Advanced List Structures – Self Organizing Lists

Introduction

The justification for skip lists was motivated by the desire to decrease the expected search time in an *n* element ordered linear list to O(log n). The basic requirement to justify the use of the skip list is that the pattern of searching is assumed to be random. In other words, having searched for a specific element within the list, the next search is expected to be a random "distance" from the previous search element. There is no expected correlation between any two search elements. However, in many search based applications there is a correlation between search elements. This is an adaptation of the "*principle of locality".* Basically this means that once an element is searched for and found, chances are high that it will be searched for again in the near future. There are many different ways that self-organizing lists can be organized; we'll look at four of the more common approaches in this set of notes.

[*Reference:* Hester, James and Hirschberg, Daniel, "Self-Organizing Linear Search," *Computing Surveys 17*, (1985), 137-138.]

[*Reference:* Valiveti, R. and Oommen, B., "Self-Organizing Doubly Linked Lists," *Journal of Algorithms 14*, (1993), 88-114.]

Organization Methods for Self Organizing Lists

Four of the more common methods for the organization protocol in self-organizing lists are:

1. *Move-to-front* method: After locating the search element it is moved to the logical front of the list.

2. *Transpose* method: After the search element is located, it is swapped with its predecessor element.

3. *Count* method: The order of the elements in the list is maintained based upon the number of times the element is referenced.

4. *Ordering* method: The order of the list is maintained using some criteria which is pertinent to the information maintained in the list. In other words, some natural ordering of the data based upon some search protocol.

We'll examine each of these organizational methods separately, however there are some similarities than run across all of the methods. For example, in the first three methods, all new nodes are inserted into the end of the list (logical and physical end), while in the fourth method a new node is inserted into the list at whatever point is appropriate for the search protocol that is employed. With the first three methods, elements most likely to be the search element are positioned physically near the beginning of the list, most explicitly with the *move-to-front* method and most cautiously with the *transpose* method.

Next, we'll look at each of these four methods is some detail, then do some analysis on the expected performance of the various methods.

Move-to-front Method

To illustrate the *move-to-front* method, consider the list shown in Figure 1:

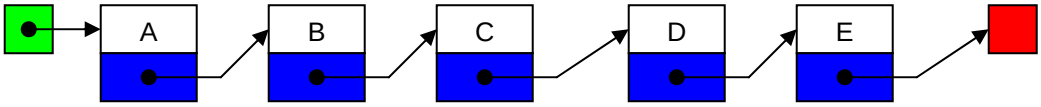

Figure 1 – Singly-linked list – ordered in terms of move-to-front organization

Now assume that an access has occurred to this list in terms of a search for element "C". This access will cause the list to reorganize into the one shown in Figure 2.

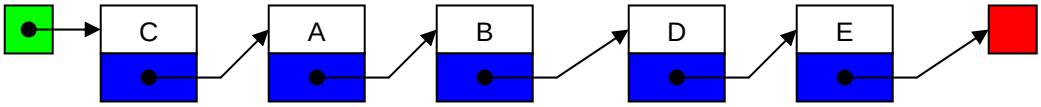

Figure 2 – Reorganized list after access to node "C" using move-to-front organization

If the next access happens to be a search for element "A", the list will reorganize into the one shown in Figure 3.

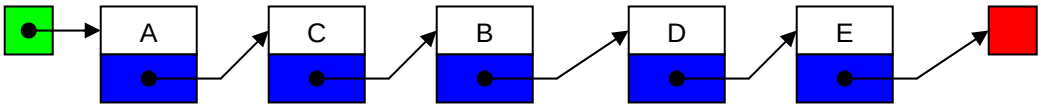

Figure 3 – Reorganization of list in Figure 2 after access to element "A"

After some time has elapsed, in terms of the number of accesses to the list, the nodes of the list will be ordered from the most recently accessed node to the least recently accessed node. [Application: Ordering page frames based upon the least recently accessed page frame is a common page replacement strategy employed by operating systems in a virtual memory environment.]

