Parallelism & Trees

15-150 M21

Lecture 0611
11 June 2021

0 Sorting, continued

Key Skill: Giving high-level algorithm descriptions

We'll be focusing on *merge sort*, which consists of the following three steps:

1. Split the input list in half
2. Sort each half
3. *merge* the sorted halves together to obtain a sorted whole

```
split : int list -> int list * int list
```
REQUIRES: true

ENSURES: `split L` evaluates to `(A,B)` where `A` and `B` differ in length by at most one, and `A@B` is a permutation of `L`

```
merge : int list * int list -> int list
```
REQUIRES: `A` and `B` are sorted

ENSURES: `merge(A,B)` evaluates to a sorted permutation of `A@B`

```
msort : int list -> int list
```
REQUIRES: true

ENSURES: `msort(L)` evaluates to a sorted permutation of `L`

`split : int list -> int list * int list`
REQUIRES: true
ENSURES: `split L` evaluates to `(A,B)` where `A` and `B` differ in length by at most one, and `A@B` is a permutation of `L`

0611.0 (sorting.sml)

```
fun split ([]):int list * int list = ([],[])
  | split ([x] : int list) = ([x],[])
  | split (x::x'::xs) =
    let
        val (A,B) = split xs
    in
      (x::A,x'::B)
    end
```

```
merge : int list * int list -> int list
```

REQUIRES: `A` and `B` are sorted

ENSURES: `merge(A,B)` evaluates to a sorted permutation of `A@B`

0611.1 (sorting.sml)

```sml
fun merge (L1:int list,[]:int list) = L1
  | merge ([],L2) = L2
  | merge (x::xs,y::ys) =
      (case Int.compare(x,y) of
          GREATER => y::merge(x::xs,ys)
        | _ => x::merge(xs,y::ys))
```

```
msort : int list -> int list
```
REQUIRES: true

ENSURES: `msort(L)` evaluates to a sorted permutation of `L`

0611.2 (sorting.sml)

```
fun msort([]:int list):int list = []
  | msort [x] = [x]
  | msort L =
      let
        val (A,B) = split L
      in
        merge(msort A,msort B)
      end
```

Analysis: `split`

Analysis: `merge`

Analysis: `msort`

1 Parallelism

Opportunity for parallelism

```
merge(msort A, msort B)
```

- Since this is functional code, there's no dependency between the evaluation of `msort A` and the evaluation of `msort B`
- An intelligent scheduler (with access to enough processors) could assign these evaluation processes to different processors, and have them calculated at the same time
- This is known as an "**opportunity for parallelism**"

```
val (x,y) = (e1,e2)
```

Opportunity for Parallelism

```
val x = e1

(*doesn't depend on x
  *)
val y = e2
```

Opportunity for Parallelism

```
val x = e1

(* DOES depend on x *)
val y = e2
```

NOT an opportunity

```
val x = case e1 of
             p1 => e2
           | ...
```

NOT an opportunity

```
val z = e1 e2
```

NOT an opportunity

- The **work** (sequential runtime) of a function is the number steps it will take to evaluate, when we do *not* take advantage of any parallelism
- The **span** (parallel runtime) of a function is the number of steps it will take to evaluate, when we take advantage of *all* opportunities for parallelism (we assume we have enough processors to do so)
- We will express both as a big-O complexity class, representing how the runtime grows as the input size grows
- We will obtain both by analyzing the code, obtaining recurrences, and solving those recurrences (using the tree method) to obtain the big-O complexity

```
val x = (e1, e2)
```

$$W_{\texttt{x}} = W_{\texttt{e1}} + W_{\texttt{e2}}$$

$$S_{\texttt{x}} = \max\left(S_{\texttt{e1}}, S_{\texttt{e2}}\right)$$

If we assume that `e1` and `e2` take approximately the same amount of time to evaluate, then

$$W_{\texttt{x}} = 2W_{\texttt{e1}} \qquad S_{\texttt{x}} = S_{\texttt{e1}} = S_{\texttt{e2}}$$

