Skip Lists vs. B-Trees

by Michael D. Black

November 15, 2001

Skip lists are a relatively new algorithm introduced in 1987 by William Pugh [1]. Their simplicity and performance makes them an attractive alternative to the well known B-tree algorithms. Testing reveals a dramatic speed advantage for skip lists when compared to B-trees. In addition to the basic speed advantage of the algorithm, skip lists also show an additional speed advantage for large data sets.

Background

The fastest general method for searching an ordered list is the binary search[5]. The binary search doubles the number of items covered at each iteration. Finding an item in a list of 256 items takes 8 iterations; a list of 65,536 items takes 16 iterations; and a list of 4,000,000,000 items takes just 32 iterations. The binary search is the way you play the old high-low guessing game. You can guess a number from 1 to 256 in 8 or less guesses if you start in the middle (i.e., 128) and keep dividing by 2. For example, if the number to find is 103 the guesses would proceed as 128, 64, 96, 104, 100, 102, and finally 103. Notice how the difference in each successive guess is halved -- 64, 32, 16, 8, 4, 2, 1.

Binary trees have a bad worst-case performance when the tree becomes one sided. A one-side tree causes a linear search time which is like playing the high-low guessing game by guessing one, two, three, four, etc. This one-sided behavior is avoided by rebalancing the tree periodically creating a Balanced Tree (i.e., B-Tree). Figure 1 gives a visual example of a balanced tree for the numbers 1 through 7. Note at this point that there is one and only one balanced tree for a given set of data. The search for this single solution is a dominant factor in the execution speed of any B-Tree algorithm.

To find an item in a B-Tree start at the highest level of the tree and then follow either the left or right node depending on whether the compare is less than or greater than the current node. The process is repeated until the item is equal or until a node is reached where there are no leafs.

One of the inherent problems of the B-Tree is the time it takes to rebalance the tree. If you are constructing a database which does not get changed much (e.g., voter registration) the time involved in rebalancing the tree is not very important. However, in a database which does get changed a lot (e.g., airline reservations) the rebalancing could become a dominant factor in the performance of the system. This rebalancing has a such a noticeable effect that many B-tree implementations go to great lengths to avoid, or at least to delay rebalancing. The additional software needed to achieve this delayed rebalancing only adds to the complexity of the B-Tree algorithm.

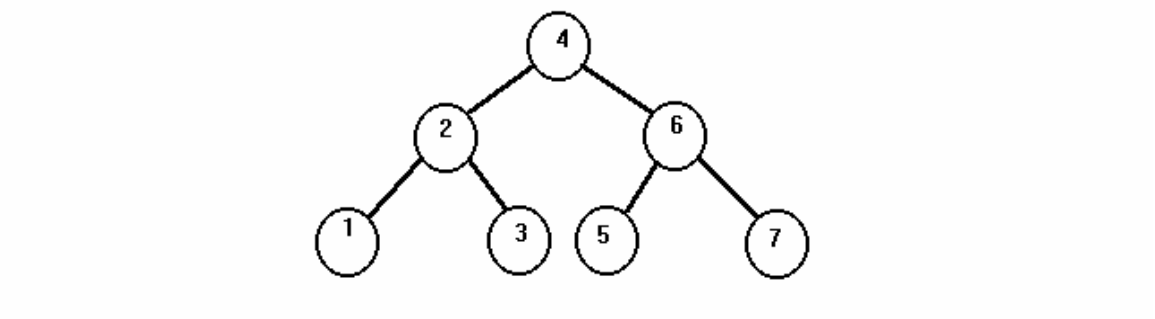

Figure 1
Balanced Binary Tree

Skip lists achieve their balance by using a random number to determine the level where a node will be placed. Although they have a poor worst-case performance, the probability of the worst case occurring is quite small. The primary disadvantage of the skip list approach is that you never know what will cause the worst case (due to its being probabilistic). However, the law of large numbers [6] gives us some assurance that the skip

list is highly unlikely to degenerate into a worst-case condition. Figure 2 shows an example of a four-level skip list generated with the numbers 1 through 7. The skip list maintains a pointer to the first item in the list which is "1" at level 4. From there you can sequentially go through the list to each successive number.

To find an item in a skip list start at the highest level and check to see if the key is greater than the first item in the level. If not, drop one level and repeat. Once the key is greater simply follow the chain. For example, to find "5" start at "1", drop to "4", then follow "4" to "5".

