15-150

Principles of Functional Programming

Lecture 8

February 6, 2025

Michael Erdmann

Parallel Sorting

Recall:

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

Int.compare : int * int -> order
```

```
datatype order =
    LESS
  | EQUAL
  | GREATER
```

We define trees to be sorted by:

- Empty is sorted;
- Node(ℓ, x, r) is sorted iff

  (i) ℓ is sorted and for every y : int in ℓ, Int.compare (y, x) returns either LESS or EQUAL,

  and (ii) r is sorted and for every z : int in r, Int.compare (z, x) returns either GREATER or EQUAL.

Let's try divide & conquer for sorting trees:

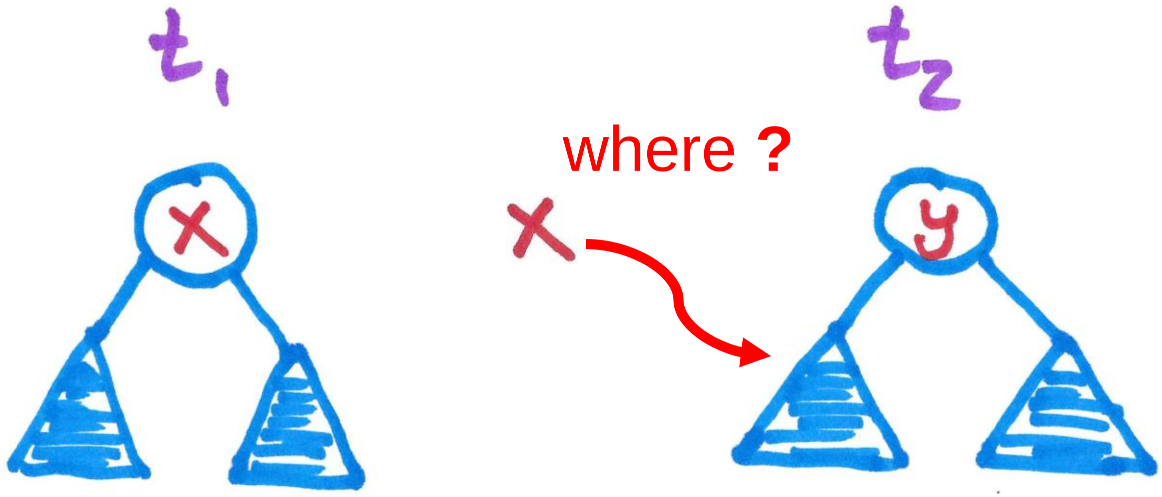
- Split the tree into subtrees
- Sort the subtrees
- Merge the results

```
(* Msort : tree → tree
   REQUIRES: true
   ENSURES:  Msort(t) returns a sorted tree
             containing exactly the elements of
             t (including duplicates).
*)

fun Msort Empty = Empty
  | Msort (Node(l, x, r)) =
      Ins(x, Merge(Msort l, Msort r))
```

```
(* Ins : int * tree -> tree
   REQUIRES: t is sorted.
   ENSURES:  Ins(x, t) returns a sorted tree
             containing x along with the
             elements of t (including duplicates).
*)