`split` doesn't have any parallelism

0611.0 (sorting.sml)

```
fun split ([]):int list * int list = ([],[])
  | split ([x] : int list) = ([x],[])
  | split (x::x'::xs) =
    let
       val (A,B) = split xs
    in
      (x::A,x'::B)
    end
```

1

$$S_{\texttt{split}}(0) = k_0$$
$$S_{\texttt{split}}(1) = k_1$$
$$\ldots + S_{\texttt{split}}(n-2)$$

0611.1 (sorting.sml)

```
fun merge (L1:int list,[]:int list) = L1
  | merge ([],L2) = L2
  | merge (x::xs,y::ys) =
      (case Int.compare(x,y) of
          GREATER => y::merge(x::xs,ys)
        | _ => x::merge(xs,y::ys))
```

1

$$S_{\texttt{merge}}(0) = k_0$$
$$S_{\texttt{merge}}(n) \le k_1 + S_{\texttt{merge}}(n-1)$$

But msort does

0611.2 (sorting.sml)

```
fun msort([]:int list):int list = []
  | msort [x] = [x]
  | msort L =
    let
       val (A,B) = split L
    in
      merge(msort A,msort B)
    end
```

1 Recurrence:

$$W_{\texttt{msort}}(0) = k_0$$
$$W_{\texttt{msort}}(1) = k_1$$
$$2W_{\texttt{msort}}(n/2) + kn$$

- Work of `msort` was $O(n \log n)$
- Making recursive calls to `msort` in parallel decreased runtime to $O(n)$ – the span
- Unable to take further advantage of parallelism, because `split` and `merge` only made one recursive call
- This is a shortcoming of `list`s themselves: they're an inherently sequential data structure and are thus limited in how much parallelism can be utilized

5-minute break

2 Trees in SML

- We define a new type `tree` with the following syntax (which we'll discuss more Monday):

0611.3 (treeDefn.sml)

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

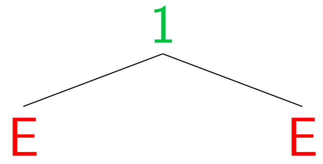
- This declares a new type called `tree` whose constructors are `Empty` and `Node`. `Empty` is a *constant constructor* because it's just a value of type `tree`. `Node` takes in an argument of type `tree*int*tree` and produces another `tree`.
- All trees are either of the form `Empty` or `Node(L,x,R)` for some `x : int` (referred to as the *root* of the tree), some `L : tree` (referred to as the *left subtree*), and some `R : tree` (referred to as the *right*

Arboretum

E

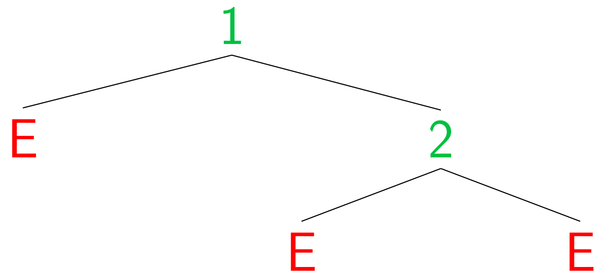
0611.9 (arboretum.sml)

```
val T0 = Empty
```



