Sorting trees — work and span revisited

15-150

Lecture 8: September 19, 2024

Stephanie Balzer
Carnegie Mellon University

Midterm I: room update

When and where:

- Thursday, **September 26**, **11:00am—12:20pm**.
- **MM 103** (Sections A—D), **PH 100** (Sections E—L).

Scope:

- Lectures: 1—8.
- Labs: 1—4 and midterm review section of Lab 5.
- Assignments: Basics, Induction, and Datatypes.

What you may have on your desk:

- Writing utensils, we provide paper, something to drink/eat, tissues.
- 8.5" x 11" cheatsheet (back and front), handwritten or typeset.
- No cell phones, laptops, or any other smart devices.

Midterm I: room update

When and where:

- Thursday, **September 26**, **11:00am—12:20pm**.
- **MM 103** (Sections A—D), **PH 100** (Sections E—L).

Scope:

- Lectures: 1—8.
- Labs: 1—4 and midterm review section of Lab 5.
- Assignments: Basics, Induction, and Datatypes.

What you may have on your desk:

- Writing utensils, we provide paper, something to drink/eat, tissues.
- 8.5” x 11” cheatsheet (back and front), handwritten or typeset.
- No cell phones, laptops, or any other smart devices.

Mergesort: divide and conquer

Mergesort: divide and conquer

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

Mergesort: divide and conquer

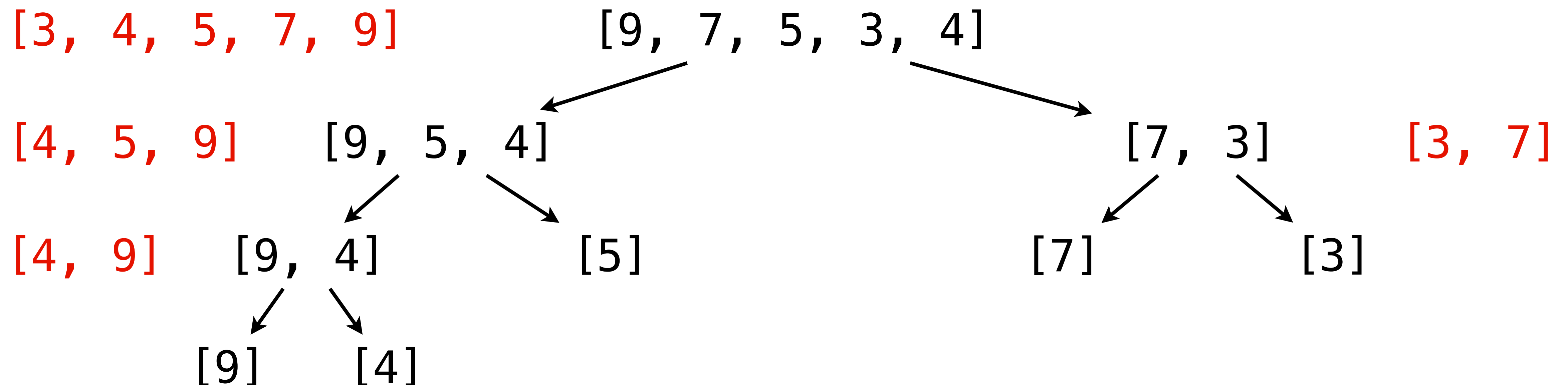
```
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let val (A, B) = split L
      in
         merge(msort A, msort B)
      end
```

recursively divide in equal sub-lists

Mergesort: divide and conquer

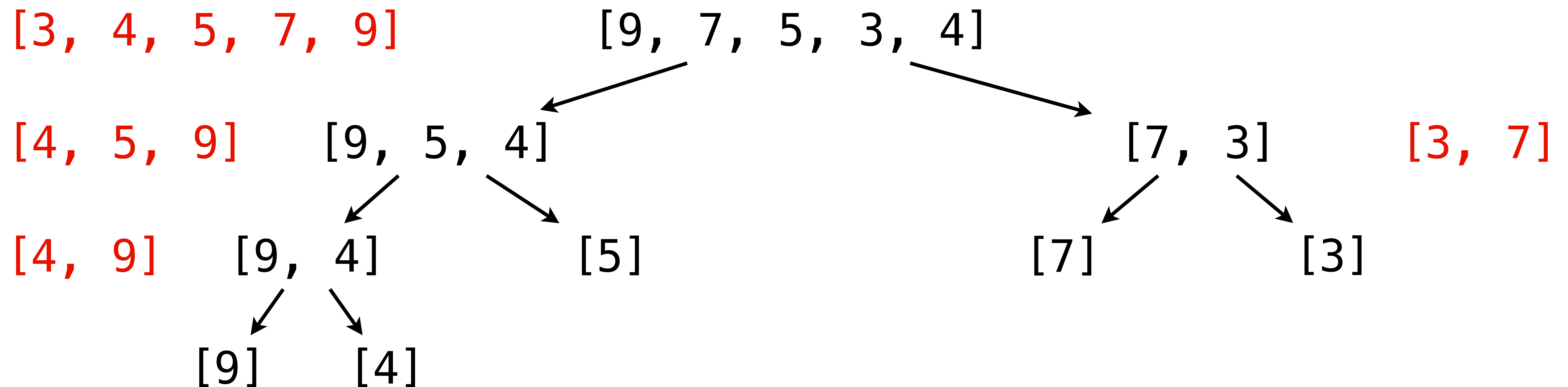
```
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let val (A, B) = split L
      in
         merge(msort A, msort B)
      end
```

recursively divide in equal sub-lists



Mergesort: divide and conquer

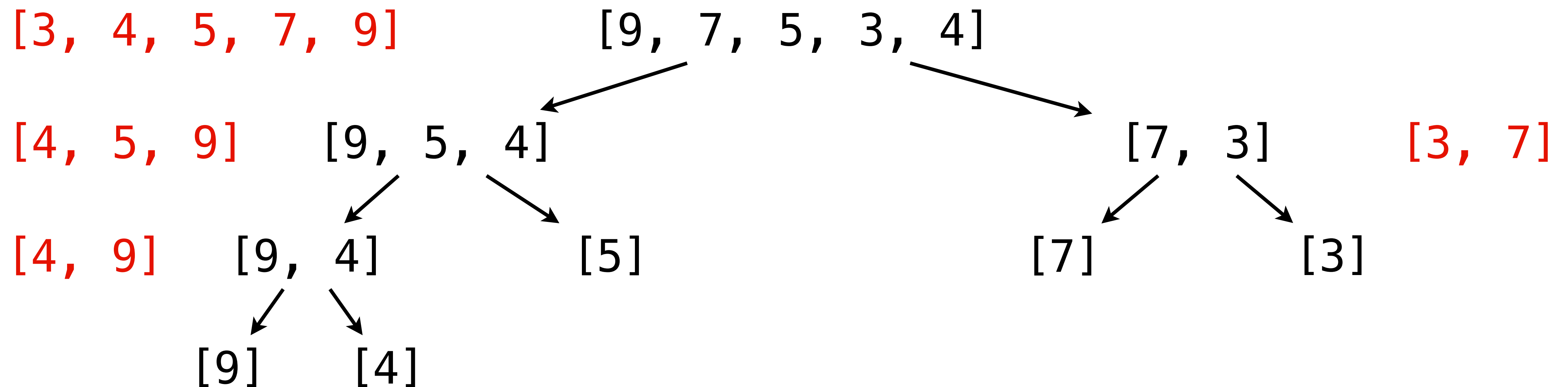
```
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let val (A, B) = split L
      in
         merge(msort A, msort B)
      end
```



Mergesort: divide and conquer

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

parallelize recursive calls on sub-lists

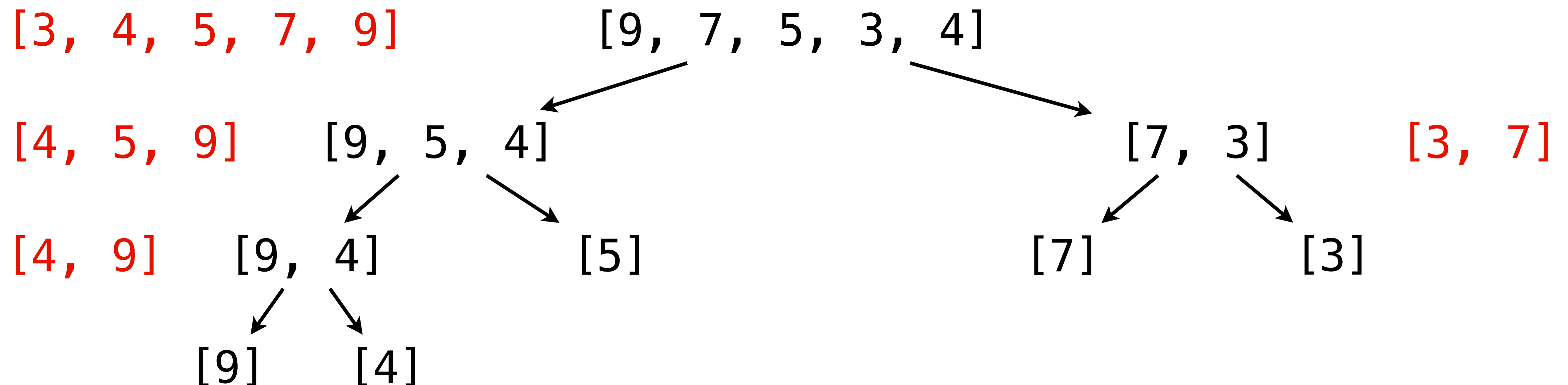


Mergesort: divide and conquer

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

parallelize recursive calls on sub-lists

Let's determine the span of mergesort!

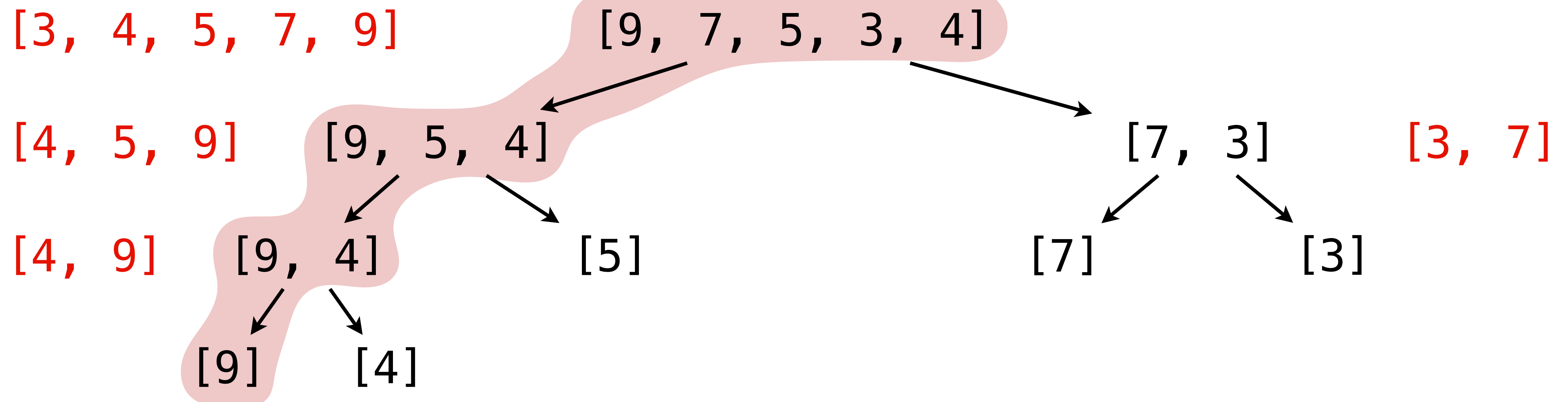


Mergesort: divide and conquer

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

parallelize recursive calls on sub-lists

Let's determine the span of mergesort!

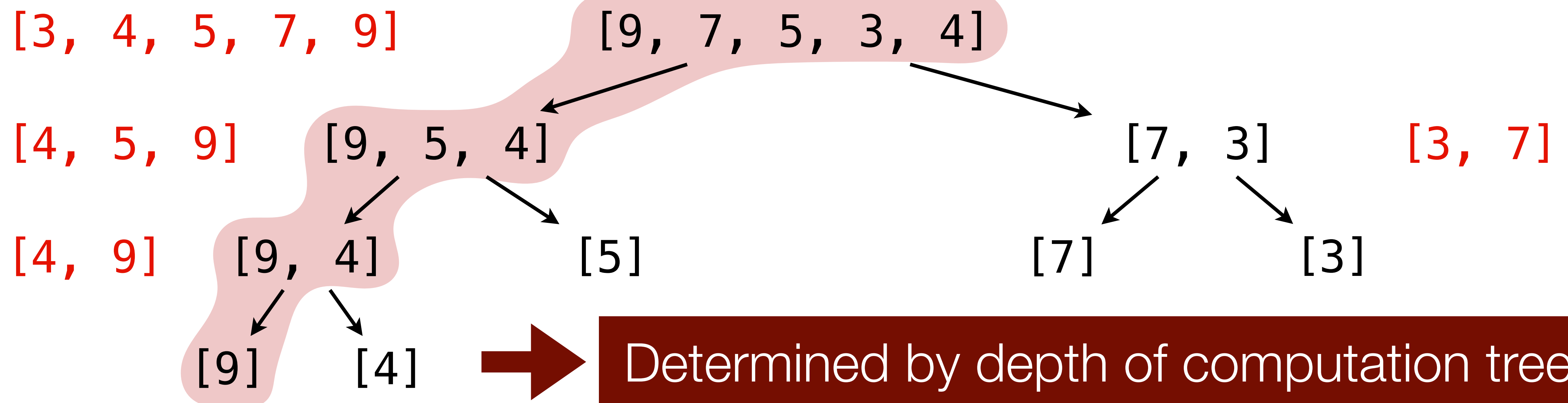


Mergesort: divide and conquer

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

parallelize recursive calls on sub-lists

Let's determine the span of mergesort!



Span for mergesort for lists

Span for mergesort for lists

Recall work: $W_{msort}(n)$ with n the list length.

Span for mergesort for lists

Recall work: $W_{\texttt{msort}}(\texttt{n})$ with $\texttt{n}$ the list length.

Equations:

$$W_{\texttt{msort}}(0) = c_0$$
$$W_{\texttt{msort}}(1) = c_1$$
$$W_{\texttt{msort}}(n) = c_2 + W_{\texttt{split}}(n) + W_{\texttt{msort}}(n_a) + W_{\texttt{msort}}(n_b)$$
$$+ W_{\texttt{merge}}(n_a, n_b), \text{ for } n = n_a + n_b \text{ and } n \geq 2$$

Span for mergesort for lists

Recall work: $W_{msort}(n)$ with n the list length.

Equations:

$= \lfloor n/2 \rfloor$ $= \lceil n/2 \rceil$

$$
\begin{aligned}
W_{msort}(0) &= c_0 \\
W_{msort}(1) &= c_1 \\
W_{msort}(n) &= c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b) \\
&\quad + W_{merge}(n_a, n_b), \text{ for } n = n_a + n_b \text{ and } n \geq 2
\end{aligned}
$$

Span for mergesort for lists

Recall work: $\texttt{W}_{\texttt{msort}}\texttt{(n)}$ with $\texttt{n}$ the list length.

Equations:

$$\lfloor n/2 \rfloor \qquad = \lceil n/2 \rceil$$

$\texttt{W}_{\texttt{msort}}\texttt{(0)} = c_0$

$\texttt{W}_{\texttt{msort}}\texttt{(1)} = c_1$

$\texttt{W}_{\texttt{msort}}\texttt{(n)} = c_2 + \texttt{W}_{\texttt{split}}\texttt{(n)} + \texttt{W}_{\texttt{msort}}\texttt{(n}_\texttt{a}\texttt{)} + \texttt{W}_{\texttt{msort}}\texttt{(n}_\texttt{b}\texttt{)}$

$+ \texttt{W}_{\texttt{merge}}\texttt{(n}_\texttt{a}\texttt{,n}_\texttt{b}\texttt{)}$, for $\texttt{n} = \texttt{n}_\texttt{a} + \texttt{n}_\texttt{b}$ and $\texttt{n} \geq 2$

$c\ \texttt{n}$

Span for mergesort for lists

Recall work: $W_{msort}(n)$ with n the list length.

Equations:

$= \lfloor n/2 \rfloor$ $= \lceil n/2 \rceil$

$W_{msort}(0) = c_0$
$W_{msort}(1) = c_1$
$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$
$+ W_{merge}(n_a, n_b)$, for $n = n_a + n_b$ and $n \geq 2$

$c\ n + c'\ n = (c + c')\ n = c_3\ n$

Span for mergesort for lists

Recall work: $W_{\texttt{msort}}(n)$ with n the list length.

Equations:

$= \lfloor n/2 \rfloor$ $= \lceil n/2 \rceil$

$$W_{\texttt{msort}}(0) = c_0$$
$$W_{\texttt{msort}}(1) = c_1$$
$$W_{\texttt{msort}}(n) = c_2 + W_{\texttt{split}}(n) + W_{\texttt{msort}}(n_a) + W_{\texttt{msort}}(n_b)$$
$$+ W_{\texttt{merge}}(n_a, n_b), \text{ for } n = n_a + n_b \text{ and } n \geq 2$$

$$c\, n + c'\, n = (c + c')\, n = c_3\, n$$

$$W_{\texttt{msort}}(n) \leq c_2 + c_3\, n + 2\, W_{\texttt{msort}}(n/2)$$
$$W_{\texttt{msort}}(n) \leq c_4\, n + 2\, W_{\texttt{msort}}(n/2)$$

Span for mergesort for lists

Recall work: $W_{msort}(n)$ with n the list length.

Equations:

$$W_{msort}(0) = c_0$$
$$W_{msort}(1) = c_1$$
$$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$$
$$+ W_{merge}(n_a, n_b), \text{ for } n = n_a + n_b \text{ and } n \geq 2$$

$$W_{msort}(n) \leq c_2 + c_3\, n + 2\, W_{msort}(n/2)$$
$$W_{msort}(n) \leq c_4\, n + 2\, W_{msort}(n/2)$$

Span for mergesort for lists

Recall work: $W_{msort}(n)$ with n the list length.

Equations:

$W_{msort}(0) = c_0$
$W_{msort}(1) = c_1$
$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$
$\quad + W_{merge}(n_a, n_b)$, for $n = n_a + n_b$ and $n \geq 2$

$W_{msort}(n) \leq c_2 + c_3\, n + 2\, W_{msort}(n/2)$
$W_{msort}(n) \leq c_4\, n + 2\, W_{msort}(n/2)$

Span for mergesort for lists

Recall work: $W_{msort}(n)$ with n the list length.

Equations:

$W_{msort}(0) = c_0$
$W_{msort}(1) = c_1$
$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$
$+ W_{merge}(n_a, n_b)$, for $n = n_a + n_b$ and $n \geq 2$

parallelize recursive calls on sub-lists

$W_{msort}(n) \leq c_2 + c_3\, n + 2\, W_{msort}(n/2)$
$W_{msort}(n) \leq c_4\, n + 2\, W_{msort}(n/2)$

Span for mergesort for lists

Recall work: $W_{msort}(n)$ with n the list length.

