Rambling through Woods on a Sunny Morning

S. Arun-Kumar*
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
Hauz Khas, New Delhi 110 016,India
Email: sak@cse.iitd.ac.in

July 15, 2019

Abstract

In this document we explore the structural information and properties of binary trees and explore the problems of recovering binary-trees from their traversals.

This document is meant to illustrate also that programs are meant to be communicated to human beings just as much as to machines. Further, algorithms require proof and programs require both proof and testing. It illustrates also that in the functional setting program proving is only as difficult as proving algorithms correct. Unlike imperative programs which require complex mathematical, logical and notational machinery, functional programming is merely a linear representation of standard mathematical notation.

1 Introduction

> *I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be* works of literature. . . .
>
> *Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a* computer *what to do, let us concentrate rather on explaining to* human beings *what we want the computer to do.*
>
> . . .
>
> *I had the feeling that top-down and bottom-up were opposing methodologies: one more suitable for program exposition and the other more suitable for program creation. But after gaining experience with WEB, I have come to realize that there is no need to choose once and for all between top-down and bottom-up, because a program is best thought of as a web instead of a tree. A hierarchical structure is present, but the most important thing about a program is its structural relationships. A complex piece of software consists of simple parts and simple relations between those parts; the programmer's task is to state those parts and those relationships, in whatever order is best for human comprehension not in some rigidly determined order like top-down or bottom-up.*
>
> Donald E. Knuth: *Literate Programming*

This document is many things rolled into one. It is about

1. binary-trees — in health and sickness and their recovery,
2. binary-trees — their representations and traversals,
3. software engineering methodology — from elementary data structures and methods of manipulation to tying it all up together into a healthy module with the appropriate abstractions,
4. the design and analysis of data structures and attendant algorithms,
5. the analysis of structural information and its representation,

6. analysis and proofs of correctness of algorithms — e.g. the use of structural induction in proofs,
7. the *correct* implementation of *correct* algorithms[1],
8. the *functional imperative* – how far to go with a functional approach and when to turn imperative,
9. testing[2] actual implementations[3],
10. module design from scattered implementations by gift-wrapping with a self-contained interface,
11. design documents — to communicate to oneself and others about the author's thinking processes that led to the design to enable systematic debugging, trouble-shooting and future improvements,

2 The binary tree datatype representation

Consider a few distinct representations of the data-type of binary trees in ML-like languages.

2a ⟨*LeafBranch* 2a⟩≡

```
datatype 'a LBbintree =
         Leaf of 'a |
         Branch of 'a * 'a bintree * 'a bintree
```

A leaf node in such a declaration is distinct from a non-leaf node. There is the problem here that intermediate nodes of out-degree 1 cannot be easily represented. An example tree is displayed in figure 1. This lacuna may be overcome with the declaration ⟨*ZeroOneTwo* 2b⟩.

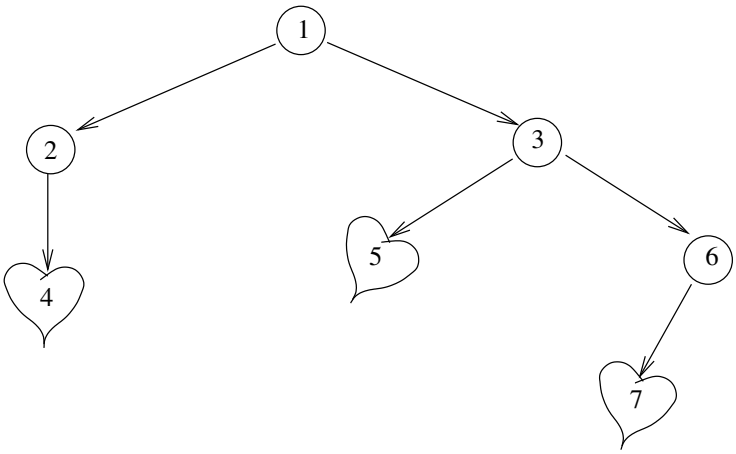

Figure 1: A binary tree with "leaf" nodes

2b ⟨*ZeroOneTwo* 2b⟩≡

```
datatype 'a bintree =
       Zero of 'a |
       One of 'a * 'a bintree |
       Two of 'a * 'a bintree * 'a bintree
```

[1] Correct algorithms never guarantee correctness of implementation

[2] Testing can seldom be exhaustive, but it can be quite exhausting! However it provides confidence in the implementation.

[3] Correct programs may not guarantee they actually work!

A leaf node in such a declaration is distinct from a non-leaf node in this case too. Further the constructor for a node of out-degree 1 is distinct here. But there is no provision for an empty tree. We may define a function called the height of a binary tree in this representation as follows

$$\begin{array}{lcl} height(\texttt{Zero}(x)) & = & 1 \\ height(\texttt{One}(x,t)) & = & 1 + height(t) \\ height(\texttt{Two}(x,t_1,t_2)) & = & 1 + max(height(t_1), height(t_2)) \end{array}$$

3a ⟨*EmptyNode2* 3a⟩≡ (5a 14b 17f 21b 37 40b)

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

On the other hand, a leaf node in the ⟨*EmptyNode2* 3a⟩ declaration 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`. The function $height$ is then defined as follows.

$$\begin{array}{lcl} height(\texttt{Empty}(x)) & = & 0 \\ height(\texttt{Node}(x,t_1,t_2)) & = & 1 + max(height(t_1), height(t_2)) \end{array}$$

Hence both representations yield the same heights for the trees in the figures 1 and 2. However we continue our discussion with the representation ⟨*EmptyNode2* 3a⟩ as it has some advantages.

- The `Empty` trees provide a kind of "zero" or identity element in the algebra of trees (analogous to the empty list in the case of list structures).
- Besides allowing for the existence of an empty tree, the presence of `Empty` permits an easy condition for recursion and/or iteration to terminate by a pattern match. It brings in a uniformity in the tree structure, since leaves are not a distinct class of elements in a tree – they are just a distinguishable type of (sub)tree.
- But the uniformity in notation can come with a cost (c.f. 4 circle nodes and 3 leaf nodes in fig. 1 versus 7 circle nodes and 8 dots in fig. 2). Assume that each node in our diagram occupies a unit cell in memory. In the case of the ⟨*ZeroOneTwo* 2b⟩ representation a complete binary tree of height h would occupy $\sum_{i=1}^{h} 2^{i-1} = 2^h - 1$ cells. However, in the case of the ⟨*EmptyNode2* 3a⟩ representation, there would actually be $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$ cells because of the extra level introduced by the `Empty` trees. However in any modern functional programming environment[4], there would be actually only one designated `Empty` cell and all copies would share it. So there would be a net increase of just one cell in this representation as compared to the ⟨*ZeroOneTwo* 2b⟩ representation (see fig. 3).

Example 2.1 *(An integer binary tree). In the binary-tree depicted in figure 2 notice that node labelled "2" has an empty left subtree and the node labelled "6" has an empty right subtree and all the leaf nodes (labelled "4", "5", "7" have both their sub-trees empty (labelled with ".").*

It is convenient to define some general-purpose functions on binary trees to prevent clutter in the code. We also include an exception for safety and technical completeness.

3b ⟨*exceptionUnexpected* 3b⟩≡ (5a)

```
exception Empty_bintree
```

3c ⟨*rootEmptyNode2* 3c⟩≡ (5a)

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

3d ⟨*leftEmptyNode2* 3d⟩≡ (5a)

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

[4] the property of *referential transparency*

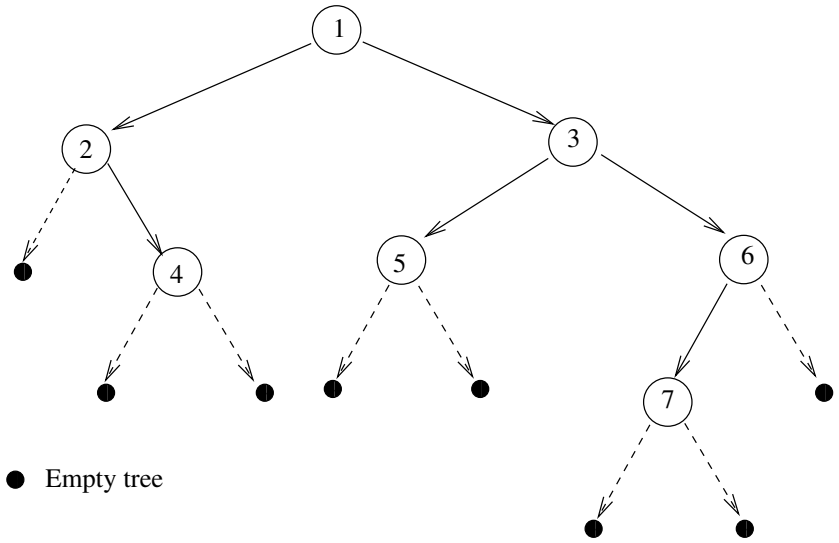

Figure 2: A binary tree with a number of `Empty` tree nodes

4a ⟨*rightEmptyNode2* 4a⟩≡ (5a)

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

We define the notion of a leaf node in the representation ⟨*EmptyNode2* 3a⟩ as

4b ⟨*isLeaf* 4b⟩≡ (5a)

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

And of course it is good to have measures such as height and size of a binary tree.

4c ⟨*height* 4c⟩≡ (5a)

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

and size is defined similarly (note that we are defining the size of `Empty` to be 0 rather than 1 because of the sharing of the lone empty cell – strictly speaking it is off from the real implementation size by one solitary cell, though this solitary cell will be shared by all other binary-trees in any single program that manipulates large numbers of binary-trees with the same representation!)

4d ⟨*size* 4d⟩≡ (5a)

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

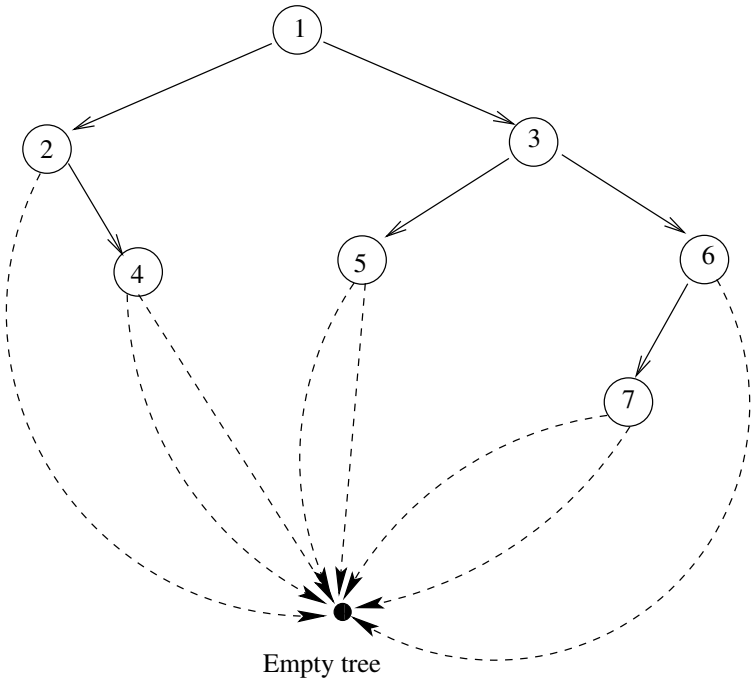

Figure 3: The same binary tree with a shared `Empty` tree node

We collect the datatype definition and the relevant deconstructor functions and some basic useful functions together into a single unit called ⟨*basicBinTreeType* 5a⟩.

5a ⟨*basicBinTreeType* 5a⟩≡ (32c 34c 36a 37)
⟨*EmptyNode2* 3a⟩
⟨*exceptionUnexpected* 3b⟩
⟨*rootEmptyNode2* 3c⟩
⟨*leftEmptyNode2* 3d⟩
⟨*rightEmptyNode2* 4a⟩
⟨*height* 4c⟩
⟨*size* 4d⟩
⟨*isLeaf* 4b⟩

An interesting but quite useless function on binary trees which we may define by induction on the structure of the tree is the "mirror image"[5] of a binary tree. It is easy to see that the mirror image of a tree will exchange the mirror images of the left and right subtrees of the original tree. It is like twisting each subtree of the original tree. So we define the recursive function

$$twist\ T \stackrel{df}{=} \begin{cases} T & \text{if } T = \texttt{Empty} \\ Node(x, (twist\ RST), (twist\ LST)) & \text{if } T = Node(x, LST, RST) \end{cases} \quad (1)$$

5b ⟨*twist* 5b⟩≡ (36a 37)

```
fun twist Empty = Empty
  | twist (Node (x, LST, RST)) =
    let
        val LST' = twist RST
        val RST' = twist LST
    in  Node (x, LST', RST')
    end
```

[5] Assume the tree is on the $x - y$ plane and is reflected on a mirror placed along the y-axis.

Exercise 2.1

1. *Prove that $twist$ is "self-dual" i.e. for any binary tree T,*

$$twist\ (twist\ T) = T \tag{2}$$

3 Tree traversals

We define traversals of trees conforming to the type definition ⟨*EmptyNode2* 3a⟩ in the following manner. Since traversals produce a list of node labels, we digress briefly to provide certain essential information about the list structure and operations on lists.

3.1 Lists

[6] We use [] to denote the empty list (*nil*) and the infix operator :: (in SML) to denote the *cons* operation to prepend an element to a (possibly empty) list. The two together define the polymorphic `list` datatype.

