COMP 761: Lecture 29 – Binary Search Trees II

David Rolnick

November 11, 2020

Problem

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

(Please don't post your ideas in the chat just yet, we'll discuss the problem soon in class.)

Course Announcements

Course Announcements

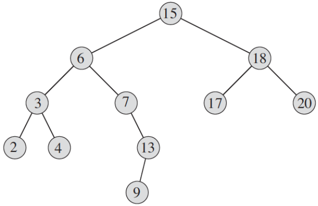
- Problem set 5 is out!

Course Announcements

- Problem set 5 is out!
- Office hours: Vincent Thu at 10:30 am, David Fri at 10 am

Review: Binary search trees

- A *binary search tree* is a binary tree, each node storing a *key*.



- We require that for every node v:
  - The left subtree has all nodes less than or equal to v.
  - The right subtree has all nodes greater than or equal to v.

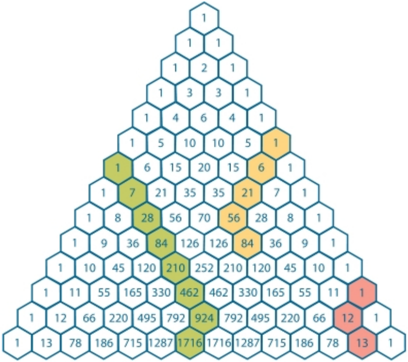
Review: Expected height

- We have a lot of algorithms running in $O(h)$.
- Maximum height with n keys: $h = n - 1$.
- Minimum height: $h = O(\log n)$.
- Let's consider a *typical* binary search tree.
- Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

Review: Hockey stick identity

- Hockey stick identity in our proof:

$$\sum_{i=0}^{n-1} \binom{i+k}{k} = \binom{n+k}{k+1}.$$



Review: Jensen's inequality

- We will also use another form of Jensen's inequality - if f is convex, then:

$$\mathbb{E}[f(x)] \geq f(\mathbb{E}[x]).$$

- This is essentially the same as the weighted form of Jensen's inequality we have already seen:

$$\sum_{i=1}^{n} p_i f(x_i) \geq f\left(\sum_{i=1}^{n} p_i x_i\right)$$

if p_i are nonnegative with $\sum_{i=1}^{n} p_i = 1$.

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- Let X_n be the height of the tree, so we are looking for $\mathbb{E}[X_n]$.

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- Let X_n be the height of the tree, so we are looking for $\mathbb{E}[X_n]$.
- What is the root of the tree?

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- Let X_n be the height of the tree, so we are looking for $\mathbb{E}[X_n]$.
- The root of the tree is whatever i we insert first, and it doesn't change by inserting new keys.

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- Let X_n be the height of the tree, so we are looking for $\mathbb{E}[X_n]$.
- The root of the tree is whatever i we insert first, and it doesn't change by inserting new keys.
- How can we use this?

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- Let X_n be the height of the tree, so we are looking for $\mathbb{E}[X_n]$.
- The root of the tree is whatever i we insert first, and it doesn't change by inserting new keys.
- Induction!

$$X_n = 1 + \max(X_{i-1}, X_{n-i}).$$

Note that i is itself a random variable.

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- Let X_n be the height of the tree, so we are looking for $\mathbb{E}[X_n]$.
- The root of the tree is whatever i we insert first, and it doesn't change by inserting new keys.
- Induction!

$$X_n = 1 + \max(X_{i-1}, X_{n-i}).$$

  Note that i is itself a random variable.
- It will be useful to define $Y_n = 2^{X_n}$:

$$Y_n = 2\max(Y_{i-1}, Y_{n-i}) \leq 2(Y_{i-1} + Y_{n-i})$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- Let X_n be the height of the tree, so we are looking for $\mathbb{E}[X_n]$.
- The root of the tree is whatever i we insert first, and it doesn't change by inserting new keys.
- Induction!

$$X_n = 1 + \max(X_{i-1}, X_{n-i}).$$

  Note that i is itself a random variable.
- It will be useful to define $Y_n = 2^{X_n}$:

$$Y_n = 2\max(Y_{i-1}, Y_{n-i}) \leq 2(Y_{i-1} + Y_{n-i})$$

- Since each i is equally likely:

$$\mathbb{E}[Y_n] \leq \sum_{i=1}^{n} \frac{1}{n}\mathbb{E}[2(Y_{i-1} + Y_{n-i})]$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- Let X_n be the height of the tree, so we are looking for $\mathbb{E}[X_n]$.
- The root of the tree is whatever i we insert first, and it doesn't change by inserting new keys.
- Induction!

$$X_n = 1 + \max(X_{i-1}, X_{n-i}).$$

Note that i is itself a random variable.

- It will be useful to define $Y_n = 2^{X_n}$:

$$Y_n = 2\max(Y_{i-1}, Y_{n-i}) \leq 2(Y_{i-1} + Y_{n-i})$$

- Since each i is equally likely:

$$\mathbb{E}[Y_n] \leq \sum_{i=1}^{n} \frac{1}{n}\mathbb{E}[2(Y_{i-1} + Y_{n-i})] = \frac{2}{n}\sum_{i=1}^{n}(\mathbb{E}[Y_{i-1}] + \mathbb{E}[Y_{n-i}])$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- Let X_n be the height of the tree, so we are looking for $\mathbb{E}[X_n]$.
- The root of the tree is whatever i we insert first, and it doesn't change by inserting new keys.
- Induction!

$$X_n = 1 + \max(X_{i-1}, X_{n-i}).$$

  Note that i is itself a random variable.
- It will be useful to define $Y_n = 2^{X_n}$:

$$Y_n = 2\max(Y_{i-1}, Y_{n-i}) \leq 2(Y_{i-1} + Y_{n-i})$$

- Since each i is equally likely:

$$\mathbb{E}[Y_n] \leq \sum_{i=1}^{n} \frac{1}{n}\mathbb{E}[2(Y_{i-1} + Y_{n-i})] = \frac{2}{n}\sum_{i=1}^{n}(\mathbb{E}[Y_{i-1}] + \mathbb{E}[Y_{n-i}])$$

$$= \frac{2}{n}\sum_{i=0}^{n-1}(\mathbb{E}[Y_i] + \mathbb{E}[Y_{n-1-i}])$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- Let X_n be the height of the tree, so we are looking for $\mathbb{E}[X_n]$.
- The root of the tree is whatever i we insert first, and it doesn't change by inserting new keys.
- Induction!

$$X_n = 1 + \max(X_{i-1}, X_{n-i}).$$

Note that i is itself a random variable.

