Lecture 6
Principles of Functional Programming
Summer 2020

Parallelism and Trees

Section 1

Recap: Work Analysis of Recursive Functions

The Tree Method

The Tree Method

0 How you're quantifying input size

The Tree Method

0 How you're quantifying input size

1 Recurrence

The Tree Method

0 How you're quantifying input size
1 Recurrence
2 Description of work tree

The Tree Method

0 How you're quantifying input size

1 Recurrence

2 Description of work tree

3 Measurements of work tree (height, and width at each level)

The Tree Method

0 How you're quantifying input size

1 Recurrence

2 Description of work tree

3 Measurements of work tree (height, and width at each level)

4 Summation

The Tree Method

0 How you're quantifying input size

1 Recurrence

2 Description of work tree

3 Measurements of work tree (height, and width at each level)

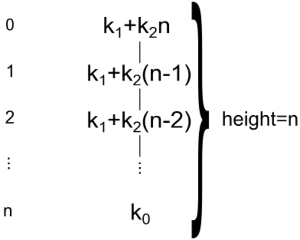
4 Summation

5 Big-O

1. Recurrence:

$$W(0) = k_0$$
$$W(n) = k_1 + k_2 n + W(n-1)$$

2. Work Tree



3. Measurements

   Height: n Work on the i-th level: $k_1 + k_2(n-i)$

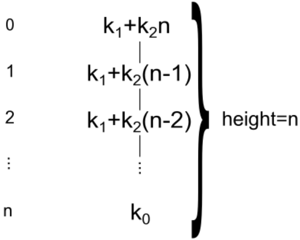
4. Sum:

$$W(n) \approx k_0 + \sum_{i=0}^{n} (k_1 + k_2(n-i)) = \dots$$

1. Recurrence:

$$W(0) = k_0$$
$$W(n) = k_1 + k_2 n + W(n-1)$$

2. Work Tree



3. Measurements
   Height: n Work on the i-th level: $k_1 + k_2(n-i)$
4. Sum:

$$W(n) \approx k_0 + \sum_{i=0}^{n} (k_1 + k_2(n-i)) = \dots$$

5. Big O:

$$W(n) \text{ is } O(n^2)$$

Section 2

Asymptotic Analysis of Multi-Step Algorithms

Sorting

Sorting is a classic algorithmic problem in computer science: finding the fastest way to put all the elements of a list in order.

Sorting

Sorting is a classic algorithmic problem in computer science: finding the fastest way to put all the elements of a list in order.

A value `[x_1,...,x_n] : int list` is **sorted** if for each $i = 1, \ldots, n-1$, `Int.compare(x_i,x_(i+1))` $\ncong$ `GREATER`.

Sorting

Sorting is a classic algorithmic problem in computer science: finding the fastest way to put all the elements of a list in order.

A value `[x_1,...,x_n] : int list` is **sorted** if for each $i = 1, \ldots, n-1$, `Int.compare(x_i,x_(i+1))` $\ncong$ `GREATER`.

Or, recursively: a value `v:int list` is **sorted** if either `v=[]` or `v=[x]` for some `x`, or `v=x::x'::xs` where `Int.compare(x,x')` $\ncong$ `GREATER` and `x'::xs` is sorted.

Spec

6.0

```
fun isSorted ([]:int list):bool = true
  | isSorted [x] = true
  | isSorted (x::x’::xs) =
          (x<=x’) andalso isSorted(x’::xs)
```

Spec

6.0

```
fun isSorted ([]:int list):bool = true
  | isSorted [x] = true
  | isSorted (x::x'::xs) =
          (x<=x') andalso isSorted(x'::xs)
```

`sort : int list -> int list`
REQUIRES: true
ENSURES: `sort(L)` evaluates to a sorted permutation of `L`

A "permutation" of `L` is just a list that contains the same elements the same number of times as `L`, just in a possibly different order. So `[1,1,2,3]` is a permutation of `[3,1,2,1]` but not of `[3,2,1]`.

Sorting Algorithms

There are many sorting algorithms: insertion sort, quick sort, merge sort, bubble sort, . . .

