Comparison of Skip List Algorithms to Alternative Data Structures

David N. Etim[1]

***Abstract*—**A skip list is a data structure that can be used as an alternative to balanced trees. This data structure uses probabilistic balancing rather than strictly enforcing balancing and causes simplicity in the algorithms for insertion and deletion for a skip list. This paper analyzes literature that compares the use of skip lists with other data structures. This project focused not only on the comparison of data structures, but the structure of skip list algorithms and their performance against similar types used in implementation by other data structures. The findings were that skip lists are a much easier and efficient algorithm to implement compared to self-adjusting tree and balanced search tree algorithms.

***Keywords*—** Skip Lists, Randomized Algorithms, And Balanced Search Trees.

I. Introduction

So what do we have as an alternative to balanced tree algorithms? Skip lists. A data structure which is a probabilistic alternative and is balanced by consulting a random generator. Although skip lists have a bad worst-case performance, no input sequence consistently produces the worst-case performance, which is much like the quicksort type of algorithm when the pivot element is chosen randomly. Skip lists have balance properties similar to that of a search tree built by random insertions, but doesn't require random insertions. A skip list is also very unlikely to be in a state of unbalanced. For example, a dictionary can have more than 250 elements. The chance that a search done on this dictionary will take more than three times the expected run time is less than one in a million. So we have good stability with a skip list balance given a solid number of elements.

Balancing a data structure probabilistically is easier than doubtlessly maintaining the balance. Skip lists are more of a representation than trees for many applications, which also leads to simple algorithms. The simplicity of a skip list algorithm is fantastic because it makes it easier for implementation and provides significant constant factor speed improvements over balanced tree and self-adjusting tree algorithms. Skip lists are also very space efficient. They can easily be configured to require an average of (4/3) pointers per element and balance is not required to be stored with each node.

[1]Computer Science and Engineering, University of Connecticut

II. Skip Lists

A skip list S for an ordered dictionary D consists of a series of sequences that we can note as {S0, S1,...,Sh}. Each sequence Si stores a subset of the items of D sorted by a non-decreasing key plus items with two special keys, denoted as $-\infty$ and $+\infty$, where $-\infty$ is smaller than every possible key that can be inserted in D and $+\infty$ is larger than every possible key that can be inserted in D. Additionally, the sequences in S satisfy the following conditions:

- Sequence S0 contains every item of dictionary D (plus the special items with keys $-\infty$ and $+\infty$).
- For i = 1,..., h − 1, sequence Si contains (in addition to $-\infty$ and $+\infty$) a randomly generated subset of the items in sequence Si−1.
- Sequence Sh contains only $-\infty$ and $+\infty$.

It is customary to visualize a skip list S with sequence S0 at the bottom and sequences S1,...,Sh−1 above it. Also, we can refer to h as the height of skip list S. The sequences are set up so that Si+1 contains more or less every other item in Si. As can be seen later in the insertion method, the items in Si+1 are chosen at random from the items in Si by picking each item from Si to also be in Si+1 with probability of 1/2. Essentially, we flip a coin for each item in Si and place that item in Si+1 if the coin comes up as "heads." Therefore, we expect S1 to have about n/2 items, S2 to have about n/4 items, and, in general, Si to have about (n/2) i items. In other words, we expect the height h of S to be about log (n). We look at a skip list as a two-dimensional collection of positions arranged horizontally into levels and vertically into towers. Each level corresponds to a sequence Si and each tower contains positions storing the same item across consecutive sequences. The positions in a skip list can be traversed using the following operations:

- after (p): the position following p on the same level
- before (p): the position preceding p on the same level
- below(p): the position below p in the same tower
- above (p): the position above p in the same tower

III. Algorithm Techniques

This section gives algorithms to search for, insert and delete elements in such collections as a dictionary or symbol table. The search operation return contents of the value associated with the success or failure of key being presented or not presented. The insert operation associates a specified key with a new value if not already presented, and the delete

operation removes the specified key. Each element is represented by a node, the level of which is chosen randomly when the node is inserted without regard for the number of elements in the data structure. A level node i has I forward pointers, pointing to a higher level, which index 1 through i. We do not need to store the level of a node in the node. The level of a list is the highest level currently in the list, or 1 given that the list is indeed empty.

