# COMP 304 Assignment 2

Daniel Braithwaite

Import the list library

```
import qualified Data.List as L
```

Define the BinTree data type

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

# Binary Trees

## hasbt

The hasbt function takes a value and a binary tree of the same time. Returns true iff the given value occors in the tree

```
hasbt :: (Ord a) => a -> BinTree a -> Bool
```

Base Cases

1. When the tree is empty clearly it dosnt contain any value so we can return False

```
hasbt x Empty = False
```

Recursive Cases

1. Here we have to things to check, if this current node contains the value we are looking for then we can return True, otherwise we check both the left and right subtrees

```
hasbt x (Node e l r)
  | x == e = True
  | otherwise = (hasbt x l) || (hasbt x r)
```

## equalbt

equalbt takes two BinTrees and recursively compares the subtrees.

```
equalbt :: (Eq a) => BinTree a -> BinTree a -> Bool
```

Base Cases

1. Two empty trees are equal.
2. Otherwise if we have an empty tree and something else they are clearly not equal.

```
equalbt Empty Empty = True
equalbt Empty _ = False
equalbt _ Empty = False
```

Recursive Cases

1. If the current value in each tree is not equal then the trees arnt equal otherwise we compare the two sub trees

```
equalbt (Node x l1 r1) (Node y l2 r2)
  | x /= y = False
  | otherwise = (equalbt l1 l2) && (equalbt r1 r2)
```

## reflecttbt

reflecttbt takes a BinTree and recursively swaps the left and right subtrees

```
reflecttbt :: BinTree a -> BinTree a
```

Base Cases

1. The empty tree reflected is just Empty

```
reflecttbt Empty = Empty
```

Recursive Cases

1. Otherwise we swap left and the right subtrees and recursivelty call reflectbt

```
reflecttbt (Node a l r) = Node a (reflecttbt r) (reflecttbt l)
```

## fringebt

fringebt takes a BinTree and returns a list containing all the elements that are contained in a leaf node.

```
fringebt :: BinTree a -> [a]
```

Base Cases

1. The fringe of an empty tree is just an empty array

```
fringebt Empty = []
```

Recursive Cases

1. If the current node is a leaf then we return a singleton list containing the current value

```
fringebt (Node x l r)
  | isLeaf l r = [x]
  | otherwise = (fringebt l) ++ (fringebt r)
```

We define an auxriliry function to tell if a node is a leaf given its left and right subtrees

```
  where isLeaf Empty Empty = True
        isLeaf _ _ = False
```

## fullbt

fullbt takes a BinTree and returns true iff the tree only contains leaf nodes or nodes with both a left and right subtree.

```
fullbt :: BinTree a -> Bool
```

Base Cases

1. An empty tree is full so we can return True
2. A leaf node is a full tree so we can return True
3. If one of the subtrees is Empty but the other isnt then the tree isnt full so we reutrn False

```
fullbt Empty = True
fullbt (Node x Empty Empty) = True
fullbt (Node x Empty _) = False
fullbt (Node x _ Empty) = False
```

Recursive Cases

1. ensure that both left and right sub trees are full trees

```
fullbt (Node x l r) = (fullbt l) && (fullbt r)
```

# Binary Tree Fold

## btfold

The btfold function takes three arguments a function, a unit (base case) and a BinTree. The given function should also take the folowing three arguments, node element, result of function applyed to left subtree, result of function applyed to right subtree. When we reach an empty node then the unit is returned rather than applying the function to it.

```
btfold :: (a -> b -> b -> b) -> b -> BinTree a -> b
btfold f u Empty = u
btfold f u (Node x l r) = f x (btfold f u l) (btfold f u r)
```

## hasbtf

The unit is False as if we reach an empty node it dosnt contain any elements. The lambda compares the current node value to the value we are searching for, and or's it with the result of the left and right sub trees.

```
hasbtf :: (Eq a) => a -> BinTree a -> Bool
hasbtf a t = btfold (\x y z -> (x == a) || y || z) False t
```

## equalbtf

Cant implement equalbtf as a fold as it dosnt make sense to do so. A fold is an operation on a single tree. A pollsible implementation could be to zip to trees together so that each node is a pair of elements and then use a fold to compare the elements of the tuple. But creating the zipped tree requires traversing both trees anyway so may as well compare them then.

