Programming in Haskell
Aug-Nov 2015

LECTURE 17

OCTOBER 15, 2015

S P Suresh
Chennai Mathematical Institute

The Set data structure

- Maintain a collection of distinct elements and support the following operations
  - `insert`: insert a given value into the set
  - `delete`: delete a given value from the set
  - `search`: check whether a given value is an element of the set
- `data Set a = Set [a]`

The Set data structure

- `data Set a = Set [a]`

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

- ```
  insert :: Eq a => a -> Set a -> Set a
  insert x (Set y)
      | elem x y  = Set y
      | otherwise = Set (x:y)
  ```

- ```
  delete :: Eq a => a -> Set a -> Set a
  delete x (Set y) = Set (filter (/=x) y)
  ```
  ```

Complexity of Set

- `search` takes $O(N)$ time
- `insert` takes $O(N)$ time
- `delete` takes $O(N)$ time
- A sequence of N operations takes $O(N^2)$ time
- We can do better if the elements are ordered

Binary search tree

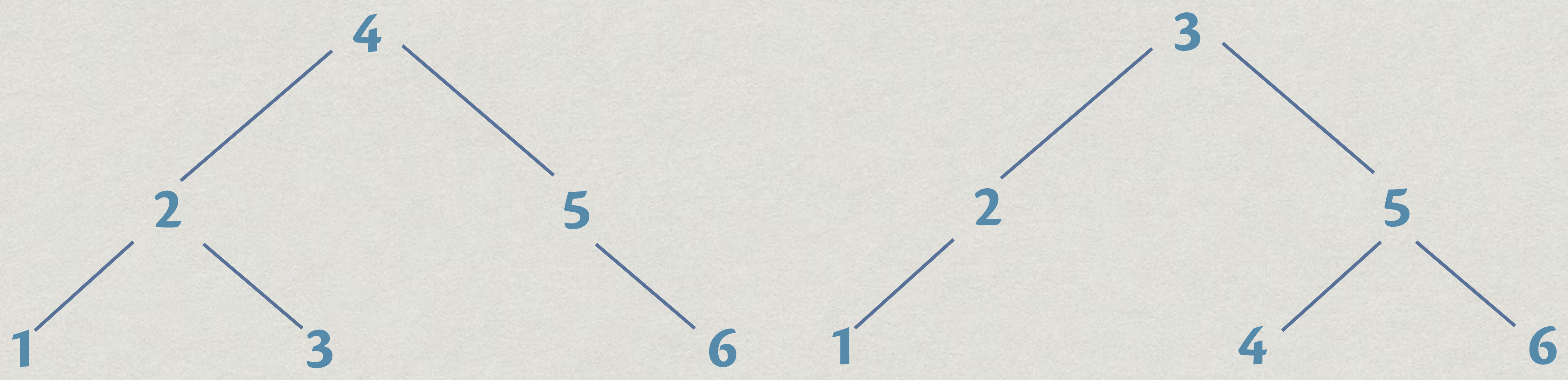
- A **binary search tree** is another way of implementing the **Set** data structure
- A binary search tree is a binary tree
- Stores values of type a, where a belongs to the typeclass `Ord`

Binary search tree

- In a binary search tree
  - Values in the left subtree are smaller than the current node
  - Values in the right subtree are larger than the current node

Examples

Two search trees with elements [1..6]



Binary search tree

- `data STree a = Nil | Node (STree a) a (STree a)`
  `deriving (Eq, Ord, Show)`
- Just calling it an `STree` does not make it a search tree

Is it a search tree?

* ```
  isstree :: Ord a => STree a -> Bool
  isstree Nil = True
  isstree (Node t1 x t2) = isstree t1 &&
                           isstree t2 &&
           (t1 == Nil || maxt t1 < x) &&
           (t2 == Nil || x < mint t2)
  ```
  ```

Maximum value in a tree

- ```
  maxt :: Ord a => STree a -> a
  -- Assume that the input tree is non-Nil
  maxt (Node t1 x t2) = max x (max y z)
    where
      y = if (t1 == Nil) then x
          else maxt t1
      z = if (t2 == Nil) then x
          else maxt t2
  ```
  ```

Minimum value in a tree

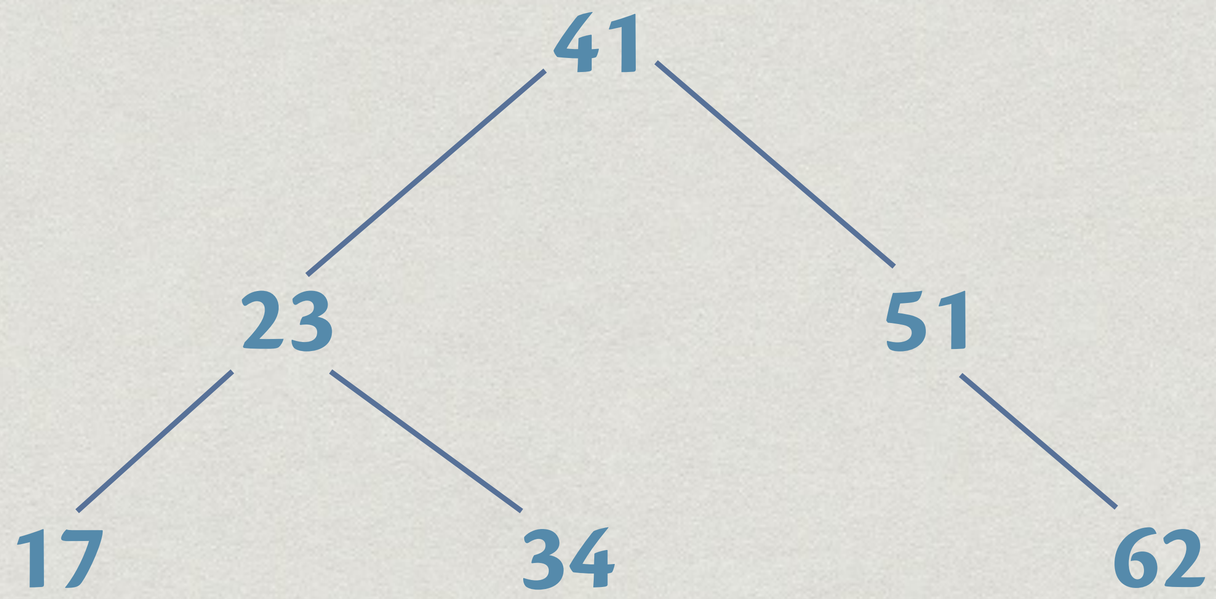
- ```
  mint :: Ord a => STree a -> a
  -- Assume that the input tree is non-Nil
  mint (Node t1 x t2) = min x (min y z)
    where
      y = if (t1 == Nil) then x
          else min t1
      z = if (t2 == Nil) then x
          else min t2
  ```
  ```

