

CSE 250: ArrayList, Amortized Complexity

Lecture 10

Sept 18, 2024

Reminders

- PA1 Implementation due Sun, Sept 22 at 11:59 PM
 - Implement a Sorted Linked List

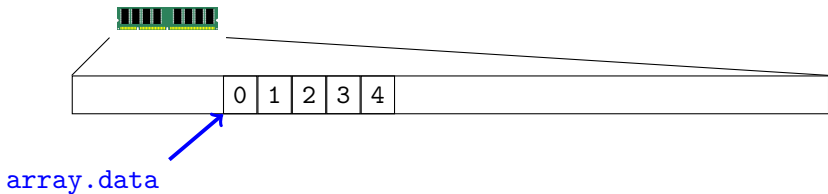
The Sequence ADT

```
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



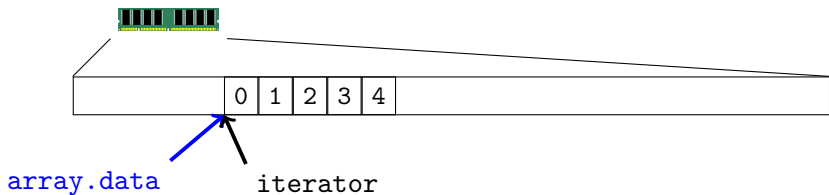
Array Iterators



`array.data`

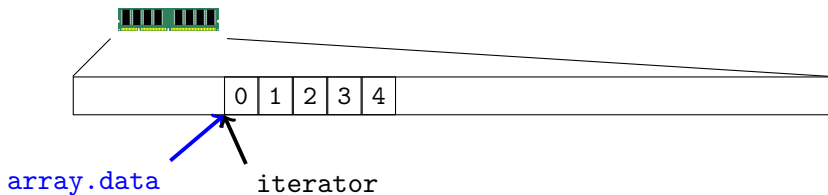
```
list.iterator()
```

Array Iterators



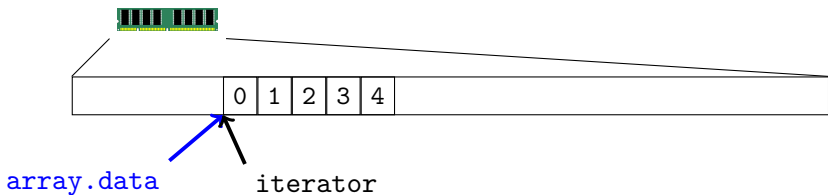
```
list.iterator() → iterator
```

Array Iterators



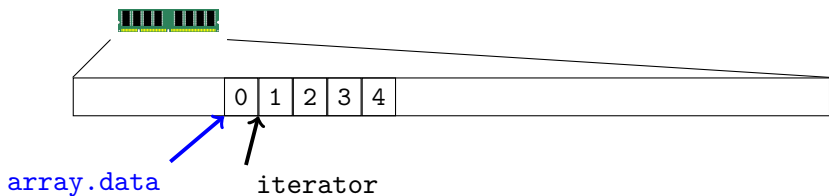
```
list.iterator() → iterator  
iterator.next()
```


Array Iterators



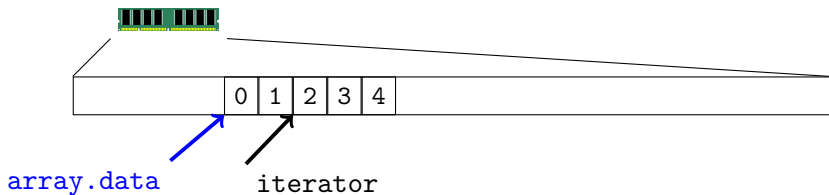
```
list.iterator() → iterator  
iterator.next() → 0
```

Array Iterators



```
list.iterator() → iterator  
iterator.next() → 0
```

Array Iterators

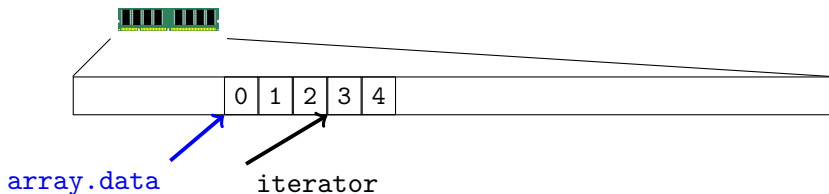


