15-150 Spring 2018
Lecture 8
Sorting Integer Trees
assessment

- `msort(L)` has $O(n \log n)$ work, where $n$ is the length of $L$

- `msort(L)` has $O(n)$ span

- So the speed-up factor obtainable by exploiting parallelism is $\log n$

  ... *in principle*, we can speed up the mergesort algorithm by a factor of $\log n$
  using parallel processing

*To do any better, need a different data structure...*
trees

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

• A user-defined type named tree
• With constructors Empty and Node

Empty : tree
Node : tree * int * tree -> tree
tree values
An inductive definition

Every tree value is either **Empty**
or **Node**(\(t_1, x, t_2\)), where \(t_1\) and \(t_2\) are tree values
and \(x\) is an integer.

**Contrast with integer lists:**

Every integer list value is either **nil**
or \(x::L\), where \(L\) is an integer list value
and \(x\) is an integer.
Inorder traversal

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

```
[2, 42, 3, 21, 7]
```

the *in-order* traversal list for t
**fun** trav Empty = [ ]

| trav (Node(l, x, r)) = trav l @ (x :: trav r)

The *in-order* traversal list for the tree is [2, 42, 3, 21, 7].
**sorted trees**

Empty is a *sorted tree*

Node(l, x, r) is a *sorted tree* iff

- every integer in l is \( \leq x \),
- every integer in r is \( \geq x \),
- and l, r are *sorted trees*

**Theorem**

t is a *sorted tree* iff

\( \text{trav(t)} \) is a *sorted list*
Divide and conquer

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

Msort : tree -> tree

REQUIRES: true

ENSURES: Msort(t) is a sorted tree consisting of the items of t

fun Msort Empty = Empty

| Msort (Node(l, x, r)) =
| Ins (x, Merge(Msort l, Msort r))
tree insertion

Ins : int * tree -> tree

REQUIRES: t is a sorted tree

ENSURES: Ins(x,t) is a sorted tree consisting of x and all of t

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

Ins(4, 3)  => 3
          /   \   \
         /     /
        /       \
       /         \
      /           \
     /             \
    /               \
   /                 \
  /                   \
 => 3
  /   \
 /     \
/       \
\       \
\       \
\       \
\       \
\       \
\   2   \ 6
\   \   \ \
\    \    \
\     \   \
\      \ 5
\      

Ins(4, 6)  => 3
          /   \   \
         /     /
        /       \
       /         \
      /           \
     /             \
    /               \
   /                 \
  /                   \
 => 3
  /   \
 /     \
/       \\ 
\       \
\       \
\       \
\       \
\   2   \ 6
\   \   \ \
\    \    \
\     \   \
\      \ 5
\      

Ins(4, 5)
Goals for Today

• State how merge sort works for integer trees
• Write code for merge sort
• Write recurrences for our code and find big-O bounds for span
fun size Empty = 0
    | size (Node(t1, _, t2)) = size tl + size t2 + 1

Uses tree patterns
Recursion is \textit{structural} 

Easy to prove \textit{by structural induction} that for all trees $t$, 
\[ \text{size}(t) = \text{a non-negative integer} \]
some facts

• Size is always non-negative
  • \( \text{size}(t) \geq 0 \)

• Children have smaller size
  • \( \text{size}(t_i) < \text{size}(\text{Node}(t_1, x, t_2)) \)

• Many recursive functions on trees make recursive calls on trees with smaller size.
  • Can use \textit{induction on size} to prove correctness.
**depth**
(or *height*)

```
fun depth Empty = 0
| depth (Node(t1, _, t2)) = Int.max(depth t1, depth t2) + 1
```

Can prove by structural induction that for all trees `t`,

```
depth(t) = a non-negative integer
```

the length of longest path from root to a leaf node
some facts

• For all trees $t$, $\text{depth}(t) \geq 0$.

• If $t = \text{Node}(t_1, x, t_2)$,
  $\text{depth}(t_1) < \text{depth}(t)$ and $\text{depth}(t_2) < \text{depth}(t)$.

