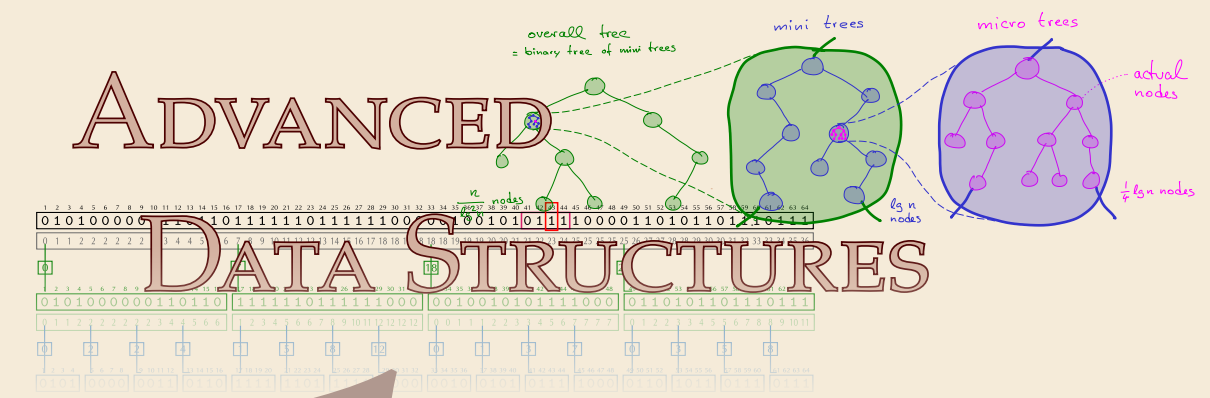


Advanced Data Structures

1 Randomized Trees

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

CS650 (Summer 2026)
Philipps-Universität Marburg
version 2026-04-26 21:35 H

Outline

1 Randomized Trees

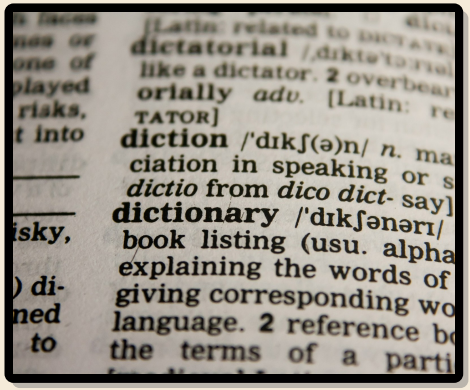
1.1 Recap: Sorted Dictionary and BSTs

Symbol table ADT

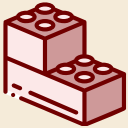
Java: `java.util.Map<K,V>`

Symbol table / Dictionary / Map / Associative array / key-value store:

Python dict `{k:v}`



- `put(`k`,`v`)` *Python* dict: `d[`k`] = `v
  Put key-value pair (k, v) into table
- `get(`k`)` *Python* dict: `d[`k`]`
  Return value associated with key k
- `delete(`k`)` *Python* dict: `del d[`k`]`
  Remove key k (any associated value) form table
- `contains(`k`)` *Python* dict: k `in d`
  Returns whether the table has a value for key k
- `isEmpty(),size()`
- `create()`

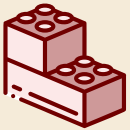


Most fundamental building block in computer science.

(Every programming library has a symbol table implementation.)

Ordered symbol tables

- `min()`, `max()`
  Return the smallest resp. largest key in the ST
- `floor(`x`)`, $\lfloor x \rfloor = \mathbb{Z}$.`floor(`$x$`)`
  Return largest key k in ST with $k \leq x$.
- `ceiling(`x`)`
  Return smallest key k in ST with $k \geq x$.
- `rank(`x`)`
  Return the number of keys k in ST $k < x$.
- `select(`i`)`
  Return the ith smallest key in ST (zero-based, i. e., $i \in [0..n)$)



With select, we can simulate access as in a truly dynamic array!.

(Might not need any keys at all then!)

Binary search trees

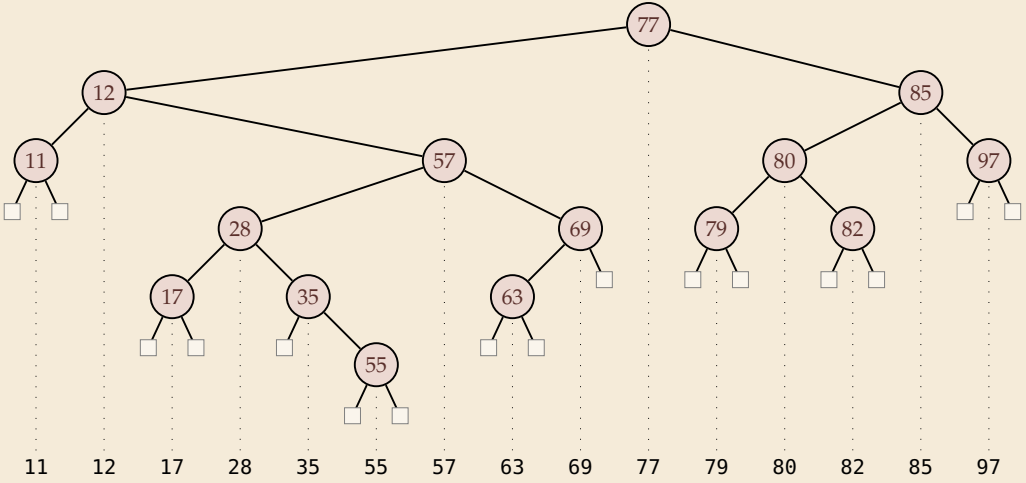
Binary search trees (BSTs) $\approx$ dynamic sorted array

- binary tree
  - Each node has left and right child
  - Either can be empty (`null`)
- Keys satisfy ***search-tree property***

  all keys in left subtree $\leq$ root key $\leq$ all keys in right subtree

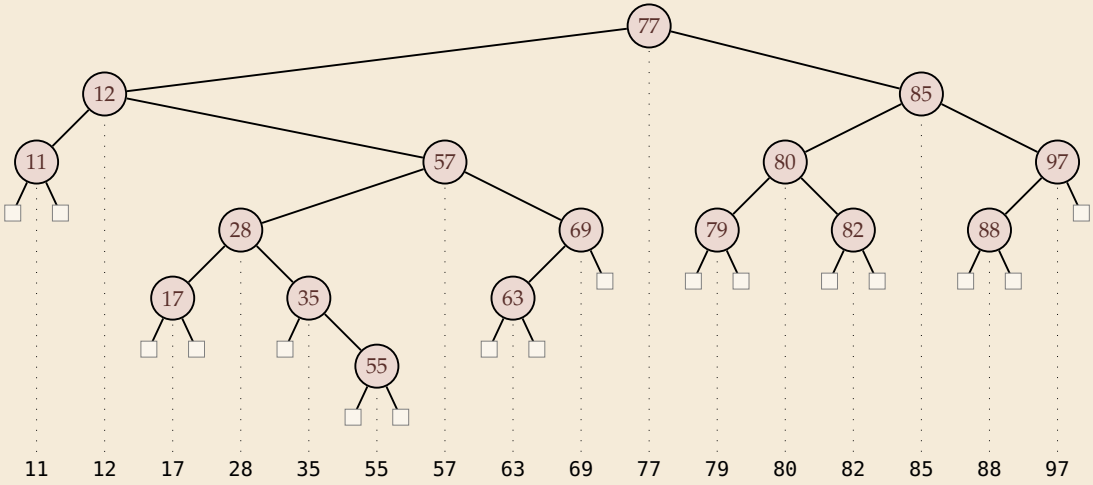
- Standard trick: ***Augmented BSTs***
  - Each node stores the **size** of its **subtree**
  - Easy to maintain upon updates
  - $\rightsquigarrow$ Allows to answer all ordered-symbol-table operations

BST example & find



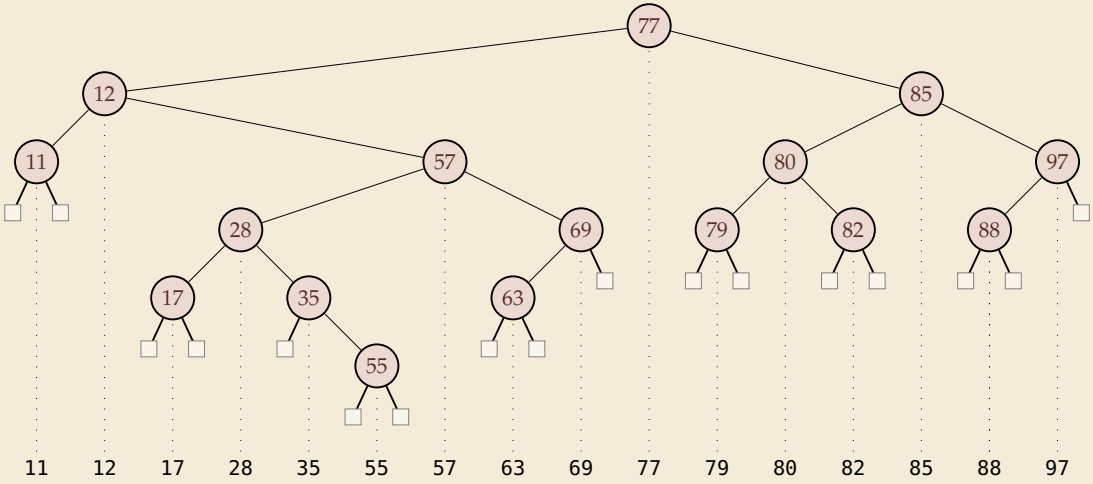
BST insert

Example: Insert 88



BST delete

- Easy case: remove leaf, e. g., 11 ⇝ replace by `null`
- Medium case: remove unary, e. g., 69 ⇝ replace by unique child
- Hard case: remove binary, e. g., 85 ⇝ swap with predecessor, recurse