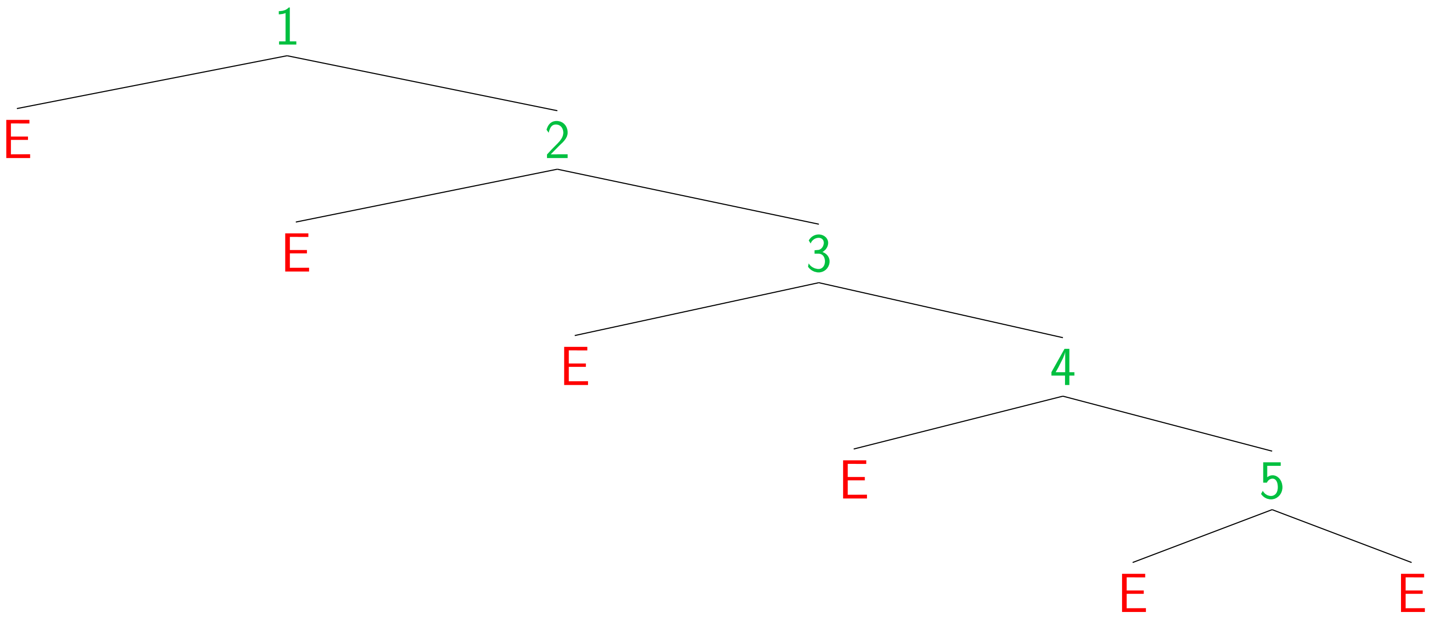
0611.10 (arboretum.sml)

```
val T1 = Node(Empty,1,Empty)
```



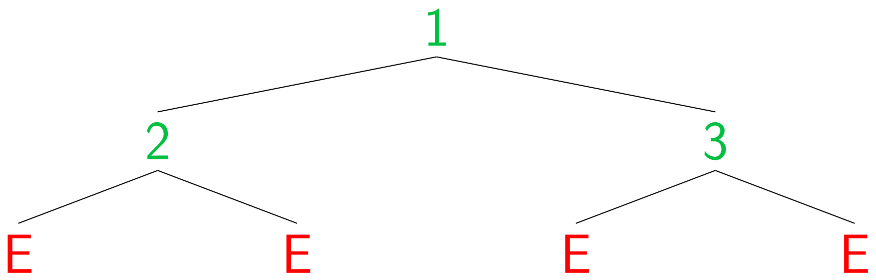
0611.11 (arboretum.sml)

```
val T2 = Node(Empty,1,Node(Empty,2,Empty))
```



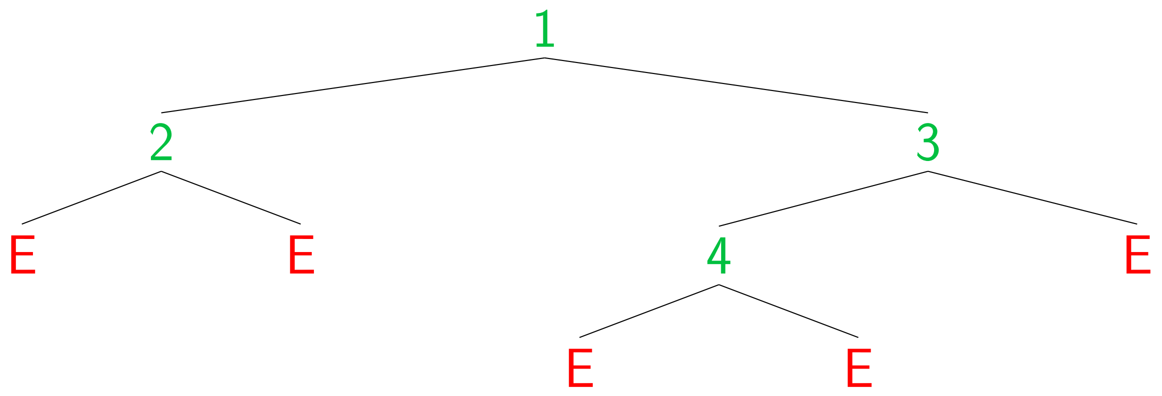
0611.12 (arboretum.sml)

```
val T3 = Node(Empty,1,Node(Empty,2,Node(Empty
  ,3,Node(Empty,4,Node(Empty,5,Empty)))))
```



