

NP-Incompleteness

Notes on programming and computer science

[Home](#) [About](#) [Book Reviews](#)

Skip Lists in Python

Posted on September 25, 2012 by kunigami

Skip list is a probabilistic data structure that allows efficient search, insertion and removal operations. It was invented by William Pugh [1] in 1989.

Other structures that have efficient operations are self-balancing binary trees, such as AVL, Red-black and splay tree. But they are often considered difficult to implement.

On the other hand, skip lists are much like multiple linked lists with some randomization.

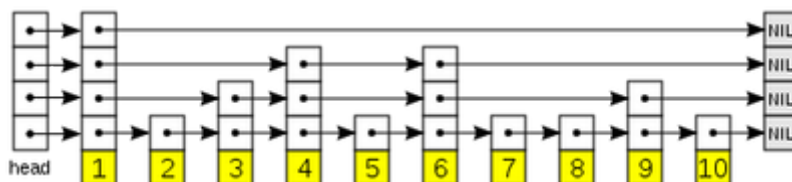


Figure 1: Skip list

In the first level, we have a regular linked list with the elements sorted. Each element of this list has a probability p to be also present in the level above. The second level will probably

Blog Stats

59,929 hits

Pages

[About](#)

[Book Reviews](#)

Categories

Select Category ▼

Archives

Select Month ▼

Blogroll

[Abstruse Goose](#)

[OR by the Beach](#)

[SMBC Comics](#)

contain fewer elements and each of these elements will also have a chance p to be on the third level, and so on. Figure 1 shows an example of a skip list.

We'll implement a simple version of the skip list in python. To start, we define a skip list node. We'll represent each level where the node appears by a list of pointers to the next nodes.

```
1 class SkipNode:
2     def __init__(self, height = 0, elem =
3         self.elem = elem
4         self.next = [None]*height
```

Our skip list is just a sentinel skip node with height initially set to 0 and that stores a null element.

```
1 class SkipList:
2     def __init__(self):
3         self.head = SkipNode()
```

Now, let's implement the search operation of this list.

Search

To search for an element q in a skip list we begin in the topmost level of the header. We go through the list in this level until we find node with the largest element that is smaller than q .

We then go to the level below and search again for node with the largest element that is smaller than q , but this time we began the search from the node we found in the level above.

When we find such node, we go down again and repeat this process until we reach the bottom level. The node x found in the bottom level will be the largest element that is smaller than q in the whole list and if q is in this list, it will be to the right of x .

Also, we want to keep the nodes found in each level right before going down to the level below since it will make the insertion and deletion operations very simple.

Computer Science Blogs

Azimuth

Combinatorics and More

Computational Complexity

Gödel's Lost Letter and P=NP

Low Dimensional Topology

Math \cap Programming

Stochastix

The Endeavour

What's new

Windows on Theory

Tags

amortized analysis Andrew Ng binary numbers binomial heaps cartography

coin-or coursera d3.js data

science data visualization es6 finite

automata Google Google Chrome

Google IO haskell introduction

Jeffrey Ullman lazy evaluation Leslie

Lamport linear regression logistic

regression Mike Bostok monads

monad transformers neural networks

ocaml parametric polymorphism

parsec paxos algorithm promise

prototype **Purely**

Funcional

Data

Structures push-

down automata puzzle queue r

random access lists **Real**

World Haskell skewed binary

numbers Spanner spoj stream

structural decomposition support

vector machines trivial pursuit turing

machines undecidability universally

quantified type

This idea can be translated into the following code:

```

1  def updateList(self, elem):
2
3      update = [None]*len(self.head.next)
4      x = self.head
5
6      for i in reversed(range(len(self.head.next))):
7          while x.next[i] != None and \
8                x.next[i].elem < elem:
9              x = x.next[i]
10             update[i] = x
11
12     return update

```

It returns a list of nodes in each level that contains the greatest value that is smaller than `elem`.

The actual `find` function returns the node corresponding to the query element or `None` if it is not present in the skip list.

```

1  def find(self, elem, update = None):
2      if update == None:
3          update = self.updateList(elem)
4      if len(update) > 0:
5          candidate = update[0].next[0]
6          if candidate != None and candidate.elem < elem:
7              return candidate
8      return None

```

Complexity

The complexity of the search is given by the following Theorem [2]:

Theorem: The number of moves in a search is $O(\log n)$ with high probability.

By high probability we mean that we can set an arbitrarily high probability by increasing the constant hidden in the $O()$ notation. The proof of this Theorem is sketched in Appendix A.

