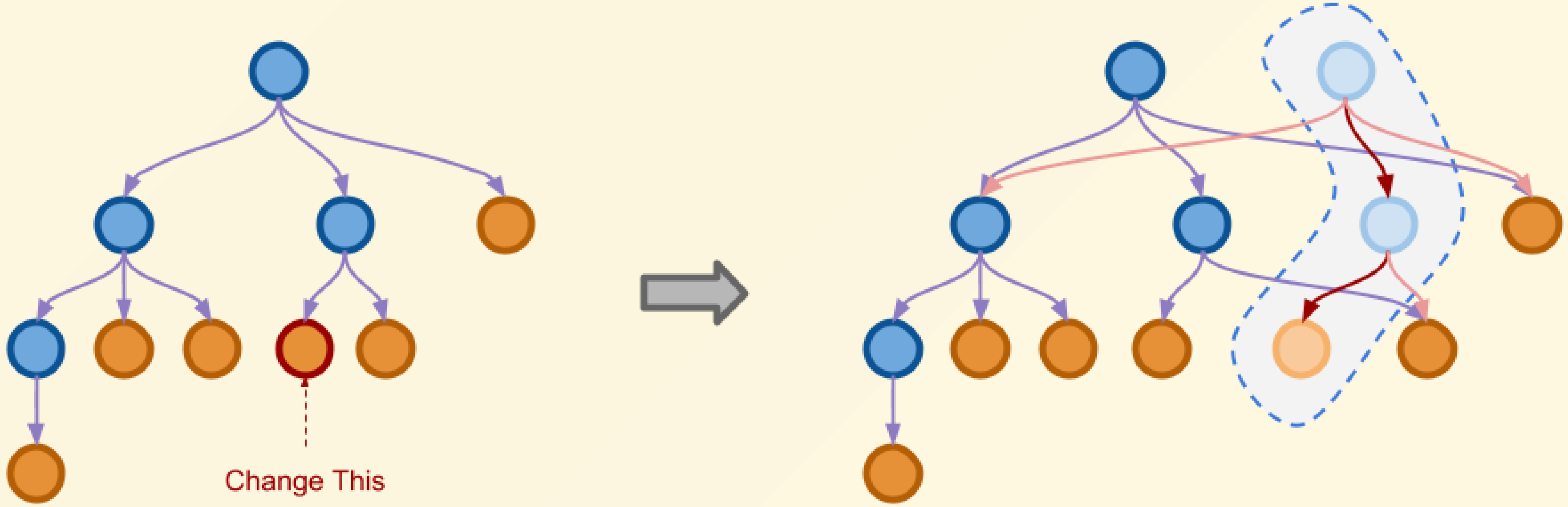
F# Data Structures



In this talk

- Immutable Data Structures - why, how, Structural sharing
- F# List
- F# Map
- F# Set
- Structural comparison
- Comparison with C# collections
- IEnumerable, seq - lazy sequences
- referential transparency
- ImmutableCollections

