1 Background

As in the previous lecture, we refer to:

datatype order = LESS | EQUAL | GREATER

(* Comparison for integers *)

(* compare : int * int -> order *)
REQUIRES: true
ENSURES:
    compare(x,y) ==> LESS if x<y
    compare(x,y) ==> EQUAL if x=y
    compare(x,y) ==> GREATER if x>y
*)
fun compare(x:int, y:int):order = 
    if x<y then LESS else
    if y<x then GREATER else EQUAL

As previously, a list of integers is *sorted* if each item in the list is no greater than all items that occur later in the list. Here is an SML function that checks for this property:

(* sorted : int list -> bool
REQUIRES: true
ENSURES: sorted(L) evaluates to true if L is sorted and to false otherwise.
*)
fun sorted [ ] = true
| sorted [x] = true
| sorted (x::y::L) = (compare(x,y) <> GREATER) andalso sorted(y::L)

* Adapted from documents by Stephen Brookes and Dan Licata.
We will also refer to the ins function, which we used as a helper function for sorting integer lists.

\[(\ast\text{ ins : int} \times\text{ int list }\rightarrow\text{ int list}\]
\[\text{REQUIRES: } L \text{ is sorted}\]
\[\text{ENSURES: } \text{ins}(x, L) \text{ evaluates to a sorted permutation of } x::L\]
\[(\ast)\]
\[\text{fun ins } (x, [\text{ ]}) = [x]\]
\[\mid \text{ ins } (x, y::L) = \text{case compare}(x, y) \text{ of}\]
\[\mid \quad \text{ GREATER } \Rightarrow y::\text{ins}(x, L)\]
\[\mid \quad _{\_} \Rightarrow x::y::L\]

2 Integer Trees in SML

We will use the following integer tree type:

\[\text{datatype tree } = \text{Empty } | \text{Node of tree} \times\text{ int} \times\text{ tree}\]

We can draw pictures of trees by putting the root integer at the top, as usual, and we may omit drawing leaf nodes. For example, let \( t \) be the tree \( \text{Node}(\text{Empty}, 42, \text{Node}(\text{Empty}, 9, \text{Empty})) \). This can be drawn as:

\[
t = \begin{array}{c}
42 \\
\downarrow \\
9
\end{array}
\]

And the tree \( \text{Node}(t, 0, t) \) looks like:

\[
\begin{array}{c}
0 \\
\downarrow \\
42 \\
\downarrow \\
9 \\
\downarrow \\
9
\end{array}
\]

depth and size

Let \( \text{max : int} \times\text{int }\rightarrow\text{ int} \) be the usual integer maximum function (predefined in SML as \( \text{Int.max} \)):

\[\text{fun max}(x:\text{int}, y:\text{int}):\text{int} = \text{if } x>y \text{ then } x \text{ else } y\]

We define the functions \text{depth} and \text{size} of type \( \text{tree }\rightarrow\text{ int} \) by:

\[\text{fun depth Empty} = 0\]
\[\mid \text{ depth (Node(t1, _, t2))} = \text{max(depth t1, depth t2)} + 1\]

\[\text{fun size Empty} = 0\]
\[\mid \text{size (Node(t1, _, t2))} = \text{size t1} + \text{size t2} + 1\]
Intuitively, size(t) computes the number of non-leaf nodes in t, and depth(t) computes the length of the longest path from the “root” of t to a leaf node. (We have also referred to depth as height in other settings.)

We refer to size(t) as “the size of t” and to depth(t) as the “depth of t”.

For all trees t, size(t) ≥ 0 and depth(t) ≥ 0; and if t’ is a child of t, then depth t’ < depth t and size t’ < size t. So we can also use induction on tree depth, or induction on tree size, as techniques for proving properties of trees, or of functions operating on trees.

Aside: structural induction on trees, induction on tree size, and induction on tree depth, as well as simple and complete induction on non-negative integers, are all special cases of a general technique known as well-founded induction.