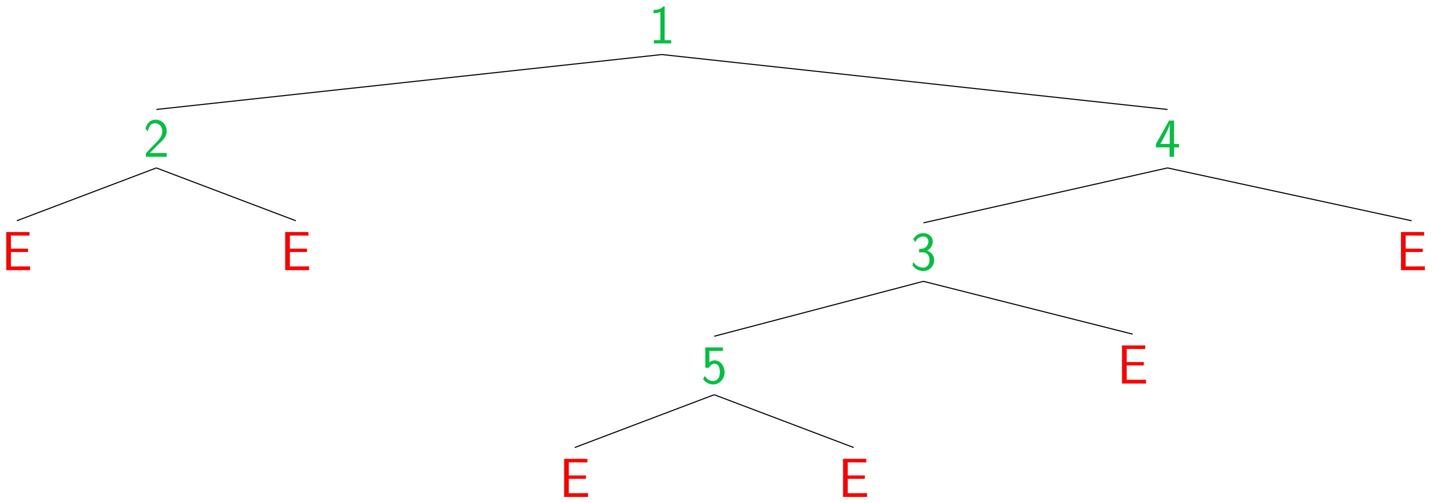
0611.13 (arboretum.sml)

```
val T4 = Node(Node(Empty,2,Empty),1,Node(Empty,3,Empty))
```



