Probabilistic Data Structures

Pedro Ribeiro

DCC/FCUP

2021/2022

(parts of this are 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

Randomized Algorithms

Randomized algorithms

We call an algorithm **randomized** if its behavior is determined not only by its input but also by values produced by a **random-number generator**

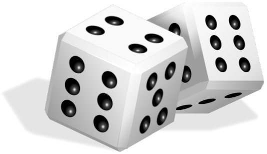
- Most programming environments offer a (deterministic) **pseudorandom-number generator**: it returns numbers that *"look"* statistically random
- We typically refer to the analysis of randomized algorithms by talking about the **expected cost** (ex: the **expected running time**)
- We can use **probabilistic analysis** to analyse randomized algorithms

Basics of Probabilistic Analysis

- Consider rolling **two dice** and observing the results.
- We call this an **experiment**.
- It has **36 possible outcomes**:

  1-1, 1-2, 1-3, 1-4, 1-5, 1-6, 2-1, 2-2, 2-3, ..., 6-4, 6-5, 6-6
- Each of these outcomes has probability $1/36$ (assuming fair dice)
- What is the probability of the sum of dice being 7?

  Add the probabilities of all the outcomes satisfying this condition: 1-6, 2-5, 3-4, 4-3, 5-2, 6-1 (probability is $1/6$)



Basics of Probabilistic Analysis

In the language of probability theory, this setting is characterized by a **sample space** S and a **probability measure** p.

- **Sample Space** is constituted by all possible outcomes, which are called **elementary events**
- In a **discrete probability distribution** (d.p.d.), the probability measure is a function $p(e)$ (or $Pr(e)$) over elementary events e such that:
    - $p(e) \geq 0$ for all $e \in S$
    - $\sum\limits_{e \in S} p(e) = 1$
- An **event** is a subset of the sample space.
- For a d.p.d. the probability of an event is just the **sum** of the probabilities of its elementary events.

Basics of Probabilistic Analysis

- A **random variable** is a function from elementary events to integers or reals:

  Ex: let X_1 be a random variable representing result of first die and X_2 representing the second die.
  $X = X_1 + X_2$ would represent the sum of the two
  We could now ask: what is the probability that $X = 7$?

- One property of a random variable we care is **expectation**:

Expectation

For a discrete random variable X over sample space S, the expected value of X is:

$$\mathbb{E}[X] = \sum_{e \in S} Pr(e)X(e)$$

Basics of Probabilistic Analysis

- In **words**: the expectation of a random variable X is just its average value over S, where each elementary event e is weighted according to its probability.

  Ex: If we roll a single die, the expected value is 3.5
  (all six elementary events have equal probability).

- One possible rewrite of the previous equation, grouping elementary events:

Expectation (possible rewrite)

$$\mathbb{E}[X] = \sum_{a} Pr(X = a)a$$

Basics of Probabilistic Analysis

- More generally:

Expectation (rewrite using disjoint events)

For any partition of the sample space into disjoint events $A_1, A_2, \ldots$:

$$\mathbb{E}[X] = \sum_i \sum_{e \in A_i} Pr(e)X(e) = \sum_i Pr(A_i)\,\mathbb{E}[X|A_i]$$

- $\mathbb{E}[X|A_i]$ is the expected value of X given A_i, defined to be: $\frac{1}{Pr(A_i)} \sum_{e \in A_i} Pr(e)X(e)$.

Basics of Probabilistic Analysis

An important fact about expected values is **Linearity of Expectation**:

Theorem - Linearity of Expectation

For any two random variables X and Y: $\mathbb{E}[\mathrm{X}+\mathrm{Y}] = \mathbb{E}[\mathrm{X}] + \mathbb{E}[\mathrm{Y}]$

Proof for discrete random variables:

$$\mathbb{E}[X+Y] = \sum_{e \in S} Pr(e)(X(e)+Y(e)) =$$
$$= \sum_{e \in S} Pr(e)X(e) + \sum_{e \in S} Pr(e)Y(e) = \mathbb{E}[X] + \mathbb{E}[Y]$$

- It is not necessary that the variables are independent
- This theorem is **very important for the analysis of algorithms**: complicated variables become a sum of simple variables which we can analyse separately.