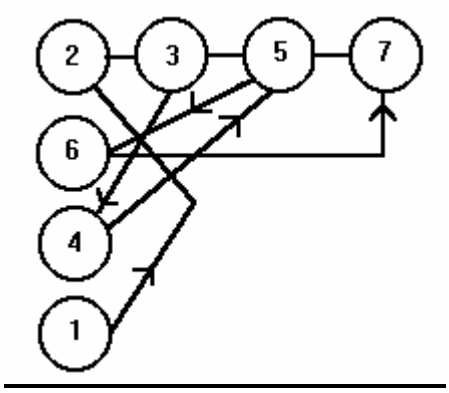

Figure 2
Skip List

The primary advantage of skip lists is speed (as I will show later on). A secondary advantage is that the coding for inserting and deleting keys is minimal compared to B-trees.

Some background on virtual memory systems is appropriate at this point as testing showed it to be a dominant factor in performance. Programs are allowed to use a large contiguous virtual memory, sometimes called a virtual address space[4]. Virtual memory is stored on disk and is mapped into real memory as needed by the program.

Storage is done in blocks of memory called pages. When a program requires a page of memory that is stored on disk the computer must read the disk. This is immensely slow compared to accessing real memory. If a program is inefficient in the way it utilizes memory it can cause a condition called "thrashing" wherein pages of memory are being written to and

read from disk very frequently. This slows down the computer considerably. The efficiency of a program with respect to this "thrashing" phenomena is called "paging behavior." In our case it is related to how many pages the algorithms must access to find a particular node.

Complexity

B-trees are an extremely popular method which makes for interesting reading. Sedgewick [2] goes so far as to say that deleting a node is "quite tricky" and leaves it to the reader to implement. This is an example where an author teaches computer theory instead of giving concrete examples. Teaching computer theory without examples is like teaching physics without lab experiments. All the theory in the world will be of little use if it can't be applied.

Kruse, Leugn, and Tondo [3] do somewhat better than Sedgewick in that they provide a full implementation. However, it is interesting to note that in their earlier discussion of binary trees they give a deletion algorithm that is "far from optimal". This self-admission demonstrates the complexity of maintaining a balanced tree. Indeed, the implementation they provide for B-trees goes on for almost 4 pages of code just to delete a node. The skip list takes only 20 lines of code to do the same thing. Also of note here is that it is dangerous for authors to present code that is "non-optimal." Many authors have regretted doing so when they see their code pop up in applications. In particular, giving "non-optimal" examples on a complex area is particularly prone to being abused as many programmers will use the "non-optimal" code just to get the job done instead of taking the time to improve it.

Skip lists are much simpler to implement though they are somewhat more difficult to understand. Their simplicity creates a condition where there is only one way to insert and one way to delete a node. There is no such thing as a non-optimal solution for skip lists.

Testing

Source code for a skip list implementation was obtained via the internet from mimsy.cs.umd.edu:/pub/skipLists. Many B-tree codes were examined but most were disk-based and could not be compared to the memory-based skip list implementation. A memory-based package from wuarchive.wustl.edu:/mirrors/unix-c/languages/c/btree1.tar.-z was the one used for testing (largely based on [3]).

Tests were run to insert N pseudo-random integers and determine the number of inserts per second for each technique. Random integer inserts were used to avoid the worst-case performance of B-trees. The skip list algorithm does not care about the ordering of the inserts due to the inserts being placed probabilistically, whereas the B-tree has to rebalance frequently when forced to create a one-sided tree. Pseudo-random numbers provide an acceptable method of avoiding this behavior.

N was varied from 50,000 to 950,000 in steps of 50,000. The skip list was configured with 32 levels and the B-tree was configured as a 2-way tree (other configurations on the B-tree did not have a significant impact on performance). Timing was done on a 40MHz Sparc 2 with 16 megabytes of memory running Sun OS. User elapsed time ("Wall Time" in Table 1), total CPU time, and system time were recorded. All code was compiled using gcc v1.41 with standard optimization (i.e. gcc -O).

Table 1 show the performance timing of the B-tree and skip list. The CPU time is a fair reflection of the complexity of an algorithm. It does not take into account disk access wait times and paging behavior. CPU time shows that the skip list was 5-10 times faster than the B-tree. Some side-testing of the B-tree was done in several different n-way modes. None of them improved performance more than a factor of 2 and several of them decreased performance by significant amounts.

