# Collection Performance

**Simon Robinson**
SOFTWARE DEVELOPER

@techiesimon   www.simonrobinson.com

# Overview

**Scalability for large collections is important**

- Measuring scalability
- Big O Notation

**TourBooker course demo**

- Searching and sorting lists
  - With an eye on performance

# Performance Principles

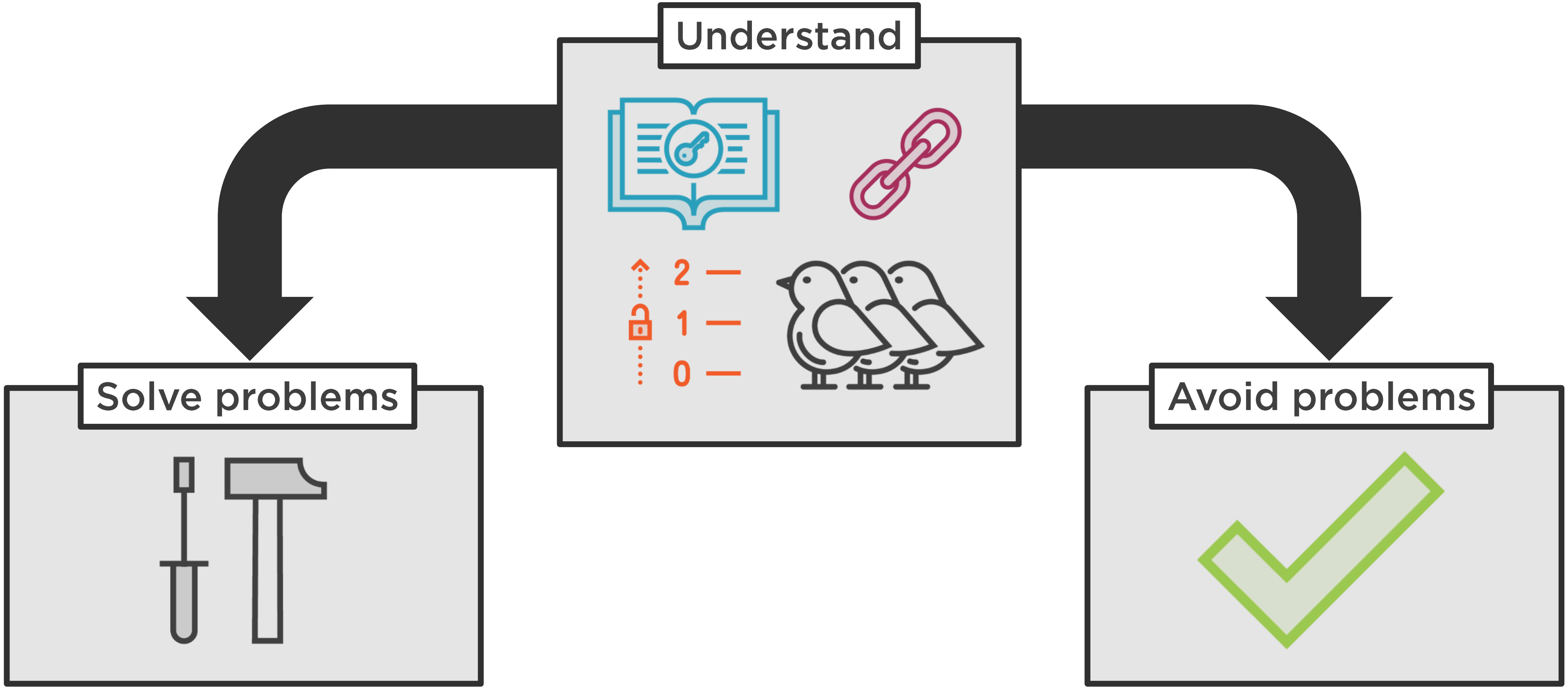
Modern processors
are fast

Write code
for robustness

Optimize
for performance
**only if**
you have issues

But for collections...?

