with $b > 0$. Show that $(n + a)^b = \Theta\big(n^b\big)$.

Proof. To prove that $(n + a)^b = O\big(n^b\big)$ we need to find c_0 such that $c_0 n^b \geq (n + a)^b$ for sufficiently large n. If $a = 0$, $(n + a)^b = n^b$ and with $c = 1$ the inequality holds. If $a < 0, (n + a)^b \leq n^b$, so

$$\frac{(n + a)^b}{n^b} = \left(\frac{n + a}{n}\right)^b < 1.$$

If $a > 0$ we need to see that $\big(1 + \frac{a}{n}\big)^b$ is bounded. (To see why, $(n+a)^b \leq cn^b \to (n + a/n)^b \leq c \to (1 + a/n)^b \leq c$).

Note that $\frac{a}{n}$ can get arbitrarily close to 0 for large enough n, so the entire expression tends to 1 as $n \to \infty$. Thus for, say, $c = 2$ it is possible to find n_0 large enough such that $(n + a)^b \leq 2n^b$. So $(n + a)^b = O\big(n^b\big)$.

To prove that $(n + a)^b = \Omega\big(n^b\big)$, by the same argument, $cn^b \leq (n + a)^b \to c \leq (n + a/n)^b = (1 + a/n)^b$. For $a \neq 0$ the expression $1 + a/n$ tends to 1, either "from below" or "from above"—at any rate, by taking some $c < 1$, say, $1/2$, it will be possible to find n large enough that the inequality will hold. Thus $(n+a)^b = \Omega\big(n^b\big)$, and finally $(n+a)^b = \Theta\big(n^b\big)$. $\square$

2-27

(a) $f(n) = o(g(n))$ and $f(n) \neq \Theta(g(n))$. If $f(n) = o(g(n))$ then $g(n) \gg f(n)$—they are on different classes. But $f(n) \neq \Theta(g(n))$ implies that either $f(n) \neq \Omega(g(n))$ or $f(n) \neq O(g(n))$. Take $f(n) = n$, $g(n) = n^2$. Clearly $f(n) \neq \Omega(g(n))$ but $g(n) \gg f(n)$ so $f(n) = o(g(n))$.

(b) $f(n) = \Theta(g(n))$, $f(n) = o(g(n))$. If $f(n) = \Theta(g(n))$ then $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, that is, they are in the same class. So it's not possible for both $f(n) = \Omega(g(n))$ and $f(n) = o(g(n))$ to hold simultaneously.

(c) $f(n) = \Theta(g(n))$ and $f(n) \neq O(g(n))$. None, by definition.

(d) $f(n) = \Omega(g(n))$ and $f(n) \neq O(g(n))$. $f(n) = n^2$, $g(n) = n$.

2-29

(a) $f(n) = n^2 + 3n + 4$, $g(n) = 6n + 7 \to f(n) = \Omega(g(n))$.

(b) $f(n) = n\sqrt{n}$, $g(n) = n^2 - n \to f(n) = \Omega(g(n))$.

(c) $f(n) = 2^n - n^2$, $g(n) = n^4 + n^2 \to f(n) = \Omega(g(n))$.

2-30

(a) Yes—$O\big(n^2\big)$ worst case time doesn't necessarily mean it ever takes n^2 steps on any input. If the algorithm was $O(n)$ worst case, it would still be $O\big(n^2\big)$.

(b) Yes—$O\big(n^2\big)$ only talks about an upper bound.

(c) Yes. It could be that the best case time complexity of the algorithm is $O(n)$.

(d) No. Some inputs will trigger worst case behavior so will necessarily be $\Theta\big(n^2\big)$.

(e) Yes. We ignore multiplicative constants and terms of lower degree.

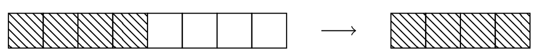
Chapter 3

Data structures

Solutions

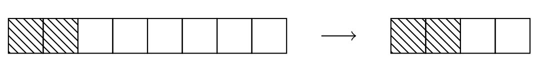
3-5

a) Suppose the array has size 2^n. This underflow strategy has us release the top half of our allocated memory when we come down to 2^{n-1} items.



