Programming in Haskell: Lecture 22

S P Suresh

October 30, 2019
A binary tree data structure is defined as follows:
Binary trees

- A binary tree data structure is defined as follows:
- The empty tree is a binary tree
Binary trees

- A binary tree data structure is defined as follows:
- The empty tree is a binary tree
- A node containing an element with left and right subtrees is a binary tree
Binary trees

• A binary tree data structure is defined as follows:
• The empty tree is a binary tree
• A node containing an element with left and right subtrees is a binary tree
• Type constructor `BTree`

```
data BTree a = Nil
             | Node (BTree a) a (BTree a)
```
Binary trees

- A binary tree data structure is defined as follows:
- The empty tree is a binary tree
- A node containing an element with left and right subtrees is a binary tree
- Type constructor `BTree`

```haskell
data BTree a = Nil
  | Node (BTree a) a (BTree a)
```

- Two value constructors:

```haskell
Nil :: BTree a
Node :: BTree a -> a -> BTree a -> BTree a
```
Binary trees

Node (Node Nil 4 Nil) 6
   (Node (Node Nil 2 Nil) 3
      (Node Nil 5 Nil))
• Yet another binary tree
Binary trees

- Yet another binary tree

- Corresponding BTree

```
(Node Nil 1 (Node Nil 2 Nil))
3
(Node (Node Nil 4 Nil) 5 Nil)
```
Functions on binary trees

• Number of nodes in a tree

\[
\text{size} :: \text{BTree } a \rightarrow \text{Int}
\]
\[
\text{size Nil} = 0
\]
\[
\text{size (Node tl x tr)} = 1 + \text{size tl} + \text{size tr}
\]
Functions on binary trees

• Number of nodes in a tree

\[
\text{size} :: \text{BTree } a \rightarrow \text{Int} \\
\text{size } \text{Nil} = 0 \\
\text{size } (\text{Node } tl \times tr) = 1 + \text{size } tl + \text{size } tr
\]

• **Height**: number of nodes on longest path from root

\[
\text{height} :: \text{BTree } a \rightarrow \text{Int} \\
\text{height } \text{Nil} = 0 \\
\text{height } (\text{Node } tl \times tr) = 1 + \max (\text{height } tl) (\text{height } tr)
\]
Creating a binary tree

- Create a binary tree from a list
Creating a binary tree

- Create a binary tree from a list
- As balanced as possible
Creating a binary tree

• Create a binary tree from a list

• As balanced as possible

• **Strategy:**
Creating a binary tree

• Create a binary tree from a list
• As balanced as possible
• Strategy:
  • Split the list in two halves
Creating a binary tree

• Create a binary tree from a list
• As balanced as possible
• **Strategy:**
  • Split the list in two halves
  • Recursively create a binary tree from each half
Creating a binary tree

• Create a binary tree from a list
• As balanced as possible
• **Strategy:**
  • Split the list in two halves
  • Recursively create a binary tree from each half
  • Join them together
Creating a binary tree

• Creating a balanced tree from a list

```haskell
createTree :: [a] -> BTree a
createTree [] = Nil
createTree xs = Node
  (createTree front) x (createTree back)
where
  n = length xs
  (front, x:back) = splitAt (n `div` 2) xs
```
Showing a binary tree

- To be able to show a binary tree, we need to derive a `Show` instance

```haskell
data BTree a = Nil | Node (BTree a) a (BTree a)
  deriving Show

createTree [0..14] =
  Node (Node (Node (Node Nil 0 Nil) 1 (Node Nil 2 Nil)))
    3 (Node (Node Nil 4 Nil) 5 (Node Nil 6 Nil)))
  7 (Node (Node Nil 8 Nil) 9 (Node Nil 10 Nil))
  11 (Node (Node Nil 12 Nil) 13 (Node Nil 14 Nil)))
```
Showing a binary tree

• To be able to show a binary tree, we need to derive a `Show` instance

```haskell
data BTree a = Nil | Node (BTree a) a (BTree a)
    deriving Show

createTree [0..14] =
    Node (Node (Node (Node Nil 0 Nil) 1 (Node Nil 2 Nil)))
    3 (Node (Node Nil 4 Nil) 5 (Node Nil 6 Nil)))
    7 (Node (Node Nil 8 Nil) 9 (Node Nil 10 Nil))
    11 (Node (Node Nil 12 Nil) 13 (Node Nil 14 Nil)))
```