The move-to-front method is a very "optimistic" appraoch to the organization of the list in the sense that the expectation is that the element on the head of the list will be searched for again in the immediate future. We'll discuss the access patterns a bit later, but for now let it suffice to say that, such a technique is clearly not optimal if the search pattern is truly random.

Transpose Method

To illustrate the *transpose* method, consider the list shown in Figure 4:

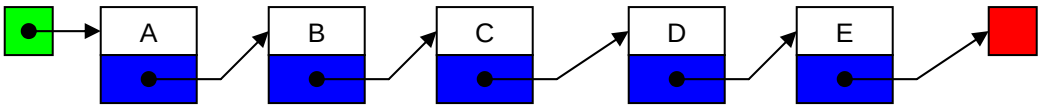

Figure 4 – Singly linked list – initial state

Suppose that the list in Figure 4 is accessed in a search for the element "D". The list shown in Figure 5 results when the list is organized using the transpose method.

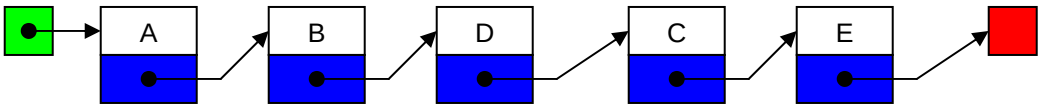

Figure 5 – List of Figure 4 after access of node "D" using transpose organization

Notice that the node just accessed has moved closer to the head of the list, but has not moved as drastically toward the head as was the case with the move-to-front method. The transpose method is a much more pessimistic approach to the reorganization. In other words, it will take repeated accesses to element "D" to literally move it to the head of the list. This approach, while still adhering to the principle of locality, does so much more cautiously (or pessimistically) since it will literally take many consecutive references to move an element to the head of the list if it is far away from the head initially. Over a period of time, the list will be ordered so that the most frequently accessed elements will tend to be positioned toward the head of the list while the less frequently accessed elements will tend to be positioned toward the tail of the list. [Application: As before, a common page

replacement strategy is to replace pages which are among the less frequently accessed pages.]

Count Method

Although quite similar in many ways to the first two organizational methods, the count method requires that a counter be associated with each element in the list which records the number of accesses to that element. The list is maintained in the order of most number of accesses down to the least number of accesses. In the event of a tie between two or more elements with the same number of accesses, the tie is typically broken arbitrarily.

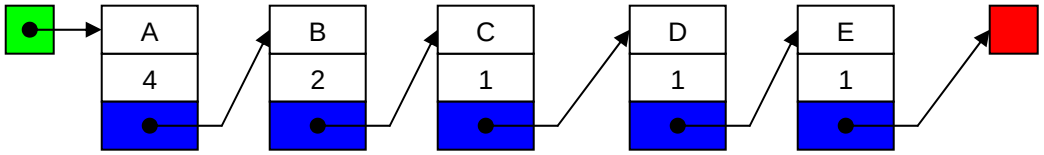

Figure 6 – Singly linked list ordered in terms of the count of references to a node

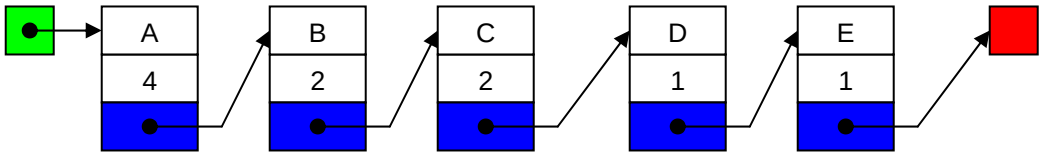

Figure 7 – List of Figure 6 after access to element C

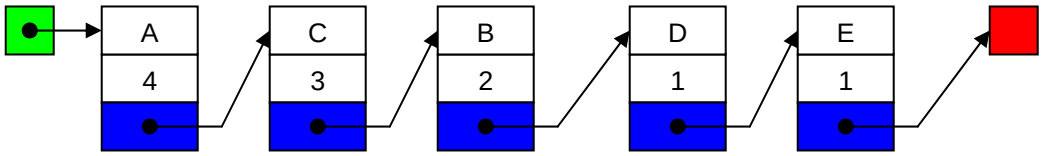

Figure 8 – List of Figure 7 after an access to element C

After a suitable period of time the list has become organized in a fashion that places the most frequently access elements near the head of the list and the least frequently accessed elements near the tail of the list.