Sorting Algorithms

There are many sorting algorithms: insertion sort, quick sort, merge sort, bubble sort, . . .

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

Sorting Algorithms

There are many sorting algorithms: insertion sort, quick sort, merge sort, bubble sort, . . .

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

1. Split the input list in half

Sorting Algorithms

There are many sorting algorithms: insertion sort, quick sort, merge sort, bubble sort, . . .

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

Sorting Algorithms

There are many sorting algorithms: insertion sort, quick sort, merge sort, bubble sort, . . .

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

Specs

`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`

Specs

`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`

Specs

`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`

```
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`

split

`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`

6.1

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

`merge`

```
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`

merge

`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`

6.2

```
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`

```
msort : int list -> int list
```

REQUIRES: true

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

msort

`msort : int list -> int list`
REQUIRES: true
ENSURES: `msort(L)` evaluates to a sorted permutation of `L`

6.3

```
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

6.1

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

Analysis

6.1

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

0 Measure of size: length of input list

Analysis

6.1

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

0 Measure of size: length of input list

1

$$W_{\texttt{split}}(0) = k_0$$
$$W_{\texttt{split}}(1) = k_1$$
$$W_{\texttt{split}}(n) = k_2 + W_{\texttt{split}}(n-2)$$

Analysis

0 Measure of size: length of input list

1

$$
\begin{aligned}
W_{\texttt{split}}(0) &= k_0 \\
W_{\texttt{split}}(1) &= k_1 \\
W_{\texttt{split}}(n) &= k_2 + W_{\texttt{split}}(n-2)
\end{aligned}
$$

Analysis

0 Measure of size: length of input list

1

$$
\begin{aligned}
W_{\texttt{split}}(0) &= k_0 \\
W_{\texttt{split}}(1) &= k_1 \\
W_{\texttt{split}}(n) &= k_2 + W_{\texttt{split}}(n-2)
\end{aligned}
$$

2-4 . . .

5 $W_{\texttt{split}}(n)$ is $O(n)$

Analysis

0 Measure of size: length of input list

1

$$\begin{aligned} W_{\texttt{split}}(0) &= k_0 \\ W_{\texttt{split}}(1) &= k_1 \\ W_{\texttt{split}}(n) &= k_2 + W_{\texttt{split}}(n-2) \end{aligned}$$

2-4 . . .

5 $W_{\texttt{split}}(n)$ is $O(n)$

Analysis

6.2

```
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))
```

Analysis

6.2

```
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))
```

0 Measure of size: sum of lengths of input lists

Analysis

6.2

```
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))
```

0 Measure of size: sum of lengths of input lists

1

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

Analysis

6.2

```
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))
```

0 Measure of size: sum of lengths of input lists

1

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

2-4 . . .

5 $W_{\texttt{merge}}(n)$ is $O(n)$

Analysis

6.2

```
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))
```

0 Measure of size: sum of lengths of input lists

1

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

2-4 ...

5 $W_{\texttt{merge}}(n)$ is $O(n)$

Analysis

6.4

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

Analysis

6.4

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

0 Measure of size: length of input list

Analysis

6.4

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

0 Measure of size: length of input list

1

$$W_{\texttt{msort}}(0) = k_0$$
$$W_{\texttt{msort}}(1) = k_1$$
$$W_{\texttt{msort}}(n) \leq k_2 + k_3 n + W_{\texttt{msort}}\left(\frac{n}{2}\right) + W_{\texttt{msort}}\left(n - \frac{n}{2}\right) + k_4 n$$
$$\approx 2W_{\texttt{msort}}(n/2) + kn$$

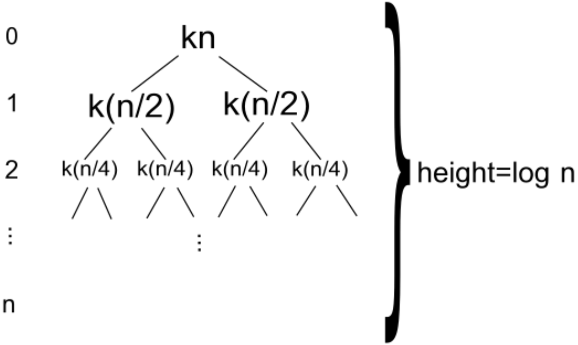
1 Recurrence:

$$\begin{aligned} W_{\texttt{msort}}(0) &= k_0 \\ W_{\texttt{msort}}(1) &= k_1 \\ W_{\texttt{msort}}(n) &\leq 2W_{\texttt{msort}}(n/2) + kn \end{aligned}$$