**In-order Traversal**

Here is a function that builds a list of integers from a tree, by making an in-order traversal of the tree, collecting data into a list. In-order traversal of a non-empty tree involves traversing the left-child, then the root, and then traversing the right-child; we also use in-order traversal on the subtrees. This description suggests that we define a recursive function!

This function is used mainly in specifications, but serves as an example of how to define a function that operates on trees: use clauses, one for the empty tree and one for non-empty trees, using pattern-matching to give names to the components of a tree.

(You have probably seen or will see this function in other settings with other names, but the idea is the same throughout: in-order traversal of a tree.)

```plaintext
(* trav : tree -> int list
    REQUIRES: true
    ENSURES: trav(t) evaluates to a list consisting of the integers in t,
             in the same order as seen during an in-order traversal of t. *)

fun trav Empty = []
| trav (Node(t1, x, t2)) = trav t1 @ (x :: trav t2)
```

For example, for the tree t on page 2, trav(t) ==> [42,9].

And

```
trav(Node(t, 0, t)) ==> trav t @ (0 :: trav t)
===> [42, 9] @ (0 :: [42, 9])
===> [42, 9, 0, 42, 9].
```

We say “x is in t” if x is a member of the list trav(t).
We prove the following theorem to build familiarity with the terms:

**Theorem 1** For all values \( T : \text{tree} \), \( \text{trav}(T) \) evaluates to a list of length \( \text{size}(T) \).

**Proof:** By structural induction on \( T \).

**BASE CASE:** \( T = \text{Empty} \).
We must show that \( \text{trav}(\text{Empty}) \) is a list of length \( \text{size}(\text{Empty}) \).
Observe that \( \text{trav}(\text{Empty}) \Rightarrow [\ ] \) and that \( \text{size}(\text{Empty}) \Rightarrow 0 \), so the base case follows.

**INDUCTIVE STEP:** \( T = \text{Node}(t_1, x, t_2) \).

**Inductive Hypotheses:** \( \text{trav}(t_1) \) is a list of length \( \text{size}(t_1) \) and \( \text{trav}(t_2) \) is a list of length \( \text{size}(t_2) \).

**Need to show:** \( \text{trav}(T) \) is a list of length \( \text{size}(T) \).

**Showing:**
Let \( L_1 \) be the list \( \text{trav}(t_1) \) and \( L_2 \) the list \( \text{trav}(t_2) \).
Let \( n_1 \) be the integer \( \text{size}(t_1) \) and \( n_2 \) the integer \( \text{size}(t_2) \).

By definition of \( \text{size} \) we have

\[
\text{size}(T) = \text{size}(\text{Node}(t_1, x, t_2)) \Rightarrow (\text{size } t_1) + 1 + \text{size}(t_2) \Rightarrow n_1 + 1 + n_2
\]

By definition of \( \text{trav} \) we have

\[
\text{trav}(T) = \text{trav}(\text{Node}(t_1, x, t_2)) \Rightarrow (\text{trav } t_1) @ (x :: \text{trav } t_2) \Rightarrow L_1 @ (x :: L_2)
\]

which, by the inductive hypotheses, reduces to a list of length \( n_1 + 1 + n_2 \), as desired. \( \square \)

**Sorted Trees**

Informally, we say that a value of type \( \text{tree} \) is sorted if the integers in the tree occur in sorted order. More precisely we intend this to mean that the in-order traversal list of the tree is sorted.

Equivalently: (i) an empty tree is sorted and (ii) a non-empty tree is sorted if and only if its two subtrees are sorted, every integer in the left subtree is less-than-or-equal to the integer at the root, and every integer in the right subtree is greater-than-or-equal to the integer at the root.