fun Ins (x, Empty) = Node(Empty, x, Empty)
  | Ins (x, Node(l, y, r)) =
    (case Int.compare (x, y) of
          GREATER => Node(l, y, Ins(x, r))
        | _       => Node(Ins(x, l), y, r))
```

Issue:

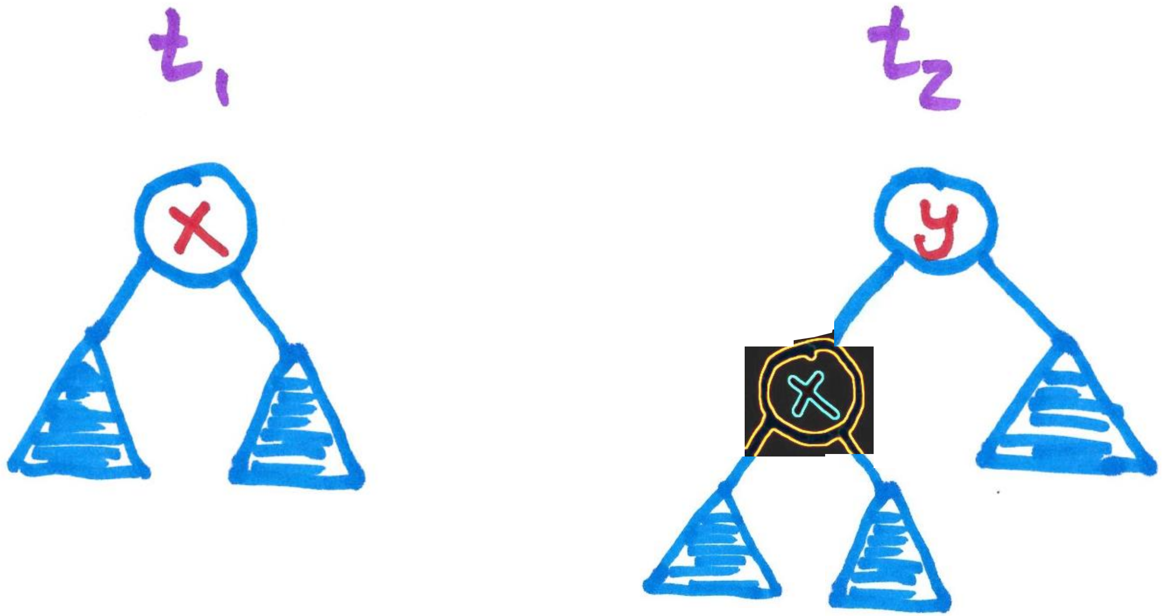
When merging two trees, the root elements may be different:



To obtain parallelism, it is useful to split t_2 at the location x would appear in t_2 (continuing to assume sorted trees), rather than at y.

Issue:

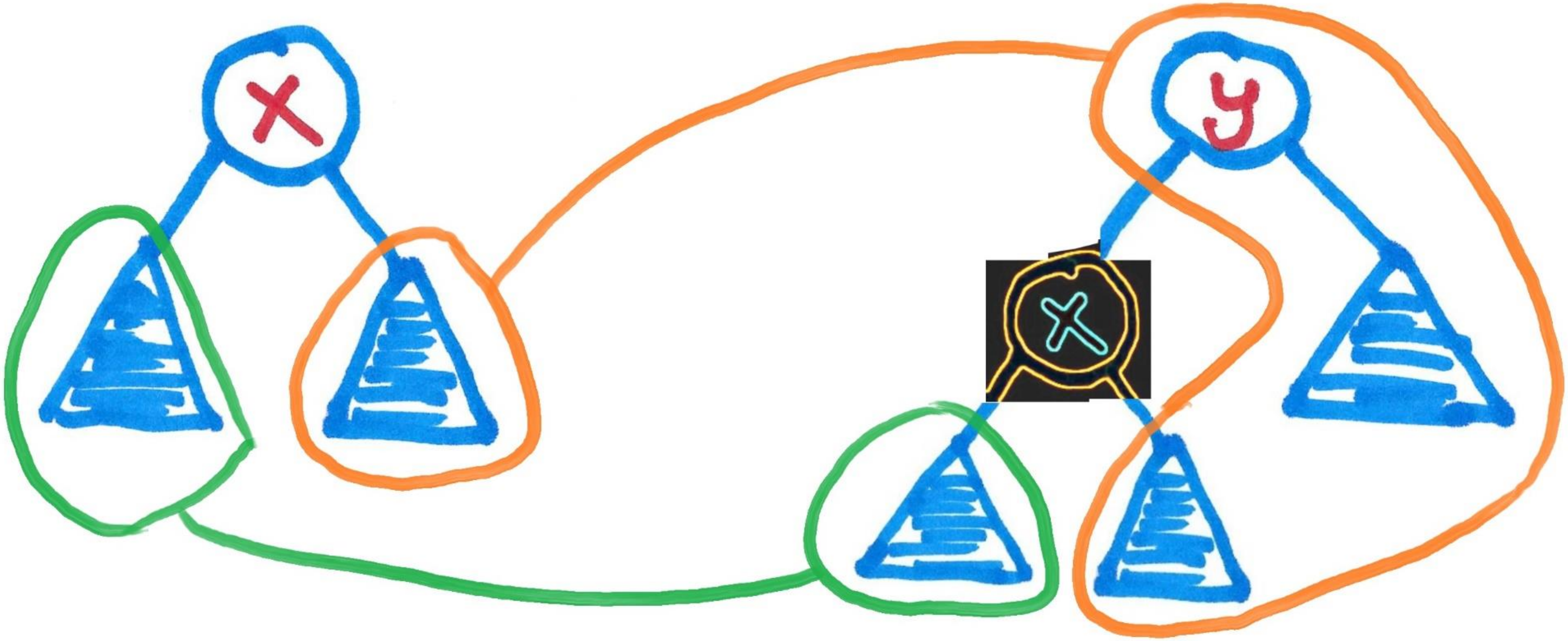
When merging two trees, the root elements may be different:



To obtain parallelism, it is useful to split t_2 at the location x would appear in t_2 (continuing to assume sorted trees), rather than at y.

Trees to merge after the split

Merge the two trees enclosed by the orange curve.



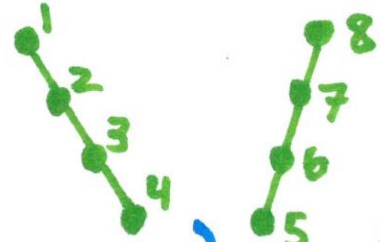
Merge the two trees enclosed by the green curve.

And then create a node with **x** and the two merged pairs of trees.

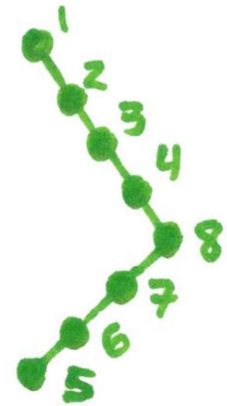
```
(* Merge : tree * tree → tree
   REQUIRES: t1 & t2 are sorted.
   ENSURES: Merge(t1, t2) returns a sorted
            tree containing exactly the elements
            of t1 & t2 together (incl. dups.).
*)

