CSE 250: ArrayList, Amortized Complexity

Lecture 10

Sept 20, 2023

Reminders

- PA1 Implementation due Sun, Sept 24 at 11:59 PM
  - Implement a Sorted Linked List
- WA2 due Sun, Oct 1 at 11:59 PM
  - Will be released over the next weekend
- Midterm 1 on Oct 2 in-class
  - Covers: Asymptotics, Sequences/Lists, Arrays, Linked Lists, Recursion
  - Bounds: Tight Upper/Lower, General vs Expected vs Amortized

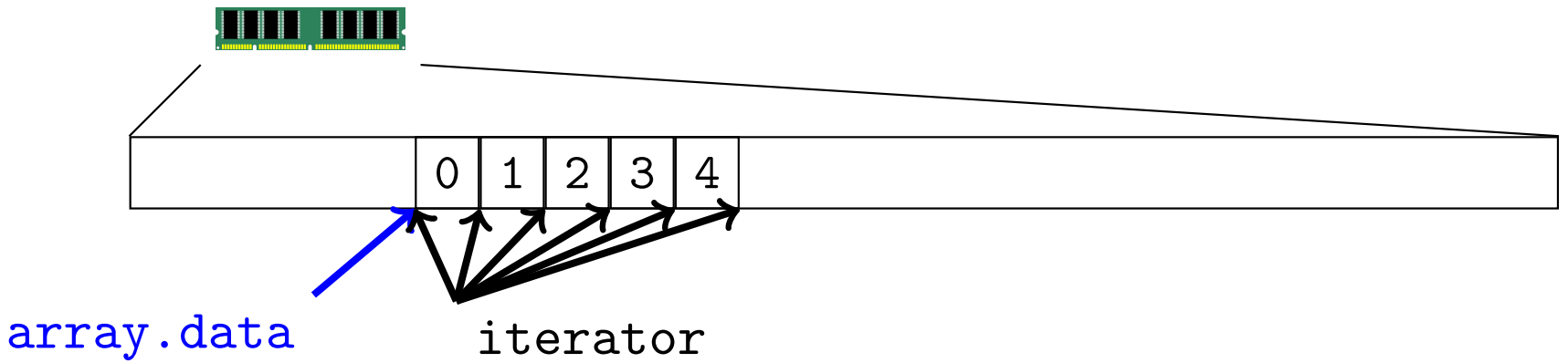
The Sequence ADT

```
1   public interface List<E>
2   {
3     // Sequence
4     public E get(idx: Int);
5     public void set(idx: Int, E value);
6     public int size();
7     public ListIterator<E> iterator();
8
9     // List
10    public void add(E value);
11    public void add(int idx, E value);
12    public void remove(int idx);
13  }
```

ListIterator (simplified)

```
1    public interface ListIterator<E>
2    {
3      public boolean hasNext();
4      public E next();
5      public void add(E value);
6      public void set(E value);
7      public void remove();
8    }
```

Array Iterators




```
list.iterator() → iterator
iterator.next() → 0
iterator.next() → 1
iterator.next() → 2
iterator.next() → 3
iterator.next() → 4
```

ArrayListIterator

```
1  public class ArrayListIterator<E>
2    implements ListIterator<E>
3  {
4    /* ??? */
5  }
```

ArrayListIterator

```
1   public class ArrayListIterator<E>
2     implements ListIterator<E>
3   {
4     ArrayList<E> array;
5     int nextIndex = 0;
6
7     /* ... */
8   }
```

ArrayListIterator

```
1   public class ArrayListIterator<E>
2     implements ListIterator<E>
3   {
4     ArrayList<E> array;
5     int nextIndex = 0;
6
7     public boolean hasNext()
8     {
9       /* ??? */
10    }
11  }
```

ArrayListIterator

```
1   public class ArrayListIterator<E>
2     implements ListIterator<E>
3   {
4     ArrayList<E> array;
5     int nextIndex = 0;
6
7     public boolean hasNext()
8     {
9       return nextIndex >= array.size() ← O(1)
10    }
11  }
```

