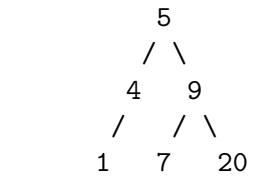
MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring 2006

Recitation 14 Solutions — 4/5/2006
Binary Search Trees

Binary Search Trees

A binary search tree is a more restricted type of tree than the general trees that appeared in lecture yesterday. A binary search tree represents a list of sorted items (we'll just use numbers today, but in general any type for which you could write a comparison function would work), and builds a data structure where you can determine if a given item is present in the set in $\Theta(logn)$ time so long as you're careful about constructing the tree (more on this later).

Each node in the tree has a left branch, a right branch, and a value. A missing branches means there is no value larger or smaller. The invariant that lets a search work is that any value anywhere in the left branch of a tree is always less than the value at the top, and the right branch is always larger.



To search for an item in a tree, you start at the root, and compare the item you're looking for to that value. If the value you want is smaller, search again in the left branch, otherwise in the right branch.

A Binary Search Tree ADT

```
;type: number,<binary-tree|null>,<binary-tree|null> -> binary-tree
(define (make-binary-tree element left-branch right-branch)
  (list 'binary-tree element left-branch right-branch))

;type: any -> boolean
(define (binary-tree? obj)
  (and (pair? obj)
       (eq? (car obj) 'binary-tree)))

;type: binary-tree -> <binary-tree|null>
(define (left-branch tree)
  (caddr tree))
```

```
;type: binary-tree -> <binary-tree|null>
(define (right-branch tree)
  (cadddr tree))

;type: binary-tree -> number
(define (tree-value tree)
  (cadr tree))
```

1. First, write functions `set-left!` and `set-right!`, which change the appropriate sub-tree:

   ```
   ;type: binary-tree, <binary-tree|null> -> unspecified
   (define (set-left! tree val)
     (set-car! (cddr tree) val))

   ;type: binary-tree, <binary-tree|null> -> unspecified
   (define (set-right! tree val)
     (set-car! (cdddr tree) val))
   ```

2. Define a procedure `tree-max` which takes a binary search tree and returns its largest value

   ```
   ;type: binary-tree -> number
   (define (tree-max tree)
   ```

3. Write a procedure `tree-search` which takes a tree and a key to search for, and returns #f if the value is not in the tree, and the value found otherwise.

   ```
   ;type binary-tree,number -> number | #f
   (define (tree-search tree key)
     (let ((v (tree-value tree)))
       (cond ((eqv? v key) v)
             ((< key v)
              (and (binary-tree? (left-branch tree))
                   (tree-search (left-branch tree) key)))
             (else
              (and (binary-tree? (right-branch tree))
                   (tree-search (right-branch tree) key))))))
   ```

4. Define a procedure `tree-insert!` which takes a tree and a new value, and adds the new value to the correct point on the tree.

   Example:

```
Add 12 to:                      results in

          8                              8
         / \                            / \
        4   10                         4  10
           /  \                           /  \
          9    15                        9   15
                                             /
                                            12
```

```
type: binary-tree, number -> unspecified
(define (tree-insert! tree val)
  (if (< val (tree-value tree))
      (cond ((not (binary-tree? (left-branch tree)))
             (set-left! tree (make-binary-tree val '() '())))
            (else
             (tree-insert! (left-branch tree) val)))
      (cond ((not (binary-tree? (right-branch tree)))
             (set-right! tree (make-binary-tree val '() '())))
            (else
             (tree-insert! (right-branch tree) val)))))
```

5. Unbalanced binary trees

   Consider the following procedure:

```
(define (fill-tree min max)
  (define (helper tree l u)
    (if (<= l u)
        (begin
          (tree-insert! tree l)
          (helper tree (+ 1 l) u))
        tree))
  (helper (make-tree min '() '())
          (+ 1 min) max))
```

   Draw the tree resulting from evaluating `(fill-tree 1 9)`. What will the running time for searches be for a tree constructed this way?

```
1
 \
  2
   \
    3
     \
      4
       \
        5
```

```
\
 6
  \
   7
    \
     8
      \
       9
```

Searches in this tree will always take linear time.

6. Modify the `tree-fold` procedure from lecture to work on binary search trees. The three operands to the combiner are the value of the top of the tree, the combined left branch value, and the combined right branch value, in that order.

```
;type (number,A,A -> A), A, <binary-tree|null> -> A
(define (tree-fold combiner init tree)
  (cond ((null? tree) init)
        (else (combiner
               (tree-value tree)
               (tree-fold combiner init (left-branch tree))
               (tree-fold combiner init (right-branch tree))))))
```

7. Write `tree-depth` using `fold-tree`. Given any binary search tree, `tree-depth` should return the distance to the most distant node, whether it is in the left of right sub-tree. The depth of a tree with a single node is 1.

```
(define (tree-depth tree)
  (tree-fold
   (lambda (v l r)
     (+ (max l r) 1))
   0
   tree))
```

8. Write a procedure `balanced-tree?` that returns #t if the tree is balances, and false otherwise. A tree is considered balanced if for every node in the tree, the depth of that nodes left and right sub-trees differs by no more than one. A balanced tree will ensure that lookups take $\Theta(logn)$ time.

   Example:

```
a balanced tree:
      5
     / \
    4   9
   /   / \
  1   7   20
```

```
an unbalanced tree:
              5
             / \
            4   9
           /   / \
          1   7   20
         /
        0
```

```
(define (balanced-tree? tree)
  (let ((ld (tree-depth (left-branch tree)))
        (rd (tree-depth (right-branch tree))))
    (if (< (abs (- ld rd)) 2)
        (and (or (null? (left-branch tree))
                 (balanced-tree?
                  (left-branch tree)))
             (or (null? (right-branch tree))
                 (balanced-tree?
                  (right-branch tree))))
        #f)))
```