Equations:

$W_{msort}(0) = c_0$

$W_{msort}(1) = c_1$

$W_{msort}(n) = c_2 + W_{split}(n) + W_{msort}(n_a) + W_{msort}(n_b)$
$+ W_{merge}(n_a, n_b)$, for $n = n_a + n_b$ and $n \geq 2$

max!

parallelize recursive calls on sub-lists

$W_{msort}(n) \leq c_2 + c_3\, n + 2\, W_{msort}(n/2)$

$W_{msort}(n) \leq c_4\, n + 2\, W_{msort}(n/2)$

Span for mergesort for lists

Span: $S_{msort}(n)$ with n the list length.

Equations:

$S_{msort}(0) = c_0$
$S_{msort}(1) = c_1$
$S_{msort}(n) = c_2 + S_{split}(n) + \max(S_{msort}(n_a) + S_{msort}(n_b))$
$\quad + S_{merge}(n_a, n_b)$, for $n = n_a + n_b$ and $n \geq 2$

max!

parallelize recursive calls on sub-lists

$W_{msort}(n) \leq c_2 + c_3\, n + 2\, W_{msort}(n/2)$
$W_{msort}(n) \leq c_4\, n + 2\, W_{msort}(n/2)$

Span for mergesort for lists

Span: $S_{msort}(n)$ with n the list length.

Equations:

$S_{msort}(0) = c_0$

$S_{msort}(1) = c_1$

$S_{msort}(n) = c_2 + S_{split}(n) + \max(S_{msort}(n_a) + S_{msort}(n_b))$
$\quad + S_{merge}(n_a, n_b)$, for $n = n_a + n_b$ and $n \geq 2$

max!

parallelize recursive calls on sub-lists

$W_{msort}(n) \leq c_2 + c_3\, n + 2\, W_{msort}(n/2)$

$W_{msort}(n) \leq c_4\, n + 2\, W_{msort}(n/2)$

Span for mergesort for lists

Span: $S_{msort}(n)$ with n the list length.

Equations:

$$S_{msort}(0) = c_0$$
$$S_{msort}(1) = c_1$$
$$S_{msort}(n) = c_2 + S_{split}(n) + \max(S_{msort}(n_a) + S_{msort}(n_b))$$
$$+ S_{merge}(n_a, n_b), \text{ for } n = n_a + n_b \text{ and } n \geq 2$$

max!

parallelize recursive calls on sub-lists

$$W_{msort}(n) \leq c_2 + c_3\, n + 2\, W_{msort}(n/2)$$
$$W_{msort}(n) \leq c_4\, n + 2\, W_{msort}(n/2)$$

Span for mergesort for lists

Span: $S_{msort}(n)$ with n the list length.

Equations:

$S_{msort}(0) = c_0$
$S_{msort}(1) = c_1$
$S_{msort}(n) = c_2 + S_{split}(n) + \max(S_{msort}(n_a) + S_{msort}(n_b))$
$\quad + S_{merge}(n_a,n_b)$, for $n = n_a + n_b$ and $n \geq 2$

max!

parallelize recursive calls on sub-lists

$S_{msort}(n) \leq c_2 + c_3\, n + S_{msort}(n/2)$
$S_{msort}(n) \leq c_4\, n + S_{msort}(n/2)$

Span for mergesort for lists

Span: $S_{msort}(n)$ with n the list length.

Equations:

$S_{msort}(0) = c_0$
$S_{msort}(1) = c_1$
$S_{msort}(n) = c_2 + S_{split}(n) + \max(S_{msort}(n_a) + S_{msort}(n_b))$
$\quad + S_{merge}(n_a, n_b)$, for $n = n_a + n_b$ and $n \geq 2$

$S_{msort}(n) \leq c_2 + c_3\, n + S_{msort}(n/2)$
$S_{msort}(n) \leq c_4\, n + S_{msort}(n/2)$

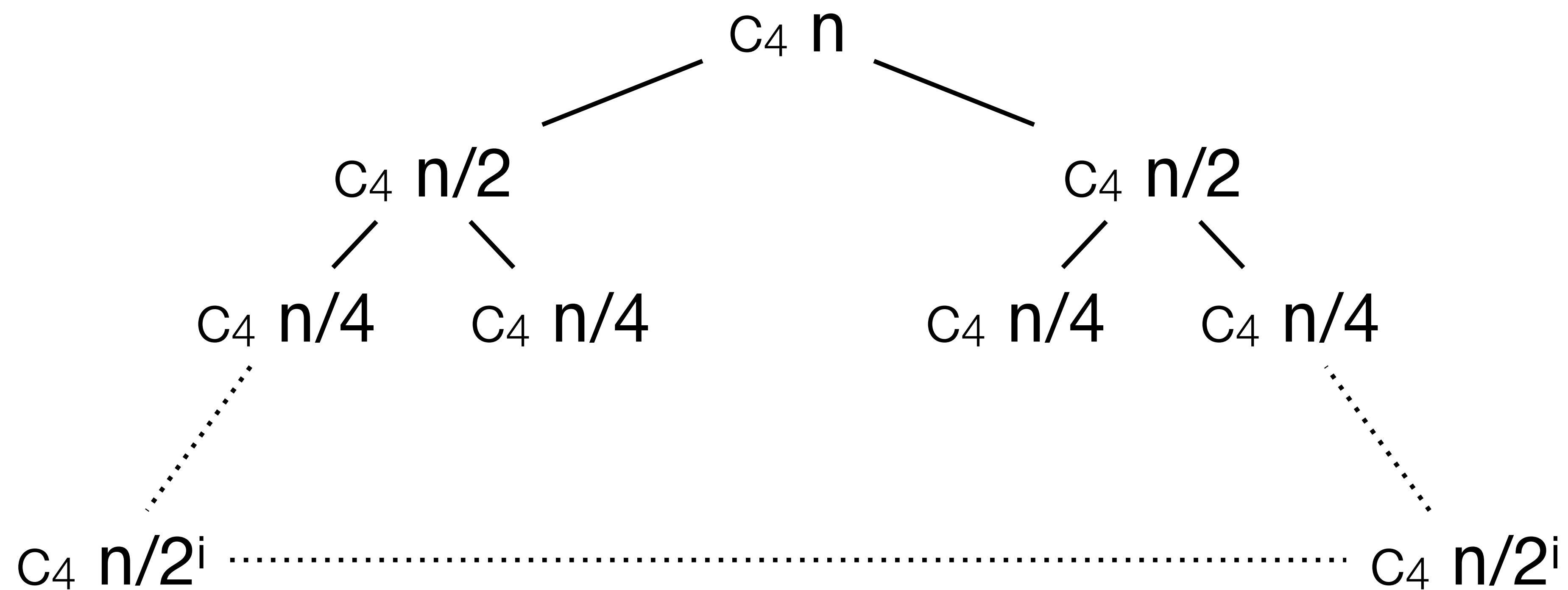
→ Let's look at the tree method to find a closed form.

Span for mergesort for lists

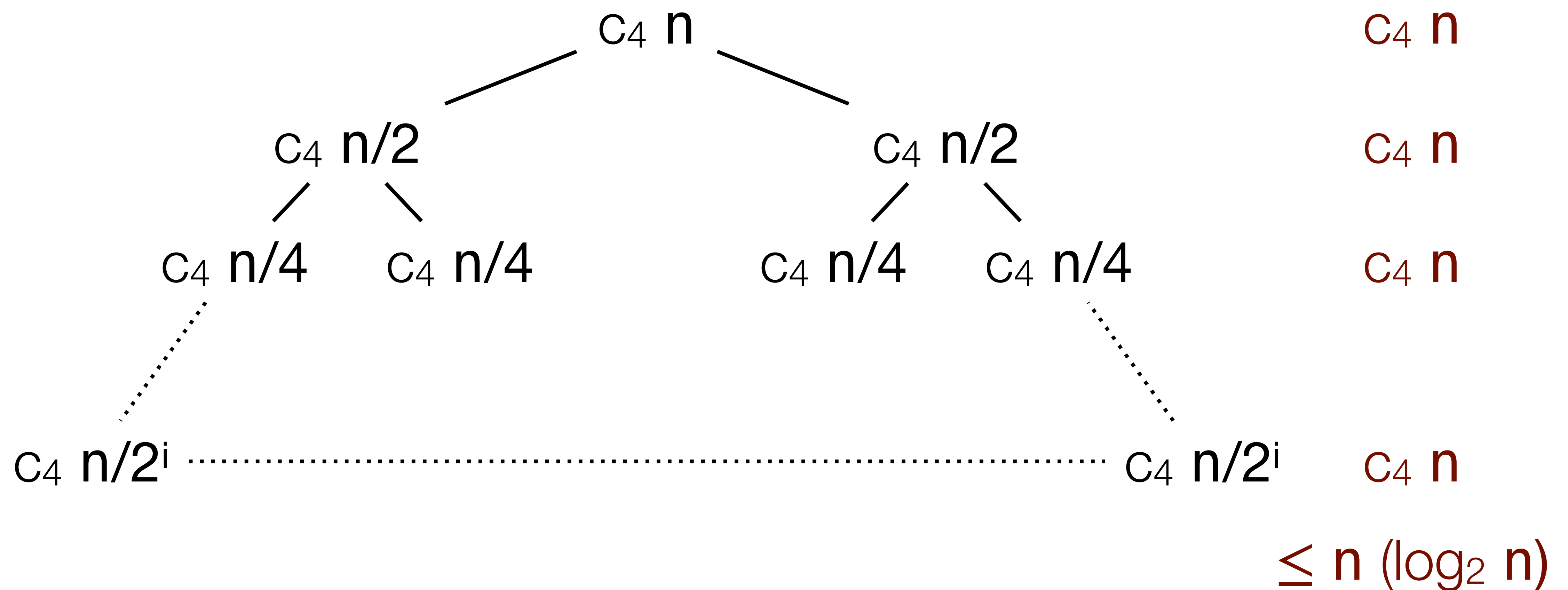
$$W_{msort}(n) \leq c_4\, n + W_{msort}(n/2)$$

Span for mergesort for lists

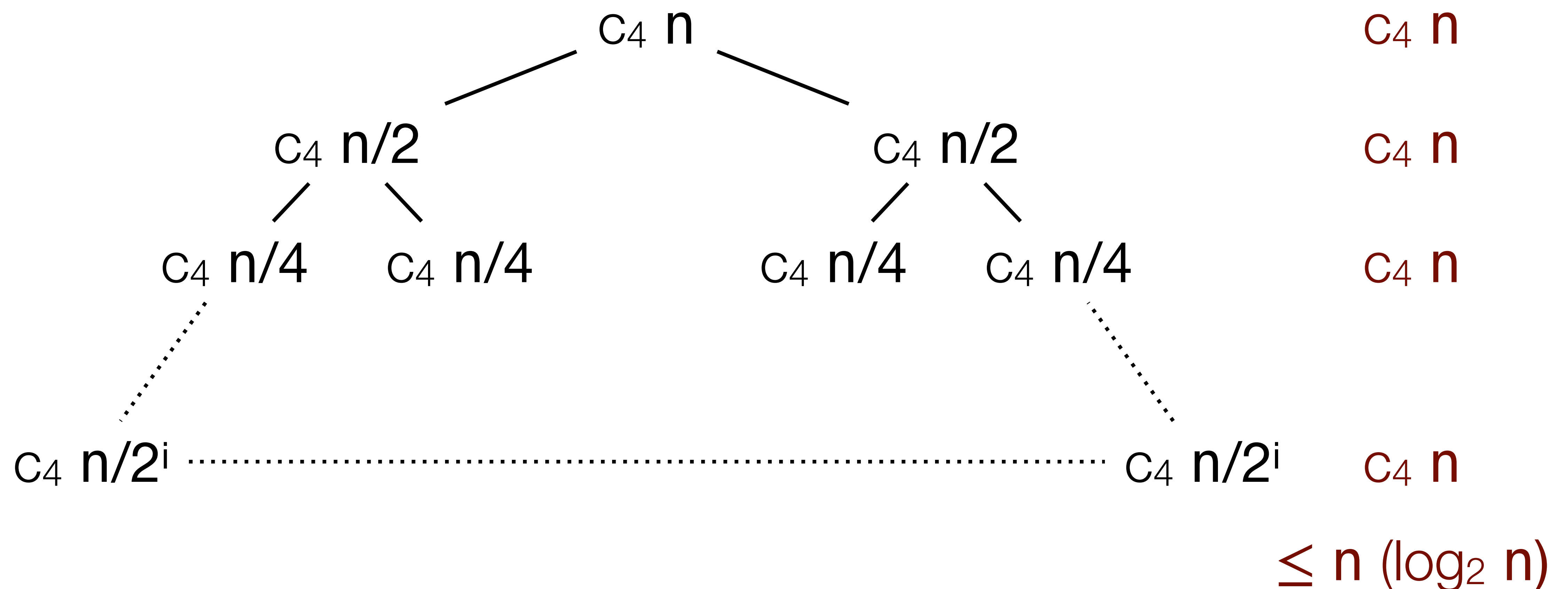
$$W_{\texttt{msort}}(n) \leq c_4\, n + W_{\texttt{msort}}(n/2)$$



Span for mergesort for lists



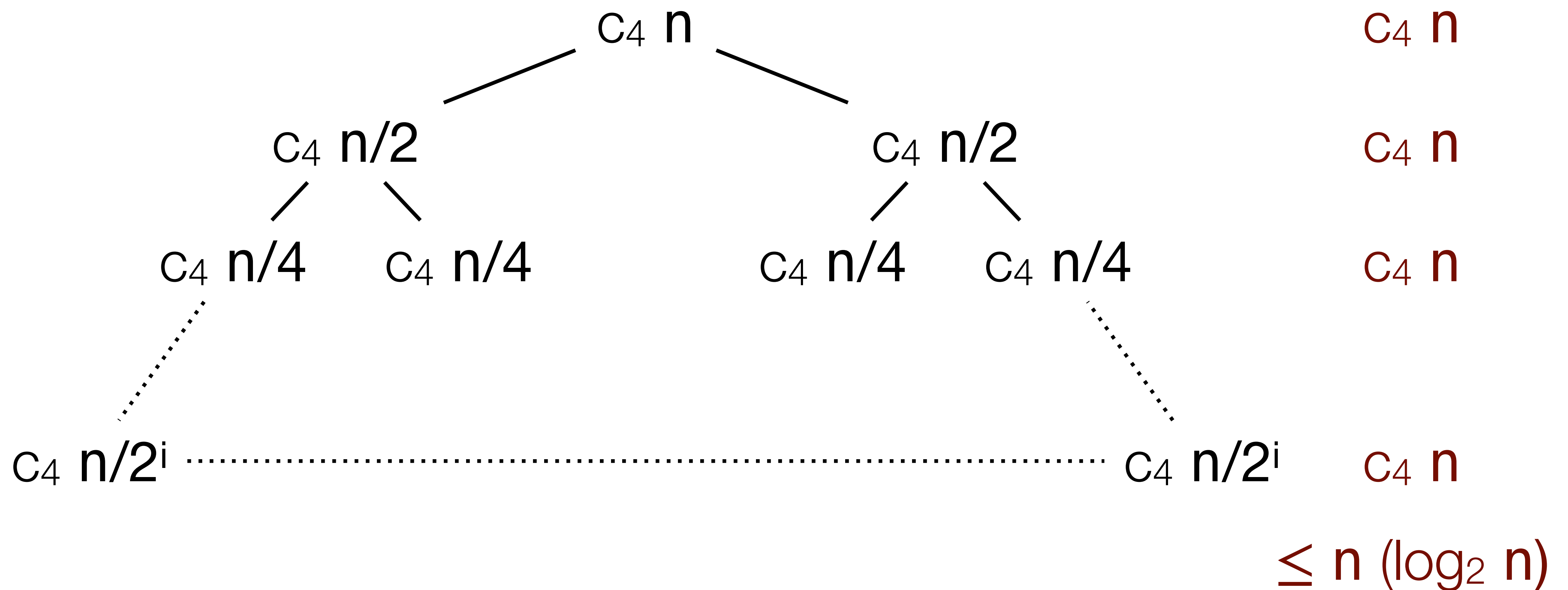
Span for mergesort for lists



Consequently: $W_{msort}(n)$ is $O(n \log n)$

Span for mergesort for lists

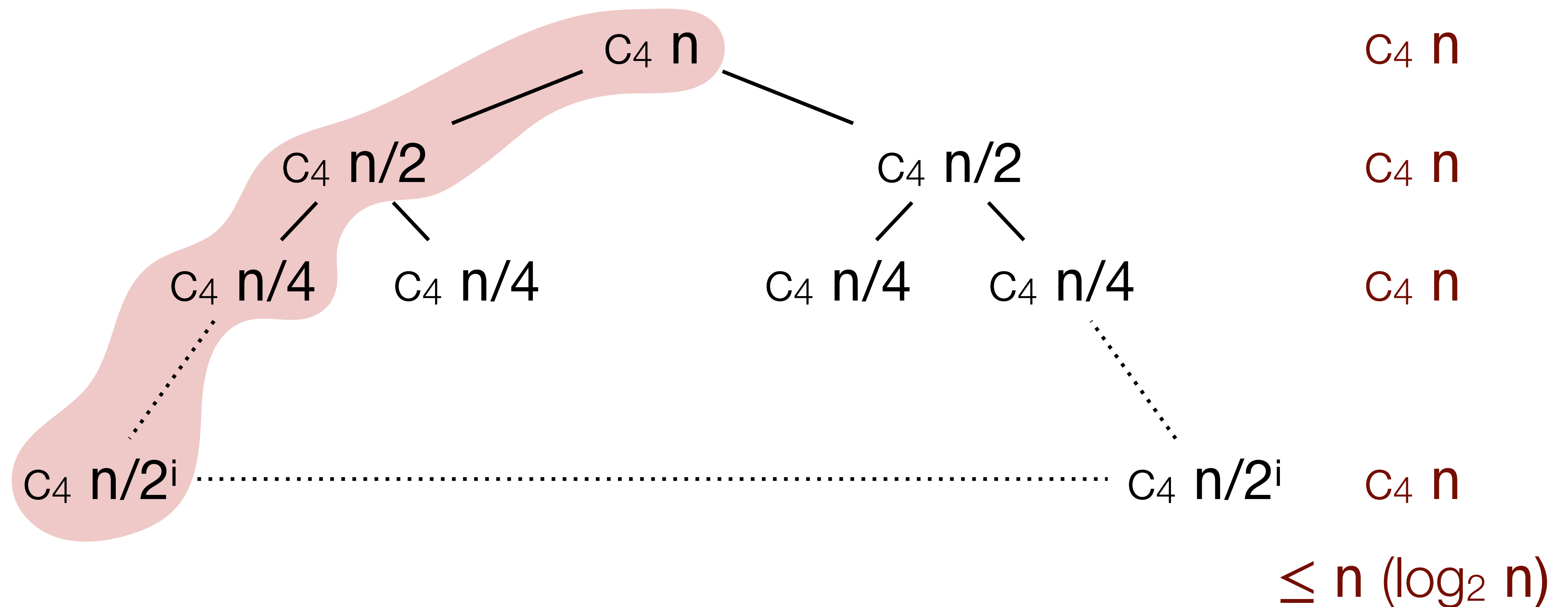
span: $W_{msort}(n) \leq c_4\, n + W_{msort}(n/2)$ work:



Consequently: $W_{msort}(n)$ is $O(n \log n)$

Span for mergesort for lists

span: $W_{\texttt{msort}}(n) \leq c_4\, n + W_{\texttt{msort}}(n/2)$ work:

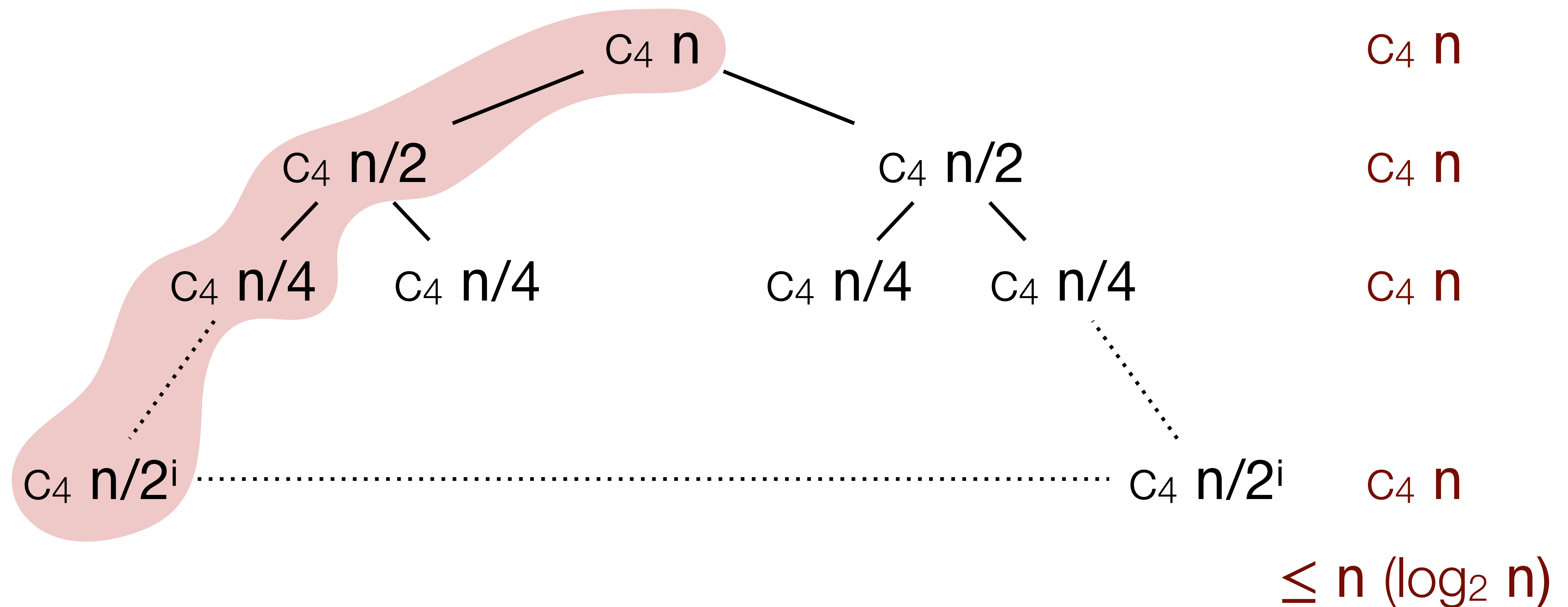


Consequently: $W_{\texttt{msort}}(n)$ is $O(n \log n)$

Span for mergesort for lists

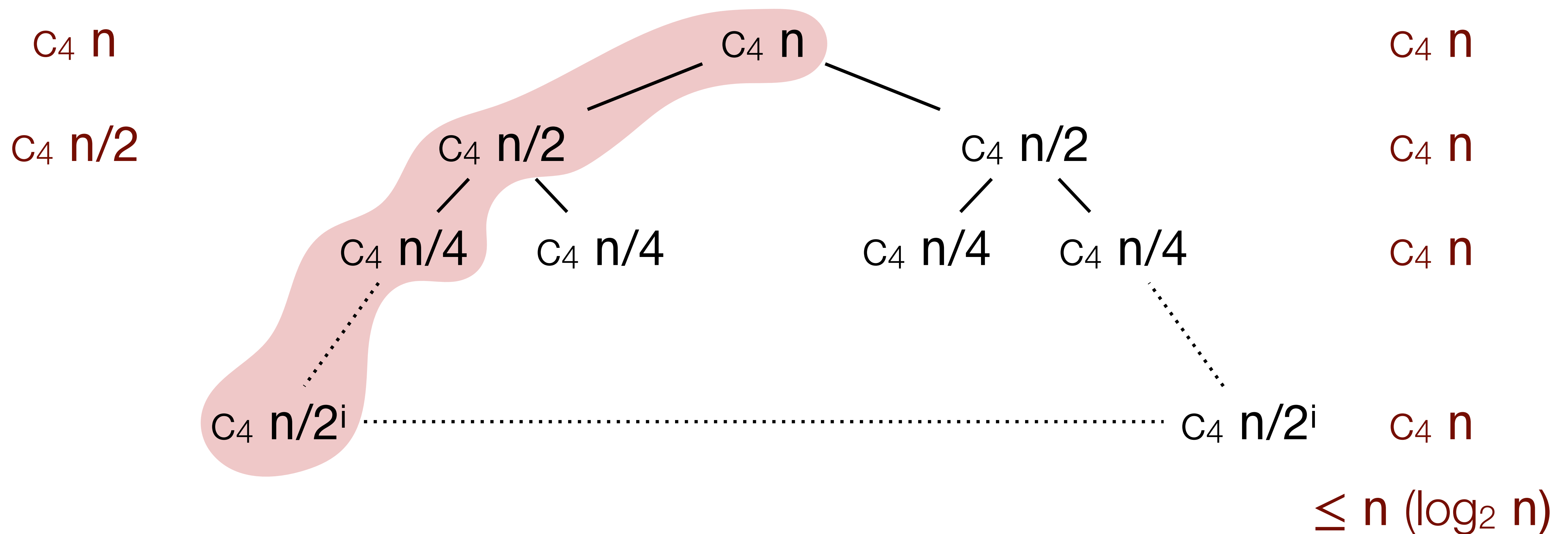
span: $W_{msort}(n) \leq c_4\, n + W_{msort}(n/2)$ work:

c_4 n



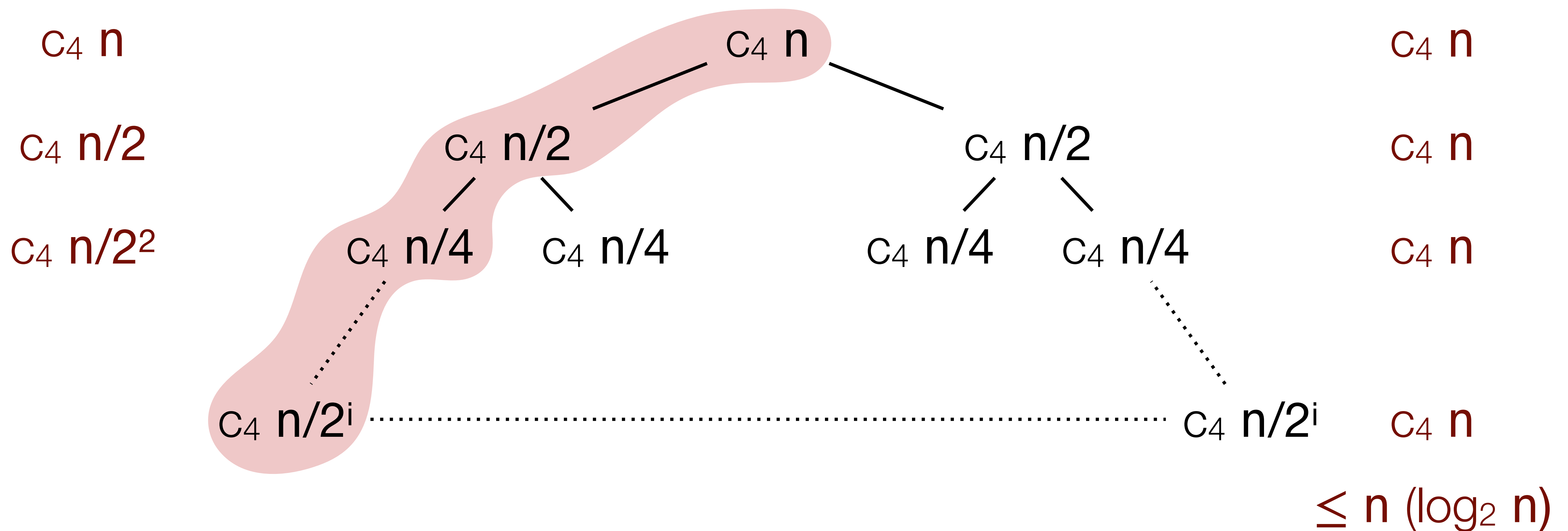
Consequently: $W_{msort}(n)$ is $O(n \log n)$

Span for mergesort for lists



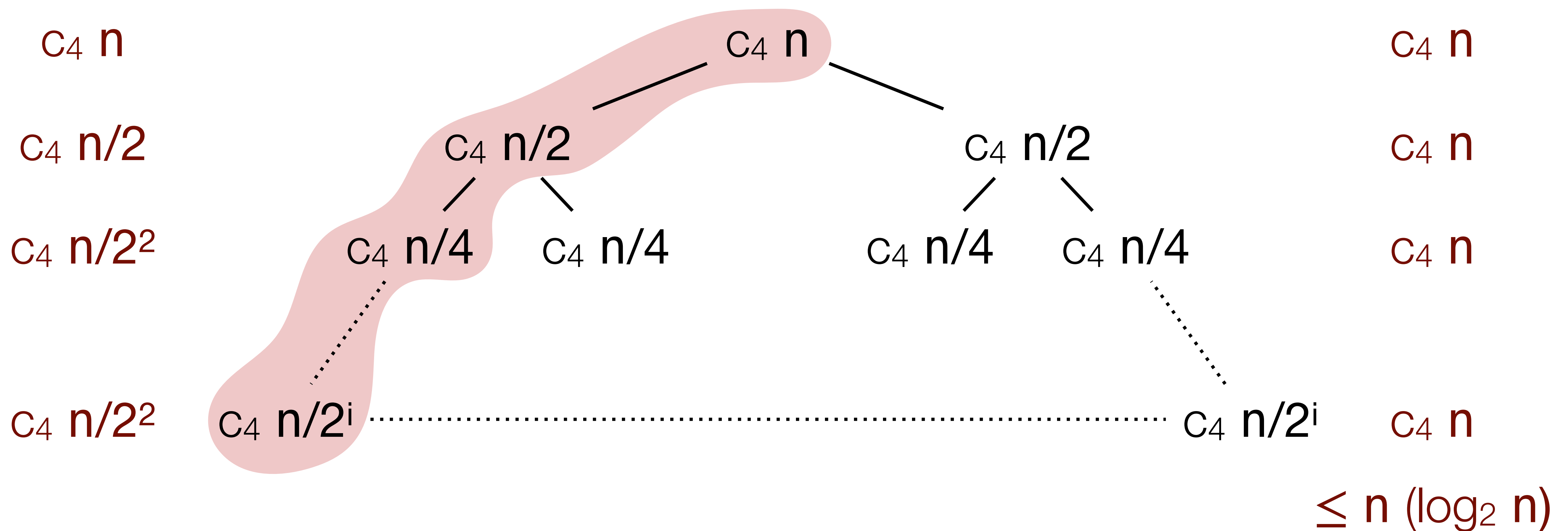
Consequently: $W_{msort}(n)$ is $O(n \log n)$

Span for mergesort for lists



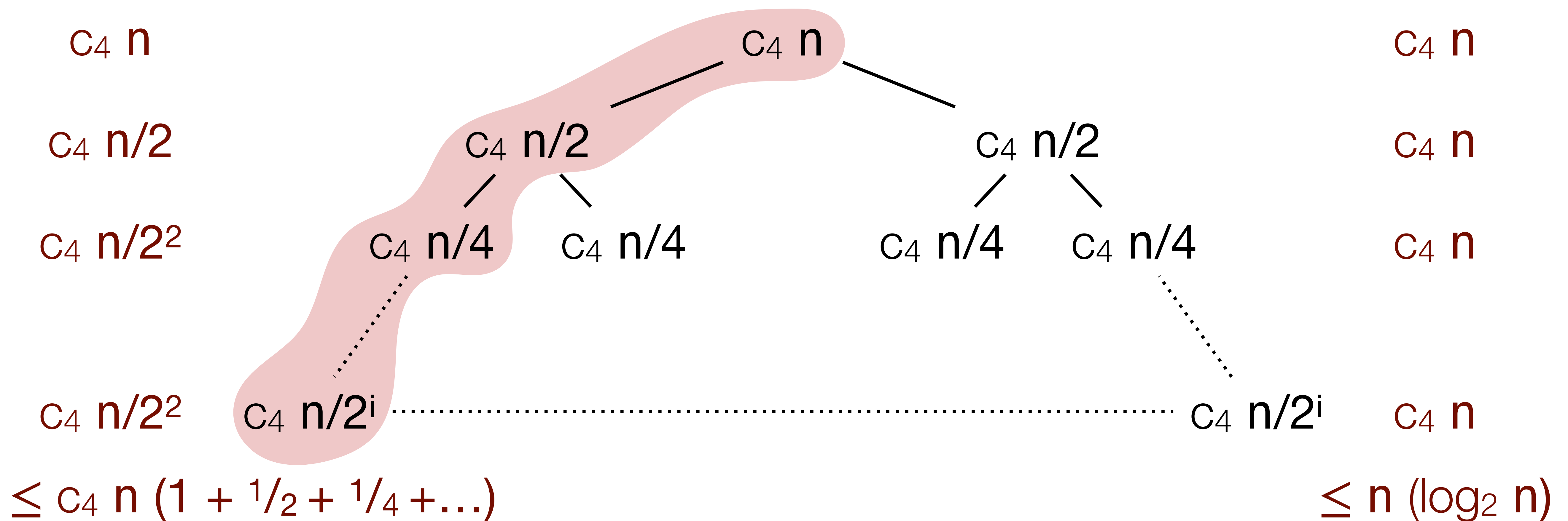
Consequently: $W_{msort}(n)$ is $O(n \log n)$

Span for mergesort for lists



Consequently: $W_{msort}(n)$ is $O(n \log n)$

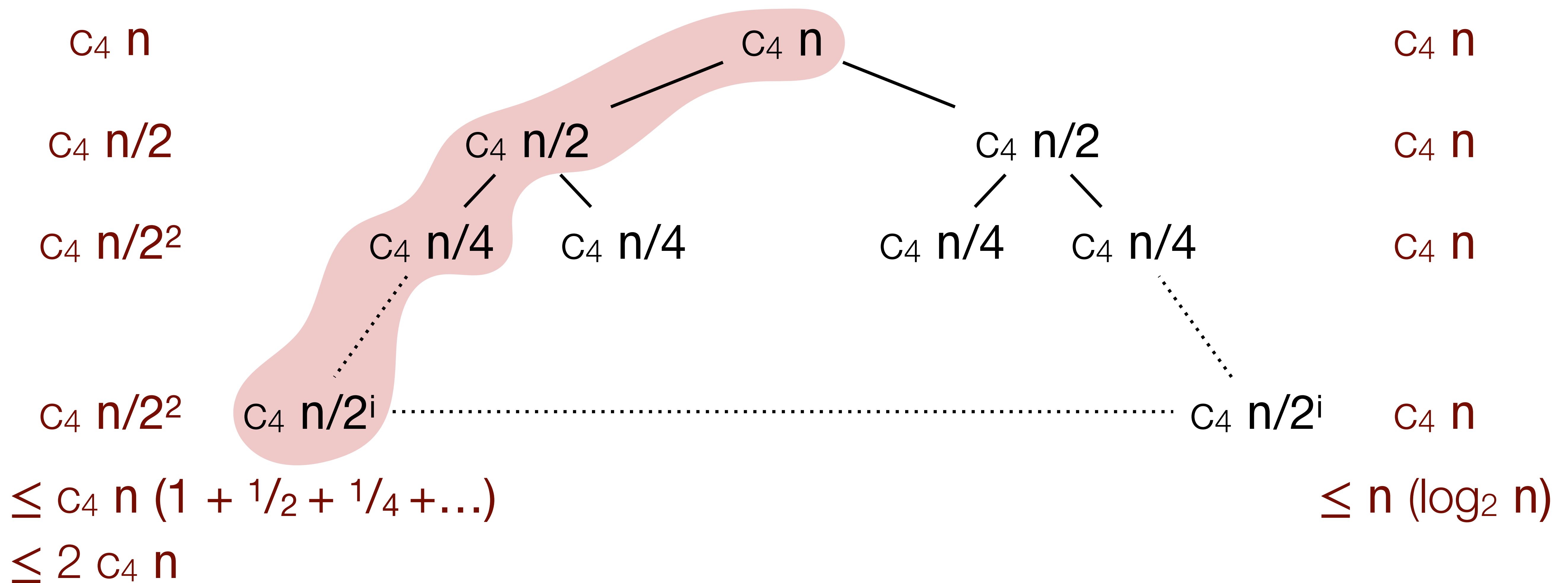
Span for mergesort for lists



Consequently: $W_{msort}(n)$ is $O(n \log n)$

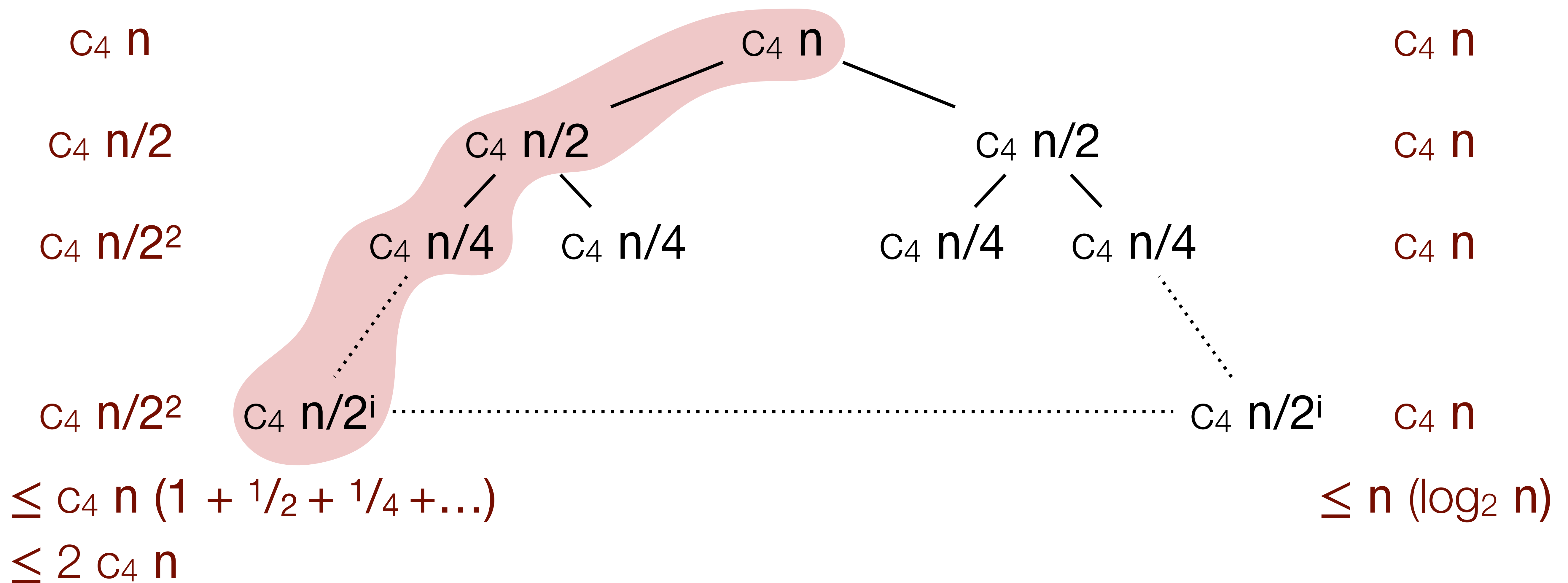
Span for mergesort for lists

span: $W_{\texttt{msort}}(n) \leq c_4\, n + W_{\texttt{msort}}(n/2)$ work:



