1 Inorder Traversal

1.1 Introduction

Inorder traversal of tree is means of traversal of tree by which we can visit every node.
Inorder traversal

1. First, visit all the nodes in the left subtree
2. Then the root node
3. Visit all the nodes in the right subtree

1.2 Recovery

We can't recover binary tree from its inorder traversal with approach discussed in the document Rambling through Woods on a Sunny Morning.
We have to add some extra information in inorder traversal to recover it.
One idea that will not work is to store the number of children for each node in inordertraversal.
Consider the following binary tree

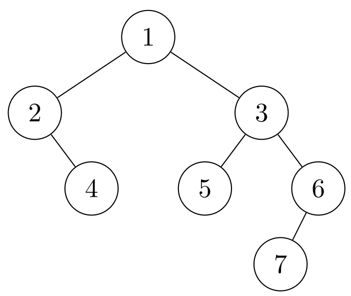

Figure 1: Binary tree

If we encode number of children information to each node then we get following inorder traversal

$$[(2,1),(4,0),(1,2),(5,0),(3,2),(7,0),(6,1)]$$

where first element in tuple is node value and second element is its number of children. But there can be another tree with same inorder traversal and same children count. Following is the another tree with the same inorder traversal and same children count for each node.

So our idea of storing children count of each node with inorder traversal not works as we have counterexample.
So we have think about another idea.

1.3 Recovery using level

What if we store level of each node along with the inorder traversal? Then we can find root of tree easily. Root of tree will be node with level 0. And there

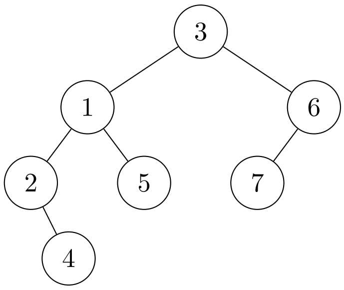

Figure 2: Binary tree with same inorder traversal and number of children

will be only one node with level 0. If we split inorder list about root, we also get inorder list for its left subtree and right subtree. The root for each subtree will be the node with minimum level.

⟨*inorder* ⟩≡ (6b)

```
local
        fun ino (Empty, Llist, Level) = Llist
                | ino (Node (N, LST, RST), Llist, Level) =
                        let val Mlist = ino (RST, Llist, Level+1)
                                val Nlist = ino (LST, (N,Level)::Mlist, Level+1)
                        in Nlist
                        end
in
        fun inorder T = ino (T, [],0)
end
```

The above inorder function do inorder traversal of binary tree. While doing inorder traversal it also store level of node in tuple. It returns the list of tuples, where each tuple is made of node value and node level.
When we do inorder on example tree we get the following inorder traversal

$$[(2,1),(4,2),(1,0),(5,2),(3,1),(7,3),(6,2)]$$

Here we can see that there is only one node with level 0 which has node value 1. So it must be root of tree and all left side of it is inorder traversal of its left subtree and all right side of it is inorder traversal of right subtree.
So here is the function of finding root of any subtree.

3a ⟨*getRoot* 3a⟩≡ (6b)

```
fun getRoot [] = NONE
        |  getRoot(List) =
                let
                        fun getMin(List: ('a * int) list) =
                                if null (tl List) then (hd List)
                                else
                                        let
                                                val tl_ans = getMin(tl List)
                                        in
                                                if #2 (hd List) < #2 (tl_ans) then (hd List)
                                                else tl_ans
                                        end
                in
                        SOME (getMin(List))
                end
```

Once we found the root, we need to split the list around root to get the inorder traversal of left subtree and right subtree.

3b ⟨*split* 3b⟩≡ (6b)

```
fun split(Min: int, L1: ('a * int) list, L2: ('a * int) list) =
        if Min = #2 (hd L2) then (L1, tl L2)
        else
                let
                        val new_l1 = L1@[hd L2]
                        val new_l2 = tl L2
                in
                        split(Min, new_l1, new_l2)
                end
```

Once we split the list we can build the tree recursively.

4a ⟨*inorderInverse* 4a⟩≡ (6b)
```
fun inorderInverse [] = Empty
        |       inorderInverse List =
                let
                        val root = valOf(getRoot(List))
                        val rootVal = #1 root
                        val rootLevel = #2 root
                        val (LSTList, RSTList) = split(rootLevel, [], List)
                        val (LST, RST) = (inorderInverse(LSTList), inorderInverse(RSTList))
                in
                        Node(rootVal, LST, RST )
                end
```

1.4 Complexity

In the above method we store node level along with the node value. So the our algorithm makes the two times size of inorder traversal list compared to normal inorder traversal.

