

DM22 - PROGRAMMERINGSSPROG

---

# Haskell Projekt 1

---

Dictionary and  
Orthogonal Line Segment Intersection

JACOB AAE MIKKELSEN  
"KOKKEN"

19 10 76

20. marts 2007

## Indhold

<b>1</b>	<b>Introduktion</b>	<b>2</b>
<b>2</b>	<b>Kravspecifikation</b>	<b>3</b>
2.1	Del 1: Dictionary . . . . .	3
2.2	Del 2: Skæringspunkter af vinkelrette linjer . . . . .	3
<b>3</b>	<b>Design</b>	<b>4</b>
3.1	Del 1: Dictionary . . . . .	4
3.2	Del 2: Skæringspunkter af vinkelrette linjer . . . . .	4
<b>4</b>	<b>Implementation</b>	<b>5</b>
4.1	Del 1: Dictionary . . . . .	5
4.2	Del 2: Skæringspunkter af vinkelrette linjer . . . . .	6
<b>5</b>	<b>Test</b>	<b>8</b>
5.1	Dictionary . . . . .	8
5.2	Mergesort . . . . .	8
5.3	Line-sweep algoritmen . . . . .	8
<b>6</b>	<b>Konklusion</b>	<b>13</b>
<b>7</b>	<b>Litteratur</b>	<b>14</b>
<b>A</b>	<b>Source Code</b>	<b>15</b>

## 1 Introduktion

Denne rapport dokumenterer den første obligatoriske aflevering i faget DM22 - Programmeringssprog på IMADA, SDU, forårssemestret 2007.

Opgaven består af to dele. I den første del skal der implementeres et 'Dictionary' med de tilhørende funktioner. Den anden del består af at implementere en algoritme der finder skæringer mellem en mængde vandrette og lodrette linjer.

Som en del af anden del skal en  $O(n \log n)$  sorteringsalgoritme implementeres.

Alle dele er testet, og beskrevet i afsnittet test.

## 2 Kravspecifikation

### 2.1 Del 1: Dictionary

Der skal i første del implementeres et Haskell modul af typen `Dictionary`, der har følgende funktioner:

- `makeDict()`
- `isEmpty(D)`
- `search(k,D)`
- `delete(k,D)`
- `rangeSearch(k1,k2,D)`

Der må benyttes at alle nøglerne er forskellige.

Modulet skal implementeres som et balanceret søgetræ af typen *locally rebuilt weight-balanced tree*. I denne type træ gælder at antallet af knuder i det ene undertræ højst er to gange antallet af knuder i det andet undertræ.

Funktionerne `makeDict` og `isEmpty` skal køre i konstant tid, og `search` logaritmisk tid. Desuden skal `insert` og `delete` udføres i amortiseret logaritmisk tid, hvilket kræver en algoritme der kører i lineær tid af antal knuder i undertræet, til at genopbygge træet til balanceret. For `rangeSearch` er kravet  $O(k + \log n)$ , hvor  $k$  er størrelsen af output.

Implementationen skal testes internt i modulet men kun de beskrevne metoder skal exporteres.

### 2.2 Del 2: Skæringspunkter af vinkelrette linjer

Problemet består af, givet to mængder af linjestykker, én mængde med lodrette stykker og én med vandrette stykker, at finde skæringspunkterne mellem linjestykkerne i de to mængder.

Der skal implementeres en 'sweep-line' algoritme, der bevæger en linje, parallel med y-aksen, langs x-aksen, og på den måde registrerer når der mødes skæringspunkter. Ved denne teknik er der så tre punkter der er interessante at se på:

**Venstre endepunkt af en vandret linje** Her begynder dette linjestykke, og støder man efterfølgende på lodrette stykker, kan der muligvis være skæring. Dette linjestykke bliver dermed aktivt i algoritmen.

**Højre endepunkt af en vandret linje** Linjestykket kan efter dets højre endepunkt er mødt ikke længere skære med lodrette linjer, og det bliver dermed inaktivt i algoritmen.

**Lodrette linjestykker** Da de lodrette linjestykker er parallelle med 'sweep-line' opträder de bare én gang. Her skal der så kontrolleres for skæring med de aktive lodrette linjestykker.