- It will be useful to define $Y_n = 2^{X_n}$:

$$Y_n = 2\max(Y_{i-1}, Y_{n-i}) \leq 2(Y_{i-1} + Y_{n-i})$$

- Since each i is equally likely:

$$\mathbb{E}[Y_n] \leq \sum_{i=1}^{n} \frac{1}{n}\mathbb{E}[2(Y_{i-1} + Y_{n-i})] = \frac{2}{n}\sum_{i=1}^{n}(\mathbb{E}[Y_{i-1}] + \mathbb{E}[Y_{n-i}])$$

$$= \frac{2}{n}\sum_{i=0}^{n-1}(\mathbb{E}[Y_i] + \mathbb{E}[Y_{n-1-i}]) = \frac{4}{n}\sum_{i=0}^{n-1}\mathbb{E}[Y_i].$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have a recurrence:

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i].$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have a recurrence:

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i].$$

- Let's try to prove by induction:

$$\mathbb{E}[Y_n] \leq \frac{1}{4} \binom{n+3}{3}.$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have a recurrence:

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i].$$

- Let's try to prove by induction:

$$\mathbb{E}[Y_n] \leq \frac{1}{4} \binom{n+3}{3}.$$

- Base case: $Y_1 = 1 \leq \frac{1}{4}\binom{4}{3}$.

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have a recurrence:

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i].$$

- Let's try to prove by induction:

$$\mathbb{E}[Y_n] \leq \frac{1}{4} \binom{n+3}{3}.$$

- Base case: $Y_1 = 1 \leq \frac{1}{4}\binom{4}{3}$.
- Assuming it holds for all $i < n$, the recurrence is just

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i] \leq \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4} \binom{i+3}{3} = \frac{1}{n} \sum_{i=0}^{n-1} \binom{i+3}{3}$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have a recurrence:

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i].$$

- Let's try to prove by induction:

$$\mathbb{E}[Y_n] \leq \frac{1}{4} \binom{n+3}{3}.$$

- Base case: $Y_1 = 1 \leq \frac{1}{4}\binom{4}{3}$.
- Assuming it holds for all $i < n$, the recurrence is just

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i] \leq \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4} \binom{i+3}{3} = \frac{1}{n} \sum_{i=0}^{n-1} \binom{i+3}{3}$$

- What is the right-hand side equal to?

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have a recurrence:

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i].$$

- Let's try to prove by induction:

$$\mathbb{E}[Y_n] \leq \frac{1}{4} \binom{n+3}{3}.$$

- Base case: $Y_1 = 1 \leq \frac{1}{4}\binom{4}{3}$.
- Assuming it holds for all $i < n$, the recurrence is just

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i] \leq \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4} \binom{i+3}{3} = \frac{1}{n} \sum_{i=0}^{n-1} \binom{i+3}{3}$$

- By the hockey stick identity, this is just

$$\frac{1}{n} \binom{n+3}{4}$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have a recurrence:

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i].$$

- Let's try to prove by induction:

$$\mathbb{E}[Y_n] \leq \frac{1}{4} \binom{n+3}{3}.$$

- Base case: $Y_1 = 1 \leq \frac{1}{4}\binom{4}{3}$.
- Assuming it holds for all $i < n$, the recurrence is just

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i] \leq \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4} \binom{i+3}{3} = \frac{1}{n} \sum_{i=0}^{n-1} \binom{i+3}{3}$$

- By the hockey stick identity, this is just

$$\frac{1}{n} \binom{n+3}{4} = \frac{1}{n} \frac{(n+3)!}{4!(n-1)!}$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have a recurrence:

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i].$$

- Let's try to prove by induction:

$$\mathbb{E}[Y_n] \leq \frac{1}{4} \binom{n+3}{3}.$$

- Base case: $Y_1 = 1 \leq \frac{1}{4}\binom{4}{3}$.
- Assuming it holds for all $i < n$, the recurrence is just

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i] \leq \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4} \binom{i+3}{3} = \frac{1}{n} \sum_{i=0}^{n-1} \binom{i+3}{3}$$

- By the hockey stick identity, this is just

$$\frac{1}{n}\binom{n+3}{4} = \frac{1}{n} \frac{(n+3)!}{4!(n-1)!} = \frac{(n+3)!}{4!n!}$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have a recurrence:

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i].$$

- Let's try to prove by induction:

$$\mathbb{E}[Y_n] \leq \frac{1}{4} \binom{n+3}{3}.$$

- Base case: $Y_1 = 1 \leq \frac{1}{4}\binom{4}{3}$.
- Assuming it holds for all $i < n$, the recurrence is just

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i] \leq \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4} \binom{i+3}{3} = \frac{1}{n} \sum_{i=0}^{n-1} \binom{i+3}{3}$$

- By the hockey stick identity, this is just

$$\frac{1}{n}\binom{n+3}{4} = \frac{1}{n} \frac{(n+3)!}{4!(n-1)!} = \frac{(n+3)!}{4!n!} = \frac{1}{4} \frac{(n+3)!}{3!n!}$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have a recurrence:

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i].$$

- Let's try to prove by induction:

$$\mathbb{E}[Y_n] \leq \frac{1}{4} \binom{n+3}{3}.$$

- Base case: $Y_1 = 1 \leq \frac{1}{4}\binom{4}{3}$.
- Assuming it holds for all $i < n$, the recurrence is just

$$\mathbb{E}[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} \mathbb{E}[Y_i] \leq \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4} \binom{i+3}{3} = \frac{1}{n} \sum_{i=0}^{n-1} \binom{i+3}{3}$$

- By the hockey stick identity, this is just

$$\frac{1}{n}\binom{n+3}{4} = \frac{1}{n}\frac{(n+3)!}{4!(n-1)!} = \frac{(n+3)!}{4!n!} = \frac{1}{4}\frac{(n+3)!}{3!n!} = \frac{1}{4}\binom{n+3}{3}.$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have proven that:

$$\mathbb{E}[Y_n] \leq \frac{1}{4}\binom{n+3}{3}.$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have proven that:

$$\mathbb{E}[Y_n] \leq \frac{1}{4}\binom{n+3}{3}.$$

- But we want $\mathbb{E}[X_n]$.

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have proven that:

$$\mathbb{E}[Y_n] \leq \frac{1}{4}\binom{n+3}{3}.$$

- But we want $\mathbb{E}[X_n]$.
- How do we go from $\mathbb{E}[Y_n] = \mathbb{E}[2^{X_n}]$ to $\mathbb{E}[X_n]$?

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have proven that:

$$\mathbb{E}[Y_n] \leq \frac{1}{4}\binom{n+3}{3}.$$

- But we want $\mathbb{E}[X_n]$.
- How do we go from $\mathbb{E}[Y_n] = \mathbb{E}[2^{X_n}]$ to $\mathbb{E}[X_n]$?
- We have Jensen's Inequality:

$$\mathbb{E}[f(x)] \geq f(\mathbb{E}[x]) \text{ if } f \text{ is convex.}$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have proven that:

$$\mathbb{E}[Y_n] \leq \frac{1}{4}\binom{n+3}{3}.$$

- But we want $\mathbb{E}[X_n]$.
- How do we go from $\mathbb{E}[Y_n] = \mathbb{E}[2^{X_n}]$ to $\mathbb{E}[X_n]$?
- We have Jensen's Inequality:

$$\mathbb{E}[f(x)] \geq f(\mathbb{E}[x]) \text{ if } f \text{ is convex.}$$

- Is $f(x) = 2^x$ convex/concave/neither?