6a ⟨*listDatatype* 6a⟩≡
```
datatype 'a list = nil | :: of 'a * 'a list
```

The infix *append* function @ on lists is defined by pattern matching on the inductive structure of lists as follows.

$$[]@M \stackrel{df}{=} M \tag{3}$$
$$(h :: L)@M \stackrel{df}{=} h :: (L@M) \tag{4}$$

It is easy to show by induction on the structure of the first list that the following identities hold.

$$x :: L = [x]@L \tag{5}$$
$$x :: (L@M) = (x :: L)@M \tag{6}$$
$$(L@M)@N = L@(M@N) \tag{7}$$

Notice that while :: is a constant time operation, @ is linear in the length of the first list L to which is appended the second list M. In general, therefore it makes sense in certain situations, to transform certain algorithms which use @ into "tail-recursive" or "iterative" ones which use :: instead. One particularly interesting example is that of reversing a list of elements which may be defined by induction on the structure of lists as follows.

$$reverse\ L \stackrel{df}{=} \begin{cases} [] & \text{if } L = [] \\ (reverse\ L')@[x] & \text{if } L = x :: L' \end{cases}$$

Since it is defined by induction on the structure of lists, it could be directly coded in ML as

6b ⟨*reverse0* 6b⟩≡
```
fun reverse0 [] = []
  | reverse0 (x::L') = (reverse0 L')@[x]
```

[6] A little known fact among imperative programmers (who obsess about whether lists should be singly-linked or doubly-linked and where the pointers point etc.), is that lists in functional languages are also actually binary trees.

However due to the presence of the @ operation the function ⟨*reverse0* 6b⟩ will actually take $O(n^2)$ *cons* operations and is usually not acceptable. The usual trick is to convert it into a tail recursive form using an auxiliary function

$$rev\ L \stackrel{df}{=} revIter\ (L, [])$$

where

$$revIter\ (L, M) \quad = \quad \begin{cases} M & \text{if } L = [] \\ revIter(L', (x :: M)) & \text{if } L = x :: L' \end{cases}$$

Notice that rev calls $revIter$ which is linear in the length of the list. The function rev may be coded in SML by localising the auxiliary function $revIter$.

7 ⟨*reverse1* 7⟩≡

```
local
        fun revIter ([], M) = M
          | revIter (x::L', M) = revIter (L', (x::M))
in
  fun rev L = revIter (L, [])
end
```

The function `rev` is directly available in the `List` structure of most functional programming languages.

rev enjoys the following algebraic identities.

$$
\begin{aligned}
rev\ L &= reverse\ L && (8)\\
rev\ (rev\ L) &= L && (9)\\
rev\ (x :: L) &= (rev\ L)@[x] && (10)\\
rev\ (L@M) &= (rev\ M)@(rev\ L) && (11)
\end{aligned}
$$

The identity (9) shows that rev is "self-dual".

Exercise 3.1

1. *Prove the identities (5), (6) and (7).*
2. *Prove the identities (8), (9), (10) and (11).*

3.2 Preorder, Inorder and Postorder traversals

Tree traversals are taught in Data Structures courses as examples of recursive definitions by induction on structure. However, the main application of tree traversals is in compiling and generating code as a linear sequence of instructions. Binary trees are particularly useful for representing the abstract syntax of arithmetic expressions (which may contain constants, unary and binary operators).

Definition 3.1 *Given a binary tree T as defined in ⟨EmptyNode2 3a⟩, the following functions define the various traversals*

Preorder

$$
preorder\ T \stackrel{df}{=} \begin{cases} [] & if\ T = \texttt{Empty} \\ x :: (Lpre@Rpre) & if\ T = \texttt{Node(x, LST, RST)} \end{cases} \quad (12)
$$

where $Lpre \stackrel{df}{=} preorder(LST)$ and $Rpre \stackrel{df}{=} preorder(RST)$.

Inorder

$$
inorder\ T \stackrel{df}{=} \begin{cases} [] & if\ T = \texttt{Empty} \\ x :: (Lin@Rin) & if\ T = \texttt{Node(x, LST, RST)} \end{cases} \quad (13)
$$

where $Lin \stackrel{df}{=} inorder(LST)$ and $Rin \stackrel{df}{=} inorder(RST)$.

Postorder

$$
postorder\ T \stackrel{df}{=} \begin{cases} [] & if\ T = \texttt{Empty} \\ Lpost@Rpost@[x] & if\ T = \texttt{Node(x, LST, RST)} \end{cases} \quad (14)
$$

where $Lpost \stackrel{df}{=} postorder(LST)$ and $Rpost \stackrel{df}{=} postorder(RST)$.

One could directly translate the above inductive definitions into ML-syntax as follows.

8a ⟨*preorder0* 8a⟩≡

```
fun preorder0 Empty = []
  | preorder0 (Node(x, LST, RST)) =
              x::(preorder0 LST) @ (preorder0 RST)
```

8b ⟨*inorder0* 8b⟩≡

```
fun inorder0 Empty = []
  | inorder0 (Node(x, LST, RST)) =
        (inorder0 LST) @ [x] @ (inorder0 RST)
```

9a ⟨*postorder0* 9a⟩≡
```
fun postorder0 Empty = nil
  | postorder1 (Node (x, LST, RST)) =
       (postorder0 LST) @ (postorder0 RST) @ [x]
```

However the above functions may be made more run-time efficient by replacing the append operations on lists by tail-recursive algorithms which utilize :: in place of @ in some places (there are also places where such a replacement produces no change in the run-time efficiency of the function).

9b ⟨*preorder* 9b⟩≡
```
local fun pre (Empty, Llist)  = Llist
        | pre (Node (N, LST, RST), Llist) =
          let val Mlist = pre (RST, Llist)
              val Nlist = pre (LST, Mlist)
          in  N::Nlist
          end
in fun preorder T = pre (T, [])
end
```

Inorder traversal is shown below.

9c ⟨*inorder* 9c⟩≡
```
local fun ino (Empty, Llist) = Llist
        | ino (Node (N, LST, RST), Llist) =
          let val Mlist = ino (RST, Llist)
              val Nlist = ino (LST, N::Mlist)
          in  Nlist
          end
in fun inorder T = ino (T, [])
end
```

Finally post-order traversal is given by

9d ⟨*postorder* 9d⟩≡
```
local fun post (Empty, Llist) = Llist
        | post (Node (N, LST, RST), Llist) =
          let val Mlist = post (RST, N::Llist)
              val Nlist = post (LST, Mlist)
          in  Nlist
          end
in fun postorder T = post (T, [])
end
```

A little-known interesting fact is the following.

Fact 3.2

rev (postorder T) = preorder (twist T)

Proof: By induction on the structure of trees.

Basis. When T is empty all its traversals are given by $[\texttt{Empty}]$ and neither rev not $twist$ change it in any way.

Induction step. Assume $T = \texttt{Node}(x, LST, RST)$ and $Lpost = postorder(LST)$ and $Rpost = postorder(RST)$. We then have

$$\begin{array}{lcl} & & rev(postorder T) \\ \{By\ def(3.1)\} & = & rev\ (Lpost@Rpost@[x]) \\ \{(10)\} & = & x :: (rev\ Rpost)@(rev\ Lpost) \\ \{Induction\ hypothesis\} & = & x :: (preorder\ (twist\ RST))@(preorder\ (twist\ LST)) \\ \{By\ def.\ (1)\} & = & preorder\ (twist\ (\texttt{Node}(x, LST, RST))) \\ & = & preorder(twist\ T) \end{array}$$

QED

Exercise 3.2

1. *Prove that for all binary trees T*
   - `preorder0 T = preorder T`
   - `inorder0 T = inorder T`
   - `postorder0 T = postorder T`
2. *Show that* ⟨*preorder* 9b⟩, ⟨*inorder* 9c⟩ *and* ⟨*postorder* 9d⟩ *are more efficient in terms of list operations than* ⟨*preorder0* 8a⟩, ⟨*inorder0* 8b⟩ *and* ⟨*postorder0* 9a⟩ *respectively.*

We will use the example binary tree (shown in figure 2) to test our functions. In SML we may read in the tree using the code ⟨*bintreeDat* 10⟩.

10 ⟨*bintreeDat* 10⟩≡

```
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);
```

The three traversals of the above tree yield

```
- val P = preorder t1;
val P = [1,2,4,3,5,6,7] : int list
- val I = inorder t1;
val I = [2,4,1,5,3,7,6] : int list
- val T = postorder t1;
val T = [4,2,5,7,6,3,1] : int list
-
```

which are clearly permutations of each other and the first element in the preorder traversal is the root of the tree.

3.3 Euler Tours

We could have actually combined the three functions into a single function called an Euler traversal (see the book by Tomassia and Goodrich). The Euler Tour around a binary tree goes as follows:

Starting from the root make a tour around the whole tree in a counter-clockwise fashion so that the nodes being visited are always on the left as you walk around the tree visiting each node.

In this tour every (non-`Empty`) node is the root of some subtree and is visited thrice (see figure 4) –

- the first time before the "tour" of its left subtree is begun,
- the second time *after* its entire left-subtree has been "toured" *before* the tour of its right-subtree begins and
- the third time after its right subtree has been toured.

The three traversals discussed earlier are then obtained by simply filtering out the first, second and third visits of each node respectively. That is,

- the *preorder* traversal is the projection of the Euler tour containing only the first visit of each node.
- the *inorder* traversal is the projection of the Euler tour containing only the second visit of each node.
- the *postorder* traversal is the projection of the Euler tour containing only the third visit of each node.

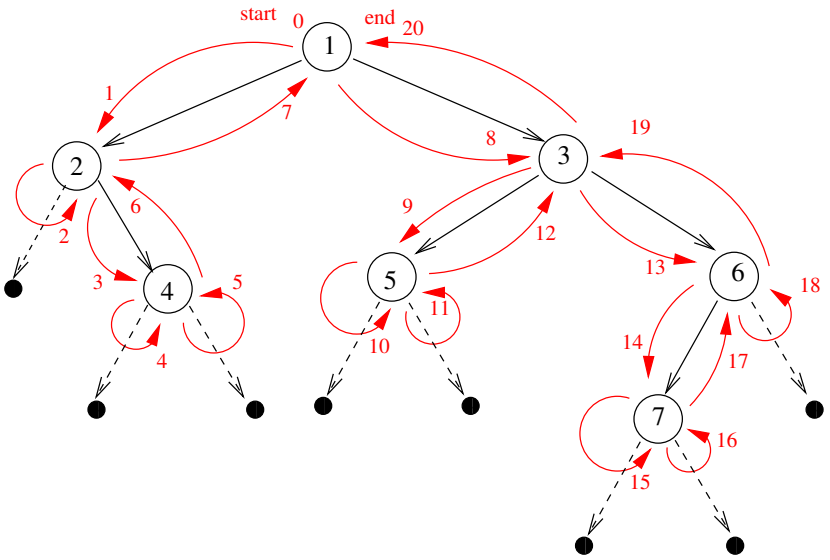

Figure 4: Euler tour of the binary tree

The following program computes the Euler tour of a binary tree by marking each visit using the following data-type.

11a ⟨*pre-in-post* 11a⟩≡
```
datatype 'a traversal = PRE of 'a | IN of 'a | POST of 'a
```

We compute the Euler tour as follows.

11b ⟨*euler0* 11b⟩≡
```
fun eulerTour0 (Empty) = nil
  | eulerTour0  (Node (x, left, right)) =
    (PRE x)::((eulerTour0 left)@(IN x)::((eulerTour0 right)@[POST x]))
```

Rather than compute the Euler tour as a single list with "tags" "PRE", "IN" and "POST", we could actually return three lists, one for each traversal.

12a ⟨*euler1* 12a⟩≡
```
fun eulerTour1 (Empty) = (nil, nil, nil)
  | eulerTour1 (Node (x, left, right)) =
        let
                val (preL, inL, postL) = eulerTour1 left
                val (preR, inR, postR) = eulerTour1 right
        in (x::preL@preR, inL@(x::inR), postL@postR@[x])
        end
fun preorder T = #1 (eulerTour1 T)
fun inorder T = #2  (eulerTour1 T)
fun postorder T = #3  (eulerTour1 T)
```

Continuing our previous session we directly use ⟨*euler1* 12a⟩ to get

```
- use "euler1.sml";
[opening euler1.sml]
val eulerTour1 = fn : 'a bintree -> 'a list * 'a list * 'a list
val it = () : unit
- eulerTour1 t1;
val it = ([1,2,4,3,5,6,7],[2,4,1,5,3,7,6],[4,2,5,7,6,3,1])
  : int list * int list * int list
-
```

4 Reconstructing the binary tree

Let `P` and `I` denote valid preorder and inorder traversals resp. of a binary tree. For simplicity we assume that every node of the tree has a distinct label which implies that there are no duplicate elements in the lists produced by any traversal. Further the two lists are permutations of each other and hence have the same length. It is easy to see that the root node would be the first element in the list `P`. We may then find the occurrence of the root node in `I`

Given two non-empty lists (which are permutations of each other and hence have the same length) we may reconstruct the tree by the following algorithm

12b ⟨*recover1* 12b⟩≡ (14b)
```
fun recover ([], []) = Empty
  | recover ((P as r::P'), I) =
    let (* P and I are permutations of each other *)
        val (LP, LI, RP, RI) = partitions (P', I, r)
        val LST = recover (LP, LI)
        val RST = recover (RP, RI)
    in  Node (r, LST, RST)
    end (* let *)
```

where `partitions (P', I, r)` works as follows.

1. Since `r` is the root of the tree, it would occur somewhere in the list `I`. So we `split` the list `I` around the occurrence of `r` into two lists `LI` and `RI` so that `I = LI@(r::RI)`. `LI` would be the inorder traversal of the left subtree in the original tree and `RI` would be the inorder traversal of the right subtree of the original tree.
2. Use `LI` and `RI` to partition `P'` into `LP` and `RP` such that the following properties hold.
   - `P'` is a permutation of `LI@RI`,
   - `LP` is a permutation of `LI`,
   - `RP` is a permutation of `RI`,
   - The relative order of elements in `P'` needs to be preserved in `LP` and `RP` respectively, i.e.
     - For any two elements a and b in `LI`, a precedes b in `LP` if and only if a precedes b in `P'` and
     - For any two elements a and b in `RI`, a precedes b in `RP` if and only if a precedes b in `P'`.