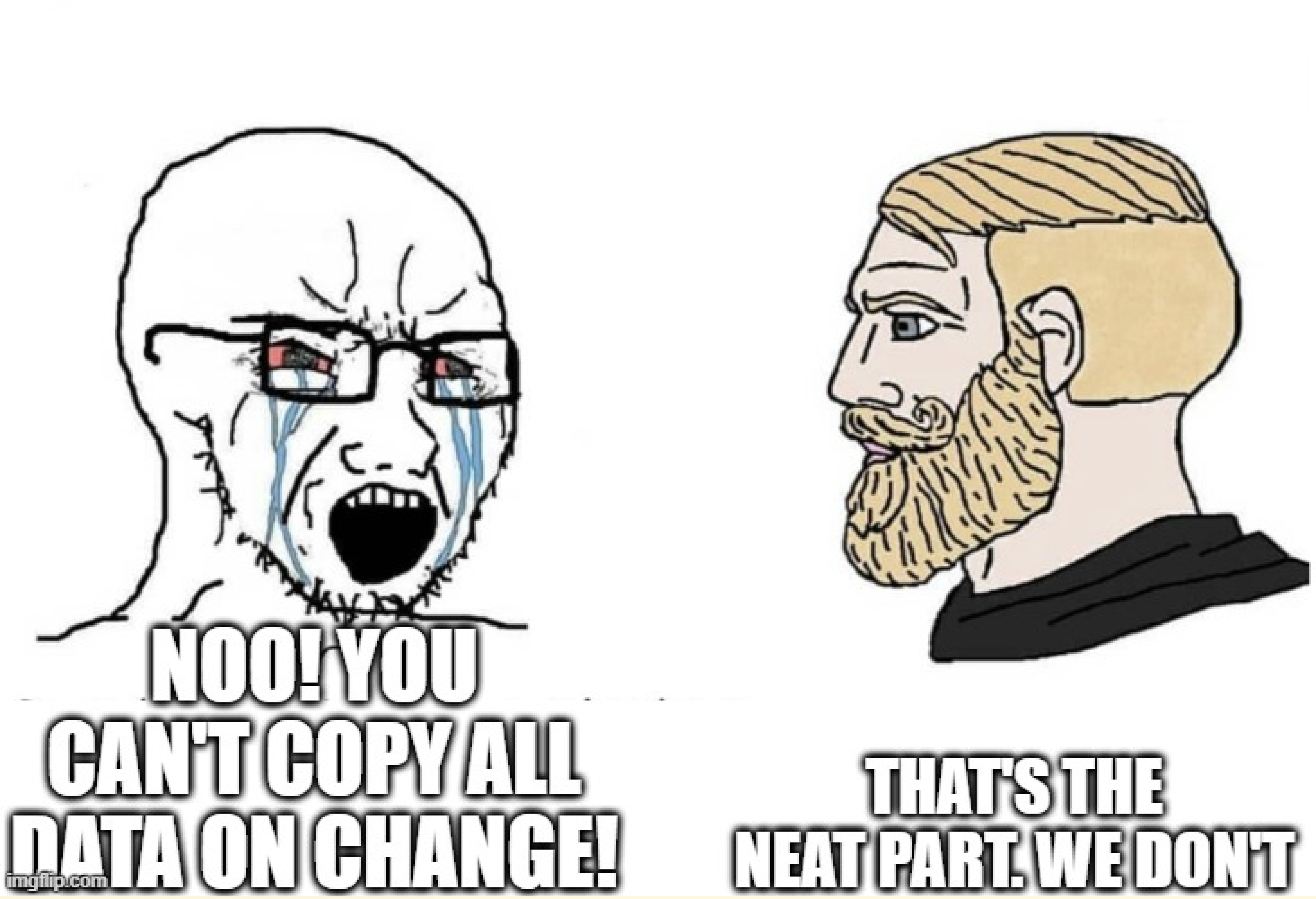
Immutable Data Structures

- no part of object can be changed after it's created

Why?

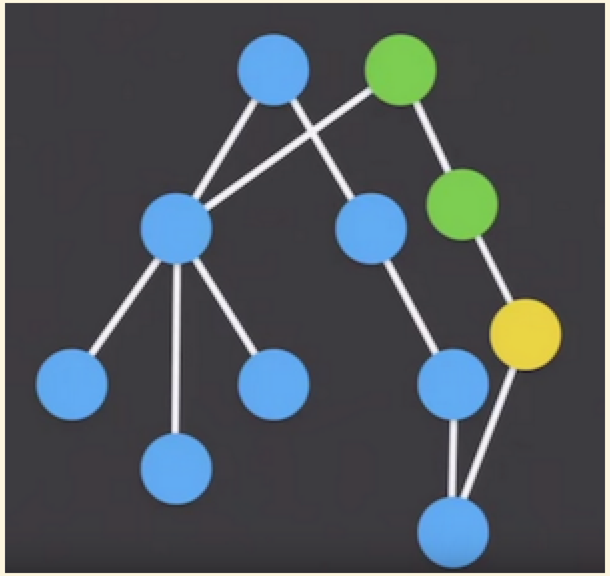
- mutation is common source of bugs
- immutable data structures are easier to reason about
    - value passed to a function, can't be changed
- immutable data structures are thread-safe
- bonus: memory efficient time travelling

MYTH: to create new immutable value, you need to copy the whole thing



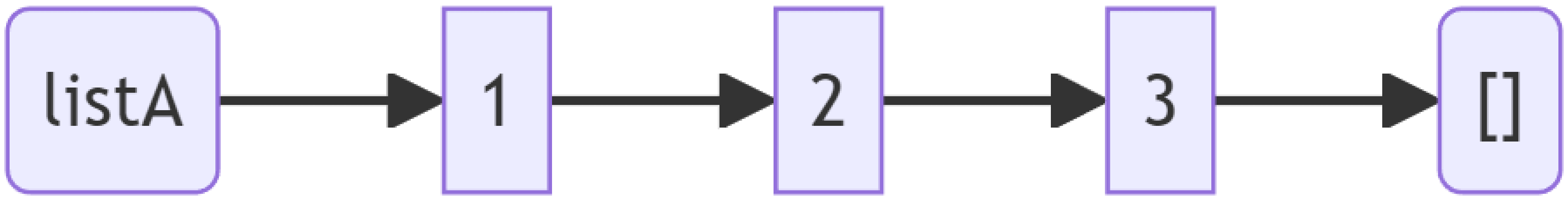
How?