1. Recurrence:

$$\begin{aligned} W_{\texttt{msort}}(0) &= k_0 \\ W_{\texttt{msort}}(1) &= k_1 \\ W_{\texttt{msort}}(n) &\leq 2W_{\texttt{msort}}(n/2) + kn \end{aligned}$$

2. Work Tree

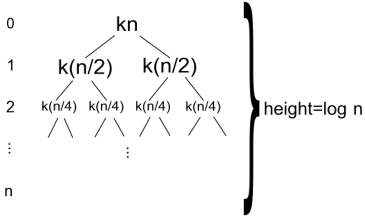


1 Recurrence:

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

1 Recurrence:

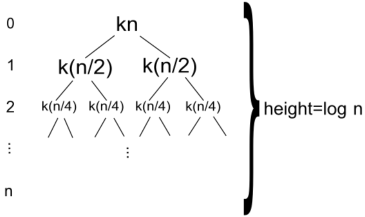
$$W_{\texttt{msort}}(0) = k_0$$
$$W_{\texttt{msort}}(1) = k_1$$
$$W_{\texttt{msort}}(n) \leq 2W_{\texttt{msort}}(n/2) + kn$$



2 Work Tree

1 Recurrence:

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



2 Work Tree

3 Measurements

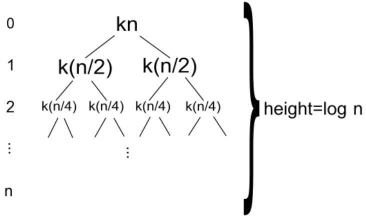
Height: $\log n$ Work on the i-th level: $2^i \frac{kn}{2^i} = kn$

1. Recurrence:

$$W_{\texttt{msort}}(0) = k_0$$

$$W_{\texttt{msort}}(1) = k_1$$

$$W_{\texttt{msort}}(n) \leq 2W_{\texttt{msort}}(n/2) + kn$$



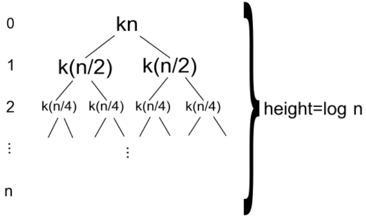
2. Work Tree
3. Measurements

   Height: $\log n$ Work on the i-th level: $2^i \frac{kn}{2^i} = kn$
4. Sum:

$$W(n) \approx \sum_{i=0}^{\log n} kn$$

1. Recurrence:

$$\begin{aligned} W_{\texttt{msort}}(0) &= k_0 \\ W_{\texttt{msort}}(1) &= k_1 \\ W_{\texttt{msort}}(n) &\leq 2W_{\texttt{msort}}(n/2) + kn \end{aligned}$$



2. Work Tree
3. Measurements

   Height: $\log n$ Work on the i-th level: $2^i \frac{kn}{2^i} = kn$

4. Sum:

$$W(n) \approx \sum_{i=0}^{\log n} kn$$

5. Big O:

$$W(n) \text{ is } O(n \log n)$$

(pause for questions)

Section 3

Parallel Cost Analysis

Opportunity for parallelism

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

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`

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

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

Opportunity for Parallelism

```
val x = e1

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

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,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 (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

```
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

Work and Span

Work and Span

- 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

Work and Span

- 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)