3.1. Searching

A skip list structure allows for simple dictionary search algorithms. We can take into account the SkipSearch algorithm in Figure 1. This method takes a key k and finds the item in a skip list S with the largest key that is less than or equal to k [2]. The algorithm begins by setting a position variable p to the top-most, left position in the skip list S. The variable p is set to the position of the special item with $-\infty$ in S_h.

Algorithm SkipSearch (k):

Input: A search key k

Output: Position p in S_0 such that the item at p has the largest key less than or equal to k

Let p be the top-most-left position of S (which should have at least 2 levels).

```
while below (p) 6= null do
        p ← below (p) {drop down}
        while key (after (p)) ≤ k do
          Let p ← after (p) {scan forward}
      end while
end while
  return p
```

Algorithm 1: A generic search in a skip list S

In the next skip list search algorithm, we search for an element by traversing forward pointers that do not go past the node with the element that's being searched for. When no more progress can be done at the current level, the forward pointers and search continues to the next level down. When no more progress can be made at the first level, the process moves to the front of the node that contains the element being searched for if it is inside the list.

Search (list, searchKey)

```
    x := list→header
    -- loop invariant: x→key < searchKey
    for i := list→level downto 1 do
    while x→forward[i]→key < searchKey do
        x := x→forward[i]
    -- x→key < searchKey ≤ x→forward[1]→key
    x := x→forward[1]
    if x→key = searchKey then return x→value
  else return failure
```

Algorithm 2: Search algorithm for skip list

3.2. Insertion

The insertion algorithm uses randomization to decide how many references to the new item (k, e) should be added to the skip list. The insertion begins with a new item (k, e) into a skip list by performing a search operation. This allows for the position p of the bottom-level item with the largest key less than or equal to k. (k, e) is then inserted in this bottom-level list immediately after position p. When the item is finished being inserted, the random() function is called which returns a number between 0 and 1. This is where we are hypothetically flipping a coin. If the number is less than ½, the flip is considered as "heads", otherwise it is considered as "tails". If the flip is tails, the algorithm stops. If heads, then the algorithm goes back to the next higher level and inserts the new item in this level at its correct position. We repeat the flipping process again and move higher if heads. The insertion continues until we get to a flip that is considered "tails".

```
Algorithm SkipInsert (k, e):
p ← SkipSearch (k)
q ← insertAfterAbove (p, null, (k, e))
while random () < 1/2 do
  while above (p) = null do
p ← before (p) {scan backward}
  end while
  p ← above (p) {jump up to higher level}
  q ← insertAfterAbove (p, q, (k, e)) end while
```

Algorithm 3: Insertion, assuming random () returns a value between 0 and 1, insertion never happens after top level

3.3. Deletion

If an insertion generates a node with a level greater than the previous maximum level of the list, we update the maximum level of the list and initialize the appropriate portions of the update vector. After every deletion made, the maximum level of the list is decreased and the maximum element is checked for whether the deletion is confirmed.

```
Delete(list, searchKey)
local update[1..MaxLevel]
x := list→header
for i := list→level downto 1 do
while x→forward[i]→key < searchKey do
  x := x→forward[i]
update[i] := x
x := x→forward[1]
if x→key = searchKey then
  for i := 1 to list→level do
          if update[i]→forward[i] ≠ x then
        break
    update[i]→forward[i] := x→forward[i]
      free(x)
     while list→level > 1 and
    list→header→forward[list→level] =
    NIL do
list→level := list→level – 1
```

Algorithm 4: Skip list deletion

The time required to execute these operations is dominated

by how much time it takes to search for the correct element. For insertion and deletion, there is an additional cost proportional to the level of the node being inserted or deleted. The required time for finding an element is proportional to the length of the search path, which is determined by the pattern in which elements with various levels show up during the list traversal.

IV. Alternative Data Structures

Balanced trees and self-adjusting trees are alternatives to skip lists and can be used for the same type of problems. Both types of trees have performance bounds of the same order. The comparison and contrast between these structures and skip lists are factors such as: difficulty of implementing the proposed algorithms, constant factors such as speed, space, and search cost, types of performance bounds, and performance on a non-uniform distribution of queries.

