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
- note about purity
- ImmutableCollections

Immutable Data Structures

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

Why?

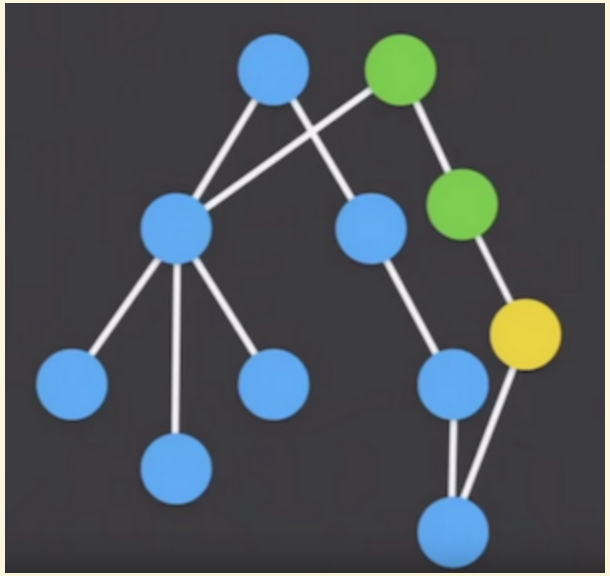
- mutation is common source of bugs
- immutable data structures are easier to reason about
  - value passed to a function, can't be changed
- immutable data structures are thread-safe
- bonus: memory efficient time travelling

How?

- MYTH: to create new immutable value, you need to copy the whole thing
- we can share parts of the structure between old and new value