```
list.iterator() → iterator
```

```
iterator.next() → 0
```

```
iterator.next() → 1
```

Array Iterators



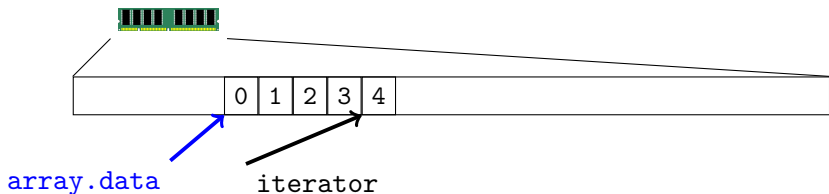
```
list.iterator() → iterator
```

```
iterator.next() → 0
```

```
iterator.next() → 1
```

```
iterator.next() → 2
```

Array Iterators



```
list.iterator() → iterator
```

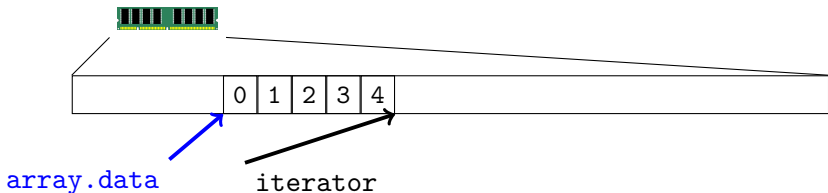
```
iterator.next() → 0
```

```
iterator.next() → 1
```

```
iterator.next() → 2
```

```
iterator.next() → 3
```

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()
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;
12         return array.get(nextIndex-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 E next()
10     {
11         nextIndex += 1;           ← O(1)
12         return array.get(nextIndex-1)
13     }
14 }
```

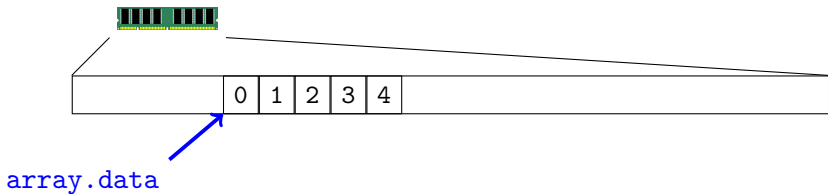
ArrayListIterator

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



`array.data`

```
array.add(idx= 2, value= 5)
```

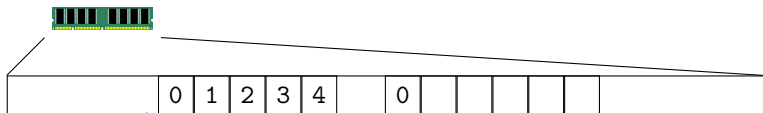
Array add(idx, value)



`array.data`

```
array.add(idx= 2, value= 5)
```

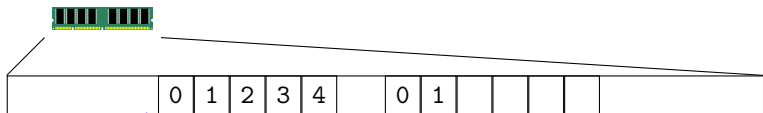
Array add(idx, value)



array.data

```
array.add(idx= 2, value= 5)
```

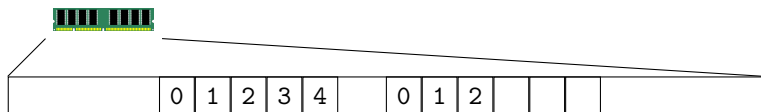
Array add(idx, value)



`array.data`

```
array.add(idx= 2, value= 5)
```

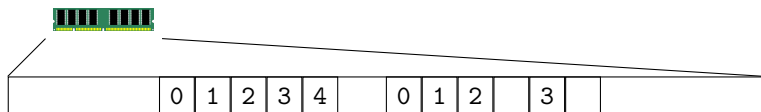
Array add(idx, value)



`array.data`

```
array.add(idx= 2, value= 5)
```

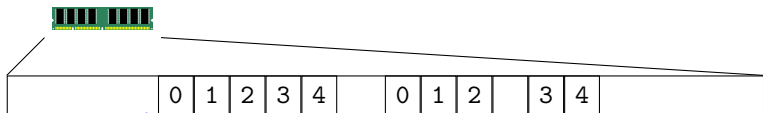
Array add(idx, value)