Searching in a tree

- Searching for **34**

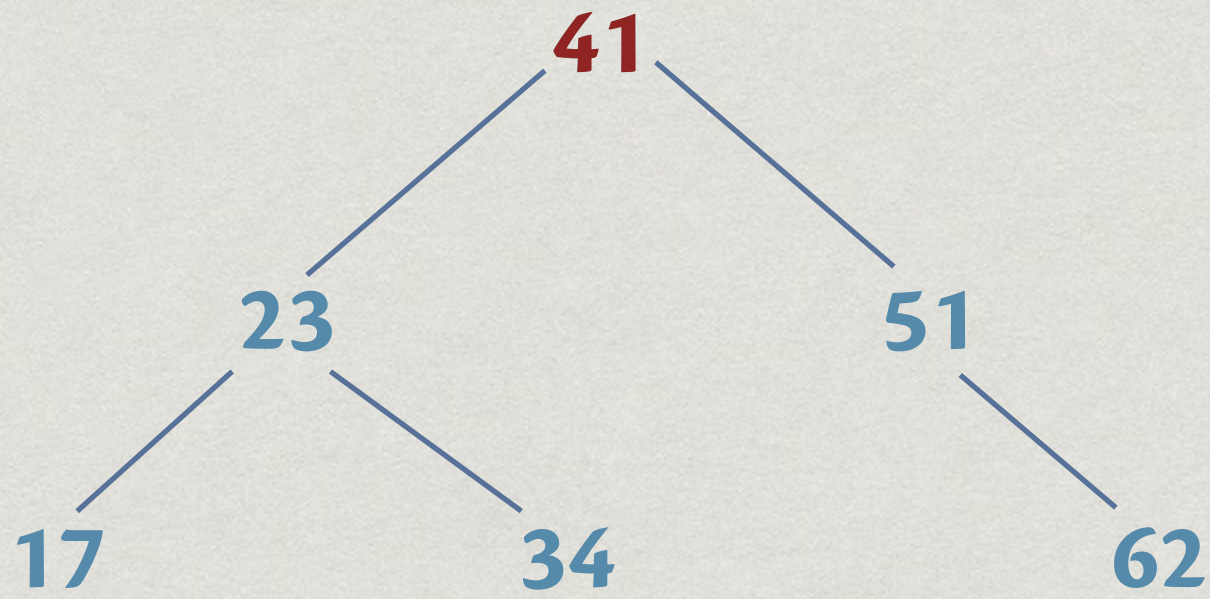
Searching in a tree

- Searching for **34**



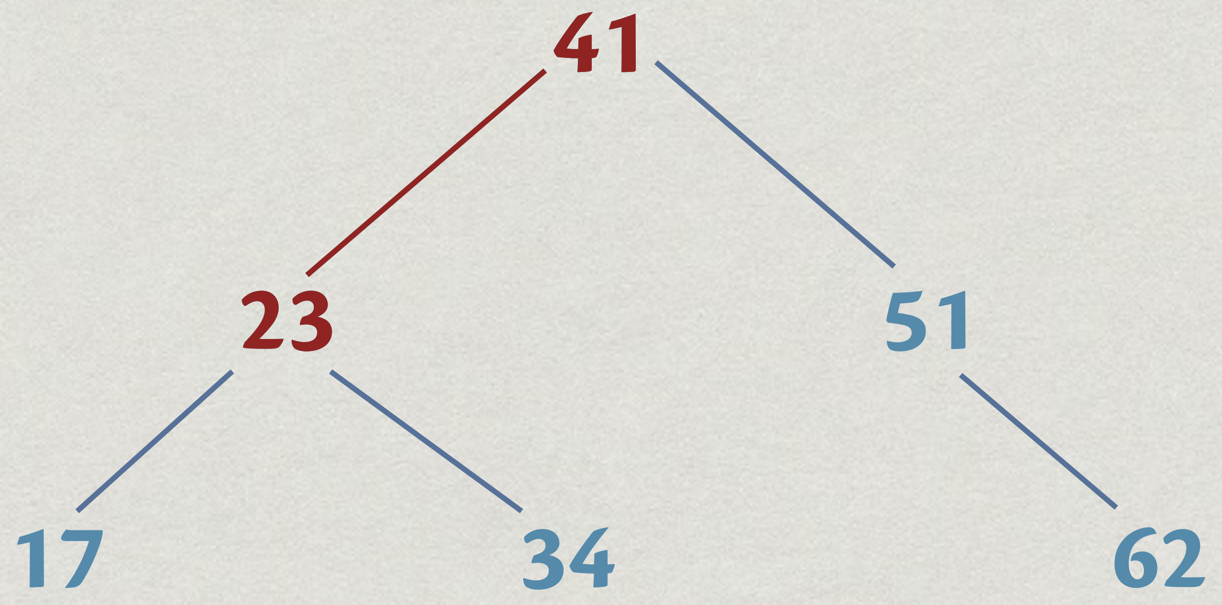
Searching in a tree

- Searching for **34**



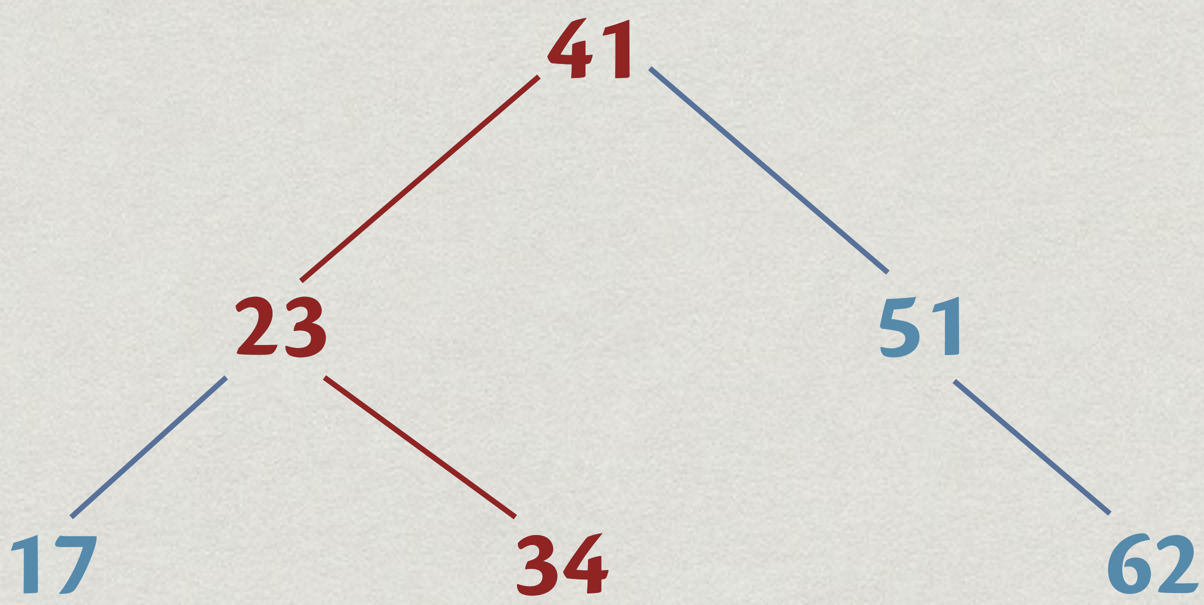
Searching in a tree

- Searching for **34**



Searching in a tree

- Searching for **34**

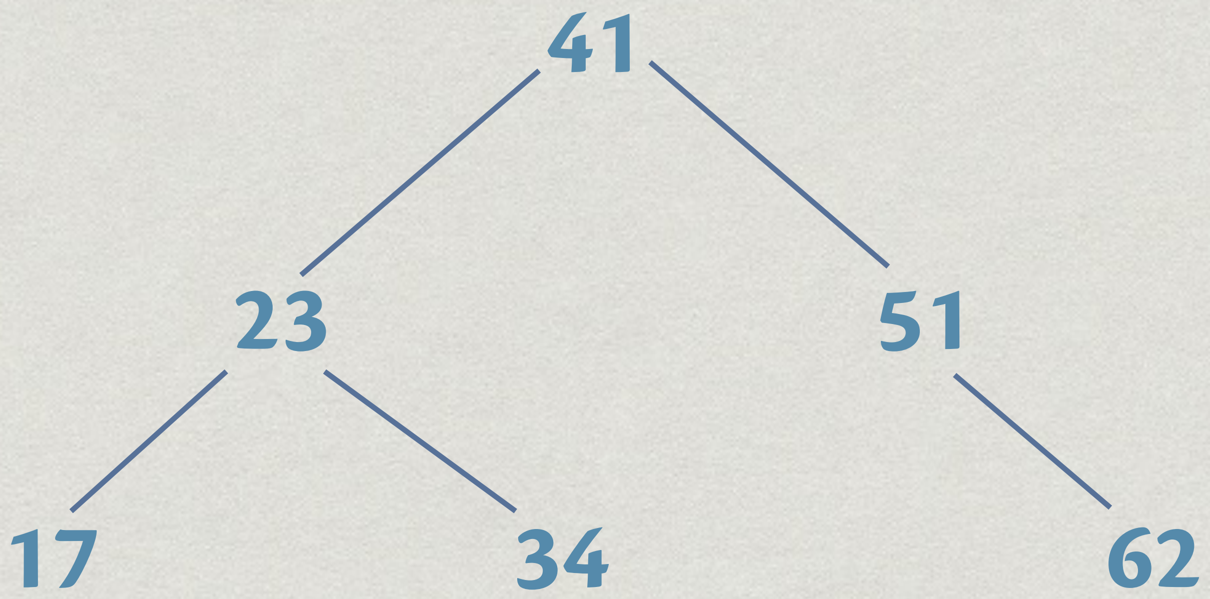


Searching in a tree

- Searching for **49**

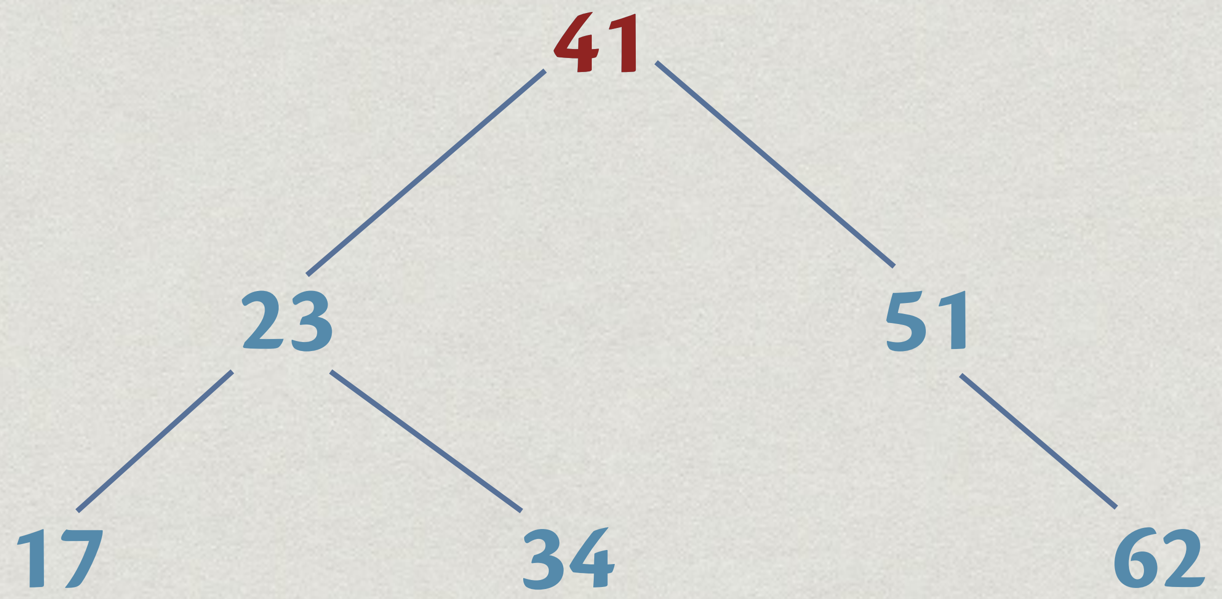
Searching in a tree

- Searching for **49**



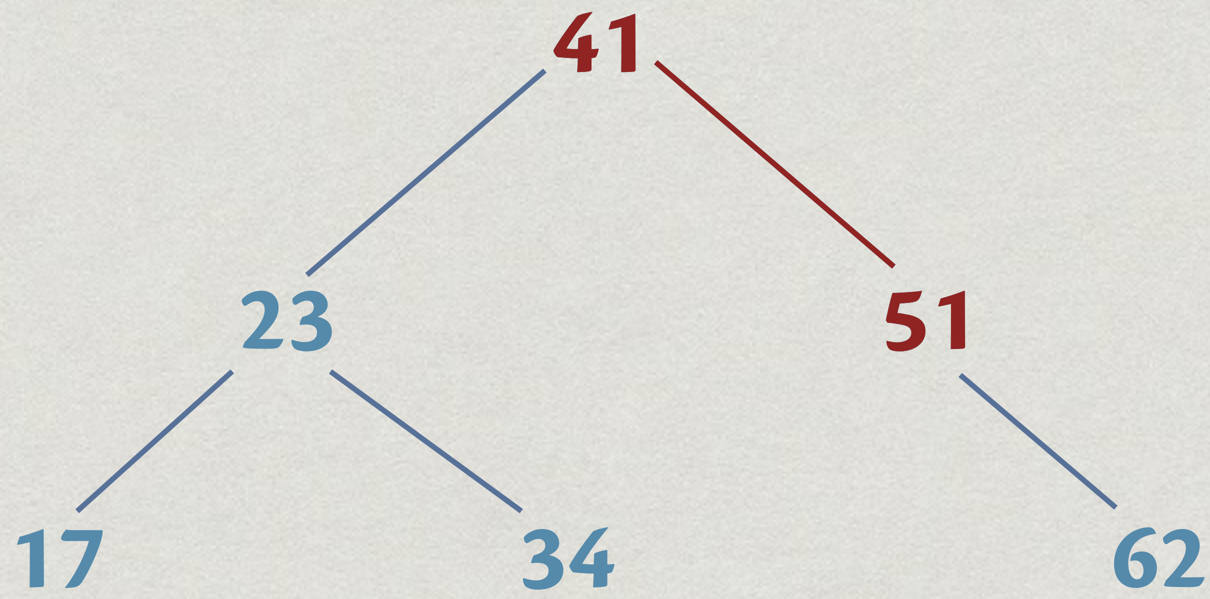
Searching in a tree

- Searching for **49**



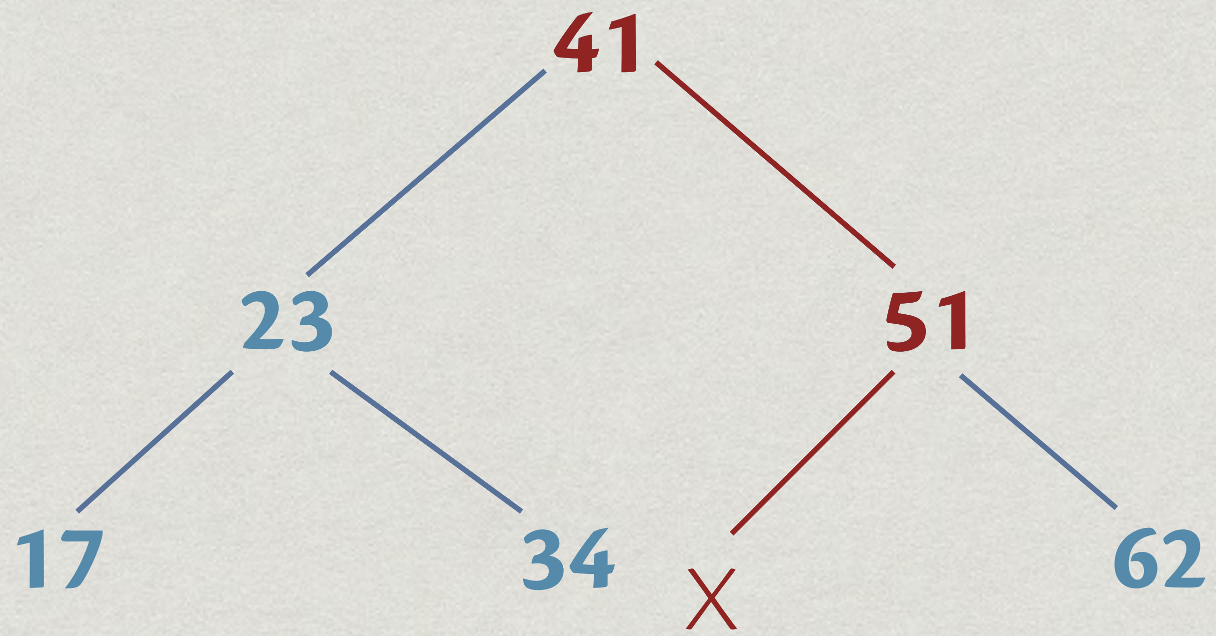
Searching in a tree

- Searching for **49**



Searching in a tree

- Searching for **49**



Searching in a tree

- Searching for value **v** in a search tree
- If the tree is empty, report **No**
- If the tree is nonempty
  - If **v** is the value at the root, report **Yes**
  - If **v** is smaller than the value at the root, search in left subtree (which could be empty)
  - If **v** is larger than the value at the root, search in right subtree (which could be empty)

