Skip Graphs and its Applications

Ajay Shankar Bidyarthy

b.ajay@iitg.ernet.in

09012305

Department of Mathematics

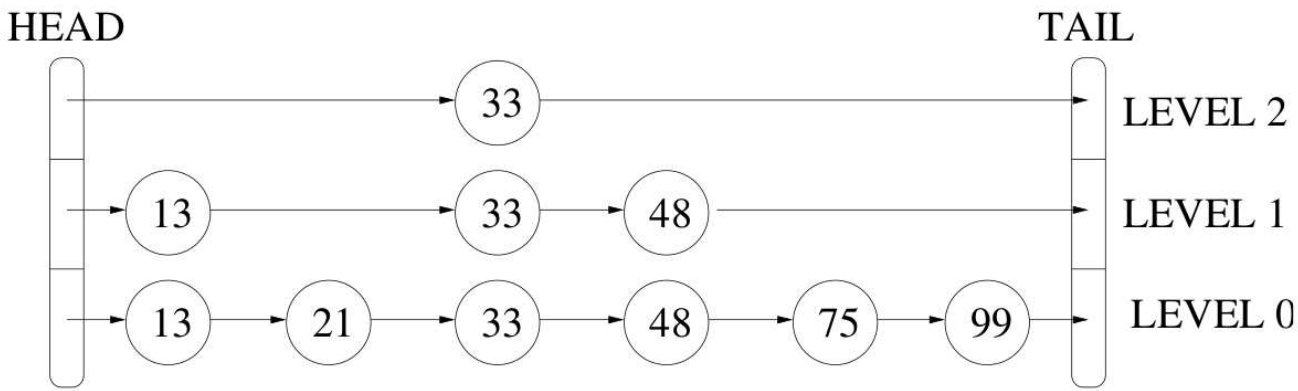
October 15, 2012

Topics to be covered

- Skip Lists
- Skip Graphs
- Search an existing node in a Skip Graphs
- Insert a new node in a Skip Graphs
- Delete an existing node from a Skip Graphs
- Skip Graphs Applications

Skip Lists

- A skip list, introduced by Pugh in 1990, is a randomized balanced tree data structure organized as a tower of increasingly sparse linked lists.



: Figure 1: A skip list with $n = 6$ nodes and $\lceil \log n \rceil = 3$ levels.

Skip Lists continue...

- In a doubly-linked skip list, each node stores a predecessor pointer and a successor pointer for each list in which it appears, for an average of $\frac{2}{1-p}$ pointers per node.

To to search for a data x, start with the leftmost node at the highest level and follow these steps:

Initialize: $\mathbf{v} = -\infty$

if $v.right > x$ **then**
| Move to the lower level

end

if $v.rigth < x$ **then**
| Move to the right

end

if $v.right = x$ **then**
| Element found // Search fails when no lower level exist.

end

Algorithm 1: Distributed algorithm to search an element x in a skip graph.

Skip Lists continue...

