MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1996

Lecture Notes – October 3, 1996

Alternative Representations for Data Structures

We are going to look at three different ways of representing the same data object, namely a **set**.

Informally, a set of just collection of distinct elements.

Operationally, a set is defined by four operations:

To compare sets of elements, we will assume some general comparison predicates: `equal?`, `less-than?`, `greater-than?`

Our first representation is as an unordered list. Here are the basic procedures for such as representation:

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set)))))
(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set)))
(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))))
(define (union-set set1 set2)
  (cond ((null? s1) s2)
        ((null? s2) s1)
        ((element-of-set? (car s1) s2)
         (union-set (cdr s1) s2))
        (else (cons (car s1)
                    (union-set (cdr s1) s2)))))
```

Orders of growth:

Procedure	Order of growth
Element	
Adjoin	
Intersect	
Union	

Our second method uses ordered lists:

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        ((less-than? x (car set)) false)
        (else (element-of-set? x (cdr set)))))
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((equal? x (car set)) set)
        ((less-than? x (car set))
         (cons x set))
        (else (cons (car set) (adjoin-set x (cdr set))))))
(define (intersection-set set1 set2)
  (if (or (null? set1) (null? set2))
      '()
      (let ((x1 (car set1)) (x2 (car set2)))
        (cond ((equal? x1 x2)
               (cons x1
                     (intersection-set (cdr set1)
                                       (cdr set2))))
              ((less-than? x1 x2)
               (intersection-set (cdr set1) set2))
              ((less-than? x2 x1)
               (intersection-set set1 (cdr set2)))))))
(define (union-set s1 s2)
  (cond ((null? s1) s2)
        ((null? s2) s1)
        (else (let ((x1 (car s1))
                    (x2 (car s2)))
                (cond ((equal? x1 x2)
                       (cons x1 (union (cdr s1) (cdr s2))))
                      ((less-than? x1 x2)
                       (cons x1 (union (cdr s1) s2)))
                      ((greater-than? x1 x2)
                       (cons x2 (union s2 (cdr s2))))))))))
```

Orders of growth:

Procedure	Order of growth
Element	
Adjoin	
Intersect	
Union	

Our third method uses binary trees:

Here is an example of a binary tree

And we can build a representation for a tree using

```
(define (entry tree) (car tree))

(define (left-branch tree) (cadr tree))

(define (right-branch tree) (caddr tree))

(define (make-tree entry left right)
  (list entry left right))
```

Now our procedures become:

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (entry set)) true)
        ((less-than? x (entry set))
         (element-of-set? x (left-branch set)))
        ((greater-than? x (entry set))
         (element-of-set? x (right-branch set)))))
(define (adjoin-set x set)
  (cond ((null? set) (make-tree x '() '()))
        ((equal? x (entry set)) set)
        ((less-than? x (entry set))
         (make-tree (entry set)
                    (adjoin-set x (left-branch set))
                    (right-branch set)))
        ((greater-than? x (entry set))
         (make-tree (entry set)
                    (left-branch set)
                    (adjoin-set x (right-branch set))))))
(define (union-set s1 s2)
  (cond ((null? s1) s2)
        ((element-of-set? (entry s1) s2)
         (union-set (left-branch s1)
                    (union-set (right-branch s1)
                               s2)))
        (else (adjoin-set (entry s1)
                          (union-set (left-branch s1)
                                     (union-set (right-branch s1)
                                                s2))))))
```

```
(define (intersect-set s1 s2)
  (cond ((or (null? s1) (null? s2)) '())
        ((element-of-set? (entry s1) s2)
         (adjoint-set (entry s1)
                      (union-set (intersect-set (left-branch s1)
                                                s2)
                                 (intersect-set (right-branch s1)
                                                s2))))
        (else (union-set (intersect-set (left-branch s1)
                                        s2)
                         (intersect-set (right-branch s1)
                                        s2)))))
```

Orders of growth:

Procedure	Order of growth
Element	
Adjoin	
Intersect	
Union	

Finally, we can look at an example of using trees to efficiently represent lists. The example is using variable length bit codes to represent characters. One nice way of doing this is with Huffman coding, which we describe at the end of the lecture.