### reflecttbtf

The reflect tree operation works nicely as a fold, At each step we swap the result of the left and right subtrees and return a new node.

```
reflecttbtf :: BinTree a -> BinTree a
reflecttbtf t = btfold (\a b c -> (Node a c b)) Empty t
```

### fringebtf

If we reach an empty node then we return an empty node. So if a node has both its left and right subtrees equal to empty lists we return the current node element as a singleton. Otherwise we concaternate the left and right subtree results. Using a fold isnt the best way to solve this problem so the solution isnt the best.

```
fringebtf :: (Eq a) => BinTree a -> [a]
fringebtf t = btfold (\a b c -> if b == [] && c == [] then [a] else b ++ c) [] t
```

### fullbtf

While you can implement fullbtf as a tree fold, there are more natural ways to do it. This method involves, counting the depth as we move up a tree and if we get to a node where the left, right depth are different the tree isnt full then we propergate -1 up the tree. If we have a -1 after the fold we know the tree isnt full.

```
fullbtf :: BinTree a -> Bool
fullbtf t = (btfold (\a b c -> compareDep b c) 0 t) /= -1
            where compareDep x y
                    | x == -1 = -1
                    | y == -1 = -1
                    | x /= y = -1
                    | x == y = x + 1
```

## Binary Search Trees

### empty

```
empty :: BinTree a
empty = Empty
```

## insert

```
insert :: (Ord a) => a -> BinTree a -> BinTree a
```

Base Cases

1. If we reach an Empty node we can insert our element here so return a node containing this element and two Empty subtrees

```
insert x Empty = Node x Empty Empty
```

Recursive Cases

1. If the item we want to insert is less than the current node the recursively insert into the left subtree, otherwise recusively insert into the right subtree.

```
insert x (Node e l r)
  | x < e = (Node e (insert x l) r)
  | otherwise = (Node e l (insert x r))
```

## has

For this problem we could of used the previously defined equalbt function but sinse we are operating on a BST we can use to properties of BST to speed up the search.

```
has :: (Ord a) => a -> BinTree a -> Bool
```

Base Cases

1. An empty tree contains no elements so we return False

```
has _ Empty = False
```

Recursive Cases

1. If the item we are searching for is less than the current element then search the left sub tree, greater than the current element search the right subtree. Otherwise return True.

```
has x (Node e l r)
  | x < e = has x l
  | x > e = has x r
  | otherwise = True
```

## delete

```
delete :: (Ord a) => a -> BinTree a -> BinTree a
```

Base Cases

1. Removing an element from an empty tree just gives an empty tree

```
delete x Empty = Empty
```

Recursive Cases

1. If the item we want to delete is less than the current element recurse on the left sub tree, if its greater then the current element recurse on the right sub tree. Otherwise call the remove function on the current node.

```
delete x n@(Node e l r)
  | x < e = (Node e (delete x l) r)
  | x > e = (Node e l (delete x r))
  | otherwise = remove x n
```

We have three cases for removing a node. If the node is a leaf node then we can simply remove it from the tree. If the node only has one child then replace the current node with the child. Finally if the node has two children we find the smallest child in the right sub tree, replace the current nodes element with the smallest value and recursively delete the smallest element.

```
  where remove x (Node _ Empty Empty) = Empty
        remove x (Node _ l Empty) = l
        remove x (Node _ Empty r) = r
        remove x (Node e l r) = Node (smallest r) l (delete (smallest r) r)
```

The smallest function takes a BinTree and returns the left most element

```
        smallest (Node e Empty Empty) = e
        smallest (Node e l r) = smallest l
```

## flatten

```
flatten :: BinTree a -> [a]
```

Base Cases

1. Flatening an empty should return an empty list

```
flatten Empty = []
```

Recursive Cases

1. We want the flattened tree to come out in sorted order so flatten the left, append it to the right

```
flatten (Node x l r) = (flatten l) ++ (x:(flatten r))
```

