

ADVANCED

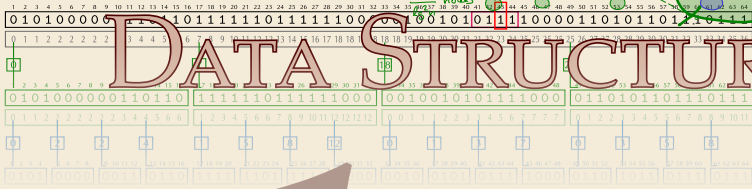
overall tree
= binary tree of mini trees

mini trees

micro trees

actual nodes

$\frac{1}{4} \lg n$ nodes



DATA STRUCTURES

1

Randomized Trees

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

Outline

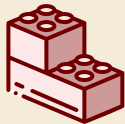
1 Randomized Trees

- 1.1 Recap: Sorted Dictionary and BSTs
- 1.2 What's wrong with BBSTs?
- 1.3 Random BSTs
- 1.4 Properties of Random BSTs
- 1.5 Treaps
- 1.6 Updates in Treaps
- 1.7 Insertion at Root
- 1.8 Randomized Binary Search Trees
- 1.9 Skiplists
- 1.10 Operations in Skiplists
- 1.11 Jumplists
- 1.12 Operations in Jumplist
- 1.13 Fringe Balancing

1.1 Recap: Sorted Dictionary and BSTs

Ordered symbol tables

- ▶ `min()`, `max()`
Return the smallest resp. largest key in the ST
- ▶ `floor(x)`, $\lfloor x \rfloor = \mathbb{Z}.\text{floor}(x)$
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 i th 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

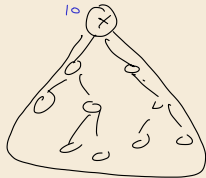
Binary search trees

Binary search trees (BSTs) \approx dynamic sorted array

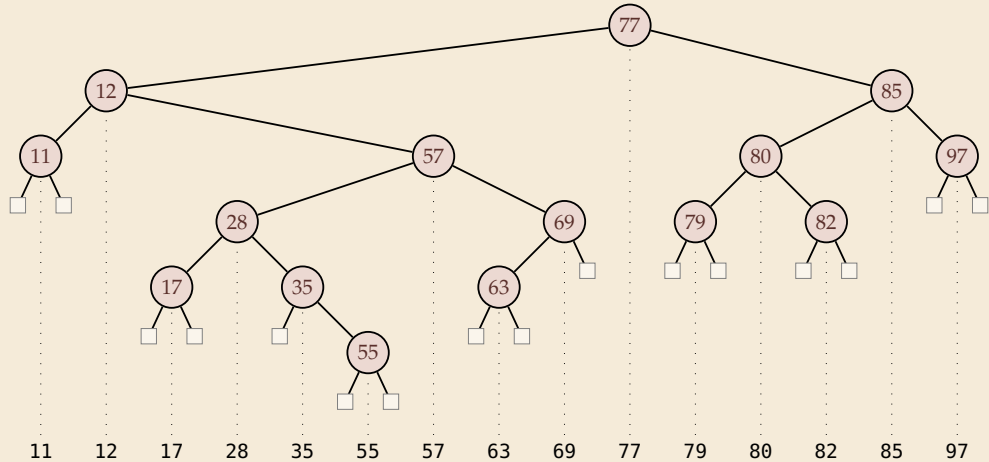
- ▶ binary tree
 - ▶ Each node has left and right child
 - ▶ Either can be empty (null)
- ▶ Keys satisfy *search-tree property*

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
 - ↪ Allows to answer all ordered-symbol-table operations

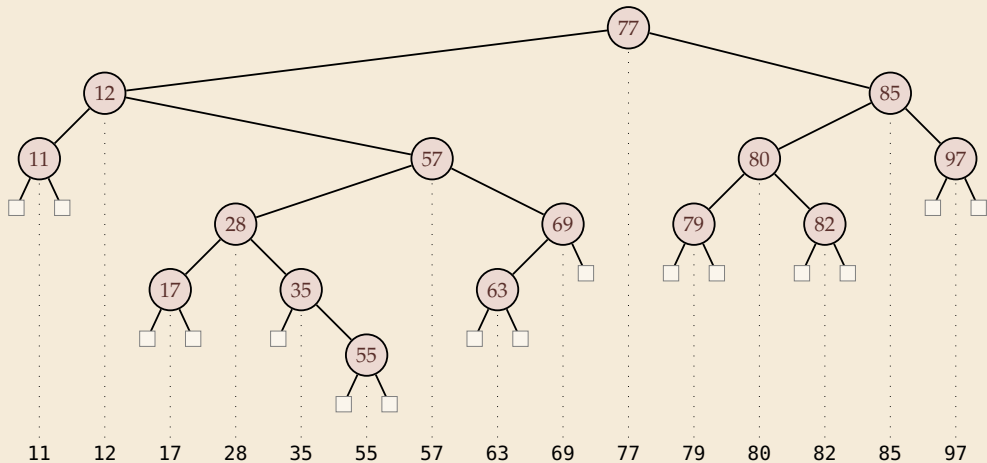


BST example & find



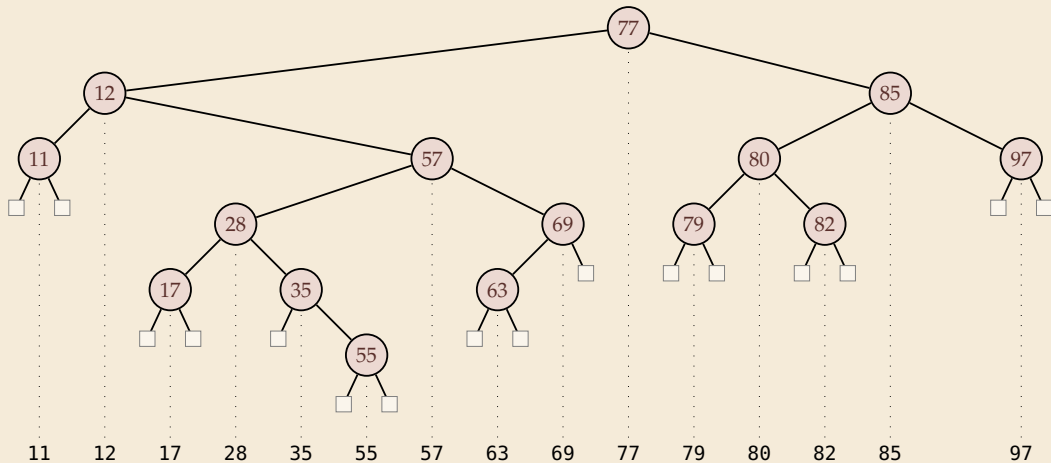
BST insert

Example: Insert 88



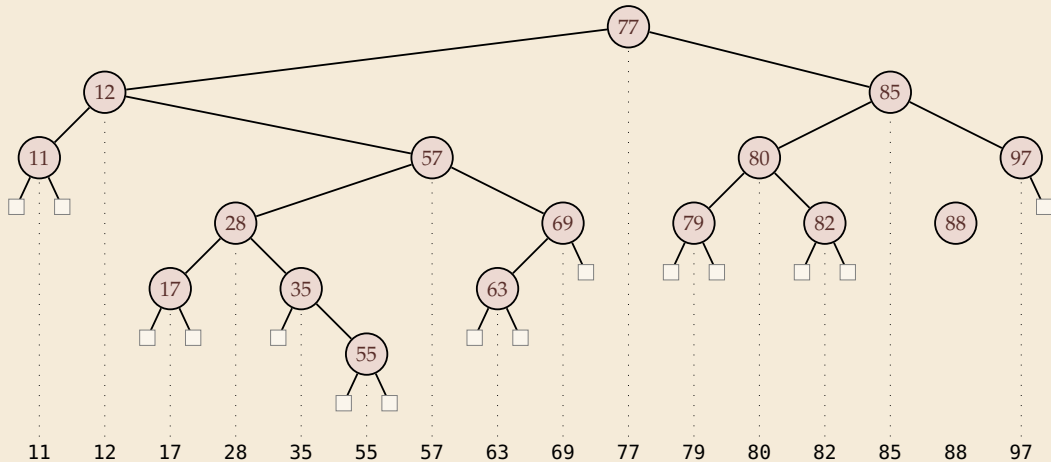
BST insert

Example: Insert 88



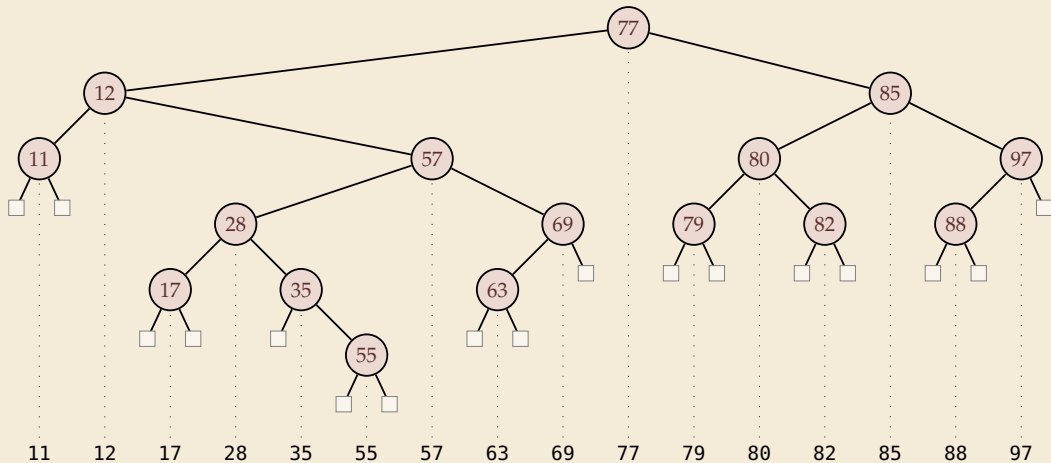
BST insert

Example: Insert 88



BST insert

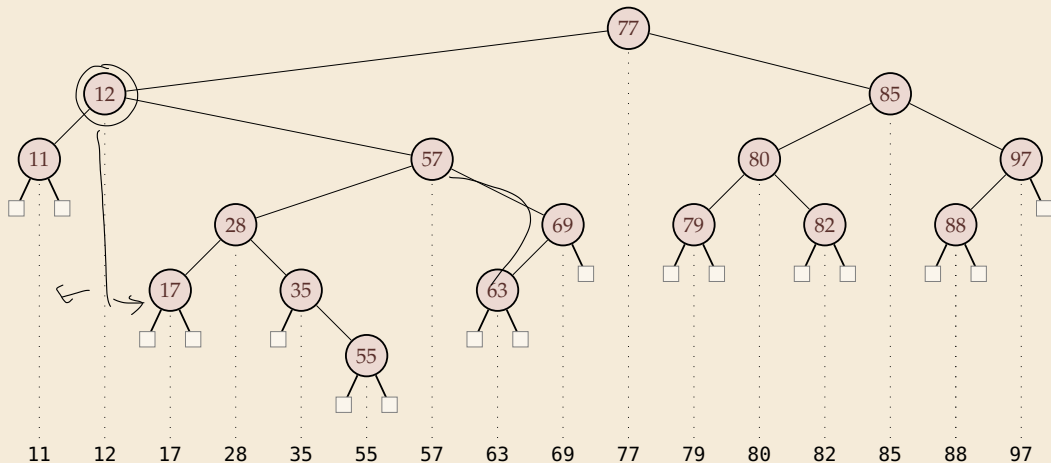
Example: Insert 88



BST delete

Hibbard's deletion

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



Unbalanced Binary Search Trees

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

$h = \text{height}$ of the BST

$$1 - 2^{-n}$$

► Height h of unbalanced BST depends on insertion order

► satisfies $\lg n \leq h \leq n$

$$\lg = \log_2$$

$$\text{Prob} = 1 - o(1)$$

► 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

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*
 - ▶ *weight-balanced trees* (BB[α] trees)
- } every undergrad data-structures module
I know includes at least one of these two

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

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*
 - ▶ *weight-balanced trees* (BB[α] trees)

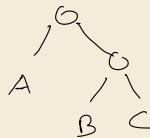
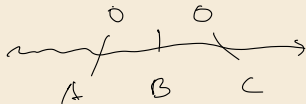
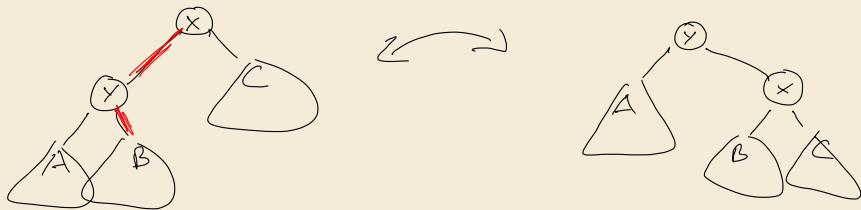
} every undergrad data-structures module
I know includes at least one of these two

▶ Properties

- ▶ All of them guarantee $h = O(\log n)$ at all times (constants differ!)
- ▶ All of them work with rotations along the search path
- ↪ $O(\log n)$ 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 Sedgwick & 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;
        else if (isRed(h.left) && !isRed(h.left.left))
            h = moveRedLeft(h);
        else 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;
    else if (isRed(h.right) && !isRed(h.right.left))
        h = moveRedRight(h);
    else 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.

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.*

Recall: Average-Case Analysis vs. Randomized Algorithms

This unit: *Randomized solutions*.

Average-Case Analysis

- ▶ algorithm is **deterministic**
same input, same computation

Randomized Algorithm (here)

- ▶ algorithm is **not** deterministic
same input, potentially different comp.

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**

Randomized Algorithm (here)

- ▶ algorithm is **not** deterministic
same input, potentially different comp.
- ▶ input is chosen **adversarially** (worst-case inputs)

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

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 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:

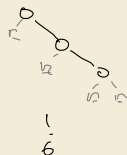
Definition 1.1 (Random BST)

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

Example: $n = 3$

$= \{1, \dots, n\}$

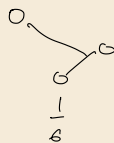
1 2 3



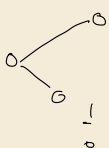
3 2 1



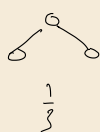
1 3 2



3 1 2



2 1 3



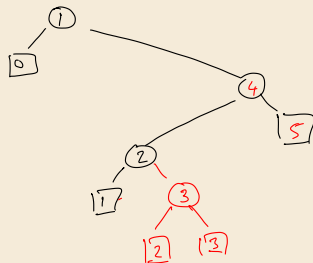
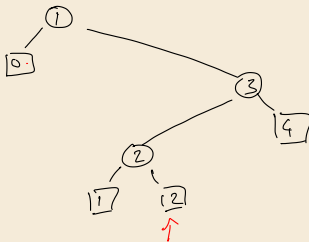
2 3 1

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. ◀

$n = 3$



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 $[n + 1] = \{1, \dots, n + 1\}$, ^{$[0..n]$}
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: $1, 2, 4, 7, 6, 5 \mid 3 \hat{=} ((1, 2, 3, 6, 5, 4), 2)$

$\{1..n+1\}$

“

$\{1, 2, \dots, n+1\}$

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*.