• Many recursive functions on trees make recursive calls on trees with smaller depth.

• Can use induction on $\text{depth}$ to prove properties or analyze efficiency.
more facts

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

\[
\text{depth}(\text{Ins}(x, t)) \leq \text{depth } t + 1
\]

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

\[
\text{size}(\text{Ins}(x, t)) \leq \text{size } t + 1
\]
tree merge

Merge : tree * tree -> tree

REQUIRES: $t_1$ and $t_2$ are sorted trees

ENSURES: $\text{Merge}(t_1, t_2)$ is a sorted tree consisting of the items of $t_1$ and $t_2$

$\text{Merge (Node}(l_1, x, r_1), t_2) = ???$

We could split $t_2$ into two subtrees $(l_2, r_2)$, then do $\text{Node}(\text{Merge}(l_1, l_2), x, \text{Merge}(r_1, r_2))$

But we need to stay sorted and not lose any data

So split should use the value of $x$ and build $(l_2, r_2)$ so that $l_2 \leq x \leq r_2$ and…
SplitAt

SplitAt : int * tree -> tree * tree

If t is sorted,
  SplitAt(x, t) returns
    a pair of trees (t₁, t₂) such that
    every integer in t₁ is ≤ x,
    every integer in t₂ is ≥ x,
    and t₁, t₂ is a perm of t.
Plan

Define SplitAt(t) using \textit{structural recursion}

- SplitAt(x, Node(l, y, r)) should
  - \textit{compare} x and y
  - call SplitAt(x, -) on a \textit{sub}tree
  - build the result
fun SplitAt(x, Empty) = (Empty, Empty)

| SplitAt(x, Node(l, y, r)) =
| case 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
Merge

Merge : tree * tree -> tree

REQUIRES: \( t_1 \) and \( t_2 \) are sorted trees

ENSURES: Merge\((t_1, t_2)\) returns a sorted tree consisting of the items of \( t_1 \) and \( t_2 \)

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

(as we promised!)
Merge:

Merge : tree * tree -> tree

REQUIRES: t₁ and t₂ are sorted trees

ENSURES: Merge(t₁, t₂) returns a sorted tree consisting of the items of t₁ and t₂

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

(as we promised!)
Span of Ins

```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)
```

(no parallelism!)

For a tree of depth $d>0$,

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

$S_{Ins}(d)$ is $O(d)$
**Span of SplitAt**

```
fun SplitAt(x, Empty) = (Empty, Empty)
  | SplitAt(x, Node(l, y, r)) =
      case 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
```

(no parallelism!)

For a tree of depth $d > 0$,

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

$S_{\text{SplitAt}}(d)$ is $O(d)$
Span of Merge

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

\[ S_{\text{Merge}}(d_1,d_2) = c_3 + S_{\text{SplitAt}}(d_2) + \max(S_{\text{Merge}}(d_1-1, d_2), S_{\text{Merge}}(d_1-1, d_2)) \]

depth of l2 and r2 are bounded above by \( d_2 \)

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

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

\[ S_{\text{Merge}}(d) \text{ is } \mathcal{O}(d_1d_2) \]
Span of Msort

\[
\text{fun } \text{Msort} \text{ Empty} = \text{Empty} \quad \text{independent}
\]

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

For a balanced tree with \( d > 0 \)

\[
S_{\text{Msort}}(d) = c_4 + \max(S_{\text{Msort}}(d-1), S_{\text{Msort}}(d-1))
\]

\[
+ S_{\text{Merge}}(d_1,d_2) + S_{\text{Ins}}(d_3)
\]

depths of trees produced by Msort   depth of tree produced by merge
Span of Msort

\[
\begin{align*}
\text{fun } & \text{ Msort Empty } = \text{ Empty } \quad \text{ independent } \\
\text{Msort } & \text{ (Node(t1, x, t2)) } = \\
\text{Ins } & \text{ (x, Merge(Msort t1, Msort t2))}
\end{align*}
\]

