Double skip-list

ABSTRACT

Skip-List is a linked list, in which, each node holds one or more links to successor nodes including the immediate one. It was originally introduced as a probabilistic list-based substitution for balanced trees. In this paper we introduce what we called double skip-list, which is a modified skip-list that takes skip-list more levels up from its original target of just being an efficient emulation of a balanced tree. We augment the ordinary skip-list with another set of forward links that enables it to keep both the user specified and the sorted orderings of its contents, by which we add to the set of operations that could be performed on the skip-list in a logarithmic time complexity. To overcome the extra memory usage due to using double sets of links, we propose an analysis for changing the randomization factor of the double skip-list to keep extra memory usage as minimum as possible, while still keeping the same logarithmic time complexity for the considered set of operations. This set includes random access, position-based insertion/deletion, and searching. The proposed modification of the skip-list promotes the skip-list to be the first data structure that offers a logarithmic time complexity for the mentioned set of operations.

Keywords

Skip-List, Balanced Search Trees, Time complexity

1. INTRODUCTION TO SKIP-LIST

By the end of 1989 William Pugh[9, 10] introduced a smart data structure that is a simple and an efficient probabilistic list-based substitution for balanced trees[1, 4]. A Skip-List is a linked list in which each node has links for multiple successor nodes including the immediate next one. Links for successor nodes other than the immediate one provide virtual express lanes that are used to skip visiting some of the unnecessary nodes. Each node has a maximum of $Log_{1/p}N$ links each lies on a level ranges from 1 to $Log_{1/p}N$, where p is the randomization factor of the Skip-List. If a node contains a link at level L then it contains a strictly equal or shorter link at level $L-1$ for each $L > 1$. Forward links are chosen using a probabilistic approach such that p^i of the total number of nodes contains i outgoing links, that is when $p = 1/2$, 50% of the nodes contain only one forward link, 25% contain two links, 12.5% contain three links, etc. This is achieved simply by extending a node to contain a forward link at level L with a fixed probability p, only if it contains a link at level $L-1$. The most common used values for p are $1/2$ and $1/4$. In general, decreasing the value of p definitely decreases the memory usage, but mostly increases the running time. The work presented in [8] provides an analysis for the effect of changing the value of p on the searching cost. According to this analysis the theoretic optimum randomization factor p is $1/e$, however, as using powers of $1/2$ is more efficient, researchers always tend to use one of the values $1/2$ or $1/4$. In our usage and analysis for skip-list, p is always equal to $1/2$.

An augmented skip-list is a special skip-list in which every link holds its length. The length of a link is the number of nodes that it skips. Keeping the length of each link enables a logarithmic time complexity for visiting any node in the list using its position.

Skip-Lists have two modes of operation,

1. Value oriented: This mode is a typical emulation of a balanced search tree. It uses the values in the list as the keys and always keeps these values sorted. Choosing a specific position for a newly inserted element can not be done when operating the skip-list in this mode, as the new value goes into the position that will keep the list sorted.
2. Position oriented: This mode is an emulation of a dynamic array. It uses the positions of the elements as the keys. To insert an element into a skip-List that is operating in this mode, a position has to be provided, at which the new element would be placed. Operating the skip-list in the second mode pushes the time complexity of the search operation to $O(N)$ due to the fact that elements in the list are no more sorted.

As an abstract difference between the two modes, the first mode navigates through the list to the appropriate position using an element's value, while the second mode does the navigation using an element's position. Only augmented skip-lists are able to use the second mode efficiently, as

lengths of links are necessary to navigate to a specific position without having to visit all the preceding nodes. Figures 1 and 2 and Table 1 illustrates the difference between the two different modes along with the time complexity for performing different operations using each of them.

Although it was initially introduced as an alternative for balanced trees, due to its efficiency and most importantly simplicity of implementation, skip-list managed to relatively replace balanced trees and is now widely used in many applications such as peer-to-peer networking, concurrent data structures and different information retrieval applications [12, 13, 5, 2].