fun Merge (Empty, t2) = t2
  | Merge (Node (l1, x, r1), t2) =
    let
       val (l2, r2) = SplitAt (x, t2)
    in
       Node (Merge (l1, l2), x, Merge(r1, r2))
    end
```

Caution:

The depth of Merge(t_1, t_2) can be the sum of the depths of t_1 & t_2.

Example: Merge(

)

$\Longrightarrow$ 

This means Merge(t_1, t_2) may not be balanced even if t_1 & t_2 are balanced.

Consequently:

In order to obtain fast code, one must rebalance.

One can do so without affecting asymptotic cost compared to what we will do today when we assume trees are balanced.

The details are beyond today's lecture.

```
fun Msort Empty = Empty
  | Msort (Node(ℓ, x, r)) =
      Ins(x, Merge(Msort ℓ, Msort r))
```

```
fun Merge (Empty, t2) = t2
  | Merge (Node(ℓ1, x, r1), t2) =
    let
      val (ℓ2, r2) = SplitAt(x, t2)
    in
      Node(Merge(ℓ1, ℓ2), x, Merge(r1, r2))
    end
```

```
fun Msort Empty = Empty
  | Msort (Node(l, x, r)) =
rebalance(Ins(x, Merge(Msort l, Msort r)))
```

```
fun Merge (Empty, t2) = t2
  | Merge (Node(l1, x, r1), t2) =
    let
      val (l2, r2) = SplitAt(x, t2)
    in
      Node(Merge(l1, l2), x, Merge(r1, r2))
    end
```

(* SplitAt : int * tree → tree * tree

REQUIRES : t is sorted.

ENSURES : SplitAt (x, t) returns a pair (t_1, t_2) of sorted trees

such that:

- t_1 & t_2 together contain exactly the elements of t (incl. dups.).
- The elements of t_1 are LESS or EQUAL to x.
- The elements of t_2 are GREATER or EQUAL to x.

*)

```
fun SplitAt (x, Empty) = (Empty, Empty)
  | SplitAt (x, Node (l, y, r)) =
    (case Int.compare (x, y) of
       LESS => let
                 val (t1, t2) = SplitAt (x, l)
               in
                 (t1, Node (t2, y, r))
               end
     | _ => let
              val (t1, t2) = SplitAt (x, r)
            in
              (Node (l, y, t1), t2)
            end)
```

Analysis

- Assume balanced trees
- Focus on span
- Write just the recursive part of the recurrence

(we will phrase this in terms of the depths d of the input trees)

```
fun Ins (x, Empty) = Node(Empty, x, Empty)
  | Ins (x, Node (l, y, r)) =
    (case Int.compare (x, y) of
          GREATER => Node (l, y, Ins(x, r))
        | _       => Node (Ins(x, l), y, r))
```

```
fun Ins (x, Empty) = Node(Empty, x, Empty)
  | Ins(x, Node(l, y, r)) =
    (case Int.compare (x, y) of
          GREATER => Node(l, y, Ins(x, r))
        | _       => Node(Ins(x, l), y, r))
```

$$S_{Ins}(d) \le c_1 + S_{Ins}(d-1)$$

```
fun Ins (x, Empty) = Node(Empty, x, Empty)
  | Ins (x, Node (l, y, r)) =
    (case Int.compare (x, y) of
          GREATER => Node (l, y, Ins(x, r))
        | _       => Node (Ins (x, l), y, r))
```

$$S_{Ins}(d) \leq c_1 + S_{Ins}(d-1),$$

so $S_{Ins}(d)$ is $O(d)$.

