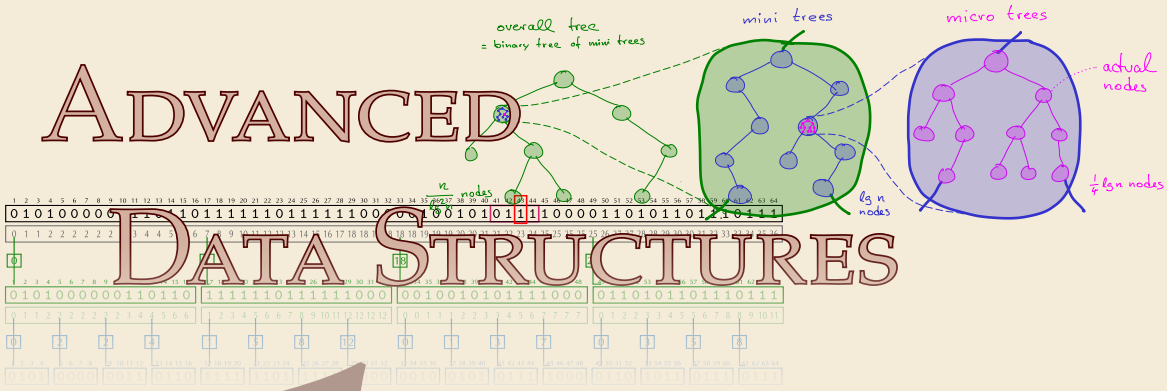


Advanced Data Structures

1 Randomized Trees

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

CS650 (Summer 2026)
Philipps-Universität Marburg
version 2026-04-13 07:31 H

Outline

1 Randomized Trees

1.1 Recap: Sorted Dictionary and BSTs
1.2 What's wrong with BBSTs?
1.3 Random BSTs
1.4 Treaps
1.5 Updates in Treaps
1.6 Insertion at Root
1.7 Randomized Binary Search Trees
1.8 Skip Lists
1.9 Jumplists

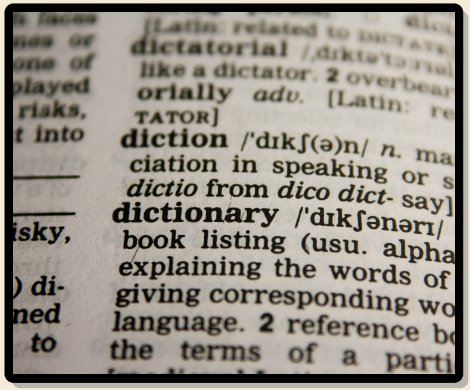
1.1 Recap: Sorted Dictionary and BSTs

Symbol table ADT

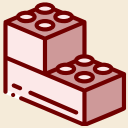
Java: `java.util.Map<K,V>`

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()`

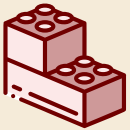


Most fundamental building block in computer science.

(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

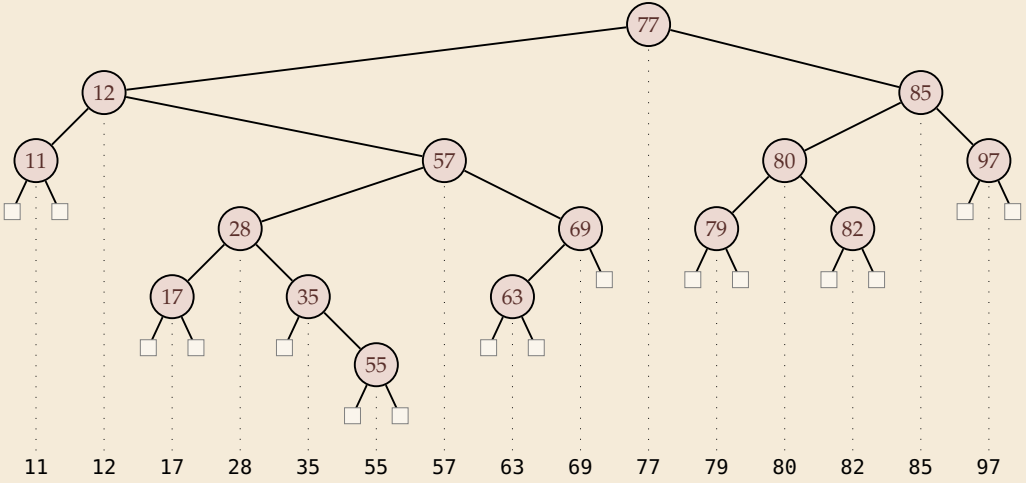
Binary search trees (BSTs) $\approx$ dynamic sorted array

