last time

• Sorting a list of integers
  • insertion sort \( O(n^2) \) for lists of length \( n \)
  • merge sort \( O(n \log n) \)

• Specifications and proofs
  • *helper functions* that really help
principles

• Every function needs a spec
• Every spec needs a proof
• Recursive functions need inductive proofs
  • Pick an *appropriate* method...
  • Choose helper functions wisely!

*proof of msort was easy,*
because of *split* and *merge*
the joy of specs

• The **proof** for `msort` relied only on the **specification** proven for `split` and the **specification** proven for `merge`.

• We can *replace* `split` and `merge` by *any* functions that satisfy the **specifications**, and the `msort` proof is still valid!
even more joy

- The work analysis for `msort` relied on the correctness of `split` and `merge` and their work.
- We can replace `split` and `merge` by any functions that satisfy the specifications and have the same work, and get a version of `msort` with the same work as before (asymptotically)!
advantages

• These joyful comments are intended to convince you of the advantages of *compositional reasoning*!

• We can reason about correctness, and analyze efficiency, in a syntax-directed way.
so far

• We proved correctness of isort and showed that the work for isort L is $O(n^2)$

• We proved correctness of msort and showed that the work for msort L is $O(n \log n)$

```fun msort [ ] = [ ]
| msort [x] = [x]
| msort L = let val (A, B) = split L in
    merge (msort A, msort B) end

W_{split}(n) = O(n)
W_{merge}(n) = O(n)
W_{msort}(n) = O(n) + 2 W_{msort}(n \text{ div } 2)```
can we do better?

Q: Would parallel processing be beneficial?

A: Find the span for isort L and msort L

- If the span is asymptotically better than the work, there’s a potential speed-up
can we do better?

Q: Would parallel processing be beneficial?
A: Find the span for isort L and msort L

- If the span is asymptotically better than the work, there’s a potential speed-up

| add | the work for sub-expressions |
| max | the span for independent sub-expressions |
| add | the span for dependent sub-expressions |
The list ops in `ins` are **sequential**

\[
\text{fun } \text{ins}(x, \text{[]} ) = [x] \\
\mid \text{ins}(x, y::L) = \text{if } y < x \text{ then } y::\text{ins}(x, L) \text{ else } x::y::L
\]

**isort** can’t be parallelized - code is dependent

\[
\text{fun } \text{isort} \text{ [ ] } = \text{ [ ]} \\
\mid \text{isort} \text{ (x::L) } = \text{ins}(x, \text{isort } L)
\]
• The list ops in \texttt{ins} are \textit{sequential}

\begin{verbatim}
fun ins(x, [ ]) = [x]
| ins(x, y::L) = if y<x then y::ins(x, L) else x::y::L

W_{ins}(0) = 1
W_{ins}(n) = W_{ins}(n-1) + 1
\end{verbatim}

• \texttt{isort} can’t be parallelized - code is dependent

\begin{verbatim}
fun isort [ ] = [ ]
| isort (x::L) = ins(x, isort L)
\end{verbatim}
• The list ops in ins are **sequential**

```plaintext
fun ins(x, [ ]) = [x]
| ins(x, y::L) = if y<x then y::ins(x, L) else x::y::L

W_{ins}(0) = 1
W_{ins}(n) = W_{ins}(n-1) + 1

W_{ins}(n) is O(n)
```

• **isort** can't be parallelized - code is dependent

```plaintext
fun isort [ ] = [ ]
| isort (x::L) = ins(x, isort L)
```
• The list ops in \texttt{ins} are \textit{sequential}

\begin{verbatim}
fun ins(x, [ ]) = [x]
| ins(x, y::L) = if y<x then y::ins(x, L) else x::y::L

W_{\text{ins}}(0) = 1
W_{\text{ins}}(n) = W_{\text{ins}}(n-1) + 1 \quad \text{W}_{\text{ins}}(n) \text{ is } O(n)

S_{\text{ins}}(0) = 1
S_{\text{ins}}(n) = S_{\text{ins}}(n-1) + 1
\end{verbatim}

• \texttt{isort} can’t be parallelized - code is dependent

\begin{verbatim}
fun isort [ ] = [ ]
| isort (x::L) = ins(x, isort L)
\end{verbatim}
• The list ops in `ins` are **sequential**

