COMP304 Programming Languages

Assignment 2: More Haskell

David Thomsen, 300052209

1. Binary trees

This is the data definition given to me assignment sheet, used throughout the assignment:

```haskell
data BinTree a = Empty | Node a (BinTree a) (BinTree a) deriving (Show, Eq)
```

Hasbt is a relatively simple function without much need for complexity, I did not consider any other way of doing it than this.

```haskell
hasbt :: Eq a => a -> BinTree a -> Bool
hasbt _ Empty = False
hasbt x (Node a t1 t2) | x == a = True
                       | otherwise = or [(hasbt x t1), (hasbt x t2)]
```

Equalbt is simple but requires a few lines of code to check all of the different possibilities, making it quite large. I originally had two lines of code, the first checking if the left subtree was empty and the right wasn't, then if the right subtree was empty and the left wasn't, but realised I could simplify that by putting a catch at the bottom.

```haskell
equalbt :: Eq a => BinTree a -> BinTree a -> Bool
equalbt Empty Empty = True
equalbt (Node a t1 t2) (Node b t3 t4)
    | a == b = and [equalbt t1 t3, equalbt t2 t4]
    | otherwise = False
equalbt _ _ = False
```

Reflectbt was the simplest of all, simply building another BinTree out of the old one but putting the branches together in reverse. Again, I did not consider any other way of completing this challenge.

```haskell
reflectbt :: BinTree a -> BinTree a
reflectbt Empty = Empty
reflectbt (Node a t1 t2) = Node a (reflectbt t2) (reflectbt t1)
```

Fringebt was written wrong at first, I made a list of all nodes instead of just the leaf ones, until I realised the error of my ways. The three lines cover the three cases and I can't find a way to simplify them further.

```haskell
fringebt :: BinTree a -> [a]
fringebt Empty = []
fringebt (Node n Empty Empty) = [n]
fringebt (Node n t1 t2) = fringebt t1++fringebt t2
```

Fullbt I have written in two different ways, firstly the first way I wrote it, then fullbt2 is another way it occurred to me to write it. I have no strong preference

either way, the second is longer but the first requires a bit more thinking to understand what is going on.

```
fullbt :: Eq a => BinTree a -> Bool
fullbt Empty = True
fullbt (Node a t1 t2) | t1 == Empty = t2 == Empty
                      | t2 == Empty = False
                      | otherwise = and [fullbt t1, fullbt t2]

fullbt2 :: Eq a => BinTree a -> Bool
fullbt2 Empty = True
fullbt2 (Node a Empty Empty) = True
fullbt2 (Node a Empty _) = False
fullbt2 (Node a _ Empty) = False
fullbt2 (Node a t1 t2) = and [fullbt2 t1, fullbt2 t2]
```

2. Binary tree folds

I wrote my btfold function similarly to the way the fold function had been written in the slides, but for a binary tree rather than a list. Looking at it now, I can kind of understand what's going on, but it's as they say, 'any sufficiently advanced Haskell is indistinguishable from magic'.

```
btfold :: (a -> b -> b -> b) -> b -> BinTree a -> b
btfold f u Empty = u
btfold f u (Node a t1 t2) = f a (btfold f u t1) (btfold f u t2)
```

Hasbtf was one of the simplest binary tree fold function to write, I don't think I need to think too hard about othe ways I could have done it.

```
hasbtf :: Eq a => a -> BinTree a -> Bool
hasbtf x t = btfold (\u v w -> x == u || v || w) False t
```

Equalbtf is incompatible with my standard btfold function, as it requires the input of two trees rather than one. Possibly I could get it to use the btfold function if I folded the trees together in some clever way, but I don't think I'm clever enough to figure that out.

I started implementing a btfold2 function that accepts two binary trees, and while I think it can be done with some effort, I decided it was redundant because it adds much more complexity than it saves, and would probably be too specialised to have an use outside of comparing if two binary trees are equal.

Reflectbtf was the other simplest binary tree fold function to write, simply swapping the two branches.

```
reflectbtf :: BinTree a -> BinTree a
reflectbtf t = btfold (\u v w -> Node u w v) Empty t
```

Fringebtf and fullbtf are tricky as they are conditional on what is returned from the two branches, for example if the two branches in a fringebtf fold return empty arrays, then we know that the node is a leaf and to return that value in

an array, otherwise just return a concatenation of the two arrays returned by the branches.

