Probability & Computing

Overview & The Power of Randomness

Why is randomness useful in computation?

- Randomness facilitates the development of algorithms and data structures.

"For many applications, a randomized algorithm is the simplest algorithm available, or the fastest, or both."

"Randomized Algorithms", Motwani & Raghavan, 1995

Why is randomness useful in computation?

- Randomness facilitates the development of algorithms and data structures.

“For many applications, a randomized algorithm is the simplest algorithm available, or the fastest, or both.”

“Randomized Algorithms”, Motwani & Raghavan, 1995

- Sometimes a randomized approach is the *only* solution!

Why is randomness useful in computation?

- Randomness facilitates the development of algorithms and data structures.

  "For many applications, a randomized algorithm is the simplest algorithm available, or the fastest, or both."

  "Randomized Algorithms", Motwani & Raghavan, 1995

- Sometimes a randomized approach is the *only* solution!

Idea

- Utilize randomness in algorithms and data structures to obtain much better performance than that of deterministic approaches

Why is randomness useful in computation?

- Randomness facilitates the development of algorithms and data structures.

https://i.imgflip.com/3ajf5v.jpg?a470534

"For many applications, a randomized algorithm is the simplest algorithm available, or the fastest, or both."

"Randomized Algorithms", Motwani & Raghavan, 1995

- Sometimes a randomized approach is the *only* solution!

Idea

- Utilize randomness in algorithms and data structures to obtain much better performance than that of deterministic approaches
- But we have to pay for that . . .

Why is randomness useful in computation?

- Randomness facilitates the development of algorithms and data structures.

"For many applications, a randomized algorithm is the simplest algorithm available, or the fastest, or both."

"Randomized Algorithms", Motwani & Raghavan, 1995

https://i.imgflip.com/3ajf5v.jpg?a470534

- Sometimes a randomized approach is the *only* solution!

Idea

- Utilize randomness in algorithms and data structures to obtain much better performance than that of deterministic approaches
- But we have to pay for that . . .
  - Maybe we only *expect* the approach to be fast

Why is randomness useful in computation?

- Randomness facilitates the development of algorithms and data structures.

"For many applications, a randomized algorithm is the simplest algorithm available, or the fastest, or both."

"Randomized Algorithms", Motwani & Raghavan, 1995

https://i.imgflip.com/3ajf5v.jpg?a470534

- Sometimes a randomized approach is the *only* solution!

Idea

- Utilize randomness in algorithms and data structures to obtain much better performance than that of deterministic approaches
- But we have to pay for that . . .
  - Maybe we only *expect* the approach to be fast
  - Maybe we only *expect* the approach to work correctly

Why is randomness useful in computation?

- Randomness facilitates the development of algorithms and data structures.

https://i.imgflip.com/3ajf5v.jpg?a470534

"For many applications, a randomized algorithm is the simplest algorithm available, or the fastest, or both."

"Randomized Algorithms", Motwani & Raghavan, 1995

- Sometimes a randomized approach is the *only* solution!

Idea

- Utilize randomness in algorithms and data structures to obtain much better performance than that of deterministic approaches
- But we have to pay for that . . .
  - Maybe we only *expect* the approach to be fast
  - Maybe we only *expect* the approach to work correctly
- Goal: develop methods that fail only rarely

Why is randomness useful in computation?

- Useful when bridging the theory-practice gap regarding the performance of an appraoch

Why is randomness useful in computation?

- Useful when bridging the theory-practice gap regarding the performance of an appraoch

Theory-Practice Gap

- Algorithm performance often measured by worst-case running time (strong guarantee)

Why is randomness useful in computation?

- Useful when bridging the theory-practice gap regarding the performance of an appraoch

Theory-Practice Gap

- Algorithm performance often measured by worst-case running time (strong guarantee)
- Observe much better performance in practice than expected

Why is randomness useful in computation?

