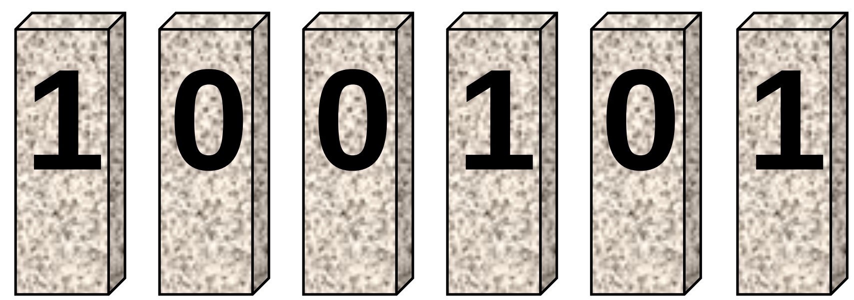
Amortized Analysis

The n-bit Counter

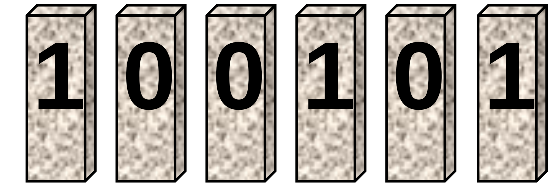
Problem of the Day

Rob has a startup. Each time he gets a new user, he increments a giant stone counter his investors (VC) erected in downtown San Francisco — that's a sequence of 6 stone tablets with 0 on one side and 1 on the other.



Every time a user signs up, he increments the counter. But the power company charges him $1 each time he turns a tablet. He is tight on funding, so he needs to pass that cost to the users. He wants to charge users as little as possible to cover his cost (the VC promised to erect new tablets as his user base grows).

How much should he charge each new user?



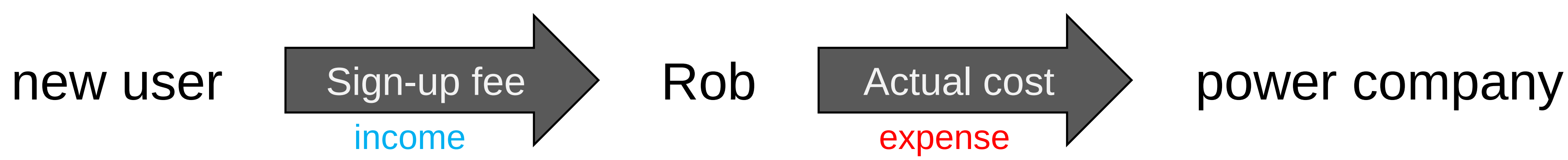
Understanding the Problem

- Each time a user signs up, increment the counter
  - pay the power company $1 per bit flipped
  - charge the user $x to cover the cost
    - make x as little as possible

This is an expense

This is income

- Cash flow:



- Implicit requirements
  - Always have enough cash to pay the power bill

1 0 0 1 0 1

Understanding the Problem

- What is the cost of signing up the first few users?

Counter	User #	Cost
0 0 0 0 0 0		
	1	1
0 0 0 0 0 1		
	2	2
0 0 0 0 1 0		
	3	1
0 0 0 0 1 1		
	4	3
0 0 0 1 0 0		
	5	1
0 0 0 1 0 1		
	6	2
0 0 0 1 1 0		
	7	1
0 0 0 1 1 1		
	8	4
0 0 1 0 0 0		

- **cost** = number of bits flipped
- Sign-up expense varies
  - many as low as $1
  - maximum gets higher and higher
    - and further apart

Let $cost_i$ be the cost of signing up user i

Solution #1

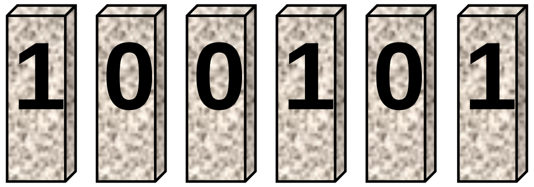
- Charge each user the actual cost
  - Rob can't charge different users different costs

He's not running an airline!

✘

- Implicit requirements
  - *Always have enough cash to pay the power bill*
  - Charge every user the same amount

New



Solution #2

- Charge each user the maximum possible cost
  - How much would that be?
    - 6 bits, so $6
    - in general, for an n bit counter, cost is $n
  - This is too much — Nobody would sign up
    - Rob would be making a big profit — Frowned upon in the startup world

✘

- Implicit requirements
  - *Always have enough cash to pay the power bill*
  - *Charge every user the same amount*
- *Recall* Goal: Charge little

1 0 0 1 0 1

Understanding the Problem

- Let's write down Rob's total cost over time

Counter	User #	Cost	Total cost
0 0 0 0 0 0			
	1	1	1
0 0 0 0 0 1			
	2	2	3
0 0 0 0 1 0			
	3	1	4
0 0 0 0 1 1			
	4	3	7
0 0 0 1 0 0			
	5	1	8
0 0 0 1 0 1			
	6	2	10
0 0 0 1 1 0			
	7	1	11
0 0 0 1 1 1			
	8	4	15
0 0 1 0 0 0			

- ○ **total_cost** = sum of all cost up to current sign-up

  $total\_cost_k = \Sigma_{i<k}\ cost_i$

- ○ Observation:
  - ➢ total_cost < 2 * user#
    - ❑ at most,
      total_cost = 2 * user# - 1
      for most expensive increments

Idea:
charge users $2!

That's
$\lceil max_k\ (\Sigma_{i<k}\ cost_i)/k \rceil$

100101

Solution #3

- Charge each user $2

This is reasonable for users

  - If the actual cost is less, put the difference in a savings account
  - If the actual cost is more, pay the difference from these savings
  - *Does this work?*
    - Does he always have enough cash to pay the power bill?

- *Implicit requirements*
  - *Always have enough cash to pay the power bill*
    - *savings ≥ 0, always*
  - *Charge every user the same amount*
- *Goal: charge little*

1 0 0 1 0 1

Understanding the Problem

- Let's write down the total income and savings over time

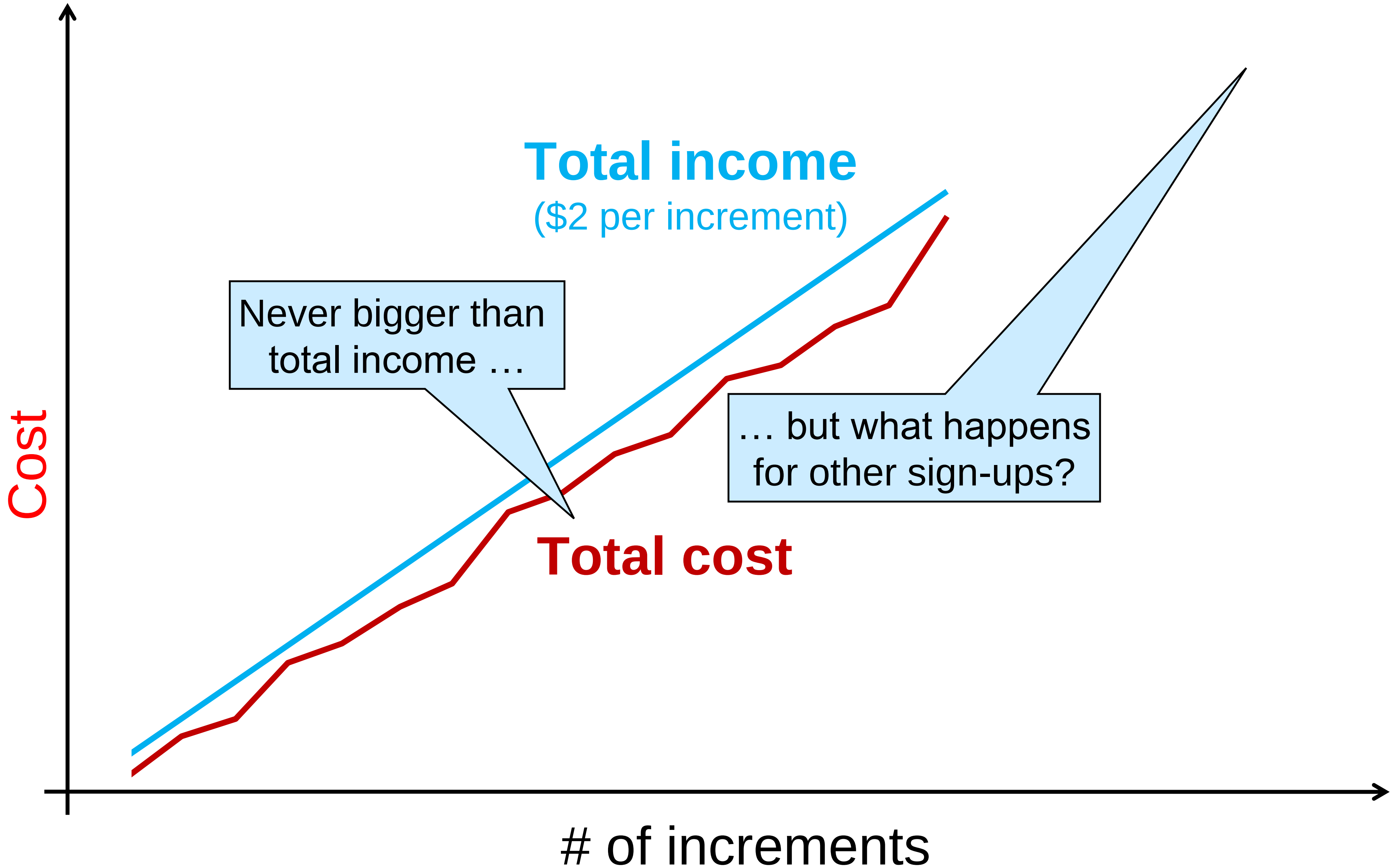
Counter	User #	Cost	Total cost	Total income	Savings
0 0 0 0 0 0					
	1	1	1	2	1
0 0 0 0 0 1					
	2	2	3	4	1
0 0 0 0 1 0					
	3	1	4	6	2
0 0 0 0 1 1					
	4	3	7	8	1
0 0 0 1 0 0					
	5	1	8	10	2
0 0 0 1 0 1					
	6	2	10	12	2
0 0 0 1 1 0					
	7	1	11	14	3
0 0 0 1 1 1					
	8	4	15	16	1
0 0 1 0 0 0					

- **total_income** = 2 * user#
  - $2 per user
- **savings** = total_income – total_cost
- enough to pay bills
  - savings + $2 ≥ next cost
- equivalently
  - savings ≥ 0
    - no need to borrow

Problem Solved?

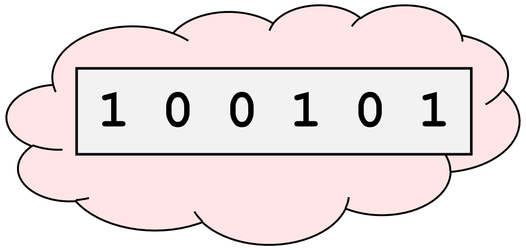
- Charging users $2 seems to work …
  - ➢ it works for the first 8 users!
- … but how can we be sure?
  - ○ at some point,
    - ➢ Rob may not have enough cash to cover the costs
    - ➢ or he may run a big profit
  - ○ or both at different times
- Let's turn this into a *computer science problem*

Problem Solved?



Analyzing the n-bit Counter

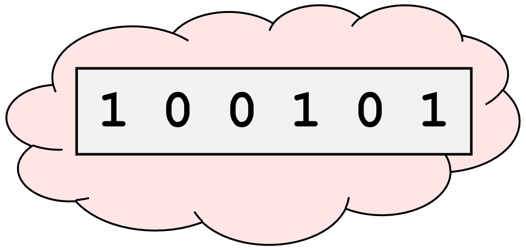
The n-bit Counter Revisited



- View the counter as a **data structure**
  - n bits
- and a user sign-up as an **operation**
  - The number of bit flips is the **cost** of performing the operation
  - Worst-case cost is O(n)
    - flip all n bits

- Then, "*enough to pay bills*" and "*savings ≥ 0*" are like **data structure invariants** …
  - … but about cost
  - Wait!
    - what are the savings in the data structure?
    - what does the $2 fee represent?

So far, data structure invariants have been about the representation of the data structure, never about cost



What are the Savings?

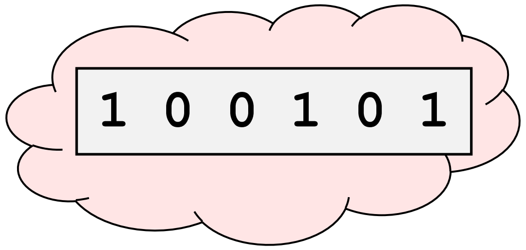
- The savings are equal to the number of bits set to 1

Counter	User #	Savings
0 0 0 0 0 0		
	1	1
0 0 0 0 0 1		
	2	1
0 0 0 0 1 0		
	3	2
0 0 0 0 1 1		
	4	1
0 0 0 1 0 0		
	5	2
0 0 0 1 0 1		
	6	2
0 0 0 1 1 0		
	7	3
0 0 0 1 1 1		
	8	1
0 0 1 0 0 0		

- Visualize this by placing a **token** on top of each 1-bit in the counter
- A token represents a unit of cost
  - = $1 = cost of one bit flip
  - we earn tokens by charging for an increment
    - 2 tokens per call to the operation
      - no matter how many bits actually get flipped
  - we spend tokens performing the increment
    - 1 token per actual bit flip
    - variable number of bit flips per increment

O(n) in worst case

The Token Invariant

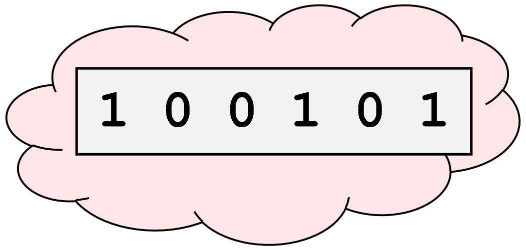


- If we
  - earn 2 tokens per increment and
  - spend 1 token for each bit flipped to carry it out,
- we claim that
  - the tokens in saving are ***always*** equal to the number of 1-bits

- This is our **token invariant**

  Well, this is a ***candidate*** invariant: we still need to show it is valid

  # tokens = # 1-bits
  - if valid, then "*saving* ≥ *0*" holds
    - because there can't be a negative number of 1-bits



Proving the Token Invariant

- To prove it is valid, we need to show that it is **preserved** by the operations
  - if the invariant holds before the operation, it also holds after

Just like loop invariants

```
while (i < n)
//@loop_invariant 0 <= i && i < \length(A);
```

In fact, just like data structure invariants!

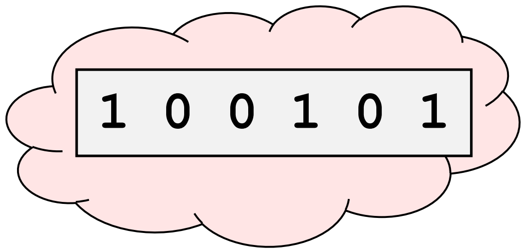
```
void enq(queue* Q, string x)
//@requires is_queue(Q);
//@ensures is_queue(Q);
```

- Preservation:

 # 1-bits before + 2 - # bit flips = # 1-bits after

tokens in savings | tokens from user | cost of operation | # tokens in savings

- i.e., if # tokens == # 1-bits *before* incrementing the counter, then # tokens == # 1-bits also *after*
- if true, then “*savings ≥ 0, always*” holds
  - because # 1-bits after can’t be negative



Proving the Token Invariant

- *To prove it is valid, we need to show that it is **preserved** by the operations*
  - *if the invariant holds before the operation, it also holds after*

- Should we also prove that it is true ***initially***?
  - kind of …
  - … we are missing an operation:
    - creating a new counter initialized to 0

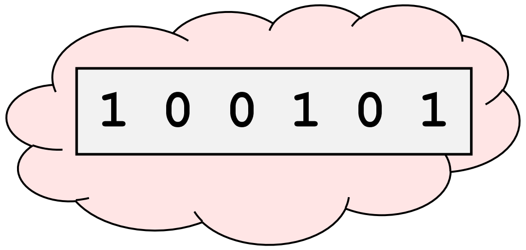
0 0 0 0 0 0

  - Does the token invariant hold for a new counter?