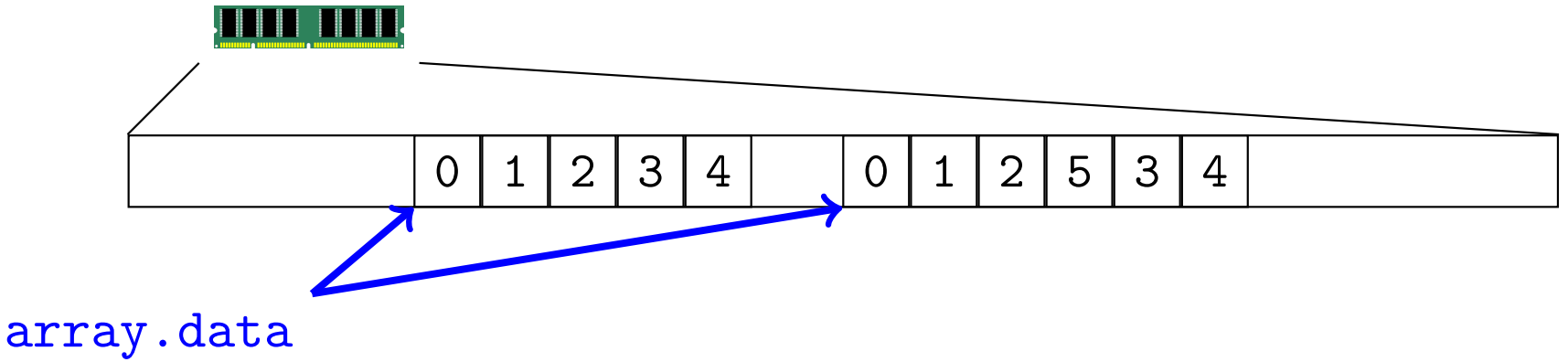
ArrayListIterator

```
1   public class ArrayListIterator<E>
2     implements ListIterator<E>
3   {
4     ArrayList<E> array;
5     int nextIndex = 0;
6
7     /* ... */
8
9     public E next()
10    {
11      nextIndex += 1;                    ← O(1)
12      return array.get(nextIndex-1)      ← O(1)
13    }
14  }
```

ArrayListIterator

```
1   public class ArrayListIterator<E>
2     implements ListIterator<E>
3   {
4     ArrayList<E> array;
5     int nextIndex = 0;
6
7     /* ... */
8
9     public void add(E element)
10    {
11      /* ??? */
12    }
13  }
```

Array add(idx, value)




```
array.add(idx= 2, value= 5)
```
$\leftarrow \theta(N)$

Today's Focus

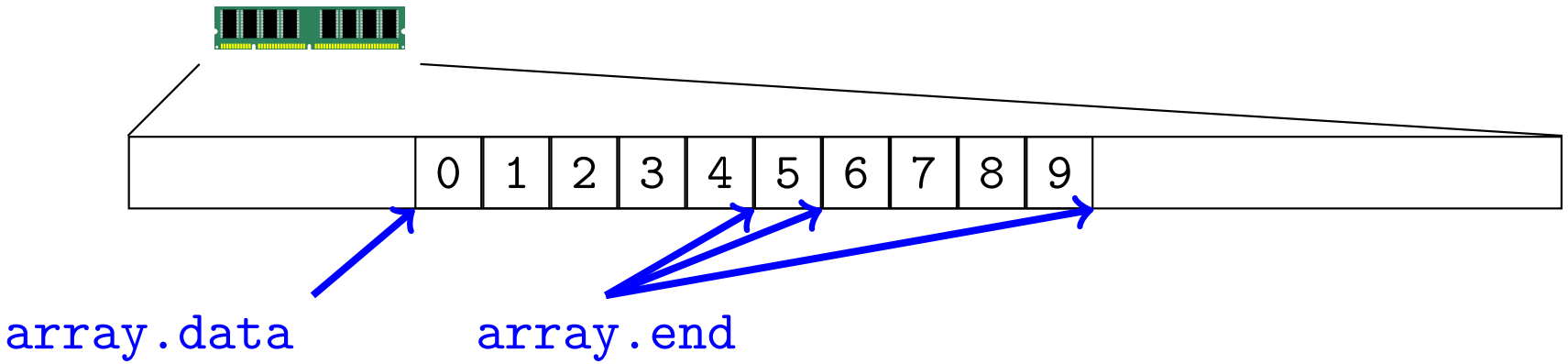
Let's start with at `add(value)` (append to end).