```
fun ins(x, [ ]) = [x]
   | ins(x, y::L) = if y<x then y::ins(x, L) else x::y::L
```

\[
W_{ins}(0) = 1 \\
W_{ins}(n) = W_{ins}(n-1) + 1 \\
W_{ins}(n) \text{ is } O(n)
\]

\[
S_{ins}(0) = 1 \\
S_{ins}(n) = S_{ins}(n-1) + 1 \\
S_{ins}(n) \text{ is } O(n)
\]

• `isort` can’t be parallelized - code is dependent

```
fun isort [ ] = [ ]
   | isort (x::L) = ins(x, isort L)
```
• The list ops in \texttt{ins} are \textit{sequential}

\begin{verbatim}
fun ins(x, []) = [x]
| ins(x, y::L) = if y<x then y::ins(x, L) else x::y::L

W_{ins}(0) = 1
W_{ins}(n) = W_{ins}(n-1) + 1 \quad W_{ins}(n) \text{ is } O(n)

S_{ins}(0) = 1
S_{ins}(n) = S_{ins}(n-1) + 1 \quad S_{ins}(n) \text{ is } O(n)
\end{verbatim}

• \texttt{isort} can’t be parallelized - code is dependent

\begin{verbatim}
fun isort [] = []
| isort (x::L) = ins(x, isort L)

W_{isort}(0) = 1
W_{isort}(n) = W_{isort}(n-1) + W_{ins}(n-1) + 1
\quad = O(n) + W_{isort}(n-1)
\end{verbatim}
• The list ops in ins are **sequential**

  ```haskell
  fun ins(x, [ ]) = [x]
  | ins(x, y::L) = if y<x then y::ins(x, L) else x::y::L
  W_{\text{ins}}(0) = 1
  W_{\text{ins}}(n) = W_{\text{ins}}(n-1) + 1  \quad W_{\text{ins}}(n) \text{ is } O(n)
  S_{\text{ins}}(0) = 1
  S_{\text{ins}}(n) = S_{\text{ins}}(n-1) + 1  \quad S_{\text{ins}}(n) \text{ is } O(n)
  ```

• isort can’t be parallelized - code is dependent

  ```haskell
  fun isort [ ] = [ ]
  | isort (x::L) = ins(x, isort L)
  W_{\text{isort}}(0) = 1
  W_{\text{isort}}(n) = W_{\text{isort}}(n-1) + W_{\text{ins}}(n-1) + 1
  = O(n) + W_{\text{isort}}(n-1)  \quad W_{\text{isort}}(n) \text{ is } O(n^2)
  ```
• The list ops in \texttt{ins} are \textit{sequential}

\[
\text{fun ins}(x, [ ]) = [x] \\
| \quad \text{ins}(x, y::L) = \text{if } y<x \text{ then } y::\text{ins}(x, L) \text{ else } x::y::L
\]

\[
W_{\text{ins}}(0) = 1 \\
W_{\text{ins}}(n) = W_{\text{ins}}(n-1) + 1 \\
\text{W}_{\text{ins}}(n) \text{ is } \mathcal{O}(n)
\]

\[
S_{\text{ins}}(0) = 1 \\
S_{\text{ins}}(n) = S_{\text{ins}}(n-1) + 1 \\
\text{S}_{\text{ins}}(n) \text{ is } \mathcal{O}(n)
\]

• \texttt{isort} can’t be parallelized - code is dependent

\[
\text{fun isort} \ [ ] = [ ] \\
| \quad \text{isort} \ (x::L) = \text{ins}(x, \text{isort} \ L)
\]

\[
W_{\text{isort}}(0) = 1 \\
W_{\text{isort}}(n) = W_{\text{isort}}(n-1) + W_{\text{ins}}(n-1) + 1 \\
= \mathcal{O}(n) + W_{\text{isort}}(n-1) \\
\text{W}_{\text{isort}}(n) \text{ is } \mathcal{O}(n^2)
\]

\[
S_{\text{isort}}(0) = 1 \\
S_{\text{isort}}(n) = S_{\text{isort}}(n-1) + S_{\text{ins}}(n-1) + 1 \\
= \mathcal{O}(n) + S_{\text{isort}}(n-1)
\]
• The list ops in ins are **sequential**

