Yesterday we looked at the sequential running time of mergesort, called the work, and showed that it was $O(n \log n)$, which is better than the work of insertion sort which we showed on Tuesday to be $O(n^2)$. We also saw that the parallel running time, or the span, was $O(n)$, which was disappointing. Today we’ll do better by switching to the tree data structure.

1 Tree Datatype

Recall that integer trees are defined as follows:

```
datatype inttree =
    Empty
  | Node of (inttree * int * inttree)
```

In lab yesterday, you defined functions like `size` and `depth` on trees. These will continue to be important.

2 Sorted Trees

Well, we’ve got trees now. But we wanted to use them to write mergesort. And if we’re going to sort trees, we should figure out what it means for a tree to be sorted. You defined one function in lab yesterday that determined if a tree was sorted. Here’s a more mathematical definition:

- **Empty** is sorted.
- **Node(l,x,r)** is sorted iff
  - $x$ is valuable, and
  - $l$ is sorted, and
  - $r$ is sorted, and
  - everything in $l$ is $\leq x$, and
  - $x \leq$ everything in $r$.
- **e** is sorted if $e \cong e'$ and $e'$ is sorted.

*Based on notes by Brandon Bohrer, Carlo Angiuli and others.*
The first two clauses say when values are sorted; the third says that sortedness respects equivalence. For example, 

\[ \text{Node(Node(Node(Empty,1,Empty),} \]
\[ \quad 2, \]
\[ \quad \text{Node(Empty,3,Empty)),} \]
\[ \quad 4, \]
\[ \quad \text{Node(Empty,5,Empty))} \]

which we can represent graphically as

```
    4
   / \
  2   5
 / \
1  3
```

is sorted using the first two clauses. We’ll use the third to state a theorem like “for all trees \( t \), \( \text{sort } t \) is sorted” and then prove it by calculating.

We can also define sortedness in terms of an in-order traversal of the tree and the \texttt{sorted} function on lists we used yesterday:

\[
\text{(* trav : inttree -> int list \*) REQUIRES: true ENSURES: trav t ==> l, a list representing an in-order traversal of t *)}
\]

\[
\text{fun trav (t : inttree) : int list = case t of Empty => [] \mid Node (l,x,r) => (trav l) @ (x :: (trav r))}
\]

\[
\text{(* tsorted : inttree -> bool \*) REQUIRES: true ENSURES: tsorted t ==> true if t is sorted, false otherwise. *)}
\]

\[
\text{fun tsorted (t : inttree) : bool = sorted (trav t)}
\]

3 Mergesort on Trees

3.1 Code

Let’s be bold and follow the template for structural recursion:

\[
\text{fun mergesort (t : inttree) : inttree = case t of Empty => Empty \mid Node (l, x, r) => ... mergesort l ... mergesort r ...}
\]
Assuming we have sorted \( l \) and \( r \), what do we need to finish off the case? We need to \textit{merge} together the two sorted results and the tree containing just \( x \). So we push one helper function on the to-do list:

\[
(* \text{merge: inttree} \ast \text{inttree} \to \text{inttree} \\
* \text{REQUIRES: t1 and t2 are sorted} \\
* \text{ENSURES: merge (t1,t2) ==> t, a sorted tree whose elements are} \\
* \quad \text{exactly those of t1 and t2 combined} \\
*)
\]

\[
\text{fun merge (t1 : inttree , t2 : inttree) : inttree = ...}
\]

and finish off \textit{mergesort} by calling it:

\[
(* \text{mergesort : inttree} \to \text{inttree} \\
* \text{REQUIRES: true} \\
* \text{ENSURES: mergesort t ==> t', a sorted tree whose elements are exactly those} \\
* \quad \text{of t} \\
*)
\]

\[
\text{fun mergesort (t : inttree) : inttree =} \\
\quad \text{case t of} \\
\quad \quad \text{Empty } \Rightarrow \text{ Empty} \\
\quad \mid \text{Node (l , x , r) } \Rightarrow \\
\quad \quad \text{merge(Node(Empty,x,Empty),} \\
\quad \quad \quad \text{merge (mergesort l , mergesort r))}
\]

We use \texttt{Node(Empty,x,Empty)} to make a one-element tree, and use \texttt{merge} twice to put these three trees together.

### 3.1.1 Splitting

Notice that the split into subproblems is \textit{constant time}—the splitting is given by the data structure itself! Second, we can merge two trees in sublinear span, which gets a sublinear span overall. This is tricky, but doable; we'll show how today.

