Lecture 11:

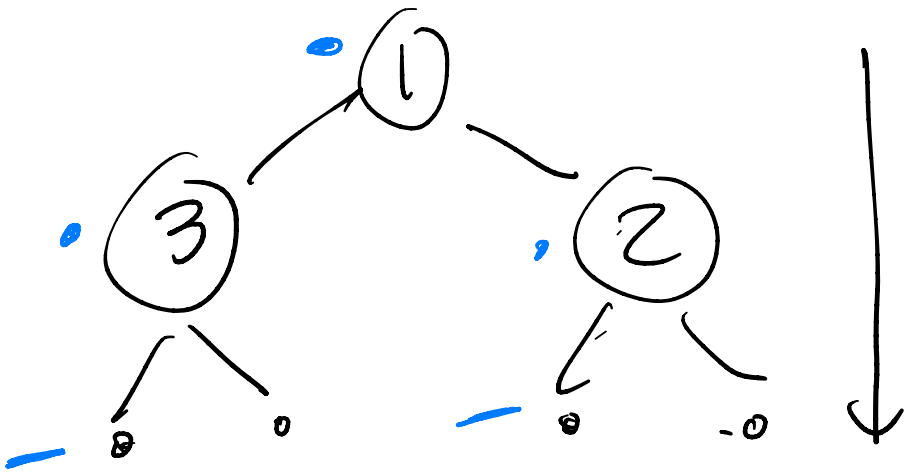
Trees

+

Sorting Trees

Trees represented as list by reading

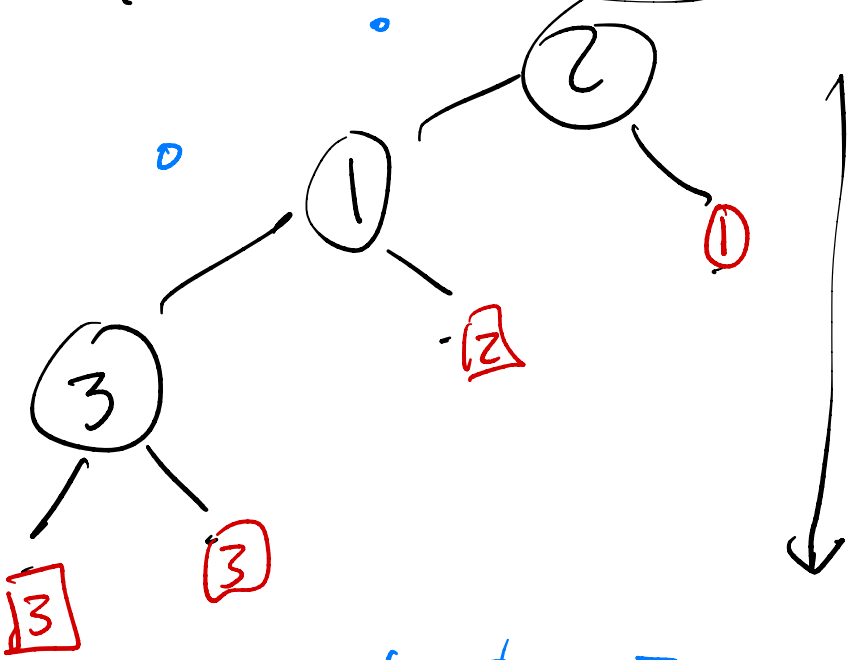
balanced



[3, 1, 2]