A first example

Suppose we unwrap a fresh deck of n cards and **shuffle** it until the cards are completely random.

How many cards do we expect to be in the same position as they were at the start?

- X: number of cards that end in the same position as they started
- We are looking for $\mathbb{E}[X]$!
- By linearity of expectation we can write this as a sum of X_i, where $X_i = 1$ if the i-th card ends up in position i, and $X_i = 0$ otherwise:
  $X = X_1 + X_2 + \ldots + X_n = \sum_{i=1}^{n} X_i$
- $Pr(X_i = 1) = 1/n$ where n is the number of cards!
- $Pr(X_i = 1)$ is also $\mathbb{E}[X_i]$ $\quad (\mathbb{E}[X_i] = 1 \cdot Pr(X_i = 1) + 0 \cdot Pr(X_i = 0))$
- $\mathbb{E}[X] = \mathbb{E}[X_1 + \ldots + X_n] = \mathbb{E}[X_1] + \ldots + \mathbb{E}[X_n] = 1$

Indicator Variables

- In the previous example we used an **indicator random variable**:

Indicator Random Variable

The indicator random variable I{A} associated with event A is defined as:

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs} \\ 0 & \text{if } A \text{ does not occur} \end{cases}$$

- Indicator random variables may be very handy in simplifying our analysis, by giving us a **simpler** way to model our desired cost
- Note that if $X_A = I\{A\}$, then $\mathbb{E}[X_A] = Pr(A)$
  $\mathbb{E}[X_A] = 1 \cdot Pr(A) + 0 \cdot Pr(\overline{A})$ where $\overline{A}$ is the complement of A)

Random Binary Search Trees

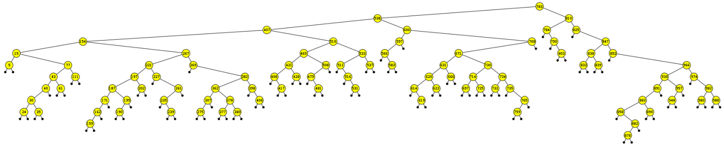
- What happens when we insert elements in **randomized order** in a binary search tree?
  - What is the **expected depth** of a node?

- 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 ·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 nodes 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:

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

$H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$ is known as the **harmonic number** and is in the range $[\ln n, \ln n + 1]$

$\square$

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, the insertion order is not always 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

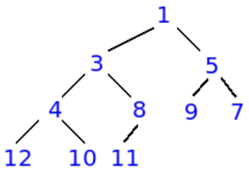
Wait! Let's first remember what a heap is...

Heaps - Concept

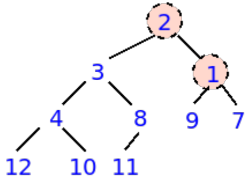
- An **heap** is a tree that obeys the following restriction:

(min)Heap Invariant

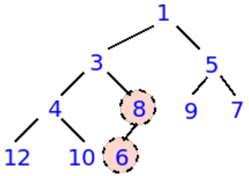
The parent of any node has always an higher priority than its children, that is, on a **minHeap**, the parent is always *smaller* than its children



Heap



Not a Heap



Not a Heap

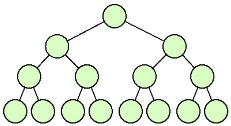
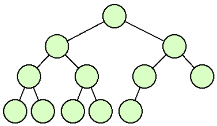
Note: on a *maxHeap*, a node would be bigger than its children

Heaps - $\mathcal{O}(\log n)$ height

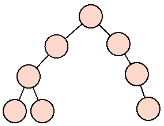
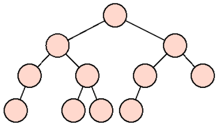
- To guarantee the efficiency of the associated operations, a heap should be a **complete binary tree**:

Complete Tree

A tree where all the levels (except perhaps the last one) are completely filled in and all nodes are as much as possible to the left.

