Introduction to Programming: Lecture 12

K Narayan Kumar

Chennai Mathematical Institute

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

17 Sep 2013

Binary Search Trees

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

Binary Search Trees

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

Binary Search Trees

- 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
```

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.

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

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.

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

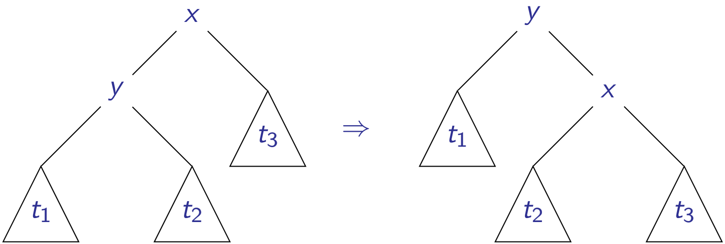
- Height is still logarithmic in size [Adelson-Velskii, Landis]
- Somewhat easier to maintain.

Height Balanced trees . . .

- Use tree rotations to maintain height balance

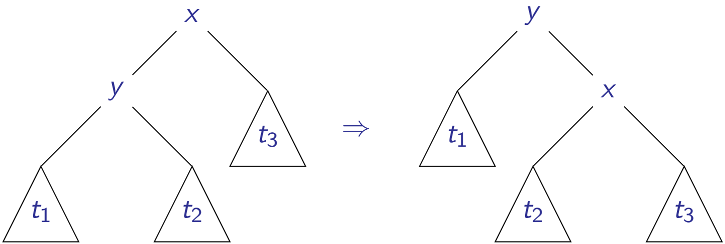
Height Balanced trees ...

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



Height Balanced trees ...

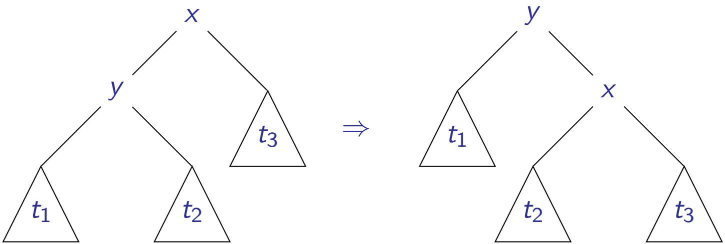
- Use tree rotations to maintain height balance
- Example: rotate right



- Useful if `t1` has large height.

Height Balanced trees ...

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

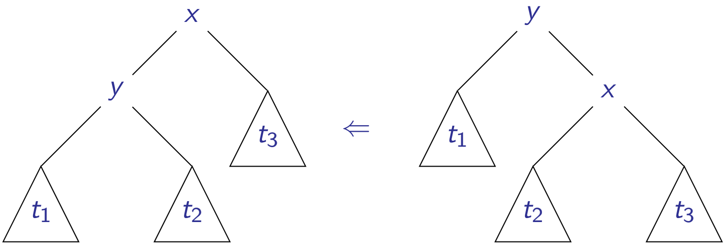


- Useful if `t1` has large height.

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

Balanced search trees . . .

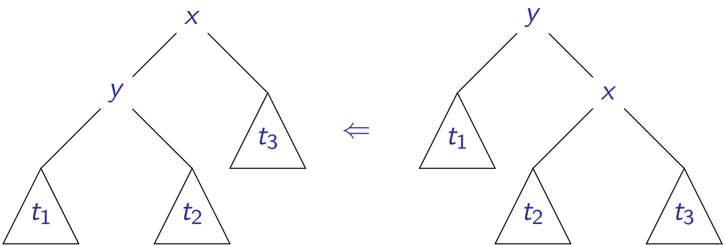
- Use tree rotations to maintain height balance
- Example: rotate left



- 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 . . .

- Assume tree is currently balanced
- Each `inserttree` or `deletetree` operation creates limited imbalance

Balanced search trees ...

- Assume tree is currently balanced
- Each `inserttree` or `deletetree` operation creates limited imbalance
- Fix imbalance using `rebalance` (to be written!)

Balanced search trees . . .

- Assume tree is currently balanced
- Each `inserttree` or `deletetree` operation creates limited imbalance
- Fix imbalance using `rebalance` (to be written!)
- Assuming `rebalance` exists, we can write

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

Rebalancing search trees

- Define `slope (Node t1 x t2)` =
  `(height t1) - (height t2)`

Rebalancing search trees

- Define `slope (Node t1 x t2)` =
  `(height t1) - (height t2)`
  - In a height balanced tree, slope is -1, 0 or 1

Rebalancing search trees