Instead of designing a whole new btfold function, for fringebtf I extracted and expanded the function being passed to the btfold.

```
fringebtf :: Eq a => BinTree a -> [a]
fringebtf t =  btfold (fringebtf') [] t

fringebtf' :: Eq a => a -> [a] -> [a] -> [a]
fringebtf' a b c | b == [] && c == [] = [a]
                 | otherwise = b++c
```

Same for fullbtf. This needed some extra thinking as to how the logic works, but it makes sense in my head. The base case is false. If both branches return false, then return true, because they are both empty. If both branches are true, return true, because they are both full. For any other combination, return false. I am quite proud of the simplicity of this solution.

```
fullbtf :: Eq a => BinTree a -> Bool
fullbtf t = btfold (fullbtf') False t

fullbtf' :: Eq a => a -> Bool -> Bool -> Bool
fullbtf' _ a b = a == b
```

3. Binary Search Tree

First, a very simple function for returning an empty BinTree. Literally just returning Empty.

```
empty :: Ord a => BinTree a
empty = Empty
```

For my insert function I attempt to make use of my fold function in the same way I had with frngebtf and fullbtf above, by extracting a submethod to handle the greater than/lesser than logic. However it ended with a tree in which every leave had my data entered into it, so it was a bit of a flop.

```
insert :: Ord a => a -> BinTree a -> BinTree a
insert x Empty = (Node x Empty Empty)
insert x (Node a t1 t2) | x < a = (Node a (insert x t1) t2)
                        | x > a = (Node a t1 (insert x t2))
                        | otherwise = Empty
```

The 'has' function could be done using the fold function as above, but it would be less efficient, searching through every node. This way is more efficient because it utilises the ordered nature of the binary search tree.

```
has :: Ord a => a -> BinTree a -> Bool
has x Empty = False
has x (Node a t1 t2) | x == a = True
```

```
    | x < a = has x t1
    | otherwise = has x t2
```

Fortunately the delete function doesn't require the tree to remain balanced, because that would be madness. Another way to implement this delete function might have been to flatten the tree, delete the value and then build the tree up from the array. This would probably result in an even more unbalanced tree though, as the flattened array would be in order, and to build a tree with values in order would result in a long list of nodes rather than a tree.

```
delete :: Ord a => a -> BinTree a -> BinTree a
delete _ Empty = Empty
delete x (Node a t1 t2)
    | x < a = (Node a (delete x t1) t2)
    | x > a = (Node a t1 (delete x t2))
    | t1 == Empty = t2
    | t2 == Empty = t1
    | otherwise = Node (getNode t1) (delete (getNode t1) t1) t2
```

This is a simple helper function to extract the value of any node, used by the delete method above.

```
getNode :: BinTree a -> a
getNode (Node a t1 t2) = a
```

A basic flatten function. Because this can actually use the fold function, I've also written it to use the fold function, just for maybe extra marks I guess.

```
flatten :: BinTree a -> [a]
flatten Empty = []
flatten (Node n t1 t2) = flatten t1++[n]++flatten t2

flatten2 :: BinTree a -> [a]
flatten2 t = btfold (\u v w -> v++[u]++w) [] t
```

Rather than write a lot of logic for 'equals', I'll just piggyback on the 'flatten' method.

```
equals :: (Eq a ) => BinTree a -> BinTree a -> Bool
equals t1 t2 = flatten t1 == flatten t2
```

4. Graph algorithms

For the graph algorithms I misunderstood the question where it said 'these functions should both be implemented using an additional function which finds all the paths from one vertex to another'. Instead I wrote a helper function that finds all Edges connected to a particular node. This was also helpful, but it meant my 'reachable' and 'minCostPath' functions contained a logic that could have been in the 'allPaths' function.

I added an Edge type to the Graph type because I thought this might be useful.

```
type Edge a = (a,Int,a)

type Graph a = [Edge a]
```

If I had implemented 'allPaths' correctly, this would have been trivial, just checking if 'allPaths' is an empty array, or actually contains paths.

