Design Documentation

Design

Data Type ¨C data Tree a = Empty | Root [a] [Tree a] deriving (Show, Read, Eq)
The data type for the 2-3-4 tree that I used consists of nodes. Each node is made up of two lists: one for storing all of the node's contents, and one for storing sub-trees. Those sub-tress are nodes themselves, and therefore can have their own data and sub-trees.

Leaf node ¨C leaf x = Root [x] [Empty]
Empty child node.

Adding a node (insertion function) ¨C addnode :: Ord a => a -> Tree a -> Tree a
Adding a node is a recursive function, where the base case is a node that does not have any sub-trees. When the base case is reached, it simply creates a "leaf node", which contains only 1 data element and no sub-trees.

The general case first checks whether the root node contains 3 elements, and if it does, it splits it accordingly.
It then proceeds to check whether any of the root's children have 3 data elements. This is done, because a node that is not the root of the tree that has 3 data elements, when split, has to pass one element upwards to its "parent" node. This could not be done if it did not remember its parent and the workaround that I chose to apply to this generally recursive program is to check all of the root's sub-trees and apply the correct function if any of them have 3 data elements. This function will be discussed later.
The general case then proceeds to call the addnode function recursively based on where the item should go on the list, until it hits the base case.

Splitting a node
There are two split functions in this program ¨C one simply splits a node into 1 root node and 2 sub-trees, the other also combines it with another node. The first split function is used where the node is a root node and does not need to be combined with anything, the second is used in all other cases.

split :: Ord a => Tree a -> Tree a
Splits the node, with the second data element becoming a root node and the other two elements becoming its sub-trees.

splitandcombine :: Ord a => Tree a -> Tree a -> Tree a
Takes two nodes ¨C a parent and a child to be split. IMPORTANT: The parent that is passed in **must not** have the child as one its sub-trees when this function is used. The child should be extracted from the parent's list and passed in as the second argument.

splitandcombine is not a recursive function ¨C it simply goes over a long list and tries to match the input based on a couple of condition variables. When it finds one that matches, it carries out the commands associated with its condition statement. The function returns a tree that is a combination of the parent + the middle node of the child + a mix of sub-trees that came from both the parent and the child. The mix is created according to the rules of splitting 2-3-4 trees.

Firstly, the function tries to match the input to 2 nodes that have no sub-trees (remember that the child has been extracted from the parent). It then proceeds to match to nodes where some of the lists are "Empty", before finally matching to a case where all lists have some data.

Displaying a tree (display) ¨C display:: Ord a => Tree a -> [a]
The display function is a recursive function that tries to map its input to a lot of different condition

variables. The reason behind this is that the order of displaying nodes in a 2-3-4 tree, as designed by me is not clear when the tree does not have a full list of sub-trees (1-node with 3 data elements may be displayed in 4 different ways, so order needs to be established first). To establish the proper order plenty of condition variables and statements are used since I could not see any better way of doing this part. The code is nevertheless well commented, so it should not be difficult to understand ¨C e.g. "case 2-node with 2 subtrees" means that the program will test for a node with 2 data elements and 2 sub-trees and decide the order.

This function keeps calling itself recursively until it goes over all of the tree's data elements. The base case is any node that does not have any sub-trees ("leaf node").

getRoot:: Ord a => Tree a -> [a]
Helper function that returns all data elements of a node as a list.

getNoRoots:: Ord a => Tree a -> Int
Helper function that returns the number of data elements of a node.

onecase :: Ord a => a -> Tree a -> Tree a
Used by the addnode function to add a new element to a "leaf node" that contains one element before the addition (two elements after onecase completes).

twocase:: Ord a => a -> Tree a -> Tree a
Used by the addnode function to add a new element to a "leaf node" that contains two elements before the addition (three elements after twocase completes).

makeTree:: Ord a => [a] -> Tree a
A function that was used primarily for testing. It takes a list and creates a 2-3-4 tree out of it. I decided to keep it in because, while not required, it is quite relevant to the program.

Testing

The program was tested mainly using the values and the tree provided by Algorithms in Action website (link in the References section).
The following were the most common values used (in the correct order):
[75,5,70,10,65,15,60,20,55,25,50,30,45,35,40]

Problems

Recursion causes the program to sometimes split a node without linking it to its parent if a couple of splits occur one after another. This problem is caused by line 25 of the program, and unfortunately I ran out of time to fix it. A possible solution to this problem may be to remember and pass the tree's root node at every step during recursion, and whenever this particular case is encountered, start the process of recursively adding a node over, which would force the splitandcombine method to work properly.

References

Algorithms in Action - http://ww2.cs.mu.oz.au/aia/
Dr. David Sinclair's homepage - http://www.computing.dcu.ie/~davids/research.html