`array.data`

```
array.add(idx= 2, value= 5)
```

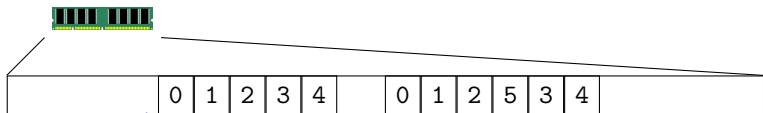
Array add(idx, value)



`array.data`

```
array.add(idx= 2, value= 5)
```

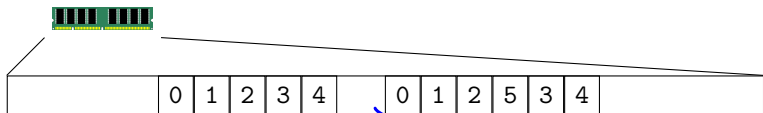
Array add(idx, value)



`array.data`

```
array.add(idx= 2, value= 5)
```

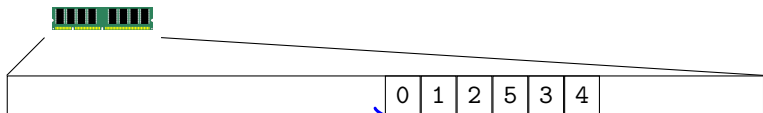

Array add(idx, value)



array.data

```
array.add(idx= 2, value= 5)
```

Array add(idx, value)



`array.data`

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

Today's Focus

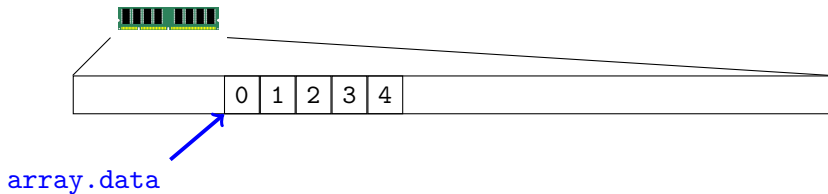
Let's start with `add(value)` (append to end).

Can we make it faster?

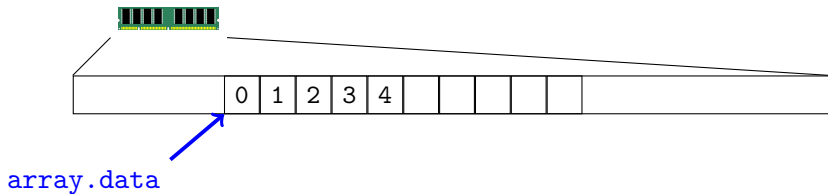
Idea 1

Idea: Allocate more memory than we need.

