COL765 - Assignment 2

Nilaksh Agarwal
2015PH10813

1 Introduction

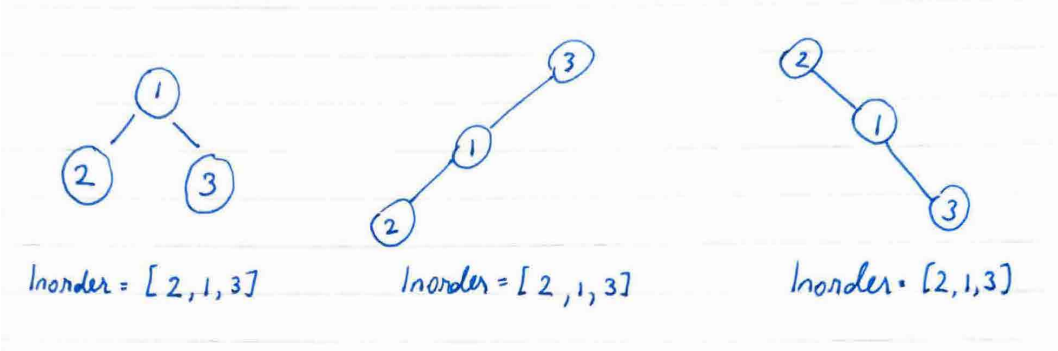
The purpose of this assignment was to develop a module to regenerate a Tree, given it's in-order traversal. I have used the document Rambling Through Woods on a Sunny Morning for reference and also used the Binary Tree signature and structure given in the same.

2 The complications with inorder traversal

Unlike preorder or postorder traversals, inorder traversals have no structural information in them. For example, in a preorder traversal, all values occurring to the right of a given node, are the descendants of that node. Similarly, to the left for postorder traversals.

In an inorder traversal however, there is no such structural information available, and moreover, there is some noise added to this as well.

This is clearly visualized with the 3 following trees, which all have the same inorder traversal:



3 The Binary Tree Signature

Here we define a basic datatype for bintree:

?? ⟨*Node* ??⟩≡ (? ? 0—1)

```
datatype 'a bintree =
        Empty
        | Node of 'a * 'a bintree * 'a bintree
```

We also define a option datatype, since we intend to store the empty Leaf nodes as well

?? ⟨*Option* ??⟩≡ (? ? 0—1)

```
datatype 'a option = NONE | SOME of 'a
```

We define our regular preorder & postorder functions like in the document. Here **'a option list** implies a list which can contain **NONE** in case of Empty subtree or **SOME of value**

?? ⟨*PrePostSig* ??⟩≡ (? 0—1)

```
val preorder : 'a bintree ->  'a option list
val postorder : 'a bintree -> 'a option list
```

However, for inorder, this is not enough. We need some addition *Cosmetic Sugar* to gain the missing structural information in this traversal. Hence, we include the depth of a node defined to be 0 for the root node, and the depth of any child is 1 + the depth of their parent.
So, our node now contains a Tuple (value, depth) which is returned from the inorder traversal. Now we can define the signatures of the inorder and inorder-Inverse

?? ⟨*InSig* ??⟩≡ (? 0—1)

```
val inorder : 'a bintree -> ('a option * int) list
val inorderInverse : ('a option * int) list -> 'a bintree
```

One last function we use to check if two given trees are equal. For this, we find their preorder and postorder traversals, and check their equality.

?? ⟨*equalSig* ??⟩≡ (? 0—1)

```
val checkTrees : ''a bintree * ''a bintree -> bool
```

Now we can put everything together in our Signature

?? ⟨*bintreeSignature-complete* ??⟩≡ (? 0—1)

```
signature BINTREE =
sig
⟨Node ??⟩
⟨Option ??⟩
        exception Empty_bintree;
        exception InvalidTraversal;
⟨PrePostSig ??⟩
⟨InSig ??⟩
⟨equalSig ??⟩
end
```

4 The Binary Tree Structure

Similarly to the document we define the preorder & postorder functions using the tail recursive forms.

