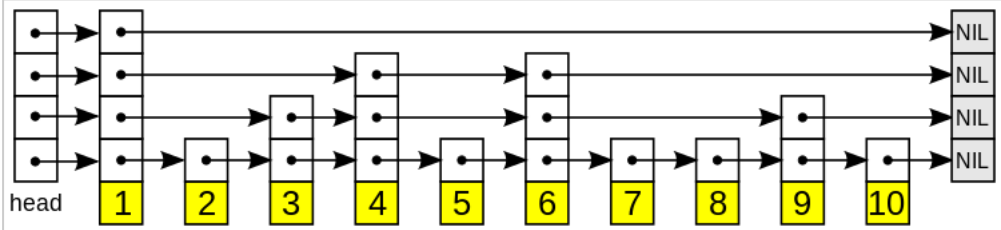
Skip list

From Wikipedia, the free encyclopedia

In computer science, a **skip list** is a data structure that allows fast search within an ordered sequence of elements. Fast search is made possible by maintaining a linked hierarchy of subsequences, with each successive subsequence skipping over fewer elements than the previous one. Searching starts in the sparsest subsequence until two consecutive elements have been found, one smaller and one larger than or equal to the element searched for. Via the linked hierarchy, these two elements link to elements of the next sparsest subsequence, where searching is continued until finally we are searching in the full sequence. The elements that are skipped over may be chosen probabilistically [2] or deterministically,[3] with the former being more common.

Skip List

Type	List	
Invented	1990	
Invented by	W. Pugh	
Time complexity in big O notation		
	Average	Worst case
Space	O(n)	O(n log n)[1]
Search	O(log n)	O(n)[1]
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)



A schematic picture of the skip list data structure. Each box with an arrow represents a pointer and a row is a linked list giving a sparse subsequence; the numbered boxes at the bottom represent the ordered data sequence. Searching proceeds downwards from the sparsest subsequence at the top until consecutive elements bracketing the search element are found.

Contents

Description

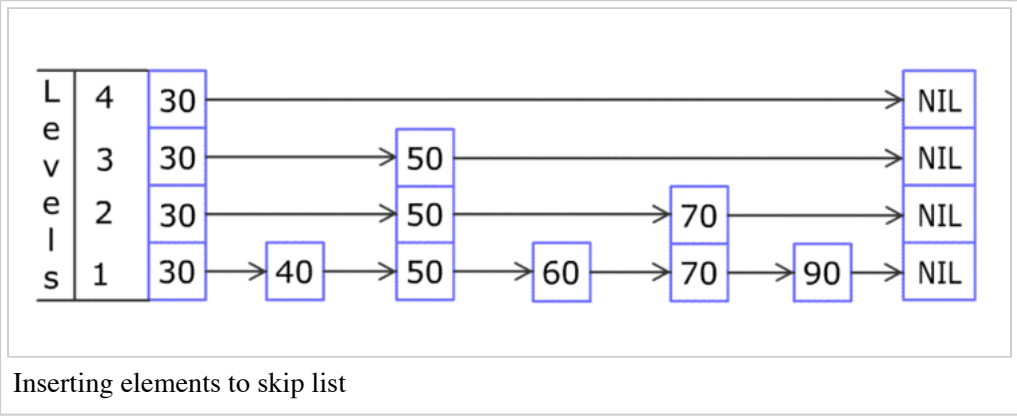
A skip list is built in layers. The bottom layer is an ordinary ordered linked list. Each higher layer acts as an "express lane" for the lists below, where an element in layer *i* appears in layer *i*+1 with some fixed probability *p* (two commonly used values for *p* are 1/2 or 1/4). On average, each element appears in 1/(1-*p*) lists, and the tallest element (usually a special head element at the front of the skip list) in all the lists. The skip list contains $\log_{1/p} n$ lists.

A search for a target element begins at the head element in the top list, and proceeds horizontally until the current element is greater than or equal to the target. If the current element is equal to the target, it has been found. If the current element is greater than the target, or the search reaches the end of the linked list, the procedure is repeated after returning to the previous element and dropping down vertically to the next lower list. The expected number of steps in each linked list is at most 1/*p*, which can be seen by tracing the search path backwards from the target until reaching an element that appears in the next higher list or reaching the beginning of the current list. Therefore, the total *expected* cost of a search is $(\log_{1/p} n)/p$, which is $\mathcal{O}(\log n)$ when *p* is a constant. By choosing different values of *p*, it is possible to trade search costs against storage costs.