depth = 2

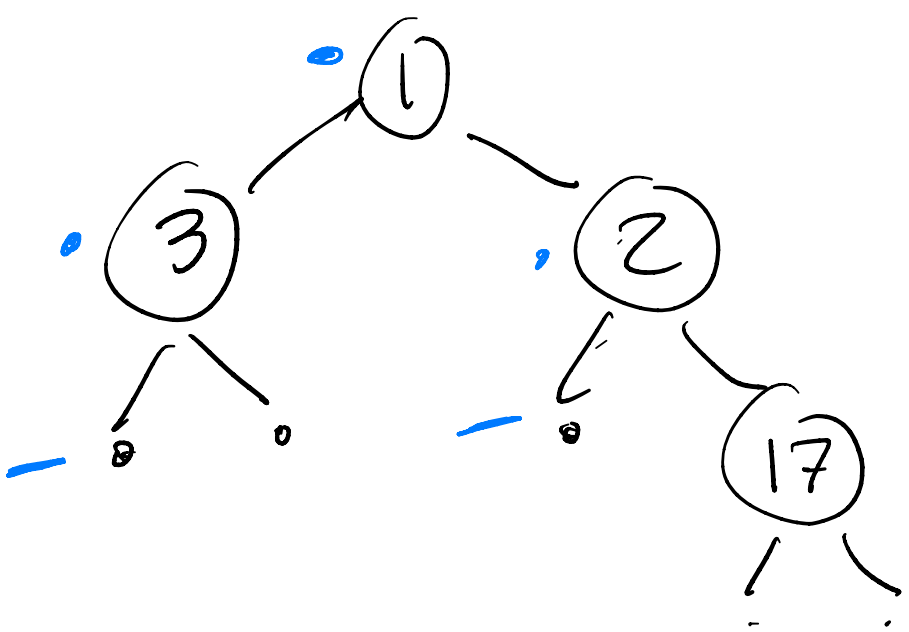
not balanced

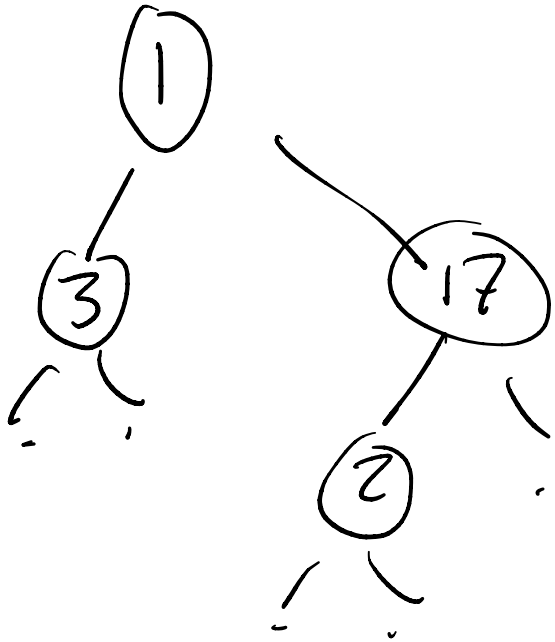


depth = 3

"depth"

balanced: all empty's have same depth (+1?)





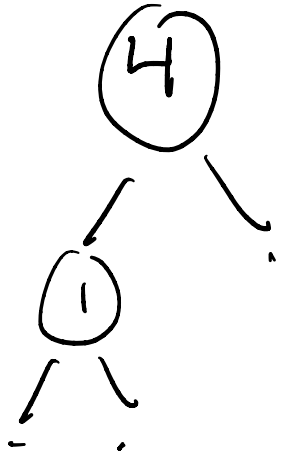
[3, 1, 2, 17]

SML

A tree is

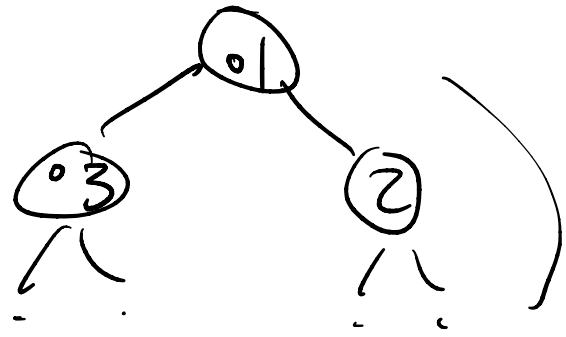
- Empty
- Node(l, x, r)

