6.001 Recitation 13: Trees

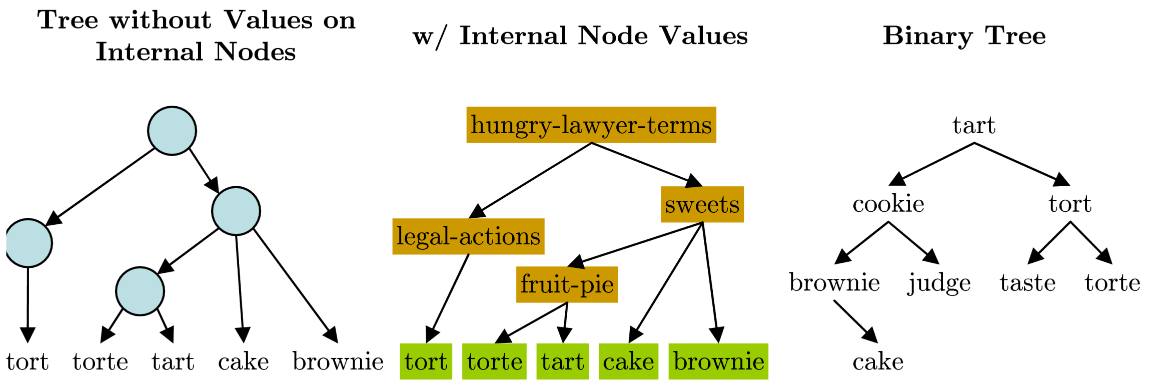
RI: Gerald Dalley, dalleyg@mit.edu, 23 Mar 2007
`http://people.csail.mit.edu/dalleyg/6.001/SP2007/`

Announcements

- Prof. Szolovits' tutorial on mutation: `http://people.csail.mit.edu/psz/6001/mutation.html`
- Google is your friend
- `(eighth lst)`

Binary Tree Data Structure

In lecture, we began talking about tree data structures. With list data structures, we think of having elements linked on to the end of other elements. Trees are a more complex recursive structure where each "element" splits into multiple sub-elements, in a way that a trunk of a tree splits into different branches, and each branch can split into smaller branches. Branches with no sub-branches are called leaves. Here are a few examples of types of trees:



Trees are useful for grouping similar items hierarchically and for optimizing sort and search behavior. One interesting special case is the binary tree: all nodes have at most two subtrees. For the binary tree shown above, the Scheme representation we'll use today is:

$$\Bigg(\bigg(\Big(()\mathit{brownie}\big(()\mathit{cake}()\big)\Big)\mathit{cookie}\Big(()\mathit{judge}()\Big)\bigg)\mathit{tart}\bigg(\Big(()\mathit{taste}()\Big)\mathit{tort}\Big(()\mathit{torte}()\Big)\bigg)\Bigg)$$

Here is some starter-code for a binary tree abstraction:

```scheme
;;;;; Special Values & Predicates
(define *empty-binary-tree*     '())
(define (empty-binary-tree? x)  (null? x))

(define (binary-tree? x)        (or (empty-binary-tree? x)
                                    (and (list? x) (= (length x) 3))))

(define (check-binary-tree x)   (if (not (binary-tree? x))
                                    (error "not␣a␣binary␣tree:" x)))

;;;;; Constructor
(define (make-node val ltree rtree)   (list ltree val rtree))

;;;;; Accessors
(define (node-value tree)      (check-binary-tree tree) (second tree))
(define (left-subtree tree)    (check-binary-tree tree) (first tree))
(define (right-subtree tree)   (check-binary-tree tree) (third tree))
```

Note: there are many possible concrete representations that can be used. Here we used one that prints out nicely, but we might on other occasions prefer to use a tagged data structure, especially if we wanted to store lists as part of the data structure.