```
fun SplitAt (x, Empty) = (Empty, Empty)
  | SplitAt (x, Node (l, y, r)) =
    (case Int.compare (x, y) of
       LESS => let
                 val (t1, t2) = SplitAt (x, l)
               in
                 (t1, Node (t2, y, r))
               end
     | _ => •••                                )
```

```
fun SplitAt (x, Empty) = (Empty, Empty)
  | SplitAt (x, Node (l, y, r)) =
    (case Int.compare (x, y) of
       LESS => let
                 val (t1, t2) = SplitAt (x, l)
               in (t1, Node (t2, y, r))
               end
     | _ => •••                                  )
```

$$S_{SplitAt}(d) \leq c_2 + S_{SplitAt}(d-1)$$

```
fun SplitAt (x, Empty) = (Empty, Empty)
  | SplitAt (x, Node (l, y, r)) =
    (case Int.compare (x, y) of
       LESS => let
                 val (t1, t2) = SplitAt (x, l)
               in
                 (t1, Node (t2, y, r))
               end
     | _ => •••                                  )
```

$$S_{\text{SplitAt}}(d) \leq c_2 + S_{\text{SplitAt}}(d-1),$$

so $S_{\text{SplitAt}}(d)$ is $O(d)$.

```
fun Merge (Empty, t2) = t2
  | Merge (Node (l1, x, r1), t2) =
      let
        val (l2, r2) = SplitAt (x, t2)
      in
        Node (Merge (l1, l2), x, Merge (r1, r2))
      end
```

```
fun Merge (Empty, t2) = t2
  | Merge (Node (l1, x, r1), t2) =
      let
        val (l2, r2) = SplitAt (x, t2)
      in
        Node (Merge (l1, l2), x, Merge (r1, r2))
      end
```

$$S_{\text{Merge}}(d_1, d_2)$$

$$\leq c_3 + S_{\text{SplitAt}}(d_2) + \max\left(S_{\text{Merge}}(d_1 - 1, ?),\ S_{\text{Merge}}(d_1', ?)\right)$$

(with $d_1' \leq d_1 - 1$)

```
fun Merge (Empty, t2) = t2
  | Merge (Node (l1, x, r1), t2) =
    let
        val (l2, r2) = SplitAt (x, t2)
    in
        Node (Merge (l1, l2), x, Merge (r1, r2))
    end
```

$$S_{\text{Merge}}(d_1, d_2)$$

$$\leq \; c_3 + S_{\text{SplitAt}}(d_2) + \max\big(S_{\text{Merge}}(d_1 - 1, ?),$$

$$S_{\text{Merge}}(d_1', ?)\big)$$

(with $d_1' \leq d_1 - 1$)

$$\leq \; c_3 + S_{\text{SplitAt}}(d_2) + S_{\text{Merge}}(d_1 - 1, d_2)$$

$$
\begin{aligned}
& S_{\mathrm{Merge}}(d_1, d_2) \\
\le\ & c_3 + S_{\mathrm{SplitAt}}(d_2) + \max\Big(S_{\mathrm{Merge}}(d_1 - 1, ?), \\
& \qquad S_{\mathrm{Merge}}(d_1', ?)\Big) \\
& \qquad (\text{with } d_1' \le d_1 - 1) \\
\le\ & c_3 + S_{\mathrm{SplitAt}}(d_2) + S_{\mathrm{Merge}}(d_1 - 1, d_2) \\
\le\ & c_3 + c_4 \cdot d_2 + S_{\mathrm{Merge}}(d_1 - 1, d_2).
\end{aligned}
$$

$$S_{\text{Merge}}(d_1, d_2)$$

$$\leq c_3 + S_{\text{SplitAt}}(d_2) + \max\Big(S_{\text{Merge}}(d_1 - 1, ?),$$

$$S_{\text{Merge}}(d_1', ?)\Big)$$

(with $d_1' \leq d_1 - 1$)

$$\leq c_3 + S_{\text{SplitAt}}(d_2) + S_{\text{Merge}}(d_1 - 1, d_2)$$

$$\leq c_3 + c_4 \cdot d_2 + S_{\text{Merge}}(d_1 - 1, d_2).$$

So $S_{\text{Merge}}(d_1, d_2)$ is $O(d_1 \cdot d_2)$.

