Lecture Notes for Chapter 17: Amortized Analysis

Chapter 17 overview

Amortized analysis

- Analyze a *sequence* of operations on a data structure.
- ***Goal:*** Show that although some individual operations may be expensive, *on average* the cost per operation is small.

Average in this context does not mean that we're averaging over a distribution of inputs.

- No probability is involved.
- We're talking about *average cost in the worst case*.

Organization

We'll look at 3 methods:

- aggregate analysis
- accounting method
- potential method

Using 3 examples:

- stack with multipop operation
- binary counter
- dynamic tables (later on)

Aggregate analysis

Stack operations

- PUSH(S, x): $O(1)$ each $\Rightarrow$ $O(n)$ for any sequence of n operations.
- POP(S): $O(1)$ each $\Rightarrow$ $O(n)$ for any sequence of n operations.

- MULTIPOP(S, k)
  while S is not empty and $k > 0$
  do POP(S)
  $k \leftarrow k - 1$

Running time of MULTIPOP:

- Linear in # of POP operations.
- Let each PUSH/POP cost 1.
- # of iterations of **while** loop is $\min(s, k)$, where $s =$ # of objects on stack.
- Therefore, total cost $= \min(s, k)$.

Sequence of n PUSH, POP, MULTIPOP operations:

- Worst-case cost of MULTIPOP is $O(n)$.
- Have n operations.
- Therefore, worst-case cost of sequence is $O(n^2)$.

Observation

- Each object can be popped only once per time that it's pushed.
- Have $\leq n$ PUSHes $\Rightarrow \leq n$ POPs, including those in MULTIPOP.
- Therefore, total cost $= O(n)$.
- Average over the n operations $\Rightarrow O(1)$ per operation on average.

Again, notice no probability.

- Showed *worst-case* $O(n)$ cost for sequence.
- Therefore, $O(1)$ per operation on average.

This technique is called ***aggregate analysis***.

Binary counter

- k-bit binary counter $A[0\,..\,k-1]$ of bits, where $A[0]$ is the least significant bit and $A[k-1]$ is the most significant bit.
- Counts upward from 0.
- Value of counter is $\sum_{i=0}^{k-1} A[i] \cdot 2^i$.
- Initially, counter value is 0, so $A[0\,..\,k-1] = 0$.
- To increment, add 1 $\pmod{2^k}$:

  INCREMENT(A, k)
  $i \leftarrow 0$
  while $i < k$ and $A[i] = 1$
  do $A[i] \leftarrow 0$
  $i \leftarrow i + 1$
  if $i < k$
  then $A[i] \leftarrow 1$

Example: $k = 3$

[Underlined bits flip. Show costs later.]

counter value	A 2 1 0	cost
0	0 0 $\underline{0}$	0
1	0 $\underline{0}$ $\underline{1}$	1
2	0 1 $\underline{0}$	3
3	$\underline{0}$ $\underline{1}$ $\underline{1}$	4
4	1 0 $\underline{0}$	7
5	1 $\underline{0}$ $\underline{1}$	8
6	1 1 $\underline{0}$	10
7	$\underline{1}$ $\underline{1}$ $\underline{1}$	11
0	0 0 $\underline{0}$	14
$\vdots$	$\vdots$	15

Cost of INCREMENT $= \Theta($# of bits flipped$)$.

Analysis: Each call could flip k bits, so n INCREMENTs takes $O(nk)$ time.

Observation

Not every bit flips every time.

[Show costs from above.]

bit	flips how often	times in n INCREMENTs
0	every time	n
1	1/2 the time	$\lfloor n/2 \rfloor$
2	1/4 the time	$\lfloor n/4 \rfloor$
	$\vdots$	
i	$1/2^i$ the time	$\lfloor n/2^i \rfloor$
	$\vdots$	
$i \geq k$	never	0

$$\begin{aligned}
\text{Therefore, total \# of flips} &= \sum_{i=0}^{k-1} \lfloor n/2^i \rfloor \\
&< n \sum_{i=0}^{\infty} 1/2^i \\
&= n \left(\frac{1}{1 - 1/2} \right) \\
&= 2n \; .
\end{aligned}$$

Therefore, n INCREMENTs costs $O(n)$.

Average cost per operation $= O(1)$.

Accounting method

Assign different charges to different operations.

- Some are charged more than actual cost.
- Some are charged less.

Amortized cost = amount we charge.

When amortized cost > actual cost, store the difference *on specific objects* in the data structure as ***credit***.

Use credit later to pay for operations whose actual cost > amortized cost.

Differs from aggregate analysis:

- In the accounting method, different operations can have different costs.
- In aggregate analysis, all operations have same cost.

Need credit to never go negative.

- Otherwise, have a sequence of operations for which the amortized cost is not an upper bound on actual cost.
- Amortized cost would tell us *nothing*.

Let c_i = actual cost of ith operation ,
$\widehat{c_i}$ = amortized cost of ith operation .

Then require $\sum_{i=1}^{n} \widehat{c_i} \geq \sum_{i=1}^{n} c_i$ for *all* sequences of n operations.