Ordering Method

This organizational method is the most flexible of the four organizational methods we are considering and as such will also prove the most difficult to analyze with any generality. The organizational criteria for this method can be any which are suitable for the data/information maintained in the list. For

example, the order of the list shown in Figure 9 appears to be simply alphabetical while the list in Figure 10 is ordered based upon GPA.

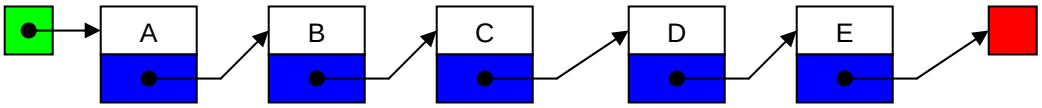

Figure 9 – Singly linked list ordered alphabetically

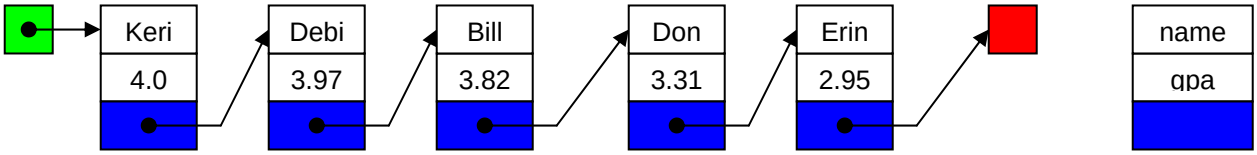

Figure 10 – Singly linked list ordered based upon decreasing GPA

a node

Maintaining a list of elements based upon what is typically a "non-key" value is a common database problem in the construction and maintenance of non-key index structures. [Discuss indices here.]

Insertions into Self-Organizing Lists

Before we get to the analysis of the various forms of the self-organizing lists, we'll look at the insertion technique for each of the four organizational methods.

Insertion with Move-to-front, Transpose, and Count Methods

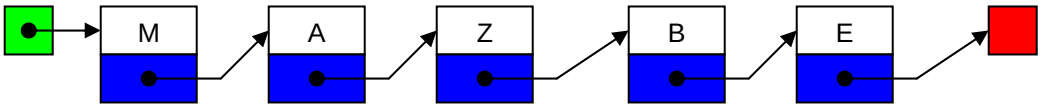

Figure 11 – Singly linked list – initial state

Assuming that the list in Figure 11 is organized using either the move-to-front, transpose, or count methods a search for the element containing "K" would, in each case the resulting list would be configured as shown in Figure 12.

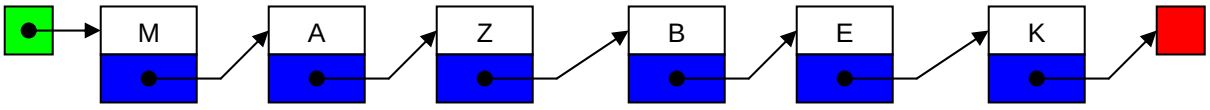

Figure 12 – List of Figure 11 after search for "K" using any one of the organizational methods of move-to-front, transpose, or count.

Insertion using Ordering Method

In this case the location for the insertion is dependent upon the ordering criteria upon which the list is ordered. Using the example from Figure 10 in which the list was ordered based upon a student's gpa, the insertion of a new student named Mark with a gpa of 3.99 would result in the list shown in Figure 13.

