Lecture 8

- Revisit Insert in Trees
- Revisit Merge Sort
  - Merge Sort on Trees
  - Analyzing Merge Sort on Trees
  - Summary of Merge Sort
  - A Peek into Rebalancing

Revisit Insert in 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)
```

The Work and Span are both $O(n)$. Or in fact, it is also $O(d)$. In the worst case, $d = n$.

For balanced trees, $d = O(log(n))$.

Revisit Merge Sort

Can we optimize the Span to be a polynomial of $log(n)$?

Merge Sort on Trees

```
(* Msort : tree -> tree
 * REQ: true
 * ENS: Msort T is a sorted tree containing the exact elements of T
 *)
(* See Lecture 7 for the concept of a sorted tree. *)
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(L, x, R)) = Ins(x, Merge (Msort L, Msort R))
```

How do we Merge then?

Consider merging trees `Node(L1, x, R1), Node(L2, y, R2)`.

Assuming `x <= y`, we want to figure out where `x` goes in the second tree, split the second tree into two parts, and recursively merge the first tree with the part where the root is less than or equal to `x`.

```
(* Merge : tree * tree -> tree
 * REQ: T1, T2 are sorted
 * ENS: Merge (T1, T2) is sorted and contains exactly the elements of T1 & T2
 *)
fun Merge (Empty : tree, T2 : tree) : tree = T2
  | Merge (T1, Empty) = T1
  | Merge (Node(L1, x, R1), T2) =
    let
      val (R_less, R_greater) = SplitAt(x, T2)
    in
      raise TODO
    end
```

What should go into `TODO`? `Node(Merge(L1, R_less), x, Merge(R1, R_greater))`

> Note: We have to inductively prove `Merge` is correct. The logic: assuming the spec, use the spec for the induction proof. Basic stuff: you have to guarantee spec holds.

> Side Note: When you have a spec that sometimes fails (we will see this in future), we have to be very careful about when it holds and when it does not.

```
(* SplitAt : int * tree -> tree * tree *)
fun SplitAt (_ : int, Empty : tree) : tree * tree = (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
```

> Micheal: Take 15 seconds to think about this. Make sure you understand this. I didn't say this is practice for the exam. :)

Analyzing Merge Sort on Trees

$$
\begin{aligned}
S_{Ins}(d) & = O(d) \\
S_{SplitAt}(d) & \leq c_1 + S_{SplitAt}(d-1) = O(d) \\
S_{Merge}(d_1, d_2) & \leq c_1 + S_{SplitAt}(d_2) + max(S_{Merge}(d_1 - 1, d_2), S_{Merge}(d_1 - 1, d_2)) \\
& \leq c_1 + c_2 d_2 + S_{Merge}(d_1 - 1, d_2) \\
& \leq c_3 d_1 d_2 \\
& = O(d_1 d_2) \\
S_{Msort}(d) & \leq c_1 + max(S_{Msort}(d-1), S_{Msort}(d-1)) + S_{Merge}(d_1, d_2) + S_{Ins}(d_1 + d_2)
\end{aligned}
$$

Now what?

Assuming trees are **balanced**, so $d = O(log(n))$, also assuming `Msort` returns balanced trees,

> Side Note: By the fact that we're stuck on the analysis now, it appears that we have not fully accomplished what we want with our current code.

$$
\begin{aligned}
S_{Msort} & \leq c_1 + S_{Msort}(d-1) + S_{Merge}(d, d) + S_{Ins}(2d) \\
& \leq c_1 c_2 d^2 + c_3 d + S_{Msort}(d-1) \\
& = O(d^3)
\end{aligned}
$$

Summary of Merge Sort

	Work	Span
Insertion Sort (list)	$O(n^2)$	$O(n^2)$
Merge Sort (list)	$O(nlogn)$	$O(n)$
Merge Sort (tree)	$O(nlogn)$	$O((logn)^3)$

A Peek into Rebalancing

```
(* To make this fast, we re-define tree datatype. *)
datatype tree = Empty
              | Node of tree * int * int * tree
fun rebalance (Empty : tree) : tree = Empty
  | rebalance T =
    let
      val(L, x, R) = halves T
    in
      Node(rebalance L, x, rebalance R)
    end
```

The Work of rebalancing is $O(n)$, and the Span is $O((logn)^2)$.