We can implement an SML function for testing tree sortedness:

(* Sorted : tree -> bool
  REQUIRES: true
  ENSURES: Sorted(T) evaluates to true if T is sorted and false otherwise.
*)
fun Sorted T = sorted(trav T)
Using `trav` like this is an easy way to make a slightly vague assertion about the contents of a tree into a rigorous one. So, a tree is called “sorted” if and only if its traversal list is sorted in a sense with which we are already familiar. Similarly, we will say that one tree `t1` is a “permutation” of another tree `t2` if and only if the list `trav(t1)` is a permutation of the list `trav(t2)`.

**Insertion for Trees**

Insertion sort is not well suited to parallel implementation, even if we are inserting into a tree. Nevertheless the tree-based analogue of the insertion function on lists is still of interest. We use capitalization to distinguish this function from the `ins` function on lists used in the previous lecture.

\[
\begin{align*}
(* & \text{ Ins : int * tree -> tree} \\
& \text{ REQUIRES: t is a sorted tree} \\
& \text{ ENSURES: Ins}(x, t) \text{ evaluates to a sorted tree such that} \\
& \quad \text{trav} (\text{Ins}(x, t)) \text{ is a sorted permutation of } x::\text{trav}(t). \\
*)
\end{align*}
\]

```haskell
fun Ins (x, Empty) = Node(Empty, x, Empty)
| Ins (x, Node(t1, y, t2)) = 
  case compare(x,y) of 
    GREATER => Node(t1, y, Ins(x, t2))
  | _ => Node(Ins(x, t1), y, t2)
```

Compare this code with the code for the list function `ins` given earlier.

In what follows we take as a fact that `Ins` is total. That is easy to prove, using structural induction.
Theorem 2 For all \( x : \text{int} \) and all sorted \( T : \text{tree} \), \( \text{trav}(\text{Ins}(x,t)) \cong \text{ins}(x, \text{trav } t) \).

Proof: By structural induction on tree \( T \).

 BASE CASE: \( T = \text{Empty} \).

Need to show that for every integer \( x \),
\( \text{trav}(\text{Ins}(x,\text{Empty})) \cong \text{ins}(x, \text{trav } \text{Empty}) \).

By the definitions of \( \text{trav}, \text{Ins} \) and \( \text{ins} \) we see

\[
\begin{align*}
\text{trav}(\text{Ins}(x,\text{Empty})) & \\
\implies & \text{trav}(\text{Node}(\text{Empty},x,\text{Empty})) & [1\text{st Ins clause}] \\
\implies & \text{trav}(\text{Empty}) \circ (x :: \text{trav}(\text{Empty})) & [2\text{nd trav clause}] \\
\implies & [ ] \circ [x] & [1\text{st trav clause}] \\
\implies & [x] & [\text{see append def}]
\end{align*}
\]

\( \text{ins}(x, \text{trav } \text{Empty}) \)
\[
\begin{align*}
\implies & \text{ins}(x, [ ]) & [1\text{st trav clause}] \\
\implies & [x] & [1\text{st ins clause}]
\end{align*}
\]

That establishes the base case.

 INDUCTIVE STEP: \( T = \text{Node}(t_1, y, t_2) \).

(Observe that \( t_1 \) and \( t_2 \) are sorted since \( T \) is.)

Inductive Hypotheses: For every integer \( x \),
\( \text{trav}(\text{Ins}(x,t_1)) \cong \text{ins}(x, \text{trav } t_1) \) and
\( \text{trav}(\text{Ins}(x,t_2)) \cong \text{ins}(x, \text{trav } t_2) \).

Need to show: For every integer \( x \),
\( \text{trav}(\text{Ins}(x,T)) \cong \text{ins}(x, \text{trav } T) \).

