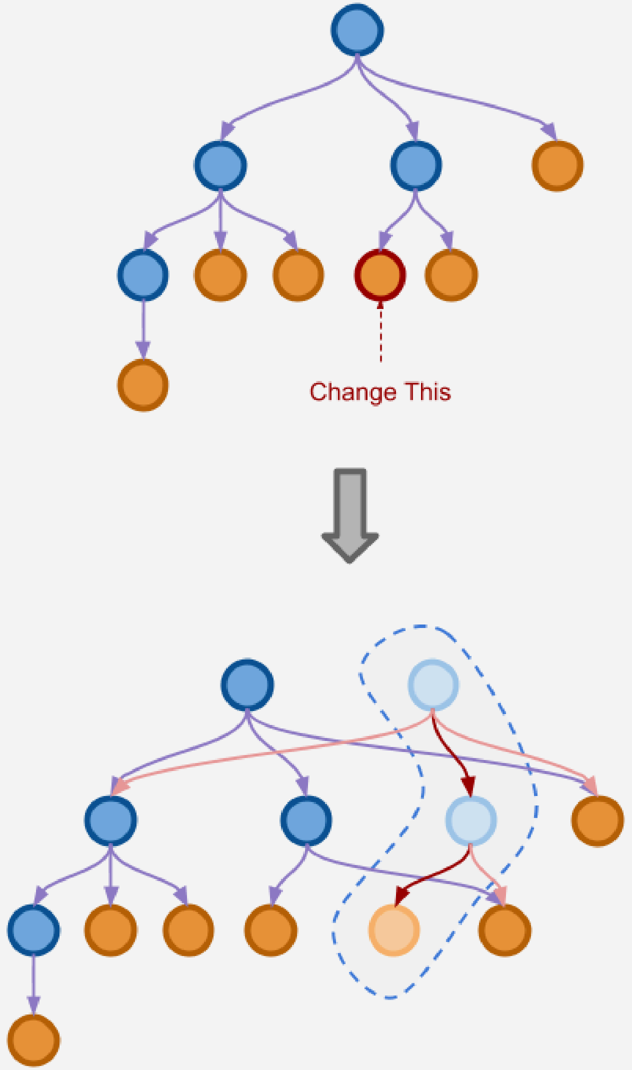
DevTalks 18.10.2024

Immutable Data Structures

Jindřich Ivánek
F# Expert at Ciklum
jindraivanek.hashnode.dev

CIKLUM



Immutable Data

Definition

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

Why use them?

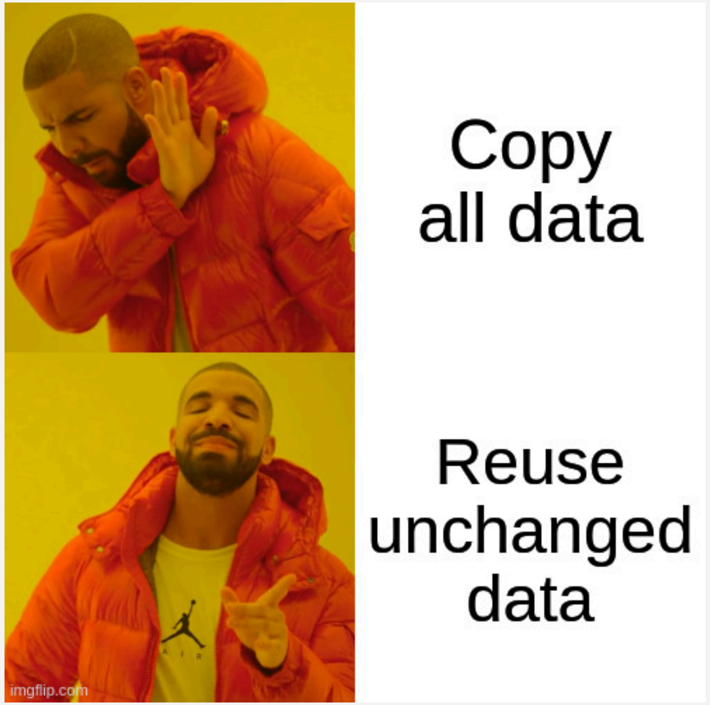
- mutation is common source of bugs
- immutable data are easier to reason about
    - value passed to a function, can't be changed
    - easier refactoring
- immutable data structures are **thread-safe**
- bonus: memory efficient time travelling

