

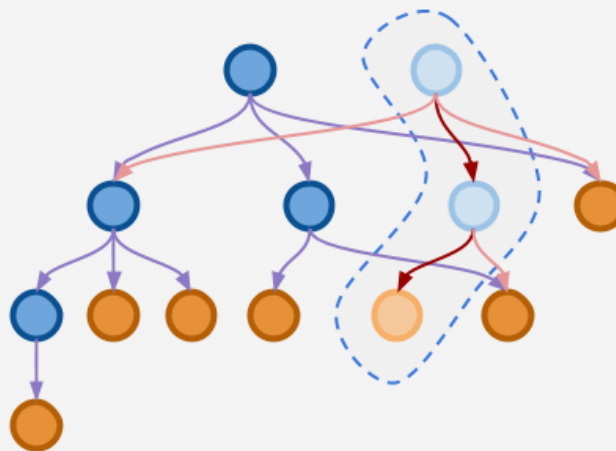
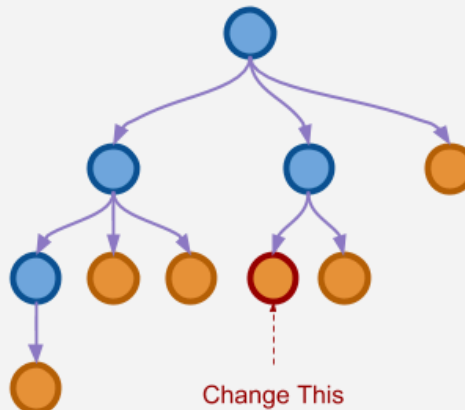
DevTalks 18.10.2024

Immutable Data Structures

Jindřich Ivánek

F# Expert at Ciklum

jindraivanek.hashnode.dev



Immutable Data

Definition

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

Why use them?

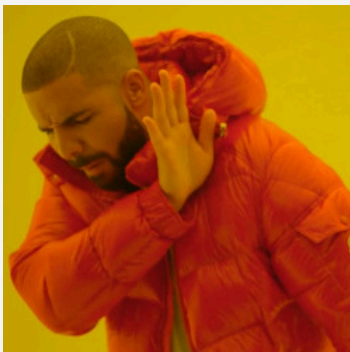
- mutation is a common source of bugs
- immutable data are easier to reason about
 - value passed to a function, can't be changed
 - easier refactoring
- immutable data structures are **thread-safe**
- bonus: memory efficient time travelling

Why immutability-example

```
1  public class Account {
2      public decimal Money { get; set; } // mutable data
3      public void Pay(decimal amount) // race condition
4      {
5          Money -= amount; // change of value, no rollback, history
6      }
7  }
8  var account = new Account { Money = 1000 };
9  Parallel.For(0, 10, _ => account.Pay(100)); // anything between 0 - 900
```

Immutable update

MYTH: to "change" immutable value, you need to copy the whole thing



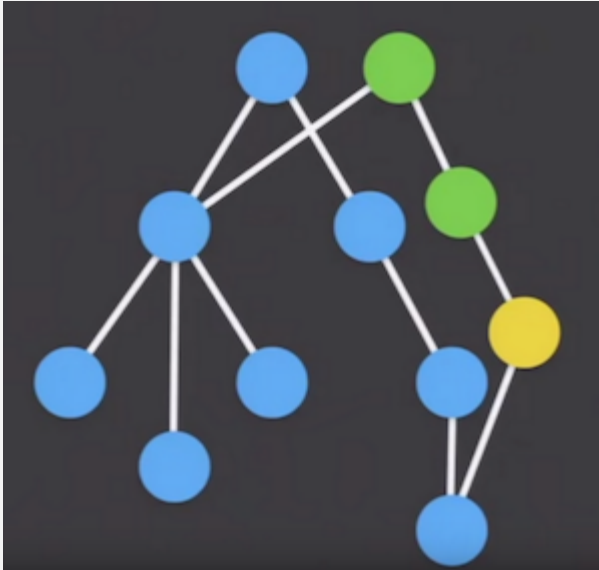
Copy
all data



Reuse
unchanged
data

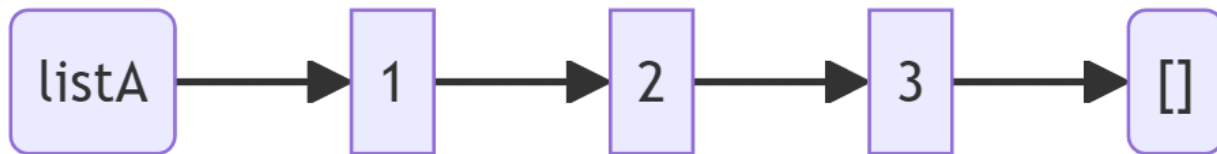
How?