2 examples of complete trees

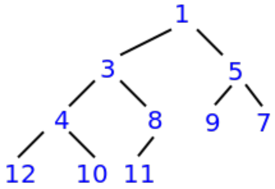



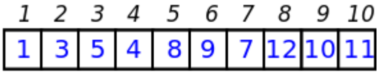
2 examples of incomplete trees

- In a complete tree with n nodes, the height is $\mathcal{O}(\log n)$
  - It is a very balanced tree and we already saw an explanation for this, but intuitively you can think that to *increment by one* the height, we would need to *duplicate* the number of elements

Heaps - Mapping into an array

- The easiest and more compact way of implementing a heap is to use an **array** that *implicitly* represents a tree.
  - The numbers appear on the array on a *breadth-first (BFS) order* (from top to bottom and then from left to right)
  - If the root is put on position 1:
    - The children of position x are in positions $x * 2$ and $x * 2 + 1$
    - The parent of position x is on position $x/2$ (integer division)
- Let's see an example:





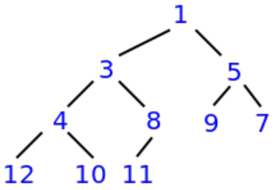
1	2	3	4	5	6	7	8	9	10
1	3	5	4	8	9	7	12	10	11

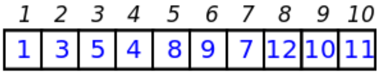
e.g. children of position 3 (node 5) are in positions $3 * 2 = 6$ (node 9) and $3 * 2 + 1 = 7$ (node 7). The parent of position 3 is the node on position $3/2 = 1$.

- Since the tree is complete, this means that the array has all consecutive positions filled in.

Heaps - min() operation

- Since each node is smaller than its children the smallest node is guaranteed to be in the root position (the first element of the array):



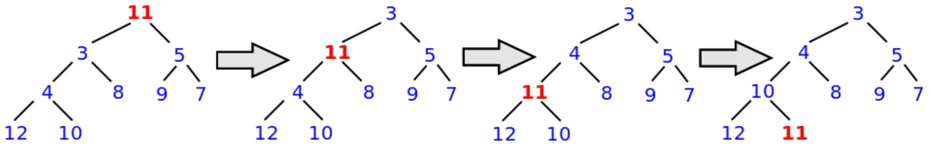


- **min():** time $\mathcal{O}(1)$

Heaps - removeMin() operation

- After removing the root we need to restore the heap conditions. For that, we can do the following:
  - We take the last element of the array and we put it on the root position (the tree is now complete)
  - The element "goes down" (**downHeap**), swapping with the smallest child, until the invariant is restored
  - At most, we traverse the height of the tree, which is $\mathcal{O}(\log n)$

An example of the previous heap, after removing the minimum (1) and putting the last element (11) on its position (the root):

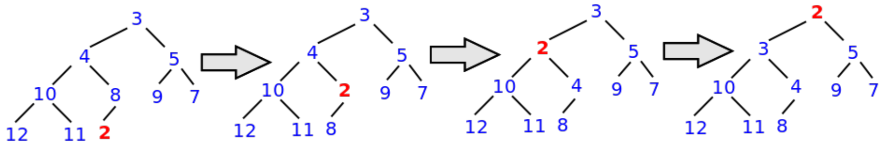


- **removeMin():** time $\mathcal{O}(\log n)$

Heaps - insert(x) operation

- To **insert** we can:
    - Position it after the last one, on the first free array position (the tree is now complete)
    - The element "goes up" (**upHeap**), swapping with its father, until the heap invariant is restored
    - At most, we traverse the height of the tree, which is $\mathcal{O}(\log n)$

An example of inserting 2 in the heap of the previous slide:

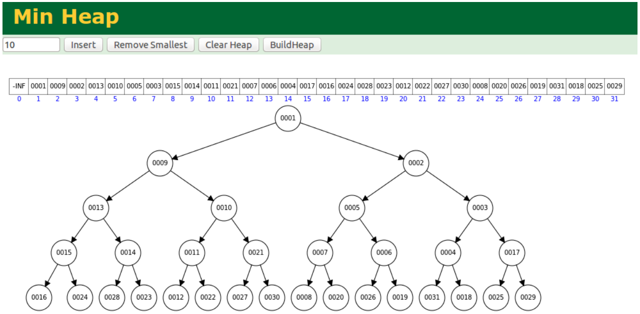


- **insert(x):** time $\mathcal{O}(\log n)$

Heaps - Visualization

- You can visualize heap insertions and removals (try the following url):