Unbalanced Binary Search Trees

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n\boldsymbol{h})$
`put(`k`,`v`)`	$O(\boldsymbol{h})$
`get(`k`)`	$O(\boldsymbol{h})$
`delete(`k`)`	$O(\boldsymbol{h})$
`contains(`k`)`	$O(\boldsymbol{h})$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`min()`, `max()`	$O(1)$ (if stored)
`floor(`x`)`, `ceiling(`x`)`	$O(\boldsymbol{h})$
`rank(`x`)`, `select(`i`)`	$O(\boldsymbol{h})$

$h =$ *height* of the BST

- Height h of unbalanced BST depends on insertion order
- satisfies $\lg n \leq h \leq n$
- For **random** insertions, $h = O(\log n)$ in expectation and with high probability

Balanced Binary Search Tree

Balanced binary search trees (BBSTs):

- imposes shape invariant that guarantees $O(\log n)$ height
- adds rules to restore invariant after updates
- **Classical examples**
  - *AVL trees* (height-balanced trees)
  - *red-black trees*

  } every undergrad data-structures module I know includes at least one of these two
  - *weight-balanced trees* (BB[α] trees)
- **Properties**
  - All of them guarantee $h = O(\log n)$ at all times (constants differ!)
  - All of them work with *rotations* along the search path

  $\rightsquigarrow$ $O(\log n)$ extra cost for maintaining balance
  - Further guarantees specific to rule known

    e. g., BB[α] trees only rotate a node again after a linear number of updates to its subtree

OPEN: *What is the exact average cost (constant in front of $\lg n$) of insertion in AVL trees / red-black trees?*

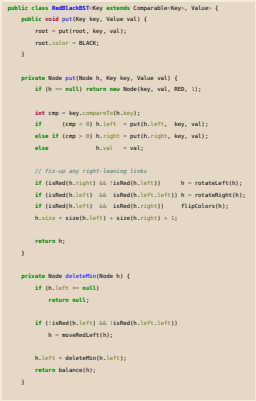
A Primitive for BSTs: Rotations

1.2 What's wrong with BBSTs?

Red-Black Tree Implementation

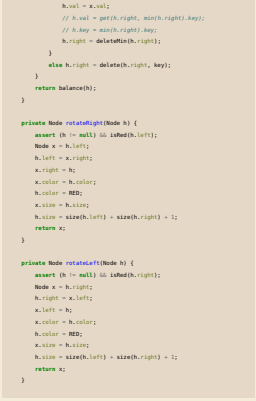
`RedBlackTree` from Sedgewick & Wayne (excerpt) `algs4.cs.princeton.edu/code`

- showing only the bare-bones core: `put` (insert) and `delete`
- no rank-based operations
- full class has 750 lines (and this is a good and compact implementation!)



```
public class RedBlackBST<Key extends Comparable<Key>, Value> {
   public void put(Key key, Value val) {
      root = put(root, key, val);
      root.color = BLACK;
   }

   private Node put(Node h, Key key, Value val) {
      if (h == null) return new Node(key, val, RED, 1);

      int cmp = key.compareTo(h.key);
      if      (cmp < 0) h.left  = put(h.left,  key, val);
      else if (cmp > 0) h.right = put(h.right, key, val);
      else              h.val   = val;

      // fix-up any right-leaning links
      if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
      if (isRed(h.left)  && isRed(h.left.left)) h = rotateRight(h);
      if (isRed(h.left)  && isRed(h.right))     flipColors(h);
      h.size = size(h.left) + size(h.right) + 1;

      return h;
   }

   private Node deleteMin(Node h) {
      if (h.left == null)
         return null;

      if (!isRed(h.left) && !isRed(h.left.left))
         h = moveRedLeft(h);

      h.left = deleteMin(h.left);
      return balance(h);
   }
```

```
   private Node deleteMax(Node h) {
      if (isRed(h.left))
         h = rotateRight(h);

      if (h.right == null)
         return null;

      if (!isRed(h.right) && !isRed(h.right.left))
         h = moveRedRight(h);

      h.right = deleteMax(h.right);

      return balance(h);
   }

   private Node delete(Node h, Key key) {
      // assert get(h, key) != null;

      if (key.compareTo(h.key) < 0)  {
         if (!isRed(h.left) && !isRed(h.left.left))
            h = moveRedLeft(h);
         h.left = delete(h.left, key);
      }
      else {
         if (isRed(h.left))
            h = rotateRight(h);
         if (key.compareTo(h.key) == 0 && (h.right == null))
            return null;
         if (!isRed(h.right) && !isRed(h.right.left))
            h = moveRedRight(h);
         if (key.compareTo(h.key) == 0) {
            Node x = min(h.right);
            h.key = x.key;
```



```
            h.val = x.val;
            // h.val = get(h.right, min(h.right).key);
            // h.key = min(h.right).key;
            h.right = deleteMin(h.right);
         }
         else h.right = delete(h.right, key);
      }
      return balance(h);
   }

   private Node rotateRight(Node h) {
      assert (h != null) && isRed(h.left);
      Node x = h.left;
      h.left = x.right;
      x.right = h;
      x.color = h.color;
      h.color = RED;
      x.size = h.size;
      h.size = size(h.left) + size(h.right) + 1;
      return x;
   }

   private Node rotateLeft(Node h) {
      assert (h != null) && isRed(h.right);
      Node x = h.right;
      h.right = x.left;
      x.left = h;
      x.color = h.color;
      h.color = RED;
      x.size = h.size;
      h.size = size(h.left) + size(h.right) + 1;
      return x;
   }
```

```
   private void flipColors(Node h) {
      h.color = !h.color;
      h.left.color = !h.left.color;
      h.right.color = !h.right.color;
   }

   private Node moveRedLeft(Node h) {
      flipColors(h);
      if (isRed(h.right.left)) {
         h.right = rotateRight(h.right);
         h = rotateLeft(h);
         flipColors(h);
      }
      return h;
   }

   private Node moveRedRight(Node h) {
      flipColors(h);
      if (isRed(h.left.left)) {
         h = rotateRight(h);
         flipColors(h);
      }
      return h;
   }

   private Node balance(Node h) {
      if (isRed(h.right) && !isRed(h.left))     h = rotateLeft(h);
      if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
      if (isRed(h.left) && isRed(h.right))     flipColors(h);

      h.size = size(h.left) + size(h.right) + 1;
      return h;
   }
```

Can we have something simpler?

In this and the next unit, we will look into alternatives to avoid lengthy case distinctions.

Disclaimer: It seems that something has to give.

All known (much) more elegant BST alternatives are either *randomized* or *amortized*.

- However, this is often tolerable.
- Complexity of code can also come with running-time penalty;
  if we avoid that, might be an overall speedup.
  (Typically not the case for well-tested library implementations, though.)

Recall: Average-Case Analysis vs. Randomized Algorithms

This unit: *Randomized solutions*.

Average-Case Analysis

- algorithm is **deterministic**
  same input, same computation
- input is chosen according to some **probability distribution**
- cost given as expectation over inputs

Randomized Algorithm (here)

- algorithm is **not** deterministic
  same input, potentially different comp.
- input is chosen **adversarially** (worst-case inputs)
- cost given as expectation over random choices of algorithm

Confusingly enough, the analysis (technique) is often the same!

But: Implications are quite different; randomization is much more versatile and robust.

1.3 Random BSTs

Random BSTs are good (enough)

*Let's first do some wishful thinking: assume we're dealing with **random** inputs only.*

- ⇝ Let's first see if we like the performance in this case.
- ▶ If so, will try and see how to **enforce** this randomness later.

Random here means:

Definition 1.1 (Random BST)

A ***Random BST*** of size n is the (random) shape of an initially empty BST after successively inserting a random permutation of $[n]$. ◀

Example: $n = 3$

Iterative randomness

Lemma 1.2 (Random insertion yields random BST)

Let $n \geq 0$ and T_n a random BST over n keys. Inserting an element equally likely in one of the $n+1$ *gaps* in T_n (external leaves) results in a new BST T_{n+1} that has the same shape as a random BST on $n+1$ keys. ◀

Proof:
Insertion order for T_{n+1} is a permutation of $[n+1] = \{1, \ldots, n+1\}$,
which is in bijection with a permutation of $[n]$ and a *insertion point (leaf)* in $[0..n]$, where we increment all values right of the insertion leaf by 1.