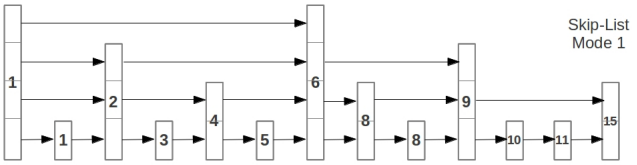

Figure 1: Skip-List operated in the first mode

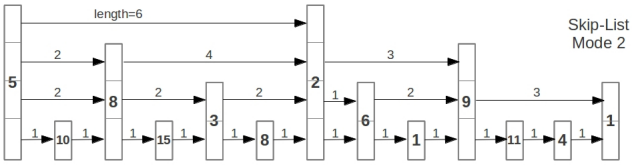

Figure 2: Augmented Skip-List operated in the second mode

	First Mode	Second Mode
Random access	expected $LogN$	expected $LogN$
Position-Based Insertion	NA	expected $LogN$
Position-Based Deletion	expected $LogN$	expected $LogN$
Searching	expected $LogN$	$O(N)$

Table 1: Time complexity for different operations on a skip-list. N is the number of elements in the list.

1.1 Proof of a logarithmic time complexity of the skip-list[9, 11]

As shown in table 1, skip-list along with its two modes of operations offer expected logarithmic time for a number of operations. In order to prove how skip-list achieves this performance, we will use a probabilistic analysis approach to study its time complexity. The analysis consists of two steps, the first is to prove that, with a high probability, the height of a skip-list is logarithmic of its size, and the second is to calculate the expected cost of performing the search operation. Recall that in our analysis p has the value $1/2$ and N is the number of nodes in the skip-list.

1.1.1 Proof of a logarithmic height of the Skip-List

- In order for a node to have a forward link at level L, L successful upward promotions must be performed, each upward promotion occurs with a probability p.
- The probability of a node to have more than $c.Log_{1/p}N$ forward links is $(p)^{c.Log_{1/p}N} = 1/N^c$.
- The probability of at least one node contains more than $c.Log_{1/p}N$ links is the union of the probability of individual nodes to contain more than $c.Log_{1/p}N$ links. That is, $N/N^c = 1/N^{c-1}$.
- The height of a Skip-List is $c.Log_{1/p}N = O(Log_{1/p}N)$ with a probability of $1-(1/N^{c-1})$. Choosing an arbitrary value for c keeps the probability of a logarithmic height quite large.

1.1.2 The expected search cost

Based on the above argument, it is quite safe to limit the growth of a Skip-List height to $Log_{1/p}N$. The search cost is determined by the path from the Skip-List root node to the required node or position. We can analyse this path backwardly starting from the destination node to the root node. At any particular point in the search path (that is k levels away from the top level), two situations may happen:

- Go left, so we still need to climb up more k levels. (situation a)
- Go up one level, so we need to climb up more k-1 levels. (situation b)

Situation b has probability p, which is the same probability of promoting a node one level up when we create it. Similarly situation a has a probability $1-p$. Let's define $C(k)$ as the expected cost (length) of a search path to climb up k levels, then the expected cost of performing the search operation is $C(Log_{1/p}N)$.

$C(0) = 0$

$C(k) = (1-p)(cost\ situ.\ a) + p(cost\ situ.\ b)$

$C(k) = (1-p)(1+C(k)) + p(1+C(k-1))$

$C(k) = 1/p + C(k-1)$

$C(k) = k/p$

The expected cost of the search is $C(Log_{1/p}N) = (Log_{1/p}N)/p$, hence for $p = 1/2$, the expected cost of the search is $2.Log_2N$. As shown in figure 3, William Pugh the inventor of the Skip-List, provided a probabilistic analysis of the situation when the actual search cost exceeds the expected one. The analysis shows that the probability of the actual search cost to substantially exceed the theoretical expected time complexity is too small, specially in skip-lists with large number of nodes. A number of researches have been conducted to offer a guaranteed worst case time complexity for different operations on the skip-list[7, 6, 3], however most applications tends to be satisfied with the very close to real expected time complexity achieved by simple implementations with simple randomization technique.

