IT UNIVERSITY OF COPENHAGEN

Functional Programming

Jesper Bengtson

Last week

We covered data types and collections

- Discriminated unions
- Maps
- Sets
- Data representation

This week

- Modules
- More maps and sets (the assignment)
- Some more programming examples
- Stateful programs - non-functional properties

This weeks assignment

- Create a multiset
- Create a dictionary (two options, easy and slow, or harder and efficient).

This weeks assignment

- Create a multiset
- Create a dictionary (two options, easy and slow, or harder and efficient).

You will be able to port the multiset and the efficient dictionary immediately into your Scrabble project

This weeks assignment

- Create a multiset
- Create a dictionary (two options, easy and slow, or harder and efficient).

You will get full assignment credit for the slow dictionary, but you will not be able to pass all of the tests, or use it in Scrabble later

This weeks assignment

- Create a multiset
- Create a dictionary (two options, easy and slow, or harder and efficient).

We will give you a dictionary for your project if you choose to do the slow one, but it will not be as good as one you make yourself

Questions?

Modules

- Modular programming design
  - Encapsulation
  - Abstraction
  - Reuse of components
- A module is characterised by
  - A signature (.fsi -file)
  - A matching implementation (.fs file)

Signatures

Signatures are given in .fsi-files

```
module ModuleName
 type T  (required type)
 val f : <type> (required function)
 val g : <type> (required function)
 …
 …
```

.fsi-files list the types (if any) that must be defined, and the visible functions (if any) that must be implemented

Implementations

Implementations are given in .fs-files

```
module ModuleName (same as .fsi)
 type T = <def> (must be discrete
                  union or record)
 let f = <implementation>
 let g = <implementation>
 …
 …
```

implementation types must match .fsi types

Implementations

Implementations are given in .fs-files

```
module ModuleName (same as .fsi)
 type T = <def> (must be discrete
                 union or record)
 let f = <implementation>
 let
 .
 .
 .
```

Important! You will get very weird error messages if you say 'type T = int', for instance

Rational numbers

Let’s write a small library for rational numbers

- Create a new project
- Create an .fsi file and define the signatures
- Create an .fs file and write the implementation

Title Text

Recall that

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd} \qquad \frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

$$\frac{a}{b} * \frac{c}{d} = \frac{ac}{bd} \qquad \frac{a}{b} / \frac{c}{d} = \frac{ad}{bc}$$

Let’s code

Rational1.fsi

```
module Rational1
    type Rat

    val mkRat : int -> int -> Rat

    val ( + ) : Rat -> Rat -> Rat
    val ( - ) : Rat -> Rat -> Rat
    val ( * ) : Rat -> Rat -> Rat
    val ( / ) : Rat -> Rat -> Rat
```

Rational1.fs

```
module Rational1
  type Rat = R of int * int

  let mkRat a b = R (a, b)

  let ( + ) (R (a, b)) (R (c, d)) =
      R (a * d + b * c, b * d)
  let ( - ) (R (a, b)) (R (c, d)) =
      R (a * d - b * c, b * d)
  let ( * ) (R (a, b)) (R (c, d)) =
      R (a * c, b * d)
  let ( / ) r (R (c, d)) =
      mkRat a d * mkRat b c
```

Program.fs

```
open Rational1

printfn "%A" (mkRat 12 2 + mkRat 12 4)
printfn "%A" (mkRat 12 2 - mkRat 12 4)
printfn "%A" (mkRat 12 2 * mkRat 12 4)
printfn "%A" (mkRat 12 2 / mkRat 12 4)

printfn "%A" (5 + 3)
```

Program.fs

```
open Rational1

printfn "%A" (mkRat 12 2 + mkRat 12 4)
printfn "%A" (mkRat 12 2 - mkRat 12 4)
printfn "%A" (mkRat 12 2 * mkRat 12 4)
printfn "%A" (mkRat 12 2 / mkRat 12 4)

printfn "%A" (5 + 3)
```

Typing error

Problem

We have replaced the definition of +. It can now only be used for operations on rational numbers

Solutions?