Can we make it faster?

Idea 1

Idea: Allocate more memory than we need.

Idea 1




```
array.add(value= 5)
array.add(value= 6)
array.add(value= 7)
array.add(value= 8)
array.add(value= 9)
array.add(value= 10)
```

$\leftarrow \theta(1)$
$\leftarrow \theta(1)$
$\leftarrow \theta(1)$
$\leftarrow \theta(1)$
$\leftarrow \theta(1)$
$\leftarrow \theta(N)$

ArrayList (extra space edition)

- `array.size()`: How many elements are currently stored.
  (actual java method)
- `array.capacity()`: How many elements can be stored before we need to make a bigger array.
  (not an actual java method)

Array `add(e)`

$$T_{\texttt{add(e)}} = \begin{cases} \theta(1) & \textbf{if } \texttt{array.size()} < \texttt{array.capacity()} \\ \theta(N) & \textbf{otherwise} \end{cases}$$

So...

$T_{\texttt{add(e)}} = O(N)$

But...

Array add(e)

```
1     public ArrayList<Integer> makeRange(int N)
2     {
3       ArrayList<Integer> list = new ArrayList<Integer>();
4       for(int i = 0; i < N; i++)
5       {
6         list.add(i);
7       }
8       return list
9     }
```

Array add(e)

```
1    public ArrayList<Integer> makeRange(int N)
2    {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4      for(int i = 0; i < N; i++)
5      {
6        O(N)
7      }
8      return list
9    }
```

Array add(e)

```
1    public ArrayList<Integer> makeRange(int N)
2    {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4      ∑_{i=0}^{N} O(N)
5      return list
6    }
```

Array add(e)

```
1    public ArrayList<Integer> makeRange(int N)
2    {
3       O(1)
4       O(N^2)
5       O(1)
6    }
```

Maybe we're overcounting: Most calls to `add(i)` are $\theta(1)$

Spoiler: We can get a tighter bound for `makeRange`!

Allocation Policy

Design Question:

1. How much extra space do we allocate at first?
2. How much extra space do we add when we run out?

Let's say 2 extra spaces.

Every 2nd call to `add(value)` will be $\theta(N)$

Array add(e)

```
1     public ArrayList<Integer> makeRange(int N)
2     {
3       ArrayList<Integer> list = new ArrayList<Integer>();
4       for(int i = 0; i < N; i++)
5       {
6         list.add(i);
7       }
8       return list
9     }
```

Array add(e)

```
1    public ArrayList<Integer> makeRange(int N)
2    {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4      for(int i = 0; i < N; i++)
5      {
6        T_add(e)(i)
7      }
8      return list
9    }
```

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 2 = 0 \\ O(N) i \cdot \theta(1) & \textbf{otherwise} \end{cases}$$

Array add(e)

```
1    public ArrayList<Integer> makeRange(int N)
2    {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4      ∑_{i=0}^{N-1} T_{add(e)}(i)
5      return list
6    }
```

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 2 = 0 \\ i \cdot \theta(1) & \textbf{otherwise} \end{cases}$$

Array add(e)

```
1    public ArrayList<Integer> makeRange(int N)
2    {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4      T_add(e)(0) + T_add(e)(1) + ... + T_add(e)(N − 1)
5      return list
6    }
```

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 2 = 0 \\ i \cdot \theta(1) & \textbf{otherwise} \end{cases}$$

Array add(e)

```
1    public ArrayList<Integer> makeRange(int N)
2    {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4      θ(1) + 2·θ(1) + θ(1) + 4·θ(1) + ... + θ(1) + N·θ(1)
                                  N terms
5      return list
6    }
```