Total credit stored $= \sum_{i=1}^{n} \widehat{c_i} - \sum_{i=1}^{n} c_i \underbrace{\geq}_{\text{had better be}} 0$.

Stack

operation	actual cost	amortized cost
PUSH	1	2
POP	1	0
MULTIPOP	$\min(k, s)$	0

Intuition: When pushing an object, pay \$2.

- \$1 pays for the PUSH.
- \$1 is prepayment for it being popped by either POP or MULTIPOP.
- Since each object has \$1, which is credit, the credit can never go negative.
- Therefore, total amortized cost, $= O(n)$, is an upper bound on total actual cost.

Binary counter

Charge \$2 to set a bit to 1.

- \$1 pays for setting a bit to 1.
- \$1 is prepayment for flipping it back to 0.
- Have \$1 of credit for every 1 in the counter.
- Therefore, credit ≥ 0.

Amortized cost of INCREMENT:

- Cost of resetting bits to 0 is paid by credit.
- At most 1 bit is set to 1.
- Therefore, amortized cost $\leq$ \$2.
- For n operations, amortized cost $= O(n)$.

Potential method

Like the accounting method, but think of the credit as *potential* stored with the entire data structure.

- Accounting method stores credit with specific objects.
- Potential method stores potential in the data structure as a whole.
- Can release potential to pay for future operations.
- Most flexible of the amortized analysis methods.

Let D_i = data structure after ith operation ,
D_0 = initial data structure ,
c_i = actual cost of ith operation ,
$\widehat{c_i}$ = amortized cost of ith operation .

Potential function $\Phi : D_i \to \mathbf{R}$

$\Phi(D_i)$ is the *potential* associated with data structure D_i.

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= c_i + \underbrace{\Delta\Phi(D_i)}_{\text{increase in potential due to } i\text{th operation}} .\end{aligned}$$

$$\begin{aligned}\text{Total amortized cost} &= \sum_{i=1}^{n} \widehat{c_i} \\ &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &\quad \text{(telescoping sum: every term other than } D_0 \text{ and } D_n \\ &\qquad \text{is added once and subtracted once)} \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) .\end{aligned}$$

If we require that $\Phi(D_i) \geq \Phi(D_0)$ for all i, then the amortized cost is always an upper bound on actual cost.

In practice: $\Phi(D_0) = 0$, $\Phi(D_i) \geq 0$ for all i.

Stack

Φ = # of objects in stack
(= # of \$1 bills in accounting method)

D_0 = empty stack $\Rightarrow \Phi(D_0) = 0$.

Since # of objects in stack is always ≥ 0, $\Phi(D_i) \geq 0 = \Phi(D_0)$ for all i.

operation	actual cost	$\Delta\Phi$	amortized cost
PUSH	1	$(s+1) - s = 1$ where s = # of objects initially	$1 + 1 = 2$
POP	1	$(s-1) - s = -1$	$1 - 1 = 0$
MULTIPOP	$k' = \min(k, s)$	$(s - k') - s = -k'$	$k' - k' = 0$

Therefore, amortized cost of a sequence of n operations $= O(n)$.

Binary counter

$\Phi = b_i$ = # of 1's after ith INCREMENT

Suppose ith operation resets t_i bits to 0.

$c_i \leq t_i + 1$ (resets t_i bits, sets ≤ 1 bit to 1)

- If $b_i = 0$, the ith operation reset all k bits and didn't set one, so $b_{i-1} = t_i = k \Rightarrow b_i = b_{i-1} - t_i$.
- If $b_i > 0$, the ith operation reset t_i bits, set one, so $b_i = b_{i-1} - t_i + 1$.
- Either way, $b_i \leq b_{i-1} - t_i + 1$.
- Therefore,

$$\begin{aligned} \Delta\Phi(D_i) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i \,. \end{aligned}$$

$$\begin{aligned} \widehat{c_i} &= c_i + \Delta\Phi(D_i) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2 \,. \end{aligned}$$

If counter starts at 0, $\Phi(D_0) = 0$.

Therefore, amortized cost of n operations $= O(n)$.

Dynamic tables

A nice use of amortized analysis.

Scenario

- Have a table—maybe a hash table.
- Don't know in advance how many objects will be stored in it.
- When it fills, must reallocate with a larger size, copying all objects into the new, larger table.
- When it gets sufficiently small, *might* want to reallocate with a smaller size.

Details of table organization not important.

Goals

1. $O(1)$ amortized time per operation.
2. Unused space always $\leq$ constant fraction of allocated space.

Load factor $\alpha = num/size$, where $num = \#$ items stored, $size =$ allocated size.

If $size = 0$, then $num = 0$. Call $\alpha = 1$.

Never allow $\alpha > 1$.

Keep $\alpha >$ a constant fraction $\Rightarrow$ goal (2).

Table expansion

Consider only insertion.