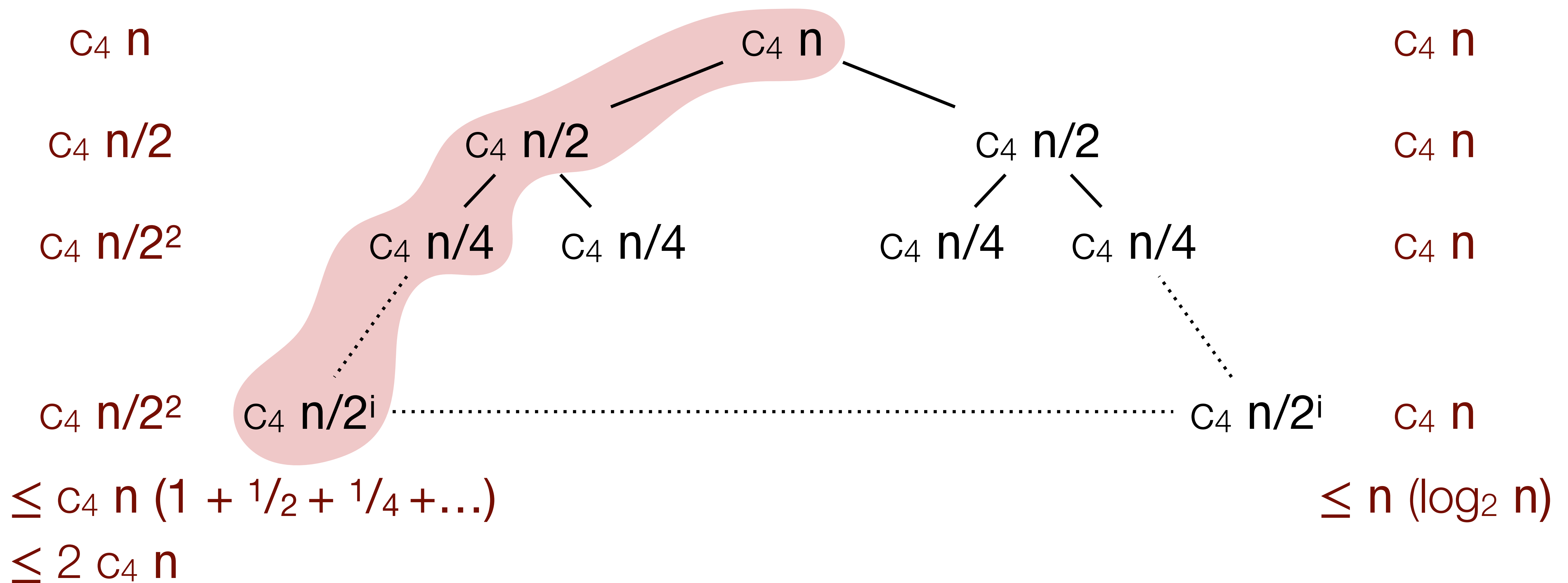
Consequently: $W_{\texttt{msort}}(n)$ is $O(n \log n)$

Span for mergesort for lists



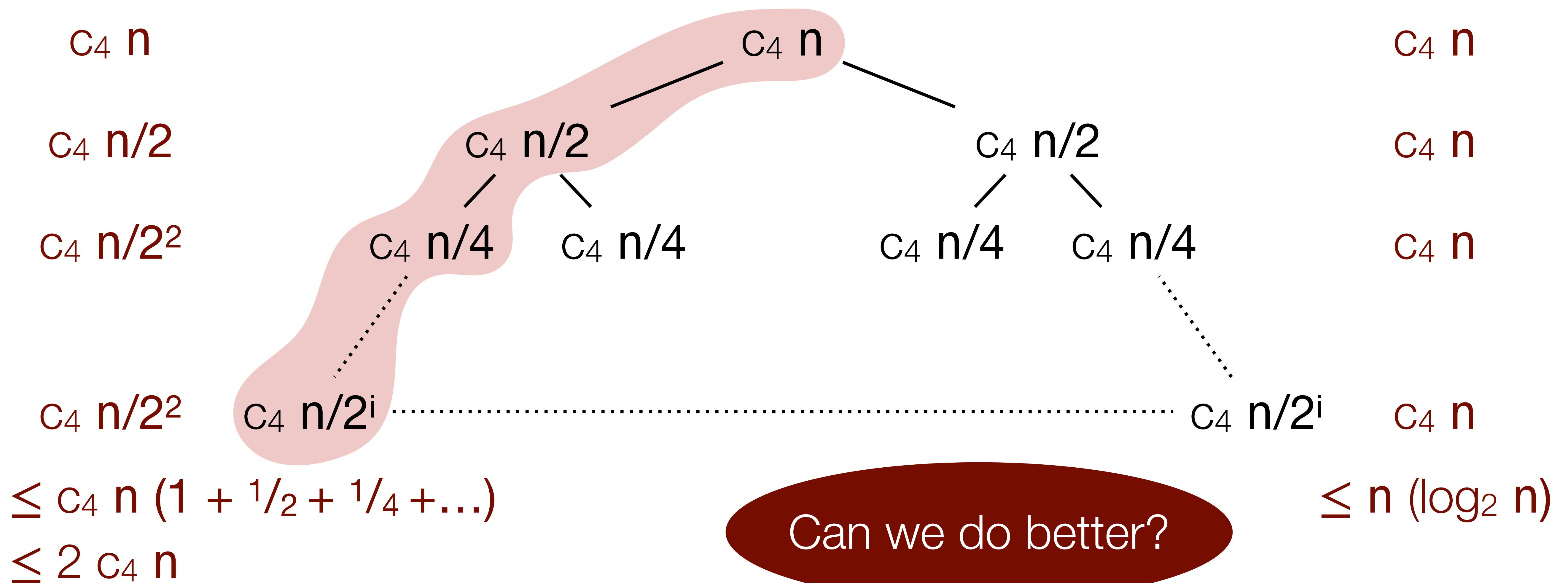
Consequently: $W_{msort}(n)$ is $O(n \log n)$ and $S_{msort}(n)$ is $O(n)$.

Span for mergesort for lists



Consequently: $W_{msort}(n)$ is $O(n \log n)$ and $S_{msort}(n)$ is $O(n)$.

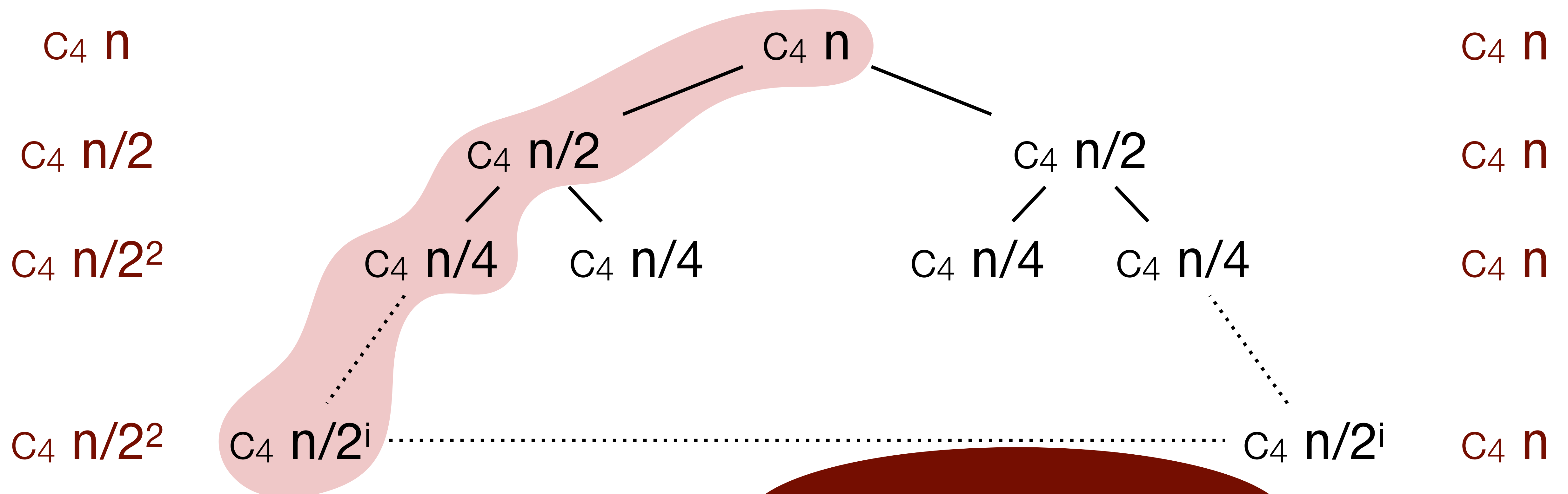
Span for mergesort for lists



Consequently: $W_{msort}(n)$ is $O(n \log n)$ and $S_{msort}(n)$ is $O(n)$.

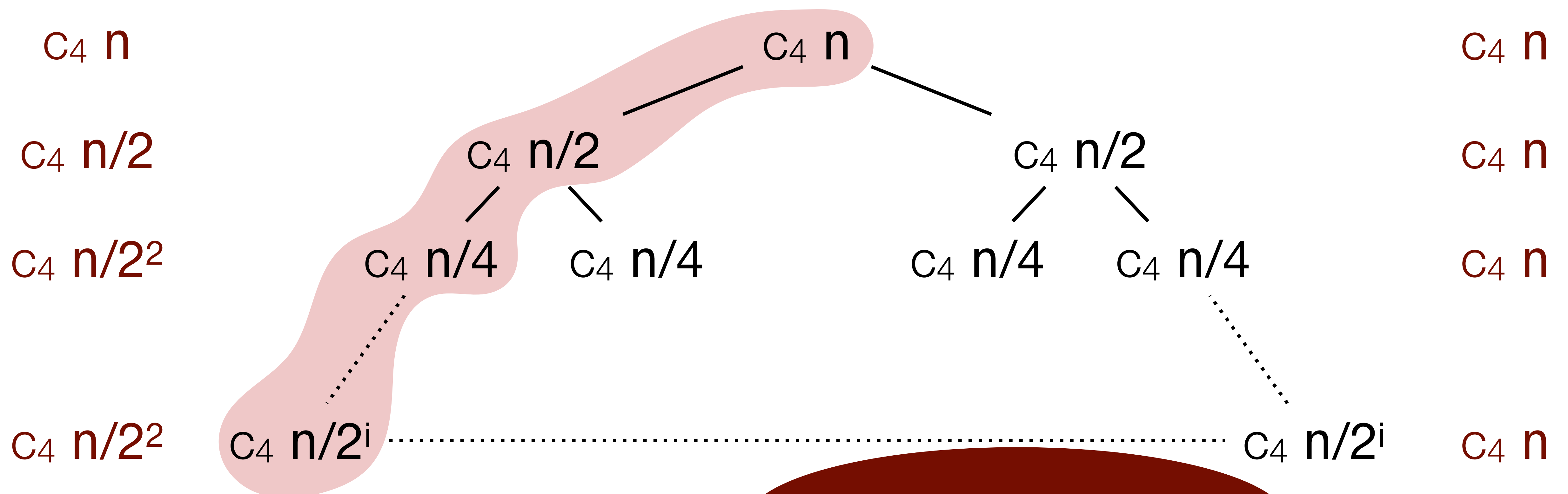
Span for mergesort for lists

span: $W_{msort}(n) \le c_4\, n + W_{msort}(n/2)$ work:



Consequently: $W_{msort}(n)$ is $O(n \log n)$ and $S_{msort}(n)$ is $O(n)$.

Span for mergesort for lists



Consequently: $W_{msort}(n)$ is $O(n \log n)$ and $S_{msort}(n)$ is $O(n)$.

What if we were given a tree, rather than a list?

Mergesort for int trees

Mergesort for int trees

Recall int tree datatype:

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

Mergesort for int trees

Recall int tree datatype:

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

Recall order datatype:

```
datatype order = LESS | EQUAL | GREATER
```

Mergesort for int trees

Recall int tree datatype:

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

Recall order datatype:

```
datatype order = LESS | EQUAL | GREATER
```

with:

```
Int.compare : int * int -> order
```

Mergesort for int trees

Mergesort for int trees

We define int trees to be sorted by:

Mergesort for int trees

We define int trees to be sorted by:

- `Empty` is sorted;

Mergesort for int trees

We define int trees to be sorted by:

- `Empty` is sorted;
- `Node(l,x,r)` is sorted iff

Mergesort for int trees

We define int trees to be sorted by:

- `Empty` is sorted;
- `Node(l,x,r)` is sorted iff
  - `l` is sorted and for every `y:int` in `l`, `Int.compare(y,x)` returns either `LESS` or `EQUAL`, and

Mergesort for int trees

We define int trees to be sorted by:

- `Empty` is sorted;
- `Node(l,x,r)` is sorted iff
  - `l` is sorted and for every `y:int` in `l`, `Int.compare(y,x)` returns either `LESS` or `EQUAL`, and
  - `r` is sorted and for every `z:int` in `r`, `Int.compare(z,x)` returns either `GREATER` or `EQUAL`.

Mergesort for int trees

We define int trees to be sorted by:

- `Empty` is sorted;
- `Node(l,x,r)` is sorted iff
  - `l` is sorted and for every `y:int` in `l`, `Int.compare(y,x)` returns either `LESS` or `EQUAL`, and
  - `r` is sorted and for every `z:int` in `r`, `Int.compare(z,x)` returns either `GREATER` or `EQUAL`.

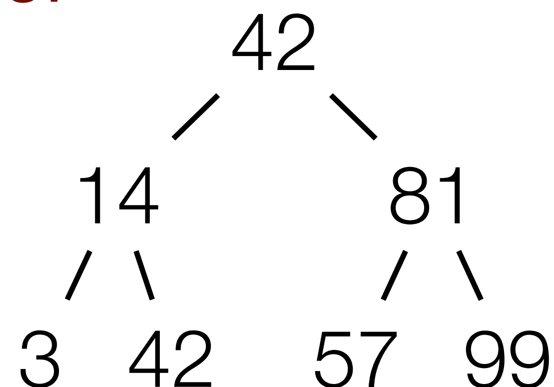
Eg, sorted tree:

Mergesort for int trees

We define int trees to be sorted by:

- `Empty` is sorted;
- `Node(l,x,r)` is sorted iff
  - `l` is sorted and for every `y:int` in `l`, `Int.compare(y,x)` returns either `LESS` or `EQUAL`, and
  - `r` is sorted and for every `z:int` in `r`, `Int.compare(z,x)` returns either `GREATER` or `EQUAL`.

Eg, sorted tree:

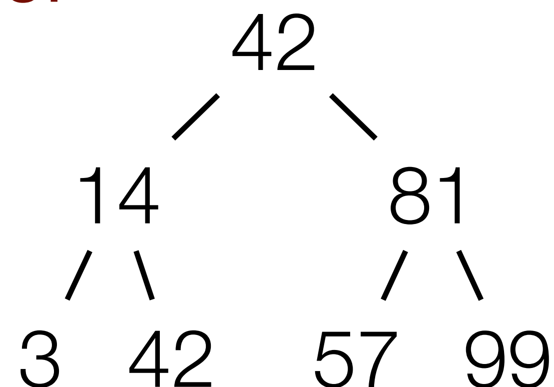


Mergesort for int trees

We define int trees to be sorted by:

- `Empty` is sorted;
- `Node(l,x,r)` is sorted iff
  - `l` is sorted and for every `y:int` in `l`, `Int.compare(y,x)` returns either `LESS` or `EQUAL`, and
  - `r` is sorted and for every `z:int` in `r`, `Int.compare(z,x)` returns either `GREATER` or `EQUAL`.

Eg, sorted tree:



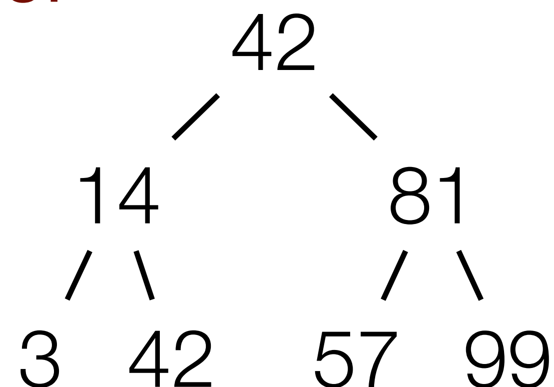
Eg, unsorted tree:

Mergesort for int trees

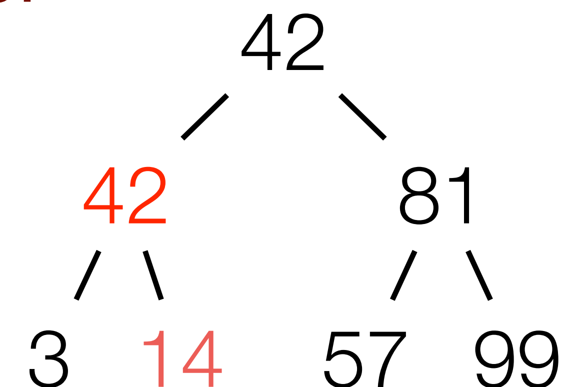
We define int trees to be sorted by:

- `Empty` is sorted;
- `Node(l,x,r)` is sorted iff
  - `l` is sorted and for every `y:int` in `l`, `Int.compare(y,x)` returns either `LESS` or `EQUAL`, and
  - `r` is sorted and for every `z:int` in `r`, `Int.compare(z,x)` returns either `GREATER` or `EQUAL`.

Eg, sorted tree:



Eg, unsorted tree:



Mergesort for int trees

Mergesort for int trees

Divide and conquer for sorting int trees:

Mergesort for int trees

Divide and conquer for sorting int trees:

- Split the tree into sub-trees;

Mergesort for int trees

Divide and conquer for sorting int trees:

- Split the tree into sub-trees;
- Sort the sub-trees;

Mergesort for int trees

Divide and conquer for sorting int trees:

- Split the tree into sub-trees;
- Sort the sub-trees;
- Merge the results.

