CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu

Dr. Oliver Kennedy
okennedy@buffalo.edu

212 Capen Hall

Day 09
Sequences, Arrays, and Array Buffers
Textbook Ch. 6.4

Announcements

- PA1 due tonight!
- WA1 posted, due Wednesday, Sept 28
  - There was small transcription error, make sure to get the newest writeup

Recap

- **ADT:** Abstract Data Type, defines what a particular data structure can be used without specifying how it is implemented
  - ie: `Seq, mutable.Seq`
- **Array:** A type of sequence with a fixed element size and fixed number of elements, allocated as a contiguous block of memory
  - Immutable
  - Constant time random access (base + index * element size)
- **ArrayBuffer:** The mutable form of an array, allows insert and remove

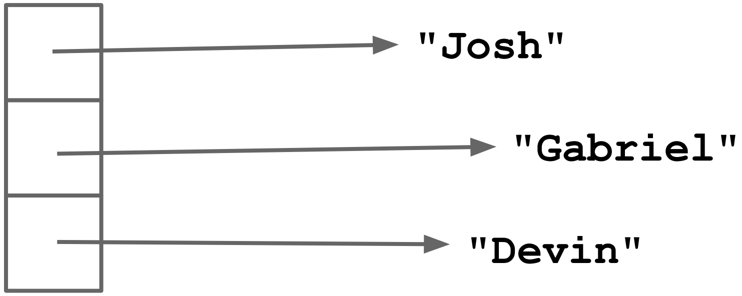
Arrays of Strings?

- We've already used **`Array[String]`** multiple times now…
  - But how does this work? Arrays have to have fixed size elements?

Arrays of Strings?

- We've already used **`Array[String]`** multiple times now...
  - But how does this work? Arrays have to have fixed size elements?
- **`String`** in Scala is a reference type! What we store is the address of the string, which is of a constant size.

Each element of the array is storing an address of a string (represented by an arrow). Addresses in Scala



Abstract Data Type vs Data Structure

ADT

The interface to a data structure

*Defines **what** the data structure can do*

Many data structures can implement the same ADT

Data Structure

The implementation of one (or more) ADTs

*Defines **how** the different tasks are carried out*

Different data structures will excel at different tasks

Types of Collections in Scala

Iterable - Any collection of items

Seq - A collection of items in a specific order

IndexedSeq - A Seq where there is guaranteed O(1) access to items

Set - A collection of unique items

Map - A collection of items identified by a key (associative collection)

Types of Sequences in Scala

mutable.Seq - Like Seq.....but mutable

mutable.Buffer - Like mutable.Seq, but "efficient" appends.

Queue - Like mutable.Seq but "efficient" append and remove first.

Think like a queue of people

Stack - Like mutable.Seq but "efficient" prepend and remove first.

Think like a stack of papers

The mutable.Seq ADT

`apply(idx: Int): [A]`
Get the element (of type **`A`**) at position **`idx`**

`iterator: Iterator[A]`
Get access to view all elements in the sequence, in order, once

`length: Int`
Count the number of elements in the seq

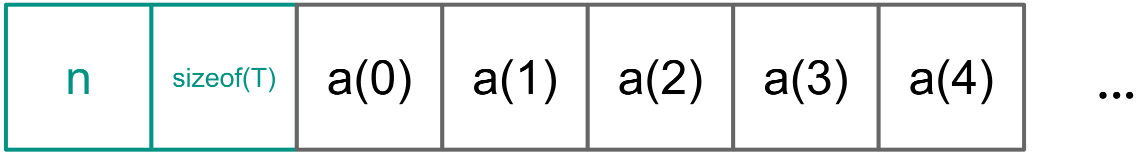
`insert(idx: Int, elem: A): Unit`
Insert an element at position **`idx`** with value **`elem`**

`remove(idx: Int): A`
Remove the element at position **`idx`**, and return the removed value

`Array[T]:Seq[T]`

What does an `Array` of *n* items of type `T` actually look like?

- 4 bytes for *n* (optional)
- 4 bytes for `sizeof(T)` (optional)
- *n* * `sizeof(T)` bytes for the data



Challenge: Operations that modify the array size require copying the array!

Challenge: Operations that modify the array size require copying the array!