- 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

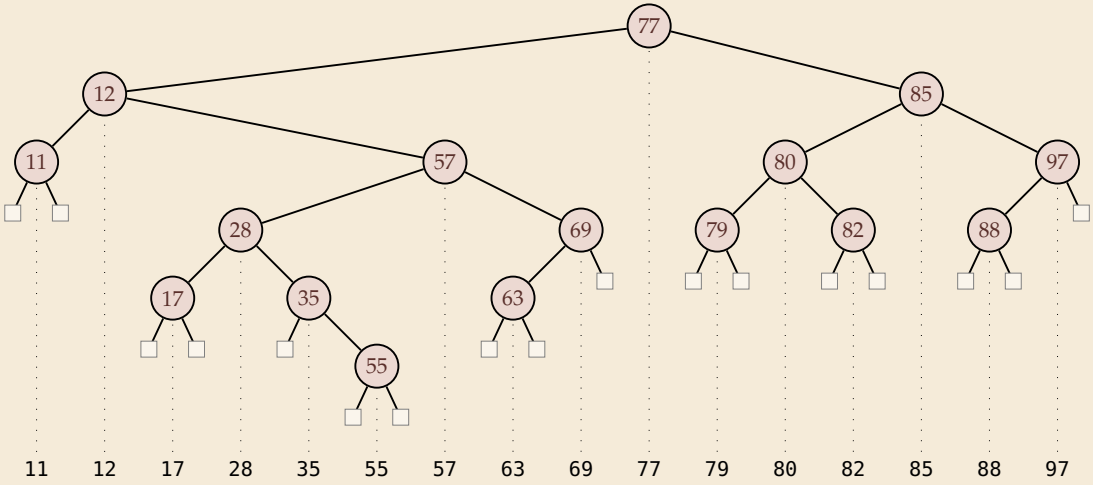
- Standard trick: ***Augmented BSTs***
  - Each node stores the **size** of its **subtree**
  - Easy to maintain upon updates
  - $\rightsquigarrow$ Allows to answer all ordered-symbol-table operations

BST example & find



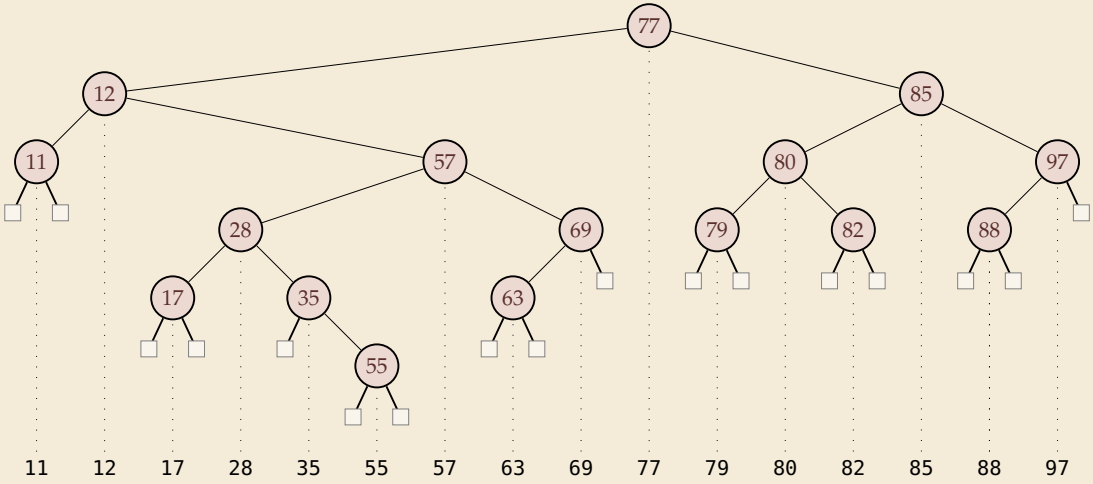
BST insert

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?*

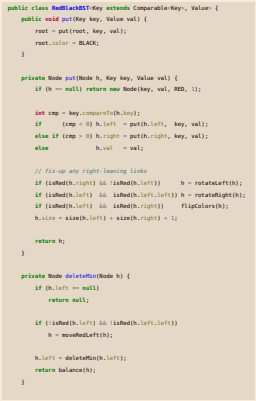
A Primitive for BSTs: Rotations

1.2 What's wrong with BBSTs?

Red-Black Tree Implementation

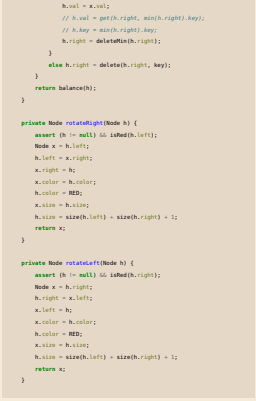
`RedBlackTree` from Sedgewick & Wayne (excerpt) `algs4.cs.princeton.edu/code`

- 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.)