- Useful when bridging the theory-practice gap regarding the performance of an appraoch

Theory-Practice Gap

- Algorithm performance often measured by worst-case running time (strong guarantee)
- Observe much better performance in practice than expected
- Example: bidirectional Breadth-First-Search
  - no asymptotic speed-up compared to standard BFS in the worst case

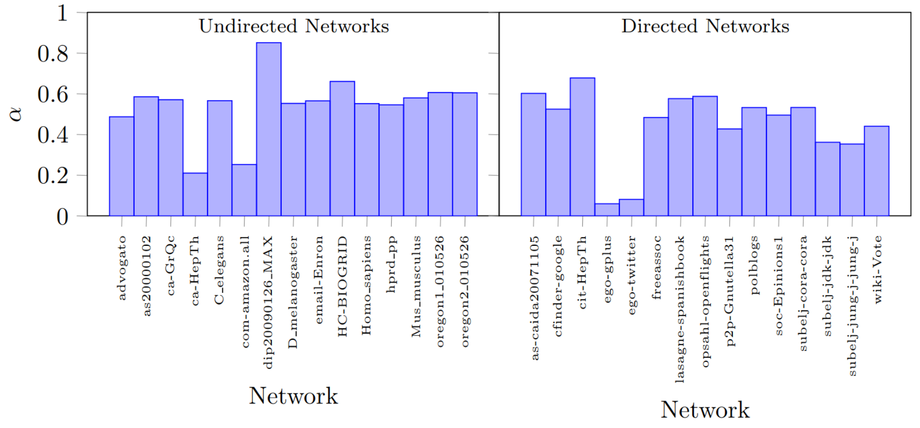
Why is randomness useful in computation?

- Useful when bridging the theory-practice gap regarding the performance of an appraoch

Theory-Practice Gap

- Algorithm performance often measured by worst-case running time (strong guarantee)
- Observe much better performance in practice than expected
- Example: bidirectional Breadth-First-Search
  - no asymptotic speed-up compared to standard BFS in the worst case
  - sublinear running time observed on many real-world networks

“KADABRA is an ADaptive Algorithm for Betweenness via Random Approximation”, Borassi & Natale, JEA, 2019



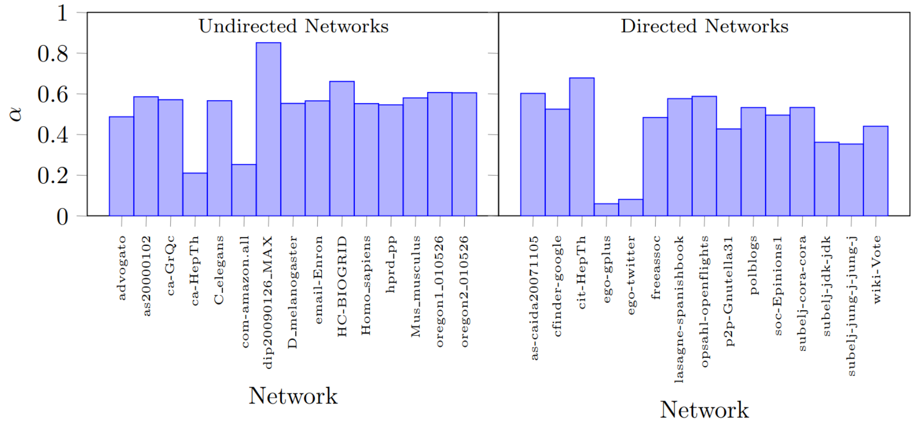
Why is randomness useful in computation?

- Useful when bridging the theory-practice gap regarding the performance of an appraoch

Theory-Practice Gap

- Algorithm performance often measured by worst-case running time (strong guarantee)
- Observe much better performance in practice than expected
- Example: bidirectional Breadth-First-Search
  - no asymptotic speed-up compared to standard BFS in the worst case
  - sublinear running time observed on many real-world networks

“KADABRA is an ADaptive Algorithm for Betweenness via Random Approximation”, Borassi & Natale, JEA, 2019



Average-Case Analysis

- Distinguish practical instances from the worst case

Why is randomness useful in computation?