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

- ▶ $\mathbb{P}[\bar{A}] = 1 - \mathbb{P}[A]$ counter-probability ($\bar{A} = \Omega \setminus A$)
- ▶ $\mathbb{P}[\bigcup A_i] \leq \sum_i \mathbb{P}[A_i]$ the *union bound* (a.k.a. Boole's inequality a.k.a. σ -subadditivity)

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

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

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.

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

discrete, $\mathcal{F} = 2^{\Omega}$ (powerset of Ω)

- ▶ $\mathbb{P}[\bar{A}] = 1 - \mathbb{P}[A]$ counter-probability ($\bar{A} = \Omega \setminus A$)
- ▶ $\mathbb{P}[\cup A_i] \leq \sum_i \mathbb{P}[A_i]$ the union bound (a.k.a. Boole's inequality a.k.a. σ -subadditivity)
- ▶ $\{A_1, \dots, A_k\}$ (mutually independent) $\iff \mathbb{P}[\cap_i A_i] = \prod_i \mathbb{P}[A_i]$
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 : $\mathbb{P}[A | B] = \mathbb{P}[A \cap B] / \mathbb{P}[B]$
generally undefined if $\mathbb{P}[B] = 0$
- ▶ law of total probability: If $\Omega = B_1 \dot{\cup} B_2 \dot{\cup} \dots$ is a partition of Ω , we have

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

Recall: Random Variables

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

Recall: Random Variables

Random variables (r.v.) $X : \Omega \rightarrow \mathcal{X}$; 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 $\mathbb{1}_A(\omega) = [\omega \in A] = \begin{cases} 1 & \omega \in A \\ 0 & \text{else} \end{cases}$

Recall: Random Variables

Random variables (r.v.) $X : \Omega \rightarrow \mathcal{X}$; 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 $\mathbb{1}_A(\omega) = [\omega \in A]$
- ▶ $F_X(x) = \mathbb{P}[X \leq x]$ is the *cumulative distribution function (CDF)*.
- ▶ for discrete r.v. X define $f_X(n) = \mathbb{P}[X = n]$ the *probability mass function (PMF)*.

Recall: Random Variables

Random variables (r.v.) $X : \Omega \rightarrow \mathcal{X}$; 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 $\mathbb{1}_A(\omega) = [\omega \in A]$
- ▶ $F_X(x) = \mathbb{P}[X \leq x]$ 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:

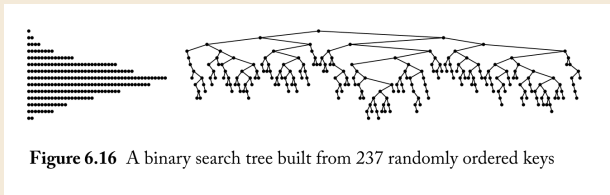
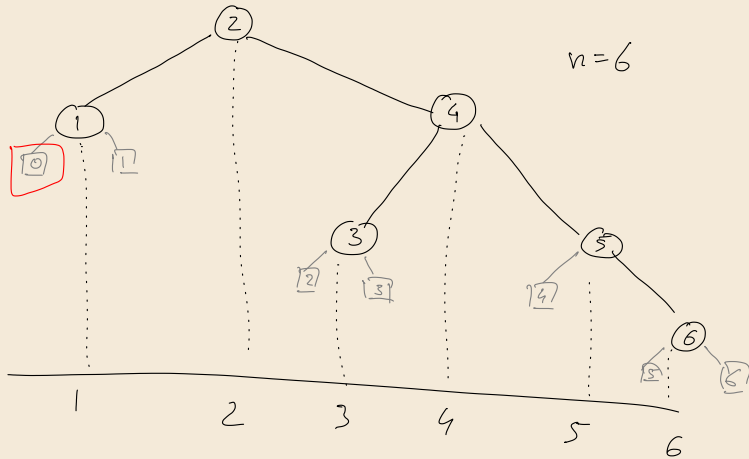
- ▶ X and Y independent $\iff \mathbb{P}[X = x \wedge Y = y] = \mathbb{P}[X = x] \cdot \mathbb{P}[Y = y]$ for all x, y .
(Naturally follows from independent events)
- ▶ a sequence of *i.i.d.* r.v. X_1, X_2, \dots (*independent and identically distributed*)
has $X_i \stackrel{\mathcal{D}}{\leftarrow} X_1$ and $\{X_i\}_{i \geq 1}$ 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



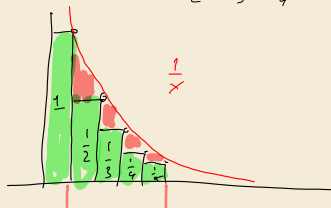
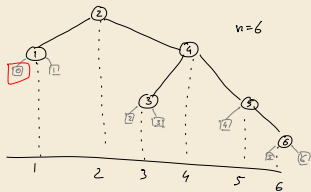
Analysis of Random BSTs I

$$\frac{H_n}{\ln n} \rightarrow 1 \quad (n \rightarrow \infty)$$

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 $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \sim \ln n$.

$$= \sum_{i=1}^n \frac{1}{i}$$



$$\int_1^n \frac{1}{x} dx \approx \ln(n) - \ln(1)$$

Analysis of Random BSTs I

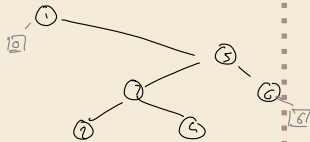
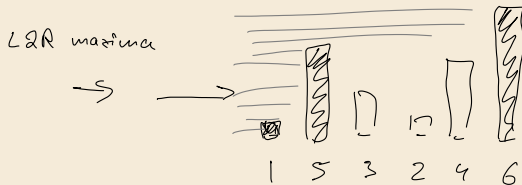
Theorem 1.4 (Expected depth of leftmost leaf)

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 $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \sim \ln n$. ◀

Proof:

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

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



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 $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \sim \ln n$. ◀

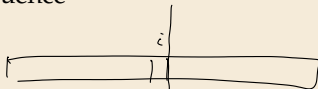
Proof:

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

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

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

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



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 $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \sim \ln n$. ◀

Proof:

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

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

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

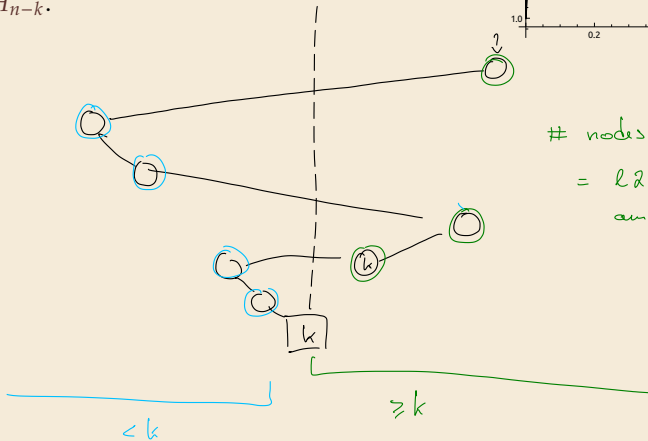
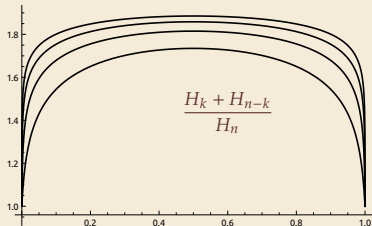
$$\rightsquigarrow X_i = [\pi_i = 1] \text{ and so } \mathbb{P}[X_i = 1] = \frac{1}{i}$$

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

Analysis of Random BSTs II

Theorem 1.5 (Expected depth of k th leaf)

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

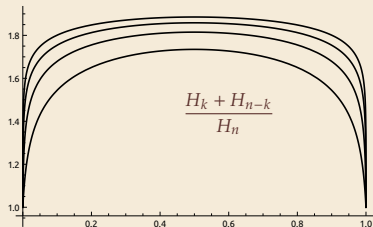


nodes right of k
 = $\ell \& r$ minima
 among keys $\geq k$

Analysis of Random BSTs II

Theorem 1.5 (Expected depth of k th leaf)

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



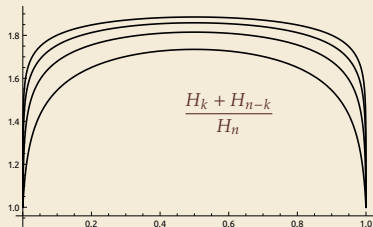
Proof:

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

Analysis of Random BSTs II

Theorem 1.5 (Expected depth of k th leaf)

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



Proof:

$$\begin{aligned} \text{depth}(\boxed{k}) &= \text{\#comparisons for unsuccessful search for a key } x \text{ that terminates in } \boxed{k}. \\ &= \text{\#comparisons with keys } < x + \text{\#comparisons with keys } > x \end{aligned}$$

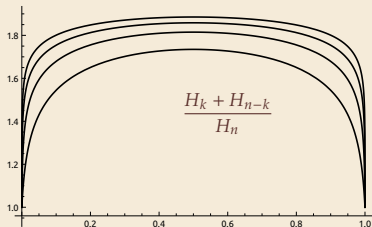
$$\ll k - \frac{1}{2}$$



Analysis of Random BSTs II

Theorem 1.5 (Expected depth of k th leaf)

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



Proof:

$$\begin{aligned} \text{depth}(\boxed{k}) &= \text{\#comparisons for unsuccessful search for a key } x \text{ that terminates in } \boxed{k}. \\ &= \text{\#comparisons with keys } < x + \text{\#comparisons with keys } > x \\ &= \text{\#} \overset{\text{max}}{2^r - 1} \text{ mins among keys } < x + \text{\#} \overset{\text{min}}{2^r - 1} \text{ maxs with keys } > x \end{aligned}$$

Analysis of Random BSTs II

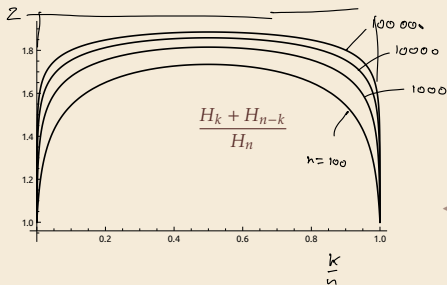
Theorem 1.5 (Expected depth of k th leaf)

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

Proof:

$$\begin{aligned} \text{depth}(\boxed{k}) &= \text{\#comparisons for unsuccessful search for a key } x \text{ that terminates in } \boxed{k}. \\ &= \text{\#comparisons with keys } < x \text{ + \#comparisons with keys } > x \\ &= \text{\#12r-mins among keys } < x \text{ + \#12r-maxs with keys } > x \end{aligned}$$

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

$$2(n+1)(H_{n+1} - 1) = 2n \ln n - 2(1-\gamma)n \pm O(\log n). \quad (\gamma \approx 0.5772 \text{ the Euler-Mascheroni constant})$$

(b) The depth of the α th leaf in a random BST of n keys $\sim \frac{2 \ln n}{1.39 \log_2 e}$ as $n \rightarrow \infty$

for any fixed $\alpha \in (0, 1)$.

$$\mathbb{E} \left[\sum_{k=0}^n \text{depth}(\lfloor \frac{k}{2} \rfloor) \right] = \sum_{k=0}^n (H_k + H_{n-k})$$

$$\sum_{i=1}^n H_i = (n+1)H_n - n, \quad = 2 \sum_{k=0}^n H_k$$

$$H_{n+1} = H_n + \frac{1}{n+1}$$

recursion tree of Quicksort

$$H_k + H_{n-k} = H_{\alpha n} + H_{(1-\alpha)n}$$

$$\sim \ln(\alpha n) + \ln((1-\alpha)n)$$

$$= 2 \ln(n) + \ln(\alpha) + \ln(1-\alpha)$$

$$\sim 2 \ln(n)$$

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
- ↪ both have expected costs of 2 steps.
- ▶ probably want A ... certainly would want to be able to distinguish them!

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
 - ↪ 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 $\mathbb{E}[X]$
 - ↪ *concentration bounds* a.k.a. *tail inequalities*
 - 👍 can then compare these typical times again
 - 👍 also obtain more reliable algorithms
 - ↪ 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
 $\forall c : \mathbb{P}[X(n)] \stackrel{\Delta}{=} 1 \pm O(n^{-c})$ as $n \rightarrow \infty$. $1 - d(c) \cdot n^{-c}$
- ▶ a random variable $X = X(n)$ is *in* $O(f(n))$ *with high probability (w.h.p.)* when
 $\forall c \exists d : \mathbb{P}[X \leq df(n)] = 1 \pm O(n^{-c})$ as $n \rightarrow \infty$.
(This means, the constant in $O(f(n))$ may depend on c .) ◀

Height / Depth of a leaf in Random BST $H = O(\lg n)$ w.h.p.

$$c=10 \quad \text{so } d_1, d_2 : \mathbb{P}[H \geq d_1 \lg n] \leq d_2 n^{-10}$$

(for example, $\mathbb{P}[\text{height} \geq 42 \lg n] \leq 2 n^{-7.4}$)

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 $n \rightarrow \infty$.
- ▶ a random variable $X = X(n)$ is *in* $O(f(n))$ *with high probability (w.h.p.)* when $\forall c \exists d : \mathbb{P}[X \leq df(n)] = 1 \pm O(n^{-c})$ as $n \rightarrow \infty$.
(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 \underline{c} from the definition of w. h. p. be given.



\rightsquigarrow events that happen with high probability can be combined

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 i th run of A fails happens with probability $O(n^{-(c+d)})$ by definition.

\rightsquigarrow events that happen with high probability can be combined



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 i th run of A fails happens with probability $O(n^{-(c+d)})$ by definition.