If our program now adds an item to the dynamic array the same space needs to be allocated that was just freed, potentially moving all 2^{n-1} items. Imagine this delete-one/add-one cycle repeats, such as may happen with a stack backed by this dynamic array. Each append-one operation copies the (now) lower half to a new location, taking linear time on the number of items.

b) The problem with that underflow strategy is that both the “grow” and “shrink” events are at the same threshold—when half full, shrink by half thus making it full again, but this guarantees the next append will trigger a “grow”. If the two thresholds are dissociated, we will avoid this pathological behavior. For this, only shrink the array to half size when it is a fourth full.



We are then at the situation after a grow operation just duplicated the size of the array for us, thus amortizing the cost of grow/shrink operations.

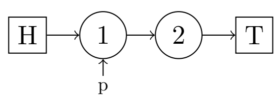
3-6 Skiena's fridge works as a stack, so unless he takes care of unstacking every food item regularly (thus emptying the fridge), that is bad news for the first items inserted.

One improvement might be to use a queue, whatever has been in there the longest is consumed first. However that is still a naive strategy—if an item expiring tomorrow is inserted after one that will expire in a year, I still risk the last item expiring.

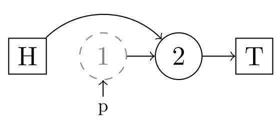
The answer is a *priority queue*. Prioritize the item that expires next. This ensures that items that have longer expiration dates wait the most.

3-7 The book says to keep a sentinel for the end of the list. There are three cases for delete: 1) delete the first node; 2) delete a node in the middle; 3) delete the last node.

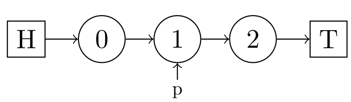
1) Deleting the head.



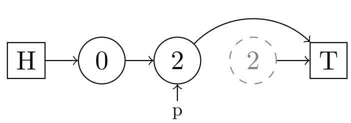
`list.head == p`, point `H` to `p->next`, free `p`



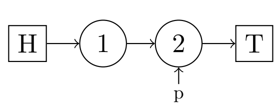
2) Deleting a node in the middle of the list.



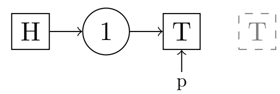
`list.head != p`, `p->next != tail`, overwrite `p` with the content of `p->next`, then free `p->next`.



3) Deleting the last node.



p->next == tail, free tail and make p the new sentinel node.



3-12 Maximum depth of a tree:

```
data Tree = Node Tree Tree
          | Nil

maxDepth :: Tree -> Int
maxDepth Nil = 0
maxDepth (Node l r) = 1 + max (maxDepth l) (maxDepth r)
```

3-14 Merging two binary search trees into a doubly linked list. The in order traversal of each tree provides ordered lists of their respective elements. A linear sweep, choosing the minimum of each head and advancing in the chosen list will merge the two structures. The cost of both the traversal and sweep/merge stages is $O(m+n)$, where m and n are the number of elements in each binary search tree.

Note that this is fundamentally the idea behind mergesort.

3-15 There is presumably an insertion order that guarantees height balance. Knowing all the elements and having them in order is a great advantage. Some thoughts: clearly the extremes have to be inserted among the last few elements–their levels have to be created for them to land in the right place. Also, the median element of the array has to be the root, as it gives the most "space" to its sides.

That median element determines two halves. Each half has its own median element to which the same thinking process applies. Therefore a potential algorithm for building a height balanced tree is:

```
balanced_insertion(lo, hi):
  m <- median(lo, hi)
  insert element m
  balanced_insertion(lo, m-1)
  balanced_insertion(m+1, hi)
```

With some provisions for ranges of length 1, in which case the element is a leaf and can be inserted with no further recursive calls.

Before doing the insertion, the binary search tree has to be traversed in order to build an array of all its elements. This takes $O(n)$ time.

Consider the call stack of `balanced_insertion()` with the full range of n elements. It makes two recursive calls, so the call stack is shaped like a binary tree. The height of the call tree is at most $\lg n$, because the range of elements in each recursive call is more than halved with respect to the range of its caller. Then this tree can have at most $2^{1+\lg n} = 2 \cdot 2^{\lg n} = 2n$ nodes, each representing a call to `balanced_insertion()`. Since each call does constant time work, this function's running time is $O(n)$ as well.