Let’s implement the mergesort function!

Let's implement the mergesort function!

```
(* Msort : tree -> tree
   REQUIRES: true
   ENSURES: Msort(t) evaluates to a sorted tree containing
            exactly the elements of t (incl duplicates).
*)
```

Let’s implement the mergesort function!

```
(* Msort : tree -> tree
   REQUIRES: true
   ENSURES: Msort(t) evaluates to a sorted tree containing
            exactly the elements of t (incl duplicates).
*)

fun Msort (Empty : tree) : tree =
```

Let's implement the mergesort function!

```
(* Msort : tree -> tree
   REQUIRES: true
   ENSURES: Msort(t) evaluates to a sorted tree containing
            exactly the elements of t (incl duplicates).
*)

fun Msort (Empty : tree) : tree = Empty
```

Let’s implement the mergesort function!

```
(* Msort : tree -> tree
   REQUIRES: true
   ENSURES: Msort(t) evaluates to a sorted tree containing
            exactly the elements of t (incl duplicates).
*)

fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      Ins (x,                                  )
```

Let's implement the mergesort function!

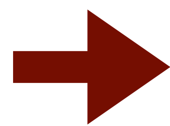
```
(* Msort : tree -> tree
   REQUIRES: true
   ENSURES: Msort(t) evaluates to a sorted tree containing
            exactly the elements of t (incl duplicates).
*)

fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      Ins (x, Merge(Msort left, Msort right))
```

Let's implement the mergesort function!

```
(* Msort : tree -> tree
   REQUIRES: true
   ENSURES: Msort(t) evaluates to a sorted tree containing
            exactly the elements of t (incl duplicates).
*)

fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      Ins (x, Merge(Msort left, Msort right))
```



Note: no splitting to create the "computation tree" necessary anymore as with int list! We are already provided with a tree.

Let's implement the insert function!

Let’s implement the insert function!

```
(* Ins : int * tree -> tree
   REQUIRES: t is a sorted tree.
   ENSURES: Ins(x,t) returns a sorted tree containing x
            along with the elements of t (incl duplicates).
*)
```

Let’s implement the insert function!

```
(* Ins : int * tree -> tree
   REQUIRES: t is a sorted tree.
   ENSURES: Ins(x,t) returns a sorted tree containing x
            along with the elements of t (incl duplicates).
*)

fun Ins (x: int, Empty: tree) : tree =
```

Let’s implement the insert function!

```
(* Ins : int * tree -> tree
   REQUIRES: t is a sorted tree.
   ENSURES: Ins(x,t) returns a sorted tree containing x
            along with the elements of t (incl duplicates).
*)

fun Ins (x: int, Empty: tree) : tree = Node(Empty, x, Empty)
```

Let's implement the insert function!

```
(* Ins : int * tree -> tree
   REQUIRES: t is a sorted tree.
   ENSURES: Ins(x,t) returns a sorted tree containing x
            along with the elements of t (incl duplicates).
*)

fun Ins (x: int, Empty: tree) : tree = Node(Empty, x, Empty)
  | Ins (x, Node(l, y, r)) =
```

Let’s implement the insert function!

```
(* Ins : int * tree -> tree
   REQUIRES: t is a sorted tree.
   ENSURES: Ins(x,t) returns a sorted tree containing x
            along with the elements of t (incl duplicates).
*)

fun Ins (x: int, Empty: tree) : tree = Node(Empty, x, Empty)
  | Ins (x, Node(l, y, r)) =
     (case Int.compare(x,y) of
```

Let's implement the insert function!

```
(* Ins : int * tree -> tree
   REQUIRES: t is a sorted tree.
   ENSURES: Ins(x,t) returns a sorted tree containing x
            along with the elements of t (incl duplicates).
*)

fun Ins (x: int, Empty: tree) : tree = Node(Empty, x, Empty)
  | Ins (x, Node(l, y, r)) =
     (case Int.compare(x,y) of
          GREATER   =>
```

Let's implement the insert function!

```
(* Ins : int * tree -> tree
   REQUIRES: t is a sorted tree.
   ENSURES: Ins(x,t) returns a sorted tree containing x
            along with the elements of t (incl duplicates).
*)

fun Ins (x: int, Empty: tree) : tree = Node(Empty, x, Empty)
  | Ins (x, Node(l, y, r)) =
    (case Int.compare(x,y) of
         GREATER   => Node(l, y, Ins(x, r))
```

Let’s implement the insert function!

```
(* Ins : int * tree -> tree
   REQUIRES: t is a sorted tree.
   ENSURES: Ins(x,t) returns a sorted tree containing x
            along with the elements of t (incl duplicates).
*)

fun Ins (x: int, Empty: tree) : tree = Node(Empty, x, Empty)
  | Ins (x, Node(l, y, r)) =
     (case Int.compare(x,y) of
          GREATER   => Node(l, y, Ins(x, r))
        | _         =>
```

Let's implement the insert function!

```
(* Ins : int * tree -> tree
   REQUIRES: t is a sorted tree.
   ENSURES: Ins(x,t) returns a sorted tree containing x
            along with the elements of t (incl duplicates).
*)

fun Ins (x: int, Empty: tree) : tree = Node(Empty, x, Empty)
  | Ins (x, Node(l, y, r)) =
     (case Int.compare(x,y) of
          GREATER   => Node(l, y, Ins(x, r))
        | _         => Node(Ins(x, l), y, r))
```

Let's implement the merge function!

Let’s implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

Let's implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
               exactly the elements of t1 and t2 (incl dupls).
*)
```

Let’s implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

Let's implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

Let's implement the merge function!

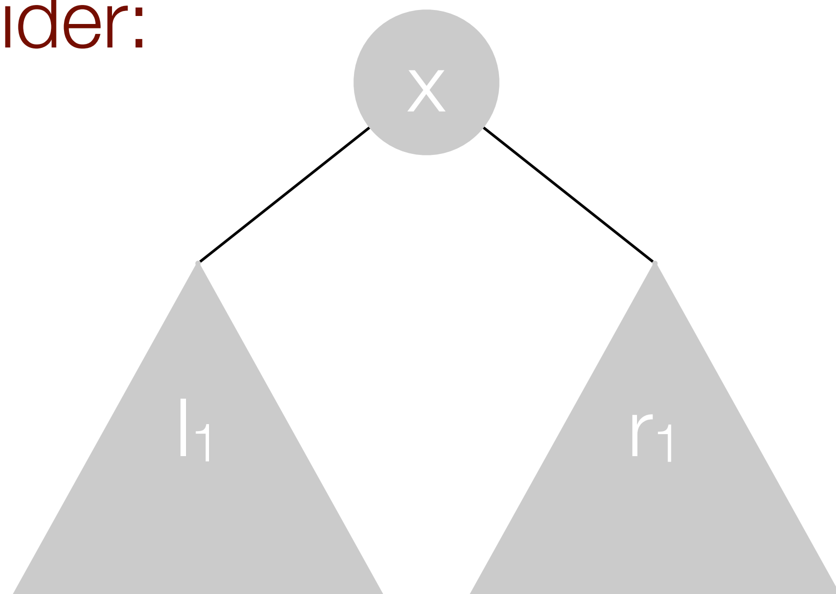
```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

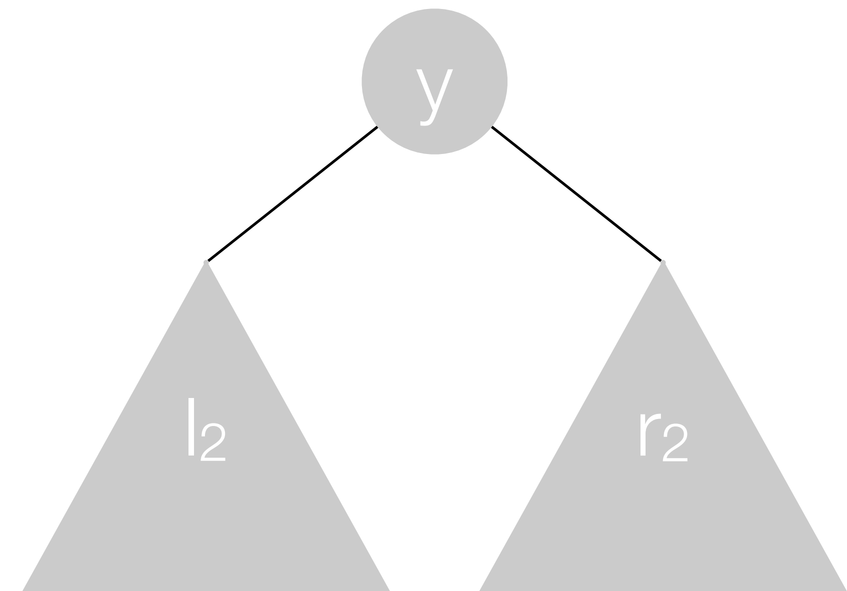
Consider:

Let’s implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

Consider:





elements in $l_1 \leq x$ and $r_1 \geq x$,
l_1 and r_1 are sorted

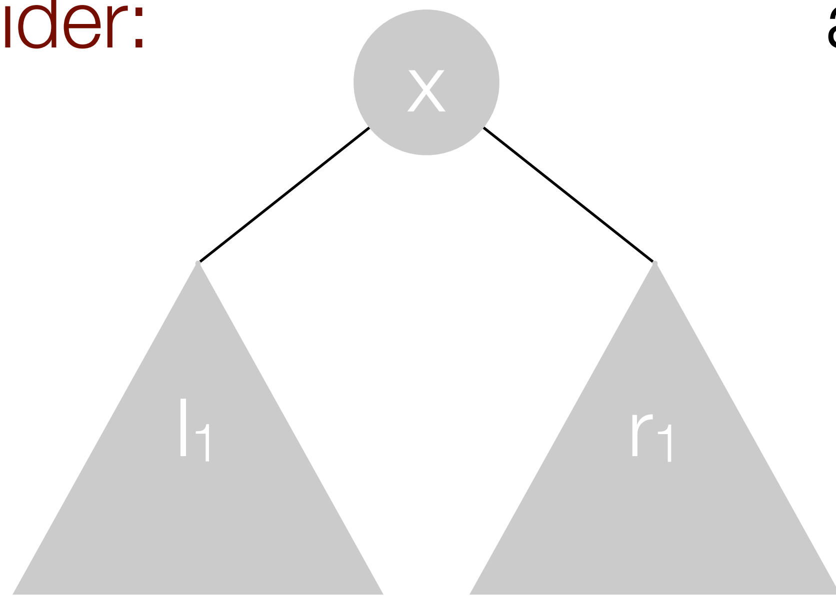
elements in $l_2 \leq y$ and $r_2 \geq y$,
l_2 and r_2 are sorted

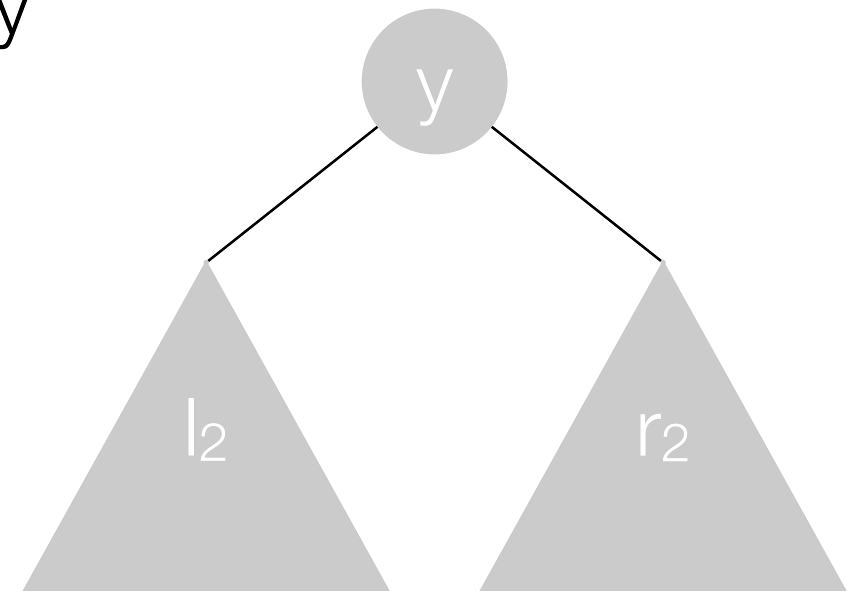
Let's implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

Consider:

assume $x \le y$





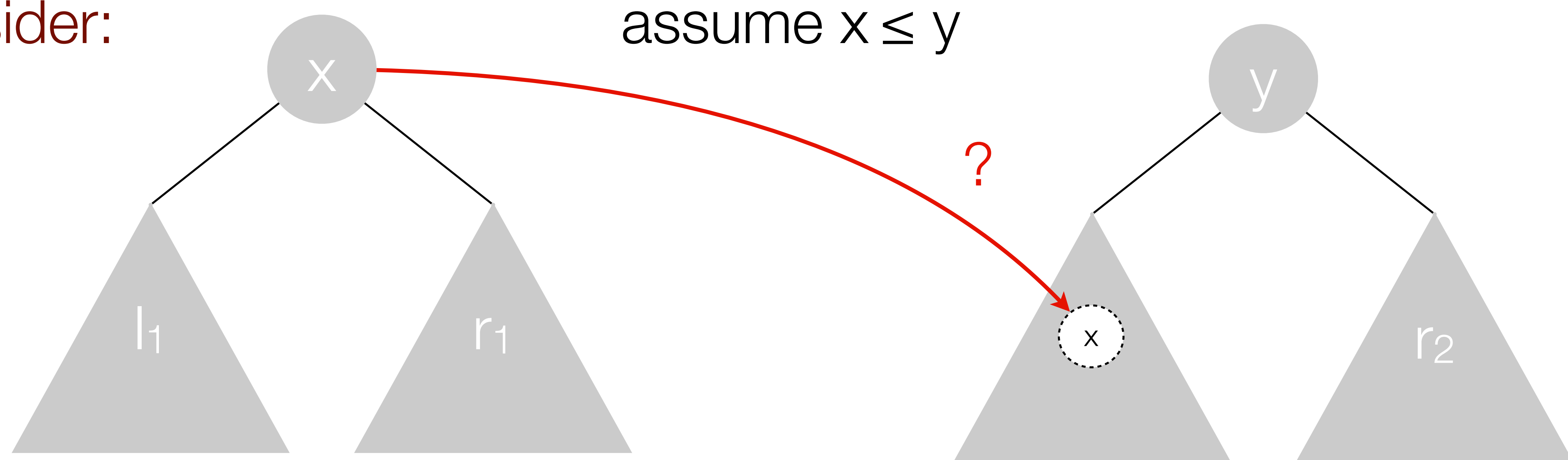
elements in $l_1 \le x$ and $r_1 \ge x$,
l_1 and r_1 are sorted

elements in $l_2 \le y$ and $r_2 \ge y$,
l_2 and r_2 are sorted

Let’s implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

Consider:

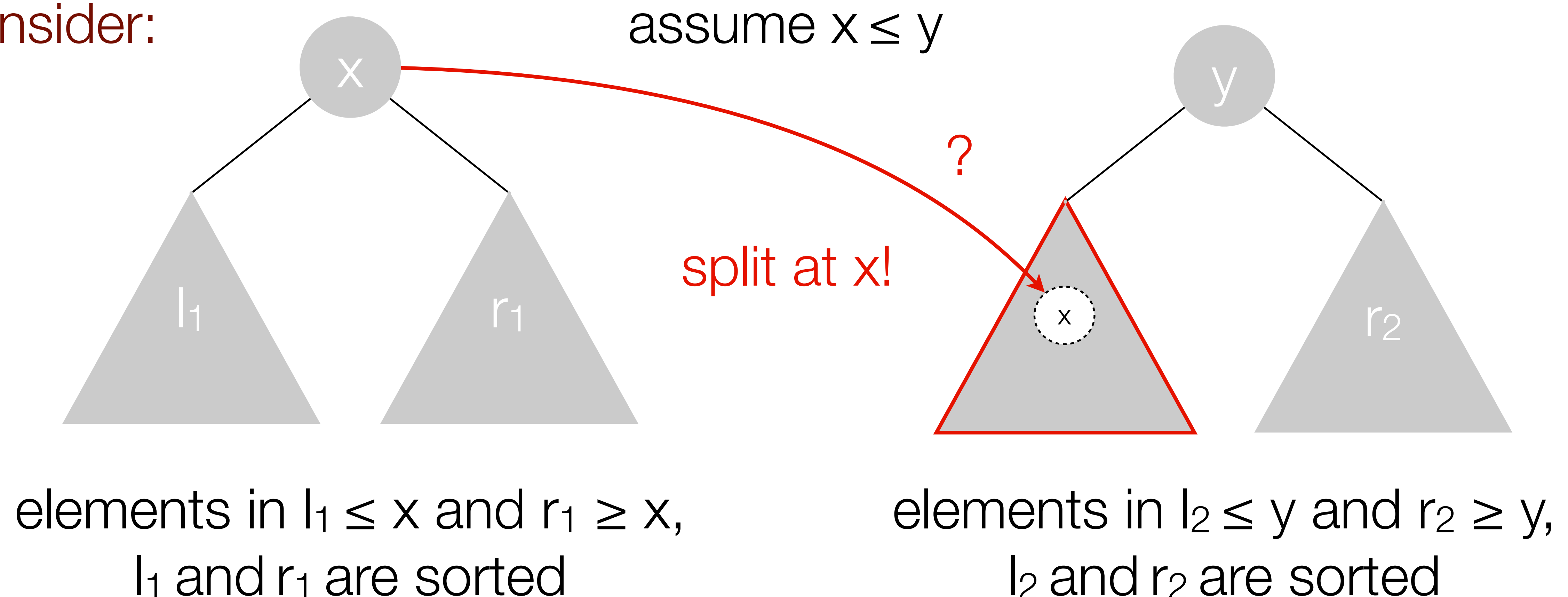


elements in $l_1 \leq x$ and $r_1 \geq x$,
l_1 and r_1 are sorted

elements in $l_2 \leq y$ and $r_2 \geq y$,
l_2 and r_2 are sorted

Let's implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

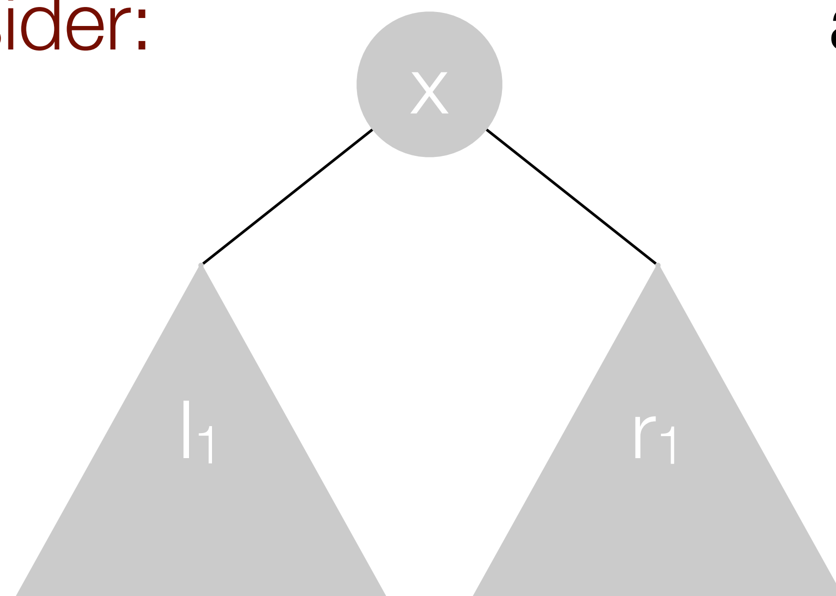


Let’s implement the merge function!