```haskell
fun ins(x, []) = [x]
|   ins(x, y::L) = if y<x then y::ins(x, L) else x::y::L
```

\[
W_{\text{ins}}(0) = 1 \\
W_{\text{ins}}(n) = W_{\text{ins}}(n-1) + 1 \\
W_{\text{ins}}(n) \text{ is } O(n)
\]

\[
S_{\text{ins}}(0) = 1 \\
S_{\text{ins}}(n) = S_{\text{ins}}(n-1) + 1 \\
S_{\text{ins}}(n) \text{ is } O(n)
\]

• **isort** can’t be parallelized - code is dependent

```haskell
fun isort [] = []
|   isort (x::L) = ins(x, isort L)
```

\[
W_{\text{isort}}(0) = 1 \\
W_{\text{isort}}(n) = W_{\text{isort}}(n-1) + W_{\text{ins}}(n-1) + 1 \\
= O(n) + W_{\text{isort}}(n-1) \\
W_{\text{isort}}(n) \text{ is } O(n^2)
\]

\[
S_{\text{isort}}(0) = 1 \\
S_{\text{isort}}(n) = S_{\text{isort}}(n-1) + S_{\text{ins}}(n-1) + 1 \\
= O(n) + S_{\text{isort}}(n-1) \\
S_{\text{isort}}(n) \text{ is } O(n^2)
\]
The list ops in \texttt{split}, \texttt{merge} are \textit{sequential}

But we \textit{could} use parallel evaluation for the recursive calls to \texttt{msort A} and \texttt{msort B}

\begin{verbatim}
fun msort [] = []
| msort [x] = [x]
| msort L = let val (A, B) = split L in
    merge (msort A, msort B) end
\end{verbatim}

How would this affect runtime?
span of msort

