Data structures and Algorithms

AMORTIZED ANALYSIS

Today

- Reminder: Asymptotics
- Linked Lists
- Amortized

Asymptotics

Exercise:

- Prove that if $f(n) = 3n^3 \log(n) - n^2\log(2^n)$, then, $f(n) \in \Theta(n^3 \log(n))$.

Asymptotics

Exercise:

- Prove that if $f(n) = 3n^3 \log(n) - n^2 \log(2^n)$, then, $f(n) \in \Theta(n^3 \log(n))$.

Solution:

In order to prove Θ, we need to prove both upper and lower bound. That is,
$3n^3 \log n - n^2 \log(2^n) \in O(n^3 \log n)$

and $3n^3 \log n - n^2 \log(2^n) \in \Omega(n^3 \log n)$

Asymptotics

Exercise:

- Prove that if $f(n) = 3n^3\log(n) - n^2\log(2^n)$, then, $f(n) \in \Theta(n^3\log(n))$.

Solution:

In order to prove Θ, we need to prove both upper and lower bound. That is, $3n^3\log n - n^2\log(2^n) \in O(n^3\log n)$,

and $3n^3\log n - n^2\log(2^n) \in \Omega(n^3\log n)$

That is, there exists $n_0, c_1 > 0$ and $c_2 > 0$ s.t.: $\forall\, n \geq n_0$

$c_1 n^3\log n \leq 3n^3\log n - n^2\log(2^n) \leq c_2 n^3\log n$

Asymptotics

Exercise:

- Prove that if $f(n) = 3n^3 \log(n) - n^2 \log(2^n)$, then, $f(n) \in \Theta(n^3 \log(n))$.

Solution:

Proof of $\boldsymbol{O}$:

$3n^3 \log(n) - n^2 \log(2^n) = 3n^3 \log(n) - n^3 \leq 3n^3 \log(n)$.

Hence for $n_0 = 1$ and $c = 3$ for every $n \geq n_0$ it holds that

$3n^3 \log(n) - n^2 \log(2^n) \leq cn \log n$.

Asymptotics

Exercise:

- Prove that if $f(n) = 3n^3\log(n) - n^2\log(2^n)$, then, $f(n) \in \Theta(n^3\log(n))$.

Solution:

Proof of Ω:

$3n^3\log(n) - n^2\log(2^n) =_{(*)} 3n^3\log(n) - n^3 \geq_{(**)} 3n^3\log(n) - n^3\log(n) = 2n^3\log(n),$

Where (*) is due to the logarithms identities, and (**) holds for every $\mathrm{n} \geq 2$: $\log n \geq 1$.

Hence for $n_0 = 2$ and $c = 2$ for every $n \geq n_0$ it holds that : $3n^3\log(n) - n^2\log(2^n) \geq cn\log n$.

Asymptotics

Exercise:

- Prove that if $f(n) = 3n^3 \log(n) - n^2\log(2^n)$, then, $f(n) \in \Theta(n^3 \log(n))$.

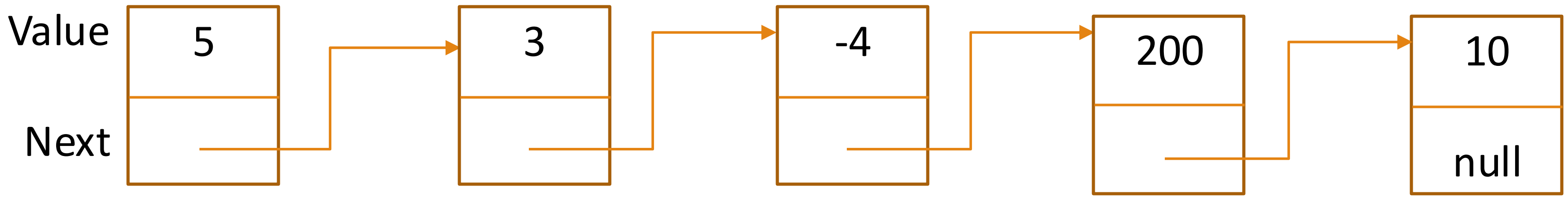
Solution:

Proof of $\boldsymbol{\Theta}$:

For $n_0 = 2, c_2 = 3$ and $c_1 = 2$

$$c_1 n^3 \log n \leq 3n^3 \log n - n^2 \log(2^n) \leq c_2 n^3 \log n$$

Linked Lists



Linked Lists - Exercise

- Describe an algorithm that given a linked list L (with integer values) returns the average of all elements in L:

Linked Lists - Exercise

- Describe an algorithm that given a linked list L (with integer values) returns the average of all elements in L:

- AverageList(L):
  - $Count \leftarrow 0$
  - $Sum \leftarrow 0$
  - $Current \leftarrow$ first node in L
  - While $Current$ not null:
    - $Sum \leftarrow Sum + Current.value$
    - $Count \leftarrow Count + 1$
    - $Current \leftarrow Current.next$
  - $Average \leftarrow \frac{Sum}{count}$
  - Return $Average$

Amortized Analysis

Imagine that you join a gym.

The gym charges a membership fee of **$60 per month, plus $3 for every time you use the gym**.

On top of the $60 monthly charge, you pay another $3 x 30 = $90 that month.

Although you can think of your fees as a flat fee of $60 and another $90 in daily fees, you can think about it in another way.

Altogether, you pay **$150 over 30 days**, or an **average of $5 per day**.

When you look at your fees in this way, you are splitting the monthly fee over the 30 days of the month, **spreading it out at $2 per day**.

Amortized Analysis - Aim

- Aim: understand how data structures perform over any sequence of operations.
- One bad operation shouldn't ruin a data structure if the operation is relatively uncommon.
- Worst-case performance per operation can be too pessimistic.
- What is the common runtime of a single operation?

Amortized Analysis - Intuition

- We learned to analyze each operation according to its worst case.
- Amortized analysis is a **worst-case analysis**, but for a **sequence of operations**, rather than for individual operations
- Mostly used to analyze data structures where **majority of the operations are cheap**, but some of the operations are expensive

Amortized Analysis - Basics

- For any sequence of operations: $op_1, op_2, \dots, op_m$ we define $Time(op_1, op_2, \dots, op_m)$ to be the time it takes to execute the entire sequence.

- Worst case analysis: define
$$T(m) = \max_{op_1, op_2, \dots, op_m} Time(op_1, op_2, \dots, op_m)\,.$$

- We say that $Amort(op)$ is an amortized cost per operation if and only if:
$$\forall\, m\colon\ T(m) \leq m\, Amort(op)$$

- Clearly, choosing $Amort(op)$ to be the worst-case time for a single operation would work. Is this choice tight?

- We should have $\sum amortized\ costs \geq \sum actual\ costs$, over all operations for any operation sequence

Amortized Analysis - Methods

- Aggregate analysis
- Accounting method

Amortized Analysis - Aggregate Analysis

When calculating amortized complexity:

1. We upper bound $T(m)$ by some function $T(m) \leq C(m)$
2. Define: $\forall i$: $Amort(op_i) = \frac{C(m)}{m}$

- How do we find $C(m)$?

Amortized Analysis - Accounting

- Each top-level operation in the algorithm is assigned a payment of tokens.
- Each “atomic” operation costs one token.
- **Allow an operation to store credit (amortized cost > actual cost)**
- **Allow an operation to pay using existing credit (amortized cost < actual cost)**
- Define $Amort(i)$ as the number of tokens assigned for operation i.
- Note: if $Amort(i)$ is constant, your amortized cost is $O(1)$

Question - Clearable DS

A clearable table has the following functions:

- add(e) – insert new element e to the next empty cell (assume there is always a next empty cell).
- clear() – delete all elements in the table.

This data structure is implemented with an array of size n.