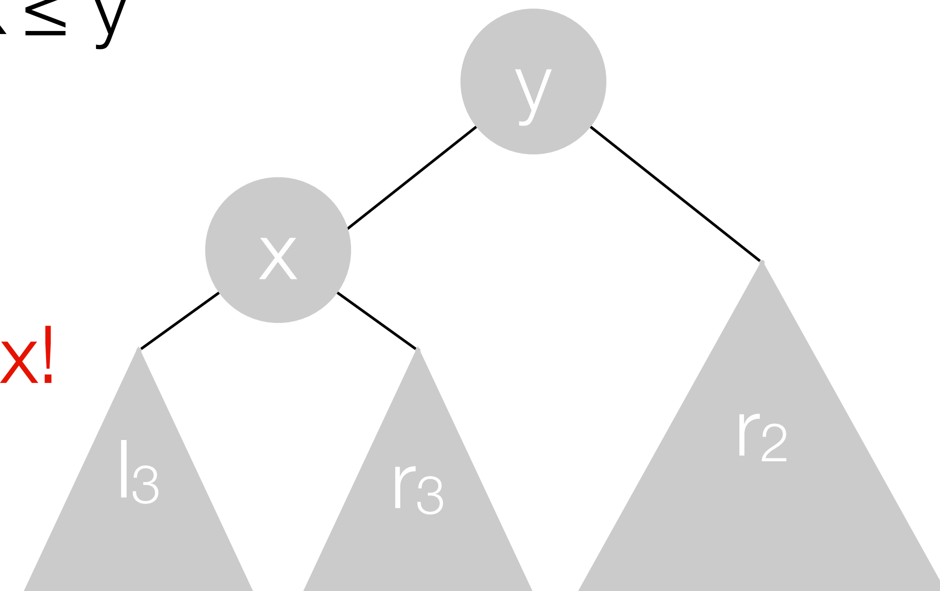
```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

Consider:

assume $x \leq y$



split at x!



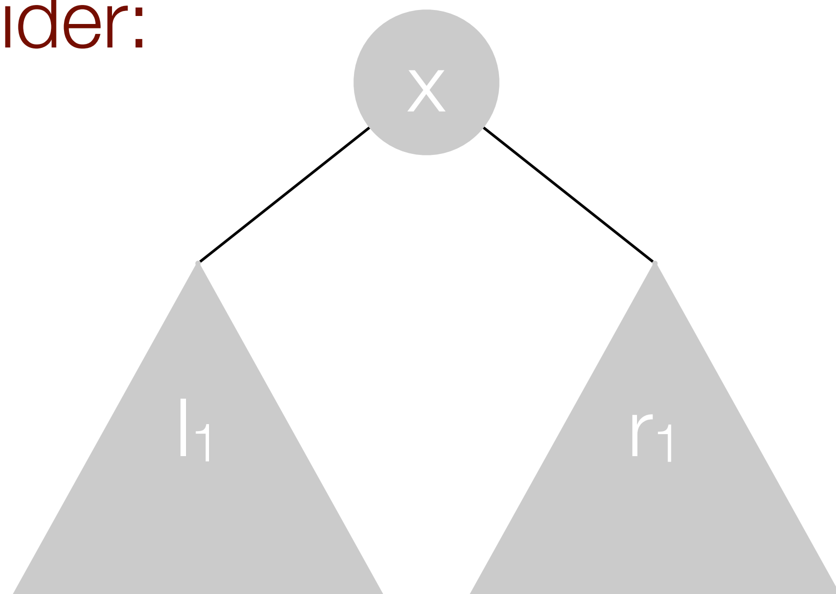
elements in $l_1 \leq x$ and $r_1 \geq x$,
l_1 and r_1 are sorted

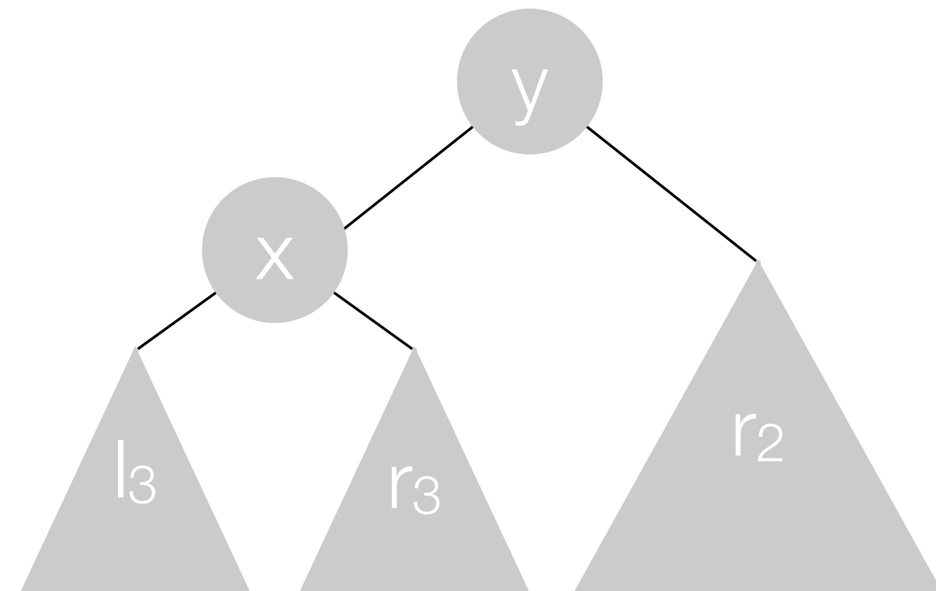
elements in l_3, $r_3 \leq y$ and $r_2 \geq y$,
elements in $l_3 \leq x$ and $r_3 \geq x$,
l_3, r_3 and r_2 are sorted

Let’s implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

Consider:





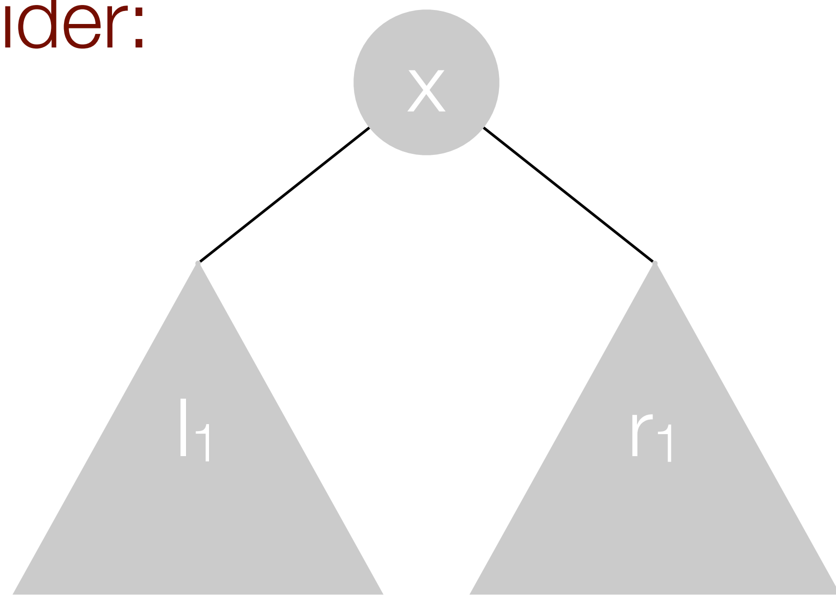
elements in $l_1 \leq x$ and $r_1 \geq x$,
l_1 and r_1 are sorted

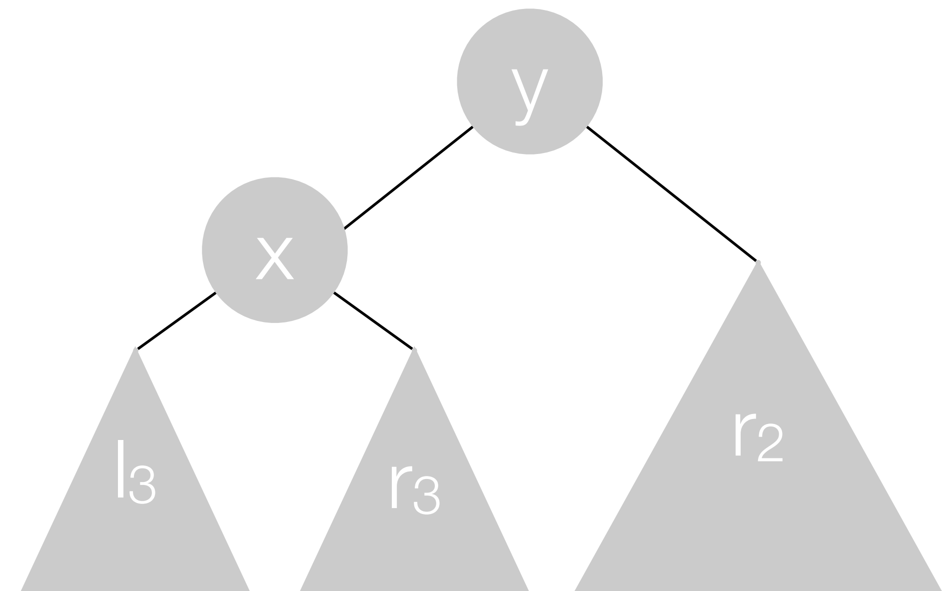
elements in l_3, $r_3 \leq y$ and $r_2 \geq y$,
elements in $l_3 \leq x$ and $r_3 \geq x$,
l_3, r_3 and r_2 are sorted

Let's implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

Consider:





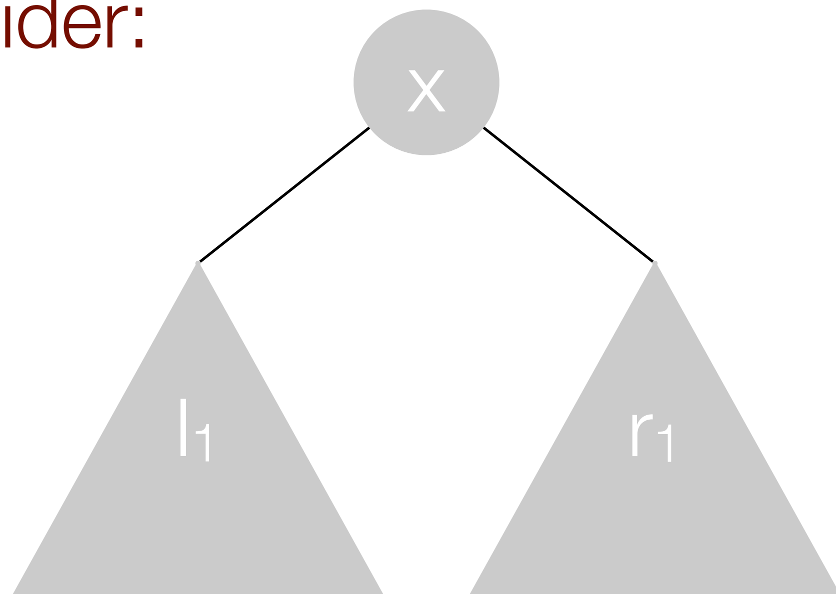
elements in $l_1 \le x$ and $r_1 \ge x$,
l_1 and r_1 are sorted

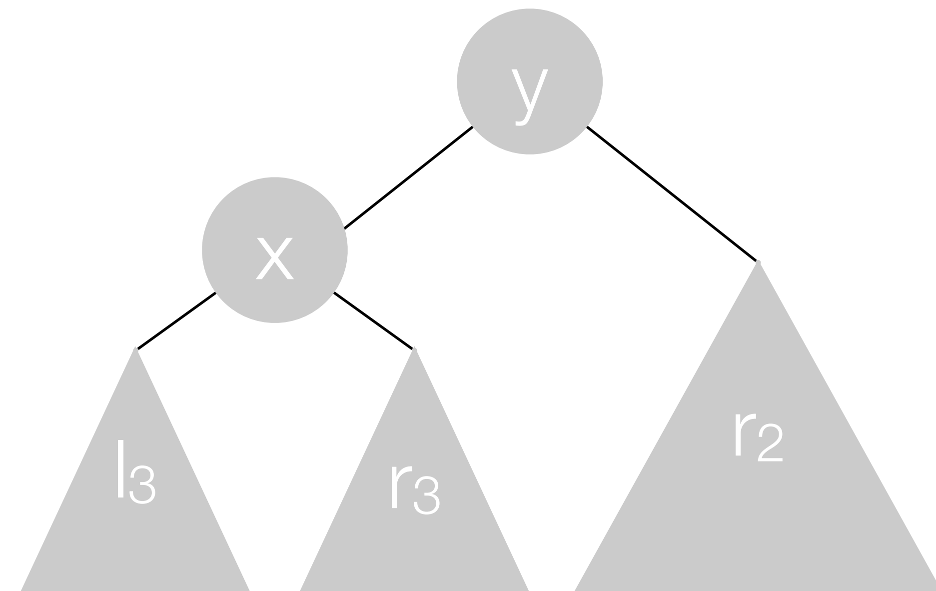
elements in l_3, $r_3 \le y$ and $r_2 \ge y$,
elements in $l_3 \le x$ and $r_3 \ge x$,
l_3, r_3 and r_2 are sorted

Let's implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

Consider:





elements in $l_1 \leq x$ and $r_1 \geq x$,
l_1 and r_1 are sorted

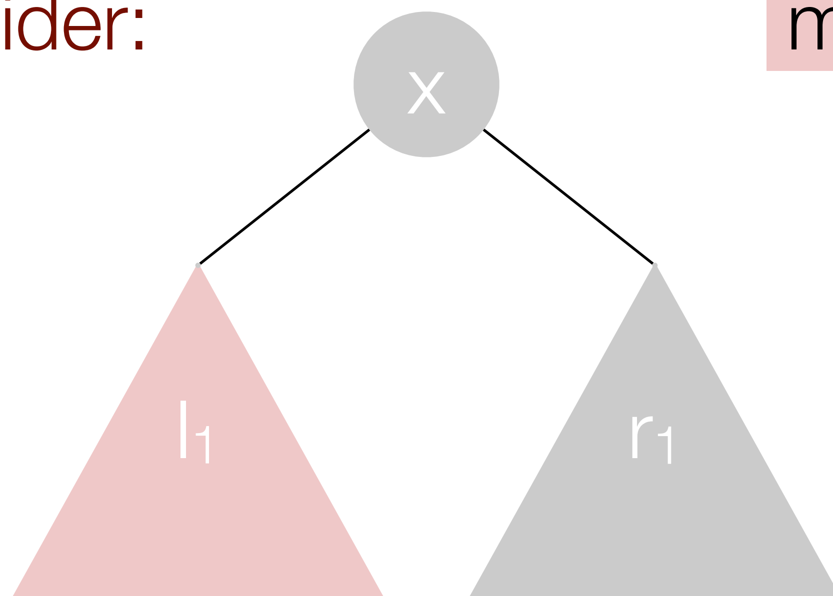
elements in l_3, $r_3 \leq y$ and $r_2 \geq y$,
elements in $l_3 \leq x$ and $r_3 \geq x$,
l_3, r_3 and r_2 are sorted

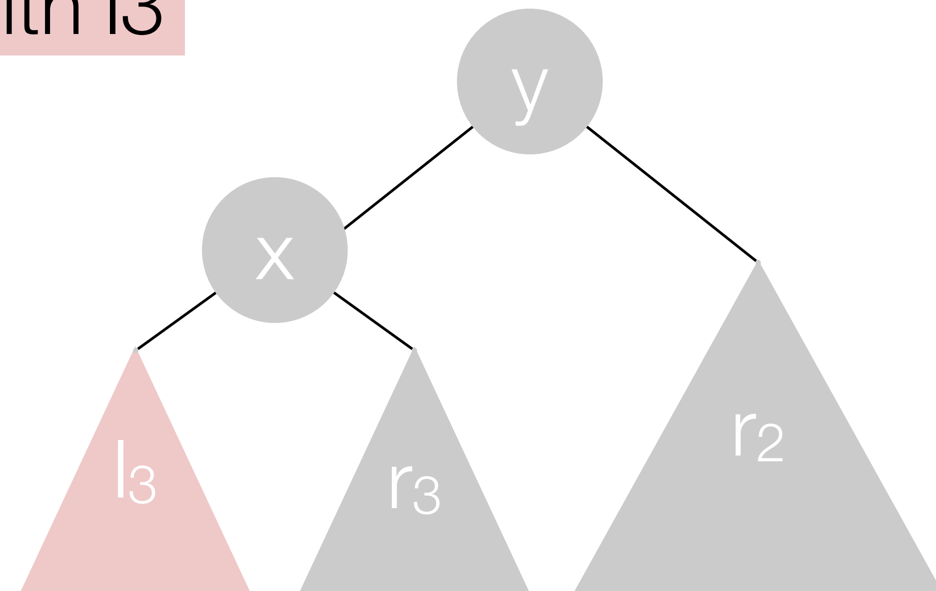
Let's implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

Consider:

merge l_1 with l3





elements in $l_1 \leq x$ and $r_1 \geq x$,
l_1 and r_1 are sorted

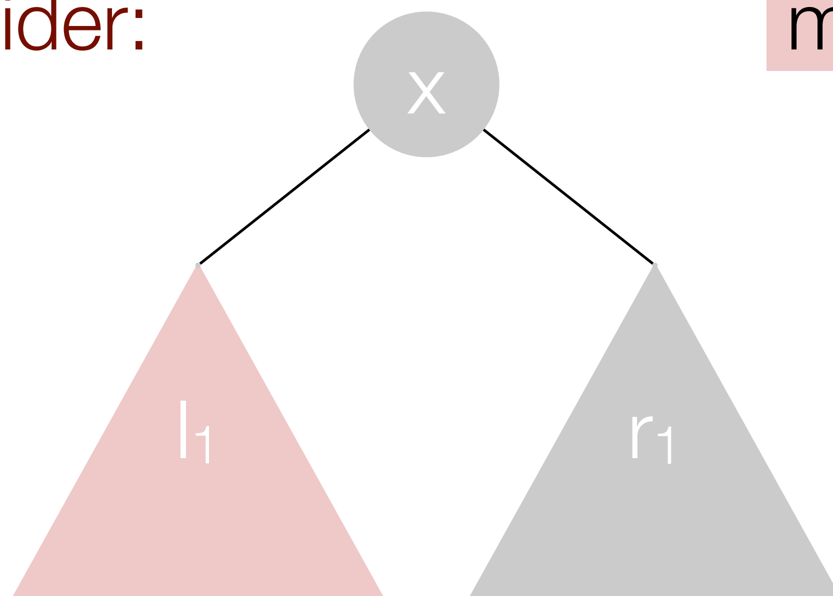
elements in l_3, $r_3 \leq y$ and $r_2 \geq y$,
elements in $l_3 \leq x$ and $r_3 \geq x$,
l_3, r_3 and r_2 are sorted

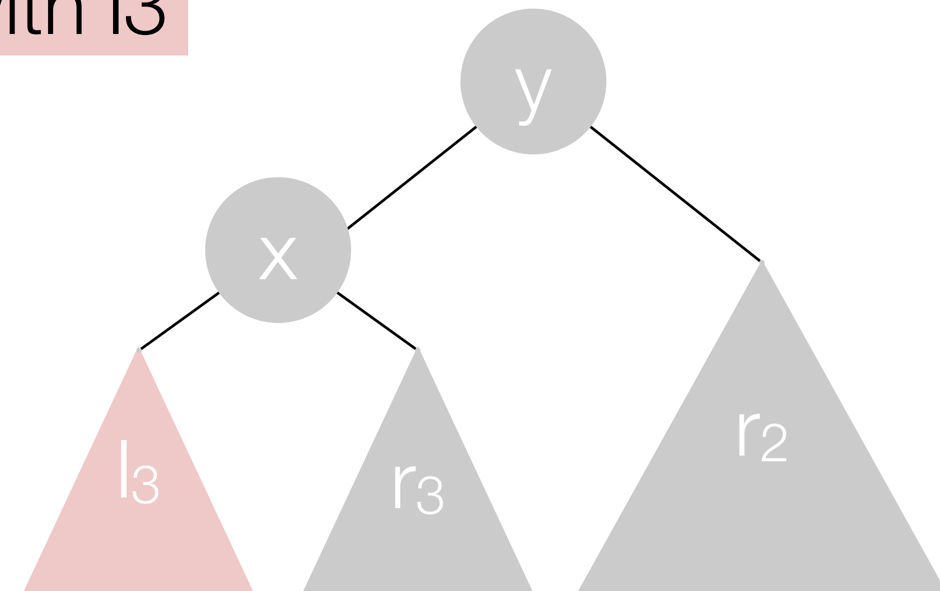
Let's implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

Consider:

merge l_1 with l3





elements in $l_1 \leq x$ and $r_1 \geq x$,
l_1 and r_1 are sorted

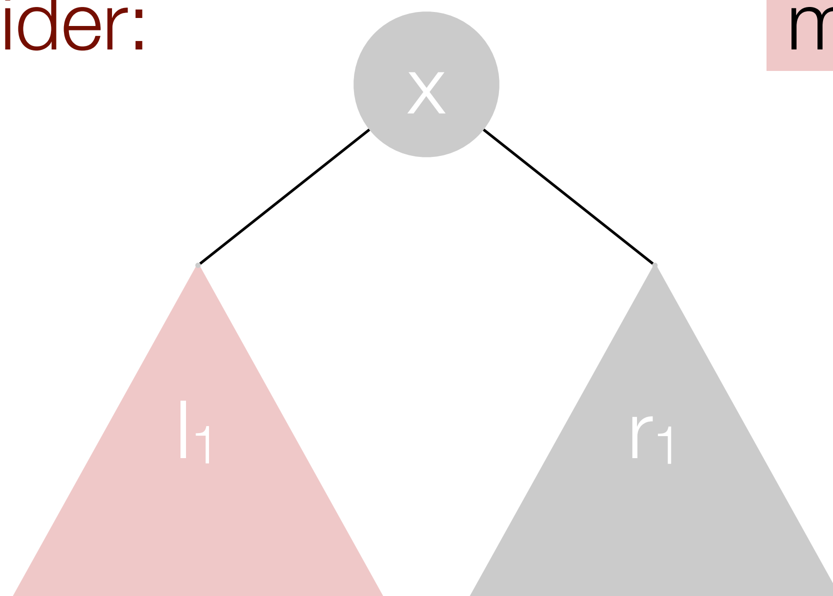
elements in l_3, $r_3 \leq y$ and $r_2 \geq y$,
elements in $l_3 \leq x$ and $r_3 \geq x$,
l_3, r_3 and r_2 are sorted

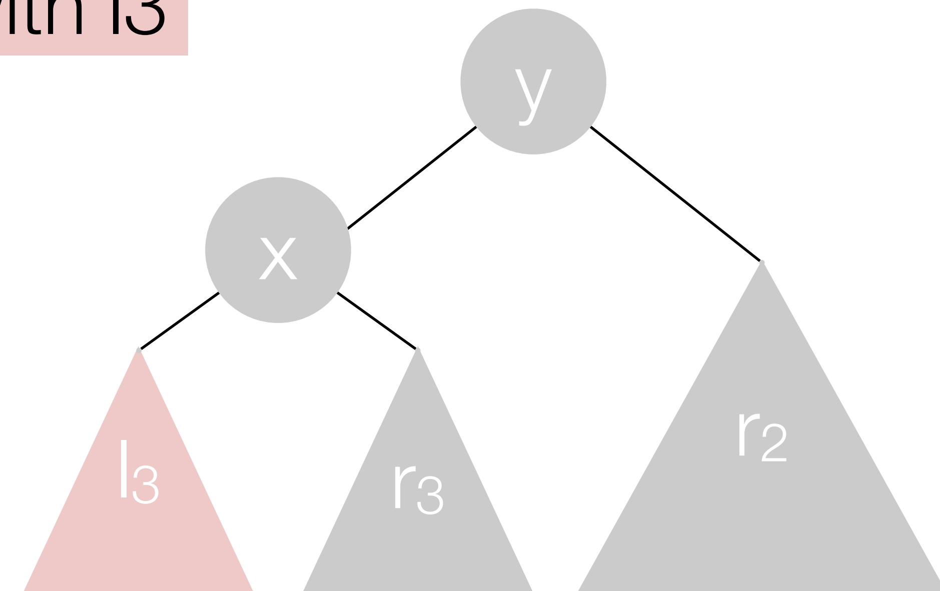
Let's implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

Consider:

merge l_1 with l3





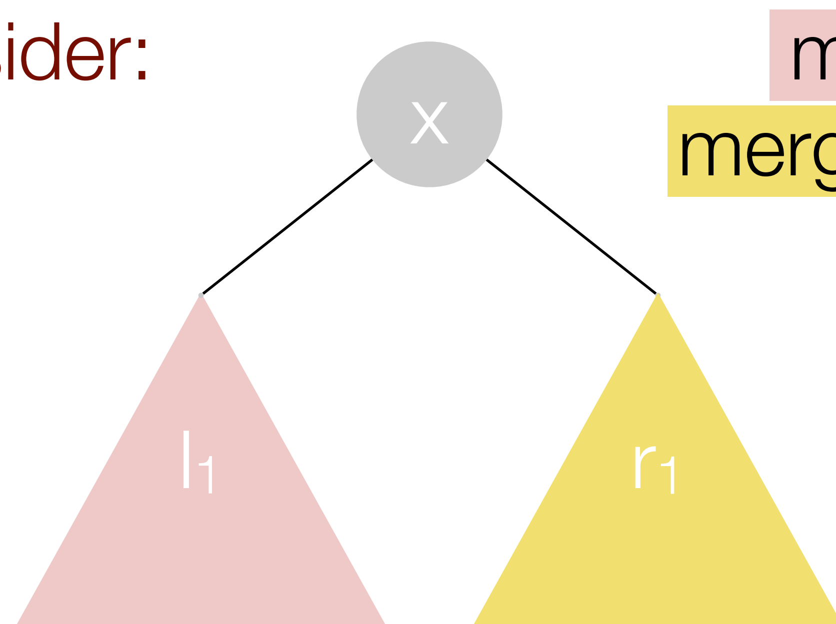
elements in $l_1 \leq x$ and $r_1 \geq x$,
l_1 and r_1 are sorted

elements in l_3, $r_3 \leq y$ and $r_2 \geq y$,
elements in $l_3 \leq x$ and $r_3 \geq x$,
l_3, r_3 and r_2 are sorted

Let's implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

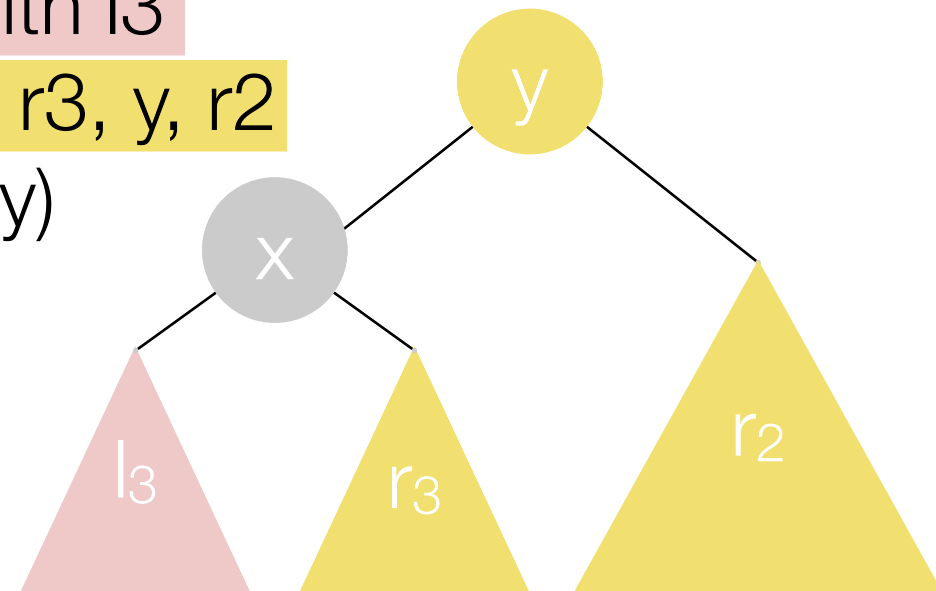
Consider:



merge l_1 with l3

merge r1 with r3, y, r2

(for $x \leq y$)



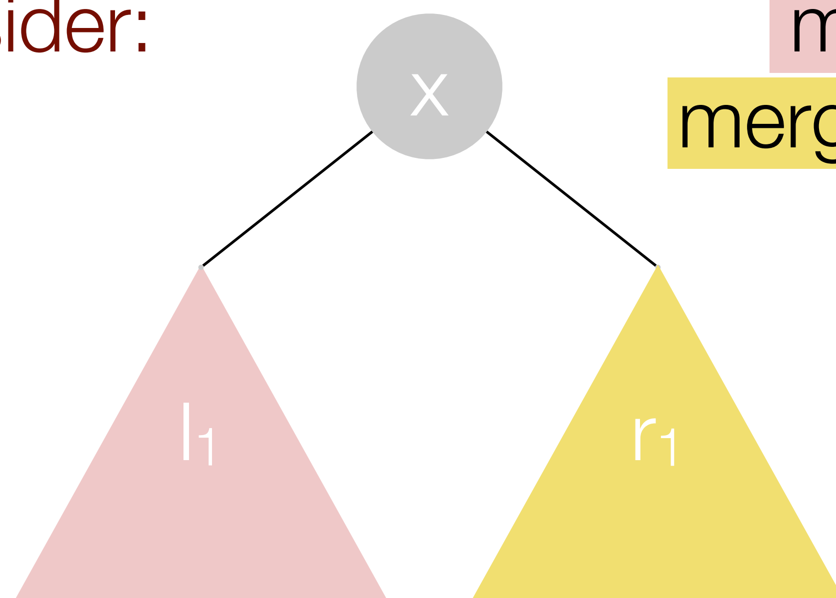
elements in $l_1 \leq x$ and $r_1 \geq x$,
l_1 and r_1 are sorted

elements in l_3, $r_3 \leq y$ and $r_2 \geq y$,
elements in $l_3 \leq x$ and $r_3 \geq x$,
l_3, r_3 and r_2 are sorted

Let’s implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)
```