# Collection Performance Characteristics



Understand

Solve problems

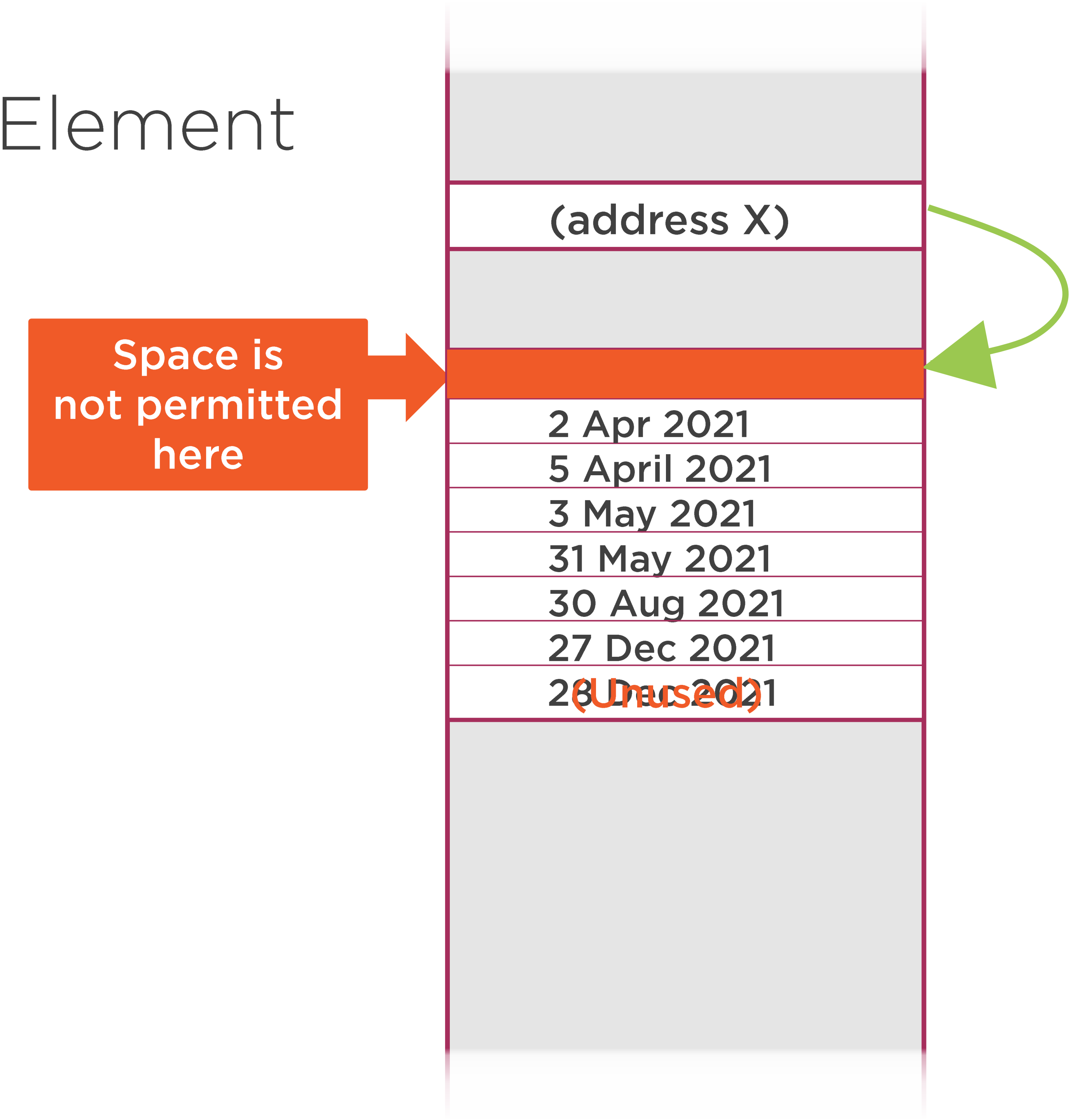
Avoid problems

# Removing a List Element

```
bankHolsLst.RemoveAt(0);
```

Remove first item....