Why immutability-example

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

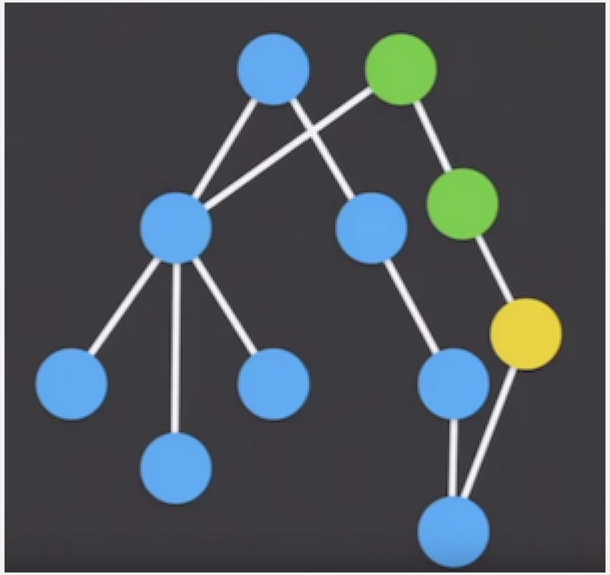
Immutable update

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



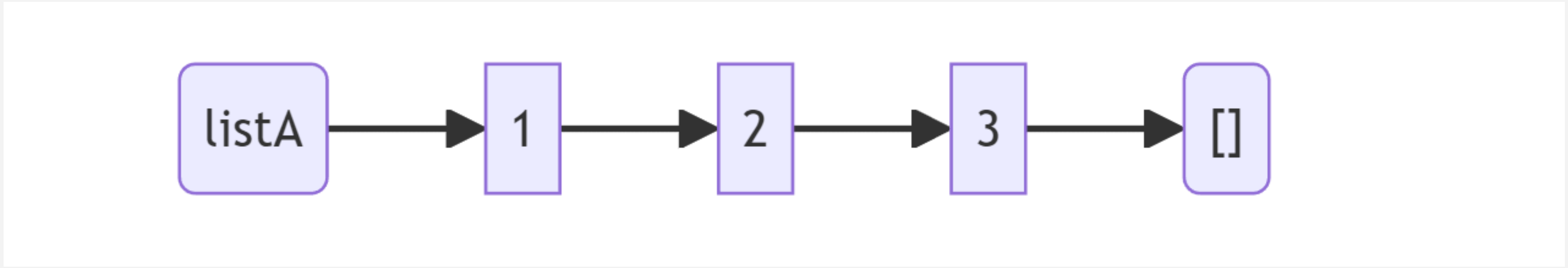
How?