Solution: Reserve extra space!

`ArrayBuffer[T]:Buffer[T](:Seq[T])`

What does an **`ArrayBuffer`** of *n* items of type **`T`** actually look like?

- 4 bytes for *n* (optional)
- 4 bytes for **`sizeof(T)`** (optional)
- 4 bytes for the number of **used** fields
- *n* * **`sizeof(T)`** bytes for the data

n	sizeof(T)	u	a(1) *or* None	a(2) *or* None	a(3) *or* None	a(4) *or* None	...

ArrayBuffer[T]:Buffer[T](:Seq[T])

```
class ArrayBuffer[T] extends Buffer[T] {
  var used = 0
  var data = Array[Option[T]].fill(INITIAL_SIZE) { None }

  def length = used

  def apply(i: Int): T = {
    if(i < 0 || i >= used){ throw new IndexOutOfBoundsException(i) }
      return data(i).get
    }

    /* ... */
  }
}
```

What is Option[T]...a brief digression

- Let's say we have a function that we know can possibly return `null`
- What can go wrong in the following code snippet?

```
val x = functionThatCanReturnNull()
x.frobulate()
```

What is Option[T]...a brief digression

- Let's say we have a function that we know can possibly return **`null`**
- What can go wrong in the following code snippet?

```
val x = functionThatCanReturnNull()
x.frobulate()
```

`java.lang.NullPointerException` (runtime error)

What is Option[T]...a brief digression

- Let's say we have a function that we know can possibly return `null`
- What can go wrong in the following code snippet?

```
val x = functionThatCanReturnNull()
if(x == null) { /* do something special */ }
else { x.frobulate() }
```

It's very easy in practice to miss doing this test!

What is Option[T]...a brief digression

- What if instead that function returns something called an Option?

```
val x = functionThatReturnsOption()
x.frobulate()
```

```
error: value frobulate is not a member of Option[MyClass]
```

What is Option[T]...a brief digression

- What if instead that function returns something called an Option?

```
val x = functionThatReturnsOption()
x.frobulate()
```

```
error: value frobulate is not a member of Option[MyClass]
```

Now it's a compile time error...Easier to catch

What is Option[T]...a brief digression

- But what is an Option (in Scala)?

Some(x)

Subclass of **`Option[T]`**

`value.isDefined == true`

A valid value exists and we can access it with **`value.get`**

None

Subclass of **`Option[T]`**

`value.isEmpty == true`

Analogous to **`null`**. No value.

Now back to ArrayBuffers…

ArrayBuffer.remove(i)

```
def remove(target: Int): T = {
    /* Sanity-check inputs */
    if(target < 0 || target >= used) {
        throw new IndexOutOfBoundsException(target)
    }
    /* Shift elements left */
    for(i <- target until (used-1)) {
        data(i) = data(i+1)
    }
    /* Update metadata */
    data(used-1) = None
    used -= 1
}
```

ArrayBuffer.remove(i)

```
def remove(target: Int): T = {
    /* Sanity-check inputs */
    if(target < 0 || target >= used) {
        throw new IndexOutOfBoundsException(target)
    }
    /* Shift elements left */
    for(i <- target until (used-1)) {
        data(i) = data(i+1)
    }
    /* Update metadata */
    data(used-1) = None
    used -= 1
}
```

What is the complexity?

ArrayBuffer.remove(i)

```
def remove(target: Int): T = {
    /* Sanity-check inputs */
    if(target < 0 || target >= used) {
        throw new IndexOutOfBoundsException(target)
    }
    /* Shift elements left */
    for(i <- target until (used-1)) {
        data(i) = data(i+1)
    }
    /* Update metadata */
    data(used-1) = None
    used -= 1
}
```

What is the complexity?