We'll want to be able to mutate this data structure for some of the later tasks. To do this, we'll need `set-node-value!`, `set-left-subtree!`, and `set-right-subtree!`. Let's implement the first one (the solutions contain implementations of all three):

```
(define (set-node-value! tree val)
```

Tree Depth

An important feature of a tree is its depth. This is the maximum number of nodes that must be visited to traverse from the root to a leaf. When we analyze the order of growth of operations on trees, we the growth is typically tightly coupled with the depth of the tree. To get started, let's write a procedure to calculate the depth for us (hint: assume `max` is a built-in procedure):

```
(define (binary-tree/depth0 tree)
```

If we look at how `binary-tree/depth0` is implemented, we can see that the structure of the procedure is very similar to `fold-right` that operates on lists, where we have some initial value, and some procedure that merges the results. Let's write:

```
(define (binary-tree/fold op init tree)
  ; binary-tree/fold : (A,B,B)->B, B, binary-tree<A> -> B
  ;   op takes a node value and the accumulated results of the node's left
  ;   and right subtrees and creates a new accumulated result.
```

Using `binary-tree/fold`, let's rewrite `binary-tree/depth`:

```
(define (binary-tree/depth tree)
```

Binary Search Trees

An especially useful type of binary tree is a "binary search tree." This is a binary tree that contains items that have at least a strict partial order. Essentially this means that, for some less-than predicate $<$, if $A \not< B$ and $B \not< A$, then it's okay to think $A \equiv B$. Real numbers and the standard $<$ satisfy this condition.

Let's build on the `binary-tree` abstraction to build a binary search tree. The most important procedure we'll need is an insertion operator that takes an existing binary search tree and inserts a new element. Here's one way to do it:

```
(define (bst/insert val tree)
  ; Space: Theta(1) ... Theta(log n) ... Theta(n)
  ; Time:  Theta(1) ... Theta(log n) ... Theta(n)

  ; note: we aren't checking to make sure it's a binary *search* tree
  ;       just to keep the discussion simpler.
  (check-binary-tree tree)
  (if (empty-binary-tree? tree)
      (make-node val *empty-binary-tree* *empty-binary-tree*)
      (let ((curr-val (node-value tree))
            (left     (left-subtree tree))
            (right    (right-subtree tree)))
        (if (< val (node-value tree))
            (make-node curr-val
                       (bst/insert val left)
                       right)
            (make-node curr-val
                       left
                       (bst/insert val right))))))
```

If we assume we have a balanced tree (all leaves are at the same level), what is the order of growth in time $\boxed{}$ and space $\boxed{}$ (counting deferred operations)? What's the best case time $\boxed{}$ and space $\boxed{}$? Worst case time $\boxed{}$ and space $\boxed{}$?
This version of insert isn't very efficient because it needs to keep rebuilding paths through the tree. Let's write a mutating version that is more space-efficient:

```
(define (bst/insert! val tree)
```

What's the space complexity here $\boxed{}$?

Conversions

It should now be pretty easy to write a procedure to convert a list to a binary search tree (assume we have an iterative `fold-left` and `bst/insert!`:

```
(define (list->bst lst)
```

What's the best-case order of growth in time $\boxed{}$ and space $\boxed{}$? How about the worst case in time $\boxed{}$ and space $\boxed{}$?
Converting from a binary tree to a list can be done in several ways. There's a really short version that uses `binary-tree/fold` and `append!`:

```
(define (binary-tree->list tree)
  ; compact recursive version
  (binary-tree/fold (lambda (x l r) (append! l (list x) r))
                    '()
                    tree))
```

Let's make a more efficient version that stops wasting time with `append!`:

```
(define (binary-tree->list tree)
```

Now, using these two procedures, we can easily write a reasonably efficient sort operation for lists of numbers:

```
(define (sort-via-bst lst)
```