- we can share parts of the structure between old and new value
- **Structural sharing**



F# (Linked) list

```
let listA = [1; 2; 3]
let listA = 1 :: 2 :: 3 :: []
```



F# list type definition

```
type List<'T> =
| ([]) : 'T list
| ( :: ) : Head: 'T * Tail: 'T list -> 'T list
```

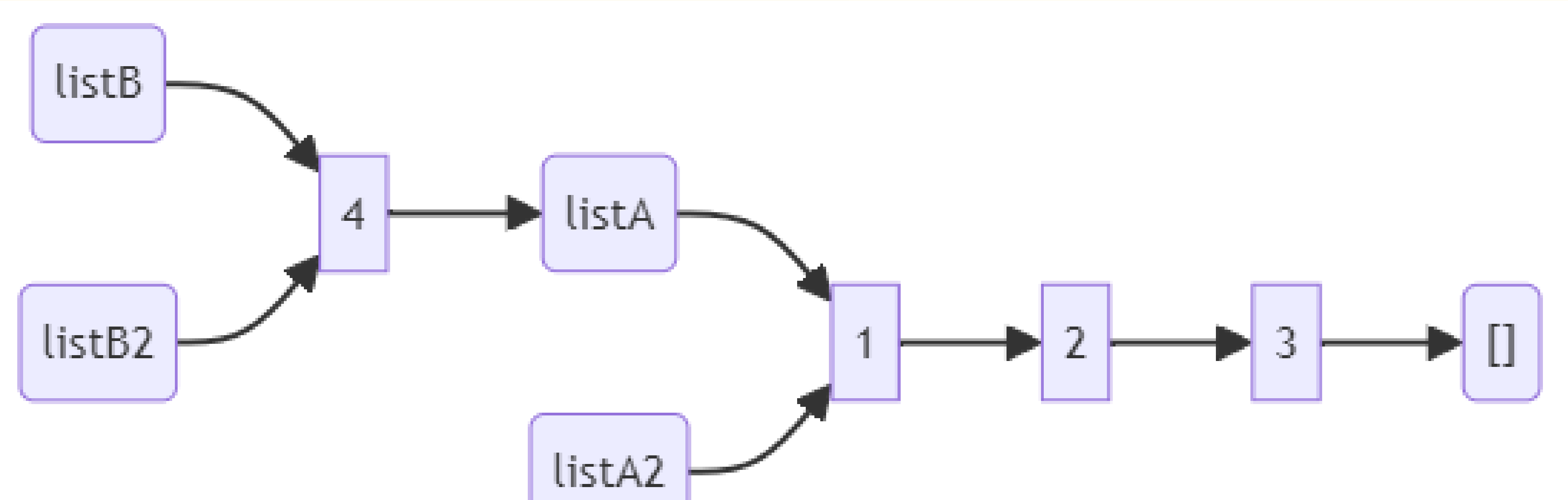
equivalently

```
type List<'T> =
| Nil : 'T list
| Cons : Head: 'T * Tail: 'T list -> 'T list

let listA = Cons(1, Cons(2, Cons(3, Nil)))
```

F# (Linked) list

```
let listA = [1; 2; 3]
let listA = 1 :: 2 :: 3 :: []
let listA2 = listA
let listB = 4 :: listA
let listB2 = [4] @ listA
```



F# (Linked) list

- fast iteration, mapping, filtering, append to start
- slow indexing, append on end
- `x :: xs` super fast
- `xs @ ys` slow

```
[<Benchmark>]
member _.ListAddToEnd() =
    let rec go i acc =
        if i = 0 then acc
        else go (i - 1) (acc @ [i])
    go size []

[<Benchmark>]
member _.ListAddToEndAcc() =
    let rec go i acc =
        if i = 0 then acc
        else go (i - 1) (i :: acc)
    go size [] |> List.rev
```