For a balanced tree with \( d > 0 \)

\[
S_{\text{Msort}}(d) = c_4 + \max(S_{\text{Msort}}(d-1), S_{\text{Msort}}(d-1)) + S_{\text{Merge}}(d-1,d-1) + S_{\text{Ins}}(2d - 2) = S_{\text{Msort}}(d-1) + O(d^2)
\]

\( S_{\text{Msort}}(d) \) is \( O(d^3) \)
Span of Msort

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

For a balanced tree with \( n \) nodes, depth \( d = O(\log n) \)

\( S_{\text{Msort}}(n) \) is \( O((\log n)^3) \)
Really?

• The balance assumptions are *not* realistic!

• But we could design a *rebalancing* function...

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

• Or implement an *abstract type* of *balanced trees* with *Ins* and *Merge* functions that *preserve balance*
## Summary

<table>
<thead>
<tr>
<th></th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>List insertion sort</strong></td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td><strong>List merge sort</strong></td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><strong>Tree merge sort</strong></td>
<td></td>
<td>$O((\log n)^3)$</td>
</tr>
</tbody>
</table>
losing balance

Msort can produce badly balanced trees
Q: How to prove that Msort is correct?
A: Use structural induction.

First prove that the helper functions Merge, SplitAt, Ins are correct. Again use structural induction.

The helper specs were carefully chosen to make the proof of Msort straightforward. (An easy structural induction, using the proven facts about helpers.)
Notes

• $\text{Ins}(x, t)$ is inductive on (structure of) $t$
• $\text{SplitAt}(x, t)$ is inductive on $t$
• $\text{Merge}(t_1, t_2)$ is inductive on $t_1$
• $\text{Msort}(t)$ is inductive on $t$

look back at the code
and see why this is true
Rebalancing
(* rebalance : tree -> tree
   REQUIRES: true
   ENSURES: rebalance(T) ==> T' such that:
       (a) trav(T) == trav(T')
       (b) depth(T') == ceiling(log_2(size(T'))))
*)

defun rebalance (Empty : tree) : tree = Empty
| rebalance (T) =
    let
        val (left, x, right) : tree*int*tree = halves(T)
    in
        Node(rebalance left, x, rebalance right)
    end
Bonus slides
(* Rebalancing code: *)
(* Caution: As is, this code is inefficient. In order to gain *)
(* efficiency, one needs to redefine Node to contain a size field, thus *)
(* allowing for a constant time implementation of the size function. *)

(* takeanddrop : tree * int -> tree * tree
  REQUIRES: 0 <= i <= size(T)
  ENSURES: takeanddrop(T,i) ==> (T1,T2) such that:
    (a) max(depth(T1), depth(T2)) <= depth(T)
    (b) size(T1) == i
    (c) trav(T) == trav(T1) @ trav(T2)
*)

fun takeanddrop (T : tree, 0 : int) : tree*tree = (Empty, T)
| takeanddrop (Empty, _) = raise Fail "not enough elements"
| takeanddrop (Node(left, x ,right), i) =
  (case Int.compare(i, size(left)) of
   LESS => let
     val (t1,t2) : tree*tree = takeanddrop(left, i)
     in
     (t1, Node(t2, x, right))
   end
   | EQUAL => (left, Node(Empty, x , right))
   | GREATER => let
     val (t1,t2) : tree*tree = takeanddrop(right, i - 1 - size(left))
     in
     (Node(left, x, t1), t2)
   end)
(* halves : tree -> tree * int * tree
  REQUIRES: T is not Empty
  ENSURES: halves(T) ==> (T1, T2) such that:
    (a) size(T1) == size(T) div 2
    (b) trav(t) == trav(T1) @ (x::trav(T2))
*)

fun halves (T : tree) : tree * int * tree =
  let
    val (T1, T2) : tree * tree = takeanddrop(T, (size T) div 2)
    val (Node(Empty, x, Empty), T3) : tree * tree = takeanddrop(T2, 1)
  in
    (T1, x, T3)
  end