```ml
fun msort [] = []
  | msort [x] = [x]
  | msort L = let val (A, B) = split L in
            merge (msort A, msort B) end
```
span of msort

fun msort [ ] = [ ]
|   msort [x] = [x]
|   msort L = let val (A, B) = split L in
|     merge (msort A, msort B) end

S_{msort}(0) = 1
S_{msort}(1) = 1
span of msort

fun msort [ ] = [ ]
| msort [x] = [x]
| msort L = let val (A, B) = split L in
merge (msort A, msort B) end

\[ S_{msort}(0) = 1 \]
\[ S_{msort}(1) = 1 \]
\[ S_{msort}(n) = S_{split}(n) + S_{msort}(n \text{ div } 2) + S_{merge}(n) + 1 \]
span of msort

fun msort [ ] = [ ]
| msort [x] = [x]
| msort L = let val (A, B) = split L in
  merge (msort A, msort B) end

S_{msort}(0) = 1
S_{msort}(1) = 1
S_{msort}(n) = S_{split}(n) + S_{msort}(n \text{ div } 2) + S_{merge}(n) + 1
span of msort

\[
\text{fun msort [ ] } = [ ] \\
| \quad \text{msort [x] } = [x] \\
| \quad \text{msort L } = \text{let val } (A, B) = \text{split L in} \\
\quad \text{merge (msort } A, \text{ msort } B) \text{ end}
\]

\[
\begin{align*}
S_{\text{msort}}(0) &= 1 \\
S_{\text{msort}}(1) &= 1 \\
S_{\text{msort}}(n) &= S_{\text{split}}(n) + S_{\text{msort}}(n \div 2) + S_{\text{merge}}(n) + 1 \\
&\text{for } n > 1
\end{align*}
\]
\[ S_{\text{msort}}(0) = 1 \]
\[ S_{\text{msort}}(1) = 1 \]
\[ S_{\text{msort}}(n) = S_{\text{split}}(n) + S_{\text{msort}}(n \div 2) + S_{\text{merge}}(n) + 1 \]
for \( n > 1 \)

\[ S_{\text{msort}}(n) = \mathcal{O}(n) + S_{\text{msort}}(n \div 2) \]
**span of msort**

\[
\text{fun } \text{msort } [ ] = [ ] \\
| \text{msort } [x] = [x] \\
| \text{msort } L = \text{let val } (A, B) = \text{split } L \text{ in} \\
| \quad \text{merge } (\text{msort } A, \text{msort } B) \text{ end}
\]

\[
\begin{align*}
S_{\text{msort}}(0) &= 1 \\
S_{\text{msort}}(1) &= 1 \\
S_{\text{msort}}(n) &= S_{\text{split}}(n) + S_{\text{msort}}(n \div 2) + S_{\text{merge}}(n) + 1 \\
& \quad \text{for } n > 1
\end{align*}
\]

\[
S_{\text{msort}}(n) = O(n) + S_{\text{msort}}(n \div 2)
\]

\[S_{\text{msort}}(n) \text{ is } O(n)\]
Deriving the **span** for `msort`

Simplify recurrence to:

\[ S(n) = n + S(n \text{ div } 2) \]
\[ = n + n/2 + S(n \text{ div } 4) \]
\[ = n + n/2 + n/4 + S(n \text{ div } 8) \]
\[ = n + n/2 + n/4 + \ldots + n/2^k \]

where \( k = \log_2 n \)

\[ = n(1 + 1/2 + 1/4 + \ldots + 1/2^k) \]
\[ \leq 2n \]

This \( S \) has same asymptotic behavior as \( S_{msort} \)

So \( S_{msort}(n) \) is \( O(n) \)
summary

• \texttt{msort(L)} has \(O(n \log n)\) work, \(O(n)\) span

• So the potential speed-up factor from parallel evaluation is \(O(\log n)\)

\[\ldots \text{in principle, we can speed up mergesort on lists by a factor of } \log n\]
• msort(L) has $O(n \log n)$ work, $O(n)$ span

• So the potential speed-up factor from parallel evaluation is $O(\log n)$

... in principle, we can speed up mersesort on lists by a factor of $\log n$

but this would require $O(n)$ parallel processors... expensive!
• \texttt{msort(L)} has $O(n \log n)$ work, $O(n)$ span

• So the potential \textit{speed-up} factor from parallel evaluation is $O(\log n)$

\ldots \textit{in principle}, we can \textit{speed up} mergesort on lists \textit{by a factor of} $\log n$
summary

• msort(L) has $O(n \log n)$ work, $O(n)$ span

• So the potential speed-up factor from parallel evaluation is $O(\log n)$

... in principle, we can speed up mergesort on lists by a factor of $\log n$

To do any better, we need a different data structure...
summary

• $\text{msort}(L)$ has $O(n \log n)$ work, $O(n)$ span

• So the potential speed-up factor from parallel evaluation is $O(\log n)$

\[\text{... in principle, we can speed up mergesort on lists by a factor of } \log n\]

To do any better, we need a different data structure...
pause for thought

• We’re going to try using trees instead of lists to represent collections of integers

• We’ll define a sorting function for trees…

• … and we’ll analyze its work and span to see if trees enable improved efficiency

• We’ll begin with some basic functions on trees

• And first, a reminder…
work is the number of evaluation steps, assuming sequential processing

span is the number of evaluation steps, assuming unlimited parallelism
work is the number of evaluation steps, assuming sequential processing

span is the number of evaluation steps, assuming unlimited parallelism

span is always $\leq$ work
**work** is the *number of evaluation steps*, assuming *sequential processing*

**span** is the *number of evaluation steps*, assuming *unlimited parallelism*

**span** is always \( \leq \) **work**

For *sequential code*, \( \text{span} = \text{work} \)
dealing with trees

• **Q:** How to represent binary trees, with data at nodes

• **A:** Use an ML datatype for binary trees

  **Q:** How to work with *special* trees

  **A:** Use an *invariant*

  • binary search trees
  • sorted trees
  • balanced trees
datatypes

• ML allows users to invent their own types

• With constructors for building values…

• … and patterns for de-constructing values

And a tailor-made form of induction…

structural induction
A datatype of trees

datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

- A user-defined type family 'a tree
- With value constructors Empty and Node
  Empty : 'a tree
  Node : 'a tree * 'a * 'a tree -> 'a tree

A value of type int tree
is a binary tree with integers at its nodes
tree values

- Empty
  an empty tree

- Node($T_1, x, T_2$)
  a non-empty tree, with $x$ at the root node, left child $T_1$, right child $T_2$

Every tree value is either Empty, or built with Node from “smaller” trees
An inductive definition

**tree values**

A tree value is either `Empty` or `Node(T_1, x, T_2)`, where `T_1` and `T_2` are tree values and `x` is a value.

**list values**

A list value is either `nil` or `x::L`, where `L` is a list value and `x` is a value.
equality (a.k.a equivalence)

for tree values

Empty = T if and only if T is Empty

Node(T₁, x, T₂) = Node(U₁, y, U₂) if and only if
T₁ = U₁, x = y and T₂ = U₂

for list values

nil = L if and only if L is nil

x::L = y::R if and only if x=y and L = R
equality types

• A type of form $t$ tree is an equality type if and only if $t$ is an equality type