- When the table becomes full, double its size and reinsert all existing items.
- Guarantees that $\alpha \geq 1/2$.
- Each time we actually insert an item into the table, it's an ***elementary insertion***.

```
TABLE-INSERT(T, x)
if size[T] = 0
  then allocate table[T] with 1 slot
       size[T] ← 1
if num[T] = size[T]                    ▷ expand?
  then allocate new-table with 2 · size[T] slots
       insert all items in table[T] into new-table      ▷ num[T] elem insertions
       free table[T]
       table[T] ← new-table
       size[T] ← 2 · size[T]
insert x into table[T]                                  ▷ 1 elem insertion
num[T] ← num[T] + 1
```

Initially, $num[T] = size[T] = 0$.

Running time: Charge 1 per elementary insertion. Count only elementary insertions, since all other costs together are constant per call.

c_i = actual cost of ith operation

- If not full, $c_i = 1$.
- If full, have $i - 1$ items in the table at the start of the ith operation. Have to copy all $i - 1$ existing items, then insert ith item $\Rightarrow c_i = i$.

n operations $\Rightarrow c_i = O(n) \Rightarrow O(n^2)$ time for n operations.

Of course, we don't always expand:

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is exact power of 2 ,} \\ 1 & \text{otherwise .} \end{cases}$$

$$\begin{aligned} \text{Total cost} &= \sum_{i=1}^{n} c_i \\ &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &= n + \frac{2^{\lfloor \lg n \rfloor + 1} - 1}{2 - 1} \\ &< n + 2n \\ &= 3n \end{aligned}$$

Therefore, **aggregate analysis** says amortized cost per operation = 3.

Accounting method

Charge \$3 per insertion of x.

- \$1 pays for x's insertion.
- \$1 pays for x to be moved in the future.
- \$1 pays for some other item to be moved.

Suppose we've just expanded, $size = m$ before next expansion, $size = 2m$ after next expansion.

- Assume that the expansion used up all the credit, so that there's no credit stored after the expansion.
- Will expand again after another m insertions.
- Each insertion will put \$1 on one of the m items that were in the table just after expansion and will put \$1 on the item inserted.
- Have \$$2m$ of credit by next expansion, when there are $2m$ items to move. Just enough to pay for the expansion, with no credit left over!

Potential method

$\Phi(T) = 2 \cdot num[T] - size[T]$

- Initially, $num = size = 0 \Rightarrow \Phi = 0$.
- Just after expansion, $size = 2 \cdot num \Rightarrow \Phi = 0$.
- Just before expansion, $size = num \Rightarrow \Phi = num \Rightarrow$ have enough potential to pay for moving all items.
- Need $\Phi \geq 0$, always.

  Always have

$$\begin{array}{lcll} size & \geq & num & \geq \frac{1}{2} \cdot size \Rightarrow \\ & & 2 \cdot num & \geq size \Rightarrow \\ & & \Phi & \geq 0 . \end{array}$$

Amortized cost of ith operation:

$$\begin{aligned} num_i &= num \text{ after } i\text{th operation} , \\ size_i &= size \text{ after } i\text{th operation} , \\ \Phi_i &= \Phi \text{ after } i\text{th operation} . \end{aligned}$$

- If no expansion:

$$\begin{aligned} size_i &= size_{i-1} , \\ num_i &= num_{i-1} + 1 , \\ c_i &= 1 . \end{aligned}$$

  Then we have

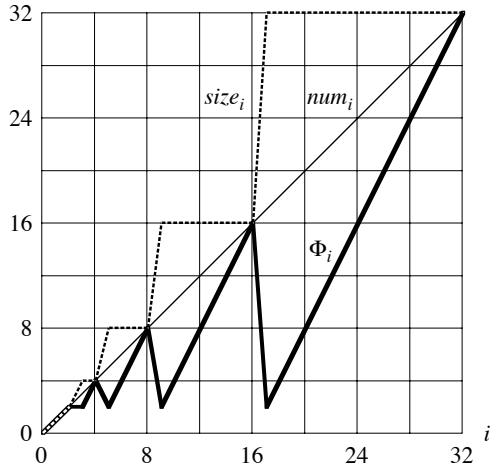
$$\begin{aligned} \widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) \\ &= 1 + 2 \\ &= 3 . \end{aligned}$$

- If expansion:

$$\begin{aligned} size_i &= 2 \cdot size_{i-1} , \\ size_{i-1} &= num_{i-1} = num_i - 1 , \\ c_i &= num_{i-1} + 1 = num_i . \end{aligned}$$

  Then we have

$$\begin{aligned} \widehat{c}_i &= c_i + \Phi_i + \Phi_{i-1} \\ &= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= num_i + (2 \cdot num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\ &= num_i + 2 - (num_i - 1) \\ &= 3 . \end{aligned}$$



Expansion and contraction

When α drops too low, contract the table.

- Allocate a new, smaller one.
- Copy all items.

Still want

- α bounded from below by a constant,
- amortized cost per operation $= O(1)$.

Measure cost in terms of elementary insertions and deletions.