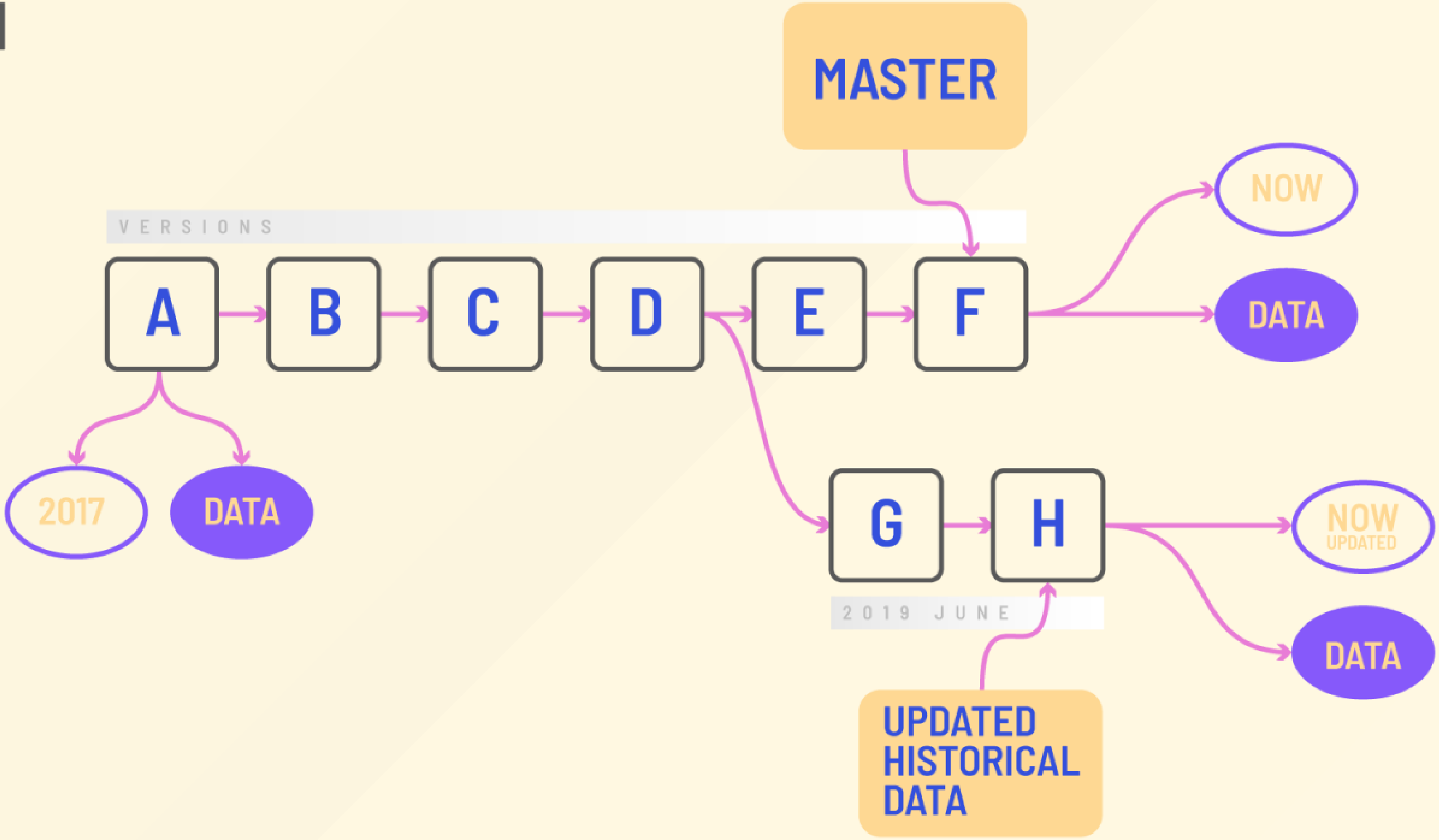
Method	Mean	Error	StdDev
ListAddToEnd	5,178.36 us	102.125 us	139.790 us
ListAddToEndAcc	15.99 us	0.308 us	0.303 us

- List.rev is fast!

search, indexing

- `List.find`, `List.nth` goes through list one by one
- `Set` is better for searching in big lists
- if you really need indexing, use array