Det er naturligt at benytte det netop implementerede `Dictionary`, hvor horisontale linjestykker indsættes med y-koordinaten når det venstre endepunkt mødes, og slettes når det højre endepunkt mødes. Når et lodret linjestykke mødes, laves en `rangeSearch` med y-koordinaterne, der så vil returnere de vandrette linjestykker der krydser den.

## 3 Design

### 3.1 Del 1: Dictionary

Det er naturligt at designe dictionary'et så det kan bruges direkte til line-sweep algoritmen. Da vi skal bruge `isEmpty` som metode, er det naturligt at lade bladene være tomme og ikke indeholde information. Dermed vil et tomt dictionary kun indeholde ét blad. Bladene har selvfølgeligt ingen undertræer, og deres vægt bliver derfor nul.

De interne knuder skal derfor indeholde deres vægt, d.v.s. antallet af knuder i deres undertræer, knudens egen data, samt pointere til hvert af venstre og højre undertræ.

Metoden `makeDict` skal blot returnere et tomt dictionary. Dette kan i Haskell implementeres ved at kalde funktionen med en tom liste. Dette udvides i denne opgave til at kunne bygge et balanceret træ, hvis der er elementer i listen, dog skal listen være i sorteret rækkefølge. Der kunne uden problemer implementeres sortering som en del af opbygningen, men dette vil konflikte med kravet til køretid for genopbygningen af træet efter `insert` eller `delete`.

Søgning skal kaldes med et dictionary og en søgenøgle, og bare returnere elementet. Der er ingen grund til at returnere dictionary'et også, men det kunne lade sig gøre ved at pakke resultatet ind i en tupel.

Funktionaliteten i `insert` og `delete` laves rekursivt, med kald til en hjælpfunktion rebuilt, der sørger for at rebalancere hvis vægtningen i træet overtrædes. For `rangeSearch` er det også naturligt at kalde rekursivt, med en hjælpfunktion som argument tager et dictionary og en liste til fundne elementer.

I rebuilt skal der, hvis vægtningen er forkert, genopbygges et træ til perfekt balance. Dette kan gøres ved at lave et kald til en flatten metode, der i lineær tid laver et dictionary til en liste, med et efterfølgende kald til `makeDict`, hvor træet bygges, igen i lineær tid. Da vi ved træet er sorteret, vil flatten med et inorder gennemløb returnere en sorteret liste, og `makeDict` vil derfor kun blive kaldt med en sorteret liste.

### 3.2 Del 2: Skæringspunkter af vinkelrette linjer

Det første der skal gøres er at lave en liste med alle de punkter der er 'interessante' dvs begge endepunkter af de vandrette og de lodrette, alle med y-koordinaten som primær sorterings nøgle. Det andet der skal sorteres efter er i rækkefølge:

1. Venstre endepunkt af en vandret linje
2. lodret linje
3. højre endepunkt af en vandret linje

Årsagen ses let ud fra figur 7. Det højre vandrette linjestykke skal blive aktivt først, så skal der testes for linjepunkter via det lodrette, og så kan det venstre af de to vandrette fjernes og blive inaktiv.

For at lave denne sortering laves et sorteringsmodul, og Mergesort er en simpel  $O(n \log n)$  sorteringsalgoritme. Skulle `makeDict` ombygges så den kan lave et sorteret dictionary af en ikke sorteret liste ville dette modul sagtens kunne bruges her også.

## 4 Implementation

### 4.1 Del 1: Dictionary

Den første del af at opbygge dictionary'et som et træ er at implementere blade og knuder.

```
data Dict a = Leaf | Node Int a (Dict a)(Dict a) deriving Show
```

#### Funktionen makeDict

Implementationen af `makeDict` laver et blad uden information hvis den er kaldt med en tom liste. Hvis listen ikke er tom, benyttes funktionen `mkDictNode` til at lave en knude af det midterste element og et rekursivt kald til hver af listerne på hver sin side. `mkDictNode` benytter så `getSize` til at få vægten rigtig. Disse hjælpemетодer er ret trivielle.