O(data.size)

or

Θ(used - target)

Analysis of `remove(i)`

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

Analysis of `remove(i)`

$$T_{remove}(n) = \begin{cases} 1 & \textbf{if } target = used - 1 \\ 2 & \textbf{if } target = used - 2 \\ 3 & \textbf{if } target = used - 3 \\ \cdots & \cdots \\ n - 1 & \textbf{if } target = 0 \end{cases}$$

$T_{remove}(n)$ is $O(n)$ and $\mathbf{\Omega}(1)$

ArrayBuffer.append(elem)

```
def append(elem: T): Unit = {
    if(used == data.size){ /* Sad case 🙁 */
        /* assume newLength > data.size, but pick it later */
        val newData = Array.copyOf(original = data, newLength = ???)
        /* Array.copyOf doesn't init elements, so we have to */
        for(i <- data.size until newData.size){ newData(i) = None }
    }
    /* Happy case 😃 */
    /* Append element, update data and metadata */
    newData(used) = Some(elem)
    data = newData
    used += 1
}
```

ArrayBuffer.append(elem)

```
def append(elem: T): Unit = {
    if(used == data.size){ /* Sad case 🙁 */
        /* assume newLength > data.size, but pick it later */
        val newData = Array.copyOf(original = data, newLength = ???)
        /* Array.copyOf doesn't init elements, so we have to */
        for(i <- data.size until newData.size){ newData(i) = None }
    }
    /* Happy case 😃 */
    /* Append element, update data and metadata */
    newData(used) = Some(elem)
    data = newData
    used += 1
}
```

What is the complexity?

…and what is newLength?

ArrayBuffer.append(elem)

```
def append(elem: T): Unit = {
    if(used == data.size){ /* Sad case 🙁 */
        /* assume newLength > data.size, but pick it later */
        val newData = Array.copyOf(original = data, newLength = ???)
        /* Array.copyOf doesn't init elements, so we have to */
        for(i <- data.size until newData.size){ newData(i) = None }
    }
    /* Happy case 😃 */
    /* Append element, update data and metadata */
    newData(used) = Some(elem)
    data = newData
    used += 1
}
```

What is the complexity?

O(`data.size`) (ie O(n)) ...but...

Analysis of `append(elem)`

$$T_{append}(n) = \begin{cases} n & \textbf{if } \texttt{used} = \texttt{n} \\ 1 & \textbf{otherwise} \end{cases}$$

Analysis of `append(elem)`

$$T_{append}(n) = \begin{cases} n & \textbf{if used} = \texttt{n} \\ 1 & \textbf{otherwise} \end{cases}$$

$T_{append}(n)$ is $O(n)$ and $\mathbf{\Omega}(1)$

Analysis of `append(elem)`

$$T_{append}(n) = \begin{cases} n & \textbf{if used} = \textbf{n} \\ 1 & \textbf{otherwise} \end{cases}$$

🙁 case

😃 case

$T_{append}(n)$ is $O(n)$ and $\Omega(1)$

How often do we hit the 🙁 case?

Analysis of `append(elem)`

$$T_{append}(n) = \begin{cases} n & \textbf{if used} = \textbf{n} \\ 1 & \textbf{otherwise} \end{cases}$$