    # tokens == # 1-bits

    - no users yet, so no tokens ✓
    - no 1-bits

- This is a special case of preservation (no "before")

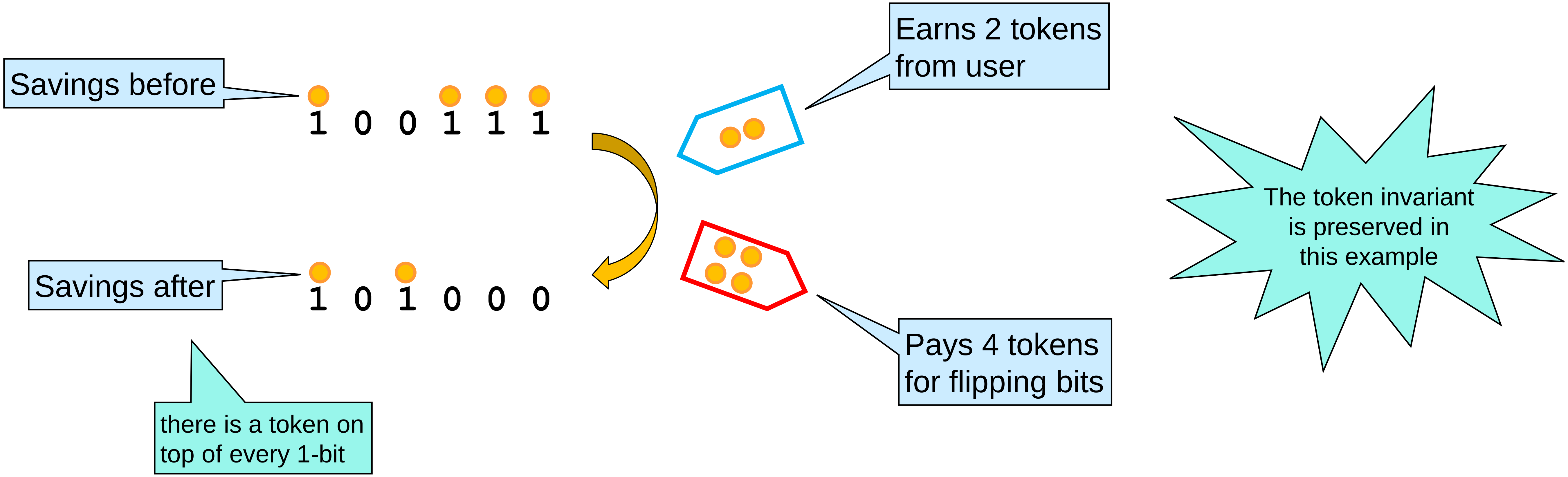


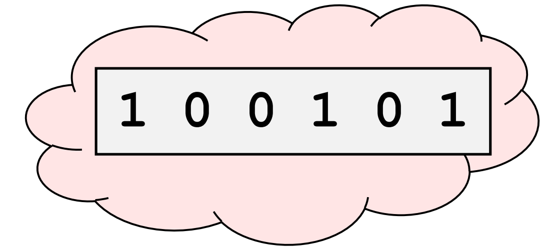
Proving the Token Invariant

1-bits before + 2 - # bit flips = # 1-bits after

- *i.e., if # tokens == # 1-bits before incrementing the counter, then # tokens == # 1-bits also after*

- Let's check it on an example





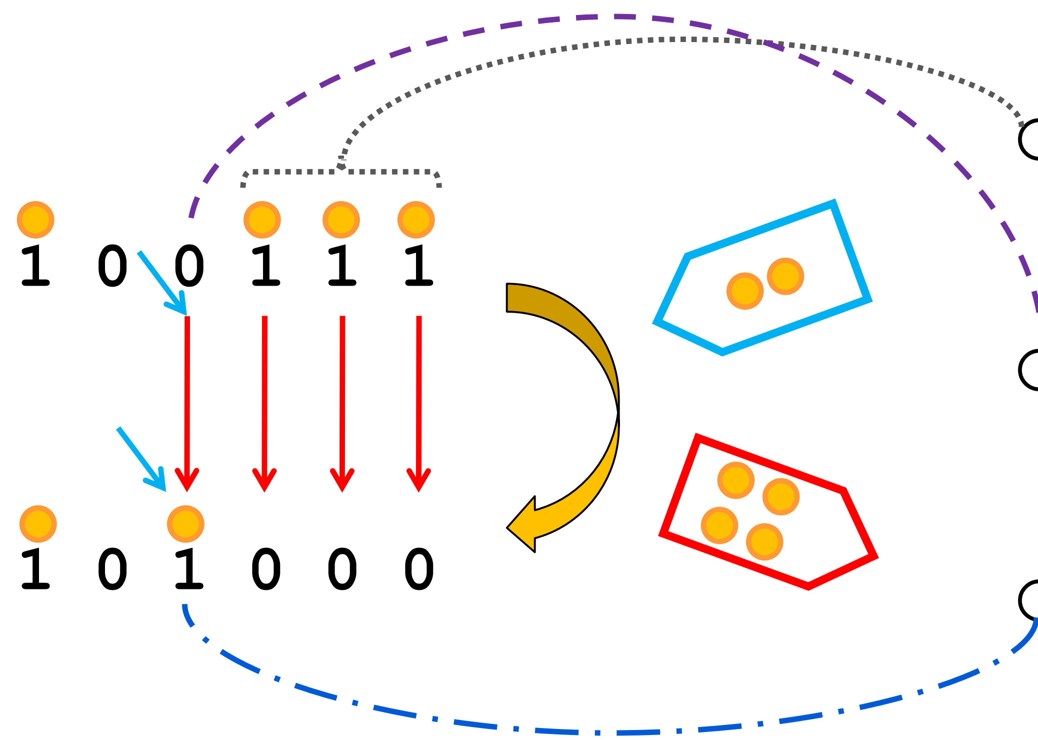
Proving the Token Invariant

1-bits before + 2 - # bit flips = # 1-bits after

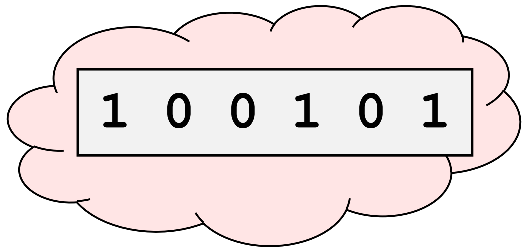
- *i.e., if # tokens == # 1-bits before incrementing the counter, then # tokens == # 1-bits also after*

- How are the tokens used?

These are all the 1-bits to the right of the rightmost 0-bit



- each 1-bit that is flipped
  - paid by associated token in savings
- 0-bit that is flipped
  - paid by 1 token from user
- token for the new 1-bit
  - paid by 1 token from user

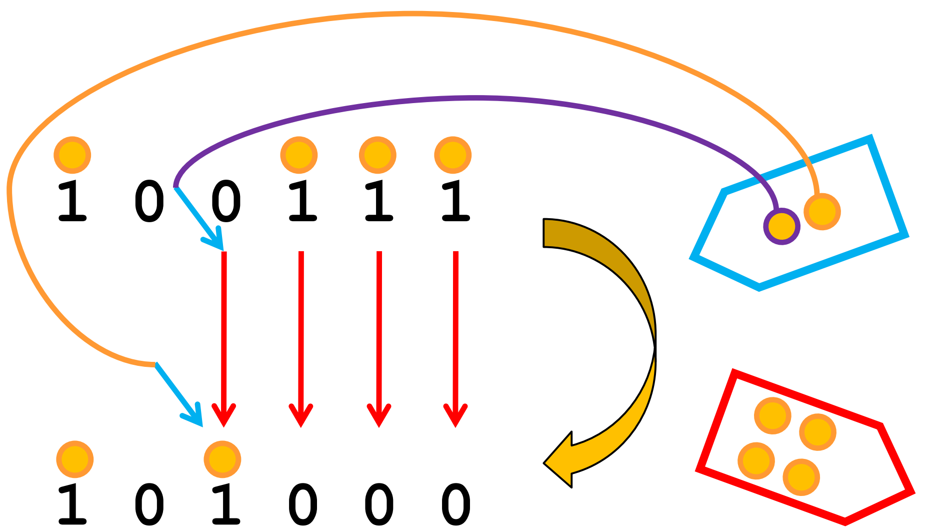


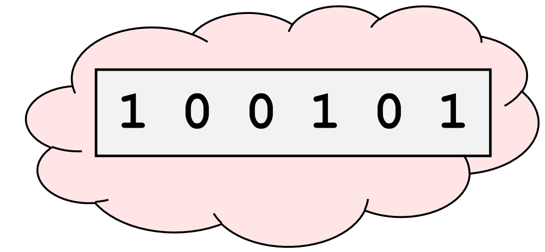
Proving the Token Invariant

1-bits before + 2 - # bit flips = # 1-bits after

- *i.e., if # tokens == # 1-bits before incrementing the counter, then # tokens == # 1-bits also after*

- How are the tokens used?
  - tokens associated to bits:
    - used to flip bit from 1 to 0
  - 2 tokens from user
    - 1 token to flip rightmost 0-bit to 1
    - 1 token to place on top of new rightmost 1-bit

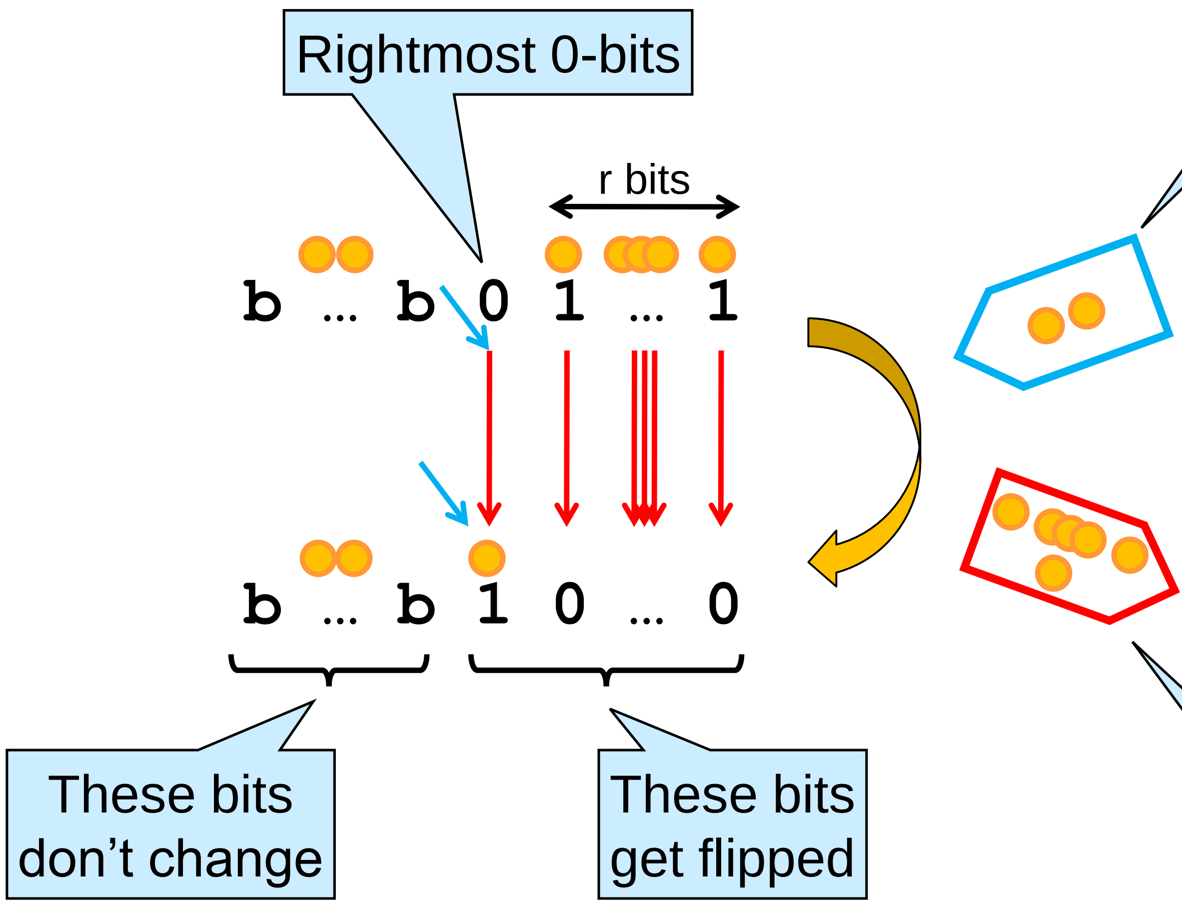




Proving the Token Invariant

 # 1-bits before + 2 - # bit flips = # 1-bits after

- General situation



  - rightmost 1-bits are flipped
    - paid by associated token in savings
  - rightmost 0-bit is flipped
    - paid by 1 token from user
  - token for the new rightmost 1-bit
    - paid by 1 token from user
  - other bits don't change

✓

Solution #3

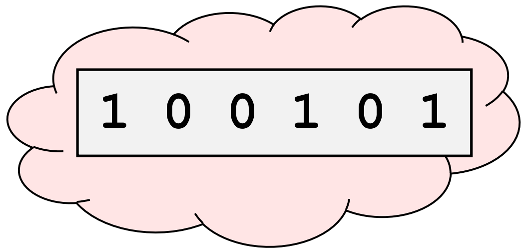
- *Charge each user $2* (This is reasonable for users)
  - *If the actual cost is less, put the difference in a savings account*
  - *If the actual cost is more, pay the difference from these savings*

  - *Does this work?*
    - YES!

✓

- *Implicit requirements*
  - *Always have enough cash to pay the power bill*
    - *savings ≥ 0, always*
  - *Charge every user the same amount*
- *Goal: charge little*

What does the $2 fee Represent?