Then $\mathbb{P}[\cup_{i=1}^{n^d} F_i]$

\rightsquigarrow events that happen with high probability can be combined



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 i th run of A fails happens with probability $O(n^{-(c+d)})$ by definition.

$$\text{Then } \mathbb{P}[\cup_{i=1}^{n^d} F_i] \leq \sum_{i=1}^{n^d} \mathbb{P}[F_i]$$

\rightsquigarrow events that happen with high probability can be combined



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 i th run of A fails happens with probability $\underbrace{O(n^{-(c+d)})}$ by definition.

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

\rightsquigarrow events that happen with high probability can be combined



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 i th run of A fails happens with probability $O(n^{-(c+d)})$ by definition.

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

\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.

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 $X_1, \dots, X_n \in \{0, 1\}$ be (*mutually independent*) with $X_i \stackrel{\text{D}}{=} B(p_i)$.

Define $X = X_1 + \dots + X_n$ and $\mu = \mathbb{E}[X_1] + \dots + \mathbb{E}[X_n] = p_1 + \dots + p_n$. Then holds

$$\begin{aligned} \forall \delta > 0 & : \mathbb{P}[X \geq (1 + \delta)\mu] < \left(\frac{e^\delta}{(1 + \delta)^{1+\delta}} \right)^\mu \\ \forall \delta \in (0, 1] & : \mathbb{P}[X \geq (1 + \delta)\mu] \leq \underline{\exp(-\mu\delta^2/3)} \end{aligned}$$

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.

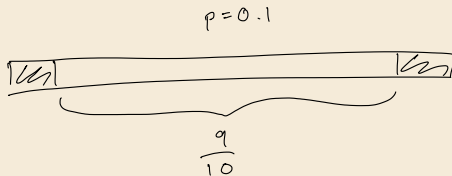
Corollary 1.10 (Chernoff Bound for Binomial Distribution)

Let $X \stackrel{d}{=} \text{Bin}(n, p)$. Then

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

$$\delta > \frac{1}{\sqrt{n}} = n^{-0.5} \quad n^{-0.499}$$

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



Analysis of Random BSTs – Concentration

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

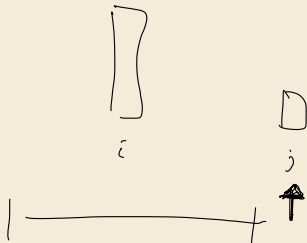
The number of left-to-right minima in a permutation of length n is in $O(\log n)$ w.h.p. Hence, the above expected results hold with high probability (up to constant factors). ◀

Proof (Idea):

Via *inversion table* of permutation:

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

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



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

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

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



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

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

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

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

*greater and left of *number* j*

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

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

randomness preservation! filtering out values $\leq x$
again a random permutation

$$\Pr \left[\begin{array}{c} \circ \\ / \quad \backslash \\ \circ \quad \circ \end{array} \right] = \frac{1}{3}$$

$$\Pr \left[\begin{array}{c} \circ \\ \backslash \\ \sigma \end{array} \right] = \frac{1}{2} \cdot \Pr \left[\begin{array}{c} \circ \\ \sigma \end{array} \right] = \frac{1}{2} \cdot \frac{1}{2} \cdot 1 = \frac{1}{4}$$

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 $\mathbb{E}[X_{nk}] = \frac{2^k}{n!} \binom{n}{k}$.
- (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 n th 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 \neq 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

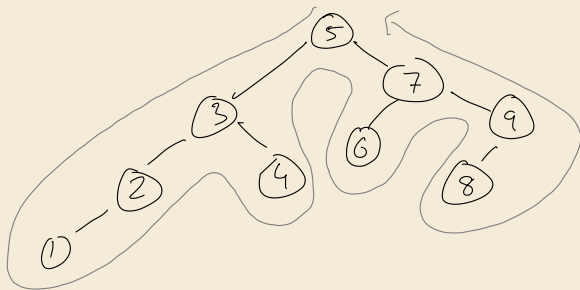
Treaps

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!

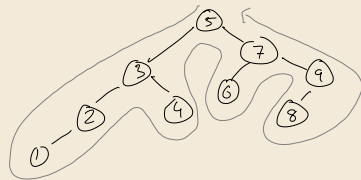
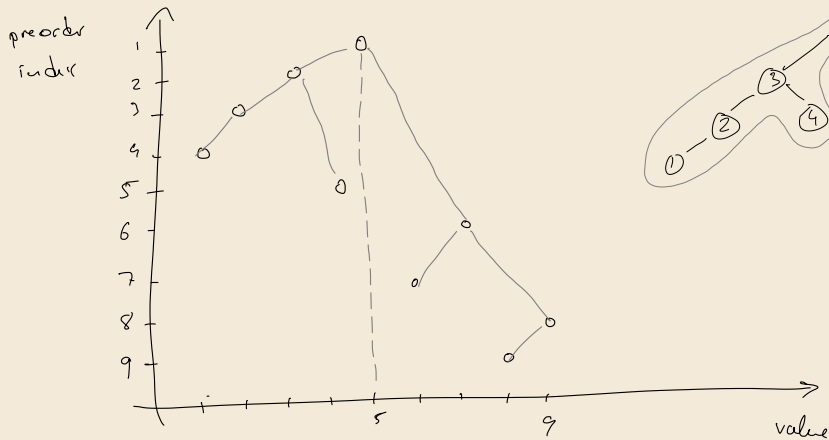
... but how? We have no control over the insertion of keys!



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 2 | 1 | 4 | 7 | 6 | 9 | 8 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

5 3 2 1 4 7 6 9 8
 1 2 3 4 5 6 7 8 9

Treap = Cartesian tree



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 $S = \{(k_1, p_1), \dots, (k_n, p_n)\}$ 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:

existence: use algorithm w/ sliding down horizontal line

uniqueness: by induction

IB $n=1$ ✓

IS: find $i = \arg \min p_j$ $k_i = \text{root}$

\Rightarrow treaps for all (k_j, p_j) with $k_j < k_i$

resp. (k_j, p_j) " $>$

are unique

□

Randomized Treaps

Definition 1.16 (Randomized Treaps)

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



Randomized Treaps

Definition 1.16 (Randomized Treaps)

A *randomized treap* is the unique treap that results from given keys k_1, k_2, \dots 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. ◀

Randomized Treaps

Definition 1.16 (Randomized Treaps)

A *randomized treap* is the unique treap that results from given keys k_1, k_2, \dots where (upon insertion) we assign k_i a priority $p_i \stackrel{\text{D}}{\equiv} \text{Uniform}(0, 1)$ 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: $\approx 1.39 \log n$

- (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, \dots, k_n into one for k_1, \dots, k_{n+1} ?

And vice versa?

Insertions and Deletions in Randomized Treaps

Up to now: *static* view on treaps.

But can we efficiently turn a randomized treap for k_1, \dots, k_n into one for k_1, \dots, k_{n+1} ?

And vice versa?

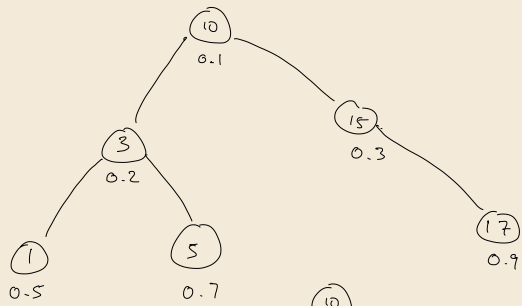
Yes!

- ▶ **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.

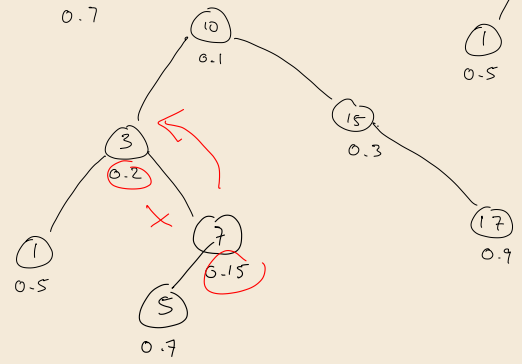
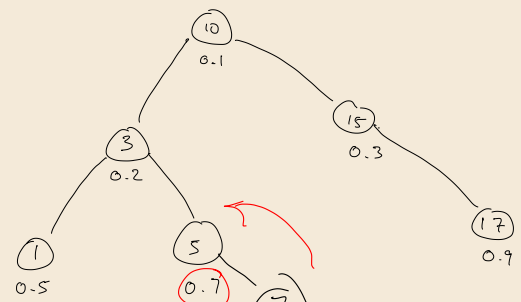
Conceptually very simple!

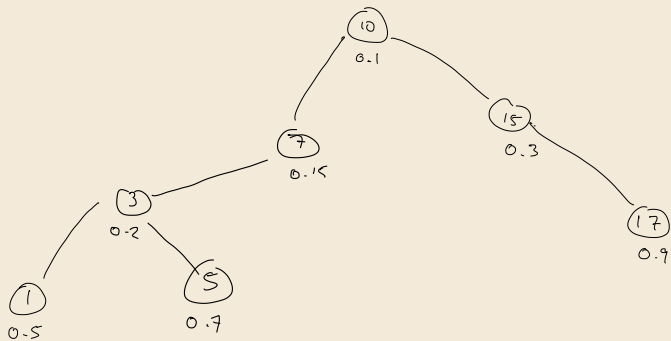
\rightsquigarrow all operations in $O(\log n)$ time w.h.p.!

in particular #rotations = $O(\log n)$ w.h.p



insert 7





Exam: Zip trees \approx treap w/ ties for priorities
are allowed / exploited

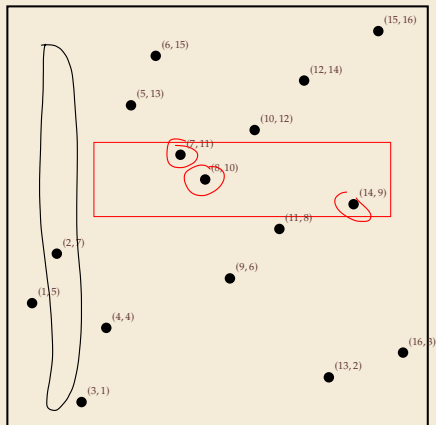
Excursion: Why care about counting rotations?

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

Example: *Range trees*

kd-trees, quad trees

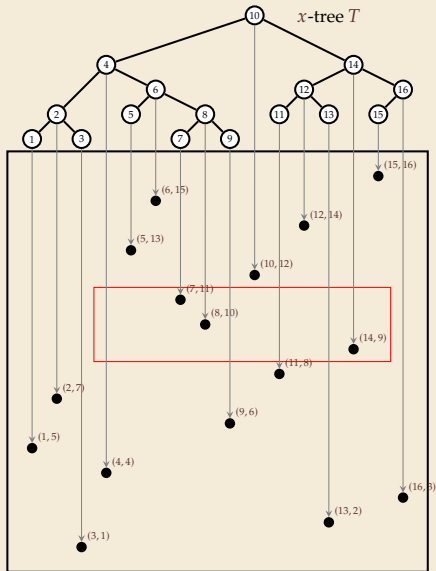
2d-orthogonal range search



Excursion: Why care about counting rotations?

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

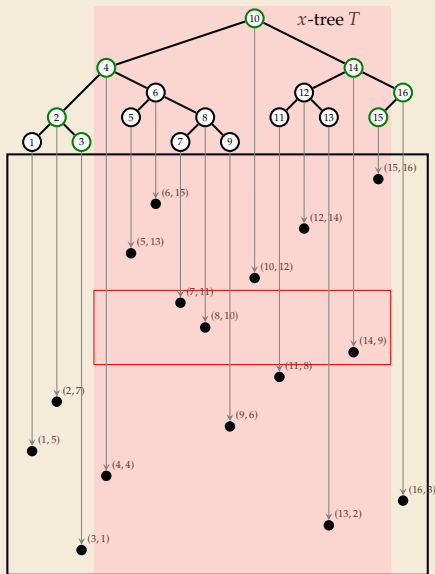
Example: *Range trees*



Excursion: Why care about counting rotations?

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

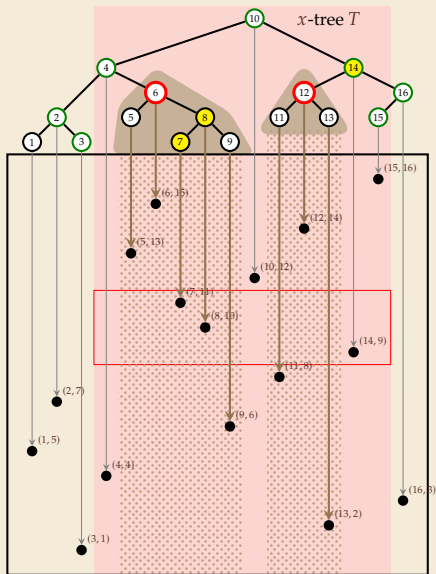
Example: *Range trees*



Excursion: Why care about counting rotations?

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

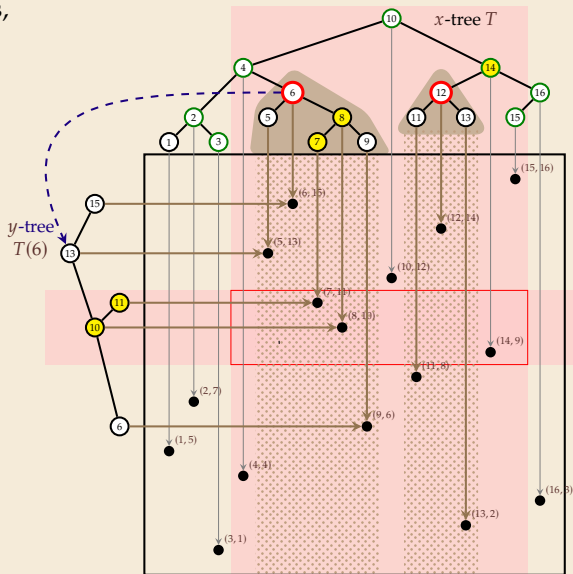
Example: *Range trees*



Excursion: Why care about counting rotations?