### 3.1.2 Merging

Here's the idea with merging: say we need to merge two sorted trees,

\[
\text{Node (Node(Empty,1,Empty), 3 , Node(Empty,5,Empty))}
\]

\[
\quad 3 \\
\quad / \quad \backslash \\
\quad 1 \quad 5
\]

and

\[
\text{Node (Node(Empty,2,Empty), 4 , Node(Empty,6,Empty))}
\]

\[
\quad 4 \\
\quad / \quad \backslash \\
\quad 2 \quad 6
\]
We will (somewhat arbitrarily) choose to be guided by the first tree, and will stipulate that the root of the first tree will be the overall root. So the question is, how do we need to fill in this\(^1\):

\[
\text{Node (merge(?,?), 3 , merge(?,?))}
\]

\[
\begin{array}{c}
/ \\
/ \\
/ \\
/ \\
/ \\
/ \\
merge (?, ?) \quad merge (?, ?)
\end{array}
\]

Clearly, the left subtree of 3 needs to go to the left, and the right to the right, for the result to be sorted and contain all the appropriate elements.

\[
\text{Node (merge(Node(Empty,1,Empty),?), 3 , merge(Node(Empty,5,Empty),?))}
\]

For similar reasons, we need to put everything in the second tree that is less than 3 to the left, and everything greater to the right.

\[
\text{Node (merge(Node(Empty,1,Empty),Node(Empty,2,Empty)), 3, merge(Node(Empty,5,Empty),Node (Empty, 4 , Node(Empty,6,Empty))))}
\]

So let’s make up another helper function:

\[
(* \text{REQUIRES: t is sorted} \quad * \text{ENSURES: splitAt (t,bound) }\rightarrow (l,r) \text{ where l and r are sorted,} \\
* \quad \quad l \text{ contains the elements of t that are } \leq \text{ bound, and} \\
* \quad \quad r \text{ contains the elements of t that are } > \text{ bound} \\
*)
\]

\[
\text{fun splitAt (t : inttree , bound : int) : inttree * inttree =}
\]

\[
\text{case t of}
\]

\[
\text{Empty } \rightarrow (\text{Empty} , \text{Empty})
\]

\[
| \quad (\text{case compare (bound, x) of} \\
| \quad \quad \text{LESS } \rightarrow \text{let } \text{val (ll , lr) } = \text{splitAt (l , bound) in (ll , Node (lr , x , r)) end} \\
| \quad \quad \_ \quad \rightarrow \text{let } \text{val (rl , rr) } = \text{splitAt (r , bound) in (Node (l , x , rl) , rr) end)
\]

Using this, it’s simple to write merge:

---

\(^1\)look at my great ASCII art skills
(* REQUIRES: t1, t2 sorted
* ENSURES: merge (t1,t2) ==> t, a sorted tree whose elements are exactly those
* of t1 and t2 combined
*)

fun merge (t1 : inttree , t2 : inttree) : inttree =
  case t1 of
    Empty => t2
  | Node (l1 , x , r1) =>
    let
      val (l2 , r2) = splitAt (t2 , x)
    in
      Node (merge (l1 , l2) ,
            x ,
            merge (r1 , r2))
    end

Let’s look at the case where $bound < x$. Here, we know that everything in the right sub-tree
is greater than the bound because the tree is sorted; we’re not sure, though, where the bound falls
inside the left sub-tree. By induction $\text{splitAt}(l,bound)$ divides up $l$ so that everything less than $bound$
is in $l1$ and everything greater is in $lr$. We know that $x$ and $r$ have to go on the right side,
because $x > bound$, everything in $r$ is greater than $x$, and therefore also greater than the bound.
The other case is exactly symmetric.

3.2 Analysis

The overall work of mergesort is $O(n \log n)$, and the span is $O((\log n)^3)$. Let’s look at the work
analysis first. Note that this is actually more complicated than the span analysis, and you don’t
need to know it in detail (though you should remember that the work is $O(n \log n)$).

First, let $d$ be the depth of the tree. Then

$$W_{\text{splitAt}}(d) = k + W_{\text{splitAt}}(d - 1)$$

because peeling off the $\text{Node}$ constructor decreases the depth by 1. Thus, $W_{\text{splitAt}}(d)$ is $O(d)$.

Let $n_1$ and $n_2$ be the sizes of trees $t1$ and $t2$ (so $\log n_2$ is the depth of $t2$, if the trees are balanced). Then

$$W_{\text{merge}}(n_1 , n_2) = k + W_{\text{splitAt}}(\log n_2) + W_{\text{merge}}(n_1/2 , n_{21}) + W_{\text{merge}}(n_1/2 , n_{22})$$

where $n_{21}$ and $n_{22}$ are the depths of the result of the split, so $n_{21} + n_{22} = n_2$.