What about memory? Skip-list was originally proposed as a competitor for balanced trees, however, as some nodes in a skip-list may contain upto $Log_{1/p}N$ forward pointers, a skip-list may be thought to consume more memory than a typical balanced tree. In fact, it does not. Recalling the

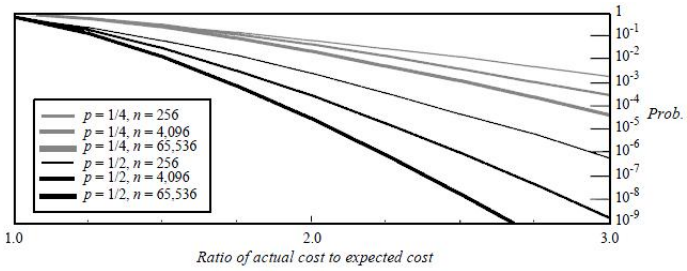

Figure 3: This graph shows a plot of an upper bound on the probability of a search taking substantially longer than expected. The vertical axis show the probability that the length of the search path for a search exceeds the average length by more than the ratio on the horizontal axis. For example, for p = 1/2 and n = 4096, the probability that the search path will be more than three times the expected length is less than one in 200 million.[9]

fact that in an optimally randomized skip-list, p^i of the total number of nodes contains i forward links, concludes that the total number of forward links T in an optimally randomized Skip-List of size N with a randomization factor p could be formulated as:

$$T = N(1 + \sum_{i=1}^{Log_{1/p}(N)} p^i)$$

$$\text{let } S = \sum_{i=1}^{Log_{1/p}(N)} p^i$$

$$T = N(1 + S)$$

$$S = p(1 + \sum_{i=1}^{Log_{1/p}(N)-1} p^i)$$

$$S = p(1 + S - p^{Log_{1/p}(N)})$$

$$S = (p - p^{Log_{1/p}(N)+1})/(1 - p)$$

$p^{Log_{1/p}(N)+1}$ can be neglected for relatively large values of N.

$$S = p/(1 - p)$$

$$T = N(1 + \frac{p}{1-p})$$

Thus in a skip-list with a randomization factor $p = 1/2$ the expected number of forward links equals to $2N$. Given that a node in a balanced tree must hold at least pointers for its left and right children along with a pointer for its parent, then an optimally randomized skip-list with $p = 1/2$ outperforms a balanced binary search tree by almost 33% of the total memory usage. As mentioned in the time complexity analysis of the Skip-List, simple methods of randomization can keep a skip-list's nodes very close to the optimal distribution at different levels, especially for skip-lists with large number of nodes.

A Skip-List may even consume less memory by changing its randomization factor p. In general, decreasing the value of p decreases the memory usage, but pays off in terms of running time. Table 2 shows the average number of pointers per node for different values of p.

p	*Avg # of pointers per node*
1/2	2
1/4	1.33
1/8	1.14
1/16	1.07

Table 2: Average number of pointers per node for different values of p.[9]

2. DOUBLE SKIP-LIST

Ordinary skip-list offers two sets of operations depending on the order in which it keeps its contents. For the sake of simplicity, in any future analysis we will consider only augmented skip-list in which every link holds its length. A possible implementation of the skip-list keeps the nodes sorted while the other keeps the nodes in the same order that they were inserted with. Each implementation supports one of the two contradicting operations, random position insertion and searching. Keeping the nodes sorted enables a logarithmic time for the search while disables the random position insertion. On the other hand, keeping the user specified order of nodes enables a logarithmic time random position insertion but pushes the searching time complexity to linear time. Double skip-list, on the other hand, is a modified skip-list that operates in both modes simultaneously. Typically it should be used in the situation when an indexable list with extensive insertions, deletions and searching queries are required. For the sake of clarity we redefine four main operations that we aim to optimize by proposing double skip-list:

1. Random access. It is the ability to fetch an item from the list using its position.
2. Random position insertion. It is the ability to insert an item into a list at a specific position.
3. Random position deletion. It is the ability to delete the item that exists at a specific position from a list.
4. Searching. It is the ability to search in a list for a specific item and return its position if it exists, or nil otherwise.