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 2 = 0 \\ i \cdot \theta(1) & \textbf{otherwise} \end{cases}$$

Array add(e)

```
1    public ArrayList<Integer> makeRange(int N)
2    {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4      θ(1) + ... + θ(1) + 2·θ(1) + 4·θ(1) + 6·θ(1) + ... + N·θ(1)
5      return list
6    }
```

($\frac{N}{2}$ terms) ($\frac{N}{2}$ terms)

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 2 = 0 \\ i \cdot \theta(1) & \textbf{otherwise} \end{cases}$$

Array add(e)

```
1    public ArrayList<Integer> makeRange(int N)
2    {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4      N/2 θ(1) + 2·θ(1) + 4·θ(1) + 6·θ(1) + ... + N·θ(1)
                          (N/2 terms)
5      return list
6    }
```

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 2 = 0 \\ i \cdot \theta(1) & \textbf{otherwise} \end{cases}$$

Array add(e)

```
1    public ArrayList<Integer> makeRange(int N)
2    {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4      θ(N) + 2·θ(1) + 4·θ(1) + 6·θ(1) + ... + N·θ(1)
                    (N/2 terms)
5      return list
6    }
```

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 2 = 0 \\ i \cdot \theta(1) & \textbf{otherwise} \end{cases}$$

Array add(e)

```
1     public ArrayList<Integer> makeRange(int N)
2     {
3       ArrayList<Integer> list = new ArrayList<Integer>();
4       θ(N) + ∑_{i=1}^{N/2} 2 · i · θ(1)
5       return list
6     }
```

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 2 = 0 \\ i \cdot \theta(1) & \textbf{otherwise} \end{cases}$$

Array add(e)

```
1    public ArrayList<Integer> makeRange(int N)
2    {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4      θ(N) + 2 · θ(1) ∑_{i=1}^{N/2} i
5      return list
6    }
```

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 2 = 0 \\ i \cdot \theta(1) & \textbf{otherwise} \end{cases}$$

Array add(e)

```
1     public ArrayList<Integer> makeRange(int N)
2     {
3       ArrayList<Integer> list = new ArrayList<Integer>();
4       θ(N) + 2 · θ(1) (N/2)((N/2)+1)/2
5       return list
6     }
```

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 2 = 0 \\ i \cdot \theta(1) & \textbf{otherwise} \end{cases}$$

Array add(e)

```
1    public ArrayList<Integer> makeRange(int N)
2    {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4      θ(N) + θ(1)(N²/4 + N/2)
5      return list
6    }
```

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 2 = 0 \\ i \cdot \theta(1) & \textbf{otherwise} \end{cases}$$

Array add(e)

```
1    public ArrayList<Integer> makeRange(int N)
2    {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4      θ(N) + θ(N^2)
5      return list
6    }
```

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 2 = 0 \\ \theta(N) & \textbf{otherwise} \end{cases}$$

Array add(e)

```
1    public ArrayList<Integer> makeRange(int N)
2    {
3      θ(1)
4      θ(N^2)
5      θ(1)
6    }
```

$$T_{\texttt{add(e)}(i)} = \begin{cases} \theta(1) & \textbf{if } i \mod 2 = 0 \\ \theta(N) & \textbf{otherwise} \end{cases}$$

More cowbell

Maybe we need to make it bigger!

100 extra spaces!

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 100 < 99 \\ i \cdot \theta(1) & \textbf{otherwise} \end{cases}$$

Bigger Allocations

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 100 < 99 \\ i \cdot \theta(1) & \textbf{otherwise} \end{cases}$$

```
1    for(i = 0; i < N; i++)
2    {
3      list.add(i);
4    }
```

Bigger Allocations

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 100 < 99 \\ i \cdot \theta(1) & \textbf{otherwise} \end{cases}$$

$$\sum_{i=0}^{N-1} T(i) = T(0) + T(1) + \ldots + T(N)$$

$$\underbrace{\theta(1) + \theta(1) + \ldots + 100 \cdot \theta(1) + \ldots + \theta(1) + N \cdot \theta(1) + \ldots}_{N \text{ terms}}$$

$$\underbrace{\theta(1) + \ldots + \theta(1)}_{\frac{99}{100} N \text{ terms}} + \underbrace{100 \cdot \theta(1) + 200 \cdot \theta(1) + \ldots + N \cdot \theta(1)}_{\frac{1}{100} N \text{ terms}}$$

