trees vs. lists

• Representing a collection as a (balanced) tree instead of a list may yield a parallel speed-up
• Using a sorted (balanced) tree may even yield faster sequential code
• With lists, even sorted lists, there’s no potential for parallelism
• But badly balanced trees are no better than lists, and balance may be hard to achieve!
balanced trees

- Empty is size-balanced
- Node(A, x, B) is size-balanced iff
  \[|\text{size}(A) - \text{size}(B)| \leq 1\]
  and A, B are size-balanced

We’ll just say “balanced”…
fun takedrop (0, L) = ([ ], L)
|  takedrop (n, x::L) = let
|    val (A, B) = takedrop (n-1, L)
|    in
|      (x::A, B)
|    end

fun list2tree [ ] = Empty
|  list2tree L =
|    let
|      val n = length L
|      val (A, x::B) = takedrop (n div 2, L)
|      in
|        Node(list2tree A, x, list2tree B)
|      end

could this pattern fail?
specs

takedrop : int * int list -> int list * int list
REQUIRES 0 <= n <= length L
ENSURES takedrop (n, L) = (A, B) such that
L = A@B and length A = n

list2tree : int list -> tree
ENSURES
    list2tree L = a balanced tree containing the items in L

list2tree : int list -> tree
ENSURES
    list2tree L = a balanced tree T such that inord(T) = L
Sorted trees

Empty is a sorted tree

Node\((t_1, x, t_2)\) is a sorted tree iff

- every integer in \(t_1\) is \(\leq x\),
- every integer in \(t_2\) is \(\geq x\),
- and \(t_1\), \(t_2\) are sorted trees

Theorem

\(T\) is a sorted tree iff

inord\((T)\) is a sorted list

\(\begin{array}{c}
42 \\
42 \\
3
\end{array}\)

\(\begin{array}{c}
42 \\
81 \\
3
\end{array}\)

\(\begin{array}{c}
14 \\
81 \\
3
\end{array}\)

\(\begin{array}{c}
42 \\
57 \\
99
\end{array}\)
fun all (p : int -> bool, T : tree) : bool =
case T of
  Empty       => true
| Node(A, x, B) =>
    (p x) andalso all (p, A) andalso all (p, B)

REQUIRES p is total

ENSURES all (p, T) = true iff every integer in T satisfies p
fun is_sorted (T : tree) : bool = 
  case T of 
    Empty => true 
  | Node(A, x, B) =>
    all (fn y => y <= x, A) andalso
    all (fn y => y >= x, B) andalso
    is_sorted A andalso is_sorted B

is_sorted T = true iff T is a sorted tree
searching a list

fun mem (x, [ ]) = false
| mem (x, y::L) = (x = y) orelse mem (x, L)

REQUIRES true
ENSURES mem (x, L) = true iff x is in L

W_{mem}(x, L) is O(length L)
Worst case: when x is not in L
  or x is last element of L

S_{mem}(x, L) is also O(length L)
fun mem (x, [ ]) = false
| mem (x, y::L) = case Int.compare(x, y) of
| LESS         => false
| EQUAL        => true
| GREATER      => mem (x, L)

REQUIRES L is a sorted list
ENSURES mem (x, L) = true iff x is in L

W_{\text{mem}}(x, L) is \(O(\text{length } L)\)

Worst case: when x is > all of L…

S_{\text{mem}}(x, L) is also \(O(\text{length } L)\)
searching

fun mem (x, Empty) = false
| mem (x, Node(A, y, B)) =
  (x = y) orelse mem (x, A) orelse mem (x, B)

REQUIRES T is a tree
ENSURES mem (x, T) = true iff x is in T

\[ W_{mem(x,T)} \text{ is } O(\text{size } T) \]
Worst case: when x is not in T
or x is inorder-last element of T

\[ S_{mem(x,T)} \text{ is also } O(\text{size } T) \]
fun mem (x, Empty) = false
|      mem (x, Node(A, y, B)) =
  (x = y) orelse
  let
    val (a, b) = (mem (x, A), mem (x, B))
  in
    a orelse b
  end.

$W_{\text{mem}}(x, T)$ is $O(\text{size } T)$

$S_{\text{mem}}(x, T)$ is $O(\text{depth } T)$
fun mem (x, Empty) = false
| mem (x, Node(A, y, B)) =
    case Int.compare(x, y) of
    LESS => mem(x, A)
    | EQUAL => true
    | GREATER => mem (x, B)

REQUIRES T is a sorted tree
ENSURES mem (x,T) = true iff x is in T

\[ W_{\text{mem}}(x,T) \text{ is } O(\text{depth } T) \]
\[ S_{\text{mem}}(x,T) \text{ is } O(\text{depth } T) \]
trees $\gg$ lists?

