CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
Capen 208

Lecture 10
ArrayLists

Announcements

- PA1 Implementation AutoLab is now open!
  - Don’t rely on AutoLab to tell you what’s wrong – that’s what **your** tests are for
  - Add more tests as needed
- WA2 releases next week

The List ADT

```
1  public interface List<E>
2      extends Sequence<E> { // Everything a Sequence has plus...
3    /** Append a new element to the end of the list */
4    public void add(E value);
5
6    /** Insert a new element at a given index */
7    public void add(int idx, E value);
8
9    /** Remove the element at a given index */
10   public void remove(int idx);
11 }
```

`List` Summary (so far...)

	`Array`	`LinkedList` *(by index)*	`LinkedList` *(by reference)*
`get(i)`	$\Theta(1)$	$\Theta(i) \subset O(n)$	$\Theta(1)$
`set(i, v)`	$\Theta(1)$	$\Theta(i) \subset O(n)$	$\Theta(1)$
`size()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`add(v)`	***???***	$\Theta(1)$	$\Theta(1)$
`add(i, v)`	***???***	$\Theta(i) \subset O(n)$	$\Theta(1)$
`remove(idx)`	***???***	$\Theta(i) \subset O(n)$	$\Theta(1)$

Implementing `List` with an `Array`

Question: How can we implement the **`List`** ADT using an **`Array`**?

Implementing `List` with an `Array`

Question: How can we implement the **`List`** ADT using an **`Array`**?

Problem: **`Array`**s have a fixed size! We **cannot** add an element to an **`Array`**...

Implementing `List` with an `Array`

Question: How can we implement the **`List`** ADT using an **`Array`**?

Problem: **`Array`**s have a fixed size! We **cannot** add an element to an **`Array`**...

...but could we fake it?

Implementing `List` with an `Array`

Question: How can we implement the **`List`** ADT using an **`Array`**?

Problem: **`Array`**s have a fixed size! We **cannot** add an element to an **`Array`**...

...but could we fake it?

How would we ***simulate*** appending an element to an **`Array`**?

Attempt #1

To "add" to an `Array` of size n...

1. Allocate a new **`Array`** of size $n + 1$	$O(1)$
2. Copy all n elements to the new **`Array`**	$\Theta(n)$
3. Set the element at index n to the inserted value	$O(1)$

Total: $\Theta(n)$

Attempt #1

To "add" to an `Array` of size n...

1. Allocate a new **`Array`** of size $n+1$	$O(1)$
2. Copy all n elements to the new **`Array`**	$\Theta(n)$
3. Set the element at index n to the inserted value	$O(1)$

Total: $\Theta(n)$

Can we do better?

Attempt #1

To "add" to an `Array` of size n...

1. Allocate a new **`Array`** of size $n+1$	$O(1)$
2. Copy all n elements to the new **`Array`**	$\Theta(n)$
3. Set the element at index n to the inserted value	$O(1)$

Total: $\Theta(n)$

Can we do better?

Idea: Keep extra space in the **`Array`** and track how many elements we have

This is the basis of an **`ArrayList`**

`ArrayList` Representation

```
1  class ArrayList<T> extends List<T> {
2      private int used;
3      private Optional<T>[] data;
4
5      public int size() { return used; }
6
…                                  …
39 }
```

Runtime of **`size`**?

`ArrayList` Representation

```
1  class ArrayList<T> extends List<T> {
2      private int used;
3      private Optional<T>[] data;
4
5      public int size() { return used; }
6
…                                    …
39 }
```

Runtime of **`size`**? $\Theta(1)$

`ArrayList` Representation

```
1  public T get(int i) {
2    if (i < 0 || i >= used) throw new IndexOutofBoundsException(i);
3    return data[i].get();
4  }
5
6  public void set(int i, T value) {
7    if (i < 0 || i >= used) throw new IndexOutofBoundsException(i);
8    data[i] = Optional.of(value);
9  }
```

Runtime of **`get`**?

Runtime of **`set`**?

`ArrayList` Representation

```
1  public T get(int i) {
2    if (i < 0 || i >= used) throw new IndexOutofBoundsException(i);
3    return data[i].get();
4  }
5
6  public void set(int i, T value) {
7    if (i < 0 || i >= used) throw new IndexOutofBoundsException(i);
8    data[i] = Optional.of(value);
9  }
```

Runtime of **`get`**? $\Theta(1)$

Runtime of **`set`**? $\Theta(1)$

`ArrayList` `get/set/size` - Analysis

The **`Sequence`** methods are all still $\Theta(1)$, great!

