# Introduction to Programming: Lecture 11

K Narayan Kumar

Chennai Mathematical Institute

http://www.cmi.ac.in/~kumar

12 Sep 2013

# The binary tree datatype

# The binary tree datatype

```
data Btree a = Nil | Node (Btree a) a (Btree a)
```

# The binary tree datatype

```
data Btree a = Nil | Node (Btree a) a (Btree a)
```

- `Nil` and `Node` are the constructors.

# The binary tree datatype

```
data Btree a = Nil | Node (Btree a) a (Btree a)
```

- `Nil` and `Node` are the constructors.

```
Node (Node Nil 4 Nil) 6
     (Node (Node Nil 2 Nil)  3 (Node Nil 5 Nil))
```

# The binary tree datatype

```
data Btree a = Nil | Node (Btree a) a (Btree a)
```

- `Nil` and `Node` are the constructors.

```
Node (Node Nil 4 Nil) 6
     (Node (Node Nil 2 Nil)  3 (Node Nil 5 Nil))
```

```
        6
       / \
      4   3
         / \
        2   5
```

# Levels

- List nodes level by level and from left to right within each level.

```
    4
   / \
  2   5
 / \
1   3
```

# Levels

- List nodes level by level and from left to right within each level.

```
      4
     / \
    2   5
   / \
  1   3

[4,2,5,1,3]
```

# Levels

- List nodes level by level and from left to right within each level.

```
        4
       / \
      2   5
     / \
    1   3
```

```
[4,2,5,1,3]
```

```
mylevels: Btree a -> [[a]]
mylevels Nil = []
mylevels (Node tl x tr) =
    [x]:(join (mylevels tl) (mylevels tr))

level t = concat (mylevels t)
```

# Levels

- List nodes level by level and from left to right within each level.

```
      4
     / \
    2   5
   / \
  1   3
```

```
[4,2,5,1,3]
```

```
mylevels: Btree a -> [[a]]
mylevels Nil = []
mylevels (Node tl x tr) =
    [x]:(join (mylevels tl) (mylevels tr))

level t = concat (mylevels t)
```

# Search trees

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

# Search trees

- In a search tree
  - Values in the left subtree are smaller than the current node
  - Values in the right subtree are bigger than the current node
- Two search trees for values `[1,2,3,4,5,7]`

```
     4                  3
    / \                / \
   2   5              2   5
  / \   \            /   / \
 1   3   7          1   4   7
```

## Search trees

- In a search tree
  - Values in the left subtree are smaller than the current node
  - Values in the right subtree are bigger than the current node
- Two search trees for values `[1,2,3,4,5,7]`

```
        4                      3
       / \                    / \
      2   5                  2   5
     / \   \                /   / \
    1   3   7              1   4   7
```

- Search Trees in Haskell

```
data Ord a => Stree a = Nil | Node (Stree a) a (Stree a
 deriving (Eq, Show)
```

# Search trees

- In a search tree
  - Values in the left subtree are smaller than the current node
  - Values in the right subtree are bigger than the current node
- Two search trees for values `[1,2,3,4,5,7]`

```
      4                  3
     / \                / \
    2   5              2   5
   / \   \            /   / \
  1   3   7          1   4   7
```

- Search Trees in Haskell

```haskell
data Ord a => Stree a = Nil | Node (Stree a) a (Stree a
 deriving (Eq, Show)
```

  - Need `Ord a` to compare values
  - No gurantee of being a search tree!

# Search trees ...

- Is it a search tree?

# Search trees ...

- Is it a search tree?

```
isstree:: Ord a => (Stree a) -> Bool
isstree Nil = True
isstree (Node tl y tr)
     = (isstree tl) && (isstree tr) &&
           (maxt tl < y) && (y < (mint tr))
```

# Search trees ...

- Is it a search tree?

```
isstree:: Ord a => (Stree a) -> Bool
isstree Nil = True
isstree (Node tl y tr)
    = (isstree tl) && (isstree tr) &&
          (maxt tl < y) && (y < (mint tr))

mint (Node Nil v Nil) = v
mint (Node tl v tr) = min (mint tl) (min v (mint tr))
```

## Search trees ...

- Is it a search tree?

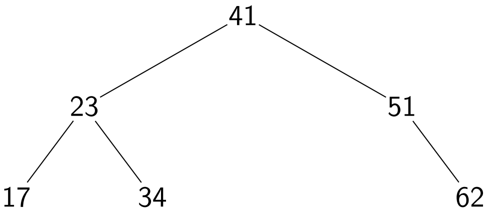
```
isstree:: Ord a => (Stree a) -> Bool
isstree Nil = True
isstree (Node tl y tr)
    = (isstree tl) && (isstree tr) &&
         (maxt tl < y) && (y < (mint tr))

mint (Node Nil v Nil) = v
mint (Node tl v tr) = min (mint tl) (min v (mint tr))
```

- In how many ways is the above program incorrect?

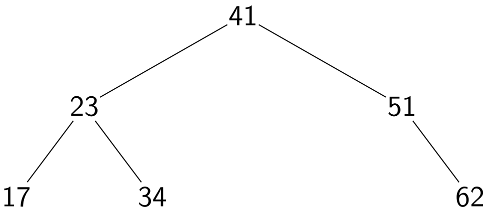
# Search trees . . .

- Searching for a value



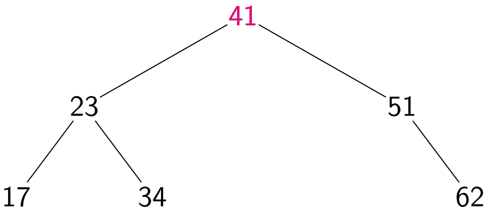
# Search trees . . .