Solutions

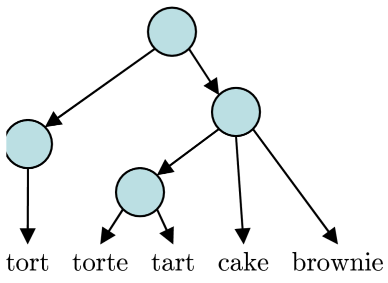
Announcements

- Prof. Szolovits' tutorial on mutation: `http://people.csail.mit.edu/psz/6001/mutation.html`
- Google is your friend
- `(eighth lst)`

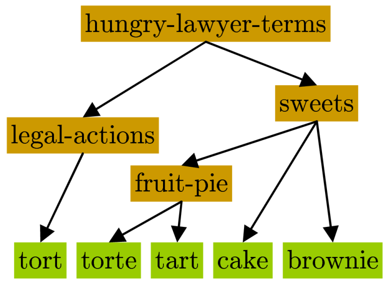
Binary Tree Data Structure

In lecture, we began talking about tree data structures. With list data structures, we think of having elements linked on to the end of other elements. Trees are a more complex recursive structure where each "element" splits into multiple sub-elements, in a way that a trunk of a tree splits into different branches, and each branch can split into smaller branches. Branches with no sub-branches are called leaves. Here are a few examples of types of trees:

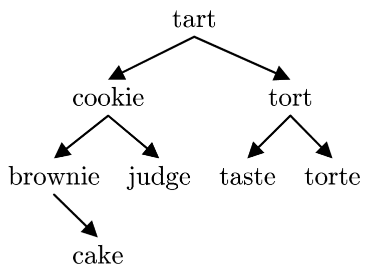
Tree without Values on Internal Nodes



w/ Internal Node Values



Binary Tree



Trees are useful for grouping similar items hierarchically and for optimizing sort and search behavior. One interesting special case is the binary tree: all nodes have at most two subtrees. For the binary tree shown above, the Scheme representation we'll use today is:

$$\Bigg(\bigg(\Big(()\mathit{brownie}\big(()\mathit{cake}()\big)\Big)\mathit{cookie}\Big(()\mathit{judge}()\Big)\bigg)\mathit{tart}\bigg(\Big(()\mathit{taste}()\Big)\mathit{tort}\Big(()\mathit{torte}()\Big)\bigg)\Bigg)$$

Here is some starter-code for a binary tree abstraction:

```scheme
;;;;; Special Values & Predicates
(define *empty-binary-tree*     '())
(define (empty-binary-tree? x)  (null? x))

(define (binary-tree? x)        (or (empty-binary-tree? x)
                                    (and (list? x) (= (length x) 3))))

(define (check-binary-tree x)   (if (not (binary-tree? x))
                                    (error "not␣a␣binary␣tree:" x)))

;;;;; Constructor
(define (make-node val ltree rtree)   (list ltree val rtree))

;;;;; Accessors
(define (node-value tree)     (check-binary-tree tree) (second tree))
(define (left-subtree tree)   (check-binary-tree tree) (first tree))
(define (right-subtree tree)  (check-binary-tree tree) (third tree))
```

Note: there are many possible concrete representations that can be used. Here we used one that prints out nicely, but we might on other occasions prefer to use a tagged data structure, especially if we wanted to store lists as part of the data structure.

We'll want to be able to mutate this data structure for some of the later tasks. To do this, we'll need `set-node-value!`, `set-left-subtree!`, and `set-right-subtree!`. Let's implement the first one (the solutions contain implementations of all three):

```
(define (set-node-value! tree val)
  (check-binary-tree tree)
  (set-car! (cdr tree) val) ; Note: cannot use set! (think why)
  tree) ; it's good to return a useful value

(define (set-left-subtree! tree ltree)
  (check-binary-tree tree)
  (set-car! tree ltree)
  tree)

(define (set-right-subtree! tree rtree)
  (check-binary-tree tree)
  (set-car! (cddr tree) rtree)
  tree)
```