Bigger Allocations

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 100 < 99 \\ \theta(N) & \textbf{otherwise} \end{cases}$$

$$\underbrace{\theta(1) + \ldots + \theta(1)}_{\frac{99}{100}N \text{ terms}} + \underbrace{100 \cdot \theta(1) + 200 \cdot \theta(1) + \ldots + N \cdot \theta(1)}_{\frac{1}{100}N \text{ terms}}$$

$$\frac{99}{100} N \cdot \theta(1) + \sum_{i=1}^{\frac{N}{100}} i \cdot 100 \cdot \theta(1)$$

$$\frac{99}{100} N \cdot \theta(1) + 100 \cdot \theta(1) \cdot \left(\frac{\frac{N}{100}\left(\frac{N}{100} + 1\right)}{2} \right)$$

Bigger Allocations

$$T_{\texttt{add(e)}}(i) = \begin{cases} \theta(1) & \textbf{if } i \mod 100 < 99 \\ \theta(N) & \textbf{otherwise} \end{cases}$$

$$\frac{99}{100} N \cdot \theta(1) + 100 \cdot \theta(1) \cdot \left(\frac{\frac{N}{100}\left(\frac{N}{100}+1\right)}{2} \right)$$

$$\frac{99}{100} N \cdot \theta(1) + \theta(1) \cdot \left(\frac{N^2}{200} + \frac{N}{2} \right)$$

$$\theta(N) + \theta(N^2)$$

$$\theta(N^2)$$

Bah!

No matter how big we go, N appends will always be $\theta(N^2)$

... or will it?

Idea: Double the array size at each step.

ArrayList

Start with a capacity of 2.

1	$\theta(1)$	(size now 1)
2	$\theta(1)$	(size now 2)
3	$2 \cdot \theta(1)$	(capacity now 4; size now 3)
4	$\theta(1)$	(size now 4)
5	$4 \cdot \theta(1)$	(capacity now 8; size now 5)
6	$\theta(1)$	(size now 6)
7	$\theta(1)$	(size now 7)
8	$\theta(1)$	(size now 8)
9	$8 \cdot \theta(1)$	(capacity now 16; size now 9)

. . . 8 more operations before next $\theta(N)$

. . . 16 more operations before next $\theta(N)$

ArrayList

- 2 insertions at $\theta(1)$
- $2 \cdot \theta(1)$ plus 2 insertions at $\theta(1)$ (up to capacity of 4)
- $4 \cdot \theta(1)$ plus 4 insertions at $\theta(1)$ (up to capacity of 8)
- $8 \cdot \theta(1)$ plus 8 insertions at $\theta(1)$ (up to capacity of 16)
- $16 \cdot \theta(1)$ plus 16 insertions at $\theta(1)$ (up to capacity of 32)
- $32 \cdot \theta(1)$ plus 32 insertions at $\theta(1)$ (up to capacity of 64)
- ...

What's the pattern?
($2^i \cdot \theta(1)$ copy on the 2^i'th insertion)

For N insertions, how many copies do we perform?
($\log_2(N)$)

ArrayList

For insertions 33 . . . 64...

$$\underbrace{32 \cdot \theta(1)}_{\text{copy 32 elements}} + \underbrace{\theta(1) + \ldots + \theta(1)}_{\text{insert 32 elements}}$$

$$= 32 \cdot \theta(1) + 32 \cdot \theta(1) = 64 \cdot \theta(1)$$

For insertions $(2^i + 1) \ldots (2^{i+1})$ (batch i)

$$\underbrace{2^i \cdot \theta(1)}_{\text{copy } 2^i \text{ elements}} + \underbrace{\theta(1) + \ldots + \theta(1)}_{\text{insert } 2^i \text{ elements}}$$

$$= 2^{i+1} \cdot \theta(1) \text{ (for } 2^i \text{ insertions)}$$

ArrayList

```
1    for(i = 0; i < N; i++)
2    {
3      list.add(i);
4    }
```

Split the N adds into $\log_2(N)$ batches (plus a zero'th batch)

$$\underbrace{\sum_{i=0}^{\log_2(N)}}_{\log_2(N) \text{ batches}} \underbrace{2^{i+1} \cdot \theta(1)}_{\text{cost for } i\text{'th batch}}$$