- Define `slope (Node t1 x t2) = (height t1) - (height t2)`
  - In a height balanced tree, slope is -1, 0 or 1
  - After inserting/deleting, slopes can be -2,-1, 0, 1 or 2. Violation happens only at nodes visited by the operation.

Rebalancing search trees

- Define `slope (Node t1 x t2)` =
  `(height t1) - (height t2)`
  - In a height balanced tree, slope is -1, 0 or 1
  - After inserting/deleting, slopes can be -2,-1, 0, 1 or 2. Violation happens only at nodes visited by the operation.
- We need to rebalance nodes whose slopes are -2 or 2 (and further assume that the sub-trees below are balanced)

Rebalancing search trees

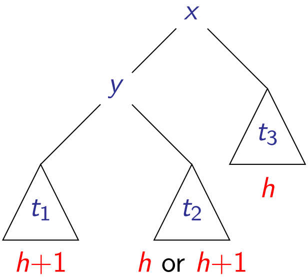
- Define `slope (Node t1 x t2) =` `(height t1) - (height t2)`
  - In a height balanced tree, slope is -1, 0 or 1
  - After inserting/deleting, slopes can be -2,-1, 0, 1 or 2. Violation happens only at nodes visited by the operation.
- We need to rebalance nodes whose slopes are -2 or 2 (and further assume that the sub-trees below are balanced)
- We consider the case where slope is 2
  - Slope -2 is symmetric

Rebalancing a node with slope 2

- Two cases
  - Slope at root of left subtree is 0 or 1
  - Slope at root of left subtree is -1

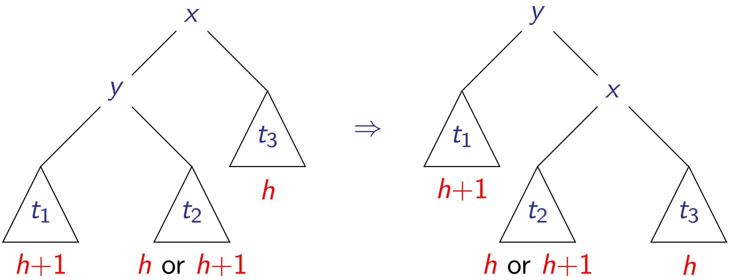
Rebalancing a node with slope 2

- Two cases
  - Slope at root of left subtree is 0 or 1
  - Slope at root of left subtree is -1
- Case 1:



Rebalancing a node with slope 2

- Two cases
  - Slope at root of left subtree is 0 or 1
  - Slope at root of left subtree is -1
- Case 1: Rotate right

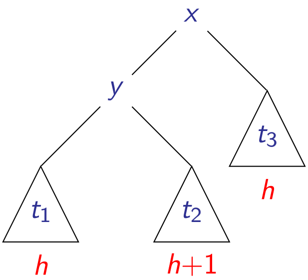


Rebalancing a node with slope 2 . . .

- Case 2: Slope at root of left subtree is -1

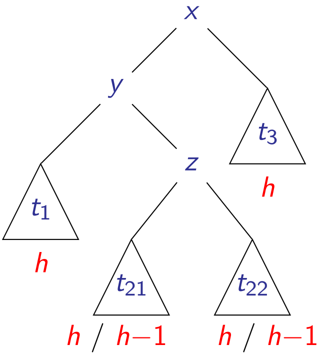
Rebalancing a node with slope 2 . . .

- Case 2: Slope at root of left subtree is -1



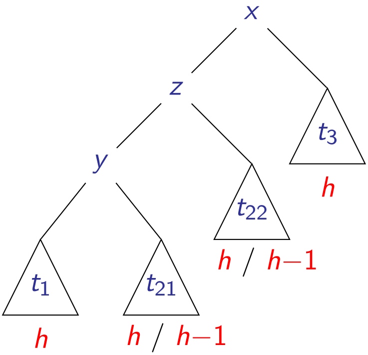
Rebalancing a node with slope 2 ...

- Case 2: Slope at root of left subtree is -1
  - Expand `t2`



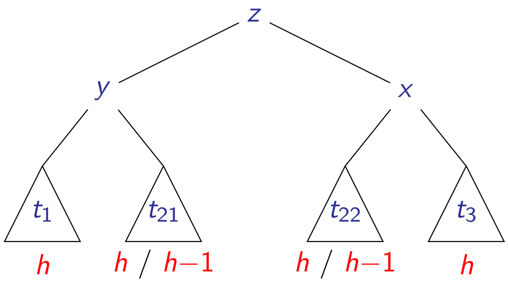
Rebalancing a node with slope 2 . . .

- Case 2: Slope at root of left subtree is -1
  - Expand t2
  - Rotate left at y—note that z may now be unbalanced



Rebalancing a node with slope 2 . . .

- Case 2: Slope at root of left subtree is -1
  - Expand `t2`
  - Rotate left at `y`—note that `z` may now be unbalanced
  - Right rotate at `x`—now `z` must be balanced



The `rebalance` function