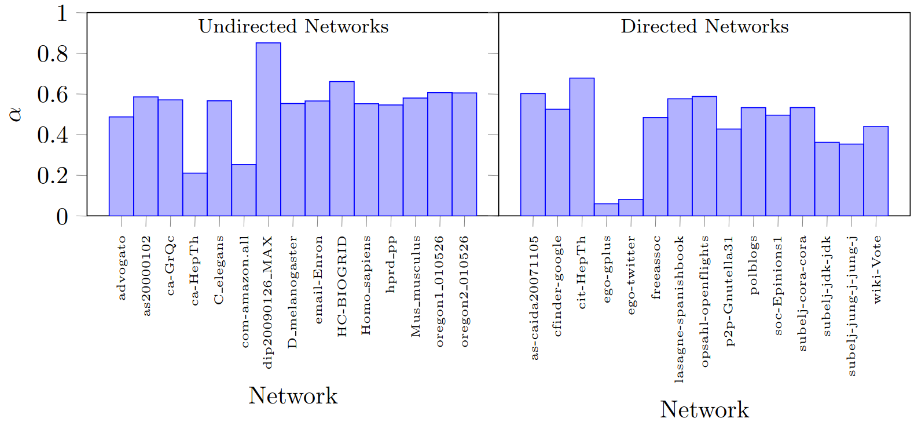
Karlsruhe Institute of Technology

- Useful when bridging the theory-practice gap regarding the performance of an appraoch

Theory-Practice Gap

- Algorithm performance often measured by worst-case running time (strong guarantee)
- Observe much better performance in practice than expected
- Example: bidirectional Breadth-First-Search
  - no asymptotic speed-up compared to standard BFS in the worst case
  - sublinear running time observed on many real-world networks

"KADABRA is an ADaptive Algorithm for Betweenness via Random Approximation", Borassi & Natale, JEA, 2019



Average-Case Analysis

- Distinguish practical instances from the worst case
- Define probabilistic distributions (over possible inputs) that favor realistic instances

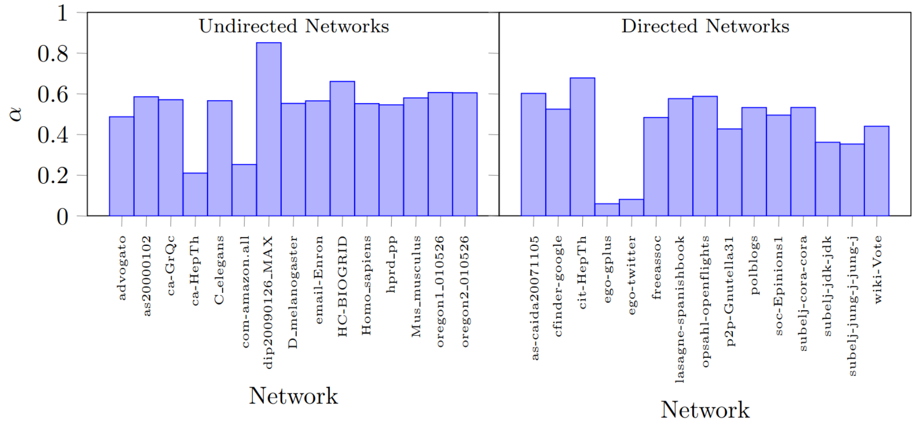
Why is randomness useful in computation?

- Useful when bridging the theory-practice gap regarding the performance of an appraoch

Theory-Practice Gap

- Algorithm performance often measured by worst-case running time (strong guarantee)
- Observe much better performance in practice than expected
- Example: bidirectional Breadth-First-Search
  - no asymptotic speed-up compared to standard BFS in the worst case
  - sublinear running time observed on many real-world networks

“KADABRA is an ADaptive Algorithm for Betweenness via Random Approximation”, Borassi & Natale, JEA, 2019



Average-Case Analysis

- Distinguish practical instances from the worst case
- Define probabilistic distributions (over possible inputs) that favor realistic instances
- Analyze performance assuming input is drawn from the distribution

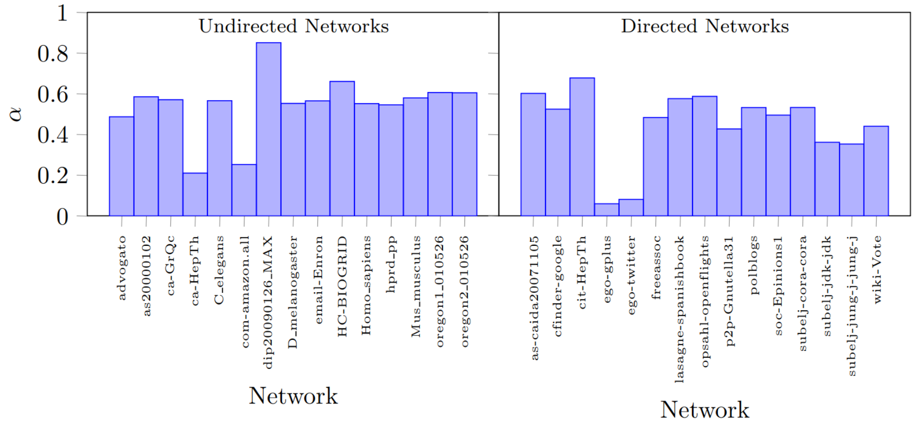
Why is randomness useful in computation?