🙁 case

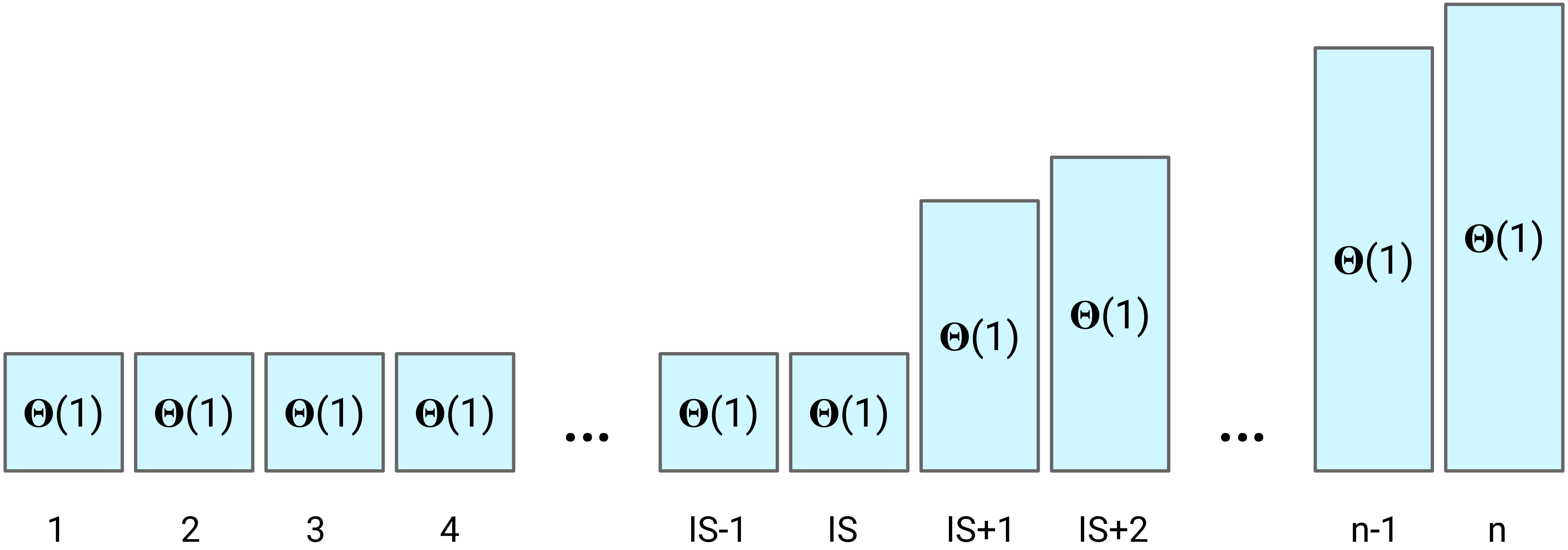
😃 case

$T_{append}(n)$ is $O(n)$ and $\boldsymbol{\Omega}(1)$

How often do we hit the 🙁 case?