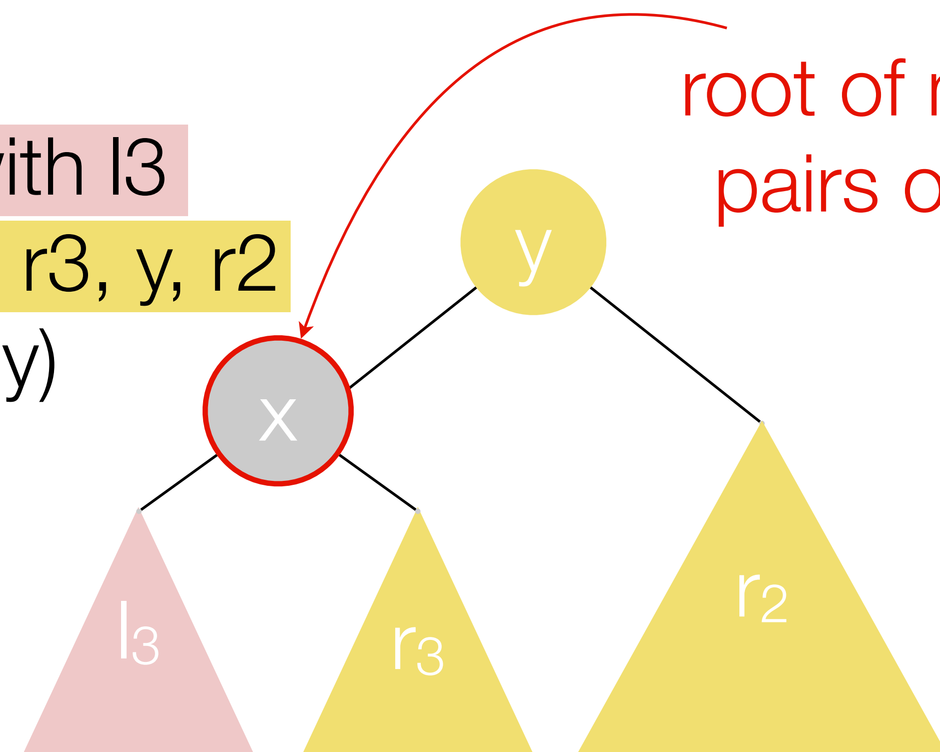
Consider:



merge l_1 with l3

merge r1 with r3, y, r2

(for $x \leq y$)



root of merged pairs of trees

elements in $l_1 \leq x$ and $r_1 \geq x$,
l_1 and r_1 are sorted

elements in l_3, $r_3 \leq y$ and $r_2 \geq y$,
elements in $l_3 \leq x$ and $r_3 \geq x$,
l_3, r_3 and r_2 are sorted

Let’s implement the merge function!

Let’s implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)

fun Merge (Empty : tree, t2 : tree) : tree =
```

Let's implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)

fun Merge (Empty : tree, t2 : tree) : tree = t2
```

Let's implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)

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

      end
```

Let's implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)

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

      end
```

Let's implement the merge function!

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted.
   ENSURES: Merge(t1,t2) returns a sorted tree containing
            exactly the elements of t1 and t2 (incl dupls).
*)

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

Merge may not return a balanced tree!

Merge may not return a balanced tree!

The depth of `Merge(t1,t2)` can be the sum of the depths of `t1`, `t2`!

Merge may not return a balanced tree!

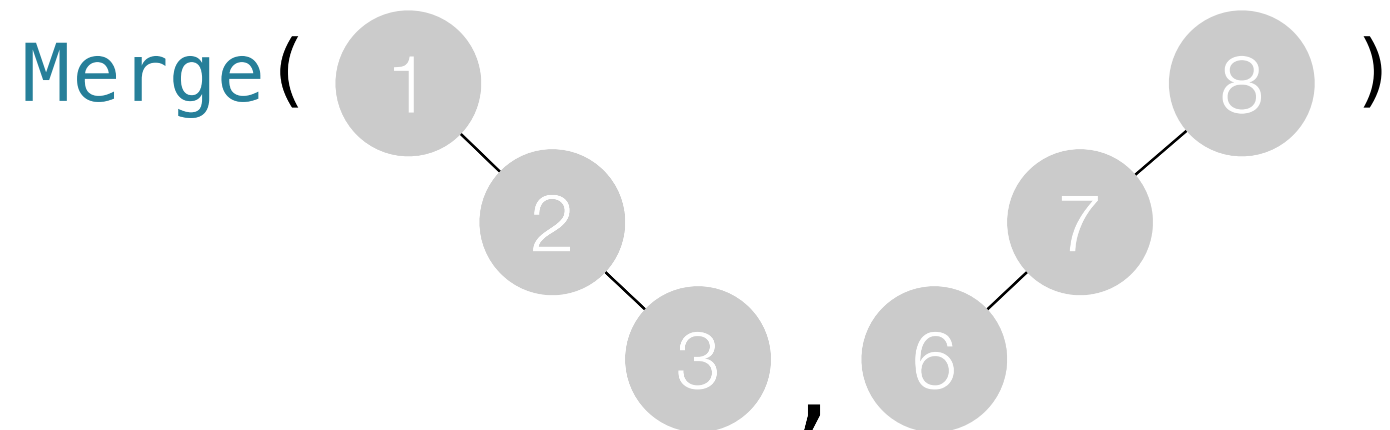
The depth of `Merge(t1,t2)` can be the sum of the depths of `t1`, `t2`!

Eg:

Merge may not return a balanced tree!

The depth of `Merge(t1,t2)` can be the sum of the depths of `t1`, `t2`!

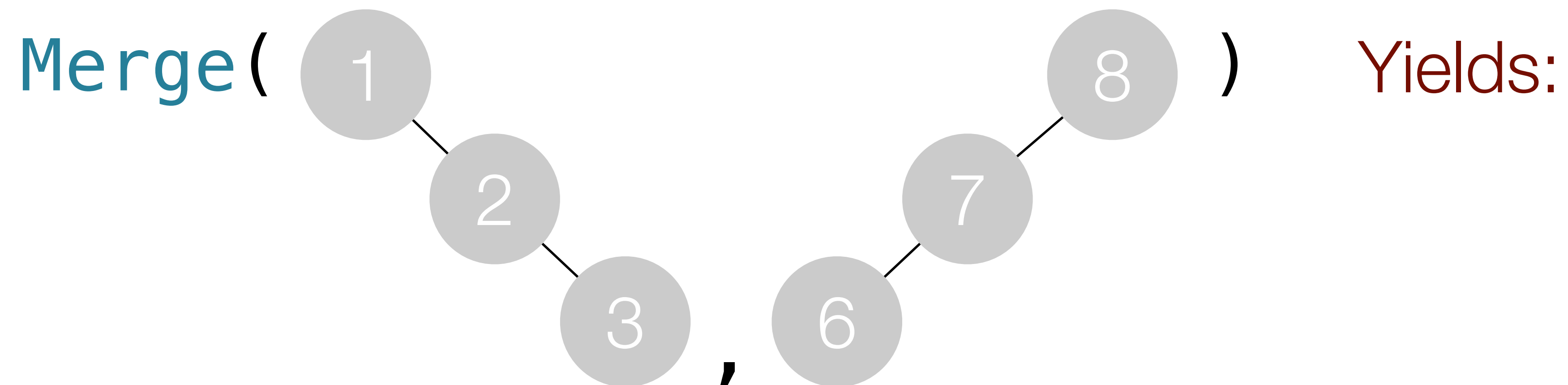
Eg:



Merge may not return a balanced tree!