Number of Inserts	Skiplist Wall time	B-tree Wall time	Skiplist CPU time	B-tree CPU time	Skiplist System time	B-tree System Time
50000	1.0	6.0	0.7	5.2	0.2	0.4
100000	2.1	12.1	1.5	11.6	0.5	0.5
150000	3.2	19.0	2.7	18.2	0.5	0.7
200000	4.2	25.7	3.5	24.6	0.7	1.1
250000	5.3	33.4	4.2	32.0	1.0	1.3
300000	6.3	41.1	5.1	39.1	1.2	1.8
350000	7.4	48.8	5.9	46.9	1.5	1.8
400000	8.5	56.6	6.9	54.5	1.6	1.9
450000	9.6	64.3	7.6	61.6	2.0	2.4
500000	12.9	72.1	8.6	69.4	2.2	2.6
550000	13.3	81.9	9.5	76.7	2.3	2.9
600000	15.2	312.0	10.3	84.6	2.6	12.9
650000	15.5	1091.3	11.2	94.6	2.9	40.1
700000	16.4	2340.7	12.0	104.3	3.2	81.4
750000	18.2	4100.1	13.2	115.3	3.2	138.9
800000	20.1	6390.9	13.9	126.3	3.6	202.9
850000	21.1	9388.9	14.8	140.5	3.9	289.0
900000	21.1	12734.4	15.8	151.7	4.0	384.5
950000	22.3	17180.5	16.7	166.8	4.4	514.7

Table 1

Wall time is the most important measure one can make of an algorithm. It reflects how long one will have to wait for the computer to finish. Even though an algorithm may be very efficient with the CPU, the wall time can be much worse due to disk I/O and paging behavior. Paging behavior occurs in a virtual memory system (like UNIX or Windows). When a program exceeds the computers' real memory it will start moving real memory to disk. The computer maintains a table of memory pages which reflects whether a memory page is real or on disk (i.e., virtual). If the software needs to access memory that is on disk, the system will take a different page and write to disk to make room for the page it has to read in. An extremely inefficient algorithm will cause paging to occur frequently, which will dramatically increase the wall time.

Table 1 reveals a major problem with the B-tree implementation. At 600,000 inserts, the Wall time and System Time start taking off exponentially. This is due to the paging behavior of the B-tree algorithm. Even though this is a memory based method, it is running on a virtual memory system. So, at some point it will need to start using virtual memory. Once this happens, the paging behavior dominates performance. This performance loss is in direct contrast to the supposed advantage of B-trees for external searching (i.e., disk-based B-tree) and at 950,000 inserts the skip list is 770 times faster than the B-tree.

Figures 3 and 4 graphically depict the inserts per second for both algorithms. The B-tree shows a linear decrease in performance that is much greater than that of the skip list. The B-tree paging behavior is also seen to start at 600000 inserts.

The memory usage for the skip list and B-tree were approximately the same. This infers a distinct difference in paging behavior. It would appear that the shortcuts created by the skip list algorithm create a distinct advantage that should carry over to a disk-based implementation.