Reachable initialises an array of visited nodes with the first node and passes it to the reachable' function.

```
reachable :: Eq a => a -> a -> Graph a -> Bool
reachable x y [] = False
reachable x y g = reachable' x y g [x]
```

If we haven't reached the goal yet, find all edges from a starting point and pass them to visitPaths

```
reachable' :: Eq a => a -> a -> Graph a -> [a] -> Bool
reachable' x y g v | x == y = True
                   | otherwise = visitPaths x y g (allPaths x g) v
```

For each edge in the list, if we've been to the destination of that edge already then go to the next edge, otherwise see if we can reach the goal from that edge or if we can't, see if we can from any other edges.

```
visitPaths :: Eq a => a -> a -> Graph a -> Graph a -> [a] -> Bool
visitPaths _ _ _ [] _ = False
visitPaths x y g ((a,k,b):gs) v
    | hasBeenTo b v = visitPaths x y g gs v
    | otherwise = reachable' b y g (v++[x]) || visitPaths x y g gs v
```

For the minCostPath I used this data type to make a binary search tree of accumuated distance, path taken, furthest node in the path and nodes visited on the path

```
data Path a = Void | Path Int (Graph a) a [a] deriving (Show, Eq)
```

First, check if the goal is actually reachable! If it is, initialise a tree and pass it to minCostPath'. Initialising the tree was important because I needed a point to start from.

```
minCostPath :: Eq a => a -> a -> Graph a -> Graph a
minCostPath x y g
    | reachable x y g = minCostPath' x y g (initialisePathTree x g)
    | otherwise = error "Unreachable"
```

If the shortest path reaches the goal, then extract that path. Otherwise expand the search and carry on.

```
minCostPath' :: Eq a => a -> a -> Graph a -> BinTree (Path a) -> Graph a
minCostPath' x y g v
    | reachesGoal y (getShortest v) = extractPath (getShortest v)
    | otherwise = minCostPath' x y g (expandSearch x y g v)
```

Initialises a binary search tree from the starting point. I would have really liked to invert the edge when it matches the case 'x == c', so that [(c,b,a)] gets added to the path for example and the final path would be in a nice sequential order, but that's not what the tutors wanted for some reason.

```
initialisePathTree :: Eq a => a -> Graph a -> BinTree (Path a)
initialisePathTree x g = initialisePathTree' x g Empty

initialisePathTree' :: Eq a
    => a -> Graph a -> BinTree (Path a) -> BinTree (Path a)
initialisePathTree' x [] t = t
initialisePathTree' x ((a,b,c):gs) t
    | x == a = initialisePathTree' x gs
        (addToPathTree (Path b [(a,b,c)] c [a]) t)
    | x == c = initialisePathTree' x gs
        (addToPathTree (Path b [(a,b,c)] a [c]) t)
    | otherwise = initialisePathTree' x gs t
```

Gets the shortest path, finds the next shortest, adds it to the binary search tree, and then deletes the shortest path. It actually just occured to me that this might be flawed, because it might not be the next shortest path that actually leads to the minimum cost path to the goal, and when I delete the shortest path I might be deleting a route to the actual shortest path. This would have been easier if I'd implemented allPaths properly.

```
expandSearch :: Eq a
    => a -> a -> Graph a -> BinTree (Path a) -> BinTree (Path a)
expandSearch x y g t = deleteFromPathTree
    (getShortest t) (addToPathTree (findNextShortest g (getShortest t)) t)
```

I needed to write new add and delete functions for my binary search tree, because it needed to compare the distance so far in a path, rather than the entire node.

```
addToPathTree :: Eq a => Path a -> BinTree (Path a) -> BinTree (Path a)
addToPathTree Void t = t
addToPathTree x Empty = (Node x Empty Empty)
addToPathTree (Path i x y z) (Node (Path j a b c) t1 t2)
    | i > j = (Node (Path j a b c) t1 (addToPathTree (Path i x y z) t2))
    | otherwise = (Node (Path j a b c) (addToPathTree (Path i x y z) t1) t2)

deleteFromPathTree :: Eq a => Path a -> BinTree (Path a) -> BinTree (Path a)
deleteFromPathTree (Path i x y z) (Node (Path j a b c)  t1 t2)
    | i > j = (Node (Path j a b c) t1 (deleteFromPathTree (Path i x y z) t2))
    | (Path i x y z) == (Path j a b c) && t1 /= Empty
        = Node (getNode t1) (deleteFromPathTree (getNode t1) t1) t2
    | (Path i x y z) == (Path j a b c) = t2
    | otherwise
        = (Node (Path j a b c) (deleteFromPathTree (Path i x y z) t1) t2)
```