4.1. Difficult Implementation

For a majority of applications, those who implement skip lists agree that the implementation of skip lists is much easier to do compared to balanced tree algorithms or self-adjusting tree algorithms. The primary reason for more difficulty in implementing balanced tree algorithms is the high demand of keeping balance. Keeping a tree balanced is a slow operation that consumes plenty resources and must be done carefully.

4.2. Constant Factors

Constant factors can make a big difference in the application of an algorithm. This is especially true for sub-linear algorithms. For example, assume that we have algorithms A and B that both require O (log n) time to process a query, but that B is twice as fast as A. In the time it takes algorithm A to process a query on a data set of size n, algorithm B can process a query on a data set of size n_2. There are two important but qualitatively different contributions to the constant factors of an algorithm. First, the inherent complexity of the algorithm places a lower bound on any implementation. Self-adjusting trees are continuously rearranged as searches are performed, which imposes a significant overhead on any implementation of self-adjusting trees. Skip list algorithms seem to have very low inherent constant-factor overheads, the inner loop of the deletion algorithm for skip lists compiles to just six instructions.

TABLE I: Relative Search Speed And Space Requirements, Depending On The Value Of P

p	Normalized search times (normalized L(n)/p)	Average # of pointers per node (1/(1-p))
½	1	2
1/e	0.94…	1.58…
1/4	1	1.33
1/8	1.33…	1.14…
1/16	2	1.07…

Second, if the algorithm is complex, users are deterred from recreating optimizations. For example, balanced tree algorithms are normally described using recursive insert and delete procedures, since that is the most simple and perceptive method of describing the algorithms. A recursive insert or delete procedure incurs a procedure call overhead. By using non-recursive insert and delete procedures, some of this overhead can be eliminated. However, the complexity of non-recursive algorithms for insertion and deletion in a balanced tree is intimidating and this complexity deters most people from eliminating recursion in these routines. Skip list algorithms are already non-recursive and they are easy enough that those programming are not disappointed from performing optimizations.

Table 2 below compares the performance of implementations of skip lists and four other techniques. All implementations were optimized for efficiency. The AVL tree algorithms were written by James Macropol of Contel and the 2–3 tree algorithms are based on those also presented in [11]. Several other existing balanced tree packages were timed and found to be a lot slower than the results presented below. The self-adjusting tree algorithms are based on those presented in [10]. The times in this table reflect the CPU time performing an operation in a data structure containing 216 elements with integer keys. The values in parentheses show the results relative to the skip list time. The times for insertion and deletion do not include the time for memory management.

TABLE II
Timings Of Implementation Of Compared Algorithms

Implementation	Search Time	Insertion Time	Deletion Time
Skip lists	0.051 msec (1.0)	0.065 msec (1.0)	0.059 msec (1.0)
Non-recursive AVL trees	0.046 msec (0.91)	0.10 msec (1.55)	0.085 msec (1.46)
Recursive 2-3 trees	0.054 msec (1.05)	0.21 msec (3.2)	0.21 msec (3.65)
Self-adjusting trees			
Top-down splaying	0.15 msec (3.0)	0.16 msec (2.5)	0.18 msec (3.1)
Bottom-up splaying	0.49 msec (9.6)	0.51 msec (7.8)	0.53 msec (9.0)

Skip lists perform more comparisons than other methods. The skip list algorithms presented here require an average of L (n)/p + 1/ (1–p) + 1 comparisons. For tests using real numbers as keys, skip lists were slightly slower than the non-recursive AVL tree algorithms and search in a skip list was slower than search in a 2–3 tree. Insertion and deletion for skip list algorithms was still quicker than using the recursive 2–3 tree algorithms. If comparisons are costly, it is possible to change the algorithms so that we never compare the search key against the key of a node more than once during a search. For p = 1/2, this produces an upper bound on the expected number of comparisons of $7/2 + 3/2 \log_2 n$.