- We **pretend** that each increment costs 2 tokens
  - even though it may cost as much as n, or as little as 1
- This is the **amortized cost** of an increment
  - not the actual cost of an increment (which varies)
  - but enough to cover the actual cost over a sequence of operations
    - inexpensive increments pay for expensive ones
    - prepay future cost
  - note that 2 is in O(1)

an increment can cost as much as O(n) …

- **Worst case cost** of increment: **O(n)**
- **Amortized cost** of increment: **O(1)**

… but it is **as if** each increment in the sequence cost O(1)

Amortized Complexity Analysis

24

Sequences of Operations

Our example

- We have a **data structure** on which we perform a **sequence of** k **operations**

n-bit counter

k increments

- Normal complexity analysis tells us that the cost of the sequence is bounded by k times the worst-case complexity of the operations

k times O(n): that's O(kn)

- The actual **total cost** of the sequence may be much less
  - total_cost = $\Sigma_{i<k}$ cost_of_operation_i

O(k) for the whole sequence

In the table, total_cost ≤ 2k-1

- Define the **amortized cost** as the actual total cost divided by the length of the sequence
  - amortized_cost = total_cost / k
    - *rounded up*

O(k) divided by k: that's O(1)

Amortized Cost

The actual total cost divided by the length of the sequence

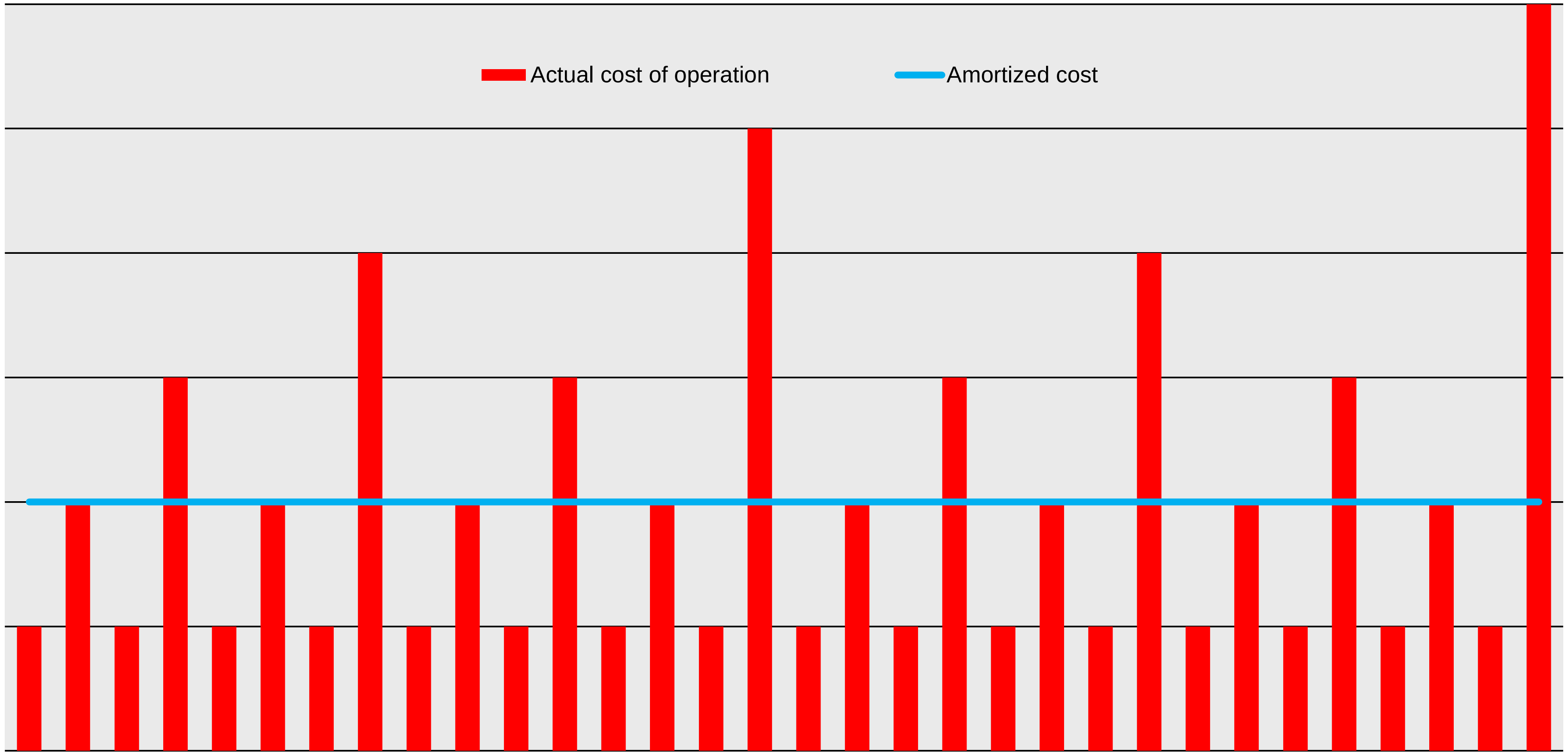
- This is the **average** of the actual total cost of the operations over the sequence
  - amortized_cost = ($\Sigma_{i<k}$ cost_of_operation_i) / k
    - *rounded up*
- **As if** every operation in the sequence cost the same amount
  - This amount is the amortized cost
- Just looking at the worst-case complexity is too pessimistic
  - it tells us about the cost of an operation in isolation
  - but here the operation is part of a sequence

a few operations may be expensive, but on average they are pretty cheap

Amortized Cost

The actual total cost divided by the length of the sequence

- amortized_cost = $(\Sigma^{k}_{i=0}$ cost_of_operation_i$) / k$
  - *rounded up*



The Old Notion of "Average"

- Recall Quicksort
  - Worst-case complexity: $O(n^2)$
    - when we were really unlucky and systematically picked bad pivots
  - ***Average-case complexity:*** O(n log n)
    - what we expected for an average array
      - very unlikely that all pivots are bad

- What were we averaging over?
  - The likelihood of a series of bad pivots in all possible arrays
    - a probability distribution
- Average-case complexity has to do with **chance**
  - There is a very low probability that the actual cost will be $O(n^2)$ on any given input
    - but it may happen
      - the actual cost depends on what array we are handed

A New Notion of "Average"

- Average-case complexity: average over input distribution
  - The actual cost has to do with chance
- Amortized complexity: average over a sequence of operations
  - We know the exact cost of every operation
    - so we know the exact cost of the sequence overall
    - this is an ***exact*** calculation
      - no chance involved

Basically an average over time

- Difference
  - average over time

  vs.
  - average over chance

Amortized complexity

Average complexity

Amortization in Practice (I)

- A baker buys a $100 sack of flour every 100 loaves of bread
  - 1^{st} loaf costs $100
  - 2^{nd}, 3^{rd}, …, 100^{th} costs nothing

Actual cost to the baker

- The baker charges $1 for each loaf
  - average cost over all 100 loafs

The baker charges you an amortized cost

$100

$1

Here, both worst case and amortized cost are O(1)

- *not as dramatic as O(n) vs. O(1)*

Amortization in Practice (II)

- Your smartphone use varies over time
  - some days you barely go online
  - other days you binge-watch movies for hours on end

Actual cost to your provider

- Your provider charges you a fixed monthly cost
  - average cost over time and over all customers (+ profit)

Your provider charges you an amortized cost

When to Use Amortized Analysis?

- We have a **sequence** of k operations on a data structure
  - the sequence starts from a well-defined state
  - each operation changes the data structure

- We expect the actual cost of the whole sequence to be **much less** than k times the worst-case complexity of the operations
  - a few operations are expensive
  - many are cheap
    - **Use the inexpensive operations to pay for the expensive operations**

We prepay for future costs

How to do Amortized Analysis?

- Invent a notion of **token**
  - represents a unit of cost
- Determine how many tokens to charge for each operation
  - ➢ this is the candidate **amortized cost**
  - (see next)
- Specify the **token invariant**
  - for any instance of the data structure, how many tokens need to be saved
- Prove that every operation **preserves** the token invariant
  - if the invariant holds before, it also holds after

what we pretend the operation costs

This is like point-to reasoning

saved tokens before + amortized cost – actual cost = saved tokens after

How to Determine the Amortized Cost?

candidate

How many tokens to charge?

1. Draw a short sequence of operations
   - ➢ make it long enough so that a pattern emerges
2. Write the cost of each operation
3. Flag the most expensive so far
4. For each operation, compute the total cost up to it
5. Divide the total cost of the most expensive operations by the operation number in the sequence
6. Round up — that's the candidate amortized cost

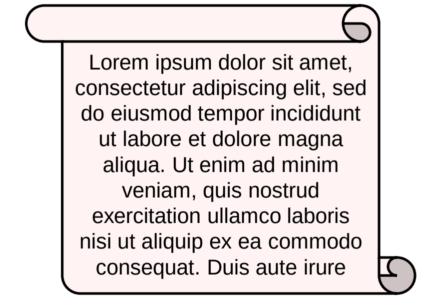
This is called the **accounting method**

Counter (1)	User # (1)	Cost (2)	Total cost (4)	Div (5)
000000				
	1 (3)	1	1	
000001				
	2 (3)	2	3	1.5
000010				
	3	1	4	
000011				
	4 (3)	3	7	1.75
000100				
	5	1	8	
000101				
	6	2	10	
000110				
	7	1	11	
000111				
	8 (3)	4	15	1.875
001000				

(6) 2

This is like operational reasoning: forming a conjecture that we then prove using point-to reasoning

Unbounded Arrays



Another Problem

- We want to store all the words in a text file into an array-like data structure so that we can access them fast
  - we don't know how many words there are ahead of time
    - we add them one at a time
- Use an array?
  - access is O(1)
  - but we don't know how big to make it! ✗
    - too small and we run out of space
    - too big and we waste lots of space
- Use a linked list?
  - we can make it the exact right size! ✗
  - but access is O(n)

where n is the number of words in the file

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure

Another Problem

- *We want to store all the words in a text file into an array-like data structure so that we can access them fast*
  - *we don't know how many words there are ahead of time*

- We want an **unbounded array**
  - ➢ a data structure that combines the best properties of arrays and linked lists
  - access is ***about*** O(1)
  - and size is ***about*** right

That's what **amortized cost** is all about!

Never too small, and not extravagantly big

- Same operations as regular arrays, plus
  - a way to add a new element at the end
  - a way to remove the end element

The Unbounded Array Interface

Unbounded Array Interface

```
// typedef ______* uba_t;

int uba_len(uba_t A)                         // O(1)
  /*@requires A != NULL;                      @*/
  /*@ensures \result >= 0;                    @*/ ;

uba_t  uba_new(int size)
  /*@requires 0 <= size ;                     @*/
  /*@ensures \result != NULL;                 @*/
  /*@ensures uba_len(\result) == size;        @*/ ;

string uba_get(uba_t A, int i)               // O(1)
  /*@requires A != NULL;                      @*/
  /*@requires 0 <= i && i < uba_len(A);       @*/ ;

void uba_set(uba_t A, int i, string x)       // O(1)
  /*@requires A != NULL;                      @*/
  /*@requires 0 <= i && i < uba_len(A);       @*/ ;

void uba_add(uba_t A, string x)              // O(1) amt
  /*@requires A != NULL;                      @*/ ;

string uba_rem(uba_t A)                      // O(1) amt
  /*@requires A != NULL;                      @*/
  /*@requires  0 < uba_len(A);                @*/ ;
```

This is exactly the self-sorting array interface with “ssa” renamed to “uba”

Doesn’t keep elements sorted this time

Add x as the last element of A
- A grows by 1 element

Constant **amortized** complexity (***worst-case*** could be a lot higher)

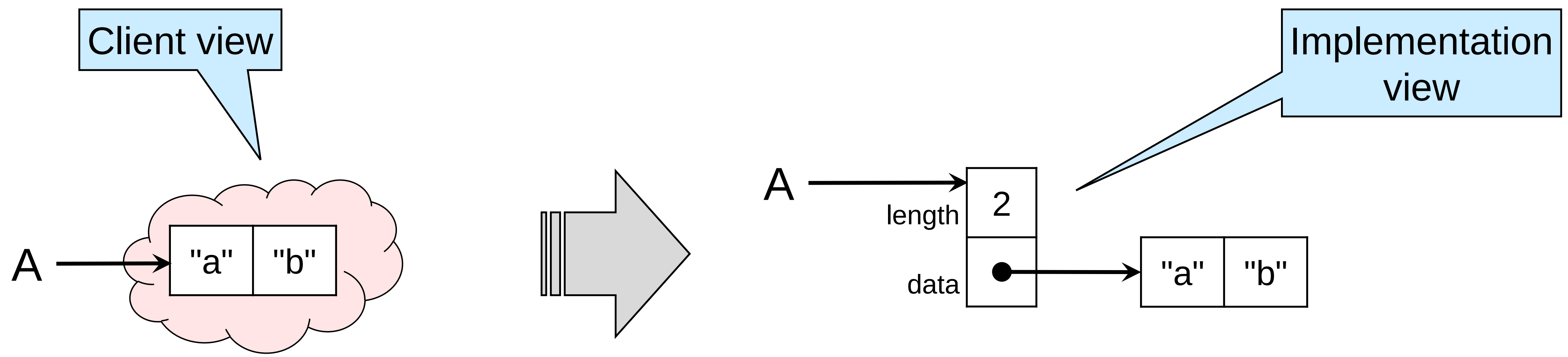
Remove and return the last element of A
- A shrinks by 1 element

Towards an Implementation

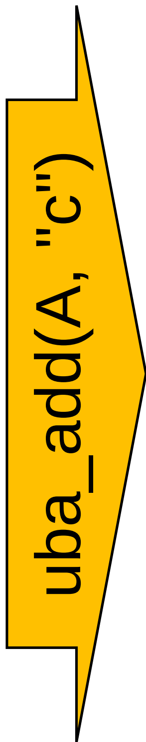
- Recall the SSA concrete type

```
// Implementation-side type
struct ssa_header {                    // Concrete type
  int length;        // 0 <= length
  string[] data;   // \length(data) == length
};
```

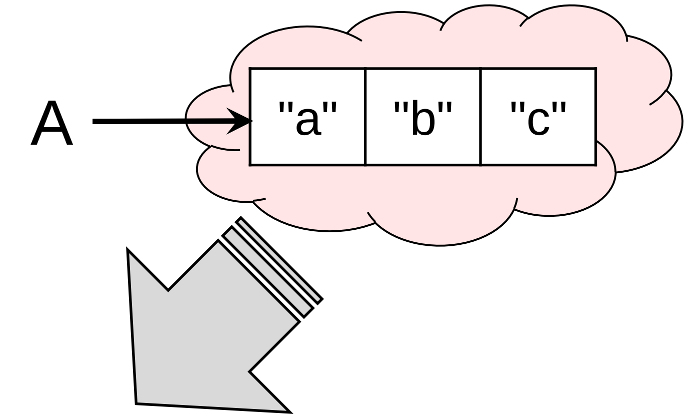
These are representation invariants



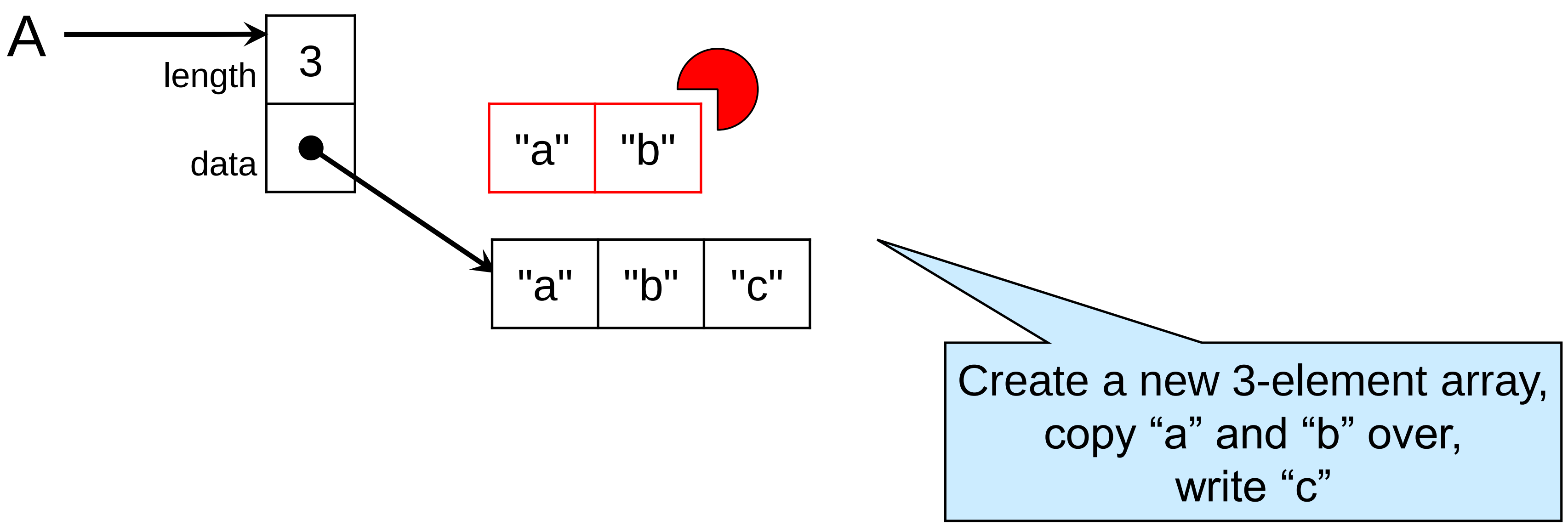
- Can we reuse it for unbounded arrays?
  - Let's add "c" to it