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



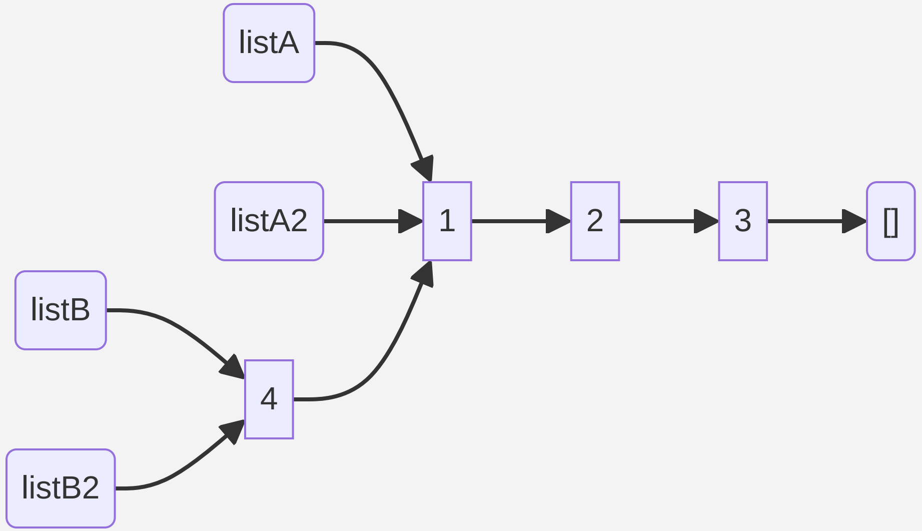
(Linked) list

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



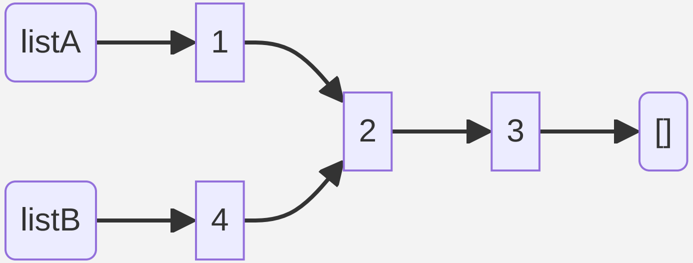
(Linked) list sharing

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

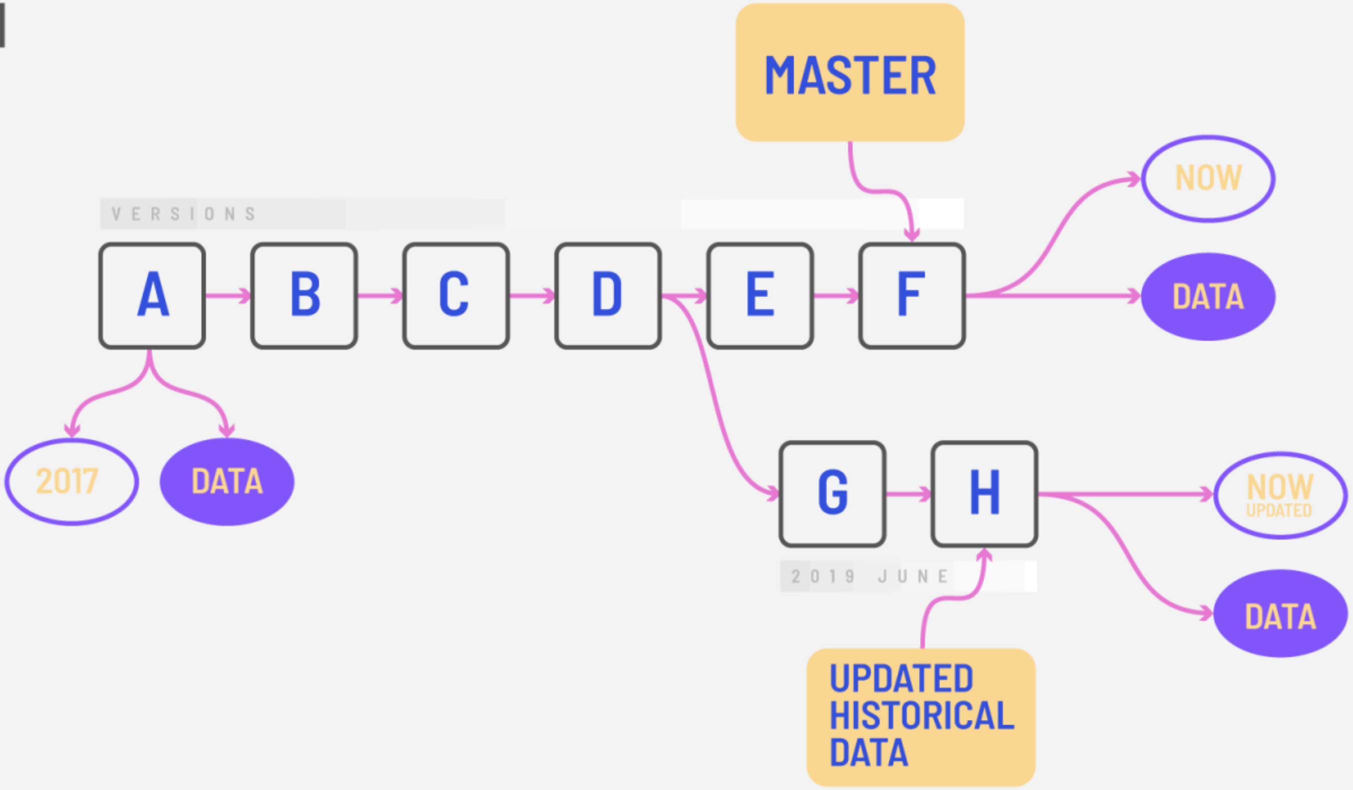

$$\begin{tikzcd}
 & \text{listA} \arrow[dr] & & & & \\
 & \text{listA2} \arrow[r] & 1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & {[]} \\
\text{listB} \arrow[dr] & & & & & \\
 & 4 \arrow[uur] & & & & \\
\text{listB2} \arrow[ur] & & & & &
\end{tikzcd}$$