- Searching for a value
  Searching for 34



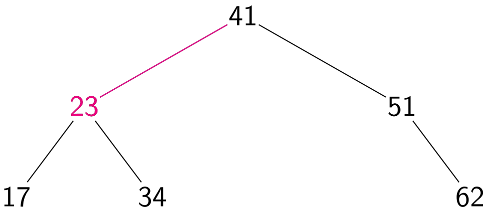
# Search trees . . .

- Searching for a value
  Searching for 34



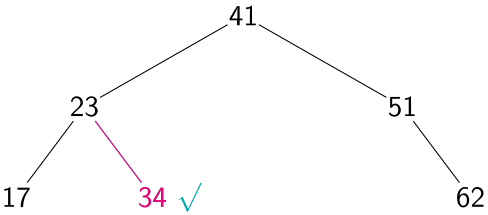
# Search trees . . .

- Searching for a value
  Searching for 34



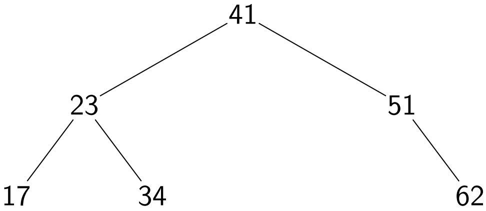
# Search trees . . .

- Searching for a value
  Searching for 34



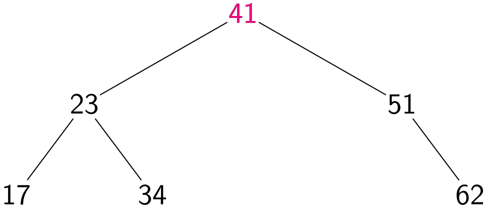
# Search trees . . .

- Searching for a value
  Searching for 49



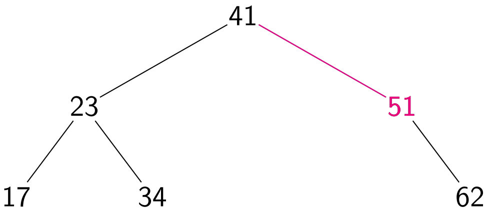
# Search trees ...

- Searching for a value
  Searching for 49



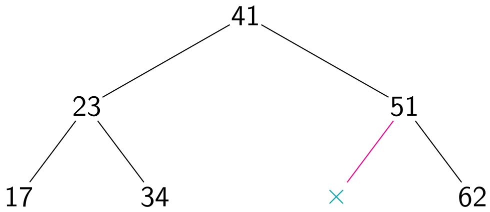
# Search trees . . .

- Searching for a value
  Searching for 49



# Search trees . . .

- Searching for a value
  Searching for 49



## Search trees . . .

- Searching for a value $v$

# Search trees . . .

- Searching for a value $v$
- Start at the root

# Search trees . . .

- Searching for a value $v$
- Start at the root
- At each node

# Search trees ...

- Searching for a value $v$
- Start at the root
- At each node
  - If the value is found, report Yes

# Search trees . . .

- Searching for a value $v$
- Start at the root
- At each node
  - If the value is found, report Yes
  - If $v$ is smaller than the current value
    - If left child exists, search for $v$ in left subtree
    - Otherwise, report No

# Search trees ...

- Searching for a value $v$
- Start at the root
- At each node
  - If the value is found, report Yes
  - If $v$ is smaller than the current value
    - If left child exists, search for $v$ in left subtree
    - Otherwise, report No
  - If $v$ is larger than the current value
    - If right child exists, search for $v$ in right subtree
    - Otherwise, report No

# Search trees . . .

- Searching for a value $v$
- Start at the root
- At each node
  - If the value is found, report Yes
  - If $v$ is smaller than the current value
    - If left child exists, search for $v$ in left subtree
    - Otherwise, report No
  - If $v$ is larger than the current value
    - If right child exists, search for $v$ in right subtree
    - Otherwise, report No
- Worst case: Number of steps is equal to the longest path from the root to a leaf

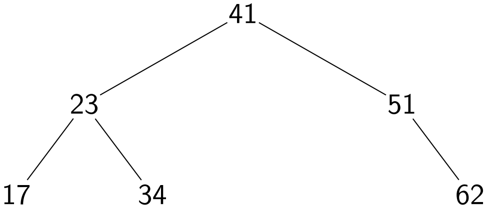
# Search trees . . .

- Searching for a value

```
searchtree :: Ord a => (Stree a) -> a -> Bool
searchtree Nil v = False
searchtree (Node tl y tr) v
  | v == y    = True
  | v < y     = searchtree tl v
  | otherwise = searchtree tr v
```

