Probabilistic Data Structures

Pedro Ribeiro

DCC/FCUP

2020/2021

(heavily inspired/based on the lecture notes by Jeff Erickson @ University of Illinois at Urbana-Champaign)

What are probabilistic data structures?

- Data structures that use some **randomized algorithm** or takes advantage of some **probabilistic characteristics** internally
- There are mainly two types of randomized algorithms:
  - **Las Vegas algorithm:** always outputs the correct answer, but runtime is a random variable
  - **Monte Carlo algorithm:** always terminates in given time bound, and outputs the correct answer with at least some (high) probability
- Likewise, we have two types of probabilistic data structures:
  - Some always give exact results, but runtime is random variable
    - E.g.: **Treaps**, **Skip Lists**
  - Some have bounded runtime, but give approximate results
    - E.g.: **Bloom Filters**, Count-Min Sketches, HyperLogLog

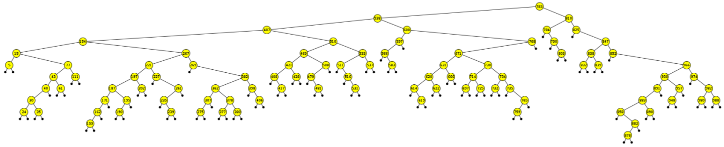
Random Binary Search Trees

- What happens when we insert elements in **randomized order** in a binary search tree?
  - What is the **expected depth** of a node?

- The average efficiency of searching, inserting and removing from that random tree is related to the **expected depth** of a node
  - Depth might n, but what is the chance that we are that "unlucky"? (all permutations are equally likely)

- Let's have a first empirical look at this:

  Gnarley Trees - Visualization of Data Structures
  https://people.ksp.sk/~kuko/gnarley-trees/



97 nodes, height $= 12 = 1.71 \cdot$opt, Avg. Depth $= 7.09$

Random Binary Search Trees

- The depth seems to be always just a (small) constant away from the optimal and therefore **logarithmic** in terms of **expected value**
- Can we prove this?

Example (Theorem - Expected Depth in Random BST)

If n values are inserted in random order on a binary search tree, the expected depth of any node will be $\mathcal{O}(\log n)$

Expected Depth of a Node in a Random BST

- Let x_k denote the node the k-th smallest of n search keys
- We will use $i \uparrow k$ to denote that x_i is a (proper) ancestor of x_k
- The depth of a node is simply the number of its ancestors, so:

$$depth(x_k) = \sum_{i=1}^{n} [i \uparrow k]$$

$[i \uparrow k]$ is just an indicator variable of $i \uparrow k$)
this is called the **Iverson bracket notation**

- We can now express the **expected** depth of a node as:

$$\mathbb{E}[depth(x_k)] = \mathbb{E}\left[\sum_{i=1}^{n} [i \uparrow k]\right] = \sum_{i=1}^{n} \mathbb{E}\left[[i \uparrow k]\right] = \sum_{i=1}^{n} Pr[i \uparrow k]$$

using linearity of expectation: $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$
and indicator variables: $\mathbb{E}[X] = Pr[X = 1]$ *if X is an indicator variable*

Expected Depth of a Node in a Random BST

- Let $X(i,k)$ denote the subset of node between x_i and x_k
  - $\{x_i, x_{i+1}, \cdots, x_k\}$ if $i < k$
  - $\{x_k, x_{k+1}, \cdots, x_i\}$ if $i > k$

- If $i \neq k$, we have $i \uparrow k$ if and only if x_i was the first element being inserted in $X(i,k)$
  - x_i must obviously be inserted before x_k
  - if j is between i and k ($i < j < k$ or $i > j > k$) and x_j is inserted before x_i, then x_i and x_j will end up in different subtrees