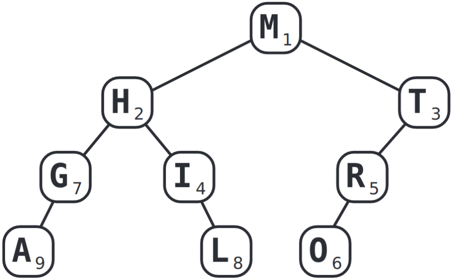
`https://www.cs.usfca.edu/~galles/visualization/Heap.html`



-INF	0001	0009	0002	0013	0010	0005	0003	0015	0014	0011	0021	0007	0006	0004	0017	0016	0024	0028	0023	0012	0022	0027	0030	0008	0020	0026	0019	0031	0018	0025	0029
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Treaps - Concept (continued)

- 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 (not necessarily complete)
- 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)

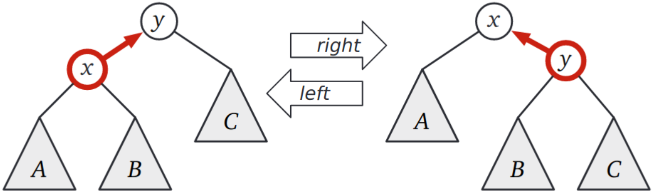
Treap Operations - Search

- **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

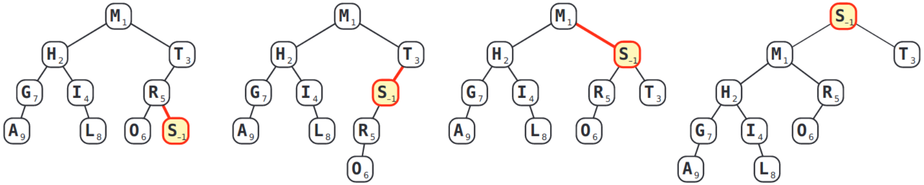
- **Insertion** is done in the following way
  - Let's call z to the new node
  - You do the insertion using the standard BST algorithm
  - This results in z being a leaf, at the bottom of the tree
  - At this point you have a BST, but priorities may no longer form a heap
  - To fix this, you do something 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 equiv. to $2\times$ of an unsuccessful 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 suggests an algorithm for **removing** an element z (as the inverse in time of insertion)
  - push z to the bottom of three using rotations
  - simply remove z, which is now a leaf
  - **time is proportional to depth of** z

Treap Operations - Split/Join

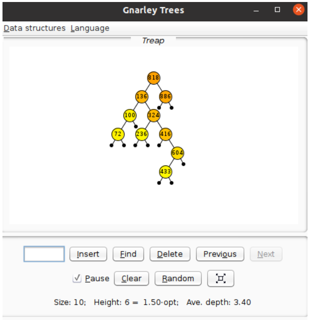
- Treaps are also very handy for **splitting**: imagine we want to split a treap T into two treaps $T_<$ and $T_>$ along some pivot π
  - $T_<$ contains all nodes with keys smaller than π
  - $T_>$ contains all nodes with keys bigger than π
- A simple way to do this is to insert a new node z with $key(z) = \pi$ 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

- (Randomized) 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/



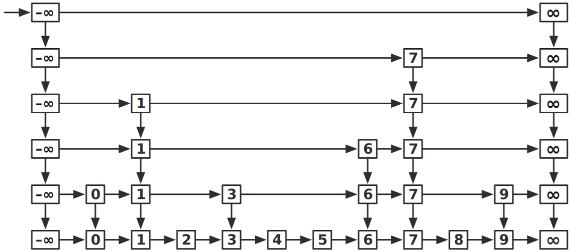
Size: 10; Height: 6 = 1.50·opt; Ave. depth: 3.40

Skip Lists

- How to have the properties of (random) BSTs without trees?