Depends on **`newLength`**

newLength = data.size + 1



`newLength = data.size + 1`

For *n* appends into an empty buffer…

While `used <= Initial_Size`: $\sum_{i=0}^{\mathtt{IS}} \Theta(1)$

And after: $\sum_{i=\mathtt{IS}+1}^{n} \Theta(i)$

`newLength = data.size + 1`

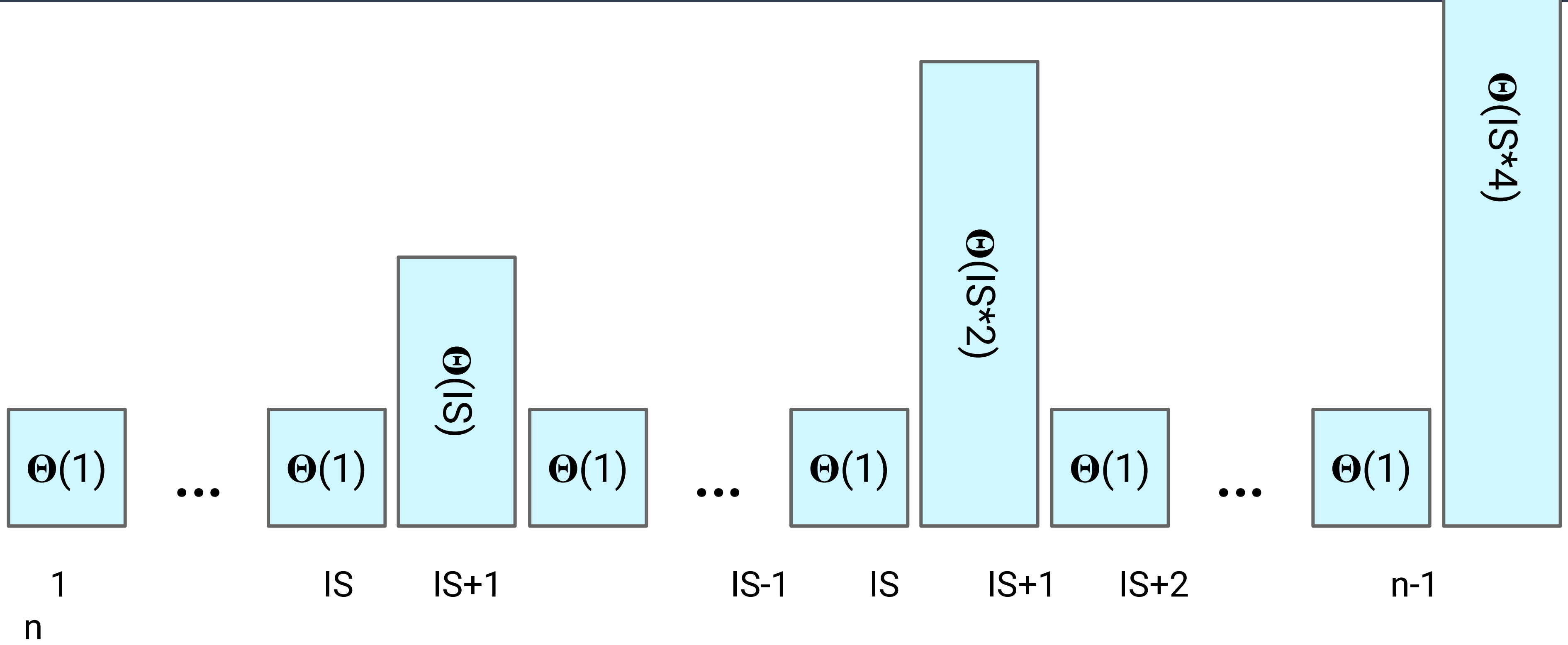
For *n* appends into an empty buffer…

While **`used <= Initial_Size`**: $\sum_{i=0}^{\texttt{IS}} \Theta(1)$

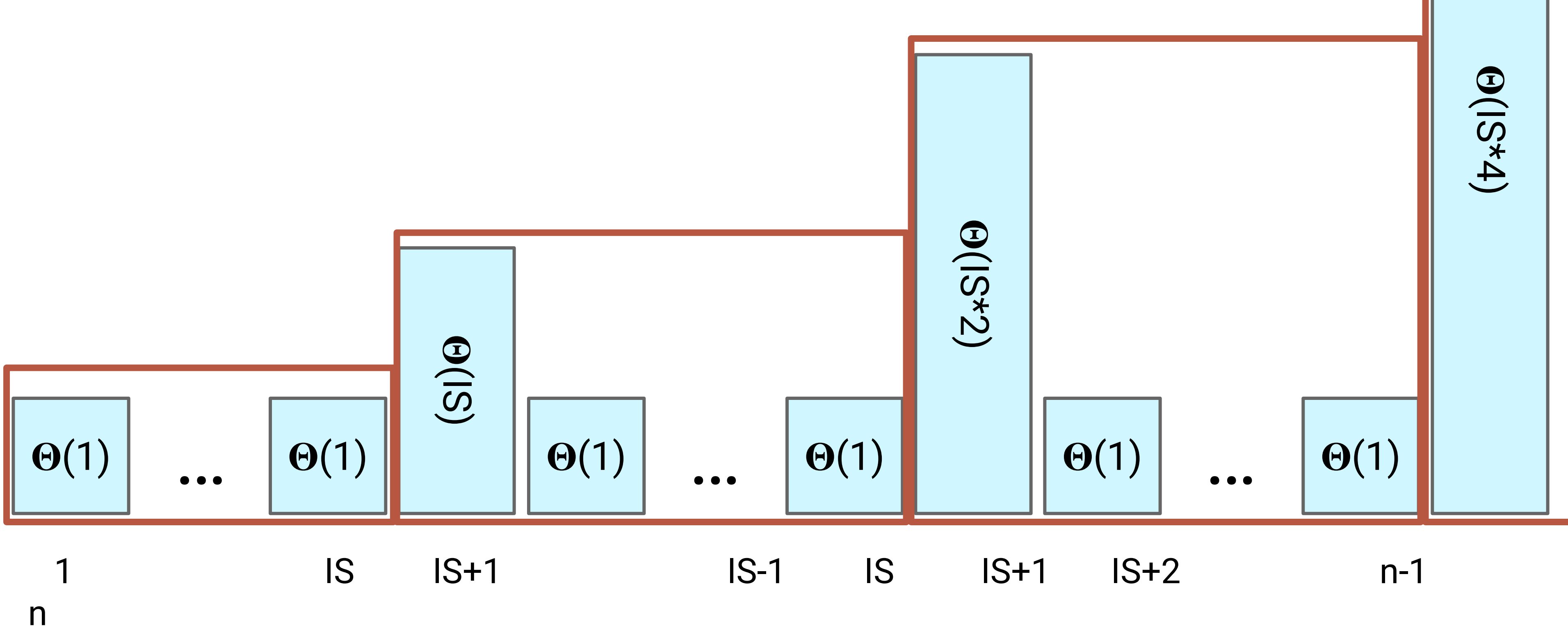
And after: $\sum_{i=\texttt{IS}+1}^{n} \Theta(i)$