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

Example: *Range trees*

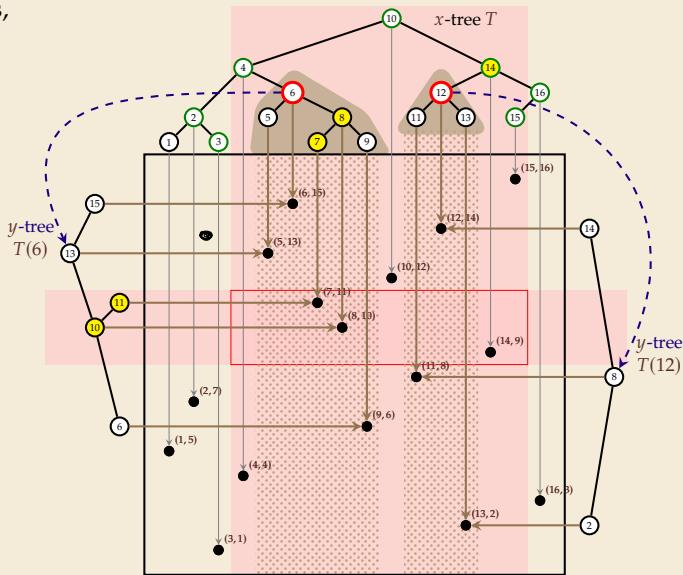


Excursion: Why care about counting rotations?

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

Example: *Range trees*

insert (x,y)
needs (potentially)
to rotate in x -heap



Ancestor Indicators

Can actually bound number of rotations much more tightly!

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.

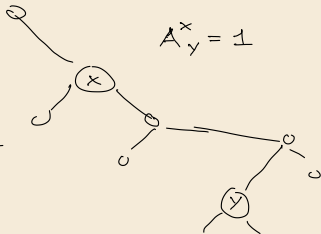
Analysis possible based on handy notion:

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 $A_y^x = [x \text{ is a } \underline{\text{proper}} \text{ ancestor of } y]$ 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 .

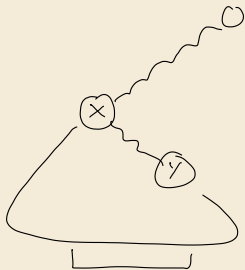
(c) $\Pr[A_y^x = 1] = \Pr[A_x^y = 1] = \frac{1}{|y - x| + 1}$ for $x \neq y$.

Ancestor Indicators – Proof

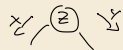
(a) $A_y^x = 1$ iff x was the *first* among the keys $[x..y] \cup [y..x]$ that was inserted into T_n .

Proof: $x < y$ wlog

(a) " \Rightarrow " $A_y^x = 1$



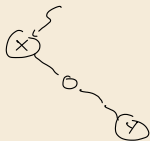
considers keys $[x..y]$
any other key $z \in (x..y)$



$\Rightarrow z$ separates x from y

$\Rightarrow x$ not ancestor of y \Leftarrow

" \Leftarrow " x inserted first among $[x..y]$



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

from (a) $\Pr[A_y^x = 1] = \Pr[x \text{ inserted first among } [x \dots y]]$

$$= \frac{1}{|y-x|+1}$$

\Rightarrow analysis of Quicksort?

never compare same pair twice

$$C = \sum_{1 \leq i < j \leq n} \underbrace{[i \text{ compared to } j]}_{\substack{\text{“(b)”} \\ A_j^i + A_i^j}} = \sum_{1 \leq i < j \leq n} \frac{2}{j-i+1} \quad \text{(c)}$$

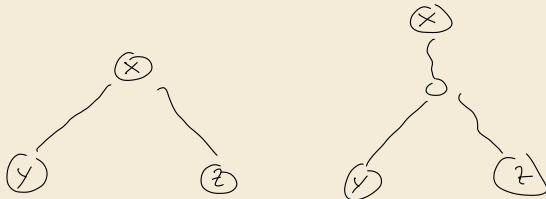
Common Ancestor Indicators

Remark 1.20 (Common ancestor indicators)

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

$$= A_y^x \cdot A_z^x$$

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



Depth of Internal Nodes

Theorem 1.21 (Expected depth of k th node)

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

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

Depth of Internal Nodes

Theorem 1.21 (Expected depth of k th node)

The *expected depth* of the k th internal node (for $k = 1, \dots, n$) 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:

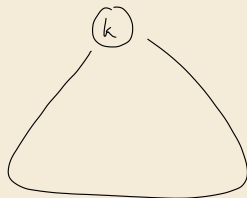
$$\text{depth}(\textcircled{k}) = \sum_{x=1}^n A_k^x$$

$$\begin{aligned} \mathbb{E}[\text{depth}(\textcircled{k})] &= \sum_{x=1}^n \mathbb{P}[A_k^x = 1] = \sum_{x=1}^{k-1} \frac{1}{k-x+1} + \sum_{x=k+1}^n \frac{1}{x-k+1} \\ &= \sum_{i=2}^k \frac{1}{i} + \sum_{i=2}^{n-k+1} \frac{1}{i} = H_k + H_{n-k+1} - 2 \end{aligned}$$

Subtree sizes

Remark 1.22 (Expected subtree size)

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



$$\Delta_x^k$$

Subtree sizes

Remark 1.22 (Expected subtree size)

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

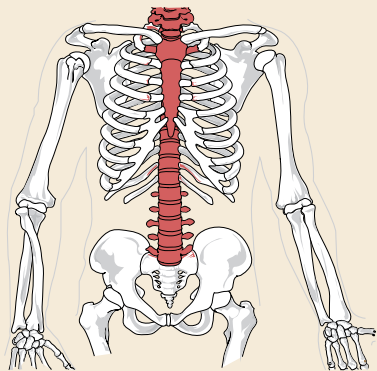
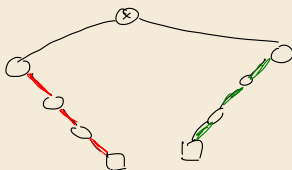
Proof:

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

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 $\underline{LS}(x) + \underline{RS}(x)$, 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 k th internal node (for $k = 1, \dots, n$) in random BST of n keys are given by

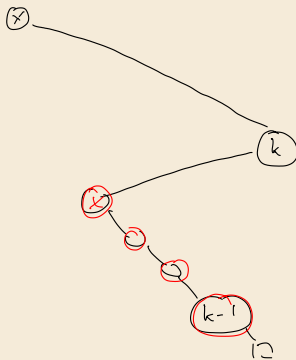
$$\mathbb{E}[LS(k)] = 1 - \frac{1}{k}$$

$$\mathbb{E}[RS(k)] = 1 - \frac{1}{n - k + 1} \quad \blacktriangleleft$$

Spines of Trees – Proof

Proof:

$$LS(k) = \sum_{x=1}^{k-1} (\underbrace{A_{k-1}^x}_{\text{red underline}} - C_{k-1,k}^x)$$



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)

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
 - ↪ 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

```
1 class Node {  
2     int key, size = 1;  
3     Node[] child = new Node[2]; // child[0] is Left, child[1] is Right  
4 }
```

Branchless Children

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

```
1 class Node {
2     int key, size = 1;
3     Node[] child = new Node[2]; // child[0] is Left, child[1] is Right
4 }
```

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

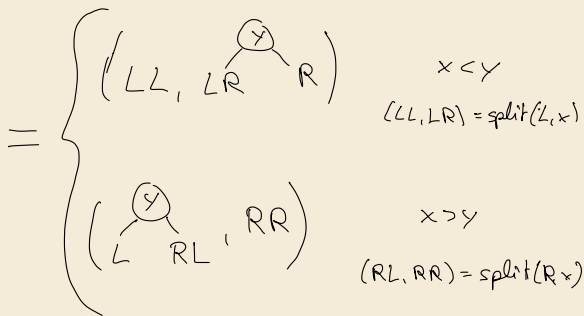
The following helper method is used to keep subtree size up to date

```
1 void updateSize(Node t) {
2     if (t != null) t.size = 1 + size(t.child) + size(t.child);
3 }
```

Split

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

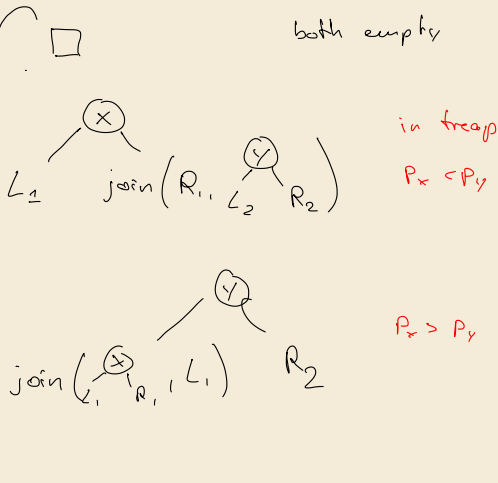
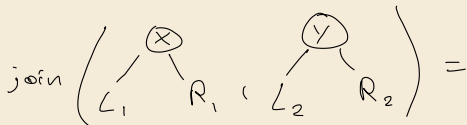
```
1 Node[] split(Node t, int x) {
2     if (t == null) return new Node[]{null, null};
3
4     int dir = x < t.key ? 0 : 1;
5     Node[] res = split(t.child[dir], x);
6
7     // Opposite child receives the remainder of split
8     t.child[dir] = res[1 - dir];
9     updateSize(t);
10
11    // Construct result dynamically based on dir
12    Node[] ans = new Node[2];
13    ans[dir] = res[dir];
14    ans[1 - dir] = t;
15    return ans;
16 }
```



Join

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

```
1 Node join(Node left, Node right) {  
2   if (left == null) return right;  
3   if (right == null) return left;  
4  
5   // new root chosen arbitrarily; here larger tree  
6   int dir = left.size > right.size ? 0 : 1;  
7   Node root = dir == 0 ? left : right;  
8   root.child = (dir == 0) ?  
9     join(root.child[1], right) :  
10    join(left, root.child[0]);  
11  updateSize(root);  
12  return root;  
13 }
```



Insert at Root, Delete

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

```
1 Node insertAtRoot(Node t, int x) {
2     Node[] splits = split(t, x);
3     Node root = new Node(x);
4     root.child = splits;
5     updateSize(root);
6     return root;
7 }
8
9 Node delete(Node t, int x) {
10    if (t == null) return null;
11    if (x == t.key) return join(t.child, t.child);
12    int dir = x < t.key ? 0 : 1;
13    t.child[dir] = delete(t.child[dir], x);
14    updateSize(t);
15    return t;
16 }
```

Tamio Nakajima's Treap Implementation

```
1 #include <random>
2 using namespace std;
3
4 struct node {
5     node *l, *r;
6     int prio, key, sum;
7 };
8 node nil_node = {&nil_node, &nil_node, 0, 0, 0};
9 node *nil = &nil_node; // Sentinel node
10
11 node *mod_child(node *n0, int id, node *child) {
12     node *n = new node;
13     *n = *n0;
14     (id == 1 ? n->r : n->l) = child;
15     n->sum = n->l->sum + n->key + n->r->sum;
16     return n;
17 }
18
19 node *join(node *l, node *r) {
20     return l == nil ? r : r == nil ? l :
21         l->prio > r->prio ?
22             mod_child(l, 1, join(l->r, r)) :
23             mod_child(r, 0, join(l, r->l));
24 }
```

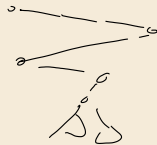
```
25
26 pair<node*,node*> split(node* n, int x) {
27     pair<node*, node*> ret = {nil, nil};
28     return n == nil ? ret : n->key < x ?
29         (ret = split(n->r, x), ret.first =
30             mod_child(n, 1, ret.first), ret) :
31         (ret = split(n->l, x), ret.second =
32             mod_child(n, 0, ret.second), ret);
33 }
34
35 node *mk_node(int x){
36     static mt19937 mt(random_device{}());
37     int prio = uniform_int_distribution<int>(
38         1, 1000000000)(mt);
39     return new node {nil, nil, prio, x };
40 }
41
42
43 node *insert(node *root, int x){
44     auto p = split(root, x);
45     return join(join(p.first, mk_node(x)), p.second);
46 }
47
48 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 $1/m$.

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

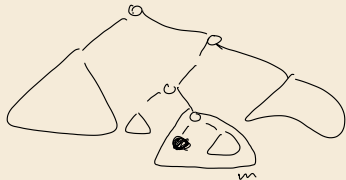
Randomized Binary Search Trees (RBSTs)

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, \dots, 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.



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 $1/m$.

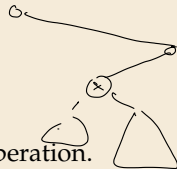
Idea of RBSTs: *enforce* this property *anew* after *each* insertion / deletion!

 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, \dots, 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.

↪ choose one of x 's children y and z

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



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

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

↪ either call insertAtRoot or recurse.

```
1 Node insert(Node root, int x) {
2     n = root.size;
3     r = random.nextInt(n); // uniform int in [0..n)
4     if (r == n) // w/p 1/n, x is new root
5         return insertAtRoot(root, x);
6     if (x < root.key)
7         root.left = insert(root.left, x);
8     else
9         root.right = insert(root.right, x);
10    updateSize(root);
11    return root;
12 }
```

Insert in RBSTs

Proceed top-down.

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

↪ either call insertAtRoot or recurse.

```
1 Node insert(Node root, int x) {
2     n = root.size;
3     r = random.nextInt(n); // uniform int in [0..n)
4     if (r == n) // w/p 1/n, x is new root
5         return insertAtRoot(root, x);
6     if (x < root.key)
7         root.left = insert(root.left, x);
8     else
9         root.right = insert(root.right, x);
10    updateSize(root);
11    return root;
12 }
```

Delete in RBSTs

```
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         root = join(root.left, root.right);
9         updateSize(root);
10        return root;
11    }
12 }
```

update size correct?