#### Funktionen isEmpty

Da bladene ikke indeholder information, returnerer `isEmpty` sandt hvis den bliver kaldt med et blad, og falskt ellers. Søgning er ligeledes simpel, hvis ikke nøglen i noden er lig med det der søges efter, kaldes rekursivt med venstre undertræ, hvis nøglen er større end, og rekursivt med højre undertræ hvis nøglen er mindre end.

#### Funktionen insert

Funktionen `insert` erstatter et blad med den indsatte knude, og efter det kaldes `rebuilt` på det opdaterede træ. `rebuilt` benytter metoden `problem` der kontrollerer om det er nødvendigt at opbygge træet på, og metoden `balance`.

`balance` er implementeret ved at tage træet, lave en liste ud af det med `flatten`:

```
flatten :: Ord a => Dict(a,b) -> [(a,b)]
flatten x = flatHelp x []

flatHelp :: Ord a => Dict (a,b) -> [(a,b)] -> [(a,b)]
flatHelp Leaf xs = xs
flatHelp (Node i x lt rt) xs = flatHelp lt (x:xs)
    where ys = flatHelp rt xs
```

Her benyttes accumulator metoden, i `flatHelp`, hvor der sendes et dictionary og en liste med, og elementerne indsættes i listen efterhånden.

Det giver os dictionary'et som en sorteret liste, og nu kan vi bruge `makeDict` til at lave et balanceret træ:

```
balance x = makeDict (flatten x)
```

#### Funktionen delete

Måden `delete` er implementeret på minder om `insert`. Den søger rekursivt efter elementet, men sletter det, ved at erstatte det af den knude der er længst til venstre i højre undertræ, dette foregår ved metoderne `joinDict`, der samler

resterne efter sletningen og `splitDict`. Denne knude og resten af træet findes ved metoden `splitDict`:

```
splitDict :: Ord a => Dict(a,b) -> ((a,b) , Dict(a,b))
splitDict (Node i x lt rt) = if isEmpty lt then (x,rt) else (y, Node 1 x ln rt)
where (y,ln) = splitDict lt
```

Her benyttes tupling til at få både elementet og resten af træet tilbage.

#### Funktionen `rangeSearch`

I `rangeSearch` benyttes igen accumulator metoden i kraft af `rangecat`, hvori det afgøres om det er til højre for intervallet, indeni intervallet eller til venstre for intervallet, og kaldes rekursivt i forhold til.

```
rangeSearch :: Ord a => Dict (a,b) -> a -> a -> [b]
rangeSearch d x y = rangecat d x y []

rangecat :: Ord a => Dict (a,b) -> a -> a -> [b] -> [b]
rangecat Leaf _ _ xs = xs
rangecat (Node i x lt rt) y z xs
| fst x == z = rangecat lt y z ((snd x):xs)
| fst x == y = (snd x):(rangecat rt y z xs)
| fst x > y && fst x < z = rangecat lt y z ((snd x):(rangecat rt y z xs))
| fst x < y = rangecat rt y z xs
| fst x > y = rangecat lt y z xs
```

## 4.2 Del 2: Skæringspunkter af vinkelrette linjer

#### Funktionen `Mergesort`

For at få køretiden ned, er `Mergesort` implementeret lige efter teorien, med den tilhørende hjælpemetode `merge`:

```
mergesort :: Ord a => [a] -> [a]
mergesort [] = []
mergesort [x] = [x]
mergesort xs = merge (mergesort ys)(mergesort zs)
where
  (ys,zs) = splitAt (length xs `div` 2) xs

merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) = if x <= y then x : merge xs (y:ys) else y : merge (x:xs) ys
```

#### Line-sweep algoritmen

Først er der oprettet en datatype til at bestemme hvilket punkt vi ser på:

```
data HV = HorSt | Ver | HorEnd deriving (Eq, Ord)
```

Starten på en horisontal linje, vertikal eller slutningen af en horisontal, respektivt. Denne type implementerer `Ord`, der tager den rækkefølge de er defineret i.

**Forarbejdet**

Først tager metoderne addHList og addVList og samler alle relevante punkter i én liste, med x - koordinaten som første koordinat, og en 5-tuple med datatypen HV mm. som anden koordinat. Mergesort sorterer nu denne liste, så punkterne bliver behandlet i den rigtige rækkefølge.