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



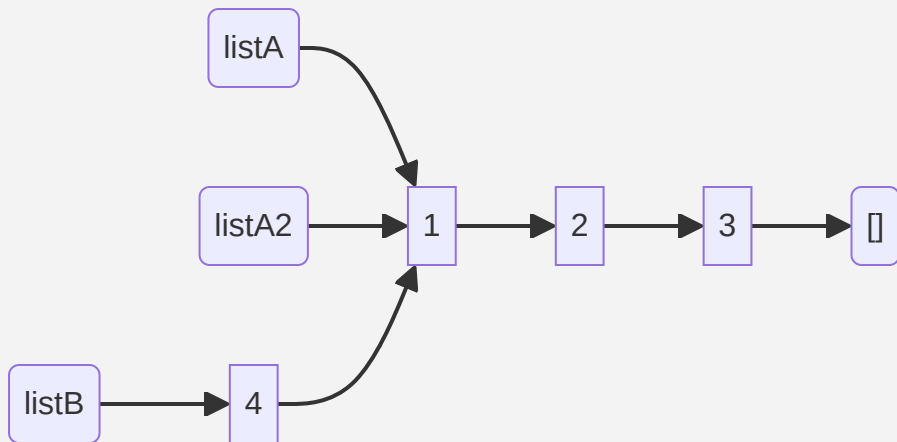
(Linked) list

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



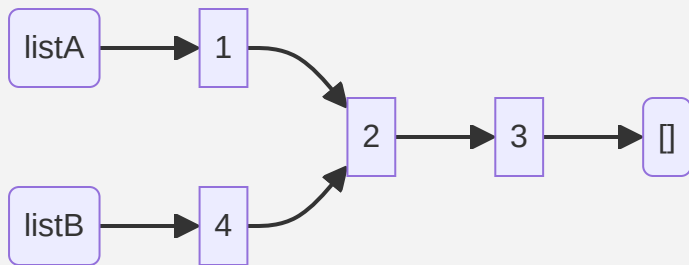
(Linked) list sharing

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

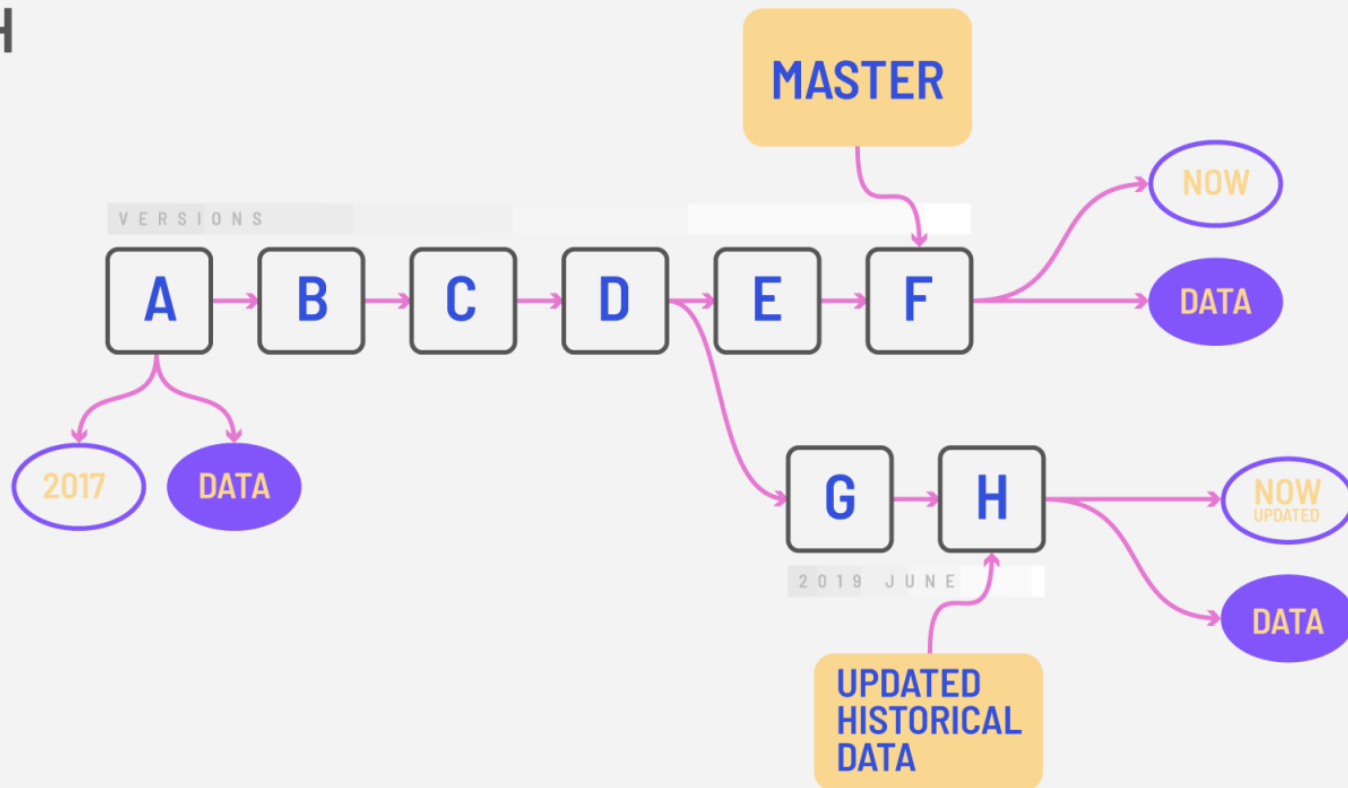


List-update head

```
1  let listA = [1; 2; 3]
2  let listB = 4 :: List.tail listA
```



COMMIT GRAPH



List Benchmark

FsListWorkload compared to CsListWorkload

```
1  member this.FsListWorkload() =
2      this.listOfRecords
3      |> List.map (fun x -> { x with Id = x.Id + 1})
4      |> List.filter (fun x -> x.Id % 2 = 0)