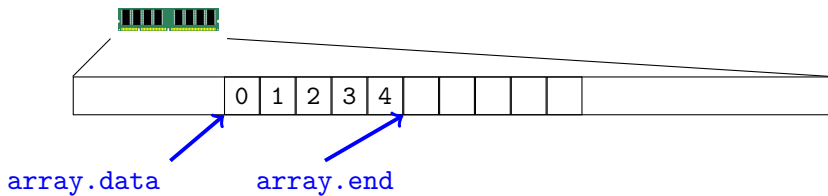
Idea 1



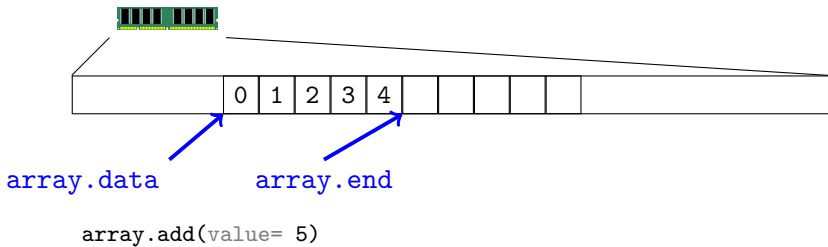
Idea 1



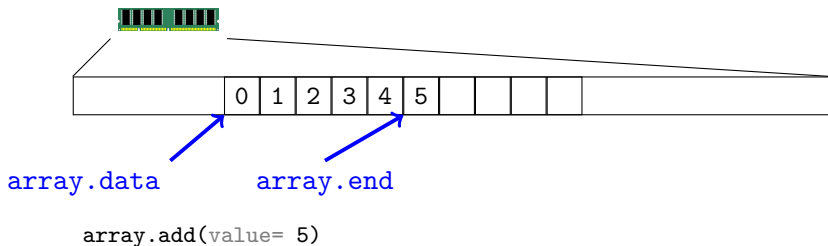
Idea 1



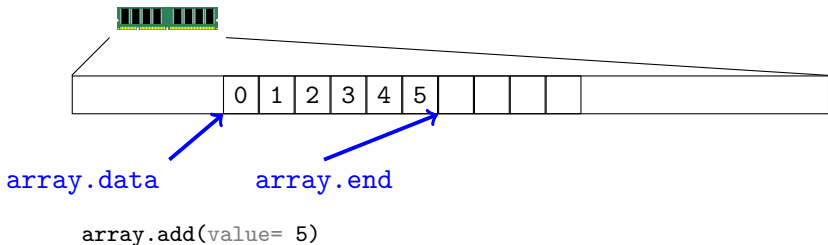
Idea 1



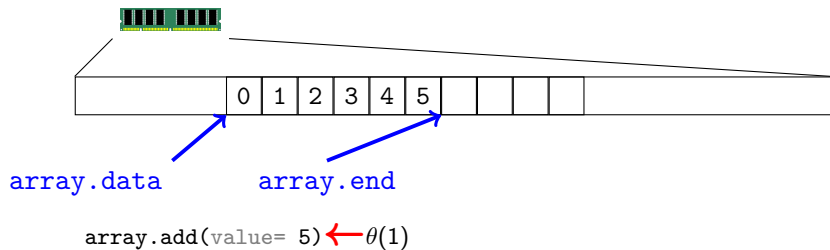
Idea 1



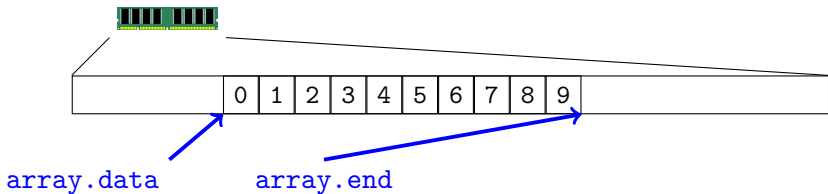
Idea 1



Idea 1



Idea 1



```
array.add(value= 5) ←  $\theta(1)$ 
```

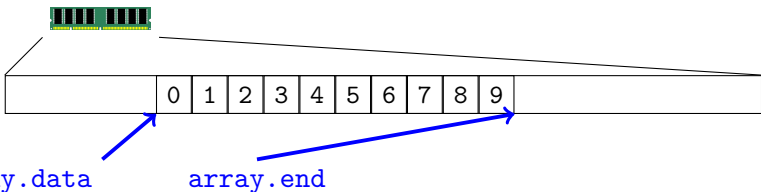
```
array.add(value= 6) ←  $\theta(1)$ 
```

```
array.add(value= 7) ←  $\theta(1)$ 
```

```
array.add(value= 8) ←  $\theta(1)$ 
```

```
array.add(value= 9) ←  $\theta(1)$ 
```

Idea 1



```
array.add(value= 5) ←  $\theta(1)$ 
```

```
array.add(value= 6) ←  $\theta(1)$ 
```

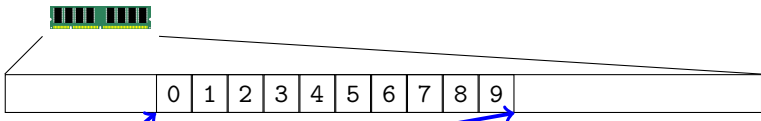
```
array.add(value= 7) ←  $\theta(1)$ 
```

```
array.add(value= 8) ←  $\theta(1)$ 
```

```
array.add(value= 9) ←  $\theta(1)$ 
```

```
array.add(value= 10)
```

Idea 1



array.data

array.end

array.add(value= 5) ← $\theta(1)$ array.add(value= 6) ← $\theta(1)$ array.add(value= 7) ← $\theta(1)$ array.add(value= 8) ← $\theta(1)$ array.add(value= 9) ← $\theta(1)$ array.add(value= 10) ← $\theta(N)$

ArrayList (extra space edition)

- `array.size()`: How many elements are currently stored.
- `array.capacity()`: How many elements can be stored before we need to make a bigger array.