**Algoritmen**

Udførelsen af skridtene i algoritmen benytter 3 metoder: `algorithm`, `algHelp` og `combineSegments`. Da vi har en liste og gerne vil gøre noget med alle elementerne, og sende noget videre er det naturligt at bruge fold med en tilpas funktion. Jeg har valgt at bruge en foldl funktion på grund af måden listen er sorteret på.

```
algorithm d xs rs = foldl algHelp (rs,d) xs
```

I skrivende stund kan jeg se at metoden nok kunne være sparet væk, og indlinet i intersections metoden, men den giver et fornuftig modulisering, af preprocesering og den egentlige algoritme.

Funktionen til foldl er algHelp, der alt efter typen af punktet indsætter, fjerner eller tester for skæringer i det medsendte dictionary.

```
algHelp (rs,d) (n,(HorSt ,s, x1, x2, y)) = (rs,insert d (y,s))
algHelp (rs,d) (n,(HorEnd, s, x1, x2, y)) = (rs,delete d y)
algHelp (rs,d) (n,(Ver, s, y1, y2, x)) =
    ((combineSegments rs (rangeSearch d y1 y2) s) ,d)
```

Funktionen combineSegments tilføjer alle skæringspunkterne som tupler til listen der sendes med og returneres.

```
combineSegments rs [] s = rs
combineSegments rs (x:xs) s = combineSegments ((x,s):rs) xs s
```

## 5 Test

### 5.1 Dictionary

For dictionary testes to ting:

- Om der kan bygges et dictionary fra en sorteret liste
- Om der kan balanceres et ikke balanceret dictionary

Resultatet af de to tests, der kan ses i A ses her:

```
Dictionary> test 1
Node 3 (5,6) (Node 1 (3,4) (Node 0 (1,2) Leaf Leaf) Leaf) (Node 0 (7,8) Leaf Leaf)
Dictionary> test 2
Node 3 (8,4) (Node 1 (6,2) (Node 0 (2,1) Leaf Leaf) Leaf) (Node 0 (10,6) Leaf Leaf)
```

Det kan kontrolleres at disse træer rent faktisk er gyldige balancerede dictionary'es.

### 5.2 Mergesort

Test af mergesort kan ses i A. Resultatet er anført her:

```
Mergesort> msTest 1
[(1,1),(2,1),(3,1),(4,1),(5,1)]
Mergesort> msTest 2
[(1,10),(2,10),(3,10),(4,10),(5,10),(6,10),(7,10),(8,10),(9,10),(10,10),(11,10),(12,10),(13,10)]
Mergesort> msTest 3
[(1,1),(1,2),(2,1),(2,2),(2,3)]
```

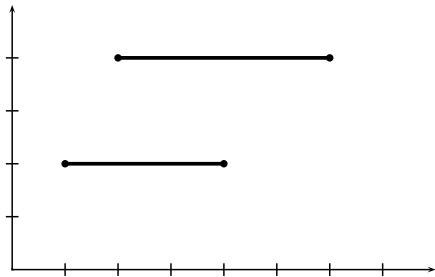
Specielt kan der bemærkes at for tuples sorteres først efter første koordinat, og derefter på anden koordinat. Derfor er ordningen af punkterne relevant når der er samme x-værdi.

### 5.3 Line-sweep algoritmen

Der er for algoritmen opstillet følgende tests:

1. Kun vandrette linjestykker
2. Kun lodrette linjestykker
3. Simpel graf med ét lodret og ét vandret linjestykke, der skærer
4. To tests med flere linjestykker, hvor nogen skærer og nogen ikke skærer
5. Seks linjestykker med maksimalt antal skæringer
6. Skæringer med samme x-koordinat, listerne permuterede på forskellig måde.

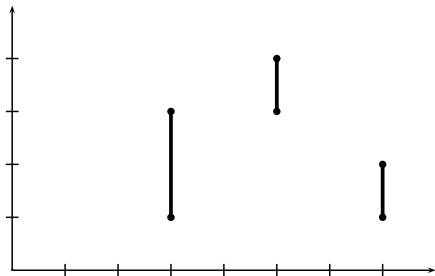
For alle testene gælder det at koden kan ses i A, og sammen med et diagram for hver test er her anført testresultatet.

