Binary Search Trees

A binary search tree is a data structure for quick access to elements. The idea is to store elements in a tree, with the property that all elements less than (or equal) to the root at in the left subtree, and all elements greater than the root are in the right subtree.

It the tree is *balanced*, that is, as close to complete as possible, to find an element in the tree (or to see that it is not in the tree), it takes time worst-case roughly $\log(N)$, where N is the number of elements in the tree. In contrast, looking for an element in a list (or seeing that the element is not in the list) takes time worst-case roughly N, where N is the number of elements in the list. When N is large, $\log(N)$ is much much smaller than N. So binary search trees are more efficient when search amongst large sets of numbers.

Why might you use binary search trees? If you wanted to implement, say, sets of numbers. You could use a binary search tree to implement the set, and seeing if an element is in the set corresponds to looking for an element in the tree.

We are interested in coming up with an ADT for binary search trees (or bst). The key is to make sure that every bst we construct satisfies the property that all elements in the left subtree of a node are less than or equal to the root value, and all elements in the right subtree of a node are greater than the root value, and this for every node in the bst.

To ensure this, instead of providing a constructor such as node (like we did for the tree ADT), we provide a constructor insert that inserts an element in the right spot in the tree.

```
empty : () -> bst
insert : (integer bst) -> bst

isEmpty : (bst) -> bool
root : (bst) -> integer
left : (bst) -> bst
right : (bst) -> bst
```

What's the algebratic specification? That's more interesting. Again, we need to come up with equations corresponding to the application of selectors to each of the constructors (`empty`, and `insert`). The constructor insert is the more interesting one. Here is the algebraic specification.

```
(isEmpty (empty)) = true
```

```
(isEmpty (insert a t)) = false

(root (insert a t)) =
    a                 if (isEmpty t)=true
    (root t)          if (isEmpty t)=false

(left (insert a t)) =
    (empty)                 if (isEmpty t)=true
    (insert a (left t))     if (isEmpty t)=false and a<=(root t)
    (left t)                if (isEmpty t)=false and a>(root t)

(right (insert a t)) =
    (empty)                 if (isEmpty t)=true
    (right t)               if (isEmpty t)=false and a<=(root t)
    (insert a (right t))    if (isEmpty t)=false and a>(root t)
```

Intuitively, the specification for left and right tell you that to get the left subtree of a bst, you keep all the inserts that inserted something that was less than the root of the tree, and symmetrically for getting the right subtree of a bst. If you're not convinced, try the specification on a few examples.

So what's an implementation for the above? We use the same representation for trees that we used last lecture, and the same functions implementation for `empty`, `isEmpty`, `left`, `right`, `root`.

```
(defun empty ()
  NIL)

(defun isEmpty (tr)
  (endp tr))

(defun root (tr)
  (first tr))

(defun left (tr)
  (if (endp tr)
      NIL
    (second tr)))

(defun right (tr)
  (if (endp tr)
      NIL
    (third tr)))
```

For `insert`, we simply compare the value to insert to the root of the tree to insert into. If we're inserting into an empty tree, we create a node for the value. If the value is smaller or equal to the root value, we recursively insert the value into the left subtree of the current tree, and if the value is greater than the root value, we recursively insert the value into the right subtree of the current tree. The value gently finds its right spot in the tree.

```
(defun insert (a tr)
  (if (endp tr)
      (list a NIL NIL)
    (if (<= a (root tr))
        (list (root tr) (insert a (left tr)) (right tr))
      (list (root tr) (left tr) (insert a (right tr))))))
```

Let's write down a predicate `bst?` that checks that a structure is actually a binary search tree, by checking the bst property that all values in the left subtree are less than or equval to the root value, and all values in the right subtree are greater than the root value. We need a couple of auxiliary functions to make the definition of `bst?` more palatable: `(all-<= a tr)` checks that every element in tree `tr` is `<=` than `a`, and `(all-> a tr)` checks that every element in tree `tr` is `>` than `a`.

```
(defun all-<= (a tr)
  (if (endp tr)
      T
    (and (<= (root tr) a)
         (all-<= a (left tr))
         (all-<= a (right tr)))))

(defun all-> (a tr)
  (if (endp tr)
      T
    (and (> (root tr) a)
         (all-> a (left tr))
         (all-> a (right tr)))))

(defun bst? (tr)
  (if (endp tr)
      (= tr NIL)
    (and (all-<= (root tr) (left tr))
         (all-> (root tr) (right tr))
         (bst? (left tr))
         (bst? (right tr)))))
```

As a sanity check, we may want to prove that inserting a value into a binary search tree yields a binary search tree.

```
(implies (bst? tr) (bst? (insert a tr))))
```

That's not so obvious to prove. I'll leave it as a (hard) exercise for now. I don't mind so much not having it, because we have the algebraic specification for binary search trees, and we can prove that our implementation satisfies the specification.