Show that the amortized cost of add(e) and clear() is O(1), when starting from an empty array.

Note: the worst case for add(e) is O(1) and for clear() is O(n).

Question – Clearable DS

A worst case naïve analysis

In a sequence of n operations where a single clear() operation can take up to $O(n)$

the sequence of n operations is $O(n^2)$.

Clearable DS – Aggregate Analysis

A is an empty array of size N.

Consider a sequence of m operations $c_1, \ldots, c_m$

- Each add operation requires 1 step
- If the array contains k elements, then the clear operation takes k steps
- Each element that was added during the sequence $c_1, \ldots, c_m$, was deleted (using the clear operation) at most **<u>once</u>**.
- Hence, we pay at most 2 steps for each element that was inserted:

  one when adding it and one when deleting it

- In total:

$$amort(m) = \frac{1}{m} \cdot \sum_{i=1}^{m} T(c_i) \leq \frac{1}{m} \cdot \sum_{i=1}^{m} 2 \leq \frac{1}{m} \cdot 2m = 2 = O(1)$$

Clearable DS – Accounting Method

- A is an empty array of size N with a pointer to the first cell.
- Assume we pay one token for each command done with O(1)
- Let's define the next new costs:
  - Add – 2 tokens.
  - Clear – 0 tokens.
  - We priced Add with a more expensive price than it really cost and Clear with cheaper price.
  - After a new element is added into the table it will leave one token in the pool (since one token was paid for adding it into the table).
  - Therefore, each element is left with one token for the clear command.
  - For m commands we paid in total O(1) per command.

Question – Increasing counter represented by bits

A is a counter of n binary bits represented by binary bits.

What is the amortized cost of the next code:

How many bits changes we have during the algorithm?

```
Increment(A[0…n-1])
  i← 0
  while i < n and A[i] = 1
      A[i]←0
      i←i + 1
  if i < n  A[i]←1
```

0000
0001
0010
0011
…

Increasing counter represented by bits

0000
0001
0010
0011
0100
0101
0110
0111
1000

Solution: **Aggregate analysis**

We have m increasing commands and we start from 0.

Let's look on each bit of A:

A[0] is changed after each command. $O(m)$ $in\ total.$

A[1] is changed every second command. $O\left(\frac{m}{2}\right)$ $in\ total.$

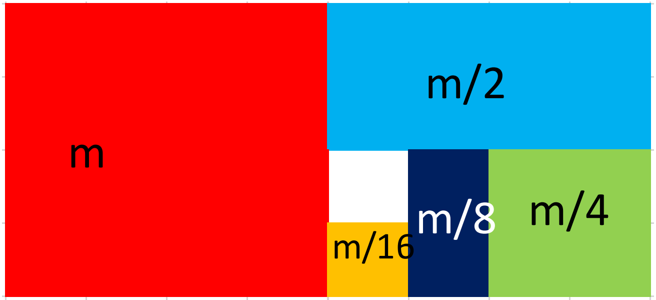
A[2] is changed every 4^{th} command. $O\left(\frac{m}{4}\right)$ $in\ total.$

And so on… In general:

A[i] is changed every 2^i commands and in total $O\left(\frac{m}{2^i}\right)$

#changes in total: $m + \frac{m}{2} + \frac{m}{4} + \cdots + \frac{m}{2^{n-1}} \leq 2m = O(m)$