Insertion

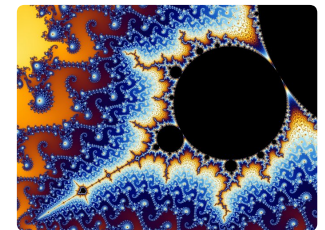
My tweets

Tweets by @kunigami



Guilherme Kuniyama
@kunigami

Polymorphic Recursion
in OCaml
kunigami.blog/2017/10/02/polymorphic-recursion-in-ocaml/



Oct 2, 2017

Embed

View on Twitter

The insertion consists in deciding the height of the new node, using `randomHeight()` and for each of the levels up to this height, insert this new node after the node specified in `update`.

```

1  def insert(self, elem):
2
3      node = SkipNode(self.randomHeight(),
4
5      while len(self.head.next) < len(node
6          self.head.next.append(None)
7
8      update = self.updateList(elem)
9      if self.find(elem, update) == None:
10         for i in range(len(node.next)):
11             node.next[i] = update[i].nex
12             update[i].next[i] = node

```

Deletion

The deletion is pretty much like the insertion, but now we delete the node found using `find()` from all levels in which it appears.

```

1  def remove(self, elem):
2
3      update = self.updateList(elem)
4      x = self.find(elem, update)
5      if x != None:
6          for i in range(len(x.next)):
7              update[i].next[i] = x.next[i]
8              if self.head.next[i] == None:

```

Note that for the sake of simplicity we do not resize `head.next` when the lists at top levels become empty. It does not change the theoretical complexity in the worst case, but in practice it may improve performance.

Computational Experiments

The complete implementation of skip list is available at [Github](#) as well as some test cases and a simple implementation of a linked list.

Our computational experiments consist in comparing the execution time for linked lists ($O(n)$ per operation), our simple skip list ($O(\log n)$ with high probability) and an implementation of a **red-black tree** (worst-case $O(\log n)$ per operation).

We ran 10000 insertions in each of these structures with a random sequence, an increasing sequence and a decreasing sequence. We measure the CPU time in seconds:

	Skip List	RB Tree	Linked List
Random	0.55	0.57	29.25
Increasing	0.36	0.55	78.50
Decreasing	0.38	0.56	0.04

Table 1: Times for insertion in different structures

As we can see, for random input Red-black tree and Skip list have similar performance. For increasing and decreasing sequences, Skip list performed better than Red-black tree because the former is unaffected by the ordering of insertions, while the latter has to make many balancing operations in such cases.

As for linked lists, we can verify it's much slower than the other two structures since it has $O(n)$ worst case insertion time, but it outperforms when the elements are inserted in decreasing order since in this case insertion is $O(1)$:)

Conclusion

There is a combination of insertions and removals that may degenerate a skip list to a linked list, so the operations of searching, inserting and deleting become $O(n)$ in the worst case scenario, though it is very unlikely to happen.

Demaine's analysis [2] is stronger than Pugh's [1]. The latter proves that the cost of the search is $O(\log n)$ in average (i.e.

expected cost) while the former proves it is $O(\log n)$ for most of the cases.

References

- [1] Skip Lists: A Probabilistic Alternative to Balanced Trees – W. Pugh.
- [2] Introduction to Algorithms MIT – Lecture 12: Skip Lists – Erik Demaine

Appendix A: Proofs

In this section we present the proof of the Theorem stated in the post. Let $L(n) = \log_{1/p} n$.

Before proving the Theorem, let's prove the following Lemma:

Lemma: The number of levels in a skip list is $O(L(n))$ with high probability.

Proof:

Let's consider the error-probability, that is, the probability that there are more than $c \cdot L(n)$ levels. We use the [Boole's inequality](#) which says that for a set of events $\{E_1, E_2, \dots, E_k\}$:

$$Pr\{E_1 \cup E_2 \cup \dots \cup E_k\} \leq Pr\{E_1\} + Pr\{E_2\} + \dots + Pr\{E_k\}$$

Thus,

$$Pr\{\text{max level} \geq c \cdot L(n)\} \leq n \cdot Pr\{\text{node level} \geq c \cdot L(n)\}$$

Since each node height is given by a [geometric distribution](#), we have that for some given level x :

$$Pr\{\text{node level} = x\} = p^{x-1}(1-p)$$

$$\Pr\{\text{node level} < x\} = \sum_{i=0}^{x-1} \Pr\{\text{node level} = i\} = (1-p) \sum_{i=0}^{x-1} p^i = 1 - p^x$$

$$\Pr\{\text{node level} \geq x\} = p^x$$

$$\text{Also, } p^{c \cdot L(n)} = p^{c \cdot \log_{1/p} n} = \frac{1}{n^c}$$

Finally,

$$\Pr\{\text{max level} < c \cdot L(n)\} \geq 1 - \frac{1}{n^{c-1}}$$

Thus, for a sufficiently large constant c , we have a very high probability, which proves the Lemma.

Theorem: The number of moves in a search is $O(\log n)$ with high probability.

Proof:

Let's prove that the number of moves in a search is $O(L(n))$ with high probability.