Double skip-list is capable of performing each of these operations in an expected logarithmic time complexity, while every other data structure either needs linear time or does not support at least one of these operations.

2.1 Structure

Double skip-list is basically two skip-lists that share their nodes. As shown in figure 4, each node in a double skip-list belongs to two virtual skip-lists, each of them keeps a different permutation of the entire set of nodes. A typical double skip-list's node consists of the following fields:

- Data: This is the actual element value.
- Set A of forward links: A set of pointers each points to a successor node in the sorted set of nodes. Similar to an ordinary skip-list, double skip-list is randomized with a randomization factor p such that $N * p^i$ of the total number of nodes carry i pointers of this type for $1 \leq i \leq Log_{1/p}N$, where N is the current number of nodes in the list. Beside the actual pointer, each link holds an integer representing its length.
- Set B of forward links: A set of pointers each points to a successor node in the original, user provided, ordering of the set of nodes. Similar to set A, $N * p^i$ of the total number of nodes carry i pointers of this type for $1 \leq i \leq Log_{1/p}N$. Similarly, each link in set B holds its length represented as an integer.

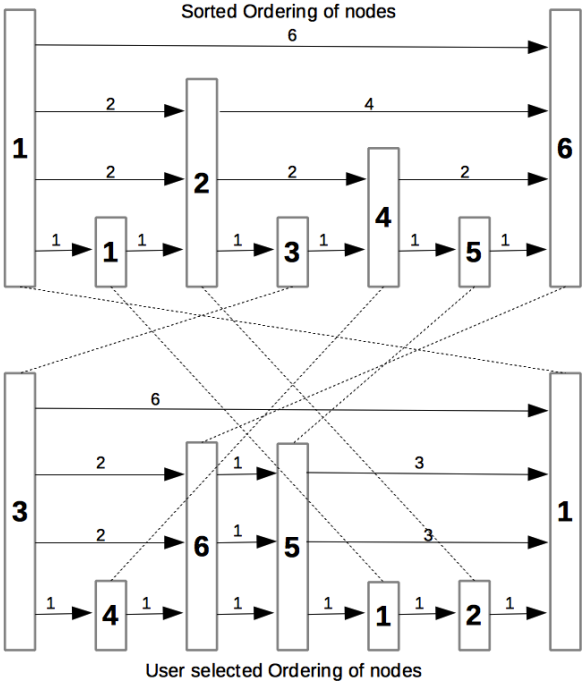

Figure 4: Internal structure of double skip-list. Each node is drawn as two virtual nodes, one in the sorted ordering of the set of nodes and the other in the user selected ordering. In reality, a single element is represented by a single node that has two sets of links by which it is attached to both the two different orderings.

2.2 Random access

As defined earlier, random access is the ability to fetch an element by its position. As double skip-list preserves both the user selected and the sorted orderings of its contents, an element could be fetched by its position in any of them. The random access operation works as the following:

1. Depending upon the targeted scenario, start by the root (first node) of the corresponding ordering, and use set A or B of links accordingly .
2. Check the highest level link. If it goes less than or equal to the required position, then go through this link. Otherwise, check the shorter links, in order, till one of them leads to a position that is less than or equal to the required one, use this link to go to the corresponding successor node.
3. Repeat the previous step for every node that is reached out by step two. For each of these nodes start checking the forward link that is on the same level as the last used link.
4. Stop when the required position is reached out. Note that every node has a level one forward link which always leads to the immediate successor node. This means that every node is reachable from any of its precedents.

The time complexity of the random access, as inherited from the time analysis of the skip-list, is expected logarithmic ($LogN$).