- Each node is in $X(i,k)$ is equally likely to be the first one inserted, so:

$$Pr[i \uparrow k] = \frac{[i \neq k]}{|k-i|+1} = \begin{cases} \frac{1}{k-i+1} & \text{if } i < k \\ 0 & \text{if } i = k \\ \frac{1}{i-k+1} & \text{if } i > k \end{cases}$$

Expected Depth of a Node in a Random BST

- Plugging it in the expectation formula:

$$
\begin{aligned}
\mathbb{E}[depth(x_k)] = \sum_{i=1}^{n} Pr[i \uparrow k] &= \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1} \\
&= \sum_{j=2}^{k} \frac{1}{j} + \sum_{j=2}^{n-k+1} \frac{1}{j} \\
&= H_k - 1 + H_{n-k+1} - 1 \\
&\leq \ln k + \ln(n-k+1) \\
&\leq 2 \ln n \in \mathcal{O}(\log n)
\end{aligned}
$$

[the sum $H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$ is known as the **harmonic series** and $H_n \leq \ln(n) + 1$]

$\square$

The expected depth of a node is **logarithmic** in relation to the nr of nodes

Using the ideas Random BSTs

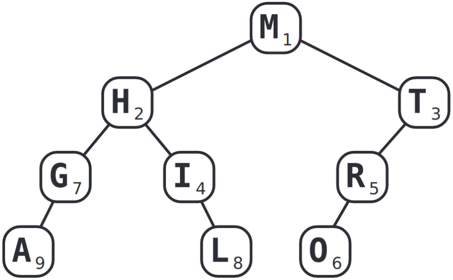
- So, on a random BST with n keys, the nodes are expected to be at $\mathcal{O}(\log n)$ depth
- However, we are not so lucky that the insertion order is random...
- Can we guarantee the same properties for any insertion order?
- *"yes we can"*, and **treaps** are here for the rescue :)

  Aragon, C. R., and Seidel, R. (1989). **Randomized search trees**. In *IEEE Symposium on Foundations of Computer Science (FOCS)* (Vol. 30, pp. 540-545).

  Note: there are other ways of obtaining the same properties, such as: Martínez, C., & Roura, S. (1998). **Randomized binary search trees**. *Journal of the ACM* (JACM), 45(2), 288-323.

Treaps - Concept

- A **treap** is a binary tree in which every node has both a **search key** and a **priority**, where the inorder sequence of search keys is sorted and each node's priority is smaller than the priorities of its children.
- In other words, it is simultaneously:
  - a **binary search tree** for the search keys
  - a **(min-)heap** for the priorities
- Let's see a first example using letters meaning search keys and numbers for the priorities:



Treaps are uniquely determined

- Let's assume that all keys and priorities are distinct
- **The structure of the treap is completely determined** by the search keys and priorities of its nodes
  - Since it's a **heap**:
    - the node v with highest priority must be the *root*
  - Since it's a **BST**:
    - any node u with $key(u) < key(v)$ must be in the left subtree
    - any node w with $key(w) > key(v)$ must be in the right subtree
  - Since the subtrees are treaps, by **induction**, their structures are completely determined (the base case is the empty treap).
- In other words, a treap is exactly the binary search tree that results of **inserting the nodes in order of increasing priority** into an initially empty tree (using the standard textbook insertion algorithm)

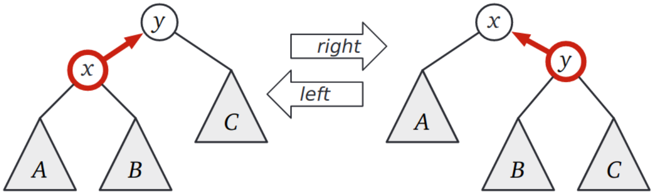
Treap Operations - Search

- **Search** is done as in a normal BST
  - Successful search has time proportional to the depth of the node
  - Unsuccessful search has time proportional to the depth of its predecessor or ancestor

(time is proportional to the depth of the node)

