Topic 11
Sets and their Representation

2.3.3

October 2008

Representing sets

Another data abstraction – here the representation choice is no so obvious. Trade-offs of different choices can be seen.

Set – collection of distinct objects. How define?

Set operations:

- union-set---union of two sets
- intersection-set---intersection of two sets
- element-of-set?---test membership in a set
- adjoin-set---add an element to a set

Sets as unordered lists (without repetition)

```
; takes an element and a set and is #t
; if element is in set
(define (element-of-set? element set)
  (cond ((null? set) #f)
        ((equal? element (car set)) #t)
        (else
            (element-of-set? element
                             (cdr set)))))
```


Adding an element to a set

```
; adds element to set
(define (adjoin-set element set)
  (if (element-of-set? element
                        set)
      set
      (cons element set)))
```


Intersection

```
; intersects set1 and set2
(define (intersection-set set1 set2)
  (cond ((or (null? set1)
             (null? set2)) ())
        ((element-of-set? (car set1)
                          set2)
         (cons (car set1)
               (intersection-set
                  (cdr set1)
                  set2)))
         (else (intersection-set
               (cdr set1)
               set2))))
```


Union

```
; returns a set that is the union of set1 and set2
(define (union-set set1 set2)
 (cond ((null? set1) set2)
       ((element-of-set? (car set1) set2)
        (union-set (cdr set1) set2))
       (else
        (cons (car set1)
              (union-set (cdr set1) set2)))))
```

Orders of growth for this representation

- `element-of-set? --- θ(n)`
- `adjoin-set --- θ(n)`
- `intersection-set --- θ(n²)`
- `union-set --- θ(n²)`

```
Could speed some of these operations if
  we change the representation of set.
Try a representation where set elements
  listed in increasing order.
```


Sets as ordered lists (of numbers, ascending order)

```
; Advantage is that now this operation
; can be written more efficiently
; returns #t if element is in the
; ordered set of numbers
(define element-of-set? element set)
  (cond ((null? set) #f)
        ((= element (car set)) #t)
        ((< element (car set)) #f)
        (else (element-of-set?
                element
                (cdr set)))))
```


intersection-set (bigger speed-up)

```
; returns an order set that is the
; intersection of ordered set1 and set2
(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2))
         ())
        ((= (car set1) (car set2))
         (cons (car set1)
               (intersection-set
                 (cdr set1)
                 (cdr set2))))
```


(continued)

```
((< (car set1) (car set2))
   (intersection-set (cdr set1)
                     set2))
  (else
    (intersection-set set1
                      (cdr
                       set2)))))
```


Orders of growth

- All four operations have order of growth equal to θ(n)
- **Operations `element-of-set?` and `adjoin-set` have been speeded up by a factor of 2**

We can do even better!

- Arrange set elements in the form of an ordered binary tree.

Binary tree

- Entry – element at that spot
- Left subtree – all elements are smaller than entry
- Right subtree – all elements are greater than entry

Notice: more than one representation for any list

- {1, 2, 4, 5, 6, 8, 10}
- (5 (2 (1 () ()) (4 () ())) (8 (6 () ()) (10 () ()))
- (2 (1 () ()) (4 () (8 (6 (5 () ())) () (10 () ()))))
- (4 (2 () ()) (6 (5 () ()) (8 () 10)
- (4 (2 (1 () ())) (5 () (6 () (8 () (10 () ())))))

Sets as (labeled) binary trees

```
; we can represent binary trees as lists
; make a tree from an entry and a left
; and right child
(define (make-tree entry
                   left-child
                   right-child)
  (list entry left-child right-child))

; selectors for a tree
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree) (caddr tree))
```


element of set

```
; takes an element and a set represented
; as a binary tree - returns #t if element
; is in set
(define (element-of-set? element set)
  (cond ((null? set) #f)
        ((= element (entry set)) #t)
        ((< element (entry set))
         (element-of-set? element
                          (left-branch set)))
        (else
          (element-of-set?
            element
            (right-branch set)))))
```


adjoin set

```
; takes an element and a set represented as
; a binary tree.  Adds element into the set
(define (adjoin-set element set)
  (cond ((null? set)
         (make-tree element () ()))
        ((= element (entry set)) set)
        ((< element (entry set))
         (make-tree (entry set)
                    (adjoin-set
                      element
                      (left-branch set))
                    (right-branch set)))
```


(continued)

```
        (else
          (make-tree
            (entry set)
            (left-branch set)
            (adjoin-set
              element
              (right-branch set))))))
```


Properties of tree represention

- If the trees are kept balanced, order of growth of `element-of-set?` **and** `adjoin-set` is θ(log n)
- Operations `intersection-set` **and** `union-set` **can be implemented to have order of growth θ(n), but the implementations are complicated**

Comparison: orders of growth (θ)

Operation	unordered	ordered	tree
element-of-set?	**n**	**n**	**log n**
adjoin-set	**n**	**n**	**log n**
intersection-set	**n^2**	**n**	**n**
union-set	**n^2**	**n**	**n**