Assignment 2
Recovery From Inorder Traversals

Navreet Kaur
2015TT10917

September 7, 2019

1 Binary Tree datatype representation

1.1 Signature

1a ⟨*BINTREEComplete* 1a⟩≡
```
signature BINTREE =
sig
        type 'a bintree
        exception Empty_bintree
        val Empty : 'a bintree
        val Node : 'a * 'a bintree * 'a bintree -> 'a bintree
        val root : 'a bintree -> 'a
        val leftSubtree : 'a bintree -> 'a bintree
        val rightSubtree : 'a bintree -> 'a bintree
        val isLeaf : 'a bintree -> bool
        val size : 'a bintree -> int
        val preorder : 'a bintree -> 'a option list
        val inorder : 'a bintree -> ('a option * int) list
        val postorder : 'a bintree -> 'a option list
        val inorderInverse : ('a option * int) list -> 'a bintree
end;
```

1.2 Structure

Binary tree data-type described in this document has the following representation.

1b ⟨*datatype* 1b⟩≡ (8a)
```
datatype 'a bintree =
        Empty
        | Node of 'a * 'a bintree * 'a bintree ;
```

Some general purpose functions along with an exception of Binary Tree are defined as follows.

1c ⟨*exceptionEmpty* 1c⟩≡ (8a)
```
exception Empty_bintree;
```

1d ⟨*root* 1d⟩≡ (8a)
```
fun root Empty = raise Empty_bintree
        | root (Node (x, _, _)) = x;
```

2a ⟨*leftSubtree* 2a⟩≡ (8a)

```
fun leftSubtree Empty = raise Empty_bintree
        | leftSubtree (Node (_, LST, _)) = LST;
```

2b ⟨*rightSubtree* 2b⟩≡ (8a)

```
fun rightSubtree Empty = raise Empty_bintree
   | rightSubtree (Node (_, _, RST)) = RST;
```

2c ⟨*isLeaf* 2c⟩≡ (8a)

```
fun isLeaf Empty = false
  | isLeaf (Node (_, Empty, Empty)) = true
  | isLeaf _ = false;
```

2d ⟨*size* 2d⟩≡ (8a)

```
fun size Empty = 0
        | size (Node (_, left, right)) =
        let
                val ls = size left
                val rs = size right
        in 1 + ls + rs
        end;
```

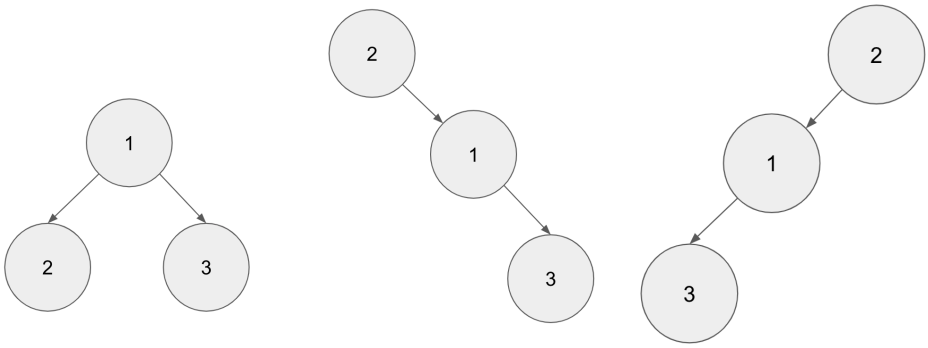
The pre-order and post-order traversals are defined as in the document *Rambling Through the Woods*, which indicate the presence of Empty trees by using the Option data-type.

2e ⟨*preorder* 2e⟩≡ (8a)

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

2f ⟨*postorder* 2f⟩≡ (8a)

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



(a) Tree 1 (b) Tree 2 (c) Tree 3

As dicussed in *Rambling Through the Woods*, unlike the case of preorder and postorder traversals, the use of null for an empty node in the traversal is not enough to guarantee distinct inorder traversals for distinct binary-trees. This is because the parent-child relation is no longer unambiguous. Consider the following trees:

They have the same in-order traversal $[\perp, 2, \perp, 1, \perp, 3, \perp]$ and hence a unique tree cannot be constructed if only the node labels are given in the inorder traversal. Contrary to pre-order and post-order traversals, which do not hide parent-child relation, i.e. a parent node will always appear before the child in pre-order and after in case of post-order, this is not the case with in-order traversals where the parent node may appear before or after the child, depending on whether the child is a right child or a left child. The main problem arrises from the failure of being able to recognise the leaf nodes. We know that the structure of a leaf node would be like $(\perp, x, \perp)$ but that does not prevent either 1 or 2 or 3 from being a leaf node in either of the trees. Hence, the information of just the node label and empty node is not enough to construct a unique tree. It seems that if we are able to unambiguougsly identify the leaf nodes in a given inorder traversal, we will be able to solve this problem. We also want that this information to be minimal. Next, we consider various kinds of information that we can use to identify a leaf node and build upon the algorithm for inverse-inorder.