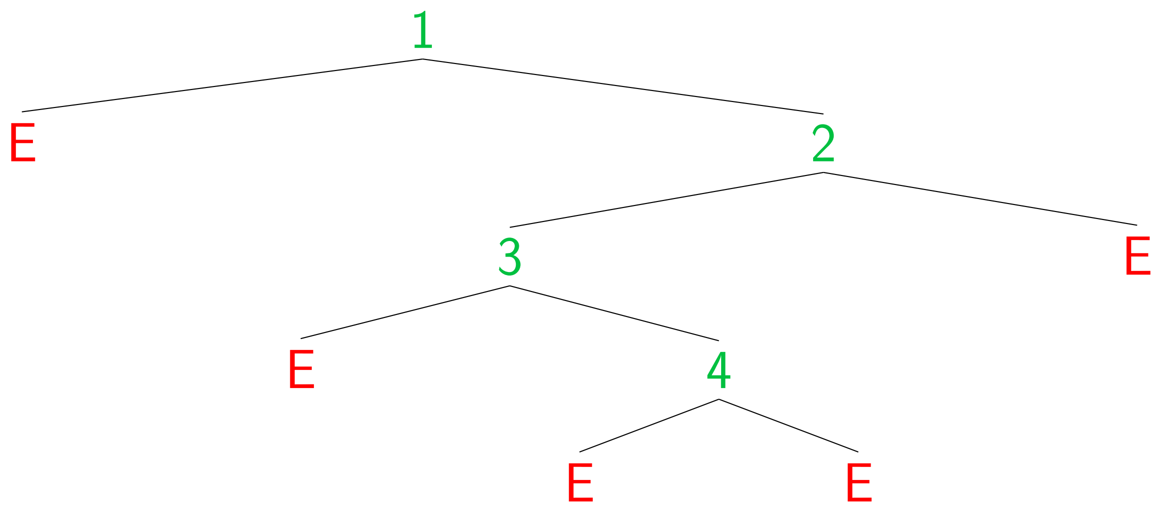
0611.14 (arboretum.sml)

```
val T5 = Node(Node(Empty,2,Empty),1,Node(Node(
  Empty,4,Empty),3,Empty))
```



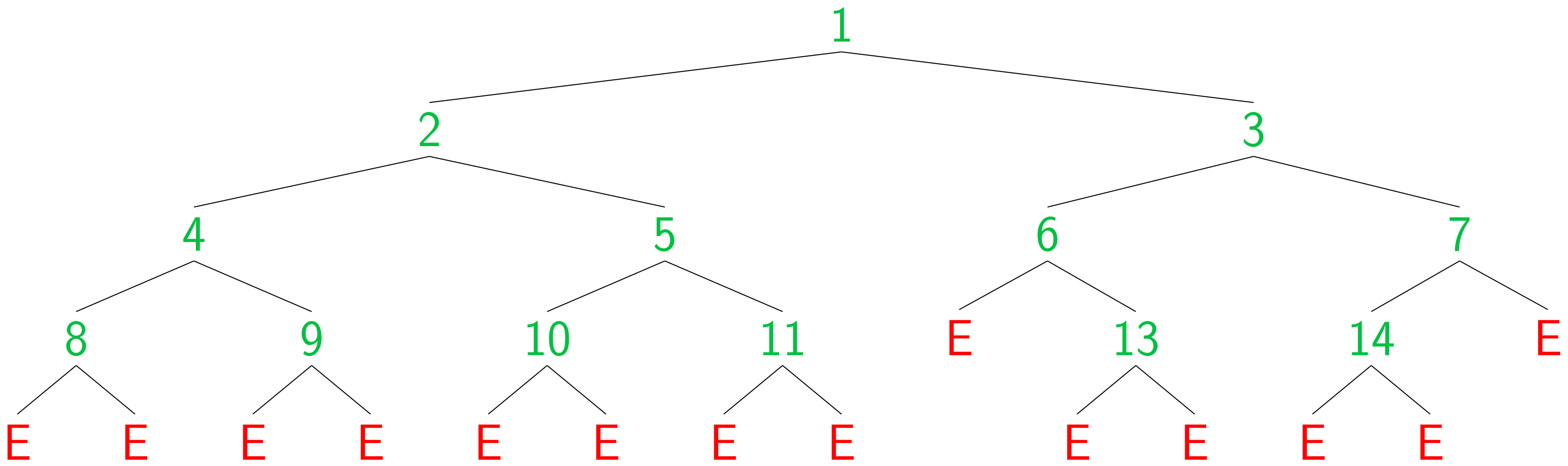
0611.15 (arboretum.sml)

```
val T6 = Node(Node(Empty,2,Empty),1,Node(Node(
  Node(Empty,5,Empty),3,Empty),4,Empty))
```



0611.16 (arboretum.sml)

```
val T7 = Node(Empty,1,Node(Node(Empty,3,Node(
  Empty,4,Empty)),2,Empty))
```