...move almost the entire list



Space is not permitted here

(address X)

2 Apr 2021

5 April 2021

3 May 2021

31 May 2021

30 Aug 2021

27 Dec 2021

28 Dec 2021

(Unused)

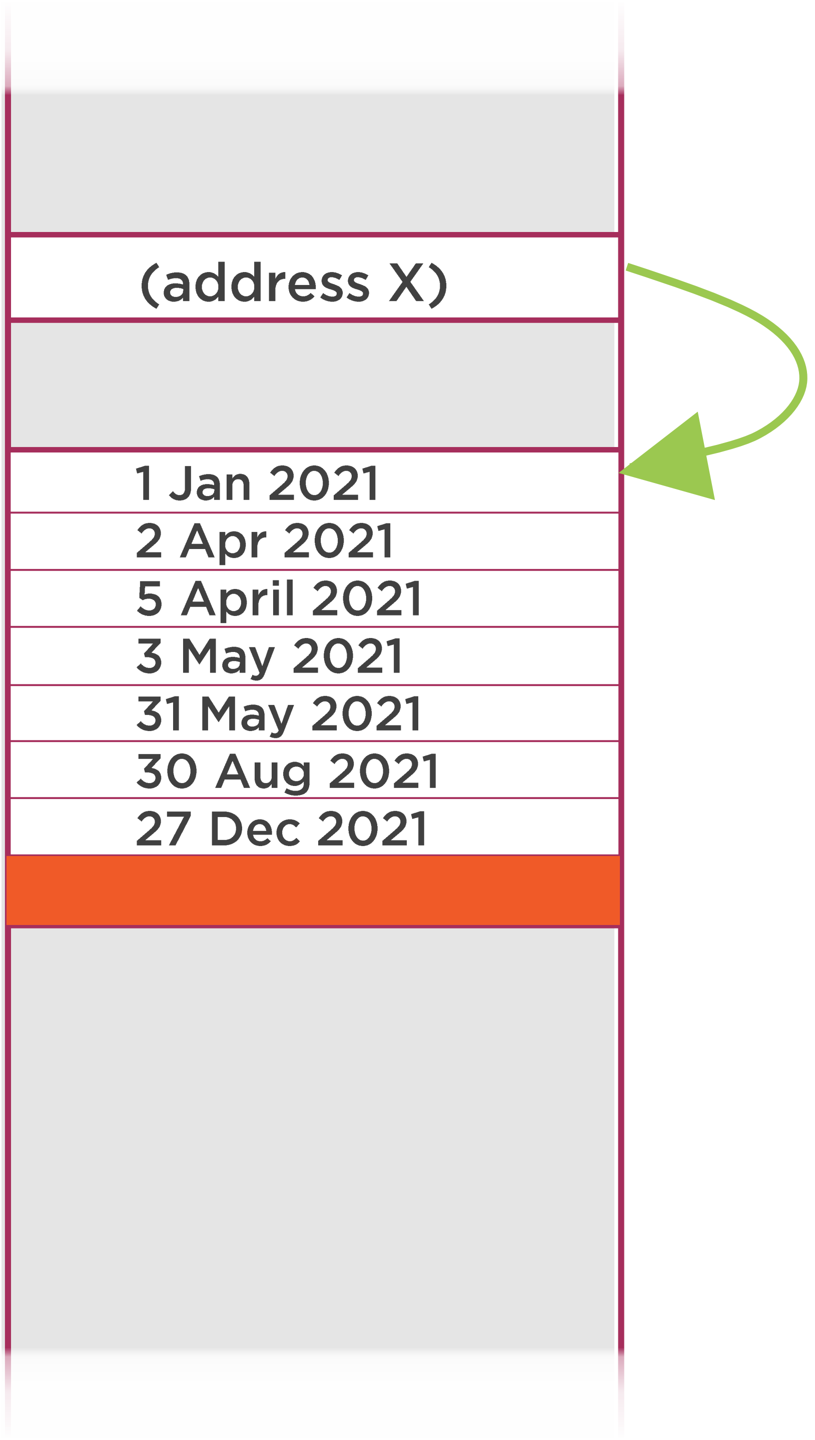
# Removing a List Element

```
bankHols1.RemoveAt(7);
```

