15-150 Fall 2013

Lecture 8

Stephen Brookes

sorted trees

- Empty is sorted
- Node(t_1, x, t_2) is sorted iff

every integer in t_1 is $\leq x$

and

every integer in t_2 is $\geq x$

and

t_1 and t_2 are sorted

t is sorted
iff
trav(t) is a sorted list

Insertion

Ins : int * tree -> 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);
```

For all sorted trees t,
Ins(x,t) = a sorted tree
consisting of x and the items of t

SplitAt

SplitAt : int * tree -> tree * tree

(* REQUIRES t is a sorted tree *)
(* ENSURES SplitAt(y, t) = a pair (t_1, t_2)
such that
every item in t_1 is $\leq$ y,
every item in t_2 is $\geq$ y,
and t_1,t_2 consist of the items in t *)

Any ideas???

Plan

Define SplitAt(t) using ***recursion***

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

SplitAt

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

Correctness

Let P(t) be

For all y:int, SplitAt(y, t) = a pair (t1, t2) such that

every item in t1 is ≤ y & every item in t2 is ≥ y

& t1, t2 consist of the items in t

Theorem

For all sorted trees t, P(t) holds

Proof: by structural induction

- Base case:
  Empty is sorted. Prove P(Empty).
- Inductive step:
  Let t be a sorted tree Node(t1, y , t2).
  Then t1 and t2 are also sorted.
  Use P(t1) and P(t2) to prove P(t)

depth lemma

For all trees t and integers y,

SplitAt(y, t) = a pair (t_1, t_2) such that

$\text{depth}(t_1) \leq \text{depth } t$ & $\text{depth}(t_2) \leq \text{depth } t$

Proof: by structural induction

(exercise!)

Merge

Merge : tree * tree -> tree

(* REQUIRES t_1 and t_2 are sorted trees *)
(* ENSURES Merge(t_1, t_2) = a sorted tree t *)
(* 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
```

Correctness

```
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
```

To prove:

For all sorted trees t_1 and t_2

Merge(t_1, t_2) = a sorted tree

consisting of the items of t_1 and t_2

Method? Induction on the depth of t_1

Mergesort

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))
```

For all trees t,
Msort(t) = a sorted permutation of t

Correct?

- How can we *prove* that Msort satisfies this specification?

  The definition of Msort is *structural*

  So use structural induction

  Use the proven specs for Ins and Merge

Mergesort

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))
```

For all trees t,
Msort(t) = a sorted permutation of t

Parallelism

- The recursive calls in

  Merge(Msort t_1, Msort t_2)

  can be evaluated in parallel

- Sequential evaluation would take the *sum* of the runtimes of the two calls

- Parallel evaluation would take the *max*

- The ***span*** is the runtime, assuming an unlimited number of parallel processors

Span, span, span,
eggs, bacon and span

- Can derive a recurrence relation for *span*
- Based on function definitions
- Dependent code: use *sum*
- Independent code: use *max*

For *sequential* code, span = work

Span of Ins

```
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 balanced tree of depth d>0,

$$S_{Ins}(d) = c + S_{Ins}(d\text{-}1)$$

$S_{Ins}(d)$ is $O(d)$

Span of SplitAt

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

(no parallelism!)

For a balanced tree of depth d>0,

$$S_{SplitAt}(d) = k + S_{SplitAt}(d-1)$$

$S_{SplitAt}(d)$ is $O(d)$

Span of Merge

independent