- Replace (+) with something else, like (.+.) (works, but wont scale)
- Override the definition of + (F# is object oriented after all)

Let’s code

Rational2.fsi

```
module Rational2
    [<Sealed>]
    type Rat =
        static member ( + ) :
                Rat * Rat -> Rat
        static member ( - ) :
                Rat * Rat -> Rat
        static member ( * ) :
                Rat * Rat -> Rat
        static member ( / ) :
                Rat * Rat -> Rat

    val mkRat : int -> int -> Rat
```

Rational2.fsi

```
module Rational2
    [<Sealed>]
    type Rat =
        static member ( + ) :

                 Rat * Rat -> Rat

    val mkRat : int -> int -> Rat
```

static member - this smells object oriented, and it is

Rational2.fs

```
module Rational2
  type Rat =
    |  R of int * int
    static member ( + ) (R (a, b), R (c, d)) =
        R (a * d + b * c, b * d)
    static member ( - ) (R (a1, b1), R (a2, b2)) =
        R (a * d - b * c, b * d)
    static member ( * ) (R (a, b), R (c, d)) =
        R (a * c, b * d)
    static member ( / ) (r, R (c, d)) =
        mkRat a d * mkRat b c

  let mkRat a b = R (a, b)
```

Program.fs

```
open Rational2

printfn "%A" (mkRat 12 2 + mkRat 12 4)
printfn "%A" (mkRat 12 2 - mkRat 12 4)
printfn "%A" (mkRat 12 2 * mkRat 12 4)
printfn "%A" (mkRat 12 2 / mkRat 12 4)

printfn "%A" (5 + 3)
```

Program.fs

```
open Rational2

printfn "%A" (mkRat 12 2 + mkRat 12 4)
printfn "%A" (mkRat 12 2 - mkRat 12 4)
printfn "%A" (mkRat 12 2 * mkRat 12 4)
printfn "%A" (mkRat 12 2 / mkRat 12 4)

printfn "%A" (5 + 3)
```

No typing error - overloading works

Problem

We do not have unique representations of rational numbers

Solution

Euclid’s algorithm

$$gcd(x, y) = \begin{cases} x & \text{if } y = 0 \\ gcd(y, remainder(x, y)) & \text{if } x >= y \text{ and } y > 0 \end{cases}$$

Let’s code

Rational3.fsi

```
module Rational2
    [<Sealed>]
    type Rat =
        static member ( + ) :
               Rat * Rat -> Rat
        static member ( - ) :
               Rat * Rat -> Rat
        static member ( * ) :
               Rat * Rat -> Rat
        static member ( / ) :
               Rat * Rat -> Rat

    val mkRat : int -> int -> Rat
```

Rational3.fsi

```
module Rational2
    [<Sealed>]
    type Rat =
                                          :
                                            Rat
                                          :
                                            Rat
                                          :
                                            Rat
                                          :
                        Rat * Rat -> Rat

    val mkRat : int -> int -> Rat
```

Exactly the same as before

Rational3.fs

```
module Rational3
  type Rat =
    |  R of int * int

  let gcd a b
      let rec aux a =
          function
          | 0 -> a
          | b -> aux b (a % b)
      aux (max a b) (min a b)
  let gcdRat (R (a, b)) =
      let c = gcd a b in R (a / c, b / c)

  type Rat with
    static member ( + ) (R (a, b), R (c, d)) =
        R (a * d + b * c, b * d) |> gcdRat
…
```

Program.fs

```
open Rational3

printfn "%A" (mkRat 12 2 + mkRat 12 4)
printfn "%A" (mkRat 12 2 - mkRat 12 4)
printfn "%A" (mkRat 12 2 * mkRat 12 4)
printfn "%A" (mkRat 12 2 / mkRat 12 4)

printfn "%A" (5 + 3)
```

Program.fs

```
open Rational3

printfn "%A" (mkRat 12 2 + mkRat 12 4)
printfn "%A" (mkRat 12 2 - mkRat 12 4)
printfn "%A" (mkRat 12 2 * mkRat 12 4)
printfn "%A" (mkRat 12 2 / mkRat 12 4)

printfn "%A" (5 + 3)
```

And now we get the desired output

Final problem

When we print we expose implementation details (R (a, b) should be internal to the module)

Solution

Override the ToString method

Let’s code

Rational4.fs

```
module Rational4

    type Rat =
        | R of int * int
        override q.ToString() =
            match q with
            | R (a, b) -> sprintf "(%d / %d)" a b

…
```

```
printfn "%A" (mkRat 3 6)
```

Now outputs "(1 / 2)"

