15-150
Fall 2025

Lecture 6

Data Types - Cost Analysis

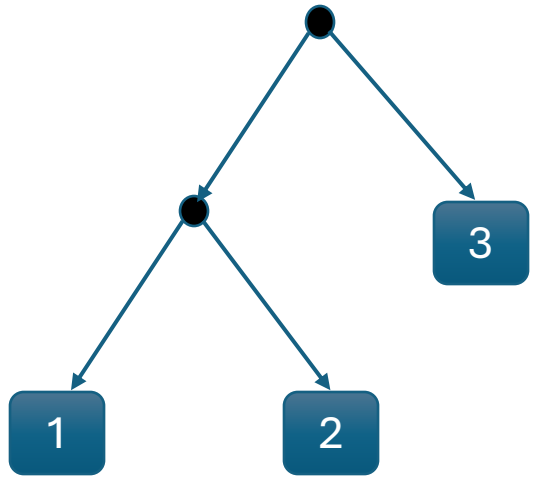
Today

- More practice with lists and trees (Part 1)
- Asymptotic cost analysis using recurrences (Part 2)

ANOTHER KIND OF TREE

A new `datatype` for trees

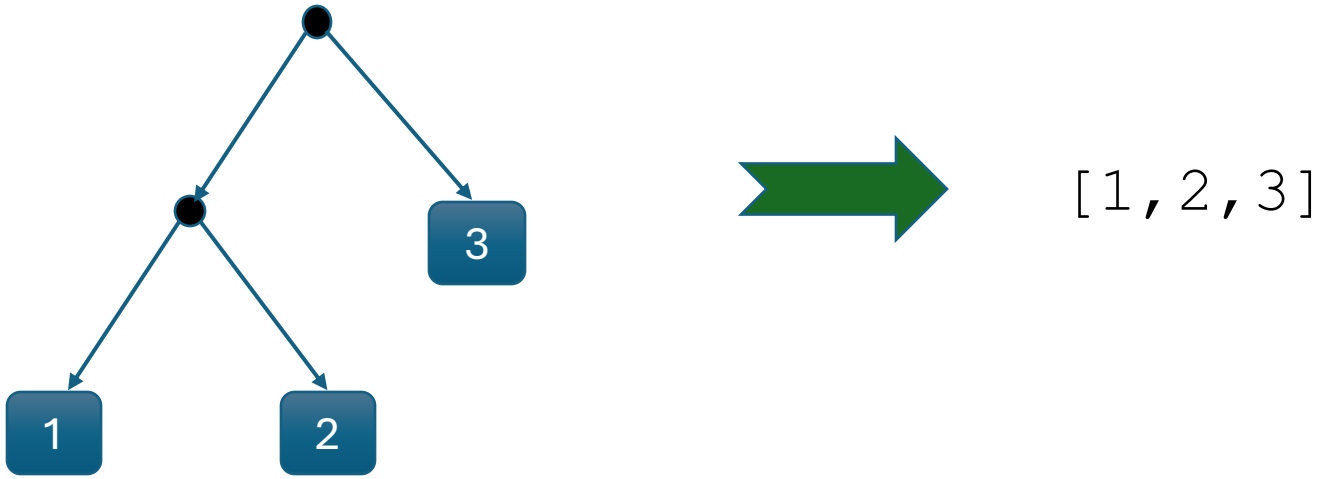
```
datatype tree = Leaf of int | Node of tree * tree
```



flatten

```
datatype tree = Leaf of int | Node of tree * tree

(* flatten : tree -> int list
   REQUIRES: true
   ENSURES: flatten(t) returns a list of the leaf
            values as they are encountered in the
            inorder traversal of t
*)
```



Appending lists

```
(* @ : int list * int list -> int list
   REQUIRES:  true
   ENSURES:  @(l,r) returns the list consisting of l
             followed by r
   NOTE: this is also predefined in SML as the right-
         associative infix  operator @.
*)

infixr (op @);

fun ([]:int list) @ (Y:int list)= Y
  | (x::xs) @ Y = x :: (xs @ Y)

val [1,2] = [] @ [1,2]
val [1,2,5,6] = [1,2] @ [5,6]
```