1.3 Presence of children

This approach requires to encode a boolean value along with the node label to indicate if the node has any children. So we assign the bit 1 to nodes which have at least one child and 0 to the leaf and empty nodes. Consider the inorder traversals for the trees in Figure 1, 2 and 3:

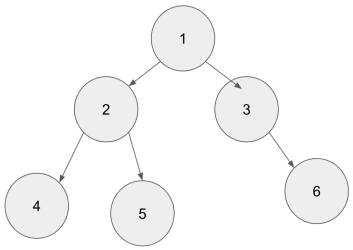
$$I_1 = [(\perp, 0), (2, 0), (\perp, 0), (1, 1), (\perp, 0), (3, 0), (\perp, 0)]$$

$$I_2 = [(\perp, 0), (2, 1), (\perp, 0), (1, 1), (\perp, 0), (3, 0), (\perp, 0)]$$

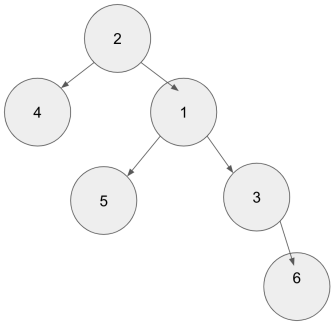
$$I_3 = [(\perp, 0), (2, 0), (\perp, 0), (1, 1), (\perp, 0), (3, 1), (\perp, 0)]$$

Recontructing from I_1 is easy since we know that 2 and 3 are the leaf nodes and the only other node, 1 has at least one child. Similarly unique trees can be constructed from I_2 and I_3 since they have two nodes with at least one child and one leaf node; in case of I_2, knowing that 2 and 1 have at least one child and there is only one leaf node 3, it is easy to say that these two will have an edge between them and 3 will be the child of only one of them. Since (2,1) occurs before (1,1) in the traversal and they have an edge between them, either 1 is the parent of 2 or 2 is the parent of 1. First case does not hold since 2 occurs before 1 in the traversal and hence we get a unique tree. Similar argument follows for I_3.

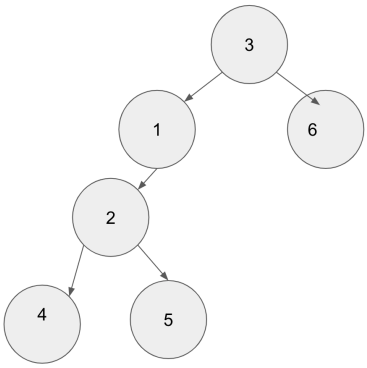
Now, consider the following 3 trees:



(a) Tree 4



(b) Tree 5



(c) Tree 6

All of them have the same inorder traversal:

$$[(\perp, 0), (4, 0), (\perp, 0), (2, 1), (\perp, 0), (5, 0), (\perp, 0), (1, 1), (\perp, 0), (3, 1), (\perp, 0), (6, 0), (\perp, 0)]$$

Even though we are now able to identify the leaf nodes, we have more than one choices for the parent of each node. Even though we know that 4, 5 and 6 are the leaf nodes, it is ambiguous that which node is the parent of 5. This happens as all the nodes occuring to the left of a non-leaf node can be considered to be in its left subtree and all the nodes occuring after the non-leaf node in the traversal can be considered to be in its right subtree, and hence, we have three choices for the root of the tree - 1, 2 and 3 nodes.

1.4 Number of children

?? It appears that if we knew how many children each node had instead of knowing if a node had at least one child or no children at all could help us to assign leaf nodes to their correct parents so that 1 would have exactly two children, 2 would have exactly two children and 3 would have only one child. Therefore, next we consider the case where we use 0, 1 and 2 to specify the number of children a node has. However, this information also does not help us reconstruct a unique tree. Consider trees from Figure 4 and 5 in which node 1 has two children, 2 has 2 children and 3 has one child, with 4, 5 and 6 being the root nodes. Their inorder traversal is represented as :

$$[(\perp, 0), (4, 0), (\perp, 0), (2, 2), (\perp, 0), (5, 0), (\perp, 0), (1, 2), (\perp, 0), (3, 1), (\perp, 0), (6, 0), (\perp, 0)]$$

In these cases, we have 3 choices of roots: 1, 2 and 3. 3 is discarded as it has 1 child but has nodes both before and after itself in the traversal. 2 and 1 can both be roots according to this traversal order and hence we do not get back a unique tree.