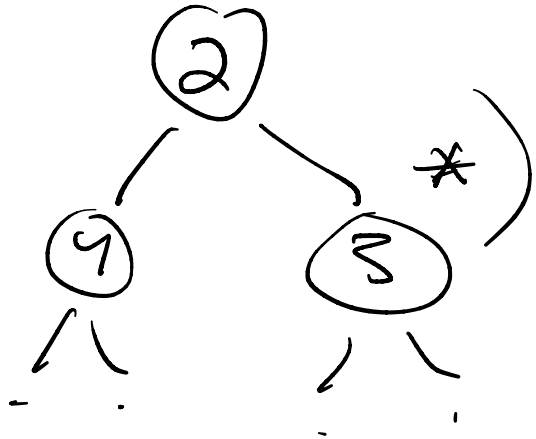
where l: tree
x: int
r: tree



Node(Node(Empty, 1, Empty), 4, Empty)

(* E.g. add one to each elt *)

(* addone 

= 

*)

```
fun addone(t: tree) : tree =
    case t of
        Empty => Empty
      | Node(l, x, r) => Node( addone l
                             , x+1
                             , addone r)
```

```
fun addone(t: tree) : tree =
    case t of
        Empty => Empty
      | Node(l, x, r) => Node( addone l ,
                               x+1 ,
                               addone r )
```

$$W_{addone}(n) = k + W_{addone}\left(\frac{n}{2}\right) + W_{addone}\left(\frac{n}{2}\right)$$