**Test 1**

Figur 1: Test af kun vandrette streger

```
Main> intersectionTest 1  
[]
```

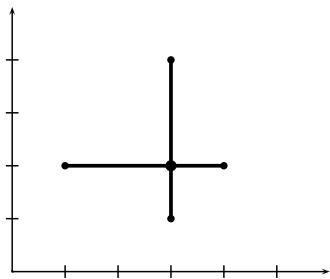
Det ses her at algoritmen korrekt ikke finder skæringspunkter

**Test 2**

Figur 2: Test af kun lodrette streger

```
Main> intersectionTest 2  
[]
```

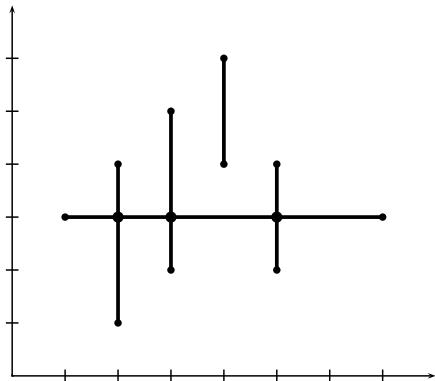
Det ses her at algoritmen igen korrekt ikke finder skæringspunkter

**Test 3**

Figur 3: Test med kun ét skæringspunkt

```
Main> intersectionTest 3  
[("H1","V1")]
```

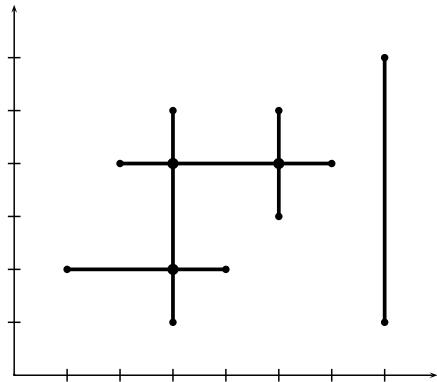
Her ses det at algoritmen korrekt finder skæringspunktet.

**Test 4**

Figur 4: Test med nogle der krydser og nogen der ikke krydser

```
Main> intersectionTest 4  
[("H1","V4"),("H1","V2"),("H1","V1")]
```

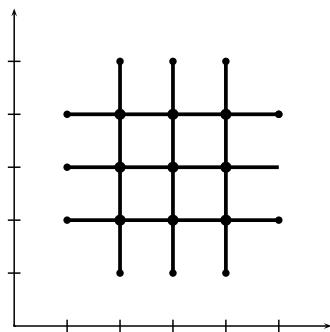
Her ses det tydeligt at algoritmen finder skæringen med de tre af de lodrette linjer der skærer.

**Test 5**

Figur 5: Endnu en test med nogle der krydser og nogen der ikke krydser

```
Main> intersectionTest 5
[("H2","V2"),("H2","V1"),("H1","V1")]
```

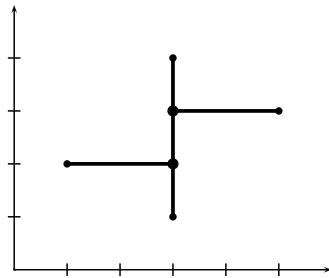
Her ses igen at de rigtige skæringspunkter er fundet

**Test 6**

Figur 6: Test af maksimalt antal skæringspunkter

```
Main> intersectionTest 6
[("H3","V3"),("H2","V3"),("H1","V3"),("H3","V2"),("H2","V2"),
 ("H1","V2"),("H3","V1"),("H2","V1"),("H1","V1")]
```

Her findes korrekt alle 9 skæringer.

**Test 7 og 8**

Figur 7: Test af flere skæringer med samme x-koordinat

```
Main> intersectionTest 7  
[("H2","V1"),("H1","V1")]  
Main> intersectionTest 8  
[("H2","V1"),("H1","V1")]
```

I denne test er rækkefølgen af de vandrette stykke byttet rundt. I begge tilfælde findes de korrekte skæringer.