```
1 Node join(Node left, Node right) {
2     if (left == null) return right;
3     if (right == null) return left;
4     int sl = left.size, sr = right.size;
5     r = random.nextInt(sl+sr); // uniform [0..sl+sr)
6     int dir = r < sl ? 0 : 1; // 0 w/p sl/(sl+sr)
7     Node root = dir == 0 ? left : right;
8     root.child = (dir == 0) ?
9         join(root.child[1], right) :
10        join(left, root.child[0]);
11    updateSize(root);
12    return root;
13 }
```

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*.

Proof see paper

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.

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 1.25

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 1.25

(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 .

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 1.25

(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.

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 1.25

(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.

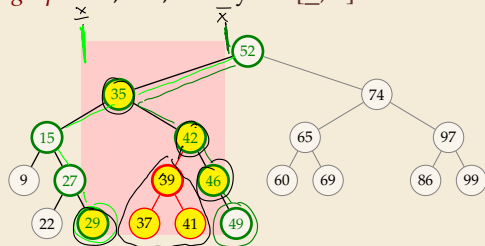
(c) Similar.

1.9 Skiplists

Maybe we don't even want a tree?

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

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

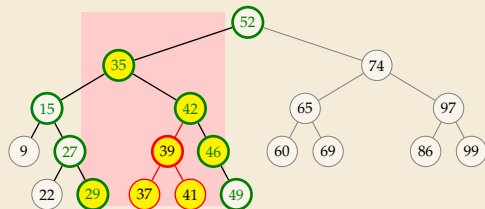


- ▶ **boundary nodes**: nodes in search paths
- ▶ **inside nodes**: nodes between search paths
- ▶ **outside nodes**

Maybe we don't even want a tree?

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

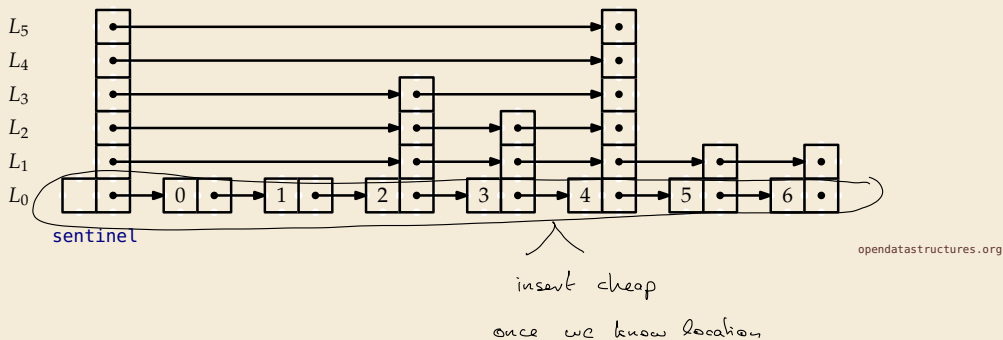
- ▶ BSTs can do that in $O(\log n)$ time
 - ▶ search for \underline{x} and \bar{x}
 - ▶ report nodes on path in range and *inside nodes* (children of path nodes)
- ↪ $O(\log n)$ nodes describe output irrespective of #keys in output



- ▶ **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!

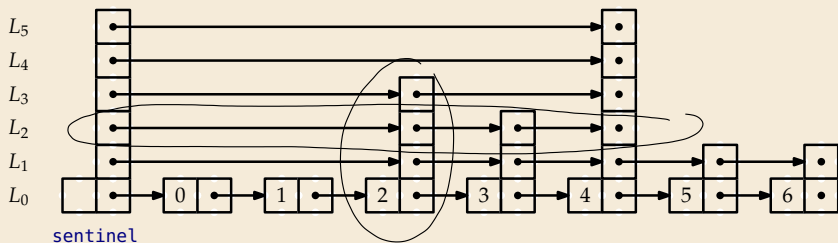
Skiplists

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



Skiplists

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

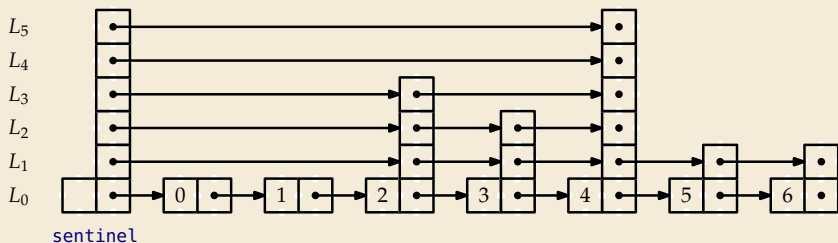


opendatastructures.org

- ▶ conceptually sequence of sorted linked lists L_0, \dots, L_h
- ▶ each L_{i+1} stores a subsequence of L_i

Skiplists

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



opendatastructures.org

- ▶ conceptually sequence of sorted linked lists L_0, \dots, 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 $C \stackrel{\mathcal{D}}{\equiv} B(\frac{1}{2})$
 - ▶ 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** $\text{next}[i]$ for each of its levels i

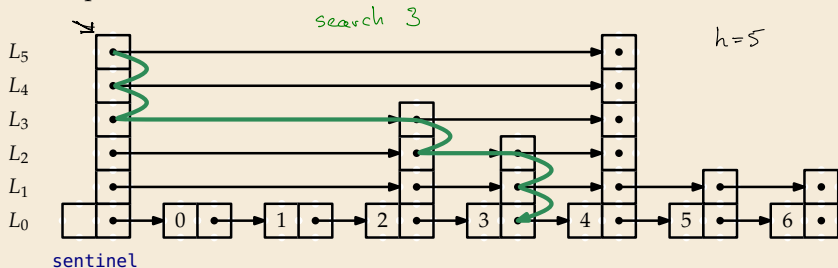
Searching key x

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 $=: T \stackrel{D}{=} \text{Geo}(p)$ with $p = \frac{1}{2}$ (Geometric distribution)

$$\blacktriangleright \mathbb{P}[T = k] = (1 - p)^{k-1} \cdot p \quad k \geq 1$$

$$\blacktriangleright \mathbb{E}[T] = 1/p \quad \mathbb{E}[T] = 2$$

$$\mathbb{E}[T > k] = (1-p)^{-k} \quad 2^{-k}$$

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 $2n$. ◀

Analysis of Skiplist

Recall: #coin tosses until first 1 =: $T \stackrel{D}{=} \text{Geo}(p)$ 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 $\lg n + 2$. ◀

"
highest level $\Rightarrow h+1$ levels

Analysis of Skiplist

Recall: #coin tosses until first 1 =: $T \stackrel{D}{=} \text{Geo}(p)$ 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:

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



Analysis of Skiplist

Recall: #coin tosses until first 1 =: $T \stackrel{D}{=} \text{Geo}(p)$ 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:

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

⋮

Analysis of Skiplist

Recall: #coin tosses until first 1 =: $T \stackrel{D}{=} \text{Geo}(p)$ 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:

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

$$\mathbb{E}[|L_k|] = \frac{1}{2} \mathbb{E}[|L_{k-1}|]$$

$$\mathbb{E}[h] = \sum_{k=1}^{\infty} \mathbb{E}[I_k]$$



Analysis of Skiplist

Recall: #coin tosses until first 1 =: $T \stackrel{D}{=} \text{Geo}(p)$ 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:

$$h = \sum_{k=1}^{\infty} \overbrace{[L_k \text{ is not empty}]}^{I_k} \leq \sum_{k=1}^{\infty} |L_k| \leq 1$$
$$\mathbb{E}[h] = \sum_{k=1}^{\infty} \mathbb{E}[I_k] = \sum_{k=1}^{\lfloor \lg n \rfloor} 1 + \sum_{k=\lfloor \lg n \rfloor+1}^{\infty} \left(\frac{n}{2^k} \right) \leq 1$$



Analysis of Skiplist

Recall: #coin tosses until first 1 =: $T \stackrel{D}{=} \text{Geo}(p)$ 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:

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

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



Analysis of Skiplist

Recall: #coin tosses until first 1 =: $T \stackrel{D}{=} \text{Geo}(p)$ 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:

$$h = \sum_{k=1}^{\infty} \overbrace{[L_k \text{ is not empty}]}^{I_k} \leq \sum_{k=1}^{\infty} |L_k|$$
$$\mathbb{E}[h] = \sum_{k=1}^{\infty} \mathbb{E}[I_k] = \sum_{k=1}^{\lfloor \lg n \rfloor} 1 + \sum_{k=\lfloor \lg n \rfloor + 1}^{\infty} \frac{n}{2^k} \leq \sum_{k=1}^{\lfloor \lg n \rfloor} 1 + \sum_{k=0}^{\infty} \frac{1}{2^k} \leq \lg n + 2. \quad \blacksquare$$

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 $2 \lg n + O(1)$. ◀

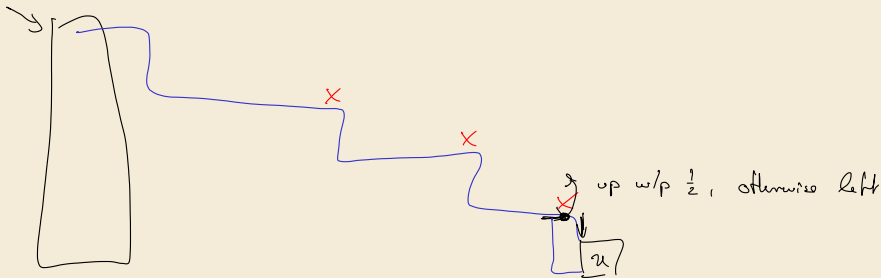
Analysis of Search

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*.



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.



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.



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.

$$S = \sum S_k + h$$

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

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 ! $\rightsquigarrow \mathbb{E}[S_k] \leq \mathbb{E}[|L_k|] \leq n/2^k$.

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 $S = h + S_0 + S_1 + S_2 + \dots$

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

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 $S = h + S_0 + S_1 + S_2 + \dots$

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

Analysis of Search

$$2 \lg n = 1.39 \lg n$$

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: $\mathbb{E}[S_k] \leq 1$.

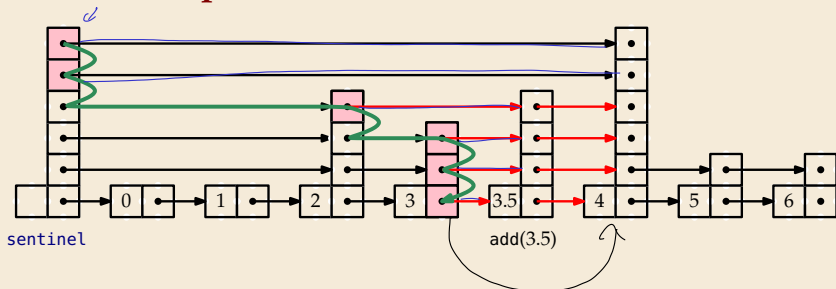
“ \leq ” since we may run out of nodes in L_k ! $\rightsquigarrow \mathbb{E}[S_k] \leq \mathbb{E}[|L_k|] \leq n/2^k$.

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

