Assignment 2:Weaving through Entangled Webs in the Wood

September 21, 2019

2019MCS2574

Abstract

In this document we explore the **Inorder Traversal** and **reconstruction of Binary tree** from given Inorer traversal by adding mininum "cosmetic sugar" in the Inorder traversal to yield the same tree.

As discussed in the document Rambling through Woods on a Sunny Morning, a binary tree can be recovered from its preorder (or postorder) traversal. Although the same approach can not be used to recover the tree from inorder traversal. The issue arises due to the difference in type of information that is abstracted by these traversals. Preorder (or postorder) traversal does not hide the parent-child relation viz. a parent node will always appear before a child node in preorder traversal (after in postorder traversal); although the parity (number of children) of the parent node is lost. However inorder traversal does not have any such guarantee.

Introduction

There are multiple ways to reconstruction of binary tree.

- **Add levels of each node.** But this information is not feasible, because in worst case of skewed tree it would add level information as much as number of nodes in the tree. Hence for a large value of nodes it would make node size large.
- **Add child information for each node**, this method is even more space consuming as each node will need to store two information i.e its left and right child.
- Hence in the document we will try to minimize this information handling of each node, a.k.a "cosmetic sugar".

1 Binary Tree representation

We define a Binary Tree of form that is either Empty or is two child. A leaf node in is actually a subtree which has two Empty subtrees. Internal nodes with out-degree 1 have to be represented with the child being either a left-child or a right-child with the other child being Empty.

2a ⟨*Datatype* 2a⟩≡ (11)

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

1.1 Root

We can retrieve root of our tree by runing a simple SML code described below. Similarly we can find LST, RST, HEIGHT, ISLEAF etc. easily with this representation of Binary tree.

2b ⟨*Root* 2b⟩≡ (6)

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

Initial thought process

- We need to focus on the constraint that we need to minimize the ”cosmetic sugar” in the node.
- Instead of storing information such as height, levels, childs which are variable for each node. we will try to add a constant value to each node.
- We will use infix property that: infix expressions require cosmetic sugar such as parentheses, associativity and precedence of operators to enable parsing of expressions unambiguously.
- Similarly we will try to add a BIT information to each node representing left or right association to a root node.

2 Inorder Traversal and adding ”cosmetic sugar”.

For Inorder Traversal we will use Euler Tour and do some modification to add mininum extra information to make it compatible for reconstruction.

We will check whether the child is left or right by a bit information where ”SOME 0” represents left child and ”SOME 1” represents right child. for this addtion of information we will use two functions **add0, add1**.The functions are defined below.

We have taken help of Option datatype where we can use NONE to define Empty and SOME (X) to define a value X.

Add0 and Add1

3 ⟨*Add0-Add1* 3⟩≡ (4a)

```
fun add0 (NONE,NONE)=(NONE,SOME (0))
|   add0 (NONE,SOME (0))=(NONE,SOME (0))
|   add0 (NONE,SOME (1))=(NONE,SOME (1))
|   add0 (SOME (x),NONE)=(SOME (x),SOME(0))
|   add0 (SOME (x),SOME(1))= (SOME (x),SOME(1))
|   add0 (SOME (x),SOME(0))= (SOME (x),SOME(0))
|   add0 (_,_) = (SOME 0, SOME 0)
fun add1 (NONE,NONE)=(NONE,SOME (1))
|   add1 (NONE,SOME (0))=(NONE,SOME (0))
|   add1 (NONE,SOME (1))=(NONE,SOME (1))
|   add1 (SOME (x),NONE)=(SOME (x),SOME(1))
|   add1 (SOME (x),SOME(1))= (SOME (x),SOME(1))
|   add1 (SOME (x),SOME(0))= (SOME (x),SOME(0))
|   add1 (_,_) = (SOME 0, SOME 0)
```

2.1 Euler-tour

Here we use the above two defined functions and recursively add the BIT information to each node.

- Root gets NONE.
- Left child gets SOME 0.
- Right child gets SOME 1.

4a ⟨*EulerTour* 4a⟩≡ (11)

```
fun     eulerTour (Empty) = [(NONE,NONE)]
|       eulerTour (Node (x, Empty, Empty)) =
        ( [(NONE,SOME (0)), (SOME (x),NONE), (NONE,SOME (1))]) (* leaf node *)
|       eulerTour (Node (x, left, right)) = (* non-leaf node *)
           let
            val  inL = eulerTour left
            val  inR = eulerTour right
⟨Add0-Add1 3⟩
            in
              (((map add0) inL)@((SOME (x),NONE)::((map add1) inR)))
            end;
```

2.2 Inorder

We get modified preorder traversal with added information using above modified Euler Tour.