The depth of `Merge(t1,t2)` can be the sum of the depths of `t1`, `t2`!

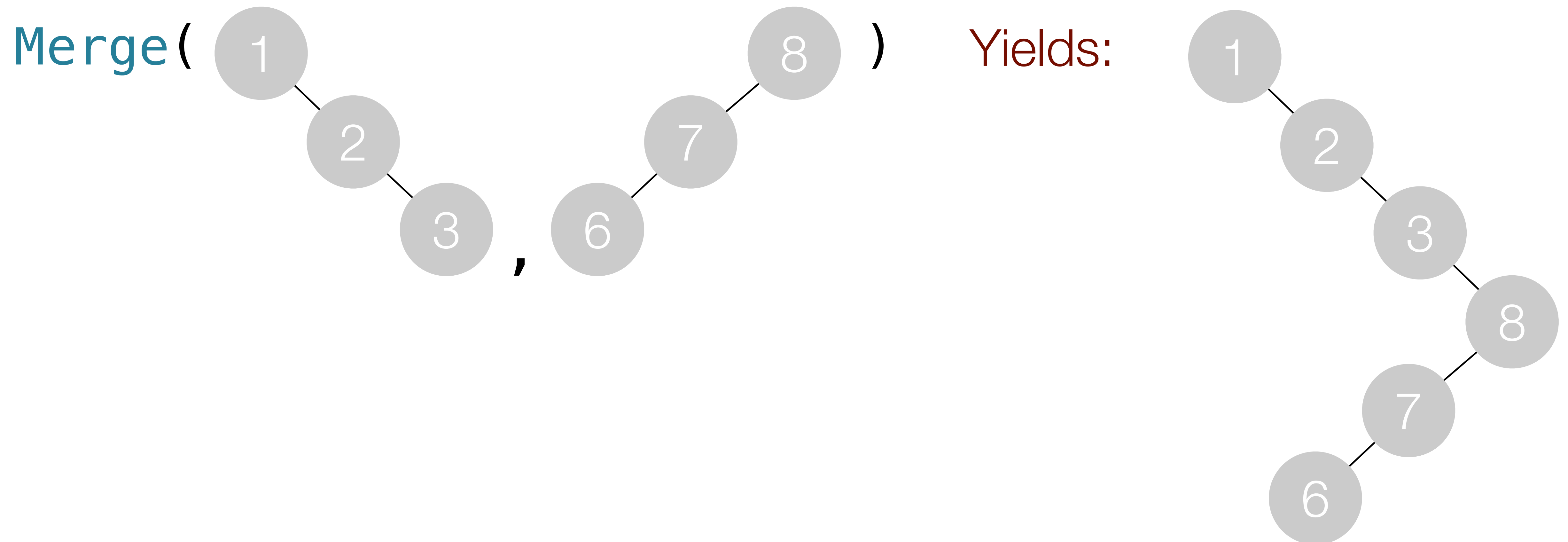
Eg:



Merge may not return a balanced tree!

The depth of `Merge(t1,t2)` can be the sum of the depths of `t1`, `t2`!

Eg:

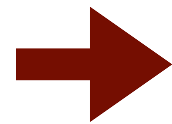


Merge may not return a balanced tree!

The depth of `Merge(t1,t2)` can be the sum of the depths of `t1`, `t2`!

Merge may not return a balanced tree!

The depth of `Merge(t1,t2)` can be the sum of the depths of `t1`, `t2`!



To obtain fast code, we must rebalance.

Merge may not return a balanced tree!

The depth of `Merge(t1,t2)` can be the sum of the depths of `t1`, `t2`!

➡ To obtain fast code, we must rebalance.

➡ One can do so without affecting asymptotic cost.

Merge may not return a balanced tree!

The depth of `Merge(t1,t2)` can be the sum of the depths of `t1`, `t2`!

- To obtain fast code, we must rebalance.
- One can do so without affecting asymptotic cost.
- You learn more about this in 15-210.

Merge may not return a balanced tree!

The depth of `Merge(t1,t2)` can be the sum of the depths of `t1`, `t2`!

- To obtain fast code, we must rebalance.
- One can do so without affecting asymptotic cost.
- You learn more about this in 15-210.
- What we will do is assume that tree are balanced.

Merge may not return a balanced tree!

The depth of `Merge(t1,t2)` can be the sum of the depths of `t1`, `t2`!

- To obtain fast code, we must rebalance.
- One can do so without affecting asymptotic cost.
- You learn more about this in 15-210.
- What we will do is assume that tree are balanced.

```
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      rebalance (Ins (x, Merge(Msort left, Msort right)))
```

Merge may not return a balanced tree!

The depth of `Merge(t1,t2)` can be the sum of the depths of `t1`, `t2`!

- To obtain fast code, we must rebalance.
- One can do so without affecting asymptotic cost.
- You learn more about this in 15-210.
- What we will do is assume that tree are balanced.

```
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
       rebalance (Ins (x, Merge(Msort left, Msort right)))
```

Merge may not return a balanced tree!

The depth of `Merge(t1,t2)` can be the sum of the depths of `t1`, `t2`!

- To obtain fast code, we must rebalance.
- One can do so without affecting asymptotic cost.
- You learn more about this in 15-210.
- What we will do is assume that tree are balanced.

```
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
       rebalance (Ins (x, Merge(Msort left, Msort right)))
```

assume rebalnce
is called here

Let’s write the function split!

Let's write the function split!

```
(* SplitAt : int * tree -> tree * tree
   REQUIRES: t is sorted.
   ENSURES: SplitAt(x,t) returns a pair (t1,t2) of sorted
            trees such that:
            (a) t1 and t2 contain exactly the elements of t;
            (b) the elements of t1 are LESS or EQUAL to x;
            (c) the elements of t2 are GREATER or EQUAL to x.
*)
```

Let’s write the function split!

```
fun SplitAt(x: int, Empty: tree): tree * tree =
```

Let's write the function split!

```
fun SplitAt(x: int, Empty: tree): tree * tree = (Empty, Empty)
```

Let's write the function split!

```
fun SplitAt(x: int, Empty: tree): tree * tree = (Empty, Empty)
  | SplitAt(x, Node(left, y, right)) =
     case compare(x, y) of
```

Let’s write the function split!

```
fun SplitAt(x: int, Empty: tree): tree * tree = (Empty, Empty)
  | SplitAt(x, Node(left, y, right)) =
     case compare(x, y) of
           LESS => let
                      val (t1, t2) = SplitAt(x, left)
                   in

                   end
```

Let’s write the function split!

```
fun SplitAt(x: int, Empty: tree): tree * tree = (Empty, Empty)
  | SplitAt(x, Node(left, y, right)) =
     case compare(x, y) of
           LESS => let
                      val (t1, t2) = SplitAt(x, left)
                   in
                      (t1, Node(t2, y, right))
                   end
```

Let's write the function split!

```
fun SplitAt(x: int, Empty: tree): tree * tree = (Empty, Empty)
  | SplitAt(x, Node(left, y, right)) =
     case compare(x, y) of
          LESS => let
                     val (t1, t2) = SplitAt(x, left)
                  in
                     (t1, Node(t2, y, right))
                  end
        |  _   => let
                     val (t1, t2) = SplitAt(x, right)
                  in

                  end
```

Let's write the function split!

```
fun SplitAt(x: int, Empty: tree): tree * tree = (Empty, Empty)
  | SplitAt(x, Node(left, y, right)) =
     case compare(x, y) of
           LESS => let
                      val (t1, t2) = SplitAt(x, left)
                   in
                      (t1, Node(t2, y, right))
                   end
          |  _  => let
                      val (t1, t2) = SplitAt(x, right)
                   in
                      (Node(left, y, t1), t2)
                   end
```

Let’s determine the span of mergesort!

Let's determine the span of mergesort!

We are going to assume balanced trees (ie call of rebalance).

Let’s determine the span of mergesort!

- We are going to assume balanced trees (ie call of rebalance).
- We are going to focus on the depth of a tree.

Let's determine the span of mergesort!

- We are going to assume balanced trees (ie call of rebalance).
- We are going to focus on the depth of a tree.
  - Depth is $\log_2$ of the number of nodes in a balanced tree.

Let's determine the span of mergesort!

- We are going to assume balanced trees (ie call of rebalance).
- We are going to focus on the depth of a tree.
  - Depth is $\log_2$ of the number of nodes in a balanced tree.
- Both Merge and Msort affort parallelism.

Let's determine the span of mergesort!

- We are going to assume balanced trees (ie call of rebalance).
- We are going to focus on the depth of a tree.
  - Depth is $\log_2$ of the number of nodes in a balanced tree.
- Both Merge and Msort affort parallelism.
  - We will focus on Msort, refer to lecture notes otherwise.

Let's determine the span of mergesort!

- We are going to assume balanced trees (ie call of rebalance).
- We are going to focus on the depth of a tree.
  - Depth is $\log_2$ of the number of nodes in a balanced tree.
- Both Merge and Msort affort parallelism.
  - We will focus on Msort, refer to lecture notes otherwise.

We have: $S_{\texttt{Ins}}(d)$ is $O(d)$.
$S_{\texttt{SplitAt}}(d)$ is $O(d)$.
$S_{\texttt{Merge}}(d)$ is $O(d_1 d_2)$.

Let’s determine the span of mergesort!

Let’s determine the span of mergesort!

```
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      Ins (x, Merge(Msort left, Msort right))
```

Let's determine the span of mergesort!

```
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      Ins (x, Merge(Msort left, Msort right))
```

Span: $S_{Msort}(d)$ with d the depth of the tree.

Let's determine the span of mergesort!

```
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      Ins (x, Merge(Msort left, Msort right))
```

Span: $S_{\texttt{Msort}}(d)$ with d the depth of the tree.

Equations:

Let's determine the span of mergesort!

```
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      Ins (x, Merge(Msort left, Msort right))
```

Span: $S_{Msort}(d)$ with d the depth of the tree.

Equations:

$S_{Msort}(d) =$

Let's determine the span of mergesort!

```
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      Ins (x, Merge(Msort left, Msort right))
```

Span: $S_{Msort}(d)$ with d the depth of the tree.

Equations:

$S_{Msort}(d) = c + \max(S_{Msort}(d-1), S_{Msort}(d'))$

Let's determine the span of mergesort!

```
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      Ins (x, Merge(Msort left, Msort right))
```

Span: $S_{Msort}(d)$ with d the depth of the tree.

Equations:

$$S_{Msort}(d) = c + \max(S_{Msort}(d-1), S_{Msort}(d')) + S_{Merge}(d_1, d_2) + S_{Ins}(d_3)$$

Let's determine the span of mergesort!

```
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      Ins (x, Merge(Msort left, Msort right))
```

Span: $S_{Msort}(d)$ with d the depth of the tree.

Equations:

$$S_{Msort}(d) = c + \max(S_{Msort}(d-1), S_{Msort}(d'))$$
$$+ S_{Merge}(d_1, d_2) + S_{Ins}(d_3)$$

Here:

$d' \leq d-1$.

d_1, d_2 depths of the trees returned by recursive calls to Msort.

d_3 depth of the tree returned by Merge.

Let's determine the span of mergesort!

```
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      Ins (x, Merge(Msort left, Msort right))
```

Span: $S_{Msort}(d)$ with d the depth of the tree.

Equations:

$$S_{Msort}(d) = c + \max(S_{Msort}(d-1), S_{Msort}(d')) + S_{Merge}(d_1,d_2) + S_{Ins}(d_3)$$

Here:

$d' \leq d-1$.

d_1, d_2 depths of the trees returned by recursive calls to Msort.

d_3 depth of the tree returned by Merge.

Let's determine the span of mergesort!

```
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      Ins (x, Merge(Msort left, Msort right))
```

Span: $S_{Msort}(d)$ with d the depth of the tree.

Equations:

$$S_{Msort}(d) = c + \max(S_{Msort}(d-1), S_{Msort}(d')) + S_{Merge}(d_1, d_2) + S_{Ins}(d_3)$$

Here:

$d' \leq d-1$.

d_1, d_2 depths of the trees returned by recursive calls to Msort.

d_3 depth of the tree returned by Merge.

Let's determine the span of mergesort!

```
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      Ins (x, Merge(Msort left, Msort right))
```

Span: $S_{Msort}(d)$ with d the depth of the tree.

Equations:

$$S_{Msort}(d) = c + \max(S_{Msort}(d-1), S_{Msort}(d')) + S_{Merge}(d_1, d_2) + S_{Ins}(d_3)$$

Here:

$d' \leq d-1$.

d_1, d_2 depths of the trees returned by recursive calls to Msort.

d_3 depth of the tree returned by Merge.

Let's determine the span of mergesort!

```
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      Ins (x, Merge(Msort left, Msort right))
```

Span: $S_{\text{Msort}}(d)$ with d the depth of the tree.

Equations:

$$S_{\text{Msort}}(d) = c + \max(S_{\text{Msort}}(d-1), S_{\text{Msort}}(d')) + S_{\text{Merge}}(d_1, d_2) + S_{\text{Ins}}(d_3)$$

Here:

$d' \leq d-1$.

d_1, d_2 depths of the trees returned by recursive calls to Msort.

d_3 depth of the tree returned by Merge.

Let’s determine the span of mergesort!

```
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      Ins (x, Merge(Msort left, Msort right))
```

Span: $S_{Msort}(d)$ with d the depth of the tree.

Equations:

$$S_{Msort}(d) = c + \max(S_{Msort}(d-1), S_{Msort}(d'))$$
$$+ S_{Merge}(d_1, d_2) + S_{Ins}(d_3)$$

Here:

$d' \leq d-1$.

d_1, d_2 depths of the trees returned by recursive calls to Msort.

d_3 depth of the tree returned by Merge.

Let's determine the span of mergesort!

```
fun Msort (Empty : tree) : tree = Empty
  | Msort (Node(left, x, right)) =
      Ins (x, Merge(Msort left, Msort right))
```

Span: $S_{Msort}(d)$ with d the depth of the tree.

Equations:

$$S_{Msort}(d) = c + \max(S_{Msort}(d-1), S_{Msort}(d')) + S_{Merge}(d_1,d_2) + S_{Ins}(d_3)$$

Here:

$d' \leq d-1$.

d_1, d_2 depths of the trees returned by recursive calls to Msort.

d_3 depth of the tree returned by Merge.

If we rebalance as a final step in Msort, then:

$d_1 \leq d$, $d_2 \leq d$, and $d_3 \leq 2d$.

Let's determine the span of mergesort!

Span: $S_{\texttt{Msort}}(\texttt{d})$ with $\texttt{d}$ the depth of the tree.

Equations:

$$S_{\texttt{Msort}}(\texttt{d}) = c + \texttt{max}(S_{\texttt{Msort}}(\texttt{d}-1), S_{\texttt{Msort}}(\texttt{d'})) + S_{\texttt{Merge}}(\texttt{d}_1, \texttt{d}_2) + S_{\texttt{Ins}}(\texttt{d}_3)$$

If we rebalance as a final step in Msort, then:

$\texttt{d}_1 \leq \texttt{d}$, $\texttt{d}_2 \leq \texttt{d}$, and $\texttt{d}_3 \leq 2\texttt{d}$.

Let's determine the span of mergesort!

Span: $S_{Msort}(d)$ with d the depth of the tree.

Equations:

$$S_{Msort}(d) = c + \max(S_{Msort}(d-1), S_{Msort}(d')) + S_{Merge}(d_1, d_2) + S_{Ins}(d_3)$$

If we rebalance as a final step in Msort, then:

$d_1 \leq d$, $d_2 \leq d$, and $d_3 \leq 2d$.

Thus:

$$S_{Msort}(d) \leq c + S_{Msort}(d-1) + S_{Merge}(d,d) + S_{Ins}(2d)$$

Let's determine the span of mergesort!

Span: $S_{Msort}(d)$ with d the depth of the tree.

Equations:

$$S_{Msort}(d) = c + \max(S_{Msort}(d-1), S_{Msort}(d')) + S_{Merge}(d_1, d_2) + S_{Ins}(d_3)$$

If we rebalance as a final step in Msort, then:

$d_1 \leq d$, $d_2 \leq d$, and $d_3 \leq 2d$.

Thus:

$$\begin{aligned} S_{Msort}(d) &\leq c + S_{Msort}(d-1) + S_{Merge}(d,d) + S_{Ins}(2d) \\ &\leq c + S_{Msort}(d-1) + c' d^2 + c'' d \end{aligned}$$

Let's determine the span of mergesort!

Span: $S_{\texttt{Msort}}(\texttt{d})$ with $\texttt{d}$ the depth of the tree.

Equations:

$$S_{\texttt{Msort}}(\texttt{d}) = c + \texttt{max}(S_{\texttt{Msort}}(\texttt{d-1}), S_{\texttt{Msort}}(\texttt{d'})) + S_{\texttt{Merge}}(\texttt{d}_1,\texttt{d}_2) + S_{\texttt{Ins}}(\texttt{d}_3)$$

If we rebalance as a final step in Msort, then:

$\texttt{d}_1 \leq \texttt{d}$, $\texttt{d}_2 \leq \texttt{d}$, and $\texttt{d}_3 \leq 2\texttt{d}$.

Thus:

$$\begin{aligned} S_{\texttt{Msort}}(\texttt{d}) &\leq c + S_{\texttt{Msort}}(\texttt{d-1}) + S_{\texttt{Merge}}(\texttt{d,d}) + S_{\texttt{Ins}}(2\texttt{d}) \\ &\leq c + S_{\texttt{Msort}}(\texttt{d-1}) + c'\, \texttt{d}^2 + c''\, \texttt{d} \\ &\leq k\, \texttt{d}^2 + S_{\texttt{Msort}}(\texttt{d-1}) \end{aligned}$$

Let's determine the span of mergesort!

Span: $S_{Msort}(d)$ with d the depth of the tree.

Equations:

$$S_{Msort}(d) = c + \max(S_{Msort}(d-1), S_{Msort}(d')) + S_{Merge}(d_1, d_2) + S_{Ins}(d_3)$$

If we rebalance as a final step in Msort, then:

$d_1 \leq d$, $d_2 \leq d$, and $d_3 \leq 2d$.

Thus:

$$\begin{aligned} S_{Msort}(d) &\leq c + S_{Msort}(d-1) + S_{Merge}(d,d) + S_{Ins}(2d) \\ &\leq c + S_{Msort}(d-1) + c'\, d^2 + c''\, d \\ &\leq k\, d^2 + S_{Msort}(d-1) \end{aligned}$$

Consequently: $S_{Msort}(d)$ is $O(d^3)$, ie $O((\log n)^3)$.

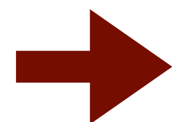
Sorting overview

Sorting overview

	list isort	list msort	tree msort
Work:	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Span:	$O(n^2)$	$O(n)$	$O((\log n)^3)$

Sorting overview

	list isort	list msort	tree msort
Work:	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Span:	$O(n^2)$	$O(n)$	$O((\log n)^3)$



In 15-210, span of tree msort can be reduced to $O((\log n)^3)$!