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

1.10 Operations in Skiplists

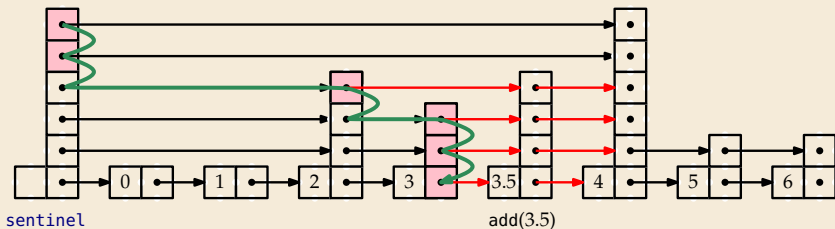
Insert in Skiplists



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.

Insert in Skiplists



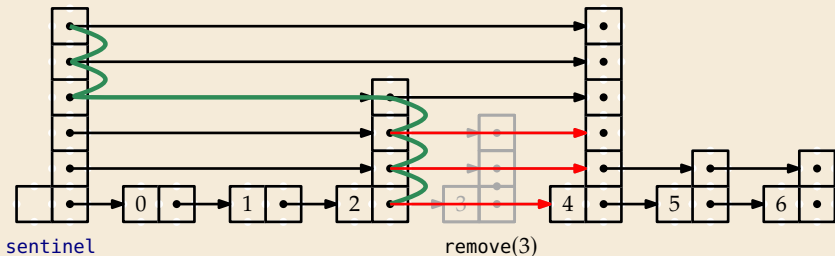
opendatastructures.org

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

$x - \epsilon$

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

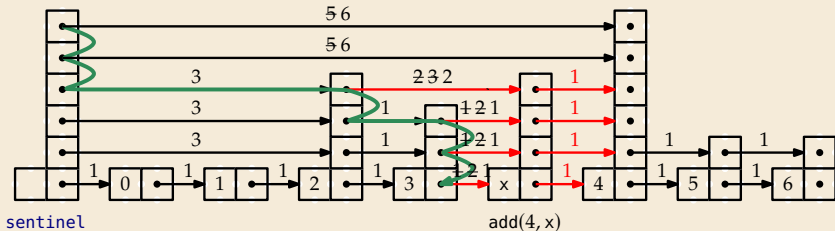
Rank-based operations

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*)



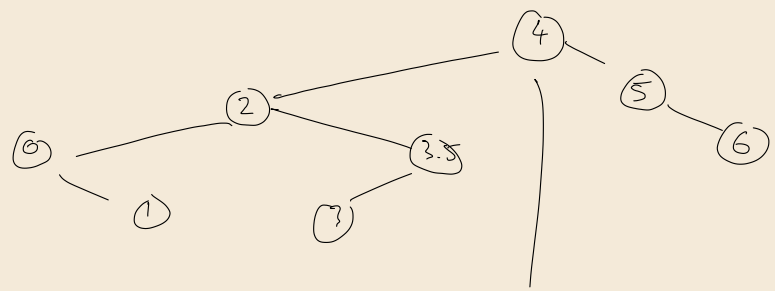
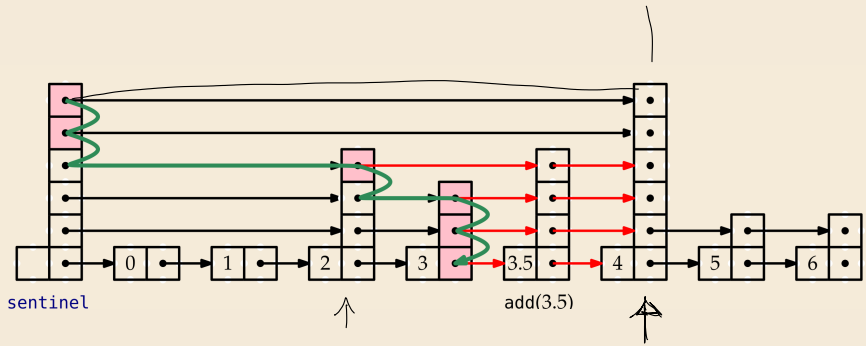
opendatastructures.org

Skiplists – Discussion

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

Skiplists – Discussion

- 👍 Extremely simple use of randomness
- 👍 Range queries simply follow linked list
- 👎 Non-constant size nodes (towers), random space usage
- 👎 Worse search costs: $2 \lg(n)$ vs $2 \ln(n) \approx 1.39 \lg n$



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!

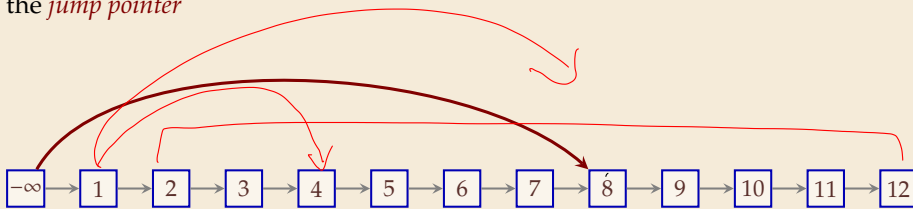
↪ the *jump pointer*

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.



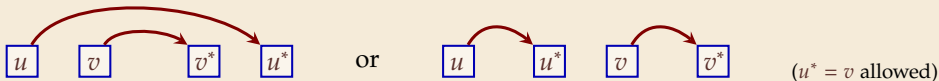
Brönnimann, Cazals, Durand: *Randomized jumplists: A jump-and-walk dictionary data structure*, STACS 2003



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

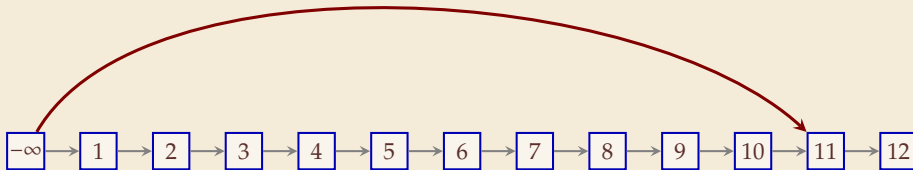
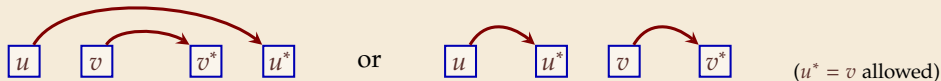
Jump Pointers

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:



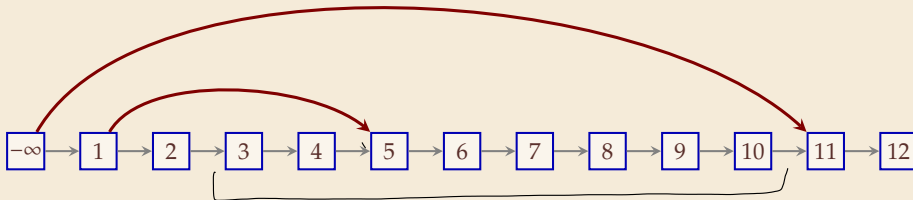
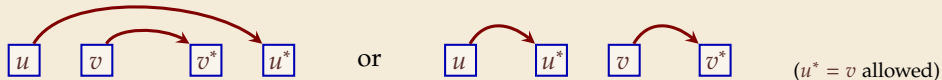
Jump Pointers

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:



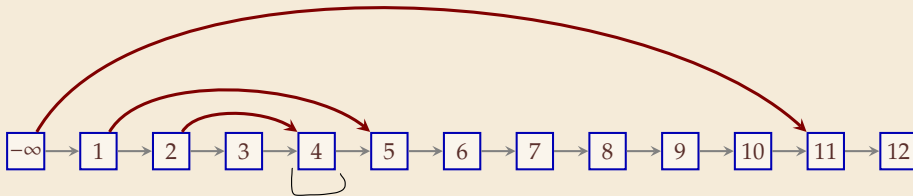
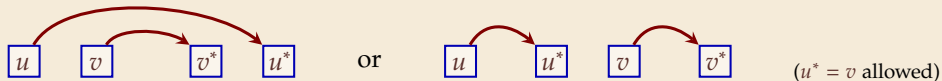
Jump Pointers

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:



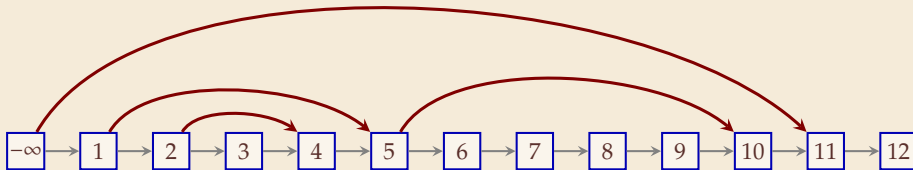
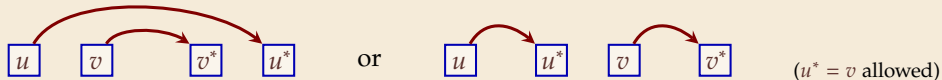
Jump Pointers

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:



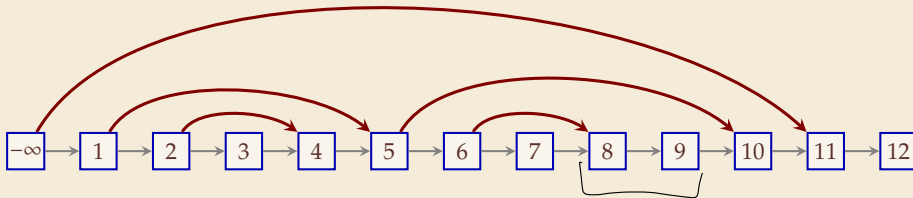
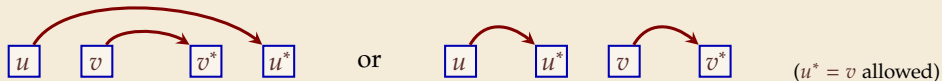
Jump Pointers

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:



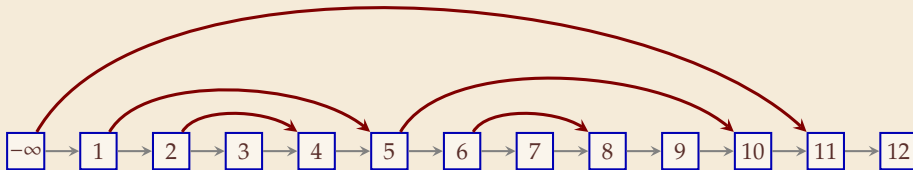
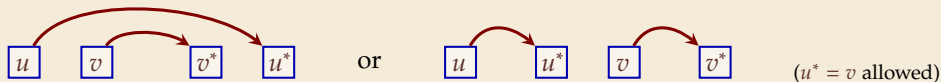
Jump Pointers

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:



Jump Pointers

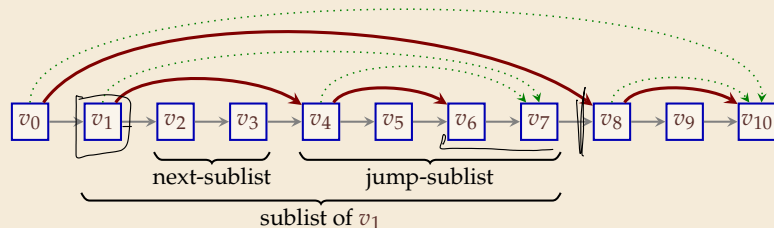
1. *Non-degeneracy*: Any node may be the target of at **most one** jump pointer, and jump pointers **never** point to the direct **successor**.
2. *Well-nestedness*: Jump pointers may **not cross**. For $v \neq u$ nodes with $v.key < u.key$, and v^* resp. u^* the nodes their jump pointers point to, order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



\rightsquigarrow By non-degeneracy, some nodes can't have jump pointer \rightsquigarrow *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 $v_2.key, \dots, v_7.key$.
(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

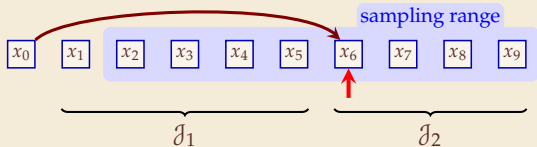
We can now define the equivalent of the random BST model: Random Jumplists

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

Random Jumlists

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

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



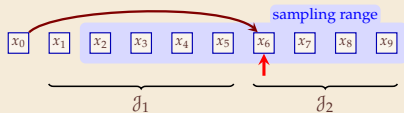
$$m = \# \text{ nodes in sublist}$$
$$n = \# \text{ keys} = m - 1$$

$$\mathbb{P}[\mathcal{J}] = \begin{cases} 1, & m \leq w; \\ \frac{1}{m-2} \cdot \mathbb{P}[\mathcal{J}_1] \mathbb{P}[\mathcal{J}_2], & m > w, \end{cases} \quad (w = 2 \text{ for simplest version})$$

1.12 Operations in Jumplist

Expected Search Costs

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

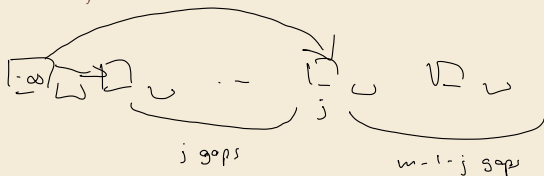


$$D_m = \begin{cases} 0 & m = 1 \\ 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} & m = 2 \\ 0 \cdot \frac{1}{m} + \frac{1}{m-2} \sum_{j=1}^{m-2} \left(\frac{j}{m} (1 + D_j) + \frac{m-1-j}{m} (1 + D_{m-1-j}) \right) & m \geq 3 \end{cases}$$

prob jump pointer to j

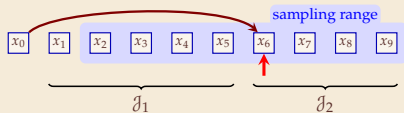
$m = 1$

$m = 2$



Expected Search Costs

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



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

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

$$D_m \sim 2 \ln m$$

Expected Search Costs [2]

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

Expected Search Costs [2]

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

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

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

Expected Search Costs [2]

$$m \cdot \underline{D}_m =: P_m = m - 1 + \begin{cases} 0 & m \leq 2 \\ \frac{2}{m-2} \sum_{j=1}^{m-2} P_j & m \geq 3 \end{cases}$$

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

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

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

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

Expected Search Costs [2]

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

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

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

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

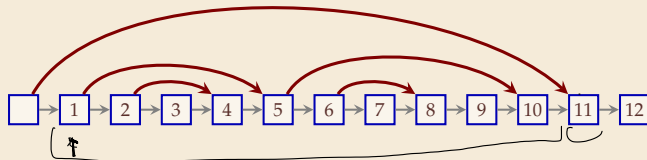
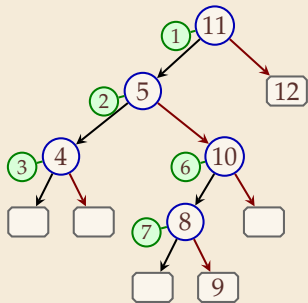
Indeed, here always $D_m \leq C_m$ for $m \geq 0$, so at least get the upper bound for free

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

Searching in Jumplists costs same as in random BSTs! 😊

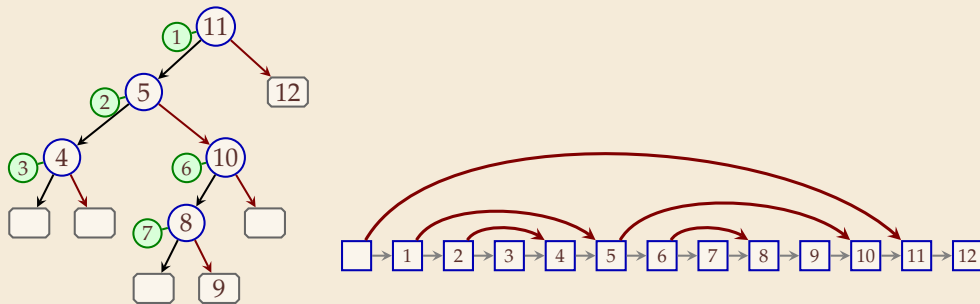
Dangling-Min BSTs

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



Dangling-Min BSTs

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



$$\text{MINBST} \left(\boxed{} \rightarrow x_1 \dots x_n \right) = \boxed{x_1, \dots, x_n} \quad \underbrace{(m \leq w)}_{\text{2}}^{\text{11}}$$

$$\text{MINBST} \left(\boxed{} \rightarrow \underbrace{x_1 \dots}_{\mathcal{J}_1} \rightarrow \underbrace{x_j \dots}_{\mathcal{J}_2} \right) = \begin{array}{c} \text{MINBST}(\mathcal{J}_1) \quad \text{MINBST}(\mathcal{J}_2) \end{array} \quad (m > w)$$

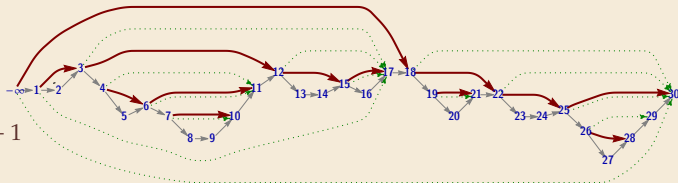
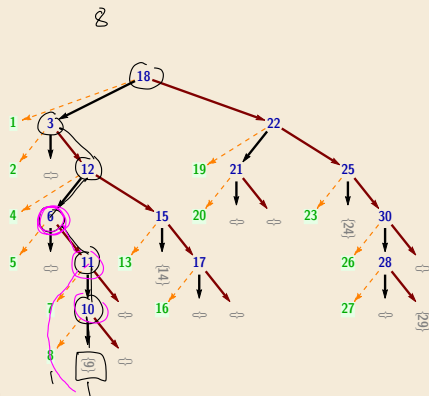
BST-View Inspired Search

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
12 // Linear search from lastJumpedTo
13 while head.next.key < x
14     head := head.next; rank := rank + 1
15 end while
16 return (head, rank)
  