What about the **`List`** methods?

ArrayList.remove(i) - Implementation

```
1  public void remove(int i) {
2    if (i < 0 || i >= used) throw new IndexOutofBoundsException(i);
3
4    for (int j = i; j < used - 1; j++) {
5      data[j] = data[j+1];
6    }
7
8    data[used - 1] = Optional.empty();
9  }
```

ArrayList.remove(i) - Implementation

```
1  public void remove(int i) {
2    if (i < 0 || i >= used) throw new IndexOutofBoundsException(i);
3
4    for (int j = i; j < used - 1; j++) {
5      data[j] = data[j+1];
6    }
7
8    data[used - 1] = Optional.empty();
9  }
```

We have to shift over elements to fill the hole...possibly ALL elements!

`ArrayList.remove(i)` - Analysis

$$T_{\text{remove}}(n) = \begin{cases} 1 & \text{if } i = \text{used} - 1 \\ 2 & \text{if } i = \text{used} - 2 \\ 3 & \text{if } i = \text{used} - 3 \\ \ldots & \ldots \\ n - 1 & \text{if } i = 0 \end{cases}$$

What are the bounds of $T_{\text{remove}}(n)$?

`ArrayList.remove(i)` - Analysis

$$T_{\text{remove}}(n) = \begin{cases} 1 & \text{if } i = \text{used} - 1 \\ 2 & \text{if } i = \text{used} - 2 \\ 3 & \text{if } i = \text{used} - 3 \\ \ldots & \ldots \\ n - 1 & \text{if } i = 0 \end{cases}$$

What are the bounds of $T_{\text{remove}}(n)$?

$$T_{\text{remove}}(n) \in O(n), \Omega(1)$$

`ArrayList.add(elem)` - Intuition

Basic Idea:

1. If the underlying **`Array`** is full:
    1. Create a new, larger, **`Array`**
    2. Copy the data over
    3. You now have some free space at the end
2. Update the next free element to hold the newly inserted value

ArrayList.add(elem) - Implementation

```
1  public void add(T elem) {
2    if (used == data.length) {
3      int newLength = computeNewLength(data.length);
4      Optional<T>[] newData = new Optional<T>[newLength];
5      System.arrayCopy(data, 0, newData, 0, data.length);
6      data = new Data;
7    }
8    data[used] = Optional.of(elem);
9    used++;
10 }
```

ArrayList.add(elem) - Implementation

```
1  public void add(T elem) {
2    if (used == data.length) {
3      int newLength = computeNewLength(data.length);
4      Optional<T>[] newData = new Optional<T>[newLength];
5      System.arrayCopy(data, 0, newData, 0, data.length);
6      data = new Data;
7    }
8    data[used] = Optional.of(elem);
9    used++;
10 }
```

Pick a new size for your data...how? TBD

ArrayList.add(elem) - Implementation

```
1  public void add(T elem) {
2    if (used == data.length) {
3      int newLength = computeNewLength(data.length);
4      Optional<T>[] newData = new Optional<T>[newLength];
5      System.arrayCopy(data, 0, newData, 0, data.length);
6      data = new Data;
7    }
8    data[used] = Optional.of(elem);
9    used++;
10 }
```

Copying the data over is costly! Need to copy all n elements.

ArrayList.add(elem) - Implementation

```
1  public void add(T elem) {
2    if (used == data.length) {
3      int newLength = computeNewLength(data.length);
4      Optional<T>[] newData = new Optional<T>[newLength];
5      System.arrayCopy(data, 0, newData, 0, data.length);
6      data = new Data;
7    }
8    data[used] = Optional.of(elem);
9    used++;
10 }
```

When no copying is needed, insertion is cheap!

`ArrayList.add(elem)` - Analysis

$$T_{\text{add}}(n) = \begin{cases} 1 \text{ if used} < \text{data.length} \\ n \text{ if used} = \text{data.length} \end{cases}$$

What are the bounds of $T_{\text{add}}(n)$?

`ArrayList.add(elem)` - Analysis

$$T_{\text{add}}(n) = \begin{cases} 1 \text{ if used} < \text{data.length} \\ n \text{ if used} = \text{data.length} \end{cases}$$

What are the bounds of $T_{\text{add}}(n)$?

$$T_{\text{add}}(n) \in O(n), \Omega(1)$$

`ArrayList.add(elem)` - Analysis

$$T_{\text{add}}(n) = \begin{cases} 1 \text{ if used} < \text{data.length} \\ n \text{ if used} = \text{data.length} \end{cases}$$

What are the bounds of $T_{\text{add}}(n)$?

$$T_{\text{add}}(n) \in O(n), \Omega(1)$$

How often do our add calls require n steps?