4b ⟨*inorder* 4b⟩≡ (11)

```
fun inorder bt= eulerTour bt;
```

3 Reconstructing the Binary Tree

Here we will try to focus on trying to retrive binary tree from single Inorder traversal using the **CII** which is a **Slice** of tree that represents a valid subtree as discussed in the document **Rambling through Woods on a Sunny Morning**.

Algorithm

1. Find the indices of all NONE elements in I and express them as CIIs i.e. as ordered pairs of the form (i, i) where i, 0 i ¡ n, is the index of a NONE element. Let this be list P I of ordered pairs representing Empty subtrees. Further let $\mid PI \mid$ be the number of CIIs in P I and$\|PI\|=$

   $$\sum_{(i,j)\in PI}(j-i+1)$$

   represent the number of elements from I that are in P I. Then n $\|PI\|$ is the number of elements in I that are not present in P I. At the end of this step $\|PI\| = \mid PI \mid$ and P I contains exactly the number of occurrences of NONE in I .

2. **Recursively**

   (a) Find all pairs of neighbours (i, j), (k, m) in P I with 0 < i ≤ j < k = j + 2 ≤ m < n and Node(i,j) is SOME (0) and Node(k,m) is SOME (1) in the list P I and $I_{j+1} \neq$ NONE. This would require looking up I for the value at index j+1 to determine neighbourhood. Further if $I_{j+1} =$ NONE then I is not a valid Inorder traversal.

   (b) Join neighbours (i, j), (k, m) to form the CII (i, m) in which both (i, j) and (k, m) are nested. In fact, the CII so formed is longer than the sum of the lengths of the CIIs (i, j) and (k, m), thus guaranteeing that$\mid PI \mid$ always decreases, but since $\|PI\|$ increases, the bound function n -$\|PI\|$ decreases with every recursive call.

until

- either P I reduces to a (list with a) single element (0, n1), in which case I is indeed a valid Inorder traversal.

- or P I reduces to a list with more than one element and no neighbours, in which case I is an invalid Inorder traversal.

3.1 Proof of correctness

- We have added BIT information in node to represent the paranthesis of INFIX notation as discussed in abstract.

- We have resolved the issue of leaf and non-leaf node problem by removing those information and adding paranthesis to each node.

- This paranthesis satisfies the stack property as each pair paranthesis is linked to its left , root and right node. hence our algorithm will always satisfy the reconstruction of binary tree.

4 Binary Tree Reconstruction Code

4.1 Inorder Inverse

This is code for reconstruction of Binary tree from modified Inorder Traversal.

6 ⟨*inorderInverse* 6⟩≡ (11)

```
fun     inorderInverse []= raise InvalidInorderTraversal
|       inorderInverse [(NONE,NONE)]= Empty
|       inorderInverse [_,_]= raise InvalidInorderTraversal
|       inorderInverse [(NONE,SOME 0),(SOME x,NONE),(NONE,SOME 1)]=
                                          Node(x,Empty,Empty)
|       inorderInverse [_,_,_]= raise InvalidInorderTraversal
|       inorderInverse Ibt= let

val arP = Array.fromList(Ibt)
val n = Array.length arP

⟨Root 2b⟩


⟨findEmpties 7a⟩


⟨areNeighbours 7b⟩


⟨joinNeighbours 8a⟩


⟨keepJoiningNeighbours 8b⟩


⟨eraseIndices 9a⟩


in

        eraseIndices(keepJoiningNeighbours(findEmpties(Ibt)))

end;
```

4.2 findEmpties

This creates a list NONEs which contains all NONE in I with their CII and BIT information.

7a ⟨*findEmpties* 7a⟩≡ (6)

```
fun findEmpties PIT =
        let fun findNONEs ([], _, T) = T
|               findNONEs (((NONE,SOME 0)::t), i, T) =
                findNONEs (t, i+1, (Node (((i,i),(NONE),SOME 0),Empty,Empty))::T)
|               findNONEs (((NONE,SOME 1)::t), i, T) =
                findNONEs (t, i+1, (Node (((i,i),(NONE),SOME 1),Empty,Empty))::T)
|               findNONEs (((SOME _,_)::t), i, T)= findNONEs (t, i+1, T)
|               findNONEs (((_,_)::t), i, T)= findNONEs (t, i+1, T)
        val NONEs = findNONEs (PIT, 0, [])
        in
        rev NONEs
        end;
```

4.3 areNeighbours

This Checks whether the given CIIs are valid neighbours .