Searching in a tree

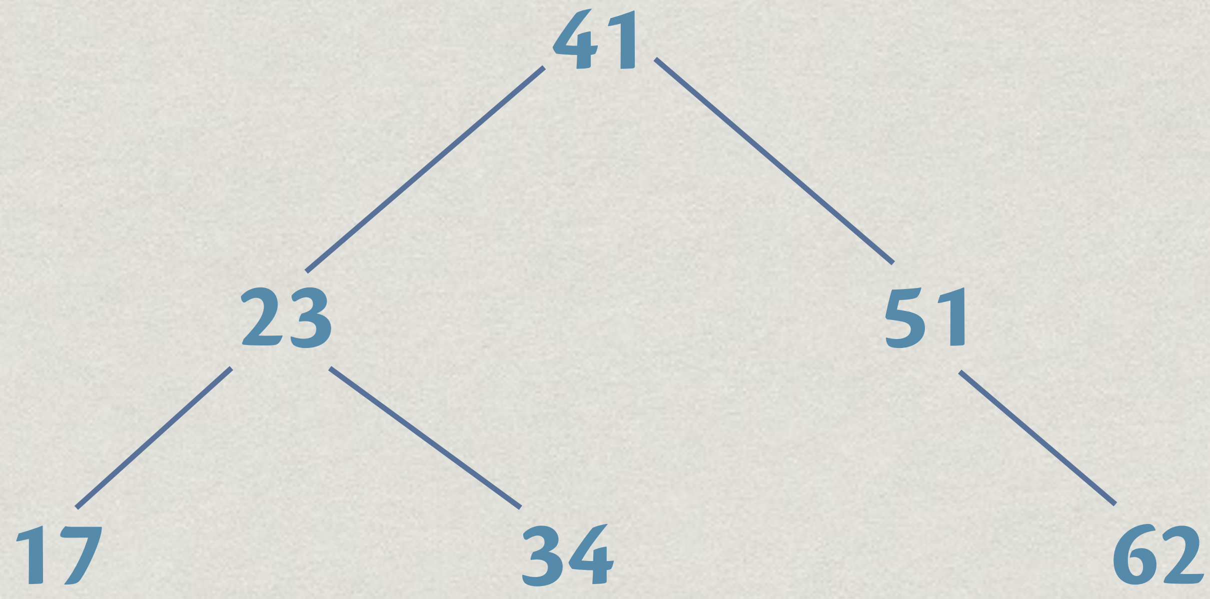
- ```
  search :: Ord a => STree a -> a -> Bool
  search Nil v                         = False
  search (Node tl x tr) v
   | x == v                            = True
   | v < x                             = search tl v
   | otherwise                         = search tr v
  ```
- Worst case: running time proportional to length of the longest path from root to a leaf (**height**)
  ```

Inserting in a tree

- Inserting **33** and **48**

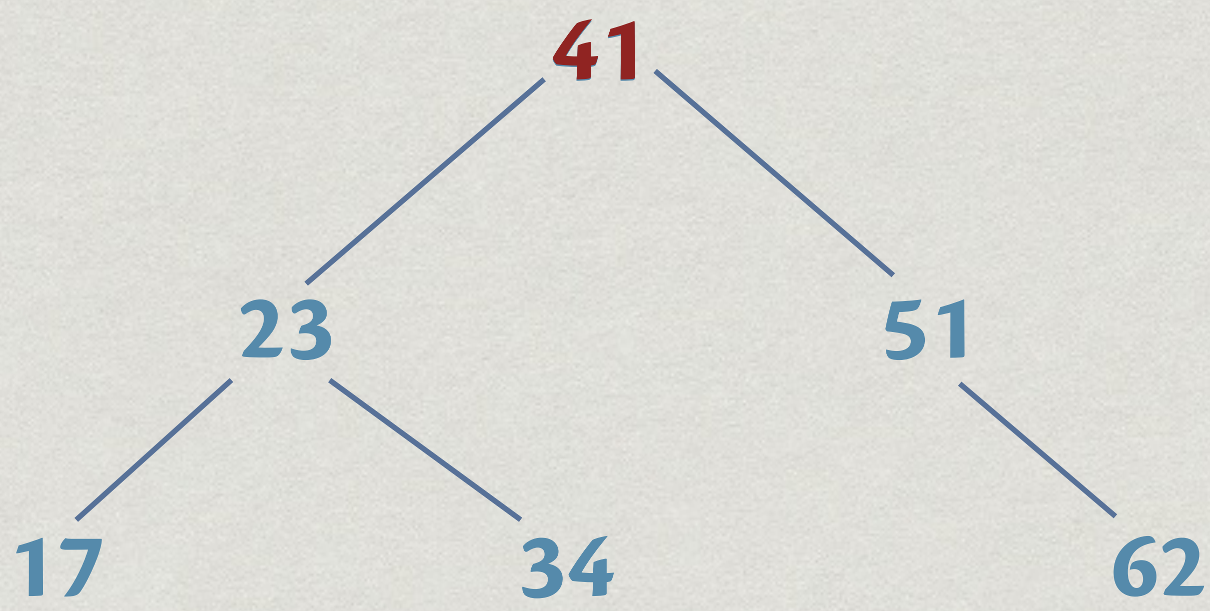
Inserting in a tree

- Inserting **33** and **48**



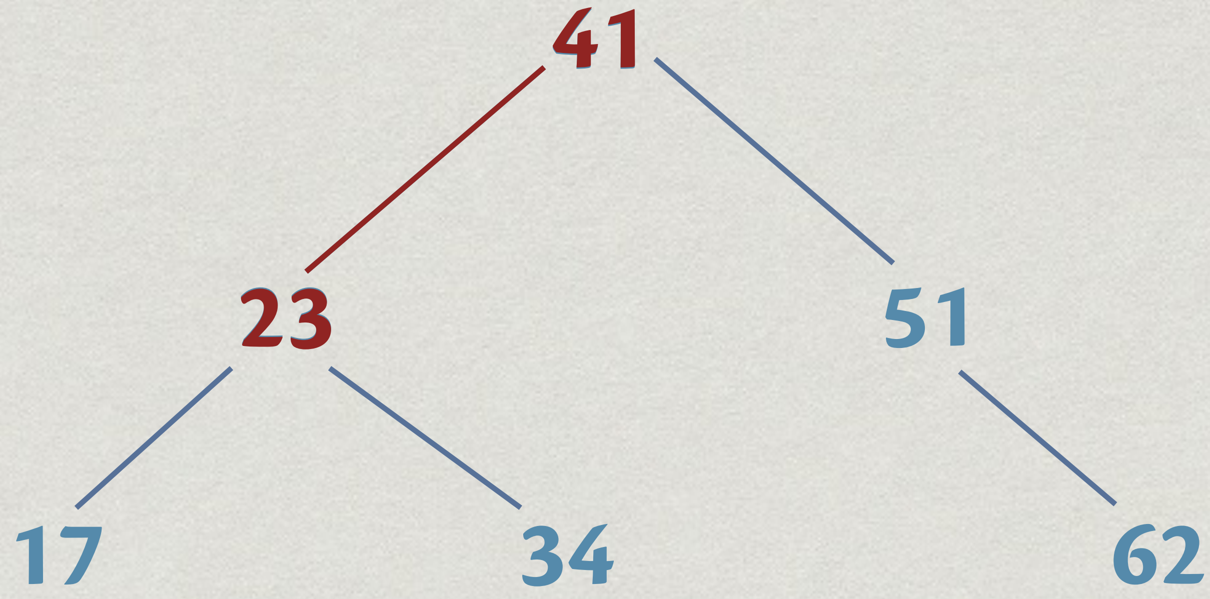
Inserting in a tree

- Inserting **33** and **48**



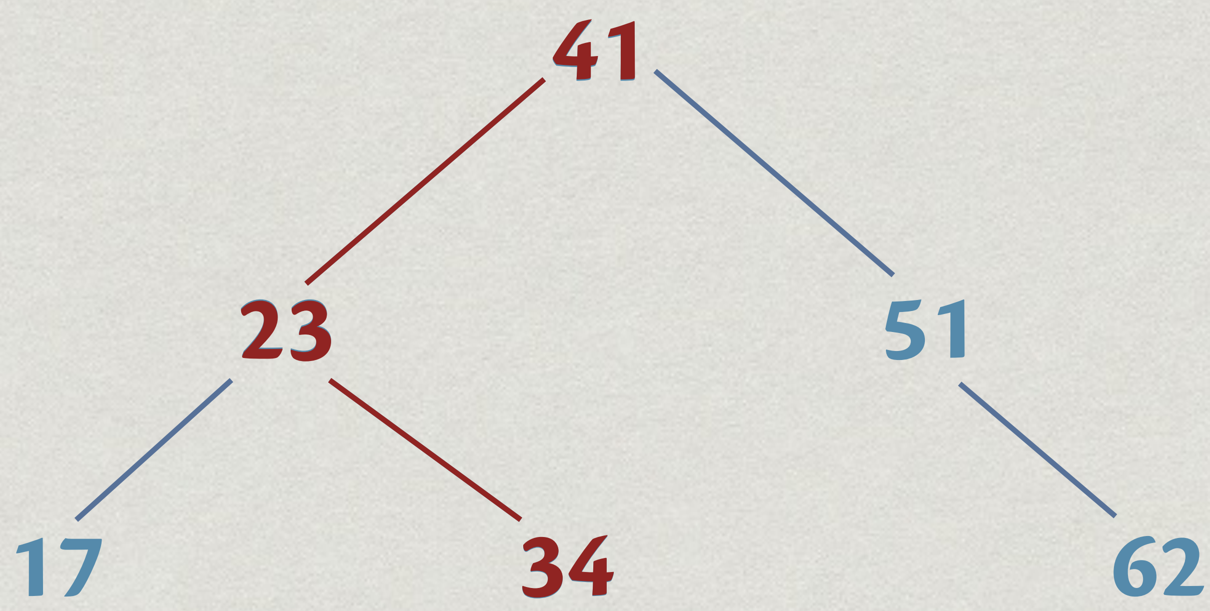
Inserting in a tree

- Inserting **33** and **48**



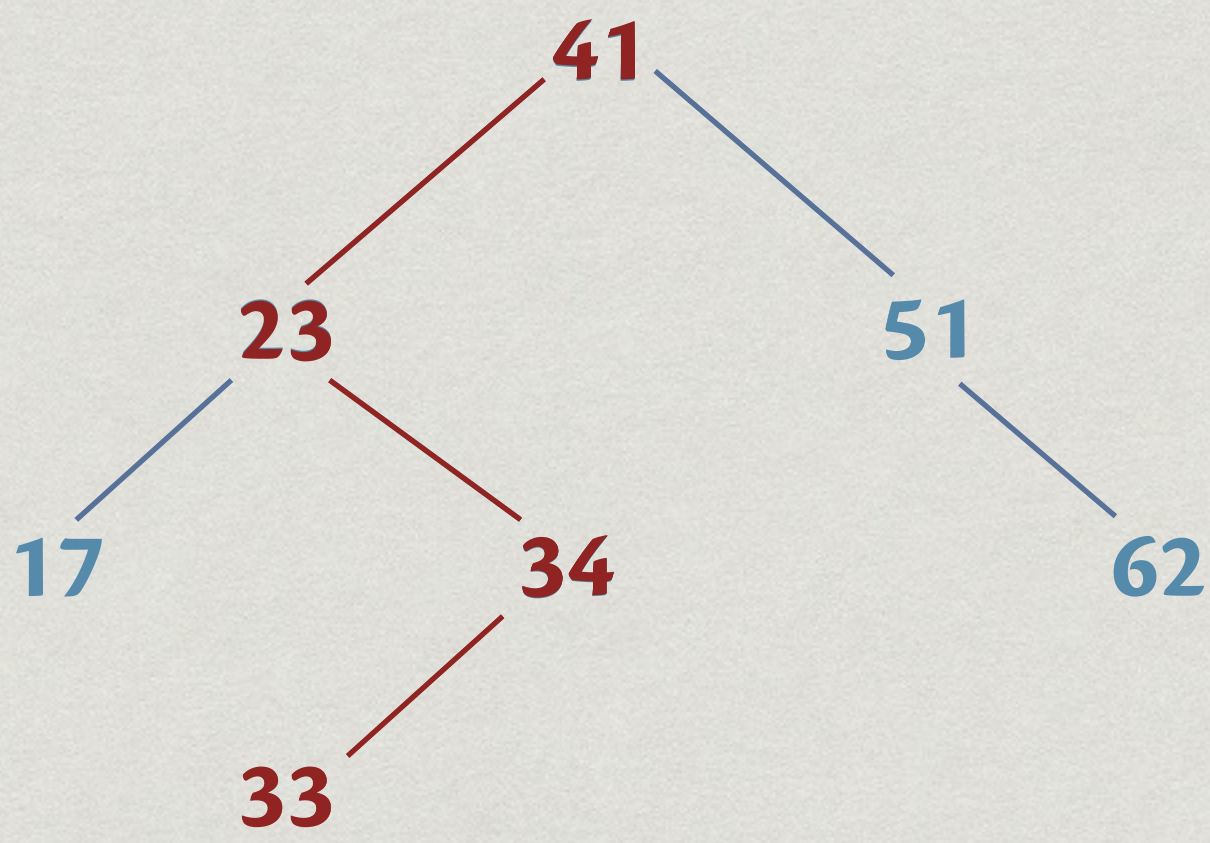
Inserting in a tree

- Inserting **33** and **48**



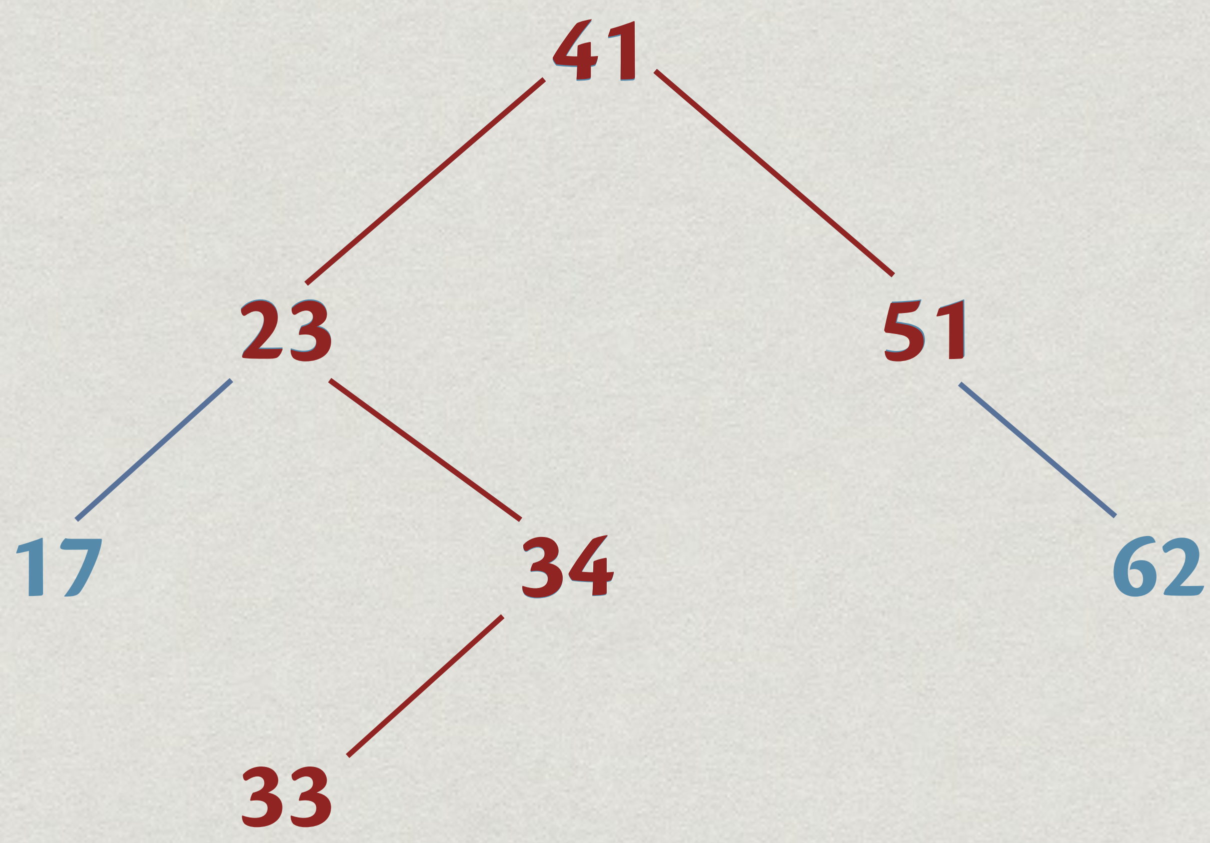
Inserting in a tree

- Inserting **33** and **48**



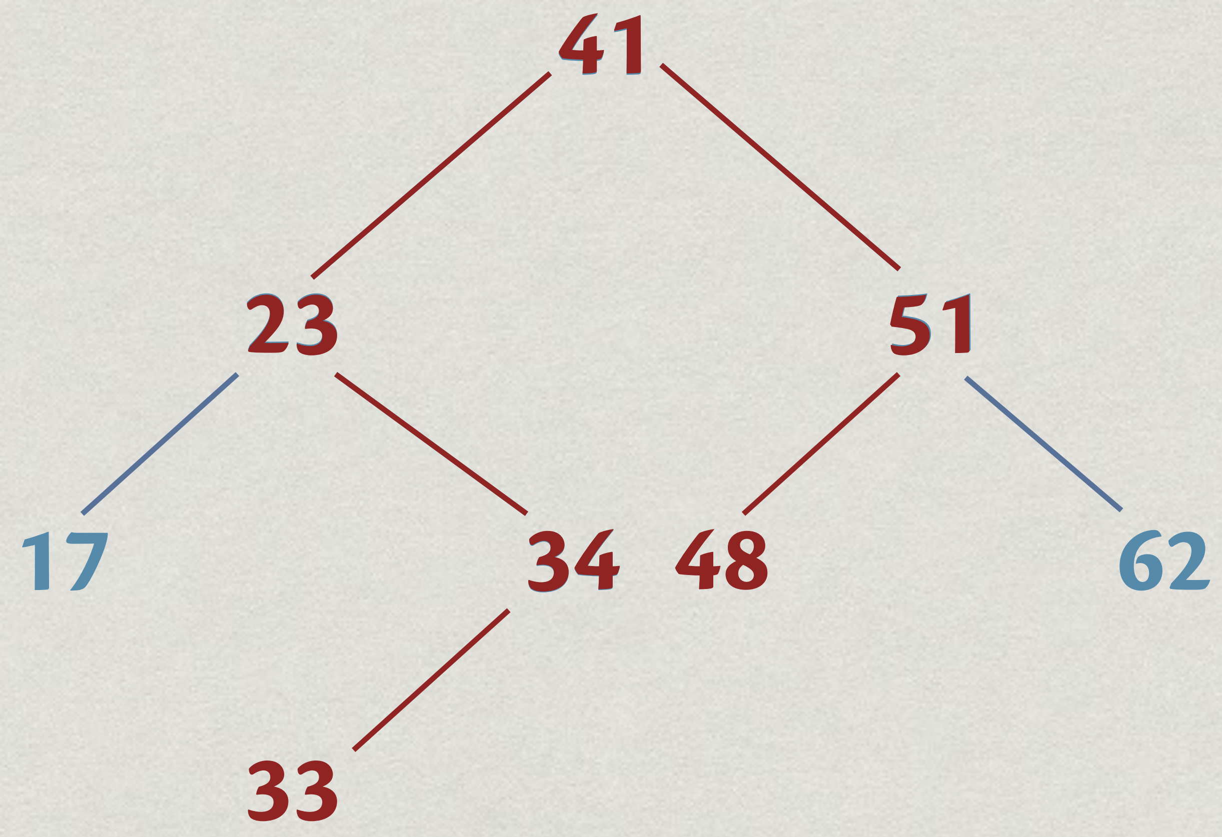
Inserting in a tree

* Inserting **33** and **48**