```ml
fun equal(Empty, Empty) = true
|   equal(Empty, Node _) = false
|   equal(Node _, Empty) = false
|   equal(Node(A1, x1, B1), Node(A2, x2, B2)) =
   (x1 = x2) andalso equal(A1, A2) andalso equal(B1, B2);

val equal = fn : 'a tree * 'a tree -> bool
```
equality types

• A type of form \texttt{t tree} is an equality type if and only if \texttt{t} is an equality type

\begin{verbatim}
fun equal(Empty, Empty) = true
|   equal(Empty, Node _) = false
|   equal(Node _, Empty) = false
|   equal(Node(A1, x1, B1), Node(A2, x2, B2)) =
  (x1 = x2) andalso equal(A1, A2) andalso equal(B1, B2);

val equal = fn : ''a tree * ''a tree -> bool
\end{verbatim}

\texttt{stdIn:10.5 Warning: calling polyEqual}

(\textit{ignore this!})
structural definition

To define a function F on trees:

• **Base clause:**
  Define \( F(\text{Empty}) \)

• **Inductive clause:**
  Define \( F(\text{Node}(T_1, x, T_2)) \)
  using \( x \) and \( F(T_1) \) and \( F(T_2) \).

That’s enough! Why?

Contrast with structural definition for *lists*
structural induction

**To prove:** For all tree values $T$, $P(T)$ holds by *structural induction* on $T$

- **Base case:** Prove $P(\text{Empty})$.
- **Inductive case:**
  - Assume IH: $P(T_1)$ and $P(T_2)$.
  - Prove $P(\text{Node}(T_1, x, T_2))$, for all values $x$.

That’s enough! Why?

Contrast with structural induction for *lists*
## Tree Patterns

- **Empty**
- **Node(\(p_1, p, p_2\))**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matching Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>an empty tree</td>
</tr>
<tr>
<td>Node((_, _, _))</td>
<td>a non-empty tree</td>
</tr>
<tr>
<td>Node(Empty, _, Empty)</td>
<td>a tree with one node</td>
</tr>
<tr>
<td>Node(_, 42, _)</td>
<td>a tree with 42 at root</td>
</tr>
</tbody>
</table>
tree patterns match tree values

Empty matches $T$ iff $T$ is Empty

Node($p_1$, $p$, $p_2$) matches $T$ iff

$T$ is Node($T_1$, $v$, $T_2$) such that

$p_1$ matches $T_1$, $p$ matches $v$, $p_2$ matches $T_2$

and combines all the bindings

Node($A$, $x$, $B$) matches and binds $x$ to 3,

A to Node(Empty, 4, Empty)

B to Node(Empty, 2, Empty)
**using trees**

- Let’s introduce some useful functions for building and manipulating tree values

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Full : int * int -&gt; int tree</code></td>
<td>Full(x, n) = a complete binary tree of depth n</td>
</tr>
<tr>
<td><code>size : 'a tree -&gt; int</code></td>
<td>the number of nodes</td>
</tr>
<tr>
<td><code>depth : 'a tree -&gt; int</code></td>
<td>the longest path length</td>
</tr>
</tbody>
</table>
fun Leaf(x:int): int tree = Node(Empty, x, Empty)

fun Full(x:int, n:int): int tree = 
  if n=0 then Empty else 
  let 
    val T = Full(x, n-1) 
  in 
    Node(T, x, T) 
  end
fun Full(x:int, n:int): int tree =  
  if n=0 then Empty else  
    Node(Full(x, n-1), x, Full(x, n-1))

Same function,  
but WAY slower!

Show why,  
using work recurrences
fun size Empty = 0
| size (Node(T1, _, T2)) = size T1 + size T2 + 1

Uses tree patterns
Recursion is structural

Easy to prove by structural induction that for all trees T,
size(T) = a non-negative integer

the number of nodes
Size matters

- Size is 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.