Imperative features

- References
- Sequencing
- Mutable data
- Arrays
- Loops

Functional values

Functional values never change

```
let x = 15;;
    val x : int = 15

let y = fib x;;
    val y : int = 987

x;;
    val it : int = 15
```

Calling fib does not change the value of x

References

References are stored on the heap

```
let x = ref 5;;
   val x : int ref = {contents = 5;}
```

dereferenced using the !-operator

```
!x;;
   val it : int = 5
```

Updated using :=

```
x := 6;;
   val it : unit = ()
```

References

References are stored on the heap

lo

deref ... g :=
the

Updating a reference is a side effect.
The value of the calculation is () (unit)

This is standard - operations with side effects (writing to files, memory, stdout…) typically return ()

```
!x;;
   val it : int = 5
```

```
x := 6;;
   val it : unit = ()
```

References

References are stored on the heap

```
let x = ref 5;;
   val x : int ref = {contents = 5;}
```

dereferenced using the !-operator

```
!x;;
   val it : int = 5
```

Updated using :=

```
x := 6;;
   val it : unit = ()
!x;;
   val it : int = 6
```

Aliasing

References can be aliased

```
let x = ref 5;;
   val x : int ref = {contents = 5;}

let y = x;;
   val y : int ref = {contents = 5;}

y := 10;;
   val it : unit = ()

x;;
   val it : int ref = {contents = 10;}
```

Sequencing

Similarly to imperative languages we can string together expressions with ;

e1; e2

- This expression has the same type as e2
- If e1 has any other type than unit then a warning will be generated
- ; separates expressions, it does not end them

Sequencing

Similarly to imperative languages we can string together expressions with ;

Sequencing should only be used to string together expressions with side effects and then finally return a value

- This … ype as e2
- If e1 … nit then a wa…
- ; separates expressions, it does not end them

You can do weird stuff

```
let ( * ) x y = x := 5; !x * y;;
   val ( * ) : x:int ref -> y:int -> int

let x = ref 10;;
   val x : int ref = {contents = 10;}

x * 7;;
   val it : int = 35

x;;
   val it : int ref = {contents = 5;}
```

You can do weird stuff

```
let ( * ) x y = x := 5; !x * y;;
   val ( * ) : x:int ref -> y:int -> int

let x = ref 10;;
   val x : int ref = {contents = 10;}

x * 7;;
   val it : int = 35

x;;
   val it : int ref = {contents = 5;}
```

There are very few places where references are useful (and **never** do this particular thing)

Mutable Variables

- Mutable variables are defined using the 'mutable' keyword
- They are stored on the stack (when possible)
- They behave much more like variables in imperative languages do
- They cannot be aliased
- They are mutated using the <- operator

Mutable Variables

```
let mutable x = 10;;
    val mutable x : int = 10

let mutable y = x;;
    val mutable y : int = 10

x <- 20;;
    val it : unit = ()

x;;
    val it : int = 20

y;;
    val it : int = 10
```

Mutable Variables

```
let mutable x = 10;;
   val mutable x : int = 10
```

New mutable variable x

```
let mutable y = x;;
   val mutable y : int = 10
```

New mutable variable y

```
x <- 20;;
   val it : unit = ()
```

Mutate x

```
x;;
   val it : int = 20
```

Evaluate x

```
y;;
   val it : int = 10
```

Evaluate y

Back to References

References are actually record types with a single mutable field.

```
type Ref<'a> =
    { mutable contents: 'a }
```

```
let (!) r = r.contents
```

```
let (:=) r x = r.contents <- x
```

Which to use

- We nearly always choose mutable variables over references (aliasing is often a very bad idea)
- Most frequently found in object-oriented code inside class declarations
- References are about as useful as singleton arrays.