`ArrayList.add(elem)` - Analysis

$$T_{\text{add}}(n) = \begin{cases} 1 \text{ if used} < \text{data.length} \\ n \text{ if used} = \text{data.length} \end{cases}$$

What are the bounds of $T_{\text{add}}(n)$?

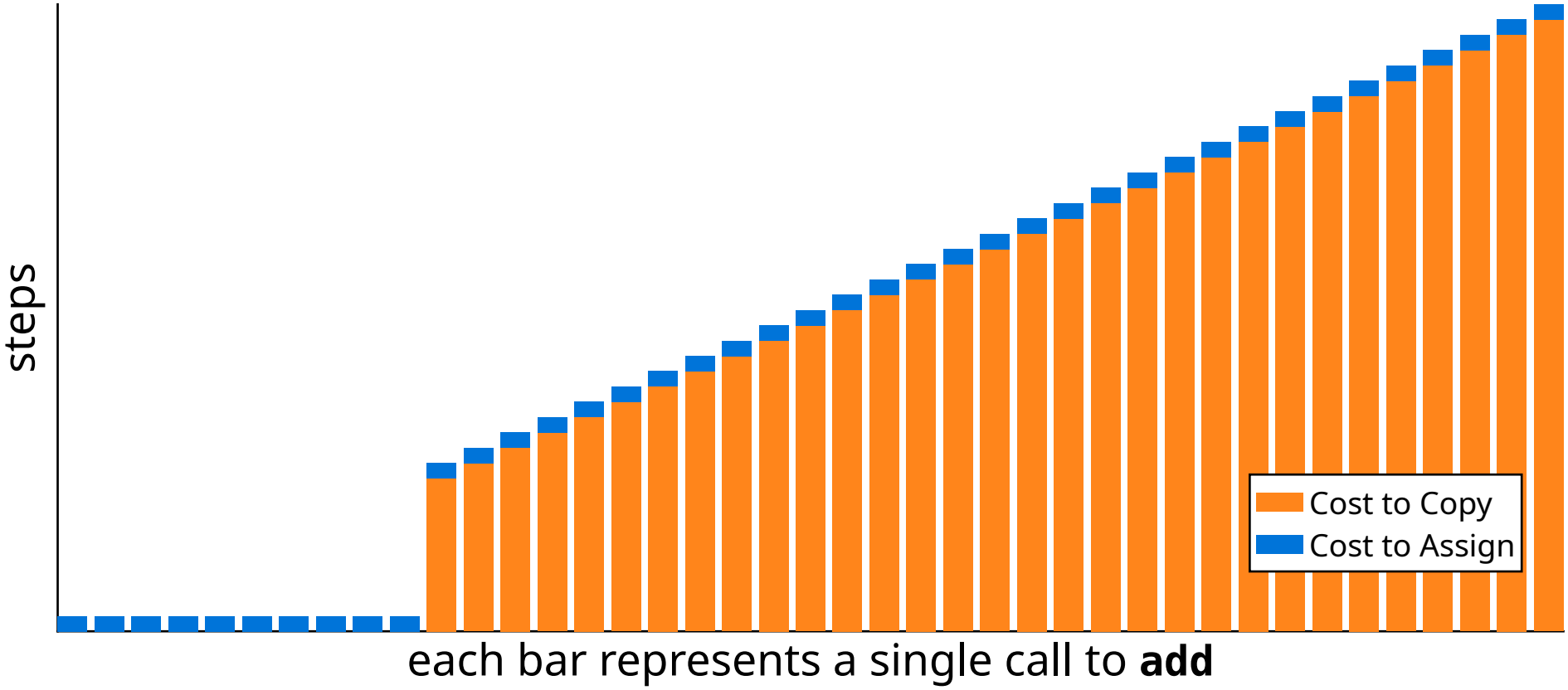
$$T_{\text{add}}(n) \in O(n), \Omega(1)$$

How often do our add calls require n steps?

It depends on how we calculate **`newLength`**!

