Binary Trees

Binary Tree Datatype

The empty binary tree "Empty" is a binary tree.

If l and r are binary trees and v is a value of **type ‘a** then Node (v, l, r) is a binary tree.

Nothing else is a binary tree.

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

Size of a Tree

size(T) = n where n is size of the tree, that is the number of nodes in T

```
fun size (t : 'a tree) : int  =
  case t of
     Empty           => 0
   | Node (a, l, r)  => size l + size r + 1
```

Binary Search Trees

Binary search trees are binary trees in which the data satisfies an ordering constraint

The values in the left subtree are all smaller than the value at the root and the values in the right subtree are all larger.

In order to construct and search these trees we have to be able to compare the values

The values in the trees we have defined can be any type, integers, reals, strings, pairs, lists, functions, etc

However:

Binary Search Trees

However not all types have relational operators and that may depend on the language

For example, in SML we can compare two lists for equality but not check if one is less than another. The type list is said to be an “equality” type

In SML real values cannot be compared for equality (Why not?)

Functions cannot be compared for equality. Equality of functions is an undecidable problem.

In OCamL lists can be compared and the result is based on a lexicographic ordering

Binary Search Trees

In the following examples, we will deal with binary trees in which each value stored at a node is a pair

The pair (k, d) is taken to represent a unique key with associated data

The keys are assumed to have an ordering which can be checked by relational operators

The associated data can be of any type

Insertion

The following function will insert a key-data pair (k, d) into a binary search tree

If there is no pair with key k in the tree, the pair will be added

Any previous occurrences of (k, d') are overwritten by (k, d)

The type of the function is ('a * 'b) * ('a * 'b) tree -> ('a * 'b) tree

Insert into a Binary Search Tree

```
fun insert ( x, t )   =
 case t of
   Empty           => Node(x, Empty, Empty)
 | Node (y, l, r) => if #1 x = #1 y then Node(x, l, r)
                     else if #1 x <  #1 y then Node(y, insert (x, l), r)
                     else Node(y, l, insert (x r))
```

Insertion – improved version

Using pattern matching and local variables improves the code

Insert into a Binary Search Tree

```
fun insert ( x, t )   =
 case t of
   Empty           => Node(x, Empty, Empty)
 | Node (y, l, r) => let val (k1,d1) = x
                         val (k2,d2) = y
                     in
                         if k1 = k2 then Node(x, l, r)
                         else if k1 < k2 then Node(y, insert (x, l), r)
                        else Node(y, l, insert (x r))
                     end
```

Search

The following function will search for a key corresponding to a key-data pair (k, d) in a binary search tree

If there is no node with the given key in the tree, return None.

If there is a node with the given key and associated data, d, return Some(d)

The type of the function is 'a -> ('a * 'b) tree -> 'b option

NOTE: Although the type of this function is polymorphic in 'a and 'b, it depends on the fact that we do have functions to compare elements of type 'a

Search

```
fun search k t =
  case t of
    Empty           => None
  | Node (y, l, r) => let val (k1, d) = y
                      in
                         if k = k1 then Some(d)
                         else
                            if k < k1 then lookup x l
                             else lookup x r)
                      end
```

Inorder traversal

Inorder traversal flattens a tree into a list

Inorder traversal first traverses the left subtree, then visits the root, and finally traverse the right subtree

```
inorder : 'a tree -> 'a list
```

```
fun inorder t =
  case
    Empty          => []
  | Node(x, l, r) -> inorder l @ [x] @ inorder r
```

r

```
let rec collect t = match t with
 | Empty       -> []
 | Node(a, l, r) -> collect l @ [a] @ collect r



(*********************************************************************)
(* Examples of trees *)

let t1 = Node(9, Node(3, Node(2, Empty, Empty),
                                                Node(5, Node(4, Empty, Empty), Node(6, Empty, Empty))),
                                Node(13, Node(11, Empty, Empty), Node(15, Empty, Empty)));;



(*         9
          /   \
          3     13
         / \   / \
        2   5  11  15
           / \
          4   6
*)
```