COMMIT GRAPH

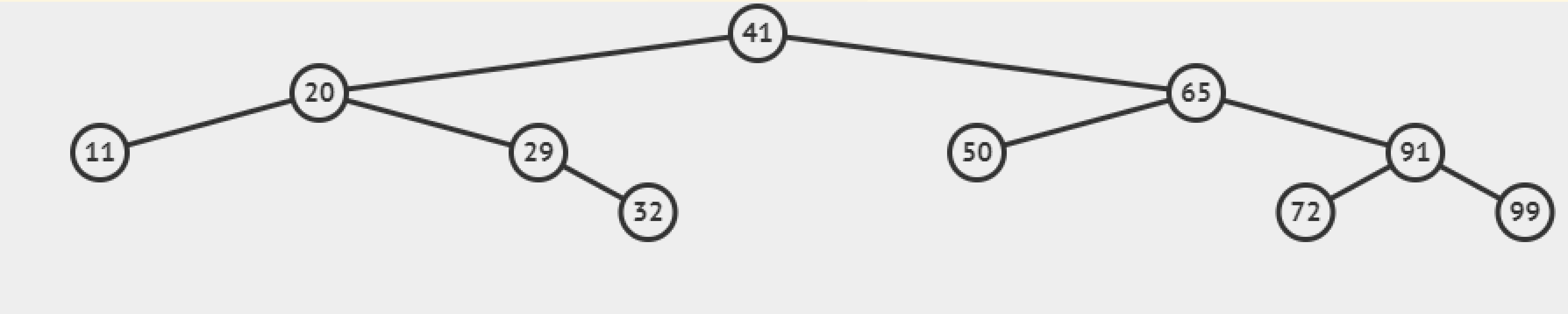


F# Set

Unordered set of values

Internally implemented as a (balanced) tree

```
let s = [11; 20; 29; 32; 41; 50; 65; 72; 91; 99] |> set
```



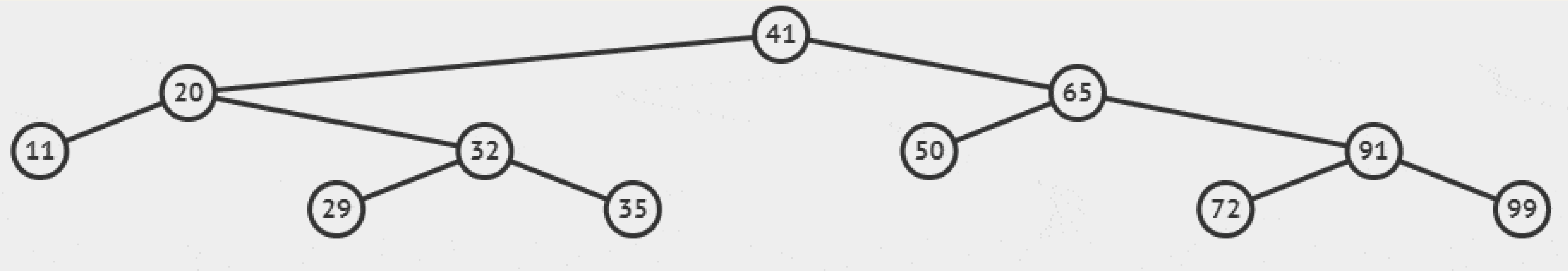
```
(* A classic functional language implementation of binary trees *)

[<CompilationRepresentation(CompilationRepresentationFlags.UseNullAsTrueValue)>]
[<NoEquality; NoComparison>]
type SetTree<'T> when 'T: comparison =
    | SetEmpty                                          // height = 0
    | SetNode of 'T * SetTree<'T> *  SetTree<'T> * int    // height = int
    | SetOne  of 'T                                     // height = 1
```

```
SetNode(41, SetNode(20, SetOne(11), SetNode(29, SetEmpty, SetOne(32), 1), 2), SetNode(65, SetOne(50), SetNode(91, SetOne(72), SetOne(99), 1), 2), 3)
```