  - 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 \( T \),
\[ \text{depth}(T) = \text{a non-negative integer} \]

the length of longest path from root to a leaf node
depth matters

- For all trees $T$, $\text{depth}(T) \geq 0$.
- Children have smaller depth
  \[
  \text{depth}(T_i) < \text{depth}(\text{Node}(T_1, x, T_2))
  \]
- Many recursive functions on trees make recursive calls on trees with smaller depth.
- Can use induction on depth to prove properties or analyze efficiency.
exercises

• Prove that for all $n \geq 0$

$$\text{size}(\text{Full}(42, n)) = 2^n - 1$$
$$\text{depth}(\text{Full}(42, n)) = n$$

Full(42, 3)

size is 7
depth is 3
useful facts

• The *children* of a full binary tree of depth $d$ are full binary trees of depth $d-1$.

• Each child of a full binary tree of size $2^{d-1}$ has size $2^{d-1}-1$ so contains about half the items.

• For all tree values $T$,

$$\text{size } T \leq 2^{\text{depth } T} - 1$$

(Exercise: prove this, using structural induction)
traversal

- We can generate a list from a tree by *traversing* it and collecting the data at nodes.
- There are many different traversal orders in wide use; each can be programmed in ML.

\[
\begin{align*}
\text{inorder} & \quad \text{preorder} & \quad \text{depth-first} \\
\text{postorder} & \quad \text{breadth-first}
\end{align*}
\]
traversal

in-order \[ [2, 42, 4, 3, 21, 7] \]

pre-order \[ [42, 2, 21, 3, 4, 7] \]

post-order \[ [2, 4, 3, 7, 21, 42] \]

breadth-first \[ [42, 2, 21, 3, 7, 4] \]
inorder traversal

inord : 'a tree -> 'a list
ENSURES inord T = the in-order traversal list for T

fun inord Empty = [ ]
| inord (Node(T1, x, T2)) = inord T1 @ (x :: inord T2)
inorder traversal

inord : ’a tree -> ’a list
ENSURES inord T = the in-order traversal list for T

fun inord Empty = [ ]
|  inord (Node(T1, x, T2)) = inord T1 @ (x :: inord T2)

  go left, then root, then right
inorder traversal

fun inord Empty = [ ]
| inord (Node(T1, x, T2)) = inord T1 @ (x :: inord T2)

inord : 'a tree -> 'a list
ENSURES inord T = the \textit{in-order} traversal list for T
inorder traversal

inord : ’a tree -> ’a list
ENSURES inord T = the in-order traversal list for T

fun inord Empty = [ ]
|  inord (Node(T1, x, T2)) = inord T1 @ (x :: inord T2)

[2, 42, 4, 3, 21, 7]
fun inord Empty = []
| inord (Node(T1, x, T2)) = inord T1 @ (x :: inord T2)

For all trees \( T \),

\[ \text{length (inord } T \text{)} = \text{size } T \]

For all lists \( L_1, L_2 \) of the same type

\[ \text{length (} L_1 \text{ @ } L_2 \text{)} = \text{length } L_1 + \text{length } L_2 \]
fun inord Empty = []
|   inord (Node(T1, x, T2)) = inord T1 @ (x :: inord T2)

Let $W_{\text{inord}}(n)$ be the work to evaluate $\text{inord}(T)$ when $T$ is a full binary tree of depth $n$

depth(T) = n, size(T) = 2^{n-1}$

- $W_{\text{inord}}(0) = 1$
- $W_{\text{inord}}(n) = 2W_{\text{inord}}(n-1) + O(2^n)$, for $n>0$
work

fun inord Empty = []
| inord (Node(T1, x, T2)) = inord T1 @ (x :: inord T2)

Let $W_{inord}(n)$ be the work to evaluate $inord(T)$ when $T$ is a full binary tree of depth $n$

$depth(T) = n$, $size(T) = 2^{n-1}$

- $W_{inord}(0) = 1$
- $W_{inord}(n) = 2W_{inord}(n-1) + O(2^n)$, for $n>0$