Work and Span

- 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

Work and Span

- 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

Calculating Work and Span

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

Calculating Work and Span

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

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

Calculating Work and Span

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

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

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

Calculating Work and Span

```
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

6.1

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

1

$$
\begin{aligned}
S_{\texttt{split}}(0) &= k_0 \\
S_{\texttt{split}}(1) &= k_1 \\
S_{\texttt{split}}(n) &= k_2 + S_{\texttt{split}}(n-2)
\end{aligned}
$$

split doesn't have any parallelism

6.1

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

1

$$S_{\texttt{split}}(0) = k_0$$
$$S_{\texttt{split}}(1) = k_1$$
$$S_{\texttt{split}}(n) = k_2 + S_{\texttt{split}}(n-2)$$

5 $S_{\texttt{split}}(n)$ is $O(n)$

merge doesn't either

6.2

```
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

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

`merge` doesn't either

6.2

```
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

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

5 $S_{\texttt{merge}}(n)$ is $O(n)$

But `msort` does

1. Recurrence:

$$
\begin{aligned}
W_{\texttt{msort}}(0) &= k_0 \\
W_{\texttt{msort}}(1) &= k_1 \\
W_{\texttt{msort}}(n) &\leq 2W_{\texttt{msort}}(n/2) + kn
\end{aligned}
$$

But `msort` does

1. Recurrence:

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

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

But `msort` does

1. Recurrence:

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

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

2. Work Tree...
3. Measurements
   Height: $\log n$ Span on the i-th level: $\frac{kn}{2^i}$

But `msort` does

1 Recurrence:

$$\begin{aligned} W_{\texttt{msort}}(0) &= k_0 \\ W_{\texttt{msort}}(1) &= k_1 \\ W_{\texttt{msort}}(n) &\le 2W_{\texttt{msort}}(n/2) + kn \end{aligned}$$

$$\begin{aligned} S_{\texttt{msort}}(0) &= k_0 \\ S_{\texttt{msort}}(1) &= k_1 \\ S_{\texttt{msort}}(n) &\le S_{\texttt{msort}}(n/2) + kn \end{aligned}$$

2 Work Tree...

3 Measurements

Height: $\log n$ Span on the i-th level: $\frac{kn}{2^i}$

4&5 Sum:

$$S(n) \approx \sum_{i=0}^{\log n} \frac{kn}{2^i} \le \sum_{i=0}^{\infty} \frac{kn}{2^i} = 2kn = O(n)$$

Conclusions

- Work of `msort` was $O(n \log n)$
- Making recursive calls to `msort` in parallel decreased runtime to $O(n)$ – the span

Conclusions

- 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

Conclusions

- 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

(pause for questions)

Section 4

Trees

Binary trees in SML

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

Binary trees in SML

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

6.5

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

Binary trees in SML

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

6.5

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

- This declares a new type called `tree` whose constructors are `Empty` and `Node`.

Binary trees in SML

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

6.5

```
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`.

Binary trees in SML

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

6.5

```
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`.

Binary trees in SML

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

6.5

```
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 subtree*)

Basic Quantities

Height (or *depth*):

6.6

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

Basic Quantities

Height (or *depth*):

6.6

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

Size

6.7

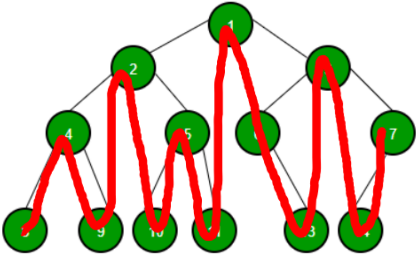
```
fun size (Empty:tree):int = 0
  | size (Node(L,_,R)) =
      1 + size L + size R
```

Traversals

Inorder

6.9

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

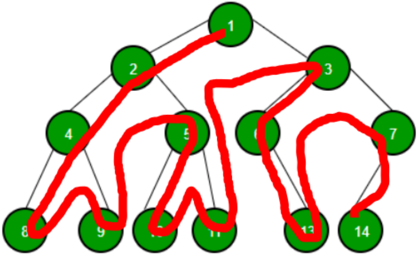


Traversals

Preorder

6.10

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



Minimum

6.11

```
fun min (Empty:tree, default:int) = default
  | min (Node(L,x,R),default) =
     let
       (* Parallel *)
       val (minL,minR) =
          (min(L,default), min(R,default))
     in
       (* Constant-time *)
       Int.min(x,Int.min(minL,minR))
     end
```

Depth-Analysis of `min`

0 Notion of size: depth d of the input tree

1 Recurrences:

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

Depth-Analysis of `min`

0. Notion of size: depth d of the input tree
1. Recurrences:

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

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

Depth-Analysis of `min`

0 Notion of size: depth d of the input tree

1 Recurrences:

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

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

2-4 . . .