"Obvious strategy":

- Double size when inserting into a full table (when $\alpha = 1$, so that after insertion α would become > 1).
- Halve size when deletion would make table less than half full (when $\alpha = 1/2$, so that after deletion α would become $< 1/2$).
- Then always have $1/2 \le \alpha \le 1$.
- Suppose we fill table.
  Then insert $\Rightarrow$ double
  2 deletes $\Rightarrow$ halve
  2 inserts $\Rightarrow$ double
  2 deletes $\Rightarrow$ halve
  . . .

Not performing enough operations after expansion or contraction to pay for the next one.

Simple solution:

- Double as before: when inserting with $\alpha = 1 \Rightarrow$ after doubling, $\alpha = 1/2$.
- Halve size when deleting with $\alpha = 1/4 \Rightarrow$ after halving, $\alpha = 1/2$.
- Thus, immediately after either expansion or contraction, have $\alpha = 1/2$.
- Always have $1/4 \le \alpha \le 1$.

Intuition:

- Want to make sure that we perform enough operations between consecutive expansions/contractions to pay for the change in table size.
- Need to delete half the items before contraction.
- Need to double number of items before expansion.
- Either way, number of operations between expansions/contractions is at least a constant fraction of number of items copied.

$$\Phi(T) = \begin{cases} 2 \cdot num[T] - size[T] & \text{if } \alpha \geq 1/2 \ , \\ size[T]/2 - num[T] & \text{if } \alpha < 1/2 \ . \end{cases}$$

T empty $\Rightarrow \Phi = 0$.

$\alpha \geq 1/2 \Rightarrow num \geq \frac{1}{2} \cdot size \Rightarrow 2 \cdot num \geq size \Rightarrow \Phi \geq 0$.

$\alpha < 1/2 \Rightarrow num < \frac{1}{2} \cdot size \Rightarrow \Phi \geq 0$.

Intuition: Φ measures how far from $\alpha = 1/2$ we are.

- $\alpha = 1/2 \Rightarrow \Phi = 2 \cdot num - 2 \cdot num = 0$.
- $\alpha = 1 \Rightarrow \Phi = 2 \cdot num - num = num$.
- $\alpha = 1/4 \Rightarrow \Phi = size/2 - num = 4 \cdot num/2 - num = num$.
- Therefore, when we double or halve, have enough potential to pay for moving all *num* items.
- Potential increases linearly between $\alpha = 1/2$ and $\alpha = 1$, and it also increases linearly between $\alpha = 1/2$ and $\alpha = 1/4$.
- Since α has different distances to go to get to 1 or $1/4$, starting from $1/2$, rate of increase of Φ differs.
  - For α to go from $1/2$ to 1, *num* increases from $size/2$ to *size*, for a total increase of $size/2$. Φ increases from 0 to *size*. Thus, Φ needs to increase by 2 for each item inserted. That's why there's a coefficient of 2 on the $num[T]$ term in the formula for Φ when $\alpha \geq 1/2$.
  - For α to go from $1/2$ to $1/4$, *num* decreases from $size/2$ to $size/4$, for a total decrease of $size/4$. Φ increases from 0 to $size/4$. Thus, Φ needs to increase by 1 for each item deleted. That's why there's a coefficient of -1 on the $num[T]$ term in the formula for Φ when $\alpha < 1/2$.

Amortized costs: more cases

- insert, delete
- $\alpha \geq 1/2$, $\alpha < 1/2$ (use α_i, since α can vary a lot)
- *size* does/doesn't change

Insert:

- $\alpha_{i-1} \geq 1/2$, same analysis as before $\Rightarrow \widehat{c}_i = 3$.
- $\alpha_{i-1} < 1/2 \Rightarrow$ *no expansion* (only occurs when $\alpha_{i-1} = 1$).

- If $\alpha_{i-1} < 1/2$ and $\alpha_i < 1/2$:

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi_i + \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\ &= 0 .\end{aligned}$$

- If $\alpha_{i-1} < 1/2$ and $\alpha_i \geq 1/2$:

$$\begin{aligned}\widehat{c}_i &= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\ &= 3 \cdot num_{i-1} - \frac{3}{2} \cdot size_{i-1} + 3 \\ &= 3 \cdot \alpha_{i-1}\, size_{i-1} - \frac{3}{2} \cdot size_{i-1} + 3 \\ &< \frac{3}{2} \cdot size_{i-1} - \frac{3}{2} \cdot size_{i-1} + 3 \\ &= 3 .\end{aligned}$$

  Therefore, amortized cost of insert is < 3.