size of l

size of r

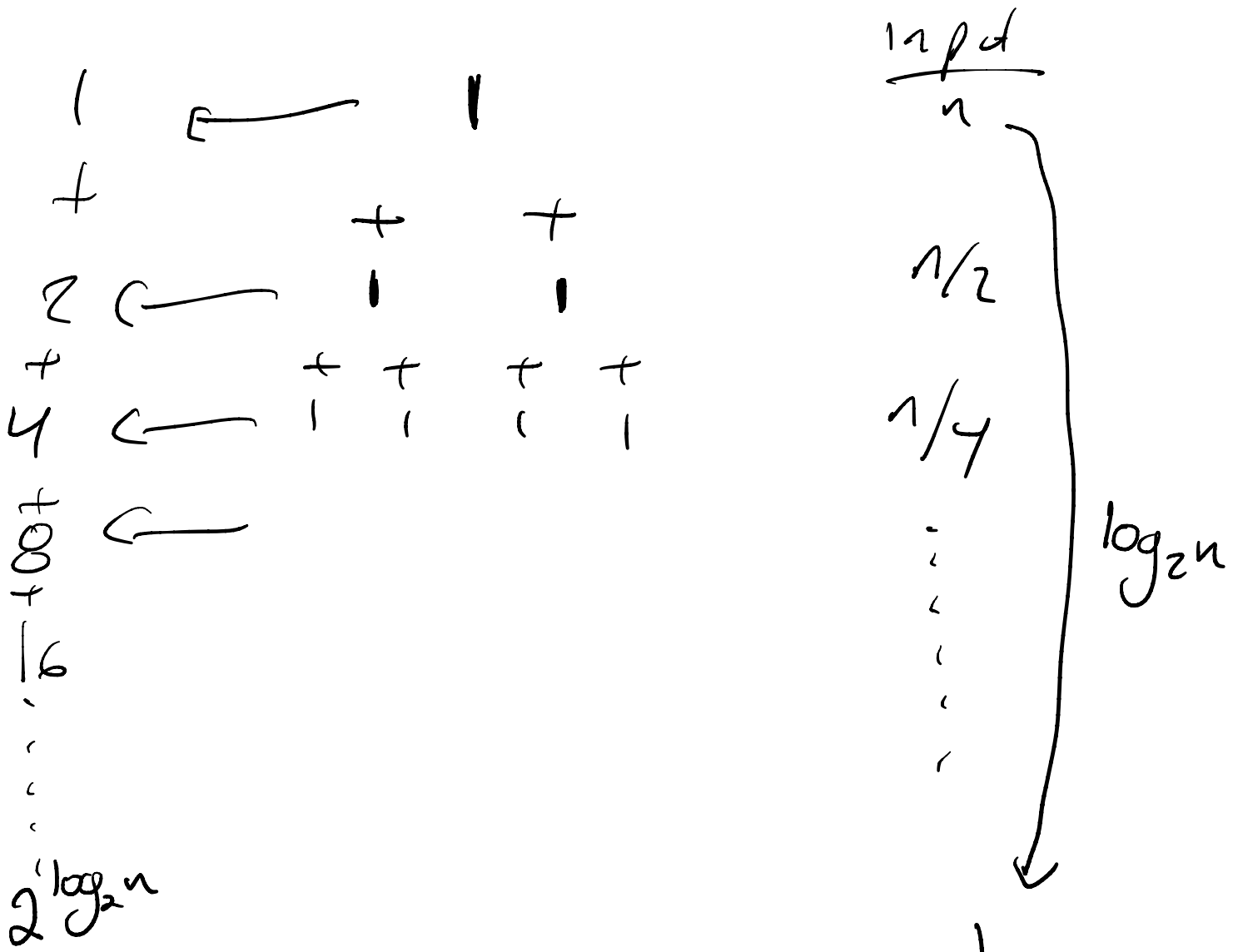
n: size of the tree

$$\approx 1 + 2\,W_{addone}\left(\frac{n}{2}\right)$$

assume t is balanced

$O(n)$

$$W(n) = 1 + 2\,W\left(\frac{n}{2}\right)$$



$$= 1 + 2 + 4 + 8 + 16 + \cdots 2^{\log_2 n}$$

$$= 2 \cdot 2^{\log_2 n} - 1 = 2n - 1$$

$O(n)$

```
fun addone(t:tree) :rec =
  case t of
    Empty => Empty
  | Node(l,x,r) => Node( addone l ,
                         x+1 ,
                         addone r )
```

$$S_{addone}(n) = k + \max\left(S_{addone}\left(\frac{n}{2}\right), S_{addone}\left(\frac{n}{2}\right)\right)$$

n: size of the tree; $\frac{n}{2}$: size of l; $\frac{n}{2}$: size of r

$$\approx 1 + S_{addone}\left(\frac{n}{2}\right)$$

assume t is balanced

$$= \underbrace{1+1+1+1+\cdots+1}_{\log_2 n \text{ times}}$$

is $O(\log_2 n)$

```
fun addone(t: tree) : tree =
  case t of
    Empty => Empty
  | Node(l, x, r) => Node( addone l ,
                           x+1 ,
                           addone r )
```

$$W_{addone}(d) \leq k + W_{addone}(d-1) + W_{addone}(d-1)$$

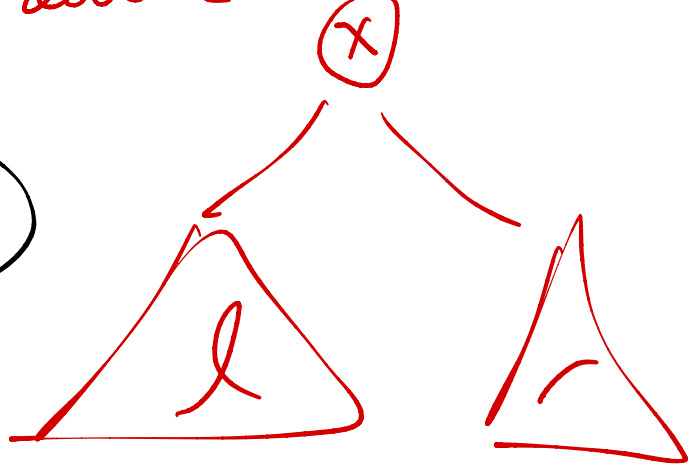
d = depth of the tree

d-1 = depth of l

d-1 = depth of r

$$\approx 1 + 2 \cdot W_{addone}(d-1)$$

is $O(2^d)$



Balanced tree:

$$\underline{\text{depth}} \approx \underline{\log_2 \text{size}}$$

$O(2^d)$ is $O(\text{size})$

```
fun addone(t: tree) : t
  case t of
    Empty => Empty
  | Node(l, x, r) => Node( addone l ,
                           x+1 ,
                           addone r )
```