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have proven that:

$$\mathbb{E}[Y_n] \leq \frac{1}{4}\binom{n+3}{3}.$$

- But we want $\mathbb{E}[X_n]$.
- How do we go from $\mathbb{E}[Y_n] = \mathbb{E}[2^{X_n}]$ to $\mathbb{E}[X_n]$?
- We have Jensen's Inequality:

$$\mathbb{E}[f(x)] \geq f(\mathbb{E}[x]) \text{ if } f \text{ is convex.}$$

- Is $f(x) = 2^x$ convex/concave/neither?
- We have $2^x = (e^{\log 2})^x = e^{(\log 2)x}$, so

$$\frac{d}{dx}2^x = \frac{d}{dx}e^{(\log 2)x} = (\log 2)e^{(\log 2)x}$$
$$\frac{d^2}{dx^2}2^x = (\log 2)\frac{d}{dx}e^{(\log 2)x} = (\log 2)^2 e^{(\log 2)x} = (\log 2)^2 2^x > 0.$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have proven that:

$$\mathbb{E}[Y_n] \leq \frac{1}{4}\binom{n+3}{3}.$$

- But we want $\mathbb{E}[X_n]$.
- How do we go from $\mathbb{E}[Y_n] = \mathbb{E}[2^{X_n}]$ to $\mathbb{E}[X_n]$?
- We have Jensen's Inequality:

$$\mathbb{E}[f(x)] \geq f(\mathbb{E}[x]) \text{ if } f \text{ is convex.}$$

- $f(x) = 2^x$ is convex, so:

$$\frac{1}{4}\binom{n+3}{3} \geq \mathbb{E}[Y_n] = \mathbb{E}[2^{X_n}] \geq 2^{\mathbb{E}[X_n]}.$$

Expected height

Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

- So we have proven that:

$$\mathbb{E}[Y_n] \leq \frac{1}{4}\binom{n+3}{3}.$$

- But we want $\mathbb{E}[X_n]$.
- How do we go from $\mathbb{E}[Y_n] = \mathbb{E}[2^{X_n}]$ to $\mathbb{E}[X_n]$?
- We have Jensen's Inequality:

$$\mathbb{E}[f(x)] \geq f(\mathbb{E}[x]) \text{ if } f \text{ is convex.}$$

- $f(x) = 2^x$ is convex, so:

$$\frac{1}{4}\binom{n+3}{3} \geq \mathbb{E}[Y_n] = \mathbb{E}[2^{X_n}] \geq 2^{\mathbb{E}[X_n]}.$$

- Therefore $\mathbb{E}[X_n] = O\left(\log\left(\frac{1}{4}\binom{n+3}{3}\right)\right) = \boxed{O(\log n)}$, since $\log(p(n)) = O(\log n)$ for any polynomial $p(n)$ (e.g. $\log(n^3) = 3\log n$).

Running time for operations

Running time for operations

- Therefore, in many cases we may expect binary search tree operations to be $O(\log n)$.

Running time for operations

- Therefore, in many cases we may expect binary search tree operations to be $O(\log n)$.
- However, this kind of average-case analysis doesn't necessarily help with any particular tree.

Running time for operations

- Therefore, in many cases we may expect binary search tree operations to be $O(\log n)$.
- However, this kind of average-case analysis doesn't necessarily help with any particular tree.
- We will now see a way to *make sure* that $h = O(\log n)$ not $O(n)$.

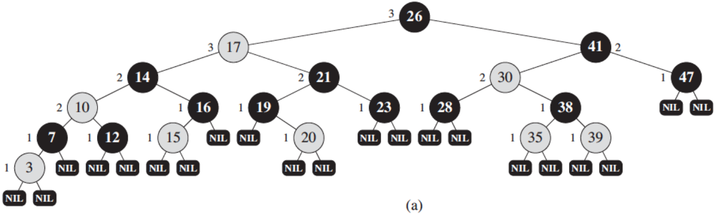
Red-black trees

Red-black trees

- Red-black trees are a type of *self-balancing tree*, in which operations on the tree ensure it has height $O(\log n)$.

Red-black trees

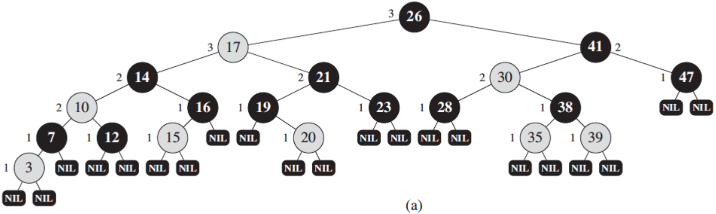
- Red-black trees are a type of *self-balancing tree*, in which operations on the tree ensure it has height $O(\log n)$.
- Specifically, a *red-black tree* is a binary search tree with the following conditions.



(a)

Red-black trees

- Red-black trees are a type of *self-balancing tree*, in which operations on the tree ensure it has height $O(\log n)$.
- Specifically, a *red-black tree* is a binary search tree with the following conditions.

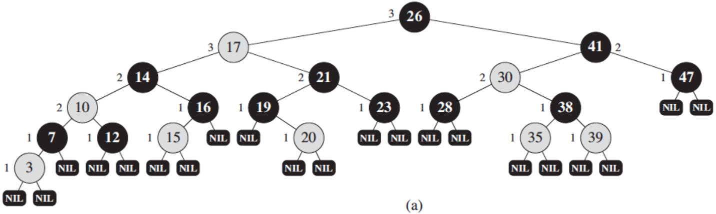


(a)

  - Each node stores a key *except the leaves*, which store NIL.

Red-black trees

- Red-black trees are a type of *self-balancing tree*, in which operations on the tree ensure it has height $O(\log n)$.
- Specifically, a *red-black tree* is a binary search tree with the following conditions.

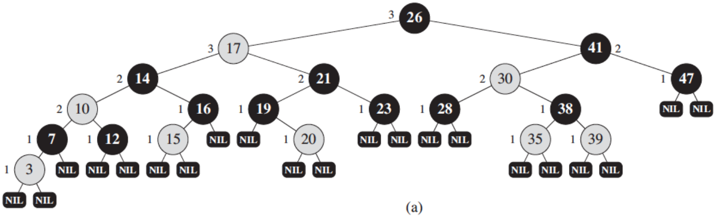


(a)

  - Each node stores a key *except the leaves*, which store NIL.
  - Each node except the leaves has two children.

Red-black trees

- Red-black trees are a type of *self-balancing tree*, in which operations on the tree ensure it has height $O(\log n)$.
- Specifically, a *red-black tree* is a binary search tree with the following conditions.

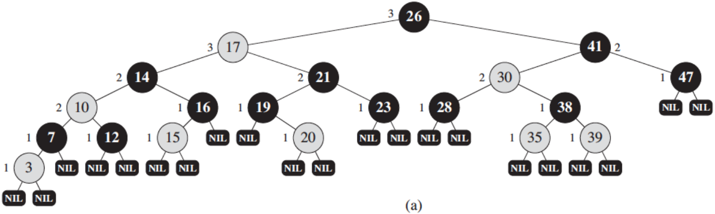


(a)

  - Each node stores a key *except the leaves*, which store NIL.
  - Each node except the leaves has two children.
  - Each node has a *color*, either red or black.

