Tree data structures MIT 6.001 Recitation

Binary trees

Ordered binary trees are a classic data structure to store and access data efficiently. Each node has only two branches, the left and the right one. Data is stored at each node (not only at leaf) and it is enforced that all the data below the left branch is smaller than the node value, and all the data on the right branch is bigger than the value.

Data structure

We will use a data structure where a tree node is either a list containing a number and the left and right trees (potentially empty). NOTE: I changed the abstraction compared to what we did in recitation. Now I redefine the empty tree to make it easier to insert values.

Draw a box-and-arrow diagram for the tree (5 (3 (1 4)) (7 (6 8))

```
(define (make-empty-tree) (list 'empty 'empty-left 'empty-right))
(define make-tree val left right) (list val left right))
(define (btree-left tree) (cadr tree))
(define (btree-right tree) (caddr tree))
(define (btree-null? tree) (eq? 'empty (car tree)))
(define (btree-leaf? tree) (and (btree-null? (btree-left tree))
                                (btree-null? (btree-right tree))))
(define (btree-get-value tree) (car tree))
(define (btree-set-left! tree l) (set-car! (cdr tree) l))
(define (btree-set-right! tree l) (set-car! (cddr tree) l))
(define (btree-set-value! tree v) (set-car! tree v))
```

Operations

```
(define (btree-depth tree)
  (if (btree-null? tree) 0
    (+ 1 (max (btree-depth (btree-left tree))
              (btree-depth (btree-right tree))))))

(define (is-in-btree N tree)
  (cond ((btree-null? tree) false)
        ((= N (btree-get-value tree)) true)
        ((< N (btree-get-value tree)) (is-in-btree? N (btree-left tree)))
        (else (is-in-btree? N (btree-right tree)))))
```

The time complexity grows as the depth of the tree. If we are lucky, the tree is balanced and it's logarithmic.

```
(define (btree-min btree)
  (if (btree-null? (btree-left tree)) (btree-get-value tree)
    (btree-min (btree-left tree))))
(define (btree-max btree)
  (if (btree-null? (btree-right tree)) (btree-get-value tree)
    (btree-max (btree-right tree))))

(define (btree-insert! N tree)
  (cond ((btree-null? tree) (btree-set-value! tree N)
                            (btree-set-left! tree (make-empty-tree))
                            (btree-set-right! tree (make-empty-tree))
        ((< N (btree-get-value tree)) (btree-insert! N (btree-left tree)))
        (else (btree-insert! N (btree-right tree)))))
```

Note how this is different from the solution in recitation. Since empty trees are stored explicitly, we can modify them and set their value. We also need to update them to give them empty branches.

```
(define (btree-map f tree) ;; potential problem if f is not monotonic. Oh well.
  (if (btree-null? tree) (make-empty-tree)
    (make-tree (f (btree-get-value tree))
               (btree-map f (btree-left tree))
               (btree-map f (btree-right tree)))))
```

```
(define (btree-delete N tree) ;;difficult. I mean it.
;; We are going to use two helpers. first we find the appropriate node. Then we delete it and replace it by values
below. This has to be done recursively

  (define (find-node N tree)
            ;;very similar to is-in-tree?. However, here we want to return a node, not a Boolean.
    (cond ((btree-null? tree) (error number not in tree"))
          ((= N (btree-get-value tree)) tree)
          ((< N (btree-get-value tree)) (find-node N (btree-left tree)))
          (else (find-node N (btree-right tree)))))

 (define (delete-current-node! tree)
     ;;This helper deletes the current node. For this it replaces it by values below, making sure to keep the ordered
     binary tree structure. We choose (arbitrarily) to replace the current node by the value on its right. We
     therefore need to recursively delete the node on the right.The base case is when the node on the right is
     empty. We then just copy the left subtree.
     (if (btree-null (btree-right tree)) ;; if empty right tree, just replace current by left tree
       (begin (btree-set-value! tree (btree-get-value (btree-left tree)))
              (btree-set-left! tree (btree-left (btree-left tree)))
              (btree-set-right! tree (btree-right (btree-left tree))))
         ;;note that it works even if there is no left subtree, because of the way we represent the empty tree.
       (begin (btree-set-value! tree (btree-get-value (btree-right tree)))
              (delete-current-node! (btree-right tree)))))

 ;;finally we use those two helpers to find and delete the node.
  (delete-current-node! (find-node N tree)))

(define (btree-flatten tree) ;; returns a list. Keep it simple (yet not efficient)
  (if (btree-null? tree) '()
    (append (list (btree-get-value tree))
            (btree-flatten (btree-left tree))
            (btree-flatten (btree-right tree)))))
```