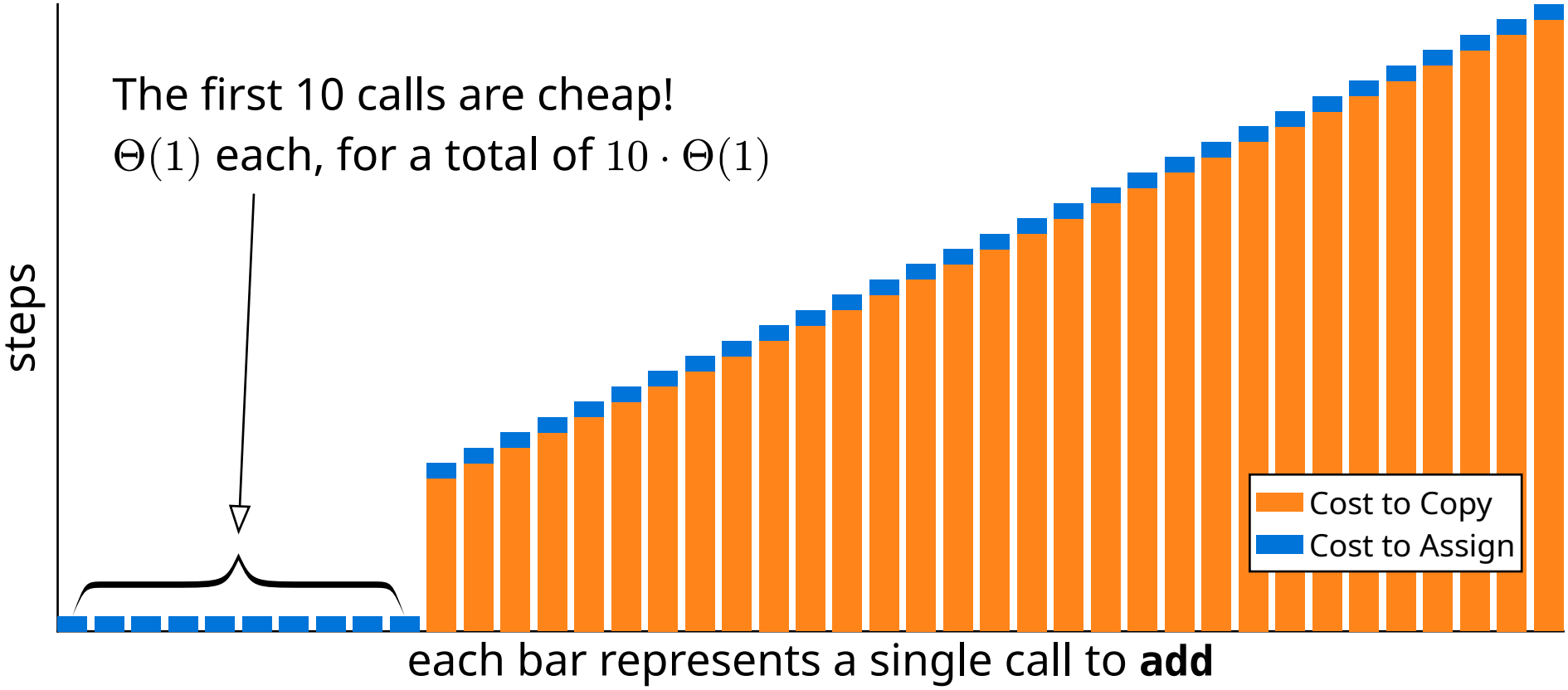
`ArrayList.add(elem)`: +1 Each Resize

Initial Size = 10, **`newLength = data.length + 1`**



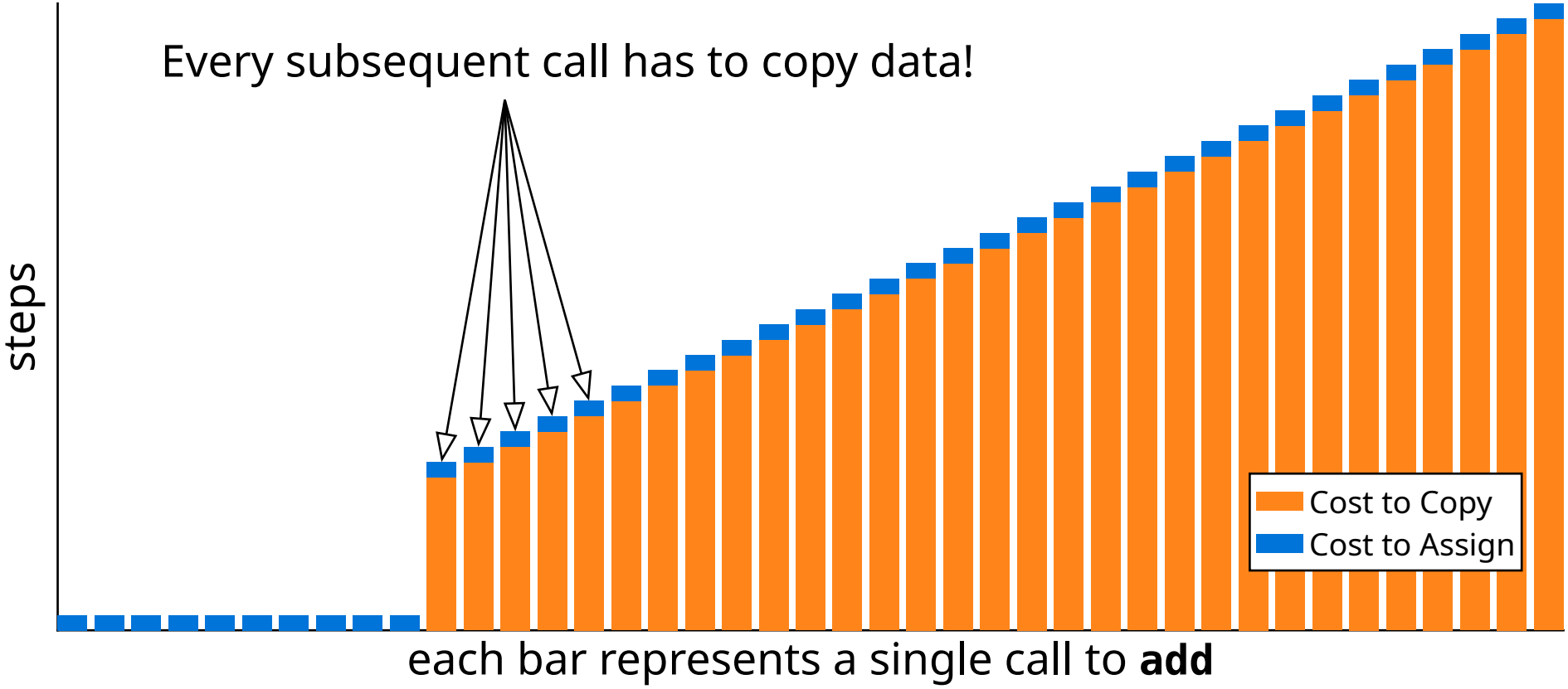
`ArrayList.add(elem)`: +1 Each Resize

Initial Size = 10, **`newLength = data.length + 1`**



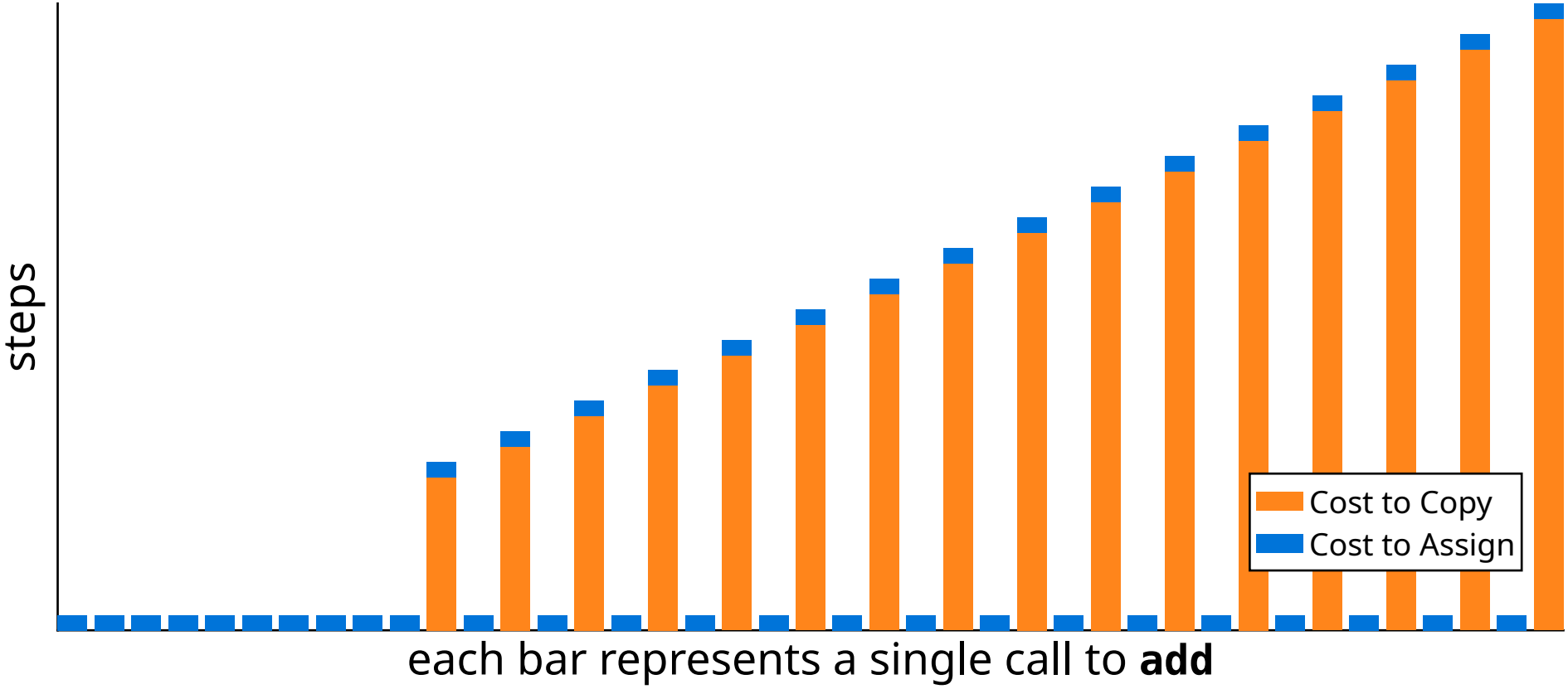
`ArrayList.add(elem)`: +1 Each Resize

Initial Size = 10, **`newLength = data.length + 1`**



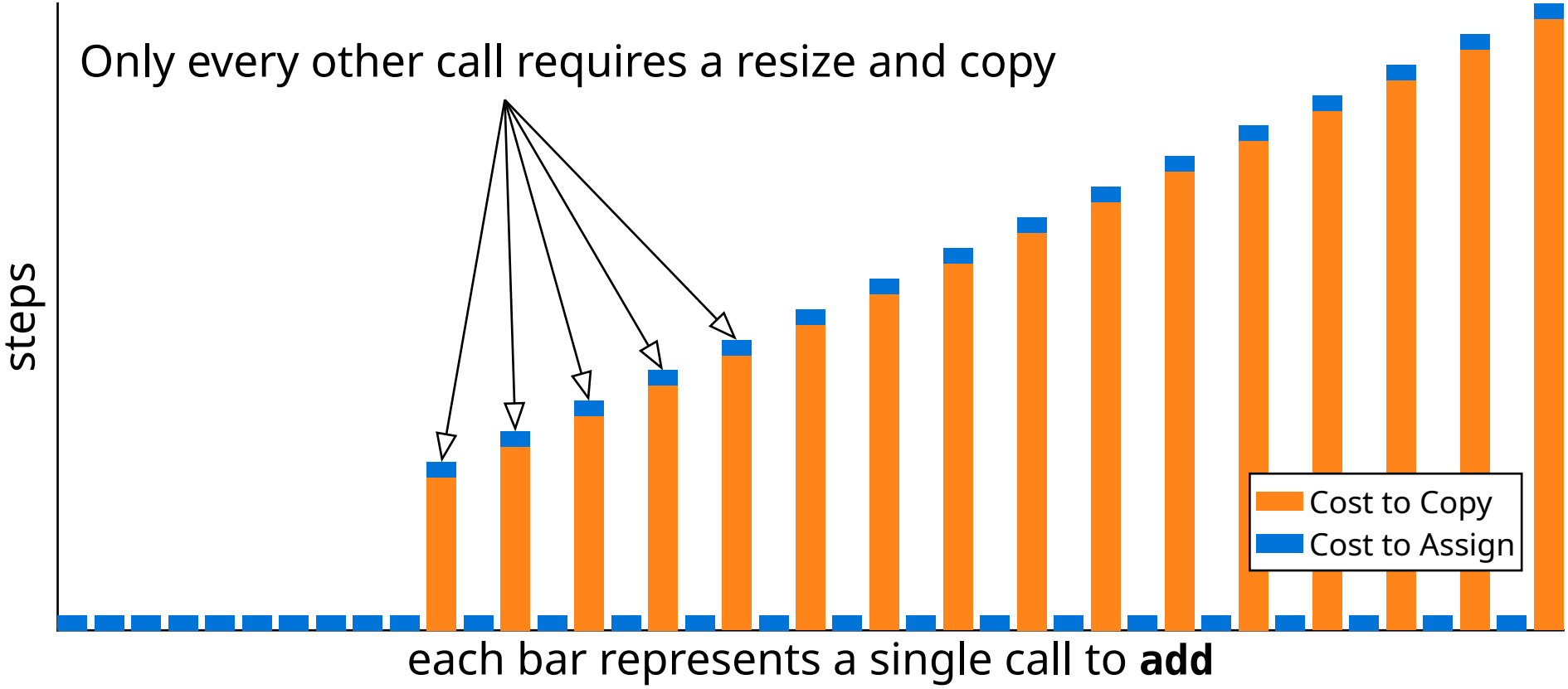
`ArrayList.add(elem)`: +2 Each Resize