Red-black trees

- Red-black trees are a type of *self-balancing tree*, in which operations on the tree ensure it has height $O(\log n)$.
- Specifically, a *red-black tree* is a binary search tree with the following conditions.

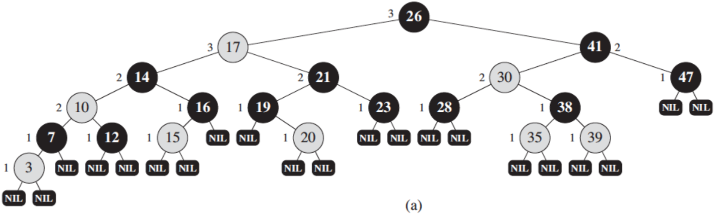


(a)

  - Each node stores a key *except the leaves*, which store NIL.
  - Each node except the leaves has two children.
  - Each node has a *color*, either red or black.
  - The root is black.

Red-black trees

- Red-black trees are a type of *self-balancing tree*, in which operations on the tree ensure it has height $O(\log n)$.
- Specifically, a *red-black tree* is a binary search tree with the following conditions.

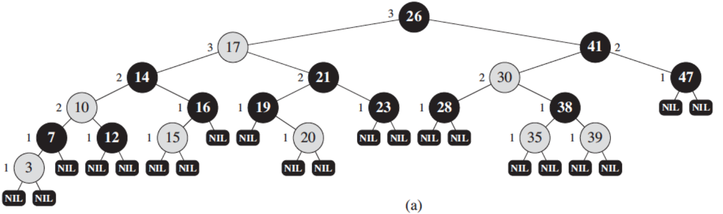


(a)

  - Each node stores a key *except the leaves*, which store NIL.
  - Each node except the leaves has two children.
  - Each node has a *color*, either red or black.
  - The root is black.
  - All the leaves are black.

Red-black trees

- Red-black trees are a type of *self-balancing tree*, in which operations on the tree ensure it has height $O(\log n)$.
- Specifically, a *red-black tree* is a binary search tree with the following conditions.

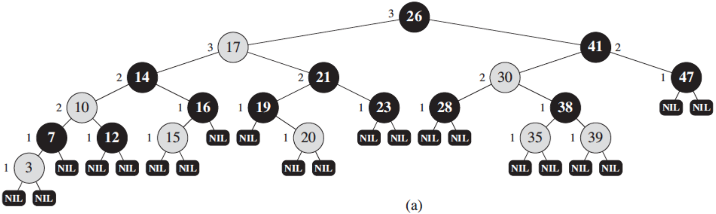


(a)

  - Each node stores a key *except the leaves*, which store NIL.
  - Each node except the leaves has two children.
  - Each node has a *color*, either red or black.
  - The root is black.
  - All the leaves are black.
  - If a node is red, both its children are colored black.

Red-black trees

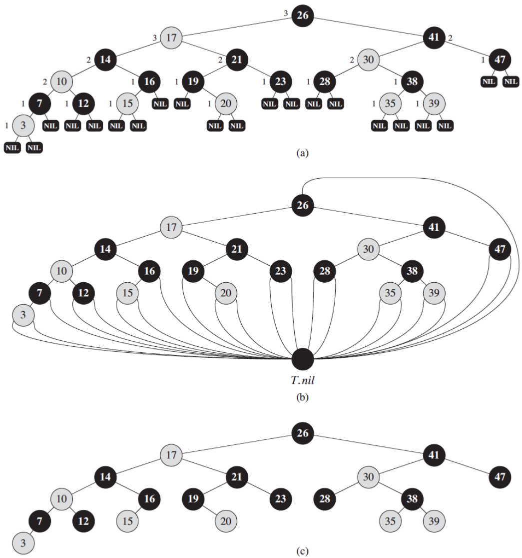
- Red-black trees are a type of *self-balancing tree*, in which operations on the tree ensure it has height $O(\log n)$.
- Specifically, a *red-black tree* is a binary search tree with the following conditions.



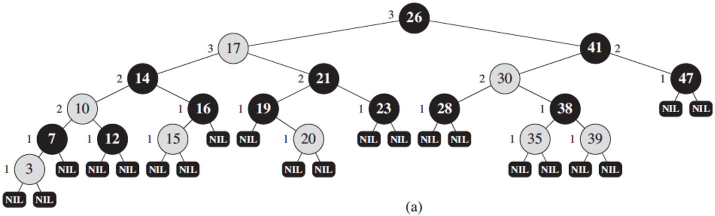
(a)

  - Each node stores a key *except the leaves*, which store NIL.
  - Each node except the leaves has two children.
  - Each node has a *color*, either red or black.
  - The root is black.
  - All the leaves are black.
  - If a node is red, both its children are colored black.
  - For each node, all paths from the node to the descendant leaves have the same number of black nodes.

Red-black trees

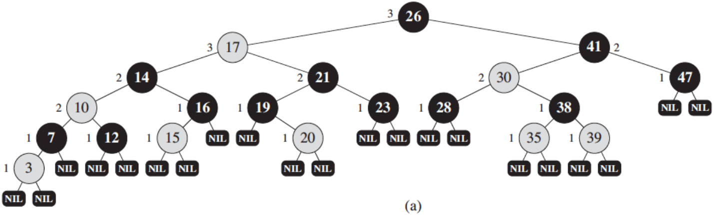


Definitions



(a)

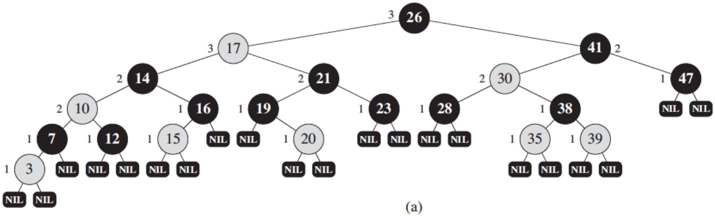
Definitions



(a)

- The *height* of a node in a rooted tree is the number of layers it is from the bottom (bottom layer = height 0, next layer = height 1, etc.)

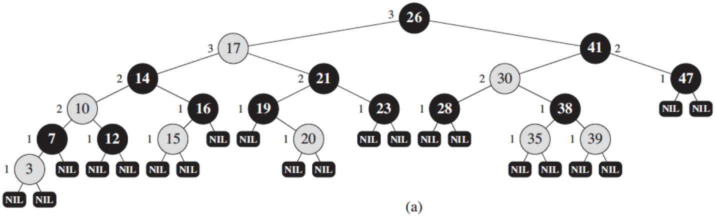
Definitions



(a)

- The *height* of a node in a rooted tree is the number of layers it is from the bottom (bottom layer = height 0, next layer = height 1, etc.)
- Formally, the height of a node equals the height of the tree minus the depth of the node.