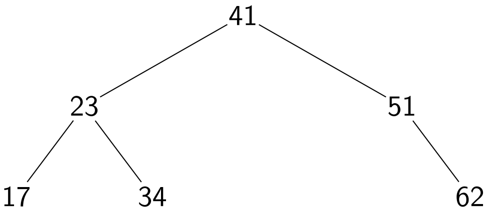
# Search trees, inserting a value

- Insert a value



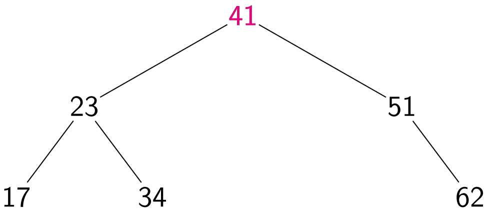
# Search trees, inserting a value

- Insert a value
  Insert 33



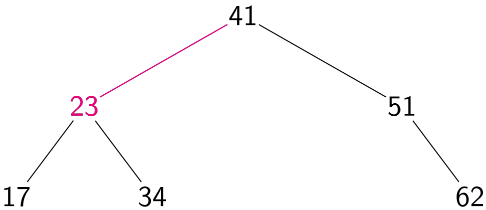
# Search trees, inserting a value

- Insert a value
  Insert 33



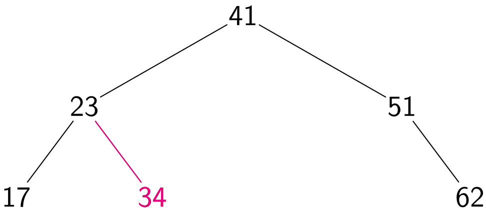
# Search trees, inserting a value

- Insert a value
  Insert 33



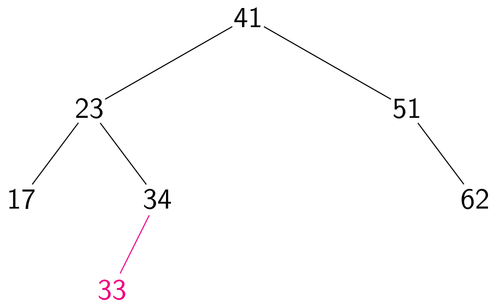
# Search trees, inserting a value

- Insert a value
  Insert 33



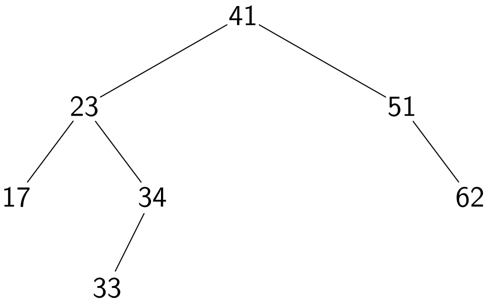
# Search trees, inserting a value

- Insert a value
  Insert 33



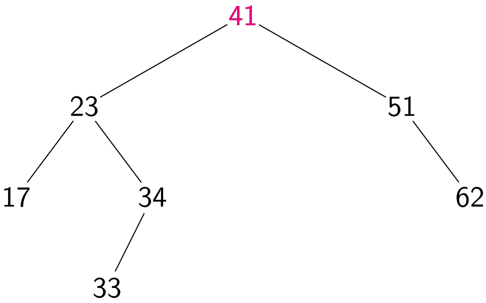
# Search trees, inserting a value

- Insert a value
  Insert 48



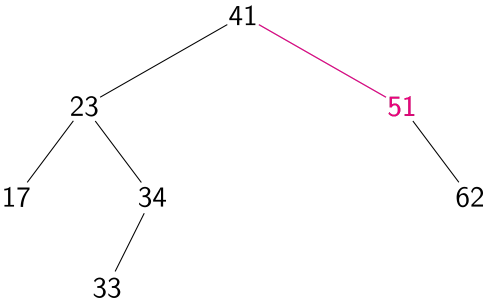
# Search trees, inserting a value

- Insert a value
  Insert 48



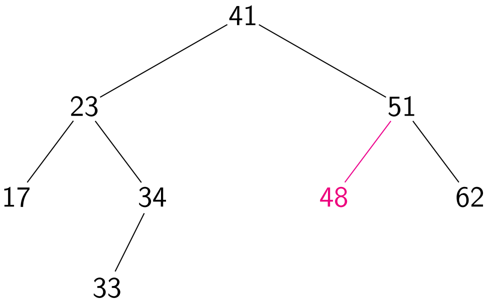
# Search trees, inserting a value

- Insert a value
  Insert 48



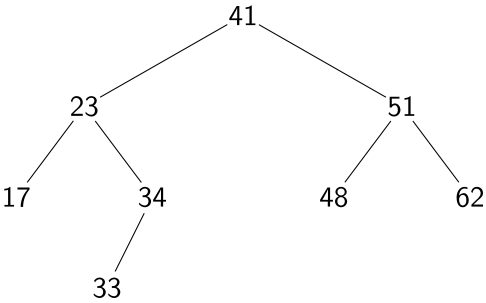
# Search trees, inserting a value

- Insert a value
  Insert 48



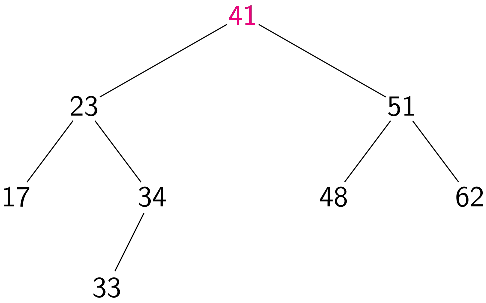
# Search trees, inserting a value

- Insert a value
  Insert 17



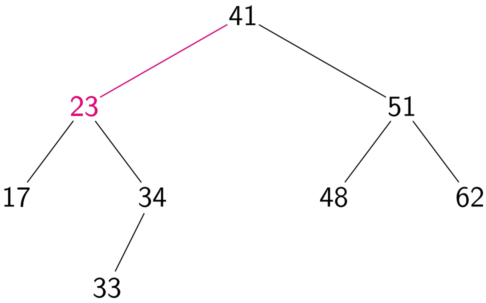
# Search trees, inserting a value

- Insert a value
  Insert 17



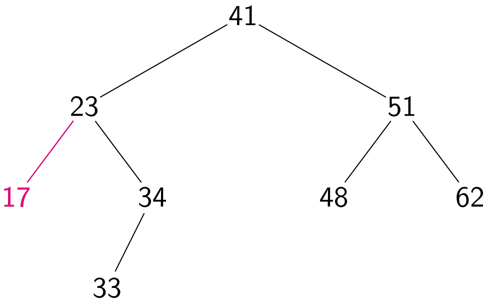
# Search trees, inserting a value

- Insert a value
  Insert 17



# Search trees, inserting a value

- Insert a value
  Insert 17



# Search trees, inserting a value . . .

- To insert a value $v$, find where it should be and add it if it is missing

# Search trees, inserting a value . . .

- To insert a value $v$, find where it should be and add it if it is missing
- Start at the root

# Search trees, inserting a value . . .

- To insert a value $v$, find where it should be and add it if it is missing
- Start at the root
- At each node

# Search trees, inserting a value . . .

- To insert a value $v$, find where it should be and add it if it is missing
- Start at the root
- At each node
  - If the value is found, exit

# Search trees, inserting a value . . .

- To insert a value $v$, find where it should be and add it if it is missing
- Start at the root
- At each node
  - If the value is found, exit
  - If $v$ is smaller than the current value
    - If left child exists, insert $v$ in left subtree
    - Otherwise, add a left child with value $v$