4.3. Performance Bounds

Balanced trees have worst-case time bounds, self-adjusting trees have amortized time bounds, and skip lists have probabilistic time bounds. An individual operation for a self-

adjusting tree will take O (n) time, but the time bound always holds over a long sequence of operations. For skip lists, any operation or sequence of operations can take longer than expected, although the probability of operations taking much longer than expected is inconsequential. For some real-time productions, it's assured that an operation will complete within a certain time bound. For such applications, self-adjusting trees may be undesirable, since they run longer than expected. For example, an individual search can take O (n) time instead of O (log n) time.

4.4. Non-uniform Query Distribution

Self-adjusting trees have the property that they adjust to non-uniform query distributions [3]. Since skip lists are faster than self-adjusting trees by a significant amount when a uniform query distribution is encountered, self-adjusting trees are faster than skip lists only for highly skewed distributions. It's possible to create self-adjusting skip lists, but the tampering of simplicity has not been a desire. In an application where highly skewed distributions are expected, either self-adjusting trees or an augmented skip list may be preferred.

V. Conclusion

In theory, it can be said that there is no need for skip lists. Balanced trees can do everything that skip lists have the ability to do and have good worst-case time bounds unlike skip lists. However, implementing balanced tree algorithms is a demanding task and as a result balanced tree algorithms are hardly implemented by those working on real world applications and systems. Skip lists are a simple data structure that is seen as a better alternative to balanced trees for most applications. There is decent simplicity in using skip list algorithms to implement and modify for various purposes. Skip lists are about as fast as highly optimized balanced tree algorithms and are considerably faster than casually implemented balanced tree algorithms.

VI. Related Work

Many papers have been written on implementing concurrent search structures using search trees, both balanced and unbalanced. Those for balanced trees [4, 5] tend to be very complicated, requiring exclusive locks and read locks, and allowing at most O (log n) busy writers. Some of the concurrency schemes for unbalanced trees [6, 7, and 8] allow O (n) busy writers and are easier than concurrent balanced tree schemes. However, certain input patterns can easily cause bad performance and the concurrency algorithms presented for skip lists appear simpler and allow as much or more concurrency. It does not seem that skip lists are particularly well suited for disk-based data structures, so this work does not provide any direct competition for concurrent B-trees.

Acknowledgment

I would like to say thank you to Professor Sanguthevar Rajasekaran for his instruction during the spring 2016 semester as well as introducing the topics during the Randomization in Computing course.

References

[1] Pugh, W. Skip Lists: A Probabilistic Alternative to Balanced Trees. Algorithms and Data Structures: Workshop WADS '89, Ottawa, Canada, August 1989, Springer-Verlag Lecture Notes in Computer Science 382, 437-449. (Revised version to appear in Comm. ACM).

[2] Goodrich, M. and Tamassia, R., Simplified Analyses of Randomized Algorithms for Searching, Sorting, and Selection.

[3] Aragon, Cecilia and Raimund Seidel, Randomized Search Trees, Proceedings of the 30th Ann. IEEE Symp on Foundations of Computer Science, pp 540–545, October 1989.

[4] Ellis, C. Concurrent search and insertion in AVL trees, IEEE Trans. on Comput. C-29 (Sept. 1980) 811-817.

[5] Ellis, C. Concurrent search and insertion in 2-3 trees. Acta Inf. 14 (1980) 63-86.

[6] Kung, H.T. and Lehman, Q. Concurrent Manipulation of Binary Search Trees, ACM Trans. on Database Systems, Vol. 5, No. 3 (Sept. 1980), 354-382.

[7] Manber, U. Concurrent Maintenance of Binary Search Trees, IEEE Transactions on Software Engineering, Vol. SE-10, No. 6 (November 1984), 777-784.

[8] Manber, U. and Ladner, P. Concurrent Control in a Dynamic Search Structure, ACM Trans. on Database Systems, Vol. 9, No. 3 (Sept 1984), 439-455.

[9] Pugh, W., Concurrent Maintenance of Skip Lists, Tech Report TR-CS-2222, Dept. of Computer Science, University of Maryland, College Park, 1989.

[10] Sleator, D. and R. Tarjan "Self-Adjusting Binary Search Trees," Journal of the ACM, Vol 32, No. 3, July 1985, pp. 652-666.

[11] Wirth, N. Algorithms + Data Structures = Programs, Prentice-Hall, 1976.