Arrays

- Arrays function as in imperative languages
- Not functional
- They still have many uses for performance reasons

Array creation

Array literals

```
[|1; 2; 3; 4; 5|];;
   val it : int [] = [|1; 2; 3; 4; 5|]
```

Array creation

```
Array.create 5 3;;
   val it : int [] = [|3; 3; 3; 3; 3|]
```

Array initialisation

```
Array.init 5 (fun x -> x * 2);;
   val it : int [] = [|0; 2; 4; 6; 8|]
```

Array update

```
let arr = Array.create 5 3;;
    val arr : int [] = [|3; 3; 3; 3; 3|]

arr.[3] <- 42;;
    val it : unit = ()

arr;;
    val it : int [] = [|3; 3; 3; 42; 3|]
```

Note that updating the array is a side effect

Array library

Go through the Array library

- Initialisers
- Maps
- Folds
- Iterators
- …

While-loops

While loops do exist, and do not behave like you are used to

```
while b do e
```

While b is true, do e, and throw away the result - only works on side effects

A while expression always has type unit

An example

```
let f arr =
    let mutable x = 0;
    while x < Array.length arr do
        arr.[x] <- arr.[x] * 3;
        x <- x + 1;;
   val f : arr:int [] -> unit

let arr = Array.init 5 (fun x -> x + 7);;
   val arr : int [] = [|7; 8; 9; 10; 11|]

f arr;;
   val it : unit = ()

arr;;
   val it : int [] = [|21; 24; 27; 30; 33|]
```

An example

```
let f arr =
    let mutable x = 0;
    while x < Array.length arr do
        arr.[x] <- arr.[x] * 3;
        x <- x + 1;;
```

This might seem familiar, but do not use loops.
We dock points if you do.

```
   val it : unit = ()

arr;;
   val it : int [] = [|21; 24; 27; 30; 33|]
```

An example

```
let arr = Array.init 5 (fun x -> x + 7);;
   val arr : int [] = [|7; 8; 9; 10; 11|]

Array.map (( * ) 3) arr;;
   val it : int [] = [|21; 24; 27; 30; 33|]
```

- Shorter, cleaner, nicer
- No risk for infinite loops

Iterators

Iterators are higher-order functions that use side effects

```
Array.iter;;
 val it : (('a -> unit) -> 'a [] -> unit)
```

Iterators

Iterators are higher-order functions that use side effects

```
Array.iter;;
 val it : (('a -> unit) -> 'a [] -> unit)
```

```
Array.iter (printfn "element: %d")
                [|7; 8; 9; 10; 11|];;
 element: 7
 element: 8
 element: 9
 element: 10
 element: 11
 val it : unit = ()
```

Iterators

Iterators are higher-order functions that use side effects

```
Array.iteri;;
 val it : ((int ->'a -> unit) -> 'a [] -> unit)
```

Iterators

Iterators are higher-order functions that use side effects

```
Array.iteri;;
 val it : ((int ->'a -> unit) -> 'a [] -> unit)

 Array.iteri (printfn "index: %d\t element: %d")
             [|7; 8; 9; 10; 11|];;
  index: 0     element: 7
  index: 1     element: 8
  index: 2     element: 9
  index: 3     element: 10
  index: 4     element: 11
  val it : unit = ()
```