# Search trees, inserting a value . . .

- To insert a value $v$, find where it should be and add it if it is missing
- Start at the root
- At each node
  - If the value is found, exit
  - If $v$ is smaller than the current value
    - If left child exists, insert $v$ in left subtree
    - Otherwise, add a left child with value $v$
  - If $v$ is larger than the current value
    - If right child exists, insert $v$ in right subtree
    - Otherwise, add a right child with value $v$

# Search trees, inserting a value . . .

- To insert a value $v$, find where it should be and add it if it is missing
- Start at the root
- At each node
  - If the value is found, exit
  - If $v$ is smaller than the current value
    - If left child exists, insert $v$ in left subtree
    - Otherwise, add a left child with value $v$
  - If $v$ is larger than the current value
    - If right child exists, insert $v$ in right subtree
    - Otherwise, add a right child with value $v$
- Worst case: Number of steps is equal to the longest path from the root to a leaf

# Inserting into a search tree

- To insert a value, search for it to identify where it should go

```
inserttree :: Ord a => Stree a  -> a -> Stree a
inserttree Nil v = Node  Nil v Nil
inserttree (Node  tl y tr) v
  | v == y    = Node  tl y tr
  | v < y     = Node (inserttree tl v) y  tr
  | otherwise = Node  tl y (inserttree tr v)
```

- `inserttree` returns the tree with the value inserted.

# Search trees, deleting a value

- Deleting $v$ from a tree

## Search trees, deleting a value

- Deleting $v$ from a tree
- If $v$ does not match current node, inductively delete from left or right subtree

# Search trees, deleting a value

- Deleting $v$ from a tree
- If $v$ does not match current node, inductively delete from left or right subtree
- What if $v$ does match?

```
        y == v
       /   \
      x     z
     / \   / \
    t1 t2 t3 t4
```

- What value should replace `y`?

# Search trees, deleting a value

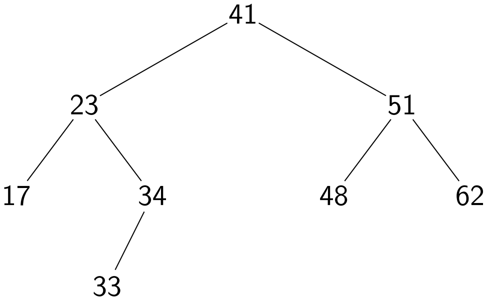
- Deleting $v$ from a tree
- If $v$ does not match current node, inductively delete from left or right subtree
- What if $v$ does match?

```
      y == v
     /   \
    x     z
   / \   / \
  t1 t2 t3 t4
```

- What value should replace `y`?
- Cannot blindly shift up `x` or `z`

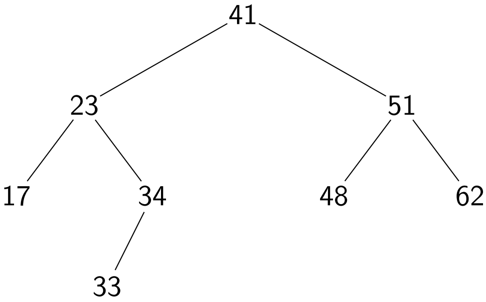
# Search trees, deleting a value . . .

- Delete a value



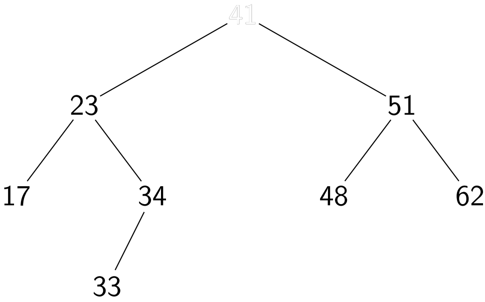
# Search trees, deleting a value . . .