5      |> List.map (fun x -> int64 x.Id)
6      |> List.sum
7
8  member this.CsListWorkload() =
9      let csList = this.csList
10     for i=0 to csList.Count - 1 do
11         csList.[i] <-
12             { csList.[i] with Id = csList.[i].Id + 1 }
13     csList.RemoveAll(fun x -> x.Id % 2 <> 0)
14     let x = csList.Sum(fun x -> int64 x.Id)
15     x
```

size	Time Ratio	Memory Ratio
100	1.41	2.54
1000	1.51	2.26
10000	1.61	2.16
100000	1.37	2.15

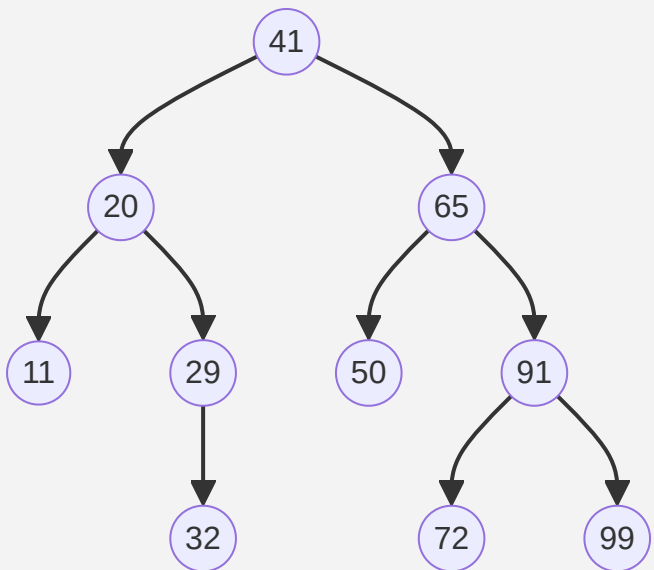
Notes on Benchmarks

- hard and time expensive to write correct benchmarks
- there are always ways to make them faster
- at best they are only indicative
- all benchmarks are wrong