Tries

Tries, or prefix trees, are efficient data structures for dictionaries

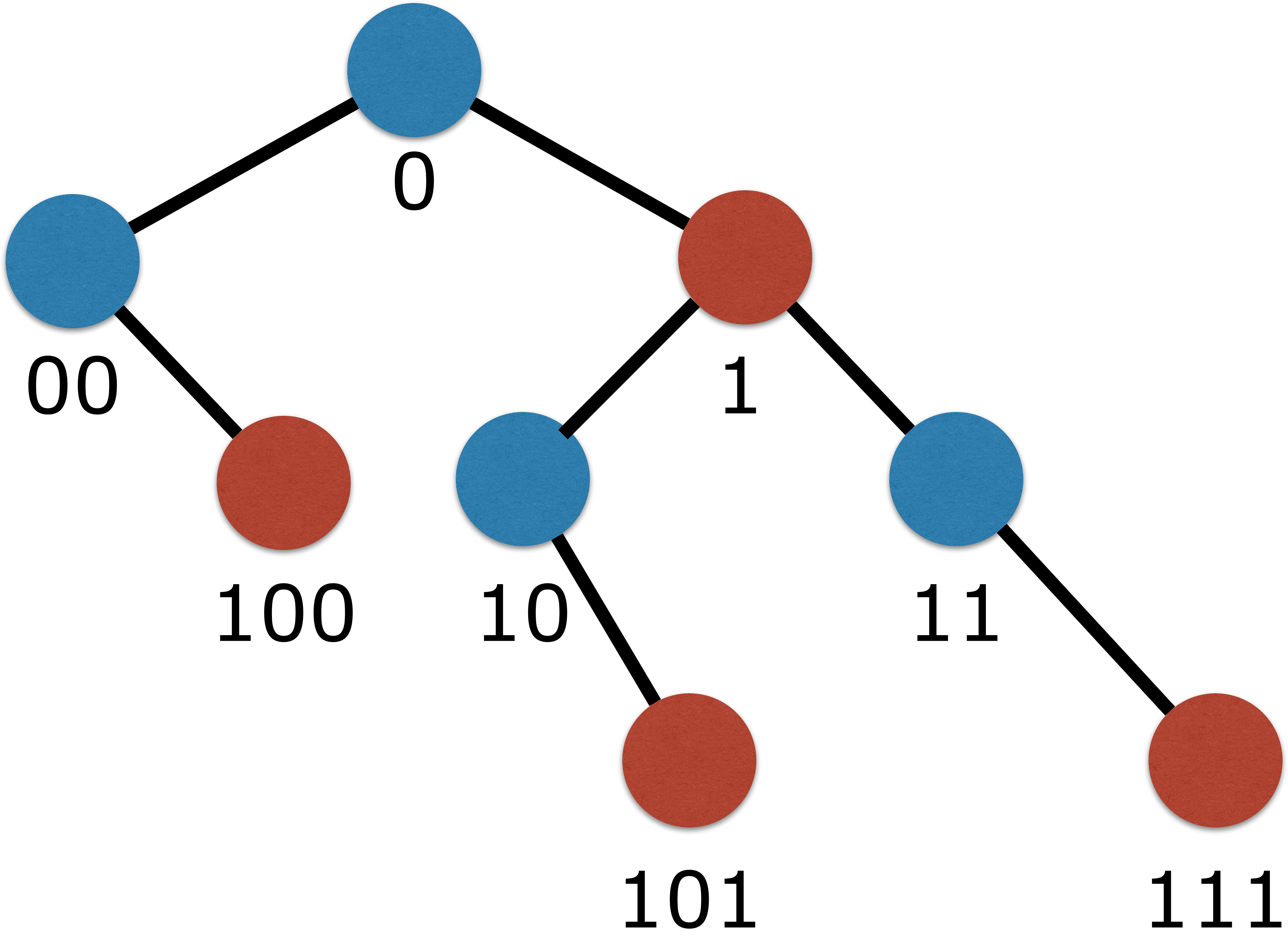
- Lookup and insertion is linear with respect to the size of the word
- Consists of a tree with one sub-tree per letter of the alphabet
- Every node and leaf has a boolean flag to tell if a complete word has been found at this point