2.3 Insertion

Insertion in double skip-list is defined the same way as it is defined in an ordinary array. To insert into a double skip-list, the new value along with its position should be provided. The data structure then updates its two internal orderings of the set of nodes to reflect the new insertion. The following procedure describes how a new element is inserted in a double skip-list,

1. A new node is created and the new value is assigned to the data field of this node.
2. The node is then placed in the skip-list. First we navigate through set B of links to the provided position at which the new node should be placed using a similar method to the one that is adapted in the random access.
3. Set B of links is created for the newly created node. Links at different levels are created sequentially starting by the first level.
4. For the first level, a forward link is added that points to the immediate successor node. The forward link of the immediate preceding node is then updated to the new node. All links at level one have length one.
5. For each of the next levels toss a coin (to simulate a randomization factor $p = 1/2$), if it is a head add a forward link at this level and proceed to the next level. If it is a tail stop adding links to the node and go to step 6. Adding forward links at any level other than level one is more complicated. The process is done by breaking the link at the same level that bypasses the new node into two smaller links which both have the new node as a start or end station. When inserting a new node at the first position, forward links must be added at all levels with no need to toss a coin for deciding whether to go one level up or not.

6. The new node is then linked to the sorted ordering of the nodes exactly the same way as described in steps 2 through 5. Set A of links is used for this step instead of set B.

Insertion in a double skip-list is a little bit expensive compared to the random access, however, it is still performed in an expected logarithmic time complexity ($LogN$).

2.4 Searching

Searching in a double skip-list is performed on two steps, the first is to find the required node through the sorted ordering of the nodes, and the second is to calculate its position in the unsorted sequencing. Each step requires $LogN$ iterations and hence the total time complexity of the searching operation is expected $LogN$. The following steps describe the process in details,

1. Navigate to the required element in the sorted ordering using set A of links, until the element is reached out or the end of the sorted set is reached out.
2. If the required node is found, switch to the unsorted ordering of the set of nodes, starting from the found node navigate to the end of the unsorted ordering through set B of links, accumulate lengths of the used links while navigating.
3. Position of the required element= Total number of nodes - Accumulated lengths.

2.5 memory overhead

Double skip-list, by nature, consumes more memory than an ordinary skip-list. This is due to the fact that every node holds two sets of links to help preserve the two orderings of the whole set of nodes. When preserving the same randomization factor p, double skip-list typically consumes double the memory size that is used by an ordinary skip-list. A good approach to overcome this drawback is to use a smaller value for p. Theoretically, as described in the skip-list analysis, the minimum number of links necessary to keep the $(Log_{1/p}N)/p$ time complexity for the search operation is $N(1 + \frac{p}{1-p})$. Hence using the value $1/4$ for p instead of $1/2$ could save approximately 33% of the memory usage while still preserving the same running time. Figure 5 shows the theoretical effect of changing the vale of p on the search cost.

3. RESULTS

Double skip-list was extensively tested against an ordinary skip-list that is operated in mode two, that is as described before, a skip-list that preserves the user selected ordering of its contents. Three operation were mainly considered in the comparison, namely, Positional Insertion, Random Access and Searching. As described earlier, an ordinary skip-list that keeps its elements sorted does not support the first operation, however it can perform the other two operations in a logarithmic time, giving that positions are always relative to the sorted ordering of the elements. On the other hand, an ordinary skip-list that keeps the user selected ordering can perform the first two operations in logarithmic time, while it suffers from a linear time for the third one. In contrary to

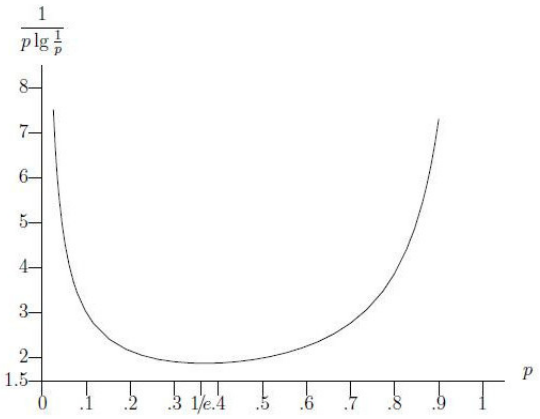

Figure 5: Theoretical searching cost Vs. different randomization factors[8]

ordinary skip-list along with its two modes, double skip-list is capable of performing the three operations in logarithmic time.

