# CS 170 Fall 2014 — Algorithms — David Wagner — Soln 7

**1. (20 pts.) Amortized running time**

We want to implement an append-only log. You can think of it as a list that allows you to append to the end. It supports one operation: Append$(x)$, which adds the value $x$ to the end of the list so far.

In memory, the data is stored in an array. If we append enough entries that the array becomes full, we allocate a new array that is twice as large (so we'll have space for the new element) and copy over the old values. In particular, our implementation of Append is as follows:

```
global variables:
    an array $A[]$, integers $s, t$
    initially $t = 0$, $s = 1$, and $A[]$ is 1 element long

Append$(x)$:
1.  If $t = s$:
2.      Allocate a new array $B$ with $2s$ elements.
3.      For $i := 0, 1, \ldots, s-1$: set $B[i] := A[i]$.
4.      Set $A := B$ (pointer assignment) and $s := 2s$.
5.  Set $A[t] := x$ and then $t := t + 1$.
```

Line 4 does not copy the arrays element by element; instead, it simply swaps a pointer, so that the name $A$ now refers to the new array just allocated on line 2. Notice that $s$ always holds the size of the array $A$ (the number of elements it can store), and $A[0..t-1]$ holds the elements that have been appended to the log so far. Also $0 \le t \le s$ is an invariant of the algorithm.

(a) Suppose we perform $m$ calls to Append, starting from the initial state. Then, we perform one more call to Append$(x)$. What is the worst-case running time of that last call, as a function of $m$?

**Solution:**

$\Theta(m)$. If $s = t$, the allocation happens (on lines 2–4) and we will have to allocate a new array with size $\Theta(m)$.

(b) Suppose we perform $m = 2^k$ calls to Append, starting from the initial state. What are the values of $t$ where calling Append causes lines 2–5 to be executed? Use this to argue that the total number of elements copied in line 3, summed across all $m$ of these calls, is $1 + 2 + 4 + 8 + \cdots + 2^{k-1}$. Then, prove that the total running time of all $m$ of these calls is $\Theta(m)$.

**Solution:**

The allocation happens when the array is full, at which point the size of the array gets doubled (starting from 1). Therefore lines 2–4 happen when $t = 2^i$, where $i = 0, 1, \ldots, k-1$. When $t = 2^i$, the total number of elements that get copied is $2^i$, therefore the total number of elements copied across all $m$ calls is $1 + 2 + \cdots + 2^{k-1} = 2^k - 1$. Since the rest of the algorithm takes $\Theta(1)$ time for every insert, the total running time is $\Theta(2^k) = \Theta(m)$.

(c) Given your answer to part (b), what is the amortized running time of Append?

**Solution:**

Let $T(m)$ be the total running time after the $m$ inserts. By part (b), $T(m) = \Theta(m)$ when $m = 2^k$. Now assume that $2^k < m < 2^{k+1}$. We have $T(m) \le T(2^{k+1}) = \Theta(2^{k+1}) = \Theta(2^k) = \Theta(m)$. Hence the amortized running time is $\Theta(1)$.

(d) Now let's see a different way to analyze the amortized running time of this data structure, using the accounting method. The running time of lines 2–4 (in a single call to Append) is $\Theta(s)$, so we will consider lines 2–4 to cost $s$ dollars. The running time of lines 1 and 5 is $\Theta(1)$, so we will consider the execution of lines 1 and 5 to cost 1 dollar.

Suppose that each time Alice calls Append, she pays us 3 dollars. If Alice calls Append at a time when $t < s$ (so lines 2–4 are not executed), how much profit do we have left over from this one call to Append?

**Solution:**

Since only line 1 and line 5 are executed, it costs \$1 and we get \$2 profit.

(e) Suppose whenever we make a profit, we save our money for a rainy day. In particular, whenever Alice calls Append, we'll put our profits from that call next to the array element $A[t]$ that was just set in line 5.

Now suppose Alice calls Append at a time when $t = s$. Which elements of $A$ have some money sitting next to them? How many dollars are there sitting next to the elements of $A$, in total? If we grab all of those dollars, does that provide enough to pay for the $s$ dollars that it will cost to execute steps 2–4?

**Solution:**

Assume $s = 2^k$. The last time lines 2–4 were executed was when $s = 2^{k-1}$. Therefore, we will have the money available from the $(2^{k-1}+1)$-th element to the $2^k$-th element. Since we have \$2 profit for each element, the total amount of money is $2^{k-1} \times 2 = 2^k$, which is sufficient to execute steps 2–4.