Insert = search + add

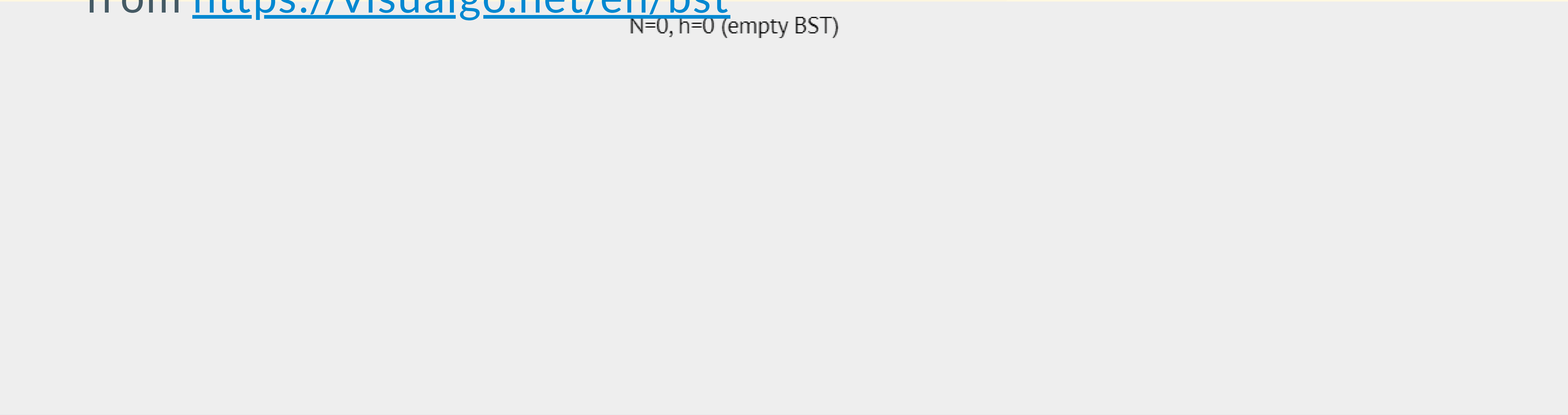
```
let s2 = s |> Set.add 35
```



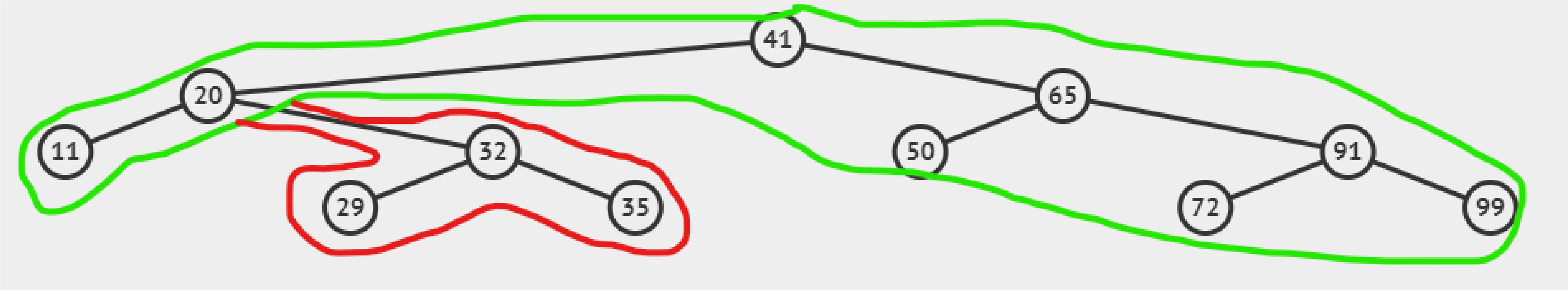
from https://visualgo.net/en/bst

```
let s = [1; 7; 3; 9; 5; 6; 2; 8; 4] |> set
```

from https://visualgo.net/en/bst



- values must be comparable
- searching for item (`Set.exists`, `Set.contains`) by binary search
- insert, remove - unchanged part of tree is shared



- functions with predicate on value (`Set.map`, `Set.filter`, `Set.partition`), goes through whole tree! (in order)
- keys cannot be duplicite - insert (`Set.add`) repace value if key already exists

When to use Set instead of List?

- generally its faster to search for item with `Set`
- but for small sizes `List.constains` is faster

When to use Set instead of List?

Method	Size	Mean	Error	StdDev
ListContains	**64**	**2.159 μs**	**0.0431 μs**	**0.0998 μs**
SetContains	64	4.561 μs	0.0833 μs	0.0780 μs
ListContains	**128**	**8.241 μs**	**0.0473 μs**	**0.0443 μs**
SetContains	128	10.347 μs	0.1933 μs	0.1985 μs
ListContains	**256**	**31.169 μs**	**0.1609 μs**	**0.1426 μs**
SetContains	256	23.488 μs	0.3803 μs	0.3557 μs
ListContains	**512**	**119.456 μs**	**0.5491 μs**	**0.5136 μs**
SetContains	512	52.889 μs	0.8146 μs	0.6802 μs
ListContains	**1024**	**467.593 μs**	**1.9139 μs**	**1.7902 μs**
SetContains	1024	149.908 μs	1.2287 μs	1.1494 μs
ListContains	**8192**	**29,487.104 μs**	**114.3813 μs**	**101.3960 μs**
SetContains	8192	1,548.127 μs	19.6668 μs	18.3963 μs

Another important functions

- `Set.union`
- `Set.intersect`
- `Set.difference`
- all of them work recursively on tree structure -> faster than the same on `list`
- `Set.isSubset`
- `Set.isSuperset`
- try to find all elements of first set in second