The above also constitutes a proof of the following theorem.

Theorem 4.1 *(Correctness) If* ⟨*partitions* 14a⟩ *satisfies the conditions 1 and 2 above then* ⟨*recover1* 12b⟩ *will reconstruct the binary-tree whose preorder and inorder traversals are given by* `P` *and* `I` *respectively.*

It is easier to consider partitions to be made up of two functions

13a ⟨*split* 13a⟩≡ (14b)
```
exception EmptyList
fun split ([], r) = raise EmptyList
  | split (I', r) =
    let
        ⟨splitIter 13b⟩
    in splitIter (I', r, [])
    end
```

where

13b ⟨*splitIter* 13b⟩≡ (13a)
```
fun splitIter ([], r, firsts) = (rev firsts, [])
  | splitIter (h::t, r, firsts) =
    if h = r  then (rev firsts, t)
    else splitIter (t, r, h::firsts)
```

to yield the lists `LI` and `RI`. We use these to separate `P'` into `LP` and `RP`. We assume the invariant property that

- at any stage of the computation `P''@LP@RP` is a permutation of `LI@RI` and
- `LP` is a permutation of some sub-list of `LI` and
- `RP` is a permutation of some sub-list of `RI`

13c ⟨*dissemble* 13c⟩≡ (14a)
```
fun dissemble ([], LI, RI, LP, RP) = (LP, RP):''a list * ''a list
  | dissemble ((h::t), LI, RI, LP, RP) =
    if  is_in (h, LI)
    then dissemble (t, LI, RI, (h::LP), RP)
    else (* it must be in RI *)
        dissemble (t, LI, RI, LP, (h::RP))
```

The function `is_in` is a general purpose function on lists that checks for membership of an element in a list.

13d ⟨*isIn* 13d⟩≡ (14b)
```
fun  is_in (x, L) = List.exists (fn y => x=y) L
```

Notice that the lists `LP` and `RP` so obtained need to be reversed in order to get the elements in the correct order. We may now define the function `partitions (P', I, r)` as follows

14a ⟨*partitions* 14a⟩≡ (14b)

```
local
    ⟨dissemble 13c⟩
in fun partitions (P', I, r) =
  let
      val (LI, RI) = split (I, r)
      val (LP, RP) = dissemble (P', LI, RI, [], [])
  in (rev LP, LI, rev RP, RI)
  end (* let *)
end (* local *)
```

We reorganize the final program so that general functions like ⟨*split* 13a⟩ and ⟨*isIn* 13d⟩ are declared global to the programs. The final program we obtain then reads as follows.

14b ⟨*recoverPI-complete* 14b⟩≡

```
⟨isIn 13d⟩
⟨split 13a⟩
⟨EmptyNode2 3a⟩
local
    ⟨partitions 14a⟩
in
⟨recover1 12b⟩
end (* local *)
```

A complete session which computes the traversals and the recovery on our example binary tree (2.1) is shown below. Some of the obvious responses from the system have been elided and replaced with ellipsis (...) in order to keep the text small and self-contained.

```
... denotes some system responses that have been elided
    denotes relevant input to the program
    denote the desirable output
    denotes undesirable output
```

The highlighting that you see in the various sessions have been colour coded as above for all the sessions in this document.

```
Standard ML of New Jersey v110.78 [built: Thu Jul 23 11:21:58 2015]
- use "recoverPI.sml";
recoverPI.sml:8.18 Warning: calling polyEqual
recoverPI.sml:1.44 Warning: calling polyEqual
val is_in = fn : ''a * ''a list -> bool
exception EmptyList
val split = fn : ''a list * ''a -> ''a list * ''a list
...
val recover = fn : ''a list * ''a list -> ''a bintree
val it = () : unit
- use "bintreeDat.sml";
[opening bintreeDat.sml]
...
val t1 = Node (1,Node (2,Empty,Node ... )) : int bintree
val it = () : unit
- use "euler1.sml";
[opening euler1.sml]
val eulerTour1 = fn : 'a bintree -> 'a list * 'a list * 'a list
val it = () : unit
- val (P,I,S) = eulerTour1 t1;
val P = [1,2,4,3,5,6,7] :  int list
val I = [2,4,1,5,3,7,6] :  int list
val S = [4,2,5,7,6,3,1] :  int list
- val t1' = recover (P, I);
val t1' = Node (1,Node (2,Empty,Node ...)) : int bintree
- t1 = t1';
val it = true :  bool
-
```

which shows that the reconstructed tree `t1'` equals the original tree `t1`.

In our example, for simplicity we assumed that the node labels are all distinct, which in general need not be the case. In fact, the presence of more than one node having the same label does make the problem of reconstruction slightly harder, but as we shall see that is more due to the lack of structural information than merely replication of values.

We leave it as an exercise for interested readers to figure out algorithms which will recontruct the binary tree

- only from the preorder and postorder traversals and
- only from the inorder and postorder traversals.

5 Positional information

In the case of each of these traversals there is a loss of structural information in each traversal which does not permit the reconstruction of the binary-tree from a single traversal. The kinds of information in a binary tree are essentially of the following kinds.

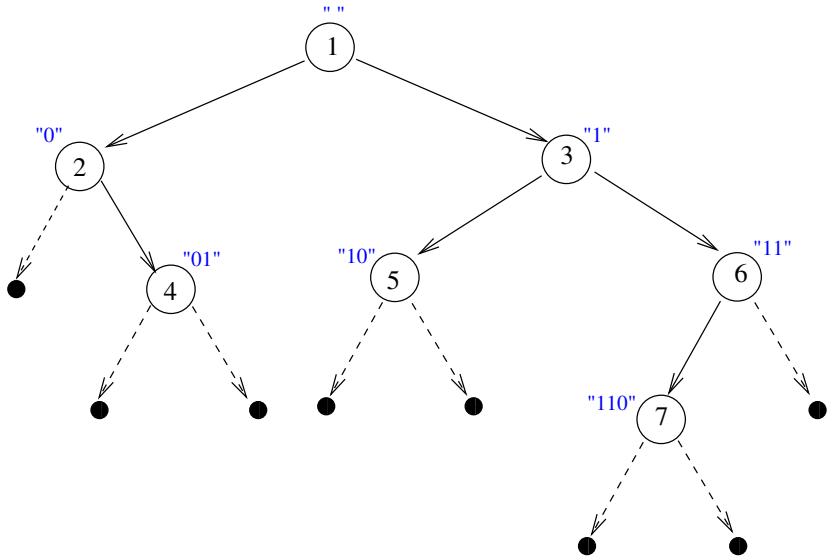

Figure 5: Encoding positions as bit-string values

- *value* information, which are labels on the individual nodes and
- *structural* information, which
  - either gives the *position* of each node,
  - or alternatively, the relative (precedence-) relationships between the nodes of the tree.

Given a binary tree, the position of each node may be represented as a bit-string *value* (see figure 5 for example) with the position of the root node represented by the empty string and each (non-`Empty`) child node having a position represented by extending the "position" of the parent by a "0" if it is left-child and a "1" if it is the right-child.

The bit-string values which encode the positions are actually absolute values and may be regarded as an encoding of structural information by values. The positional information actually gives the following information.

1. The string representing the position of a node gives "directions" about the path to follow from the root in order to reach the node.
2. If the height of the binary-tree is h, then there must be at least one node at each level l, $0 \leq l \leq h$.
3. The length of the string gives the "level" at which the node is located.
4. Since we are limiting ourselves to binary-trees, at each level l there can be at most 2^l distinct nodes.
5. For any bit-string s let $|s|$ denote the length of s, $\prec$ denote the prefix ordering on strings and $(s)_2$ denote the value of the bit-string as a number written in binary.

   The list of positions as bit-strings gives an implicit "precedence" ordering $<_{prec}$ on the positions of the nodes. If $n_1 = (v_1, s_1)$ and $n_2 = (v_2, s_2)$ are nodes represented by their labels and positions then $n_1 <_{prec} n_2$ iff

   - either n_1 is at a higher level than n_2 i.e. $|s_1| < |s_2|$. In particular, if $s_1 \prec s_2$ then additionally it says that n_1 is strictly an ancestor of n_2,
   - or they are at the same level but n_1 is to the left of n_2 i.e. $|s_1| = |s_2|$ and $(s_1)_2 < (s_2)_2$.

   That is, $n_1 <_{prec} n_2$ iff $(|s_1|, (s_1)_2) <_{lex} (|s_2|, (s_2)_2)$ where $<_{lex}$ is the lexicographic ordering on ordered pairs of numbers[7].

[7]Notice that $s_1 \prec s_2$ implies $(|s_1|, (s_1)_2) <_{lex} (|s_2|, (s_2)_2)$.

The following higher-order function (analogous to the map function on lists) may be used to include the positional values to the tree.

17a ⟨*bt-map* 17a⟩≡ (37)
```
fun BTmap F =
    let fun BTM Empty = Empty
          | BTM (Node (x, left, right)) = Node ((F x), BTM (left), BTM (right))
    in  BTM
    end
```

For any binary tree `T` we have

17b ⟨*bt-map-01* 17b⟩≡
```
fun BTmap0 T = BTmap (prefix "0") T
fun BTmap1 T = BTmap (prefix "1") T
```

where

17c ⟨*prefix* 17c⟩≡ (17f 21b)
```
fun prefix s (x, p) = (x, s^p)
fun prefix0 (x, p) = prefix "0" (x,p)
fun prefix1 (x, p)= prefix "1" (x,p)
```

Note that `BTmap0` and `BTmap1` require the argument `T`.

The function ⟨*bt-pos* 17d⟩ may then be employed to decorate the nodes with their positions too.

17d ⟨*bt-pos* 17d⟩≡
```
fun btpos Empty = Empty
  | btpos (Node (x, L, R)) =
          let val posL = btpos L
              val posR = btpos R
          in  Node ((x,""), (BTmap0 posL), (BTmap1 posR))
          end
```

We could have modified the Euler tour to one which gives the positional information along with node labels as follows. We then use the `List.map` function instead of `BTmap`.

17e ⟨*euler-positions-bare* 17e⟩≡ (17f)
```
fun eulerTour2 Empty = (nil, nil, nil)
  | eulerTour2 (Node (x, left, right)) =
    let
        val (preL, inL, postL) = eulerTour2 left
        val (preR, inR, postR) = eulerTour2 right
    in ((x, ""):::((map prefix0 preL)@(map prefix1 preR)),
        (map prefix0 inL)@((x,""):::(map prefix1 inR)),
        (map prefix0 postL)@(map prefix1 postR)@[(x,"")]
       )
    end
```

In this case (unlike the previous cases) we could allow multiple nodes with the same label, since the positional information is absolute and recovery of the original tree depends entirely upon the positional information.

A complete program which may be executed in a stand-alone fashion is given below.

17f ⟨*euler-positions-complete* 17f⟩≡
```
⟨EmptyNode2 3a⟩
⟨prefix 17c⟩
⟨euler-positions-bare 17e⟩
```

An SML session on our example tree then looks as follows.

```
Standard ML of New Jersey v110.78 [built: Thu Jul 23 11:21:58 2015]
- [opening ... euler-positions-complete.sml]
...
val prefix = fn : string -> 'a * string -> 'a * string
val prefix0 = fn : 'a * string -> 'a * string
val prefix1 = fn : 'a * string -> 'a * string
val eulerTour2 = fn
  : 'a bintree -> ('a * string) list * ('a * string) list * ('a * string) list
val it = () : unit
-
[opening ... bintreeDat.sml]
...
val t1 = Node (1,Node (2,Empty,Node ...)) : int bintree
val it = () : unit
- eulerTour2 t1;
val it =
([(1,""),(2,"0"),(4,"01"),(3,"1"),(5,"10"),(6,"11"),(7,"110")],
[(2,"0"),(4,"01"),(1,""),(5,"10"),(3,"1"),(7,"110"),(6,"11")],
[(4,"01"),(2,"0"),(5,"10"),(7,"110"),(6,"11"),(3,"1"),(1,"")])
 : (int * string) list * (int * string) list * (int * string) list
-
```

For any list (which is some permutation of node-position pairs that make up a binary tree), it is possible to recover the original tree from this list, by the following algorithm 5.1, which takes a list of pairs of node labels and positions and returns a binary tree whenever possible. Since this function takes an arbitrary list as input, we need to also make sure our definitions are not just "technically complete", they can also give us a unique binary tree. Otherwise suitable exceptions need to be raised.

- `makeBintree: ('a * string) list -> 'a bintree`
  which the type inference engine will ensure or point out as error. The other exceptions which require explicit programming for "technical completeness" are the following.
- There is exactly one root node with position `""`.
- There are no duplicate positions occurring anywhere in the list. This is checked recursively in the calls to ⟨*partition-into-3* 19b⟩ from ⟨*makeBintree* 19d⟩.
- There are no missing levels in the tree. For instance, the length of the position string determines the depth of a node in the tree.
- There are no redundant nodes.

Algorithm 5.1 (Make Binary tree) Partitioning. *Partition the list into three parts*

1. `rt`*: a unique root node with the empty string as its position. For reasons of technical completeness this is also a list. In* ⟨*makeBintree* 19d⟩ *the presence of multiple root-nodes at the level of each sub-tree is also checked and raised.*
2. `nl0`*: a (possibly empty) list of nodes whose position strings begin with "0",*
3. `nl1`*: a (possibly empty) list of nodes whose position strings begin with "1".*

See ⟨*partition-into-3* 19b⟩.

Strip head. *Strip the first character in the position of each element in* `nl0` *and* `nl1` *of , yielding* `nl0'` *and* `nl1'`. *See* ⟨*strip1* 19c⟩.