Towards an Implementation

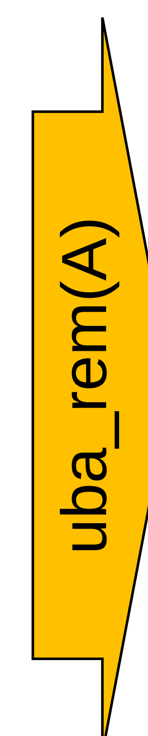


- Let's add "c" to it

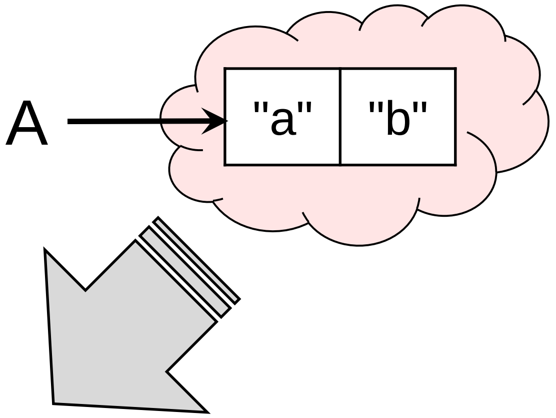


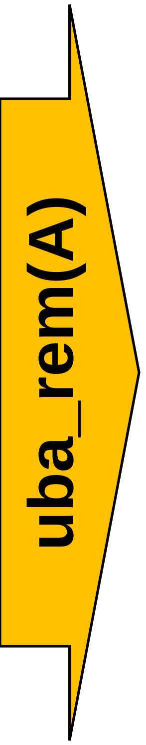
  - ○ Copying the old elements to the new array is expensive
    - ➢ O(n) for an n-element array

- Next, let's remove the last element

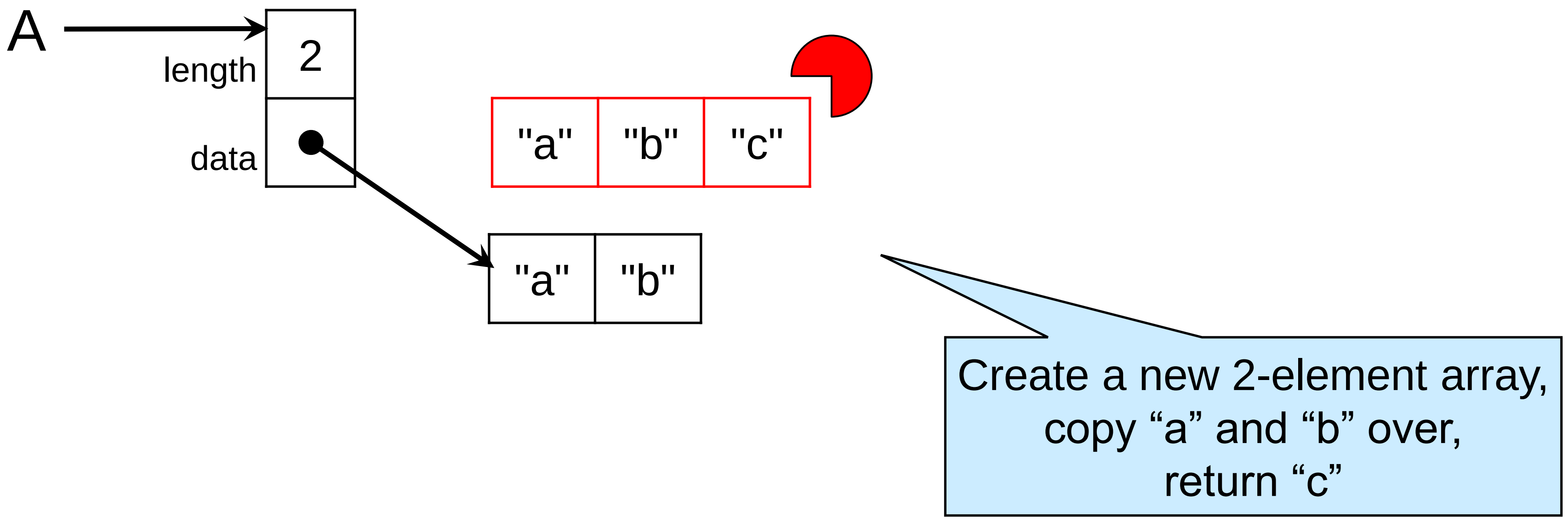


Towards an Implementation



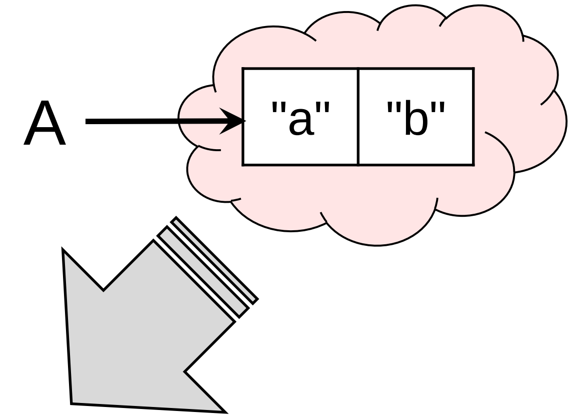


- Next, let's remove the last element

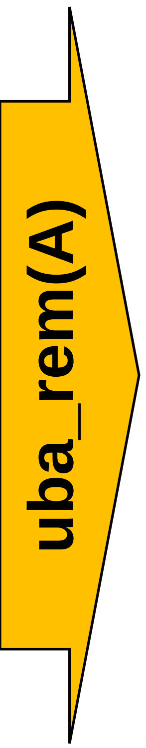


  - Copying the remaining elements to the new array is expensive
    - again, O(n)

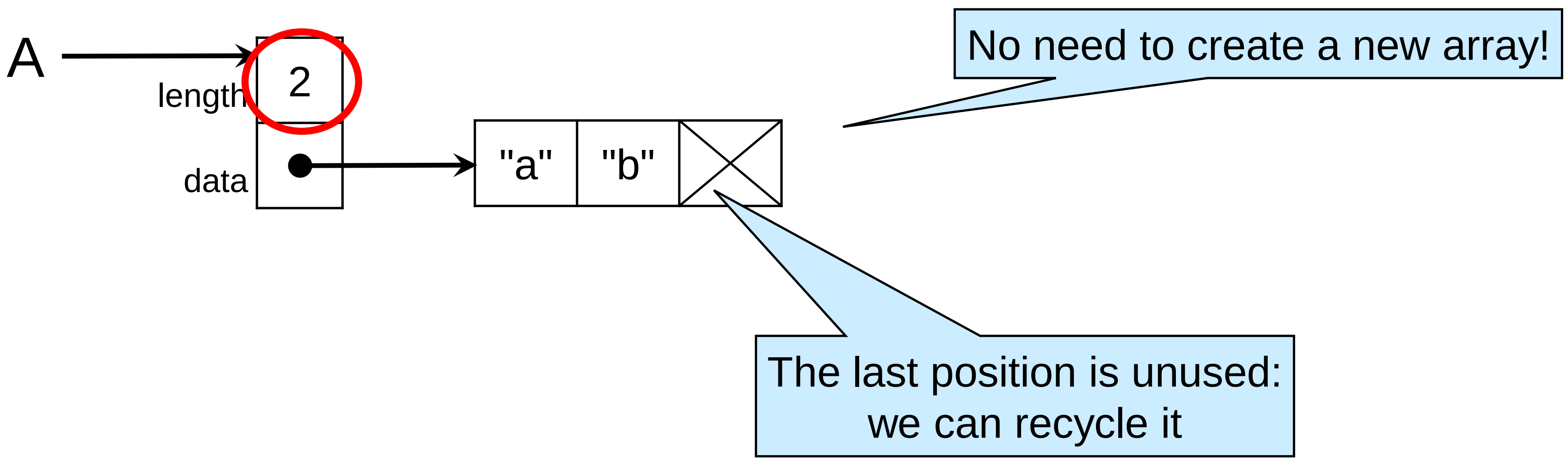
- *Can we do better?*



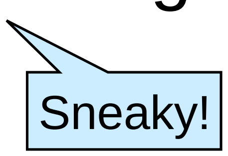
Towards an Implementation



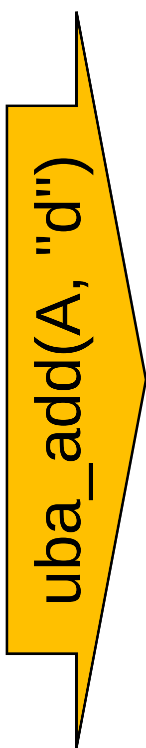
- Can we do better?
  - Maybe leave the array alone and just change the length!



  - We did not do any copying, just updated the length
    - O(1) for an n-element array

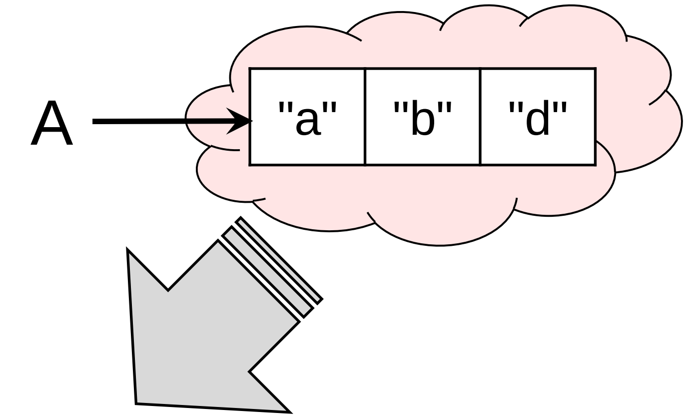


- Let’s continue by adding “d”

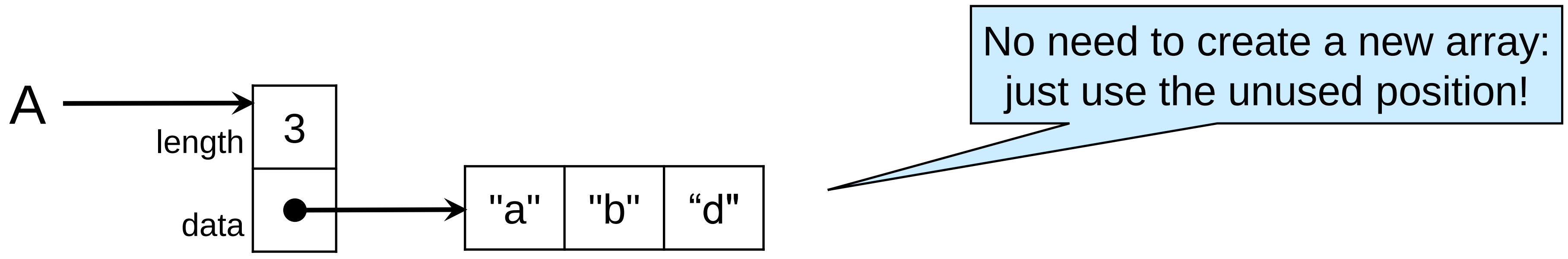


uba_add(A, "d")

Towards an Implementation



- Let's continue by adding "d"



  - All we did is one write!
    - O(1)

- *But is it safe?*
  - We have no way to know the true length of the array!
    - it used to be that A->length == \length(A->data)
    - when executing

      A->data[2] = "d"

      we don't know if we are writing out of bounds
    - now, all we know is that A->length <= \length(A->data)

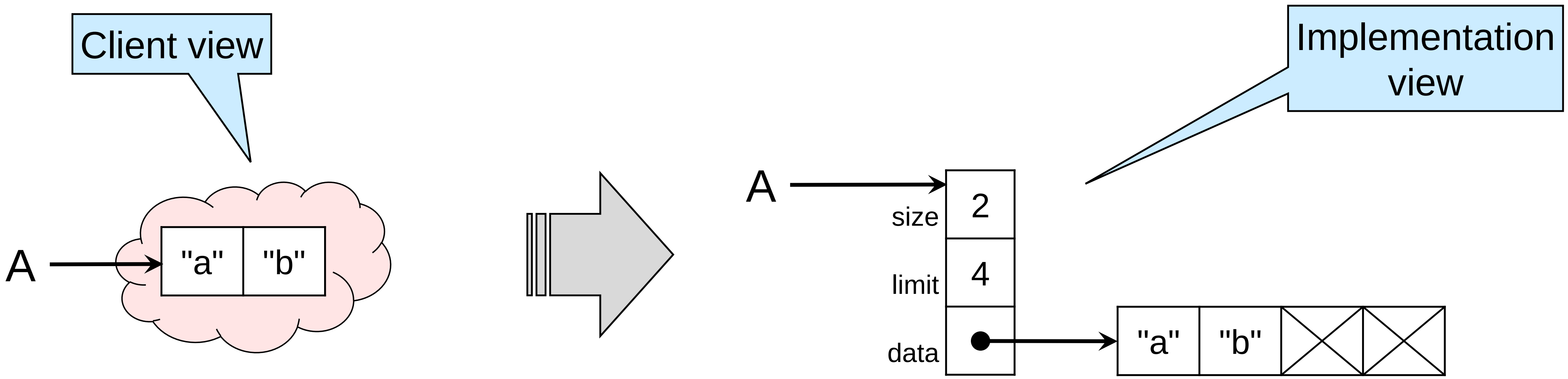
Towards an Implementation

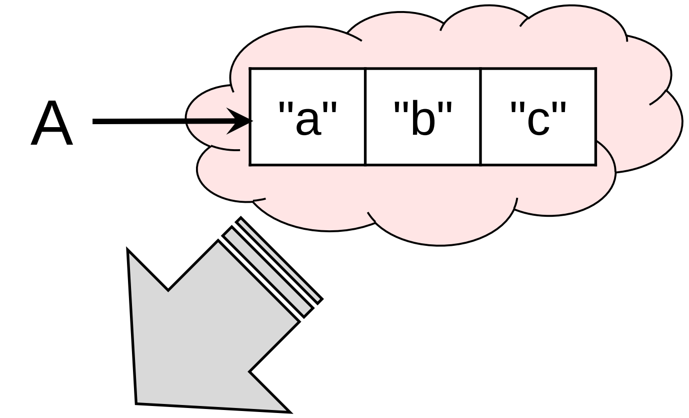
- Fix this by splitting **length** into two fields
  - **size** is the size of the unbounded array reported to the user
  - **limit** is the true length of the underlying array

```
// Implementation-side type
struct uba_header {                    // Concrete type
  int size;          // 0 <= size && size < limit
  int limit;         // 0 < limit
  string[] data;   // \length(data) == limit
};
```

These are representation invariants

It will be convenient to have size < limit rather than size <= limit

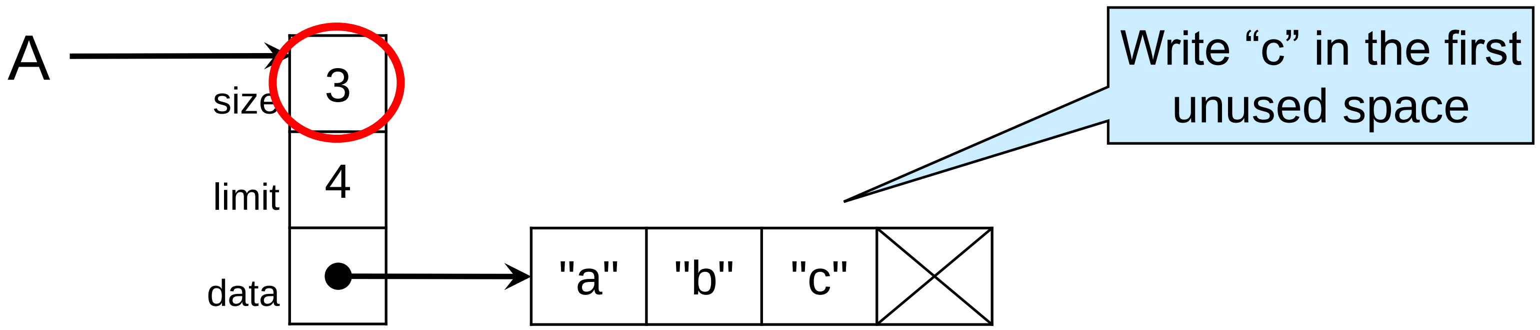




Towards an Implementation

uba_add(A, "c")