$$S_{addone}(d) = k + \max\left(S_{addone}(d-1), S_{addone}(d-1)\right)$$

d: depth of the tree

depth of l

depth of r

$$\approx 1 + S_{addone}(d-1)$$

$$\underbrace{1+1+1+1+\cdots+1}_{d \text{ times}}$$

is $O(d)$

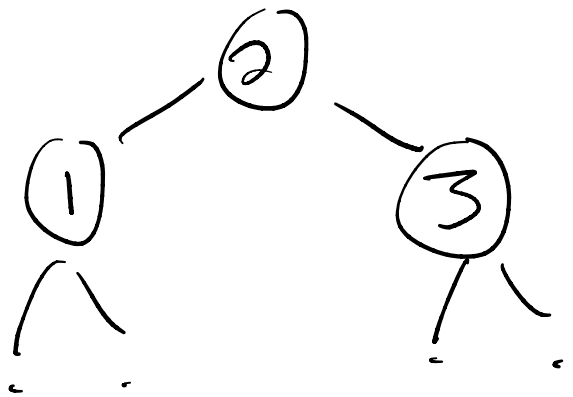
Sorting Trees

HW:
binarySearch/
contain
in $O(\log_2 \text{size})$
work

A tree is sorted iff

1. Empty, or
2. Node(l, x, r)

where
l is sorted
r is sorted
everything in $l \leq x$
everything in $r \geq x$



[1, 2, 3]

(* Goal: sort a tree with n elements with $O(n \log n)$ work and $O((\log_2 n)^k)$ span

"sublinear"

*)

```
fun mergesort (t: tree): tree =
    case t of
        Empty => Empty
      | Node(l,x,r) =>
          merge(merge (mergesort(l),
                       mergesort(r)),
                Node(Empty, x, Empty))
```

(* Goal: sort a tree with n elements

*)

$O(n \log n)$ work

$O((\log_2 n)^k)$ span

"sublinear"

```
fun mergesort (t: tree): tree =
  case t of
    Empty => Empty
  | Node(l,x,r) =>
      merge(merge (mergesort(l),
                   mergesort(r)), Node(Empty, x, Empty))
```