Recursive construction. *Construct the left and right subtrees recursively and then make the binary tree from the root node (see* ⟨*makeBintree* 19d⟩*).*

which gives us the following code

19a ⟨*makebintree-frompos-complete* 19a⟩≡
⟨*partition-into-3* 19b⟩
⟨*strip1* 19c⟩
⟨*makeBintree* 19d⟩

where

19b ⟨*partition-into-3* 19b⟩≡ (19a)
```
exception invalid_position_string of string
fun partInto3 [] (rt, nl0, nl1) = (rt, nl0, nl1)
  | partInto3 ((a,s)::nl)  (rt, nl0, nl1) =
                if s = "" then partInto3 nl ((a,s)::rt, nl0, nl1)
                else if (String.isPrefix "0" s) then partInto3 nl (rt, (a,s)::nl0, n
                else if (String.isPrefix "1" s) then partInto3 nl (rt, nl0, (a,s)::n
                else raise invalid_position_string (s)
```

19c ⟨*strip1* 19c⟩≡ (19a)
```
local
    exception striphd_empty
    fun striphd s =
      let val l = String.size s
      in if l = 0 then raise striphd_empty
         else substring (s, 1, l-1)
      end
in
fun strip1 (a,b) = (a, (striphd b))
end
```

19d ⟨*makeBintree* 19d⟩≡ (19a)
```
exception invalid_nodelist of string
exception No_root
exception Multiple_roots
fun makeBintree [] = Empty
  | makeBintree [(a,"")] = Node(a, Empty, Empty)
  | makeBintree [(_, _)] = raise invalid_nodelist ("ERROR: Expected empty string in
  | makeBintree nl = (* nl has at least two elements *)
    let
        val (rt, nl0, nl1) = partInto3 nl ([], [], [])
        val valrt = if List.null (rt) then raise No_root
                    else if (List.length rt) = 1 then (#1 (hd rt))
                    else raise Multiple_roots
        val nl0' = List.map strip1 nl0
        val nl1' = List.map strip1 nl1
        val t0 = makeBintree nl0'
        val t1 = makeBintree nl1'
    in
        Node (valrt, t0, t1)
    end (* let *)
```

Notice that the above algorithm is independent of any particular traversal. Any permutation of a valid traversal of the binary tree would yield the tree. The following SML seession illustrates this (see especially the last 10 lines where the list `Post` has been permuted to yield `Post'`).

```
Standard ML of New Jersey v110.78 [built: Thu Jul 23 11:21:58 2015]
- [opening ... euler-positions-complete.sml]
...
val prefix = fn : string -> 'a * string -> 'a * string
val prefix0 = fn : 'a * string -> 'a * string
val prefix1 = fn : 'a * string -> 'a * string
val eulerTour2 = fn
  : 'a bintree -> ('a * string) list * ('a * string) list * ('a * string) list
val it = () : unit
-
[opening ... bintreeDat.sml]
...
val t1 = Node (1,Node (2,Empty,Node ...)) : int bintree
val it = () : unit
- val (Pre, In, Post) = eulerTour2 t1;
val Pre = [(1,""),(2,"0"),(4,"01"),(3,"1"),(5,"10"),(6,"11"),(7,"110")]
  : (int * string) list
val In = [(2,"0"),(4,"01"),(1,""),(5,"10"),(3,"1"),(7,"110"),(6,"11")]
  : (int * string) list
val Post = [(4,"01"),(2,"0"),(5,"10"),(7,"110"),(6,"11"),(3,"1"),(1,"")]
  : (int * string) list
-
[opening ... makebintree-frompos-complete.sml]
[autoloading]
[library \$SMLNJ-BASIS/basis.cm is stable]
[autoloading done]
...
val makeBintree = fn : ('a * string) list -> 'a bintree
val it = () : unit
- val t1Pre = makeBintree Pre;
val t1Pre = Node (1,Node (2,Empty,Node ...))
  : int bintree
- t1Pre = t1;
val it = true : bool
- val t1In = makeBintree In;
val t1In = Node (1,Node (2,Empty,Node ...)) : int bintree
- t1In = t1; \colorbox{green}{val it = true : bool}\colorbox{yellow}{- val t1Post = ma
t1Post = Node (1,Node (2,Empty,Node \colorbox{blue!50}{...})):  int bintree\colorbox{y
val it = true : bool
- val Post' =  [(5,"10"),(7,"110"),(3,"1"),(1,""),(2,"0"),(6,"11"),(4,"01")];
val Post' = [(5,"10"),(7,"110"),(3,"1"),(1,""),(2,"0"),(6,"11"),(4,"01")]
  : (int * string) list
- val t1Post' = Post';
val t1Post' = [(5,"10"),(7,"110"),(3,"1"),(1,""),(2,"0"),(6,"11"),(4,"01")]
  : (int * string) list
- val t1Post' = makeBintree Post';
val t1Post' = Node (1,Node (2,Empty,Node ...))
  : int bintree
- t1Post' = t1;
val it = true : bool
-
```

6 Stuctural Information

In sections 4 and 5 we have seen how to reconstruct a binary tree given either two distinct traversals or absolute positional information for each node. Both of these contain large amounts of extra information which may not actually be needed. Instead we could use the information about the presence of empty trees while performing the traversal and use it appropriately. The loss of structural information may be remedied by the introduction of some markers for the leaf nodes during the traversal.

For this purpose Standard ML provides a convenient polymorphic data type called the option type whose signature is shown below.

```
datatype 'a option = NONE | SOME of 'a
exception Option
val getOpt : 'a option * 'a -> 'a
val isSome : 'a option -> bool
val valOf : 'a option -> 'a
val filter : ('a -> bool) -> 'a -> 'a option
val join : 'a option option -> 'a option
val app : ('a -> unit) -> 'a option -> unit
val map : ('a -> 'b) -> 'a option -> 'b option
val mapPartial : ('a -> 'b option) -> 'a option -> 'b option
val compose : ('a -> 'c) * ('b -> 'a option) -> 'b -> 'c option
val composePartial : ('a -> 'c option) * ('b -> 'a option)
                     -> 'b -> 'c option
```

It consists of a datatype declaration which for any type `'a` tags each element of type `'a` with a `SOME` constructor and adds a new constructor called `NONE` to signify elements which do not belong to type `'a`. This may be used for raising exceptions and handling them in contexts far removed from where the exception occurred and thus provides a much cleaner solution to the problem of propagating exceptions to scopes well outside the context in which they are raised.

6.1 Encoding emptiness using option datatype

But we could use `NONE` to mark the positions in the traversal which are *leaf* nodes (both of whose children are `Empty`) and nodes of out-degree 1 (one of whose children is `Empty`). Our algorithm for the Euler Tour (⟨*euler1* 12a⟩) then gets modified as follows.

21a ⟨*euler3* 21a⟩≡ (21b 32c 34c 36a 37)

```
fun eulerTour3 (Empty) = ([NONE], [NONE], [NONE])
  | eulerTour3 (Node (x, left, right)) =
        let
                val (preL, inL, postL) = eulerTour3 left
                val (preR, inR, postR) = eulerTour3 right
        in ((SOME x)::preL@preR, inL@((SOME x)::inR), postL@postR@[(SOME x)])
        end
fun preorder bt = #1 (eulerTour3 bt)
fun inorder bt = #2 (eulerTour3 bt)
fun postorder bt = #3 (eulerTour3 bt)
```

A complete program which may be executed in a stand-alone fashion is the given below.

21b ⟨*euler-structural-complete* 21b⟩≡

```
⟨EmptyNode2 3a⟩
⟨prefix 17c⟩
⟨euler3 21a⟩
```

Here is an SML session on our familiar example.

```
Standard ML of New Jersey v110.78 [built: Thu Jul 23 11:21:58 2015]
- use "EmptyNode2.sml";
[opening EmptyNode2.sml]
...
- use "bintreeDat.sml";
[opening bintreeDat.sml]
val t2 = Node (2,Empty,Node (4,Empty,Empty)) : int bintree
val t1 = Node (1,Node (2,Empty,Node ...)) : int bintree
val it = () : unit
- use "euler3.sml";
[opening euler3.sml]
val eulerTour3 = fn
  : 'a bintree -> 'a option list * 'a option list * 'a option list
val it = () : unit
- eulerTour3 t1;
val it =
  ([SOME 1,SOME 2,NONE,SOME 4,NONE,NONE,SOME 3,SOME 5,NONE,NONE,SOME 6,SOME 7,
    ...],
   [NONE,SOME 2,NONE,SOME 4,NONE,SOME 1,NONE,SOME 5,NONE,SOME 3,NONE,SOME 7,
    ...],
   [NONE,NONE,NONE,SOME 4,SOME 2,NONE,NONE,SOME 5,NONE,NONE,SOME 7,NONE,...])
  : int option list * int option list * int option list
-
```

6.2 Reconstruction from a Preorder traversal

Now we need to tackle the problem of reconstructing the tree from a single traversal. We initially choose only the preorder traversal. An initial attempt at reconstructing the tree from the preorder traversal goes as follows.

22 ⟨*makeTreeFromP0* 22⟩≡

```
exception InvalidPreorderTraversal
fun makeTreeFromP0 [] = raise  InvalidPreorderTraversal
  | makeTreeFromP0 [NONE] = Empty
  | makeTreeFromP0 [_,_] = raise  InvalidPreorderTraversal
  | makeTreeFromP0 [SOME x, NONE, NONE] = Node (x, Empty, Empty)
  | makeTreeFromP0 [_,_,_] =  raise  InvalidPreorderTraversal
  | makeTreeFromP0 (h::t) =
    ( case h of
          NONE => raise  InvalidPreorderTraversal
        | SOME x =>
          let
              val (LST, RST) = makeTwoTrees t
          in  Node (x, LST, RST)
          end
    )
```

where we need to figure out how to construct *two* trees out of the tail of the list. How does one make two trees out of a single list? Clearly the list may be split into two lists, retaining the elements in each part in the same order in which they occur in the original list. However, we don't know the correct place in the list to split the list into two which will yield exactly two trees for a valid preorder traversal. We may require a deeper analysis to solve the problem.

6.3 More analysis

For brevity, we simply use x to denote `SOME` x and $\perp$ for `NONE`. Think of the set of all elements of the type of labels as consisting of the labels with a distinguished new symbol $\perp$ added to the set.

We refer to a contiguous sub-sequence of a traversal as a *slice* of the list. More precisely,

Definition 6.1 *A list S is a* **slice** *of a list L, if there exist lists M and N such that $L = M@S@N$.*

However, not every contiguous slice defines a subtree of the original tree. Due to the loss of some structural information, it is even possible that an algorithm that we design yields a different tree than the one we started out with. The following observations about the traversals are obvious and will prove useful when we try to reconstruct the tree from any traversal.

Facts 6.2

1. *No traversal can yield an empty list. The* `Empty` *tree yields* $[\perp]$ *as the result of each traversal. Each* $\perp$ *is an* `Empty` *sub-tree.*
2. *No traversal can yield a list of length* 2*. We need at least 3 elements for the list to be a valid traversal.*
3. *Any element $m \neq \perp$ in any traversal is the root of a subtree (could possibly be a leaf node too).*
4. *The (pre/in/post)-order traversal of each subtree of a tree is a slice of the (pre/in/post)-order traversal resp. of the whole tree.*

Hence any traversal of a subtree of the original tree yields a *slice* in the traversal of the whole tree. However, not every slice in a traversal may occur as a subtree.

Facts 6.3 *Let T be a binary tree whose preorder, inorder and postorder traversals are L_{pre}, L_{in} and L_{post} respectively.*

1. *Every* leaf *node l of the tree (denoting the subtree* `Node (`l`, Empty, Empty)`*) occurs as a slice of the form*
   - $[l, \perp, \perp]$ *in the* preorder *traversal,*
   - $[\perp, l, \perp]$ *in the* inorder *traversal and*
   - $[\perp, \perp, l]$ *in the* postorder *traversal.*

   This is also the case when the original tree consists of a single node which is both the root as well as the unique leaf.
2. *Every non-$\perp$ element in any traversal is the root of a sub-tree of the original tree.*
3. *Given distinct labels $m \neq \perp \neq l$, such that* `Node (`m`, Node (`l`, Empty, Empty), Empty)` *is a subtree,*
   - $[m, l, \perp, \perp, \perp]$ *is a slice in the* preorder *traversal,*
   - $[\perp, l, \perp, m, \perp]$ *is a slice in the* inorder *traversal,*
   - $[\perp, \perp, l, \perp, m]$ *is a slice in the* postorder *traversal,*

   l is the left child of m and the right child of m is `Empty`*.*
4. *Given distinct labels $m \neq \perp \neq r$, such that* `Node (`m`, Empty, Node (`r`, Empty, Empty))` *is a subtree,*

- $[m, \bot, r, \bot, \bot]$ *is a slice in the* preorder *traversal,*
- $[\bot, m, \bot, r, \bot]$ *is a slice in the* inorder *traversal,*
- $[\bot, \bot, \bot, r, m]$ *is a slice in the* postorder *traversal,*

r *is the right child of* m *in the tree and* m *has an* `Empty` *left child,*

5. *Given distinct labels* $m, l, r \neq \bot$*, such that* `Node (`m`, Node (`l`, Empty, Empty), Node (`r`, Empty, Empty))` *is a subtree,*
   - $[m, l, \bot, \bot, r, \bot, \bot]$ *is a slice in the* preorder *traversal,*
   - $[\bot, l, \bot, m, \bot, r, \bot]$ *is a slice in the* inorder *traversal,*
   - $[\bot, \bot, l, \bot, \bot, r, m]$ *is a slice in the* postorder *traversal,*
6. *Let* `Node (`m`,` T^l`,` T^r`)` *be a (sub-)tree rooted at* $m \neq \bot$*. Further let* l, r*, where one or both of* l *and* r *could be* $\bot$*, be distinct labels such that* l *and* r *are roots of subtrees* T^l *and* T^r *respectively. If* L^l_{pre} L^l_{in}*,* L^l_{post}*,* L^r_{pre}*,* L^r_{in}*,* L^r_{post}*, are respectively the slices in the various traversals then*
   - $m :: (L^l_{pre} @ L^r_{pre})$ *is a slice in the* preorder *traversal,*
   - $L^l_{in} @ (m :: L^r_{in})$ *is a slice in the* inorder *traversal,*
   - $L^l_{post} @ L^r_{post} @ [m]$ *is a slice in the* postorder *traversal,*

   then having T^l *and* T^r *respectively as its left and right subtrees.*
7. *Any preorder, inorder or postorder traversal is a slice that represents the whole tree as a tree-slice.*

Some facts that are specific to preorder traversals are the following.