ArrayList

Split the N adds into $\log_2(N)$ batches (plus a zero'th batch)

$$\underbrace{\sum_{i=0}^{\log_2(N)}}_{\log_2(N) \text{ batches}} \underbrace{2^{i+1} \cdot \theta(1)}_{\text{cost for } i\text{'th batch}}$$

$$= 2^{\log_2(N)+1} - 1$$

$$= (2^1 \cdot 2^{\log_2(N)} - 1) \cdot \theta(1)$$

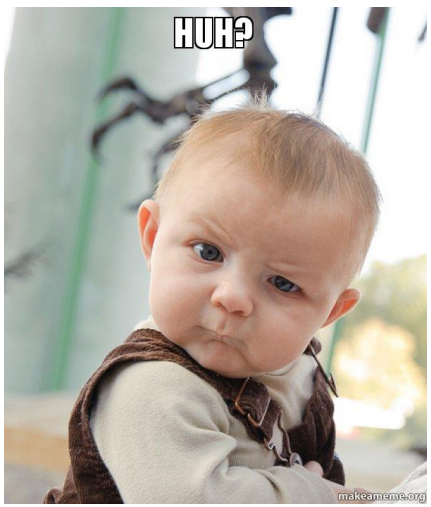
$$= (2 \cdot 2^{\log_2(N)} - 1) \cdot \theta(1)$$

ArrayList

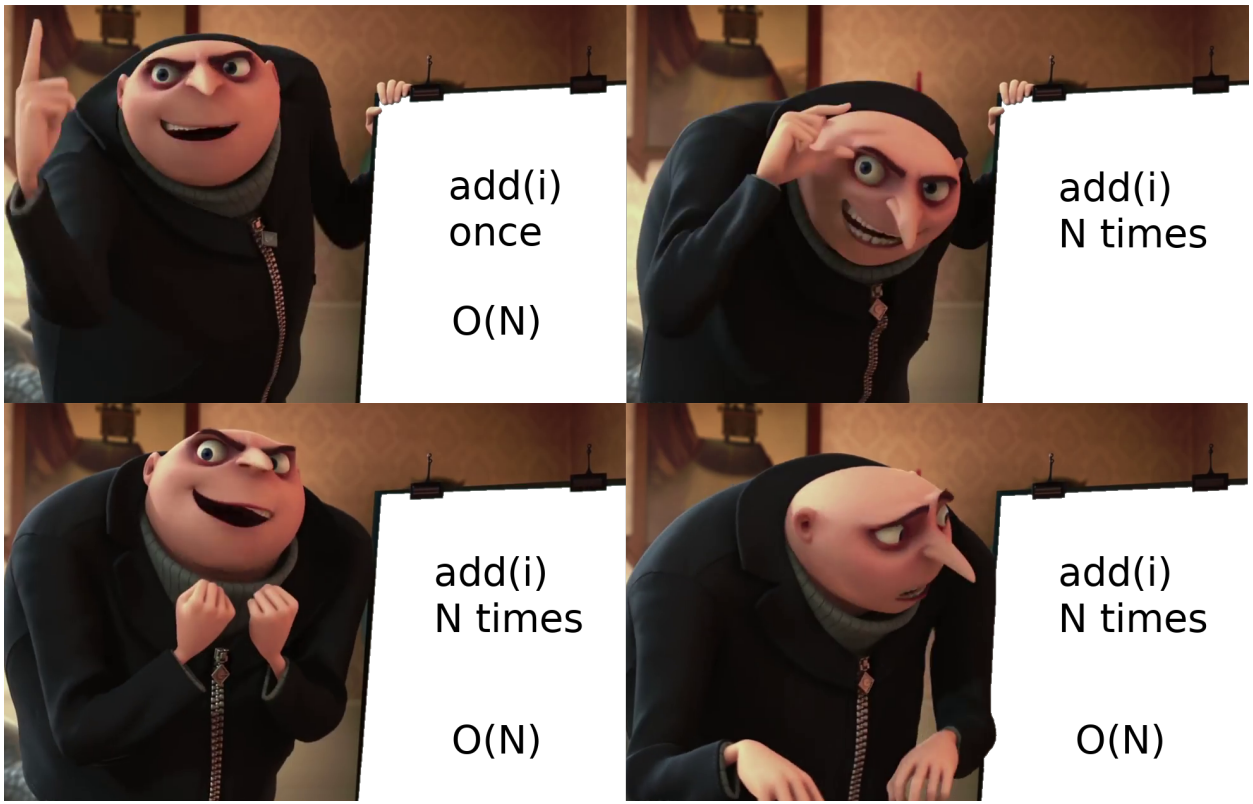
$$= (2 \cdot 2^{\log_2(N)} - 1) \cdot \theta(1)$$