```



So far: Random Jumplists

How do we get to *Randomized Jumplists*?

So far: Random Jumlists

How do we get to *Randomized Jumlists*?

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.

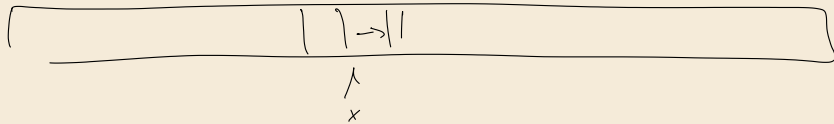
$$\text{REB}\left(\boxed{v_0 \dots v_n}\right) = \boxed{v_0 \dots v_n} \quad (m \leq \overset{2}{w})$$
$$\text{REB}\left(\boxed{v_0 \dots v_n}\right) = \boxed{v_0} \rightarrow \text{REB}\left(\boxed{}\right) \rightarrow \text{REB}\left(\boxed{v_j }\right) \quad (m > w)$$

Draw $j \stackrel{\mathcal{D}}{=} \text{Uniform}\{v_2, \dots, v_n\}$

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 .



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).

$$\text{RESTINS}\left(\begin{array}{|c|} \hline \color{red}{\blacksquare} \\ \hline \end{array}\right) = \text{REB}\left(\begin{array}{|c|} \hline \color{red}{\blacksquare} \\ \hline \end{array}\right) \quad (m \leq w) \quad p = \frac{1}{n}.$$

$$\text{RESTINS}(\mathcal{J}) = p \cdot \begin{array}{|c|} \hline v_0 \\ \hline \end{array} \xrightarrow{\text{REB}} \left(\begin{array}{|c|} \hline \\ \hline \end{array}\right) \xrightarrow{\text{REB}} \begin{array}{|c|c|} \hline v_j & v_n \\ \hline \end{array}$$

$$+ (1-p) \cdot \left\{ \begin{array}{l} \begin{array}{|c|} \hline x \\ \hline \end{array} \xrightarrow{\text{RESTINS}} \left(\begin{array}{|c|} \hline \color{red}{\blacksquare} \\ \hline \end{array}\right) \xrightarrow{\text{REB}} \begin{array}{|c|c|} \hline v_j & \\ \hline \end{array} \quad \mathcal{J} = \begin{array}{|c|c|c|c|} \hline \color{red}{x} & v_0 & & v_j \\ \hline \end{array} \\ \\ \begin{array}{|c|} \hline v_0 \\ \hline \end{array} \xrightarrow{\text{RESTINS}} \left(\begin{array}{|c|} \hline \color{red}{\blacksquare} \\ \hline \end{array}\right) \xrightarrow{\text{REB}} \begin{array}{|c|c|} \hline v_j & \\ \hline \end{array} \quad \mathcal{J} = \begin{array}{|c|c|c|c|} \hline v_0 & & \color{red}{\blacksquare} & v_j \\ \hline \end{array} \\ \\ \begin{array}{|c|} \hline v_0 \\ \hline \end{array} \xrightarrow{\text{REB}} \left(\begin{array}{|c|} \hline \\ \hline \end{array}\right) \xrightarrow{\text{RESTINS}} \begin{array}{|c|c|} \hline v_j & \color{red}{\blacksquare} \\ \hline \end{array} \quad \mathcal{J} = \begin{array}{|c|c|c|c|} \hline v_0 & & & v_j \\ \hline \end{array} \end{array} \right.$$

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

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 ...)

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 ...)
- ↪ 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

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | 4 | 2 | 9 | 1 | 3 | 8 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|

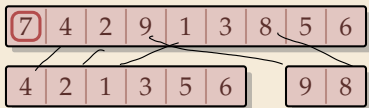
Quicksort & Binary Search Trees

Quicksort

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | 4 | 2 | 9 | 1 | 3 | 8 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|

Quicksort & Binary Search Trees

Quicksort



Quicksort & Binary Search Trees

Quicksort

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | 4 | 2 | 9 | 1 | 3 | 8 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 1 | 3 | 5 | 6 | 7 | 9 | 8 |
|---|---|---|---|---|---|---|---|---|

Quicksort & Binary Search Trees

Quicksort

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | 4 | 2 | 9 | 1 | 3 | 8 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 1 | 3 | 5 | 6 | 7 | 9 | 8 |
|---|---|---|---|---|---|---|---|---|

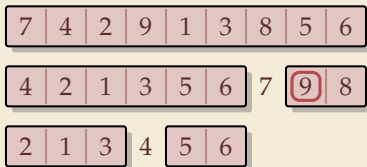
Quicksort & Binary Search Trees

Quicksort



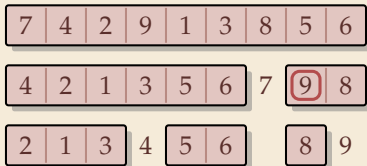
Quicksort & Binary Search Trees

Quicksort



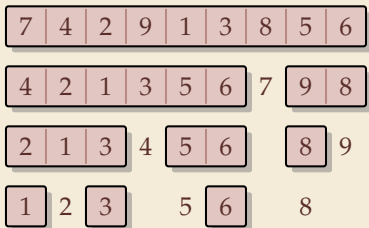
Quicksort & Binary Search Trees

Quicksort



Quicksort & Binary Search Trees

Quicksort



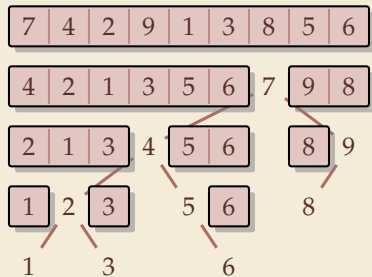
Quicksort & Binary Search Trees

Quicksort



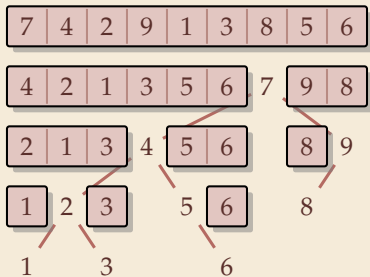
Quicksort & Binary Search Trees

Quicksort



Quicksort & Binary Search Trees

Quicksort

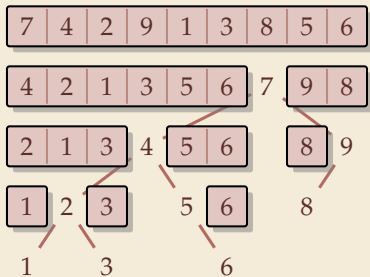


Binary Search Tree (BST)

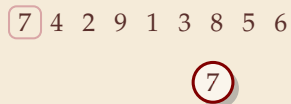
7 4 2 9 1 3 8 5 6

Quicksort & Binary Search Trees

Quicksort

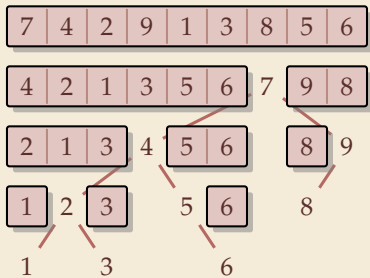


Binary Search Tree (BST)

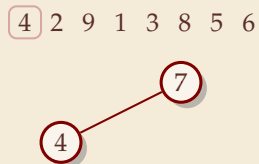


Quicksort & Binary Search Trees

Quicksort

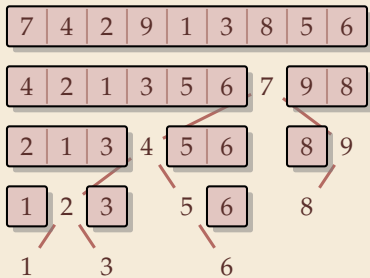


Binary Search Tree (BST)

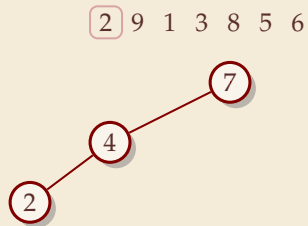


Quicksort & Binary Search Trees

Quicksort

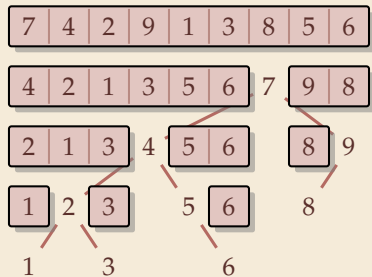


Binary Search Tree (BST)

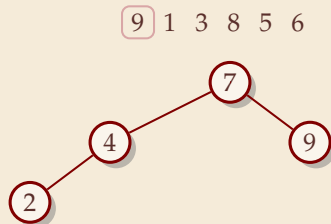


Quicksort & Binary Search Trees

Quicksort

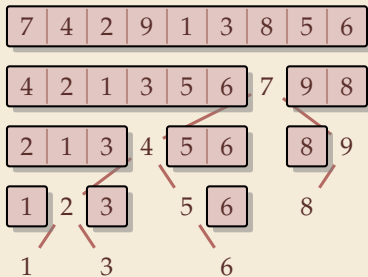


Binary Search Tree (BST)

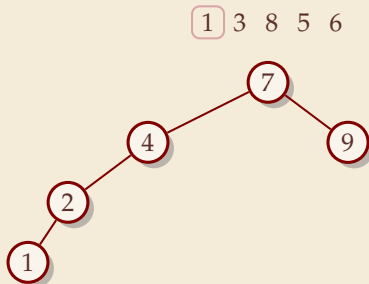


Quicksort & Binary Search Trees

Quicksort

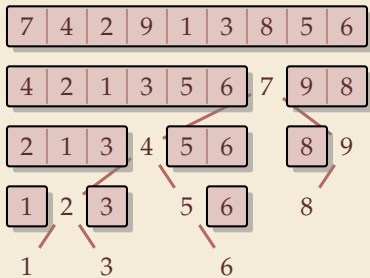


Binary Search Tree (BST)

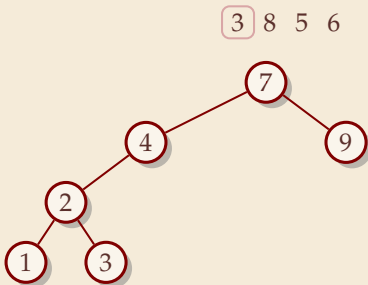


Quicksort & Binary Search Trees

Quicksort

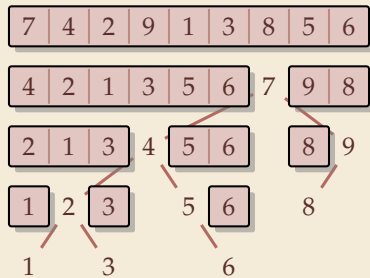


Binary Search Tree (BST)

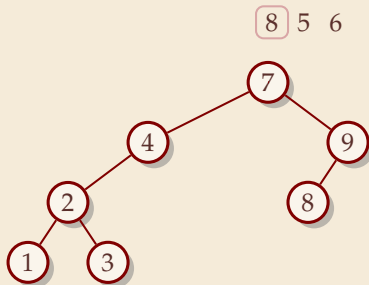


Quicksort & Binary Search Trees

Quicksort

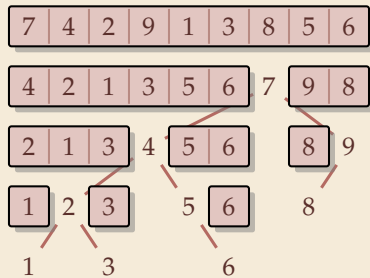


Binary Search Tree (BST)

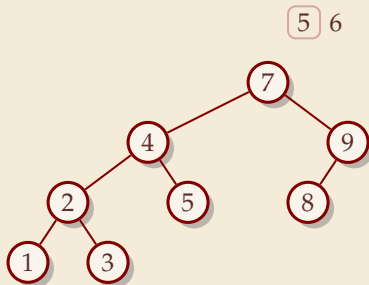


Quicksort & Binary Search Trees

Quicksort

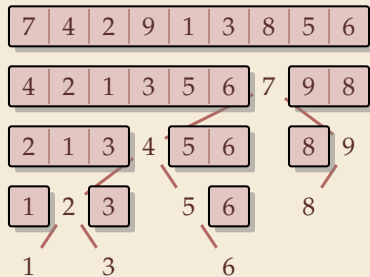


Binary Search Tree (BST)

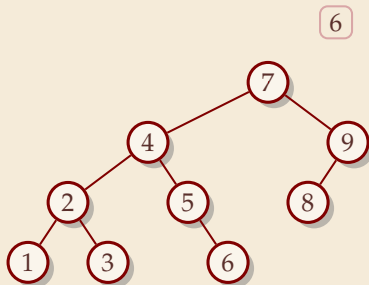


Quicksort & Binary Search Trees

Quicksort



Binary Search Tree (BST)



Fringe-Balanced BSTs

k-Fringe-Balanced Search Trees:

- ▶ Leaves *buffer* $k = 2t + 1$ elements.

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*.

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

($t = 2$)

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 1 | 3 | 3 | 5 | 4 | 4 | 3 | 5 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|

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

($t = 2$)

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 1 | 3 | 3 | 5 | 4 | 4 | 3 | 5 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|

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

($t = 2$)

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 1 | 3 | 3 | 5 | 4 | 4 | 3 | 5 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|

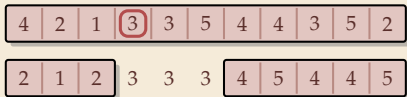
Fringe-Balanced BSTs

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

($t = 2$)



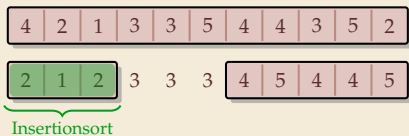
Fringe-Balanced BSTs

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

($t = 2$)



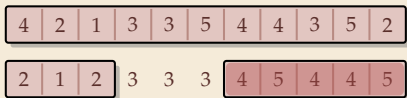
Fringe-Balanced BSTs

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

($t = 2$)



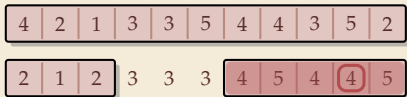
Fringe-Balanced BSTs

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

($t = 2$)



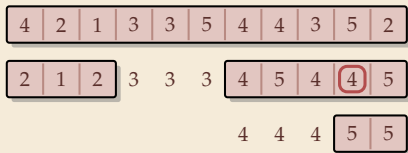
Fringe-Balanced BSTs

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

($t = 2$)



Fringe-Balanced BSTs

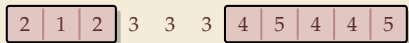
k-Fringe-Balanced Search Trees:

- ▶ 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



Insertionsort

Fringe-Balanced BSTs

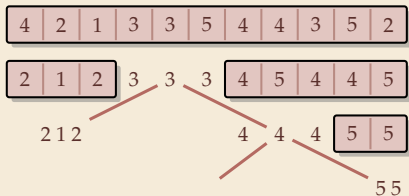
k-Fringe-Balanced Search Trees:

- ▶ 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

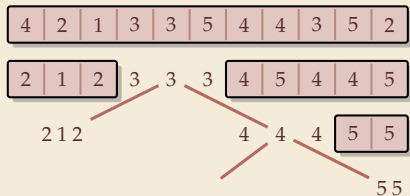


Fringe-Balanced BSTs

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

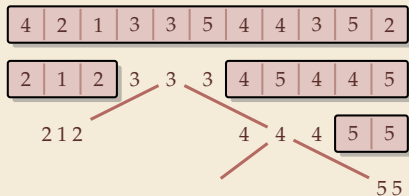
4 2 1 3 3 5 4 4 3 5 2

Fringe-Balanced BSTs

k-Fringe-Balanced Search Trees:

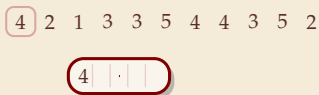
- ▶ Leaves *buffer* $k = 2t + 1$ elements.
- ▶ If buffer is full, leaf is **split** \rightsquigarrow new internal node with *chosen pivot*.