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

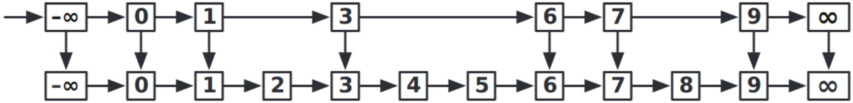
- At a high level, a skip list is just a **sorted linked list with some random shortcuts**



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

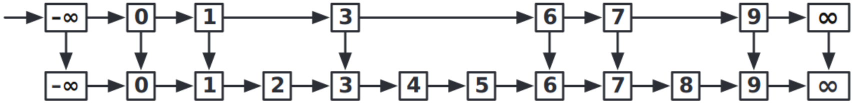
Skip Lists: Concept

- 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



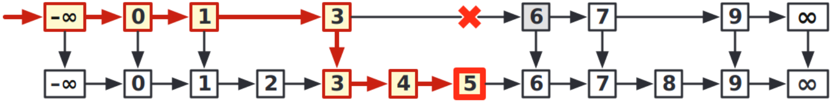
A linked list with some randomly-chosen shortcuts.

Skip Lists: Concept



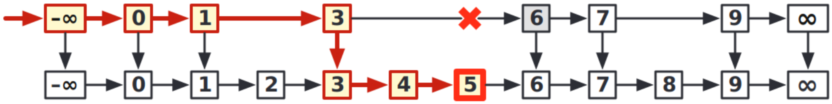
A linked list with some randomly-chosen shortcuts.

- 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

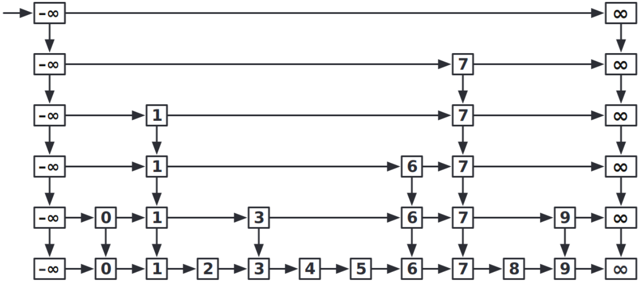


Searching for 5 in a list with shortcuts.

- 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.

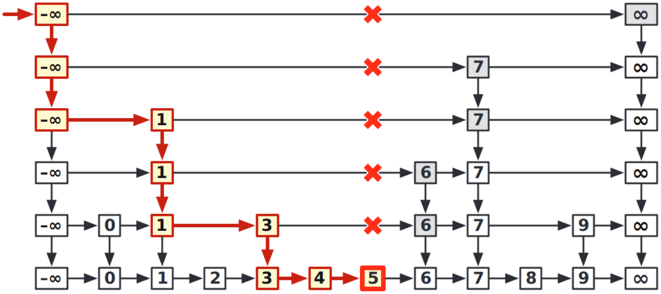


A skip list is a linked list with recursive random shortcuts.

Skip Lists: Searching

- The search algorithm for skip lists is very simple:

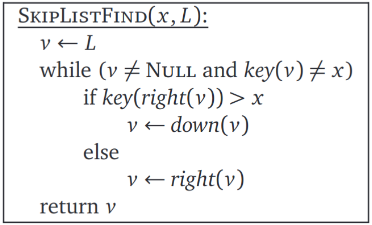
```
SkipListFind($x, L$):
    $v \leftarrow L$
    while ($v \neq$ Null and $key(v) \neq x$)
        if $key(right(v)) > x$
            $v \leftarrow down(v)$
        else
            $v \leftarrow right(v)$
    return $v$
```



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.