Example: $1,2,4,7,6,5 \mid 3 \;\widehat{=}\; \big((1,2,3,6,5,4),2\big)$ ■

Recall: Events

something we can assign a probability to

$A \in \mathcal{F}$ is called an *event* of probability space $(\Omega, \mathcal{F}, \mathbb{P})$; also a *measurable set*.

Basic properties

- $\mathbb{P}\left[\overline{A}\right] = 1 - \mathbb{P}[A]$ counter-probability $\quad (\overline{A} = \Omega \setminus A)$
- $\mathbb{P}\left[\bigcup A_i\right] \leq \sum_i \mathbb{P}[A]$ the *union bound* (a.k.a. Boole's inequality a.k.a. σ-subadditivity)
- $\{A_1, \ldots, A_k\}$ *(mutually) independent* $\iff \mathbb{P}\left[\bigcap_i A_i\right] = \prod_i \mathbb{P}[A_i]$

  An infinite set of events is mutually independent if every finite subset is so.

  k-wise independence means that only all size-k subsets are independent.
- *conditional probability* for A given B: $\mathbb{P}[A \mid B] = \mathbb{P}[A \cap B] / \mathbb{P}[B]$
  generally undefined if $\mathbb{P}[B] = 0$
- *law of total probability*: If $\Omega = B_1 \dot\cup B_2 \dot\cup \cdots$ is a partition of Ω, we have

$$\mathbb{P}[A] = \sum_{\substack{i \\ \mathbb{P}[B_i] \neq 0}} \mathbb{P}[A \mid B_i] \cdot \mathbb{P}[B_i].$$

Recall: Random Variables

Random variables (r.v.) $X : \Omega \to \mathcal{X}$; often $\mathcal{X} = \mathbb{R}$ (in general spaces: only *measurable* functions)

Basic properties and conventions:

- event $\{X = x\}$ is defined as $\{\omega \in \Omega : X(\omega) = x\}$.
- For event A define the indicator r.v. $\mathbb{1}_A$ via $\mathbb{1}_A(\omega) = [\omega \in A]$
- $F_X(x) = \mathbb{P}[X \le x]$ is the *cumulative distribution function (CDF)*.
- for discrete r.v. X define $f_X(n) = \mathbb{P}[X = n]$ the *probability mass function (PMF)*.

Independence:

- X and Y independent $\iff \mathbb{P}[X = x \wedge Y = y] = \mathbb{P}[X = x] \cdot \mathbb{P}[Y = y]$ for all x, y.

  (Naturally follows from independent events)

- a sequence of *i.i.d.* r.v. $X_1, X_2, \ldots$ (*independent and identically distributed*) has $X_i \overset{\mathcal{D}}{=} X_1$ and $\{X_i\}_{i \ge 1}$ are mutually independent
  - typical example: sequence of coin tosses (with same coin)

Alternative Random Model

Corollary 1.3

A BST built by inserting n i.i.d. Uniform$(0,1)$ r.v. has the shape of a random BST. ◀

1.4 Properties of Random BSTs

Analysis of Random BSTs I

Theorem 1.4 (Expected depth of leftmost leaf)

The *expected depth* (number of edges from root) of the leftmost external leaf (leaf for $-\infty$) in a random BST on $n \geq 1$ nodes is $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \sim \ln n$. ◀

Proof:

$\text{depth}(\boxed{0}) = \#\text{left-to-right minima } \text{L2R}_n \text{ in insertion sequence}$

$\text{L2R}_n = \sum_{i=1}^{n} X_i \qquad \text{for } X_i = [\text{position } i \text{ is a l2r min}]$

First i inserted elements are in bijection with random permutation π of $[i]$

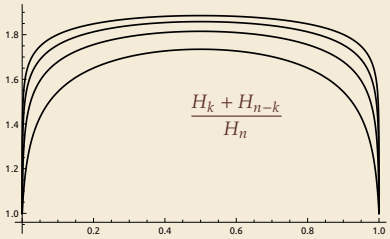
$\rightsquigarrow \quad X_i = [\pi_i = 1]$ and so $\mathbb{P}[X_i = 1] = \frac{1}{i}$

$\mathbb{E}[\text{L2R}_n] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \frac{1}{i} = H_n.$ ■

Analysis of Random BSTs II

Theorem 1.5 (Expected depth of kth leaf)

The *expected depth* of the *kth external leaf* (for $k = 0, \ldots, n$) in a random BST on $n \geq 1$ keys is $H_k + H_{n-k}$.



Proof:

$\text{depth}(\boxed{k})$ = #comparisons for unsuccessful search for a key x that terminates in $\boxed{k}$.

= #comparisons with keys $< x$ + #comparisons with keys $> x$

= #l2r-mins among keys $< x$ + #l2r-maxs with keys $> x$

Since there are k keys $< x$ and $n - k$ keys $> x$, the same derivation as above applies. ■

Analysis of Random BSTs III

Corollary 1.6 (Depth of typical leaf)

Consider a random BST T_n of n keys.

(a) The *expected external path length* of T_n is
$2(n+1)(H_{n+1}-1) \;=\; 2n\ln n - 2(1-\gamma)n \;\pm\; O(\log n)$. ($\gamma \approx 0.5772$ the *Euler-Mascheroni constant*)

(b) The depth of the αnth leaf in a random BST of n keys $\sim 2\ln n$ as $n \to \infty$ for any fixed $\alpha \in (0,1)$.

◀

Detour: When Expectation Isn't Enough

- Two hypothetical algorithms:
  - A takes 1 step in half the cases and 3 steps otherwise
  - B takes 1 step in 99% of cases and **101** steps otherwise
  - $\rightsquigarrow$ both have expected costs of 2 steps.
  - probably want A ... certainly would want to be able to distinguish them!

- **Goal:** Strengthen algorithms so $time(x)$ rarely far from $\mathbb{E}[time(x)]$
  - formally: bound probability that X (far) exceeds $\mathbb{E}[X]$
    - $\rightsquigarrow$ *concentration bounds* a.k.a. *tail inequalities*
  - can then compare these typical times again
  - also obtain more reliable algorithms

  $\rightsquigarrow$ Let's establish some tools for that!

Detour: With High Probability

Definition 1.7 (With high probability)

We say

- an event $X = X(n)$ happens *with high probability (w.h.p.)* when $\forall c \,:\, \mathbb{P}[X(n)] \,=\, 1 \pm O(n^{-c})$ as $n \to \infty$.
- a random variable $X = X(n)$ is *in* $O(f(n))$ *with high probability (w.h.p.)* when $\forall c\ \exists d \,:\, \mathbb{P}[X \le d f(n)] \,=\, 1 \pm O(n^{-c})$ as $n \to \infty$.
  (This means, the constant in $O(f(n))$ may depend on c.) ◀

- Very strong notion: failure probability smaller than any polynomial

$\rightsquigarrow$ If A succeeds w.h.p. then also polynomially many repetitions of A succeed w.h.p.

Lemma 1.8 (Repetitions w.h.p.)

Suppose A is an algorithm that w.h.p. does not fail.
In n^d *independent* repetitions of A on inputs of size n, w.h.p. *no* repetition fails. ◀

Detour: With High Probability [2]

Proof (Lemma 1.8):
Let c from the definition of w.h.p. be given.
The event F_i that the ith run of A fails happens with probability $O(n^{-(c+d)})$ by definition.

Then $\mathbb{P}[\cup_{i=1}^{n^d} F_i] \leq \sum_{i=1}^{n^d} \mathbb{P}[F_i] = n^d \cdot O(n^{-(c+d)}) = O(n^{-c})$. ■

$\rightsquigarrow$ events that happen with high probability can be combined

Detour: Chernoff Bounds

Strong concentration inequalities require assumptions on distribution of X.
A classical one is that X consists of many **small and independent** parts.

Theorem 1.9 (Chernoff Bound for Bernoulli trials)

Let $X_1, \ldots, X_n \in \{0,1\}$ be *(mutually) independent* with $X_i \overset{\mathcal{D}}{=} \mathrm{B}(p_i)$.
Define $X = X_1 + \cdots + X_n$ and $\mu = \mathbb{E}[X_1] + \cdots + \mathbb{E}[X_n] = p_1 + \cdots + p_n$. Then holds

$$\forall \delta > 0 \;:\; \mathbb{P}[X \geq (1+\delta)\mu] \;<\; \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu$$

$$\forall \delta \in (0,1] \;:\; \mathbb{P}[X \geq (1+\delta)\mu] \;\leq\; \exp(-\mu\delta^2/3)$$

◀

Proof (Sketch):

Apply Markov's Inequality to $Y = e^{tX}$, then choose convenient t. ■

(more details in *Advanced Algorithms*)

Detour: Chernoff Bound for Binomial Distribution

The algorithmically most widely used special case has identical coin flips.