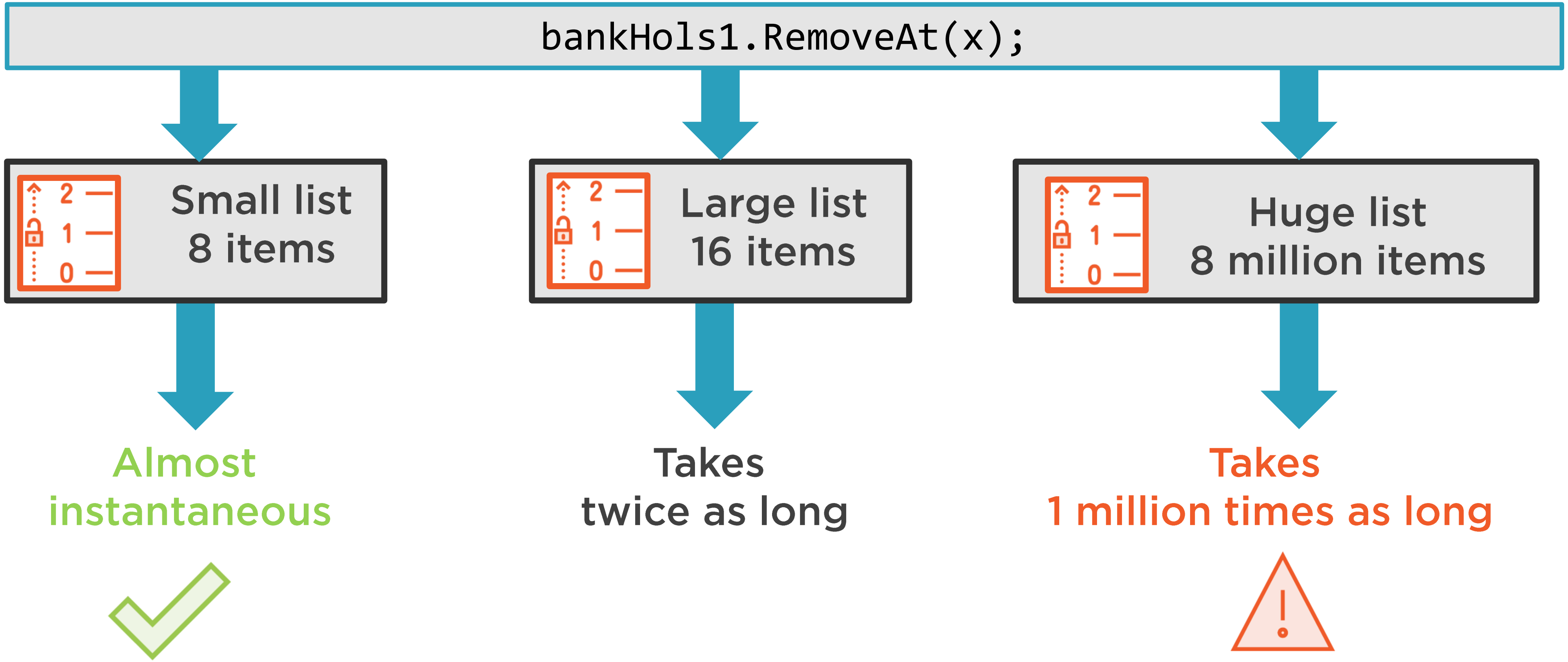
Remove last item....