- Let's do it all over again: we first add "c"

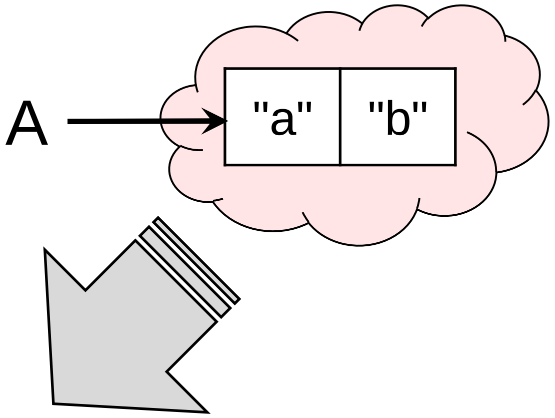


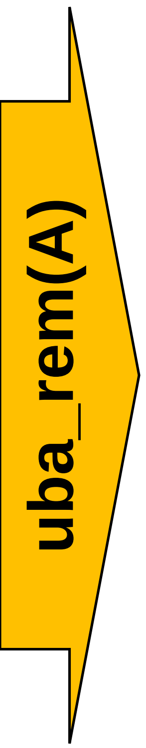
  - No need to copy old array elements
    - write new element in the first unused space
    - update size
  - O(1) for an n-element array
    - very cheap this time

- Next, let's remove the last element

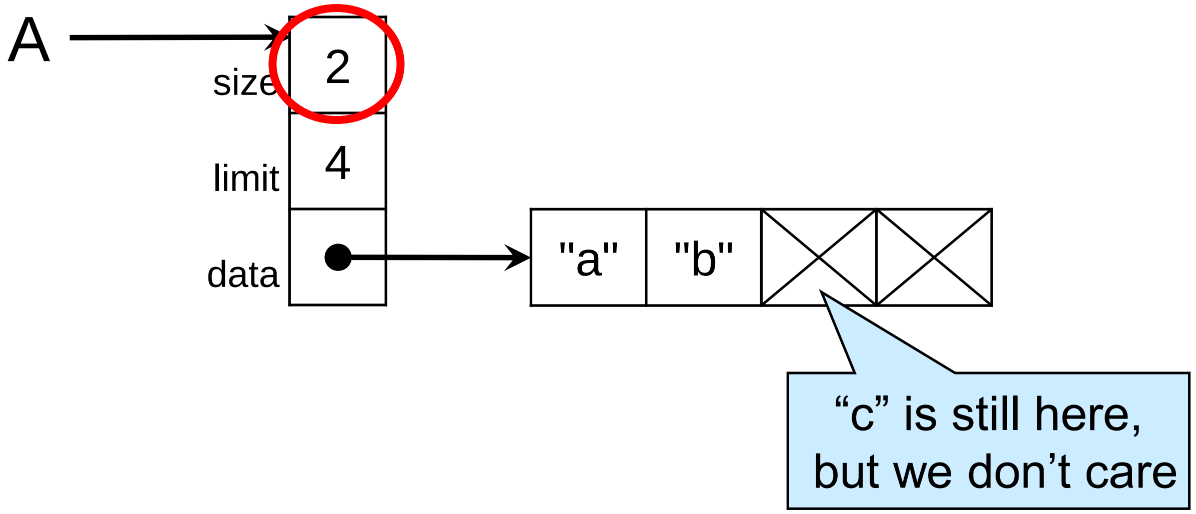
uba_rem(A)

Towards an Implementation



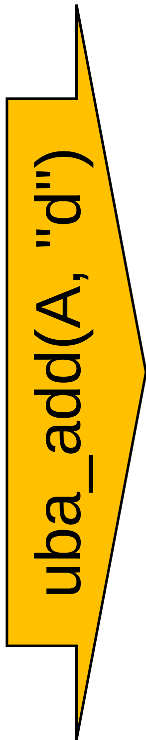


- Next, let’s remove the last element

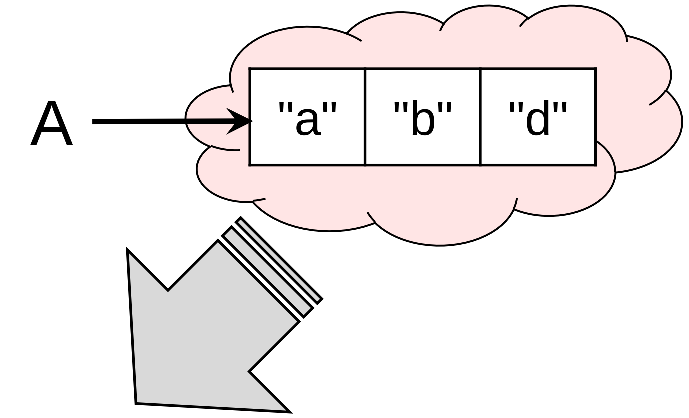


  - Simply decrement size and return element
  - O(1)

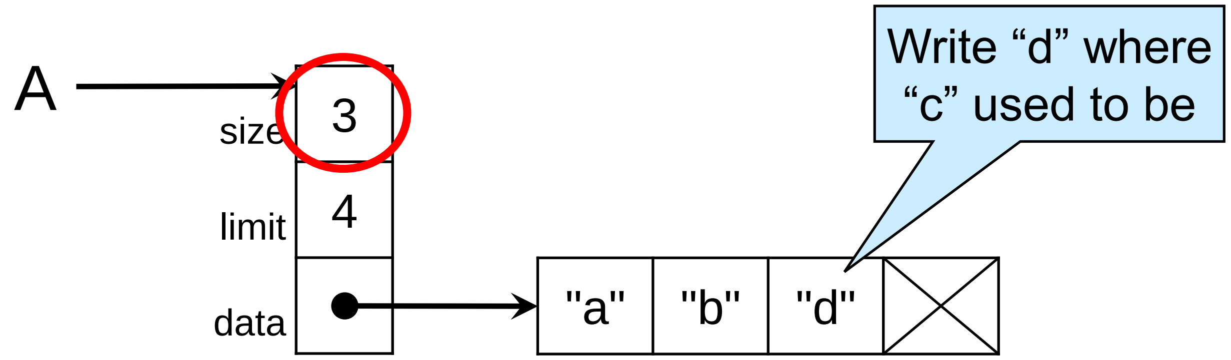
- Let’s continue by adding “d”



Towards an Implementation

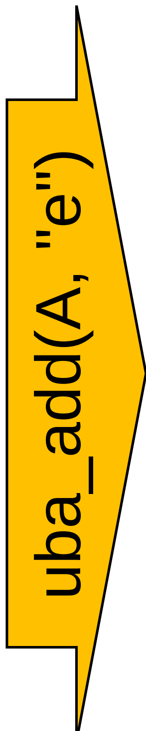


- Let's continue by adding "d"

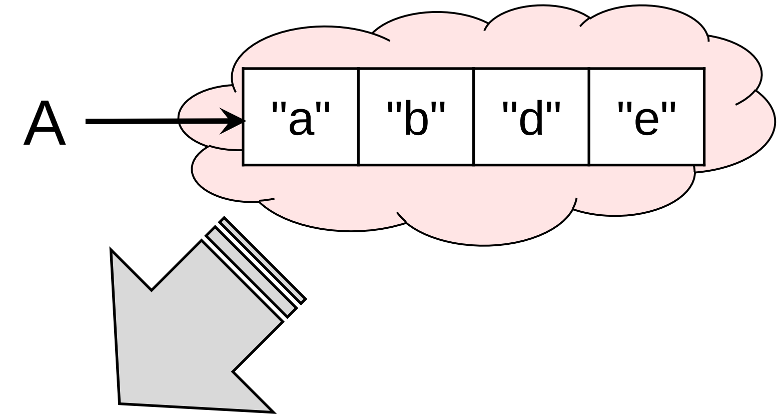


  - As before, just update size
  - O(1)

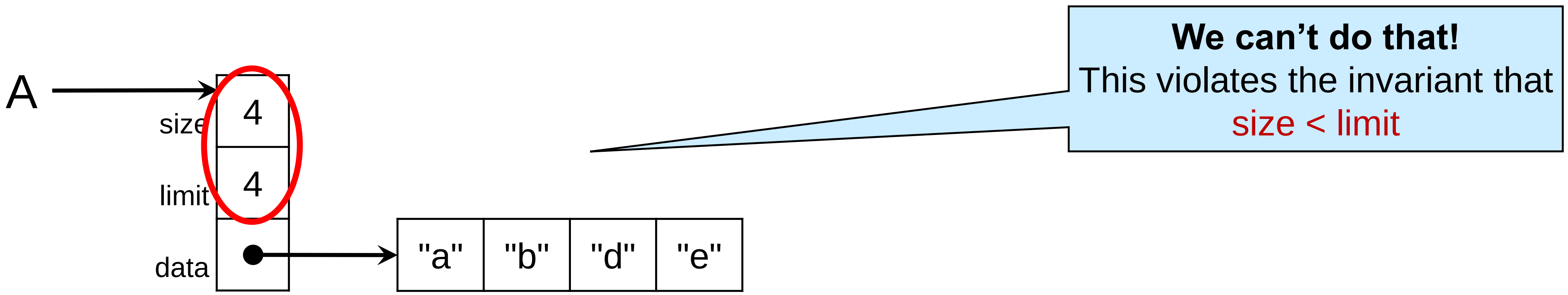
- This is where we got stuck earlier
  - Let's carry on and add "e"



Towards an Implementation



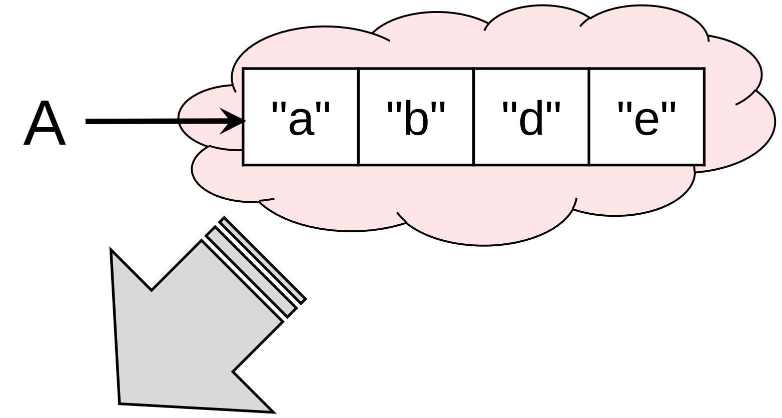
- Let’s carry on and add “e”



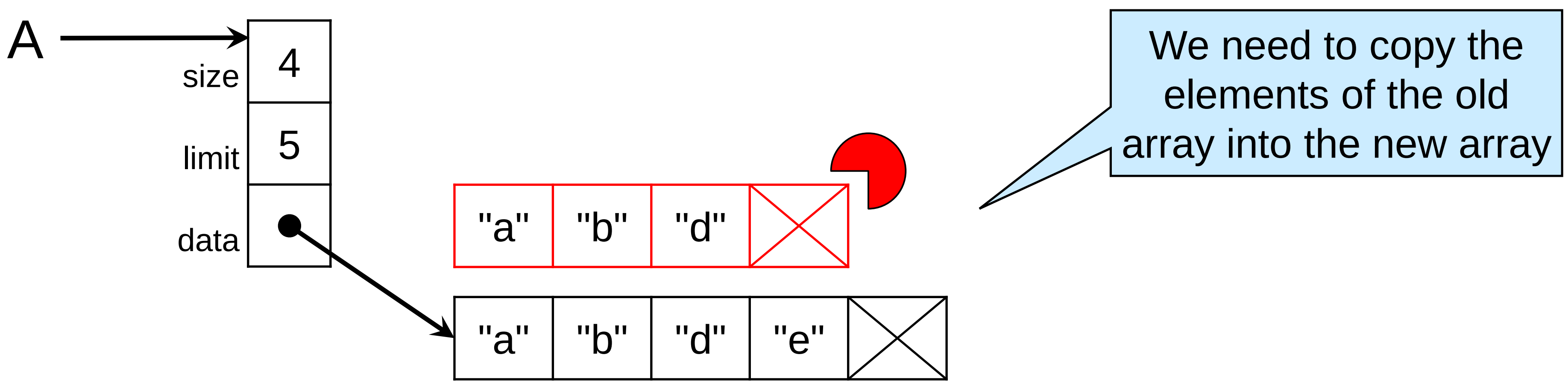
- We need to **resize** the array to accommodate “e”
  - while satisfying the representation invariants

- *How big should the new array be?*

Resizing the Array

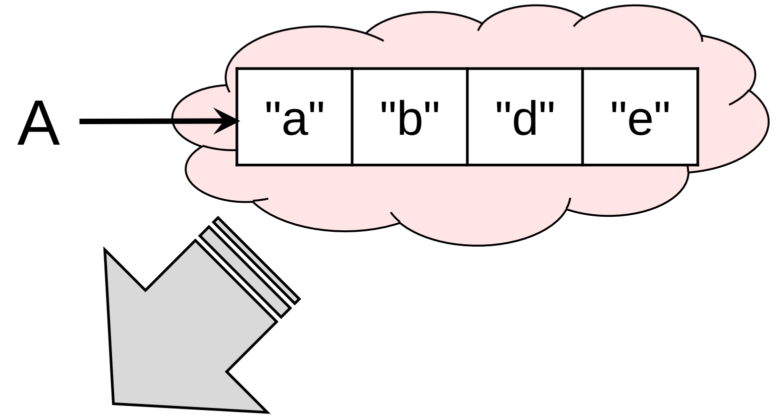


- How big should the new array be?
  - One longer: just enough to accommodate “e”



  - O(n) for an n-element array
- The next uba_add will also be O(n)
  - and the next after that, and the one after, and …

Resizing the Array



- *How big should the new array be?*
  - *one longer: just enough to accommodate "e"*
  - *O(n) for an n-element array, but the next add will also be O(n), …*

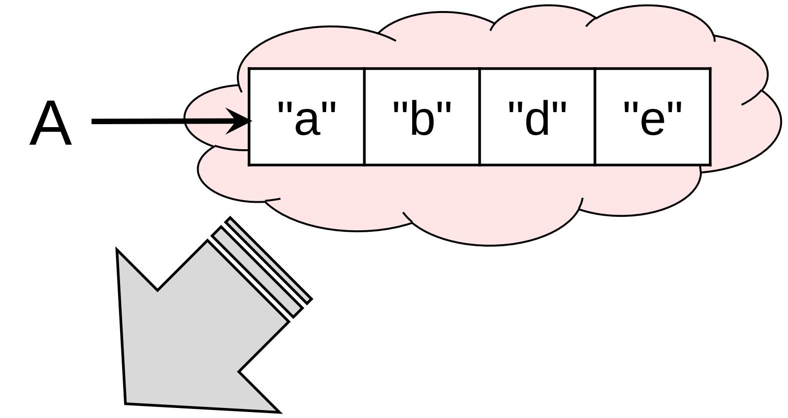
- A sequence of n uba_add starting from a limit-1 array costs

  $$1 + 2 + 3 + \ldots + (n-1) + n = n(n+1)/2$$

  That's $O(n^2)$
  - The amortized cost of each operation is O(n), like the worst-case

