CA320

Continuous Assessment: 2-3 Trees

Renso Guilalas (21422182)

28th November, 2023

Assignment Description

2-3 Trees are an extension of binary trees. A *2-3 Tree* is either:

- An empty tree;
- A node with 1 root element, a left subtree in which all the elements as less than or equal to the root element, and a right subtree in which all the elements are greater than the root element; or
- A node with 2 root elements, a left subtree in which all the elements are less than or equal to the first root element, a middle subtree in which all the elements are greater than the first root element and less than or equal to the second root element, and a right subtree in which all the elements are greater than the second root element.

The assignment is to implement the following Haskell functions for *2-3 Trees*.

- *add(X, T)* returns the *2-3 Tree* from adding *X* to the *2-3 Tree T*.
- *member(X, T)* returns *true* if *X* is in the *2-3 Tree T*.
- *height(T)* returns the height of *T*.
- *prettyPrint(T)* is always *true* and displays the *2-3 Tree T*.

Description of Implementation

Data Type

In my haskell file I've written a data type called 'TwoThreeTree' which can either be an 'Empty' tree, a 'TwoNode' tree which has one root element and two branches (a binary tree) or a 'ThreeNode' tree which has two root elements and three branches. I've created this data type by modifying the data type from lab session 5 to accommodate a tree with two root elements and 3 branches (See Appendix A).

Implementation of *add(X, T)*

My implementation of the add function is a modified version of the 'addNode' function from lab 5. This function adds a value *X* to a specified tree. It uses pattern matching to check whether the tree is 'Empty', a '2-node' or '3-node' tree and acts accordingly:

- If an element is being added to an empty tree, then the returned tree is a '2-node' tree which contains the added element at the root and two empty subtrees (See Appendix B).
- Elements added to a '2-node' tree will create a '3-node' tree. Depending on the element being added to the '2-node tree', the smaller element is added as the left root while the bigger element is added as the right root. The created tree will contain empty subtrees (See Appendix B).
- I use recursion in my implementation for adding an element to a '3-node' tree. This is done depending on the element being added. If an element x is to be added to a '3-node' tree and the roots of the '3-node' tree are y and z in respective order, then x will be added to the left subtree if it is less than or equal to y. If the element x is greater than y and less than or equal to z, then x will be added to the middle subtree. If the element x is greater than z, then the element is added to the right subtree (See Appendix B).
- I also defined a 'twoLeaf' node pattern as a shortcut for trees with an element and two empty subtrees. However, I've only used it once in my implementation (See Appendix C).

Implementation of *member(X, T)*

The member function returns ‘True’ or ‘False’ based on whether the value ‘X’ is in the tree ‘T’. My implementation uses pattern matching to check whether the tree is ‘Empty’, a ‘2-node’ or ‘3-node’ tree and acts accordingly:

- When checking if an element ‘x’ is in an ‘Empty’ tree it returns ‘False’ (See Appendix D).
- Using recursion, the pattern for checking if an element ‘x’ in a ‘2-node’ tree has a base case which returns ‘True’ if ‘x’ is equal to the element ‘y’, which is the root of the tree. Recursion comes in when ‘x’ is not equal to ‘y’. If ‘x’ is less than ‘y’ then continue the search in the left subtree. And if ‘x’ is greater than ‘y’ then the search is continued in the right subtree. Otherwise, if none of the patterns match then ‘False’ is returned (See Appendix D).
- The pattern for checking whether an element ‘x’ is in a ‘3-node’ tree is similar to the pattern for checking if ‘x’ is in a ‘2-node’ tree. The additions to this are checking if ‘x’ is equal to the second root element ‘z’ of the ‘3-node’ tree and checking whether ‘x’ is greater than ‘y’ and less than or equal to ‘z’. If it matches this then it continues searching within the middle subtree. Everything else is similar to the pattern for ‘2-node’ trees (See Appendix D).

Implementation of *height(T)*

The height function returns the height of a tree ‘T’. It uses pattern matching to check whether the tree is ‘Empty’, a ‘2-node’ or ‘3-node’ tree and acts accordingly:

- If the tree is ‘Empty’ then the height is -1 (See Appendix E).
- Checking the height of a ‘2-node’ tree involves recursion. Need to find the height of its left subtree and right subtree recursively and add 1 to them. Recursion would apply until the subtrees are ‘Empty’. Once the heights of both subtrees are calculated, the greater height is returned (See Appendix E). [1]
- The pattern for checking the height of a ‘3-node’ is similar to the pattern for ‘2-node’ trees. The only addition is checking the height of the middle subtree. The height of the middle subtree is compared to the height of the right subtree. The greater of those two are then compared to the left subtree and the greater of that comparison is taken as the height of the tree (See Appendix E).