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)

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

Array add(e)

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

So...

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

Array add(e)

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

So...

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

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      $\sum_{i=0}^N O(N)$ 
5     return list
6 }
```

Array add(e)

```
1 public ArrayList<Integer> makeRange(int N)
2 {
3     O(1)
4     O(N2)
5     O(1)
6 }
```

Array add(e)

```
1 public ArrayList<Integer> makeRange(int N)
2 {
3     O(1)
4     O(N2)
5     O(1)
6 }
```

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

Array add(e)

```
1 public ArrayList<Integer> makeRange(int N)
2 {
3     O(1)
4     O(N2)
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?

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.

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          $T_{\text{add}(e)}(i)$ 
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         Tadd(e)(i)
7     }
8     return list
9 }
```

$$T_{\text{add}(e)}(i) = \begin{cases} \theta(1) & \text{if } i \bmod 2 = 0 \\ O(N) & \text{otherwise} \end{cases}$$

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              Tadd(e)(i)
7          }
8      return list
9  }
```

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

Array add(e)

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

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

Array add(e)

```

1  public ArrayList<Integer> makeRange(int N)
2  {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4       $T_{\text{add}(e)}(0) + T_{\text{add}(e)}(1) + \dots + T_{\text{add}(e)}(N-1)$ 
5      return list
6  }
```

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

Array add(e)

```

1  public ArrayList<Integer> makeRange(int N)
2  {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4       $\theta(1) + 2 \cdot \theta(1) + \theta(1) + 4 \cdot \theta(1) + \dots + \theta(1) + N \cdot \theta(1)$ 
5      return list
6  }

```

N terms

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

Array add(e)

```

1  public ArrayList<Integer> makeRange(int N)
2  {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4       $\underbrace{\theta(1) + \dots + \theta(1)}_{\frac{N}{2} \text{ terms}} + \underbrace{2 \cdot \theta(1) + 4 \cdot \theta(1) + 6 \cdot \theta(1) + \dots + N \cdot \theta(1)}_{\frac{N}{2} \text{ terms}}$ 
5      return list
6  }
```

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

Array add(e)

```

1  public ArrayList<Integer> makeRange(int N)
2  {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4       $\frac{N}{2}\theta(1) + 2 \cdot \theta(1) + 4 \cdot \theta(1) + 6 \cdot \theta(1) + \dots + N \cdot \theta(1)$ 
5
6      return list
  }
```

$\underbrace{\hspace{15em}}_{\frac{N}{2} \text{ terms}}$

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

Array add(e)

```

1  public ArrayList<Integer> makeRange(int N)
2  {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4       $\theta(N) + 2 \cdot \theta(1) + 4 \cdot \theta(1) + 6 \cdot \theta(1) + \dots + N \cdot \theta(1)$ 
5
6      return list
  }
```

$\underbrace{\hspace{15em}}_{\frac{N}{2} \text{ terms}}$

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

Array add(e)

```

1  public ArrayList<Integer> makeRange(int N)
2  {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4       $\theta(N) + \sum_{i=1}^{N/2} 2 \cdot i \cdot \theta(1)$ 
5      return list
6  }
```

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

Array add(e)

```

1  public ArrayList<Integer> makeRange(int N)
2  {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4       $\theta(N) + 2 \cdot \theta(1) \sum_{i=1}^{N/2} i$ 
5      return list
6  }
```

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

Array add(e)

```

1  public ArrayList<Integer> makeRange(int N)
2  {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4       $\theta(N) + 2 \cdot \theta(1) \frac{N(\frac{N}{2} + 1)}{2}$ 
5      return list
6  }

```

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

Array add(e)

```

1  public ArrayList<Integer> makeRange(int N)
2  {
3      ArrayList<Integer> list = new ArrayList<Integer>();
4       $\theta(N) + \theta(1) \left( \frac{N^2}{4} + \frac{N}{2} \right)$ 
5      return list
6  }
```

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

Array add(e)

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

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

Array add(e)

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

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

More cowbell

Maybe we need to make it bigger!