Facts 6.4 *Let* T *be a binary tree.*

1. *In any preorder traversal,* $\bot$ *is the last element in the traversal.*
2. *In any preorder traversal* $\bot$ *cannot be the first element unless the original tree is* `Empty`*.*
3. *The descendants of a node* $l \neq \bot$ *always occur to its right in any preorder traversal.*
4. *Any* m *that occurs to the right of a node* l *is either a descendant of* l *(if* $l \neq \bot$*) or a descendant of a right sibling of* l*.*
5. *If* $l \neq \bot \neq m$ *then* m *is the successor of* l *in the preorder traversal if and only if* m *is the left-child of* l *in the tree.*
6. $\bot$ *is the successor of* $l \neq \bot$ *in the preorder traversal if and only if* `Empty` *is the left-child of* l*.*
7. *Any slice of the form* $[l, \bot, \bot]$ *where* $l \neq \bot$ *determines a unique leaf node in the original tree.*
8. *Any slice of the form* $[l, m, \bot, \bot, \bot]$ *where* $l \neq \bot \neq m$ *determines a unique subtree rooted at* l *whose left child is the leaf node* m *and the right child is empty.*
9. *Any slice of the form* $[l, \bot, m, \bot, \bot]$ *where* $l \neq \bot \neq m$ *determines a unique subtree rooted at* l *whose right child is the leaf node* m *and the left child is empty.*
10. *Any slice of the form* $l :: S_0 @ S_1$ *where* $l \neq \bot$ *and* S_0 *and* S_1 *are slices that determine unique sub-trees* T_0 *and* T_1 *resp., determines a unique sub-tree rooted at* l *with left substree* T_0 *and right subtree* T_1 *resp.*

We then have the following theorem for preorder traversals.

Theorem 6.5 (Uniqueness of Preorder traversals) . *No two distinct binary trees can yield the same preorder traversal.*

Proof: The proof follows from the definition (3.1) of preorder traversal and the facts 6.4 (esp. if and only if conditions). Further, the facts 7-10 in 6.4 actually yield an inductive proof along with a case analysis of the inductive step. QED

Theorem 6.5 and in particular the facts 7-10 in 6.4 actually give us a systematic bottom-up construction of the tree from the `Empty` nodes via the construction of leaf nodes and subtrees by an inductive process which effectively answers the question raised by `makeTwoTrees` in ⟨*makeTreeFromP0* 22⟩.

Even if the reader is not convinced by the proof given for Theorem 6.5, we hope to convince the reader later through theorem 6.12 after developing some suitable technical machinery.

We denote slices in a traversal that result in subtrees by an ordered pair of indices called *closed index interval (CII)*. Given a list of length $n > 0$ representing a tree traversal[8] of node-labels and $\perp$ we then have the following observations. Let $I(n) = \{(i, j) \mid 0 \leq i \leq j < n\}$ denote the set of all non-empty closed index intervals of slices of a list L of length $n > 0$. For any valid index i, $0 \leq i < n$, L_i denotes the element in L at index i Then we have

Facts 6.6

1. *A CII of the form (i, i) can only represent an* `Empty` *(sub-)tree and that too provided $L_i = \perp$. We denote the (sub-)tree it represents by $T^{(i,i)}$.*
2. *No CII of the form $(i, i + 1)$ can represent a valid sub-tree.*
3. *Every leaf node (also subtree $T^{(i,j)}$) is represented by a CII (i, j) where $j = i + 2$, in a*
   - preorder *traversal provided $L_i \neq \perp$, $L_{i+1} = L_j = \perp$,*
   - inorder *traversal provided $L_{i+1} \neq \perp$, $L_i = L_j = \perp$, and*
   - postorder *traversal provided $L_j \neq \perp$, $L_i = L_{i+1} = \perp$,*
4. *Regardless of whether the traversal is preorder, inorder or postorder if $T^{(k,m)}$ is a subtree of $T^{(i,j)}$ then the CII (k, m) is entirely contained within the CII (i, j) i.e. $0 \leq i \leq k \leq m \leq j < n$.*
5. *In a preorder traversal, $T^{(k,m)}$ is a subtree of $T^{(i,j)}$ if and only if (k, m) is entirely contained within the CII (i, j) i.e. $0 \leq i \leq k \leq m \leq j < n$.*
6. *Given $0 \leq i \leq j < k \leq m < n$ with (i, j) and (k, m) representing valid subtrees $T^{(i,j)}$ and $T^{(k,m)}$ of the original tree respectively, (i, j) and (k, m) are* neighbours *(more precisely (i, j) is the* left neighbour *and (k, m) is the* right neighbour*) in a*
   - preorder *traversal if $k = j + 1$, $i > 0$ and $L_{i-1} \neq \perp$,*
   - inorder *traversal if $k = j + 2$ and $L_{j+1} \neq \perp$ and*
   - postorder *traversal if $k = j + 1$, $m < n - 1$ and $L_{m+1} \neq \perp$.*

   Notice that the above definition of neighbours holds even in the case of `Empty` *subtrees i.e. whenever $i = j$ and/or $k = m$.*
7. *Given $0 \leq i \leq j < k \leq m < n$ with (i, j) and (k, m) representing valid subtrees $T^{(i,j)}$ and $T^{(k,m)}$ of a preorder traversal, (i, j) and (k, m) are* neighbours *if and only if $k = j + 1$, $i > 0$ and $L_{i-1} \neq \perp$.*
8. *Given neighbours (i, j) and (k, m), representing subtrees $T^{(i,j)}$ and $T^{(k,m)}$ of the original tree respectively,*
   - preorder: *if $i > 0$ and $L_{i-1} = h \neq \perp$, then $(i-1, m)$ is the CII representing $Node(h, T^{(i,j)}, T^{(k,m)}) = T^{(i-1,m)}$.*
   - inorder: *if $L_{j+1} = h \neq \perp$, then (i, m) is the CII representing the subtree $Node(h, T^{(i,j)}, T^{(k,m)}) = T^{(i,m)}$.*
   - postorder: *if $m < n - 1$ and $L_{m+1} = h \neq \perp$, then $(i, m + 1)$ is the CII representing the subtree $Node(h, T^{(i,j)}, T^{(k,m)}) = T^{(i,m+1)}$.*
9. *$(0, n - 1)$ is the slice representing a valid tree provided all the above conditions are satisfied.*

Exercise 6.1 *There is a certain sense in which the postorder traversal of a binary tree is like the "dual" of the preorder traversal.*

[8] a putative traversal cannot yield an empty list on any traversal

1. *Translate each of the facts 6.4 into facts about postorder traversals and justify your statements.*
2. *Prove a uniqueness theorem (à la theorem 6.5) for postorder traversals.*
3. *Translate the parts 5 and 7, in facts 6.6 into dual statements that hold for postorder traversals.*

Definition 6.7 *Let*

- $\subseteq$ *denote the* nesting *relation on intervals i.e.* $(i,j) \subseteq (k,m)$ *iff* $k \leq i \leq j \leq m$,
- $\subset$ *denote the* proper nesting *relation i.e.* $(i,j) \subset (k,m)$ *iff* $(i,j) \subseteq (k,m)$ *and* $(i,j) \neq (k,m)$ *and*
- $\asymp$ *denote the* disjointness *relation i.e.* $(i,j) \asymp (k,m)$ *iff* $i \leq j < k \leq m$ *or* $k \leq m < i \leq j$.

If two CIIs are neither disjoint nor nested they are said to be strictly overlapping.

Facts 6.8

1. *The nesting relation is a partial order on* $I(n)$.
2. $(i,j) \subset (k,m)$ *iff* $(k < i \leq j \leq m)$ *or* $(k \leq i \leq j < m)$.
3. *The proper nesting relation is an irreflexive transitive relation.*
4. *Neighbours are disjoint.*
5. *The operation of joining neighbours as in fact 6.6 part 8 always yields a CII in which the neighbours are properly nested but themselves disjoint.*

Fact 6.9

1. *In any traversal of length* $n > 0$*, two CIIs* (i,j) *and* (k,m) *could represent valid subtrees only if one of the following conditions is satisfied*
   - *one of them is* nested *in the other or*
   - *they are* disjoint *i.e.*
2. *Given any two CIIs that are strictly overlapping at least one of them does not represent a subtree of the original tree.*

Lemma 6.10 *Given any traversal of a binary-tree, and three slices* (i,j), (k,m) *and* (p,q) *representing valid subtrees such that* $0 \leq i \leq j < k \leq m < p \leq q < n$*, it is impossible for* (k,m) *to be a* right *neighbour of* (i,j) *as well as a* left *neighbour of* (p,q).

Proof: Let L be any traversal of a binary-tree such that $0 \leq i \leq j < k \leq m < p \leq q < n$ and $T^{(i,j)}$, $T^{(k,m)}$ and $T^{(p,q)}$ are subtrees of the original tree. Assume (k,m) to be a *right* neighbour of (i,j) as well as a *left* neighbour of (p,q). Then we proceed by case analysis on the traversal.

Preorder $i > 0$ and $L_{i-1} \neq \perp \neq L_{k-1}$ and $i - 1 < k - 1$ and by 8 we may obtain subtrees represented by CIIs $(i-1,m)$ and $(k-1,q)$ which violate fact 6.9 (strictly overlapping).

Inorder $p = m + 2$ and $L_{m+1} \neq \perp$ and $k = j + 2$ and $L_{j+1} \neq \perp$, and by 8 we may obtain subtrees represented by CIIs (i,m) and (k,q) which violate fact 6.9 (strictly overlapping).

Postorder $m < q < n-1$ and $L_{q+1} \neq \perp \neq L_{m+1}$ yielding subtrees represented by CIIs $(i,m+1)$ and $(k,q+1)$ which violate fact 6.9 (strictly overlapping).

QED

We may note the following interesting facts (all of which may not be relevant to our algorithm).

Facts 6.11

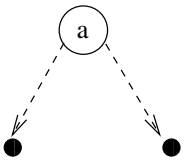

Figure 6: A binary tree with one non-`Empty` node

1. *There is no tree whose preorder traversal is the empty list.*
2. *The* `Empty` *tree has a unique preorder, inorder and postorder traversal viz.* $[\bot]$.
3. *A tree with only a single non-*`Empty` *node (which is both root and leaf) has preorder, inorder and postorder traversals of length 3 (see figure 6).*
4. *A tree with exactly two non-*`Empty` *nodes has preorder, inorder and postorder traversals of length 5 and the preorder traversals are all distinct from each other. See figure 7*

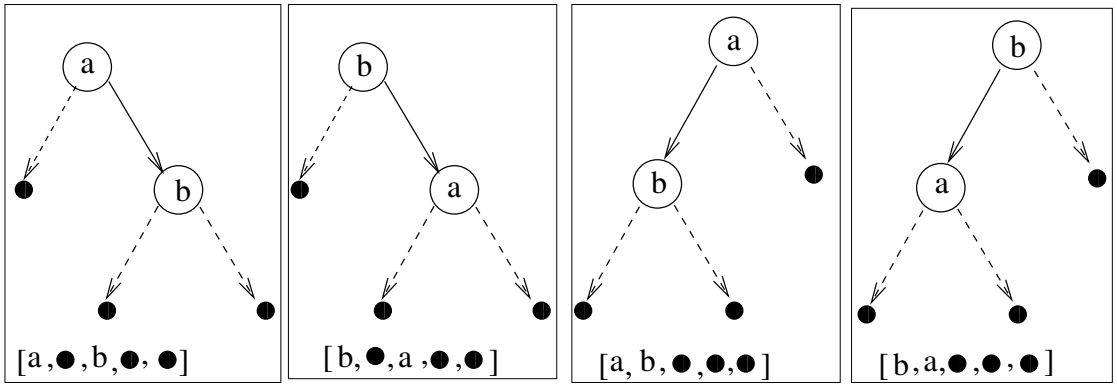

Figure 7: All binary trees with two non-`Empty` nodes

5. *Any binary tree with more than two non-*`Empty` *nodes has traversal lists of length greater than 5.*

Theorem 6.12 (Distinctness of Preorder traversals) *No two distinct binary-trees can be constructed from the same preorder traversal.*

Proof: Assume the contrary. Hence let $L = [a_0, \ldots, a_{n-1}]$ be a list of length $n > 3$ be the preorder traversal of two distinct trees T_1 and T_2. We know that $n > 3$, $a_0 \neq \bot$ and is the root of both trees and $a_{n-1} = \bot$. In fact, from facts 6.11 and figure 7 it follows that n must be greater than 5.