Total: $\boldsymbol{\Theta}(n^2)$

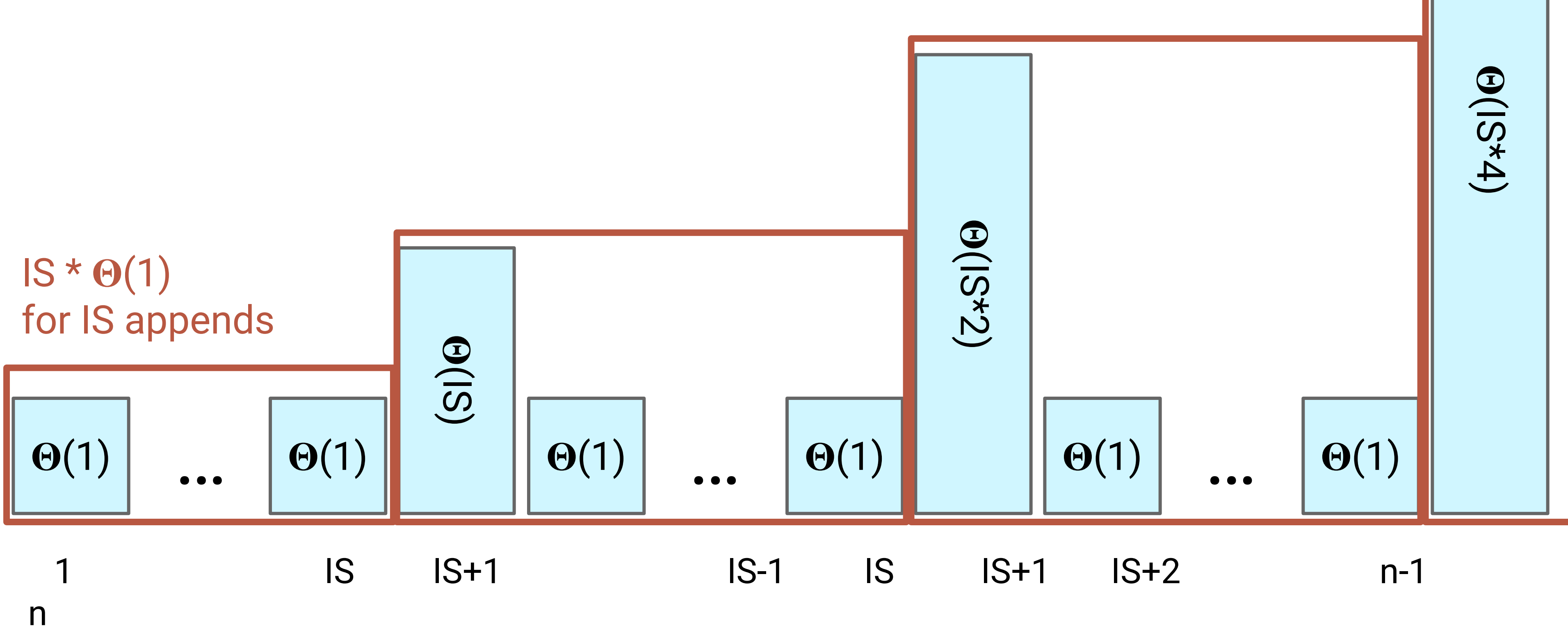
`newLength = data.size * 2`



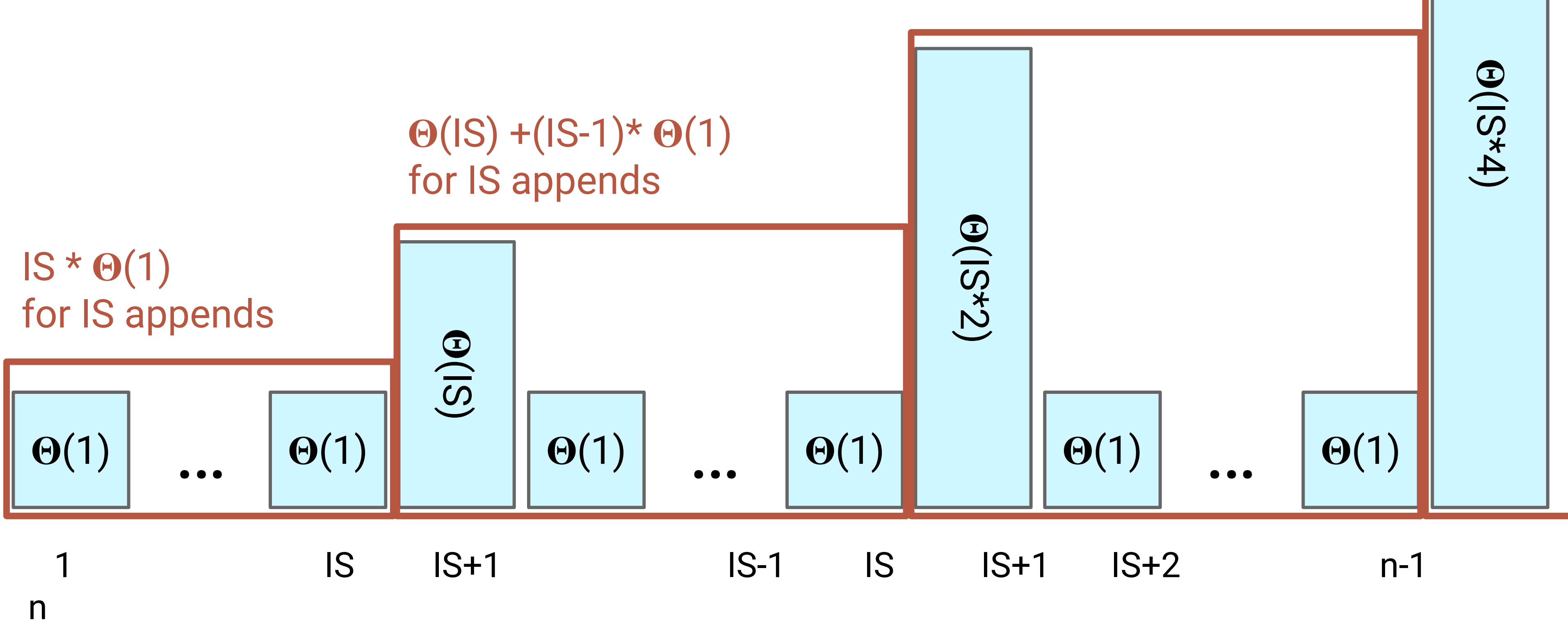