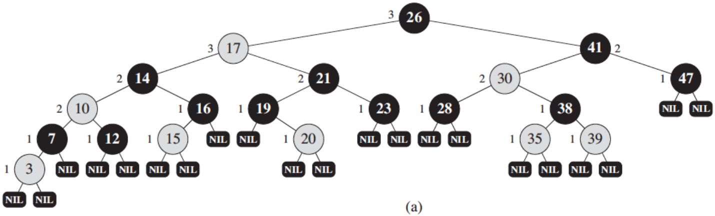
Definitions



(a)

- The *height* of a node in a rooted tree is the number of layers it is from the bottom (bottom layer = height 0, next layer = height 1, etc.)
- Formally, the height of a node equals the height of the tree minus the depth of the node.
- In a red-black tree, the *black-height* bh(x) of a node x is the number of black nodes on a path from x to a leaf descendant (not including the node x itself if it is black).

Definitions



(a)

- The *height* of a node in a rooted tree is the number of layers it is from the bottom (bottom layer = height 0, next layer = height 1, etc.)
- Formally, the height of a node equals the height of the tree minus the depth of the node.
- In a red-black tree, the *black-height* bh(x) of a node x is the number of black nodes on a path from x to a leaf descendant (not including the node x itself if it is black).
- We say that an *internal node* of a red-black tree is any node that isn't a leaf (so any node containing a key).

Height

Claim: The subtree rooted at a node x has at least $2^{\text{bh}(x)} - 1$ internal nodes.

Height

Claim: The subtree rooted at a node x has at least $2^{\text{bh}(x)} - 1$ internal nodes.

- What technique can we try to prove this?

Height

Claim: The subtree rooted at a node x has at least $2^{\mathrm{bh}(x)} - 1$ internal nodes.

- Let's try induction. What can we induct on?

Height

Claim: The subtree rooted at a node x has at least $2^{\text{bh}(x)} - 1$ internal nodes.

- Let's try induction. What can we induct on?
- Let's induct on the height of x (not the black-height).

Height

Claim: The subtree rooted at a node x has at least $2^{\mathrm{bh}(x)} - 1$ internal nodes.

- Let's try induction. What can we induct on?
- Let's induct on the height of x (not the black-height).
- What is a good base case?

Height

Claim: The subtree rooted at a node x has at least $2^{\mathrm{bh}(x)} - 1$ internal nodes.

- Let's try induction. What can we induct on?
- Let's induct on the height of x (not the black-height).
- Base case as small as possible: height=0.

Height

Claim: The subtree rooted at a node x has at least $2^{\text{bh}(x)} - 1$ internal nodes.

- Let's try induction. What can we induct on?
- Let's induct on the height of x (not the black-height).
- Base case as small as possible: height=0.
- Then x is a leaf and $\text{bh}(x) = 0$. There are indeed $2^0 - 1 = 0$ internal nodes in the subtree.

Height

Claim: The subtree rooted at a node x has at least $2^{\text{bh}(x)} - 1$ internal nodes.

- Let's try induction. What can we induct on?
- Let's induct on the height of x (not the black-height).
- Base case as small as possible: height=0.
- Then x is a leaf and $\text{bh}(x) = 0$. There are indeed $2^0 - 1 = 0$ internal nodes in the subtree.
- Now assume true for height h, look at x with height $h + 1$.

Height

Claim: The subtree rooted at a node x has at least $2^{\text{bh}(x)} - 1$ internal nodes.

- Let's try induction. What can we induct on?
- Let's induct on the height of x (not the black-height).
- Base case as small as possible: height=0.
- Then x is a leaf and $\text{bh}(x) = 0$. There are indeed $2^0 - 1 = 0$ internal nodes in the subtree.
- Now assume true for height h, look at x with height $h + 1$.
- If x is a leaf, then $\text{bh}(x) = 0$, so claim is true.

Height

Claim: The subtree rooted at a node x has at least $2^{\mathrm{bh}(x)} - 1$ internal nodes.

- Let's try induction. What can we induct on?
- Let's induct on the height of x (not the black-height).
- Base case as small as possible: height=0.
- Then x is a leaf and $\mathrm{bh}(x) = 0$. There are indeed $2^0 - 1 = 0$ internal nodes in the subtree.
- Now assume true for height h, look at x with height $h + 1$.
- If x is a leaf, then $\mathrm{bh}(x) = 0$, so claim is true.
- Otherwise, x has two children y and z, with $\mathrm{bh}(y)$ and $\mathrm{bh}(z)$ both either $\mathrm{bh}(x)$ or $\mathrm{bh}(x) - 1$.

Height

Claim: The subtree rooted at a node x has at least $2^{\text{bh}(x)} - 1$ internal nodes.

- Let's try induction. What can we induct on?
- Let's induct on the height of x (not the black-height).
- Base case as small as possible: height=0.
- Then x is a leaf and $\text{bh}(x) = 0$. There are indeed $2^0 - 1 = 0$ internal nodes in the subtree.
- Now assume true for height h, look at x with height $h + 1$.
- If x is a leaf, then $\text{bh}(x) = 0$, so claim is true.
- Otherwise, x has two children y and z, with $\text{bh}(y)$ and $\text{bh}(z)$ both either $\text{bh}(x)$ or $\text{bh}(x) - 1$.
- Since height of y and z less than x, inductive hypothesis implies the subtrees rooted at y and z each have at least $2^{\text{bh}(x)-1} - 1$ internal nodes.

Height

Claim: The subtree rooted at a node x has at least $2^{\text{bh}(x)} - 1$ internal nodes.

- Let's try induction. What can we induct on?
- Let's induct on the height of x (not the black-height).
- Base case as small as possible: height=0.
- Then x is a leaf and $\text{bh}(x) = 0$. There are indeed $2^0 - 1 = 0$ internal nodes in the subtree.
- Now assume true for height h, look at x with height $h + 1$.
- If x is a leaf, then $\text{bh}(x) = 0$, so claim is true.
- Otherwise, x has two children y and z, with $\text{bh}(y)$ and $\text{bh}(z)$ both either $\text{bh}(x)$ or $\text{bh}(x) - 1$.
- Since height of y and z less than x, inductive hypothesis implies the subtrees rooted at y and z each have at least $2^{\text{bh}(x)-1} - 1$ internal nodes.
- So subtree rooted at x has at least

$$1 + \left(2^{\text{bh}(x)-1} - 1\right) + \left(2^{\text{bh}(x)-1} - 1\right) = 2^{\text{bh}(x)} - 1$$

  internal nodes, finishing the induction.

Height

The height of a red-black tree with n internal nodes is $O(\log n)$.

Height

The height of a red-black tree with n internal nodes is $O(\log n)$.

- Let us prove this using the claim we just proved: The subtree rooted at a node x has at least $2^{\mathrm{bh}(x)} - 1$ internal nodes.

Height

The height of a red-black tree with n internal nodes is $O(\log n)$.

- Let us prove this using the claim we just proved: The subtree rooted at a node x has at least $2^{\mathrm{bh}(x)} - 1$ internal nodes.
- What x should we take?

Height

The height of a red-black tree with n internal nodes is $O(\log n)$.

- Let us prove this using the claim we just proved: The subtree rooted at a node x has at least $2^{\mathrm{bh}(x)} - 1$ internal nodes.
- What x should we take?
- If x is the root, we have:

$$n \geq 2^{\mathrm{bh}(x)} - 1.$$

Height

The height of a red-black tree with n internal nodes is $O(\log n)$.