`[1,2] @ [3,4] @[5,6,7]` means `[1,2] @ ([3,4] @ [5,6,7])`

flatten

```
datatype tree = Leaf of int | Node of tree * tree

(* flatten : tree -> int list
   REQUIRES: true
   ENSURES: flatten(t) returns a list of the leaf
            values as they are encountered in the
            inorder traversal of t
*)
```

```
fun flatten (Leaf(x) : tree) : int list =
  | flatten (Node(t1, t2)) =
```

flatten

```
datatype tree = Leaf of int | Node of tree * tree

(* flatten : tree -> int list
   REQUIRES: true
   ENSURES: flatten(t) returns a list of the leaf
            values as they are encountered in the
            inorder traversal of t
*)
```

```
fun flatten (Leaf(x) : tree) : int list = [x]
  | flatten (Node(t1, t2)) =
```

flatten

```
datatype tree = Leaf of int | Node of tree * tree

(* flatten : tree -> int list
   REQUIRES: true
   ENSURES: flatten(t) returns a list of the leaf
            values as they are encountered in the
            inorder traversal of t
*)
```

```
fun flatten (Leaf(x) : tree) : int list = [x]
  | flatten (Node(t1, t2)) = flatten (t1) @ flatten (t2)
```

`flatten` with accumulator

```
(* flatten2 : tree * int list-> int list
   REQUIRES: true
   ENSURES: …
*)
```

`flatten` with accumulator

```
(* flatten2 : tree * int list -> int list
   REQUIRES: true
   ENSURES: flatten2(t, acc) ≅ flatten(t) @ acc
*)
```

`flatten` with accumulator

```
(* flatten2 : tree * int list -> int list
   REQUIRES: true
   ENSURES: flatten2(t, acc) ≅ flatten(t) @ acc
*)

fun flatten2 (Leaf(x), acc) =
```

`flatten` with accumulator

```
(* flatten2 : tree * int list -> int list
   REQUIRES: true
   ENSURES: flatten2(t, acc) ≅ flatten(t) @ acc
*)

fun flatten2 (Leaf(x), acc) = x :: acc
  | flatten2 …
```

`flatten` with accumulator

```
(* flatten2 : tree * int list -> int list
   REQUIRES: true
   ENSURES: flatten2(t, acc) ≅ flatten(t) @ acc
*)

fun flatten2 (Leaf(x), acc) = x :: acc
  | flatten2 (Node(t1,t2), acc) =
```

`flatten` with accumulator

```
(* flatten2 : tree * int list -> int list
   REQUIRES: true
   ENSURES: flatten2(t, acc) ≅ flatten(t) @ acc
*)

fun flatten2 (Leaf(x), acc) = x :: acc
  | flatten2 (Node(t1,t2), acc) =
               flatten2(t1,(flatten2(t2,acc)))
```

Is `flatten2` tail recursive?

`flatten` with accumulator

```
(* flatten2 : tree * int list -> int list
   REQUIRES: true
   ENSURES: flatten2(t, acc) ≅ flatten(t) @ acc
*)

fun flatten2 (Leaf(x), acc) = x :: acc
  | flatten2 (Node(t1,t2), acc) =
               flatten2(t1,(flatten2(t2,acc)))


fun flatten' (t: tree) : int list =
                                 flatten2(t,[])
```