## 6 Konklusion

Alle de stillede opgaver er implementeret og testet. Der er ikke under testningen forekommet problemer der indikerer at programmet ikke fungerer korrekt.

Enkelte steder kunne programkode komprimeres, men dette er ikke sket, da det i stedet giver et bedre overblik over de enkelte faser i datastrukturen og algoritmen.

Det vurderes at opgaven er løst tilfredsstillende

## 7 Litteratur

- *Introduction to Functional Programming using Haskell* Richard Bird, 2nd edition. Prentice Hall, 1998. ISBN 0-13-484346-0.
- *Haskell: The Craft of Functional Programming* Simon Thompson, Addison-Wesley, 2nd edition, 1999. ISBN 0-201-34275-8.
- *trees.hs* Slides fra kurset, Rolf Fagerberg, <http://www.imada.sdu.dk/Courses/DM22/trees.hs>

## A Source Code

### Dictionary.hs

```

1  module Dictionary (Dict(Leaf,Node) , makeDict , insert , delete ,
2                      search , rangeSearch)where
3
4  —— Node consists of a weight , 'a' which when storing data must be
5  —— a tuple (x,y) where 'x' is the key, and 'y' is the data, left
6  —— subtree and right subtree
7  data Dict a = Leaf | Node Int a (Dict a)(Dict a) deriving Show
8
9  —— Makes a dictionary of the tuples , with a as the search key, and
10 —— b as the data . This implementation returns an empty dictionary ,
11 —— when called with an empty list – If called with a non–empty
12 —— list , a balanced dictionary with the elements is returned
13 —— Note, if called with a list , the list must be sorted
14  makeDict :: Ord a => [(a,b)] -> Dict (a,b)
15  makeDict [] = Leaf
16  makeDict [x] = Node 0 x (Leaf)(Leaf)
17  makeDict xs = mkDictNode r (makeDict lt)(makeDict rt)
18  where
19      m = length xs `div` 2
20      (lt , (r:rt)) = splitAt m xs
21
22  —— Creates an internal node to the tree ,
23  —— which has the right weight .
24  mkDictNode :: a -> Dict a -> Dict a -> Dict a
25  mkDictNode r lt rt = Node (getSize lt + getSize rt) r lt rt
26
27  —— Helping function for the number of leaves
28  —— Returns the number of leaves in the subtrees of the node
29  getSize :: Dict a -> Int
30  getSize Leaf = 0
31  getSize (Node n _ _ _) = n+1
32
33  —— Returns true if the dictionary is empty, and false if there
34  —— is more elements in it .
35  isEmpty :: Ord a => Dict (a,b) -> Bool
36  isEmpty Leaf = True
37  isEmpty (Node _ _ _ _) = False
38
39  —— Searches the dictionary for the data b ,
40  —— corresponding to the search key a
41  search :: Ord a => Dict (a,b) -> a -> b
42  search (Node i x lt rt) y
43  | fst x == y = snd x
44  | fst x > y = search lt y
45  | fst x < y = search rt y
46  | otherwise = error "No match in dictionary"
47
48  —— Inserts the tuple the right place in the search tree
49  —— FIXME doesn't implement balancing yet
50  insert :: Ord a => Dict (a,b) -> (a,b) -> Dict (a,b)
51  insert Leaf y = mkDictNode y (Leaf)(Leaf)
52  insert (Node i x lt rt) y
53  | fst x > fst y = rebuilt (Node (i+1) x (insert lt y) rt)
54  | fst x < fst y = rebuilt (Node (i+1) x lt (insert rt y))
55
56  —— Deletes the entry with key a
57  —— FIXME doesn't implement balancing