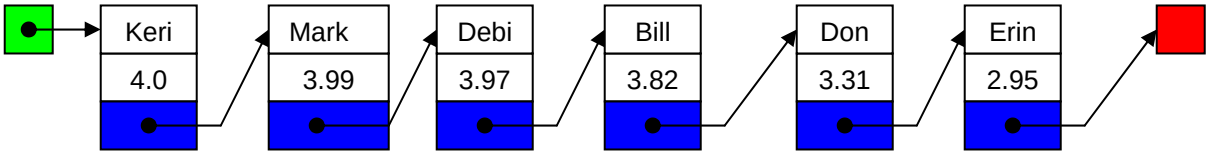

Figure 13 – List of Figure 10 reorganized after the insertion of new student (Mark, 3.99)

Analysis of Self-Organizing Lists

Analysis of the efficiency of self-organizing lists operating under one of these four organizational methods is customarily done by comparing their efficiency to that of *optimal static ordering*. In an optimal static ordering, all data is ordered by the frequency of occurrence in the body of data so that the list is used only for searching and not for inserting new data. With this approach, two passes through the body of data are required, one to build the list and another to use the list for searching alone.

What follows in this section has been excerpted primarily from the two references listed above (as well as some other references) and essentially amounts to experimental measurements of the efficiency of the self-organizing lists as a comparison of the actual number of comparisons to the maximum number of possible comparisons. This latter number is determined by adding the lengths of the list at the moment of processing the search element. To illustrate this evaluation technique, we'll use the information represented in Table 1 which incorporates a specific access pattern of searching against that list. Since the list is initially empty, it must be first constructed based upon the pattern of searches. In addition to the four self-organization methods, we'll also keep track of the structure of the list as if it were ordered based simply on the order of insertion of new elements (i.e., not self-organizing).

Search Element	List Length Prior to Search	Simple List	Self-Organization Method			
			Move-to-front	Transpose	Count	Ordering
A	0	A (0)	A (0)	A	A	A
C	1	A C (1)	A C (1)	A C	A C	A C
B	2	A C B (2)	A C B (2)	A C B	A C B	A B C
C	3	A C B (2)	C A B (2)	C A B	C A B	A B C
D	3	A C B D (3)	C A B D (3)	C A B D	C A B D	A B C D
A	4	A C B D (1)	A C B D (2)	A C B D	C A B D	A B C D
D	4	A C B D (4)	D A C B (4)	A C D B	D C A B	A B C D
A	4	A C B D (1)	A D C B (2)	A C D B	A D C B	A B C D
C	4	A C B D (2)	C A D B (3)	C A D B	C A D B	A B C D
A	4	A C B D (1)	A C D B (2)	A C D B	A C D B	A B C D
C	4	A C B D (2)	C A D B (2)	C A D B	C A D B	A B C D
C	4	A C B D (2)	C A D B (1)	C A D B	C A D B	A B C D
E	4	A C B D E (4)	C A D B E (4)	C A D B E	C A D B E	A B C D E
E	5	A C B D E (5)	E C A D B (5)	C A D E B	C A E D B	A B C D E

Table 1 – Illustration of the four self-organization methods

Notice in Table 1 that the access pattern (Column 1 reading down) to the list is a specific one, namely: A, C, B, C, D, A, D, A, C, A, C, C, E, E consisting of 14 letters, 5 of which are different. The length of the list prior to processing the search element is shown in the second column, the sum of these numbers is 46. This number is used to compare the number of all made comparisons to this combined length. Using this technique we can determine what percentage of the list was scanned during the entire process. For all of the lists except optimal ordering this combined length is the same; only the number of comparisons can change. The optimal list for this access pattern in C, A, D, E, B and requires 32 comparisons. For example, when using the move-to-front method, 0+1+2+2+3+2+4+2+3+2+2+1+4+5 = 33 comparisons were made (the number of comparisons made using the different organization strategies is shown in parenthesis at the end of the lists in Table 1), which is 71.7% when compared to 46. [33/46 * 100 = 71.7%] Notice that the number 46 represents the worst case scenario, the combined length of intermediate lists every time all the nodes in the list are compared. Plain search, with no reorganization, requires 30 comparisons, which is 65.2%. [30/46 * 100 = 65.2%] [For practice,

you should determine the efficiency, using this technique for the transpose, count, and ordering methods of organization.]

