CS 4104: Data and Algorithm Analysis

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Fall 2010

Searching

Assumptions for search problems:

- Target is well defined.
- Target is fixed.
- Search domain is finite.
- We (can) remember all information gathered during search.

We search for a record with a **key**.

A Search Model (1)

Problem:
Given:

- A list L, of n elements
- A search key X

Solve: Identify one element in L which has key value X, if any exist.

A Search Model (1)

Problem:
Given:

- A list L, of n elements
- A search key X

Solve: Identify one element in L which has key value X, if any exist.

Model:

- The key values for elements in L are unique.
- One comparison determines $<$, $=$, $>$.
- Comparison is our only way to find ordering information.
- Every comparison costs the same.

A Search Model (2)

Goal: Solve the problem using the minimum number of comparisons.

- Cost model: Number of comparisons.
- (Implication) Access to every item in *L* costs the same (array).

Is this a reasonable model and goal?

Linear Search

General algorithm strategy: Reduce the problem.

- Compare X to the first element.
- If not done, then solve the problem for $n - 1$ elements.

```
Position linear_search(L, lower, upper, X) {
  if L[lower] = X then
    return lower;
  else if lower = upper then
    return -1;
  else
    return linear_search(L, lower+1, upper, X);
}
```

What equation represents the worst case cost?

Worst Cost Upper Bound

$$f(n) = \begin{cases} 1 & n = 1 \\ f(n-1) + 1 & n > 1 \end{cases}$$

Reasonable to guess that $f(n) = n$.
Prove by induction:
Basis step: $f(1) = 1$, so $f(n) = n$ when $n = 1$.
Induction hypothesis: For $k < n$, $f(k) = k$.
Induction step: From recurrence,

$$\begin{aligned} f(n) &= f(n-1) + 1 \\ &= (n-1) + 1 \\ &= n \end{aligned}$$

Thus, the worst case cost for n elements is linear.
Induction is great for verifying a hypothesis.

Approach #2

- What if we couldn't guess a solution?
- Try: Substitute and Guess.
  - Iterate a few steps of the recurrence, and look for a summation.

$$\begin{aligned} f(n) &= f(n-1)+1 \\ &= \{f(n-2)+1\}+1 \\ &= \{\{f(n-3)+1\}+1\}+1\} \end{aligned}$$

- Now what? Guess $f(n) = f(n-i) + i$.
- When do we stop? When we reach a value for f that we know.

$$f(n) = f(n-(n-1)) + n - 1 = f(1) + n - 1 = n$$

- Now, go back and test the guess using induction.

Approach #3

Guess and Test: Guess the form of the solution, then solve the resulting equations.

Guess: $f(n)$ is linear.
$f(n) = rn + s$ for some r, s.

What do we know?

- $f(1) = r(1) + s = r + s = 1.$
- $f(n) = r(n) + s = r(n-1) + s + 1.$

Solving these two simultaneous equations, $r = 1$, $s = 0$.

Final form of guess: $f(n) = n$.

Now, prove using induction.

Lower Bound on Problem

Theorem: Lower bound (in the worst case) for the problem is n comparisons.

Proof: By contradiction.

- Assume an algorithm A exists that requires only $n - 1$ (or less) comparisons of X with elements of L.
- Since there are n elements of L, A must have avoided comparing X with $L[i]$ for some value i.
- We can feed the algorithm an input with X in position i.
- Such an input is legal in our model, so the algorithm is incorrect.

Is this proof correct?

Fixing the Proof (1)

Error #1: An algorithm need not consistently skip position i.
Fix:

- On any given run of the algorithm, *some* element i gets skipped.
- It is possible that X is in position i at that time.

Fixing the Proof (2)

Error #2: Must allow comparisons between elements of L.
Fix:

- Include the ability to “preprocess” L.
- View L as initially consisting of n “pieces.”
- A comparison can join two pieces (without involving X).
- The total of these comparisons is k.
- We must have at least $n - k$ pieces.
- A comparison of X against a piece can reject the whole piece.
- This requires $n - k$ comparisons.
- The total is still at least n comparisons.