Delete:

- If $\alpha_{i-1} < 1/2$, then $\alpha_i < 1/2$.
  - If no contraction:

$$\begin{aligned}\widehat{c}_i &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) \\ &= 2 .\end{aligned}$$

  - If contraction:

$$\begin{aligned}\widehat{c}_i &= \underbrace{(num_i + 1)}_{\text{move + delete}} + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &\quad [size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1] \\ &= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1)) \\ &= 1 .\end{aligned}$$

- If $\alpha_{i-1} \geq 1/2$, then no contraction.
  - If $\alpha_i \geq 1/2$:

$$\begin{aligned}\widehat{c}_i &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_i + 2 - size_i) \\ &= -1 .\end{aligned}$$

  - If $\alpha_i < 1/2$, since $\alpha_{i-1} \geq 1/2$, have

$$num_i = num_{i-1} - 1 \geq \frac{1}{2} \cdot size_{i-1} - 1 = \frac{1}{2} \cdot size_i - 1 .$$

Thus,

$$\begin{aligned}
\widehat{c}_i &= 1 + (size_i/2 - num_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= 1 + (size_i/2 - num_i) - (2 \cdot num_i + 2 - size_i) \\
&= -1 + \frac{3}{2} \cdot size_i - 3 \cdot num_i \\
&\le -1 + \frac{3}{2} \cdot size_i - 3\left(\frac{1}{2} \cdot size_i - 1\right) \\
&= 2 .
\end{aligned}$$

Therefore, amortized cost of delete is ≤ 2.

Solutions for Chapter 17: Amortized Analysis

Solution to Exercise 17.1-3

Let c_i = cost of ith operation.

$$c_i = \begin{cases} i & \text{if } i \text{ is an exact power of 2 ,} \\ 1 & \text{otherwise .} \end{cases}$$

Operation	Cost
1	1
2	2
3	1
4	4
5	1
6	1
7	1
8	8
9	1
10	1
$\vdots$	$\vdots$

n operations cost

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lg n} 2^j = n + (2n - 1) < 3n \ .$$

(Note: Ignoring floor in upper bound of $\sum 2^j$.)

Average cost of operation $= \dfrac{\text{Total cost}}{\#\text{ operations}} < 3$

By aggregate analysis, the amortized cost per operation $= O(1)$.

Solution to Exercise 17.2-1

[We assume that the only way in which COPY *is invoked is automatically, after every sequence of k* PUSH *and* POP *operations.]*

Charge \$2 for each PUSH and POP operation and \$0 for each COPY. When we call PUSH, we use \$1 to pay for the operation, and we store the other \$1 on the item pushed. When we call POP, we again use \$1 to pay for the operation, and we store the other \$1 in the stack itself. Because the stack size never exceeds k, the actual cost of a COPY operation is at most \$$k$, which is paid by the \$$k$ found in the items in the stack and the stack itself. Since there are k PUSH and POP operations between two consecutive COPY operations, there are \$$k$ of credit stored, either on individual items (from PUSH operations) or in the stack itself (from POP operations) by the time a COPY occurs. Since the amortized cost of each operation is $O(1)$ and the amount of credit never goes negative, the total cost of n operations is $O(n)$.

Solution to Exercise 17.2-2

Let c_i = cost of ith operation.

$$c_i = \begin{cases} i & \text{if } i \text{ is an exact power of 2 ,} \\ 1 & \text{otherwise .} \end{cases}$$

Charge each operation \$3 (amortized cost $\widehat{c}_i$).

- If i is not an exact power of 2, pay \$1, and store \$2 as credit.
- If i is an exact power of 2, pay \$$i$, using stored credit.

Operation	Cost	Actual cost	Credit remaining
1	3	1	2
2	3	2	3
3	3	1	5
4	3	4	4
5	3	1	6
6	3	1	8
7	3	1	10
8	3	8	5
9	3	1	7
10	3	1	9
⋮	⋮	⋮	⋮

Since the amortized cost is \$3 per operation, $\sum_{i=1}^{n} \widehat{c}_i = 3n$.

We know from Exercise 17.1-3 that $\sum_{i=1}^{n} c_i < 3n$.

Then we have $\sum_{i=1}^{n} \widehat{c}_i \geq \sum_{i=1}^{n} c_i \Rightarrow$ credit = amortized cost − actual cost ≥ 0.

Since the amortized cost of each operation is $O(1)$, and the amount of credit never goes negative, the total cost of n operations is $O(n)$.

Solution to Exercise 17.2-3

We introduce a new field $max[A]$ to hold the index of the high-order 1 in A. Initially, $max[A]$ is set to -1, since the low-order bit of A is at index 0, and there are initially no 1's in A. The value of $max[A]$ is updated as appropriate when the counter is incremented or reset, and we use this value to limit how much of A must be looked at to reset it. By controlling the cost of RESET in this way, we can limit it to an amount that can be covered by credit from earlier INCREMENTs.

```
INCREMENT(A)
i ← 0
while i < length[A] and A[i] = 1
    do A[i] ← 0
       i ← i + 1
if i < length[A]
  then A[i] ← 1
       ▷ Additions to book's INCREMENT start here
       if i > max[A]
         then max[A] ← i
  else max[A] ← −1
```

```
RESET(A)
for i ← 0 to max[A]
    do A[i] ← 0
max[A] ← −1
```

As for the counter in the book, we assume that it costs \$1 to flip a bit. In addition, we assume it costs \$1 to update $max[A]$.

Setting and resetting of bits by INCREMENT will work exactly as for the original counter in the book: \$1 will pay to set one bit to 1; \$1 will be placed on the bit that is set to 1 as credit; the credit on each 1 bit will pay to reset the bit during incrementing.