Treap Operations - Insert

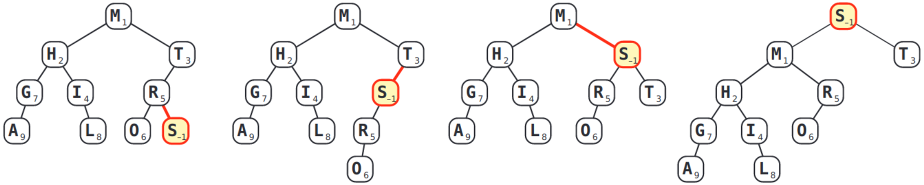
- **Insertion** is done in the following way
  - Let's call z to the new node
  - You do the insertion using the standard BST algorithm
  - This results in z being a leaf, at the bottom of the tree
  - At this point you have a BST, but priorities may no longer form a heap
  - To fix this, you do something similar to a standard heap:
    - ★ As long as the parent of z has a smaller priority, you perform a **rotation** at z, decreasing the depth of z (and increasing the depth of the parent), while keeping the BST property
    - ★ One rotation takes constant time



A right rotation at x and a left rotation at y are inverses.

Treap Operations - Insert/Remove

- The overall time to insert z is **proportional to the depth of** z before the rotations (roughly equi. to $2\times$ of an unsucessfull search for z):
  - we have to walk down the treap to insert z
  - and then walk back up the treap doing rotations



Left to right: After inserting S with priority -1, rotate it up to fix the heap property.
Right to left: Before deleting S, rotate it down to make it a leaf.

- This suggest an algorithm for **removing** an element z (as the inverse in time of insertion)
  - push z to the bottom of three using rotation
  - simply remove z, which is now a leaf
  - **time is proportional to depth of** z

Treap Operations - Split/Join

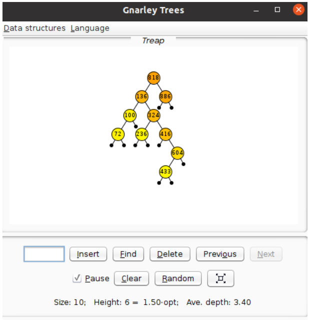
- Treaps are also very handy for **splitting**: imagine we want to split a treap T into two treaps $T_<$ and $T_>$ along some pivot π
  - $T_<$ contains all nodes with keys smaller than π
  - $T_>$ contains all nodes with keys bigger than π
- A simple way to do this is to insert a new node z with $key(z) = \pi$ and $priority(z) = -\infty$
  - after the insertion z is the root
  - deleting the root the left and right subtrees are $T <$ and $T >$
- **Joining** two treaps $T <$ and $T >$ is just splitting in reverse
  - create a dummy root, rotate it to the bottom and chop it off
- For both splitting or joining, time is again **proportional to the depth**

Randomized Treaps

- A **randomized treap** is a treap in which the priorities are **independently and uniformly distributed random variables**.
- Whenever we insert a new search key into the treap, we generate a random real number between (say) 0 and 1 and use that number as the priority of the new node.
- As we saw before, a treap is exactly the BST that results of **inserting the nodes in order of increasing priority** into an initially empty tree (using the standard textbook insertion algorithm)
- Because the priorities are random and independent this is equivalent to inserting them in random order; as we know, this means the expected depth of any node is $\mathcal{O}(\log n)$
- The cost of a treap operation (search/insert/remove/split/join) is proportional to the depth of the node, so **all these operations are logarithmic** :)

Treaps: Conclusion

- Treaps are a **Las Vegas Algorithm**: they always output the correct answer, but runtime is a random variable
- Treaps are very easy to implement (and extend), while providing **(expected) logarithmic time** for all main operations
- You can try out a visualization with **Gnarley trees**: https://people.ksp.sk/~kuko/gnarley-trees/



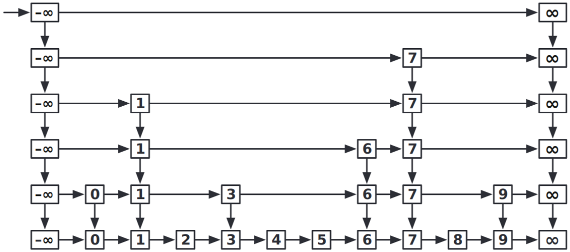
Size: 10; Height: 6 = 1.50·opt; Ave. depth: 3.40

Skip Lists

- How to have the properties of BSTs without trees?

