Updated on April 4th 2009, Saturday 01:00

CENG 242

Homework #3
(Due: April 14th 2009, Tuesday 23:55)

In this homework you will implement an abstract data type `QuadTree` for Quad trees in Haskell. You will implement interface functions to manipulate this data type. In addition, you will be required operator instances so that `QuadTree` will be `overloading (+,-,*,negate) functions of the Num class` and be a member of `Show` class.

The region quadtree represents a partition of space in two dimensions by decomposing the region into four equal quadrants, subquadrants, and so on with each leaf node containing data corresponding to a specific subregion. Each node in the tree either has *exactly four children*, or has *no children* (a leaf node). [1]

We will not give you the type signature of `QuadTree`. You will implement it anyway you want and hide the type internals. That's the spirit of abstract data type as you shall remember.

Your module should look like:

```
module QuadTree (QuadTree, interface functions) where
   data QuadTree = internal definition of type
    definitions of interface and auxiliary functions
```

interface functions are the interface functions the module will export. These functions are explained below. *internal definitions of type* can be anything useful. You will not export any type constructor given here.

You will implement the following interface functions:

new n

new :: Integer → QuadTree

Creates an initial QuadTree using (0,0) and (n-1,n-1) as diagonal corners where,

(0,0) → bottom-left point

(n-1,n-1) → top-right point

(0,n-1) → top-left point

(n-1,0) → bottom-right point.

insert (point1,point2) tree

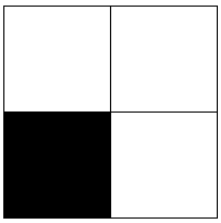
insert :: ((Integer,Integer),(Integer,Integer))->QuadTree->QuadTree

Gets a rectangle with diagonal corner points point1 and point2 and inserts this rectangle to the given QuadTree. Keep in mind that (point1,point2) can be (top-right,bottom-left) or (bottom-left,top-right).

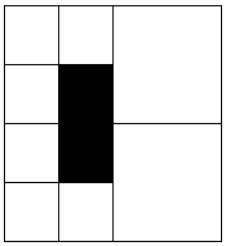
Updated on April 4th 2009, Saturday 01:00

Examples:

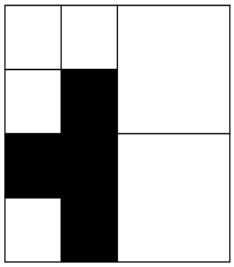
insert ((0,0), (3,3)) (new 8)



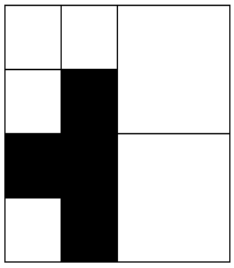
insert ((3,5) , (2,2)) (new 8)



insert ((1,3) , (0,2)) (insert ((2,0) , (3,5)) (new 8))



insert ((0,8) , (15,15)) (insert ((8,0) , (15,23)) (new 32))



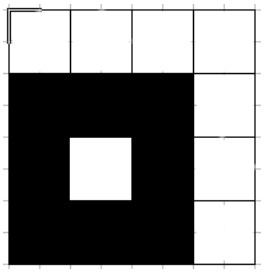
Updated on April 4th 2009, Saturday 01:00

delete (point1,point2) tree

delete :: ((Integer,Integer),(Integer,Integer))->QuadTree->QuadTree

Gets a rectangle with diagonal corner points point1 and point2 and deletes this rectangle from the given QuadTree.

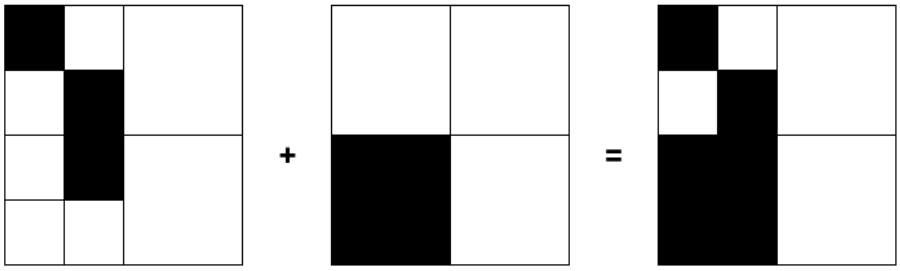
delete ((256,256) , (511,511)) (insert ((767,767) , (0,0)) (new 1024))



(+) tree1 tree2

(+) :: QuadTree -> QuadTree -> QuadTree

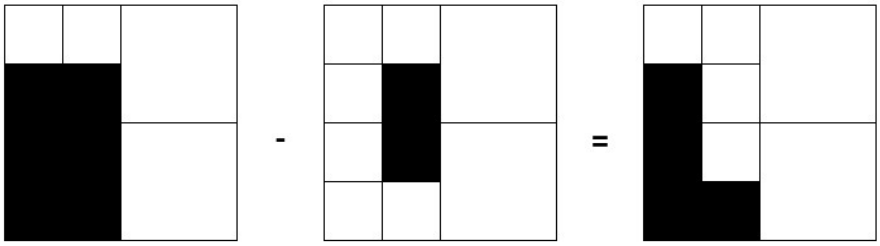
Returns the union of two QuadTrees.



(-) tree1 tree2

(-) :: QuadTree -> QuadTree -> QuadTree

Returns the difference of two QuadTrees.