In addition, we'll use \$1 to pay to update max, and if max increases, we'll place an additional \$1 of credit on the new high-order 1. (If max doesn't increase, we can just waste that \$1—it won't be needed.) Since RESET manipulates bits at positions only up to $max[A]$, and since each bit up to there must have become the high-order 1 at some time before the high-order 1 got up to $max[A]$, every bit seen by RESET has \$1 of credit on it. So the zeroing of bits of A by RESET can be completely paid for by the credit stored on the bits. We just need \$1 to pay for resetting max.

Thus charging \$4 for each INCREMENT and \$1 for each RESET is sufficient, so the sequence of n INCREMENT and RESET operations takes $O(n)$ time.

Solution to Exercise 17.3-3

Let D_i be the heap after the ith operation, and let D_i consist of n_i elements. Also, let k be a constant such that each INSERT or EXTRACT-MIN operation takes at most $k \ln n$ time, where $n = \max(n_{i-1}, n_i)$. (We don't want to worry about taking the log of 0, and at least one of n_{i-1} and n_i is at least 1. We'll see later why we use the natural log.)

Define

$$\Phi(D_i) = \begin{cases} 0 & \text{if } n_i = 0 \,, \\ k n_i \ln n_i & \text{if } n_i > 0 \,. \end{cases}$$

This function exhibits the characteristics we like in a potential function: if we start with an empty heap, then $\Phi(D_0) = 0$, and we always maintain that $\Phi(D_i) \ge 0$.

Before proving that we achieve the desired amortized times, we show that if $n \ge 2$, then $n \ln \frac{n}{n-1} \le 2$. We have

$$\begin{aligned} n \ln \frac{n}{n-1} &= n \ln \left(1 + \frac{1}{n-1}\right) \\ &= \ln \left(1 + \frac{1}{n-1}\right)^n \\ &\le \ln \left(e^{\frac{1}{n-1}}\right)^n && \text{(since } 1 + x \le e^x \text{ for all real } x\text{)} \\ &= \ln e^{\frac{n}{n-1}} \\ &= \frac{n}{n-1} \\ &\le 2 \,, \end{aligned}$$

assuming that $n \ge 2$. (The equation $\ln e^{\frac{n}{n-1}} = \frac{n}{n-1}$ is why we use the natural log.)

If the ith operation is an INSERT, then $n_i = n_{i-1} + 1$. If the ith operation inserts into an empty heap, then $n_i = 1$, $n_{i-1} = 0$, and the amortized cost is

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le k \ln 1 + k \cdot 1 \ln 1 - 0 \\ &= 0 \,. \end{aligned}$$

If the ith operation inserts into a nonempty heap, then $n_i = n_{i-1} + 1$, and the amortized cost is

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le k \ln n_i + k n_i \ln n_i - k n_{i-1} \ln n_{i-1} \\ &= k \ln n_i + k n_i \ln n_i - k(n_i - 1) \ln(n_i - 1) \\ &= k \ln n_i + k n_i \ln n_i - k n_i \ln(n_i - 1) + k \ln(n_i - 1) \\ &< 2k \ln n_i + k n_i \ln \frac{n_i}{n_i - 1} \\ &\le 2k \ln n_i + 2k \\ &= O(\lg n_i) \,. \end{aligned}$$

If the ith operation is an EXTRACT-MIN, then $n_i = n_{i-1} - 1$. If the ith operation extracts the one and only heap item, then $n_i = 0$, $n_{i-1} = 1$, and the amortized cost

is

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le k \ln 1 + 0 - k \cdot 1 \ln 1 \\ &= 0 .\end{aligned}$$

If the ith operation extracts from a heap with more than 1 item, then $n_i = n_{i-1} - 1$ and $n_{i-1} \ge 2$, and the amortized cost is

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le k \ln n_{i-1} + k n_i \ln n_i - k n_{i-1} \ln n_{i-1} \\ &= k \ln n_{i-1} + k(n_{i-1} - 1) \ln(n_{i-1} - 1) - k n_{i-1} \ln n_{i-1} \\ &= k \ln n_{i-1} + k n_{i-1} \ln(n_{i-1} - 1) - k \ln(n_{i-1} - 1) - k n_{i-1} \ln n_{i-1} \\ &= k \ln \frac{n_{i-1}}{n_{i-1} - 1} + k n_{i-1} \ln \frac{n_{i-1} - 1}{n_{i-1}} \\ &< k \ln \frac{n_{i-1}}{n_{i-1} - 1} + k n_{i-1} \ln 1 \\ &= k \ln \frac{n_{i-1}}{n_{i-1} - 1} \\ &\le k \ln 2 \qquad (\text{since } n_{i-1} \ge 2) \\ &= O(1) .\end{aligned}$$

A slightly different potential function—which may be easier to work with—is as follows. For each node x in the heap, let $d_i(x)$ be the depth of x in D_i. Define

$$\begin{aligned}\Phi(D_i) &= \sum_{x \in D_i} k(d_i(x) + 1) \\ &= k \left(n_i + \sum_{x \in D_i} d_i(x) \right) ,\end{aligned}$$

where k is defined as before.