Tree Depth

An important feature of a tree is its depth. This is the maximum number of nodes that must be visited to traverse from the root to a leaf. When we analyze the order of growth of operations on trees, we the growth is typically tightly coupled with the depth of the tree. To get started, let's write a procedure to calculate the depth for us (hint: assume `max` is a built-in procedure):

```
(define (binary-tree/depth0 tree)
  (check-binary-tree tree)
  (if (empty-binary-tree? tree)
      0
      (+ 1 (max (binary-tree/depth0 (left-subtree tree))
                (binary-tree/depth0 (right-subtree tree))))))
```

If we look at how `binary-tree/depth0` is implemented, we can see that the structure of the procedure is very similar to `fold-right` that operates on lists, where we have some initial value, and some procedure that merges the results. Let's write:

```
(define (binary-tree/fold op init tree)
  ; binary-tree/fold : (A,B,B)->B, B, binary-tree<A> -> B
  ;   op takes a node value and the accumulated results of the node's left
  ;   and right subtrees and creates a new accumulated result.
  (check-binary-tree tree)
  (if (empty-binary-tree? tree)
      init
      (op (node-value tree)
          (binary-tree/fold op init (left-subtree tree))
          (binary-tree/fold op init (right-subtree tree)))))
```

Using `binary-tree/fold`, let's rewrite `binary-tree/depth`:

```
(define (binary-tree/depth tree)
  (binary-tree/fold (lambda (curr-val lresult rresult)
                      (+ 1 (max lresult rresult)))
                    0
                    tree))
```

Binary Search Trees

An especially useful type of binary tree is a "binary search tree." This is a binary tree that contains items that have at least a strict partial order. Essentially this means that, for some less-than predicate $<$, if $A \not< B$ and $B \not< A$, then it's okay to think $A \equiv B$. Real numbers and the standard $<$ satisfy this condition.

Let's build on the `binary-tree` abstraction to build a binary search tree. The most important procedure we'll need is an insertion operator that takes an existing binary search tree and inserts a new element. Here's one way to do it:

```
(define (bst/insert val tree)
  ; Space: Theta(1) ... Theta(log n) ... Theta(n)
  ; Time:  Theta(1) ... Theta(log n) ... Theta(n)

  ; note: we aren't checking to make sure it's a binary *search* tree
  ;       just to keep the discussion simpler.
  (check-binary-tree tree)
  (if (empty-binary-tree? tree)
      (make-node val *empty-binary-tree* *empty-binary-tree*)
      (let ((curr-val (node-value tree))
            (left     (left-subtree tree))
            (right    (right-subtree tree)))
        (if (< val (node-value tree))
            (make-node curr-val
                       (bst/insert val left)
                       right)
            (make-node curr-val
                       left
                       (bst/insert val right))))))
```