Average Cost

How many comparisons does linear search do on average?

We must know the probability of occurrence for each possible input.

(Must X be in L?) Ignore everything except the position of X in L. Why?

What are the $n + 1$ events?

$$\mathbf{P}(X \notin L) = 1 - \sum_{i=1}^{n} \mathbf{P}(X = L[i]).$$

Average Cost Equation

Let $k_i = i$ be the number of comparisons when $X = L[i]$.
Let $k_0 = n$ be the number of comparisons when $X \notin L$.

Let p_i be the probability that $X = L[i]$.
Let p_0 be the probability that $X \notin L[i]$ for any i.

$$\begin{aligned} f(n) &= k_0 p_0 + \sum_{i=1}^{n} k_i p_i \\ &= n p_0 + \sum_{i=1}^{n} i p_i \end{aligned}$$

What happens to the equation if we assume all p_i's are equal (except p_0)?

Computation

$$
\begin{aligned}
f(n) &= p_0 n + \sum_{i=1}^{n} ip \\
&= p_0 n + p \sum_{i=1}^{n} i \\
&= p_0 n + p \frac{n(n+1)}{2} \\
&= p_0 n + \frac{1-p_0}{n} \frac{n(n+1)}{2} \\
&= \frac{n+1+p_0(n-1)}{2}
\end{aligned}
$$

Depending on the value of p_0, $\frac{n+1}{2} \leq f(n) \leq n$.

Problems with Average Cost

- Average cost is usually harder to determine than worst cost.
- We really need also to know the variance around the average.
- Our computation is only as good as our knowledge (guess) on distribution.

Sorted List

Change the model: Assume that the elements are in ascending order.

Is linear search still optimal? Why not?

Optimization: Use linear search, but test if the element is greater than X. Why?

Observation: If we look at $L[5]$ and find that X is bigger, then we rule out $L[1]$ to $L[4]$ as well.

More is Better: If we look at $L[n]$ and find that X is bigger, then we know in one test that X is not in L. Great!

- What is wrong here?

Jump Search

Algorithm:

- From the beginning of the array, start making jumps of size k, checking $L[k]$ then $L[2k]$, and so on.
- So long as X is greater, keep jumping by k.
- If X is less, then use linear search on the last sublist of k elements.

This is called Jump Search.

What is the right amount to jump?

Analysis of Jump Search

- If $mk \leq n < (m+1)k$, then the total cost is at most $m+k-1$ 3-way comparisons.

$$f(n,k) = m + k - 1 = \left\lfloor \frac{n}{k} \right\rfloor + k - 1.$$

- What should k be?

$$\min_{1 \leq k \leq n} \left\{ \left\lfloor \frac{n}{k} \right\rfloor + k - 1 \right\}$$

- Take the derivative and solve for $f'(x) = 0$ to find the minimum.
- This is a minimum when $k = \sqrt{n}$.
- What is the worst case cost?
  - Roughly $2\sqrt{n}$.

Lessons

We want to balance the work done while selecting a sublist with the work done while searching a sublist.

In general, make subproblems of equal effort.

This is an example of **divide and conquer**

What if we extend this to three levels?

- We'd jump to get a sublist, then jump to get a sub-sublist, then do sequential search
- While it might make sense to do a two-level algorithm (like jump search), it almost never makes sense to do a three-level algorithm
- Instead, we resort to recursion

Binary Search

```
int binary(int K, int* array, int left, int right) {
  // Return position of element (if any) with value K
  int l = left-1;
  int r = right+1;    // l and r beyond array bounds
  while (l+1 != r) {  // Stop when l and r meet
    int i = (l+r)/2;  // Middle of remaining subarray
    if (K < array[i]) r = i;     // In left half
    if (K == array[i]) return i; // Found it
    if (K > array[i]) l = i;     // In right half
  }
  return UNSUCCESSFUL; // Search value not in array
}
```

Worst Case for Binary Search (1)

$$f(n) = \begin{cases} 1 & n = 1 \\ f(\lfloor n/2 \rfloor) + 1 & n > 1 \end{cases}$$

Since $n/2 \geq \lfloor n/2 \rfloor$, and since $f(n)$ is assumed to be non-decreasing (why?), we can use

$$f(n) = f(n/2) + 1.$$

Alternatively, assume n is a power of 2.

