

We will consider searching under the comparison model

Binary tree –  $\lg n$  upper and lower bounds  
This also holds in “average case”  
Updates also in  $O(\lg n)$

Linear search – worst case  $n$   
If all elements have equal probability of search, expected time is  $(n+1)/2$  in  
successful case ( $n$  for unsuccessful search)

Real Questions:

1. What if probabilities of access differ? How well can we do under stochastic model?  
(Easiest example:  $p_i$  is probability of requesting element  $i$ , probabilities are independent)
2. What if these (independent) probabilities are not given?
3. How do these methods compare with the best we could do?
  - given the frequencies (and later – given the sequence of requests)

This leads to issues of

- expected behaviour (given independent probabilities)
- what if probabilities change
- amortized behaviour (running versus an adversary so we are considering worst-case but for a sequence of operations)
- amortized or expected cost could be compared with but possible for the given probabilities/sequence ... perhaps compare with optimal static structure
- consider our structures that will adapt, (self-adjust, though perhaps on-line and compare with the optimal adjusting structure that knows the sequence in advance (off-line))

Model is a key issue

4. How can we develop self-adjusting data structures? – Adapt to changing probabilities.

Start with linear search. As noted

Worst case  $n$  (succ); also amortized  
“Average (all equal probabilities)  $(n+1)/2$ ”

– it doesn’t matter how you change the structure for worst case or for all probabilities same and independent.

But

- 1) Start with  $\{P_i\}$   $i = 1, n$   $P_i > P_{i+1}$  independent (Stochastic process)
- 2,3) Clearly the best order is  $a_1 \dots a_n$ . Expected cost would be  $S_{opt} = \sum_{i=1}^n i p_i$

Clearly, we could count accesses and converge



Sopt order is cb  
and there are k “unsuccessful comparisons”

What order of requests maximizes this number under MTF? Clearly

$$c^{m-k} (bc)^k$$

So there are 2k “unsuccessful compares” (one for each b & one for each of the last k c’s)

Now observe that this holds in any list for any pair of elements.

Sum over all

$$\text{Cost} \leq 2 \text{Sopt}$$



Note: This bound is tight.

Given  $a_1, a_2, \dots, a_n$ , repeatedly ask for last in list, so all requested equally often.

Cost is n per search

Whereas “do nothing” =  $\text{Sopt} \approx (n+1)/2$

Note again Transpose is a disaster if we always ask for the last in its list.

Observe, for some sequences we do a lot better than static optimal  $a_1^n a_2^n a_3^n \dots a_n^n$

$$\text{Sopt} \quad - \quad (n+1)/2$$

$$\text{MTF} \quad - \quad \left( \frac{n+1}{2} + n - 1 \right) / n = \frac{n+1+2n-2}{2n}$$

$$= \frac{3}{2} - O(1/n)$$

Next – Suppose you are given the entire sequence in advance. Sleator & Tarjan (1985).

The model we will discuss may or may not be realistic, but it is a benchmark.

We consider the idea of a “dynamic offline” method. The issue is:

on line: Must respond as requests come

vs

offline: Get to see entire schedule and determine how to move values

$$\text{Competitive Ratio of Alg} \equiv \text{Worst Case of } \frac{\text{Online time of Alg}}{\text{Optimal Offline time}}$$

A method is “competitive” if this ratio is a constant.

But the model becomes extremely important

Basics – Search for or change element in position i: scan to location i at cost i.

Unpaid exchange – can move element i closer to front of list free (keep others in same order)

Paid – can swap adjacent pairs at cost 1

Borodin and El-Yaniv prohibit going past location i

Sleator-Tarjan proof seems ok though.

Issues – Long/Short scan (ie passing location i)

Exchanging only adjacent values

Further

Access costs i if element in position i

Delete costs i if element in position i

Insert costs n+1 if n elements already there.

After any we can apply update.

**Theorem:** Under the model described, MTF is within a factor of 2 of offline optimal i.e. is 2-competitive.

Let A denote any algorithm, and MF denote the move to front heuristic.

We will consider the situation in which we deal with a sequence, S, having a total of M queries and a maximum of n data values. By convention we start with the empty list. The cost model for a search that ends by finding the element in position i is

$$i + \# \text{ paid exchanges}$$

Recall the element sought may be moved forward in the list at no charge (free exchange) while any other moves must be made by exchanging adjacent values.

Notation

$$\begin{aligned} C_A(S) &= \text{total cost of all operations in } S \text{ with algorithm } A \\ X_A(S) &= \# \text{ paid exchanges} \\ F_A(S) &= \# \text{ free exchanges} \end{aligned}$$

Note:

$$\begin{aligned} X_{MF}(S) &= X_T(S) = X_{FC}(S) = 0 \\ &(\text{T denotes the transpose heuristic, FC denotes frequency count}) \\ F_A(S) &\leq C_A(S) - M \\ &(\text{Since after accessing the } i^{\text{th}} \text{ element there are at most } i-1 \text{ free exchanges}) \end{aligned}$$

**Theorem:**  $C_{MF}(S) \leq 2C_A(S) + X_A(S) - F_A(S) - M$ , for any algorithm A starting with the empty set.

**Proof:** The key idea is the use of a potential function  $\Phi$ . We run algorithm A and MF, in parallel, on the same sequence, S.