If we assume we have a balanced tree (all leaves are at the same level), what is the order of growth in time $\boxed{\Theta(\log n)}$ and space $\boxed{\Theta(\log n)}$ (counting deferred operations)? What's the best case time $\boxed{\Theta(1)}$ and space $\boxed{\Theta(1)}$? Worst case time $\boxed{\Theta(n)}$ and space $\boxed{\Theta(n)}$?
This version of insert isn't very efficient because it needs to keep rebuilding paths through the tree. Let's write a mutating version that is more space-efficient:

```
(define (bst/insert! val tree)
  ; Space: Theta(log n) if balanced, up to Theta(n) otherwise
  ; Time : Theta(log n) if balanced, up to Theta(n) otherwise
  ;
  ; Recursive version (deferred set-left-subtree!)
  (check-binary-tree tree)
  (cond ((empty-binary-tree? tree)
         (make-node val *empty-binary-tree* *empty-binary-tree*))
        ((< val (node-value tree))
         (set-left-subtree! tree
                            (bst/insert! val (left-subtree tree))))
        (else
         (set-right-subtree! tree
                             (bst/insert! val (right-subtree tree))))))


; below are a set of increasingly "better" solutions that try to
; use iteration and do so efficiently.


(define (bst/insert! val tree)
  ; Space: Theta(log n) if balanced, up to Theta(n) otherwise
  ; Time : Theta(log n) if balanced, up to Theta(n) otherwise
  ;
  ; looks iterative, but isn't (the tree return values count as deferred operations)
  (cond ((not (binary-tree? tree))
         (error "not␣a␣tree:" tree))
        ((empty-binary-tree? tree)
         (make-node val *empty-binary-tree* *empty-binary-tree*))

        ((< val (node-value tree))
         (if (empty-binary-tree? (left-subtree tree))
             (set-left-subtree! tree
                                (bst/insert! val (left-subtree tree)))
             (begin
               (bst/insert! val (left-subtree tree))
               tree)))

        (else
         (if (empty-binary-tree? (right-subtree tree))
             (set-right-subtree! tree
                                 (bst/insert! val (right-subtree tree)))
             (begin
               (bst/insert! val (right-subtree tree))
               tree)))))


(define (bst/insert! val tree)
  ; Space: Theta(1) always
  ; Time : Theta(log n) if balanced, up to Theta(n) otherwise
  ;
  ; Okay, this one actually is iterative.  We create a handle: a cons cell that
  ; has a dummy car part and whose cdr part points to the subtree we're handling.
  ; We use the handle to simulate pass-by-reference (note: this is actually quite
  ; similar to the way C simulates pass-by-reference by using pointers).
  (define (iter handle)
    (cond
      ; Here it's easiest to handle the error condition first
      ((not (binary-tree? (cdr handle)))
       (error "not␣a␣tree:" (cdr handle)))

      ; special base case: create a new node if the original tree was empty.  Note:
      ; the handle will never be null (it's an extra cons cell that was tacked onto
      ; our tree), so it's always safe to set-cdr! to it.
      ((empty-binary-tree? (cdr handle))
       (set-cdr! handle (make-node val *empty-binary-tree* *empty-binary-tree*)))

      ; Need to go down the left subtree?
      ((< val (node-value (cdr handle)))
       (if (empty-binary-tree? (left-subtree (cdr handle)))
           ; we're about to walk off the tree to the left, so stop early and
           ; attach a new node
           (set-left-subtree! (cdr handle)
                              (make-node val *empty-binary-tree* *empty-binary-tree*))
           ; create a new handle and step down the left subtree
           (iter (cons '() (left-subtree (cdr handle))))))

      ; right subtree case looks similar to the left subtree...
      (else
       (if (empty-binary-tree? (right-subtree (cdr handle)))
           (set-right-subtree! (cdr handle)
                               (make-node val *empty-binary-tree* *empty-binary-tree*))
           (iter (cons '() (right-subtree (cdr handle))))))))

  ; First, create and keep a handle to the root of our tree.  This allows us to
  ; handle null trees
  (let ((handle (cons '() tree)))
    ; Do the mutating insertion.  Its return value is undefined.
    (iter handle)
    (cdr handle)))


(define (bst/insert! val tree)
  ; Space: Theta(1) always
  ; Time : Theta(log n) if balanced, up to Theta(n) otherwise
  ;
  ; Here's another iterative version.  In this case, we avoid using the handle
  ; by handling the special empty tree case outside of the iterator.
  (define (iter tree)
    (cond
      ((not (binary-tree? tree))
       (error "not␣a␣tree:" tree))

      ((< val (node-value tree))
       (if (empty-binary-tree? (left-subtree tree))
           (set-left-subtree! tree
                              (make-node val *empty-binary-tree* *empty-binary-tree*))
           (iter (left-subtree tree))))

      (else
       (if (empty-binary-tree? (right-subtree tree))
           (set-right-subtree! tree
                               (make-node val *empty-binary-tree* *empty-binary-tree*))
           (iter (right-subtree tree))))))

  (if (empty-binary-tree? tree)
      ; keep iter simple by handling the difficult case of any empty tree out here
      (make-node val *empty-binary-tree* *empty-binary-tree*)
      (begin
        ; otherwise, do the simplified iteration
        (iter tree)
        ; and don't forget to return the final result since iter returns junk (not
        ; worrying about the return value is what made changing to an iterative
        ; procedure easier than it would have been)
        tree)))
```

What's the space complexity here $\boxed{\Theta(1)}$?