We use **NONE** to indicate empty nodes and **SOME of val** to indicate nodes with value **val**

?? ⟨*preorder* ??⟩≡ (? 0—1)

```
local
    fun pre(Empty, Llist) = NONE::Llist
            | pre(Node(N,Ltree, Rtree),Llist) =
            let
                    val Mlist = pre(Rtree,Llist)
                    val Nlist = pre(Ltree,Mlist)
            in
                    SOME N :: Nlist
            end
    in
            fun preorder T = pre(T,[])
    end
```

?? ⟨*postorder* ??⟩≡ (? 0—1)

```
local
    fun post(Empty, Llist) = NONE::Llist
            | post(Node(N,Ltree, Rtree),Llist) =
            let
                    val Mlist = post(Rtree,SOME N::Llist)
                    val Nlist = post(Ltree,Mlist)
            in
                    Nlist
            end
    in
            fun postorder T = post(T,[])
    end
```

In inorder however, we need to store the depth of the node as well, since we are unable to figure out any strucural information from the traversal

?? ⟨*inorder* ??⟩≡ (? 0—1)

```
local
fun ino(Empty, Llist,i) = (NONE,i)::Llist
        | ino(Node(N,Ltree, Rtree),Llist,i) =
        let
                val Mlist = ino(Rtree,Llist,i+1)
                val Nlist = ino(Ltree,(SOME N,i)::Mlist,i+1)
        in
                Nlist
        end
in
        fun inorder T = ino(T,[],0)
end
```

Here we store (value,depth) as a tuple in the node.

4.1 The Inorder Inverse

Before we start the inorder inverse function, we define some facts about our updated inorder Traversal (with heights)

1. *In any inorder traversal ($\perp$,_) is the first and last element*
2. *If any inorder traversal has more than 1 element, the tree is non-empty*
3. *The descendants of a node always have a depth greater than the depth of the node. (**m**,h1) & (**n**,h2) : n is a descendant of m if (h2 >h1)*
4. *Any node m that has a height greater than a node l = (v,h) is either a descendant of l or a descendant of a sibling of l*
5. *Any slice of the form [($\perp$,h+1), (**v**,h), ($\neq \perp \neq$,h+1)] determines a unique leaf node in the tree with height h.*
6. *Any slice of the form [($\perp$,h+2), (**m**,h+1), ($\perp$, h+2), (**l**,h), ($\perp$,h+1)] where $m \neq \perp \neq l$ are values of the nodes, determines a unique subtree rooted at l whose left child is the leaf node m and the right child is empty*
7. *Any slice of the form [($\perp$,h+1), (**l**,h), ($\perp$, h+2), (**m**,h+1), ($\perp$,h+2)] where $m \neq \perp \neq l$ are values of the nodes, determines a unique subtree rooted at l whose right child is the leaf node m and the left child is empty.*
8. *Any slice of the form [(**T1**,h+1), (**v**,h), (**T2**,h+1)] where T1 $\neq \perp \neq$ T2 are subtrees/leaf nodes determines a subtree rooted at v with T1 and T2 as it's left and right children*

Uniqueness of Inorder Traversal: No two distinct binary trees can yield the same inorder traversal. *Proof:* The proof follows from the definition of inorder traversal and the previous facts (especially the if and only if conditions) Further the statements yield an inductive proof along with a case analysis of the inductive step

Now we define some helper functions to create this inorder inverse. The first function converts a inorder traversal into a list of Bintree nodes.

?? ⟨*Nodify* ??⟩≡ (? 0—1)

```
fun  Nodify[] = []
           | Nodify (h::t) =
           Node(h,Empty,Empty)::Nodify(t)
```

The next function joins 3 nodes into a single node if the height of the middle node is one less than the other two.

?? ⟨*joinNodes* ??⟩≡ (? 0—1)

```
fun joinNodes(T1 as Node((v1,h1),_,_), T2 as Node((v2,h2),_,_), T3 as Node
if(h1=h3 andalso (h1-1)=h2) then
        [Node((v2,h2),T1,T3)]
else
        [T1,T2,T3]
| joinNodes(T1,T2,T3) = raise InvalidTraversal
```

The next function goes over a list of notes and tries combining all the successive nodes triplets

?? ⟨*CombineIter* ??⟩≡ (? 0—1)

```
fun combineIter(h1::h2::h3::[]) =
        joinNodes(h1,h2,h3)
| combineIter(L as h1::h2::h3::tl) =
let
        val M = joinNodes(h1,h2,h3) @ tl
in
        List.hd(M) :: combineIter(List.tl(M))
end
| combineIter (L as h1::h2::[]) = L
| combineIter L = raise InvalidTraversal
```

Now, a function keeps calling the iterative joining function until only one node remains

?? ⟨*Combine* ??⟩≡ (? 0—1)

```
fun combine(hd::[]) = hd
| combine L =
let
        val M = combineIter L
in
        combine M
end
```

This tree created has Complete nodes even for Empty Leaf nodes. We need to clean this up and remove the extra Empty Nodes