  Pugh, W. (1990). **Skip lists: a probabilistic alternative to balanced trees.** Communications of the ACM, 33(6), 668-676.

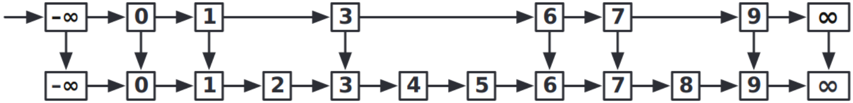
- At a high level, a skip list is just a **sorted linked list with some random shortcuts**



A skip list is a linked list with recursive random shortcuts.

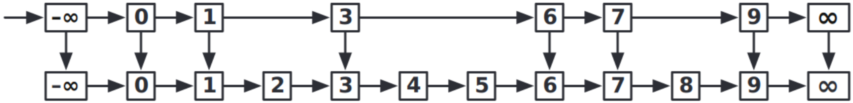
Skip Lists: Concept

- To search in a normal singly-linked list of length n, we obviously need to look at n items in the worst case.
- To speed up this process, we can make a second-level list that contains roughly half the items from the original list
  - For each item in the original list, we duplicate it with probability $1/2$
  - We string together all the duplicates into a second sorted linked list, and add a pointer from each duplicate back to its original
  - (to be safe) We add sentinel nodes at the start and end of both lists



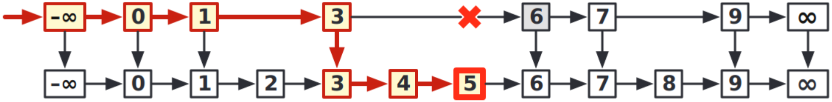
A linked list with some randomly-chosen shortcuts.

Skip Lists: Concept



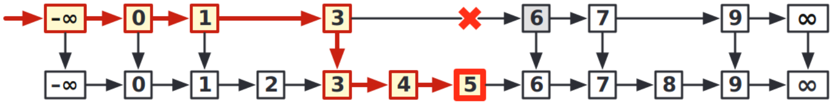
A linked list with some randomly-chosen shortcuts.

- We can now find a value x using a two-stage algorithm:
  - First, we scan for x in the shortcut list, starting at $-\infty$
  - If we find x, we're done
  - Otherwise, we reach a value $> x$ and we know that $x \notin$ shortcut list
  - In the second phase, we scan for x in the original list, starting from the largest item less than x



Searching for 5 in a list with shortcuts.

Skip Lists: Concept

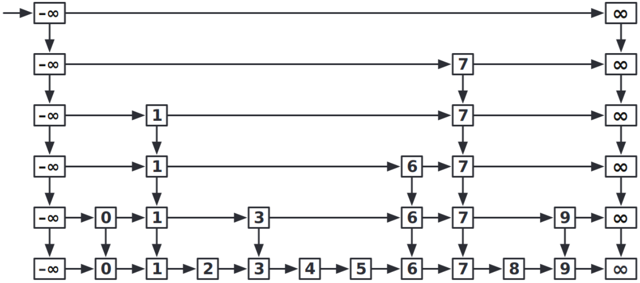


Searching for 5 in a list with shortcuts.