...nothing moves

On average, half the list must be moved



# Removing a List Item



```
bankHols1.RemoveAt(x);
```

Small list
8 items

Large list
16 items

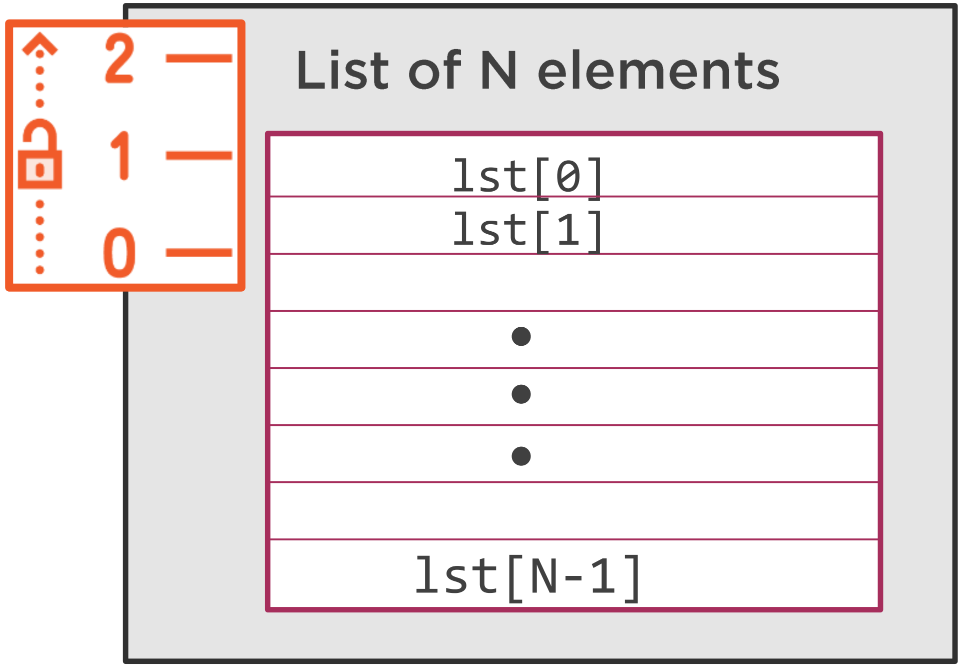
Huge list
8 million items

Almost instantaneous

Takes twice as long

Takes 1 million times as long

# Removing a List Item



On average,