```
rebalance :: Ord a =>  Stree a  -> Stree a
rebalance (Node t1 y t2)
  | abs (sy) < 2          = Node t1 y t2
-- Slope = 2
  | sy == 2 && st1 /= -1 = rotateright (Node t1 y t2)
  | sy == 2 && st1 == -1 =
              rotateright (Node (rotateleft t1) y t2)
  ...
  where
    sy  = slope (Node t1 y t2)
    st1 = slope t1
    ...
```

Computing the slope

- How do we compute `slope`?

Computing the slope

- How do we compute `slope`?
- Naively

```
slope :: Stree a  -> Int
slope Nil = 0
slope (Node t1 x t2) = (height t1) - (height t2)
```

Computing the slope

- How do we compute `slope`?
- Naively

```
slope :: Stree a  -> Int
slope Nil = 0
slope (Node t1 x t2) = (height t1) - (height t2)
```

- To compute `height`, we examine each node in the tree
- Computing `slope` proportional to size, not height!

Balanced search trees ...

- Instead, store height at each node
- Modify definition of `Stree` to add an extra `Int` to record height

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

Balanced search trees ...

- Instead, store height at each node
- Modify definition of `Stree` to add an extra `Int` to record height

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

- Now, computing `height` and `slope` takes constant time

```
height Nil = 0
height (Node t1 x m t2) = m

slope Nil = 0
slope (Node t1 x m t2) = (height t1) - (height t2)
```

Insertion

```
inserttree :: Ord a => BStree a -> a -> BStree a

inserttree Nil x = Node  Nil x 1  Nil
inserttree (Node  tl y h tr) x
  | x == y    = Node tl y h tr
  | x < y     = rebalance (Node ntl y nhl tr)
  | otherwise = rebalance (Node tl y nhr ntr)
    where
      ntl = inserttree tl x
      nhl  = 1 + max (height ntl) (height tr)
      ntr = inserttree tr x
      nhr = 1 + max (height tl) (height ntr)
```

Deletion

```
deletetree :: Ord a => BStree a -> a -> BStree  a

deletetree Nil x = Nil
deletetree (Node tl y h tr) x
  | x < y    = rebalance (Node ntl y nhl tr)
  | x > y    = rebalance (Node tl y nhr ntr)
    where
     ntl = deletetree tl x
     nhl = 1 + max (height ntl) (height tr)
     ntr =  deletetree tr x
     nhr  = 1 + max (height tl) (height ntr)
```

Cont'd ...

Deletion ...

```
-- In all cases below, we must have x == y

deletetree (Node Nil y h tr) x    = tr
deletetree (Node tl y h tr) x =
        rebalance (Node tz z nh tr)
   where
     (z,tz) = deletemax tl
     nh = 1 + max (height tz) (height tr)
```

Deletemax

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

deletemax (Node  tl y h Nil) = (y,tl)

deletemax (Node tl y h tr) =
          (z, rebalance (Node tl y nh tz))
          where
            (z,tz) = deletemax tr
            nh     = 1 + max (height tl) (height tz)
```

Searching

```
searchtree  :: Ord a => BStree a -> a -> Bool

searchtree Nil v = False
searchtree  (Node tl y h tr)  v
     | v == y       = True
     | v < y        = searchtree  tl v
     | v > y        = searchtree  tr v
```

Rebalance:

```
rebalance :: Ord a =>  Stree a  -> Stree a
rebalance (Node t1 y h t2)
  | abs (sy) < 2          = Node t1 y h t2
  | sy == 2 && st1 /= -1 = rotateright (Node t1 y h t2)
  | sy == 2 && st1 == -1 =
              rotateright (Node (rotateleft t1) y nya  t2)
  ...
  where
    sy  = slope (Node t1 y h t2)
    nya = 1 + max (height t2) (height (rotateleft t1))
    st1 = slope t1
    ...
```

Rotations

```
rotateright (Node (Node t1 y hy t2) x hx t3) =
             Node t1 y nhy (Node t2 x nhx t3)
              where
               nhx = 1 + max (height t2) (height t3)
               nhy = 1 + max (height t1) nhx
```

...