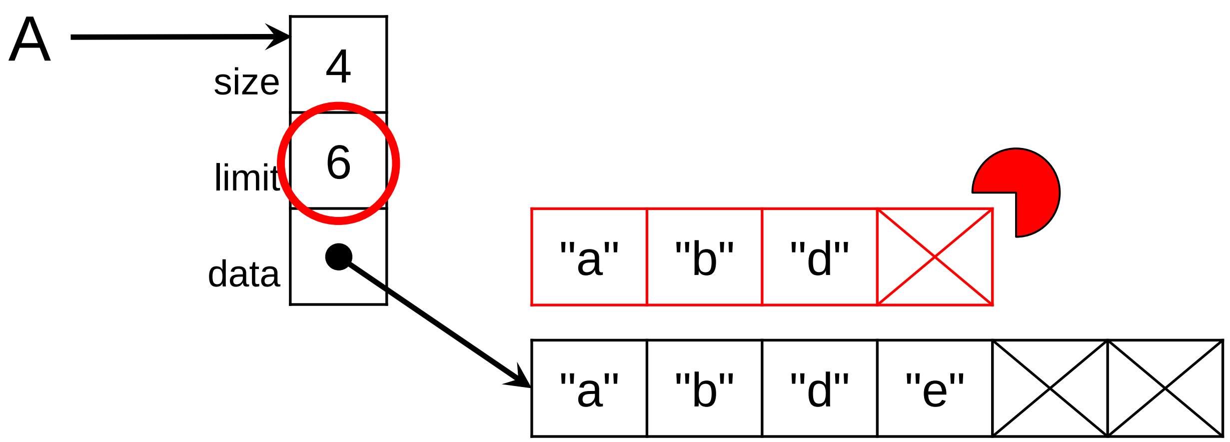
- *Can we do better?*
  - **Observation:** if there is space in the array, uba_add costs just O(1)
  - **Idea:** make the new array bigger than necessary

uba_add(A, "e")



Resizing the Array

- How big should the new array be?
  - **Two** longer: enough to accommodate “e” and a next element

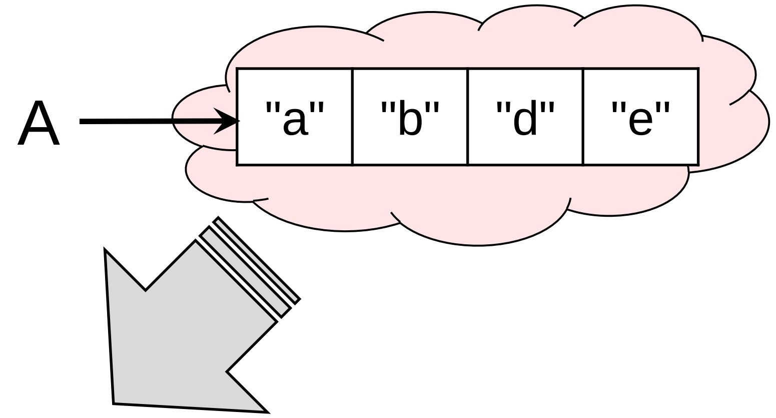


  - O(n) for an n-element array

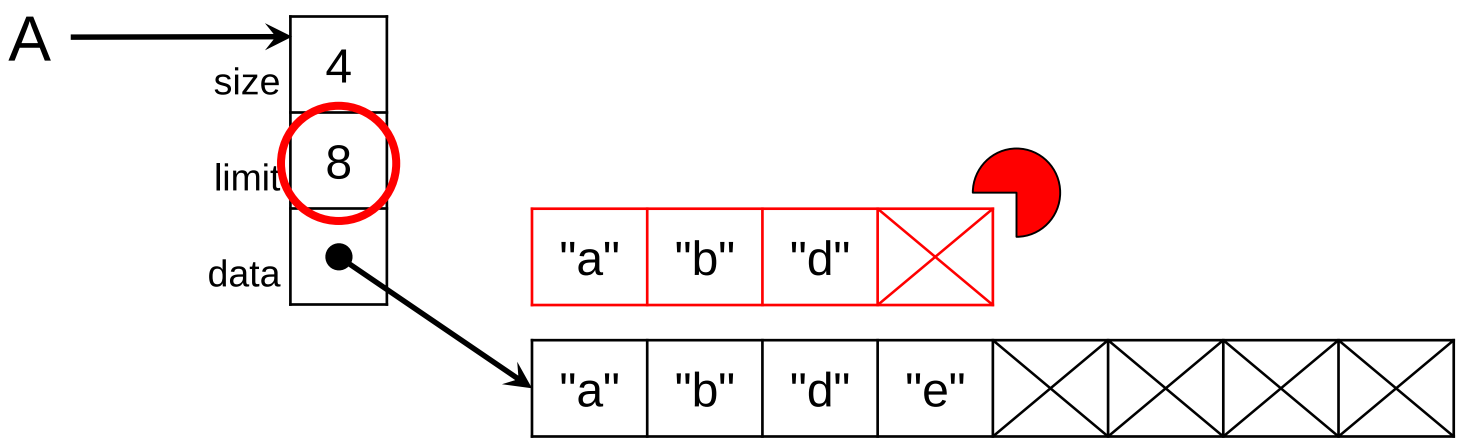
- The next add will be O(1) but the one after that is O(n) again
  - The cost of a sequence of n uba_add is still $O(n^2)$
  - The amortized cost stays at O(n)
- Same if we grow the array by *any* **fixed** amount c

1 + 1 + 3 + 1 + 5 + 1 + … + (2n+1) + 1
= 2 + 4 + 6 + … + (2n+2)
= 2(1 + 2 + 3 + … (n+1))
= (n+1)(n+2)

Resizing the Array



- How big should the new array be?
  - **Double** the length!



  - O(n) for an n-element array

- The next n uba_add will be O(1)
  - We get good amortized cost when
    - the expensive operations are further and further apart
    - most operations are cheap
  - Does doubling the size of the array give us **O(1) amortized** cost?

Analyzing Unbounded Arrays

Amortized Cost of uba_add

- **Conjecture:** doubling the size of the array on resize yields O(1) amortized complexity
- Let's follow our methodology

- Invent a notion of **token**
  - represents a unit of cost
- Determine how many tokens to charge
  - the candidate **amortized cost**
- Specify the **token invariant**
  - for any instance of the data structure, how many tokens need to be saved
- Prove that the operation **preserves** it
  - if the invariant holds before, it also holds after
    - saved tokens before + amortized cost – actual cost = saved tokens after

1. Draw a short sequence of operations
2. Write the cost of each operation
3. Flag the most expensive so far
4. For each operation, compute the total cost up to it
5. Divide the total cost of the most expensive operations by the operation number in the sequence
6. Round up — that's the candidate amortized cost

Amortized Cost of uba_add

- Invent a notion of **token**
  - represents a unit of cost

- For us, the unit of cost will be an **array write**
  - 1 array write costs 1 token
  - all other instructions are cost-free
    - we could also assign a cost to them

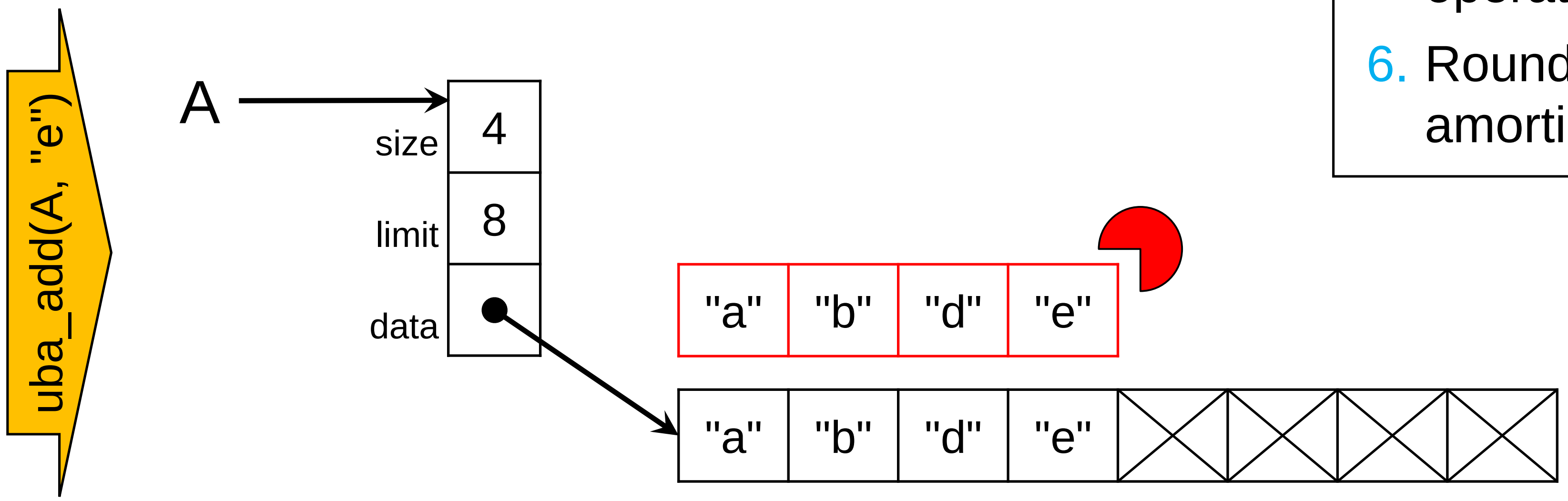
but let's keep things simple

Amortized Cost of uba_add

- Determine how many tokens to charge
  - that's the candidate **amortized cost**

1. Draw a short sequence of operations
2. Write the cost of each operation
3. Flag the most expensive so far
4. For each operation, compute the total cost up to it
5. Divide the total cost of the most expensive operations by the operation number in the sequence
6. Round up — that's the candidate amortized cost

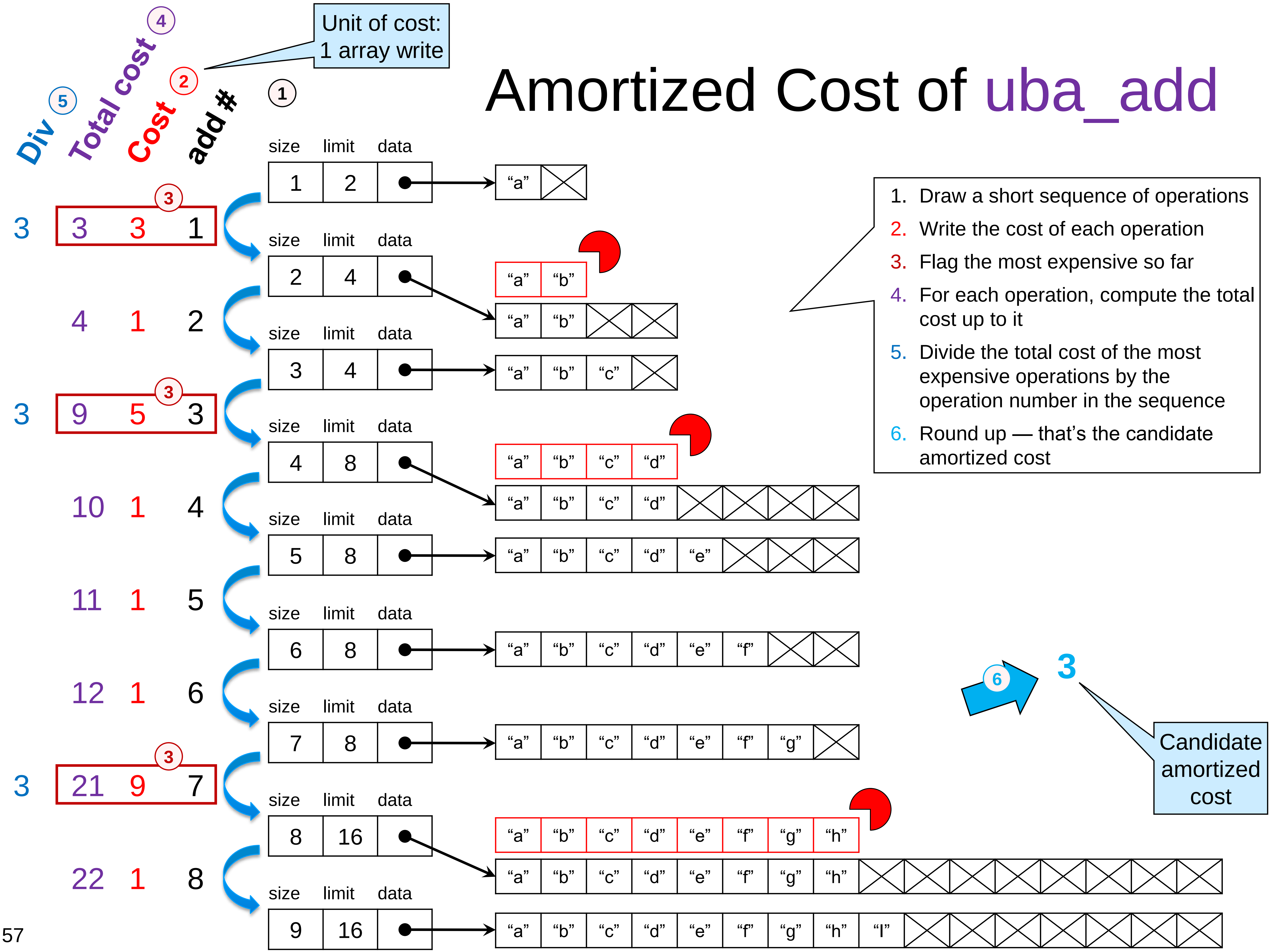
- When adding an element
  - we first write it in the old array, and then
  - if full, copy everything to the new array



  - This costs 5 tokens
    - write "e" in the old array
    - copy "a", "b", "d", "e" to the new array

a bit silly, but it makes the math simpler

Amortized Cost of uba_add



1. Draw a short sequence of operations
2. Write the cost of each operation
3. Flag the most expensive so far
4. For each operation, compute the total cost up to it
5. Divide the total cost of the most expensive operations by the operation number in the sequence
6. Round up — that's the candidate amortized cost

Amortized Cost of uba_add

It looks like we need to charge 3 tokens per uba_add

that's our candidate amortized cost

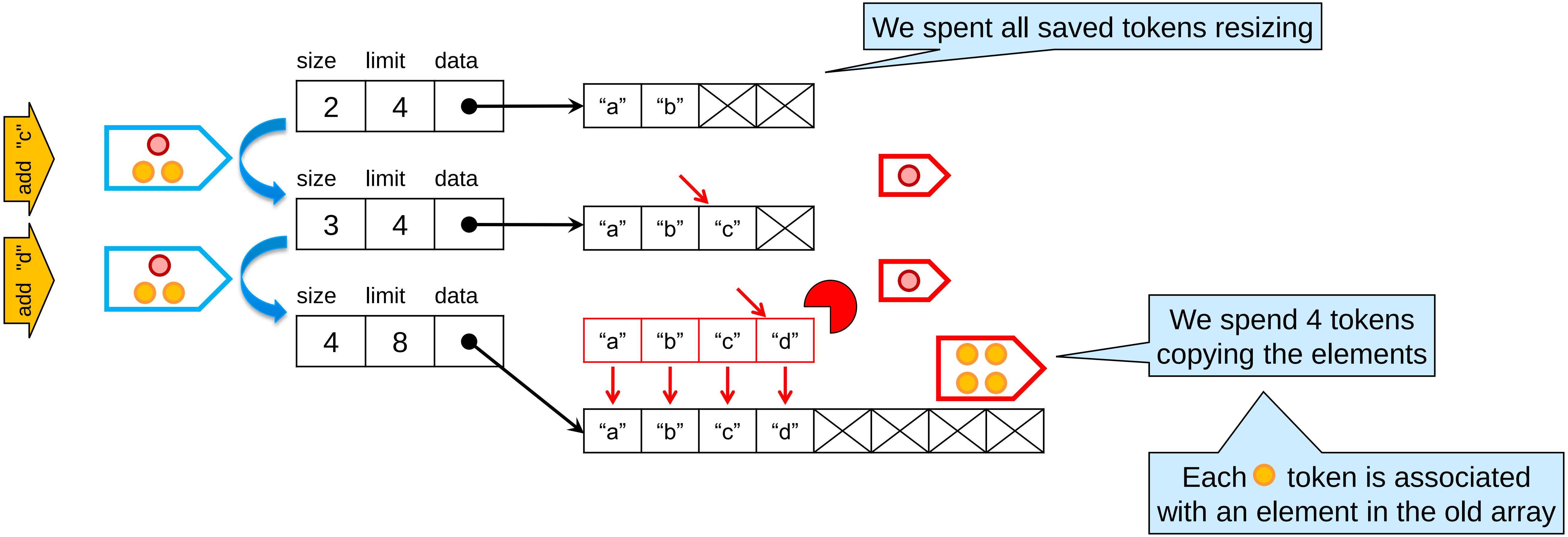
- Specify the **token invariant**
  - for any instance of the data structure, how many tokens need to be saved