Takes a full graph and a path and returns a new path with the next minimum

edge, or Void if there isn't one. There would probably have been a cleaner way to have one function rather than both findNextShortest and findNextShortest', but it still would have been messy.

```
findNextShortest :: Eq a => Graph a -> Path a -> Path a
findNextShortest [] a = Void
findNextShortest ((n,j,m):gs) (Path l g p z)
    | p == n && hasBeenTo m z == False
        = findNextShortest' gs (n,j,m) (Path l g p z)
    | p == m && hasBeenTo n z == False
        = findNextShortest' gs (n,j,m) (Path l g p z)
    | otherwise = findNextShortest gs (Path l g p z)
```

Takes the full graph, the minimum edge found so far, a path and returns a path with the actual next min edge

```
findNextShortest' :: Eq a => Graph a -> Edge a -> Path a -> Path a
findNextShortest' [] (x, i, y) (Path l g p z)
    | p == x = Path (l+i) (g++[(x,i,y)]) y (z++[p])
    | p == y = Path (l+i) (g++[(x,i,y)]) x (z++[p])
findNextShortest' ((n,j,m):gs) (x,i,y) (Path l g p z)
    | p == n && hasBeenTo m z == False && j < i
        = findNextShortest' gs (n,j,m) (Path l g p z)
    | p == m && hasBeenTo n z == False && j < i
        = findNextShortest' gs (n,j,m) (Path l g p z)
    | otherwise = findNextShortest' gs (x,i,y) (Path l g p z)
```

Just so we can print out the path as a graph rather than as a Path data type.

```
extractPath :: Eq a => Path a -> Graph a
extractPath (Path _ g _ _) = g
```

Returns the shortest path in the binary search tree. There was something awkward where because 't1' could also match empty, I had to make it explicit that it wasn't an empty node.

```
getShortest :: Eq a => BinTree (Path a) -> Path a
getShortest Empty = Void
getShortest (Node n Empty Empty) = n
getShortest (Node n (Node m s1 s2) t2) = getShortest (Node m s1 s2)
getShortest (Node n Empty t2) = getShortest t2
```

Just checking if a particular Path reaches the goal.

```
reachesGoal :: Eq a => a -> Path a -> Bool
reachesGoal _ Void = False
reachesGoal a (Path _ _ b _) | a == b = True
                             | otherwise = False
```

Methods supposed to be used by both reachable and minCostPath

First one simply checks a is contained within an array of [a].

```
hasBeenTo :: Eq a => a -> [a] -> Bool
hasBeenTo _ [] = False
hasBeenTo a (e:es) | a == e = True
                   | otherwise = hasBeenTo a es
```

This should be named 'allEdges' because that it is what it does, finding all edges of a graph that connect to a particular node. Was not actually used in minCostPath in the end.

```
allPaths :: Eq a => a -> Graph a -> Graph a
allPaths _ [] = []
allPaths x ((a,v,b):gs) | x == a = [(a,v,b)]++allPaths x gs
                        | x == b = [(b,v,a)]++allPaths x gs
                        | otherwise = allPaths x gs
```

Lazy evaluation ensure that execution of 'reachable' is less expensive than that of 'minCostPath' because it will stop evaluation once it receive an affirmative answer.

Here I set up a type 'Clique', which is an array of type 'Graph', which is an array of type 'Edge'. I think this was more useful than if I had tried to work with type [[(a,Int,a)]].

```
type Clique a = [Graph a]
```

My 'cliques' method works by comparing each edge to each graph in an existing set of cliques, and if it connect to anything in the graph then add it to that graph, otherwise ignore.

I thought about other tools I can use to accomplish this, like fold techniques, but as none of the type signatures match, it would result in a folding method created specifically for the use of one method, which seems pointless.

There are other implementations that could have been more efficient, but I feel that the added complexity of those methods would be too much of a compromise for only a small amount of gain for this assignment.