$Amortized\ time\ is:\ \frac{2m}{m} = 2 = O(1)$



Increasing counter represented by bits

0000
0001
0010
0011
0100
0101
0110
0111
1000

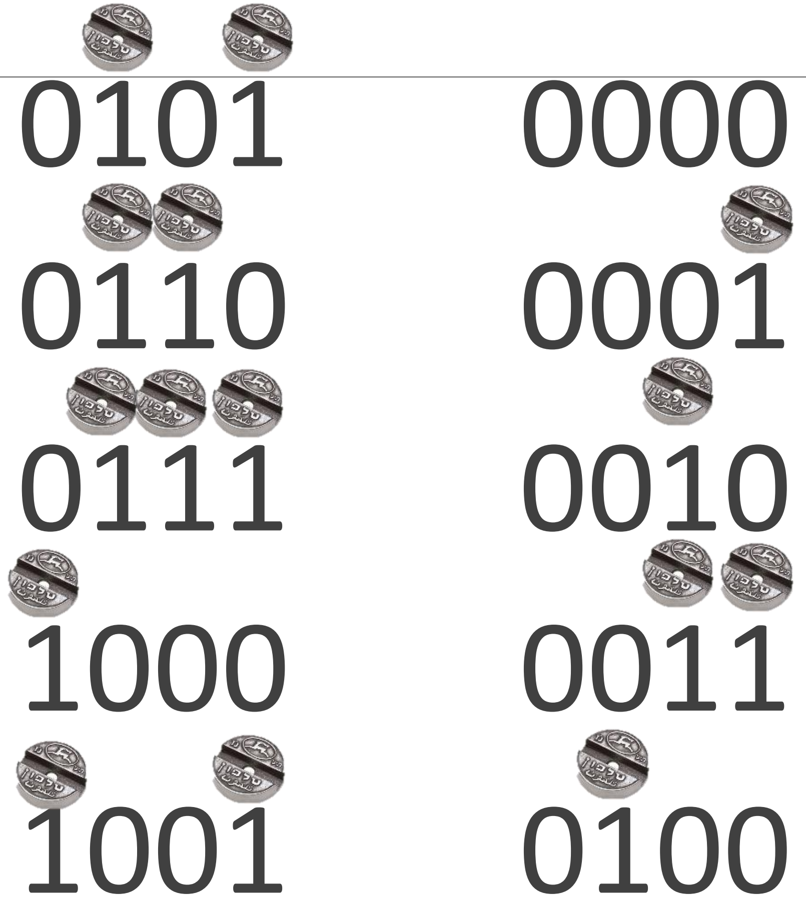
Solution: **Accounting method**

- We have m increment commands, and we start with all bits 0.
- We will allocate 2 tokens for each command.
- Switching a bit costs 1 token.
- Keep the second token if possible.
- Note: In each increment, a single 0 flips to 1
  - Multiple 1s can flip to 0 though

- **Need to prove: we will never run out of tokens.**

Question – Increasing counter represented by bits

```
Increment(A[0…n-1])
  i← 0
  while i < n and A[i] = 1
      A[i]←0
      i←i + 1
  if i < n  A[i]←1
```



Increasing counter represented by bits

0000
0001
0010
0011
0100
0101
0110
0111
1000

- **Claim: we will never run out of tokens.**

Increasing counter represented by bits

0000
0001
0010
0011
0100
0101
0110
0111
1000

- Claim: we will never run out of tokens.
- **Helper claim: the number of available tokens = number of 1's in our current state (n's binary representation).**

Increasing counter represented by bits

0000
0001
0010
0011
0100
0101
0110
0111
1000

- Claim: we will never run out of tokens.
- **Helper claim: the number of available tokens = number of 1's in our current state (n's binary representation).**
- Proof by induction:
  - For n_0=0 we have 0 tokens.

Increasing counter represented by bits

0000
0001
0010
0011
0100
0101
0110
0111
1000

- Claim: we will never run out of tokens.
- **Helper claim: the number of available tokens = number of 1's in our current state (n's binary representation).**

- Proof by induction:
  - For n_0=0 we have 0 tokens.
  - Assume correctness for n.
  - In step n+1: we switch k*1→0 and 1*0→1 (by algorithm definition)
  - We lost k-1 tokens from our pool. k+1 operations, 2 income.
  - We lost k-1 1's. k switched to zero, 1 gained.
  - → Number of tokens = number of 1's