$$W_{ms}(n) = W_{merge} + 2\,W_{ms}\left(\frac{n}{2}\right)$$

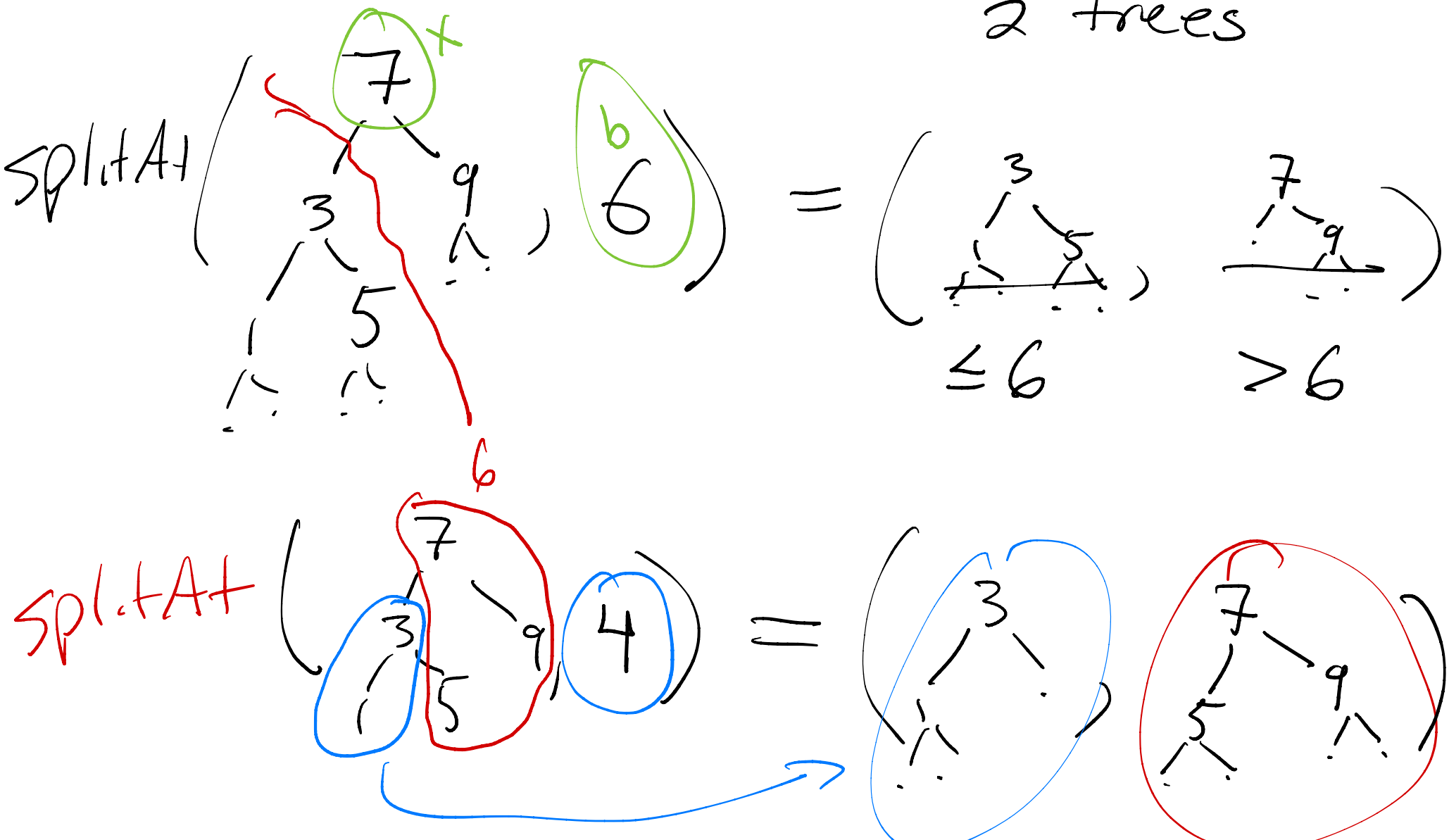
n = size, assume balanced

should be $O(n)$

$$S_{ms}(n) = S_{merge} + 1 \cdot S_{ms}\left(\frac{n}{2}\right)$$

S_{merge} → should be $O((\log_2 n)^k)$

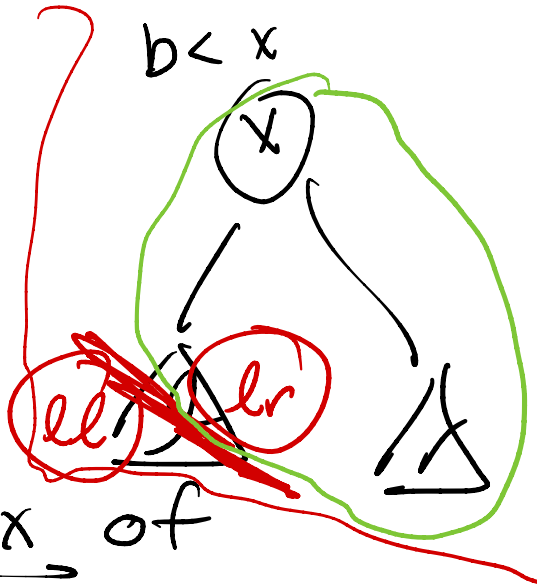
Splitting a sorted tree based on a bound



(*Spec: Given a sorted tree t
splitAt(t, b) = (l, r) where
l and r partition t
and every in l ≤ b
every in r > b *)

```
fun splitAt(t : tree, b : int) : tree * tree =
  case t of
    Empty => (Empty, Empty)
  | Node(l, x, r) => case bound < x of
      true =>
    | false =>
```