Expand the recurrence:

$$\begin{aligned} f(n) &= f(n/2) + 1 \\ &= \{f(n/4) + 1\} + 1 \\ &= \{\{f(n/8) + 1\} + 1\} + 1 \end{aligned}$$

Worst Case for Binary Search (2)

Collapse to

$$f(n) = f(n/2^i) + i = \log n + 1$$

Now, prove it with induction.

$$\begin{aligned} f(n/2) + 1 &= (\log(n/2) + 1) + 1 \\ &= (\log n - 1 + 1) + 1 \\ &= \log n + 1 = f(n). \end{aligned}$$

Lower Bound (for Problem Worst Case)

How does n compare to $\sqrt{n}$ compare to $\log n$?

Can we do better?

Model an algorithm for the problem using a decision tree.

- Consider only comparisons with X.
- Branch depending on the result of comparing X with $L[i]$.
- There must be at least n leaf nodes in the tree. (Why?)
- Some path must be at least $\log n$ deep. (Why?)

Thus, binary search has optimal worst cost under this model.

Average Cost of Binary Search (1)

An estimate given these assumptions:

- X is in L.
- X is equally likely to be in any position.
- $n = 2^k$ for some non-negative integer k.

Cost?

Average Cost of Binary Search (1)

An estimate given these assumptions:

- X is in L.
- X is equally likely to be in any position.
- $n = 2^k$ for some non-negative integer k.

Cost?

- One chance to hit in one probe.
- Two chances to hit in two probes.
- 2^{i-1} to hit in i probes.
- $i \leq k$.

Average Cost of Binary Search (1)

An estimate given these assumptions:

- X is in L.
- X is equally likely to be in any position.
- $n = 2^k$ for some non-negative integer k.

Cost?

- One chance to hit in one probe.
- Two chances to hit in two probes.
- 2^{i-1} to hit in i probes.
- $i \leq k$.

What is the equation?

Average Cost (2)

$$\frac{1 \times 1 + 2 \times 2 + 3 \times 4 + ... + \log n 2^{\log n - 1}}{n} = \frac{1}{n} \sum_{i=1}^{\log n} i 2^{i-1}$$

$$\begin{aligned}
\sum_{i=1}^{k} i2^{i-1} &= \sum_{i=0}^{k-1} (i+1)2^i = \sum_{i=0}^{k-1} i2^i + \sum_{i=0}^{k-1} 2^i \\
&= 2\sum_{i=0}^{k-1} i2^{i-1} + 2^k - 1 \\
&= 2\sum_{i=1}^{k} i2^{i-1} - k2^k + 2^k - 1
\end{aligned}$$

Average Cost (3)

Now what? Subtract from the original!

$$\sum_{i=1}^{k} i2^{i-1} = k2^k - 2^k + 1 = (k-1)2^k + 1.$$

Result (1)

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{\log n} i2^{i-1} &= \frac{(\log n - 1)2^{\log n} + 1}{n}\\ &= \frac{n(\log n - 1) + 1}{n}\\ &\approx \log n - 1\end{aligned}$$

So the average cost is only about one or two comparisons less than the worst cost.

Result (2)

If we want to relax the assumption that $n = 2^k$, we get:

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \frac{\lceil \frac{n}{2} \rceil - 1}{n} f(\lceil \frac{n}{2} \rceil - 1) + \frac{1}{n} 0 + & \\ \frac{\lfloor \frac{n}{2} \rfloor}{n} f(\lfloor \frac{n}{2} \rfloor) + 1 & n > 1 \end{cases}$$

Average Cost Lower Bound

- Use decision trees again.
- **Total Path Length**: Sum of the level for each node.
- The cost of an outcome is the level of the corresponding node plus 1.
- The average cost of the algorithm is the average cost of the outcomes (total path length$/n$).
- What is the tree with the least average depth?
- This is equivalent to the tree that corresponds to binary search.
- Thus, binary search is optimal.

