

Searching Methods

Summary: We consider a typical problem of computing and a variety of algorithms used to solve that problem.

Contents:

- Introduction
- Sequential Search
 - Alternative Return Values
 - Searching For Keyed Values
- Binary Search
 - An Example
 - Another Example: Searching for Primes
 - Verifying That a Vector is Sorted

Introduction

To *search* a data structure is to examine its elements one-by-one until either (a) an element that has a desired property is found or (b) it can be concluded that the 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.

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. We used such a strategy for searching association lists. Here are a few alternate versions of the algorithm.

```
;;; Procedure:
;;;   sequential-search-list
;;; Parameters:
;;;   pred?, predicate
;;;   lst, a list
;;; Purpose:
;;;   Searches the list for a value that matches the predicate.
;;; Produces:
;;;   match, a value
;;; Preconditions:
;;;   pred? can be applied to all values in lst.
;;; Postconditions:
;;;   If lst contains an element for which pred? holds, match
;;;   is one such value.
;;;   If lst contains no elements for which pred? holds, match
```

```

;;; is false (#f).
(define sequential-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 (sequential-search-list pred? (cdr lst))))))

;;; Procedure:
;;; sequential-search-vector
;;; Parameters:
;;; pred?, predicate
;;; vec, a vector
;;; Purpose:
;;; Searches the vector for a value that matches the predicate.
;;; Produces:
;;; match, a value
;;; Preconditions:
;;; pred? can be applied to all elements of vec.
;;; Postconditions:
;;; If vec contains an element for which pred? holds, match
;;; is the index of one such value. That is,
;;; (pred? (vector-ref vec match)) holds.
;;; If vec contains no elements for which pred? holds, match
;;; is false (#f).
(define sequential-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))
> (sequential-search-vector even? sample)
4
> (sequential-search-vector (right-section = 12) sample)
#f

```

Alternative Return Values

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 or we might search a phone book for a person using the number as 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:
;;; match, a Scheme value
;;; #f, otherwise.
;;; Preconditions:
;;; The get-key procedure can be applied to each element of values.
;;; Postconditions:
;;; If there is no index for which
;;; (equal? key (get-key (list-ref values index)))
;;; holds then
;;; match is #f.
;;; Otherwise,
;;; match is a member of values: (member match values)
;;; (equal? key (get-key match))
(define search-list-for-keyed-value
  (lambda (key values get-key)
    (sequential-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, such as a vector, as opposed to one that offers only sequential access, such as a list. Binary search 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.

In binary search, we keep track of the vector, the value searched for, and the lower and upper bounds of the region still of interest. The key idea is to divide the region of interest of 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.

There are usually three possibilities for the relationship between the value sought and the element at the point of division.

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

(1) The value sought cannot follow the element at the point of division 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 region of interest. The search procedure invokes itself recursively to search just the left half of that region.

(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 region -- if it is present at all. The search procedure invokes itself recursively to search just the right half of the region.

There is one other way in which the recursion can terminate: If, in some recursive call, the region 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 region 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-than?, a binary predicate that tells us whether or not
;;;     one key is less-than another.
;;; Produces:
;;;   match, a number.
;;; Preconditions:
;;;   The vector is "sorted". That is,
;;;     (less-than? (get-key (vector-ref vec i))
;;;                 (get-key (vector-ref vec (+ i 1))))
;;;   holds for all reasonable i.
;;;   The less-than? procedure can be applied to all pairs of keys
;;;   in the vector (and to the supplied key)
;;;   The less-than? procedure is transitive. That is, if
;;;     (less-than? a b) and (less-than? b c) then it must
```

```

;;; be that (less-than? a c).
;;; The less-than? procedure is inclusive. If a is not less-than b
;;; and b is not less-than a, then a equals b.
;;; Postconditions:
;;; If vector contains no element whose key matches key, match is -1.
;;; If vec contains an element whose key equals key, match is the
;;; index of one such value. That is, key is
;;; (get-key (vector-ref vec match))
(define binary-search
  (lambda (key vec get-key less-than?)
    ; 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 the portion is empty
      (if (> lower-bound upper-bound)
          ; Indicate the value cannot be found
          -1
          ; Otherwise, identify middle point, element, and key
          (let* ((midpoint (quotient (+ lower-bound upper-bound) 2))
                (middle-element (vector-ref vec midpoint))
                (middle-key (get-key middle-element)))
            (cond
              ; If the middle key is too large, look in the left half
              ; of the region.
              ((less-than? key middle-key)
               (search-portion lower-bound (- midpoint 1)))
              ; If the middle key is too small, look in the right half
              ; of the region.
              ((less-than? middle-key key)
               (search-portion (+ midpoint 1) upper-bound))
              ; If the middle key is neither too large nor too small,
              ; it's just right.
              (else midpoint)))))))

```

An Example

So, how do we use binary search to search a sorted vector? It depends on what the vector contains. Let's suppose it contains a list of name/grade pairs, sorted by name. Here's one such vector

```

(define grades
  (vector (cons "Amy" 85)
          (cons "Bob" 60)
          (cons "Charlotte" 91)
          (cons "Danielle" 80)
          (cons "Devon" 85)
          (cons "Erin" 100)
          (cons "Fred" 70)
          (cons "Greg" 0)
          (cons "Heather" 50)
          (cons "Ira" 80)
          (cons "Jesse" 90)
          (cons "Karla" 85)
          (cons "Leo" 75)
          (cons "Maria" 88)
          (cons "Ned" 90)
          (cons "Otto" 55))

```

```

(cons "Paula" 56)
(cons "Quentin" 88)
(cons "Rebecca" 95)
(cons "Sam" 110)
(cons "Ted" 5)
(cons "U" 99)
(cons "Violet" 82)
(cons "Xerxes" 67)
(cons "Yvonne" 95)
(cons "Zed" 100))

```

Now, `binary-search` has four parameters: a key to search for, a vector to search, the procedure that extracts a key from each element in the vector, and the procedure used to compare keys. For this example, the vector to search will be `grades` and the name to search for will be whatever name we want. To get the name from a pair, we use `car`. To compare two names, we use `string-ci<?`.

So, to find out the index of my entry, I would write something like the following:

```

> (binary-search "Sam" grades car string-ci<?)
19
> (vector-ref grades 19)
("Sam" . 110)

```

We might even use this strategy to write a procedure that looks up grades.