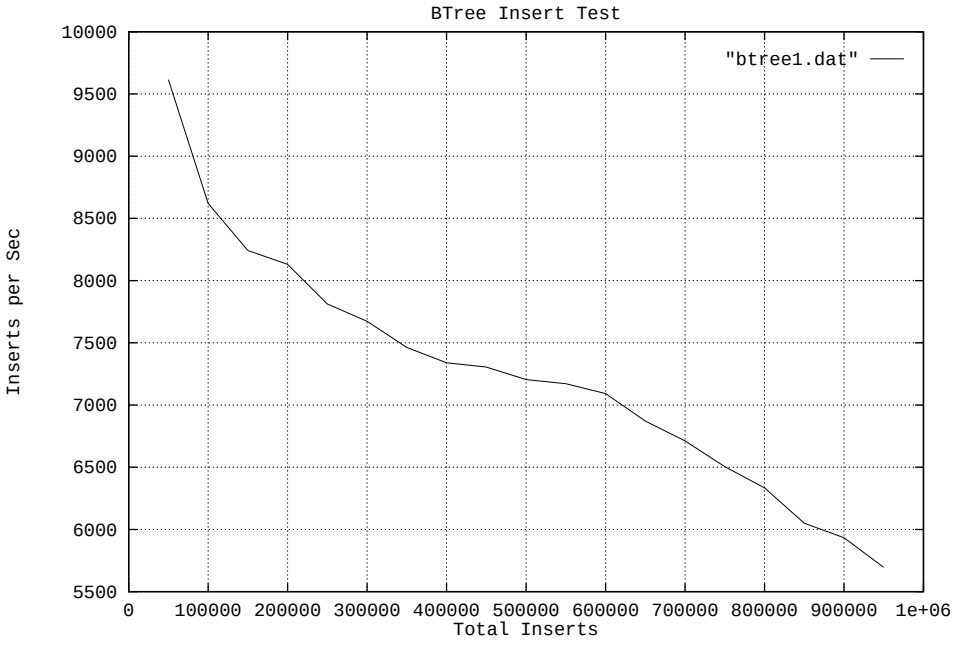

Figure 3

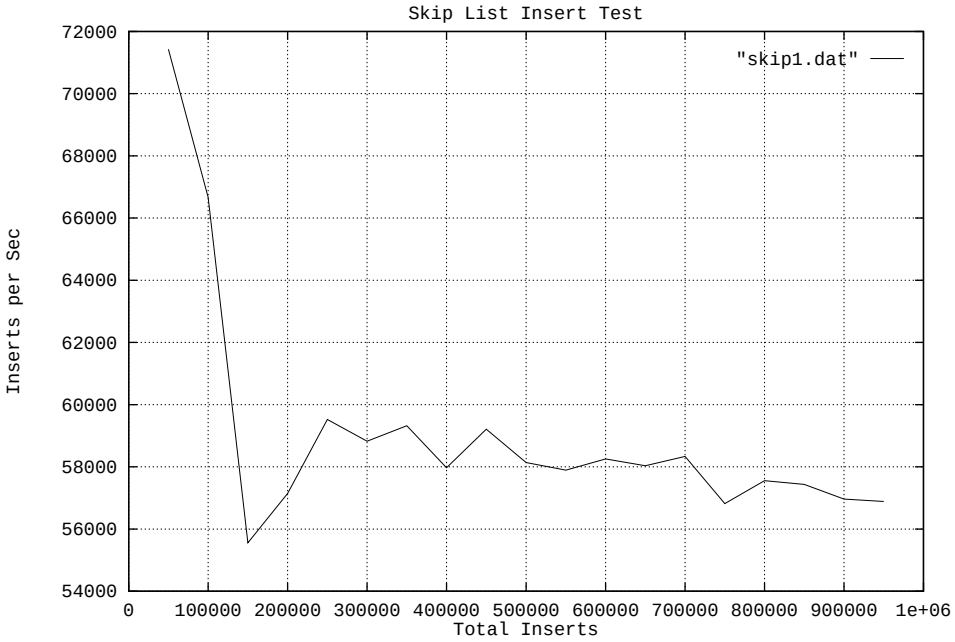

Figure 4

Effects on Database Algorithms

Multi-user databases must use some form of record locking to prevent collisions. One method frequently used is page locking. The vast difference in paging behavior shown in Table 1 would translate to improved performance for this locking method.

Another consideration is the locking necessary for updating indexes. The B-Tree algorithm will generally have to lock the entire index fairly frequently as it rebalances the tree. Skip lists would only have to lock 1-3 pages. This would allow many more asynchronous processes and reduce the chance of collisions.

Date discusses database performance issues and states the prevalence of the B-tree algorithm [7]. He goes so far as state that "B-trees are the obvious choice." Date's writings only consider queries, though, and not updates. A heavily transaction-oriented system could benefit greatly from a skip list approach. Indeed, there are some systems which are mostly

updates instead of queries. Inventory ordering systems, for example, would have a small percentage of queries compared to the number of orders placed.

Conclusions

Skip lists provide an efficient means of maintaining an ordered list. They out-perform B-trees by a large factor (5-770) for random inserts. Large, frequently changed databases could potentially gain large performance increases by utilizing skip lists due to the paging behavior of the algorithm.

Future work should address search speeds and a measure of real performance in a disk-based implementation. In addition, more expansive testing of different configurations for the algorithms should be performed to determine tradeoffs for insert and search speeds..

References

1 William Pugh, Skip Lists: A Probabilistic Alternative to Balanced Trees, Internet mimsy.cs.umd.edu:pub/skipLists/skiplist.ps.Z, June 1990

2 Robert Sedgewick, Algorithms, Addison-Wesley Publishing Company, Inc., 1983

3 Robert L. Kreuse, Bruce P. Leung, and Clovis L. Tondo, Data Structures and Program Design in C, Prentice-Hall, Inc., 1991

4 Leland L. Beck, System Software: An Introduction to Systems Programming, Addison-Wesley Publishing Company Inc., 1985, p. 332

5 C. William Gear, Applications and Algorithms in Computer Science, Science Research Associates, Inc, 1978, p. A107

6 Murray R. Siegel, Schaum's Outline Series - Theory and Problems of Probability and Statistics, McGraw-Hill Book Company, 1975

7 C.J Date, Relational Database Selected Writings, Addison-Wesley Publishing Company Inc., 1988, p. 69