TODO: meme

Structural sharing



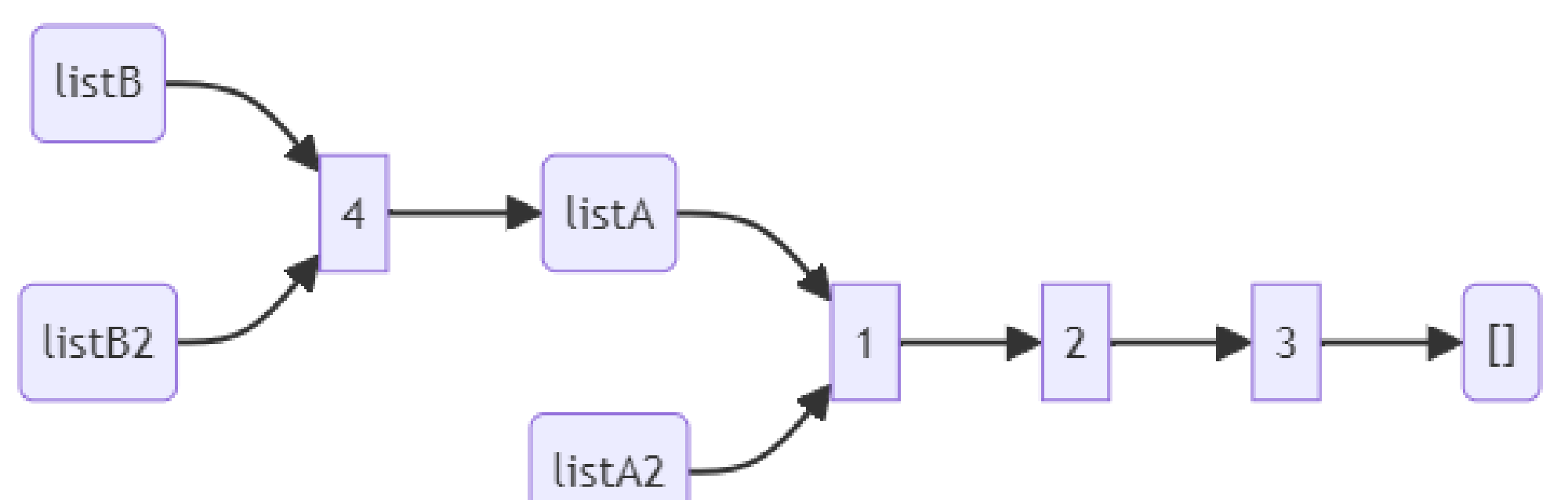
F# (Linked) list

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

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

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

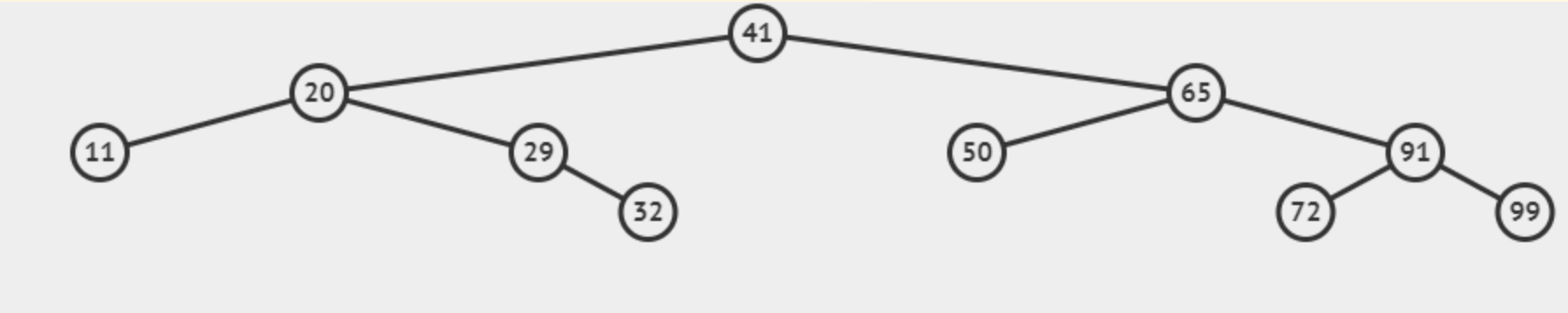
- `List.find`, `List.nth` goes through list one by one
- `Set` is better for searching in big lists
- if you really need indexing, use array

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
```

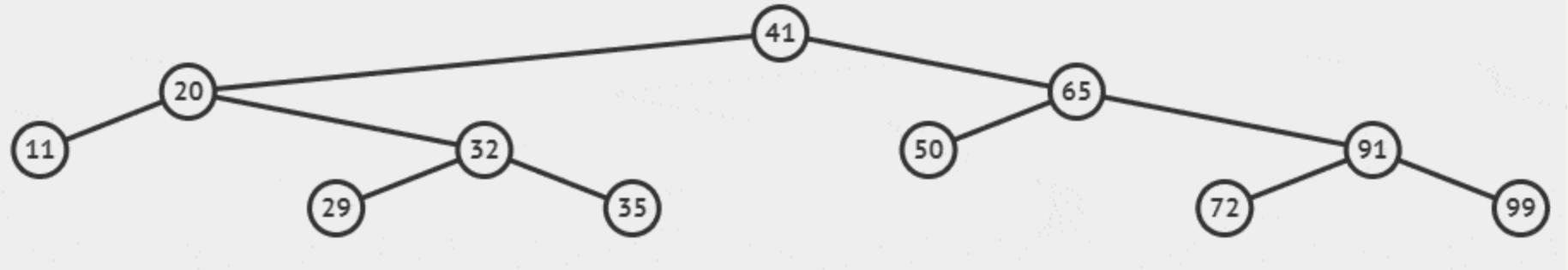


```
(* A classic functional language implementation of binary trees *)

[<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
```

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
```

N=0, h=0 (empty 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) example
- keys cannot be duplicite - insert (`Map.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**

Another important functions

- `Set.union`
- `Set.intersect`
- `Set.difference`

all of them works recursively on tree structure -> faster than the same on `list`

- `Set.isSubset`
- `Set.isSuperset`

try 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)

```
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)
  example
- keys cannot be duplicite - insert (`Map.add`) repace value if key already exists

Creation of `Map` - List.groupBy

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

Comparison with C# collections

Naming

<small>

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)		

Other useful C# collections

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

Enumerable, seq - lazy sequences

`seq<'t>`

- 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 is 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())
```

Pure functions

- **Pure** function:
  - always returns the same result for the same input (**referential transparency**)
  - no side effects
- Immutable data structures allows us to write **pure** functions.
- no mutable variables / data structures, no side effects => **referential transparency**

- BUT:
- **referential transparency** can be achieved even with mutable data structures
- 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
```

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

Method	Mean	Error	StdDev	Gen0	Gen1
'int - List cons'	2.375 us	0.0473 us	0.1059 us	2.5482	0.4234
'int - ImmutableList cons'	95.410 us	1.7462 us	1.6334 us	40.0391	9.6436
'int - List.reverse'	2.511 us	0.0413 us	0.0606 us	2.5482	0.4234
'int - ImmutableList.reverse'	71.121 us	0.6854 us	0.6411 us	3.7842	0.8545
'int - List.map'	2.781 us	0.0543 us	0.0687 us	2.5482	0.5074