newLength = data.size * 2



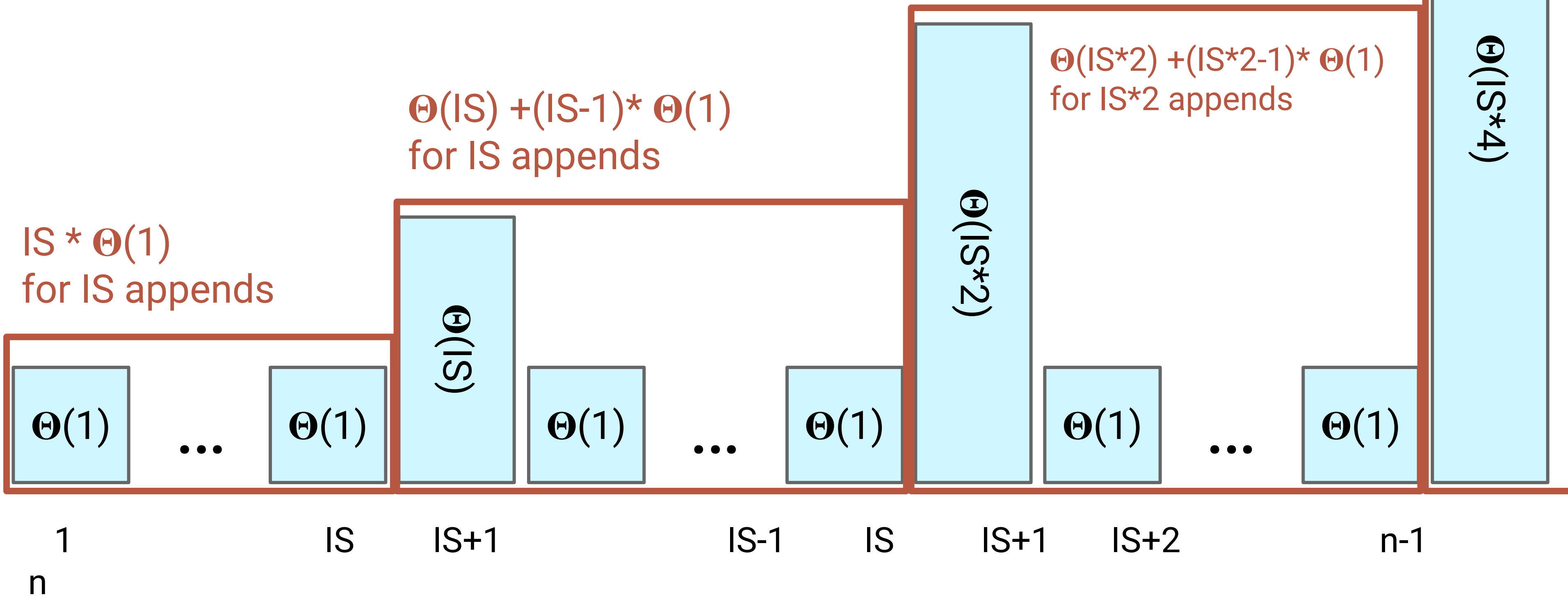
newLength = data.size * 2



```
newLength = data.size * 2
```