List-update head

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



COMMIT GRAPH



List Benchmark

CIKLUM

```
member this.FsListWorkload() =
    this.listOfRecords
    |> List.map (fun x -> { x with Id = x.Id + 1})
    |> List.filter (fun x -> x.Id % 2 = 0)
    |> List.map (fun x -> int64 x.Id)
    |> List.sum

member this.CsListWorkload() =
    let csList = this.csList
    for i=0 to csList.Count - 1 do
        csList.[i] <-
          { csList.[i] with Id = csList.[i].Id + 1 }
    csList.RemoveAll(fun x -> x.Id % 2 <> 0)
    let x = csList.Sum(fun x -> int64 x.Id)
    x
```

FsListWorkload compared to CsListWorkload

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

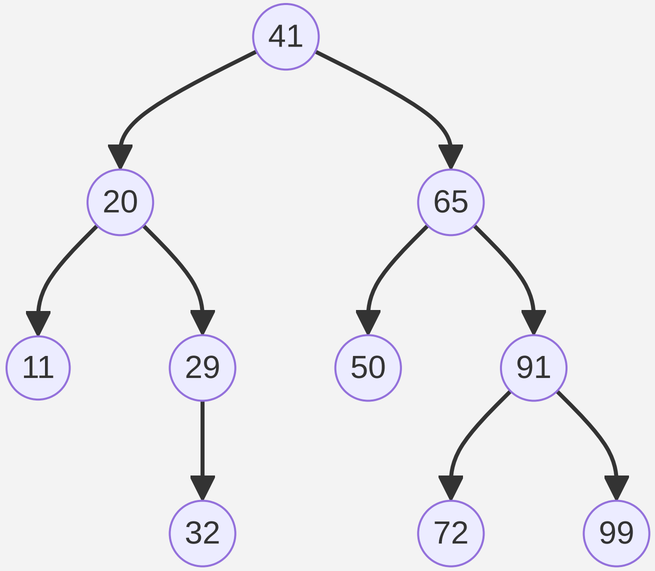
- hard and time expensive to write correct benchmarks
- there are always ways to make them faster
- at best they are only indicative
- all benchmarks are wrong

Set

Unordered set of values

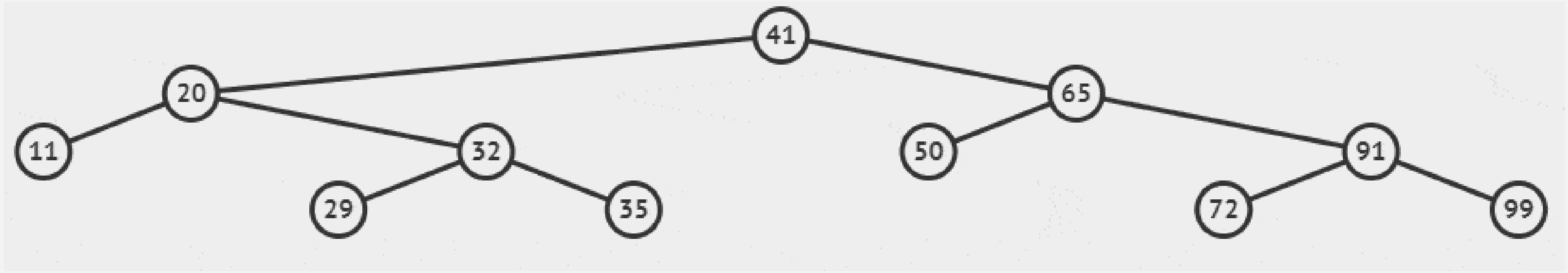
Typically implemented as a balanced tree (AVL)