Let LST_1, RST_1 be the left and right subtrees of T_1, whose preorder traversals are LL_1 and LR_1 respectively. Similarly let LST_2, RST_2 be the left and right subtrees of T_2 respectively whose preorder traversals are LL_2 and LR_2 respectively. Since both trees have the same root viz. a_0, by definition 3.1 we have

$$\begin{array}{rcl} L & = & a_0 :: LL_1 @ LR_1 \\ L & = & a_0 :: LL_2 @ LR_2 \end{array}$$

from which it follows that $LL_1 @ LR_1 = LL_2 @ LR_2$. This implies that the two trees can be distinct only if there are two indices $1 \leq k < n$ and $1 \leq m < n$ such that

$$\begin{array}{rcccl} preorderLST_1 & = & LL_1 & = & [a_1, \ldots, a_{k-1}] \\ preorderRST_1 & = & LR_1 & = & [a_k, \ldots, a_{n-1}] \\ preorderLST_2 & = & LL_2 & = & [a_1, \ldots, a_{m-1}] \\ preorderRST_2 & = & LR_2 & = & [a_m, \ldots, a_{n-1}] \end{array}$$

Since the two trees are distinct $k \neq m$ and $k, m \geq 3$ by facts 6.11 part 3 we have that $a_1 \neq \bot$ (by facts 6.3 part 2). Similarly we also have $a_k \neq \bot$ and $a_m \neq \bot$. By facts 6.4 part 1 the last elements in preorder traversals have to be $\bot$. Hence $a_{k-1} = a_{m-1} = a_{n-1} = \bot$.

Translating the above into CIIs we get that $LST_1 = T^{(1,k-1)}$, $RST_1 = T^{(k,n-1)}$, $LST_2 = T^{(1,m-1)}$ and $RST_2 = T^{(m,n-1)}$. Without loss of generality we assume $k < m$. Then clearly $(1, k-1) \subset (1, m-1)$ and $(m, n-1) \subset (k, n-1)$. This implies that LST_1 is a subtree of LST_2 and RST_2 is a subtree of LST_2.

Clearly the only way two distinct trees T_1 and T_2 (with the same root) could have been formed is by joining the neighbours $(1, k-1)$ and $(k, n-1)$ in the case of T_1 and joining the neighbours $(1, m-1)$ and $(m, n-1)$ in the case of T_2. Now consider the CII $(k, m-1)$ which also defines a subtree $T^{(k,m-1)}$ (since $a_k \neq \perp$). Therefore in the case of T_1 the CII $(k, n-1)$ could have been formed by joining the neighbours $(k, m-1)$ and $(m, n-1)$. Likewise in the case of T_2 the CII $(1, m-1)$ could have been formed by joining the neighbours $(1, k-1)$ and $(k, m-1)$. This implies that at some point the list of CIIs could have been $[(1, k-1), (k, m-1), (m, n-1)]$. These are three consecutive slices such that $(k, m-1)$ is the right neighbour of $(1, k-1)$ as well as the left neighbour of $(m, n-1)$ which by lemma 6.10 is impossible. Hence our assumption that there could be two distinct trees that could be constructed from the same preorder traversal is wrong!. QED

Equipped with the above observations, our next attempt is as follows. Assume P represents a putative preorder traversal with $|P| = n > 3$ elements and hence could represent a non-trivial[9] binary-tree. We proceed as follows.

Algorithm 6.1 (Reconstruct from Preorder traversal)

1. *Find the indices of all $\perp$ elements in P and express them as CIIs i.e. as ordered pairs of the form (i, i) where i, $0 \leq i < n$, is the index of a $\perp$ element. Let this be list PI of ordered pairs representing* `Empty` *subtrees. Further let $|PI|$ be the number of CIIs in PI and $\|PI\| = \sum_{(i,j)\in PI}(j - i + 1)$ represent the number of elements from P that are in PI. Then $n - \|PI\|$ is the number of elements in P that are not present in PI. At the end of this step $\|PI\| = |PI|$ and PI contains exactly the number of occurrences of $\perp$ in P.*
2. **Recursively**
   (a) *Find all pairs of* neighbours (i, j), (k, m) *in PI with $0 < i \leq j < k = j + 1 \leq m < n$ in the list PI and $P_{i-1} \neq \perp$. This would require looking up P for the value at index $i - 1$ to determine neighbourhood. Further if $P_{i-1} = \perp$ then P is not a valid preorder traversal.*
   (b) *Join* neighbours (i, j), (k, m) *to form the CII $(i - 1, m)$ 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 $|PI|$ always decreases, but since $\|PI\|$ increases, the bound function $n - \|PI\|$ decreases with every recursive call.*

   until

   - **either** *PI reduces to a (list with a) single element $(0, n-1)$, in which case P is indeed a valid preorder traversal,*
   - **or** *PI reduces to a list with more than one element and no neighbours, in which case P is an invalid preorder traversal.*

The reader may have realised some interesting facts about the set of CIIs $I(n)$. Most of these facts are actually true about closed intervals on linear orders in general.

A curious fact about the operation of joining neighbours as outlined in algorithm 6.1 (which in turn uses the joining operation described in fact 6.6 part 8) is that the process when carried out creates a binary-tree in the ⟨*LeafBranch* 2a⟩ style in which the $\perp$ entries form the leaf nodes and the (directed) edges are the proper nesting relation. The proper nesting relation on the intervals exactly corresponds to the proper subtree relation in the original tree.

Getting back to the algorithm 6.1, we implement the first step 1 as follows.

28 ⟨*findEmpties* 28⟩≡
```
fun findEmpties PIT =
    let fun findNONEs ([], _, T) = T
          | findNONEs ((NONE::t), i, T) =
            findNONEs (t, i+1, Leaf ((i,i), List.nth (PIT, i))::T)
        val NONEs = findNONEs (PIT, 0, [])
    in
       rev NONEs
    end
```

[9] a *trivial* binary-tree is either `Empty` or consists of a single leaf node i.e. it is of the form `Node (x, Empty, Empty)`

Notice that ⟨*findEmpties* 28⟩ is general enough and independent of the traversal considered[10]. Having obtained the CIIs and values of the empty elements in order, we have them organised as a list of elements of the type `((int*int)* 'a) LBbintree` defined in ⟨*LeafBranch* 2a⟩, where each node in the tree is an ordered pair consisting of a CII and the node value in the original tree.

We now need to find the neighbours (facts 6.6 part 6) and join them as described in facts 6.6 part 8. Since the notion of neighbours differs between the various kinds of traversals, we suffix a "P" to indicate that it is applicable only to preorder traversals.

29 ⟨*areNeighboursP0* 29⟩≡

```
fun areNeighboursP0 ((i,j), (k, m)) =
  if  (i>0) andalso (i<=j) andalso (j<=k) andalso (k<= m)  then
          (case List.nth (P, i-1) of
                   NONE => false
                 | SOME _ => (k=j+1)
          )
  else false
```

Notice that there is a requirement of `P` the preorder traversal list to be always available and we need to access the `i`-th element which requires a traversal of `i` steps through the list `P`. Since this would be required repeatedly as we recursively keep joining neighbours, it is better to use an array (`arP`) and indices `i`, `j`, `k`, `m` to it for this purpose. But then one could also transform ⟨*findEmpties* 28⟩ into an iterative version viz. ⟨*findEmptiesIter* 31b⟩ in which we continue to create the binary tree using the datatype definition ⟨*EmptyNode2* 3a⟩.

Further, for technical completeness reasons, we need to make sure that all the indices used in the argument of `areNeighboursP0` are non-negative integers smaller than the length of the list `P` (equivalently the length of array `arP`). We put the various initial and exceptional conditions already illustrated in ⟨*makeTreeFromP0* 22⟩ and combine them with ⟨*findEmptiesIter* 31b⟩ and ⟨*keepJoiningNeighboursP* 32a⟩. See figure 8 to see the resulting tree that is constructed.

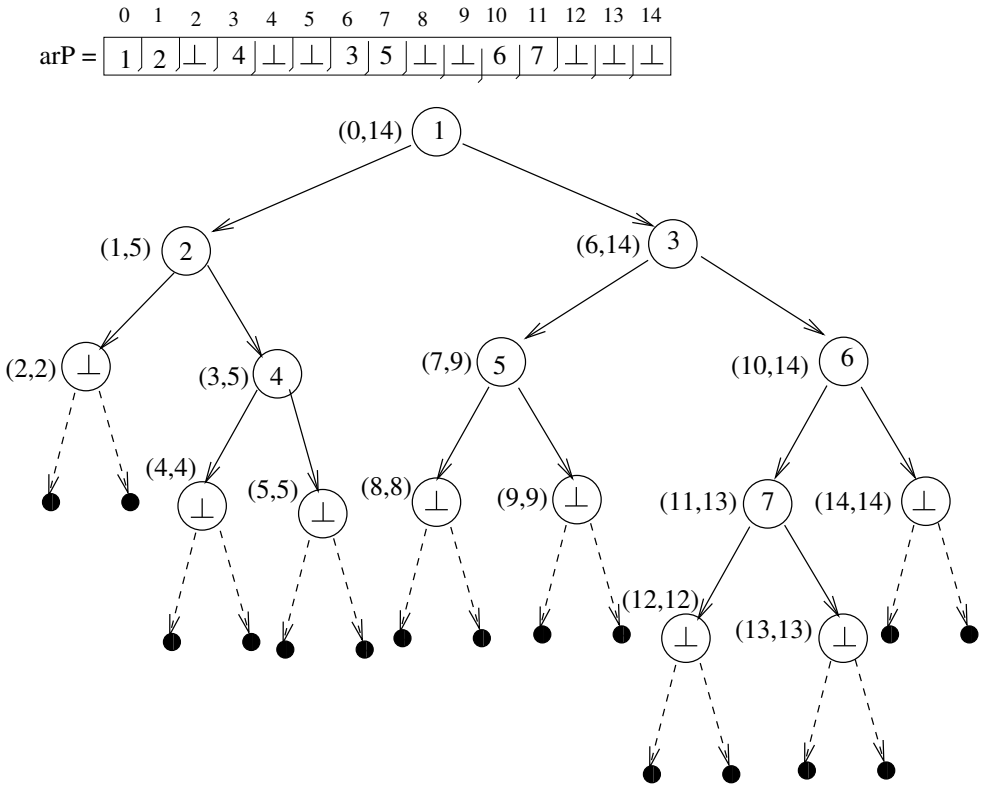

Figure 8: The binary tree constructed from preorder traversal

"Look upon this figure 8 and upon this 2". What differences do you see? It is necessary to do a transformation which erases indices (and converts leaves to `Empty`; see ⟨*eraseIndices* 32b⟩) to recover the original binary tree shown in figure 2.

30 ⟨*makeTreeFromP1* 30⟩≡ (32c 36a 37)

```
exception InvalidPreorderTraversal
fun makeTreeFromP1 [] = raise  InvalidPreorderTraversal
  | makeTreeFromP1 [NONE] = Empty
  | makeTreeFromP1 [_,_] = raise  InvalidPreorderTraversal
  | makeTreeFromP1 [SOME x, NONE, NONE] = Node (x, Empty, Empty)
  | makeTreeFromP1 [_,_,_] =  raise  InvalidPreorderTraversal
  | makeTreeFromP1 P =
    let val arP = Array.fromList P
        val n = Array.length arP
        ⟨areNeighboursP1 31a⟩
        ⟨findEmptiesIter 31b⟩
        val NONEs = findEmptiesIter (arP)
        ⟨keepJoiningNeighboursP 32a⟩
        val bt = keepJoiningNeighboursP NONEs
```

[10]The variable `PIT` indicates that it could be any of the three traversals.

```
    ⟨eraseIndices 32b⟩
in  (eraseIndices bt)
end
```

where

31a ⟨*areNeighboursP1* 31a⟩≡ (30)

```
exception Out_of_Range
fun areNeighboursP1 ((i,j), (k, m)) =
    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>0) andalso (i<=j) andalso (j<=k) andalso (k<= m)
             then (case Array.sub (arP, i-1) of
                       NONE => false
                     | SOME _ => (k=j+1)
                   )
             else false
        else raise Out_of_Range
     end
```

Since `arP` is declared "global" to the `makeTreeFromP1 P` clause, there is no need for either an argument, nor is there any need to define a nested function like `findNONEs`. We may directly compute ⟨*findEmptiesIter* 31b⟩ as shown below. As an added feature we create a list of binary trees with a single node rather than merely return the ordered pair with a the CII and node value `NONE`.

31b ⟨*findEmptiesIter* 31b⟩≡ (30 33)

```
fun findEmptiesIter (ar) =
    let val i = ref (n-1)
                (* val NONEs = ref []:((int*int) * 'a option) bintree list ref *)
                val NONEs = ref []
     in while (!i >= 0) do
        ( (case Array.sub (ar, !i) of
                NONE => NONEs := (Node (((!i,!i), NONE), Empty, Empty))::(!NONEs)
              | SOME _ => NONEs := !NONEs
          );
          i := (!i)-1
        );
        !NONEs
    end
```

We know that none of the binary trees in the list of NONEs can ever be EMPTY. To determine whether two elements in the list are neighbours, we need to look at their root nodes to see whether they satisfy the neighbourhood property (facts 6.6 part 6). Notice the expression `bt::(joinNeighboursP btList')` which is obtained after joining `bt0` and `bt1`. This follows from lemma 6.10, which shows that neighbours occur only in distinct pairs.

32a ⟨*keepJoiningNeighboursP* 32a⟩≡ (30)

```
local
    fun joinNeighboursP [] = []
      | joinNeighboursP [bt] = [bt]
      | joinNeighboursP (bt0::(bt1::btList')) =
        let val ((i,j), rootval0) = root bt0
            val ((k,m), rootval1) = root bt1
        in  if areNeighboursP1 ((i,j), (k, m))
            then let val cii = (i-1, m)
                     val rt  = Array.sub (arP, i-1)
                     val bt = Node ((cii, rt), bt0, bt1)
                 in bt::(joinNeighboursP btList')
                 end
            else if null (btList') (* list does not reduce *)
            then raise InvalidPreorderTraversal
            else bt0::(joinNeighboursP (bt1::btList'))
    end
in
fun keepJoiningNeighboursP [] = raise InvalidPreorderTraversal
  | keepJoiningNeighboursP [bt] = bt
  | keepJoiningNeighboursP btList =
    let val btList1 = joinNeighboursP btList
    in  keepJoiningNeighboursP btList1
    end (* let *)
end (* local *)
```

If indeed the original list `P` was a genuine preorder traversal of a binary tree, the result of applying `keepJoiningNeighboursP` to the list `P` would be to yield a binary tree of type `((int*int)*'a) bintree`. Finally we need to remove the indices and get a bare binary tree of type `'a bintree` shorn off all indices and option tags. Further we need to replace all subtrees of the form `Node (((i,i) NONE), Empty, Empty)` by the `Empty` subtree. We do this by the function ⟨*eraseIndices* 32b⟩

32b ⟨*eraseIndices* 32b⟩≡ (30 33)

```
exception Unexpected_Empty_Node
exception Unexpected_Node_Value
fun eraseIndices Empty = raise Empty_bintree
  | eraseIndices (Node (((i, j), NONE), Empty, Empty)) =
      if (i=j) then Empty else raise Unexpected_Empty_Node
  | eraseIndices (Node (((i, j), 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
```

Let us now try to recover the original binary tree of our running example.