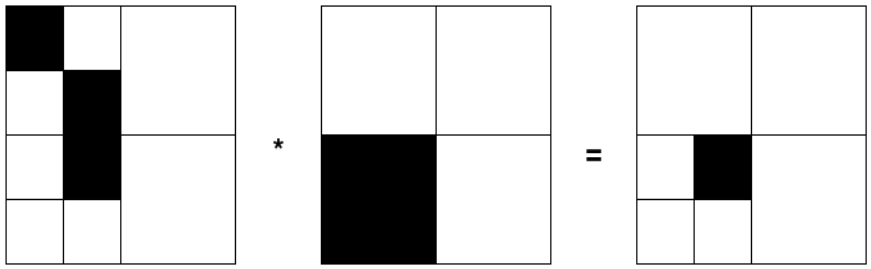


Updated on April 4th 2009, Saturday 01:00

(*) tree1 tree2

(*) :: QuadTree -> QuadTree -> QuadTree

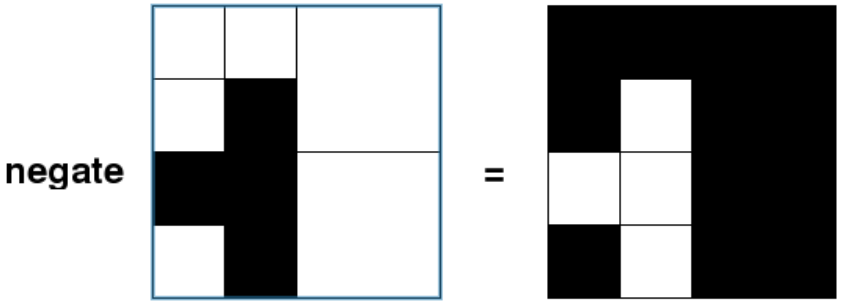
Returns the intersection of two QuadTrees.



negate tree

negate :: QuadTree->QuadTree

Returns the complement of a QuadTree.



show tree

`show` is inherited from `Show` class. You will implement `QuadTree` as an instance of this class. Type signature of `show` is given as:

show :: Show a => a → String

You will implement the instance as:

instance Show QuadTree

Updated on April 4th 2009, Saturday 01:00

The output string will be in Lisp-like syntax with some indentation. Leaf nodes are shown as: **((xmin,ymin),(xmax,ymax) Empty)** or **((xmin,ymin),(xmax,ymax) Full)**

where (x1,y1), (x2,y2) are the corners of the leaf node.

Internal nodes are shown as:

```
(
        child1 (north east)
        child2 (north west)
        child3 (south west)
        child4 (south east)
)
```

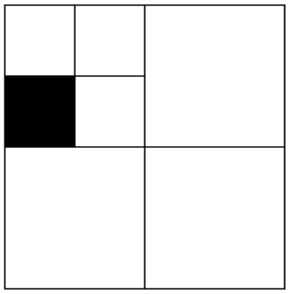
new 10 -- this line is a command

((0,0) (9,9) Empty)

insert ((7,23) , (0,16)) (new 32) --this line is a command

```
(
        ((16,16) (31,31) Empty)
        (
                ((8,24) (15,31) Empty)
                ((0,24) (7,31) Empty)
                ((0,16) (7,23) Full)
                ((8,16) (15,23) Empty)
        )
        ((0,0) (15,15) Empty)
        ((16,0) (31,15) Empty)
)
```

Corresponding quadtree is:



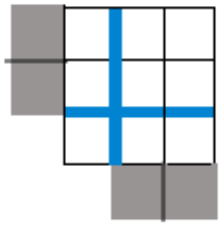
Updated on April 4th 2009, Saturday 01:00

Notes

- Check the size compatibility of QuadTrees with guards in your operator definitions. Your program should give a pattern match error for QuadTree operations on incompatible trees.
- When inserted/deleted rectangles are off the boundary, use the part that stays between boundaries for further steps. If the whole rectangle is off boundary, you can just return the QuadTree without change.
- Put the module and nothing else in file QuadTree`.hs`. You can have any auxiliary definitions under `where` in module definition as long as they are not exported outside.

- **IMPORTANT**: When you cannot divide the QuadTree area into 4 equal parts, divide the area in this manner (Xdim,Ydim represents X and Y dimension size of the area):
  - top-right → biggest area. Corners: (Xdim/2,Ydim/2) (Xdim-1,Y-dim-1)
  - bottom-left → smallest area. Corners: (0,0) (Xdim/2-1,Ydim/2-1)
  - top-left / bottom-right → equal areas. bigger than bottom-left, smaller than top-right.
- When splitting 1x2 and 2x1 areas, use null leaves for non-existing points of the square:
  - For 1x2 area:
    - upper square → top-right
    - lower square → bottom-right
  - For 2x1 area:
    - left square → top-left
    - right square → top-right

  - Related 3x3 quadtree diagram (grey squares are null leaves):



  - When showing these QuadTrees with null leaves, you won't print anything for null leaves:
    - insert ((0,0) , (0,1)) (new 3) -- this line is a command

```
(
        ((1,1) (2,2) Empty)
        (
                ((0,2) (0,2) Empty)
                ((0,1) (0,1) Full)
        )
        ((0,0) (0,0) Full)
        ((1,0) (2,0) Empty)
)
```

Updated on April 4th 2009, Saturday 01:00

Specifications

- All the work should be done **individually**.
- Your codes should be written in Haskell and have the name "**QuadTree.hs**"
- In evaluation, black box method will be used. So be careful about the name of functions, data structures etc.
- You will submit your code through **Cow** system.
- You should test your codes in **inek** machines with **hugs** before submitting.

References

[1] - http://en.wikipedia.org/wiki/Quadtree#The_region_quadtree