?? ⟨*treeClean* ??⟩≡ (? 0—1)

```
fun treeClean(Node((SOME v,_),Ltree,Rtree)) =
let
        val LClean = treeClean(Ltree)
        val RClean = treeClean(Rtree)
in
        Node(v, LClean, RClean)
end
| treeClean(T) = Empty
```

Now we can put it all together in our inorderInverse

?? ⟨*inorderInverse* ??⟩≡ (? 0—1)

```
local
    ⟨Nodify ??⟩
    ⟨joinNodes ??⟩
    ⟨CombineIter ??⟩
    ⟨Combine ??⟩
    ⟨treeClean ??⟩
in
    fun inorderInverse(L) =
                    treeClean (combine (Nodify L))

    end
```

Having gotten a tree back from an inorder inverse, we would like to define a function to check if the tree is the same as our original tree. We can do the same by comparing their preorder & postorder traversal. (Since in the document the uniqueness of preorder/postorder traversals has been proved. For this we need to check if two lists generated by the traversals are the same

?? ⟨*checkList* ??⟩≡ (? 0—1)

```
fun checkList([],[]) = true
            | checkList([],_) = false
            | checkList(_,[]) = false
            | checkList(NONE::t1, NONE::t2) = checkList(t1,t2)
            | checkList(SOME h1::t1, SOME h2::t2) =
            if(h1=h2) then
                    checkList(t1,t2)
            else
                    false
            | checkList(_,_) = false
```

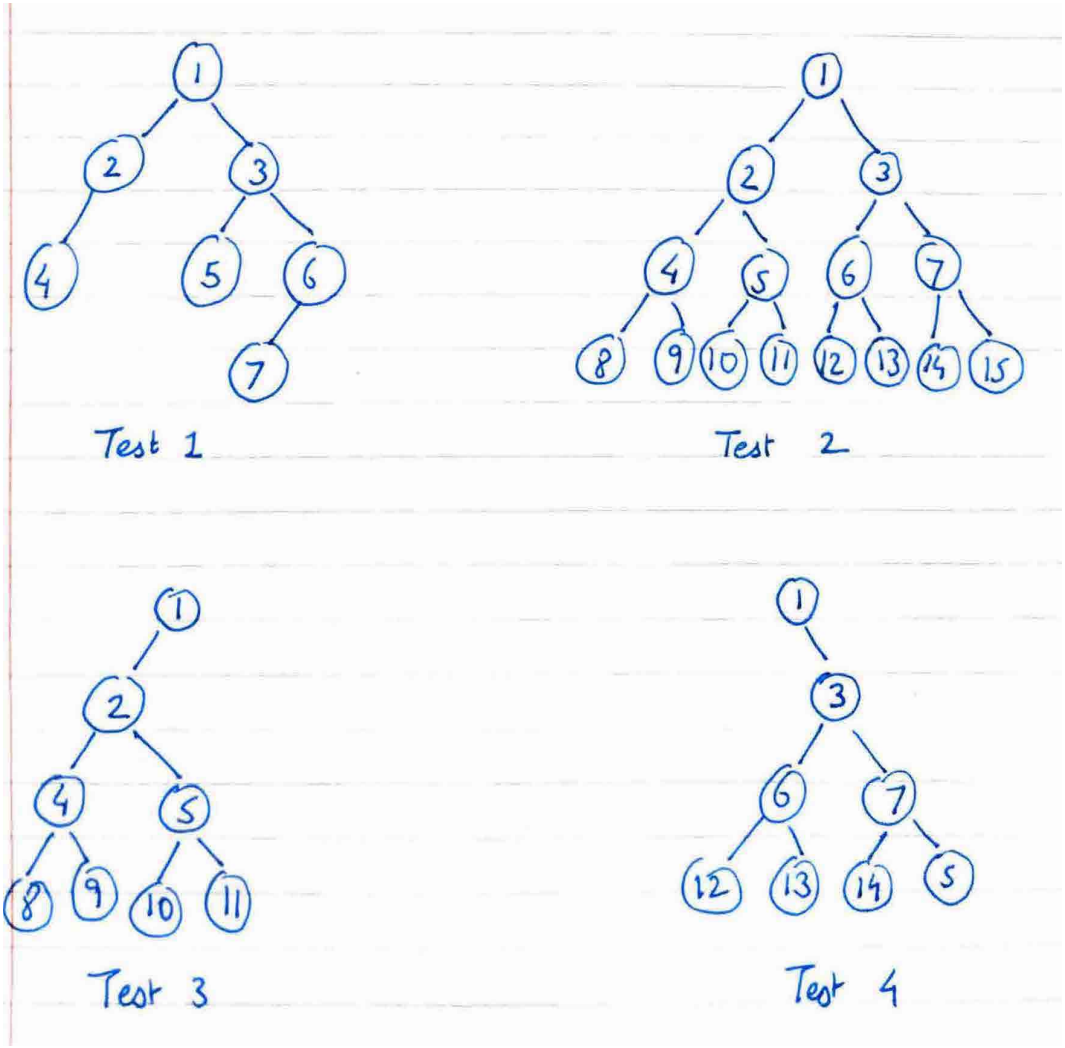
```
??    ⟨checkTrees ??⟩≡                                                    (? 0—1)
          local
                      ⟨checkList ??⟩
              in
                      fun checkTrees(T1,T2) =
                      let
                              val pre1 = preorder(T1)
                              val pre2 = preorder(T2)
                              val pos1 = postorder(T1)
                              val pos2 = postorder(T2)
                      in
                              if(checkList(pre1,pre2) andalso checkList(pos1,pos2)) then
                                      true
                              else
                                      false
                      end
              end
```