- Let us prove this using the claim we just proved: The subtree rooted at a node x has at least $2^{\text{bh}(x)} - 1$ internal nodes.
- What x should we take?
- If x is the root, we have:
$$n \geq 2^{\text{bh}(x)} - 1.$$
- Remember we assumed that in a red-black tree, if a node is red, both its children are colored black.

Height

The height of a red-black tree with n internal nodes is $O(\log n)$.

- Let us prove this using the claim we just proved: The subtree rooted at a node x has at least $2^{\mathrm{bh}(x)} - 1$ internal nodes.
- What x should we take?
- If x is the root, we have:

$$n \geq 2^{\mathrm{bh}(x)} - 1.$$

- Remember we assumed that in a red-black tree, if a node is red, both its children are colored black.
- How does this help?

Height

The height of a red-black tree with n internal nodes is $O(\log n)$.

- Let us prove this using the claim we just proved: The subtree rooted at a node x has at least $2^{\text{bh}(x)} - 1$ internal nodes.
- What x should we take?
- If x is the root, we have:

$$n \geq 2^{\text{bh}(x)} - 1.$$

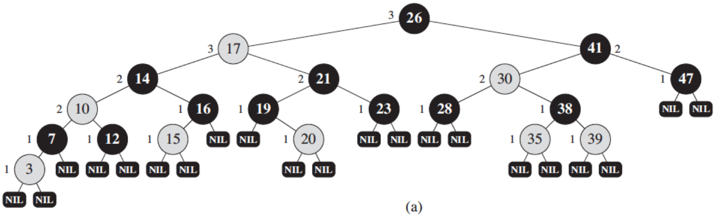
- Remember we assumed that in a red-black tree, if a node is red, both its children are colored black.
- How does this help?
- We know that $\text{bh}(x)$ is at least half the height of x, so

$$n + 1 \geq 2^{\text{height}(x)/2},$$

implying

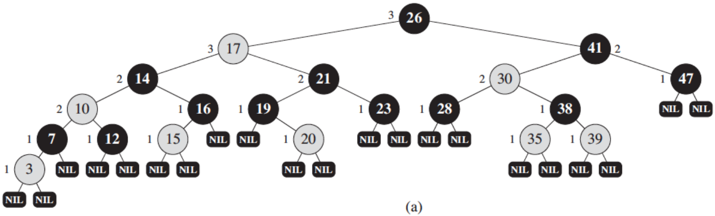
$$\text{height}(x) \leq 2 \log_2(n + 1) = O(\log n).$$

Tree operations



(a)

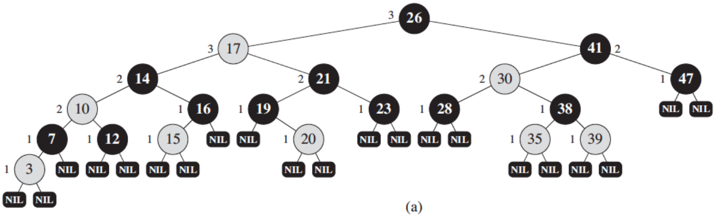
Tree operations



(a)

- Last time, we looked at the tree operations Search, Maximum, Minimum, Successor, Predecessor, Insert, and Delete.

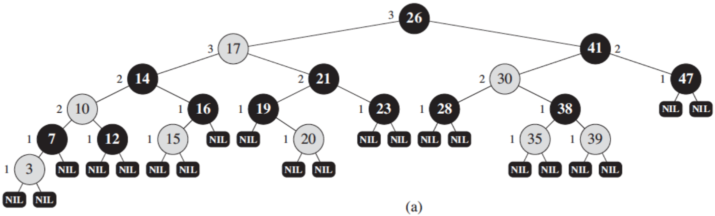
Tree operations



(a)

- Last time, we looked at the tree operations Search, Maximum, Minimum, Successor, Predecessor, Insert, and Delete.
- Which ones require special attention for a red-black tree?

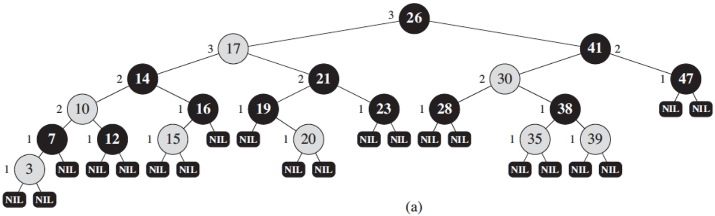
Tree operations



(a)

- Last time, we looked at the tree operations Search, Maximum, Minimum, Successor, Predecessor, Insert, and Delete.
- Which ones require special attention for a red-black tree?
- Search, Maximum, Minimum, Successor, and Predecessor all work normally since they don't change the tree – a red-black tree is a binary search tree, just with additional information.

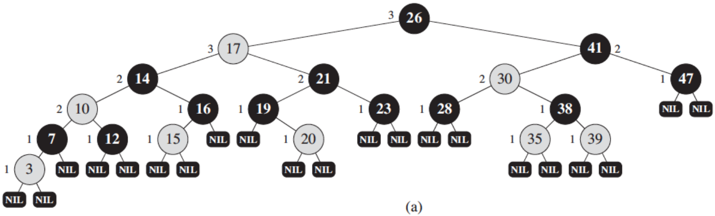
Tree operations



(a)

- Last time, we looked at the tree operations Search, Maximum, Minimum, Successor, Predecessor, Insert, and Delete.
- Which ones require special attention for a red-black tree?
- Search, Maximum, Minimum, Successor, and Predecessor all work normally since they don't change the tree – a red-black tree is a binary search tree, just with additional information.
- So all these operations run naturally in time $O(\log n)$.

Tree operations



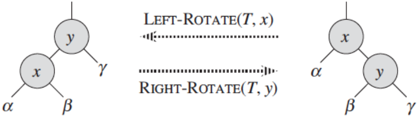
(a)

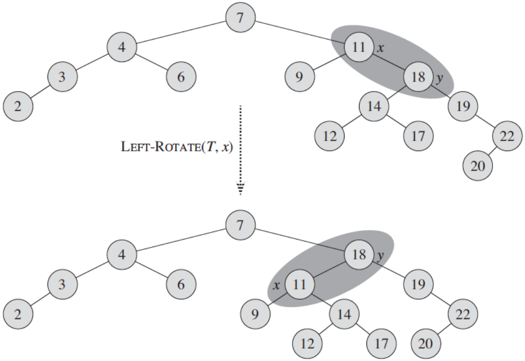
- Last time, we looked at the tree operations Search, Maximum, Minimum, Successor, Predecessor, Insert, and Delete.
- Which ones require special attention for a red-black tree?
- Search, Maximum, Minimum, Successor, and Predecessor all work normally since they don't change the tree – a red-black tree is a binary search tree, just with additional information.
- So all these operations run naturally in time $O(\log n)$.
- Insert and Delete must be changed so the red/black conditions work.