```
fun Msort Empty = Empty
  | Msort (Node(l, x, r)) =
      Ins(x, Merge(Msort l, Msort r))
```

```
fun Msort Empty = Empty
  | Msort (Node(l, x, r)) =
      Ins(x, Merge(Msort l, Msort r))
```

$$S_{\mathbf{Msort}}(d)$$

$$\le c_5 + \max\left(S_{\mathbf{Msort}}(d-1),\, S_{\mathbf{Msort}}(d')\right) + S_{\mathbf{Merge}}(d_1, d_2) + S_{\mathbf{Ins}}(d_3).$$

```
fun Msort Empty = Empty
  | Msort (Node(l, x, r)) =
      Ins(x, Merge(Msort l, Msort r))
```

$$S_{\text{Msort}}(d)$$

$$\leq c_5 + \max\left(S_{\text{Msort}}(d-1), S_{\text{Msort}}(d')\right)$$

$$+ S_{\text{Merge}}(d_1, d_2) + S_{\text{Ins}}(d_3).$$

Here:
- $d' \leq d-1$.
- d_1 & d_2 are the depths of the trees returned by the recursive calls to Msort.
- d_3 is the depth of the tree returned by Merge.

$$S_{\mathrm{Msort}}(d)$$

$$\leq c_5 + \max\left(S_{\mathrm{Msort}}(d-1),\ S_{\mathrm{Msort}}(d')\right)$$

$$+ S_{\mathrm{Merge}}(d_1, d_2) + S_{\mathrm{Ins}}(d_3).$$

Here:

- $d' \leq d-1$.
- d_1 & d_2 are the depths of the trees returned by the recursive calls to Msort.
- d_3 is the depth of the tree returned by Merge.

If we rebalance as a final step in Msort, then $d_1 \leq d$, $d_2 \leq d$, & $d_3 \leq 2 \cdot d$.

Thus: $S_{\text{Msort}}(d)$

$$\leq c_5 + S_{\text{Msort}}(d-1) + S_{\text{Merge}}(d,d) + S_{\text{Ins}}(2d)$$

$$\leq c_5 + S_{\text{Msort}}(d-1) + c_6 \cdot d^2 + c_7 \cdot d$$

$$\leq c_8 \cdot d^2 + S_{\text{Msort}}(d-1).$$

Thus: $S_{Msort}(d)$

$$\le c_5 + S_{Msort}(d-1) + S_{Merge}(d,d) + S_{Ins}(2d)$$

$$\le c_5 + S_{Msort}(d-1) + c_6 \cdot d^2 + c_7 \cdot d$$

$$\le c_8 \cdot d^2 + S_{Msort}(d-1).$$

So $S_{Msort}(d)$ is $O(d^3)$.

Sorting

	list isort	list merge sort	tree merge sort
Work	$O(n^2)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$
Span	$O(n^2)$	$O(n)$	$O((\log n)^3)$ $O((\log n)^2)$
		(previous lecture)	(today) (in 15-210)

```
(* rebalance : tree -> tree
   REQUIRES: true
   ENSURES: rebalance(t) returns a tree t'
            containing exactly the elements of t,
            and in the same order, such that:
               depth(t') = ⌈log2(size(t'))⌉.
*)

fun rebalance (Empty) = Empty
  | rebalance (t) =
      let
         val (l, x, r) = halves(t)
      in
         Node (rebalance l, x, rebalance r)
      end
```

Comments

- **halves** is nontrivial.

- If the input tree t to **rebalance** is roughly balanced, then

$$W_{\text{rebalance}}(n) \text{ is } O(n)$$

&

$$S_{\text{rebalance}}(d) \text{ is } O(d^2).$$

Here $n = \text{size}(t)$ & $d = \text{depth}(t)$.

"Roughly balanced" means $d \leq c \cdot \log_2 n$, for some fixed constant c ($c = 2$, for instance).

(Analysis takes some effort.)

That is all.

Have a good weekend.

See you Tuesday, when we will talk about polymorphism.