



## Linear Search Function

11. Searching 3

The simplest technique for searching a list is to just "walk" the list, examining each element in turn until you either find a match or reject the last used cell:

```
const int MISSING = -1 ;

int LinearSearch(const int List[], int Target, int Size) {

    int Scan = 0;          // begin search at first cell

    // Continue searching while there are more entries
    // to consider and the target value has not been
    // found:
    while ( ( Scan < Size ) && (List[Scan] != Target) )
        Scan++ ;

    if ( Scan < Size )     // Target was found
        return( Scan );
    else
        return ( MISSING ); // Target was not found
}
```

## Application: a Simple Database

11. Searching 4

Consider implementing an application that will store data for a list of trips and perform lookup operations on that list (e.g., report the distance or time for a trip given a specific origin and destination).

Clearly we may use an array of `Trip` variables, as defined earlier, to store the data.

Assume that the application will use two input files, one containing the trip data as we've seen before, and a second input file that will contain the lookup commands the program is supposed to process, using the general syntax:

```
<command string><tab><tab-separated command parameters>
```

For example:

|           |                |                 |
|-----------|----------------|-----------------|
| mileage   | Knoxville, TN  | Nashville, TN   |
| time      | Tucumcari, NM  | Albuquerque, NM |
| neighbors | Birmingham, AL |                 |
| . . .     |                |                 |

## Searching a List of Structured Data

11. Searching 5

In this case, the array elements are complex structures and the search function must access the appropriate member(s) to make the comparison:

```
int reportMileage(const Trip& toFind, const Trip dB[], int numTrips) {
    int Idx;
    for (Idx = 0; Idx < numTrips; Idx++) {
        if ( (toFind.Origin == dB[Idx].Origin    &&
             toFind.Destination == dB[Idx].Destination)
            ||
            (toFind.Origin == dB[Idx].Destination    &&
             toFind.Destination == dB[Idx].Origin) ) {
            return dB[Idx].Miles;
        }
    }
    return MISSING;
}
```

The designer has determined that the direction of the trip does not matter, which complicates the Boolean test for a match somewhat. How would the implementation change if direction did matter?

Note the use of `const` in the parameter list. Is this good design? Why or why not?

## Binary Search

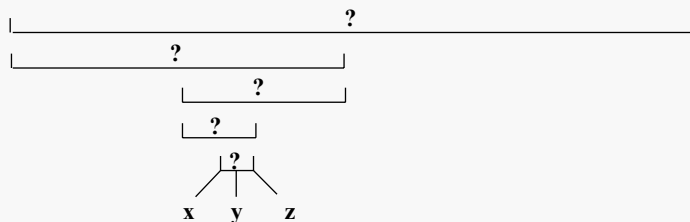
11. Searching 6

Linear search will always work, but there is an alternative that is, on average, much faster. The difficulty is that binary search may only be applied under the following:

Assumption: the list is sorted in ascending (or descending) order:

$$\text{List}[0] \leq \text{List}[1] \leq \dots \leq \text{List}[\text{Size}-1]$$

Binary search: Examine the middle element. If the target element is not found, determine to which side it falls. Divide that section in half and examine the middle element, etc, etc ...



## Binary Search Function

11. Searching 7

An implementation of binary search is more complex logically than a linear search, but the performance gain is impressive:

```

const int MISSING = -1;

int BinSearch(const int List[] , int Target, int Lo, int Hi) {
    int Mid;

    while ( Lo <= Hi ) {

        Mid = ( Lo + Hi ) / 2;           // find "middle" index

        if ( List[Mid] == Target )      // check for target
            return ( Mid );
        else if ( Target < List[Mid] )
            Hi = Mid - 1;               // look in lower half
        else
            Lo = Mid + 1;               // look in upper half
    }
    return ( MISSING );                // Target not found
}
    
```

Note this implementation allows the caller to search a specified portion of List[ ].

## Binary Search Trace

11. Searching 8

Suppose an array contains the values:

|   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 |
| A | 21 | 35 | 41 | 43 | 51 | 82 | 85 | 86 | 93 | ?? | ?? | ?? | ?? | ?? |

Consider searching A[ ] for the value 86.

What would the call look like?

Trace the execution:

|     |  |  |  |  |
|-----|--|--|--|--|
| Lo  |  |  |  |  |
| Mid |  |  |  |  |
| Hi  |  |  |  |  |

Consider searching A[ ] for the value 50.

What would the call look like?

Trace the execution:

|     |  |  |  |  |
|-----|--|--|--|--|
| Lo  |  |  |  |  |
| Mid |  |  |  |  |
| Hi  |  |  |  |  |

## Cost of Searching

11. Searching 9

Suppose the array to be searched contains  $N$  elements. The cost of searching may be measured by the number of array elements that must be compared to `Target`.

Using that measure:

|               | Best Case | Worst Case | Average Case |
|---------------|-----------|------------|--------------|
| Linear Search | 1         | $N$        | $N/2$        |
| Binary Search | 1         | $\log_2 N$ | $\log_2 N$   |

To get an idea of how much cheaper binary search is, note that

$$\begin{array}{ll} N = 1024 & \log_2 N = 10 \\ N = 1024^2 & \log_2 N = 20 \end{array}$$

Of course, binary search is only feasible if the array is sorted . . .