### equals

For determining wether two binary trees are equal we can use our equalbt function as the fact our trees are BST's dosnt help us, we still have to compare every subtree.

```
equals :: (Eq a) => BinTree a -> BinTree a -> Bool
equals = equalbt
```

## Graph Algorythms

```
type Graph a = [(a, Int, a)]
```

### Part 1

We find all the paths between two nodes by using a recursive depth first search and building our set of paths as we go

```
paths :: (Eq a) => a -> a -> Graph a -> [a] -> [Graph a]
```

If our start vertex is equal to the finish vertex then return a list with an empty path (as the path from a vertex to its self contains no edges)

```
paths s f g v
  | s == f = [[]]
  | otherwise = subpaths
```

we use the concatMap function to generate a list of subpaths from the current vertex by extending the current path though each of the verticies neighbours. We use a concatMap because otherwise we get a list of lists of graphs.

```
    where subpaths = concatMap (\e@(a, b, c) -> add e (paths (direction s e) f g (s:v))) n
```

Given a current node and an edge return the neighbour of that node. An edge works in both directions so we have to check both ends of the edge.

```
direction c (a, _, b)
  | c == a = b
  | otherwise = a
```

n is defined as the neighbours of the current vertex which we havent allready visited

```
n = unvisited (neghbours s g) v
```

neighbours function takes a vertex name and a graph. It returns the list of edges that link our given vertex to any other vertex

```
neghbours x r = filter (\(f, c, t) -> (x == f || x == t)) r
```

unvisited takes a list of edges in a graph and a list of verticy names which have allready been visited. It returns the edges in the origonal set which arnt leading to a visited vertex

```
unvisited n v = filter (\(a, b, c) -> (not (c `elem` v)) && (not (a `elem` v))) n
```

add appends an item to the front of each list in a list of lists

```
add e x = map (\a -> e:a) x
```

the find all paths function is just an alias for paths, the reason for this is it simplifys the use of paths, the user dosnt need to pass in the empty array at end.

```
findAllPaths :: (Eq a) => a -> a -> Graph a -> [Graph a]
findAllPaths s f g = paths s f g []
```

Reachable takes two verticies and a graph, returns true iff you can traverse the graph starting from ether of the verticies and reach the other one.

```
reachable :: (Eq a) => a -> a -> Graph a -> Bool
```

reachable gets the array of all paths and checks to ensure the array is non empty. If the array is non empty then there is atleast one path between the nodes and thus they f is reachable from s

```
reachable s f g
  | p == [] = False
  | otherwise = True
  where p = findAllPaths s f g
```

minCostPath works by getting all the paths between the two nodes, creating a list of tuples of the form (cost, path). Then uses the minmumBy functions to find the path with the minumum cost

```
minCostPath :: (Eq a) => a -> a -> Graph a -> Graph a
```

We find the minumum in a list by comparing the cost of each of the paths between the two verticies.

```
minCostPath s f g = snd (L.minimumBy (\(c1, p1) (c2, p2) -> c1 `compare` c2) paths)
```

We create paths by mapping over all the paths between the given verticies and creating a tuple (cost, path)

```
                    where paths = map (\p -> (cost p, p)) (findAllPaths s f g)
```

cost function takes a path and computes the cost of the path

```
                        cost p = sum (map (\(s, c, f) -> c) p)
```

Lazy execution makes the reachable function more efficient because rather than computing all the paths we only have to compute 1 path to show the two verticies can reach each other. Where as the minCostPath has to explore all the paths as we have to compute the cost for all of them and find the smallest.

# Part 2

```
cliques :: (Eq a) => Graph a -> [Graph a]
```

We think of this as two edges are related if you can reach one from the other, we use reachable as our relation. The groupBy function will group two items together if the given lambda returns True when given the edges as input.

```
cliques g = L.groupBy (\(s1, c1, f1) (s2, c2, f2) -> reachable s1 s2 g) g
```