Correctness of `flatten2`

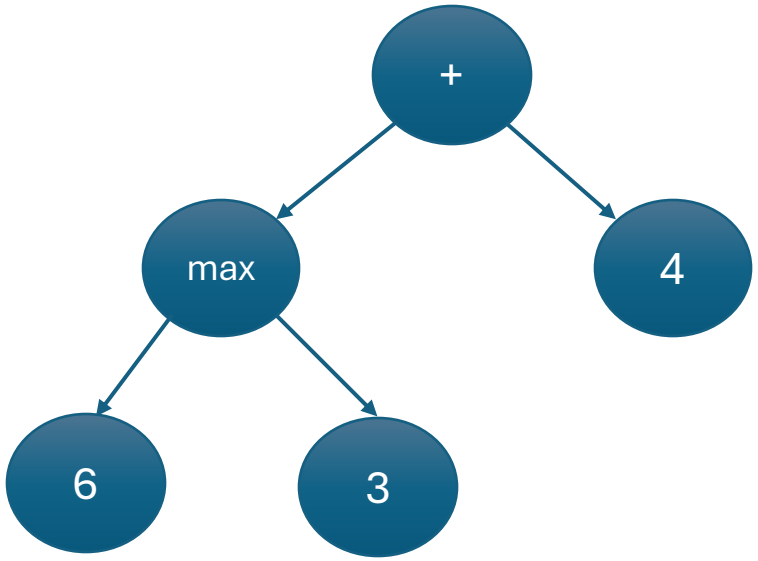
Theorem: For all values `T : tree` and `acc : int list`,

`flatten2(t,acc) ≅ flatten(t)@ acc.`

PLEASE READ THE NOTES

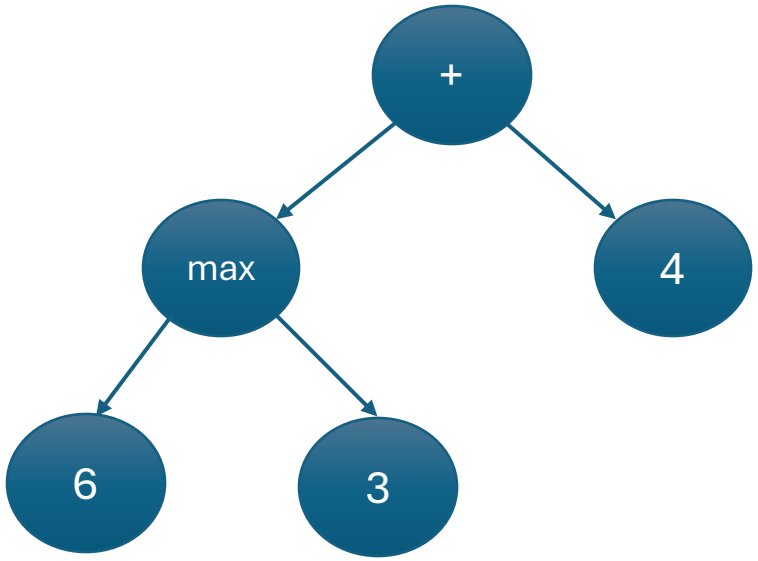
Another kind of tree

```
(Int.max(6,3)) + 4
```



Operator/operand tree

```
datatype optree = Op of optree * (int * int -> int) * optree
                | Val of int
```




```
Oper(Oper(Val 6,Int.max,Val 3),
     (fn (x,y)=>x+y),
     Val 4)
```

Could also write `op +`

Operator/operand tree

```
datatype optree = Op of optree * (int * int -> int) * optree
                | Val of int

(* eval : optree -> int
   REQUIRES: all functions in T are total
   ENSURES: eval(T) reduces to the integer value that is the
            result of the computation
            described by T (assuming post-order traversal)
*)


fun eval(Val x : optree ) : int = x
   |eval(Op(l,f,r)) =
```

Operator/operand tree

```
datatype optree = Op of optree * (int * int -> int) * optree
                | Val of int

(* eval : optree -> int
   REQUIRES: all functions in T are total
   ENSURES: eval(T) reduces to the integer value that is the
            result of the computation
            described by T (assuming post-order traversal)
*)


fun eval(Val x : optree ) : int = x
   |eval(Op(l,f,r)) = f(eval l, eval r)
```