32c ⟨*tryRecoverPre-complete* 32c⟩≡
⟨*basicBinTreeType* 5a⟩
⟨*euler3* 21a⟩
⟨*makeTreeFromP1* 30⟩

The following is a complete SML session which reconstructs the original tree from its preorder traversal.

```
Standard ML of New Jersey v110.78 [built: Thu Jul 23 11:21:58 2015]
- [opening ... tryRecoverPre-complete.sml]
...
val makeTreeFromP1 = fn : int option list -> int bintree
val it = () : unit
- use "bintreeDat.sml";
[opening bintreeDat.sml]
...
val t1 = Node (1,Node (2,Empty,Node ...)) : int bintree
val it = () : unit
- val P = preorder t1;
val P =
  [SOME 1,SOME 2,NONE,SOME 4,NONE,NONE,SOME 3,SOME 5,NONE,NONE,SOME 6,SOME 7,
   ...] : int option list
- val t1P = makeTreeFromP1 P;
val t1P = Node (1,Node (2,Empty,Node ...)) : int bintree
- t1P = t1;
val it = true : bool -
```

Since we have shown that preorder traversals are unique and distinct for each binary-tree and there exists a function `makeTreeFromP1` which reconstructs the binary tree from its preorder traversal, we will rename it `preorderInverse` and mathematically it is just the inverse of the $preorder$ function, so we represent it as $preorder^{-1}$ in the sequel.

6.4 Reconstruction from Inorder traversals

One would assume that we would require only very small changes to reconstruct the tree from the other traversals now. For example one could conjecture that for inorder traversals we replace the suffix "I" for "P" and implement the changes as per the definitions in (6) and (8) resulting code then looks as follows.

33 ⟨*makeTreeFromI1* 33⟩≡ (34c)

```
exception Invalid_Inorder_Traversal
fun makeTreeFromI1 [] = raise  Invalid_Inorder_Traversal
  | makeTreeFromI1 [NONE] = Empty
  | makeTreeFromI1 [_,_] = raise  Invalid_Inorder_Traversal
  | makeTreeFromI1 [NONE, SOME x, NONE] = Node (x, Empty, Empty) (* Inorder *)
  | makeTreeFromI1 [_,_,_] =  raise  Invalid_Inorder_Traversal
  | makeTreeFromI1 I =
    let val arI = Array.fromList I
        val n = Array.length arI
        ⟨areNeighboursI1 34a⟩
        ⟨findEmptiesIter 31b⟩
        val NONEs = findEmptiesIter (arI)
        ⟨keepJoiningNeighboursI 34b⟩
        val bt = keepJoiningNeighboursI NONEs
        ⟨eraseIndices 32b⟩
    in  (eraseIndices bt)
    end
```

where

34a ⟨*areNeighboursI1* 34a⟩≡ (33)

```
exception Out_of_Range
fun areNeighboursI1 ((i,j), (k, m)) =
    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)
             then (case Array.sub (arI, j+1) of
                      NONE => false
                    | SOME _ => (k=j+2) (* Inorder *)
                   )
             else false
        else raise Out_of_Range
     end
```

and

34b ⟨*keepJoiningNeighboursI* 34b⟩≡ (33)

```
local
    fun joinNeighboursI [] = []
      | joinNeighboursI [bt] = [bt]
      | joinNeighboursI (bt0::(bt1::btList')) =
        let val ((i,j), rootval0) = root bt0
            val ((k,m), rootval1) = root bt1
        in  if areNeighboursI1 ((i,j), (k, m))
            then let val cii = (i, m)
                     val rt  = Array.sub (arI, j+1) (* Inorder *)
                     val bt = Node ((cii, rt), bt0, bt1)
                 in bt::(joinNeighboursI btList')
                 end
            else bt0::(joinNeighboursI (bt1::btList'))
    end
in
fun keepJoiningNeighboursI [] = raise Invalid_Inorder_Traversal
  | keepJoiningNeighboursI [bt] = bt
  | keepJoiningNeighboursI btList =
    let val btList1 = joinNeighboursI btList
    in  keepJoiningNeighboursI btList1
    end (* let *)
end (* local *)
```

34c ⟨*tryRecoverI-complete* 34c⟩≡

⟨*basicBinTreeType* 5a⟩
⟨*euler3* 21a⟩
⟨*makeTreeFromI1* 33⟩

However the following session shows that this does not work for our example.

```
...
val makeTreeFromI1 = fn : int option list -> int bintree
val it = () : unit
- use "bintreeDat.sml";
[opening bintreeDat.sml]
...
val t1 = Node (1,Node (2,Empty,Node ...)) : int bintree
val it = () : unit
- val I = inorder t1;
val I =
  [NONE,SOME 2,NONE,SOME 4,NONE,SOME 1,NONE,SOME 5,NONE,SOME 3,NONE,SOME 7,
   ...] : int option list
- val t1I = makeTreeFromI1 I;
val t1I = Node (5,Node (4,Node ...  ))  :  int bintree
- t1I = t1; \colorbox{red}{val it = false : bool}\colorbox{yellow}{- val I1 = inorder
I1 =[NONE,SOME 2,NONE,SOME 4,NONE,SOME 1,NONE,SOME 5,NONE,SOME 3,NONE,SOME 7,...]
:  int option list\colorbox{yellow}{- I1 = I;}
val it = true :  bool -
```

The resulting tree `t1I` is shown in figure 9 and is clearly different from the original tree in figure 2 and hence is wrong! Further, we also see from the subsequent lines that the inorder traversal of tree `t1I` is `I1` which equals `I` which is the inorder traversal of the original tree `t1`. Hence unlike the case of preorder (and postorder) traversal, the use of the marker $\perp$ for an empty node in the traversal is not enough to guarantee distinct inorder traversals for distinct binary-trees (c.f. theorem 6.5 for preorder traversals and problem 2 in exercise 6.1).

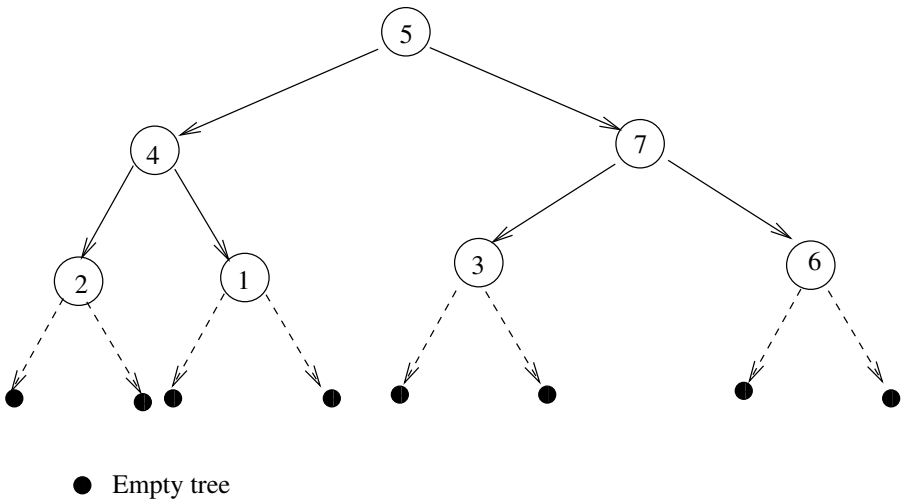

Figure 9: The binary tree constructed from the inorder traversal of the tree in figure 2

The reason seems to be the following.

The inorder traversal of example 2.1 yields the list $[\perp, 2, \perp, 4, \perp, 1, \perp, 5, \perp, 3, \perp, 7, \perp, 6, \perp]$. Given this list, the definition of neighbours yields the following balanced binary tree, since there is nothing in the list which prevents the node 1 being identified as a leaf node. The left to right traversal of the list to identify leaf nodes and using the definition of neighbours then makes the node 5 the root node as the following grouping of nodes shows. The first line below shows the identification of `Empty` by parenthesising the $\perp$. The subsequent lines – again using a pair of matching parentheses – show the subtrees formed by joining neighbours using ⟨*keepJoiningNeighboursI* 34b⟩ (which is simply one particular way of implementing algorithm 2, but is not exactly the same!). The bold numerals indicate the root of the (sub)tree.

$$[(\perp), (\perp), (\perp), (\perp), (\perp), (\perp), (\perp), (\perp)]$$
$$[((\perp), \mathbf{2}, (\perp)), ((\perp), \mathbf{1}, (\perp)), ((\perp), \mathbf{3}, (\perp)), ((\perp), \mathbf{6}, (\perp))]$$

$$[(((\bot), 2, (\bot)), \mathbf{4}, ((\bot), 1, (\bot))), (((\bot), 3, (\bot)), \mathbf{7}, ((\bot), 6, (\bot)))]$$
$$[(((\bot), 2, (\bot)), 4, ((\bot), 1, (\bot))), \mathbf{5}, (((\bot), 3, (\bot)), 7, ((\bot), 6, (\bot)))]$$

Hence the main problem in the case of inorder traversals as opposed to preorder traversals, is the failure of being able to identify leaf nodes of the original tree uniquely. That is, every slice of the form $[\bot, l, \bot]$ in a valid inorder traversal need not represent a leaf node of the original tree. Further preorder traversals allow an unique left-to-right scheme of scanning for leaf nodes which fails in the inorder case. Notice that left-to-right scanning is not present in the original algorithm 6.1, it is only present in the implementations ⟨*keepJoiningNeighboursP* 32a⟩ and ⟨*keepJoiningNeighboursI* 34b⟩ due to the inductive structure of lists.

6.5 Reconstruction from Postorder traversals

It is clear from the duality properties (see exercise 6.1) of postorder traversals vis-à-vis preorder traversals that an algorithm and program along similar lines may be designed which will reconstruct the tree from its postorder traversal.

Exercise 6.2

1. *Use the solutions to exercise 6.1 to reconstruct a binary tree from its postorder traversal.*
2. *When does a list of node labels consitute an invalid postorder traversal of a given tree?*

However given that we have a complete algorithm and program to reconstruct a tree from its preorder traversal, we could use it in conjunction with fact 3.2 to derive another method of reconstruction viz. ⟨*tryRecoverPost-complete* 36a⟩ from a postorder traversal. Let $preorder^{-1}$ denote the function that constructs a binary-tree from a valid preorder traversal. We then have the following.

Corollary 6.13 (corollary of fact 3.2)

$$twist\ (preorder^{-1}\ (rev\ (postorder\ T))) = T \tag{15}$$

which we may use to derive the following code.

36a ⟨*tryRecoverPost-complete* 36a⟩≡
⟨*basicBinTreeType* 5a⟩
⟨*euler3* 21a⟩
⟨*makeTreeFromP1* 30⟩
⟨*twist* 5b⟩
⟨*makeTreeFromRevP1* 36b⟩

where

36b ⟨*makeTreeFromRevP1* 36b⟩≡ (36a 37)
```
fun makeTreeFromRevP1 S = twist  (makeTreeFromP1  (rev S))
```

That is, for any given postorder traversal S, of a binary tree T let $RS = rev\ (S)$. This yields a preorder traversal of the binary tree $twist(T)$. Using the function ⟨*makeTreeFromP1* 30⟩ we construct $twist(T)$ and apply $twist$ once again and by the self-dual property of $twist$ (exercise 2.1) we obtain T.

The following SML session supports this intuition.

```
Standard ML of New Jersey v110.78 [built: Thu Jul 23 11:21:58 2015]
- use "tryRecoverPre-complete.sml";
...
val it = () : unit
- use "twist.sml";
[opening twist.sml]
val twist = fn : 'a bintree -> 'a bintree
val it = () : unit
- use "bintreeDat.sml";
[opening bintreeDat.sml]
...
val t1 = Node (1,Node (2,Empty,Node ...)) : int bintree
val it = () : unit
- val S = postorder t1;
val S =
  [NONE,NONE,NONE,SOME 4,SOME 2,NONE,NONE,SOME 5,NONE,NONE,SOME 7,NONE,...]
  : int option list
- val twistedP = makeTreeFromP1 (rev S);
val twistedP = Node (1,Node (3,Node ...))
  : int bintree
- val t1' = twist twistedP;
val t1' = Node (1,Node (2,Empty,Node ...)) : int bintree
- t1 = t1';
val it = true : bool
-
```

7 Making a module

We have seen that marking the `Empty` nodes and listing them as part of the traversals helps in reconstructing the tree from preorder and postorder traversals. Using the data-types ⟨*LeafBranch* 2a⟩ or ⟨*ZeroOneTwo* 2b⟩ would not have helped in reconstructing the binary tree from the preorder or postorder traversals.

However in the case of inorder traversals, there is either some information missing and/or some "noisy" information gets added, resulting in the ambiguity of inorder traversals. Simply the fact that leaf nodes are not uniquely identifiable in an inorder traversal makes the traversal ambiguous and hence two structurally distinct trees (with the same node labels) can have identical inorder traversals with the markers. The question that remains open is, "What structural information accounts for this ambiguity?" We leave this question unanswered for some future investigation.