Showing:
Suppose \( x > y \) (the other cases are similar, as usual). Then:

\[
\begin{align*}
\text{trav}(\text{Ins}(x, \text{Node}(t_1, y, t_2))) & \\
\cong & \text{trav}(\text{Node}(t_1, y, \text{Ins}(x, t_2))) & [\text{step, 2nd Ins clause, } x>y] \\
\cong & (\text{trav } t_1) \circ (y :: \text{trav}(\text{Ins}(x, t_2))) & [\text{step, 2nd trav clause, Ins totality}] \\
\cong & (\text{trav } t_1) \circ (y :: \text{ins}(x, \text{trav } t_2)) & [\text{IH, RefTrans}] \\
\cong & (\text{trav } t_1) \circ (\text{ins}(x, y :: \text{trav } t_2)) & [\text{step, 2nd ins clause, } x>y, \text{RefTrans}] \\
\cong & \text{ins}(x, (\text{trav } t_1) \circ (y :: \text{trav } t_2)) & [T \text{ is sorted, } x>y]^1 \\
\cong & \text{ins}(x, \text{trav } (\text{Node}(t_1, y, t_2))) & [\text{step, 2nd trav clause}]
\end{align*}
\]

Reminder: The proof methodology we employed to establish the base case was of the form: If expression \( e_1 \) reduces to some value \( v \) and if expression \( e_2 \) also reduces to that same value \( v \), then \( e_1 \cong e_2 \). Alternatively, we could have shown that \( e_1 \cong e_2 \) by establishing a string of equivalences, as we did in the Inductive Step. Often a proof can be done either way, though one must be careful to employ the correct reasoning for the particular methodology one uses. In particular, it is false to say that \( e_1 \Rightarrow e_1 \) if in fact no such reduction occurs, if merely \( e_1 \cong e_2 \). And establishing

\[^1\]Technically, this step requires its own proof. In an exam, we would probably supply it as a separate fact or lemma.
that $e_1 \cong e_2$ must itself be done with care, since one must ensure that the expressions of interest really do both have equivalent values or both loop forever or both raise the same exception. See the guide to extensional equivalence on the course’s tools webpage for further advice.

3 Splitting a Tree

In adapting the mergesort algorithm to operate on trees we need a suitable analog to the `split` function. It isn’t easy to figure out a good way to hew a tree into two roughly equal sized pieces, based solely on the structure of the tree. Instead, we will start from a tree and an integer, and break the tree into two trees that consist of items in the tree less-than-or-equal to the integer and items greater-than-or-equal to the integer, respectively. We will only ever need to use this method on a sorted tree, as you will observe when we develop the code. Indeed, the design of this function takes advantage of the assumption that the tree is already sorted, a fact that we echo in the way we write the function’s specification.

\begin{verbatim}
(* SplitAt : int * tree -> tree * tree
   REQUIRES: t is sorted
   ENSURES: SplitAt(x,t) evaluates to a pair (t1,t2) of sorted trees such that:
   (a) for every Node(_,y,_) in t1, y <= x,
   and (b) for every Node(_,y,_) in t2, y >= x,
   and (c) trav(t1)@trav(t2) is a sorted permutation of trav(t).
   *)
   fun SplitAt(x, Empty) = (Empty, Empty)
   | SplitAt(x, Node(left, y, right)) =
     case compare(x, y) of
     LESS => let
       val (t1, t2) = SplitAt(x, left)
     in (t1, Node(t2, y, right))
     end
     | _ => let
       val (t1, t2) = SplitAt(x, right)
     in (Node(left, y, t1), t2)
     end
\end{verbatim}
4 Merging Two Trees

The tree-based analog of \texttt{merge} is a function that takes a pair of sorted trees and combines their data into a single (also sorted) tree.

\[
(* \text{ Merge : tree * tree -> tree} \\
\text{REQUIRES: t1 and t2 are sorted} \\
\text{ENSURES: Merge(t1,t2) evaluates to a sorted tree t such that} \\
\text{trav(t) is a sorted permutation of trav(t1)@trav(t2).} \\
*)
\]

\[
\text{fun Merge (Empty, t2) = t2} \\
| \text{Merge (Node(l1,x,r1), t2) = let} \\
| \quad \text{val (l2, r2) = SplitAt(x, t2)} \\
| \quad \text{in} \\
| \quad \text{Node(Merge(l1, l2), x, Merge(r1, r2))} \\
\end{align}
\]