Corollary 1.10 (Chernoff Bound for Binomial Distribution)

Let $X \overset{\mathcal{D}}{=} \mathrm{Bin}(n,p)$. Then

$$\forall \delta \geq 0 \;:\; \mathbb{P}\left[\left|\frac{X}{n} - p\right| \geq \delta\right] \leq 2\exp(-2\delta^2 n)$$

◀

$\rightsquigarrow$ $\boxed{\mathrm{Bin}(n,p) \in np \pm n^{0.501} \text{ w.h.p.}}$

Analysis of Random BSTs – Concentration

Theorem 1.11 (Concentration of left-to-right minima)

The number of left-to-right minima in a permutation of length n is in $O(\log n)$ w.h.p.
Hence, the above expected results hold with high probability (up to constant factors). ◀

Proof (Idea):

Via *inversion table* of permutation:

$a_1, \ldots, a_n$ permutation of $[n]$ in bijection with $b_1, \ldots, b_n$ where

$$b_j = \#\text{inversions of form } (\bullet, j)$$
$$= \#\text{values left of } i \text{ that are} > i$$

random permutation $\rightsquigarrow$ b_j **independent** and $b_j \stackrel{\mathcal{D}}{=} \text{Uniform}[0..n-j]$

$$\text{L2RMax}(a) = \sum_{j=1}^{n} [b_j = 0] \rightsquigarrow \text{a sum of independent Bernoulli trials.}$$ ■

Hook-Length Formula for Random BSTs

For a given tree, the probability to see this shape can be computed recursively over the tree structure:

Theorem 1.12 (Random BST Distribution)

Let T_n be a binary tree. The probability that Random BSTs attains the shape T_n after n insertions is

$$\Pr[T_n] = \begin{cases} 1 & n = 0, \\ \frac{1}{n} \cdot \Pr[T_L] \cdot \Pr[T_R] & n \geq 1, \quad \text{(for } T_L \text{ and } T_R \text{ the left resp. right subtrees of } T\text{).} \end{cases}$$

◀

More AofA (Analysis of Algorithms)

Remark 1.13 (Knowledge on Random BSTs)

Random BSTs are extremely well-studied in Analysis of Algorithms.
A few more results (all proven in the literature):

(a) The **expected height** is $\alpha \ln n - \beta \ln \ln n \pm O(1)$ with $\alpha \approx 4.311$ and $\beta \approx 1.953$.

(b) The **height** divided by $\ln n$ **converges in probability** to the constant α.

(c) The number X_{nk} of **external leaves at depth** k satisfies $\mathbb{E}[X_{nk}] = \frac{2^k}{n!}\left[{n \atop k}\right]$.

(d) The **depth** of a typical **leaf** divided by $\ln n$ **converges in probability** to 2.

(e) The standardized **depth** of a random leaf **converges** in distribution to a standard **normal distribution**.

(f) The same is true for the standardized depth of a random internal node.

(g) Let D_n be the **depth of the nth inserted node**. Then $(D_n - \ln n)/\sqrt{\ln n}$ converges in distribution to a standard **normal distribution**. ◀

⇝ In many ways, random BSTs are close to perfectly balanced.

⇝ If only we can get the performance of random BSTs, we're happy.

Caveat: Random Deletions ≠ Random BSTs!

$\rightsquigarrow$ *Unbalanced BSTs have **great** performance **if** insertions come in random order.*

Interesting fact: *no longer true* if there are also *deletions*!

After long sequence of random inserts and deletes: expected height $\Theta(\sqrt{n})$, not $\Theta(\log n)$ (!)

Reason: Hibbard's deletion algorithm destroys randomness!

Animations: `http://algs4.cs.princeton.edu/32bst`

1.5 Treaps

Treaps

Observation: The *preorder* sequence of the keys fully determines a BST since

- each BST has unique preorder, and
- each preorder generates a unique BST by inserting keys in preorder into an initially empty tree.

⇝ Enforcing the preorder corresponding to a **random** BST suffices!
. . . but how? We have no control over the insertion of keys!

Idea: Separate *key values* from *rank in insertion order* using random **"priorities"**.

Definition 1.14 (Treap)

Let $S = \{(k_1, p_1), \ldots, (k_n, p_n)\}$ be a set of *key-priority pairs* where $k_i \in K$ and $p_i \in [0, 1]$ for K some totally ordered universe.
A *treap* for S is a binary tree with n internal nodes labeled by the key-priority pairs so that

(a) the *search tree property* holds w.r.t. the keys, and

(b) the *heap property* holds w.r.t. the priorities.

There can be only one

Theorem 1.15 (Treaps are unique)

Let S be a set of n key-priority pairs where all keys and all priorities are distinct. Then there is *exactly one treap* for S.

Proof:

Randomized Treaps

Definition 1.16 (Randomized Treaps)

A *randomized treap* is the unique treap that results from given keys $k_1, k_2, \ldots$ where (upon insertion) we assign k_i a priority $p_i \stackrel{\mathcal{D}}{=} \text{Uniform}(0,1)$ independent of all previous priorities. ◀

Theorem 1.17 (Shape of randomized treaps)

The (random) shape of a randomized treap for n keys has the *same distribution* as random BST with n keys. ◀

Corollary 1.18 (Search Costs)

All results for random BSTs apply, in particular:

(a) Expected search costs (#comparisons) $< 2 \ln n + 1$.

(b) Search costs in $O(\log n)$ w.h.p. ◀

1.6 Updates in Treaps

Insertions and Deletions in Randomized Treaps

Up to now: *static* view on treaps.

But can we efficiently turn a randomized treap for $k_1, \ldots, k_n$ into one for $k_1, \ldots, k_{n+1}$?
And vice versa?

Yes!

- **Insert:** Start as in plain BST, then *rotate up* until heap property holds.
- **Delete:** Rotate node down (as if priority was $-\infty$) until it is a leaf, then remove it.

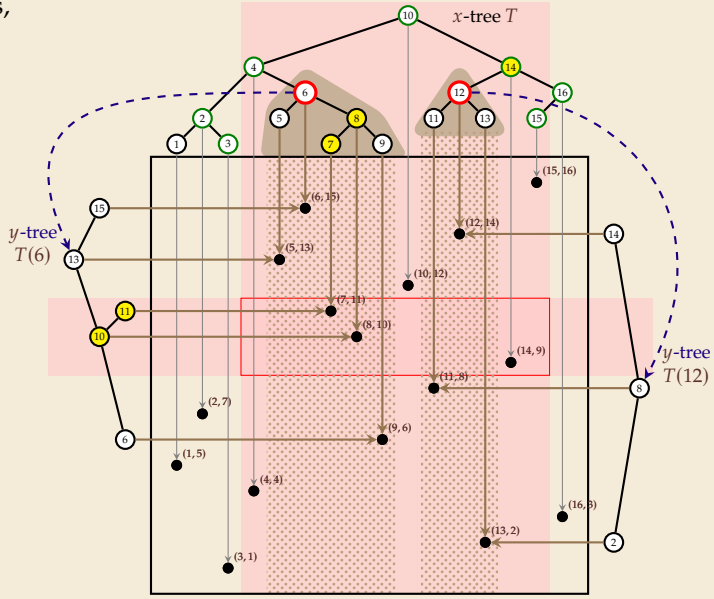
Conceptually very simple!

$\rightsquigarrow$ *all operations in* $O(\log n)$ *time w.h.p.!*

Excursion: Why care about counting rotations?

For secondary data structures, cost of a rotation can be *linear in subtree size*.

Example: *Range trees*



Ancestor Indicators

Can actually bound number of rotations much more tightly!

For that, we need closer look at depths of *internal nodes* in random BSTs.
Analysis possible based on handy notion:

Lemma 1.19 (Ancestor indicators)

Let T_n be a random BST with keys $[n]$ and denote by $A_y^x = [x \text{ is a } \textit{proper} \text{ ancestor of } y]$ for $x, y \in [n]$.
(This means $A_x^x = 0$ and for $x \neq y$, $A_y^x = 1$ iff x lies on the path from the root to y.)
Then holds:

(a) $A_y^x = 1$ iff x was the *first* among the keys $[x..y] \cup [y..x]$ that was inserted into T_n.

(b) $A_y^x = 1$ iff x and y are *directly compared* by randomized Quicksort during a partitioning step using pivot x.

(c) $\Pr[A_y^x = 1] = \Pr[A_x^y = 1] = \dfrac{1}{|y - x| + 1}$ for $x \neq y$. ◀

Ancestor Indicators – Proof

Proof:

Common Ancestor Indicators

Remark 1.20 (Common ancestor indicators)

Idea generalizes to $C^x_{y,z} = [x \text{ is common ancestor of } y \text{ and } z]$:

$$\Pr[C^x_{y,z} = 1] \;=\; \frac{1}{\max\{x,y,z\} - \min\{x,y,z\} + 1}.$$ ◀

Depth of Internal Nodes

Theorem 1.21 (Expected depth of kth node)

The *expected depth* of the *kth internal node* (for $k = 1, \ldots, n$) in a random BST on $n \geq 1$ nodes is $H_k + H_{n-k+1} - 2$. ◀

Recall: $\mathbb{E}[\text{depth}(\boxed{k})] = H_k + H_{n-k}$.