time to remove an item = (something) times N

We write this as

O(N)

(‘O’ stands for ‘order of’)

# Big O Notation

Tells you how collection performance scales as a collection gets bigger

(It doesn’t tell you about absolute performance)

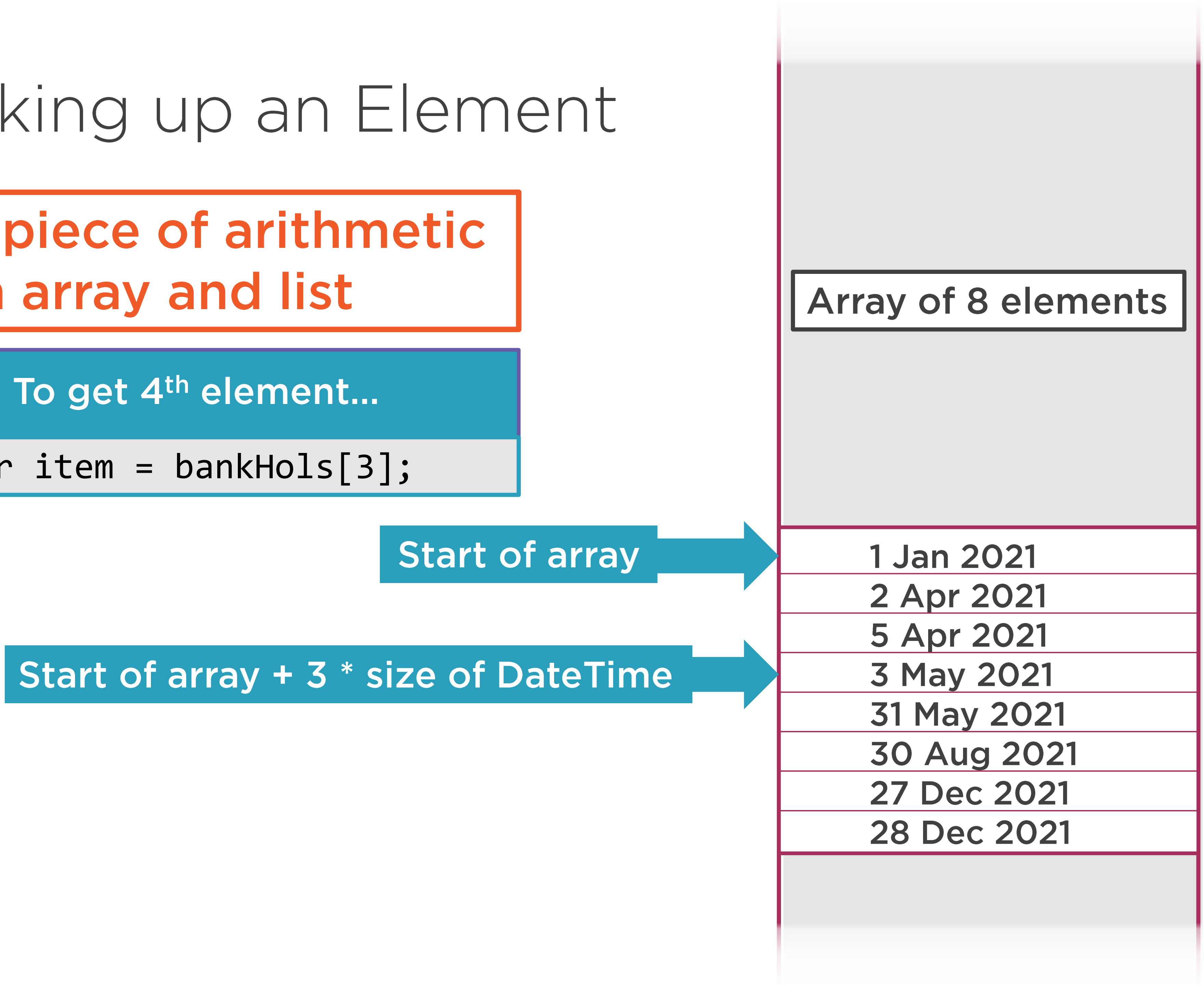
For collections – it’s scalability that matters!