• Not particularly readable!
A custom show

```
4
  2
  1
  0
  *
  3
  6
  5
  7
```
A custom show
A custom show

instance Show a => Show (BTree a) where
    show = intercalate "\n" (draw t)
draw :: Show a => BTree a -> [String]
draw Nil = ["*"]
draw (Node Nil x Nil) = [show x]
draw (Node tl x tr) = [show x] ++
    shift (draw tl) ++
    shift (draw tr)

where shift = zipWith (++) (repeat " ")
A custom show

```
0
  |   +-1
  |   |   +--*
  |   |   |   `-2
  |   |   `-3
  |   `-4
  `-5
```
A custom show

instance Show a ⇒ Show (BTree a) where
  show = intercalate "\n" (draw2 t)
draw2 :: Show a ⇒ BTree a → [String]
draw2 Nil = ["*"]
draw2 (Node Nil x Nil) = [show x]
draw2 (Node tl x tr) = [show x] ++
  shiftl (draw2 tl) ++
  shiftr (draw2 tr)
  where shiftl = zipWith (++) ("+-":repeat "| ")
  shiftr = zipWith (++) ("\-":repeat " ")
Creating a binary tree

- Creating a balanced tree from a list

```haskell
createTree :: [a] -> BTree a
createTree [] = Nil
createTree xs = Node
  (createTree front) x (createTree back)
where
  n = length xs
  (front, x:back) = splitAt (n `div` 2) xs
```
Creating a binary tree

- `length` and `splitAt` take linear time
Creating a binary tree

- **length** and **splitAt** take linear time
- \( T(n) = 2T(n/2) + O(n) \) and hence \( T(n) = O(n \log n) \)
Creating a binary tree

- **length** and **splitAt** take linear time
- \[ T(n) = 2T(n/2) + O(n) \] and hence \[ T(n) = O(n \log n) \]
- Can we improve?

```plaintext
can we improve?
can we improve? can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
can we improve?
Creating a binary tree

- **length** and **splitAt** take linear time
- \( T(n) = 2T(n/2) + O(n) \) and hence \( T(n) = O(n \log n) \)
- Can we improve?
  - Culprit is the repeated use of **length** and **splitAt**
Creating a binary tree

- **length** and **splitAt** take linear time
- \( T(n) = 2T(n/2) + O(n) \) and hence \( T(n) = O(n \log n) \)
- Can we improve?
  - Culprit is the repeated use of **length** and **splitAt**
  - Can we avoid that?
Creating a binary tree

- **length** and **splitAt** take linear time
- \( T(n) = 2T(n/2) + O(n) \) and hence \( T(n) = O(n \log n) \)
- Can we improve?
  - Culprit is the repeated use of **length** and **splitAt**
  - Can we avoid that?
- Consider the following function:

  \[
  \text{go :: Int} \rightarrow \text{[a]} \rightarrow \text{(BTree a, [a])}
  \]
Creating a binary tree

- **length** and **splitAt** take linear time

- \( T(n) = 2T(n/2) + O(n) \) and hence \( T(n) = O(n \log n) \)

- Can we improve?
  - Culprit is the repeated use of **length** and **splitAt**
  - Can we avoid that?

- Consider the following function:

  \[
  \text{go} :: \text{Int} \to [a] \to (\text{BTree } a, [a])
  \]

- \( \text{go } n \text{ } l = (\text{createTree } (\text{take } n \text{ } l), \text{drop } n \text{ } l) \)
Creating a binary tree

- **length** and **splitAt** take linear time
- \( T(n) = 2T(n/2) + O(n) \) and hence \( T(n) = O(n \log n) \)
- Can we improve?
  - Culprit is the repeated use of **length** and **splitAt**
  - Can we avoid that?

- Consider the following function:

  \[
  \text{go} :: \text{Int} \rightarrow [a] \rightarrow (\text{BTree } a, [a])
  \]

- \( \text{go } n \ l = \) (createTree (take \( n \) \ l), drop \( n \) \ l)
- Can we do this efficiently?
Creating a binary tree

• Creating a tree in linear time

```haskell
createTree :: [a] -> BTree a
createTree l = fst (go (length l) l)

where

  go :: Int -> [a] -> (BTree a, [a])
  go 0 xs = (Nil, xs)
  go n xs = (Node tl y tr, zs)
    where m = n `div` 2
          (tl, y:ys) = go m xs
          (tr, zs) = go (n-m-1) ys
```

Suresh
PRGH 2019: Lecture 22
October 30, 2019
Creating a binary tree

• Creating a tree in linear time