Orders of growth in time?

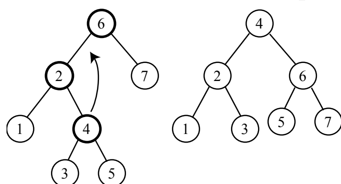
depth is linear.

is-in-tree grows as the depth of the tree. Linear in the worst case, log in the best case.

insert and delete are similar.

Balanced binary tree

A binary tree is more efficient when it is balanced, that is the depth of the two subtrees is always equal or different by one. This ensures log n performance for most operations. One way to improve the balance of a tree is to perform rotations. Look at the example below for rotate-left. Analyze it and implement a general version.



It looks scary, but all you have to do is check all the nodes that have changed and use the appropriate mutators to update things. Also, we need to make sure we don't loose pointers to subtrees. We will use a let to save the appropriate subnodes. In our example, the affected nodes are 6, 2 and 4. In general, we'll call the equivalent of 2 left, and the equivalent of 4 left-right. Of course, 6 is the input, tree.

```
(define (rotate-left! tree)
  (let ((left (btree-left tree))
        (left-right (btree-right (btree-left tree))))
    ;; first, let's move the subtrees of left-right before we lose references to them
    (btree-set-right! left (btree-left left-right))
    (btree-set-left! tree (btree-right left-right))
    ;;next, let's replace the sub-trees of left-right
    (btree-set-left! left-right left)
    (btree-set-right left-right tree)
    ;;Finally, we return the new root of the tree, left-right
    left-right)
```

We could have done things differently and only move values, between left-right and tree. The advantage is that variables pointing to tree would still have the root of the tree. In our version, they have to be updated, which is why we return the new root.

rotate-right is just the symmetric.

A tree can be balanced recursively using these procedures. In fact, the insertion and deletion should be updated to always maintain good balance. More about this in your typical algorithm class.

Tree manipulation

(define my-tree (list 1 (list 2 3 (list 4 5)) (list (list 6))))

* Draw the box & pointer diagram for my-tree. How does my-tree print? Draw a "stylized tree diagram" for my-tree

```
 (define (leaf? tree) (not (pair? tree)))
(define (tree-manip leaf-op init merge tree)
  (if (null? tree)
      init
      (if (leaf? tree)
        (leaf-op tree)
        (merge (tree-manip leaf-op init merge (car tree))
               (tree-manip leaf-op init merge (cdr tree))))))
```

* type of tree-manip:

(A→B), B, (B, B→B) → B

* use tree-manip to sum up the values of the leaves of the tree

In this case, B is integer.

```
(sum-leaves my-tree)  =>  21
(tree-manip identity 0 + my-tree)
```

* use tree manip to square every element of a tree

In this case, B is a tree of integers.

```
(square-tree my-tree)  =>  (1 (4 9 (16 25)) ((36)))
(tree-manip square '() cons my-tree)
```

* use tree-manip to compute the product of all the even-valued leaves of the tree

B is integer

```
(product-even my-tree)   =>  48
(tree-manip (lambda(x)(if (even? x) x 1)) 1 * my-tree)
```