Implementation of *prettyPrint(T)*

The prettyPrint function should try to display the *2-3 Tree* in an easily readable format. There were many different sources that I tried to use for my implementation. However, there was one source [2] which I found would be the closest to the format asked in the assignment.

This implementation involves an 'indent' function which takes a list of strings and adds an indentation of eight spaces to each string.

The 'layoutTree' function takes a *2-3 Tree* and returns a list of strings representing a formatted layout of the tree.

- If the tree is "Empty", it returns a list with a single string "nil" (See Appendix F).
- If the tree is a '2-node' with a value 'x' and left and right subtrees, it indents the layout of the left subtree, adds a string representation of the single root 'x', and then indents the layout of the right subtree (See Appendix F).
- If the tree is a '3-node' with values 'x' and 'y' and left, middle and right subtrees, it similarly indents the layouts of the left, middle and right subtrees, and adds a string representation of the two root elements 'x' and 'y' (See Appendix F).

The 'prettyPrint' function takes a *2-3 Tree* and returns a formatted string representation of the tree. It uses the 'layoutTree' function to obtain a list of strings representing the layout of the tree and then uses 'unlines' to concatenate these strings into a single string with newline characters between them.

I ran into a problem when writing this function for printing '3-node' trees. The problem is that I wasn't able to print the middle subtree inline with the root elements. It would instead print the middle subtree one line below its respective root elements (See Appendix G).

References

[1] A. Chugh, "Height of a tree data structure," DigitalOcean, https://www.digitalocean.com/community/tutorials/height-of-a-tree-data-structure (accessed Nov. 28, 2023).

[2] A. Chugh, "Height of a tree data structure," DigitalOcean, https://www.digitalocean.com/community/tutorials/height-of-a-tree-data-structure (accessed Nov. 28, 2023).

Appendices

Appendix A

TwoThreeTree Data Type

```
data TwoThreeTree t
  = Empty
  | TwoNode t (TwoThreeTree t) (TwoThreeTree t)
  | ThreeNode t t (TwoThreeTree t) (TwoThreeTree t) (TwoThreeTree t)
  deriving (Eq, Ord, Show)
```

Appendix B

add(X, T) function

```
add :: (Ord t) => t -> TwoThreeTree t -> TwoThreeTree t
add x Empty = twoLeaf x
add x (TwoNode y left right)
  | x < y = ThreeNode x y left Empty right
  | otherwise = ThreeNode y x left Empty right
add x (ThreeNode y z left middle right)
  | x <= y = ThreeNode y z (add x left) middle right
  | x > y && x <= z = ThreeNode y z left (add x middle) right
  | x > z = ThreeNode y z left middle (add x right)
  | otherwise = error "Something wrong happened"
```

Appendix C

Definition of twoLeaf

```
twoLeaf x = TwoNode x Empty Empty
```

Appendix D

member(X, T) function

```
member :: (Ord t) => t -> TwoThreeTree t -> Bool
member x Empty = False
member x (TwoNode y left right)
   | x == y = True
   | x < y = member x left
   | x > y = member x right
   | otherwise = False
member x (ThreeNode y z left middle right)
   | x == y = True
   | x == z = True
   | x <= y = member x left
   | x > y && x <= z = member x middle
   | x > z = member x right
   | otherwise = False
```

Appendix E

height(T) function

```
height :: TwoThreeTree t -> Int
height Empty = -1
height (TwoNode x left right) = max (height left) (height right) + 1
height (ThreeNode x y left middle right) = max (height left) (max (height middle) (height right)) + 1
```

Appendix F

prettyPrint(T) function

```
indent :: [String] -> [String]
indent = map ("        "++)

layoutTree :: (Show a) => TwoThreeTree a -> [String]
layoutTree Empty = ["nil"]
layoutTree (TwoNode x left right) = indent (layoutTree left) ++ [show x] ++ indent (layoutTree right)
layoutTree (ThreeNode x y left middle right) = indent (layoutTree left) ++ [show (x, y)] ++  indent (layoutTree middle) ++ indent (layoutTree right)

prettyPrint :: (Show a) => TwoThreeTree a -> String
prettyPrint = unlines.layoutTree
```

Appendix G

prettyPrint(T), where T is a *2-3 Tree*