Inserting in a tree

- Inserting **33** and **48**



Inserting in a tree

- Inserting a value **v** in a search tree
- Search for **v** in the tree
- If **v** is in the tree, there is nothing to do
- If not, add a node with value **v** at the place where **v** is missing

Inserting in a tree

- If the tree is empty, create a node with value **v** and empty subtrees
- If the tree is nonempty
  - If **v** is the value at the root, exit
  - If **v** is smaller than the value at the root, insert **v** in left subtree (which could be empty)
  - If **v** is larger than the value at the root, insert **v** in right subtree (which could be empty)

Inserting in a tree

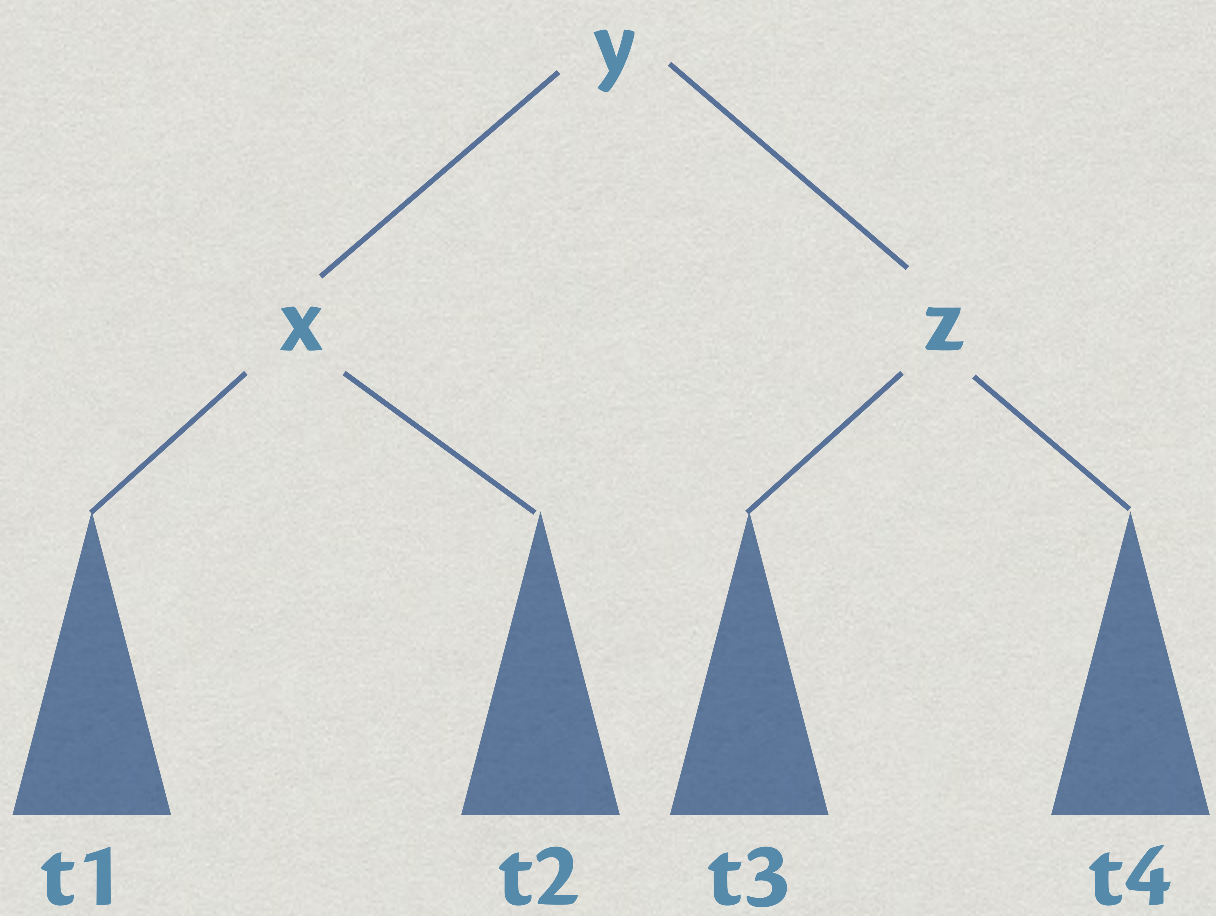
- ```
  insert :: Ord a => STree a -> a -> STree a
  insert Nil v  = Node Nil v Nil
  insert (Node tl x tr) v
   | x == v      = Node tl x tr
   | v < x       = Node (insert tl v) x tr
   | otherwise   = Node tl x (insert tr v)
  ```
- Worst case: running time proportional to length of the longest path from root to a leaf (**height**)
  ```