Tables 3, 4 and 5 illustrates a running time comparison for the three different operations on both a mode two ordinary skip-list and the proposed double skip-list. The comparison shows that both of them have very close performance doing the insertion and the random access, while the searching operation makes a great difference in the overall performance. This is basically due to the linear vs logarithmic time complexity difference between the two data structures.

Size	*Ordinary Skip-List*	*Double Skip-List*
10^4	0.01 s	0.01 s
10^5	0.21 s	0.23 s
$5 * 10^5$	1.0 s	1.4 s
10^6	2.3 s	2.9 s
10^7	38.8 s	48.7 s

Table 3: Average time consumed to insert N elements into initially empty lists

Size	*Ordinary Skip-List*	*Double Skip-List*
10^4	0.01 s	0.01 s
10^5	0.01 s	0.01 s
$5 * 10^5$	0.2 s	0.2 s
10^6	0.4 s	0.4 s
10^7	5.5 s	5.5 s

Table 4: Average time consumed to perform the random access operation N times on a list of size N nodes

4. CONCLUSIONS

A novel skip-list structure is proposed that promotes the skip-list to be the first data structure to offer a logarithmic time complexity for the a set of operations namely, random access, position-based insertion, position-based deletion, and searching. The new structured skip-list was given the name double skip-list as a notation for its behavior of

Size	*Ordinary Skip-List*	*Double Skip-List*
10^4	1.4 s	0.01 s
10^5	940 s	0.1 s
$5 * 10^5$	INF	0.9 s
10^6	INF	2.2 s
10^7	INF	39.5 s

Table 5: Average time in seconds consumed to perform the searching operation N times on a list of size N nodes

keeping two different permutations for its contents. Extensive theoretical and practical comparisons have been conducted against the ordinary skip-list. Results showed that double skip-list is slightly slower doing the insertion but extremely faster performing the searching. As double skip-lists tends to consume more memory than an ordinary skip-list, a detailed time vs. memory analysis was proposed which concluded that double skip-list may consume only 33% of the original memory consumed by an ordinary skip-list while preserving the same running time. For applications which can sacrifice this price, double skip-list is a great option specially when both the contradicting operations, position-based insertion and searching, are extensively required.

5. REFERENCES

[1] M. AdelsonVelskii and E. M. Landis. *An algorithm for the organization of information*. Defense Technical Information Center, 1963.

[2] J. Aspnes and G. Shah. Skip graphs. *ACM Trans. Algorithms*, 3(4), Nov. 2007.

[3] A. Bagchi, A. L. Buchsbaum, and M. T. Goodrich. Biased skip lists. *Algorithmica*, 42:31–48, 2005.

[4] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[5] P. Boldi and S. Vigna. Compressed perfect embedded skip lists for quick inverted-index lookups. In M. Consens and G. Navarro, editors, *String Processing and Information Retrieval*, volume 3772 of *Lecture Notes in Computer Science*, pages 25–28. Springer Berlin Heidelberg, 2005.

[6] T. Clouser, M. Nesterenko, and C. Scheideler. Tiara: A self-stabilizing deterministic skip list. In S. Kulkarni and A. Schiper, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 5340 of *Lecture Notes in Computer Science*, pages 124–140. Springer Berlin Heidelberg, 2008.

[7] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, SODA '92, pages 367–375, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[8] T. Papadakis. *Skip Lists and Probabilistic Analysis of Algorithms*. PhD thesis, University of Waterloo, 1993.

[9] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[10] W. Pugh. A skip list cookbook. 1998.

[11] S. Sen. Some observations on skip-lists. *Information Processing Letters*, 39(4):173 – 176, 1991.

[12] D. Wang and J. Liu. A dynamic skip list-based overlay for on-demand media streaming with vcr interactions. *Parallel and Distributed Systems, IEEE Transactions on*, 19(4):503–514, April.

[13] D. Wang and J. Liu. Peer-to-peer asynchronous video streaming using skip list. In *Multimedia and Expo, 2006 IEEE International Conference on*, pages 1397–1400, July.