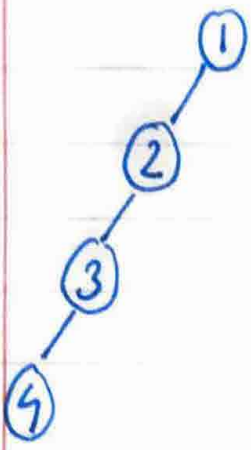
Now we put it all together into our Bintree structure

```
??    ⟨bintreeStructure-complete ??⟩≡                                     (? 0—1)
        ⟨bintreeSignature-complete ??⟩
        structure Bintree : BINTREE =
        struct
            ⟨Node ??⟩
            ⟨Option ??⟩

            exception Empty_bintree
                exception InvalidTraversal

                ⟨preorder ??⟩
                ⟨postorder ??⟩
                ⟨inorder ??⟩
            ⟨inorderInverse ??⟩
            ⟨checkTrees ??⟩
        end
```

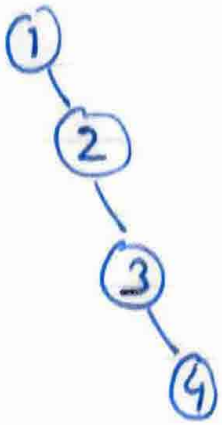
5 Test Cases

We define the following test cases to check the performance of our inorderInverse

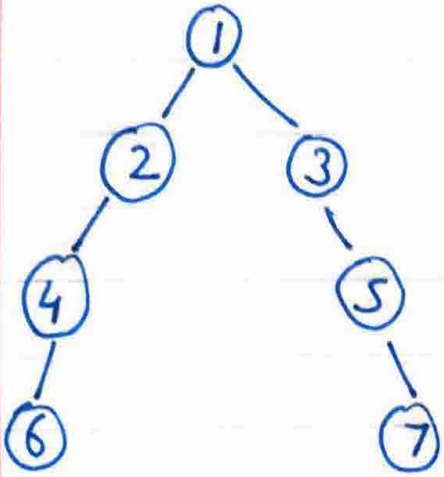




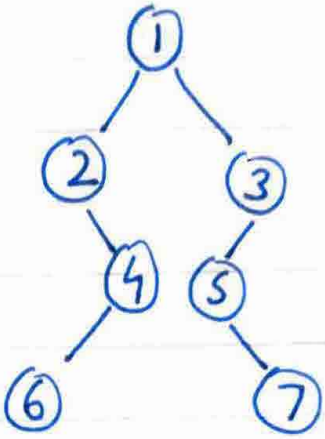
Test 5



Test 6



Test 7



Test 8

?? ⟨*test1* ??⟩≡ (? 0—1)

```
local
        val t7 = Node (7, Empty, Empty);
        val t6 = Node (6, t7, Empty);
        val t5 = Node (5, Empty, Empty);
        val t4 = Node (4, Empty, Empty);
        val t3 = Node (3, t5, t6);
        val t2 = Node (2, Empty, t4);
in
        val test1 = Node (1, t2, t3);
end
```