I could also have used some of the methods above in various ways, but I think that that would have resulted in unnecessary coupling, and also some awkward alterations to make the methods fit in usefully.

```
cliques :: Eq a => Graph a -> Clique a
cliques a = cliques' a []

cliques' :: Eq a => Graph a -> Clique a -> Clique a
cliques' [] l = l
cliques' (e:es) l = cliques' es (sortCliques e l [e] [])

sortCliques :: Eq a => Edge a -> Clique a ->
    Graph a -> Clique a -> Clique a
```

```
sortCliques _ [] a b = [a]++b
sortCliques e (l:ls) a b | containsEdge e l = sortCliques e ls (l++a) b
                         | otherwise = sortCliques e ls a ([l]++b)

containsEdge :: Eq a => Edge a -> Graph a -> Bool
containsEdge _ [] = False
containsEdge (a,x,b) ((c,y,d):gs)
    | a == c || a == d || b == c || b == d = True
    | otherwise = containsEdge (a,x,b) gs
```

TESTING

Testing data

```
emptyTree = (Empty :: BinTree Int)
partialTree1 = (Node 3 (Node 2 Empty Empty) (Node 4 (Node 1 Empty Empty) Empty)
    :: BinTree Int)
partialTree2 = (Node 3 (Node 2 Empty Empty) (Node 4 (Node 2 Empty Empty) Empty)
    :: BinTree Int)
partialTree3 = (Node 3 (Node 4 Empty (Node 1 Empty Empty)) (Node 2 Empty Empty)
    :: BinTree Int)
fullTree = (Node 1 (Node 2 Empty Empty) (Node 3 (Node 4 Empty Empty)
    (Node 5 Empty Empty)) :: BinTree Int)
binarySearchTree1 = (Node 3 (Node 1 Empty (Node 2 Empty Empty))
    (Node 4 Empty (Node 5 Empty Empty)) :: BinTree Int)
binarySearchTree2 = (Node 1 Empty (Node 2 Empty (Node 3 Empty
    (Node 4 Empty (Node 5 Empty Empty)))) :: BinTree Int)
binarySearchTree3 = (Node 'e' (Node 'd' (Node 'c' (Node 'b' Empty Empty) Empty)
    Empty) Empty :: BinTree Char)
minCostPathPart1 = [('e',2,'d'),('c',4,'a'),('b',2,'c'),('d',4,'b')]
minCostPathPart2 = [('c',1,'d'),('e',7,'a'),('b',1,'a')]
fullMinCostPath = minCostPathPart1++minCostPathPart2
edgeList = [(1,2,3),(4,5,6),(6,7,8),(8,9,10),(11,12,13),(13,14,15)]
cliqueList = [[(11,12,13),(13,14,15)],[(4,5,6),(6,7,8),(8,9,10)],[(1,2,3)]]
```

Tests for part 1

```
testHasbt1 = hasbtf 1 fullTree == True
testHasbt2 = hasbtf 6 fullTree == False
testHasbt3 = hasbt 1 emptyTree == False
testHasbt4 = hasbt 1 partialTree1 == True
testHasbt5 = hasbt 8 partialTree1 == False
testHasbt = and [testHasbt1, testHasbt2, testHasbt3, testHasbt4, testHasbt5]

testEqualbt1 = equalbt emptyTree emptyTree == True
testEqualbt2 = equalbt fullTree emptyTree == False
testEqualbt3 = equalbt emptyTree fullTree == False
testEqualbt4 = equalbt partialTree1 partialTree1 == True
testEqualbt5 = equalbt partialTree1 partialTree2 == False
```

```
testEqualbt = and [testEqualbt1, testEqualbt2,
    testEqualbt3, testEqualbt4, testEqualbt5]

testReflectbt1 = reflectbt partialTree1 == partialTree3
testReflectbt2 = reflectbt partialTree2 /= partialTree3
testReflectbt = and [testReflectbt1, testReflectbt2]

testFringebt1 = fringebt emptyTree == []
testFringebt2 = fringebt partialTree1 == [2,1]
testFringebt3 = fringebt partialTree3 == [1,2]
testFringebt = and [testFringebt1, testFringebt2, testFringebt3]

testFullbt1 = fullbt partialTree1 == False
testFullbt2 = fullbt fullTree == True
testFullbt3 = fullbt (reflectbt fullTree) == True
testFullbt = and [testFullbt1, testFullbt2, testFullbt3]
```