Deleting from a tree

- Deleting **v** from the tree
- If the tree is empty, exit
- If the tree is nonempty
  - If **v** is smaller than the value at the root, delete **v** from left subtree (which could be empty)
  - If **v** is larger than the value at the root, delete **v** from right subtree (which could be empty)

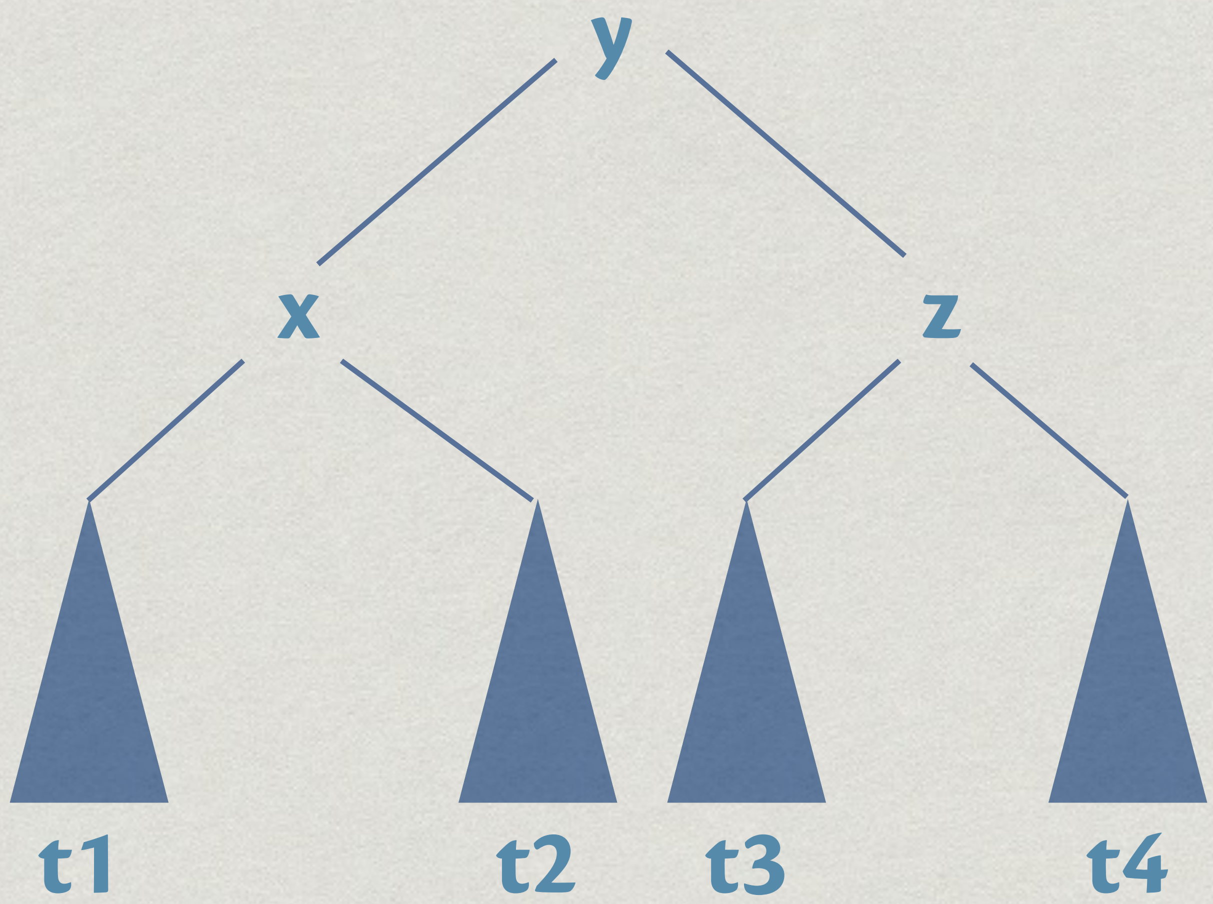
Deleting from a tree

- What if **v** is the value at the root? v = y



- What value should replace **y**?

Deleting from a tree



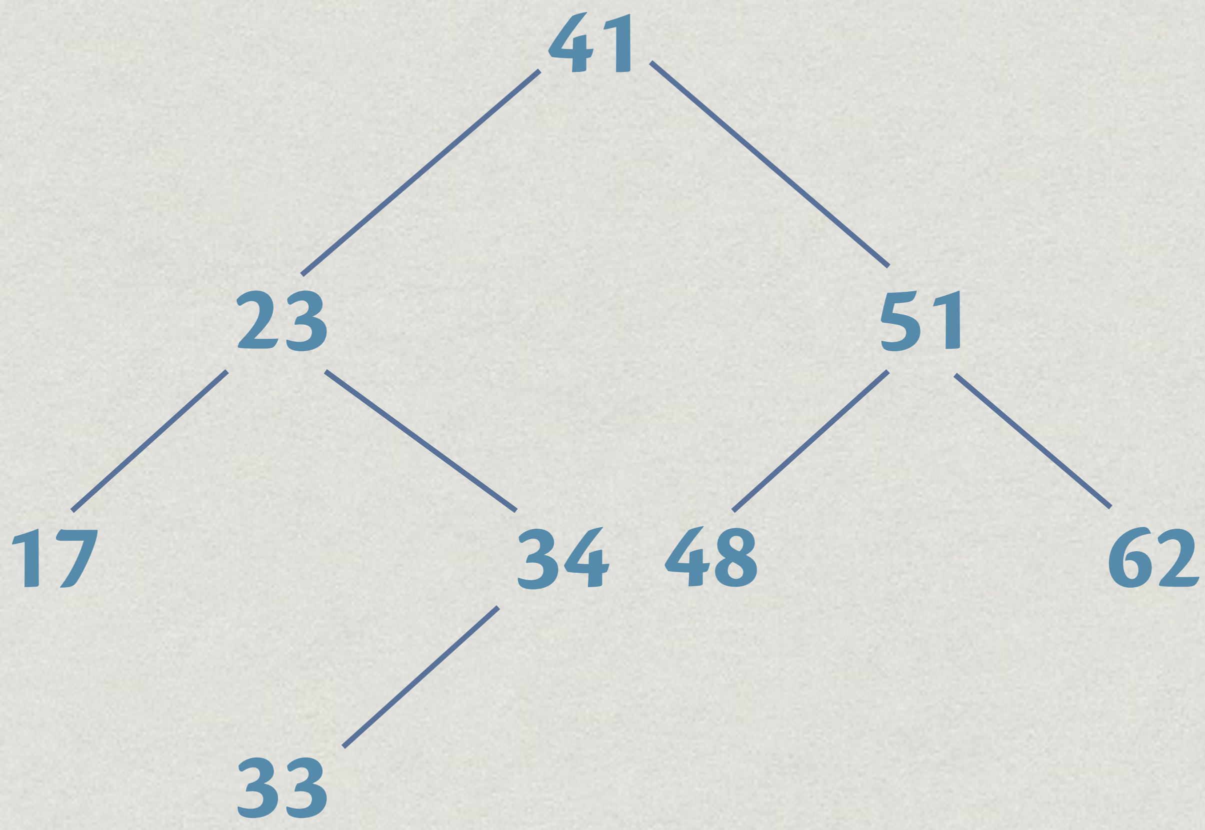
- What if v = y?
- Cannot blindly push **x** or **z** up the tree