3-17 The definition of height balanced tree is recursive, but the recursive aspect is implicit in the way it's worded in the book. An explicit way to say it is: a binary tree is height balanced if both its children are height balanced and the difference between their heights is at most 1. A node with no children is balanced and has height 1 (or, equivalently, a "null" node is balanced and has height 0).

```
data Tree = Node Tree Tree
          | Nil

isBalanced :: Tree -> (Bool, Int)
isBalanced Nil = (True, 0)
isBalanced (Node l r) = (balanced, height)
  where (lb, lh) = isBalanced l
        (rb, rh) = isBalanced r
        height   = 1 + max lh rh
        balanced = lb && rb && abs (lh - rh) <= 1
```

Each call performs constant-time work, and there is only one call for each node in the tree. Therefore this is an $O(n)$ time algorithm.

3-18 We have a balanced tree where all of `search()`, `insert()`, `delete()`, `minimum()` and `maximum()` take $O(\log n)$ time, and we want to ensure it supports `successor()` and `predecessor()` in $O(1)$ time. Each node will have pointers to its logical predecessor and successor that will have to be maintained during update operations. This would effectively add a doubly linked list structure on top of the binary tree.

`successor()`: see pp. 85 in the book. The in order successor can be found, with a parent pointer, by considering two different cases: if the node has a right subtree, and if it hasn't.

`predecessor()`: finding it is symmetric to the successor.

- If a node is a leaf, and is the left child of its parent, traverse up until finding a node that is the right child of its parent. The parent will be the predecessor.
- If a node is a leaf and is a right child, this is the trivial case of the previous point—the parent is the predecessor.
- If a node has a left subtree, the predecessor is the maximum of that left subtree.

When deleting an item from the tree, we can get ahold of its successor and predecessor in $O(1)$ time via the pointers, delete the item, then link the pointers as done in a doubly linked list.

Note that the exercise says the tree is balanced. We can assume there is a balancing operation that takes place after each insert and delete, before the successor and predecessor pointers are updated.

3-19 We have a dictionary with $O(\log n)$ `search()`, `insert()`, `delete()`, `min()`, `max()`, `predecessor()` and `successor()`, and we want to make some changes to `insert()` and `delete()` so `min()` and `max()` will take $O(1)$ time while the update operations still take $O(\log n)$ time.

After `insert()` we can query for the new element's predecessor in $O(\log n)$ time—if there isn't one, our new element is the new minimum. Similarly with `maximum()` and `successor()`.

Before `delete()` we can query for the predecessor of the element—if there isn't one we can find the new minimum by querying for its successor (although in reality we'd know we are trying to delete the minimum because we have a $O(1)$ minimum); likewise when deleting the maximum. If before delete we do find a more extreme element, we know we are not deleting the minimum or maximum and no adjustment is necessary.

3-20 The exercise calls this structure a "set", and by its operations it does look like a set, so we'll assume the elements are unique. Furthermore, we will assume we have a balancing operation that keeps the tree height balanced in $O(\log n)$ time.

Our set structure is backed by a balanced binary search tree. The `member()` query searches for the element, taking $O(\log n)$ time. The `insert()` operation is the regular BST insert—if the key already exists, it is overwritten; this also takes $O(\log n)$ time. Finally, `delete()` has to find the k-th smallest element. Assume our BST has $O(\log n)$ `minimum()` and $O(1)$ `successor()`, as in the previous exercise. Then we need to query for the minimum element and then k calls to `successor()`, then one $O(\log n)$ delete and one $O(\log n)$ rebalance.

Unfortunately this doesn't work—consider always deleting the last element. After finding the minimum, we always do n calls to `successor()`, turning this into a $O(n)$ algorithm in the worst case. `:\`

3-21 This may be cheating somehow, but if the two sets are disjoint and all keys in S_1 are less than every key in S_2, we can concatenate both trees by finding the minimum element in S_2 and making the root of S_1 its left child. It would wreak havoc on the balance of the tree, but rebalancing should take at most $O(n + m)$ time, for n elements in S_1 and m elements in S_2. The traversal of S_2 to its minimum would be $O(logm)$ and the concatenation—a simple matter of pointer manipulation—would be constant time.

3-22 We have to design a data structure that supports `insert(x)` and `median()` operations, both in $O(\log n)$ time. Inserting is easy enough, but the median is an element relative to all other elements in the structure; it depends on its logical position within the set of all the elements, so it's not clear that it can be determined without somehow keeping track of counts.

Suppose the structure keeps two binary search trees, the median, and counters for the elements in each tree. The elements in the left tree are less than the median, while those on the right tree are greater than the median.