* use tree-manip flatten a tree

B is list of A

```
(flatten my-tree)   =>   (1 2 3 4 5 6)
(tree-manip list '() append my-tree)
```

* use tree-manip to compute the depth of a tree

B is integer

```
(tree-manip 1 0
   (lambda(result-car result-cdr) (max (+ 1 result-car) result-cdr)) my-tree)
```

* use tree-manip to "deep-reverse" a tree (reverse the order of children at each node)

B is tree of A

```
  (deep-reverse my-tree)  =>  (((6)) ((5 4) 3 2) 1)
(tree-manip identity '() (lambda(result-car result-cdr)
                           (append result-cdr (list result-car))) my-tree)
```

* use tree-manip to create a new tree, which keeps the odd-valued leaves of the original tree within the same tree structure, but removes even valued leaves

B is a tree of A. Note that this is pretty much the tree equivalent of filter

```
 (keep-odd-leaves my-tree) =>  (1 (3 (5)))
  (keep-odd-leaves (list 1 2 3 4 5 6))  =>  (1 3 5)
  (keep-odd-leaves (list 2 4 6))  =>  nil
  (keep-odd-leaves (list (list 1) (list 4) (list 6))) =>  ((1))
(tree-manip (lambda(x)(if (odd? x) x '())) '()
  (lambda(result-car result-cdr)
    (if (null? result-car) result-cdr (cons result-car result-cdr)))
   my-tree)
```

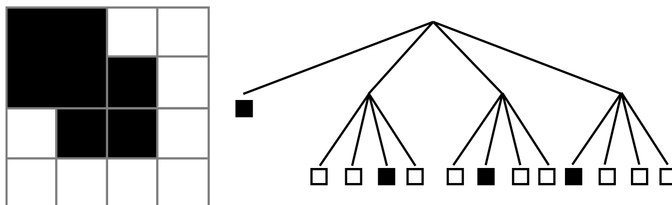
* for each of the above exercises, write down the types of arguments & return value for the calls to tree-manip

Quadtrees

Sorry, no solution for this problem!

A Quadtree is a popular data structure in computer graphics that allows for the encoding of spatial information using recursive subdivisions of a square. In this problem, we define a quadtree abstract datatype to represent black-and-white bitmap, where a pixel is either white or black.

The leaves of a quadtree are either black or white, and other nodes have exactly four children representing the four quadrants of the subdivision of the square in the order North-West, North-East, South-West, and South-East. See the example below:



We provide the following abstraction:

```
(define (black-qt) 'black)
(define (white-qt) 'white)
(define (node-qt nw ne sw se) (list nw ne sw se))

(define (qt-black? qt) (eq? qt 'black))
(define (qt-white? qt) (eq? qt 'white))
(define (qt-node? qt) (list? qt))

(define (qt-nw qt) (car qt))
(define (qt-ne qt) (cadr qt))
(define (qt-sw qt) (caddr qt))
(define (qt-se qt) (cadddr qt))
```

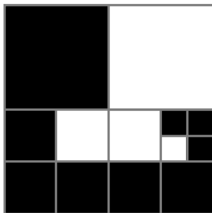
For example, the above tree can be obtained using

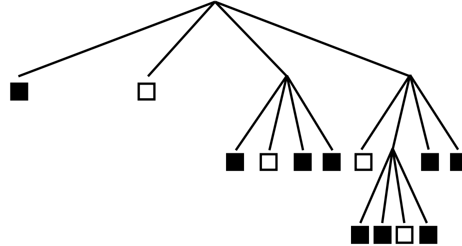
```
(define tree1
          (let ((ne (node-qt (white-qt) (white-qt) (black-qt) (white-qt)))
```

```
                                    (sw (node-qt (white-qt) (black-qt)
(white-qt) (white-qt)))
                                    (se (node-qt (black-qt) (white-qt)
(white-qt) (white-qt)))
                        (node-qt (black) ne sw se))
```