5 Mergesort for Trees

Using \texttt{Ins} and \texttt{Merge}, and guided by their specs, we may now define a mergesorting function for integer trees.

\[
(* \text{ Msort : tree -> tree} \\
\text{REQUIRES: true} \\
\text{ENSURES: Msort(t) evaluates to a sorted tree such that} \\
\text{trav(Msort(t)) is a sorted permutation of trav(t).} \\
*)
\]

\[
\text{fun Msort Empty = Empty} \\
| \text{Msort (Node(t1, x, t2)) = Ins (x, Merge(Msort t1, Msort t2))}
\]

Compare and contrast this code with the mergesorting function \texttt{msort} for integer lists that we examined during the last lecture. This code no longer needs to do any significant work to split the input data before sorting recursively, since the tree structure already encapsulates a natural split (into left and right parts). However, it may be necessary now to perform some splitting when merging the results of the recursive calls.

6 Depth Analysis

There can be many different trees containing the same integers. Indeed, there can be many different \textit{sorted} trees containing the same integers. So the specifications and proofs so far don’t really tell us much about the shapes of the trees produced by sorting.

We state some intuitive results about depth, that will be helpful when we analyze the runtime behavior of our code. One may prove these results by induction.

1. For all trees \( t \) and integers \( x \),

\[
\text{depth(Ins(x, t)) \leq depth(t) + 1.}
\]
2. For all trees \( t \) and integers \( x \), if \( \text{SplitAt}(x, t) \Rightarrow (t_1, t_2) \) then
\[
\text{depth}(t_1) \leq \text{depth}(t) \quad \text{depth}(t_2) \leq \text{depth}(t).
\]
3. For all trees \( t_1 \) and \( t_2 \),
\[
\text{depth}(\text{Merge}(t_1, t_2)) \leq \text{depth}(t_1) + \text{depth}(t_2).
\]
4. For all trees \( t \),
\[
\text{depth}(\text{Msort } t) \leq 2*\text{depth}(t).
\]

7 Size Analysis

We further state some intuitive results about size. Again, one may use a suitable inductive method to prove these facts.

1. For all trees \( t \) and integers \( x \),
\[
\text{size}(\text{Ins}(x, t)) = \text{size}(t) + 1.
\]
2. For all trees \( t \) and integers \( x \), if \( \text{SplitAt}(x, t) \Rightarrow (t_1, t_2) \) then
\[
\text{size}(t_1) + \text{size}(t_2) = \text{size}(t).
\]
3. For all trees \( t_1 \) and \( t_2 \),
\[
\text{size}(\text{Merge}(t_1, t_2)) = \text{size}(t_1) + \text{size}(t_2).
\]
4. For all trees \( t \),
\[
\text{size}(\text{Msort } t) = \text{size}(t).
\]

8 Span Analysis

The overall work\(^2\) of our \texttt{Msort} implementation is \( O(n \log n) \), and the span is \( O((\log n)^3) \). We now analyze the span in detail. We focus on span, since the purpose of using trees was to enable parallel-friendly sorting.

- The span for \( \text{Ins}(x, t) \) is \( O(d) \), where \( d \) is the depth of \( t \). Reason: \( \text{Ins}(x, t) \) makes a single recursive call, on a subtree whose depth is \( d - 1 \) or less. This produces the recurrence
\[
\begin{align*}
S_{\text{Ins}}(0) &= c_0 \\
S_{\text{Ins}}(d) &\leq c_1 + S_{\text{Ins}}(d - 1) \quad \text{for } d > 0
\end{align*}
\]
for some constants \( c_0, c_1 \). This recurrence has a solution that is \( O(d) \).