Comment: If you answered $2^k + 1$, with a suitable explanation, that's also OK. The case $s = t = 1$ is an exception; for that case, we have \$2 available, and only need \$1 for steps 2–4, so we'll have \$1 left over. This \$1 can be carried forward forever. Therefore, when $s = t = 2^k$, we have $2^k$ dollars available from the $(2^{k-1}+1)$-th element to the $2^k$-th element, plus the \$1 carry-over from the very beginning, for a total of $2^k + 1$.

(f) For the accounting-based amortized analysis to be valid, we have to be sure that any time Append is called, there is enough money available to pay for its running time (i.e., our total balance will never go negative). Using the method outlined in parts (d)–(e), is this guaranteed?

**Solution:**

Since we're making profit when lines 2–4 are not executed, we only need to worry about having enough money when $s = 2^k$. To prove that we always have sufficient funds, we can use induction on $k$. In the base case when $k = 0$, $s = 2^k = 1$. We have \$2 left over from the first call to Append, which is enough to pay for this call. Now assume that we always have enough money for the first $s = 2^{k-1}$ calls to Append. When $s = 2^k$, notice that from part (e), we only used the profit from the $2^{k-1}+1$-th Append to $2^k$-th Append to pay for the $2^k$-th Append, therefore by induction we always have sufficient money during the whole process.

(g) If Alice makes $m$ calls to Append, she will have paid us a total of $3m$ dollars. Based on parts (d)–(f), what is an upper bound on the total running time needed to execute all $m$ of those calls to Append?

**Solution:**

Since the running time never exceeds the total money paid at any point of time during the algorithm (by part (f)), the total running time after $m$ calls is $\Theta(m)$.

(h) Based on part (g), what is the amortized running time of a call to Append?

**Solution:**

$\Theta(m)/m = \Theta(1)$.

**2. (15 pts.) Proof of correctness for greedy algorithms**

A doctor's office has $n$ customers, labeled $1,2,\dots,n$, waiting to be seen. They are all present right now and will wait until the doctor can see them. The doctor can see one customer at a time, and we can predict exactly how much time each customer will need with the doctor: customer $i$ will take $t(i)$ minutes.

(a) We want to minimize the average waiting time (the average of the amount of time each customer waits before they are seen, not counting the time they spend with the doctor). What order should we use?

Hint: sort the customers by ____.

**Solution:**

Sort the customers by $t(i)$, starting with the smallest $t(i)$.

(b) Let $x_1, x_2, \dots, x_n$ denote an ordering of the customers (so we see customer $x_1$ first, then customer $x_2$, and so on). Prove that the following modification, if applied to any order, will never increase the average waiting time:

- If $i < j$ and $t(x_i) \geq t(x_j)$, swap customer $i$ with customer $j$.

(For example, if the order of customers is $3,1,4,2$ and $t(3) \geq t(4)$, then applying this rule with $i = 1$ and $j = 3$ gives us the new order $4,1,3,2$.)

**Solution:**

First observe that swapping $x_i$ and $x_j$ does not affect the waiting time customers $x_1, x_2, \dots, x_i$ or customers $x_{j+1}, x_{j+1}, \dots, x_n$ (i.e., for customers $x_k$ where $k \leq i$ or $k > j$). Therefore we only have to deal with customers $x_{i+1}, \dots, x_j$, i.e., for customer $k$, where $i < k \leq j$. For customer $x_k$, the waiting time before the swap is

$$T_k = \sum_{1 \leq l < k} t(x_l),$$

and the waiting time after the swap is

$$T'_k = \sum_{1 \leq l < i} t(x_l) + t(x_j) + \sum_{i < l < k} t(x_l) = T_k - t(x_i) + t(x_j).$$

Since $t(x_i) \geq t(x_j)$, $T'_k \leq T_k$, so the waiting time is never increased for customers $x_{i+1}, \dots, x_j$, hence the average waiting time for all the customers will not increase after the swap.

(c) Let $u$ be the ordering of customers you selected in part (a), and $x$ be any other ordering. Prove that the average waiting time of $u$ is no larger than the average waiting time of $x$—and therefore your answer in part (a) is optimal.