Question 1

Write the code to create the following tree :





```
(define (tree2)
        YOUR-CODE-HERE
```

Question 2: Depth-first encoding

We want to represent quadtrees in a compact and flat manner using a list composed of '*n'* to indicate a node, *'0'* for black, and *'1'* for white. More precisely, we define the *depth-first encoding* of the quadtree as:

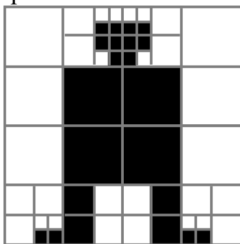
- 0 if the tree is a black leaf, and 1 if it is a white leaf
- the concatenation of 'n and the depth-order encoding of the sub-trees in the order described above (NW, NE, SW, SE) otherwise.

For this, we will write a function qt-depth-first-encode, and your job will be to fill in the code for CODE-1, CODE-2, and CODE-3. Be careful to respect the abstraction barrier.

```
(define (qt-depth-first-encode qt)
        (cond (  (qt-black? qt) CODE-1
                 CODE-2
                 (else CODE-3))))
```

Question 3: depth-first construction

Since creating quadtrees with the constructors is tedious, we want to use the above depth-first encoding to create quadtrees. Your task is to write the inverse of depth-first-encode-qt so that we can create the following tree



using

```
 (qt-depth-first-create '(n n 0 n 0 n 0 0 1 1 0 n 1 1 0 1 0 1 n n n 0 0 1 1
0 n 1 1 1 0 0 0 1 0 n 0 1 n 0 0 0 n 0 0 0 n 0 0 1 1 n 1 0 1 0 n 1 0 n 0 1 0
1 n 0 0 n 0 0 1 1 0)

(define (qt-depth-first-create ls)
   YOUR-CODE-HERE
```

Warning, this is difficult: the function has to return a pair with the quadtree and the list.

Question 4: Simplification

Some quadtrees contain redundant information when subdivision was used in regions of constant color. We want to simplify such tree into the most-compact quadtree. We will do this in two steps: first, we simplify nodes that have only leaves as children, next we will recursively simplify the full tree.

First, we want to simplify a node that has four black children or four white children. Write a function qt-conflate that takes a node and verifies if its four children are all black (resp all white), in which case it returns a black (resp. white) leaf. Otherwise, return the node unchanged.

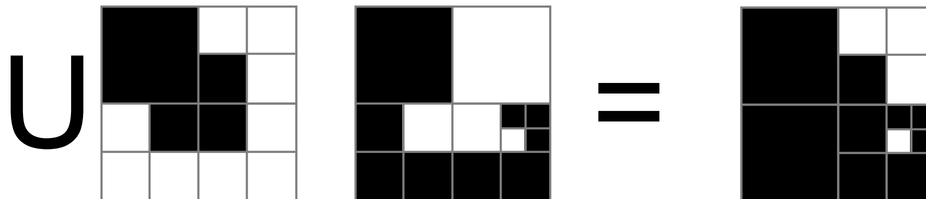
```
(define (qt-conflate n)
         YOUR-CODE
```

Next, we want to recursively simplify the tree using the qt-conflate procedure above.

```
(define (qt-simplify qt)
         (if qt-node? qt)
                   YOUR-CODE-HERE
         qt))
```

Question 5: Union

We want to take two images described by a quadtree and compute the union of the black parts. Note that the subdivision structure of the union quadtree might be simpler than that of the two sub-trees. In this case, we want the most compact quadtree. For example, the union of tree1 and tree2 is



```
(define (qt-union qt1 qt2)
         (cond
              ((qt-black? qt1) YOUR-CODE-HERE
              ((qt-white? qt1) YOUR-CODE-HERE
              ((qt-black? qt2) YOUR-CODE-HERE
              ((qt-white? qt2) YOUR-CODE-HERE
              (else
                   YOUR-CODE-HERE)
```