Set

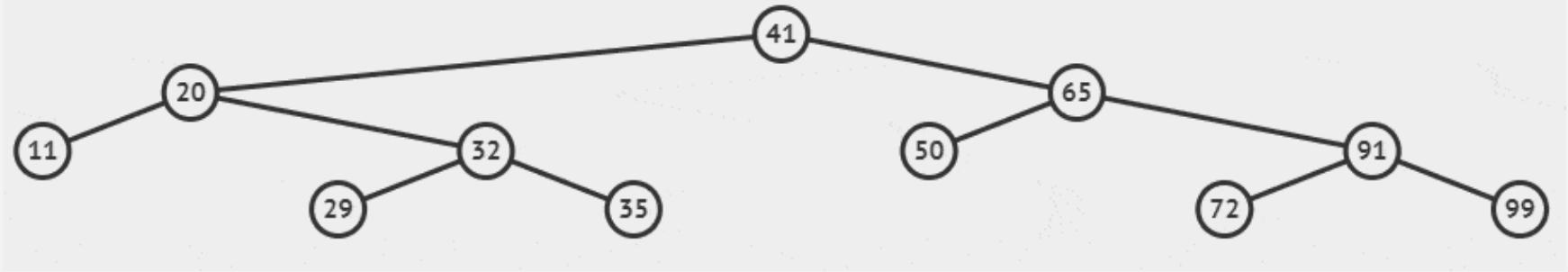
- unordered set of values
- typically implemented as a balanced tree (AVL)

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



Insert = search + add

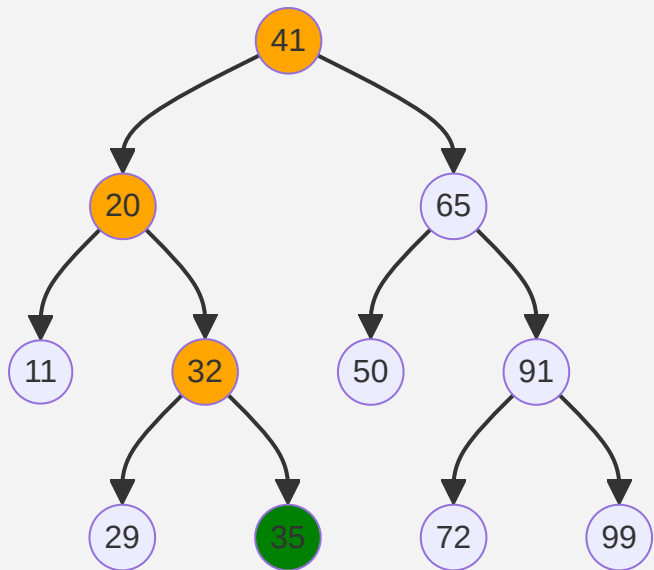
```
1 let s2 = s |> Set.add 35
```



source: <https://visualgo.net/en/bst>

Insert-structural sharing

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



Building new Set

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

N=0, h=0 (empty BST)

source: <https://visualgo.net/en/bst>

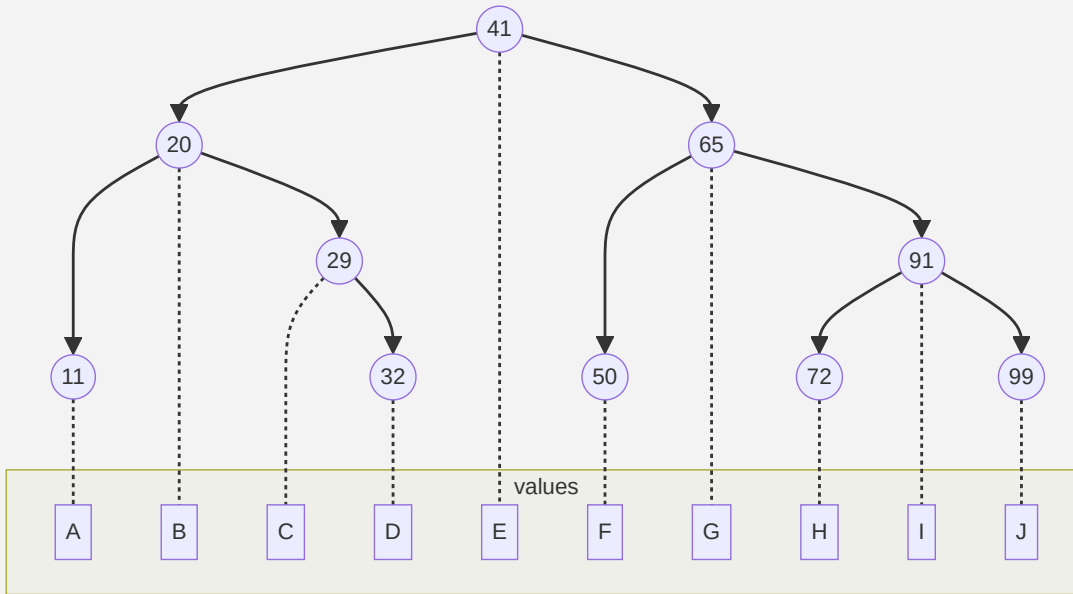
Set Benchmark

Immutable `Set` / mutable `HashSet`

Method	size	Time Ratio	Memory Ratio
'create + contains'	100	3.52	1.46
'create + contains'	1000	4.32	1.82
'create + contains'	10000	4.28	2.23
'create + contains'	100000	3.02	2.72
'contains'	100	0.97	1.01
'contains'	1000	1.09	1.00
'contains'	10000	1.09	1.00
'contains'	100000	0.92	1.00