Implementation details

The elements used for a skip list can contain more than one pointer since they can participate in more than one list.

Insertions and deletions are implemented much like the corresponding linked-list operations, except that "tall" elements must be inserted into or deleted from more than one linked list.



Inserting elements to skip list

$\mathcal{O}(n)$ operations, which force us to visit every node in ascending order (such as printing the entire list), provide the opportunity to perform a behind-the-scenes derandomization of the level structure of the skip-list in an optimal way, bringing the skip list to $\mathcal{O}(\log n)$ search time. (Choose the level of the i'th finite node to be 1 plus the number of times we can repeatedly divide i by 2 before it becomes odd. Also, i=0 for the negative infinity header as we have the usual special case of choosing the highest possible level for negative and/or positive infinite nodes.) However this also allows someone to know where all of the higher-than-level 1 nodes are and delete them.

Alternatively, we could make the level structure quasi-random in the following way:

```
make all nodes level 1
j ← 1
while the number of nodes at level j > 1 do
  for each i'th node at level j do
    if i is odd
      if i is not the last node at level j
        randomly choose whether to promote it to level j+1
      else
        do not promote
      end if
    else if i is even and node i-1 was not promoted
      promote it to level j+1
    end if
  repeat
  j ← j + 1
repeat
```

Like the derandomized version, quasi-randomization is only done when there is some other reason to be running an $\mathcal{O}(n)$ operation (which visits every node).

The advantage of this quasi-randomness is that it doesn't give away nearly as much level-structure related information to an adversarial user as the de-randomized one. This is desirable because an adversarial user who is able to tell which nodes are not at the lowest level can pessimize performance by simply deleting higher-level nodes. (Bethea and Reiter however argue that nonetheless an adversary can use probabilistic and timing methods to force performance degradation.[4]) The search performance is still guaranteed to be logarithmic.

It would be tempting to make the following "optimization": In the part which says "Next, for each i'th...", forget about doing a coin-flip for each even-odd pair. Just flip a coin once to decide whether to promote only the even ones or only the odd ones. Instead of $\mathcal{O}(n \log n)$ coin flips, there would only be $\mathcal{O}(\log n)$ of them. Unfortunately, this gives the adversarial user a 50/50 chance of being correct upon guessing that all of the even numbered nodes (among the ones at level 1 or higher) are higher than level one. This is despite the property that he has a very low probability of guessing that a particular node is at level *N* for some integer *N*.

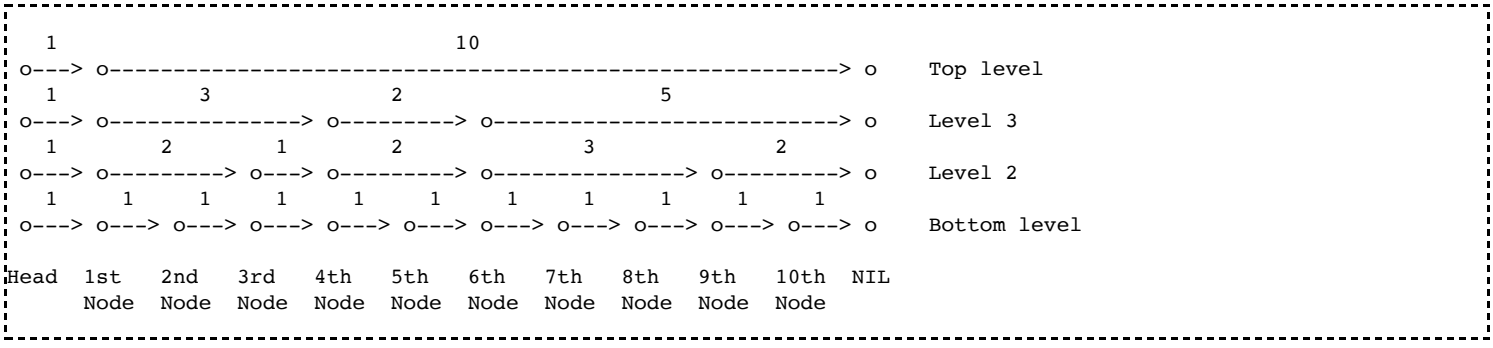
A skip list does not provide the same absolute worst-case performance guarantees as more traditional balanced tree data structures, because it is always possible (though with very low probability) that the coin-flips used to build the skip list will produce a badly balanced structure. However, they work well in practice, and the randomized balancing scheme has been argued to be easier to implement than the deterministic balancing schemes used in balanced binary search trees. Skip lists are also useful in parallel computing, where insertions can be done in different parts of the skip list in parallel without any global rebalancing of the data structure. Such parallelism can be especially advantageous for resource discovery in an ad-hoc wireless network because a randomized skip list can be made robust to the loss of any single node.[5]