More cowbell

Maybe we need to make it bigger!

100 extra spaces!

More cowbell

Maybe we need to make it bigger!

100 extra spaces!

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

Bigger Allocations

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

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


Bigger Allocations

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

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

Bigger Allocations

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

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

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

Bigger Allocations

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

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

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

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

Bigger Allocations

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

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

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

Bigger Allocations

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

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

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

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

Bigger Allocations

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

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

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

Bigger Allocations

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

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

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

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

Bigger Allocations

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

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

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

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

$$\theta(N^2)$$

Bah!

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

Bah!

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

... or will it?

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 $\theta(1)$

ArrayList

Start with a capacity of 2.

1 $\theta(1)$

(size now 1)

ArrayList

Start with a capacity of 2.

1 $\theta(1)$

2 $\theta(1)$

(size now 1)

ArrayList

Start with a capacity of 2.

1 $\theta(1)$

(size now 1)

2 $\theta(1)$

(size now 2)

3 $2 \cdot \theta(1)$

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)

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 $4 \cdot \theta(1)$

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)

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) |

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 $\theta(N)$

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 $\theta(1)$

ArrayList

- 2 insertions at $\theta(1)$
- $2 \cdot \theta(1)$ plus 2 insertions at $\theta(1)$ (up to capacity of 4)

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)

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)

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)

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)

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)
- ...

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?

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)

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?

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...

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

ArrayList

For insertions 33 ...64...

$$\underbrace{32 \cdot \theta(1)}_{\text{copy 32 elements}} + \underbrace{\theta(1) + \dots + \theta(1)}_{\text{insert 32 elements}}$$
$$= 32 \cdot \theta(1) + 32 \cdot \theta(1) = 64 \cdot \theta(1)$$

ArrayList

For insertions 33 ...64...

$$\underbrace{32 \cdot \theta(1)}_{\text{copy 32 elements}} + \underbrace{\theta(1) + \dots + \theta(1)}_{\text{insert 32 elements}}$$
$$= 32 \cdot \theta(1) + 32 \cdot \theta(1) = 64 \cdot \theta(1)$$

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

ArrayList

For insertions 33 ...64...

$$\begin{aligned}
 & \underbrace{32 \cdot \theta(1)}_{\text{copy 32 elements}} + \underbrace{\theta(1) + \dots + \theta(1)}_{\text{insert 32 elements}} \\
 & = 32 \cdot \theta(1) + 32 \cdot \theta(1) = 64 \cdot \theta(1)
 \end{aligned}$$

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

ArrayList

For insertions 33 ...64...

$$\begin{aligned}
 & \underbrace{32 \cdot \theta(1)}_{\text{copy 32 elements}} + \underbrace{\theta(1) + \dots + \theta(1)}_{\text{insert 32 elements}} \\
 & = 32 \cdot \theta(1) + 32 \cdot \theta(1) = 64 \cdot \theta(1)
 \end{aligned}$$

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

$$\begin{aligned}
 & \underbrace{2^i \cdot \theta(1)}_{\text{copy } 2^i \text{ elements}} + \underbrace{\theta(1) + \dots + \theta(1)}_{\text{insert } 2^i \text{ elements}} \\
 & = 2^{i+1} \cdot \theta(1)
 \end{aligned}$$

ArrayList

For insertions 33 ...64...

$$\begin{aligned}
 & \underbrace{32 \cdot \theta(1)}_{\text{copy 32 elements}} + \underbrace{\theta(1) + \dots + \theta(1)}_{\text{insert 32 elements}} \\
 & = 32 \cdot \theta(1) + 32 \cdot \theta(1) = 64 \cdot \theta(1)
 \end{aligned}$$

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

$$\begin{aligned}
 & \underbrace{2^i \cdot \theta(1)}_{\text{copy } 2^i \text{ elements}} + \underbrace{\theta(1) + \dots + \theta(1)}_{\text{insert } 2^i \text{ elements}} \\
 & = 2^{i+1} \cdot \theta(1) \text{ (for } 2^i \text{ insertions)}
 \end{aligned}$$

ArrayList

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

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)

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)