Expanding this out, we see that the work at level $i$ is

$$\log n_{21} + \cdots + \log n_{2i}$$

where $n_{21} + \cdots + n_{2i} = n_2$. Let’s assume that $n_{21} = \cdots = n_{2i}$, as this turns out to be the worst
case. So the work at level $i$ is

$$2^i \log \frac{n_2}{2^i} = 2^i (\log n_2 - i)$$
\[ \sum_{i=0}^{\log n_1} (2^i (\log n_2 - i)) \]
\[ = \log n_2 \sum_{i=0}^{\log n_1} 2^i - \sum_{i=0}^{\log n_1} i2^i \]
\[ = (2^{\log n_1+1} - 1) \log n_2 - (2^{\log n_1+1} (\log n_1 - 1) + 2) \]
\[ = (2n_1 - 1) \log n_2 - (2n_1 (\log n_1 - 1) + 2) \]
\[ = 2n_1 (\log n_2 - \log n_1 + 1) - \log n_2 - 2 \]
\[ \in O(n_1 (\log(1 + \frac{n_2}{n_1}))) \]

(The closed form for \( \sum_{i=0}^{\log n_1} i2^i \) uses a couple tricks that are outside the scope of this class. See, e.g. Chapter 12 of http://www.parallel-algorithms-book.com/).

Finally, we analyze the work for mergesort.

\[ W_{\text{mergesort}}(n) = k + W_{\text{mergesort}} \left( \frac{n}{2} \right) + W_{\text{mergesort}} \left( \frac{n}{2} \right) + W_{\text{merge}}(n, n) + W_{\text{merge}}(1, 2n) \]
\[ \leq k + 2W_{\text{mergesort}} \left( \frac{n}{2} \right) + k_1 n + k_2 \log n \]

Note that, modulo some constants and the \( \log n \) term (which is dominated by \( k_1 n \), this is the same recurrence we had for mergesort on lists, so this is \( O(n \log n) \).

Now for the span analysis.

Again, let \( d \) be the depth of the tree. Then

\[ S_{\text{splitAt}}(d) = k + S_{\text{splitAt}}(d - 1) \]

because peeling off the Node constructor decreases the depth by 1. Thus, \( S_{\text{splitAt}}(d) \) is \( O(d) \).

Let \( d_1 \) and \( d_2 \) be the depths of trees \( t_1 \) and \( t_2 \). Then

\[ S_{\text{merge}}(d_1, d_2) = k + S_{\text{splitAt}}(d_2) + \max(S_{\text{merge}}(d_1 - 1, d_2_1), S_{\text{merge}}(d_1 - 1, d_2_2)) \]

where \( d_2_1 \) and \( d_2_2 \) are the depths of the result of the split. These are no deeper than the original tree, so we can overapproximate as

\[ S_{\text{merge}}(d_1, d_2) \leq k + S_{\text{splitAt}}(d_2) + \max(S_{\text{merge}}(d_1 - 1, d_2), S_{\text{merge}}(d_1 - 1, d_2)) \]
\[ \leq k'd_2 + S_{\text{merge}}(d_1 - 1, d_2) \]

Expanding this out, you can see that we do \( k'd_2 \) work \( d_1 \) times, so \( S_{\text{merge}}(d_1, d_2) \) is \( O(d_1 \cdot d_2) \).

Now for mergesort. Let \( n \) be the size of the tree, and assume it is balanced, so the depth is about \( \log n \).

\[ S_{\text{mergesort}}(n) = k + \max \left( S_{\text{mergesort}} \left( \frac{n}{2} \right), S_{\text{mergesort}} \left( \frac{n}{2} \right) \right) + S_{\text{merge}}(\log n, \log n) + S_{\text{merge}}(1, 2 \log n) \]
\[ \leq k + S_{\text{mergesort}} \left( \frac{n}{2} \right) + k_1 (\log n)^2 + k_2 \log n \]
\[ \leq S_{\text{mergesort}} \left( \frac{n}{2} \right) + k_3 (\log n)^2 \]
The second call to \texttt{merge} is on the output of the first, so we need to know how \texttt{merge} changes the depth of the tree. Fortunately, we can prove that

\[
\text{depth } (\text{merge}(l,r)) \leq \text{depth } l + \text{depth } r
\]

Overall the recurrence says that we do \(O((\log n)^2)\) steps (the divisions inside the log don’t help you, because it’s just subtracting off a constant \(\log n\) times, so the overall span is \(O((\log n)^3)\). Thus (ignoring constants), when we try to sort a billion elements, the length of the longest critical path is about 27000 operations, so we can exploit over a million processors!