```haskell
createTree :: [a] -> BTree a
createTree l = fst (go (length l) l)

  where
    go :: Int -> [a] -> (BTree a, [a])
    go 0 xs = (Nil, xs)
    go n xs = (Node tl y tr, zs)
      where m = n `div` 2
          (tl, y:ys) = go m xs
          (tr, zs) = go (n-m-1) ys
```

• \( T(n) = 2T(n/2) + c \) and hence \( T(n) = (2n - 1)c \)
The Set ADT

- Maintain a collection of distinct elements and support the following operations
The Set ADT

- Maintain a collection of distinct elements and support the following operations
  - `insertInto` – insert a given value into the set
The Set ADT

- Maintain a collection of distinct elements and support the following operations
  - `insertInto` – insert a given value into the set
  - `deleteFrom` – delete a given value from the set
The Set ADT

- Maintain a collection of distinct elements and support the following operations
  - `insertInto` – insert a given value into the set
  - `deleteFrom` – delete a given value from the set
  - `search` – check whether a given value is an element of the set
The Set ADT

- Maintain a collection of distinct elements and support the following operations
  - `insertInto` – insert a given value into the set
  - `deleteFrom` – delete a given value from the set
  - `search` – check whether a given value is an element of the set

- Straightforward implementation

```haskell
module Set(Set, insertInto, deleteFrom, search) where

data Set a = Set [a]
```
The Set ADT

```haskell
data Set a = Set [a]

search :: Eq a => a -> Set a -> Bool
search x (Set xs) = x `elem` xs

insertInto :: Eq a => a -> Set a -> Set a
insertInto x (Set xs) = if x `elem` xs then Set xs
                          else Set (x:xs)

deleteFrom :: Eq a => a -> Set a -> Set a
deleteFrom x (Set xs) = Set (filter (/=x) xs)
```
Set: complexity

- **search** takes $O(n)$ time
Set: complexity

- **search** takes $O(n)$ time
- **insertInto** takes $O(n)$ time
Set: complexity

- search takes $O(n)$ time
- insertInto takes $O(n)$ time
- deleteFrom takes $O(n)$ time
Set: complexity

- **search** takes $O(n)$ time
- **insertInto** takes $O(n)$ time
- **deleteFrom** takes $O(n)$ time
- A sequence of $n$ operations takes $O(n^2)$ time
Set: complexity

• search takes $O(n)$ time
• insertInto takes $O(n)$ time
• deleteFrom takes $O(n)$ time
• A sequence of $n$ operations takes $O(n^2)$ time
• We can do better if the elements admit an order
Binary search trees

- A **binary search tree** is another way of implementing the Set ADT.
Binary search trees

• A binary search tree is another way of implementing the Set ADT
• A binary search tree is a binary tree
Binary search trees

• A **binary search tree** is another way of implementing the **Set** ADT
• A binary search tree is a binary tree
• Stores values of type \(a\) such that \(\text{Ord } a\)
Binary search trees

- A binary search tree is another way of implementing the Set ADT
- A binary search tree is a binary tree
- Stores values of type $a$ such that $\text{Ord } a$
- In a binary search tree:
Binary search trees

- A **binary search tree** is another way of implementing the Set ADT
- A binary search tree is a binary tree
- Stores values of type \( a \) such that \( \text{Ord} \ a \)
- In a binary search tree:
  - values in the left subtree are smaller than the root
Binary search trees

- A **binary search tree** is another way of implementing the **Set** ADT
- A binary search tree is a binary tree
- Stores values of type $a$ such that $\text{Ord } a$
- In a binary search tree:
  - values in the left subtree are smaller than the root
  - values in the right subtree are larger than the root
Binary search trees

• The binary search tree

Node $tl \times tr$
Binary search trees

- The binary search tree

Node tl x tr

- Pictorially ...

```
  x
 /   \
<  x  >  x
 tl   tr
```
Binary search trees

• Examples

• Non-examples
• Binary search trees in Haskell:

```haskell
data BST a = Nil | Node (BST a) a (BST a)
deriving (Eq, Ord)
```
• Binary search trees in Haskell:

```haskell
data BST a = Nil | Node (BST a) a (BST a)
deriving (Eq, Ord)
```

• The empty tree:

```haskell
emptyBST :: BST a
emptyBST = Nil
```
• Binary search trees in Haskell:

```haskell
data BST a = Nil | Node (BST a) a (BST a)
  deriving (Eq, Ord)
```

• The empty tree:

```haskell
emptyBST :: BST a
emptyBST = Nil
```

• Is a tree empty?