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

ArrayList

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

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

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

ArrayList

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

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

ArrayList

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

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

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

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

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

ArrayList

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

ArrayList

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

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

ArrayList

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

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

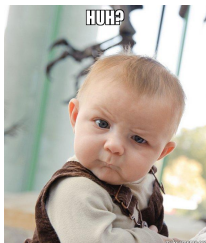
$$= \theta(N)$$

ArrayList

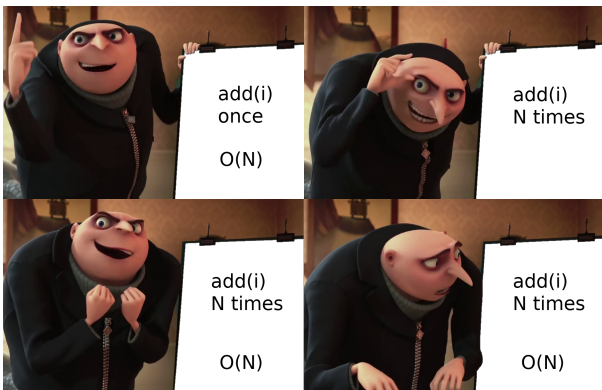
$$= (2 \cdot 2^{\log_2(N)} - 1) \cdot \theta(1)$$

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

$$= \theta(N)$$



Huh?



Despicable Me; ©2010 Universal Pictures

`add(i)`

This is weird!

Amortized Runtimes

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

Amortized Runtimes

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

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

Amortized Runtimes

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

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

- Any **one** call could be $O(N)$
- But the $O(N)$ case happens rarely.

Amortized Runtimes

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

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

- 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		
add(i) N times		

LinkedList vs ArrayList

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

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

LinkedList vs ArrayList

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

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

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	$O(N)$	$O(N)$

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 loop.

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 loop.

Amortized Runtimes

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

Amortized Runtime

- The tight upper bound on $\text{add}(i)$ is $O(N)$

Amortized Runtime

- The tight upper bound on $\text{add}(i)$ is $O(N)$
Any one call to $\text{add}(i)$ could take up to $O(N)$.

Amortized Runtime

- The tight upper bound on $\text{add}(i)$ is $O(N)$
Any one call to $\text{add}(i)$ could take up to $O(N)$.
- The tight amortized upper bound on $\text{add}(i)$ is $O(1)$

Amortized Runtime

- The tight upper bound on $\text{add}(i)$ is $O(N)$
Any one call to $\text{add}(i)$ could take up to $O(N)$.
- The tight amortized upper bound on $\text{add}(i)$ is $O(1)$
 N calls to $\text{add}(i)$ average out to $O(1)$ each.

Amortized Runtime

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

Amortized Runtime

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

Amortized Runtime

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

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

Amortized Runtime

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

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

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
 - Copy elements to new array
- Append new element to next open spot
- Update size

ArrayList

add(E value)

- If insufficient capacity:
 - Allocate new array at $2\times$ capacity
 - Copy elements to new array
- Append new element to next open spot
- Update size

← $O(1)$

ArrayList

add(E value)

- If insufficient capacity:
 - Allocate new array at $2\times$ capacity
 - Copy elements to new array
- Append new element to next open spot
- Update size

← $O(1)$

← $O(N)$

ArrayList

add(E value)

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

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: $O(N)$

ArrayList

add(E value)

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

Unqualified Runtime: $O(N)$

Amortized Runtime: $O(1)$

ArrayList

`add(E value, int idx)`

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

ArrayList

`add(E value, int idx)`

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

← $O(1)$

ArrayList

`add(E value, int idx)`

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

← $O(1)$

← $O(N)$

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
- Update size

ArrayList

`add(E value, int idx)`

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

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: $O(N)$

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: $O(N)$

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
- Update size

ArrayList

`remove(int idx)`

- Shift elements after `idx` left one spot
- Update size

← $O(N)$

ArrayList

`remove(int idx)`

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

ArrayList

`remove(int idx)`

- Shift elements after `idx` left one spot
- Update size

← $O(N)$

← $O(1)$

Unqualified Runtime: $O(N)$

ArrayList

`remove(int idx)`

- Shift elements after `idx` left one spot
- Update size

← $O(N)$

← $O(1)$

Unqualified Runtime: $O(N)$

Amortized Runtime: $O(N)$