Rotations

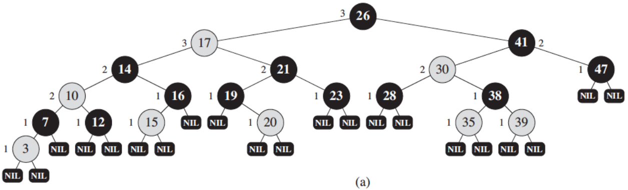
Rotations

- We will use the following operations, called *left rotation* and *right rotation*:



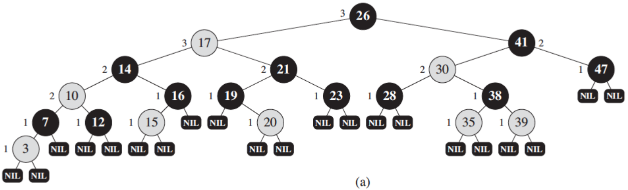


Insert



(a)

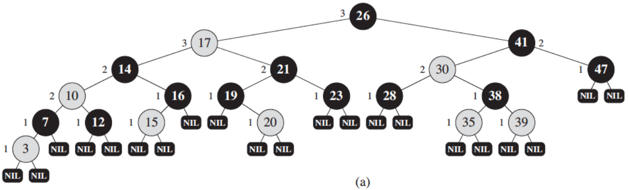
Insert



(a)

- Let's remember our procedure from last time: Run Search on the key we want to insert and add it at a leaf.

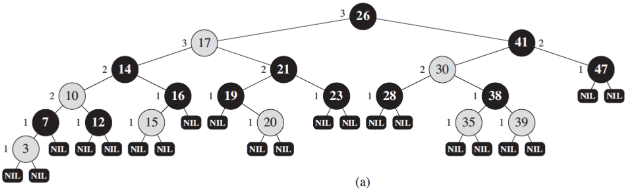
Insert



(a)

- Let’s remember our procedure from last time: Run Search on the key we want to insert and add it at a leaf.
- First minor difference we need: since the leaves of a red-black tree store NIL, we add the key and make two new leaves for its children.

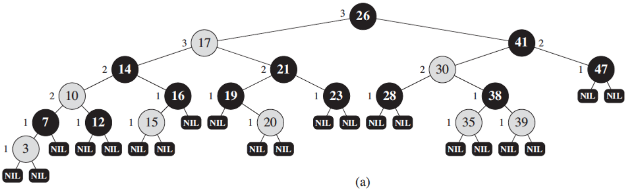
Insert



(a)

- Let's remember our procedure from last time: Run Search on the key we want to insert and add it at a leaf.
- First minor difference we need: since the leaves of a red-black tree store NIL, we add the key and make two new leaves for its children.
- Big difference: working out the colors.

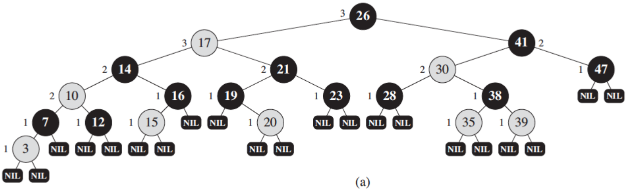
Insert



(a)

- Let's remember our procedure from last time: Run Search on the key we want to insert and add it at a leaf.
- First minor difference we need: since the leaves of a red-black tree store NIL, we add the key and make two new leaves for its children.
- Big difference: working out the colors.
- Let's start by doing a normal Insert with the new node colored red.

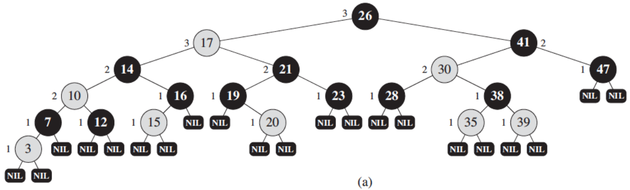
Insert



(a)

- Let's remember our procedure from last time: Run Search on the key we want to insert and add it at a leaf.
- First minor difference we need: since the leaves of a red-black tree store NIL, we add the key and make two new leaves for its children.
- Big difference: working out the colors.
- Let's start by doing a normal Insert with the new node colored red.
- Which of these red-black tree conditions might be violated?
  - The root is black.
  - All the leaves are black.
  - Both children of a red node are colored black.
  - For each node, all paths from the node to the descendant leaves have the same number of black nodes.

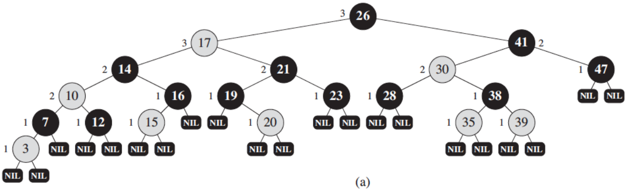
Insert



(a)

- Let's remember our procedure from last time: Run Search on the key we want to insert and add it at a leaf.
- First minor difference we need: since the leaves of a red-black tree store NIL, we add the key and make two new leaves for its children.
- Big difference: working out the colors.
- Let's start by doing a normal Insert with the new node colored red.
- Which of these red-black tree conditions might be violated?
  - The root is black. Might be false if we added the root.
  - All the leaves are black.
  - Both children of a red node are colored black.
  - For each node, all paths from the node to the descendant leaves have the same number of black nodes.

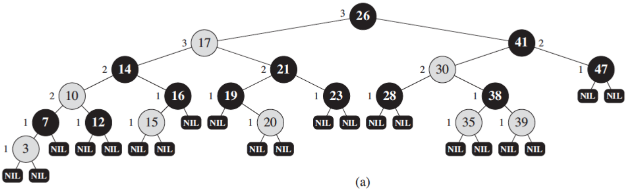
Insert



(a)

- Let's remember our procedure from last time: Run Search on the key we want to insert and add it at a leaf.
- First minor difference we need: since the leaves of a red-black tree store NIL, we add the key and make two new leaves for its children.
- Big difference: working out the colors.
- Let's start by doing a normal Insert with the new node colored red.
- Which of these red-black tree conditions might be violated?
  - The root is black. Might be false if we added the root.
  - All the leaves are black. Still true.
  - Both children of a red node are colored black.
  - For each node, all paths from the node to the descendant leaves have the same number of black nodes.

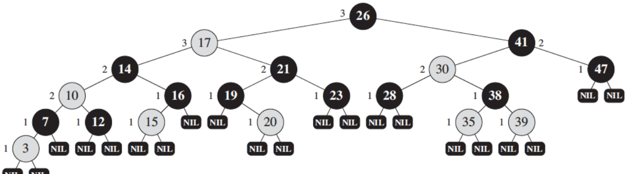
Insert



(a)

- Let's remember our procedure from last time: Run Search on the key we want to insert and add it at a leaf.
- First minor difference we need: since the leaves of a red-black tree store NIL, we add the key and make two new leaves for its children.
- Big difference: working out the colors.
- Let's start by doing a normal Insert with the new node colored red.
- Which of these red-black tree conditions might be violated?
    - The root is black. Might be false if we added the root.
    - All the leaves are black. Still true.
    - Both children of a red node are colored black. Might be false if parent is red.
    - For each node, all paths from the node to the descendant leaves have the same number of black nodes.