1.5 Depth of node

In the previous example, we saw that even though we knew the out degree of each node, we had different options for the selecting the root of the tree and hence no unique tree was gauranteed. If we can preserve the parent child relation in another way, we will be abel to get a unique tree from the inorder traversal. Suppose, along with the empty tree information, we also use the depth of the node in a tree. The inorder traversal for

all the previosly mentioned trees would be:

$$
\begin{aligned}
I_1 &= [(\perp,2),(2,1),(\perp,2),(1,0),(\perp,2),(3,2),(\perp,2)] \\
I_2 &= [(\perp,1),(2,0),(\perp,2),(1,1),(\perp,3),(3,2),(\perp,3)] \\
I_3 &= [(\perp,3),(2,2),(\perp,3),(1,1),(\perp,2),(3,0),(\perp,1)] \\
I_4 &= [(\perp,3),(4,2),(\perp,3),(2,1),(\perp,3),(5,2),(\perp,3),(1,0),(\perp,2),(3,1),(\perp,3),(6,2),(\perp,3)] \\
I_5 &= [(\perp,2),(4,1),(\perp,2),(2,0),(\perp,3),(5,2),(\perp,3),(1,1),(\perp,3),(3,2),(\perp,4),(6,3),(\perp,4)] \\
I_6 &= [(\perp,4),(4,3),(\perp,4),(2,2),(\perp,4),(5,3),(\perp,4),(1,1),(\perp,2),(3,0),(\perp,2),(6,1),(\perp,2)]
\end{aligned}
\tag{1}
$$

All the above in-order traversals are different and represent a unique tree. Recovery of the original tree depends on the positional information given in the traversal. In this case, depth of the node is sufficient to construct a unqiue tree. The argument for the same is given later in this section. The following function can be used to find the inorder traversal of the tree.

5 ⟨*inorder* 5⟩≡ (8a)

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

For any list (which is a permutation of node-depth pairs that make up a binary tree), it is possible to recover the original tree from this list, by the following algorithm. Before we present this algorithm, we need to state the following fact: For any three node-depth pairs represented as $(v_1, d_1), (v_2, d_2), (v_3, d_3)$ in this order in any inorder traversal, $v2$ is the parent of both v_1 and v_3, where v_1 is the left child and v_3 is the right child of v_2 if and only if $d_1 = d_3$ and $d_2 = d_3 - 1 = d_1 - 1$.

Claim:
Such $(v_1, d_1), (v_2, d_2), (v_3, d_3), (v_4, d_4), (v_5, d_5)$ will not exist where $d_1 = d_3 = d_5 = d_2 + 1 = d_4 + 1$

Theorem (Distinctness of Inorder Traversal) *No two distinct binary-trees can be constructed from the same inorder traversal.*

Assume I represents a putative inorder traversal with $|I| = n > 3$ elements and hence could represent a non-trivial binary-tree. We proceed as follows.

Algorithm(Reconstruct from Inorder traversal):

1. *Convert each node-depth (v_1, d_1) pair into a Node with value $(SOMEv_1, d_1)$ and left and right subtree as Empty. For nodes with values $(\perp, d)$, convert it into Node with value (NONE, d)*
2. ***While*** $|I| = n > 1$*:*
   - *Find all node triples (N_1, N_2, N_3) in I having values of the form $(v_1, d+1), (v_2, d), (v_3, d+1)$*
   - *Combine them into one Node N' having value (v_2, d), N_1 as the left child and N_3 as the right child. In the original list I, replace these triples with node N'. In this way, the length of the list I keeps decreasing by combibing the node triples and at last we are left with onle one node representing the whole tree.*
3. *Convert the values of all nodes in the returned tree from $(SOMEv, d)$ to v and $(NONE, d)$ to Empty*

From the algorithm, the first step is implemented as follows.

6a ⟨*convert2node* 6a⟩≡ (7d)
```
fun convert2node (SOME x,y) = Node((SOME x ,y),Empty,Empty)
                | convert2node (NONE, y) = Node((NONE,y),Empty,Empty);
```

6b ⟨*recover* 6b⟩≡ (7d)
```
fun recover L =
        let val node_list = map convert2node L
                in List.hd(recover_node node_list)
        end;
```

The following function combines the valid triples in a given list.