- \( \text{SplitAt}(x, t) \) has span \( O(d) \), where \( d \) is the depth of \( t \). Again the reason is that \( \text{SplitAt} \) makes a single recursive call, on a subtree whose depth is \( d - 1 \) or less.

\(^2\)Using the techniques from this course, you should be able to show that the work is \( O(n \log n)^2 \). A finer analysis using concavity of the log function can then establish that the work is in fact \( O(n \log n) \). You will revisit this problem in 15-210, where you will see an algorithm with span \( O((\log n)^2) \), better than what we have here.
- **Merge**\((t_1, t_2)\) has span \(O(d_1d_2)\), where \(d_1\) and \(d_2\) are the depths of \(t_1\) and \(t_2\), respectively. To see this, observe that **Merge** calls **SplitAt** on \(t_2\) and **Merge** makes two (independent, i.e., parallelizable) recursive calls. The first arguments in these recursive calls are subtrees of \(t_1\) with depths no greater than \(d_1 - 1\). The second arguments are subtrees of \(t_2\), one of which might have depth \(d_2\). Thus (plus a base case):

\[
S_{\text{Merge}}(d_1, d_2) \leq k_0 + S_{\text{SplitAt}}(d_2) + S_{\text{Merge}}(d_1 - 1, d_2) \\
\leq k_1 + k_2 \cdot d_2 + S_{\text{Merge}}(d_1 - 1, d_2)
\]

for some constants \(k_0\), \(k_1\), and \(k_2\). This recurrence has a solution that is \(O(d_1d_2)\).

- Assuming that the trees produced by **Msort** are **balanced**, so that their depth is about the logarithm of their size, **Msort**\((t)\) has span \(O(d^3)\), where \(d\) is the depth of \(t\). To see this, observe that **Msort** makes two (independent, i.e., parallelizable) recursive calls to subtrees of depth \(d - 1\) or less, followed by a call to **Merge** and a call to **Ins**. Thus (plus a base case):

\[
S_{\text{Msort}}(d) \leq k + S_{\text{Ins}}(2d) + S_{\text{Merge}}(d - 1, d - 1) + S_{\text{Msort}}(d - 1) \\
\leq c_0 + c_1 \cdot d + c_2 \cdot (d - 1)^2 + S_{\text{Msort}}(d - 1)
\]

for balanced trees of depth \(d > 1\), for some constants \(k\), \(c_0\), \(c_1\), and \(c_2\). Expanding out, and observing that the sum of the first \(d\) squares is proportional to \(d^3\), we deduce that the span is \(O(d^3)\). For a balanced tree, with size \(n\) and depth \(d\), \(d\) is \(O(\log n)\). Our analysis therefore shows that the span for **Msort**\((t)\) on balanced trees of size \(n\) is \(O((\log n)^3)\).

Thus (ignoring constants), when we sort a billion integers in a balanced tree, the length of the longest critical path is about 27000 operations, so we can exploit over a million processors!

**Caution:** This would be true, except that we haven’t justified our balance assumption. Indeed, there is a bug in the previous analysis. Even if we assume that the original tree passed to **Msort** is balanced, we cannot guarantee that the recursive sorting within **Msort** will produce balanced trees or that calling **Merge** on balanced trees will maintain balance. In fact, as written, **Msort** need not maintain balance. There is a simple fix: rebalance within **Msort**. This can be done in a way that does not affect the asymptotic work and span results, but we will not discuss details here. More generally, later in the course, we will discuss how to implement binary trees that maintain balance.

**Comment:**
There is a way to define a version of **Msort** that avoids using **Ins**, instead calling **Merge**:

```haskell
fun Msort' Empty = Empty
| Msort' (Node(t1, x, t2)) = 
    Merge(Node(Empty, x, Empty), Merge(Msort' t1, Msort' t2))
```

Are **Msort** and **Msort’** extensionally equivalent?

Which of **Msort** and **Msort’** is more efficient?