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 enable faster code, e.g. for searching

• With lists, even sorted lists, there’s less potential for parallelism

• But badly balanced trees are no better than lists, and balance may be hard to achieve!
balanced

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

An inductive characterization of the set of balanced trees
balanced trees

We can build a balanced tree from a list...

... and (if we do it right) we can get the same list back again by in-order traversal

\[ [4, 1, 2] \xrightarrow{\text{list2tree}} \begin{tikzpicture} \node [circle, draw] {1} child {node [circle, draw] {4}} child {node [circle, draw] {2}} \end{tikzpicture} \xleftarrow{\text{inord}}\]
fun takedrop (0, L) = ([ ], L)
| takedrop (n, x::R) = let
|   val (A, B) = takedrop (n-1, R)
| in
|   (x::A, B)
| end

takedrop : int * 'a list -> 'a list * 'a list

  takedrop (m, L) = a pair (A, B)
such that length A = m and A@B = L

  “chops list into two”
fun list2tree [ ] = Empty
| list2tree [x] = Node(Empty, x, 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

list2tree [4,1,2] = ???

QUESTION
Why is the pattern (A, x::B) justifiable here?
question

- Would it have been OK to omit the [x] clause?

```plaintext
fun list2tree [ ] = Empty
| list2tree [x] = Node(Empty, x, 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
```

```plaintext
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
```

list2tree [4] = ???
takedrop : int * int list -> int list * int list
REQUIRES 0 <= n <= length L
ENSURES takedrop (n, L) = a pair of lists (A, B) such that L = A@B and length A = n

list2tree : int list -> int tree
ENSURES
list2tree L = a balanced tree T such that inord(T) = L
**takedrop proof**

```plaintext
fun takedrop (0, L) = ([ ], L)

|     takedrop (n, x::R) = let
|       val (A, B) = takedrop (n-1, R)
|       in
|         (x::A, B)
|       end
```

For all L : int list and n : int with 0 <= n <= length L,
takedrop (n, L) = a pair of lists (A, B)
such that L = A @ B and length A = n

**PROOF**

By induction on length of L
or by structural induction on L

- *the recursive call is on the tail of list*
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

For all L : int list,
list2tree L = a balanced tree T
such that inord(T) = L

PROOF
By (strong) induction on length of L
- in recursive calls, length A and length B are less than length L
examples

val A = list2tree [4, 1, 2]
val B = list2tree [3, 5]
val T = Node(A, 42, B)
sorted trees

Empty is sorted

Node(A, x, B) is sorted iff

  every integer in A is \( \leq x \),
  every integer in B is \( \geq x \),
  and A and B are sorted

Theorem

T is a sorted tree iff

inord T is a sorted list
fun all (p : int -> bool, T : int 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
all

all : (int -> bool) * int tree -> bool

fun all (p, Empty) = true
| all (p, 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 : int 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
is_sorted

fun is_sorted Empty = true
| is_sorted (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

\[
\text{fun mem (x, [ ])} = \text{false} \\
| \quad \text{mem (x, y::L)} = (x = y) \text{ orelse mem (x, L)}
\]

- **REQUIRES** true
- **ENSURES** \( \text{mem (x, L)} = \text{true iff x is in L} \)

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

Worst case: when \( x \) is not in \( L \)

or \( x \) is last element of \( L \)

\[ S_{\text{mem}}(x, L) \text{ is also } O(\text{length } L) \]
**searching a sorted list**

```haskell
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
```

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

Worst case: when x is > all of L…

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

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

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

Worst case: when \( x \) is not in \( T \)
or \( x \) is \textit{inorder-last} element of \( T \)

\[ S_{\text{mem}}(x, T) \] 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
in
| a orelse b
end

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

$W_{mem}(x,T)$ is $O(\text{depth } T)$
$S_{mem}(x,T)$ 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 support faster code, e.g. for searching

• With lists, even sorted lists, there is less opportunity for parallelism

• But an unbalanced tree may be no better than a list
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

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)

(contrast with list insertion)
inserting in a list

\[
\text{ins : int } \times \text{ int list } \rightarrow \text{ int list}
\]

\[
\text{fun } \text{ins} \ (x, \ [ \ ]) = [x] \\
\mid \text{ins} \ (x, \ y::L) = \\
\quad \text{case} \ \text{compare}(x, \ y) \ \text{of} \\
\quad \quad \text{GREATER} \Rightarrow y::\text{ins}(x, \ L) \\
\quad \quad \_ \_ \_ \_ \_ \Rightarrow x::y::L
\]

For all sorted integer lists \(L\),

\[\text{ins}(x, \ L) = \text{a sorted permutation of } x::L.\]
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)
  1
   6
  2 5

= 3
  1
   Ins(4, 6)
  2 5

= 3
  1
   6
  2
   Ins(4, 5)
value equations

\[ \text{Ins}(x, \text{Empty}) = \text{Node}(\text{Empty}, x, \text{Empty}) \]

\[ \text{Ins}(x, \text{Node}(t_1, y, t_2)) = \begin{cases} 
\text{Node}(t_1, y, \text{Ins}(x, t_2)) & \text{if } x > y \\
\text{Node}(\text{Ins}(x, t_1), y, t_2) & \text{if } x \leq y 
\end{cases} \]

These equations hold, for all integer values \( x, y \) and all tree values \( t_1, t_2 \)

By definition of \( \text{Ins} \)

\[ \text{Ins}(4, 3) = 3 \]

\begin{align*}
\text{T sorted} & \quad \text{Ins}(4, T) \text{ sorted}
\end{align*}
merging trees

Merge : int tree * int tree -> int tree

REQUIRES t_1 and t_2 are sorted trees

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

Merge (Node(L_1,x,R_1), t_2) = ???

We could split t_2 into two subtrees (L_2, R_2), then do Node(Merge(L_1,L_2), x, Merge(R_1,R_2))

But we need to stay sorted and not lose data…

… so our split should use x and build (L_2, R_2) so that L_2 ≤ x ≤ R_2 …
splitting a tree

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

If t is sorted,

\[ \text{SplitAt}(x, t) = \text{a pair of trees } (u_1, u_2) \text{ such that} \]
\[ \text{every integer in } u_1 \text{ is } \leq x, \]
\[ \text{every integer in } u_2 \text{ is } \geq x, \]
\[ \text{and } u_1, u_2 \text{ is a perm of } t. \]

Any ideas???
Plan

Define SplitAt(t) using *structural recursion*

- SplitAt(x, Node(t_1, y, t_2)) should
  - *compare* x and y
  - call SplitAt(x, -) on a *subtree*
  - build the result
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

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

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

Merge : int tree * int tree -> int tree

REQUIRES $t_1$ and $t_2$ are sorted trees

ENSURES $\text{Merge}(t_1, t_2) = \text{a sorted tree}$

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) = \text{SplitAt}(x, t2)
|   in
|     Node(Merge(l1, l2), x, Merge(r1, r2))
| end

(as we promised!)
Merge

Merge : int tree * int tree -> int tree

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

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
depth lemma

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

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

For all int 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 : int tree -> int 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 : int tree -> int 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