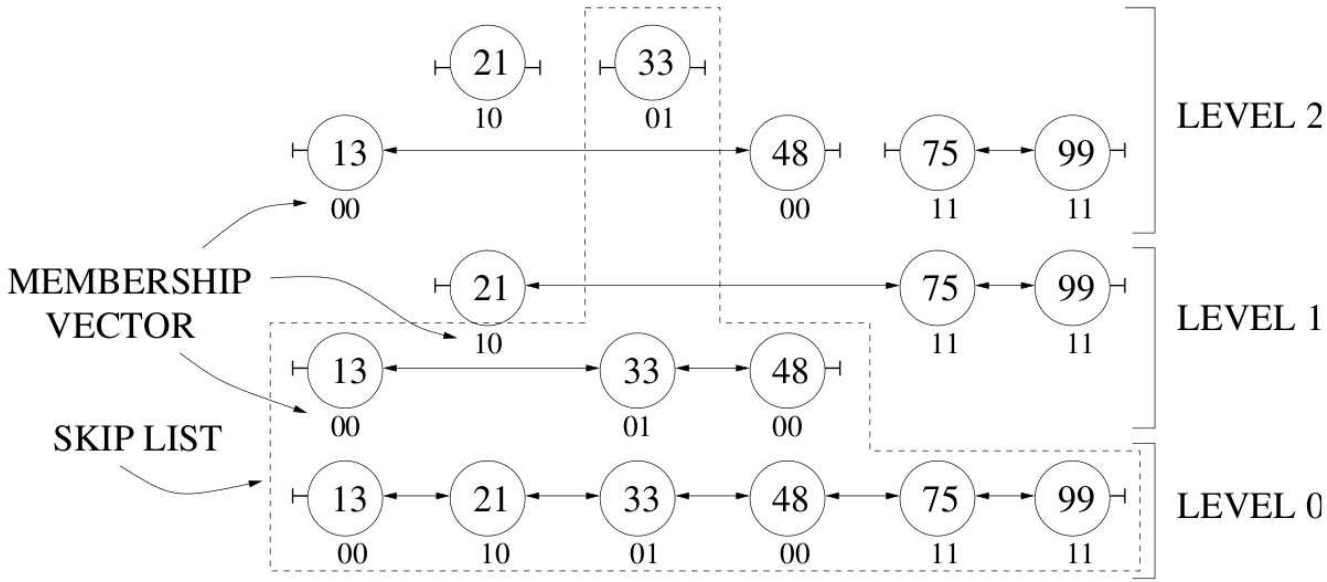
- Let N be the number of nodes in the respective graph. If the list is doubly linked, then the expected number of edges in the entire graph is

$$2N + N + N/2 + N/4 + ... = 4N.$$

The average degree of per node is still a constant, but the routing latency is reduced to $O(\log N)$

Skip Graphs

- Skip graph is a generalization of skip list



: Figure 2: A skip Graph with $n = 6$ nodes and $\lceil \log n \rceil = 3$ levels.

Skip Graphs continue...

- a skip graph is distinguished from a skip list is that there may be many lists at level i, and every node participates in one of these lists, until the nodes are splintered into singletons after $O(\log n)$ levels on average.

Skip Graphs continue...

- Which lists a node x belongs to is controlled by a membership vector $m(x)$.
- We think of $m(x)$ as an infinite random word over some fixed alphabet, in practice, only an $O(\log n)$ length prefix of $m(x)$ needs to be generated on average.
- The idea of the membership vector is that every linked list in the skip graph is labeled by some finite word w, and a node x is in the list labeled by w if and only if w is a prefix of $m(x)$.

Skip Graphs continue...

- Define A particular list S_w is part of level i if $|w| = i$. The bottom level is always a doubly-linked list S_ϵ consisting of all nodes in order as shown in figure 2.
- A skip graph is precisely a family $\{S_w\}$ of doubly-linked lists.
- Notation: If $|w| > i$, we write $w|i$ for the prefix of w of length i

Skip Grapsh Continue...

- **Result 1:** Let $\{S_w\}$ be a skip graph with alphabet Σ. for any $z \in \Sigma^{\infty}$, the sequence $S_0, S_1, S_2, ...$, where each $S_i = S_{z|i}$, is a skip list with parameter $p = |\Sigma|^{-1}$

The search algorithm

- Denote the pointer to x's successor and predecessor at level l as $x.neighbor[R][l]$ and $x.neighbor[L][l]$ respectively. Define xR_l formally to be the value of $x.neighbor[R][l]$, if $x.neighbor[R][l]$ is a non-nil pointer to a non-faulty node, and $\perp$ otherwise. Define xL_l similarly.

The search algorithm continue...

- The variables stored at each node is presented in Table *I*.