Insert



(a)

- Let's remember our procedure from last time: Run Search on the key we want to insert and add it at a leaf.
- First minor difference we need: since the leaves of a red-black tree store NIL, we add the key and make two new leaves for its children.
- Big difference: working out the colors.
- Let's start by doing a normal Insert with the new node colored red.
- Which of these red-black tree conditions might be violated?
  - The root is black. Might be false if we added the root.
  - All the leaves are black. Still true.
  - Both children of a red node are colored black. Might be false if parent is red.
  - For each node, all paths from the node to the descendant leaves have the same number of black nodes. Still true.

Insert

Insert

- We insert a new node as before and color it red.

Insert

- We insert a new node as before and color it red.
- If the node is the root, then we simply recolor it black.

Insert

- We insert a new node as before and color it red.
- If the node is the root, then we simply recolor it black.
- What if the parent of the new node is red?

Insert

- We insert a new node as before and color it red.
- If the node is the root, then we simply recolor it black.
- What if the parent of the new node is red?
- In that situation, define the *uncle* of the new node to be the other child of its parent's parent.

Insert

- We insert a new node as before and color it red.
- If the node is the root, then we simply recolor it black.
- What if the parent of the new node is red?
- In that situation, define the *uncle* of the new node to be the other child of its parent's parent.
- **Case 1.** The uncle of the new node is red.

Insert

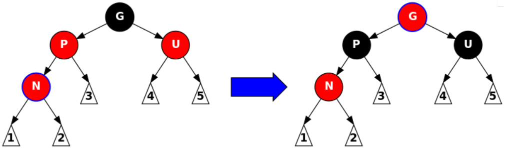
- We insert a new node as before and color it red.
- If the node is the root, then we simply recolor it black.
- What if the parent of the new node is red?
- In that situation, define the *uncle* of the new node to be the other child of its parent's parent.
- **Case 1.** The uncle of the new node is red.
- **Case 2.** The uncle is colored black, and the new node is a right child.

Insert

- We insert a new node as before and color it red.
- If the node is the root, then we simply recolor it black.
- What if the parent of the new node is red?
- In that situation, define the *uncle* of the new node to be the other child of its parent's parent.
- **Case 1.** The uncle of the new node is red.
- **Case 2.** The uncle is colored black, and the new node is a right child.
- **Case 3.** The uncle is colored black, and the new node is a left child.

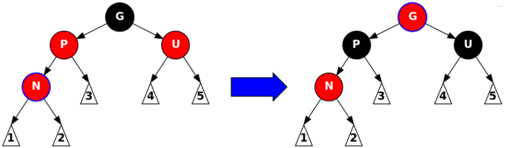
Insert

Case 1. The uncle of the new node is red.



Insert

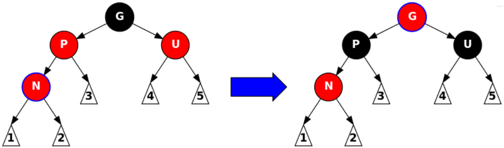
Case 1. The uncle of the new node is red.



- We swap colors as shown.

Insert

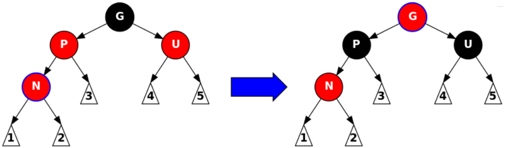
Case 1. The uncle of the new node is red.



- We swap colors as shown.
- Can check doesn't lead to any new violations.

Insert

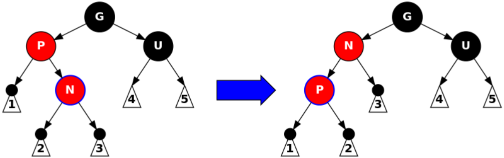
Case 1. The uncle of the new node is red.



- We swap colors as shown.
- Can check doesn't lead to any new violations.
- Except that the grandparent may now be a red violation if its own parent is red – in that case, we can recursively repeat the correction process we are now doing.

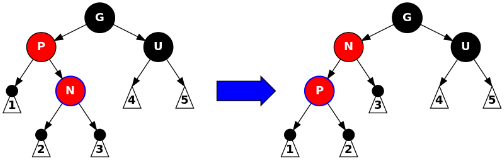
Insert

Case 2. The uncle is colored black, and the new node is a right child.



Insert

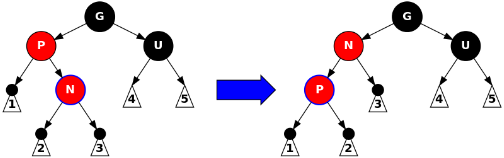
Case 2. The uncle is colored black, and the new node is a right child.



- We run a left rotation on the parent to reduce to the next case, case 3.

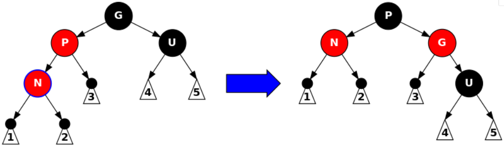
Insert

Case 2. The uncle is colored black, and the new node is a right child.



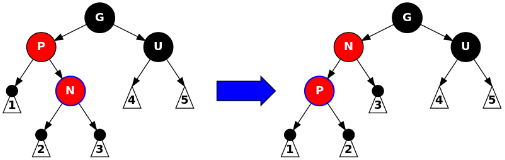
- We run a left rotation on the parent to reduce to the next case, case 3.

Case 3. The uncle is colored black, and the new node is a left child.



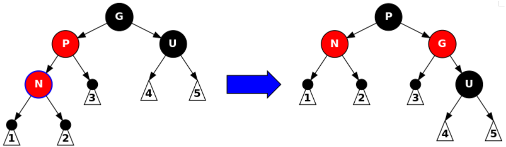
Insert

Case 2. The uncle is colored black, and the new node is a right child.



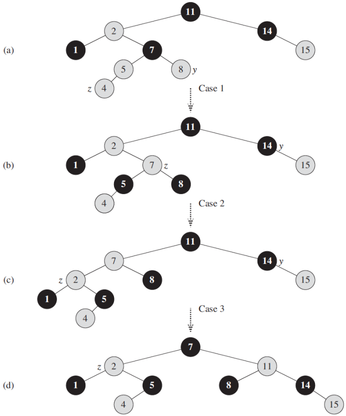
- We run a left rotation on the parent to reduce to the next case, case 3.

Case 3. The uncle is colored black, and the new node is a left child.



- We run a right rotation on the grandparent, and then swap the colors of the parent and grandparent.

Insert summary



Red-black conditions:

- The root is black.
- Both children of a red node are colored black.
- For each node, all paths from the node to the descendant leaves have the same number of black nodes.

Delete

Delete

- Delete in a red-black tree is a bit more complicated.

Delete

- Delete in a red-black tree is a bit more complicated.
- But there is a way to do it in $O(\log n)$ time.

Delete

- Delete in a red-black tree is a bit more complicated.
- But there is a way to do it in $O(\log n)$ time.
- So all our operations on a red-black tree run in time $O(\log n)$.

Next time!

Hashing