Or it would be, if there wasn’t a bug in this analysis! When we instantiated \(S_{merge}(\log n, \log n)\) and \(S_{merge}(\log n, 1)\), we were relying on the fact that the trees we passed to \texttt{merge} were balanced. However, this is not necessarily the case, because we call these functions on the output of mergesort. Moreover, mergesort doesn’t necessarily produce a balanced tree. We can fix this by rebalancing each time through:

\[
\text{fun mergesort } (t : \texttt{inttree}) : \texttt{inttree} =
\begin{align*}
\text{case } t \text{ of} \\
\quad \text{Empty } => \text{Empty} \\
\mid \text{Node } (l , x , r) => \\
\quad \text{rebalance}(\text{merge}(\text{Node}(\text{Empty},x,\text{Empty}), \\
\quad \quad \text{merge } (\text{mergesort } l , \text{mergesort } r)))
\end{align*}
\]

We might take a look at \texttt{rebalance} later; it doesn’t change the overall work or span.

### 3.3 Correctness

How would you prove that \texttt{mergesort} returns a sorted tree? \textit{Structural induction on trees!} Because there are two cases in the definition of trees, our proof will have two cases. And because a \texttt{Node} has two subtrees, we will get two inductive hypotheses, one for each subtree.

\textbf{Theorem 1.} \textit{For all values } \texttt{t:tree}, \texttt{mergesort t sorted}.

Here’s the template:

\textit{Proof. Case for } \texttt{Empty}:

\textbf{To show: } \texttt{mergesort Empty sorted}

\textbf{Proof: } \texttt{mergesort Empty == Empty} in 2 steps, and \texttt{Empty} is sorted by definition.

\textbf{Case for } \texttt{Node(l,x,r)} \textit{Let } \texttt{l} \text{ and } \texttt{r} \textit{ be any trees, and } \texttt{x} \textit{ be any int.}

\textbf{IH1: } \texttt{mergesort l sorted}

\textbf{IH2: } \texttt{mergesort r sorted}

\textbf{To show: } \texttt{mergesort Node(l,x,r) sorted}

\textbf{Proof: } By stepping the code twice and appealing to transitivity of equivalence, we get that \texttt{mergesort Node(l,x,r)} is equivalent to

\[
\text{merge } (\text{merge } (\text{mergesort l, mergesort r}), \text{Node}(\text{Empty},x,\text{Empty}))
\]
By the two inductive hypotheses, \texttt{mergesort l} and \texttt{mergesort r} are sorted. By the spec for \texttt{merge}, which says that \texttt{merge} takes two sorted trees to a third, \texttt{merge(mergesort l, mergesort r)} is sorted. Applying this spec again, the whole expression is sorted. So because sortedness respects equivalence, \texttt{mergesort (Node (l,x,r))} is sorted as well.

It’s worth pointing out a subtlety in the proof: The easiest thing to prove about \texttt{merge} is

\textbf{Lemma 1.} \textit{For all values l:tree and r:tree, if l is sorted and r is sorted then merge(l,r) is sorted.}

When we state the theorem for \textit{values}, then we can do a proof by induction, because the values of type \texttt{tree} are inductively defined.

However, in the proof, we need to appeal to the lemma on non-values, like \texttt{mergesort l} and \texttt{mergesort r}. To do this, we can lift the lemma to \textit{valuables}:

\textbf{Corollary 1.} \textit{For all valuable expressions e1:tree and e2:tree, if e1 is sorted and e2 is sorted then merge(e1,e2) is sorted.}

\textit{Proof.} By definition of valuability \(e1 \cong v1\) and \(e2 \cong v2\) for some values \(v1\) and \(v2\). Because sorted respects equivalence, \(v1\) and \(v2\) are sorted. By the lemma above, this means that \(merge(v1,v2)\) is sorted. Again because sorted respects equivalence, \(merge(e1,e2)\) is sorted.

This is the same lifting we’ve done for lists, just now for a theorem about trees.

In the proof, we then need to know that \texttt{mergesort e1} and \texttt{mergesort e2} and \texttt{merge(mergesort e1, mergesort e2)} are valuable. But it is simple to prove that all of \texttt{splitAt}, \texttt{merge}, and \texttt{mergesort} are total, because they are structurally recursive.