Deleting from a tree

Delete **41** from the tree

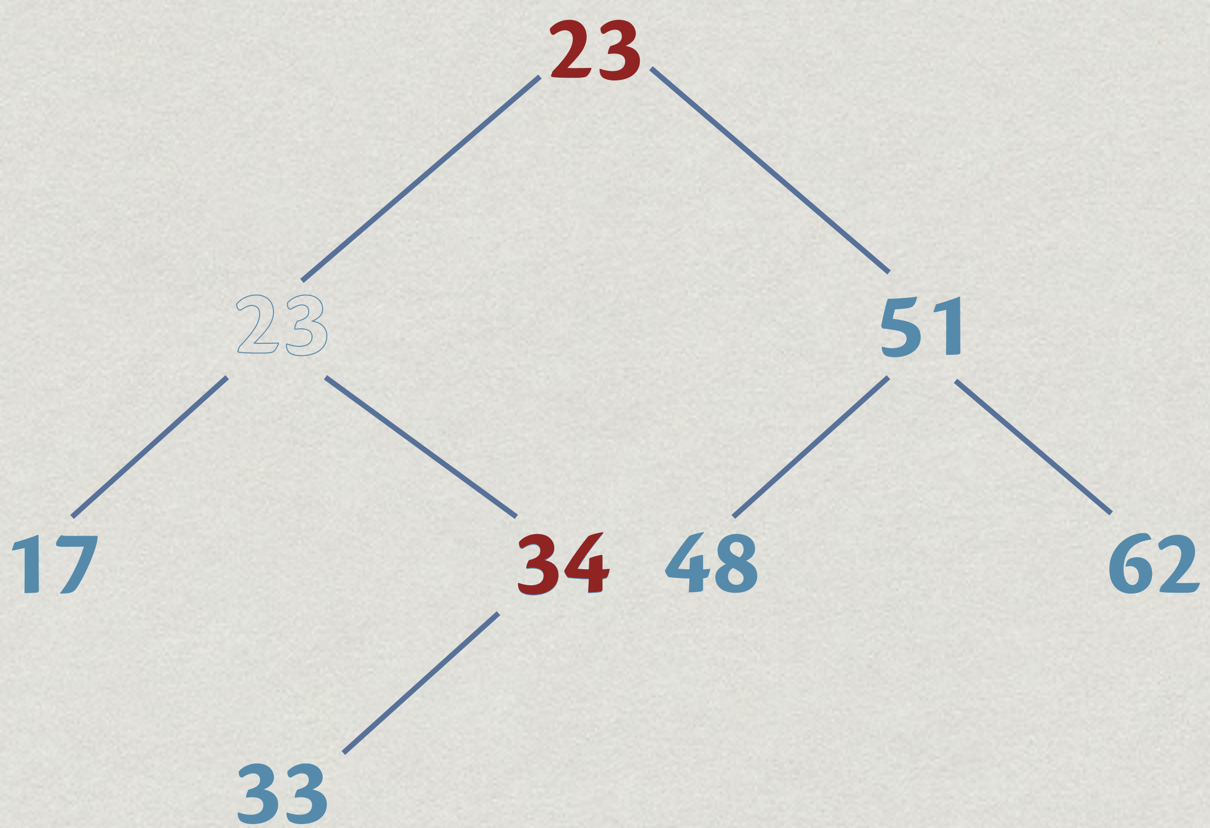
Deleting from a tree

Delete **41** from the tree



Deleting from a tree

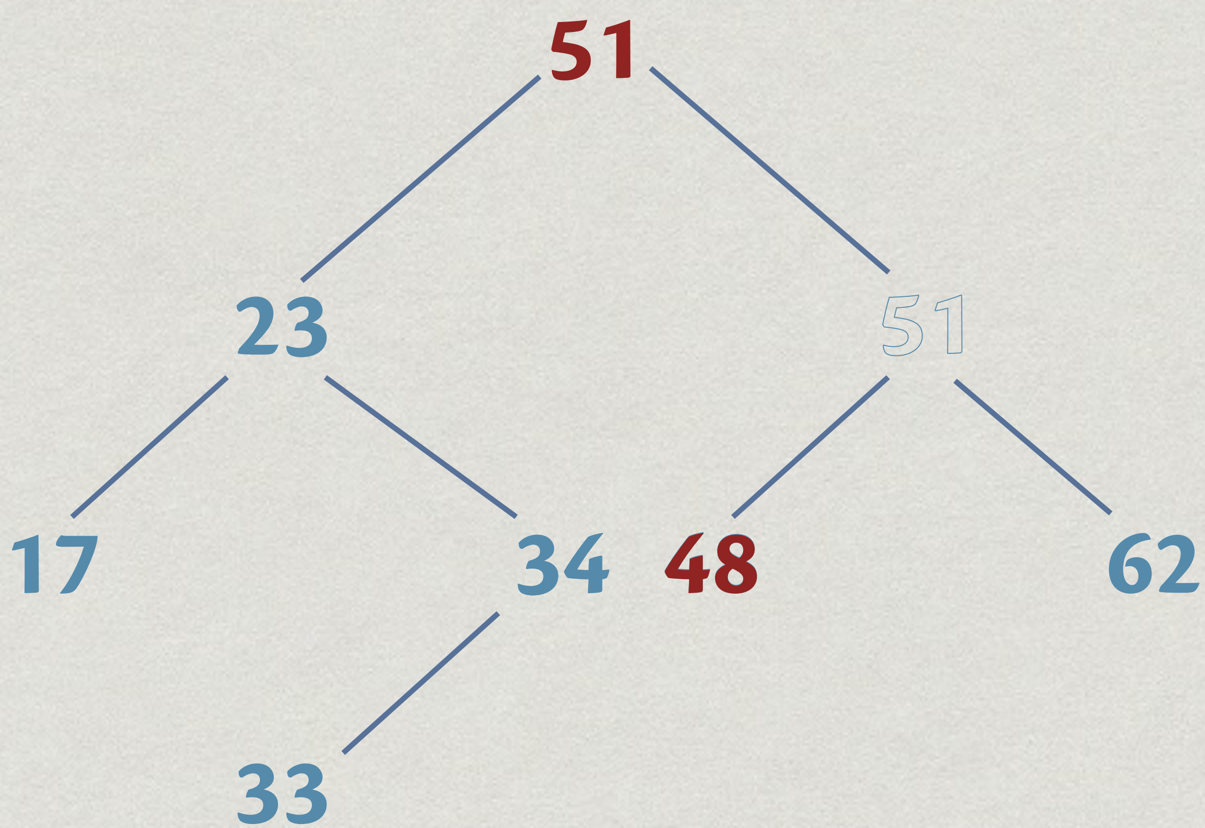
Delete **41** from the tree



Cannot shift 23 up
Conflict with 34

Deleting from a tree

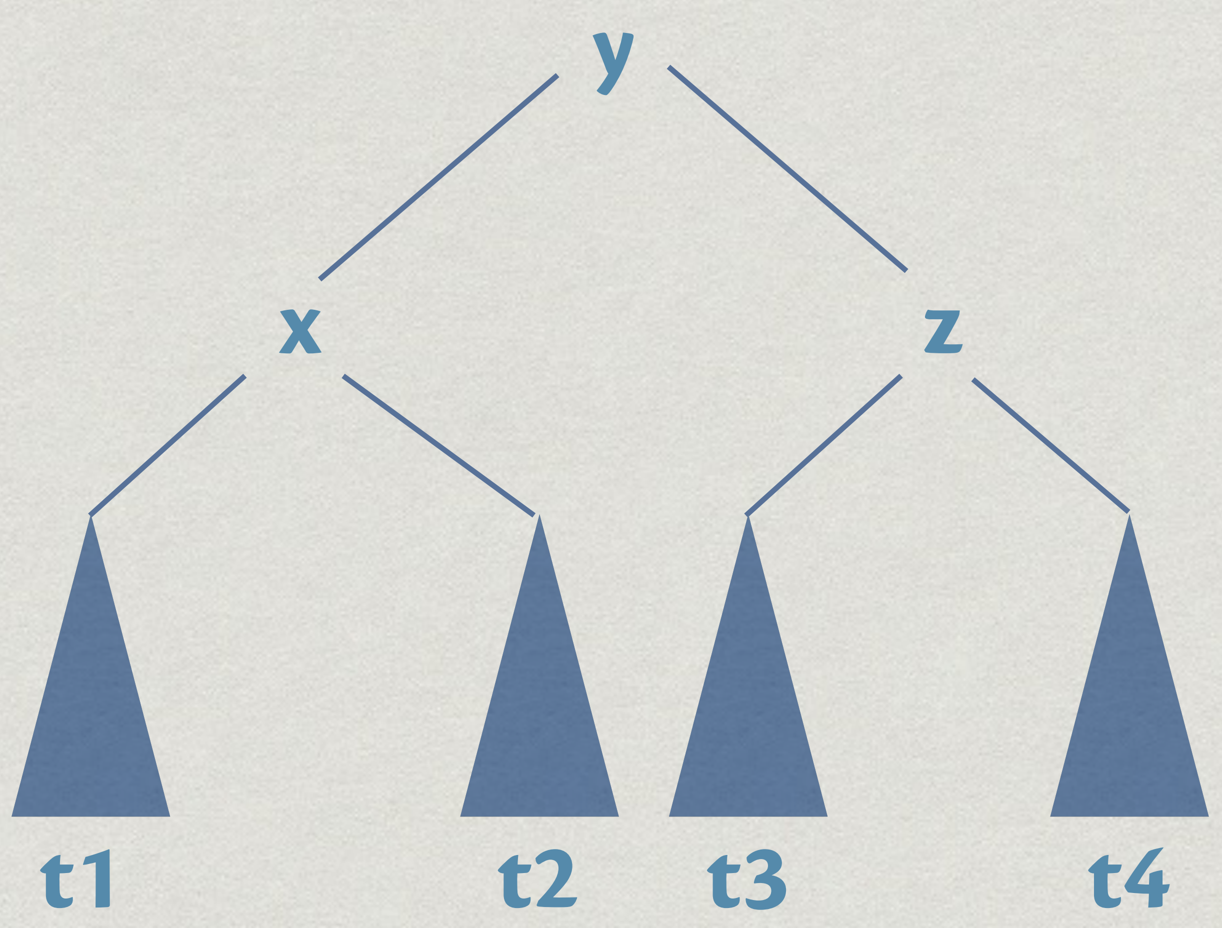
Delete **41** from the tree



Cannot shift 51 up
Conflict with 48

Deleting from a tree

- What if v = y?
- Cannot blindly push **x** or **z** up the tree
- Move up a value that is larger than the left and smaller than the right
- Either **maximum** value in **left subtree**, or **minimum** value in **right subtree**