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 $Pr(L(x) \geq \ell) = 2^{-\ell}$
- 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:

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

Skip Lists: Number of Levels

$$\mathbb{E}[L] \leq \log n + 1$$

Expected Number of Levels in a Skip List

The expected depth of a skip list with n keys is $\leq \log n + 1$

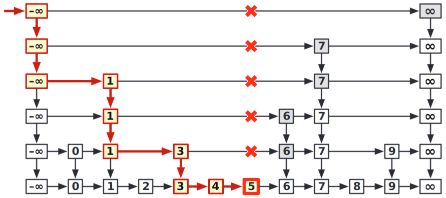
So in expectation, a skip list has at most one more level than an ideal version where each level contains exactly half the nodes of the next level below. Notice that this is an **additive** penalty over a perfectly balanced structure, as opposed to **treaps**, where the expected depth is a constant **multiple**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

ᗡИIᖷTƧI⅃qIꓘƧ 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).

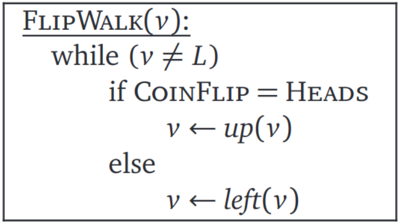
```
ᗡИIᖷTƧI⅃qIꓘƧ($v$):
    while ($level(v) \neq L$)
        if $up(v)$ exists
            $v \leftarrow up(v)$
        else
            $v \leftarrow left(v)$
```



Searching for 5 in a skip list.

Skip Lists: Logarithmic Search Time

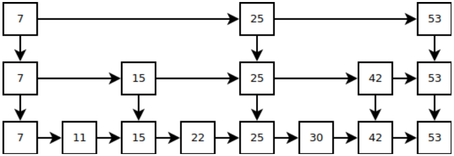
For any v in the skip list, $up(v)$ exists with probability $1/2$. So, for the purpose of analysis, SkipListFind is equivalent to the following algorithm:



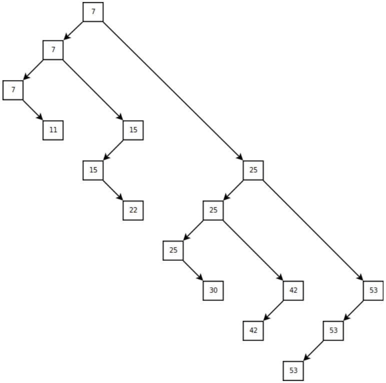
```
FlipWalk($v$):
    while ($v \neq L$)
        if CoinFlip = Heads
            $v \leftarrow up(v)$
        else
            $v \leftarrow left(v)$
```

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:

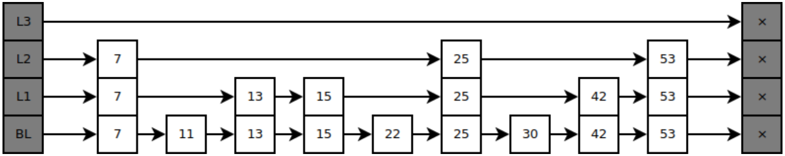
https://www.singlestore.com/blog/what-is-skiplist-why-skiplist-index-for-memsql/

Skip Lists: Implementation

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

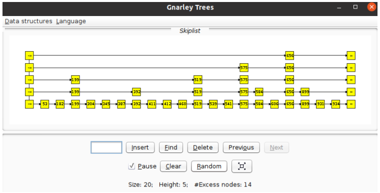
Skip Lists: Done Right
http://ticki.github.io/blog/skip-lists-done-right/

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



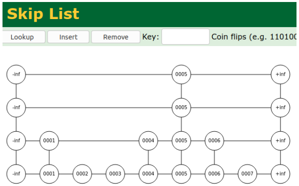
Skip Lists: Visualizations

- 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 by **Yves Lucet** (@ UBritishColumbia):
  https://cmps-people.ok.ubc.ca/ylucet/DS/SkipList.html


Skip List
Lookup Insert Remove Key: Coin flips (e.g. 110100
-inf 0001 0002 0003 0004 0005 0006 0007 +inf