Hint: Let $i$ be the smallest index such that $u_i \neq x_i$. Use what you learned in part (b). Then, use proof by induction (maybe backwards, in the order $i = n, n-1, n-2, \dots, 1$, or in some other way).

**Solution:**

Let $u$ be the ordering in part (a), and $x$ be any other ordering. Let $i$ be the smallest index such that $u_i \neq x_i$. Let $j$ be the index of $x_i$ in $u$, i.e, $x_i = u_j$ and $k$ be the index of $u_i$ in $x$. It's easy to see that $j > i$. By the construction of $x$, we have $T(x_i) = T(u_j) \geq T(u_i) = T(x_k)$, therefore by swapping $x_i$ and $x_k$, we will not increase the average waiting time. If we keep doing this, eventually we will transform $x$ into $u$. Since we never increase the average waiting time throughout the process, $u$ is the optimal ordering.

**3. (15 pts.) Job Scheduling**

You are given a set of $n$ jobs. Each takes one unit of time to complete. Job $i$ has an integer-valued deadline time $d_i \geq 0$ and a real-valued penalty $p_i \geq 0$. Jobs may be scheduled to start at any non-negative integer time (0, 1, 2, etc), and only one job may run at a time. If job $i$ completes at or before time $d_i$, then it incurs no penalty; otherwise, it is late and incurs penalty $p_i$. The goal is to schedule all jobs so as to minimize the total penalty incurred.

For each of the following greedy algorithms, either prove that it is correct, or give a simple counterexample (with at most three jobs) to show that it fails.

(a) Among unscheduled jobs that can be scheduled on time, consider the one whose deadline is the earliest (breaking ties by choosing the one with the highest penalty), and schedule it at the earliest available time. Repeat.

**Solution:**

This is incorrect. Consider a listing of $(d_i, p_i)$ as $(1,1),(2,10),(2,20)$. The algorithm schedules $(1,1)$ at time 0, $(2,20)$ at time 1, and $(2,10)$ late, incurring a penalty of 10. One better (actually optimal) scheduling is to schedule $(2,10)$ at time 0, $(2,20)$ at time 1, and $(1,1)$ late, incurring a penalty of 1.

(b) Among unscheduled jobs that can be scheduled on time, consider the one whose penalty is the highest (breaking ties by choosing the one with the earliest deadline), and schedule it at the earliest available time. Repeat.

**Solution:**

Incorrect. Consider a listing of $(d_i, p_i)$ as $(1,1),(2,10)$. The algorithm schedules $(2,10)$ at time 0, and $(1,1)$ late, incurring a penalty of 1. One better (actually optimal) scheduling is to schedule $(1,1)$ at time 0, $(2,10)$ at time 1, incurring no penalty.

(c) Among unscheduled jobs that can be scheduled on time, consider the one whose penalty is the highest (breaking ties arbitrarily), and schedule it at the latest available time before its deadline. Repeat.

**Solution:**

This order is correct (optimal). Proof: Consider any partial scheduling $P$ of jobs, and let $j_h$ be a highest penalty job among those unscheduled jobs in $P$ that can still be scheduled on time. We will show that there is an optimal schedule $S'$ that schedules $j_h$ at the latest available time before its deadline. Once this "greedy choice property" is established, then clearly the algorithm gives optimal schedule (by a standard induction argument).

Now we prove the property. Consider any optimal schedule $S$. Now construct $S'$ from $S$, by scheduling $j_h$ at the latest available time before its deadline. If in $S$ that time slot is taken by a job $j_k$, then $S'$ just swaps $j_h$ with $j_k$. (You may want to draw a picture to see clearly what is happening. We are implicitly assuming that both $S$ and $S'$ extend $P$, in the sense that all the jobs scheduled in $P$ are scheduled the same way in $S$ and $S'$.)

- If $S$ schedules $j_h$ no later than its deadline: then $S'$ schedules $j_k$ earlier (hence no more penalty for $j_k$) and $j_h$ later (but still before deadline, hence no more penalty for $j_h$) than $S$ does, so $S'$ incurs no more penalty than $S$;
- If $S$ schedules $j_h$ late: then $S'$ schedules $j_h$ on time, at the expense of possibly scheduling $j_k$ late, but $S'$ incurs no more penalty than $S$, since $p_h \geq p_k$.

**4. (15 pts.) Timesheets**

Suppose we have $N$ jobs labeled $1,\ldots,N$. For each job, there is a bonus $V_i \geq 0$ for completing the job, and a penalty $P_i \geq 0$ per day that accumulates for each day until the job is completed. It will take $R_i \geq 0$ days to successfully complete job $i$.