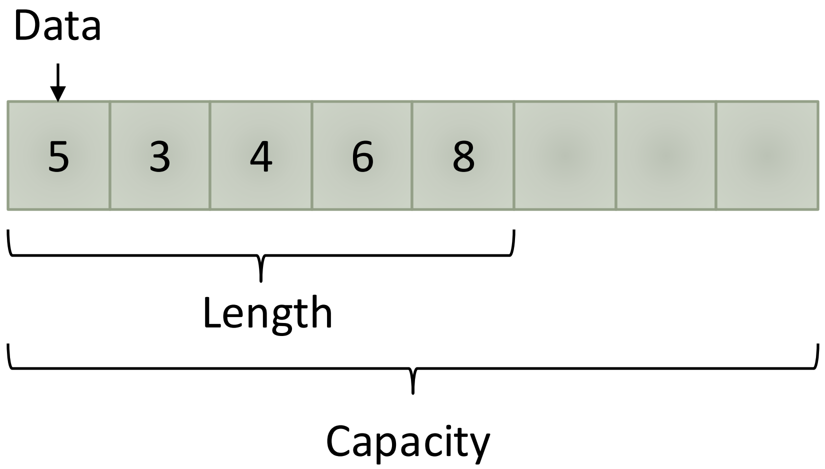
Self-Expanding List

- A list that mimics infinite capacity – you can always add more elements.
- Implemented as a simple table.
- Supports retrieval(i) and append(x).

Self-Expanding List

- A list that mimics infinite capacity – you can always add more elements.
- Implemented as a simple table.
- Supports retrieval(i) and append(x).

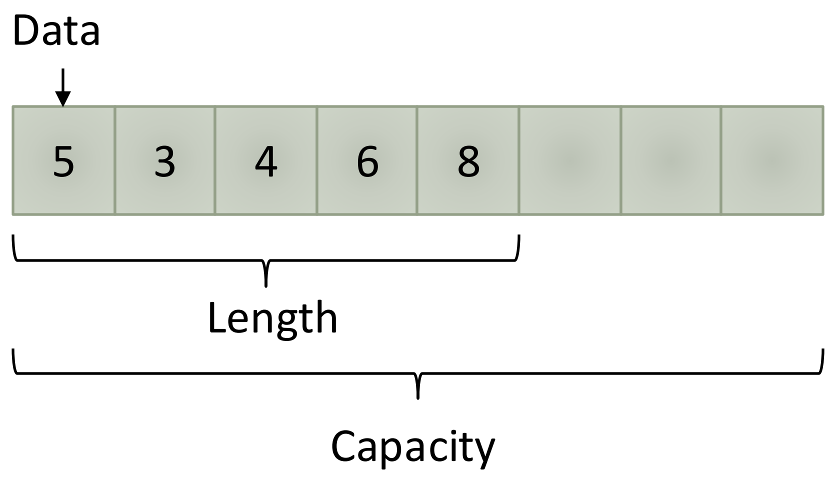
- Implementation:
  - **Data**: pointer to the first cell
  - **Capacity**: number of cells
  - **Length**: number of elements (used cells)



Self-Expanding List - Append

Append element x to list A

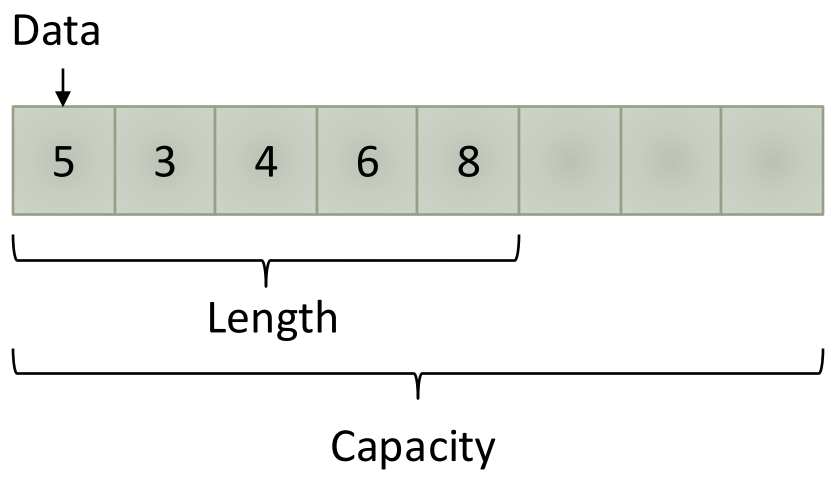
- If Length(A) = Capacity(A): // Out of available space
  - Allocate memory sized Capacity*2
  - Copy data to new memory
  - Data(A) ← new pointer
  - Capacity(A) ← Capacity(A)*2
- A[Length(A)+1] = x
- Length(A) ← Length(A)+1