7a ⟨*recoverIt* 7a⟩≡ (7d)

```
fun recover_it [] = []
        | recover_it [N] = [N]
        | recover_it [a,b] = [a,b]
        | recover_it [Node((v1,h1),l1,r1), Node((v2,h2),l2,r2), Node((v3,h3),l3,r3)] =
        if h1=h3 andalso h2=h1-1
        then [Node((v2,h2), Node((v1,h1),l1,r1), Node((v3,h3),l3,r3))]
        else [Node((v1,h1),l1,r1), Node((v2,h2),l2,r2), Node((v3,h3),l3,r3)]
        | recover_it (h1::h2::h3::t) =
                let
                        val head = recover_it([h1,h2,h3])@t
                in List.hd(head)::recover_it(List.tl(head))
                end;
```

The following function calls the previous function iteratively till only one node is left in the list.

7b ⟨*recoverNode* 7b⟩≡ (7d)

```
fun recover_node [] = []
        | recover_node [X] = [X]
        | recover_node L =
        let val L_ = recover_it L
                in recover_node L_
        end;
```

This function is for transforming a Node having label-depth pair as its value to only its label, where NONEs are converted to *Empty*.

7c ⟨*convert* 7c⟩≡ (7d)

```
fun convert (Empty) = Empty
        | convert (Node((NONE,y), Empty, Empty)) = Empty
        | convert (Node((SOME x, y),Empty,Empty)) = Node(valOf(SOME x),Empty,Empty)
        | convert (Node((SOME x, y), left, right)) =
        let val leftval = convert (left);
                val rightval = convert (right);
        in Node(valOf(SOME x),leftval,rightval)
        end;
```

Finally, the inorder-inverse function :

7d ⟨*inorderInverse* 7d⟩≡ (8a)

```
local
⟨convert2node 6a⟩
⟨recoverIt 7a⟩
⟨recoverNode 7b⟩
⟨convert 7c⟩
⟨recover 6b⟩
in
  fun inorderInverse L =
        let val T_ = recover L
                in convert T_
        end;
end;
```

We are now ready to present a complete module as follows. The signature of this structure is 1.1.

8a ⟨*BinarytreeComplete* 8a⟩≡

```
use "BINTREEComplete.sml";
structure Binarytree : BINTREE = struct
  ⟨datatype 1b⟩
  ⟨exceptionEmpty 1c⟩
  ⟨root 1d⟩
  ⟨leftSubtree 2a⟩
  ⟨rightSubtree 2b⟩
  ⟨isLeaf 2c⟩
  ⟨size 2d⟩
  ⟨preorder 2e⟩
  ⟨postorder 2f⟩
  ⟨inorder 5⟩
  ⟨inorderInverse 7d⟩
end;
```

Note: *The code chunks having "Complete" as their suffix represent the complete signature and structure files*

1.6 Test Cases

The following code can be used to test the *inorderInverse* function for different cases.

8b ⟨*testComplete* 8b⟩≡

```
use "BinarytreeComplete.sml";
open Binarytree;
⟨tree1 8c⟩
⟨tree2 9a⟩
⟨tree3 9b⟩
⟨tree4 9c⟩
⟨tree5 10a⟩
⟨tree6 10b⟩
⟨tree7 10c⟩
⟨tree8 11⟩
```

where This is the example from *Rambling throught the woods* - an unbalanced tree with max depth 2.

8c ⟨*tree1* 8c⟩≡ (8b)

```
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 tree1 = Node (1, t2, t3);
        val I1 = inorder tree1;
        val P1 = preorder tree1;
        val T1 = inorderInverse I1;
end;
```

This is a tree fully balanced binary tree with maximum depth 3.

9a ⟨*tree2* 9a⟩≡ (8b)

```
local
        val t8 = Node(8,Empty,Empty);
        val t9 = Node(9,Empty,Empty);
        val t10 = Node(10,Empty,Empty);
        val t11 = Node(11,Empty,Empty);
        val t12 = Node(12,Empty,Empty);
        val t13 = Node(13,Empty,Empty);
        val t14 = Node(14,Empty,Empty);
        val t15 = Node(15,Empty,Empty);
        val t4 = Node(4,t8,t9);
        val t5 = Node(5,t10,t11);
        val t6 = Node(6,t12,t13);
        val t7 = Node(7,t14,t15);
        val t2 = Node(2,t4,t5);
        val t3 = Node(3,t6,t7);
in
        val tree2 = Node(1,t2,t3);
        val I2 = inorder tree2;
        val P2 = preorder tree2;
        val T2 = inorderInverse I2;
end;
```