Let $W_{\text{inord}}(n)$ be the work to evaluate $\text{inord}(T)$ when $T$ is a full binary tree of depth $n$.

depth($T$) = $n$, size($T$) = $2^{n-1}$

- $W_{\text{inord}}(0) = 1$
- $W_{\text{inord}}(n) = 2W_{\text{inord}}(n-1) + O(2^n)$, for $n > 0$

\begin{itemize}
  \item if Node($T_1$, $x$, $T_2$) is full, depth $n$ then $T_1$ and $T_2$ are full, depth $n-1$
  \item work for $L_1 @ L_2$ is $O(\text{length } L_1)$
\end{itemize}
work

fun inord Empty = [ ]
  | inord (Node(T1, x, T2)) = inord T1 @ (x :: inord T2)

Let \( W_{inord}(n) \) be the work to evaluate \( \text{inord}(T) \) when \( T \) is a full binary tree of depth \( n \)

\[
\text{depth}(T) = n, \text{size}(T) = 2^{n-1}
\]

- \( W_{inord}(0) = 1 \)
- \( W_{inord}(n) = 2W_{inord}(n-1) + O(2^n), \text{ for } n>0 \)

if Node(T_1, x, T_2) is full, depth n then T_1 and T_2 are full, depth n-1

work for L_1@L_2 is O(length L_1)

length(inord T_1) = 2^{n-1}
Let $W_{\text{inord}}(n)$ be the work to evaluate $\text{inord}(T)$ when $T$ is a full binary tree of depth $n$.

depth($T$) = n, size($T$) = $2^n - 1$

- $W_{\text{inord}}(0) = 1$
- $W_{\text{inord}}(n) = 2W_{\text{inord}}(n-1) + O(2^n)$, for $n > 0$

fun inord Empty = []
| inord (Node(T1, x, T2)) = inord T1 @ (x :: inord T2)

work for $L_1 @ L_2$ is $O(\text{length } L_1)$
length(inord $T_1$) = $2^{n-1}$
work

fun inord Empty = []
  | inord (Node(T1, x, T2)) = inord T1 @ (x :: inord T2)

Let $W_{inord}(n)$ be the work to evaluate $\text{inord}(T)$ when $T$ is a full binary tree of depth $n$

depth(T) = n, size(T) = 2^{n-1}

• $W_{inord}(0) = 1$

• $W_{inord}(n) = 2W_{inord}(n-1) + O(2^n)$, for $n>0$

length(inord T1) = $2^{n-1}$
Let $W_{\text{inord}}(n)$ be the work to evaluate $\text{inord}(T)$ when $T$ is a full binary tree of depth $n$

depth$(T) = n$, size$(T) = 2^{n-1}$

• $W_{\text{inord}}(0) = 1$

• $W_{\text{inord}}(n) = 2W_{\text{inord}}(n-1) + O(2^n)$, for $n>0$

fun inord Empty = []
| inord (Node(T1, x, T2)) = inord T1 @ (x :: inord T2)
Let $W_{\text{inord}}(n)$ be the work to evaluate $\text{inord}(T)$ when $T$ is a full binary tree of depth $n$

- $\text{depth}(T) = n$, $\text{size}(T) = 2^{n-1}$
- $W_{\text{inord}}(0) = 1$
- $W_{\text{inord}}(n) = 2W_{\text{inord}}(n-1) + O(2^n)$, for $n > 0$

$W_{\text{inord}}(n)$ is $O(n2^n)$
inorder : 'a tree * 'a list -> 'a list

fun inorder (Empty, L) = L
| inorder (Node(T1, x, T2), L) =
    inorder (T1 , x :: inorder (T2, L))

Theorem

For all trees T and lists L,

\[ \text{inorder (T, L)} = (\text{inord T}) \ @ \ L \]

The work for \text{inorder(T, L)},
when T is a full tree of depth n, is \(O(2^n)\)
balanced trees

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

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

“chops list into two”
takedrop spec

For all $L : \text{int list}$ and $n : \text{int}$ with $0 \leq n \leq \text{length } L$, 
$takedrop \ (n, L) = \text{a pair of lists } (A, B) \text{ such that } L = A \@ B$ and $\text{length } A = n$

PROOF

By induction on length of $L$
- the recursive call is on a shorter list

or by structural induction on $L$
- the recursive call is on the tail
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

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

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