- Delete a value
  Delete 41



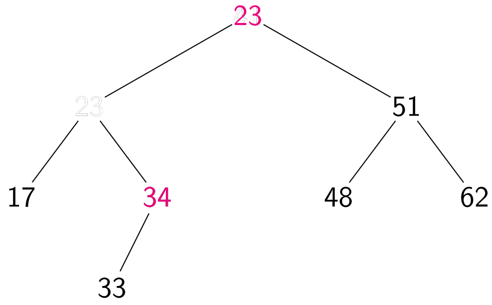
# Search trees, deleting a value . . .

- Delete a value
  Delete 41



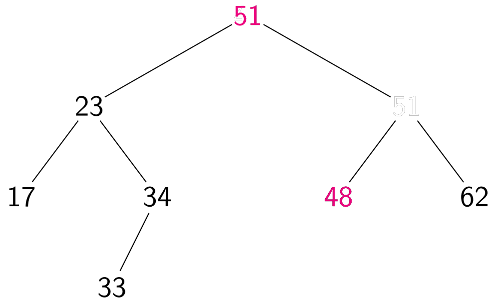
# Search trees, deleting a value . . .

- Delete a value
  Delete 41
  Cannot shift up 23



# Search trees, deleting a value . . .

- Delete a value
  Delete 41
  Cannot shift up 51



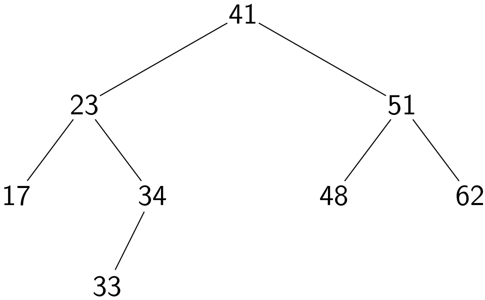
# Search trees, deleting a value . . .

```
        y == v
       /   \
      x     z
     / \   / \
    t1 t2 t3 t4
```

- Cannot blindly shift up `x` or `z`
- Need to move up a value that is bigger than left and smaller than right
    - Move up maximum value in left subtree . . .
    - . . . or minimum value in right subtree

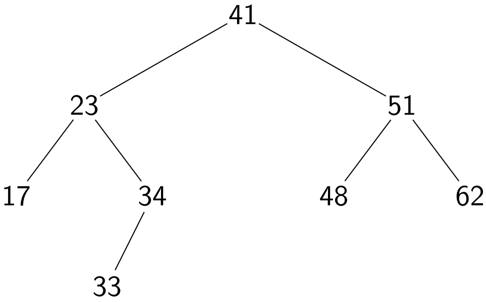
# Search trees, deleting a value . . .

- Delete a value



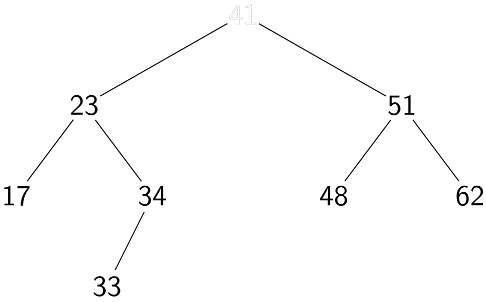
# Search trees, deleting a value ...

- Delete a value
  Delete 41



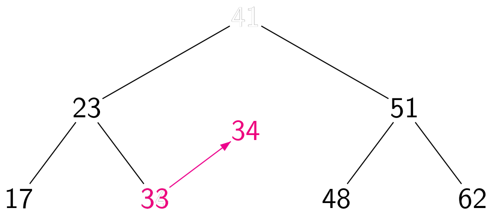
# Search trees, deleting a value ...

- Delete a value
  Delete 41



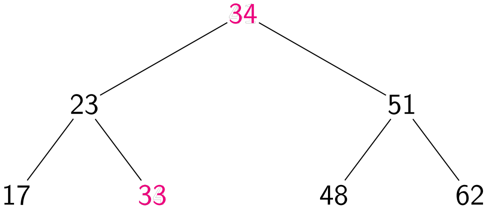
# Search trees, deleting a value . . .

- Delete a value
  Delete 41
  Remove maximum value in left subtree, 34



# Search trees, deleting a value . . .

- Delete a value
  Delete 41
  Remove maximum value in left subtree, 34
  . . . and use it to replace 41



# Search trees . . .

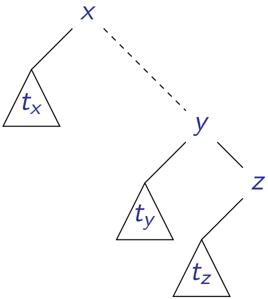
- Deleting the maximum value in a search tree

## Search trees . . .

- Deleting the maximum value in a search tree
- Keep going right till you run out of values
    - Rightmost value has no right subtree
    - Replace rightmost value by its left subtree

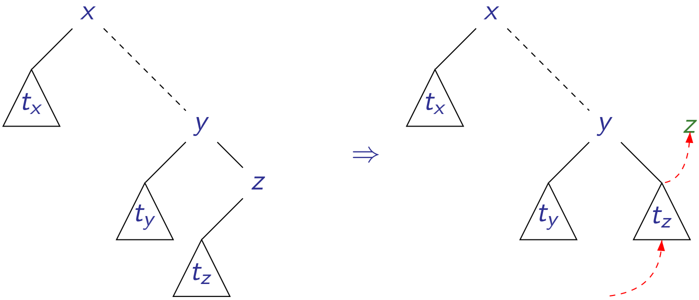
# Search trees . . .