This tree has only left subtree, with a choice between 1 and 2 as the root. This is similar to the case of Trees 4, 5 and 6 discussed in section **??**

9b ⟨*tree3* 9b⟩≡ (8b)

```
local
        val t3 = Node(3,Empty,Empty);
        val t4 = Node(4,Empty,Empty);
        val t2 = Node(2,t3,t4);
in
        val tree3 = Node(1,t2,Empty);
        val I3 = inorder tree3;
        val P3 = preorder tree3;
        val T3 = inorderInverse I3;
end;
```

This tree has only right subtree, with a choice between 1 and 2 as the root. This is similar to the case of Trees 4, 5 and 6 discussed in section **??**

9c ⟨*tree4* 9c⟩≡ (8b)

```
local
        val t3 = Node(3,Empty,Empty);
        val t4 = Node(4,Empty,Empty);
        val t2 = Node(2,t3,t4);
in
        val tree4 = Node(1,Empty,t2);
        val I4 = inorder tree4;
        val P4 = preorder tree4;
        val T4 = inorderInverse I4;
end;
```

A tree having only left branches at each node.

10a ⟨*tree5* 10a⟩≡ (8b)

```
local
        val t4 = Node(4,Empty,Empty);
        val t3 = Node(3,t4,Empty);
        val t2 = Node(2,t3,Empty);
in
        val tree5 = Node(1,t2,Empty);
        val I5 = inorder tree5;
        val P5 = preorder tree5;
        val T5 = inorderInverse I5;
end;
```

A tree having only right branches at each node.

10b ⟨*tree6* 10b⟩≡ (8b)

```
local
        val t4 = Node(4,Empty,Empty);
        val t3 = Node(3,Empty,t4);
        val t2 = Node(2,Empty,t3);
in
        val tree6 = Node(1,Empty,t2);
        val I6 = inorder tree6;
        val P6 = preorder tree6;
        val T7 = inorderInverse I6;
end;
```

A tree having only left branches at the left subtree and only right branches at the right subtree.

10c ⟨*tree7* 10c⟩≡ (8b)

```
local
        val t6 = Node(6,Empty,Empty);
        val t7 = Node(7,Empty,Empty);
        val t4 = Node(4,t6,Empty);
        val t5 = Node(5,Empty,t7);
        val t2 = Node(2,t4,Empty);
        val t3 = Node(3,Empty,t5);
in
        val tree7 = Node(1,t2,t3);
        val I7 = inorder tree7;
        val P7 = preorder tree7;
        val T7 = inorderInverse I7;
end;
```

A zigzag tree

11 ⟨*tree8* 11⟩≡ (8b)

```
local
        val t6 = Node(6,Empty,Empty);
        val t7 = Node(7,Empty,Empty);
        val t4 = Node(4,t6,Empty);
        val t5 = Node(5,Empty,t7);
        val t2 = Node(2,Empty,t4);
        val t3 = Node(3,t5,Empty);
in
        val tree8 = Node(1,t2,t3);
        val I8 = inorder tree8;
        val P8 = preorder tree8;
        val T8 = inorderInverse I8;
end;
```

It can be seen by comparing the preorder traversal of the recovered tree and the original tree that the recovered tree is indeed the original tree from which the inorder traversal was constructed.

1.7 Complexity

In this section, we discuss the complexity of the *inorderInverse* function.

- *convert2node*: If n is the total number of nodes in the tree, including the empty nodes, it takes $O(n)$ time to convert each value-depth pair to a Node
- *recoverIt*: For finding the valid triples and combining them in one pass, $O(n)$ time is spent.
- *recoverNode*: Assuming a complete binary tree, at depth d there are d^2 number of nodes. At each iteration, we can assume that $d^2 + (d-1)^2$ nodes are combined to form $(d-1)^2$ nodes which requires $O(n)$ time. With each iteration, the maximum d keeps decreasing so that total number of uncombined nodes in the list keep decreasing. At max, the number of iterations that run are $O(n)$. Hence this function takes $O(n^2)$ time.
- *convert*: It takes $O(1)$ time to convert a Node having value as label-depth pair to a Node having value as label.
- *recover*: For converting n nodes to have only their label as the value, it takes $O(n)$ time.

Therefore, the overall complexity of the algorithm is $O(n^2)$