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

Depth-Analysis of `min`

0 Notion of size: depth d of the input tree

1 Recurrences:

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

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

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)

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

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

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

Size-Analysis of `preord`

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

1 Recurrences:

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

NOTE: This assumes the tree is balanced

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

2-4 . . .

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

Upcoming lectures

- Thursday: Structural induction on trees, datatypes
- Friday: Sorting with trees

Section 5

Lecture 6.5 : Trees in SML

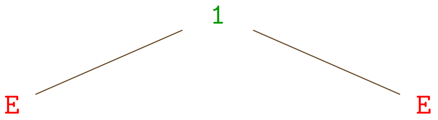
Section 6

Arboretum

`E`

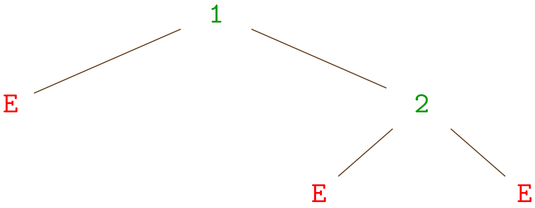
6.12

```
Empty
```



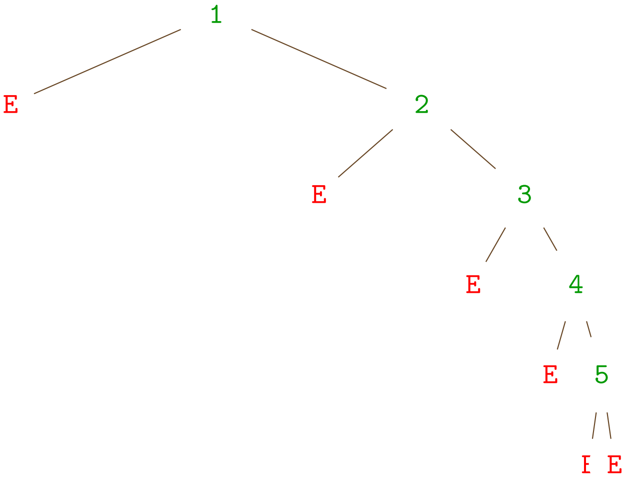
6.13

```
Node(Empty,1,Empty)
```



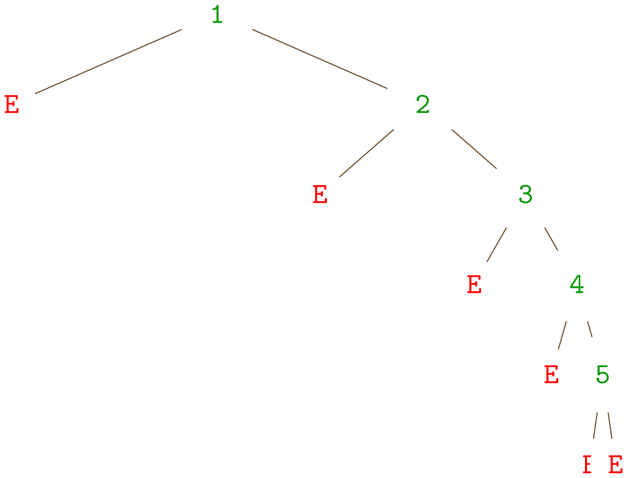
6.14

```
Node(Empty,1,Node(Empty,2,Empty))
```



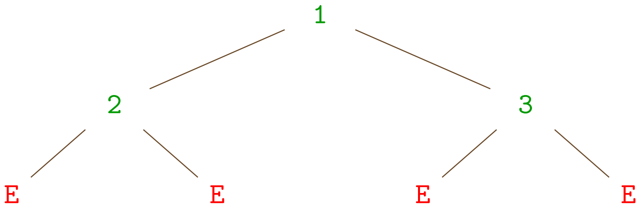
6.15

```
Node(Empty,1,Node(Empty,2,Node(Empty,3,Node(
    Empty,4,Node(Empty,5,Empty)))))
```