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



Insert = search + add

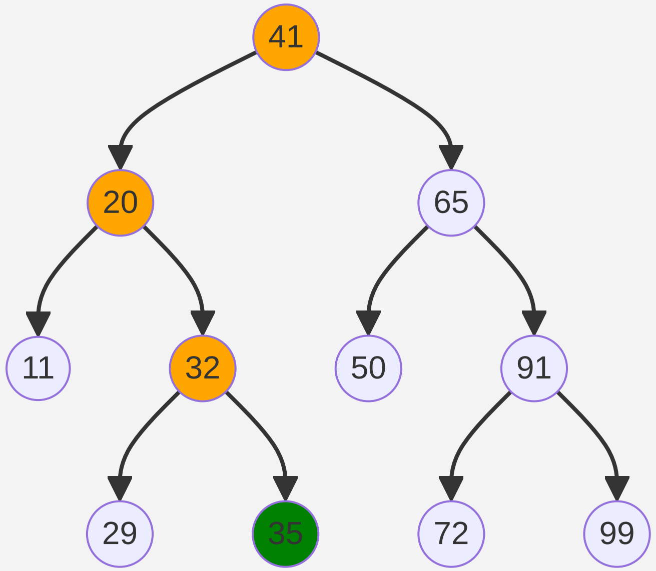
```
let s2 = s |> Set.add 35
```



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

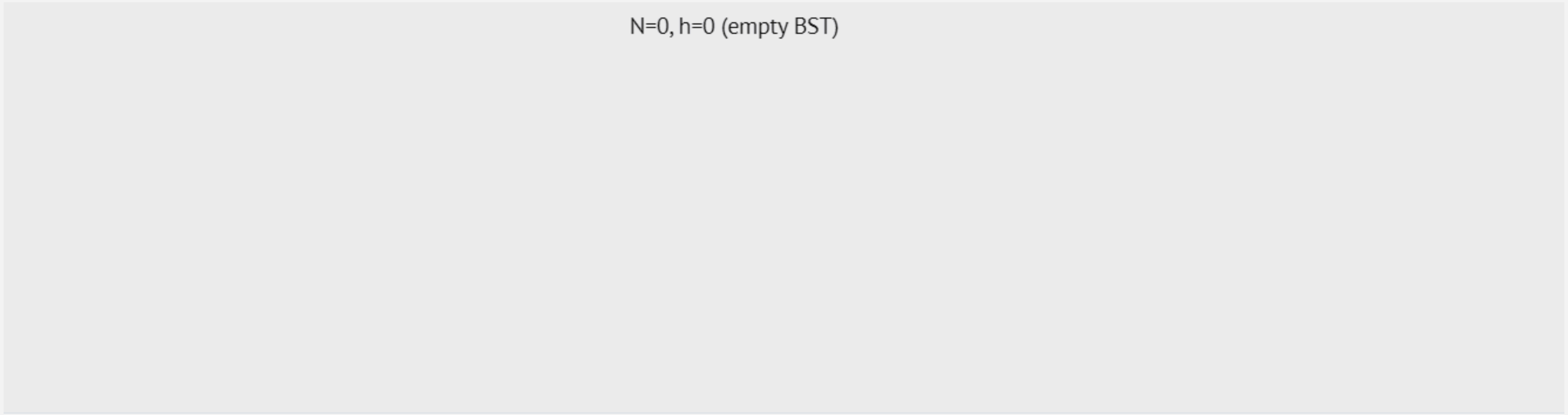
Insert-structural sharing

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



Building new Set

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


N=0, h=0 (empty BST)