```

;;; Procedure:
;;; get-grade
;;; Parameters:
;;; name, a string
;;; grades, a vector of name/grade pairs, sorted by name
;;; Purpose:
;;; Find the grade given to name.
;;; Produces:
;;; grade, a grade (or #f)
;;; Preconditions:
;;; For all reasonable i,
;;; (string-ci<? (car (vector-ref grades i)) (car (vector-ref grades (+ i 1))))
;;; Postconditions:
;;; If there exists i s.t. (equal? name (car (vector-ref grades i))),
;;; grade is (cdr (vector-ref grades i)).
;;; Otherwise, grade is #f.
(define get-grade
  (lambda (name grades)
    (let ((tmp (binary-search name grades car string-ci<?)))
      (if (= tmp -1) #f
          (cdr (vector-ref grades tmp))))))

```

Let's see it work

```

> (get-grade "Sam" grades)
110
> (get-grade "Janet" grades)
#f
> (get-grade "Amy" grades)
85
> (get-grade "Zed" grades)
100

```

Another Example: Searching for Primes

There are a number of ways to determine whether or not a value is prime. For small primes, the easiest technique is to search through a vector of known primes.

```

(define small-primes
  (vector 2 3 5 7 11 13 17 19 23 29 31 37
         41 43 47 53 59 61 67 71 73 79 83 89 97
         101 103 107 109 113 127 131 137 139 149
         151 157 163 167 173 179 181 191 193 197 199
         211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293
         307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397
         401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499
         503 509 521 523 541 547 557 563 569 571 577 587 593 599
         601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691
         701 709 719 727 733 739 743 751 757 761 769 773 787 797
         809 811 821 823 827 829 839 853 857 859 863 877 881 883 887
         907 911 919 929 937 941 947 953 967 971 977 983 991 997 ))

```

We could, of course, use a sequential search technique to look for a value in this vector. However, binary search is much more efficient. What procedure should we use for `get-key`? Well, each value is its own key, so we use `(lambda (x) x)`. The values are ordered numerically, so we use `<` for less-than.

For example,

```

> (binary-search 231 small-primes (lambda (x) x) <)
-1
> (binary-search 241 small-primes (lambda (x) x) <)
52
> (vector-length small-primes)
168

```

In procedure form, we might write

```

(define is-small-prime
  (lambda (candidate)
    (binary-search candidate small-primes (lambda (x) x) <)))

```

Now, how many recursive calls do we do in determining whether or not a candidate value is a small prime? If we were doing a linear search, we'd need to look at all 168 primes less than 1000, so approximately 168 recursive calls would be necessary. In binary search, we split the 168 into two groups of approximately 84 (one step), split one of those groups of 84 into two groups of 42 (another step), split one of those groups into two groups of 21 (another step), split one of those groups of 21 into two groups of 10 (we'll assume that we don't find the value), split 10 into 5, 5 into 2, 2 into 1, and then either find it or

don't. That's only about six recursive calls. Much better than the 168.

Now, suppose we listed another 168 or so primes. In linear search, we would now have to do 336 recursive calls. With binary search, we'd only have to do one more recursive call (splitting the 336 or so primes into two groups of 168).

This slow growth in the number of recursive calls (that is, when you double the number of elements to search, you double the number of recursive calls in sequential search, but only add one to the number of recursive calls in binary search) is one of the reasons that computer scientists love binary search.

Verifying That a Vector is Sorted

For `binary-search` to work correctly, we need to have a sorted vector. Checking that a vector is sorted will require looking at every neighboring pair of values, so it is not something we want to do every time we call `binary-search`. However, it is helpful to have such a procedure available.

```
;;; Procedure:
;;; sorted?
;;; Parameters:
;;;   vec, a vector
;;;   get-key, a procedure that extracts keys from the elements of vec
;;;   less-than?, a procedure that compares keys
;;; Purpose:
;;;   Determine if vec is sorted by key
;;; Produces:
;;;   is-sorted?, a Boolean
;;; Preconditions:
;;;   get-key should be applicable to any value in vec.
;;;   less-than? should be applicable to any two values returned by get-key.
;;; Postconditions:
;;;   If, for all reasonable i,
;;;     (less-than? (get-key (vector-ref vec i)) (get-key (vector-ref vec (+ i 1))))
;;;     then is-sorted is #t.
;;;   Otherwise,
;;;     is-sorted is #f.
(define sorted?
  (lambda (vec get-key less-than?)
    (let ((veclen (vector-length vec)))
      (letrec ((kernel (lambda (i)
                        (or (= i (- veclen 1))
                            (and (less-than? (get-key (vector-ref vec i))
                                               (get-key (vector-ref vec (+ i 1))))
                                (kernel (+ i 1)))))))
        (kernel 0))))))
```

Here are some tests for the vectors we defined earlier.

```
> (sorted? small-primes id <)
#t
> (sorted? grades car string-ci<?)
#t
```

Copyright © 2006 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.