- How are the 3 tokens charged for an uba_add used?
  - We always write the added element to the old array
    - 1 token used to write the new element
  - The remaining 2 tokens are saved
    - *where do they go?*

Amortized Cost of uba_add

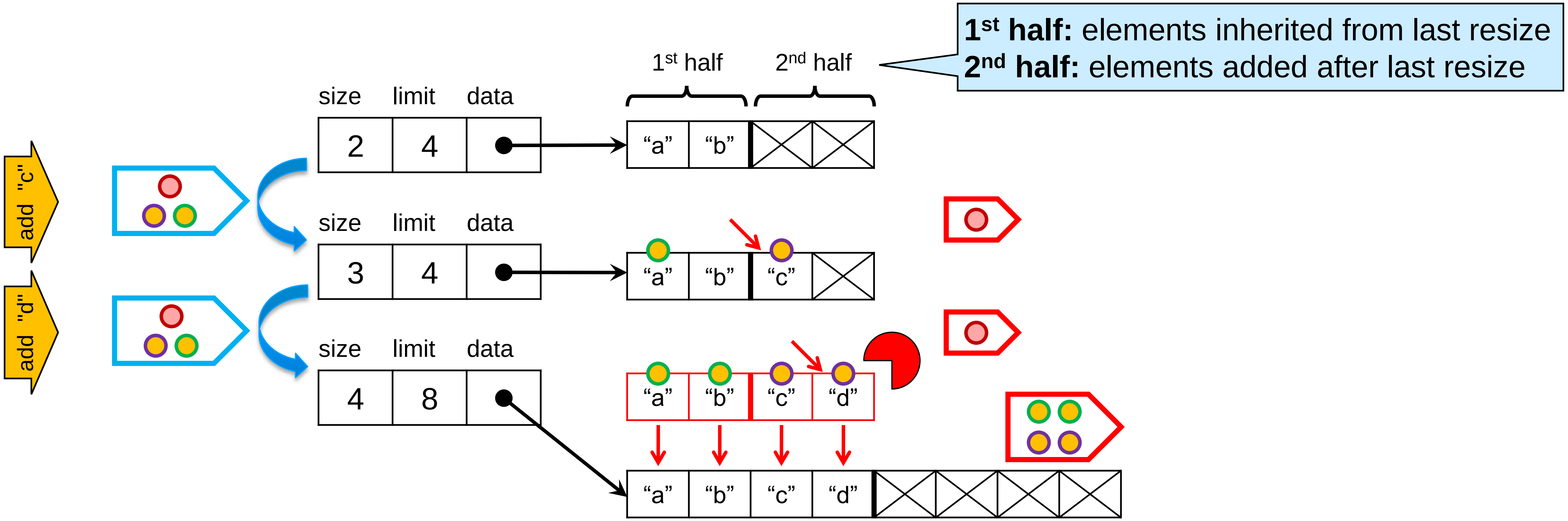
- How are the 3 tokens charged for an uba_add used?
  - 1 token used to write the new element
  - *Where do the remaining 2 tokens go?*

- Assume
  - we have just resized the array and have no tokens left



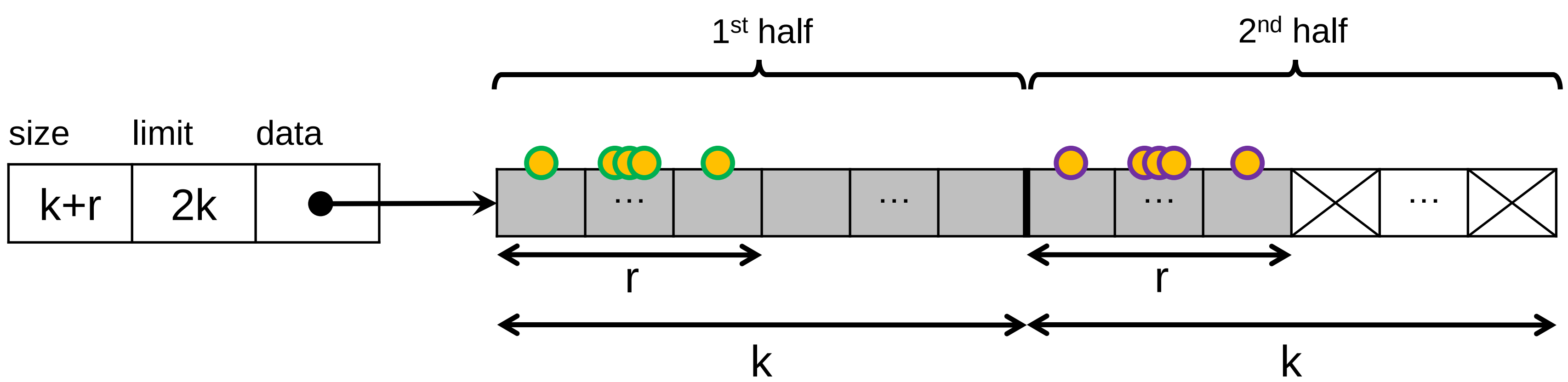
Amortized Cost of uba_add

- How are the 3 tokens charged for an uba_add used?
  - 1 token used to write the new element
  - Each of the remaining 2 tokens is associated with an element in the old array
    - 1 token to copy the element we just wrote
      - always in the 2nd half of the array
    - 1 token to copy the matching element in the first half of the array
      - element that was copied on the last resize



Amortized Cost of uba_add

- The **token invariant**
  - every element in the 2nd half of the array has a token
  - and the corresponding element in the 1st half of the array has a token



- *Alternative formulation:*
  - an array with limit 2k and size k+r holds 2r tokens (for 0 ≤ r < k)
    - # tokens == 2r

both assume a resize has happened previously

Amortized Cost of uba_add

- Prove that the operation **preserves** the token invariant
  - if the invariant holds before, it also holds after
    - saved tokens before + amortized cost – actual cost = saved tokens after

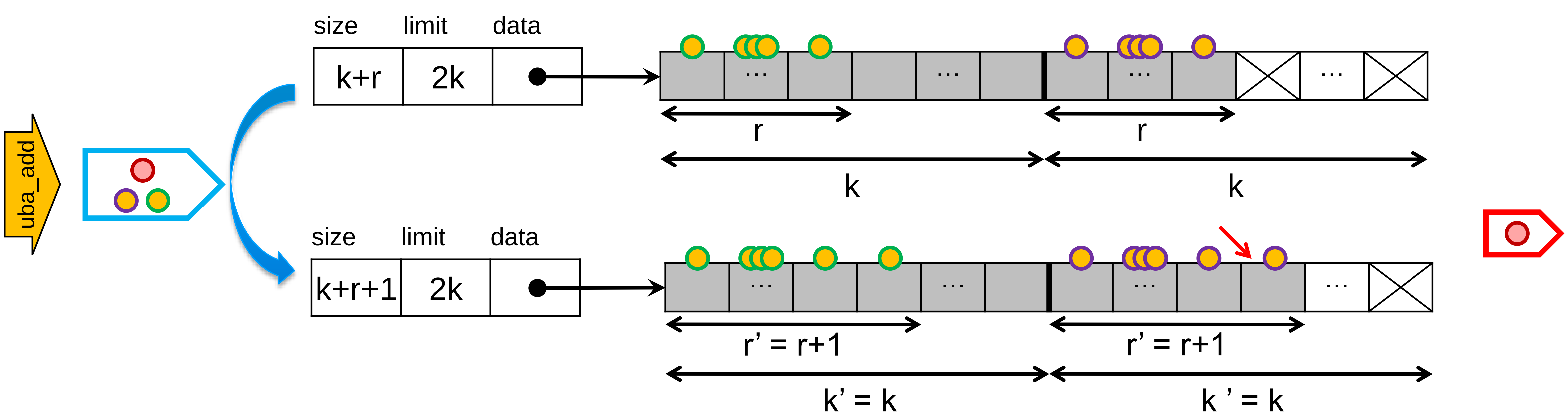
- We need to distinguish two cases
  1. Adding the element does not trigger a resize
  2. Adding the element does trigger a resize

  … and we will need to see what happens before the first resize

Amortized Cost of uba_add

saved tokens before + amortized cost – actual cost = saved tokens after

1. Adding the element **does not** trigger a resize
   - We receive 3 tokens
     - we spend 1 to write the new element
     - we put 1 on top of the new element
     - we put 1 on top of the matching element in the 1st half of the array

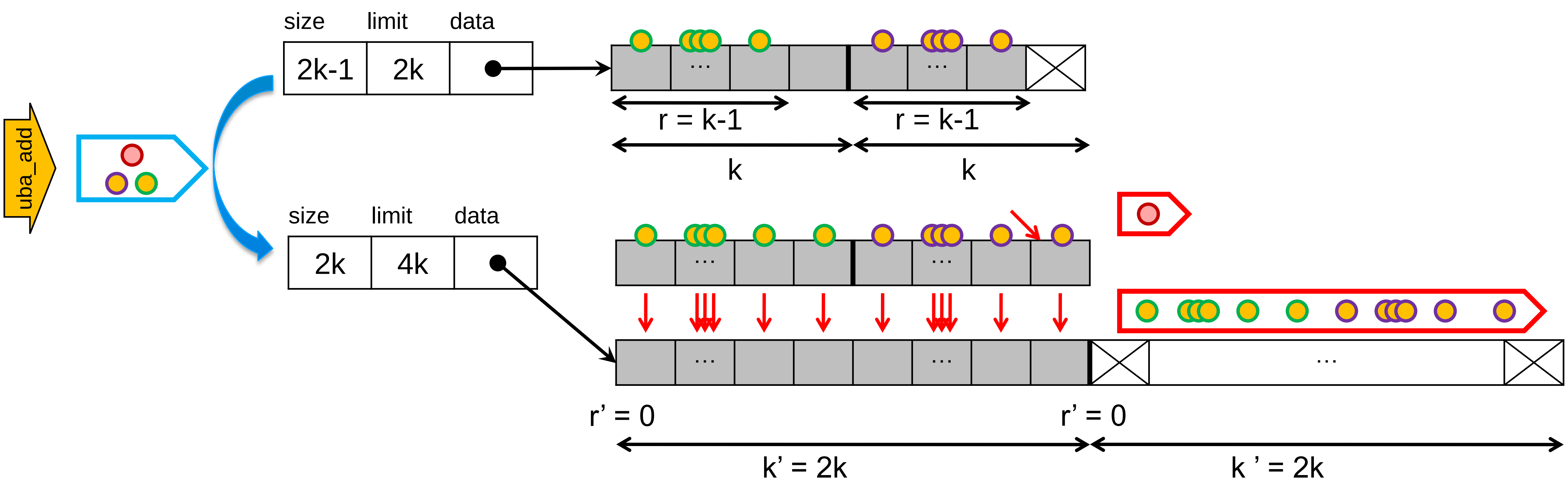


- Alternatively,
  - # tokens after = # tokens before + 3 – 1 = 2r + 2 = 2(r+1) = 2r'

Amortized Cost of uba_add

saved tokens before + amortized cost – actual cost = saved tokens after

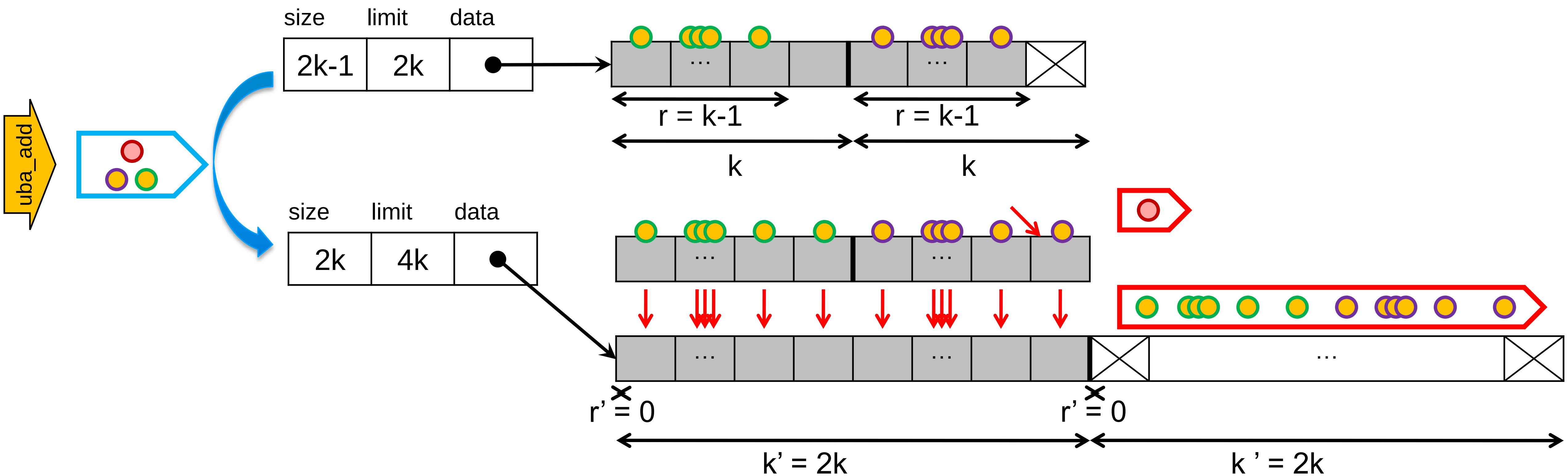
2. Adding the element **does** trigger a resize
   - We receive 3 tokens
     - we spend 1 to write the new element
     - we put 1 on top of the new element
     - we put 1 on top of the matching element in the 1st half of the array
   - We spend all tokens associated with array elements



Amortized Cost of uba_add

saved tokens before + amortized cost – actual cost = saved tokens after

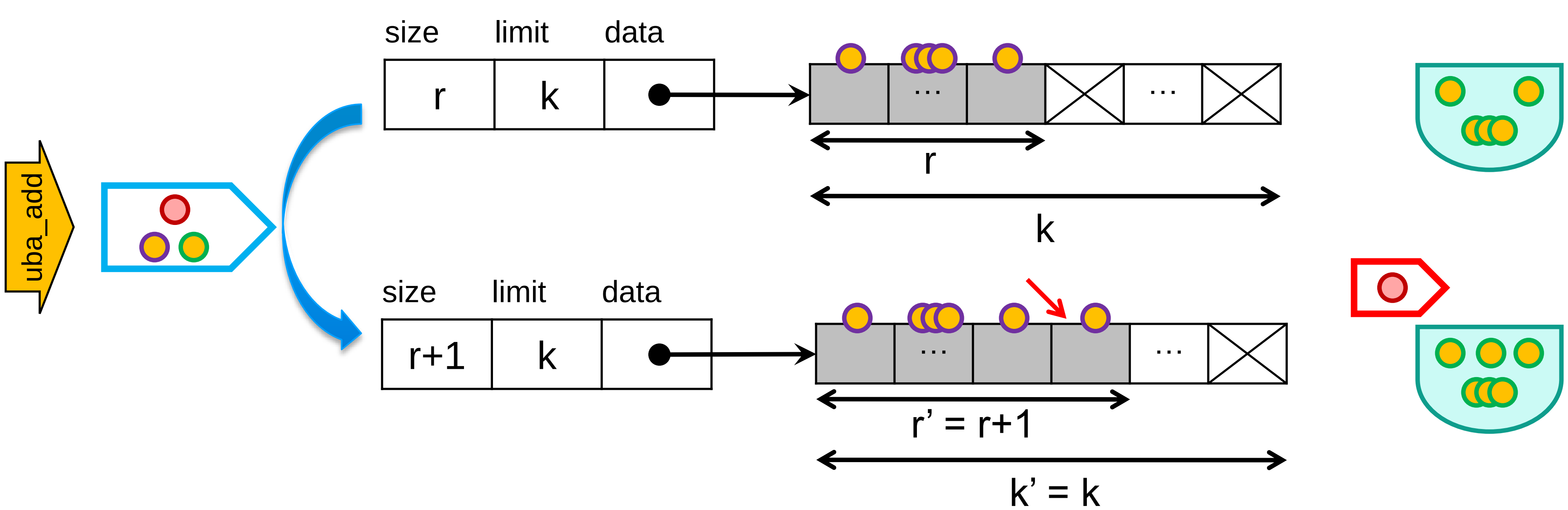
2. Adding the element **does** trigger a resize



- Alternatively,
  - # tokens after = # tokens before + 3 – 1 – (# tokens before + 2) = 2r + 2 – (2r+2) = 0 = 2r'

Amortized Cost of uba_add

- What happens before the first resize?
  - there is no 1st half of the array where to put matching tokens
  - put it in an extra savings account
    - that will not be used when resizing
    - update the token invariant to: # tokens ≥ 2r



  - It doesn't matter if we have extra savings
    - we are charging 3 tokens for uba_add
    - amortized cost is still O(1)

Amortized Cost of uba_add

- We followed our methodology

  - Invent a notion of **token**
    - represents a unit of cost
  - Determine how many tokens to charge
    - the candidate **amortized cost**
  - Specify the **token invariant**
    - for any instance of the data structure, how many tokens need to be saved
  - Prove that the operation **preserves** it
    - if the invariant holds before, it also holds after
      - saved tokens before + amortized cost – actual cost = saved tokens after

  1. Draw a short sequence of operations
  2. Write the cost of each operation
  3. Flag the most expensive so far
  4. For each operation, compute the total cost up to it
  5. Divide the total cost of the most expensive operations by the operation number in the sequence
  6. Round up — that's the candidate amortized cost

- and found that
  - we can charge 3 tokens for uba_add
  - the **amortized complexity** of uba_add is **O(1)**
  - although its **worst-case complexity** is **O(n)**

where n is the number of elements in the array

What about the Other Operations?