newLength = data.size * 2



`newLength = data.size * 2`

So…how many red boxes for *n* inserts?

`newLength = data.size * 2`

So…how many red boxes for *n* inserts? $\Theta(\log(n))$

newLength = data.size * 2

So…how many red boxes for *n* inserts? $\Theta(\log(n))$

How much work for box *j*?

`newLength = data.size * 2`

So…how many red boxes for *n* inserts? $\Theta(\log(n))$

How much work for box *j*? $\Theta(\texttt{IS} \cdot 2^j) + \sum_{1}^{\texttt{IS}\cdot 2^j} \Theta(1)$

`newLength = data.size * 2`

So…how many red boxes for *n* inserts? $\Theta(\log(n))$

How much work for box *j*? $\Theta(\texttt{IS} \cdot 2^j) + \sum_{1}^{\texttt{IS} \cdot 2^j} \Theta(1) = \Theta(2^j)$

newLength = data.size * 2

So…how many red boxes for *n* inserts? $\Theta(\log(n))$

How much work for box *j*? $\Theta(\texttt{IS} \cdot 2^j) + \sum_{1}^{\texttt{IS} \cdot 2^j} \Theta(1) = \Theta(2^j)$

How much work for *n* inserts?

`newLength = data.size * 2`

So…how many red boxes for *n* inserts? $\Theta(\log(n))$

How much work for box *j*? $\Theta(\texttt{IS} \cdot 2^j) + \sum_{1}^{\texttt{IS}\cdot 2^j} \Theta(1) = \Theta(2^j)$

How much work for *n* inserts? $\sum_{j=0}^{\Theta(\log(n))} \Theta(2^j)$

`newLength = data.size * 2`

So…how many red boxes for *n* inserts? $\Theta(\log(n))$

How much work for box *j*? $\Theta(\texttt{IS} \cdot 2^j) + \sum_{1}^{\texttt{IS} \cdot 2^j} \Theta(1) = \Theta(2^j)$

How much work for *n* inserts? $\sum_{j=0}^{\Theta(\log(n))} \Theta(2^j)$

Total for n insertions: $\Theta(n)$

Amortized Runtime

`append(elem)` is $O(n)$

n calls to **`append(elem)`** are $O(n)$

Amortized Runtime

`append(elem)` is $O(n)$

n calls to **`append(elem)`** are $O(n)$

The cost of n calls is <u>guaranteed</u> to be $O(n)$.

Amortized Runtime

If n calls to a function take $O(T(n))$...

We say the **Amortized Runtime** is $O(T(n) / n)$

e.g. the amortized runtime of **`append`** on an **`ArrayBuffer`** is:

$O(n/n) = O(1)$