```haskell
isEmpty :: BST a -> Bool
isEmpty Nil = True
isEmpty _   = False
```
Is it a search tree?

- Just naming it **BST** does not make it a binary search tree

```haskell
isBST :: Ord a => BST a -> Bool
isBST Nil = True
isBST (Node tl x tr) = isBST tl && isBST tr &&
                      (is Empty tl || max tl < x) &&
                      (is Empty tr || x < min tr)
```
Minimum in a tree

- \texttt{mint} gives the minimum value in a non-empty tree:

\begin{verbatim}
mint :: Ord a => BST a -> a
mint (Node tl x tr) = min x (min y z)
  where y = if isEmpty tl then x else mint tl
       z = if isEmpty tr then x else mint tr
\end{verbatim}
Minimum in a tree

- \texttt{mint} gives the minimum value in a non-empty tree:

\begin{verbatim}
mint :: Ord a => BST a -> a
mint (Node tl x tr) = min x (min y z)
  where y = if isEmpty tl then x else mint tl
       z = if isEmpty tr then x else mint tr
\end{verbatim}

- \texttt{maxt} is similar (uses \texttt{max} instead of \texttt{min})
Searching in a tree

- Searching for value $v$ in a search tree
Searching in a tree

- Searching for value $v$ in a search tree
- If the tree is empty, report $\text{False}$
Searching in a tree

• Searching for value $v$ in a search tree
• If the tree is empty, report False
• If the tree is nonempty
Searching in a tree

• Searching for value $v$ in a search tree
• If the tree is empty, report \texttt{False}
• If the tree is nonempty
  • If $v$ is the value at the root, report \texttt{True}
Searching in a tree

- Searching for value $v$ in a search tree
- If the tree is empty, report False
- If the tree is nonempty
  - If $v$ is the value at the root, report True
  - If $v$ is smaller than the value at the root, search in left subtree
Searching in a tree

- Searching for value $v$ in a search tree
- If the tree is empty, report $\text{False}$
- If the tree is nonempty
  - If $v$ is the value at the root, report $\text{True}$
  - If $v$ is smaller than the value at the root, search in left subtree
  - If $v$ is larger than the value at the root, search in right subtree
Searching in a tree

- Haskell code transcribe the search strategy:

```haskell
searchBST :: Ord a => a -> BST a -> Bool
searchBST v Nil       = False
searchBST v (Node tl x tr)
    | v == x       = True
    | v < x        = searchBST v tl
    | v > x        = searchBST v tr
```
Searching in a tree

- Haskell code transcribe the search strategy:

```haskell
searchBST :: Ord a => a -> BST a -> Bool
searchBST v Nil = False
searchBST v (Node tl x tr)
  | v == x    = True
  | v < x     = searchBST v tl
  | v > x     = searchBST v tr
```

- **Worst case running time**: Height of the tree
Inserting in a tree

• Inserting an element into a tree follows a similar strategy:

```
insertBST :: Ord a => a -> BST a -> BST a
insertBST v Nil = Node Nil v Nil
insertBST v t@(Node tl x tr)
  | v < x    = Node (insertBST v tl) x tr
  | v > x    = Node tl x (insertBST v tr)
  | v == x   = t
```
Inserting in a tree

• Inserting an element into a tree follows a similar strategy:

\[
\begin{align*}
\text{insertBST} :: \text{Ord } a \Rightarrow a \to \text{BST } a \to \text{BST } a \\
\text{insertBST } v \text{ Nil } &= \text{Node Nil } v \text{ Nil} \\
\text{insertBST } v \text{ t@}(\text{Node } tl \ x \ tr) &= \\
| v < x &= \text{Node } (\text{insertBST } v \ tl) \ x \ tr \\
| v > x &= \text{Node } tl \ x \ (\text{insertBST } v \ tr) \\
| v == x &= t
\end{align*}
\]

• **Worst case running time**: Height of the tree
Inserting in a tree

• Inserting an element into a tree follows a similar strategy:

```haskell
insertBST :: Ord a => a -> BST a -> BST a
insertBST v Nil = Node Nil v Nil
insertBST v t@(Node tl x tr)
  | v < x     = Node (insertBST v tl) x tr
  | v > x     = Node tl x (insertBST v tr)
  | v == x    = t
```

• **Worst case running time**: Height of the tree

• `@` allows one to refer to a data value by a name
Deleting from a tree

- We first tackle a simpler problem
Deleting from a tree

- We first tackle a simpler problem
  - Delete the maximum value
Deleting from a tree

• We first tackle a simpler problem
  • Delete the maximum value
  • and return the value as well as the modified tree
Deleting from a tree

- We first tackle a simpler problem
  - Delete the maximum value
  - and return the value as well as the modified tree
- Maximum is the rightmost node:

```
deleteMax :: Ord a => BST a -> (a, BST a)
deleteMax (Node tl x Nil) = (x, tl)
deleteMax (Node tl x tr) = let (y, ty) = deleteMax tr
                           in (y, Node tl x ty)
```
Deleting from a tree

- We first tackle a simpler problem
  - Delete the maximum value
  - and return the value as well as the modified tree
- Maximum is the rightmost node:

  \[
  \text{deleteMax} :: \text{Ord } a \Rightarrow \text{BST } a \rightarrow (a, \text{BST } a)
  \]

  \[
  \text{deleteMax} (\text{Node } tl \times x \times \text{Nil}) = (x, tl)
  \]

  \[
  \text{deleteMax} (\text{Node } tl \times x \times \text{tr}) = \text{let } \langle y, ty \rangle = \text{deleteMax } tr \text{ in } \langle y, \text{Node } tl \times x \times ty \rangle
  \]

- **Worst case running time**: Height of the tree
Deleting from a tree

- Deleting a value $v$ from a search tree

If the tree is empty, nothing to do.

If the tree is non-empty,

- If $v$ is smaller than the value at the root, delete from the left subtree.
- If $v$ is larger than the value at the root, delete from the right subtree.
- If $v$ is equal to the value at the root, remove the root and
  - If the left subtree is empty, just slide the right subtree up one level.
  - If the left subtree is non-empty, replace the root by the maximum element of the left subtree.

Search tree property is preserved.
Deleting from a tree

- Deleting a value $v$ from a search tree
- If the tree is empty, nothing to do
Deleting from a tree

- Deleting a value $v$ from a search tree
- If the tree is empty, nothing to do
- If the tree is nonempty
Deleting from a tree

- Deleting a value $v$ from a search tree
- If the tree is empty, nothing to do
- If the tree is nonempty
  - If $v$ is smaller than the value at the root, delete from left subtree
Deleting from a tree

- Deleting a value \( v \) from a search tree
- If the tree is empty, nothing to do
- If the tree is nonempty
  - If \( v \) is smaller than the value at the root, delete from left subtree
  - If \( v \) is larger than the value at the root, delete from right subtree
  - If \( v \) is equal to the value at the root, remove root and
    - If left subtree is empty, just slide right subtree up one level
    - If left subtree is non-empty, replace root by maximum element of left subtree
  - Search tree property is preserved
Deleting from a tree

• Deleting a value $v$ from a search tree
• If the tree is empty, nothing to do
• If the tree is nonempty
  • If $v$ is smaller than the value at the root, delete from left subtree
  • If $v$ is larger than the value at the root, delete from right subtree
  • If $v$ is equal to the value at the root, remove root and
Deleting from a tree

• Deleting a value \( v \) from a search tree
• If the tree is empty, nothing to do
• If the tree is nonempty
  • If \( v \) is smaller than the value at the root, delete from left subtree
  • If \( v \) is larger than the value at the root, delete from right subtree
  • If \( v \) is equal to the value at the root, remove root and
    • If left subtree is empty, just slide the right subtree up a level

Search tree property is preserved
Deleting from a tree

- Deleting a value $v$ from a search tree
- If the tree is empty, nothing to do
- If the tree is nonempty
  - If $v$ is smaller than the value at the root, delete from left subtree
  - If $v$ is larger than the value at the root, delete from right subtree
  - If $v$ is equal to the value at the root, remove root and
    - If left subtree is empty, just slide the right subtree up a level
    - If left subtree is non-empty, replace root by maximum element of left subtree

Search tree property is preserved
Deleting from a tree

- Deleting a value $v$ from a search tree
- If the tree is empty, nothing to do
- If the tree is nonempty
  - If $v$ is smaller than the value at the root, delete from left subtree
  - If $v$ is larger than the value at the root, delete from right subtree
  - If $v$ is equal to the value at the root, remove root and
    - If left subtree is empty, just slide the right subtree up a level
    - If left subtree is non-empty, replace root by maximum element of left subtree
  - Search tree property is preserved
Deleting from a tree

• Code for delete:

```haskell
deleteBST :: Ord a => a -> BST a -> BST a
deleteBST v Nil = Nil
deleteBST v (Node tl x tr)
  | v < x       = Node (deleteBST v tl) x tr
  | v > x       = Node tl x (deleteBST v tr)
  | v == x      = if isEmpty tl then tr else Node ty y tr
where (y, ty) = deleteMax tl
```
Deleting from a tree

- Code for delete:

```haskell
deleteBST :: Ord a => a -> BST a -> BST a
deleteBST v Nil = Nil
deleteBST v (Node tl x tr)
  | v < x = Node (deleteBST v tl) x tr
  | v > x = Node tl x (deleteBST v tr)
  | v == x = if isEmpty tl then tr else Node ty y tr

where (y, ty) = deleteMax tl
```

- **Worst case running time:** Height of the tree