Simple Searching

1. Table of Contents
2. Sequential Searching
3. Probability Ordering
4. Sequential Search Code
5. Sequential Search Code (cont)
6. Binary Search
7. Interpolation Search

Sequential Searching

Unsorted List
Each element is compared to locate the desired element one after another starting at the head of the list.

Worst Case Order = $O(N)$
† desired element is at the end of the list.

Average Case Order = $O(N/2) \in O(N)$
† one half of the list must be scanned on the average.

Assumes that the probability of each element in the list being searched for is equal.

Sequential Searching on a Sorted list
Search stops when element is located or a larger element (ascending order) is encountered.

Worst case and average case orders are the same as the unordered list.

Simple Searching
- Internal (primary memory) searching

External => File Search
- (Indexes, BTrees, files, etc.)
Probability Ordering

Unequal Access Probabilities
- Implemented when a small subset of the list elements are accessed more frequently than other elements.

Static Probabilities
- When the contents of the list are static the most frequently accessed elements are stored at the beginning of the list.
  - Assumes that access probabilities are also static

Dynamic Probabilities
- For nonstatic lists or lists with dynamic probability element accesses, a dynamic element ordering scheme is required:
  - Sequential Swap Scheme
    † Move each element accessed to the start of the list if it is not within some threshold units of the head of the list.
  - Bubble Scheme
    † Swap each element accessed with the preceding element to allow elements to “bubble” to the head of the list.
  - Access Count Scheme
    † Maintain a counter for each element that is incremented anytime an element is accessed.
    † Maintain a sorted list ordered on the access counts.

Sequential Search Code

Normal Sequential Search Function
```c
const int MISSING = -1;

int SeqSearch (const Item A[], Item K, int size) {
    int i;
    for ( i = 0; i < size; i++ ) {
        if ( K == A[i] )
            return ( i );
    }
    return (MISSING);
}
```

Coded inline to avoid function call overhead:
```c
inline int SeqSearch2 (const Item A[], Item K, int size) {
    int i;
    for ( i = 0; ((i < size) && !( K == A[i] )); i++ )
        ;
    return ( ( i < size ) ? ( i ) : ( MISSING ) );
}
```

Problem: two comparisons in the loop are inefficient
Search sequentially down to 0 using 0 as limit test.
```c
const int MISSING = -1;

int SeqSearch3 (const Item A[], Item K, int size) {
    int i;
    for ( i = size -1; (!(K == A[i]) && (i)); i--);
    if ( K == A[i] )
        return ( i );
    else
        return (MISSING);
}
14. Searching

Sequential Search continued

Sentinel Method
- Store the desired element at the end of the array:

```
const int MISSING = -1;
int SeqSearch4 (Item A[], Item K, int size) {
    int i;
    A[size] = K;
    for ( i = 0; !(K == A[i]); i++ )
        ;
    if ( i < size )
        return ( i );
    else
        return ( MISSING);
}
```

- Requires storage at the end of the array to always be available.
- Ensures that the loop will terminate.
- Array parameter must be passed by reference to allow the sentinel insertion.

Binary Search

Algorithm
IF desired element = middle element of list THEN
    found
ELSE
    IF desired element < middle element
        THEN set list to lower half & repeat process
    ELSE set list to upper half & repeat process

Recursive Binary Search Function

```
const int MISSING = -1;
int BinarySearch ( const Item A[], Item K, int L, int R) {
    int Midpoint = (L+R) / 2 ; //compute midpoint
    if ( L > R ) // If search interval is empty return -1
        return MISSING ;
    else if ( K == A[Midpoint] ) //successful search
        return Midpoint;
    else if ( A[Midpoint] < K ) //search upper half
        return BinarySearch(A, K, Midpoint + 1, R);
    else //search lower half
        return BinarySearch(A, K, L, Midpoint - 1);
}
```

Worst Case Order = $O(\log_2 N)$

Note: for small lists a sequential search will usually be faster due to the midpoint computation and comparisons.

Subtle Algorithm Adjustments
- Minor changes to highly efficient algorithms (e.g., binary search) can have a drastic negative effect on execution.
- Changing the indexes to longints can increase execution time by a factor of 3.
- Using real division and truncating for the midpoint computation may slow execution by more than 10 times.
Interpolation Search

Variation of Binary Searching
- Attempts to more accurately predict where the item may fall within the list. Similar to looking up telephone numbers
- Standard Binary Search Midpoint Computation:
  \[
  \text{Midpoint} = \frac{(L+R)}{2};
  \]
- General Binary Search Midpoint Computation:
  \[
  \text{Midpoint} = L + \frac{1}{2} \times (R - L);
  \]
- Interpolation replaces the \(1/2\) (in the above formula) with an estimate of where the desired element is located in the range, based on the available values (be careful of int arithmetic):
  \[
  \text{Interp} = L + \left(\frac{(K - A[L])}{(A[R] - A[L])} \times \frac{(R - L)}{(R - L)}\right);
  \]
  where:
  \(L\) = \text{base loc} +
  \(K\) = \text{element to search for} +
  \(R\) = \text{end of the range} +
  \(A[L]\) = \text{lvalues before \(K\)} +
  \(A[R]\) = \text{lvalues after \(K\)} +
  \(L\) = \text{start of the range} +
  \(R\) = \text{end of the range} +
  \(L\) = \text{start of the range} +

- Example:
  - Assume 30K recs of SSNs in the range from 0 ... 600 00 0000
  - Searching for 222 22 2222 yields an initial estimate of:
    \[
    \text{Interpolation} = 0 + \left(\frac{(222222222 - 0)}{(600000000 - 0)} \times \frac{(30000 - 0)}{(30000 - 0)}\right);
    \]
    \[= 11111\]
  - Worst Case Order approximately = \(O(\log \log N)\)
  - Can be assumed to be a constant of about 5 since \(\approx (\log \log 10^9)\)
  - Assumes the search values are evenly distributed over the search range, ("True for SSNs")
  - Inefficient for searching small number of elements