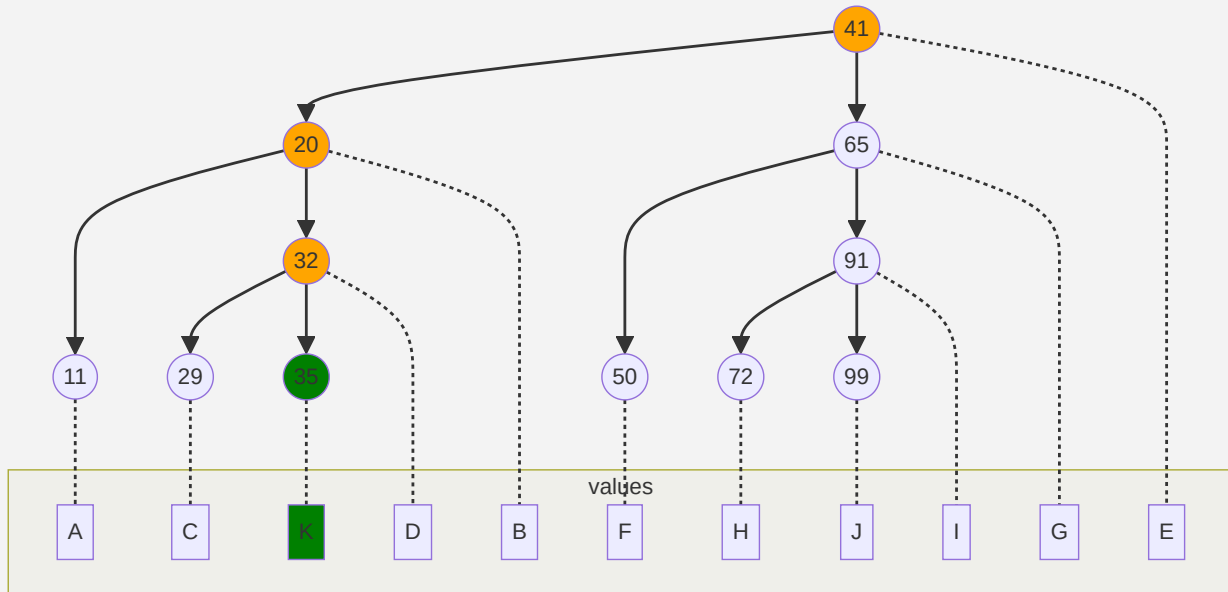
Map

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



Map sharing

```
1 let mapA = Map.ofList [11, "A"; 20, "B"; 29, "C"; 32, "D"; 41, "E"; 50, "F"; 65, "G"; 72, "H"; 91, "I"]
2 let mapB = Map.add 35 "K" mapA
```



Map Benchmark



Immutable **Map** / mutable **Dictionary**

Method	size	Time Ratio	Memory Ratio
'containsKey'	100	1.08	1.01
'containsKey'	1000	0.85	1.00
'containsKey'	10000	1.07	1.00
'containsKey'	100000	0.99	1.00
'create + containsKey'	100	2.07	1.92
'create + containsKey'	1000	2.98	2.21
'create + containsKey'	10000	1.79	2.61
'create + containsKey'	100000	2.19	3.13



Thank you!



Records

```
1 { Id: int; Name: string; Data: BigObject }
```

- immutable by default
- no special immutable structure
- update syntax creates new record with not-changed fields shared with old record

- ```
{ oldRecord with Name = "Bob" }
```

- only reference is copied
- `Data` is shared

# Structural comparison in .NET



- definition of equality based on values, not references
- all F# data types have defined structural comparison and ordering
- immutability and structural comparison are different features, but it is common that immutable data structures have defined structural comparison
  - same value with different references is more common when working with immutable data structures

DevTalks 18.10.2024

# Immutable Data Structures

# Jindřich Ivánek

## F# Expert at Ciklum

jindraivanek.hashnode.dev