Proof:

$$\text{depth}(\textcircled{k}) = \sum_{x=1}^{n} A_k^x$$

Subtree sizes

Remark 1.22 (Expected subtree size)

The *expected size* of the *subtree* rooted at the kth internal node is also $H_k + H_{n-k+1} - 1$. ◀

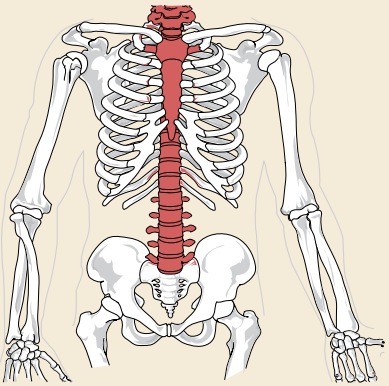
Proof:

Subtree size of $\textcircled{k} = 1 + \sum_{x=1}^{n} A_x^k$

Result follows from similar calculation as above. ■

⇝ Size of secondary data structures to be rebuilt in expectation $2 \ln n$.

Spines of Trees



Lemma 1.23 (Bound on Rotations)
The number of *rotations* to insert or delete a node x in a randomized treap is at most $LS(x) + RS(x)$, where $LS(x)$ and $RS(x)$ are the *lengths of the left resp. right spine* of (the subtree of) x in the treap (after insertion resp. before deletion). ◀

Lemma 1.24 (Expected Spine Lengths)
The expected length of the left and right spine of (the subtree of) the kth internal node (for $k = 1, \ldots, n$) in random BST of n keys are given by

$$\mathbb{E}[LS(k)] = 1 - \frac{1}{k} \qquad \mathbb{E}[RS(k)] = 1 - \frac{1}{n-k+1}$$ ◀

Spines of Trees – Proof

Proof:

$$LS(k) = \sum_{x=1}^{k-1} \left(A_{k-1}^{x} - C_{k-1,k}^{x} \right)$$

1.7 Insertion at Root

Simplerer!

- The details of the implementation of Treaps still need
  - cases distinctions for which rotation to use
  - both code to bubble up nodes (insert) and trickle down (delete)
- We can indeed simplify this further
  (at the expense for more pointer writes)
- Idea actually an alternative to standard BST insert / delete
  - Instead of adding a new leaf upon insert,
    we force the new key to **become the root** upon insert

  $\rightsquigarrow$ need a method to ***split*** a BST into $< x$ and $> x$ for a key x.
  (assume that x not in the tree)
  - if we also have a complementary ***join***,
    deleting an arbitrary node works by joining its children.

Branchless Children

A neat trick makes the following code more compact: store children in 2-element array

```
class Node {
    int key, size = 1;
    Node[] child = new Node[2]; // child[0] is Left, child[1] is Right
}
```

For rank-based operations, we store the subtree size of a node

The following helper method is used to keep subtree size up to date

```
void updateSize(Node t) {
    if (t != null) t.size = 1 + size(t.child) + size(t.child);
}
```

Split

Split BST into two BSTs containing keys $< x$ and $\geq x$ resp.

```
Node[] split(Node t, int x) {
    if (t == null) return new Node[]{null, null};

    int dir = x < t.key ? 0 : 1;
    Node[] res = split(t.child[dir], x);

    // Opposite child receives the remainder of split
    t.child[dir] = res[1 - dir];
    updateSize(t);

    // Construct result dynamically based on dir
    Node[] ans = new Node[2];
    ans[dir] = res[dir];
    ans[1 - dir] = t;
    return ans;
}
```

Join

Given two BSTs where all keys in the left are smaller than all keys in the right, merge them into a single BST.

```
Node join(Node left, Node right) {
    if (left == null) return right;
    if (right == null) return left;

    // new root chosen arbitrarily; here larger tree
    int dir = left.size > right.size ? 0 : 1;
    Node root = dir == 0 ? left : right;
    root.child = (dir == 0) ?
        join(root.child[1], right) :
        join(left, root.child[0]);
    updateSize(root);
    return root;
}
```

Insert at Root, Delete

Based on split and merge, insert and delete easy to implement.

```
Node insertAtRoot(Node t, int x) {
    Node[] splits = split(t, x);
    Node root = new Node(x);
    root.child = splits;
    updateSize(root);
    return root;
}

Node delete(Node t, int x) {
    if (t == null) return null;
    if (x == t.key) return join(t.child, t.child);
    int dir = x < t.key ? 0 : 1;
    t.child[dir] = delete(t.child[dir], x);
    updateSize(t);
    return t;
}
```

Tamio Nakajima's Treap Implementation

```
#include <random>
using namespace std;

struct node {
    node *l, *r;
    int prio, key, sum;
};
node nil_node = {&nil_node, &nil_node, 0, 0, 0};
node *nil = &nil_node; // Sentinel node

node *mod_child(node *n0, int id, node *child) {
    node *n = new node;
    *n = *n0;
    (id == 1 ? n->r : n->l) = child;
    n->sum = n->l->sum + n->key + n->r->sum;
    return n;
}

node *join(node *l, node *r) {
    return l == nil ? r : r == nil ? l :
        l->prio > r->prio ?
            mod_child(l, 1, join(l->r, r)) :
            mod_child(r, 0, join(l, r->l));
}
```

```

pair<node*,node*> split(node* n, int x) {
    pair<node*, node*> ret = {nil, nil};
    return n == nil ? ret : n->key < x ?
        (ret = split(n->r, x), ret.first =
            mod_child(n, 1, ret.first), ret) :
        (ret = split(n->l, x), ret.second =
            mod_child(n, 0, ret.second), ret);
}

node *mk_node(int x){
    static mt19937 mt(random_device{}());
    int prio = uniform_int_distribution<int>(
        1, 1000000000)(mt);
    return new node {nil, nil, prio, x };
}


node *insert(node *root, int x){
    auto p = split(root, x);
    return join(join(p.first, mk_node(x)), p.second);
}

node *empty_treap(){ return nil; }
```

1.8 Randomized Binary Search Trees

What's wrong with Treaps?

Weaknesses of treaps:

- priorities *fixed once and for all* $\rightsquigarrow$ never recovers from "bad luck"
- have to store *priorities* (at least in a direct implementation), but these are *not helpful* algorithmically.

Randomized Binary Search Trees (RBSTs)

Recall: Key property in random BSTs is that in every subtree of size m, each key value is the root of the subtree with probability $1/m$.

Idea of RBSTs: *enforce* this property *anew* after *each* insertion / deletion!

Martínez, Roura: *Randomized Binary Search Trees*, JACM **1998**

- Store in each node x the size of its subtree $S(x)$.
- **Insert:** Insert x as new leaf and let $y_1, \ldots, y_d$ be the nodes on the path from the root. For each y_j, x should have a $1/S(y_j)$ chance to replace y_j as the subtree root.
- **Delete:** After x is gone, one of the remaining $S(x) - 1$ nodes must become subtree root.

  $\rightsquigarrow$ choose one of x's children y and z

  with probabilities $\dfrac{S(y)}{S(y) + S(z)}$ resp. $\dfrac{S(z)}{S(y) + S(z)}$.

Benefits: Tree occasionally rebuilt, subtree sizes useful for rank-based operation.

Insert in RBSTs

Proceed top-down.

At each step, flip a biased coin to decide whether x should be subtree root or not

$\rightsquigarrow$ either call insertAtRoot or recurse.

```
Node insert(Node root, int x) {
    n = root.size;
    r = random.nextInt(n); // uniform int in [0..n)
    if (r == n) // w/p 1/n, x is new root
        return insertAtRoot(root, x);
    if (x < root.key)
        root.left  = insert(root.left,  x);
    else
        root.right = insert(root.right, x);
    updateSize(root);
    return root;
}
```

Note: In expectation over all random priorities

Delete in RBSTs

```
Node delete(Node root, int x) {
    if (root == null) return null;
    if      (x < root.key)
        root.left = delete(root.left, x);
    else if (x > root.key)
        root.right = delete(root.right, x);
    else { // must delete root
        root = join(root.left, root.right);
        updateSize(root);
        return root;
    }
}
```

```
Node join(Node left, Node right) {
    if (left == null) return right;
    if (right == null) return left;
    int sl = left.size, sr = right.size;
    r = random.nextInt(sl+sr); // uniform [0..sl+sr)
    int dir = r < sl ? 0 : 1; // 0 w/p sl/(sl+sr)
    Node root = dir == 0 ? left : right;
    root.child = (dir == 0) ?
        join(root.child[1], right) :
        join(left, root.child[0]);
    updateSize(root);
    return root;
}
```

RBST Analysis

Theorem 1.25 (Correctness)

Any sequence of insert and delete operations results in a tree whose shape is that of a *random BST*. ◀

Theorem 1.26 (Operation Costs)

The costs (#visited nodes) of operations in RBSTs on n keys are

- **(a)** same as in random BSTs for **(un)successful search**, i. e., $\sim 2 \ln n$ in expectation and $O(\log n)$ w.h.p.;
- **(b)** **insert** additionally needs $O(1)$ in expectation during insertAtRoot / split, and
- **(c)** **delete** needs $O(1)$ expected cost in join. ◀

Proof:

(a) follows from Theorem **??**

(b) cost of split.

In the *resulting* tree, nodes touched in split are precisely those on the left and right spine of new node x. $\rightsquigarrow$ In expectation at most 2.