Anonymous Functions

- So far, every Haskell function had a name.

Anonymous Functions

- So far, every Haskell function had a name.
- Anonymous functions are functions without names.

Anonymous Functions

- So far, every Haskell function had a name.
- Anonymous functions are functions without names.
- Useful, for instance, if a function is needed just once.

Anonymous Functions

- So far, every Haskell function had a name.
- Anonymous functions are functions without names.
- Useful, for instance, if a function is needed just once.

  ```
  \x -> (head (tail x))
  ```

Anonymous Functions

- So far, every Haskell function had a name.
- Anonymous functions are functions without names.
- Useful, for instance, if a function is needed just once.

  ```
  \x -> (head (tail x))
  ```

- The `\x` is to be read as lambda x and comes from λ-Calculus.

Anonymous Functions

- So far, every Haskell function had a name.
- Anonymous functions are functions without names.
- Useful, for instance, if a function is needed just once.

  ```
  \x -> (head (tail x))
  ```
- The `\x` is to be read as lambda x and comes from λ-Calculus.
- Writing ...

  ```
  allseconds  = map (\x -> head (tail x))
  ```

Anonymous Functions

- So far, every Haskell function had a name.
- Anonymous functions are functions without names.
- Useful, for instance, if a function is needed just once.

```
\x -> (head (tail x))
```

- The \x is to be read as lambda x and comes from λ-Calculus.

- Writing ...

```
allseconds  = map (\x -> head (tail x))
```

... is better than writing

```
allseconds  = map second
 where
 second x  = head (tail x)
```

Functional Composition

- Composing functions is a natural operation.

```
second x  = head (tail x)
```

Functional Composition

- Composing functions is a natural operation.

  ```
  second x  = head (tail x)
  ```

- Given `f :: a -> b` and `g :: b -> c` their composition is defined in Haskell using the `.` operator:

  ```
  g.f :: a -> c
  (g.f) x = g (f x)
  ```

Functional Composition

- Composing functions is a natural operation.

```
second x  = head (tail x)
```

- Given `f :: a -> b` and `g :: b -> c` their composition is defined in Haskell using the `.` operator:

```
g.f :: a -> c
(g.f) x = g (f x)
```

- `second = head.tail`

Functional Composition

- Composing functions is a natural operation.

```
second x  = head (tail x)
```

- Given `f :: a -> b` and `g :: b -> c` their composition is defined in Haskell using the `.` operator:

```
g.f :: a -> c
(g.f) x = g (f x)
```

- `second = head.tail`
- `.` is a function like any other in Haskell and can be easily defined.

Functional Composition

- Composing functions is a natural operation.

  ```
  second x  = head (tail x)
  ```

- Given `f :: a -> b` and `g :: b -> c` their composition is defined in Haskell using the `.` operator:

  ```
  g.f :: a -> c
  (g.f) x = g (f x)
  ```

- `second = head.tail`
- `.` is a function like any other in Haskell and can be easily defined.
- What is its type?

Examples

- Here is a more elaborate example that counts the number of words that begin with a captial letter:

Examples

- Here is a more elaborate example that counts the number of words that begin with a captial letter:
- `words :: String -> [String]` returns the words in the given `String`

Examples

- Here is a more elaborate example that counts the number of words that begin with a captial letter:
- `words :: String -> [String]` returns the words in the given `String`

```
capcount = length . filter (isUpper.head) . words
```

The `if-then-else` function

- The `if-then-else` construct works as one expects it to:

The `if-then-else` function

- The `if-then-else` construct works as one expects it to:

  `if b then e1 else e2`

  If the value of `b`, a boolean expression, is `True` then the value of the expression is `e1` else `e2`

The `if-then-else` function

- The `if-then-else` construct works as one expects it to:

  `if b then e1 else e2`

  If the value of `b`, a boolean expression, is `True` then the value of the expression is `e1` else `e2`

```
fact n = if (n==0) 1 else (fact (n-1))*n
```

The `let` construct

- Allows for the introduction of local names much as the `where` command does.

The `let` construct

- Allows for the introduction of local names much as the `where` command does.

```
let f x = x*x in
  map f [1,2,3]
```

The `let` construct

- Allows for the introduction of local names much as the `where` command does.

```
let f x = x*x in
  map f [1,2,3]
```

- There are differences betweeen `let` and `where` but we do not discuss them here.