Median-of-5 Quicksort



($t = 2$)

5-Fringe-Balanced Tree



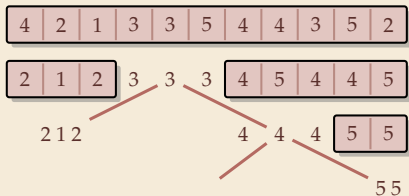
Fringe-Balanced BSTs

k-Fringe-Balanced Search Trees:

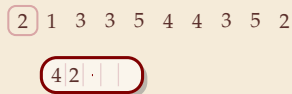
- ▶ Leaves *buffer* $k = 2t + 1$ elements.
- ▶ If buffer is full, leaf is **split** \rightsquigarrow new internal node with *chosen pivot*.

Median-of-5 Quicksort

($t = 2$)



5-Fringe-Balanced Tree



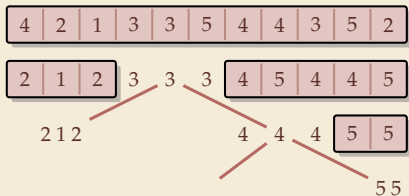
Fringe-Balanced BSTs

k-Fringe-Balanced Search Trees:

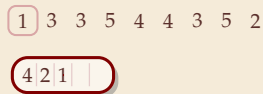
- ▶ Leaves *buffer* $k = 2t + 1$ elements.
- ▶ If buffer is full, leaf is **split** \rightsquigarrow new internal node with *chosen pivot*.

Median-of-5 Quicksort

($t = 2$)



5-Fringe-Balanced Tree



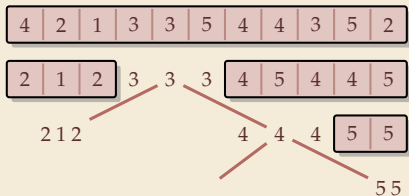
Fringe-Balanced BSTs

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



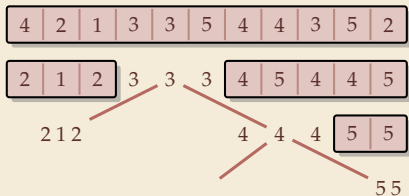
Fringe-Balanced BSTs

k-Fringe-Balanced Search Trees:

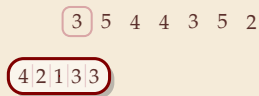
- ▶ Leaves *buffer* $k = 2t + 1$ elements.
- ▶ If buffer is full, leaf is **split** \rightsquigarrow new internal node with *chosen pivot*.

Median-of-5 Quicksort

($t = 2$)



5-Fringe-Balanced Tree



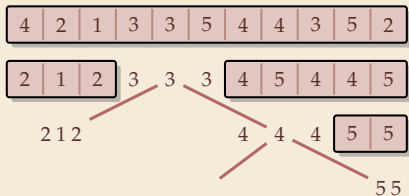
Fringe-Balanced BSTs

k-Fringe-Balanced Search Trees:

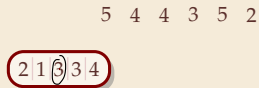
- ▶ Leaves *buffer* $k = 2t + 1$ elements.
- ▶ If buffer is full, leaf is **split** \rightsquigarrow new internal node with *chosen pivot*.

Median-of-5 Quicksort

($t = 2$)



5-Fringe-Balanced Tree



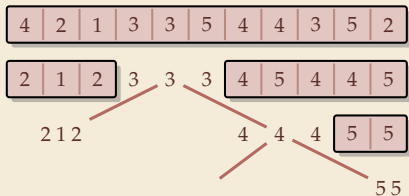
Fringe-Balanced BSTs

k-Fringe-Balanced Search Trees:

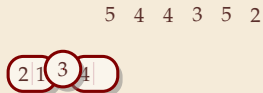
- ▶ Leaves *buffer* $k = 2t + 1$ elements.
- ▶ If buffer is full, leaf is **split** \rightsquigarrow new internal node with *chosen pivot*.

Median-of-5 Quicksort

($t = 2$)



5-Fringe-Balanced Tree

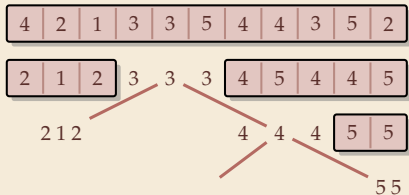


Fringe-Balanced BSTs

k-Fringe-Balanced Search Trees:

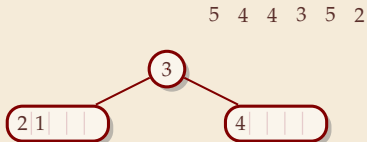
- ▶ Leaves *buffer* $k = 2t + 1$ elements.
- ▶ If buffer is full, leaf is **split** \rightsquigarrow new internal node with *chosen pivot*.

Median-of-5 Quicksort



($t = 2$)

5-Fringe-Balanced Tree

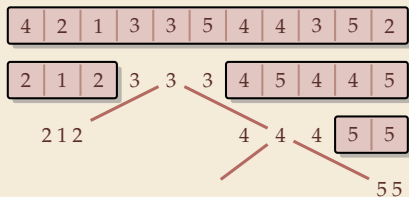


Fringe-Balanced BSTs

k-Fringe-Balanced Search Trees:

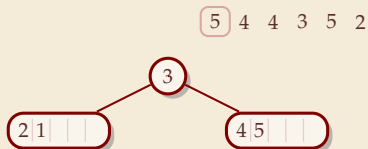
- ▶ Leaves *buffer* $k = 2t + 1$ elements.
- ▶ If buffer is full, leaf is **split** \rightsquigarrow new internal node with *chosen pivot*.

Median-of-5 Quicksort



($t = 2$)

5-Fringe-Balanced Tree

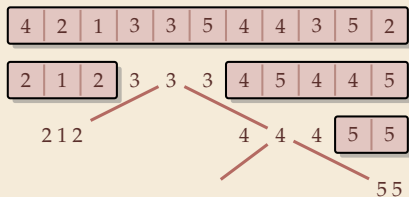


Fringe-Balanced BSTs

k-Fringe-Balanced Search Trees:

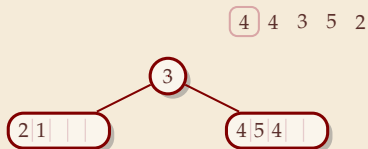
- ▶ Leaves *buffer* $k = 2t + 1$ elements.
- ▶ If buffer is full, leaf is **split** \rightsquigarrow new internal node with *chosen pivot*.

Median-of-5 Quicksort



($t = 2$)

5-Fringe-Balanced Tree

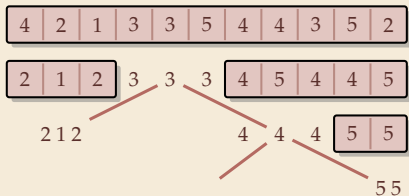


Fringe-Balanced BSTs

k-Fringe-Balanced Search Trees:

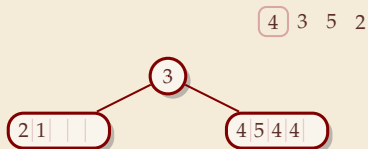
- ▶ Leaves *buffer* $k = 2t + 1$ elements.
- ▶ If buffer is full, leaf is **split** \rightsquigarrow new internal node with *chosen pivot*.

Median-of-5 Quicksort



($t = 2$)

5-Fringe-Balanced Tree

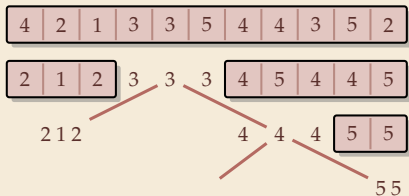


Fringe-Balanced BSTs

k-Fringe-Balanced Search Trees:

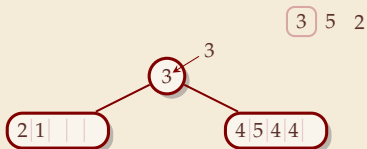
- ▶ Leaves *buffer* $k = 2t + 1$ elements.
- ▶ If buffer is full, leaf is **split** \rightsquigarrow new internal node with *chosen pivot*.

Median-of-5 Quicksort



($t = 2$)

5-Fringe-Balanced Tree



Fringe-Balanced BSTs

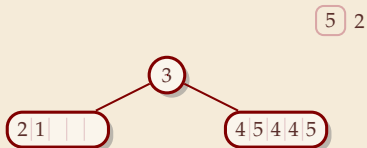
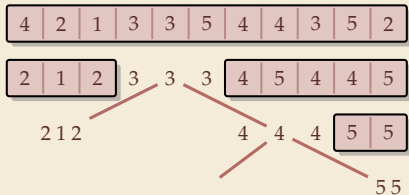
k-Fringe-Balanced Search Trees:

- ▶ 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

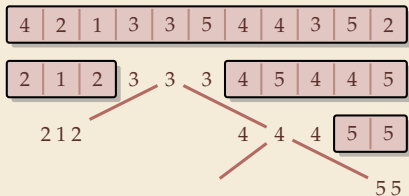


Fringe-Balanced BSTs

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



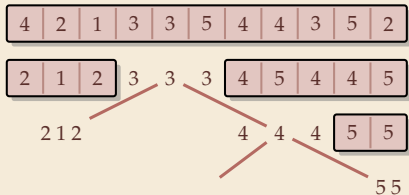
2

Fringe-Balanced BSTs

k-Fringe-Balanced Search Trees:

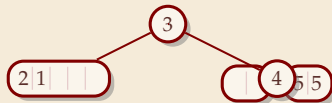
- ▶ Leaves *buffer* $k = 2t + 1$ elements.
- ▶ If buffer is full, leaf is **split** \rightsquigarrow new internal node with *chosen pivot*.

Median-of-5 Quicksort



($t = 2$)

5-Fringe-Balanced Tree



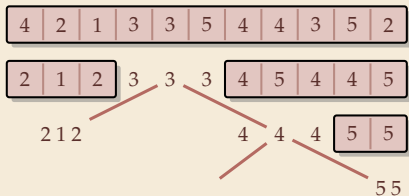
2

Fringe-Balanced BSTs

k-Fringe-Balanced Search Trees:

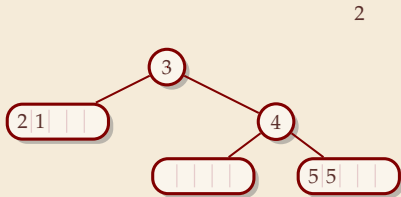
- ▶ Leaves *buffer* $k = 2t + 1$ elements.
- ▶ If buffer is full, leaf is **split** \rightsquigarrow new internal node with *chosen pivot*.

Median-of-5 Quicksort



($t = 2$)

5-Fringe-Balanced Tree

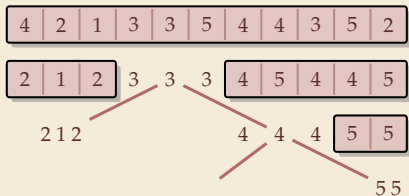


Fringe-Balanced BSTs

k-Fringe-Balanced Search Trees:

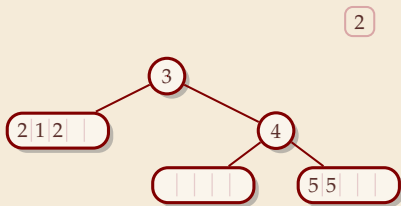
- ▶ Leaves *buffer* $k = 2t + 1$ elements.
- ▶ If buffer is full, leaf is **split** \rightsquigarrow new internal node with *chosen pivot*.

Median-of-5 Quicksort



($t = 2$)

5-Fringe-Balanced Tree

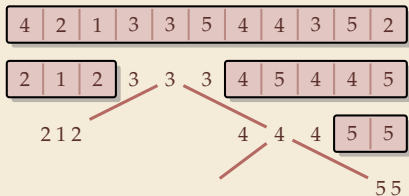


Fringe-Balanced BSTs

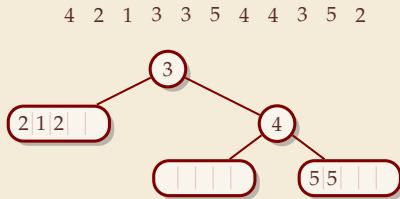
k-Fringe-Balanced Search Trees:

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

Median-of-5 Quicksort



5-Fringe-Balanced Tree



\rightsquigarrow Correspondence extends to

- ▶ Larger Insertionsort Cutoff
- ▶ (Handling equal keys)

Fringe-Balanced BSTs

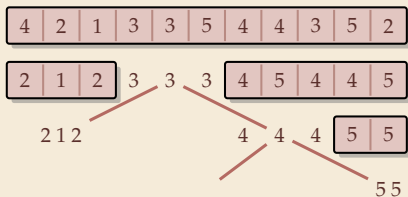
k-Fringe-Balanced Search Trees:

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

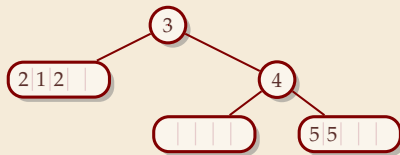
Median-of-5 Quicksort

($t = 2$)

5-Fringe-Balanced Tree



4 2 1 3 3 5 4 4 3 5 2



\rightsquigarrow Correspondence extends to

- ▶ Larger Insertionsort Cutoff
- ▶ (Handling equal keys)

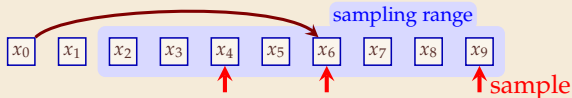
\rightsquigarrow Expected depth of typical node: $\frac{1}{H_{k+1} - H_{t+1}} \ln n$ instead of $\underline{2 \ln n}$

$$k = 5 \rightsquigarrow \overset{1.39}{1.12} \lg n$$

Randomized Fringe-Balanced Jumplists

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

$\rightsquigarrow J \stackrel{D}{=} \text{BetaBin}(m - 2 - k; t + 1, t + 1) + t + 2$ 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:

- The expected number of key comparisons in a **spine search** is asymptotic to $1/H(k) \cdot \ln n$, as $n \rightarrow \infty$, when each position is equally likely to be requested.
- The expected number of rebalanced elements in the **cleanup after insertion** is asymptotic to $k/H(k) \cdot \ln n$, as $n \rightarrow \infty$, when each of the $n + 1$ possible gaps is equally likely.
- The expected number of rebalanced elements in the **cleanup after deletion** is asymptotic to $k/H(k) \cdot \ln n$, as $n \rightarrow \infty$, when each key is equally likely to be deleted.
- 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 $n \rightarrow \infty$.

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

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


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.