```

```

58  delete :: Ord a => Dict (a,b) -> a -> Dict (a,b)
59  delete (Node i x lt rt) y
60    | fst x == y = rebuilt (joinDict lt rt)
61    | fst x > y = rebuilt (Node (i-1) x (delete lt y) rt)
62    | fst x < y = rebuilt (Node (i-1) x lt (delete rt y))
63
64
65  joinDict :: Ord a => Dict (a,b) -> Dict (a,b) -> Dict (a,b)
66  joinDict lt rt = if isEmpty rt then lt else Node 1 x lt rn
67    where (x, rn) = splitDict rt
68
69
70  splitDict :: Ord a => Dict (a,b) -> ((a,b), Dict (a,b))
71  splitDict (Node i x lt rt) = if isEmpty lt then (x, rt) else (y,
72    Node 1 x ln rt)
73    where (y, ln) = splitDict lt
74
75  — Returns a list of the elements in the specified range
76  rangeSearch :: Ord a => Dict (a,b) -> a -> a -> [b]
77  rangeSearch d x y = rangecat d x y []
78
79  — Helping function for rangeSearch
80  rangecat :: Ord a => Dict (a,b) -> a -> a -> [b] -> [b]
81  rangecat Leaf xs = xs
82  rangecat (Node i x lt rt) y z xs
83    | fst x == z = rangecat lt y z ((snd x):xs)
84    | fst x == y = (snd x):(rangecat rt y z xs)
85    | fst x > y && fst x < z = rangecat lt y z ((snd x):(rangecat
86      rt y z xs))
87    | fst x < y = rangecat rt y z xs
88    | fst x > y = rangecat lt y z xs
89
90  — Balances if necessary, using problem and balance
91  rebuilt :: Ord a => Dict (a,b) -> Dict (a,b)
92  rebuilt a = if (problem a) then balance a else a
93
94  — Determines if balancing is needed
95  problem :: Dict a -> Bool
96  problem Leaf = True
97  problem (Node i x lt rt) = if (2*mi < ma) then False else True
98    where
99      mi = (min (getSize lt) (getSize rt))
100     ma = (max (getSize lt) (getSize rt))
101
102  — Balances the Dictionary, using flatten and makeDict, which both
103  — run in linear time of the number of nodes
104  balance :: Ord a => Dict (a,b) -> Dict (a,b)
105  balance x = makeDict (flatten x)
106
107  flatten :: Ord a => Dict (a,b) -> [(a,b)]
108  flatten x = flathelp x []
109
110  flathelp :: Ord a => Dict (a,b) -> [(a,b)] -> [(a,b)]
111  flathelp Leaf xs = xs
112  flathelp (Node i x lt rt) xs = flathelp lt (x:ys)
113    where ys = flathelp rt xs
114
115  — Test of dictionary
116  test x
117    | x == 1 = makeDict [(1,2),(3,4),(5,6),(7,8)]

```

```
118    | x == 2 = balance (Node 2 (10,6) (Node 2 (8,4) (Node 1 (6,2)
           | (Node 1 (2,1) Leaf Leaf) Leaf) Leaf)
```