F# Map

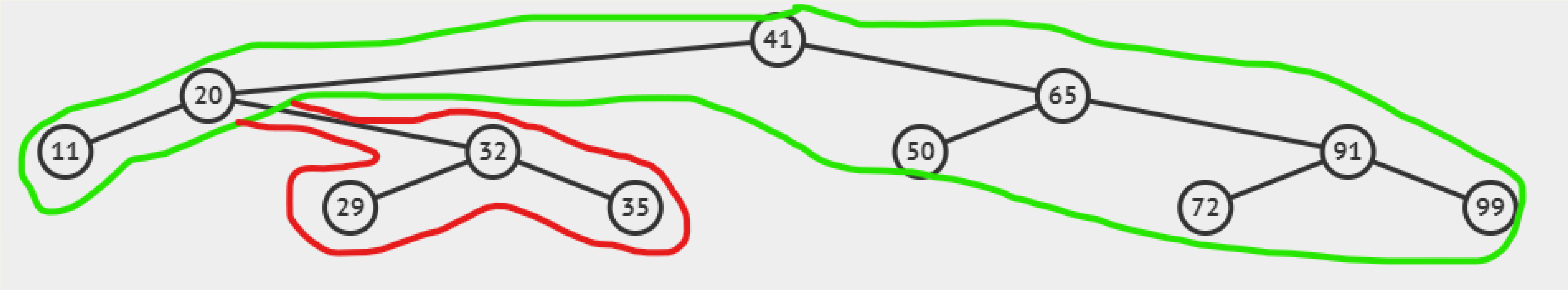
- Dictionary like immutable data structure
- Like `Set`, but with value linked with each key (node)

```fsharp
let mapA = Map.ofList [1, "A"; 2, "B"; 3, "C"]
let mapB = Map.ofList [1, "A"; 2, "B"; 3, "C"; 4, "D"]
let mapB2 = Map.add 4 "D" mapA
mapB = mapB2 // true
```

```
[<NoEquality; NoComparison>]
[<AllowNullLiteral>]
type internal MapTree<'Key, 'Value>(k: 'Key, v: 'Value, h: int) =
    member _.Height = h
    member _.Key = k
    member _.Value = v
    new(k: 'Key, v: 'Value) = MapTree(k, v, 1)

[<NoEquality; NoComparison>]
[<Sealed>]
[<AllowNullLiteral>]
type internal MapTreeNode<'Key, 'Value>
    (
        k: 'Key,
        v: 'Value,
        left: MapTree<'Key, 'Value>,
        right: MapTree<'Key, 'Value>,
        h: int
    ) =
    inherit MapTree<'Key, 'Value>(k, v, h)
    member _.Left = left
    member _.Right = right
```

- keys must be comparable
- searching for item (`Map.find` , `Map.containsKey`) by binary search
- insert, remove - unchanged part of tree is shared



- functions with predicate on key (`Map.pick` , `Map.findKey`), goes through whole tree! (in keys order)
- keys cannot be duplicite - insert (`Map.add`) replace value if key already exists

Creation of `Map` - List.groupBy

```
[1..1000] |> List.groupBy (fun x -> x % 100) |> Map.ofList
```

F# data types

- unit
- primitive types - `int`, `float`, `string`, `bool`, ...
- records
- tuples
- discriminated unions

composed types

- `list`
- `Set`

Ordering

Ordering by field/case position, then recurse or prim. type ordering

```
type R = {A: int; B: string}
{A = 1; B = "b"} < {A = 2; B = "a"}
{A = 1; B = "a"} = {A = 1; B = "a"}
{A = 1; B = "a"} < {A = 1; B = "b"}

type R2 = {B: string; A: int}
{B = "b"; A = 1} > {B = "a"; A = 2}
{B = "a"; A = 2} > {B = "a"; A = 1}

("a", 1) < ("a", 2)
```

```
//DU - by order of cases

Some 1 < Some 2
None < Some System.Int32.MaxValue
```

(Ab)use of ordering example

```
type PokerHand =
    | HighCard of int
    | Pair of int
    | TwoPair of int * int
    | ThreeOfAKind of int
    | Straight of int
    | Flush of int
    | FullHouse of int * int
    | FourOfAKind of int
    | StraightFlush of int
    | RoyalFlush
```

Comparison with C# collections

Collection	F#	C#
Linked list	`list<'T>`	`LinkedList<T>`
Resizeable array	`ResizeArray<'T>`	`List<T>`
Array	`array<'T>`, `'T[]`	`T[]`
Map (immutable dictionary)	`Map<'K, 'V>`	`ImmutableDictionary<K, V>`
Set (immutable set)	`Set<'T>`	`ImmutableHashSet<T>`
Dictionary (mutable)	-	`Dictionary<K, V>`
HashSet (mutable)	-	`HashSet<T>`
Enumerable	`seq<'T>`	`IEnumerable<T>`

Other useful C# collections

- Queue<T>
- PriorityQueue<T>
- ConcurrentDictionary<K, V>

Enumerable, seq - lazy sequences

- Every collection implements `seq<'T>` (alias for `IEnumerable<T>`) interface.
- Interface for reading elements one by one.
- Lazy abstraction - elements are computed on demand.