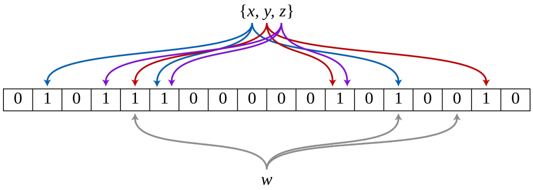
Bloom Filters

- What about a **Monte Carlo** Data Structure? (always terminate in a given time bound, outputs the correct answer with high probability)
- Can we provide efficient **membership queries** on a dynamic set in less then $\mathcal{O}(n)$ space?
  - Having a set X of n elements from a universe $\mathcal{U}$, answer whether a given element $x \in \mathcal{U}$ is an element of X
- **Bloom filters** are capable of doing this!
  - Hash Tables already provide $\mathcal{O}(1)$ expected time, using $\mathcal{O}(n)$ space
  - Bloom filters can be considered as an extension of hash tables
  - By allowing **false positives** - occasionally reporting $x \in X$ when in fact $x \notin X$ we can still answer queries in $\mathcal{O}(1)$ expected time using considerably less space.
  - This makes bloom filters unsuitable as an exact membership data structure, but because of their speed and low false positive rate, they are commonly used as **filters or sanity checks for more complex data structures**.

Bloom Filters - Concept

- A **bloom filter** consists of an array $B[0 \ldots m-1]$ of bits, together with k hash functions $h_1, h_2, \ldots, h_k : \mathcal{U} \rightarrow \{0, 1, \ldots, m-1\}$
- A Bloom filter for a set $X = \{x_1, x_2, \ldots, x_n\}$ is initialized by setting the bit $B[h_j(x_i)]$ to 1 for all indices i and j. Because of collisions, some bits may be set more than once, but that's fine.

```
MakeBloomFilter(X):
  for $h \leftarrow 0$ to $m-1$
      $B[h] \leftarrow 0$
  for $i \leftarrow 1$ to $n$
      for $j \leftarrow 1$ to $k$
          $B[h_j(x_i)] \leftarrow 1$
  return $B$
```

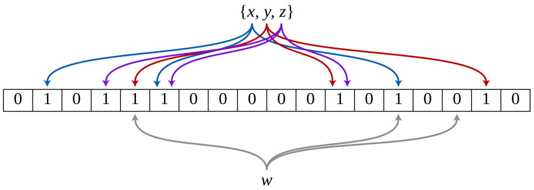


Example bloom filter, representing the set $\{x, y, z\}$ with $k = 3$ and $m = 18$. Colored arrows show the positions in the bit array that each set element is mapped to.

Bloom Filters - Concept

- Given a new item w, the bloom filter determines whether $w \in X$ by checking each bit $B[h_j(w)]$.
- If any of those bits is 0, it correctly reports that $w \notin X$.
- If all bits are 1, it reports that $w \in X$, although this is not necessarily correct.

```
BLOOMMEMBERSHIP($B, y$):
  for $j \leftarrow 1$ to $k$
      if $B[h_j(y)] = 0$
          return FALSE
  return MAYBE
```

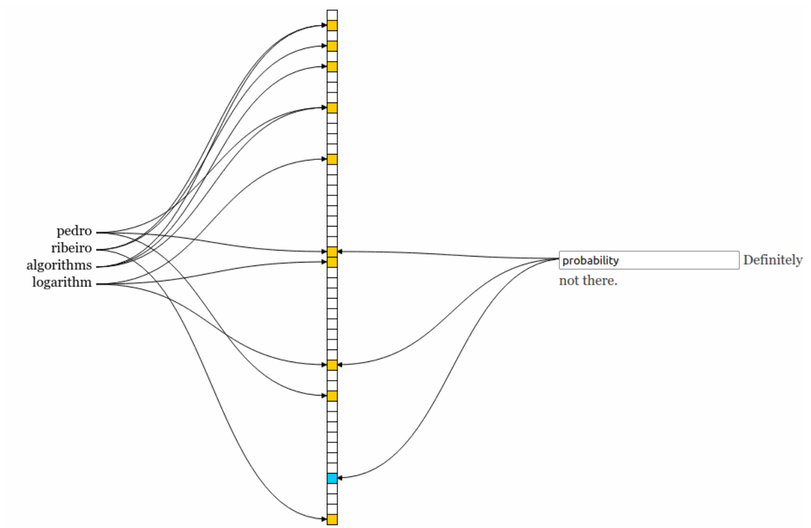


The same bloom filter of the previous slide. The element w is not in the set $\{x, y, z\}$, because it hashes to one bit-array position containing 0.