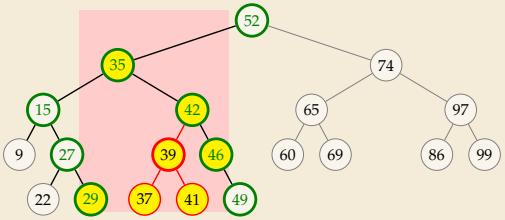
(c) Similar. ■

1.9 Skiplists

Maybe we don't even want a tree?

Typical application for sorted dictionaries: *(1D) range queries*, i. e., all keys in $[\underline{x}, \overline{x}]$

- BSTs can do that in $O(\log n)$ time
  - search for $\underline{x}$ and $\overline{x}$
  - report nodes on path in range and *inside nodes* (children of path nodes)
  - $\rightsquigarrow$ $O(\log n)$ nodes describe output irrespective of #keys in output

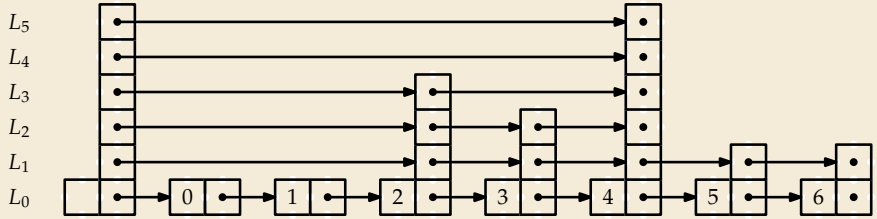


- boundary nodes: nodes in search paths
- inside nodes: nodes between search paths
- outside nodes

- still, iteration over sorted range rather slow
- needs stack or parent pointers
- a sorted linked list does that faster!

Skiplists

Basic structure of a *skip list*: linked list with shortcuts



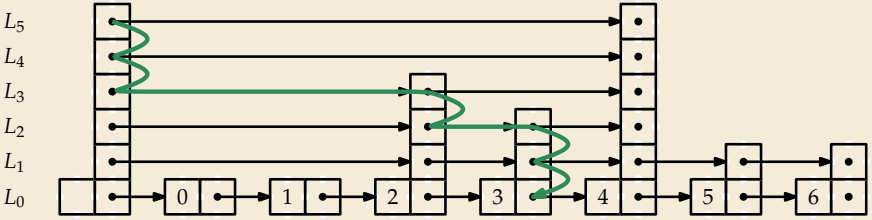
- conceptually sequence of sorted linked lists $L_0, \ldots, L_h$
- each L_{i+1} stores a subsequence of L_i
- obtained by randomized procedure:
  - for each element x in L_i, flip a fair coin $C \stackrel{\mathrm{D}}{=} \mathrm{B}(\frac{1}{2})$
  - if $C = 1$, include x in L_{i+1} (otherwise not)
- eventually stop when next layer empty

Searching in a Skiplist

- head of all lists is sentinel (key $-\infty$)
- each node stores one key and
  a **tower of pointers** `next`$[i]$ for each of its levels i

Searching key x

1. Start in top left corner: $\ell := h$, $u :=$ first node of L_h
2. If u.`next`$[\ell]$.`key` $\leq x$ set $u := u$.`next`$[\ell]$ (move right)
   else $\ell := \ell + 1$ (move down)
3. Repeat until $\ell == 0$



Analysis of Skiplist

Recall: #coin tosses until first 1 =: $T \stackrel{\mathcal{D}}{=} \text{Geo}(p)$ with $p = \frac{1}{2}$ (Geometric distribution)

- $\mathbb{P}[T = k] = (1-p)^{k-1} \cdot p$
- $\mathbb{E}[T] = 1/p$

Fact 1.27 (Space usage of Skiplist)

The expected total height of n pointer towers (excluding sentinel) is $2n$. ◀

Lemma 1.28 (Height of Skiplist)

The expected height $\mathbb{E}[h]$ of a skiplist on n elements is at most $\lg n + 2$. ◀

Proof:

$$h = \sum_{k=1}^{\infty} \overbrace{[L_k \text{ is not empty}]}^{I_k} \leq \sum_{k=1}^{\infty} |L_k|$$

$$\mathbb{E}[h] = \sum_{k=1}^{\infty} \mathbb{E}[I_k] = \sum_{k=1}^{\lfloor \lg n \rfloor} 1 + \sum_{k=\lfloor \lg n \rfloor + 1}^{\infty} \frac{n}{2^k} \leq \sum_{k=1}^{\lfloor \lg n \rfloor} 1 + \sum_{k=0}^{\infty} \frac{1}{2^k} \leq \lg n + 2.$$ ■

Analysis of Search

Theorem 1.29 (Skiplist path lengths)

The expected length of the search path for any node, u, in L_0 is at most $2 \lg n + O(1)$. ◀

Proof:

Consider search path *in reverse*.

In each step, if current tower allows to go up, the path does so.
If it cannot go up, it goes left.

How often do we go left before we move up in expectation? $\rightsquigarrow$ coin toss until success!
$\rightsquigarrow$ Expected 1 step left before we go up.

S_k = #horizontal steps at level k. $\rightsquigarrow$ Above: $\mathbb{E}[S_k] \leq 1$.

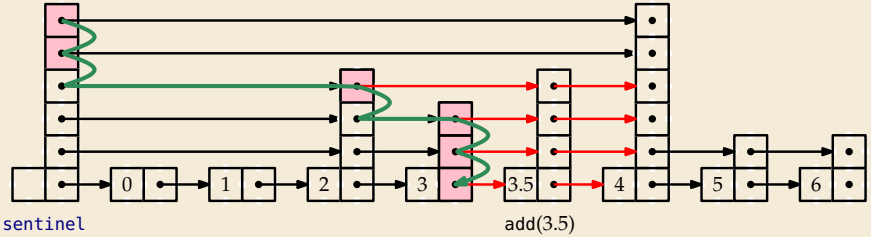
"$\leq$" since we may run out of nodes in L_k! $\rightsquigarrow$ $\mathbb{E}[S_k] \leq \mathbb{E}[|L_k|] \leq n/2^k$.

Total length of path $S = h + S_0 + S_1 + S_2 + \cdots$

$$\mathbb{E}[S] \leq \mathbb{E}[h] + \sum_{k=0}^{\lfloor \lg n \rfloor} \mathbb{E}[S_k] + \sum_{k=\lfloor \lg n \rfloor + 1}^{\infty} \mathbb{E}[S_k] \leq \mathbb{E}[h] + \sum_{k=0}^{\lfloor \lg n \rfloor} 1 + \sum_{k=\lfloor \lg n \rfloor + 1}^{\infty} \frac{n}{2^k} \leq \lg n + 2 + \lg n + 3$$ ■

1.10 Operations in Skiplists

Insert in Skiplists



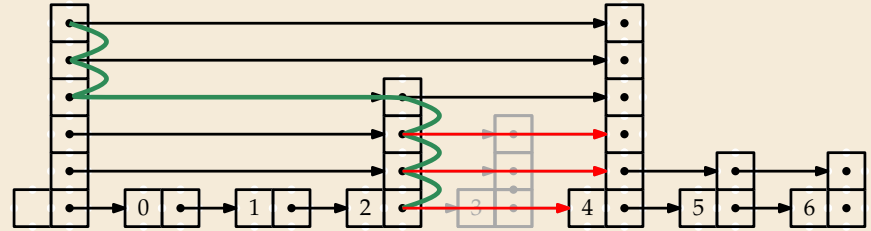
opendatastructures.org

Insert x into skiplist

1. Search for x, keep "hand" of predecessor nodes on all levels
2. Insert x in a new node u into L_0
3. Add a tower of height $T = \text{Geo}(\frac{1}{2})$ to u
4. Use hand to update predecessors on bottom T levels.

Cost bounded in terms of search path $\rightsquigarrow$ $O(\log n)$ in expectation.

Delete in Skiplists



opendatastructures.org

Delete x from skiplist

1. Search for x, keep "hand" of predecessor nodes on all levels
2. Remove node u containing x from linked lists

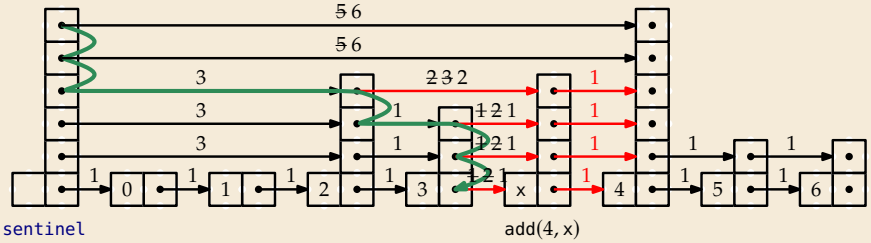
Rank-based operations

Can also support rank-based operations in skiplists.

- Need to know how much we skip!