Each day, we choose one unfinished job to work on. A job $i$ has been finished if we have spent $R_i$ days working on it. This doesn't necessarily mean you have to spend $R_i$ consecutive days working on job $i$. We start on day 1, and we want to complete all our jobs and finish with maximum reward. If we finish job $i$ at the end of day $t$, we will get reward $V_i - t \cdot P_i$. Note, this value can be negative if you choose to delay a job for too long.

Given this information, what is the optimal job scheduling policy to complete all of the jobs?

**Solution:**

Sort the jobs in order of decreasing $P_i/R_i$ and allocate the first $R_i$ available days to that job. Repeat until all $N$ jobs have been completed.

**Proof of correctness.** Because interleaving the work for several jobs will delay all the jobs' completion dates, interleaving will necessarily have a higher penalty than a contiguous scheduling, and cannot be a part of any optimal solution. Therefore, we focus our attention only on orders where each job is scheduled on contiguous days: once you start a job, you complete it before starting any other job. Also, since we must complete all jobs, the $V_i$ values are irrelevant: the base reward $V_i$ is always received, no matter what order we schedule the jobs. Therefore, we can treat all the $V_i$ as zero and just minimize the total penalties accrued, and this won't change the optimal solution.

To prove that our order is optimal, we will use the following swapping rule:

- Suppose the jobs are numbered $1, 2, \ldots, n$ in the order they appear in the ordering, and suppose $P_j/R_j \leq P_{j+1}/R_{j+1}$. Then we can swap jobs $j$ and $j+1$.

We'll first prove that this modification, if applied to any order, will never make things worse (it will never increase the penalty). Why? Well, the penalty for the days when jobs $j$ and $j+1$ are being done is

$$\rho = R_j P_j + (R_j + R_{j+1}) P_{j+1} + (R_j + R_{j+1})(P_{j+2} + P_{j+2} + \cdots + P_n),$$

before the swap. After the swap, the penalty for that time period becomes

$$\rho' = (R_j + R_{j+1}) P_j + R_{j+1} P_{j+1} + (R_j + R_{j+1})(P_{j+2} + P_{j+2} + \cdots + P_n).$$

Notice that

$$\rho' = \rho + R_{j+1} P_j - R_j P_{j+1}.$$

Now if $P_j/R_j \leq P_{j+1}/R_{j+1}$, then $R_{j+1}P_j \leq R_j P_{j+1}$, so $\rho' \leq \rho$. The penalty for the other days is unaffected by the swap. This proves that the swapping rule above can never increase the penalty.

Now let's prove that our algorithm generates an optimal ordering. If we start from any order, then we can repeatedly apply the swapping rule above until the jobs are ordered by decreasing $P_i/R_i$-value, i.e., until $P_1/R_1 \geq P_2/R_2 \geq \cdots \geq P_N/R_N$. Basically, we just apply bubble sort: take the job with largest $P_i/R_i$ and swap it forward until it is at the start of the order; then take the job with second-largest $P_i/R_i$ and repeatedly swap to move it forward until it as second in the order; and so on. We can see that each step, the swap moves a job with larger $P_i/R_i$-value forward in front of a job with smaller $P_i/R_i$-value, so is allowed by the swapping rule. Therefore, the swapping rule above allows us to transform any order into the one output by our algorithm, without increasing the total penalty. As a consequence, the order produced by our algorithm must be optimal.

**5. (15 pts.) A greedy algorithm—so to speak**

The founder of LinkedIn, the professional networking site, decides to crawl LinkedIn's relationship graph to find all of the *super-schmoozers*. (He figures he can make more money from advertisers by charging a premium for ads displayed to super-schmoozers.) A *super-schmoozer* is a person on LinkedIn who has a link to at least 20 other super-schmoozers on LinkedIn.

We can formalize this as a graph problem. Let the undirected graph $G = (V, E)$ denote LinkedIn's relationship graph, where each vertex represents a person who has an account on LinkedIn. There is an edge

$\{u,v\} \in E$ if $u$ and $v$ have listed a professional relationship with each other on LinkedIn (we will assume that relationships are symmetric). We are looking for a subset $S \subseteq V$ of vertices so that every vertex $s \in S$ has edges to at least 20 other vertices in $S$. And we want to make the set $S$ as large as possible, subject to these constraints.