The sample data shown in Table 1 is in agreement with theoretical analyzes which indicate that count and move-to-front methods are, in the long run, at most twice as costly as the optimal static ordering; the transpose method approaches, in the long run, the cost of the move-to-front method. In particular, using amortized analysis, it can be proven that the cost of accessing a list element with the move-to-front method is at most twice the cost of accessing this element on the list that uses optimal static ordering. The basics of this proof follow:

The proof of the statement above uses the concept of *inversion*. For two lists containing the same elements, an inversion is defined to be a pair of elements (*x,y*) such that in one of the lists *x* precedes *y* and in the other list *y* precedes *x*. For example, the list [C, B, D, A] has four inversions with respect to the list [A, B, C, D], which are: (C,A), (B,A), (D,A), and (C,B). (Recall that the same concept was used in CS2 when we discussed the various sorting algorithms that operated on the basis of removing inversions.) The *amortized cost* is defined to be the sum of the actual cost and the difference between the number of inversions before accessing an element and after accessing it,

amCost(x) = cost(x) + (inversionsBefore Access(x) – inversionsAfterAccess(x))

To determine this value, consider an optimal list OL = [A, B, C, D] and a move-to-front list MTF = [C, B, D, A]. The access of elements usually changes the balance of inversions. Let *displaced(x)* be the number of elements preceding *x* in MTF but following *x* in OL. For example, *displaced(A)* = 3, *displaced(B)* = 1, *displaced(C)* = 0, and *displaced(D)* = 0. *Displaced(A)* = 3 since in MTF elements C, B, and D precede A and in OL these elements follow A. Similarly, *displaced(B)* = 1, since in MTF the element C precedes B yet C follows B in OL; *displaced(C)* = 0 since in MTF no elements precede C; *displaced(D)* = 0 since although elements C and B precede D in MTF, no elements follow D in OL.

Let $pos_{MTF}(x)$ be the current position of *x* in MTF, then $pos_{MTF}(x) – 1 –$ *displaced(x)* is the number of elements which precede *x* in both lists. For D, this value will be 2 , and for all other elements this value will be 0. ($pos_{MTF}(D) – 1 –$ *displaced(D) = 3 – 1 – 0 = 2;* $pos_{MTF}(A) – 1 –$ *displaced(A) = 4 – 1 – 3 = 0;* $pos_{MTF}(C) – 1-$ *displaced(C) = 1 – 1 – 0 = 0;* $pos_{MTF}(B) – 1 –$ *displaced(B) = 2 – 1 – 1 = 0).*

Now, accessing an element *x* and moving it to the front of MTF creates a total of $pos_{MTF}(x) – 1-$ *displaced(x)* new inversions and removes a total of *displaced(x)* inversions. The amortized time to access *x* becomes:

$amCost(x) = pos_{MTF}(x) + pos_{MTF}(x) - 1 - displaced(x) - displaced(x)$

$= 2\,(pos_{MTF}(x) - displaced(x)) - 1$

where $cost(x) = pos_{MTF}(x)$.

Accessing element A transforms MTF = [C, B, D, A] into [A, C, B, D] and $amCost(A) = 2(4 - 3) - 1 = 1$.
Accessing element B transforms MTF = [C, B, D, A] into [B, C, D, A] and $amCost(B) = 2(2 - 1) - 1 = 1.$
Accessing element C transforms MTF = [C, B, D, A] into [C, B, D, A] and $amCost(C) = 2(1 - 0) - 1 = 1.$
Accessing element D transforms MTF = [C, B, D, A] into [D, C, B, A] and $amCost(D) = 2(3 - 0) - 1 = 5.$

Notice however, that the common elements which precede *x* on the two lists cannot exceed the number of all elements preceding *x* on OL; therefore it must be that: $pos_{MTF}(x) - 1 - displaced(x) \leq pos_{OL}(x) - 1,$ so that we have:

$amCost(x) \leq 2pos_{OL}(x) - 1$

The amortized cost of accessing an element *x* in MTF is in excess of $pos_{OL}(x) - 1$ units to its actual cost of access in OL. This excess is used to cover an additional cost of accessing elements in MTF for which $pos_{MTF}(x) > pos_{OL}(x),$ that is, elements that require more accesses in MTF than in OL.