0611.17 (arboretum.sml)

```
val T8 = Node(Node(Node(Node(Empty,8,Empty),4,
  Node(Empty,9,Empty)),2,Node(Node(Empty,10,
  Empty),5,Node(Empty,11,Empty))),1,Node(Node(
  Empty,6,Node(Empty,13,Empty)),3,Node(Node(
  Empty,14,Empty),7,Empty)))
```

Height (or *depth*):

0611.4 (trees.sml)

```
fun height (Empty:tree):int = 0
  | height (Node(L,_,R)) =
      1 + Int.max(height L,height R)
```

Size

0611.5 (trees.sml)

```
fun size (Empty:tree):int = 0
  | size (Node(L,_,R)) =
      1 + size L + size R
```

Live Coding: Traversal

0611.6 (trees.sml)

```
fun inord (Empty:tree):int list = []
  | inord (Node(L,x,R)) =
      (inord L) @ (x::inord R)
```

0611.7 (trees.sml)

```
fun preord (Empty:tree):int list = []
  | preord (Node(L,x,R)) =
      x::((preord L) @ (preord R))
```

Live Coding: Minimum

`min : tree * int -> int`
REQUIRES: true
ENSURES: `min(T,default)` evaluates to the smallest value in `T`, or `default` if `T` is empty

```
fun min (Empty:tree, default:int) = default
  | min (Node(L,x,R),default) =
        Int.min(min(L,x),min(R,x))

fun min' Empty = NONE
  | min' (Node(L,x,R)) =
      (case (min' L, min' R) of
          (NONE,NONE) => SOME x
        | (NONE,SOME z) => SOME(Int.min(x,z))
        | (SOME y, NONE) => SOME(Int.min(x,y))
        | (SOME y, SOME z) =>
              SOME(Int.min(x,Int.min(y,z))))
```

When analyzing tree function, we have *two* standard notions of size:

- Depth/height, d
- Size (number of nodes), n

To simplify our analysis, we often assume the tree in question is **balanced**. A tree `Node(L,x,R)` is balanced iff

- `L` and `R` have approximately the same number of nodes
- Both `L` and `R` are balanced

A balanced tree of depth d will have approximately 2^d nodes

Demonstration: `min` runtime analysis

0 Notion of size: depth d of the input tree

1 Recurrences:

$$W_{\texttt{min}}(0) = k_0$$
$$W_{\texttt{min}}(d) \leq k_1 + 2W_{\texttt{min}}(d-1)$$

$$S_{\texttt{min}}(0) = k_0$$
$$S_{\texttt{min}}(d) \leq k_1 + S_{\texttt{min}}(d-1)$$

2-4 ...

5 $W_{\texttt{min}}(d)$ is $O(2^d)$, $S_{\texttt{min}}(d)$ is $O(d)$

If the input tree is **balanced**, then $2^d \approx n$, where n is the size (number of nodes)

Demonstration: `preord` runtime analysis

Size-Analysis of `preord`

0 Notion of size: number of nodes n of the input

1 Recurrences:

$$\begin{aligned} W_{\texttt{preord}}(0) &= k_0 \\ W_{\texttt{preord}}(n) &= 2W_{\texttt{preord}}(n/2) + kn \end{aligned}$$

NOTE: This assumes the tree is balanced

$$\begin{aligned} S_{\texttt{preord}}(0) &= k_0 \\ S_{\texttt{preord}}(n) &\leq S_{\texttt{preord}}(n/2) + kn \end{aligned}$$

2-4 ...

5 $W_{\texttt{preord}}(n)$ is $O(n \log n)$, $S_{\texttt{preord}}(n)$ is $O(n)$

- We can implement and analyze sorting algorithms using the tools we've developed so far
- We can identify opportunities for parallelism and analyze how fast the code would run if a scheduler could take advantage of all such opportunities.
- We can encode binary `int` trees in SML, write functions operating on them, and analyze their parallel & sequential runtimes
- Trees typically have more opportunities for parallelism than lists

Next Time

- Tree Search
- Structural Induction on Trees
- Custom Datatypes
- Parametrized Polymorphism

Thank you!