⇝ Store *length* of pointers

⇝ Can be maintained upon updates (details a bit messy, see *Open Data Structures*)



opendatastructures.org

Skiplists – Discussion

- 👍 Extremely simple use of randomness
- 👍 Range queries simply follow linked list

- 👎 Non-constant size nodes (towers), random space usage
- 👎 Worse search costs: $2\,\mathbf{lg}(n)$ vs $2\,\mathbf{ln}(n) \approx 1.39 \lg n$

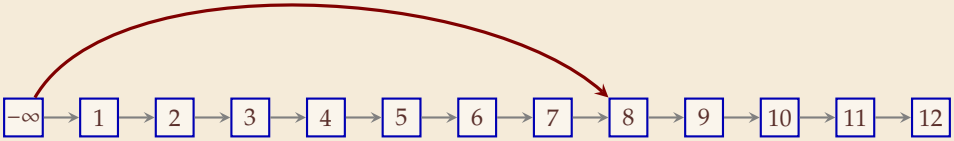
1.11 Jumplists

Jumplists

Binary trees get by with *two pointers* per node; can't we do that, too?

- one pointer used for single linked lists, so only one pointer left!

⇝ the *jump pointer*



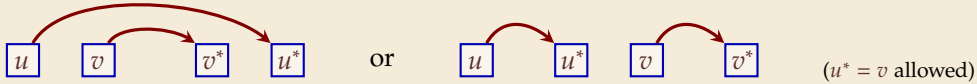
⇝ **Search strategy:** Always try shortcuts, otherwise linear search.

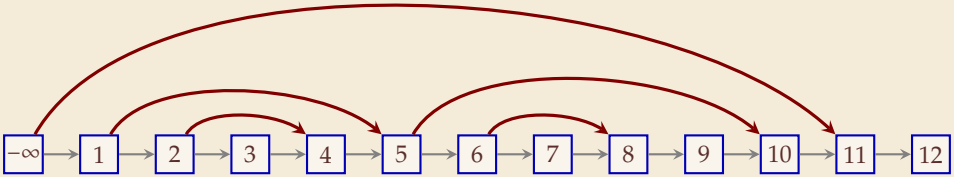
Brönnimann, Cazals, Durand: *Randomized jumplists: A jump-and-walk dictionary data structure*, STACS **2003**

Nebel, Neumann, Wild: *Median-of-k Jumplists and Dangling-Min BSTs*, ANALCO **2019**

Jump Pointers

1. *Non-degeneracy:* Any node may be the target of at **most one** jump pointer, and jump pointers **never** point to the direct **successor**.
2. *Well-nestedness:* Jump pointers may **not cross**. For $v \neq u$ nodes with $v.key < u.key$, and v^* resp. u^* the nodes their jump pointers point to, order must be:

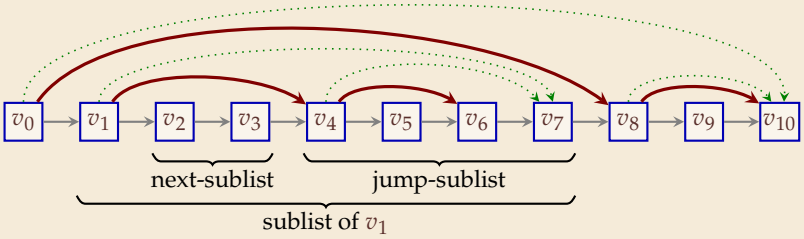




⇝ By non-degeneracy, some nodes can't have jump pointer ⇝ *plain nodes* and *jump nodes*

Sublists

The *sublist of node v* starts at v (inclusive) and ends just before the first node targeted by a jump pointer originating before v (or the end of the list)

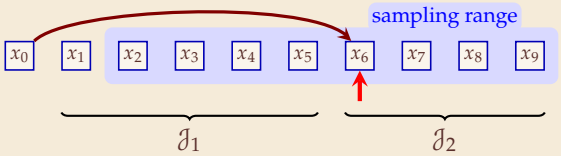


Example: The sublist of node v_1 contains $m(v_1) = 7$ nodes and stores the 6 keys $v_2.key, \ldots, v_7.key$.
(The key of the header is always ignored within a sublist.)
The sizes of the next- and jump-sublist are $J_1(v_1) = 2$ and $J_2(v_1) = 4$, respectively.

Random Jumplists

We can now define the equivalent of the random BST model: *Random Jumplists*

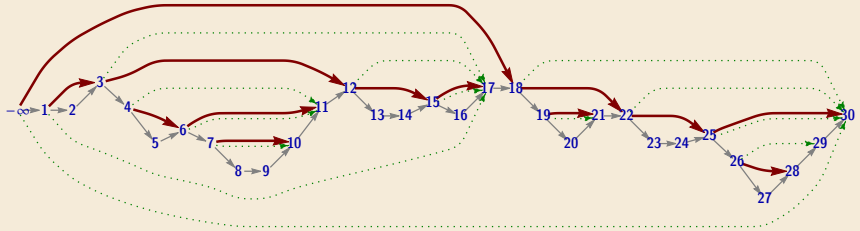
1. Pick jump pointer target of header uniformly from eligible targets
2. Recurse on next-sublist and jump-sublist



$$\mathbb{P}[\mathcal{J}] = \begin{cases} 1, & m \le w; \\ \dfrac{1}{m-2} \cdot \mathbb{P}[\mathcal{J}_1]\,\mathbb{P}[\mathcal{J}_2], & m > w, \end{cases}$$

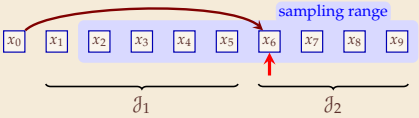
($w = 2$ for simplest version)

Typical Jumplist



Expected Search Costs

⇝ Expected search cost (#pointers followed) of a random gap:



$$D_m = \begin{cases} 0 & m = 1 \\ 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} & m = 2 \\ 0 \cdot \frac{1}{m} + \frac{1}{m-2} \sum_{j=1}^{m-2} \left(\frac{j}{m}(1 + D_j) + \frac{m-1-j}{m}(1 + D_{m-1-j}) \right) & m \geq 3 \end{cases}$$

$$m \cdot D_m = \begin{cases} 0 & m = 1 \\ 1 & m = 2 \\ (m-1) + \frac{1}{m-2} \sum_{j=1}^{m-2} \left(j \cdot D_j + (m-1-j) \cdot D_{m-1-j} \right) & m \geq 3 \end{cases}$$

1.12 Operations in Jumplist

Expected Search Costs [2]

$$m \cdot D_m \quad =: \quad P_m \quad = \quad m - 1 + \begin{cases} 0 & m \leq 2 \\ \dfrac{2}{m-2} \displaystyle\sum_{j=1}^{m-2} P_j & m \geq 3 \end{cases}$$

Reminiscent of Quicksort average case = internal path length of random BST

$$C_n \quad = \quad (n-1) + \frac{2}{n} \sum_{j=0}^{n-1} C_j \quad C_0 \; = \; 0$$

for a systematic and precise way, see Nebel, Neumann, Wild 2019

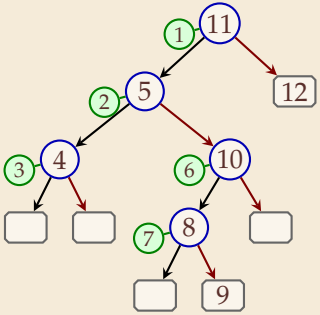
Indeed, here always $D_m \leq C_m$ for $m \geq 0$, so at least get the upper bound for free

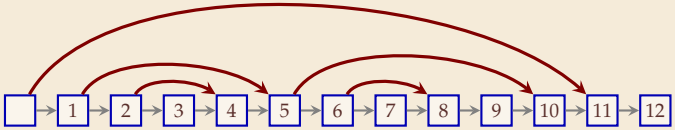
$$P_m \quad \leq \quad 2m \ln m \; \pm \; O(m) \quad \rightsquigarrow \quad D_m \quad \leq \quad \mathbf{2 \ln m} \pm O(1)$$

Searching in Jumplists costs same as in random BSTs! 🤔

Dangling-Min BSTs

Indeed, jumplists are in bijection with a kind of BST . . . but a slightly awkward one.





$$\textsc{minBST}\left(\square \to x_1 \ \ldots \ x_n\right) = \boxed{x_1, \ldots, x_n} \qquad (m \le w)$$

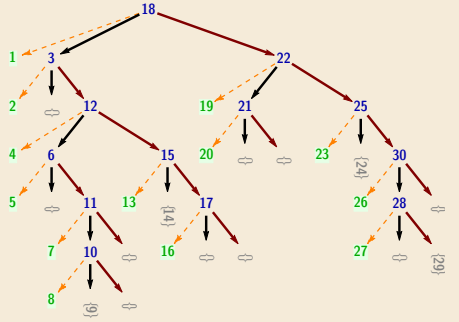
$$\textsc{minBST}\left(\square \to x_1 \square \to x_j \square\right) = \begin{array}{c} x_1 \quad x_j \\ \textsc{minBST}(\mathcal{J}_1) \quad \textsc{minBST}(\mathcal{J}_2) \end{array} \qquad (m > w)$$

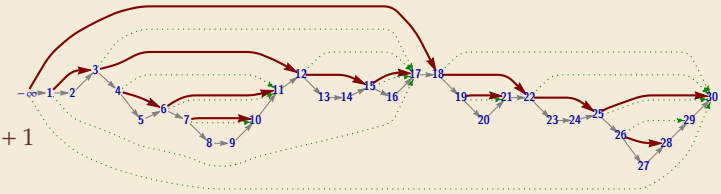
$\mathcal{J}_1$ $\mathcal{J}_2$

BST-View Inspired Search

SpineSearch(*head*, *x*)

```
    // Returns last node with key < x and its zero-based rank,
    // i.e., the number of nodes with key < x
1   rank := 0;  steppedOver := 0;  lastJumpedTo := head
2   repeat    // BST-style search
3       if head.jump.key < x
4           rank := rank + head.nsize + 1 + steppedOver
5           head := head.jump
6           steppedOver := 0; lastJumpedTo := head
7       else
8           head := head.next;  steppedOver := steppedOver + 1
9       end if
10  until head is PlainNode
11  head := lastJumpedTo
    // Linear search from lastJumpedTo
12  while head.next.key < x
13      head := head.next;  rank := rank + 1
14  end while
15  return (head, rank)
```





So far: Random Jumplists

How do we get to *Randomized Jumplists*?