Design an efficient algorithm to find the set of super-schmoozers (the largest set $S$ that is consistent with these constraints), given the graph $G$.

Hint: There are some vertices you can rule out immediately as not super-schmoozers.

**Solution:**

**Main idea.**

The basic idea here is that we iteratively remove non-schmoozers until we reach a fixed point. Any node whose degree is below 20 is certainly a non-schmoozer. We keep a worklist of pending non-schmoozers. In each iteration we remove a non-schmoozer from the worklist, delete it, adjust the degree of its neighbors, and add them to the worklist if their degree has fallen below 20.

**Pseudocode.**

```
1.  Set $d[v] := 0$ for each $v \in V$.
2.  For each edge $\{u,v\} \in E$:
3.      Increment $d[u]$. Increment $d[v]$.
4.  Initialize a list $W$ as $W := \{v \in V : d[v] < 20\}$.
5.  While $W$ is non-empty:
6.      Remove a vertex from $W$; call it $v$.
7.      For each $\{v,w\} \in E$:
8.          Decrement $d[w]$.
9.          If $d[w] < 20$ and $w \notin W$, add $w$ to $W$.
10.     Remove $v$ from the graph.
11. Output the set of nodes left in the graph (they are the super-schmoozers).
```

**Proof of correctness.**

Lemma: At each iteration of the while loop, for each vertex $v$ that has not yet been deleted, $d[v] =$ the degree of $v$ (in the graph that remains).

Proof: This invariant can be easily proven by induction on the number of iterations of the loop. Whenever we delete a vertex, we decrement each of its neighbors to reflect the change in their degree.

Claim: After the algorithm ends, we've identified a subgraph $G'$ such that every user in $G'$ has degree at least 20, in that subgraph. In other words, the algorithm outputs a valid set of vertices who satisfy the requirements to be super-schmoozers.

Proof: At the beginning of every iteration of the loop, every node either has degree at least 20, or is in the worklist $W$. Nodes in the worklist could not possibly be super-schmoozers. This is true before the first iteration because that's how line 4 initialized the worklist. After that, a node's degree can only change when one of its neighbors is removed, and whenever a node is deleted we check all of its neighbors' degrees. Therefore, after the loop, when the worklist is empty, every remaining vertex [if any] will have degree $\geq 20$, and therefore can be validly labelled a super-schmoozer.

Claim: Everyone who can be validly labelled as a super-schmoozer is included in the output of the algorithm.

Proof: Let $S$ be the largest set of vertices that can be validly labeled as super-schmoozers. Then it is an invariant that $d[s] \geq 20$ and $s \notin W$, throughout the algorithm, for all $s \in S$. This is true before the first iteration because of how line 4 initialized the worklist. Also, if it is true before one iteration of the loop, it remains true after that iteration. In particular, consider an iteration where we remove $v$ from the worklist. By the contrapositive of the inductive hypothesis and using the fact that we had $v \in W$ at the start of this iteration, it follows that $v \notin S$. Also, for each neighbor $w$ of $v$, if $w \in S$, by the inductive hypothesis $w$ had an edge to $\geq 20$ other members of $S$; since $v \not{i}nS$, after deleting $v$ it still has an edge to $\geq 20$ other members of $S$. Therefore, the invariant remains true after this iteration of the loop, and the claim follows by induction.

The first claim shows that the algorithm's output is not "too large", and the second claim shows that the algorithm's output is not "too small". This implies that our algorithm finds the largest possible set of super-schmoozers, as desired.

**Running time.** $O(|V|+|E|)$. We examine each vertex twice (in lines 1 and 4), and do a constant amount of work per vertex in each case. Each vertex is inserted into the worklist at most once, so the number of iterations of the loop in lines 5–10 is at most $|V|$, and each iteration takes $O(1)$ time, ignoring the inner loop at lines 7–9. (Notice that you can remove an element from a list in $O(1)$ time if you remove the item at the head of the list, so line 6 takes $O(1)$ time.) Also, the inner loop at lines 7–9 examines each edge at most once, when we remove one of its endpoints, and we we do a constant amount of work per edge in that case. This accounts for all of the work done in this algorithm, and we can see that we do $O(1)$ work per vertex plus $O(1)$ work per edge. Hence, the running time is $O(|V|+|E|)$.