This week: Randomized solutions.

- Recall: Randomization means we explicitly add randomness that we control

≠ average-case running time where we assume the **input** to be random
(much weaker!)

⇝ either way, performance is now a random variable

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 in T_{n+1} is in bijection with permutations of $[n+1]$,
which is in bijection with a permutation of $[n]$ and a *last value* in $[n+1]$, where we increment all values $\geq$ the last value by 1.

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

Corollary 1.3

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

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 $[i]$

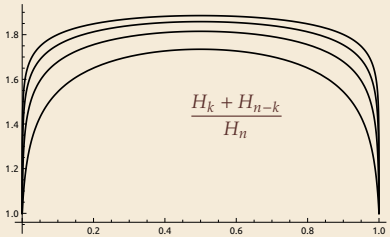
$\rightsquigarrow \quad X_i = [\pi_i = 1]$ and so $\mathbb{P}[X_i = 1] = \frac{1}{i}$

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

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 $\alpha \in (0,1)$. ◀

Analysis of Random BSTs – Concentration

Theorem 1.7 (Concentration of left-to-right minima)

One can show that the number of left-to-right minima in a permutation of length n is in $O(\log n)$ w.h.p. (using general Chernoff bound).
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

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

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

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

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 1.8 (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

Remark 1.9 (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 *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`

1.4 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.

$\rightsquigarrow$

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 1.10 (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 1.11 (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. ◀

Randomized Treaps

Definition 1.12 (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 $p_i \overset{\mathcal{D}}{=} \text{Uniform}(0,1)$ independent of all previous priorities. ◀

Theorem 1.13 (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 1.14 (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. ◀

1.5 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 bother about rotations?

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

Depth of Internal Nodes

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 1.15 (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 $x \neq y$. ◀

Depth of Internal Nodes – Proof

Proof:

Depth of Internal Nodes [2]

Remark 1.16 (Common ancestor indicators)

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

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

Theorem 1.17 (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 $H_k + H_{n-k+1} - 2$. ◀

Proof:

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

Subtree sizes

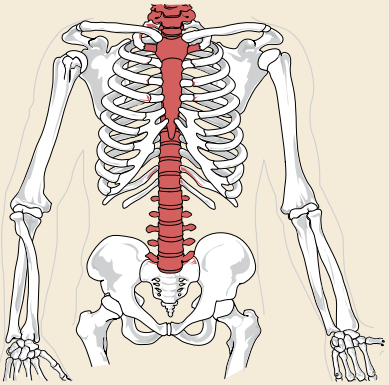
Remark 1.18 (Expected subtree size)

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 $\textcircled{k} = \sum_{x=1}^{n} A_x^k$ ■

Spines of Trees



Lemma 1.19 (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 1.20 (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:

1.6 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

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

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

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

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

Join

```
private 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, right) :
        join(left, root.child);
    updateSize(root);
    return root;
}
```

Insert at Root, Delete

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

Node delete(Node t, int key) {
    if (t == null) return null;
    if (key == t.key) return join(t.child, t.child);

    int dir = key < t.key ? 0 : 1;
    t.child[dir] = delete(t.child[dir], key);
    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; }
```

1.7 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, x should have a $1/S(y)$ chance to replace y 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

(Update of size fields omitted)

```
1  Node insert(Node root, int x)
2      n = root.size;
3      r = Uniform[0..n] // uniform int between 0 and n
4      if (r == n) return insertAtRoot(root,x)
5      if (x < root.key)
6          root.left = insert(root.left, x)
7      else
8          root.right = insert(root.right, x)
9      return root
10
11 Node insertAtRoot(Node root, int x)
12     (smallerRoot,largerRoot) = split(root, x)
13     return new Node(x, smallerRoot, largerRoot)
```

Delete in RBSTs

(Update of size fields omitted)

```
1 Node delete(Node root, int x)
2     if (root == null) return null
3     if (x < root.key)
4         root.left = delete(root.left, x)
5     else if (x > root.key)
6         root.right = delete(root.right, x)
7     else // must delete root
8         return join(root.left, root.right)
```

RBST Analysis

Theorem 1.21 (Correctness)

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

Theorem 1.22 (Operation Costs)

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

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

◀

1.8 Skip Lists

1.9 Jumplists