CS61A Notes – Disc 5: Trees, deep lists

Fake Plastic Trees

A tree is, abstractly, an acyclic, connected set of nodes (of course, that's not a very friendly definition). Usually, it is a node that contains two kinds of things – data and children. Data is whatever information may be associated with a tree, and children is a set of subtrees with this node as the parent. Concretely, it's often just a list of lists of lists of lists in Scheme, but **it's best NOT to think of trees as lists at all**. Trees are trees, lists are lists. They are completely different things, and if you, say, call `(car tree)` or something like that, we'll yell at you for violating data abstraction. `car`, `cdr`, `list` and `append` are for lists, not trees! And don't bother with box-and-pointer diagrams – they get way too complicated for trees. Just let the data abstraction hide the details from you, and trust that the procedures like `make-tree` work as intended.

Of course, that means we need our own procedures for working with trees analogous to `car`, `cdr`, etc. Different representations of trees use different procedures. You're already seen the ones for a general tree, which is one that can have any number of children (not just two) in any order (not grouped into smaller-than and larger-than). Its operators are things like:

```
(define (make-tree label children) ...)
;; takes in a datum and a LIST of trees that will be
;; the children of this tree, and returns a tree

(define (datum tree) ...)
;; returns the datum at this node

(define (children tree) ...)
;; returns a LIST of trees that are the children of the tree.
;; we call a list of trees a FOREST
```

With general trees, you'll often be working with mutual recursion. This is a common structure:

```
(define (foo-tree tree)
    ...
    (foo-forest (children tree)))

(define (foo-forest forest)
    ...
    (foo-tree (car forest))
    ...
    (foo-forest (cdr forest)))
```

Note that `foo-tree` calls `foo-forest`, and `foo-forest` calls `foo-tree`! Mutual recursion is absolutely mind-boggling if you think about it too hard. The key thing to do here is – of course – **TRUST THE RECURSION!** If, when you're writing `foo-tree`, you BELIEVE that `foo-forest` is already written, then `foo-tree` should be easy to write. Same thing applies the other way around.

QUESTIONS

In Class:

1. **Write `(square-tree tree)`, which returns the same tree structure, but with every element squared. Don't use "`map`"!**

2. **Write `(max-of-tree tree)` that does the obvious thing. The tree has at least one element.**

Extra Practice:

3. **Write `(listify-tree tree)` that turns the tree into a list in any order.**

4. **A maximum heap is a tree whose children's data are all less-than-or-equal-to the root's datum. Of course, its children are all maximum heaps as well. Write `(max-heap? tree)` that checks if a given tree is a maximum heap. You should use `max-of-tree` above.**

Binary Search Trees

A binary search tree is a special kind of tree with an interesting restriction – each node only has two children (called the "left subtree" and the "right subtree", and every node in the left subtree has datum smaller than the node of the root, and every node in the right subtree has datum larger than the node of the root. Here are some operators:

```
(define (make-tree datum left-branch right-branch) ...)
;; takes a datum, a left subtree and a right subtree and make a bst

(define (datum bst) ...)
;; returns the datum at this node

(define (left-subtree bst) ...)
;; returns the left-subtree of this bst

(define (right-subtree bst) ...)
;; returns the right-subtree of this bst
```

QUESTIONS

In Class:

1. **Write `(sum-of bst)` that takes in a binary search tree, and returns the sum of all the data in the tree.**

2. **Write `(max-of bst)` that takes in a binary search tree, and returns the maximum datum in the tree. The tree has at least one element. (Hint: This should be easy.)**

3. **Write `(remove-leaves bst)` that takes a binary search tree and returns a new binary search tree with all the leaves removed from the original.**

4. **Write `(height-of tree)` that takes in a tree and returns the height – the length of the longest path from the root to a leaf.**

Extra Practice:

5. **(HARD!) Write `(width-of tree)` that takes in a tree and returns the width – the length of the longest path from one leaf to another leaf.**

6. **Louie Reasoner was told to write `(valid-bst? bst)` that checks whether a tree satisfies the binary-search-tree property – elements in left subtree are smaller than datum, and elements in right subtree are larger than datum. He came up with this:**

```
(define (valid-bst? bst)
    (cond ((null? bst) #t)
          (else
              (and (or (null? (left-branch bst))
                       (and (< (datum (left-branch bst)) (datum bst))
                            (valid-bst? (left-branch bst))))
                   (or (null? (right-branch bst))
                       (and (> (datum (right-branch bst)) (datum bst))
                            (valid-bst? (right-branch bst)))))))))
```

Why will Louie never succeed in life? Give an example that would fool his pitiful procedure.

7. **Write `(listify bst)` that converts elements of the given `bst` into a list. The list should be in NON-DECREASING ORDER.**

Deep Lists

"Deep lists" are lists that contain sublists. You've already been working with them in the lab with `deep-reverse`, and in homeworks with `substitute2`. The key difference is that with deep-lists, you find that most of the time you will have to recurse not just on the cdr, but on both the car and the cdr, here are some exercises:

QUESTIONS

In Class:

1. Write deep-sum, that takes in a deep-list, and returns the sum of every element in the deep-list.

> (deep-sum '(1 (2 3) (4 (5) 6) (7 (8 9)))
> 45

2. Write a procedure replace-with-depth, that takes in a deep-list, and returns the same list structure, but with each element replaced by its depth in the list

> (hello (my name (is)) garply)
> (1 (2 2 (3)) 1)

Extra Practice:

3. Write deep-accumulate, that works like accumulate, but on deep-lists:

> (deep-accumulate * 1 '(1 2 (3 4) (5 (6) (7 8)))
> 40320