Variable	Type
key	Resource Key
neighbor[R]	Array of successor pointers
neighbor[L]	Array of predecessor pointers
m	Membership vector
maxLevel	Integer
deleteFlag	Boolean

Table: List of all the variables stored at each node.

```
Upon receiving (searchOp, startNode, searchKey, level)
// If the current key is the search key, return
if (v.key = searchKey) then
    send (foundOp, v) to startNode
end
// Coming from left, current key is less than search key
if (v.key < searchKey) then
    // Go one level down till a right neighbor with key smaller than the search key is
    reached
    while level ≥ 0 do
        if ((v.neighbor[R][level]).key ≤ searchKey) then
            send (searchOp, startNode, searchKey, level) to v.neighbor[R][level]
            break
        else
            level = level - 1
        end
    end
else
    // Coming from right, current key is greater than search key
    while level ≥ 0 do
        // Go one level down till a left neighbor with key larger than the search key is
        reached
        if ((v.neighbor[L][level]).key ≥ searchKey) then
            send (searchOp, startNode, searchKey, level) to v.neighbor[L][level
            break
        else
            level = level - 1
        end
    end
end
// Reached the bottom-most level, key is not found
if (level < 0) then
    send (notFoundOp, v) to startNode
end
```

Algorithm 1: Algorithm 1: search for node v.

The search algorithm continue....

- **Result 2:** The search operation in a skip graph S with n nodes takes expected $O(\log n)$ messages and $O(\log n)$ time.

The insert algorithm

A new node u knows some introducing node v in the network that will help it to join the network. Node u inserts itself in one linked list at each level till it finds itself in a singleton list at the topmost level. The insert operation consists of two stages:

- Node u starts a search for itself from v to find its neighbors at *level* 0, and links to them.
- Node u finds the closest nodes s and y at each *level* ≥ 0, $s < u < y$, such that $m(u)|(l+1) = m(s)|(l+1) = m(y)|(l+1)$, if they exist, and links to them at level $l+1$.

Execute search from introducer to find max $s < u$

Initialize: $l = 0$

while *true* **do**

insert u in list at level l starting from s

scan backwards at level l to find s' such that

$m(s')|(l+1) = m(u)|(l+1)$

if *no such s exists* **then**

exit

else

$s = s'$

l = l+1

end

end

Algorithm 2: Algorithm 2: insert for a new node u.

- **Result 3:** The insert operation in a skip graph S with n nodes takes expected $O(\log n)$ messages and $O(\log n)$ time.

The delete algorithm

The delete operation is very simple. When node u wants to leave the network, it deletes itself in parallel from all lists above *level* 0 and then deletes itself from *level* 0.

for $l = 1$ *to* $u.maxLevel$ *in parallel* **do**
  delete u from list at *level* l
  delete u from list at *level* 0
end

Algorithm 3: Algorithm 3: delete for existing node u.

The delete algorithm continue...

- **Result 4:** In the absence of concurrency, the delete operation in a skip graph S with n nodes takes expected $O(\log n)$ messages and $O(1)$ time.

Applications

The skip graph as a peer-to-peer data storage system

A natural way to group data items is to let each server hold a contiguous segment of the key space. Each server puts only one element of that segment in the skip graph (this data item effectively serves as the key of the skip graph node), and holds the rest in some internal data structure (which allows range queries). Now each node corresponds to a server in the system and not to a data item, and the capability to range query the data set is preserved.

Applications continue...

Fault tolerance

Skip graphs are highly resilient to random failure of nodes. The expansion property of skip graphs gives the theoretical support to these empirical findings.

Applications continue...

Load balancing

A server that wishes to join the skip graph performs a random walk of length $\Omega(\log n)$, recording the number of data items held by each node it encounters.

Applications continue...

Locating highly replicated data items

A skip graph may serve as an excellent hybrid data structure; i.e., may serve as a structured and unstructured P2P system simultaneously.

References

1. William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. Communications of the ACM, 33(6):668676, June 1990.
2. Aspnes, J. and Shah, G. 2003. Skip graphs. In Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms. 384393.
3. Sukumar Ghosh. Distributed Systems: An Algorithmic Approach. Page 374-376.
4. Gauri Shah. Ph.D. Thesis: Distributed Data Structures for Peer-to-peer Systems. Department of Computer Science, Yale University. Page 61-117.

References continue...

5. Aspnes, J. and Wieder, U. 2005. The expansion and mixing time of skip graphs with applications. In Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures. 126134.

Thank you!!!