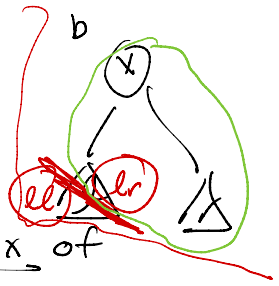
```
fun splitAt(t:tree, b:int):tree * tree =
  case t of
    Empty => (Empty, Empty)
  | Node(l,x,r) => case b < x of
      true =>
        let val (ll, lr) = splitAt(l)
        in
          (ll , Node(lr, x, r))
        end
    | false => let val(rl, rr) = split(r)
        in
          (Node(l, x, rl), rr)
        end
```



ll: $\leq b$; Node(lr, x, r): $\geq b$

Node(l, x, rl): $\leq b$; rr: $\geq b$

```
fun splitAt(t:tree, b:int):tree * tree =
  case t of
    Empty => (Empty, Empty)
  | Node(l,x,r) => case b < x of
      true =>
        let val (ll, lr) = splitAt(l)
        in
          (ll, Node(lr, x, r))
        end
    | false => let val (rl, rr) = split(r)
        in
          (Node(l,x,rl), rr)
        end
```



Tree induction

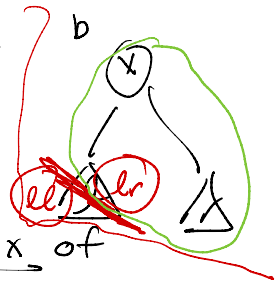
Theorem: for any sorted tree t,

splitAt(t,b) = (l,r) where

$l \leq b$

and $r > b$

```
fun splitAt(t:tree, b:int):tree * tree =
  case t of
    Empty => (Empty, Empty)
  | Node(l,x,r) => case b < x of
      true =>
        let val (ll, lr) = splitAt(l)
        in
          (ll, Node(lr, x, r))
        end
    | false => let val (rl, rr) = split(r)
        in
          (Node(l, x, rl), rr)
        end
```



ll: $\leq b$; Node(lr, x, r): $\geq b$

rr: $\geq b$; Node(l, x, rl): $\leq b$

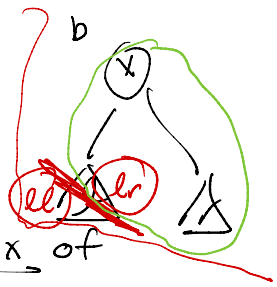
Case for Empty:

splitAt(Empty) = (Empty, Empty)

Know: Empty $\leq b$ ✓

Empty $> b$ ✓

```
fun splitAt(t:tree, b:int):tree * tree =
  case t of
    Empty => (Empty, Empty)
  | Node(l,x,r) => case b < x of
      true =>
        let val (ll, lr) = splitAt(l)
        in
          (ll, Node(lr, x, r))
        end
    | false => let val (rl, rr) = split(r)
        in
          (Node(l, x, rl), rr)
        end
```



Case for Node(l, x, r)

IH for l: split(l, b) = (ll, lr) with ll ≤ b and lr > b

IH for r: split(r, b) = (rl, rr) with rl ≤ b and rr > b

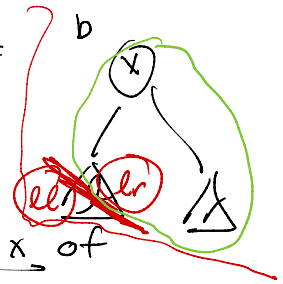
To show: splitAt() =

① ↦ (ll, Node(lr, x, r))

To show ≤ b, > b

② ↦ (Node(l, x, rl), rr)

```
fun splitAt(t:tree, b:int):tree * tree =
  case t of
    Empty => (Empty, Empty)
  | Node(l,x,r) => case b < x of
      true =>
        let val (ll, lr) = splitAt(l)
        in
          (ll, Node(lr, x, r))
        end
    | false => let val (rl, rr) = split(r)
        in
          (Node(l, x, rl), rr)
        end
```



If balanced

d is log size

$O(\log size)$

$$W_{split}(d) = k + W_{split}(d-1)$$

$$\text{Empty} \approx 1 + W_{split}(d-1)$$

$O(d)$

$$S_{split}(d) = 1 + S_{split}(d-1)$$

$O(d)$