# Looking up an Element

**Simple piece of arithmetic in array and list**

To get 4th element…

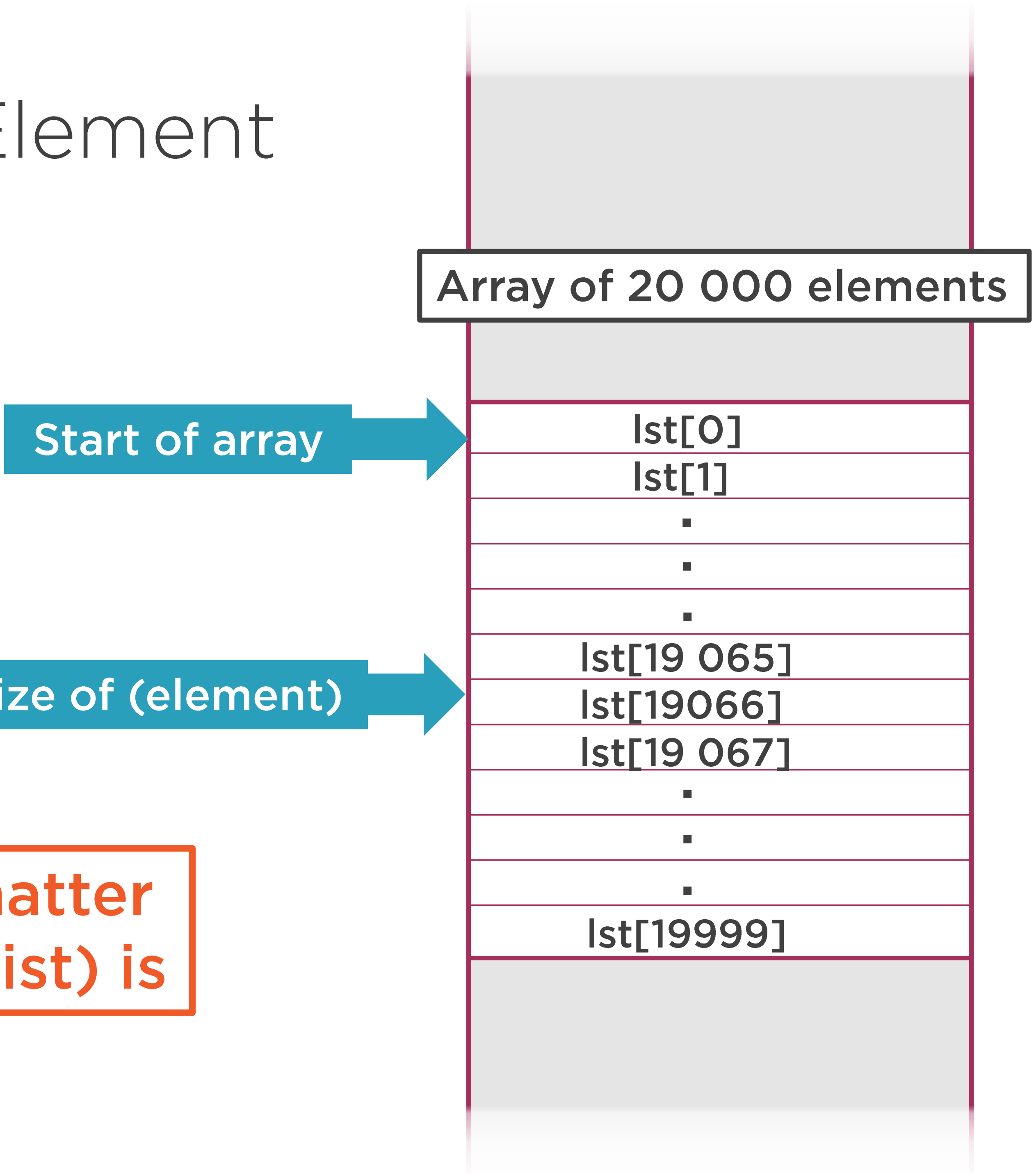
```
var item = bankHols[3];
```



# Looking up an Element

To get 19067th element…

```
var item = bankHols[19066];
```



Array of 20 000 elements

Start of array

Start of array + 19 066 * size of (element)

lst[0]

lst[1]

.

.

.

lst[19 065]

lst[19066]

lst[19 067]

.

.

.

lst[19999]

**Same calculation no matter how big the array (or list) is**

# O(1) Operation

Time taken is the same, no matter how big the collection is

# Making an $O(N^2)$ Operation

```
// lst is of type List<Something>
for (int i=lst.Count-1; i>= 0; i--)
{
    if (someExpression(bankHols[i]))
        lst.RemoveAt(i);
}
```

This O(N) operation might be executed up to N times

RemoveAt() is O(N)

This entire loop could be $O(N^2)$

Beware of putting O(N) operations inside loops!

# Making an $O(N^2)$ Operation

```
// lst is of type List<Something>
for (int i=lst.Count-1; i>= 0; i--)
{
    if (someExpression(bankHols[i]))
        lst.RemoveAt(i);
}
```

RemoveAt() is O(N)

Therefore this entire loop could be $O(N^2)$

```
// lst is of type List<Something>
lst.RemoveAll(x=>someExpression(x));
```

RemoveAll() is O(N)

Entire operation is O(N)

# Review of Collection Scaling

| | | |
|---|---|---|
| O(1) | Same time for any size of collection | Look up item in array or list |
| O(log N) | Almost as good as O(1) | |
| O(N) | Scales as size of collection | Remove item from list<br>Enumerate most collections |
| O(N log N) | Almost as good as O(N) | |
| $O(N^2)$ | Very slow for large collections | (Rare in .NET Framework)<br>Put O(N) operation in a loop |

# Performance – Key Takeaways

Check documented scalability of collection methods

Be wary of O(N) operations in tight loops

# The TourBooker Demo App

# Demo

**TourBooker will eventually support:**

- Create, view and book tours
- Many different collections

**But for now, select a country:**

- Collections will be too small to worry about performance
- But for practice, will still keep an eye on scalability

# To Sort a List

## Using `List.Sort()`

Slightly quicker

More awkward to code

## Using LINQ

LINQ has more overhead than collection methods

But performance doesn’t matter as much if only doing once

# Summary

## Performance

- Measured by how well operations scale
- O(1) and O(log N) are ideal
- Take care calling O(N) repeatedly

## TourBooker demo code

- Searching and sorting performance

**Next up: Dictionaries**