Self-Expanding List - Append

<u>Append element x to list A</u>

- If Length(A) = Capacity(A):   // Out of available space
  - Allocate memory sized Capacity*2
  - Copy data to new memory
  - Data(A) ← new pointer
  - Capacity(A) ← Capacity(A)*2
- A[Length(A)+1] = x
- Length(A) ← Length(A)+1



Self-Expanding List - Append

<u>Example</u>

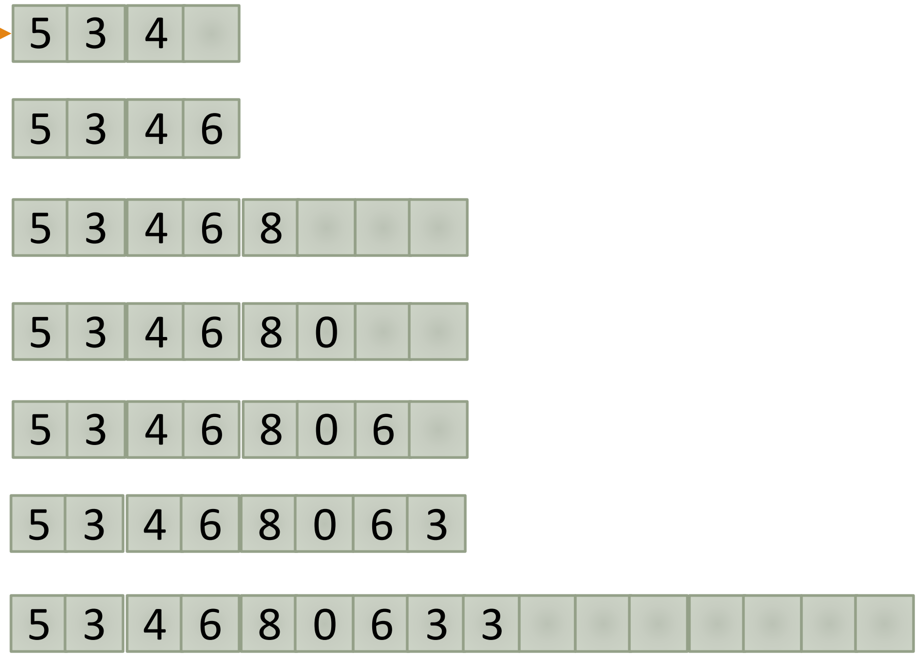
- Start empty:
  - Capacity=1, Length=0, Data= $\boxed{}$
- Add 5:
  - A[1] ← 5
  - Capacity=1, Length=1, Data= $\boxed{5}$
- Add 3:
  - Reallocate memory $\boxed{}\boxed{}$
  - Copy data $\boxed{5}\boxed{}$
  - A[2] ← 3 $\boxed{5}\boxed{3}$
  - Capacity=2, Length=2

Self-Expanding List - Append

<u>Example</u>

- Start empty:
  - Capacity=1, Length=0, Data=
- Add 5:
  - A[1] ← 5
  - Capacity=1, Length=1, Data= 5
- Add 3:
  - Reallocate memory
  - Copy data 5
  - A[2] ← 3 5 3
  - Capacity=2, Length=2

Add more elements



Self-Expanding List - Append

- **Claim: append is amortized $O(1)$ operations**

Self-Expanding List - Append

- **Claim: append is amortized $O(1)$ operations**
- Sketch for proof:
  - How much does each command cost?

