

## Searching Methods

To *search* a data structure is to examine its elements singly until one has either found an element that has a desired property or concluded that the data structure contains no such element. For instance, one might search a vector of integers for an even element, or a vector of pairs for a pair having the string "elephant" as its cdr. Scheme's predefined `assq`, `assv`, and `assoc` procedures search association lists.

## Sequential Search

In a linear data structure -- such as a flat list, a vector, or a file -- there is an obvious algorithm for conducting a search: Start at the beginning of the data structure and traverse it, testing each element. Eventually one will either find an element that has the desired property or reach the end of the structure without finding such an element, thus conclusively proving that there is no such element. Here are a few versions of the algorithm.

```
;;; Procedure:
;;;   linear-search-list
;;; Parameters:
;;;   lst, a list
;;;   pred?, predicate
;;; Purpose:
;;;   Searches the list for a value that matches
;;;   the predicate.
;;; Produces:
;;;   A matching value, if one exists.
;;;   #f, otherwise.
;;; Preconditions:
;;;   The first parameter is a list.
;;;   The second parameter is a unary predicate.
;;; Postconditions:
;;;   If the procedure returns #f, no element of the
;;;   list matches the predicate.
;;;   If the procedure returns some other value, v, then
;;;   (1) (pred? v) returns #t and (2) v is in lst.
(define linear-search-list
  (lambda (pred? lst)
    (cond
      ; If the list is empty, no values match the predicate.
      ((null? lst) #f)
      ; If the predicate holds on the first value, use that one.
      ((pred? (car lst)) (car lst))
      ; Otherwise, look at the rest of the list
      (else (linear-search-list pred? (cdr lst))))))

;;; Procedure:
;;;   linear-search-vector
;;; Parameters:
;;;   vec, a vector
;;;   pred?, predicate
```

```

;;; Purpose:
;;; Searches the vector for a value that matches
;;; the predicate.
;;; Produces:
;;; The index of the matching value, if there is one.
;;; #f, otherwise.
;;; Preconditions:
;;; The first parameter is a vector.
;;; The second parameter is a unary predicate.
;;; Postconditions:
;;; If the procedure returns #f, no element of the
;;; vector matches the predicate.
;;; If the procedure returns some other value, i, then
;;; (pred? (vector-ref vec i)) returns #t.
(define linear-search-vector
  (lambda (pred? vec)
    ; Grab the length of the vector so that we don't have to
    ; keep recomputing it.
    (let ((len (vector-length vec)))
      ; Helper: Keeps track of the position we're looking at.
      (let kernel ((position 0)) ; Start at position 0
        (cond
         ; If we've run out of elements, give up.
         ((= position len) #f)
         ; If the current element matches, use it.
         ((pred? (vector-ref vec position)) position)
         ; Otherwise, look in the rest of the vector.
         (else (kernel (+ position 1))))))))

> (define sample (vector 1 3 5 7 8 11 13))
> (linear-search-vector even? sample)
4
> (linear-search-vector (right-section = 12) sample)
#f

```

These search procedures return #f if the search is unsuccessful. The first returns the matched value if the search is successful. The second returns returns the position in the specified vector at which the desired element can be found. There are many variants of this idea: One might, for instance, prefer to signal an error or display a diagnostic message if a search is unsuccessful. In the case of a successful search, one might simply return #t (if all that is needed is an indication of whether an element having the desired property is present in or absent from the list).

## Searching For Keyed Values

One of the most common “real-world” searching problems is that of searching a collection compound values for one which matches a particular portion of the value, known as the *key*. For example, we might search a phone book for a phone number using a person’s name as the key. As you’ve probably noted, association lists implement this kind of searching if we use the first value of a list as the key for that list.