Binary tries

Binary tries can be used to store integers

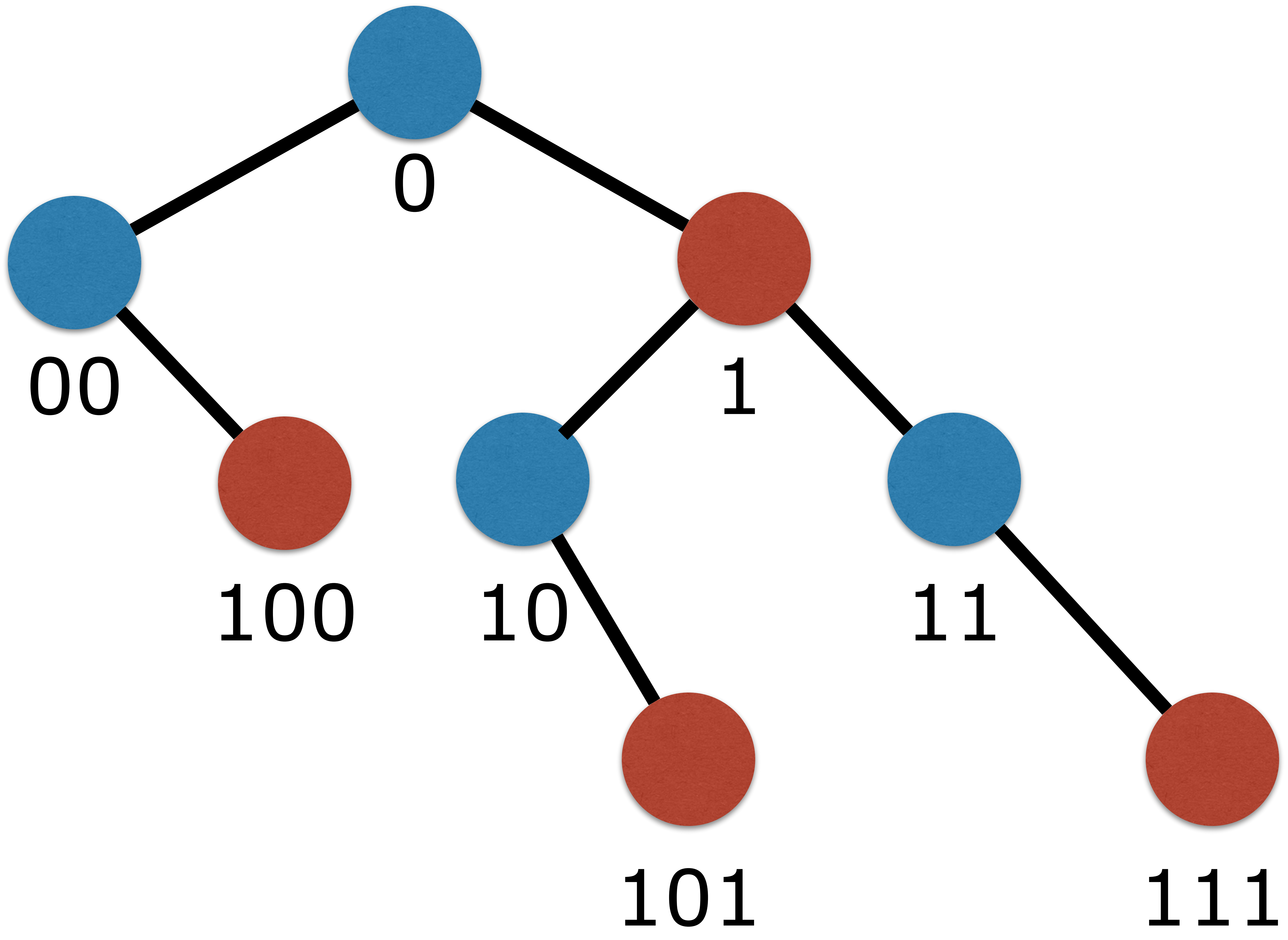
- Lookup and insertion is O(log n) where n is the number we are working with
- We go left if the last bit is 0
- We go right if the last bit is 1
- We shift n right when we go down
- We set or check the flag if n is 0