Tests for part 2

```
testBtfold1 = btfold (\u v w -> 1 + v + w) 0 fullTree == 5
testBtfold2 = btfold (\u v w -> u + v + w) 0 fullTree == 15
testBtfold = and [testBtfold1, testBtfold2]

testHasbtf1 = hasbtf 1 fullTree == True
testHasbtf2 = hasbtf 6 fullTree == False
testHasbtf3 = hasbt 1 emptyTree == False
testHasbtf4 = hasbt 1 partialTree1 == True
testHasbtf5 = hasbt 8 partialTree1 == False
testHasbtf = and [testHasbtf1, testHasbtf2,
    testHasbtf3, testHasbtf4, testHasbtf5]

testReflectbtf1 = reflectbtf partialTree1 == partialTree3
testReflectbtf2 = reflectbtf partialTree1 /= partialTree2
testReflectbtf = and [testReflectbtf1, testReflectbtf2]

testFringebtf1 = fringebt emptyTree == []
testFringebtf2 = fringebtf partialTree1 == [2,1]
testFringebtf3 = fringebtf partialTree3 == [1,2]
testFringebtf = and [testFringebtf1, testFringebtf2, testFringebtf3]

testFullbtf1 = fullbt partialTree1 == False
testFullbtf2 = fullbt fullTree == True
testFullbtf3 = fullbt (reflectbt fullTree) == True
testFullbtf = and [testFullbtf1, testFullbtf2, testFullbtf3]
```

Tests for part 3

```
testEmpty = delete 2 (Node 2 Empty Empty) == empty

testInsert1 = insert 3 Empty == (Node 3 Empty Empty)
testInsert2 = insert 1 (Node 3 Empty Empty)
    == Node 3 (Node 1 Empty Empty) Empty
```

```
testInsert3 = insert 2 (Node 3 (Node 1 Empty Empty) Empty)
    == Node 3 (Node 1 Empty (Node 2 Empty Empty)) Empty
testInsert4 = insert 1 (Node 3 (Node 1 Empty Empty) Empty)
    == Node 3 (Node 1 Empty Empty) Empty
testInsert = and [testInsert1, testInsert2, testInsert3, testDelete4]

testDelete1 = delete 2 (Node 3 (Node 1 Empty (Node 2 Empty Empty)) Empty)
    == Node 3 (Node 1 Empty Empty) Empty
testDelete2 = delete 1 (Node 3 (Node 1 Empty Empty) Empty)
    == Node 3 Empty Empty
testDelete3 = delete 3 (Node 3 Empty Empty) == Empty
testDelete4 = delete 4 (Node 3 (Node 1 Empty Empty) Empty)
    == Node 3 (Node 1 Empty Empty) Empty
testDelete5 = delete 5 Empty == Empty
testDelete = and [testDelete1, testDelete2,
    testDelete3, testDelete4, testDelete5]

testFlatten1 = flatten binarySearchTree1 == [1,2,3,4,5]
testFlatten2 = flatten binarySearchTree2 == [1,2,3,4,5]
testFlatten3 = flatten binarySearchTree3 == ['b','c','d','e']
testFlatten4 = flatten (empty :: BinTree Int) == []
testFlatten = and [testFlatten1, testFlatten2, testFlatten3, testFlatten4]

testEquals1 = equals binarySearchTree1 binarySearchTree2
testEquals2 = equals (empty :: BinTree Int) (empty :: BinTree Int)
testEquals = and [testEquals1, testEquals2]
```

Tests for part 4

I did not test these extensively as the tutor said it was more important to demonstrate that I knew how to test effectively than to actually write tests for everything

```
testGraph = ([('a',4,'b'),('b',1,'b'),('b',5,'c'),('d',3,'c'),('e',1,'f')]
    :: Graph Char)

testAllPaths = allPaths 'b' testGraph == [('b',4,'a'),('b',1,'b'),('b',5,'c')]

testReachable1 = reachable 'a' 'd' testGraph
testReachable2 = reachable 'a' 'f' testGraph == False
testReachable = and [testReachable1, testReachable2]

testMinCostPath = minCostPath 'a' 'e' fullMinCostPath
    == [('b',1,'a'),('b',2,'c'),('c',1,'d'),('e',2,'d')]

testCliques = cliques edgeList
    == cliqueList
```

Tests for each part

```
testPart1 = and [testHasbt, testEqualbt,
    testReflectbt, testFringebt, testFullbt]
```

```
testPart2 = and [testBtfold, testReflectbtf, testFringebtf]
testPart3 = and [testEmpty, testInsert, testDelete, testFlatten, testEquals]
testPart4 = and [testReachable, testMinCostPath, testAllPaths, testCliques]
```

All tests

```
test = and [testPart1, testPart2, testPart3, testPart4]
```