- Since each node appears in the shortcut list with probability $1/2$, the expected number of nodes examined in the first phase is at most $n/2$
- Only one of the nodes examined in the second phase has a duplicate
- The probability that any node is followed by k nodes without duplicates is 2^{-k}.
- So, the expected number of nodes examined in the second phase is at most $1 + \sum_{k>0} 2^{-k} = 2$
- Thus, by adding these random shortcuts, we've reduced the cost of a search from n to $n/2 + 2$ **(roughly a factor of two in savings)**.

Skip Lists: Recursive Random Shortcuts

- There's an obvious improvement: **add shortcuts to the shortcuts, and repeat recursively!** That's exactly how skip lists are made...
  - For each original node, we repeatedly flip a coin until we get tails
  - Each time we get heads, we make a new copy of the node
  - The duplicates are stacked up in levels, and the nodes on each level are strung together into sorted linked lists
  - Each node v stores a search key $key(v)$, a pointer $down(v)$ to its next lower copy, and a pointer $right(v)$ to the next node in its level.

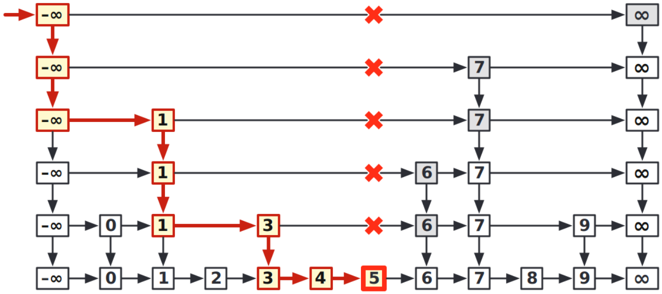


A skip list is a linked list with recursive random shortcuts.

Skip Lists: Searching

- The search algorithm for skip lists is very simple:

```
SKIPLISTFIND($x, L$):
    $v \leftarrow L$
    while ($v \neq$ NULL and $key(v) \neq x$)
        if $key(right(v)) > x$
            $v \leftarrow down(v)$
        else
            $v \leftarrow right(v)$
    return $v$
```



Searching for 5 in a skip list.

Skip Lists: Searching

- The search algorithm for skip lists is very simple:

```
SKIPLISTFIND($x, L$):
    $v \leftarrow L$
    while ($v \neq$ NULL and $key(v) \neq x$)
        if $key(right(v)) > x$
            $v \leftarrow down(v)$
        else
            $v \leftarrow right(v)$
    return $v$
```

- Intuitively, since each level of the skip lists has about half the number of nodes as the previous level, the total number of levels should be about $\mathcal{O}(\log n)$
- Similarly, each time we add another level of random shortcuts to the skip list, we cut the search time roughly in half, except for a constant overhead, so $\mathcal{O}(\log n)$ levels should give us an overall expected search time of $\mathcal{O}(\log n)$.
- Let's try to **formalize** each of these two **intuitive observations**.

Skip Lists: Number of Levels

The actual values of the search keys don't affect the skip list analysis, so let's assume the keys are the integers 1 through n

- Let $L(x)$ be the number of levels of the skip list that contain some search key x, not counting the bottom level
- Each new copy of x is created with probability $1/2$ from the previous level
- We can compute the expected value of $L(x)$ recursively:

$$\mathbb{E}[L(x)] = \frac{1}{2} \cdot 0 + \frac{1}{2}(1 + \mathbb{E}[L(x)]$$

- Solving this equation gives us $\mathbb{E}[L(x)] = 1$

In order to analyze the expected worst-case cost of a search, however, we need a bound on the number of levels $L = max_x L(x)$.
Unfortunately, we can't compute the average of a maximum the way we would compute the average of a sum.

Skip Lists: Number of Levels

Instead, we derive a stronger result:

Number of Levels in a Skip List

The depth of a skip list with n keys is $\mathcal{O}(\log n)$ with high probability.

"High probability" is a technical term that means the probability is at least $1 - \frac{1}{n^c}$ for some constant $c \geq 1$; the hidden constant in the $\mathcal{O}(\log n)$ bound could depend on c.