Here: Expected subtree size in Random BST $O(\log n)$ nodes $\rightsquigarrow$ could rebuild entire subtree

- $\rightsquigarrow$ Simulate Randomized BST algorithms (top-down, coin flips),
- $\rightsquigarrow$ rebuild subtree if coin says so.

Jumplist Rebalance

Takes a sublist (via its header) and draws new random jump pointers for entire sublist.

$$\textsc{Reb}\left(\boxed{v_0 \ \ldots \ v_n}\right) = \boxed{v_0 \ \ldots \ v_n} \qquad (m \leq w)$$

$$\textsc{Reb}\left(\boxed{v_0 \ \ldots \ v_n}\right) = \boxed{v_0} \longrightarrow \textsc{Reb}\left(\boxed{}\right) \longrightarrow \textsc{Reb}\left(\boxed{v_{\mathcal{J}}}\boxed{\ v_n}\right) \qquad (m > w)$$

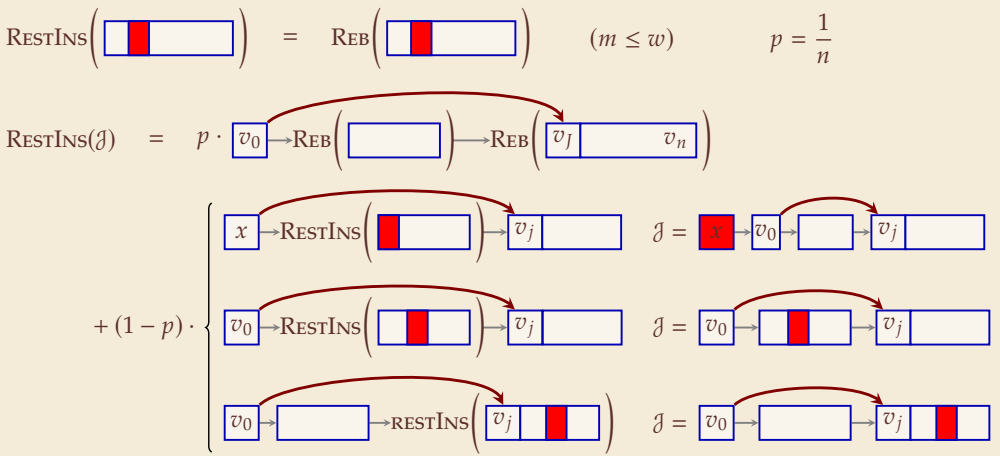
Draw $\mathcal{J} \stackrel{\mathcal{D}}{=} \text{Uniform}\{v_2, \ldots, n_n\}$

Can be implemented to use a single traversal and set the jump pointers upon recursive ascent

Restore After Insert

Insert(x) searches for correct position and adds a new node v for x.

RestoreAfterInsert draws a jump pointer for v and, potentially, makes v the target of an earlier jump pointer ($\approx$ InsertAtRoot).



1.13 Fringe Balancing

Can we be more balanced than Random BSTs?

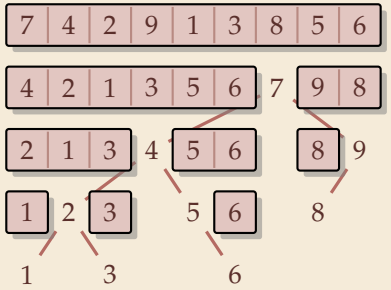
- worst-case height bounds of balanced BSTs usually worse than average depth of Random BST
- but **average** costs empirically much better (just open how to analyze . . .)

$\rightsquigarrow$ can randomized data structures have better expected search times than $2 \ln n$?

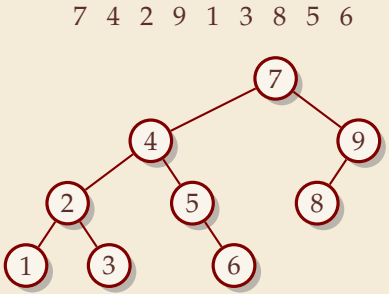
- What could a suitable random model for BSTs look like?

Quicksort & Binary Search Trees

Quicksort



Binary Search Tree (BST)



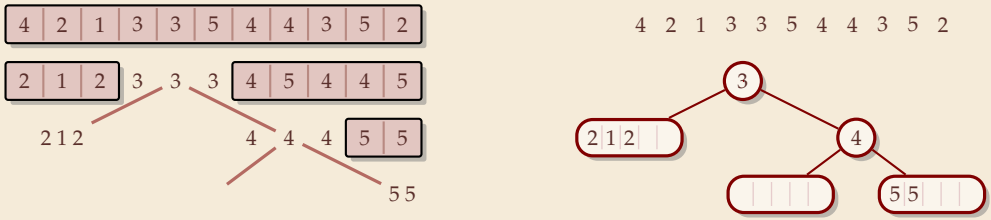
- recursion tree of quicksort = binary search tree from successive insertion
- comparisons in quicksort = comparisons to built BST
- comparisons in quicksort $\approx$ comparisons to search each element in BST

Fringe-Balanced BSTs

***k*-Fringe-Balanced Search Trees:**

- Leaves *buffer* $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with *chosen pivot*.

Median-of-5 Quicksort ($t = 2$) **5-Fringe-Balanced Tree**



$\rightsquigarrow$ **Correspondence extends to**

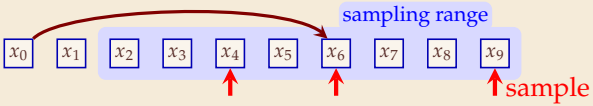
- Larger Insertionsort Cutoff
- (Handling equal keys)

$\rightsquigarrow$ Expected depth of typical node: $\dfrac{1}{H_{k+1} - H_{t+1}} \ln n$ instead of $2 \ln n$ $\qquad k = 5 \rightsquigarrow 1.12 \lg n$

Randomized Fringe-Balanced Jumplists

Rebalancing of Jumplists can draw jump pointers as median-of-k.

$\rightsquigarrow$ $J \stackrel{\mathcal{D}}{=} \mathrm{BetaBin}(m-2-k; t+1, t+1) + t + 2$ instead of uniform



Details in **Nebel, Neumann, Wild**: *Median-of-k Jumplists and Dangling-Min BSTs*, ANALCO **2019**

+ full Java implementation

Theorem 7.1:
Consider randomized median-of-k jumplists with leaf size w on n keys, where k and w are fixed constants. Abbreviate by $H(k) = H_{k+1} - H_{(k+1)/2}$ for H_n the harmonic numbers. Then the following holds:

(a) *The expected number of key comparisons in a* ***spine search*** *is asymptotic to $1/H(k) \cdot \ln n$, as $n \to \infty$, when each position is equally likely to be requested.*
(b) *The expected number of rebalanced elements in the* ***cleanup after insertion*** *is asymptotic to $k/H(k) \cdot \ln n$, as $n \to \infty$, when each of the $n+1$ possible gaps is equally likely.*
(c) *The expected number of rebalanced elements in the* ***cleanup after deletion*** *is asymptotic to $k/H(k) \cdot \ln n$, as $n \to \infty$, when each key is equally likely to be deleted.*
(d) *The expected number of additional machine words per key required to store the jumplist is asymptotically at most $1 + \frac{2}{(w+1)H(k)}$ as $n \to \infty$.*

Median-of-k Jumplists

- At increase in update cost can obtain more balanced shape
- Search cost slightly better
- *Same* relative effect as median-of-k in Quicksort

- Similar approach possible for RBSTs (nobody worked out the details)

Summary

Randomization for Sorted Dictionaries


various schemes to keep data structures in good shape via randomization

- rich design space

⇝ even better may be ready to be discovered


clear winner in code size and simplicity


some variants bring additional benefits

- traversal in sorted order
- more balanced than random BSTs


individual items may still be deep in tree (almost unlikely)

- repeatedly accessing such an item, adversary might drive up costs

⇝ it would be cool to have frequently / recently accessed items close to root!
Aiming at random BSTs doesn't do that.