`seq<'t>`

```
xs |> Seq.map (fun x -> expensiveFun x) |> Seq.take 10 |> Seq.toList
```

Only first 10 elements are computed.

```
xs |> Seq.filter (...) |> Seq.map (fun x -> expensiveFun x) |> Seq.tryFind (...)
```

Only elements that pass the filter are computed.

`seq<'t>`

There are cases where using `Seq` can be faster than `List`.

Example: expensive filtering and then taking first *k* elements.

```
xs |> Seq.filter (fun x -> expensiveFun x) |> Seq.take k |> Seq.toList
```

Infinite sequences

Seq can be also used for generating (possible infinite) sequences.

```
let cycle xs =
    let arr = Array.ofSeq xs
    Seq.initInfinite (fun i -> arr.[i % arr.Length])
```

Or sequnce of random numbers:

```
let r = System.Random()
Seq.initInfinite (fun _ -> r.Next())
```

Referential transparency

- replace the function call with its result doesn't change meaning of the program
  - always returns the same result for the same input ("math-y" function)
- Immutable data structures allows us to write **Referential transparent** functions.
- no mutable variables / data structures, no side effects => **referential transparency**

- BUT:
- **referential transparency** can be achieved even with mutable data structures or side-effects
- mutable variables and data structures are perfectly fine when not leaking outside of function

```
[<CompiledName("Fold")>]
let fold<'T, 'State> folder (state: 'State) (list: 'T list) =
    match list with
    | [] -> state
    | _ ->
        let f = OptimizedClosures.FSharpFunc<_, _, _>.Adapt (folder)
        let mutable acc = state

        for x in list do
            acc <- f.Invoke(acc, x)

        acc
```

Memoize function:

```
let memoizeBy projection f =
    let cache = System.Collections.Concurrent.ConcurrentDictionary()
    fun x -> cache.GetOrAdd(projection x, lazy f x).Value
```

Pure functions

- **Pure** function:
  - always returns the same result for the same input (**referential transparency**)
  - no side effects
- no mutable variables / data structures, no side effects <=> **pure function**
- every **pure** function is **referential transparent**
- **pure function** is more strict, but can be checked by compiler - one of idea behind Haskell

C# Immutable collections

- Immutable collections are persistent data structures for C# from .NET 7
- `ImmutableList<T>` is indexable, represented as tree (similar to `Map<int, T>`)
- `ImmutableArray<T>` copying whole array on change (!)
- `ImmutableDictionary<K, V>` is similar to `Map<K, V>`
- `ImmutableStack<T>` is actually linked list - similar to `list<T>`
- `ImmutableQueue<T>` - no std. F# equivalent\

https://learn.microsoft.com/en-us/archive/msdn-magazine/2017/march/net-framework-immutable-collections

F# Data Structures

Method	Mean	Error	StdDev	Gen0	Gen1	Allocated
'int - List cons'	2.375 us	0.0473 us	0.1059 us	2.5482	0.4234	32000 B
'int - ImmutableList cons'	95.410 us	1.7462 us	1.6334 us	40.0391	9.6436	502896 B
'int - List.reverse'	2.511 us	0.0413 us	0.0606 us	2.5482	0.4234	32000 B
'int - ImmutableList.reverse'	71.121 us	0.6854 us	0.6411 us	3.7842	0.8545	48024 B
'int - List.map'	2.781 us	0.0543 us	0.0687 us	2.5482	0.5074	32000 B
'int - ImmutableList map by LINQ Select'	31.375 us	0.5986 us	0.7571 us	4.1504	0.9766	52200 B
'int - ImmutableList map by SetItem'	113.180 us	2.1415 us	2.4661 us	36.2549	-	455376 B
'int - ImmutableList map by Builder'	36.315 us	0.6762 us	0.6944 us	3.7842	1.0376	48072 B
'int - List.filter'	1.756 us	0.0350 us	0.0623 us	1.2741	0.1411	16000 B
'int - ImmutableList filter by LINQ Where'	13.979 us	0.2794 us	0.3825 us	2.2736	0.2747	28672 B
'int - ImmutableList filter by RemoveAll'	57.953 us	0.9039 us	0.8455 us	2.3804	0.2441	30376 B
'int - List.reduce'	1.095 us	0.0148 us	0.0138 us	-	-	-
'int - ImmutableList.reduce'	4.495 us	0.0656 us	0.0806 us	0.0076	-	112 B
'int - List.contains'	5.087 us	0.0649 us	0.0607 us	-	-	40 B
'int - ImmutableList.contains'	12.743 us	0.1634 us	0.1448 us	-	-	72 B

QUESTIONS?

Ask question now, or I start talking about how to make mutable data structures immutable! :)