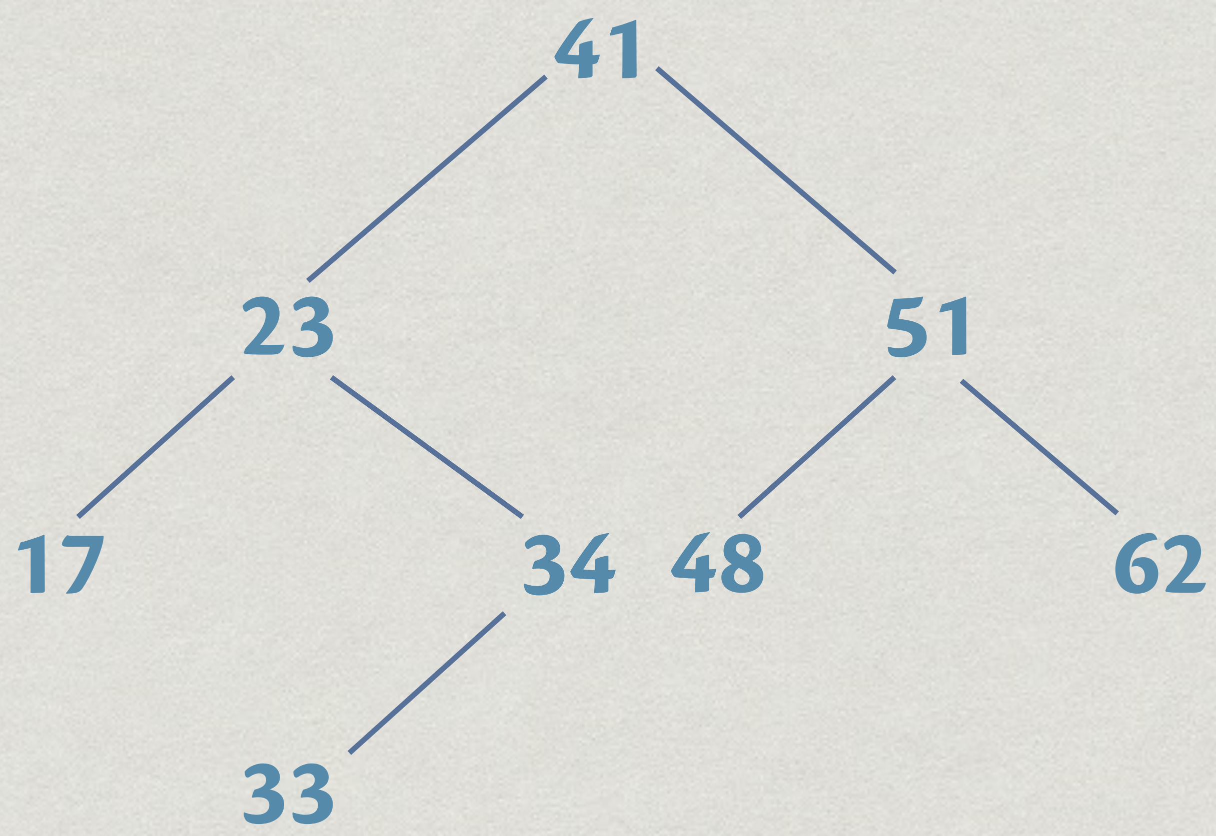


Deleting from a tree

Delete **41** from the tree

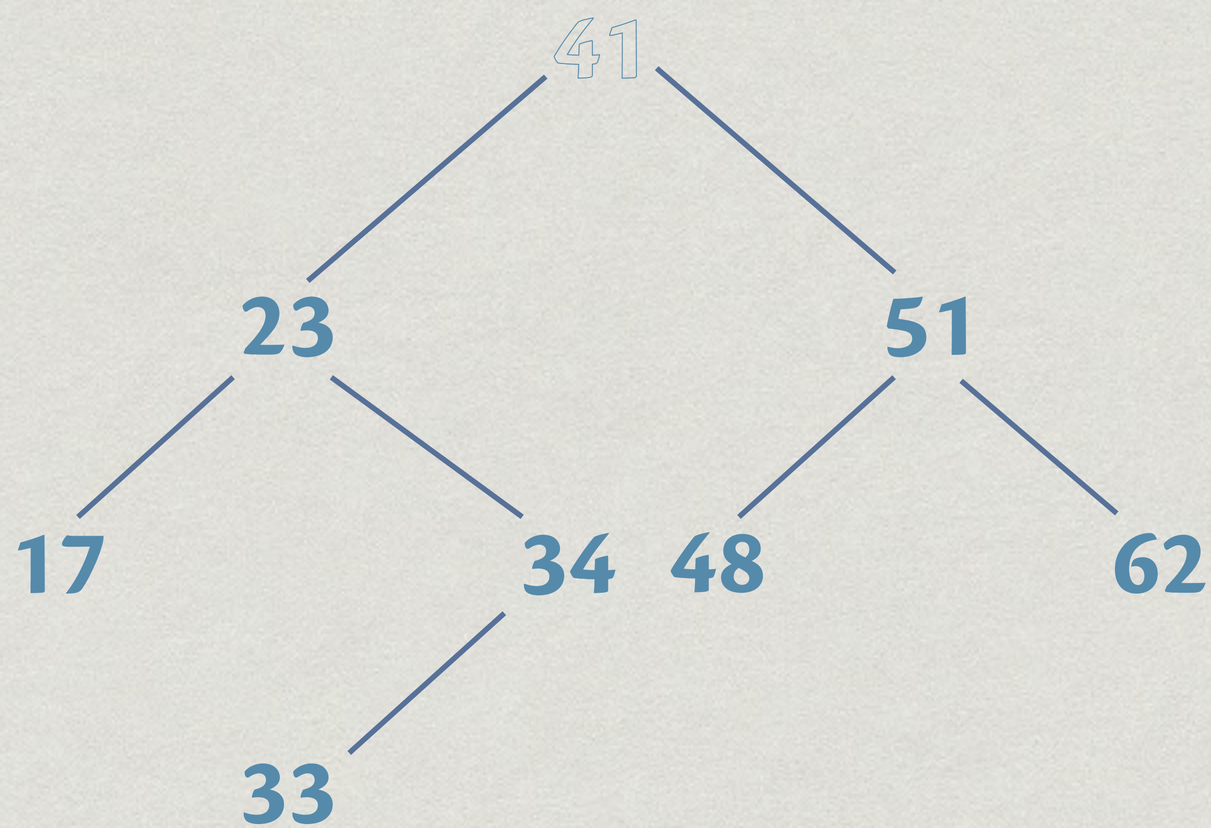
Deleting from a tree

Delete **41** from the tree



Deleting from a tree

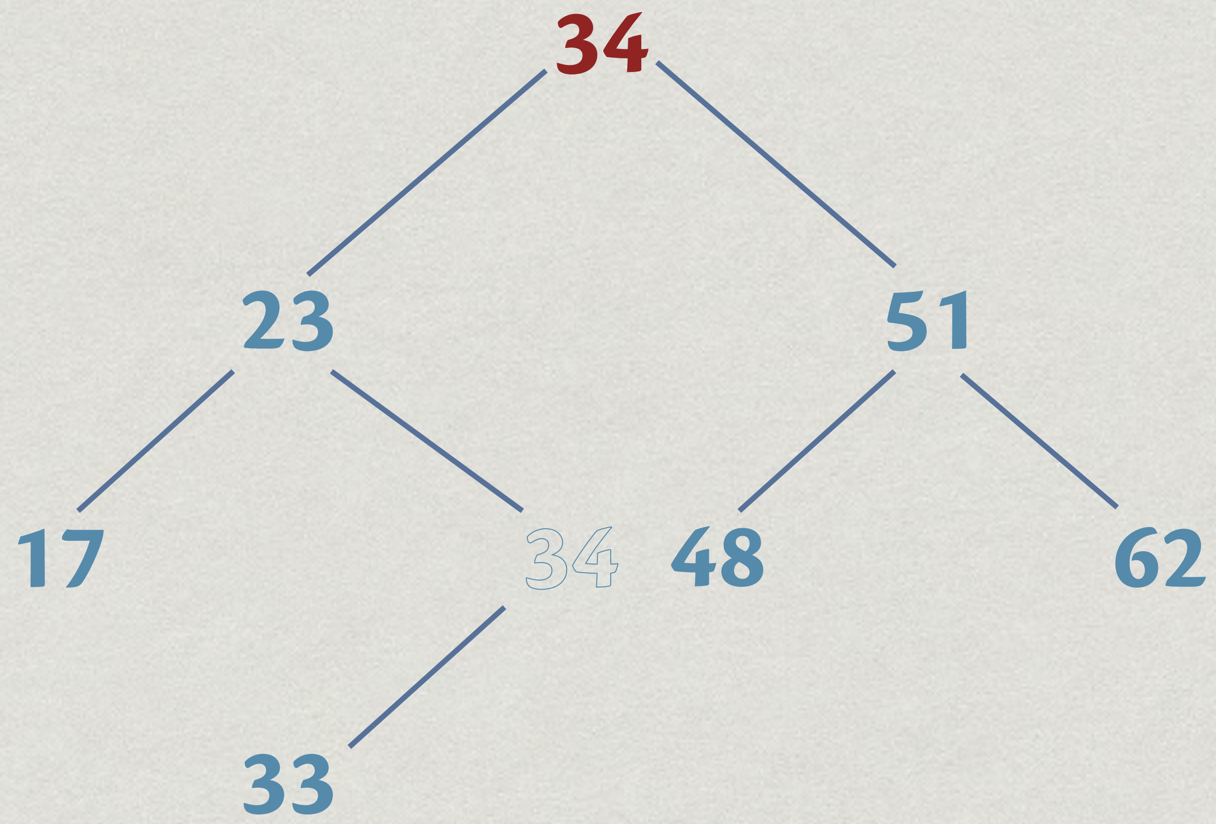
Delete **41** from the tree



Remove 41

Deleting from a tree

Delete **41** from the tree

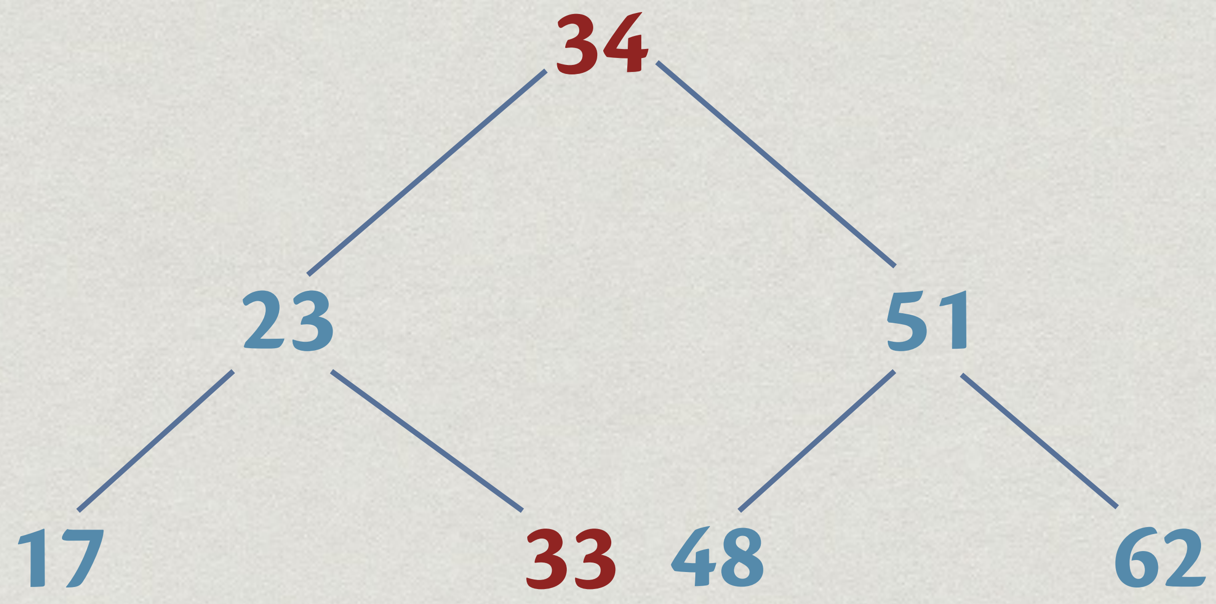


Remove 41

Move up maximum in left subtree, 34

Deleting from a tree

Delete **41** from the tree



Remove 41

Move up maximum in left subtree, 34

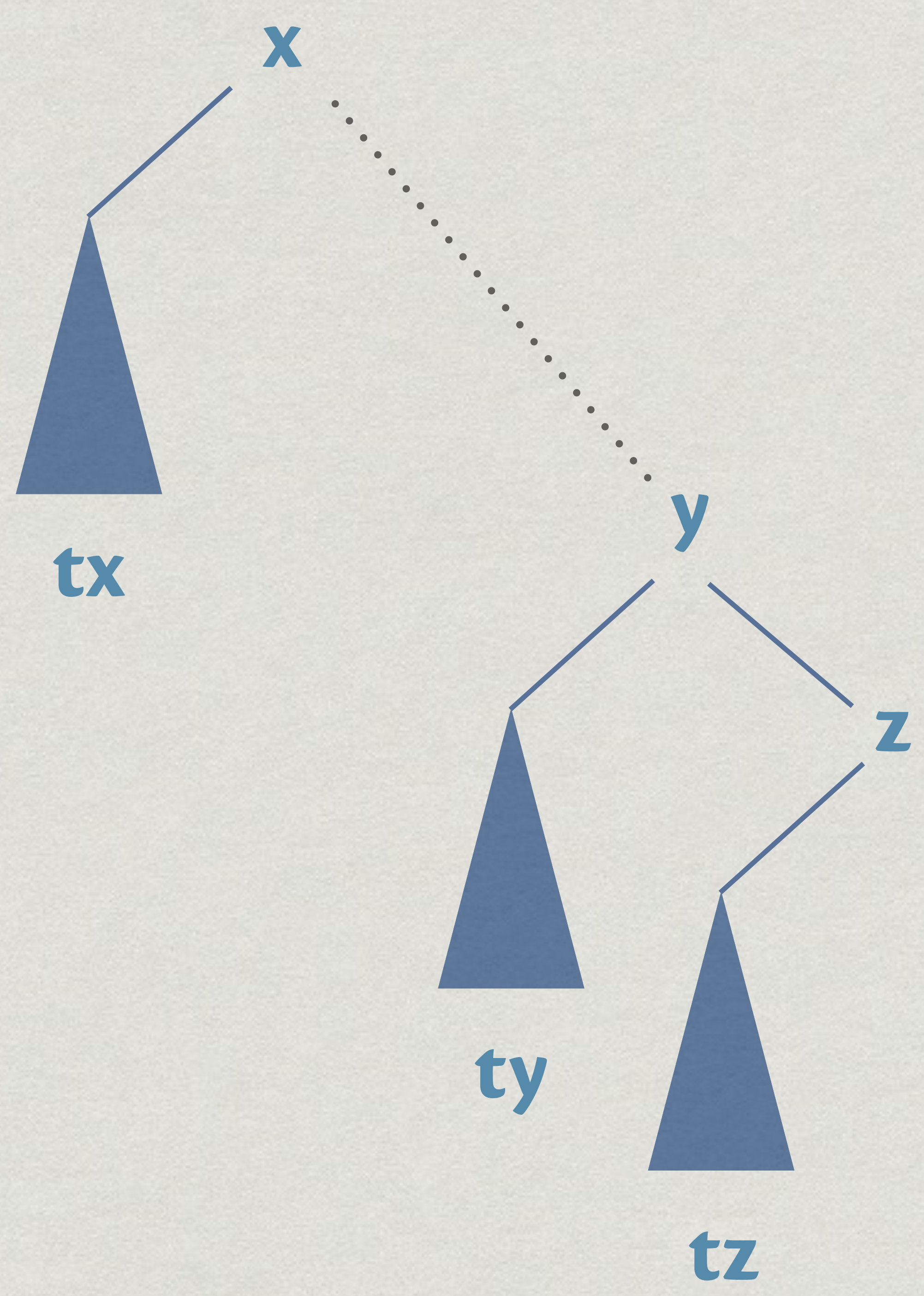
Move 33 up to occupy 34's place

Deleting the maximum value

* Keep going right till you reach a node whose right subtree is empty
* Remove the node
* Replace the node by its left subtree

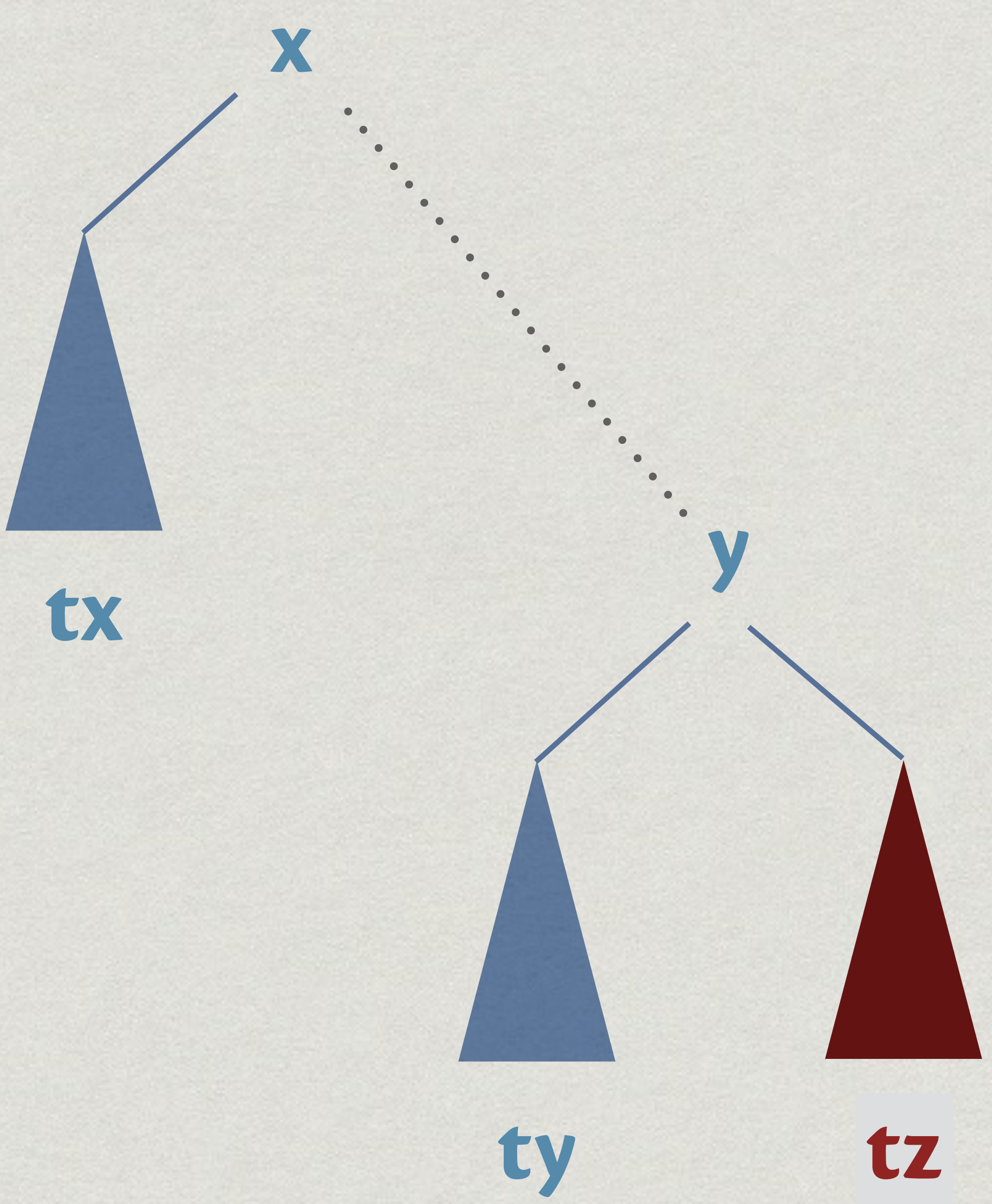
Deleting the maximum value

- Keep going right till you reach a node whose right subtree is empty
- Remove the node
- Replace the node by its left subtree



Deleting the maximum value

* Keep going right till you reach a node whose right subtree is empty
* Remove the node
* Replace the node by its left subtree



Deleting the maximum value

* `deletemax :: Ord a => STree a -> (a, STree a)`

```
-- At the rightmost node
deletemax (Node tl x Nil) = (x, tl)

-- Always descend right