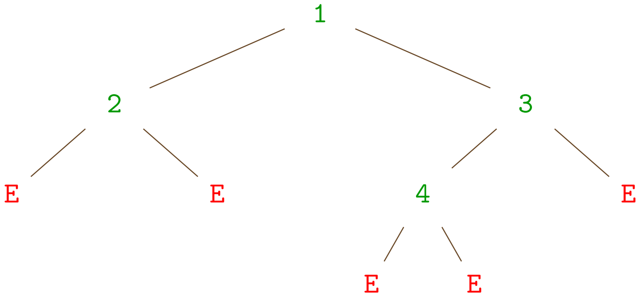
6.15

```
Node(Empty,1,Node(Empty,2,Node(Empty,3,Node(
    Empty,4,Node(Empty,5,Empty)))))
```



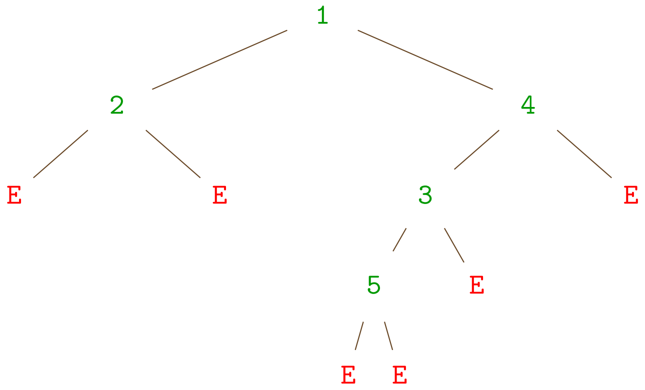
6.16

```
Node(Node(Empty,2,Empty),1,Node(Empty,3,Empty))
```



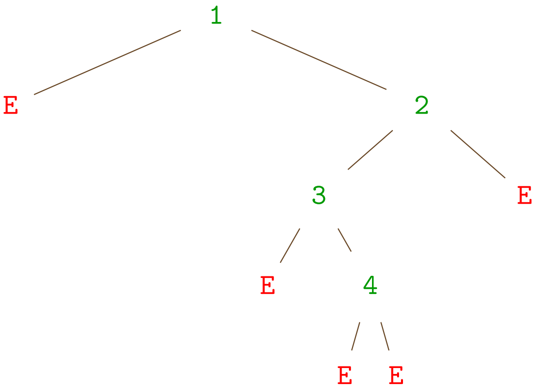
6.17

```
Node(Node(Empty,2,Empty),1,Node(Node(Empty,4,Empty),3,Empty))
```



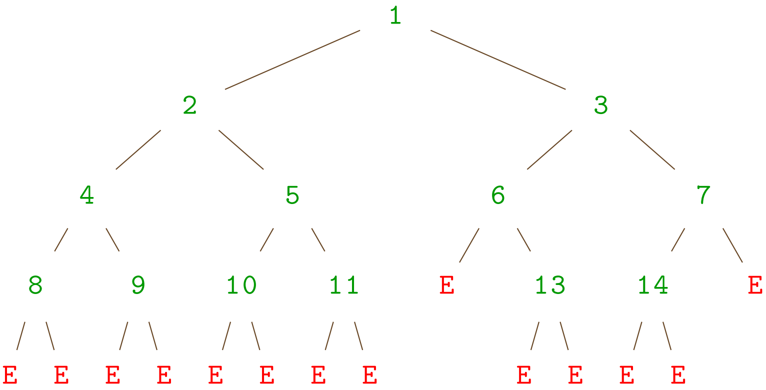
6.18

```
Node(Node(Empty,2,Empty),1,Node(Node(Node(Empty,5,Empty),3,Empty),4,Empty))
```



6.19

```
Node(Empty,1,Node(Node(Empty,3,Node(Empty,4,
    Empty)),2,Empty))
```



6.20

```
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)))
```

Thank you!