Number of Levels in a Skip List

The depth of a skip list with n keys is $\mathcal{O}(\log n)$ with prob. $\geq 1 - \frac{1}{n^c}$.

Skip Lists: Number of Levels

Number of Levels in a Skip List

The depth of a skip list with n keys is $\mathcal{O}(\log n)$ with prob. $\geq 1 - \frac{1}{n^c}$.

- For a search key x to appear on level ℓ, it must have flipped heads in a row when it was inserted, so $Pr(L \geq \ell) = 2^{-\ell}$
- The skip list has at least ℓ levels if and only if $L(x) \geq \ell$ for at least one of the n search keys, so:

$$Pr[L \geq \ell] = Pr[(L(1) \geq \ell) \vee (L(2) \geq \ell) \vee \cdots \vee (L(n) \geq \ell)]$$

- Using the **union bound** — $Pr[A \vee B] \leq Pr[A] + Pr[B]$ for any random events A and B — we can simplify this as follows:

$$Pr[L \geq \ell] \leq \sum_{x=1}^{n} Pr[L(x) \geq \ell] = n \cdot Pr[L(x) \geq \ell] = \frac{n}{2^\ell}$$

Skip Lists: Number of Levels

Number of Levels in a Skip List

The depth of a skip list with n keys is $\mathcal{O}(\log n)$ with prob. $\geq 1 - \frac{1}{n^c}$.

$$Pr[L \geq \ell] \leq \frac{n}{2^\ell}$$

- When $\ell \leq \log n$, this bound is trivial (since $Pr \leq 1$)
- However, for any constant $c > 1$, we have a strong upper bound:

$$Pr[L \geq c \log n] \leq \frac{1}{n^{c-1}}$$

We conclude that **with high probability, a skip list has $\mathcal{O}(\log n)$ levels.**

Skip Lists: Number of Levels

This high-probability bound indirectly implies a bound on the *expected* number of levels:

$$\mathbb{E}[L] = \sum_{\ell \geq 0} \ell \cdot Pr[L = \ell] = \sum_{\ell \geq 0} Pr[L \geq \ell]$$

Clearly, if $\ell < \ell'$ then $Pr[L(x] \geq \ell] > Pr[L(x) \geq \ell']$. So we can derive an upper bound on the expected number of levels as follows:

$$\begin{aligned}
\mathbb{E}[L] = \sum_{\ell \geq 1} Pr[L \geq \ell] &= \sum_{\ell=1}^{\log n} Pr[L \geq \ell] + \sum_{\ell \geq \log n + 1} Pr[L \geq \ell] \\
&\leq \sum_{\ell=1}^{\log n} 1 + \sum_{\ell \geq \log n + 1} \frac{n}{2^\ell} \\
&= \log n + \sum_{i \geq 1} \frac{1}{2^i} \\
&= \log n + 2
\end{aligned}$$

Skip Lists: Number of Levels

$$\mathbb{E}[L] \leq \log n + 2$$

Expected Number of Levels in a Skip List

The expected depth of a skip list with n keys is $\leq \log n + 2$

So in expectation, a skip list has at most two more levels than an ideal version where each level contains exactly half the nodes of the next level below. Notice that this is an **additive** penalty over a perfectly balanced structure, as opposed to **treaps**, where the expected depth is a constant **multiple** of the ideal $\log n$.