Indexable skiplist

As described above, a skiplist is capable of fast $\mathcal{O}(\log n)$ insertion and removal of values from a sorted sequence, but it has only slow $\mathcal{O}(n)$ lookups of values at a given position in the sequence (i.e. return the 500th value); however, with a minor modification the speed of random access indexed lookups can be improved to $\mathcal{O}(\log n)$.

For every link, also store the width of the link. The width is defined as the number of bottom layer links being traversed by each of the higher layer "express lane" links.

For example, here are the widths of the links in the example at the top of the page:



Notice that the width of a higher level link is the sum of the component links below it (i.e. the width 10 link spans the links of widths 3, 2 and 5 immediately below it). Consequently, the sum of all widths is the same on every level (10 + 1 = 1 + 3 + 2 + 5 = 1 + 2 + 1 + 2 + 5).

To index the skiplist and find the i'th value, traverse the skiplist while counting down the widths of each traversed link. Descend a level whenever the upcoming width would be too large.

For example, to find the node in the fifth position (Node 5), traverse a link of width 1 at the top level. Now four more steps are needed but the next width on this level is ten which is too large, so drop one level. Traverse one link of width 3. Since another step of width 2 would be too far, drop down to the bottom level. Now traverse the final link of width 1 to reach the target running total of 5 (1+3+1).

```
function lookupByPositionIndex(i)
    node ← head
    i ← i + 1                          # don't count the head as a step
    for level from top to bottom do
         while i ≥ node.width[level] do # if next step is not too far
             i ← i - node.width[level]  # subtract the current width
             node ← node.next[level]    # traverse forward at the current level
         repeat
    repeat
    return node.value
end function
```

This method of implementing indexing is detailed in Section 3.4 Linear List Operations in "A skip list cookbook" by William Pugh (http://cg.scs.carleton.ca/~morin/teaching/5408/refs/p90b.pdf).

History

Skip lists were first described in 1989 by William Pugh.[6]

To quote the author:

> *Skip lists are a probabilistic data structure that seem likely to supplant balanced trees as the implementation method of choice for many applications. Skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space.*

Usages

List of applications and frameworks that use skip lists:

- MemSQL uses skiplists as its prime indexing structure for its database technology.
- Cyrus IMAP server offers a "skiplist" backend DB implementation (source file (http://git.cyrusimap.org/cyrus-imapd/tree/lib/cyrusdb_skiplist.c))
- Lucene uses skip lists to search delta-encoded posting lists in logarithmic time.
- QMap (http://qt-project.org/doc/qt-4.8/qmap.html#details) (up to Qt 4) template class of Qt that provides a dictionary.
- Redis, an ANSI-C open-source persistent key/value store for Posix systems, uses skip lists in its implementation of ordered sets.[7]
- nessDB (https://github.com/shuttler/nessDB), a very fast key-value embedded Database Storage Engine (Using log-structured-merge (LSM) trees), uses skip lists for its memtable.
- skipdb (http://www.dekorte.com/projects/opensource/skipdb/) is an open-source database format using ordered key/value pairs.
- ConcurrentSkipListSet (http://download.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentSkipListSet.html) and ConcurrentSkipListMap (http://download.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentSkipListMap.html) in the Java 1.6 API.
- Speed Tables (https://github.com/flightaware/speedtables) are a fast key-value datastore for Tcl that use skiplists for indexes and lockless shared memory.
- leveldb (https://github.com/google/leveldb), a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values

Skip lists are used for efficient statistical computations (http://code.activestate.com/recipes/576930/) of running medians (also known as moving medians). Skip lists are also used in distributed applications (where the nodes represent physical computers, and pointers represent network connections) and for implementing highly scalable concurrent priority queues with less lock contention,[8] or even without locking,[9][10][11] as well as lockless concurrent dictionaries.[12] There are also several US patents for using skip lists to implement (lockless) priority queues and concurrent dictionaries.[13]