Initially, the heap has no items, which means that the sum is over an empty set, and so $\Phi(D_0) = 0$. We always have $\Phi(D_i) \ge 0$, as required.

Observe that after an INSERT, the sum changes only by an amount equal to the depth of the new last node of the heap, which is $\lfloor \lg n_i \rfloor$. Thus, the change in potential due to an INSERT is $k(1 + \lfloor \lg n_i \rfloor)$, and so the amortized cost is $O(\lg n_i) + O(\lg n_i) = O(\lg n_i) = O(\lg n)$.

After an EXTRACT-MIN, the sum changes by the negative of the depth of the old last node in the heap, and so the potential *decreases* by $k(1 + \lfloor \lg n_{i-1} \rfloor)$. The amortized cost is at most $k \lg n_{i-1} - k(1 + \lfloor \lg n_{i-1} \rfloor) = O(1)$.

Solution to Problem 17-2

a. The SEARCH operation can be performed by searching each of the individually sorted arrays. Since all the individual arrays are sorted, searching one of them using a binary search algorithm takes $O(\lg m)$ time, where m is the size of the array. In an unsuccessful search, the time is $\Theta(\lg m)$. In the worst case, we may

assume that all the arrays $A_0, A_1, \ldots, A_{k-1}$ are full, $k = \lceil \lg(n+1) \rceil$, and we perform an unsuccessful search. The total time taken is

$$\begin{aligned} T(n) &= \Theta(\lg 2^{k-1} + \lg 2^{k-2} + \cdots + \lg 2^1 + \lg 2^0) \\ &= \Theta((k-1) + (k-2) + \cdots + 1 + 0) \\ &= \Theta(k(k-1)/2) \\ &= \Theta(\lceil \lg(n+1) \rceil \, (\lceil \lg(n+1) \rceil - 1)/2) \\ &= \Theta(\lg^2 n) \ . \end{aligned}$$

Thus, the worst-case running time is $\Theta(\lg^2 n)$.

b. We create a new sorted array of size 1 containing the new element to be inserted. If array A_0 (which has size 1) is empty, then we replace A_0 with the new sorted array. Otherwise, we merge sort the two arrays into another sorted array of size 2. If A_1 is empty, then we replace A_1 with the new array; otherwise we merge sort the arrays as before and continue. Since array A_i is of size 2^i, if we merge sort two arrays of size 2^i each, we obtain one of size 2^{i+1}, which is the size of A_{i+1}. Thus, this method will result in another list of arrays in the same structure that we had before.

Let us analyze its worst-case running time. We will assume that merge sort takes $2m$ time to merge two sorted lists of size m each. If all the arrays $A_0, A_1, \ldots, A_{k-2}$ are full, then the running time to fill array A_{k-1} would be

$$\begin{aligned} T(n) &= 2\,(2^0 + 2^1 + \cdots + 2^{k-2}) \\ &= 2(2^{k-1} - 1) \\ &= 2^k - 2 \\ &= \Theta(n) \ . \end{aligned}$$

Therefore, the worst-case time to insert an element into this data structure is $\Theta(n)$.

However, let us now analyze the amortized running time. Using the aggregate method, we compute the total cost of a sequence of n inserts, starting with the empty data structure. Let r be the position of the rightmost 0 in the binary representation $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$ of n, so that $n_j = 1$ for $j = 0, 1, \ldots, r-1$. The cost of an insertion when n items have already been inserted is

$$\sum_{j=0}^{r-1} 2 \cdot 2^j = O(2^r) \ .$$

Furthermore, $r = 0$ half the time, $r = 1$ a quarter of the time, and so on. There are at most $\lceil n/2^r \rceil$ insertions for each value of r. The total cost of the n operations is therefore bounded by

$$O\left(\sum_{r=0}^{\lceil \lg(n+1) \rceil} \left(\left\lceil \frac{n}{2^r} \right\rceil \right) 2^r \right) = O(n \lg n) \ .$$

The amortized cost per INSERT operation, therefore is $O(\lg n)$.

We can also use the accounting method to analyze the running time. We can charge $\$k$ to insert an element. \$1 pays for the insertion, and we put $\$(k-1)$ on the inserted item to pay for it being involved in merges later on. Each time it is merged, it moves to a higher-indexed array, i.e., from A_i to A_{i+1}. It can

move to a higher-indexed array at most $k - 1$ times, and so the $\$(k - 1)$ on the item suffices to pay for all the times it will ever be involved in merges. Since $k = \Theta(\lg n)$, we have an amortized cost of $\Theta(\lg n)$ per insertion.

c. DELETE(x) will be implemented as follows:

1. Find the smallest j for which the array A_j with 2^j elements is full. Let y be the last element of A_j.
2. Let x be in the array A_i. If necessary, find which array this is by using the search procedure.
3. Remove x from A_i and put y into A_i. Then move y to its correct place in A_i.
4. Divide A_j (which now has $2^j - 1$ elements left): The first element goes into array A_0, the next 2 elements go into array A_1, the next 4 elements go into array A_2, and so forth. Mark array A_j as empty. The new arrays are created already sorted.