Conversions

It should now be pretty easy to write a procedure to convert a list to a binary search tree (assume we have an iterative `fold-left` and `bst/insert!`:

```
(define (list->bst lst)
  ; Space: Theta(n)
  ; Time:  Theta(n log n) ... Theta(n^2)
  ;
  ; Q: is there any danger in using bst/insert!?
  ; A: Nope! We're just clobbering our own internal binary-tree.
  (fold-left bst/insert! *empty-binary-tree* lst))
```

What's the best-case order of growth in time $\boxed{\Theta(n \log n)}$ and space $\boxed{\Theta(n)}$? How about the worst case in time $\boxed{\Theta(n^2)}$ and space $\boxed{\Theta(n)}$?

Converting from a binary tree to a list can be done in several ways. There's a really short version that uses `binary-tree/fold` and `append!`:

```
(define (binary-tree->list tree)
  ; compact recursive version
  (binary-tree/fold (lambda (x l r) (append! l (list x) r))
                    '()
                    tree))
```

Let's make a more efficient version that stops wasting time with `append!`:

```
(define (binary-tree->list tree)
  ; efficient version
  ; Space: Theta(n)
  ; Time:  Theta(n)
  (define (iter tail node)
    (check-binary-tree node)
    (if (empty-binary-tree? node)
        tail
        (begin
          (set! tail (iter tail (left-subtree node)))
          (set-cdr! tail (list (node-value node)))
          (iter (cdr tail) (right-subtree node)))))
  (let ((answer-handle (list '())))
    (iter answer-handle tree)
    (cdr answer-handle)))
```

Now, using these two procedures, we can easily write a reasonably efficient sort operation for lists of numbers:

```
(define (sort-via-bst lst)
  ; Space: Theta(n)
  ; Time:  Theta(n log n) ... Theta(n^2)
  (binary-tree->list (list->bst lst)))
```

Test Driver

Here's the code used to test our implementation:

```
(load "helpers.scm")

; this version uses non-tagged representation for nicer printing

(load "bst-def.scm")

;;;;; Primitive Mutators

(load "primitive-mutators.scm")

;;;;; Higher-level operations

(load "depth0.scm")
(load "fold.scm")
(load "depth.scm")

(define (check-depth tree expected-depth)
  (check= (binary-tree/depth0 tree) expected-depth)
  (check= (binary-tree/depth  tree) expected-depth))

(define lr-tree
  (make-node 'top
             (make-node 'l *empty-binary-tree* *empty-binary-tree*)
             (make-node 'r
                        (make-node 'rl *empty-binary-tree* *empty-binary-tree*)
                        *empty-binary-tree*)))
(check-depth lr-tree 3)
(check-depth '(() 1 (() 2 (() 3 (() 4 ())))) 4)

;;;;; Binary Search Tree

(load "bst-insert.scm")
(load "bst-insert!.scm")
(load "list-bst.scm")

(list->bst '(1 2 3 4 5))
(list->bst '(5 3 1 2 7 6 9))

(load "bt-list.scm")
; we could write an efficient version with binary-tree/fold too, but
; the implementation gets a bit involved.
```

```
(load "sort.scm")

(define (check-sort lst)
  (let ((built-in (sort         lst <))
        (ours     (sort-via-bst lst  )))
    (check-equal ours built-in)
    ours))

(check-sort '(1 2 3 4 5))
(check-sort '(5 4 3 2 1))
(check-sort '(5 3 1 2 7 6 9))
(check-sort '())
```