Note how we translated the above algebraic specification into ACL2.

```
(defthm isempty-empty
  (isEmpty (empty)))

(defthm isempty-insert
  (implies (and (bst? tr)
                (integerp a))
           (not (isEmpty (insert a tr)))))

(defthm root-insert-1
  (implies (and (bst? tr)
                (integerp a)
                (isEmpty tr))
           (= (root (insert a tr)) a)))

(defthm root-insert-2
  (implies (and (bst? tr)
                (integerp a)
                (not (isEmpty tr)))
           (= (root (insert a tr)) (root tr))))

(defthm left-insert-1
  (implies (and (bst? tr)
                (integerp a)
                (isEmpty tr))
           (= (left (insert a tr)) (empty))))

(defthm left-insert-2
  (implies (and (bst? tr)
                (integerp a)
                (not (isEmpty tr))
                (<= a (root tr)))
           (= (left (insert a tr)) (insert a (left tr)))))
```

```
(defthm left-insert-3
  (implies (and (bst? tr)
                (integerp a)
                (not (isEmpty tr))
                (> a (root tr)))
           (= (left (insert a tr)) (left tr))))

(defthm right-insert-1
  (implies (and (bst? tr)
                (integerp a)
                (isEmpty tr))
           (= (right (insert a tr)) (empty))))

  (defthm right-insert-2
    (implies (and (bst? tr)
                  (integerp a)
                  (not (isEmpty tr))
                  (<= a (root tr)))
             (= (right (insert a tr)) (right tr))))

  (defthm right-insert-3
    (implies (and (bst? tr)
                  (integerp a)
                  (not (isEmpty tr))
                  (> a (root tr)))
             (= (right (insert a tr)) (insert a (right tr)))))
```

There, so we have an implementation of binary search trees (which is, by the way, the standard implementation), and we showed formally that the implementation satisfied the algebraic specification. If you believe that the algebraic specification truly captures what a binary search tree is doing, then we're done.

Let's look at a very interesting property of binary search trees. You remember inorder traversals, right, from last time? Here's the code for a tree inorder traversal.

```
(defun inorder (tr)
  (if (isEmpty tr)
      NIL
    (append (inorder (left tr))
            (cons (root tr)
                  (inorder (right tr))))))
```

Binary search trees have an interesting property with respect to inorder traversals. To get a sense for it, consider the following function that lets us construct binary search trees easily, given a list of integers.

```
(defun make-tree (L)
  (if (endp L)
      (empty)
    (insert (car L) (make-tree (cdr L)))))
```

Note that the elements are inserted in the tree in the reverse order of that in which they appear in the list. Thus, for instance:

```
ACL2 !>(make-tree '(3 5 6 4 7 4 7 9))
(9 (7 (4 (4 (3 NIL NIL) NIL)
         (7 (6 (5 NIL NIL) NIL) NIL))
      NIL)
   NIL)
```

Now, let's take an inorder traversal of that tree:

```
ACL2 !>(inorder (make-tree '(3 5 6 4 7 4 7 9)))
(3 4 4 5 6 7 7 9)
```

Interesting. It produced a sort of the list of elements. Try it for several other binary search trees, you'll see that you always get a sorted list of the elements. Let's prove at least one part of this conjecture, namely, that the result of making an inorder traversal of a binary search tree always resulsts in an ordered list, ordered in increasing order. Let's define that predicate first.

```
(defun ordered->= (L)
  (if (endp L)
      T
    (if (endp (cdr L))
        T
      (and (<= (car L) (car (cdr L)))
           (ordered->= (cdr L))))))
```

The theorem we want is basically that

```
(implies (bst? tr) (ordered->= (inorder tr)))
```

Before we can prove it, though, we need a few lemmas. And to express those lemmas, we needs a few auxiliary functions. How did I know I needed to prove those lemmas? Well, I

tried proving the above directly, it didn't work. ACL2 got stuck basically trying to prove that the result of appending the inorder traversal of the left subtree with the root and the inorder traversal of the right subtree yielded an ordered list. By the induction hypothesis, we know that the inorder traversal of the left and the right subtrees of a bst are ordered, but then you need a lemma that says that if you append two ordered lists such that every element in the first list is less than the first element of the second list, then the result is ordered. Also, we need a lemma that says that consing an element in front of an ordered list with the property that that element is less than or equal to every element of the list yields a list which is itself ordered. With two auxiliary functions `(list-all-<= a L)` that says that `a` is `<=` than every element of `L`, and `(list-all-> a L)` that says that `a` is `>` than every element of `L`, we can state and prove those lemmas.

```
(defun list-all-<= (a L)
  (if (endp L)
      T
    (and (<= (car L) a)
         (list-all-<= a (cdr L)))))

(defun list-all-> (a L )
  (if (endp L)
      T
    (and (> (car L) a)
         (list-all-> a (cdr L)))))

(defthm lemma-1
  (implies (and (ordered->= L)
                (list-all-> a L))
           (ordered->= (cons a L))))

(defthm lemma-2
  (implies (and (ordered->= L)
                (not (endp M))
                (ordered->= M)
                (list-all-<= (car M) L))
           (ordered->= (append L M))))
```

The next two lemmas make the link between the `all->` function on trees and the `list-all->` function on lists, and similarly for `all-<=` and `list-all-<=`

```
(defthm lemma-3
  (implies (all-> a tr)
           (list-all-> a (inorder tr))))
```

```
(defthm lemma-4
  (implies (all-<= a tr)
           (list-all-<= a (inorder tr))))
```

We can now prove our main theorem.

```
(defthm ordered-inorder
  (implies (bst? tr)
           (ordered->= (inorder tr))))
```

Voilà.