- Deleting the maximum value in a search tree
- Keep going right till you run out of values
  - Rightmost value has no right subtree
  - Replace rightmost value by its left subtree



# Search trees . . .

- Deleting the maximum value in a search tree
- Keep going right till you run out of values
  - Rightmost value has no right subtree
  - Replace rightmost value by its left subtree



# Deleting maximum value in a search tree ...

- deletemax

```
deletemax :: Ord a => Stree a  -> (a ,Stree a)

-- We are at rightmost value
deletemax (Node  t1 y Nil) = (y,t1)

-- We are not yet at rightmost value
deletemax (Node t1 y t2) = (z, Node t1 y tz)
  where (z,tz) = deletemax t2
```

# Deleting maximum value in a search tree ...

- `deletemax`

```
deletemax :: Ord a => Stree a  -> (a ,Stree a)

-- We are at rightmost value
deletemax (Node  t1 y Nil) = (y,t1)

-- We are not yet at rightmost value
deletemax (Node t1 y t2) = (z, Node t1 y tz)
  where (z,tz) = deletemax t2
```

- Note that `deletemax` returns the maximum value and the modified tree

# Search trees, deleting a value ...

- To delete a value $v$

# Search trees, deleting a value . . .

- To delete a value $v$
- Start at the root

# Search trees, deleting a value ...

- To delete a value $v$
- Start at the root
- At each node

# Search trees, deleting a value . . .

- To delete a value $v$
- Start at the root
- At each node
  - If $v$ is smaller than the current value
    - If left child exists, delete $v$ from left subtree

# Search trees, deleting a value ...

- To delete a value $v$
- Start at the root
- At each node
  - If $v$ is smaller than the current value
    - If left child exists, delete $v$ from left subtree
  - If $v$ is larger than the current value
    - If right child exists, delete $v$ from right subtree

# Search trees, deleting a value . . .

- To delete a value $v$
- Start at the root
- At each node
  - If $v$ is smaller than the current value
    - If left child exists, delete $v$ from left subtree
  - If $v$ is larger than the current value
    - If right child exists, delete $v$ from right subtree
  - Otherwise, current value is $v$
    - If there is no left child, shift up right subtree to current node
    - If there is a left child
      - Delete maximum value $x$ from left subtree
      - Replace current value $v$ by $x$

# Search trees, deleting a value . . .

- To delete a value $v$
- Start at the root
- At each node
  - If $v$ is smaller than the current value
    - If left child exists, delete $v$ from left subtree
  - If $v$ is larger than the current value
    - If right child exists, delete $v$ from right subtree
  - Otherwise, current value is $v$
    - If there is no left child, shift up right subtree to current node
    - If there is a left child
      - Delete maximum value $x$ from left subtree
      - Replace current value $v$ by $x$

# Search trees, deleting a value ...

- To delete a value $v$
- Start at the root
- At each node
  - If $v$ is smaller than the current value
    - If left child exists, delete $v$ from left subtree
  - If $v$ is larger than the current value
    - If right child exists, delete $v$ from right subtree
  - Otherwise, current value is $v$
    - If there is no left child, shift up right subtree to current node
    - If there is a left child
      - Delete maximum value $x$ from left subtree
      - Replace current value $v$ by $x$
- Worst case: Number of steps is equal to the longest path from the root to a leaf

# Deleting from a search tree

- deletetree — deletes a given value from the given tree and returns the resulting tree

```
deletetree :: Ord a => Stree a  -> a -> Stree a
deletetree Nil v = Nil
deletetree (Node tl y  tr) v
  | v < y   = Node (deletetree tl v)  y tr
  | v > y   = Node  tl  y (deletetree tr v)

-- In all cases below, we must have v == y

deletetree (Node Nil y tr) v   = tr
deletetree (Node  tl y tr) v = Node  tx x tr
  where (x,tx) = deletemax tl
```

# Balance

- The complexity of all the operations depend on the height of the tree.

# Balance

- The complexity of all the operations depend on the height of the tree.
- In general, a search tree will not be balanced

# Balance

- The complexity of all the operations depend on the height of the tree.
- In general, a search tree will not be balanced
- Inserting values in ascending or descending order results in highly skewed tree

```
          6
         /
        5
       /
      4
     /
    3
   /
  2
 /
1
```

# Balanced search trees . . .

- Ideally, for each node size of left and right subtrees differ by at most one

# Balanced search trees . . .

- Ideally, for each node size of left and right subtrees differ by at most one
  - Height of the tree is logarithmic in size.

# Balanced search trees ...

- Ideally, for each node size of left and right subtrees differ by at most one
  - Height of the tree is logarithmic in size.

    Lemma: Let $T$ be a size balanced tree with $s$ nodes. The height of $T$ is at most $log(s) + 1$.

# Balanced search trees ...

- Ideally, for each node size of left and right subtrees differ by at most one
  - Height of the tree is logarithmic in size.

    Lemma: Let $T$ be a size balanced tree with $s$ nodes. The height of $T$ is at most $log(s) + 1$.

    Proof: By induction on the size $s$.

# Balanced search trees . . .

- Ideally, for each node size of left and right subtrees differ by at most one
  - Height of the tree is logarithmic in size.

    Lemma: Let $T$ be a size balanced tree with $s$ nodes. The height of $T$ is at most $log(s) + 1$.

    Proof: By induction on the size $s$.
    Basis: If $s = 1$ then the tree has height $= 1$

# Balanced search trees . . .

- Ideally, for each node size of left and right subtrees differ by at most one
  - Height of the tree is logarithmic in size.

    Lemma: Let $T$ be a size balanced tree with $s$ nodes. The height of $T$ is at most $log(s) + 1$.