- uba_len and uba_get don't write to the array
  - they cost 0 tokens
- uba_set does exactly 1 write to the array
  - it costs 1 token

Worst-case complexity is O(1)

By charging this number of tokens, they trivially preserve the token invariant

- our analysis of uba_add remains valid even for sequences of operations that make use of them

- uba_new: doesn't write to the array
  - it costs 0 tokens
  - but we need to account for alloc_array

Worst-case complexity is O(size)

- uba_rem is … interesting
  - *left as exercise!*

It turns out that Its amortized complexity is also O(1)

Implementing Unbounded Arrays

Let’s implement them!

- Things we need to do
  - Define the concrete type for uba_t
  - Define its representation invariants
  - write code for every interface function
    - make sure it’s safe and correct

Left as an exercise

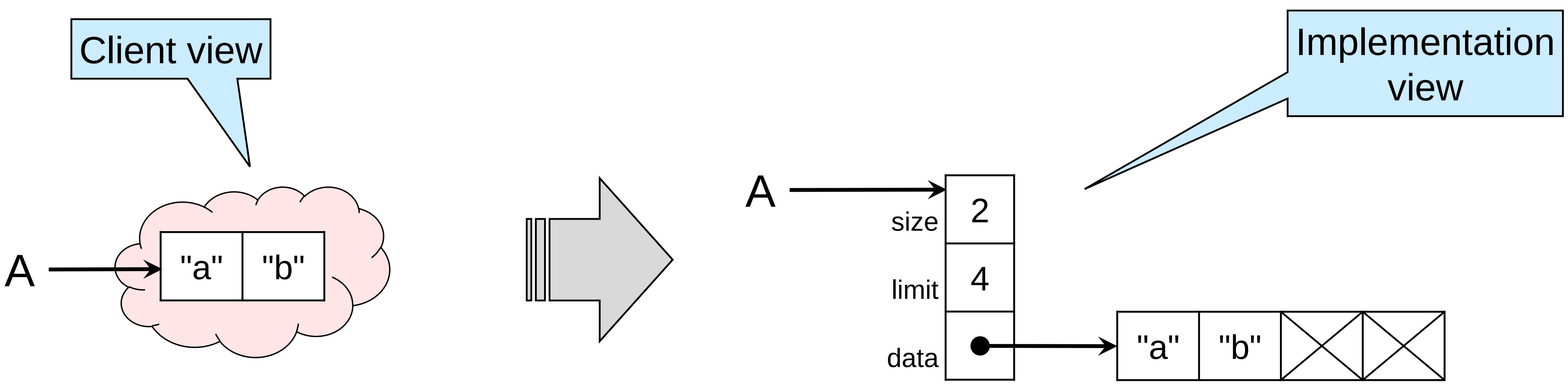
Unbounded Array Interface

```
// typedef ______* uba_t;

int uba_len(uba_t A)                          // O(1)
  /*@requires A != NULL;                      @*/
  /*@ensures \result >= 0;                    @*/ ;

uba_t  uba_new(int size)                      // O(size)
  /*@requires 0 <= size ;                     @*/
  /*@ensures \result != NULL;                 @*/
  /*@ensures uba_len(\result) == size;        @*/ ;

string uba_get(uba_t A, int i)                // O(1)
  /*@requires A != NULL;                      @*/
  /*@requires 0 <= i && i < uba_len(A);       @*/ ;

void uba_set(uba_t A, int i, string x)        // O(1)
  /*@requires A != NULL;                      @*/
  /*@requires 0 <= i && i < uba_len(A);       @*/ ;

void uba_add(uba_t A, string x)               // O(1) amt
  /*@requires A != NULL;                      @*/ ;

string uba_rem(uba_t A)                       // O(1) amt
  /*@requires A != NULL;                      @*/
  /*@requires  0 < uba_len(A);                @*/ ;
```

Concrete Type

- We did this earlier!

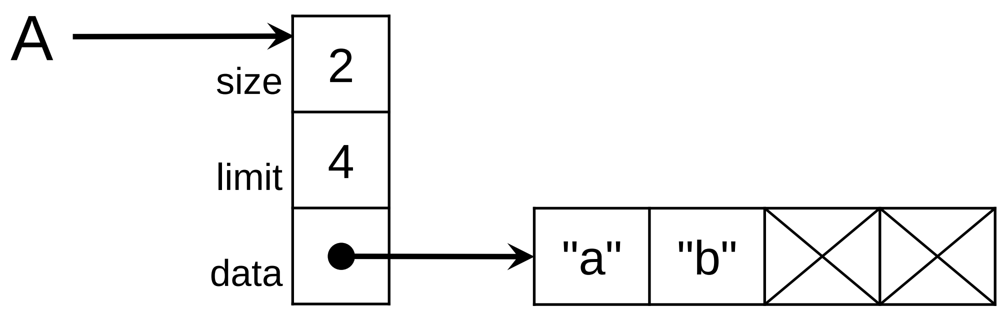
```
// Implementation-side type
struct uba_header {                   // Concrete type
  int size;          // 0 <= size && size < limit
  int limit;         // 0 < limit
  string[] data;  // \length(data) == limit
};
typedef struct uba_header uba;          // Internal name

// … rest of implementation …

// Client-side type (abstract)
typedef uba* uba_t;
```



Representation Invariants

```
struct uba_header {
  int size;        // 0 <= size && size < limit
  int limit;       // 0 < limit
  string[] data;   // \length(data) == limit
};
typedef struct uba_header uba;
```

- Internally, unbounded arrays are values of type uba*
  - non-NULL
  - satisfies the requirements in the type



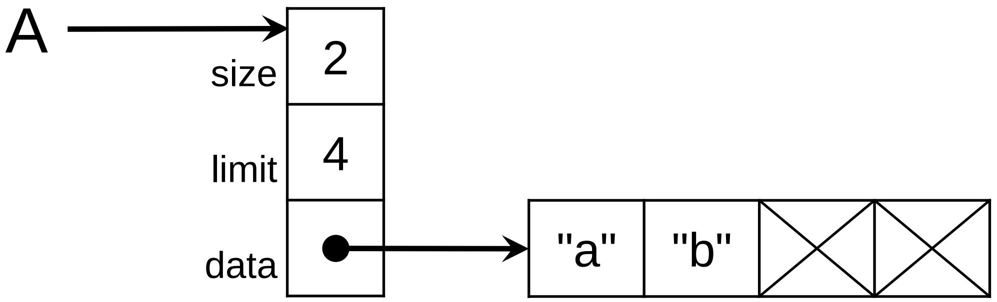

```
bool is_array_expected_length(string[] A, int length) {
  //@assert \length(A) == length;
  return true;
}

bool is_uba(uba* A) {
  return A != NULL
      && 0 <= A->size
      && A->size < A->limit
      && is_array_expected_length(A->data, A->limit);
}
```

Our trick to check that the length is Ok

Basic Array Operations

```
struct uba_header {
  int size;
  int limit;
  string[] data;
};
typedef struct uba_header uba;
```

- The code is as expected




```
int uba_len(uba* A)
//@requires is_uba(A);
//@ensures 0 <= \result && \result < \length(A->data);
{
  return A->size;
}
```

```
void uba_set(uba* A, int i, string x)
//@requires is_uba(A);
//@requires 0 <= i && i < uba_len(A);
//@ensures is_uba(A);
{
  A->data[i] = x;
}
```

```
string uba_get(uba* A, int i)
//@requires is_uba(A);
//@requires 0 <= i && i < uba_len(A);
{
  return A->data[i];
}
```

```
uba* uba_new(int size)
//@requires 0 <= size;
//@ensures is_uba(\result);
//@ensures uba_len(\result) == size;
{
  uba* A = alloc(uba);
  int limit = size == 0 ? 1 : size*2;
  A->data = alloc_array(string, limit);
  A->size = size;
  A->limit = limit;

  return A;
}
```

- *if* size == 0, *then* limit = 1
- *otherwise* limit = size*2

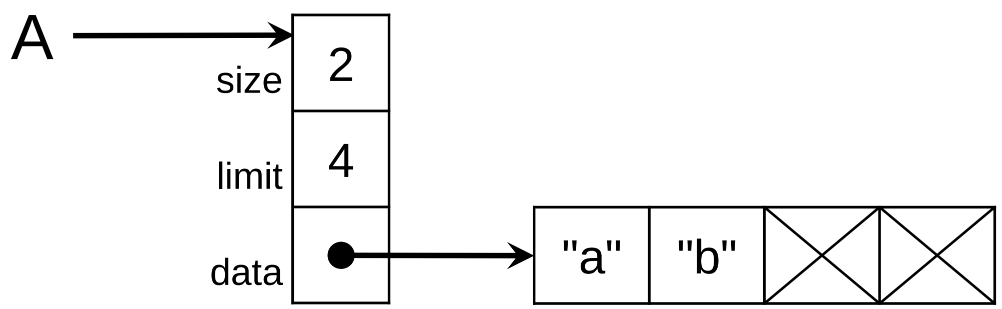
This ensures that
size < limit
(and leaves room to grow)

We are not considering overflow

```
struct uba_header {
  int size;
  int limit;
  string[] data;
};
typedef struct uba_header uba;
```

Adding an Element

- We write the new element,
- increment size,
- if array is full, we resize it
  - but only if there can't be overflow




```
void uba_add(uba* A, string x)
//@requires is_uba(A);
//@ensures is_uba(A);
{
  A->data[A->size] = x;
  (A->size)++;

  if (A->size < A->limit) return;
  assert(A->limit <= int_max() / 2);
  uba_resize(A, A->limit * 2);
}
```

Fail if new limit would overflow

Resize A with the new limit double the old limit

Resizing the Array

```
struct uba_header {
  int size;
  int limit;
  string[] data;
};
typedef struct uba_header uba;
```

- Create an array with the new limit,
- copy the elements over
- update the fields of the header

```
void uba_resize(uba* A, int new_limit)
//@requires A != NULL;
//@requires 0 <= A->size && A->size < new_limit;
//@requires \length(A->data) == A->limit;
//@ensures is_uba(A);
{
  string[] B = alloc_array(string, new_limit);

  for (int i = 0; i < A->size; i++)
    //@loop_invariant 0 <= i && i <= A->size;
    {
      B[i] = A->data[i];
    }

  A->limit = new_limit;
  A->data = B;
}
```

//@requires is_uba(A);
would be incorrect:
we may have size==limit

uba_resize may be passed an **invalid UBA**:
one that violates the representation invariant

Part of its job is to restore
the representation invariant

Unbounded Arrays in the Wild

Python "Lists"

- The Python programming language does not have arrays
- It has "lists" that can be indexed, extended and shrunk
  - nothing to do with linked list

```
data = ['A', 'B', 'C']
data.append('D')
data[2]

data = []
for i in range(100000):
  data.append('A')
data[99888]
```

- Create a 3-element list with 'A', 'B', and 'C'
- Extend it with 'D'
- Get the element at index 2 (that's 'C')
- Set data to the empty list
- Extend it with a bunch of 'A'
- Access one of them

- Python lists work just like unbounded arrays
  - **append** is what we called uba_add

How are Python Lists Implemented?

- Source code available at
  https://github.com/python/cpython/blob/master/Objects/listobject.c
  - It is written in C

- Let's look at the code for **`append`**

```c
int
PyList_Append(PyObject *op, PyObject *newitem)
{
    if (PyList_Check(op) && (newitem != NULL))
        return app1((PyListObject *)op, newitem);
    PyErr_BadInternalCall();
    return -1;
}
```

If all Ok, call app1

Otherwise, raise an error

How are Python Lists Implemented?

- Let’s look at the code of app1

```
static int
app1(PyListObject *self, PyObject *v)
{
    Py_ssize_t n = PyList_GET_SIZE(self);

    assert (v != NULL);
    if (n == PY_SSIZE_T_MAX) {
        PyErr_SetString(PyExc_OverflowError,
            "cannot add more objects to list");
        return -1;
    }

    if (list_resize(self, n+1) < 0)
        return -1;

    Py_INCREF(v);
    PyList_SET_ITEM(self, n, v);
    return 0;
}
```

Calls list_resize to resize array if needed

This code writes the new element after any resizing

How are Python Lists Implemented?

- Let’s look at the code of list_resize

```c
static int
list_resize(PyListObject *self, Py_ssize_t newsize)
{
 ...

    /* This over-allocates proportional to the list size, making room
     * for additional growth.  The over-allocation is mild, but is
     * enough to give linear-time amortized behavior over a long
     * sequence of appends() in the presence of a poorly-performing
     * system realloc().
     * The growth pattern is:  0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...
     * Note: new_allocated won't overflow because the largest possible value
     *       is PY_SSIZE_T_MAX * (9 / 8) + 6 which always fits in a size_t.
     */
    new_allocated = (size_t)newsize + (newsize >> 3) + (newsize < 9 ? 3 : 6);
    if (new_allocated > (size_t)PY_SSIZE_T_MAX / sizeof(PyObject *)) {
        PyErr_NoMemory();
        return -1;
    }
```

unimportant code

== newsize / 8

new_allocated = 1.125 * newsize + *change*

doesn’t quite double the size, but grows as a multiple of newsize

Exercise: check that the amortized cost is still O(1)

Wrap Up

What have we done?

- We introduced **amortized complexity**
  - average cost over a sequence of operations
- We learned how to determine the amortized complexity
  - **amortized analysis** using the accounting method
- We used it to analyze **unbounded arrays**

Operation	Worst-case complexity	Amortized complexity
uba_len	O(1)	*(same)*
uba_new	O(n)	
uba_get	O(1)	
uba_set	O(1)	
uba_add	O(n)	O(1)
uba_rem	O(n)	O(1)

Exercise

- We implemented unbounded arrays