Example



Example

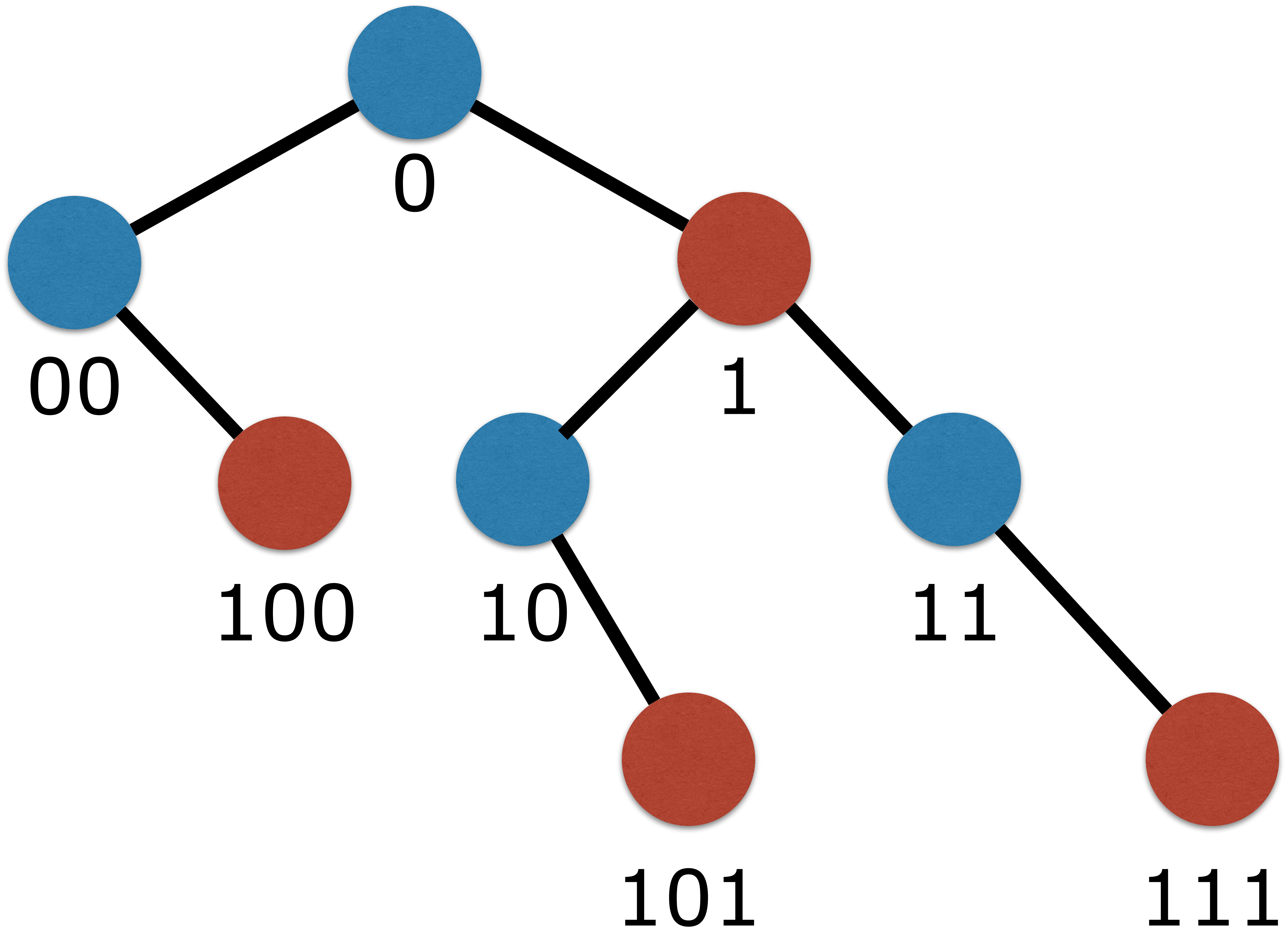
Red nodes means flag is set and number exists in the trie



Example

The Trie contains the numbers

- 4 (100)
- 1 (1)
- 5 (101)
- 7 (111)



Let’s code

Binary tries (empty)

```
type BTrie =
    | Leaf of bool
    | Node of bool * BTrie * BTrie

let empty = Leaf false
```

Binary tries (insertion)

```
let rec insert x =
    function
    | Leaf _           when x = 0u -> Leaf true
    | Node (_, l, r) when x = 0u -> Node(true, l, r)

    | Leaf b           when x % 2u = 0u ->
      Node(b, insert (x / 2u) empty, empty)
    | Leaf b                                 ->
      Node(b, empty, insert (x / 2u) empty)

    | Node (b, l, r) when x % 2u = 0u ->
      Node(b, insert (x / 2u) l, r)
    | Node (b, l, r)                         ->
      Node(b, l, insert (x / 2u) r)
```

Binary tries (insertion)

```
let rec insert x =
    function
    |
    |                                             l, r)

    |

    |

    |

    | Node (b, l, r)                         ->
        Node(b, l, insert (x / 2u) r)
```

We could of course have used integers here but since one of the assignments calls for unsigned integers here is an example of how to use them

Binary tries (lookup)

```
let rec lookup x =
    function
    | Leaf b            when x = 0u        -> b
    | Leaf _                               -> false

    | Node (b, _, _) when x = 0u           -> b
    | Node (_, l, _) when x % 2u = 0u      ->
        lookup (x / 2u) l
    | Node (_, _, r)                       ->
        lookup (x / 2u) r
```

n-ary tries

- Work the same, but typically use maps or arrays in the nodes - one element per letter in the alphabet
- If using arrays:
  - Keep size minimal to keep down construction time and memory usage.
  - Be aware that arrays are imperative (dictionaries are permanently changed by insertion)

n-ary tries

- Work the same, but typically use maps or arrays in the nodes - one element per letter in the alphabet
- If using a
  - Keep siz
    constru
  - Be aware that arrays are imperative (dictionaries are permanently changed by insertion)

In our performance tests maps perform as well as arrays and they are easier to work with as you do not have to know the size of your alphabet before you start.

Questions?