**6. (20 pts.) A funky kind of coloring**

Let $G=(V,E)$ be an undirected graph where every vertex has degree $\leq 51$. Let's find a way of coloring each vertex blue or gold, so that no vertex has more than 25 neighbors of its own color.

Consider the following algorithm, where we call a vertex "bad" if it has more than 25 neighbors of its own color:

```
1. Color each vertex arbitrarily.
2. Let $B := \{v \in V : v \text{ is bad}\}$.
3. While $B \neq \emptyset$:
4.      Pick any bad vertex $v \in B$.
5.      Reverse the color of $v$.
6.      Update $B$ to reflect this change, so that it again holds the set of bad vertices.
```

Notice that if this algorithm terminates, it is guaranteed to find a coloring with the desired property.

(a) Prove that this algorithm terminates in a finite number of steps. I suggest that you define a *potential function* that associates a non-negative integer (the potential) to each possible way of coloring the graph, in such a way that each iteration of the while-loop is guaranteed to strictly reduce the potential.

**Solution:**

Define the potential function to be the number of edges that connect same-color nodes. This is clearly a non-negative integer.

It decreases at each step since when we flip a node's color, it assumes a color that a strict majority of its neighbors *didn't* have.

The initial value of the potential function is finite. Since it is a non-negative integer that monotonically diminishes as the algorithm runs, the algorithm must stop, either when the potential reaches zero, or possibly before.

(b) Prove that the algorithm terminates after at most $|E|$ iterations of the loop.

Hint: You should figure out the largest value the potential could take on.

**Solution:**

The potential function defined in part (a) is clearly upper bounded by $|E|$: the number of same-color edges can't be more than the total number of edges. The potential decreases by at least 1 in each iteration of the loop, and the potential cannot go below 0, so the number of iterations of the loop must be at most the initial value of the potential: i.e., at most $|E|$.

Optional: Think about how to implement the algorithm so that its total running time is $O(|V|+|E|)$ — this won't be graded.

**Solution:**

The maximum degree of any node is 51, a constant. So, the algorithm can process "all a node's neighbors" in constant time. The data structure we need is a set that has constant-time insert, remove, and extract-an-element. As it happens, we can get this by combining a doubly linked list of vertices, along with a pointer from each vertex to its list element if it appears in the list (or null if it is not in the list).

**Comment:** This question introduces you to the notion of a potential function. These can be useful for proving termination or analyzing the running of an algorithm, as seen here. In particular, they provide a way to view a complicated algorithm as monotonic process: the potential always strictly decreases at each step. This is useful, because analyzing the termination of monotonic processes is usually pretty easy. Of course, finding the right potential function that will provide such a monotonicity function can require considerable creativity.

Potential functions can also be used for analyzing amortized running time. We define a potential function, where the potential is determined by the state of the data structure. Intuitively, the potential represents the amount of work we may have to do in the future to deal with future operations; a state with a large potential means that soon we may hit an operation that takes a long time to perform. More precisely, suppose the $i$th operation takes time $t_i$ and transforms the data structure from a state with potential $\Phi_{i-1}$ to a state with potential $\Phi_i$; then the effective cost of this operation is $c_i = t_i + \Phi_i - \Phi_{i-1}$. Suppose that the effective cost is always at most a fixed value $c$, and the potential is initially zero and never negative. Then it follows that after $n$ operations, the total amount of computation done will be at most $cn$, hence the amortized cost of each operation will be at most $c$. Proof:

$$\begin{aligned} t_1 + \cdots + t_n &\le t_1 + \cdots + t_n + \Phi_n - \Phi_0 \\ &= t_1 + \Phi_1 - \Phi_0 + t_2 + \Phi_2 - \Phi_1 + \cdots + t_n + \Phi_n - \Phi_{n-1} \\ &= c_1 + \cdots + c_n \le cn. \end{aligned}$$

For instance, consider Question 1 on this homework. We could define the potential to be the number of array elements that are occupied minus the number of empty array elements, i.e., $\Phi = s - t$. An ordinary Append operation takes 1 operation, and increases the potential by 2, so has effective cost $3 = O(1)$. An Append operation that requires doubling the size of the array takes $n$ time (where $n =$ the size of the array), but decreases the potential by $n$ (the potential changes from $n - 0 = n$ to $n - n = 0$), so has effective cost 0. In each case, the effective cost is $O(1)$, so the amortized running time of such a data structure is $O(1)$ time per operation.