But to complete the program, we need to put it all together to form a module. Since the function `makeTreefromP1` is the inverse of `preorder`, we rename it `preorderInverse`. Likewise we rename `makeTreefromRevP1` to `postorderInverse`. It is rather odd to have a function called `eulerTour3`. So we rename it `euler`. We are now ready to present a complete module as follows. We first create the structure `Bintree` containing all the functions we need. We would also like to create a signature which we call `BINTREE`[11]

The association between the structure `Bintree` and the signature is specified either transparently in the structure declaration using a `:` or opaquely using a `:>`.

37 ⟨*module-structure-bintree* 37⟩≡ (38b)

```
structure Bintree : BINTREE =
struct
```

[11] It has now become standard ML convention to use lower-case letters for the name of a data-type and use the same name with the initial letter in upper-case for the structure which defines the implementation of the datatype and use all upper-case letters with the same name for the name of the signature. Example: datatype `int` is implemented in the structure `Int` whose signature is defined in `INT`.

```
        ⟨EmptyNode2 3a⟩
        ⟨basicBinTreeType 5a⟩
        ⟨twist 5b⟩
        ⟨bt-map 17a⟩
        ⟨euler3 21a⟩
        val euler = eulerTour3
        ⟨makeTreeFromP1 30⟩
        val preorderInverse = makeTreeFromP1
        ⟨makeTreeFromRevP1 36b⟩
        val postorderInverse = makeTreeFromRevP1
end (* struct *)
```

To define the signature we need to specify the type of each function and object that needs to be made visible. We use our sessions to define the types of the various function.

38a ⟨*module-signature-bintree* 38a⟩≡ (38b)

```
signature BINTREE =
sig
    datatype 'a bintree = Empty |
                          Node of 'a * 'a bintree * 'a bintree
    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 BTmap       : ('a -> 'b) -> 'a bintree -> 'b bintree
    val euler       : 'a bintree -> 'a option list * 'a option list * 'a option list
    val preorder    : 'a bintree -> 'a option list
    val inorder     : 'a bintree -> 'a option list
    val postorder   : 'a bintree -> 'a option list
    exception InvalidPreorderTraversal
    val preorderInverse : 'a option list -> 'a bintree
    val postorderInverse: 'a option list -> 'a bintree
end (* sig *)
```

Finally we could leave the signature and structure in different files (and remember to use them in the right order when we do use them) or we could combine them further into a single file.

38b ⟨*module-bintree-complete* 38b⟩≡

```
    ⟨module-signature-bintree 38a⟩

    ⟨module-structure-bintree 37⟩
```

and try it out. Here is the session that we tried.

```
Standard ML of New Jersey v110.78 [built: Thu Jul 23 11:21:58 2015]
[opening module-bintree.sml]
[autoloading]
[library $SMLNJ-BASIS/basis.cm is stable]
[autoloading done]
module-bintree.sml:124.30 Warning: calling polyEqual
signature BINTREE =
  sig
    datatype 'a bintree = Empty ...
    ...
  end
structure Bintree : BINTREE
-
- open Bintree;
opening Bintree
  datatype 'a bintree = ...
- ...
- use "bintreeDat.sml";
[opening bintreeDat.sml]
...
val t1 = Node (1,Node (2,Empty,Node ...)) : int bintree
val it = () : unit
- val (P, I, S) = euler t1;
val P =
  [SOME 1,SOME 2,NONE,SOME 4,NONE,NONE,SOME 3,SOME 5,NONE,NONE,SOME 6,SOME 7,
   ...] : int option list
val I =
  [NONE,SOME 2,NONE,SOME 4,NONE,SOME 1,NONE,SOME 5,NONE,SOME 3,NONE,SOME 7,
   ...] : int option list
val S =
  [NONE,NONE,NONE,SOME 4,SOME 2,NONE,NONE,SOME 5,NONE,NONE,SOME 7,NONE,...]
  : int option list
- val tP = preorderInverse P;
val tP = Node (1,Node (2,Empty,Node  ...)) : int bintree
- tP = t1;
val it = true :  bool
- val tS = postorderInverse S;
val tS = Node (1,Node (2,Empty,Node ...)) : int bintree
- tS = t1;
val it = true :  bool
-
```

8 Inorder traversals Revisited

While we do have a nicely structured module, it still lacks a good inorder-inverse function to complete it. We would like to add the minimum amount of information in our traversals to permit the calculation of inverse functions for all the traversals. The positional information that we studied in section 5 is way too much information as it should now be evident. We do not actually require the absolute position of each node in the tree (even if there are duplicate occurrences of node labels).

We revisit the problem of the ambiguity in identifying leaves in inorder traversals. It appears that if we could simply add leaf information while performing the traversal, we would be able to recover the tree from the inorder traversal. This requires just an extra bit (literally a "bit") of information viz. whether a non-empty node has any children at all. Naturally adding this bit of information for inorder traversals also affects the preorder and postorder traversals in an obvious way, since we would like to have the same types assigned to all the traversals. Further for our lists to

have a uniform representation, the non-leaf nodes should also have a bit with complementary information. So we assign the bit `1` to all nodes which have at least one child and the bit `0` to the leaf-nodes.

We first redefine the Euler tour to include this information in the traversals uniformly. This yields four clauses (actually four distinct patterns) in the definition of the tour since one or both subtrees rooted at a node may be `Empty`.

The traversals now look as follows:

40a ⟨*euler4* 40a⟩≡ (40b)

```
fun eulerTour4 (Empty) = ([NONE], [NONE], [NONE])
  | eulerTour4 (Node (x, Empty, Empty)) = (* leaf node *)
     ([SOME (x,0), NONE, NONE], [NONE, SOME (x,0), NONE], [NONE, NONE, SOME (x,0)])
  | eulerTour4 (Node (x, left, right)) = (* non-leaf node *)
    let
        val (preL, inL, postL) = eulerTour4 left
        val (preR, inR, postR) = eulerTour4 right
    in ((SOME (x,1))::preL@preR, inL@((SOME (x,1))::inR), postL@postR@[(SOME (x,1))]
    end
fun preorder bt = #1 (eulerTour4 bt)
fun inorder bt = #2 (eulerTour4 bt)
fun postorder bt = #3 (eulerTour4 bt)
```

A complete program for the traversals then is given by

40b ⟨*euler-bit-complete* 40b⟩≡

⟨*EmptyNode2* 3a⟩

⟨*euler4* 40a⟩

A session using this with our favourite example is shown below.

```
Standard ML of New Jersey v110.78 [built: Thu Jul 23 11:21:58 2015]
[opening euler-bit-complete.sml]
...
val eulerTour4 = fn
  : 'a bintree
    -> ('a * int) option list * ('a * int) option list *
       ('a * int) option list
val preorder = fn : 'a bintree -> ('a * int) option list
val inorder = fn : 'a bintree -> ('a * int) option list
val postorder = fn : 'a bintree -> ('a * int) option list
- use "bintreeDat.sml";
[opening bintreeDat.sml]
...
val t1 = Node (1,Node (2,Empty,Node ...)) : int bintree
val it = () : unit
- val (P, I, S) = eulerTour4 t1;
val P =[SOME (1,1),SOME (2,1),NONE,SOME (4,0),...] : ...
val I =[NONE,SOME (2,1),NONE,SOME (4,0),NONE,SOME (1,1),...] : ...
val S =[NONE,NONE,NONE,SOME (4,0),SOME (2,1),...] : ...
-
```

Put more succinctly in our notation we have

$$\begin{aligned} P &= [(1,1),(2,1),\bot,(4,0),\bot,\bot,(3,1),(5,0),\bot,\bot,(6,1),(7,0),\bot,\bot,\bot] \\ I &= [\bot,(2,1),\bot,(4,0),\bot,(1,1),\bot,(5,0),\bot,(3,1),\bot,(7,0),\bot,(6,1),\bot] \\ S &= [\bot,\bot,\bot,(4,0),(2,1),\bot,\bot,(5,0),\bot,\bot,(7,0),\bot,(6,1),(3,1),(1,1)] \end{aligned}$$

One reason why we cannot avoid outputting the `Empty` trees in the traversals is that then we may again be faced with ambiguity because then we would not be able to clearly identify the nodes which have out-degrees of 1 and even if we did we would not know which of their children is `Empty`[12].

First of all we have

Theorem 8.1 (Uniqueness of traversals) . *No two distinct binary trees can yield the same preorder, inorder or postorder traversal.*

whose proof is left as an exercise (see part 1 of exercise 8.1.

Let us now get back to the problem of reconstruction. Look at algorithm 6.1. We now do the reconstruction for all three traversals by modifying the algorithm as follows.

Algorithm 8.1 (Reconstruct from each traversal) *1. Assume each traversal is a list of length $n > 2$, i.e. the indices range from 0 to $n-1$.*

(a) Find the indices of all the leaf *nodes (by checking for a 0 in the second component of a non-$\bot$ ordered pair) in the traversal.*

(b) For each leaf node (of index i) identify the indices of the `Empty` *nodes associated with them. These would be their immediate adjacent elements as stated in Fact 6.3 and 6.6 form the CII*

Preorder: $(i, i+2)$, *where* $0 \leq i < n-2$, $P_i \neq \bot$, $P_{i+1} = P_{i+2} = \bot$
Inorder: $(i-1, i+1)$, *where* $0 < i < n-1$, $I_i \neq \bot$, $I_{i-1} = I_{i+1} = \bot$,
Postorder: $(i-2, i)$, *where* $2 \leq i < n$, $S_i \neq \bot$, $S_{i-2} = S_{i-1} = \bot$,

Otherwise raise the `InvalidTraversal` *exception.*

Let these be the lists PI, II and SI respectively of ordered pairs representing leaf *subtrees.*

2. **Recursively**

[12] Also see part 5 of exercise 8.1

(a) *Find all pairs of* neighbours (i, j), (k, m) *(see Facts 6.6 part 6) with* $0 \leq i \leq j < k \leq m < n$ *such that*

Preorder: $0 < i \leq j < k = j + 1 \leq m < n$ *in the list* PI *and* $P_{i-1} \neq \perp$,
Inorder: $k = j + 2$ *and* $L_{j+1} \neq \perp$ *in the list* II *and* $I_{i-1} \neq \perp$,
Postorder: $k = j + 1$, $m < n - 1$ *in the list* S_i *and* $S_{m+1} \neq \perp$.

This would require looking up P, I *and* S *respectively for the value at indices* $i - 1$, $j + 1$ *and* $m + 1$ *resp. to determine neighbourhood. Further, if any of* $P_{i-1} = \perp$ *or* $I_{i-1} = \perp$ *or* $S_{m+1} = \perp$ *then that list is not a valid traversal and an* `InvalidTraversal` *exception should be raised.*

(b) *Join* neighbours (i, j), (k, m) *to form the CII* (p, q) *where*

Preorder: $(p, q) = (i - 1, m)$
Inorder: $(p, q) = (i, m)$
Postorder: $(p, q) = (i, m + 1)$

such that both (i, j) *and* (k, m) *are nested* (p, q). *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* $|PI|$, $|II|$ *and* $|SI|$ *always decrease, but since* $\|PI\|$, $\|II\|$ *and* $\|SI\|$ *increase, the bound functions* $n - \|PI\|$, $n - \|II\|$ *and* $n - \|SI\|$ *decrease with every recursive call.*

until

- **either** PI, II, SI *reduces to single element lists* $(0, n - 1)$, *in which case* P, I *and* S *are indeed valid traversals,*
- **or** PI, II, SI *reduce to lists with more than one element and no neighbours, in which case* P, I, S *are invalid traversals.*

Exercise 8.1 *Create a noweb document which provides the solution to the following problems.*

1. *Prove theorem 6.5 for inorder and postorder traversals.*
2. *Complete the functions* `preorderInverse`, `inorderInverse` *and* `postorderInverse` *of algorithm 8.1 respectively.*
3. *Create the binary-tree module (with an appropriate signature and structure) that are mutually compatible and implement it in SML.*
4. *Design a suite of tests from the most trivial to the most complex and test your module and present the results. Your test data should exhaustively explore all possible cases of execution. Keep the following in mind.*
   - *Each test datum should be structurally different from every other and explore a different condition. Do not simply replace values by other values to create a new test datum.*
   - *Even if the same test datum is used for different tests, each test should explore a different aspect of the module.*
5. *Suppose instead of a "bit" information we actually used the numbers* 0, 1 *and* 2 *to specify the number of children each non-*`Empty` *node has. Then*
   (a) *Redefine the Euler tour to incorporate this information.*
   (b) *Does the uniqueness theorem 8.1 hold in this case too? Prove or disprove the claim.*
   (c) *If the uniqueness theorem holds, derive an algorithm to reconstruct the binary tree.*

9 Concluding Remarks

While this document is mostly about a data-structure that is very commonly taught in undergraduate CS courses, there are nevertheless certain fairly interesting facts and theorems that I never knew before I started working on it and which I consider original. Fact 3.2 is one such. Others (some of which are more specific to the datatype representation) are the properties of CIIs (see facts 6.6 and their relation-ship to subtrees of a binary-tree which helped in creating the algorithm. But most importantly, everybody who has studied binary-trees knows that one can reconstruct a binary tree from the preorder (or postorder) and inorder traversals as shown in section 4, but

to the best of my knowledge, it is not known that you can reconstruct the binary tree from only the preorder or post-order traversal alone by simply choosing markers for empty trees. This does not suffice for inorder traversals, and we require to put in literally a "bit" more information to be able to reconstruct the tree.

Last but not least this document illustrates the use of `noweb` as a tool for developing algorithms, proofs, programs and documentation. With a variety of tools like the LaTeXpackages, noweb, xfig and my own scripts written in perl and shell for the purpose of creating both stand-alone functions and complete modules, it is a wholesome exercise in software engineering combining mathematical rigour, algorithms along with their proofs and programming (including testing) to create a piece of software that is proven, tested and fully justified. In addition, components can be tested individually and in combination with others.