- Representing a collection of integers as a (balanced) tree may yield a parallel speed-up.
- Using a sorted (and balanced) tree may even support faster sequential code.
- Using lists, even sorted lists, only allows sequential code, and precludes parallelism.
- Badly balanced trees are no better than lists!
sorting a tree

- If the tree is Empty, do nothing
- Otherwise
  (recursively) sort the two children, then
  merge the sorted children, then
  insert the root value

We’ll design helpers to insert and merge

merge will also need a helper to split a tree in two
inserting in a tree

\[ \text{Ins} : \text{int} \times \text{tree} \rightarrow \text{tree} \]

**REQUIRES** \( t \) is a sorted tree

**ENSURES** \( \text{Ins}(x, t) \) is a sorted tree consisting of \( x \) and all of \( t \)

\[
\text{fun } \text{Ins} \ (x, \text{Empty}) = \text{Node}(\text{Empty}, x, \text{Empty}) \\
| \quad \text{Ins} \ (x, \text{Node}(t1, y, t2)) = \\
\quad \quad \text{case } \text{compare}(x, y) \text{ of} \\
\quad \quad \quad \text{GREATER} \Rightarrow \text{Node}(t1, y, \text{Ins}(x, t2)) \\
\quad \quad \quad \_ \quad \Rightarrow \text{Node}(\text{Ins}(x, t1), y, t2)
\]

(contrast with list insertion)
inserting in a list

ins : int * int list -> int list

fun ins (x, [ ]) = [x]

| ins (x, y::L) = case compare(x, y) of
|     GREATER => y::ins(x, L)  as before
|          _        => x::y::L

For all sorted integer lists L,
ins(x, L) = a sorted permutation of x::L.
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

\[
\begin{array}{c}
\text{Ins}(4, 3) \\
\text{Ins}(4, 6) \\
\text{Ins}(4, 5)
\end{array}
\]
merging trees

Merge : tree * tree -> tree

REQUIRES t₁ and t₂ are sorted trees

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

Merge (Node(L₁,x,R₁), t₂) = ???

We could split t₂ into two subtrees (L₂, R₂), then do Node(Merge(L₁,L₂), x, Merge(R₁,R₂))

But we need to stay sorted and not lose data…

… so our split should use x and build (L₂, R₂) so that L₂ ≤ x ≤ R₂ …
splitting a tree

SplitAt : int * tree -> tree * tree

REQUIRES t is a sorted tree

ENSURES SplitAt(x, t) =
    a pair of sorted trees (u₁, u₂) such that
    u₁ ≤ x ≤ u₂ and u₁, u₂ is a perm of t

Not completely specific, but that’s OKAY!
SplitAt

SplitAt : int * tree -> tree * tree

If t is sorted,
    SplitAt(x, t) = a pair of sorted trees (u₁, u₂) such that
        every integer in u₁ is ≤ x,
        every integer in u₂ is ≥ x,
        and u₁, u₂ is a perm of t.
Plan

Define SplitAt(x, t) by a *structural recursion*

• SplitAt(x, Node(t₁, y, t₂)) should
  • *compare* x and y
  • call SplitAt(x, -) on a *subtree*
  • build the result
**SplitAt**

**SplitAt :** int * tree -> tree * tree

**REQUIRES t** is a sorted tree

**ENSURES SplitAt(x, t) =** a pair of sorted trees \((u_1, u_2)\)

such that \(u_1 \leq x \leq u_2\) and \(u_1, u_2\) is a perm of \(t\)

```plaintext
fun SplitAt(x, Empty) = (Empty, Empty)

| SplitAt(x, Node(t1, y, t2)) =
| case compare(y, x) of
|   GREATER =>
|     let val (l1, r1) = SplitAt(x, t1) in (l1, Node(r1, y, t2)) end
| _ =>
|     let val (l2, r2) = SplitAt(x, t2) in (Node(t1, y, l2), r2) end
```
**Merge**

Merge : tree * tree -> tree

REQUIRES $t_1$ and $t_2$ are sorted trees

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

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

(as we promised!)
Merge

Merge : tree * tree -> tree

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

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

fun Merge (Empty, t2) = t2

let
val (l2, r2) = SplitAt(x, t2)

in

Node(Merge(l1, l2), x, Merge(r1, r2))
end
depth lemma

For all trees $t$ and integers $x$,
$$\text{depth}({\text{Ins}}(x, t)) \leq \text{depth} t + 1$$

For all trees $t$, if $\text{SplitAt}(y, t) = (t_1, t_2)$, then
$$\text{depth}(t_1) \leq \text{depth} t \land \text{depth}(t_2) \leq \text{depth} t$$

For all trees $t_1$ and $t_2$,
$$\text{depth}(\text{Merge}(t_1, t_2)) \leq \text{depth} t_1 + \text{depth} t_2$$

(no, we won’t prove this!)
Msort

Msort : tree -> tree
REQUIRES true
ENSURES Msort(t) = a sorted tree consisting of the items of t

fun Msort Empty = Empty
| Msort (Node(t1, x, t2)) =
  Ins (x, Merge(Msort t1, Msort t2))
Correct?

- **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.)
Msort

Msort : tree -> tree

REQUIRES true
ENSURES Msort(t) = a sorted tree consisting of the items of t

fun Msort Empty = Empty
  | Msort (Node(t1, x, t2)) =
    Ins (x, Merge(Msort t1, Msort t2))
example

val A = list2tree [4,1,2]
val B = list2tree [3,5,0]
val T = Node(A, 42, B)
val S = Msort T