Initial Size = 10, **`newLength = data.length + 2`**



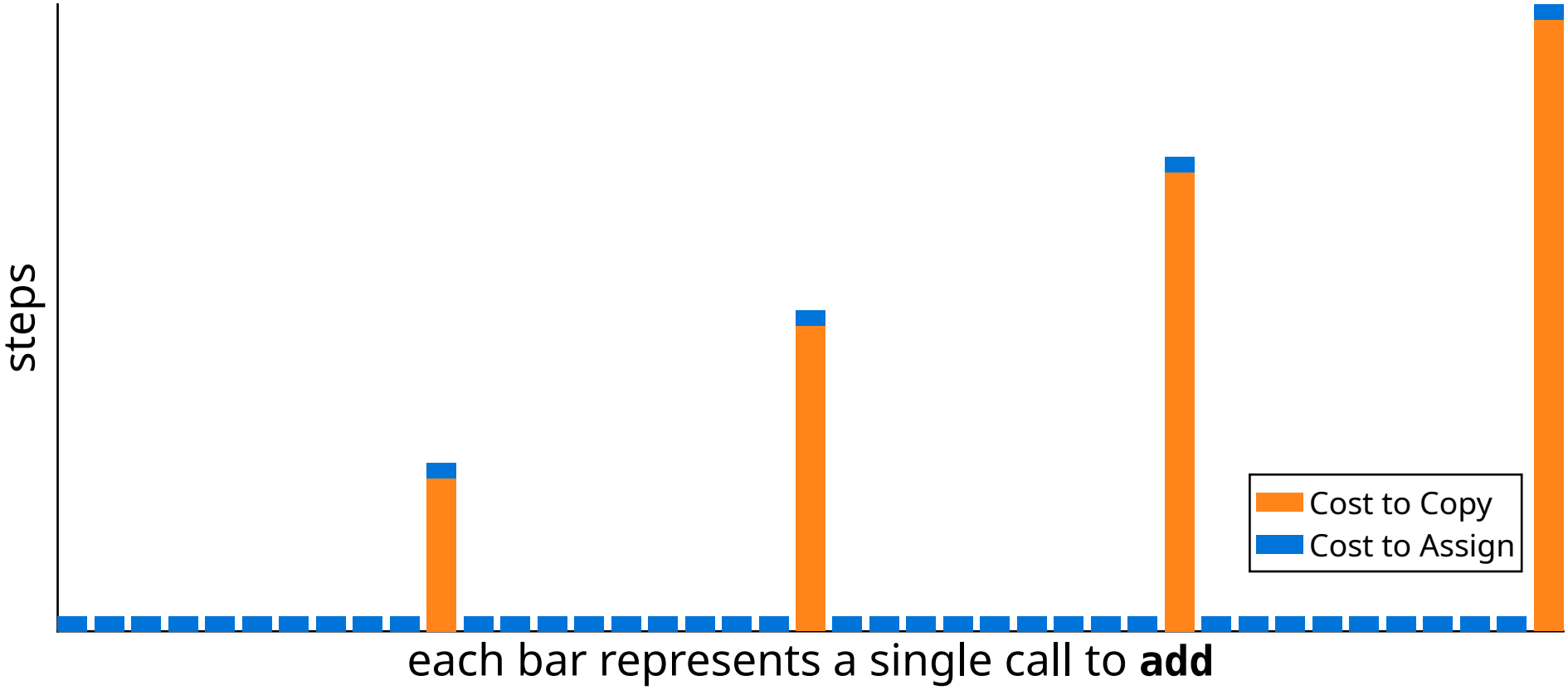
`ArrayList.add(elem)`: +2 Each Resize

Initial Size = 10, **`newLength = data.length + 2`**



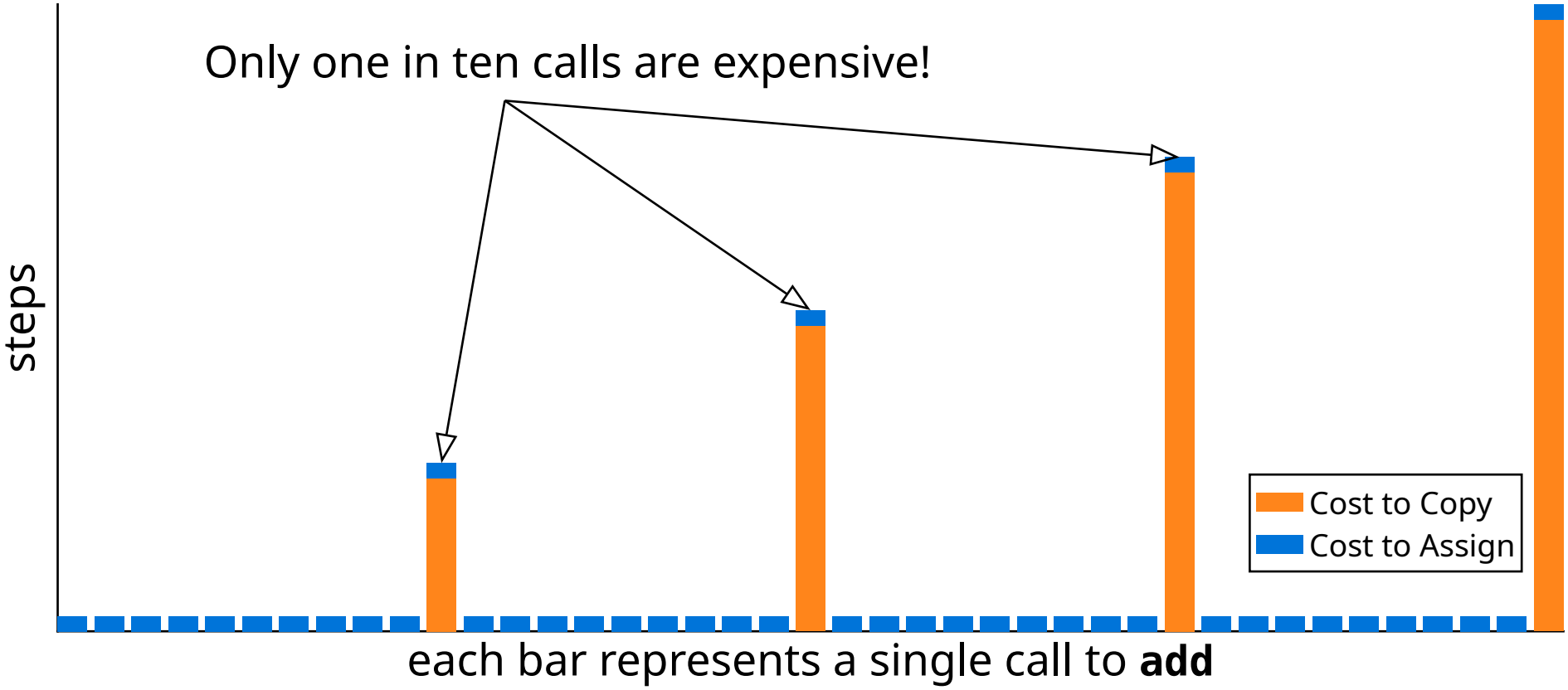
`ArrayList.add(elem)`: +10 Each Resize

Initial Size = 10, **`newLength = data.length + 10`**



`ArrayList.add(elem)`: +10 Each Resize

Initial Size = 10, **`newLength = data.length + 10`**



A Note on Runtime Complexity