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

proof

By induction on list length

- in recursive calls, length A and length B are less than length L
correctness

By induction on length of L

• Base case (i) L is [ ]
  \text{list2tree }[ ] = \text{Empty}, \text{ a balanced tree, and inord Empty } = [ ] \text{. QED}

• Base case (ii) L is \([x]\)
  \text{list2tree }[x] = \text{Node(Empty, x, Empty)}
  \text{ and this is a balanced tree. Its inorder traversal list is } [x]. \text{ QED.}

• Inductive step: next slide…
correctness

• Inductive step: \( L \) has length \( n \geq 2 \).

Assume IH

For all shorter lists \( R \),

\( \text{list2tree } R = \) a balanced tree with inorder traversal list \( R \).

Show that

\( \text{list2tree } L = \) a balanced tree with inorder traversal list \( L \).

• Let \((A, C) = \text{takedrop } (n \text{ div } 2, L)\). Remember that \( n \geq 2 \).

\( A \) and \( C \) are shorter than \( L \), and \( A@C=L \).

\( C \) is non-empty, so let \( C = x::B \).

\( B \) is also shorter than \( L \), and \( A @ x::B = L \).

By IH, \( \text{list2tree } A = \) a balanced tree \( T_1 \) with \( \text{inord } T_1 = A \).

By IH, \( \text{list2tree } B = \) a balanced tree \( T_2 \) with \( \text{inord } T_2 = B \).

• \( \text{list2tree } L = \text{Node } (T_1, x, T_2) \)

This is a balanced tree, and its inorder traversal list is

\( (\text{inord } T_1) @ x :: (\text{inord } T_2) = A @ x::B = L \)
That proof was sketchy… but a good outline

• Add *justifications* to the proof steps above

• See how we used the function definitions for *inord*, *takedrop*, *list2tree*

• See where the proven specs for *takedrop* and *inord* were used (sometimes *implicitly*)

• Notice the subtle details that justify steps, e.g. when \( n \geq 2 \) we get \( 1 \leq n \div 2 < n \) (so \( A \) and \( C \) are shorter than \( L \), \( C \) is non-empty)
example

takedrop \((6 \text{ div } 2, [4,1,2,42,3,5])\) = ([4,1,2], [42,3,5])

list2tree \([4,1,2]\) = \[ \begin{array}{c} 4 \\ \downarrow \\ \downarrow \\ 2 \\ 1 \end{array} \] balanced

list2tree \([3,5]\) = \[ \begin{array}{c} 3 \\ \downarrow \\ 5 \end{array} \] balanced

list2tree \([4,1,2,42,3,5]\) = \[ \begin{array}{c} 42 \\ \Downarrow \\ 4 \\ 2 \\ \Downarrow \\ 1 \\ \downarrow \\ 3 \\ \downarrow \\ 5 \end{array} \] balanced

inorder traversals are as intended
exercise
(in class, if time)

• Write an ML function
  \texttt{layers : } 'a \texttt{ tree} \rightarrow ('a \texttt{ list}) \texttt{ list}
  \texttt{such that}
  \texttt{layers} \texttt{T} = \texttt{a list of the cross-sections of } \texttt{T}
exercise
(in class, if time)

- Write an ML function
  \[ \text{layers} : \text{'a tree -> ('a list) list} \]
such that
  \[ \text{layers } T = \text{ a list of the cross-sections of } T \]
exercise

• Using layers : 'a tree -> 'a list list, define a breadth-first traversal function
  bftrav : 'a tree -> 'a list
exercise

- Using `layers : 'a tree -> 'a list list`, define a breadth-first traversal function `bftrav : 'a tree -> 'a list`
coming soon

• How to sort a tree of integers
  • What’s a sorted tree?
  • Can we exploit parallelism here?
How to Recognise Different Trees From Quite a Long Way Away

No. 1 The Larch