When inserting, we compare the element to the median to select which side to insert it into. It's then inserted in the correct tree, taking $O(\log n)$ time (actually $\log m$, with m the number of elements on that side of the median). We increase the counter on this side, and if the difference in elements between both sides is greater than 1, we know we need to shift the median.

To rebalance the structure, suppose the left side has two more elements than the right side. We can insert the median in the right tree in $O(\log n)$ time, then find and delete the maximum from the left tree, also in $O(\log n)$ time, and set it as the new median. The right counter increases by one, the left counter decreases by one, and we are balanced again.

This actually gives $O(1)$ access to the median, so it is probably wrong for the exercise.

Chapter 4

Sorting

Solutions

4-1 The Grinch wants the most unbalanced game possible. This is the same as asking to maximize the difference in total skill between the two teams.

In $O(n \log n)$, sort the player pool by skill and make each half of the result a team. Any swap between teams would bring a higher skilled player into the lower half, and a lower skilled player into the higher half, making the difference in total skill lower.

4-2

(a) Unsorted array. Find x, y that maximize $|x - y|$ in $O(n)$ time. In one sweep through the array, find the minimum and maximum values. These maximize the difference.

(b) Sorted array. Find x, y that maximize $|x - y|$ in $O(1)$ time. These are the first and last elements of the array.

(c) Unsorted array. Find x, y that minimize $|x - y|$, for $x \neq y$, in $O(n \log n)$ time. Sort the array in $O(n \log n)$ then do a linear pairwise search of consecutive values for the pair with the smallest difference in $O(n)$ time.

(d) Sorted array. Find x, y that minimize $|x - y|$, for $x \neq y$, in $O(n)$ time. Do a pairwise comparison of every element with the next in $O(n)$ and find the consecutive pair with the smallest difference.

4-3 What we want to do here is first sort the array in $O(n \log n)$ time. Then we want to pair up the biggest offender with the least problematic number, so the largest with the smallest, the second largest with the second smallest, and so on.

$$x_1 < x_2 < \ldots < x_{2n-1} < x_{2n} \quad \rightarrow \quad (x_1, x_{2n}), (x_2, x_{2n-1}), \ldots$$

Why does this work? Suppose in my pairing I find that (x, y) are my maximum pair. By way of contradiction, I claim that there is a different partition of the $2n$ numbers whose maximum pair is less than (x, y).

Let's say that $x < y$. The claim that a different partition does better implies that, whatever its maximum pair is, element y in this hypothetical partition has to be paired up with an element t such that $t < x$, otherwise the maximum pair could not be less than $x + y$. But the same logic applies to every element larger than y. They need to be paired up with elements smaller than x certainly, or for any $z > y$ we'd have a pair (z, x) where $z + x > x + y$ is larger than the claimed maximum.

Now suppose there are i elements larger than y. In my original pairing, that means there are i elements smaller than x. But the claim forced me to pair y with one of those i elements smaller than x, so there are now $i - 1$ numbers available to pair with the i numbers larger than y. We see that we won't be able to find pairs for every element that don't surpass our claimed maximum pair. The contradiction shows that our original pairing was indeed optimal, and the algorithm correct.

4-4 Keep three linked lists and pointers to the end of each. Iterate over the input pairs, appending each to the list corresponding to its color. By the pointers to the ends, this can be done in $O(1)$ time. Finally concatenate all three lists, which is again constant time.

4-5 $O(n)$ is the best we can hope for, since we need to at least visit every element in the input. By using a hash table with (amortized) $O(1)$ query and insertion we can check for each value's membership, set it to 1 on first sight, then increment it on subsequent visits to construct a frequency table of all values. There are n of them, and constant work for each (query, increment, insert), so building the table is $O(n)$ time. Iterating over all the key-value pairs and selecting the maximum frequency is another $O(n)$ effort after which we have found the mode.

4-6 We can afford to sort one of the sets into a sorted array A in $O(n \log n)$ time. It's now possible to query A for membership of a number in $O(\log n)$ time by binary search. Then for each number b in the other set, let $a = x - b$, and query A for a. If $a \in A$ then we have a number $a = x - b$, which is to say $x = a + b$, and we can answer positively. If we test every number in the second set unsuccessfully, there is no pair that adds to x. In this (worst) case, we do n binary searches for a total $O(n \log n)$ time.