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

  - $\boldsymbol{T(m) = \sum_{i=1}^{m} c_i \le m + \sum_{j=0}^{\lfloor log(m) \rfloor} 2^j = m + \frac{2^{\lfloor log(m) \rfloor + 1} - 1}{2-1} \le m + 2m}$

Self-Expanding List - Append

- **Accounting method:**

Self-Expanding List - Append

- **Accounting method:**
- On each operation we receive 3 tokens.
- Assignment costs 1 token.
- Reallocating to length k costs $\frac{k}{2}$ tokens.

i	1	2	3	4	5	6	7	8	9
$t(i)$	1	2	3	1	5	1	1	1	9

Self-Expanding List - Append

- **Accounting method:**
- On each operation we receive 3 tokens.
- Assignment costs 1 token.
- Reallocating to length k costs $\frac{k}{2}$ tokens.
- **Need to prove: we always have enough tokens.**

Self-Expanding List - Append

- **Proof:**

Self-Expanding List - Append

- **Proof:**
- Claim: Each time append requires a resize, pool is left with 3 tokens.

Self-Expanding List - Append

- **Proof:**
- Claim: Each time append requires a resize, pool is left with 3 tokens.
- Base: Let’s count together

insert	capacity	pool
0	1	0
1	1	2
2	2	3

Self-Expanding List - Append

- **Proof:**
- Claim: Each time append requires a resize, pool is left with 3 tokens.
- Base: Let's count together

insert	capacity	pool
0	1	0
1	1	2
2	2	**3**
3	4	**3**
1	4	5
5	8	**3**

Self-Expanding List - Append

- **Proof:**
- Claim: Each time append requires a resize, pool is left with 3 tokens.
- Base: Let's count together
- Let's assume correctness for resize to capacity $2k$ and prove for resize to $4k$.

insert	capacity	pool
0	1	0
1	1	2
2	2	3

Self-Expanding List - Append

- **Proof:**
- Claim: Each time append requires a resize, pool is left with 3 tokens.
- Base: Let’s count together
- Let’s assume correctness for resize to capacity $2k$ and prove for resize to $4k$.
- Let's assume we just entered the $k+1$’s element. Pool has currently 3 tokens (I.H.)

insert	capacity	pool
0	1	0
1	1	2
2	2	3

Self-Expanding List - Append

- **Proof:**
- Claim: Each time append requires a resize, pool is left with 3 tokens.
- Base: Let's count together
- Let's assume correctness for resize to capacity $2k$ and prove for resize to $4k$.
- Let's assume we just entered the $k+1$'s element. Pool has currently 3 tokens (I.H.)
- Before adding the $2k+1$'s item (causing a resize), we add $k-1$ items. Pool will have $2k+1$ tokens. ((k-1)*3 – (k-1) = 2k-2 , +3 from I.H.)

insert	capacity	pool
0	1	0
1	1	2
2	2	3

Self-Expanding List - Append

- **Proof:**
- Claim: Each time append requires a resize, pool is left with 3 tokens.
- Base: Let’s count together
- Let’s assume correctness for resize to capacity $2k$ and prove for resize to $4k$.
- Let's assume we just entered the $k+1$’s element. Pool has currently 3 tokens (I.H.)
- Before adding the $2k+1$'s item (causing a resize), we add $k-1$ items. Pool will have $2k+1$ tokens. ((k-1)*3 – (k-1) = 2k-2 , +3 from I.H.)
- When adding the $2k+1$’s item, we gain 3 tokens ($2k+4$ total), use $2k$ to copy all existing elements, and one more token for new element. We end up with 3 tokens.

insert	capacity	pool
0	1	0
1	1	2
2	2	3