$\Phi$  maps the configuration of the current status of the two methods onto the reals.

Running an operation (or sequence) maps  $\Phi$  to  $\Phi'$  and the amortized time of the operation is

$$T + \Phi' - \Phi$$

(i.e. amortized time = real time +  $\Delta\Phi$ )

(so we will aim at amortized time as an overestimate)

The  $j^{\text{th}}$  operation takes actual time (cost)  $t_j$  and amortized cost  $a_j$

$$\sum_j t_j = \Phi - \Phi' + \sum_j a_j$$

where  $\Phi$  is the initial potential and  $\Phi'$ , the final.

$\Phi$  is defined as the number of inversions between the status of A and MF, at the given time.

So  $\Phi \leq n(n-1)/2$

We want to prove that the amortized time to access element i in A's list is at most  $2i - 1$  in MF's.

Similarly inserting in position i+1 has amortized cost  $2(i+1) - 1$ . (Deletions are similar).

Furthermore: we can make the amortized time charged to MF when A does an exchange

-1 for free exchanges

at most +1 for paid exchanges

Initial configuration – empty; so  $\Phi = 0$

Final value of  $\Phi$  is nonnegative

So actual MF cost  $\leq \sum$  amortized time  $\leq$  our bound

{ access or insertion amortized time  $\leq 2C_A - 1$ ;

amortized delete time  $\leq 2C_A - 1$ .

The -1's, one per operation, sum to -M }

Now we must bound the amortized times of operations.

Consider access by A to position  $i$  and assume we go to position  $k$  in MF.

$$x_i = \# \text{ items preceding } i \text{ in MF, but not in A}$$

$$\text{so } \# \text{ items preceding } i \text{ in both is } (k - 1 - x_i)$$

Moving  $i$  to front in MF creates

$$k - 1 - x_i \text{ inversions}$$

and destroys  $x_i$  others

so amortized time is

$$k + (k - 1 - x_i) - x_i = 2(k - x_i) - 1$$

But  $(k - x_i) < i$  as  $k-1$  items precede  $i$  in MF and only  $i-1$  in A.

So amortized time  $\leq 2i - 1$ .

The same argument goes through for insert and delete.

An exchange by A has zero cost to MF,

so amortized time of an exchange is just increase in # inversions caused by exchange,

i.e. 1 for paid, -1 for free. ◆

### Extension

Let MF(d) ( $d \geq 1$ ) be a rule by which the element inserted or accessed in position  $k$  is moved at least  $k/d - 1$  units closer to the front. Then

$$C_{MF(d)}(S) \leq d (2C_A(S) + X_A(S) - F_A(S) - M)$$

{ eg. MF  $\equiv$  MF(1) }

Also

**Theorem:** Given any algorithm A running on a sequence S, there exists another algorithm for S that is no more expensive and does no paid exchanges.

**Proof Sketch:** Move elements only after an access, to corresponding position. There are details to work through.

Further applications can be made to paging.

However – there is a question about the model.

Given the sequence

$$1 \rightarrow 2 \rightarrow \dots \rightarrow n/2 \rightarrow n/2+1 \rightarrow \dots \rightarrow n$$

suppose we want to convert it to

$$n/2 \rightarrow \dots \rightarrow n \rightarrow 1 \rightarrow \dots \rightarrow n/2$$

what “should” I pay?

Observe that all only 3 pointers are changed, though presumably I have to scan the list ( $\Theta(n)$ ).

Sleator and Tarjan model says  $\Theta(n^2)$ , perhaps  $\Theta(n)$  is a fairer charge.

So for an offline algorithm: To search for element in position  $i$ , we probe up to position  $k$  ( $k \geq i$ ) and can reorder elements in positions 1 through  $k$ . Cost is  $k$ .

[J.I. Munro: On the Competitiveness of Linear Search](#). ESA '00 (LNCS 1879 pp 338-345)

Consider the following rule, Order by Next Request (ONR):

To search for element in position  $i$ , continue scan to position  $2^{\lceil \lg i \rceil}$ .

Then reorder these  $2^{\lceil \lg i \rceil}$  elements according to the time until their next requests.

(The next of these to be accessed goes in position 1)

$$\text{Cost } 2^{\lceil \lg i \rceil} = \Theta(i)$$

How does this do? First try a permutation, say  $1, \dots, n$  (let  $n = 16$ ) and assume  $1$  is initially in the last half).

(To simplify diagram we move requested value to front)

Request	Cost	New Ordering
1	16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
2	2	2 1 • •
3	4	3 4 1 2 • •
4	2	4 3 • •
5	8	5 6 7 8 1 2 3 4 • •
6	2	6 5 • •
7	4	7 8 5 6 • •
8	2	8 7 • •
9	16	9 10 11 12 13 13 14 15 16 1 2 3 4 5 6 7 8

Clearly the same cost applies to any permutation

Under our model this cost is  $\sim n \lg n$ .