See also

- Bloom filter
- Skip graph

References

1. http://www.cs.uwaterloo.ca/research/tr/1993/28/root2side.pdf
2. Pugh, W. (1990). "Skip lists: A probabilistic alternative to balanced trees" (PDF). *Communications of the ACM*. **33** (6): 668. doi:10.1145/78973.78977.
3. Munro, J. Ian; Papadakis, Thomas; Sedgewick, Robert (1992). "Deterministic skip lists". *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms (SODA '92)*. Orlando, Florida, USA: Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. pp. 367–375. alternative link (http://www.ic.unicamp.br/~celio/peer2peer/skip-net-graph/deterministic-skip-lists-munro.pdf)
4. Darrell Bethea and Michael K. Reiter, Data Structures with Unpredictable Timing https://www.cs.unc.edu/~djb/papers/2009-ESORICS.pdf, section 4 "Skip Lists"
5. Shah, Gauri (2003). *Distributed Data Structures for Peer-to-Peer Systems* (PDF) (Ph.D. thesis). Yale University.
6. William Pugh (April 1989). "Concurrent Maintenance of Skip Lists", Tech. Report CS-TR-2222, Dept. of Computer Science, U. Maryland.
7. "Redis ordered set implementation".
8. Shavit, N.; Lotan, I. (2000). "Skiplist-based concurrent priority queues". *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000* (PDF). p. 263. doi:10.1109/IPDPS.2000.845994. ISBN 0-7695-0574-0.
9. Sundell, H.; Tsigas, P. (2003). "Fast and lock-free concurrent priority queues for multi-thread systems". *Proceedings International Parallel and Distributed Processing Symposium*. p. 11. doi:10.1109/IPDPS.2003.1213189. ISBN 0-7695-1926-1.
10. Fomitchev, Mikhail; Ruppert, Eric (2004). *Lock-free linked lists and skip lists* (PDF). Proc. Annual ACM Symp. on Principles of Distributed Computing (PODC). pp. 50–59. doi:10.1145/1011767.1011776. ISBN 1581138024.
11. Bajpai, R.; Dhara, K. K.; Krishnaswamy, V. (2008). "QPID: A Distributed Priority Queue with Item Locality". *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*. p. 215. doi:10.1109/ISPA.2008.90. ISBN 978-0-7695-3471-8.
12. Sundell, H. K.; Tsigas, P. (2004). "Scalable and lock-free concurrent dictionaries". *Proceedings of the 2004 ACM symposium on Applied computing - SAC '04* (PDF). p. 1438. doi:10.1145/967900.968188. ISBN 1581138121.
13. US patent 7937378 (http://worldwide.espacenet.com/textdoc?DB=EPODOC&IDX=US7937378)

External links

- "Skip list" entry (https://xlinux.nist.gov/dads/HTML/skiplist.html) in the Dictionary of Algorithms and Data Structures
- Skip Lists: A Linked List with Self-Balancing BST-Like Properties (http://msdn.microsoft.com/en-us/library/ms379573(VS.80).aspx#datastructures20_4_topic4) on MSDN in C# 2.0
- Skip Lists lecture (MIT OpenCourseWare: Introduction to Algorithms) (http://videolectures.net/mit6046jf05_demaine_lec12/)
- Open Data Structures - Chapter 4 - Skiplists (http://opendatastructures.org/versions/edition-0.1e/ods-java/4_Skiplists.html)

- Skip trees, an alternative data structure to skip lists in a concurrent approach (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.514)
- Skip tree graphs, a distributed version of skip trees (http://www0.cs.ucl.ac.uk/staff/a.gonzalezbeltran/pubs/icc2007.pdf)
- More on skip tree graphs, a distributed version of skip trees (http://www0.cs.ucl.ac.uk/staff/a.gonzalezbeltran/pubs/AGB-comcom08.pdf)

Demo applets

- Skip List Applet (http://people.ksp.sk/~kuko/bak/index.html) by Kubo Kovac
- Thomas Wenger's demo applet on skiplists

Implementations

- Algorithm::SkipList, implementation in Perl on CPAN (https://metacpan.org/module/Algorithm::SkipList)
- Raymond Hettinger's implementation in Python (http://code.activestate.com/recipes/576930/)
- ConcurrentSkipListSet documentation for Java 6 (http://java.sun.com/javase/6/docs/api/java/util/concurrent/ConcurrentSkipListSet.html) (and sourcecode (http://www.docjar.com/html/api/java/util/concurrent/ConcurrentSkipListSet.java.html))

Categories: 1989 introductions | Linked lists | Probabilistic data structures

---