$$= (2N - 1) \cdot \theta(1)$$

$$= \theta(N)$$



Huh?



Despicable Me; ©2010 Universal Pictures

`add(i)`

This is weird!

Amortized Runtimes

$$T_{add}(N) = \begin{cases} \theta(1) & \textbf{if } capacity > size \\ \theta(N) & \textbf{otherwise} \end{cases}$$

$T_{add}(N) \in O(N)$

- Any **one** call could be $O(N)$
- But the $O(N)$ case happens rarely.
  - ... rarely enough (with doubling) that the expensive call *amortizes* over the cheap calls.

LinkedList vs ArrayList

```
1    for(i = 0; i < N; i++)
2    {
3      list.add(i);
4    }
```

	`LinkedList`	`ArrayList`
`add(i)` once	$O(1)$	$O(N)$
`add(i)` N times	$O(N)$	$O(N)$

`ArrayList.add(i)` behaves like it's $O(1)$, but only when it's in a loopbut only when it's in a loop**.**

Amortized Runtimes

Let's give this behavior a name: **Amortized Runtime**

Amortized Runtime

- The tight *unqualified* upper bound on `add(i)` is O(N)
  Any one call to `add(i)` could take up to $O(N)$.
- The tight *amortized* upper bound on `add(i)` is O(1)
  N calls to `add(i)` average out to $O(1)$ each.
  ($O(N)$ for all N calls)

Amortized Runtime

If $T(N)$ runs in *amortized* $O(f(N))$, then:

$$\sum_{i=0}^{N} T(N) = \quad N \cdot O(f(N)) = \quad O(N \cdot f(N))$$

Even if $T(N) \notin O(f(N))$

Amortized Runtime

- **Unqualified Bounds**: Always true (no qualifiers)
- **Amortized Bounds**: Only valid in $\sum_{i=0}^{N} T(i)$
  - One call may be expensive, many calls average out cheap

ArrayList

`add(E value)`

- If insufficient capacity:
  - Allocate new array at $2\times$ capacity ← O(1)
  - Copy elements to new array ← O(N)
- Append new element to next open spot ← O(1)
- Update size ← O(1)

Unqualified Runtime: $O(N)$
Amortized Runtime: $O(1)$

ArrayList

`add(E value, int idx)`

- If insufficient capacity:
  - Allocate new array at $2\times$ capacity ←O(1)
  - Copy elements to new array ←O(N)
- Shift elements after `idx` right one spot ←O(N)
- Insert new element into open spot ←O(1)
- Update size ←O(1)

Unqualified Runtime: $O(N)$
Amortized Runtime: $O(N)$ (shift occurs on every call)

ArrayList

`remove(int idx)`

- Shift elements after `idx` left one spot ←O(N)
- Update size ←O(1)

Unqualified Runtime: $O(N)$
Amortized Runtime: $O(N)$