So far, when discussing runtime bounds, we have done so without taking any extra information/context into account

For example:

- The worst-case runtime of **`ArrayList.add`** is $O(n)$
- Our analysis doesn't capture the fact that oftentimes it is faster than $O(n)$

A Note on Runtime Complexity

So far, when discussing runtime bounds, we have done so without taking any extra information/context into account

For example:

- The worst-case runtime of **`ArrayList.add`** is $O(n)$
- Our analysis doesn't capture the fact that oftentimes it is faster than $O(n)$

This type of analysis is referred to as unqualified analysis:

- Analyze a single run of the algorithm without any extra qualifications/context
- We would say the **unqualified runtime** of **`ArrayList.add`** is $O(n)$

A Note on Runtime Complexity

So far, when discussing runtime bounds, we have done so without taking any extra information/context into account

For example:

- The worst-case runtime of **`ArrayList.add`** is $O(n)$
- Our analysis doesn't capture the fact that oftentimes it is faster than $O(n)$

This type of analysis is referred to as unqualified analysis:

- Analyze a single run of the algorithm without any extra qualifications/context
- We would say the **unqualified runtime** of **`ArrayList.add`** is $O(n)$

But sometimes extra context is relevant...how can we include it in our analysis?

***Next Time**: Amortized Analysis*