Changing the Model

What are factors that might make binary search either unusable or not optimal?

- We know something about the distribution.
- Data are not sorted. (Preprocessing?)
- Data sorted, but probes not all the same cost (not an array).
- Data are static, know all search requests in advance.

Interpolation Search

(Also known as Dictionary Search)

Search L at a position that is appropriate to the value of X.

$$p = \frac{X - L[1]}{L[n] - L[1]}$$

Repeat as necessary to recalculate p for future searches.

Quadratic Binary Search

This is easier to analyze:

- Compute p and examine $L[\lceil pn \rceil]$.
- If $X < L[\lceil pn \rceil]$ then sequentially probe

$$L[\lceil pn - i\sqrt{n} \rceil], i = 1, 2, 3, ...$$

  until we reach a value less than or equal to X.
- Similar for $X > L[\lceil pn \rceil]$.
- We are now within $\sqrt{n}$ positions of X.
- ASSUME (for now) that this takes a constant number of comparisons.
- Now we have a sublist of size $\sqrt{n}$.
- Repeat the process recursively.
- What is the cost?

QBS Probe Count (1)

Cost is $\Theta(\log \log n)$ IF the number of probes on jump search is constant.

Number of comparisons needed is:

$$\sum_{i=1}^{\sqrt{n}} i\mathbf{P}(\text{need exactly } i \text{ probes})$$

$$= 1\mathbf{P}_1 + 2\mathbf{P}_2 + 3\mathbf{P}_3 + \cdots + \sqrt{n}\mathbf{P}_{\sqrt{n}}$$

This is equal to:

$$\sum_{i=1}^{\sqrt{n}} \mathbf{P}(\text{need at least } i \text{ probes})$$

QBS Probe Count (2)

$$\sum_{i=1}^{\sqrt{n}} \mathbf{P}(\text{need at least } i \text{ probes})$$

$$\begin{aligned}
&= 1 + (1 - \mathbf{P}_1) + (1 - \mathbf{P}_1 - \mathbf{P}_2) + \cdots + \mathbf{P}_{\sqrt{n}} \\
&= (\mathbf{P}_1 + ... + \mathbf{P}_{\sqrt{n}}) + (\mathbf{P}_2 + ... + \mathbf{P}_{\sqrt{n}}) + \\
&\qquad (\mathbf{P}_3 + ... + \mathbf{P}_{\sqrt{n}}) + \cdots \\
&= 1\mathbf{P}_1 + 2\mathbf{P}_2 + 3\mathbf{P}_3 + \cdots + \sqrt{n}\mathbf{P}_{\sqrt{n}}
\end{aligned}$$

QBS Probe Count (3)

We require at least two probes to set the bounds, so cost is:

$$2 + \sum_{i=3}^{\sqrt{n}} \mathbf{P}(\text{need at least } i \text{ probes})$$

Useful fact (Čebyšev's Inequality):
The probability that we need probe i times ($\mathbf{P}_i$) is:

$$\mathbf{P}_i \leq \frac{p(1-p)n}{(i-2)^2 n} \leq \frac{1}{4(i-2)^2}$$

since $p(1-p) \leq 1/4$.

This assumes uniformly distributed data.

QBS Probe Count (4)

Final result:

$$2 + \sum_{i=3}^{\sqrt{n}} \frac{1}{4(i-2)^2} \approx 2.4112$$

Is this better than binary search?

What happened to our proof that binary search is optimal?

Comparison (1)

Let's compare $\log \log n$ to $\log n$.

n	$\log n$	$\log \log n$	Diff
16	4	2	2
256	8	3	2.7
$64K$	16	4	4
2^{32}	32	5	6.4

Now look at the actual comparisons used.

- Binary search $\approx \log n - 1$
- Interpolation search $\approx 2.4 \log \log n$

n	$\log n - 1$	$2.4 \log \log n$	Diff
16	3	4.8	worse
256	7	7.2	$\approx$ same
$64K$	15	9.6	1.6
2^{32}	31	12	2.6

Comparison (2)

Not done yet! This is only a count of comparisons!