7b ⟨*areNeighbours* 7b⟩≡ (6)

```
exception Out_of_Range
fun     areNeighbours((i,j), (k, m),LR0,LR1) =
                        let val inRange = (i >= 0) andalso (i < n) andalso
                        (j >= 0) andalso (j < n) andalso
                        (k >= 0) andalso (k < n) andalso
                        (m >= 0) andalso (m < n)
                        in if inRange
                        then if (i<=j) andalso (j<k) andalso (k<= m)
                        andalso (LR0=(SOME 0)) andalso  (LR1=(SOME 1))
                        then (case Array.sub (arP, j+1) of
                        (NONE,_) => false
                        | (SOME _,_) => (k=j+2) (* Inorder *)
                        )
                        else false
                        else raise Out_of_Range
                        end;
```

4.4 joinNeighbours

This joins two CIIs to single CII if both are valid neighbours.

8a ⟨*joinNeighbours* 8a⟩≡ (6)

```
fun     joinNeighbours[] = []
|       joinNeighbours[bt] = [bt]
|       joinNeighbours(bt0::(bt1::btList')) =
        let val ((i,j), rootval0,LR0) = root bt0
        val ((k,m), rootval1,LR1) = root bt1
        in if areNeighbours((i,j), (k, m),LR0,LR1)
        then let val cii = (i, m)
                 val (rt,LR) = Array.sub (arP, j+1) (* Inorder *)
                 val bt = Node ((cii, rt,LR), bt0, bt1)
                 in bt::(joinNeighbours btList')
                 end
           else bt0::(joinNeighbours(bt1::btList'))
        end;
```

4.5 keepJoiningNeighbours

It recursively calls joinNeighbours untill single CII is left. Hence all Tree is constructed if it is valid Inorder Traversal.

8b ⟨*keepJoiningNeighbours* 8b⟩≡ (6)

```
fun     keepJoiningNeighbours[] = raise InvalidInorderTraversal
|       keepJoiningNeighbours[bt] = bt
|       keepJoiningNeighbours btList =
                                        let val btList1 = joinNeighbours btList
                                        in keepJoiningNeighbours btList1
                                        end;
```

4.6 eraseIndices

Finally after the Tree is generated to will need to remove the extra information. and trim the tree to original tree with no extra information i.e "cosmetic sugar".

9a ⟨*eraseIndices* 9a⟩≡ (6)

```
exception Unexpected_Empty_Node
exception Unexpected_Node_Value
fun      eraseIndices Empty = raise Empty_bintree
|        eraseIndices (Node (((i:int, j:int), NONE,_), Empty, Empty)) =
                if (i=j) then Empty else raise Unexpected_Empty_Node
|        eraseIndices (Node (((i:int, j:int), x,_), LST, RST)) =
                let val left = eraseIndices LST
                val right = eraseIndices RST
                in ( case x of
                NONE => raise Unexpected_Node_Value
                | SOME y => Node (y, left, right)
                   )
                end;
```

5 Module

Now with all the code chunks now will we make a Module merging into all into one chunk

5.1 Signature

This is the Signature to be incorporated into module

9b ⟨*Signature* 9b⟩≡ (11)

```
signature BINTREE =
sig
        exception Empty_bintree
        exception InvalidInorderTraversal
        datatype 'a bintree = Empty | Node of 'a * 'a bintree * 'a bintree
        val inorder  : int bintree -> (int option * int option) list
        val inorderInverse  : ('a option * int option) list -> 'a bintree
        val test0 : int bintree
        val test1 : int bintree
        val test2 : int bintree
        val test3 : int bintree

end; (*SIG ENDS*)
```

5.2 exception

Here we define all the exceptions that we are going to raise is any invalid CASES and PATTERN are matched.

10a ⟨*Exception* 10a⟩≡ (11)

```
exception Empty_bintree
exception InvalidInorderTraversal
```

5.3 TestCases

These are all kinds of test cases

- Empty Tree
- Skewed Tree
- Complete tree
- Unbalanced Tree

10b ⟨*TestCases* 10b⟩≡ (11)

```
val test0 = Empty
val test1 = Node(1,Empty,Node(2,Empty,Node(3,Empty,Empty)))
val test2 = Node(1000,Node(999,Node(997,Empty,Empty),Node(996,Empty,Empty)),Node(998,Node(995,Empt
val test3 = Node(1,Node (2,Empty,Node (4,Empty,Empty)),Node (3,Node (5,Empty,Empty),Node (6,Node (
```

5.4 COMPLETE MODULE

This is the Complete Module which is designed for the Binary Tree.

11 ⟨*2019MCS2574-Module-complete* 11⟩≡

```
⟨Signature 9b⟩

structure Bintree : BINTREE =
struct

⟨Exception 10a⟩

⟨Datatype 2a⟩

⟨TestCases 10b⟩


local
⟨EulerTour 4a⟩
in

⟨inorder 4b⟩

end

⟨inorderInverse 6⟩
end;
open Bintree;
```