**Intersection.hs**

```

1  module Intersection (intersections ,
2                      HSegment(HSeg) ,VSegment(VSeg)) where
3
4  import Dictionary
5  import Mergesort
6
7  data HSegment = HSeg SegID MinX MaxX Y
8  data VSegment = VSeg SegID MinY MaxY X
9  type SegID = String
10 type MinX = Float
11 type MaxX = Float
12 type MinY = Float
13 type MaxY = Float
14 type Y = Float
15 type X = Float
16 --Datatype to determine which xcoordinate we are looking at:
17 -- HorSt: start of a horizontal line
18 -- HorEnd: End of horizontal line
19 -- Ver: Vertical line
20 data HV = HorSt | Ver | HorEnd deriving (Eq, Ord)
21
22 -- Exported function from this module.
23 -- Used to find Orthogonal Line Segment Intersections
24 intersections :: [HSegment] -> [VSegment] -> [(SegID,SegID)]
25 intersections [] [] = []
26 intersections xs ys = fst (algorithm (makeDict [] :: Dict(X,SegID)) zs [])
27   where zs = mergesort (addVList ys (addHList xs))
28
29 -- Adds all the horisontal lines to the list ,
30 -- using both their endpoints and their starting points .
31 addHList :: [HSegment] -> [((Float,(HV,[Char]),Float,Float,Float))]
32 addHList [] = []
33 addHList ((HSeg s x1 x2 y):hs) = (x1,(HorSt ,s, x1, x2,
34   y)):(x2,(HorEnd , s, x1, x2, y)): addHList hs
35 -- Adds all the vertical lines to the list , using the x-coordinate
36 addVList :: [VSegment] -> [((Float,(HV,[Char]),Float,Float,Float))]
37   -> [((Float ,(HV,[Char]),Float,Float,Float))]
38 addVList [] xs = xs
39 addVList ((VSeg s y1 y2 x):vs) hs = addVList vs ((x,(Ver, s, y1,
40   y2, x)):hs)
41
42 -- Implements the algorithm described in the assignment
43 -- Keeps a tuple of the results until now: rs ,
44 -- and the dictionary of linepieces already met: d
45 algorithm :: Ord a => Dict (a,[Char]) -> [(b,(HV,[Char],a,a,a))]
46   -> [((String, String)) -> (((String, String)), Dict (a,[Char]))]
47 algorithm d xs rs = foldl algHelp (rs,d) xs
48
49 algHelp :: Ord a => (((String, String)), Dict (a,[Char])) ->
50   (b,(HV,[Char],a,a,a)) -> (((String, String)), Dict (a,[Char]))
51 algHelp (rs,d) (n,(HorSt ,s, x1, x2, y)) = (rs,insert d (y,s))
52 algHelp (rs,d) (n,(HorEnd , s, x1, x2, y)) = (rs,delete d y)
53 algHelp (rs,d) (n,(Ver, s, y1, y2, x)) = ((combineSegments rs
54   (rangeSearch d y1 y2) s),d)
55
56 -- Helping function that adds the found pairs of linesegments

```

```
53  —— to the list of those already found.  
54  combineSegments :: [(String, String)] -> [String] -> String ->  
      [(String, String)]  
55  combineSegments rs [] s = rs  
56  combineSegments rs (x:xs) s = combineSegments ((x,s):rs) xs s
```

**Mergesort.hs**

```

1  module Mergesort (mergesort) where
2
3
4  -- O(n log n) sorting algorithm
5  mergesort :: Ord a => [a] -> [a]
6  mergesort [] = []
7  mergesort [x] = [x]
8  mergesort xs = merge (mergesort ys) (mergesort zs)
9    where
10      (ys, zs) = splitAt (length xs `div` 2) xs
11
12  -- Helping function for mergesort
13  merge :: Ord a => [a] -> [a] -> [a]
14  merge [] ys = ys
15  merge xs [] = xs
16  merge (x:xs) (y:ys) = if x <= y then x : merge xs (y:ys) else y :
17    merge (x:xs) ys
18
19  -- Test of mergesort
20  msTest :: Int -> [(Int, Int)]
21  msTest n
22  | n == 1     = mergesort [(5,1),(4,1),(3,1),(2,1),(1,1)]
23  | n == 2     = mergesort [(9,10),(8,10),(7,10),(10,10),(6,10),(5,10),(4,10),(11,10),(13,10),
24  | n == 3     = mergesort [(2,3),(1,2),(2,2),(2,1),(1,1)]
```

**Test.hs**

```
1 import Intersection
2
3 -- Test of intersections
4 intersectionTest n
5   | n == 1 = intersections [(HSeg "H1" 1 4 2),(HSeg "H2" 2 6 4)]
6   | n == 2 = intersections [][(VSeg "V1" 1 5 3),(VSeg "V2" 3 5
7   | n == 3 = intersections [(HSeg "H1" 1 4 2)] [(VSeg "V1" 1 5
8   | n == 4 = intersections [(HSeg "H1" 1 7 3)] [(VSeg "V1" 1 4
9   | n == 5 = intersections [(HSeg "H1" 1 4 2),(HSeg "H2" 2 6 4)
10  | n == 6 = intersections [(HSeg "H1" 1 5 2),(HSeg "H2" 1 5
11  | n == 7 = intersections [(HSeg "H1" 1 3 2),(HSeg "H2" 3 5 4)
12  | n == 8 = intersections [(HSeg "H2" 3 5 4),(HSeg "H1" 1 3 2)
   | (VSeg "V1" 1 4 3)]
```