It is important to remember that the amortized costs of single operations are meaningful only in the context of sequences of operations. The cost of an isolated operation will seldom equal its amortized cost; however, in a sufficiently long sequence of accesses, each access on the average will take at most $2pos(x) - 1$ time. Table 2 illustrates the performance of self-organizing lists with data taken from actual experimental results. The first two columns of numbers are based upon data from programs and the remaining columns represent straight English text. Except for alphabetic ordering, all the methods improve their efficiency as the size of the list increases. The move-to-front and count methods are essentially the same in their efficiency, and both outperform the transpose, plain, and ordering methods. The poor performance for smaller lists is due to the fact that all of the methods are busy including new elements into the lists, which requires an exhaustive search of the list. Later, the methods will concentrate on the reorganization geared toward reducing the cost of subsequent searches.

Table 2 also contains data for the skip list. Notice the overwhelming difference in the efficiency of the skip list compared to any of the self-organizing list techniques. This is to some extent misleading due to the way the data is presented in Table 2. In Table 2, only comparisons of data are included with no indication of any other operations required to execute the analyzed methods. In particular, there is no indication of how many references are used and "relinked" (the "backtracking" that occurs in the movement through the hierarchy of lists that comprise the skip list). If this information were included, the difference between the various self-organizing methods and the skip list would be less dramatic that it appears in Table 2.

Different Words/ All Words	Type of data in the List					
	Program data		English Text			
Reorganization Strategy	149/423	550/2847	156/347	609/1510	1163/5866	2013/23065
Optimal	26.4	17.6	28.5	24.5	16.2	10.0
Plain	71.2	56.3	70.3	67.1	51.7	35.4
Move-to-Front	49.5	31.3	61.3	54.5	30.5	18.4
Transpose	69.5	53.3	68.8	66.1	49.4	32.9
Count	51.6	34.0	61.2	54.7	32.0	19.8
Alphabetic Order	45.6	55.7	50.9	48.0	50.4	50.0
Skip List	12.3	5.5	15.1	6.6	4.8	3.8

Table 2 – Efficiency of Self-organizing lists using (*number of data comparisons)/(combined length) expressed as a percentage.*

Summary

As the data in Table 2 suggests, empirical results indicate that for lists of modest size, the generic linked list suffices. With an increase in the amount of data and/or an increase in the frequency with which list elements need to be accessed, more sophisticated methods and data structures will be required.

Answers to practice problem mentioned on page 7.

Search Element	List Length Prior to Search	Simple List (insert order)	Self-organization Methods		
			Transpose	Count	Ordering
A	0	A (0)	A (0)	A (0)	A (0)
C	1	A C (1)	A C (1)	A C (1)	A C (1)
B	2	A C B (2)	A C B (2)	A C B (2)	A B C (2)
C	3	A C B (2)	C A B (2)	C A B (2)	A B C (3)
D	3	A C B D (3)	C A B D (3)	C A B D (3)	A B C D (3)
A	4	A C B D (1)	A C B D (2)	C A B D (2)	A B C D (1)
D	4	A C B D (4)	A C D B (4)	D C A B (4	A B C D (4)
A	4	A C B D (1)	A C D B (1)	A D C B (3)	A B C D (1)
C	4	A C B D (2)	C A D B (2)	C A D B (3)	A B C D (3)
A	4	A C B D (1)	A C D B (2)	A C D B (2)	A B C D (1)
C	4	A C B D (2)	C A D B (2)	C A D B (2)	A B C D (3)
C	4	A C B D (2)	C A D B (1)	C A D B (1)	A B C D (3)
E	4	A C B D E (4)	C A D B E (4)	C A D B E (5)	A B C D E (5)
E	5	A C B D E (5)	C A D E B (5)	C A E D B (4)	A B C D E (5)

For the transpose method the count of comparisons is 31. This gives us: (31/46)*100 = 67.39%.

For the count method the total number of comparisons is 34. This gives us: (34/46)*100 = 73.91%.

For the ordered method the total number of comparisons is 35. This gives us: (35/46)*100 = 76.08%