    Proof: By induction on the size $s$.
    Basis: If $s = 1$ then the tree has height $= 1$
    Induction: Let $T$ have $s$ nodes.

# Balanced search trees ...

- Ideally, for each node size of left and right subtrees differ by at most one
  - Height of the tree is logarithmic in size.

    Lemma: Let $T$ be a size balanced tree with $s$ nodes. The height of $T$ is at most $log(s) + 1$.

    Proof: By induction on the size $s$.
    Basis: If $s = 1$ then the tree has height $= 1$
    Induction: Let $T$ have $s$ nodes.
    Consider the left and right subtrees of this tree.

# Balanced search trees ...

- Ideally, for each node size of left and right subtrees differ by at most one
  - Height of the tree is logarithmic in size.

    Lemma: Let $T$ be a size balanced tree with $s$ nodes. The height of $T$ is at most $log(s) + 1$.

    Proof: By induction on the size $s$.
    Basis: If $s = 1$ then the tree has height $= 1$
    Induction: Let $T$ have $s$ nodes.
    Consider the left and right subtrees of this tree.
    The largest of them has at most $s/2$ nodes. (Why?)

# Balanced search trees ...

- Ideally, for each node size of left and right subtrees differ by at most one
  - Height of the tree is logarithmic in size.

    Lemma: Let $T$ be a size balanced tree with $s$ nodes. The height of $T$ is at most $log(s) + 1$.

    Proof: By induction on the size $s$.
    Basis: If $s = 1$ then the tree has height $= 1$
    Induction: Let $T$ have $s$ nodes.
    Consider the left and right subtrees of this tree.
    The largest of them has at most $s/2$ nodes. (Why?)
    So, both subtrees have height at most

$$log(s/2) + 1 \ = \ (log(s) + 1) - 1$$

# Balanced search trees ...

- Ideally, for each node size of left and right subtrees differ by at most one
  - Height of the tree is logarithmic in size.

    Lemma: Let $T$ be a size balanced tree with $s$ nodes. The height of $T$ is at most $log(s) + 1$.

    Proof: By induction on the size $s$.
    Basis: If $s = 1$ then the tree has height $= 1$
    Induction: Let $T$ have $s$ nodes.
    Consider the left and right subtrees of this tree.
    The largest of them has at most $s/2$ nodes. (Why?)
    So, both subtrees have height at most

$$log(s/2) + 1 \;=\; (log(s) + 1) - 1$$

- However, it is not easy to maintain size-balance.

# Height-balanced trees

- Maintain height balanced trees instead of size-balanced trees.
  - Height of left subtree and height of right subtree differ by at most one at any node.

# Height-balanced trees

- Maintain height balanced trees instead of size-balanced trees.
  - Height of left subtree and height of right subtree differ by at most one at any node.
- Height is still logarithmic in size [Adelson-Velskii, Landis]

# Height-balanced trees

- Maintain height balanced trees instead of size-balanced trees.
  - Height of left subtree and height of right subtree differ by at most one at any node.
- Height is still logarithmic in size [Adelson-Velskii, Landis]
- Somewhat easier to maintain.

# Height balanced trees ...

```
    4
   / \
  2   5
 / \   \
1   3   6
```

# Height balanced trees ...

```
      4
     / \
    2   5
   / \   \
  1   3   6
```

- A height and weight balanced tree.

# Height balanced trees ...

```
        4
       / \
      2   5
     / \   \
    1   3   6
```

- A height and weight balanced tree.

```
        4
       / \
      2   5
     / \
    1   3
```

# Height balanced trees ...

```
      4
     / \
    2   5
   / \   \
  1   3   6
```

- A height and weight balanced tree.

```
      4
     / \
    2   5
   / \
  1   3
```

- A height balanced tree that is not weight balanced.

# Height Balanced Trees ...

# Height Balanced Trees ...

- Let $S(h)$ be the size of the smallest height balanced tree with height $h$.

# Height Balanced Trees ...

- Let $S(h)$ be the size of the smallest height balanced tree with height $h$.
  - $S(0) = 0$

# Height Balanced Trees ...

- Let $S(h)$ be the size of the smallest height balanced tree with height $h$.
  - $S(0) = 0$
  - $S(1) = 1$

# Height Balanced Trees ...

- Let $S(h)$ be the size of the smallest height balanced tree with height $h$.
  - $S(0) = 0$
  - $S(1) = 1$
  - $S(2) = 2$

# Height Balanced Trees ...

- Let $S(h)$ be the size of the smallest height balanced tree with height $h$.
  - $S(0) = 0$
  - $S(1) = 1$
  - $S(2) = 2$
- If $T$ is a height balanced tree of height $h$, then one of its two subtrees has height $h - 1$ and the other has height at least $h - 2$. So,

# Height Balanced Trees ...

- Let $S(h)$ be the size of the smallest height balanced tree with height $h$.
  - $S(0) = 0$
  - $S(1) = 1$
  - $S(2) = 2$
- If $T$ is a height balanced tree of height $h$, then one of its two subtrees has height $h-1$ and the other has height at least $h-2$. So,

$$S(h) \geq S(h-1) + S(h-2) + 1$$

# Height Balanced Trees ...

- Let $S(h)$ be the size of the smallest height balanced tree with height $h$.
  - $S(0) = 0$
  - $S(1) = 1$
  - $S(2) = 2$
- If $T$ is a height balanced tree of height $h$, then one of its two subtrees has height $h - 1$ and the other has height at least $h - 2$. So,
$$S(h) \geq S(h-1) + S(h-2) + 1$$
- Grows like the Fibonacci numbers, exponentially.