Of course, it is also possible to make a `get-key` procedure a parameter to the search procedure.

```

;;; Procedure:
;;;   search-list-for-keyed-value
;;; Parameters:
;;;   key, a key to search for.
;;;   values, a list of compound values.
;;;   get-key, a procedure that extracts a key from a compound value.
;;; Purpose:
;;;   Finds a member of the list that has a matching key.
;;; Produces:
;;;   A matching value, if found.
;;;   #f, otherwise.
;;; Preconditions:
;;;   The get-key procedure can be applied to each element of values.
;;; Postconditions:
;;;   If the procedure returns #f, there is no value for which
;;;     (equal? key (get-key val))
;;;   holds. Otherwise, returns some value for which that holds.
(define search-list-for-keyed-value
  (lambda (key values get-key)
    (linear-search-list
     (lambda (val) (equal? key (get-key val)))
     values)))

```

## Binary Search

The linear search algorithms just described can be quite slow if the data structure to be searched is large. If one has a number of searches to carry out in the same data structure, it is often more efficient to “pre-process” the values, sorting them and transferring them to a vector, before starting those searches. The reason is that one can then use the much faster *binary search* algorithm.

Binary search is a more specialized algorithm than linear search. It requires a random-access structure, as opposed to one that offers only sequential access, and it is limited to the kind of test in which one is looking for a particular value that has a unique relative position in some ordering. For instance, one could use a binary search to look for an element equal to 12 in a vector of integers, since 12 is uniquely located between integers less than 12 and integers greater than 12; but one wouldn’t use binary search to look for an even integer, since the even integers don’t have a unique position in any natural ordering of the integers.

The idea in a binary search is to divide the sorted vector into two approximately equal parts, examining the element at the point of division to determine which of the parts must contain the value sought. Actually, there are usually three possibilities:

- (1) The element at the point of division cannot precede the value sought in the ordering that was used to sort the vector. In this case, the value sought must be in a position with a lower index than the element at the point of division (if it is present at all) -- in other words, it must be in the left half of the vector. The search procedure invokes itself recursively to search just the left half of the vector.
- (2) The value sought cannot precede the element at the point of division. In this case, the value sought must be in a higher-indexed position -- in the right half of the vector -- if it is present at all. The search procedure invokes itself recursively to search just the right half of the vector.

(3) The value sought is the element at the point of division. The search has succeeded.

There is one other way in which the recursion can terminate: If, in some recursive call, the subvector to be searched (which will be half of a half of a half of ... of the original vector) contains no elements at all, then the search obviously cannot succeed and the procedure should take the appropriate failure action.

Here, then, is the basic binary-search algorithm. The identifiers `lower-bound` and `upper-bound` denote the starting and ending positions of the part of the vector within which the value sought must lie, if it is present at all. (We use the convention that the starting and ending positions are *inclusive* in that they are positions within the vector that we must include in the search.)

```
;;; Procedure:
;;;   binary-search
;;; Parameters:
;;;   key, a key we're looking for
;;;   vec, a vector to search
;;;   get-key, a procedure of one parameter that, given a data item,
;;;     returns the key of a data item.
;;;   less-equal?, a binary predicate that tells us whether or not
;;;     one key is less-than-or-equal-to another.
;;; Produces:
;;;   The index of a value with key sought, if found.
;;;   #f, otherwise.
;;; Preconditions:
;;;   The vector is "sorted". That is,
;;;     (less-equal? (get-key (vector-ref vec i))
;;;                  (get-key (vector-ref vec (+ i 1))))
;;;     holds for all reasonable i.
;;;   The less-equal? procedure can be applied to all pairs of keys
;;;     in the vector (and to the supplied ot key)
;;; Postconditions:
;;;   If the procedure returns #f, no element of the
;;;     vector matches the predicate.
;;;   If the procedure returns some other value, i, then
;;;     the key of (vector-ref vec i) is equal to key.
(define binary-search
  (lambda (key vec get-key less-equal?)
    ; Search a portion of the vector from lower-bound to upper-bound
    (let search-portion ((lower-bound 0)
                        (upper-bound (- (vector-length vec) 1)))
      (if (<= lower-bound upper-bound)
          (let* ((midpoint (quotient (+ lower-bound upper-bound) 2))
                 (middle-element (vector-ref vec midpoint))
                 (middle-key (get-key middle-element)))
            (cond ((not (less-equal? middle-key key))
                   (search-portion lower-bound (- midpoint 1)))
                  ((not (less-equal? key middle-key))
                   (search-portion (+ midpoint 1) upper-bound))
                  (else midpoint)))
            #f))))))
```