Recall cost model for reordering a list:

1. Must scan, at least, to element requested, but may scan further covering a subtree whose root is the original root of the tree.
2. May reorder the section scanned “free”. That is charge only for initial scan.

Using it for offline computation we are likely to move an element forward if it is to be reused in the near future.

So, assume each element has associated with it an extra value or “due date” indicating when it is next to be accessed. (Actual accesses still search by value). When value accessed, it gets a new “due date”. This leads the idea of Order by Next Request (ONR). We augment this idea by saying:

Order by Next Request (ONR): On accessing the element in position  $i$  continue the scan to position  $2^{\lceil \lg i \rceil}$ , i.e. the power of 2 encountered first at or after the scan position.

Reorder all value up to position  $2^{\lceil \lg i \rceil}$  based on their due dates. (Element to be retrieved first goes in position 1.)

How expensive is this, really?

Fact: When doing the scan, each block of elements from position  $2^j + 1$  to position  $2^{j+1}$  ( $j = 0 \dots$ ) will be ordered by next request (increasing order by “due date”).

The element requested has to lowest due date of all, so is in the first position of one of these blocks, indeed first position of largest block inspected.

Its “due date” is increased.

Therefore: Each of blocks 0 through  $\lceil \lg i \rceil$  remains in sorted order, except for first element of block  $\lceil \lg i \rceil$ .

And so

To reorder the entire list from position 1 to  $2^{\lceil \lg i \rceil}$  is a matter of merging a pair of blocks of size  $2^j$ ,  $j \equiv 0, \lceil \lg i \rceil - 1$  (and handling the single value “out of place”). This can be done in  $2 * 2^{\lceil \lg i \rceil} = \Theta(i)$  comparisons.

Indeed it can be done in linear time in place if we are working with an array implementation.

Hence:

**Theorem:** The order by next request heuristic (ONR) can be implemented in time linear in the number elements inspected, provided each element has associated with it the time of its next request (due date).

**Proof.** Outlined above

As a consequence we define the cost of a search/reordering as  $2^{\lceil \lg i \rceil}$  as the number of elements inspected. The main concern, however, is to show that this offline heuristic does very well on any input sequence. We work toward an amortization policy that:

- charges a constant amount to each element charged for a reordering. This means a constant fraction of elements inspected must be charged.

and

- charges only elements whose “state” changes in some identifiable way. This will imply a bound on how many times an element is charged.

**Theorem:** The cost of ONR can be amortized to at most  $1 + 4 \lceil \lg r \rceil$  element inspections where  $r-1$  other distinct values have been retrieved since the last request for the given value. By convention we start with the list ordered by first request, alternatively we could start with an empty list and insert on first request.

**Proof:** I’ve called the approach “soak the middle class”

- if the request is for the first or second element charge each inspected element 1
- otherwise charge the  $p$  probe requests to the  $p/4$  elements in the penultimate block. (The requested value will be the first element in the last block, so it will in fact not be charged.)

The key point: An element,  $v$ , is charged at most once in any block (between two requests).

There are two cases:

- $v$  remains in the current block,  $b$ , or moves forward
- $v$  moves to block  $b+1$

**Case 1:**  $v$  remains in block  $b$  or moves “in”, to a lower numbered block.

- Then all elements from  $v$ 's new position to end of  $b+1$  are ordered by next request.
- This relative ordering will be maintained, (though others may be inserted among them)
- None of them requested until after  $v$ .
- Hence  $v$  cannot be charged again until after a request for one beyond block  $b+1$
- On such a request  $v$  is not charged, though if moved it could go to block  $b+1$  or beyond (and eligible to be charged again).

**Case 2:**  $v$  moved “out” to block  $b+1$ . It cannot be charged for access to its block or an earlier one.

Hence element is charged at most once in  $\lceil \lg r \rceil$  blocks up to position  $r$  (latest place it can get to). Hence amortized cost  $\leq 4\lceil \lg r \rceil$ . ◆

**Corollary:** Total cost for a sequence of requests in which  $i^{\text{th}}$  request occurs  $r_i$  steps after the last request for the same element is at most  $m + \sum 4\lceil \lg(r_i + 1) \rceil$ .

**Corollary:** Let  $f_i$  denote frequency of access to element  $i$ . Then total cost of sequence is at most  $m + 4\sum \lceil \lg(m / f_i) \rceil$ .

Rewriting in terms of “probabilities”,  $p_i = f_i / m$ .

**Corollary:** Amortized cost of a search for element  $i$  is  $O(p_i \lg(1 / p_i))$ .

Pushing a little further.

Rather than scanning to one less than a power of 2 locations, scan to position  $a^{k+1} - 1$  and optimize  $a$ . Setting  $a = 4.24429\dots$  we reduce the constant 4 in all above results to  $2.6641\dots$ .

So

- Be careful about models
- Interesting  $\lg$  time sequential search.