- Which is more expensive: calculating the midpoint or calculating the interpolation point?

Which algorithm is dependent on good behavior by the input?

Hashing

Assume we can preprocess the data.

- How should we do it to minimize search?

Put record with key value X in $L[X]$.

If the range is too big, then use hashing.

How much can we get from this?

Simplifying assumptions:

- We hash to each slot with equal probability
- We probe to each (new) slot with equal probability
- This is called uniform hashing

Hashing Insertion Analysis (1)

Define $\alpha = N/M$ (Records stored/Table size)

Insertion cost: sum of costs times probabilities for looking at 1, 2, ..., $N + 1$ slots

- Probability of collision on insertion? $\alpha = N/M$
- Probability of initial collision and another collision when probing? α^2

$$\sum_{i=0}^{i=N} i(\frac{N}{M})^i \frac{M-N}{M} = \sum_{i=0}^{i=N} i\alpha^i(1-\alpha)$$

Hashing Insertion Analysis (2)

Simpler formulation: Always look at least once, look at least twice with probability α, look at least three times with probability α^2, etc.

$$\sum_{i=0}^{\infty} \alpha^i = 1 + \alpha + \alpha^2 \cdots = \frac{1}{1-\alpha}$$

How does this grow?

Searching Linked Lists

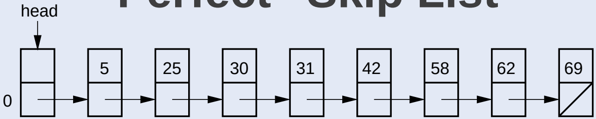
Assume the list is sorted, but is stored in a linked list.

Can we use binary search?

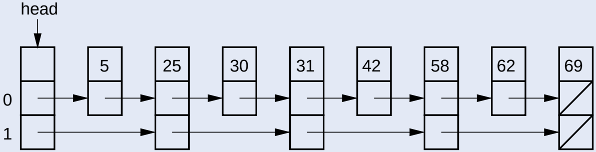
- Comparisons?
- “Work?”

What if we add additional pointers?

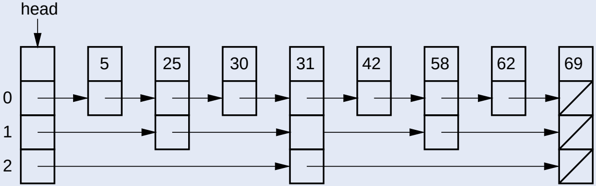
"Perfect" Skip List



(a)



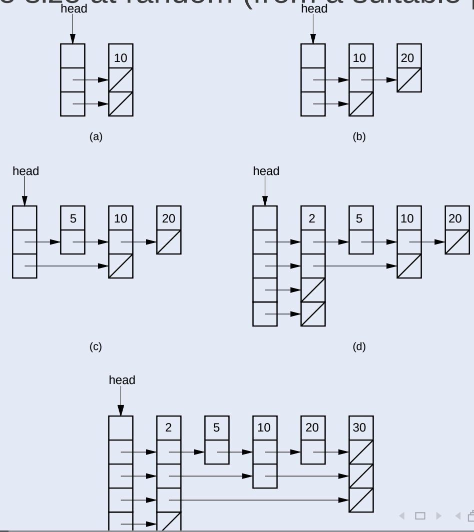
(b)



(c)

Building a Skip List

Pick the node size at random (from a suitable probability distribution).



Skip List Analysis (1)

What distribution do we want for the node depths?

```
int randomLevel(void) { // Exponential distrib
  for (int level=0; Random(2) == 0; level++);
  return level;
}
```

What is the worst cost to search in the “perfect” Skip List?

What is the average cost to search in the “perfect” Skip List?

What is the cost to insert?

What is the average cost in the “typical” Skip List?

Skip List Analysis (2)

How does this differ from a BST?

- Simpler or more complex?
- More or less efficient?
- Which relies on data distribution, which on basic laws of probability?

Other Types of Search

- Nearest neighbor (if X not in L).
- Exact Match Query.
- Range query.
- Multi-dimensional search.
- Is L static?

Is linear search on a sorted list ever better than binary search?