deletemax (Node tl x tr) = (y, Node tl x ty)
  where (y, ty) = deletemax tr
```

* `deletemax` returns the **maximum** value and the **modified tree**

Deleting from the tree

- ```
  delete :: Ord a => STree a -> a -> STree a
  delete Nil v = Nil

  delete (Node tl x tr) v
    | v < x          = Node (delete tl v) x tr
    | v > x          = Node tl x (delete tr v)
    | otherwise   = if (tl == Nil) then tr
                    else (Node ty y tr)
        where (y, ty) = deletemax tl
  ```

- **Worst case**: running time proportional to length of the longest path from root to a leaf (**height**)
  ```

Other useful functions

- ```
  makeTree :: Ord a => [a] -> STree a
  makeTree = foldl insert Nil
  ```

- ```
  inorder :: STree a -> [a]
  inorder Nil = []
  inorder (Node tl x tr) = inorder tl
                   ++ [x] ++ inorder tr
  ```

- `inorder t` prints out the values in `t` in ascending order

- `sort = inorder . makeTree`

Summary

- Binary search trees can be used to store a set of elements and support the operations insert, delete and search
- Fundamental data structure
- `insert`, `delete` and `search` takes time proportional to the **height** of the tree
- But height can be as large as the number of nodes in a tree
- **Improvements in the next lecture**