The cost of DELETE is $\Theta(n)$ in the worst case, where $i = k - 1$ and $j = k - 2$: $\Theta(\lg n)$ to find A_j, $\Theta(\lg^2 n)$ to find A_i, $\Theta(2^i) = \Theta(n)$ to put y in its correct place in array A_i, and $\Theta(2^j) = \Theta(n)$ to divide array A_j. The following sequence of n operations, where $n/3$ is a power of 2, yields an amortized cost that is no better: perform $n/3$ INSERT operations, followed by $n/3$ pairs of DELETE and INSERT. It costs $O(n \lg n)$ to do the first $n/3$ INSERT operations. This creates a single full array. Each subsequent DELETE/INSERT pair costs $\Theta(n)$ for the DELETE to divide the full array and another $\Theta(n)$ for the INSERT to recombine it. The total is then $\Theta(n^2)$, or $\Theta(n)$ per operation.

Solution to Problem 17-4

a. For RB-INSERT, consider a complete red-black tree in which the colors alternate between levels. That is, the root is black, the children of the root are red, the grandchildren of the root are black, the great-grandchildren of the root are red, and so on. When a node is inserted as a red child of one of the red leaves, then case 1 of RB-INSERT-FIXUP occurs $(\lg(n + 1))/2$ times, so that there are $\Omega(\lg n)$ color changes to fix the colors of nodes on the path from the inserted node to the root.

For RB-DELETE, consider a complete red-black tree in which all nodes are black. If a leaf is deleted, then the double blackness will be pushed all the way up to the root, with a color change at each level (case 2 of RB-DELETE-FIXUP), for a total of $\Omega(\lg n)$ color changes.

b. All cases except for case 1 of RB-INSERT-FIXUP and case 2 of RB-DELETE-FIXUP are terminating.

c. Case 1 of RB-INSERT-FIXUP reduces the number of red nodes by 1. As Figure 13.5 shows, node z's parent and uncle change from red to black, and z's grandparent changes from black to red. Hence, $\Phi(T') = \Phi(T) - 1$.

d. Lines 1–16 of RB-INSERT cause one node insertion and a unit increase in potential. The nonterminating case of RB-INSERT-FIXUP (Case 1) makes three

color changes and decreases the potential by 1. The terminating cases of RB-INSERT-FIXUP (cases 2 and 3) cause one rotation each and do not affect the potential. (Although case 3 makes color changes, the potential does not change. As Figure 13.6 shows, node z's parent changes from red to black, and z's grandparent changes from black to red.)

e. The number of structural modifications and amount of potential change resulting from lines 1–16 of RB-INSERT and from the terminating cases of RB-INSERT-FIXUP are $O(1)$, and so the amortized number of structural modifications of these parts is $O(1)$. The nonterminating case of RB-INSERT-FIXUP may repeat $O(\lg n)$ times, but its amortized number of structural modifications is 0, since by our assumption the unit decrease in the potential pays for the structural modifications needed. Therefore, the amortized number of structural modifications performed by RB-INSERT is $O(1)$.

f. From Figure 13.5, we see that case 1 of RB-INSERT-FIXUP makes the following changes to the tree:

- Changes a black node with two red children (node C) to a red node, resulting in a potential change of -2.
- Changes a red node (node A in part (a) and node B in part (b)) to a black node with one red child, resulting in no potential change.
- Changes a red node (node D) to a black node with no red children, resulting in a potential change of 1.

The total change in potential is -1, which pays for the structural modifications performed, and thus the amortized number of structural modifications in case 1 (the nonterminating case) is 0. The terminating cases of RB-INSERT-FIXUP cause $O(1)$ structural changes. Because $w(v)$ is based solely on node colors and the number of color changes caused by terminating cases is $O(1)$, the change in potential in terminating cases is $O(1)$. Hence, the amortized number of structural modifications in the terminating cases is $O(1)$. The overall amortized number of structural modifications in RB-INSERT, therefore, is $O(1)$.

g. Figure 13.7 shows that case 2 of RB-DELETE-FIXUP makes the following changes to the tree:

- Changes a black node with no red children (node D) to a red node, resulting in a potential change of -1.
- If B is red, then it loses a black child, with no effect on potential.
- If B is black, then it goes from having no red children to having one red child, resulting in a potential change of -1.

The total change in potential is either -1 or -2, depending on the color of B. In either case, one unit of potential pays for the structural modifications performed, and thus the amortized number of structural modifications in case 2 (the nonterminating case) is at most 0. The terminating cases of RB-DELETE cause $O(1)$ structural changes. Because $w(v)$ is based solely on node colors and the number of color changes caused by terminating cases is $O(1)$, the change in potential in terminating cases is $O(1)$. Hence, the amortized number of structural changes in the terminating cases is $O(1)$. The overall amortized number of structural modifications in RB-DELETE-FIXUP, therefore, is $O(1)$.

h. Since the amortized number structural modification in each operation is $O(1)$, the actual number of structural modifications for any sequence of m RB-INSERT and RB-DELETE operations on an initially empty red-black tree is $O(m)$ in the worst case.