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

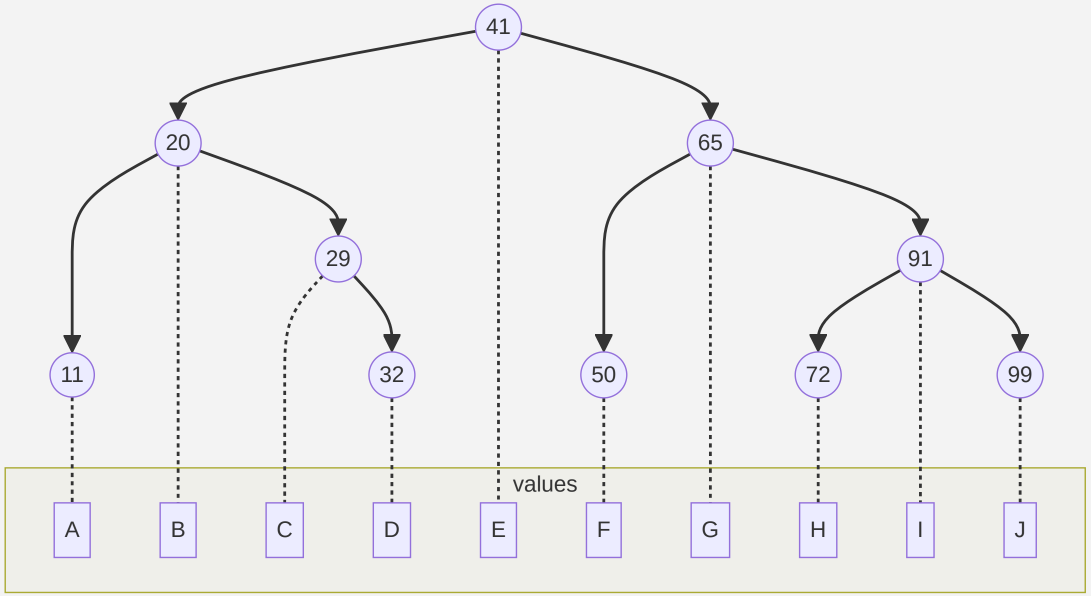
Set Benchmark

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

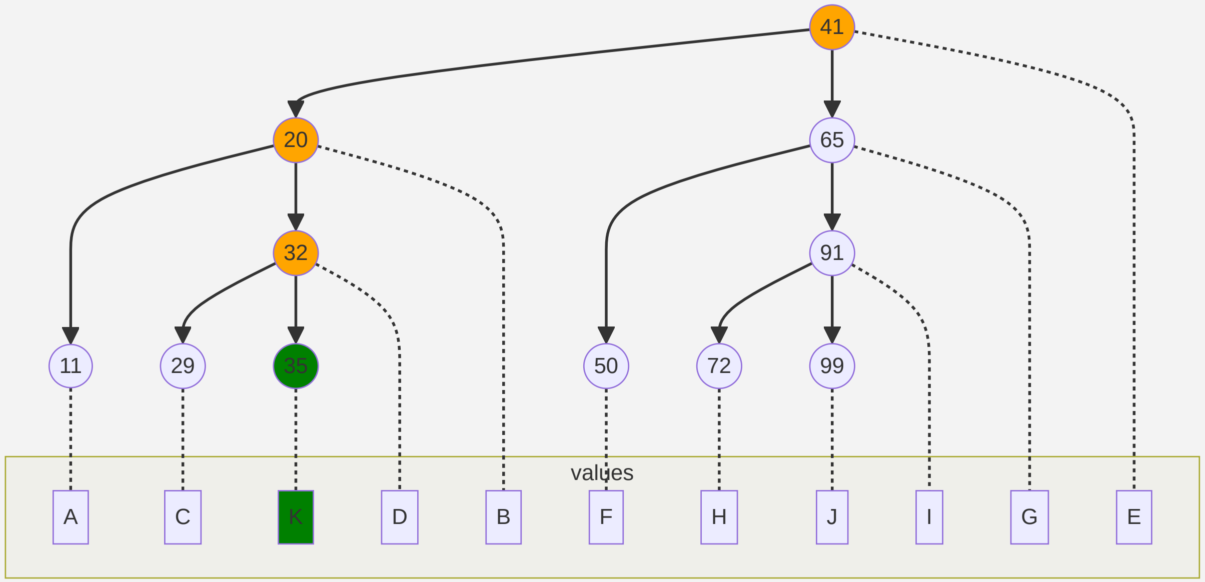
Map

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



Map sharing

```
let mapA = Map.ofList [11, "A"; 20, "B"; 29, "C"; 32, "D"; 41, "E"; 50, "F"; 65, "G", 72, "H"; 91, "I"; 99, "J"]
let mapB = Map.add 35 "K" mapA
```



Map Benchmark

CIKLUM

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

Records

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

- Immutable by default
- No special immutable structure
- Update syntax create 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

CIKLUM

Thank you!