```
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;
```

For balanced trees of same depth d>0,
assuming that the trees got by splitting
have the same depth (d-1) and are balanced

$$S_{Merge}(d) = S_{SplitAt}(d) + S_{Merge}(d-1)$$
$$+ \max(S_{Merge}(d-1), S_{Merge}(d-1))$$

$S_{Merge}(d)$ is $O(d^2)$

Span of Msort

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

independent

For a balanced tree of size n, depth log n

$$S_{Msort}(n) \leq \max(S_{Msort}(n \text{ div } 2), S_{Msort}(n \text{ div } 2)) + S_{Merge}(\log n) + S_{Ins}(2 \log n)$$

Span of Msort

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

independent

For a balanced tree of size n, depth log n

Span of Msort

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

independent

For a balanced tree of size n, depth log n

$$S_{Msort}(n) \leq S_{Msort}(n \text{ div } 2) + S_{Merge}(\log n) + S_{Ins}(2 \log n)$$

Span of Msort

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

independent

For a balanced tree of size n, depth log n

Span of Msort

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

independent

For a balanced tree of size n, depth log n

$$S_{Msort}(n) \leq S_{Msort}(n \text{ div } 2) + c(\log n)^2$$

Span of Msort

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

independent

For a balanced tree of size n, depth log n

$S_{Msort}(n) \leq S_{Msort}(n \text{ div } 2) + c(\log n)^2$

$S_{Msort}(n)$ is $O((\log n)^3)$

Really?

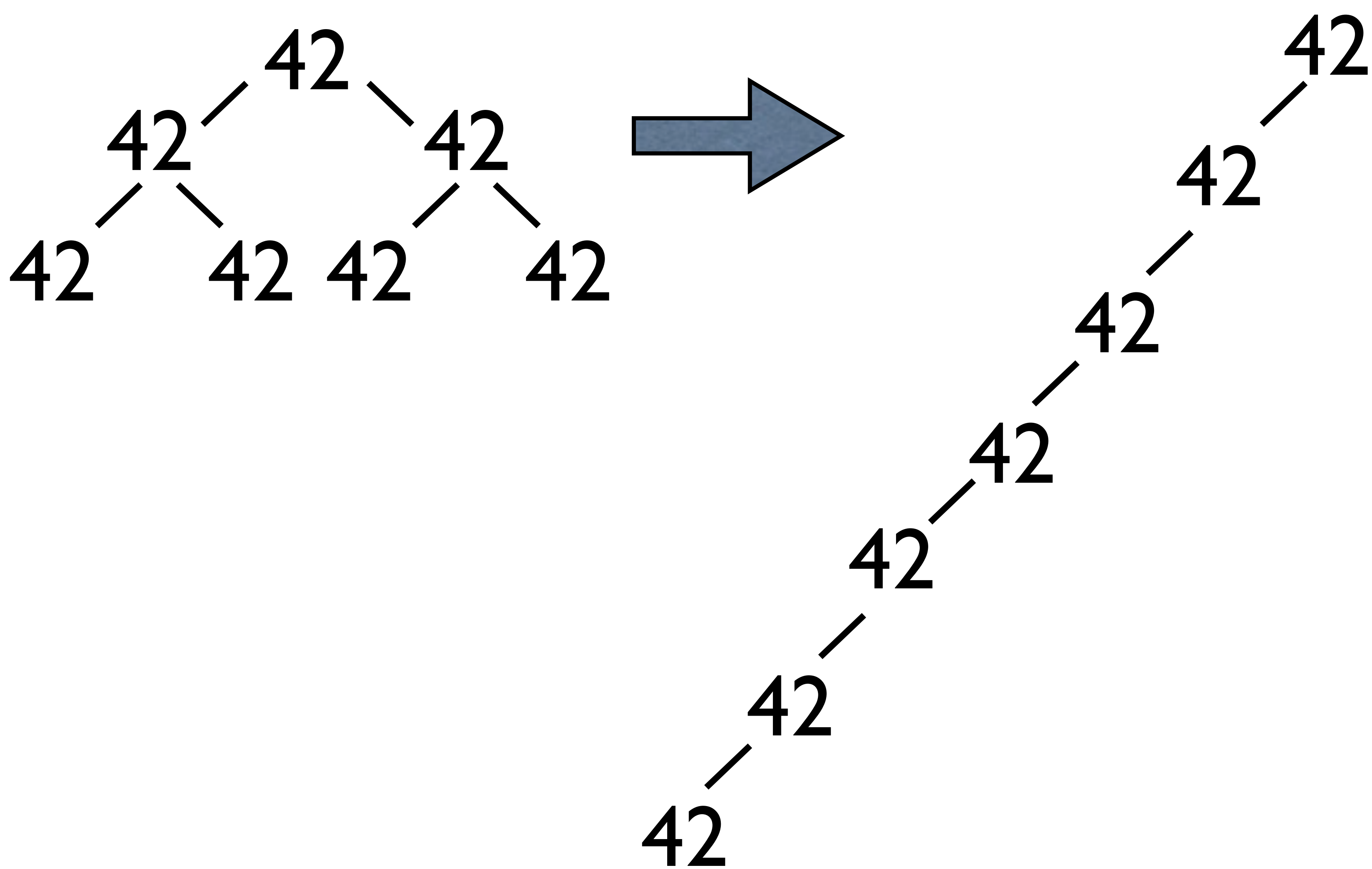
- Are the balance assumptions realistic? No!
- But we could design a *rebalancing* function...

```
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*...

Why bother?

- Msort can produce badly balanced trees



Back to lists

- The mergesort function on integer lists can also exploit parallel evaluation
- When length(L)>1 and (A,B) = split(L),

  msort L = merge(msort A, msort B)

  independent

What's the span?