- One nice feature of bloom filters is that the various hash functions h_i can be evaluated in **parallel** (e.g. on a multicore machine)

Bloom Filters - Visualization

- You can check a visualization of a bloom filter in: https://www.jasondavies.com/bloomfilter/



Bloom Filters - False Positive Rate

- For the purposes of theoretical analysis, we assume the hash functions h_i are **mutually independent, ideal random functions**.
- This assumption is of course unsupportable in practice, but may be necessary to guarantee theoretical performance.
- Fortunately, the actual real-world behavior of Bloom filters appears to be consistent with this unrealistic theoretical analysis.

- Let's estimate the probability of a **false positive**, as a function of the various parameters n, m and k.
- For all indices *pos*, i, and j, we have $Pr(h_j(x_i) = p) = \frac{1}{m}$, so ideal randomness gives us probability p (for every position *pos*):

$$p = Pr(B[pos] = 0) = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

$$\text{given that } \lim_{x \to \infty} \left(1 - \frac{1}{x}\right)^{x} = \frac{1}{e}$$

Bloom Filters - False Positive Rate

- The expected number of 0-bits in the array is approximately mp
- Thus, the probability ϵ of a false positive can be approximated as:

$$\epsilon \approx (1-p)^k = (1-e^{-kn/m})^k$$

This is not strictly correct as it assumes independence for the probabilities of each bit being set

- If all other parameters are held constant, then the false positive rate increases with n (the number of items) and decreases with m (the number of bits)
- The dependence on k (the number of hash functions) is a bit more complicated, but we can derive the best value for k given n and m:

$$k = \frac{m}{n} \ln 2$$

Bloom Filters - Choosing the parameters

- Armed we this we can achieve any desired fale positive ratio $\epsilon > 0$ by choosing the right parameters:

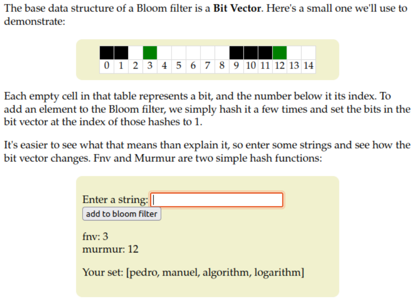
  $m = -\frac{n \ln \epsilon}{(\ln 2)^2}$

  that uses $k = -\frac{\ln \epsilon}{\ln 2} = -\log_2 \epsilon$ hash functions (ignoring integrality)

- For example, we can achieve a 1% false-positive rate using a Bloom filter of size $10n$ bits with 7 hash functions; in practice, this is considerably fewer its than we would need to store all the elements of S explicitly.

Bloom Filters - Conclusion

- It is out of this scope of this course to provide a fully fledged analysis of what hash functions to choose
- The course website provides links to possible hash functions that work well in practice:
  - Murmur, FNV, Jenkins, ...
- Here is a final visualization using real hash functions: http://llimllib.github.io/bloomfilter-tutorial/



The base data structure of a Bloom filter is a **Bit Vector**. Here's a small one we'll use to demonstrate:

■	■		🟩						■	■	■	🟩		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Each empty cell in that table represents a bit, and the number below it its index. To add an element to the Bloom filter, we simply hash it a few times and set the bits in the bit vector at the index of those hashes to 1.

It's easier to see what that means than explain it, so enter some strings and see how the bit vector changes. Fnv and Murmur are two simple hash functions:

Enter a string:

fnv: 3
murmur: 12

Your set: [pedro, manuel, algorithm, logarithm]

Bloom Filters - Variations

- What about element **deletion**?
- What if we want to have the **frequency** of the elements?
  - **Count-Min Sketches**
- There are many more possible variations (e.g. **HyperLogLog**) and there is still much to learn!
- The website provides further links for exploration