?? ⟨*test2* ??⟩≡ (? 0—1)

```
local
        val t15 = Node (15, Empty, Empty);
        val t14 = Node (14, Empty, Empty);
        val t13 = Node (13, Empty, Empty);
        val t12 = Node (12, Empty, Empty);
        val t11 = Node (11, Empty, Empty);
        val t10 = Node (10, Empty, Empty);
        val t9 = Node (9, Empty, Empty);
        val t8 = Node (8, Empty, Empty);
        val t7 = Node(7, t14, t15);
        val t6 = Node (6, t12, t13);
        val t5 = Node (5, t10, t11);
        val t4 = Node (4, t8, t9);
        val t3 = Node (3, t6, t7);
        val t2 = Node (2, t4, t5);
in
        val test2 = Node (1, t2, t3);
end
```

?? ⟨*test3* ??⟩≡ (? 0—1)

```
local
        val t11 = Node (11, Empty, Empty);
        val t10 = Node (10, Empty, Empty);
        val t9 = Node (9, Empty, Empty);
        val t8 = Node (8, Empty, Empty);
        val t5 = Node (5, t10, t11);
        val t4 = Node (4, t8, t9);
        val t2 = Node (2, t4, t5);
in
        val test3 = Node (1, t2, Empty);
end
```

?? ⟨*test4* ??⟩≡ (? 0—1)

```
local
        val t15 = Node (15, Empty, Empty);
        val t14 = Node (14, Empty, Empty);
        val t13 = Node (13, Empty, Empty);
        val t12 = Node (12, Empty, Empty);
        val t7 = Node(7, t14, t15);
        val t6 = Node (6, t12, t13);
        val t3 = Node (3, t6, t7);
in
        val test4 = Node (1, Empty, t3);
end
```

?? ⟨*test5* ??⟩≡ (? 0—1)

```
local
        val t4 = Node (4, Empty, Empty);
        val t3 = Node (3, t4, Empty);
        val t2 = Node (2, t3, Empty);
in
        val test5 = Node (1, t2, Empty);
end
```

?? ⟨*test6* ??⟩≡ (? 0—1)

```
local
        val t4 = Node (4, Empty, Empty);
        val t3 = Node (3, Empty, t4);
        val t2 = Node (2, Empty, t3);
in
        val test6 = Node (1, Empty, t2);
end
```

?? ⟨*test7* ??⟩≡ (? 0—1)

```
local
        val t7 = Node (7, Empty, Empty);
        val t6 = Node (6, Empty, Empty);
        val t5 = Node (5, Empty, t7);
        val t4 = Node (4, t6, Empty);
        val t3 = Node (3, Empty, t5);
        val t2 = Node (2, t4, Empty);
in
        val test7 = Node (1, t2, t3);
end
```

?? ⟨*test8* ??⟩≡ (? 0—1)

```
local
        val t7 = Node (7, Empty, Empty);
        val t6 = Node (6, Empty, Empty);
        val t5 = Node (5, Empty, t7);
        val t4 = Node (4, t6, Empty);
        val t3 = Node (3, t5, Empty);
        val t2 = Node (2, Empty, t4);
in
        val test8 = Node (1, t2, t3);
end
```

After this, to check if our inorderInverse works for each test case, we simply checkTrees between the inorderInverse generated tree and the original tree

?? ⟨*testCheck* ??⟩≡ (? 0—1)

```
val test1_check = checkTrees(inorderInverse(inorder(test1)),test1);
val test2_check = checkTrees(inorderInverse(inorder(test2)),test2);
val test3_check = checkTrees(inorderInverse(inorder(test3)),test3);
val test4_check = checkTrees(inorderInverse(inorder(test4)),test4);
val test5_check = checkTrees(inorderInverse(inorder(test5)),test5);
val test6_check = checkTrees(inorderInverse(inorder(test6)),test6);
val test7_check = checkTrees(inorderInverse(inorder(test7)),test7);
val test8_check = checkTrees(inorderInverse(inorder(test8)),test8);
```

?? ⟨*testCase-complete* ??⟩≡

```
⟨bintreeStructure-complete ??⟩
open Bintree;

⟨test1 ??⟩
⟨test2 ??⟩
⟨test3 ??⟩
⟨test4 ??⟩
⟨test5 ??⟩
⟨test6 ??⟩
⟨test7 ??⟩
⟨test8 ??⟩
⟨testCheck ??⟩
```

The reason to choose these particular test cases since it covers some particular ambiguous cases such as a fully dense tree (Test2), a highly skewed tree (Test5 and Test6) as well as a initially skewed and then dense tree (Test3 and Test4) or a highly hollow tree (Test7) as well as a tree containing alternative oriented children (Test8)

The checkTrees function checks the inorderInverse generated tree vs the original tree. For all these testcases, the output is true.