# Height Balanced Trees ...

- Let $S(h)$ be the size of the smallest height balanced tree with height $h$.
  - $S(0) = 0$
  - $S(1) = 1$
  - $S(2) = 2$
- If $T$ is a height balanced tree of height $h$, then one of its two subtrees has height $h-1$ and the other has height at least $h-2$. So,

  $$S(h) \geq S(h-1) + S(h-2) + 1$$
- Grows like the Fibonacci numbers, exponentially.
- $S(h) \geq 2^{h/1.44}$ or equivalently

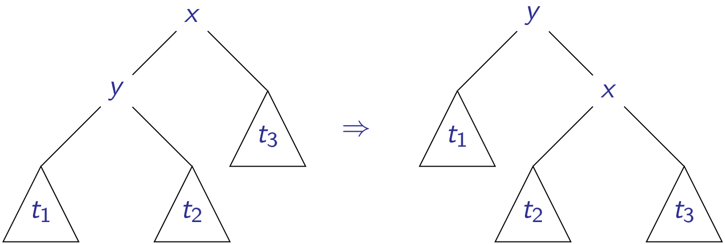
$$h(T) \leq 1.44 log(s(T))$$

# Height Balanced trees . . .

- Use tree rotations to maintain height balance

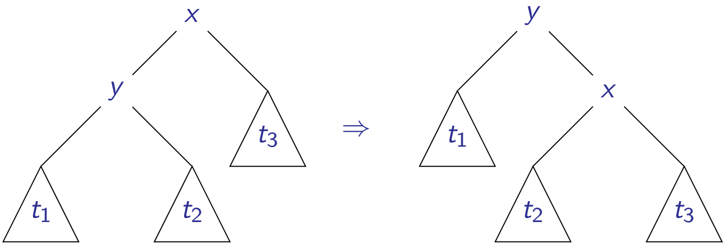
# Height Balanced trees . . .

- Use tree rotations to maintain height balance
- Example: rotate right



# Height Balanced trees . . .

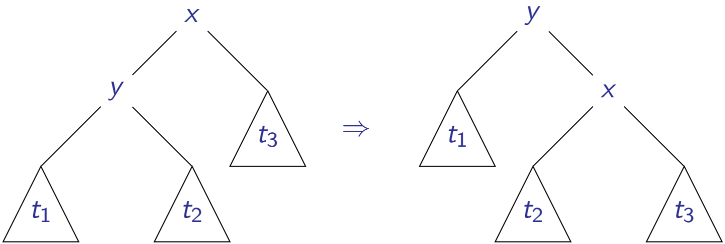
- Use tree rotations to maintain height balance
- Example: rotate right




- Useful if `t1` has large height.

# Height Balanced trees ...

- Use tree rotations to maintain height balance
- Example: rotate right

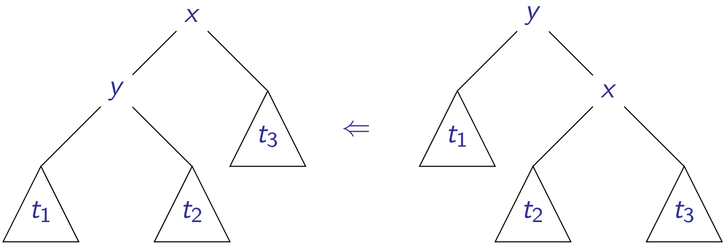



- Useful if `t1` has large height.

- ```
  rotateright (Node (Node t1 y t2) x t3) =
                                  Node t1 y (Node t2 x t3)
  ```
  ```

# Balanced search trees ...

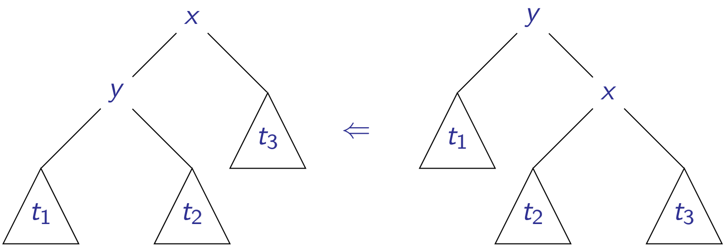
- Use tree rotations to maintain height balance
- Example: rotate left




- Useful if `t3` has large height.

# Balanced search trees ...

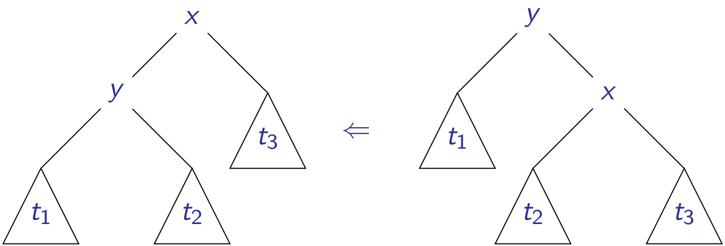
- Use tree rotations to maintain height balance
- Example: rotate left




- Useful if `t3` has large height.
- ```
  rotateleft (Node t1 y (Node t2 x t3)) =
                           Node (Node t1 y t2) x t3
  ```
  ```

# Balanced search trees ...

- Use tree rotations to maintain height balance
- Example: rotate left




- Useful if `t3` has large height.

- ```
  rotateleft (Node t1 y (Node t2 x t3)) =
                               Node (Node t1 y t2) x t3
  ```
  ```