1.4.1 Comparison with leaf information storing

If binary tree have N nodes then it have total floor $\frac{N+1}{2}$ leaf node. So the leaf node information method will take at least $2*\frac{N+1}{2}$ = N+1 extra information to store along with node information. One extra information will also have to store for degree-1 node. So we were storing 2 extra information for each leaf node and 1 extra information for each degree-1 node
Our algorithm of storing level for each node only takes N extra information in all cases.
So our algorithm for storing extra information along with traversal take less number of information but important information.

2 Making Module

Now we will make module
Datatype representation of binary tree

4b ⟨*datatype* 4b⟩≡ (6c)
```
datatype 'a bintree =Empty |
        Node of 'a * 'a bintree * 'a bintree
```

Signature of binary tree

5a ⟨*signature* 5a⟩≡ (6c)

```
signature BINTREE =
        sig

                val root : 'a bintree -> 'a
                val leftSubtree : 'a bintree -> 'a bintree
                val rightSubtree: 'a bintree -> 'a bintree
                val height : 'a bintree -> int
                val size : 'a bintree -> int
                val isLeaf : 'a bintree -> bool
                val inorder : 'a bintree -> ('a *int) list
                val getRoot : ('a * int) list -> ('a * int) option
                val split : int * ('a * int) list * ('a * int) list -> ('a * int) list * ('a * int
                val inorderInverse: ('a * int) list -> 'a bintree
        end (* sig *)
```

Now we will define basic binary tree functions

5b ⟨*emptyexcetion* 5b⟩≡ (6a)

```
exception Empty_bintree;
```

5c ⟨*root* 5c⟩≡ (6a)

```
fun root Empty = raise Empty_bintree
        | root (Node (x, _, _)) = x;
```

5d ⟨*leftsubtree* 5d⟩≡ (6a)

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

5e ⟨*rightsubtree* 5e⟩≡ (6a)

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

5f ⟨*height* 5f⟩≡ (6a)

```
fun height Empty = 0
        | height (Node (_, left, right)) =
                let
                        val lh = height left
                        val rh = height right
                in 1 + Int.max(lh, rh)
                end;
```

5g ⟨*size* 5g⟩≡ (6a)

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

5h ⟨*isleaf* 5h⟩≡ (6a)

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

So the basic binary tree functions will look like this

6a ⟨*basicbintree* 6a⟩≡ (6b)
⟨*emptyexcetion* 5b⟩
⟨*root* 5c⟩
⟨*leftsubtree* 5d⟩
⟨*rightsubtree* 5e⟩
⟨*height* 5f⟩
⟨*size* 5g⟩
⟨*isleaf* 5h⟩

Now structure of module will look like this

6b ⟨*structure* 6b⟩≡ (6c)
```
structure Bintree : BINTREE =
   struct
   ⟨basicbintree 6a⟩
   ⟨inorder ⟩
   ⟨getRoot 3a⟩
   ⟨split 3b⟩
   ⟨inorderInverse 4a⟩
   end
```

Now complete module will look like this

6c ⟨*2019MCS2565-module-complete.sml* 6c⟩≡
⟨*datatype* 4b⟩
⟨*signature* 5a⟩
⟨*structure* 6b⟩

3 Test cases

Test cases

6d ⟨*use* 6d⟩≡ (6 7)
```
use "2019MCS2565-module-complete.sml";
```

6e ⟨*test* 6e⟩≡ (6 7)
```
val traversal = Bintree.inorder t1;
val tree = Bintree.inorderInverse traversal;
```

Test case 1
This is test case for skewed tree which has only right child

6f ⟨*2019MCS2565-case1-complete.sml* 6f⟩≡
```
⟨use 6d⟩
val t4 = Node (4, Empty, Empty);
val t3 = Node (3, Empty, t4);
val t2 = Node (2, Empty, t3);
val t1 = Node (1, Empty, t2);
⟨test 6e⟩
```

Test case 2

This is test case for example tree given in this document

7a ⟨*2019MCS2565-case2-complete.sml* 7a⟩≡

```
⟨use 6d⟩
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);
val t1 = Node (1, t2, t3);
⟨test 6e⟩
```

Test case 3

This is test case for skewed tree which has only left child

7b ⟨*2019MCS2565-case3-complete.sml* 7b⟩≡

```
⟨use 6d⟩
val t4 = Node (4, Empty, Empty);
val t3 = Node (3, t4, Empty);
val t2 = Node (2, t3, Empty);
val t1 = Node (1, t2, Empty);
⟨test 6e⟩
```

Test case 4

This is test case single node tree

7c ⟨*2019MCS2565-case4-complete.sml* 7c⟩≡

```
⟨use 6d⟩
val t1 = Node (1, Empty, t1);
⟨test 6e⟩
```