Skip Lists: Logarithmic Search Time

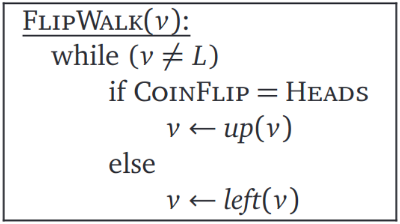
It's a little easier to analyze the cost of a search if we imagine running the algorithm backwards

DNIFTSILPIKS takes the output from SkipListFind as input and traces back through the data structure to the upper left corner (for simplification, imagine you have up and left pointers).

```
DNIFTSILPIKS($v$):
    while ($level(v) \neq L$)
        if $up(v)$ exists
            $v \leftarrow up(v)$
        else
            $v \leftarrow left(v)$
```

Skip Lists: Logarithmic Search Time

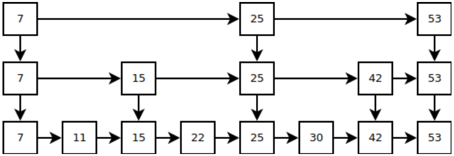
For any v in the skip list, $up(v)$ exists with probability $1/2$. So, for the purpose of analysis, SkipListFind is equivalent to the following algorithm:



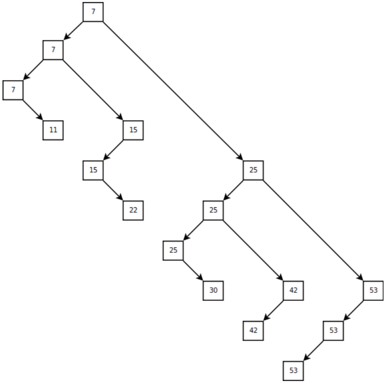
```
FlipWalk(v):
    while ($v \neq L$)
        if CoinFlip = Heads
            $v \leftarrow up(v)$
        else
            $v \leftarrow left(v)$
```

Obviously, the expected number of heads is exactly the same as the expected number of tails. Thus, the expected running time of this algorithm is twice the expected number of upward jumps. But we already know that the number of upward jumps is $\mathcal{O}(\log n)$ with high probability. It follows the running time of FlipWalk is $\mathcal{O}(\log n)$ with high probability (and therefore in expectation)

Skip Lists and Binary Trees



If you rotate the skip list and remove duplicate edges, you can see how it resembles a binary search tree (images from `http://ticki.github.io`):



Skip Lists: Some Advantages

- Skip lists perform very well on insertions (no rotations or reallocations).
- They are simple to implement and extend
- You can easily retrieve the next element in constant time
- "Easy" to parallelize (lock-free skip lists)

Skips lists are used in some (in memory) databases / search engines:

SingleStore   APACHE LUCENE

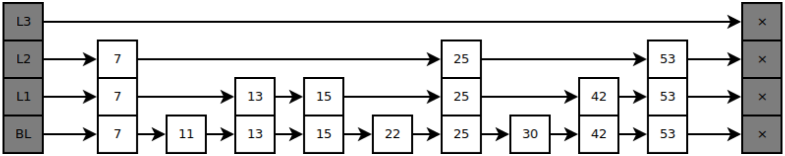
https://www.singlestore.com/blog/what-is-skiplist-why-skiplist-index-for-memsql/

Skip Lists: Implementation

The following article gives very nice hints on how to implement skip lists:

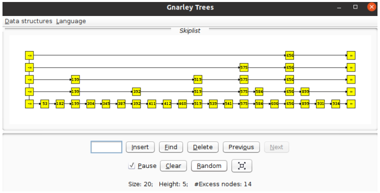
Skip Lists: Done Right
http://ticki.github.io/blog/skip-lists-done-right/

For example, to avoid wasting memory, one could consider each node as an array (avoiding the downward pointers), while also reducing cache misses.



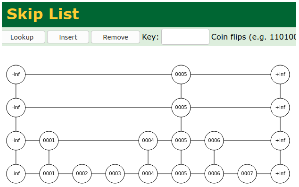
Skip Lists: Visualizations

- You can try out a visualization with **Gnarley trees**: https://people.ksp.sk/~kuko/gnarley-trees/



Size: 20; Height: 5; #Excess nodes: 14

- Or a visualization **Yves Lucet** (@ UBritishColumbia): https://people.ok.ubc.ca/ylucet/DS/Algorithms.html



Bloom Filters: Concept

(slides to be added)