First, consider the reversed path to find the returned element. This reversed path consists of up and left movements along the skip list. Note first that the number of up movements is bounded by the levels of the skip list.

An up movement is done with probability p , which is the case that the current element has at least one more level. Otherwise a left movement is done with probability $1 - p$.

Then, the length of the path is given by the number of movements until we reach $c \cdot L(n)$ up movements. We claim that such number of movements is $O(L(n))$ with high probability.

To prove our claim, let the number of movements be $c'cL(n)$ for some other constant c' . In [2], some combinatoric relations are used to show that

$$Pr\{\# \text{ up moves} \leq cL(n) \text{ among } c'cL(n) \text{ moves}\} \leq \frac{1}{n^\alpha}$$

where $\alpha = ((c' - 1) - L(c'e)) \cdot c$. We also have the converse:

$$Pr\{\# \text{ up moves} > cL(n) \text{ among } O(cL(n)) \text{ moves}\} = 1 - \frac{1}{n^\alpha}$$


Since $(c' - 1) > L(c'e)$ for sufficiently large values of c' , we can choose c' to make α arbitrarily large, which makes the probability above very high, proving the claim.

We conclude that the number of movements is $O(cL(n)) = O(L(n))$ with high probability.

Advertisements


[10000mAh External Backu...](#)
\$14.14
Buy Now

[S530 Mini Stereo Bluetoot...](#)
\$2.25
Buy Now

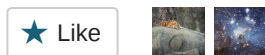
Ads by tvc-mall.com 

[10000mAh External Backu...](#)
\$14.14
Buy Now

[OEM Camera Flash Frame ...](#)
\$0.95
Buy Now

Ads by tvc-mall.com 

Share this:



2 bloggers like this.

Related

[The Algorithm X and the Dancing Links](#)
In "data structures"

[Polymorphic Recursion in OCaml](#)
In "data structures"

[Numerical Representations as inspiration for Data Structures](#)
In "data structures"

This entry was posted in [data structures](#), [probabilistic algorithms](#), [python](#).
Bookmark the [permalink](#).

13 thoughts on “Skip Lists in Python”

[alexmatto](#) says:

September 25, 2012 at 1:43 pm

Great post!



It's usually hard for me to understand your posts because of the heavy math theory. But this one was very easy to understand! (with the exception of the proof, because I'm not used to reading proofs)

↩ Reply

[kunigami](#) says:

September 26, 2012 at 10:09 am

Thanks, I'm glad you follow my blog!



↩ Reply

[brongondwana](#) says:

September 25, 2012 at 6:44 pm

Yay skiplists. There's an on-disk format implementation of them in Cyrus IMAPd – and the (not yet released) version has a crash safe extended skiplist format called 'twoskip' because it has two “complete” linked lists as well as the skip lists.



↩ Reply

[kunigami](#) says:

September 26, 2012 at 10:09 am

Cool! I'll take a look on that.



↩ Reply

James says:

September 26, 2012 at 9:58 am



Since you never attempt to splice the list the gain you are seeing is the same as the re-balance cost in the rb tree.

If left running for a long time adding and removing items it will almost always certainly de-grade into a linked list.

Can you put together some longer running benchmarks?

↩ Reply

kunigami says:

September 26, 2012 at 10:08 am



I can't see why it will degrade into a linked list, unless we are (very!) unlucky to delete only nodes with height greater than 1. With random deletions we're probably deleting nodes with all heights.

Anyway, I'll try to include a benchmark mixing insertions and deletions ASAP.

↩ Reply

mortoray says:

September 26, 2012 at 2:18 pm



A major use of skip lists is for concurrent data structures. With some slight modifications they can be used to create a concurrent ordered set. Multiple processors can modify the same list at the same time in a lock-free fashion.

(Of course this aspect of skip-lists won't be usable in Python.)

↩ Reply

Pingback: [Visto nel Web – 46 « Ok, panico](#)

[badc0re](#) says:

August 5, 2013 at 2:59 pm

Reblogged this on [Ownagezone](#) and commented:

Great post for implementing skip list in python.



↩ Reply

[badc0re](#) says:

August 7, 2013 at 2:53 am

I am curious about what is the purpose of:

```
node.next[i] = update[i].next[i]
```

```
update[i].next[i] = node
```

in insert function. In which cases `update[i].next[i]` be different than `None`, and what is the purpose of assigning `None` to `node.next[i]`?



↩ Reply

[badc0re](#) says:

August 7, 2013 at 3:00 am

Oh my bad i found my error :D sry



↩ Reply

Pingback: [那些小众的数据结构（一）：Skip List | Ace's Blog](#)

Pingback: [2014 in Review | NP-Incompleteness](#)

Leave a Reply (sorry, due to SPAM, the blog requires users to be logged in)

Enter your comment here...

Blog at WordPress.com.