- Useful when bridging the theory-practice gap regarding the performance of an appraoch

Theory-Practice Gap

- Algorithm performance often measured by worst-case running time (strong guarantee)
- Observe much better performance in practice than expected
- Example: bidirectional Breadth-First-Search
  - no asymptotic speed-up compared to standard BFS in the worst case
  - sublinear running time observed on many real-world networks

"KADABRA is an ADaptive Algorithm for Betweenness via Random Approximation", Borassi & Natale, JEA, 2019



Average-Case Analysis

- Distinguish practical instances from the worst case
- Define probabilistic distributions (over possible inputs) that favor realistic instances
- Analyze performance assuming input is drawn from the distribution
- Expect good performance when hard instances are sufficiently unlikely

Overview

Karlsruhe Institute of Technology

Randomized Algorithms & Data Structures

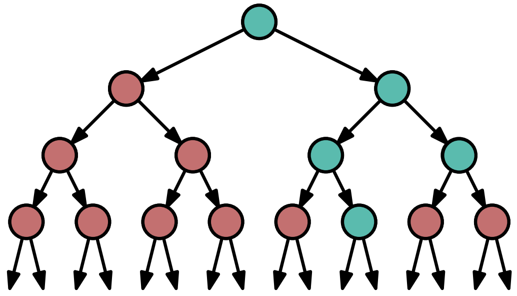
- Probability Amplification
- Streaming / Online-algorithms
- Hashing

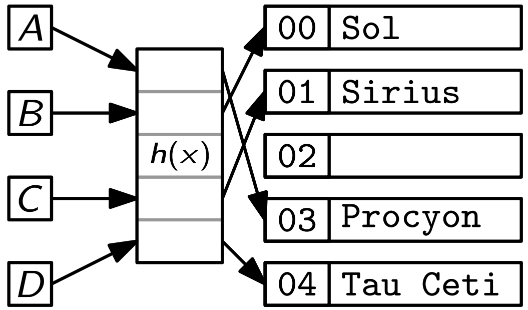
Average-Case Analysis

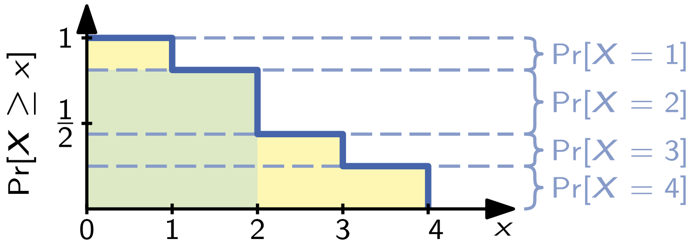
- Random Graphs
- Algorithm Analysis

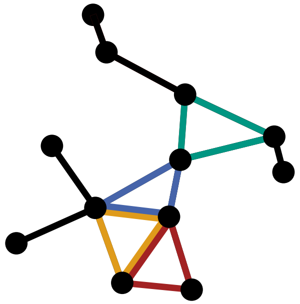
Toolbox

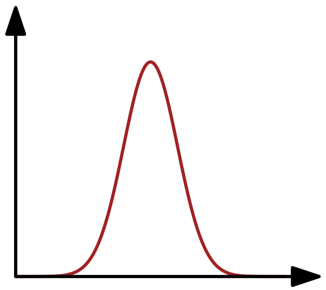
- Probabilistic Method
- Yao’s Principle
- Coupling
- Dealing with stochastic dependencies
- Concentration bounds

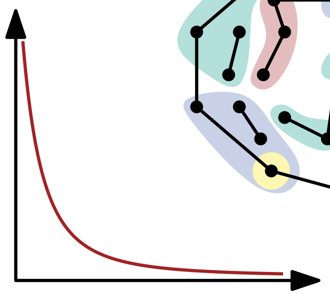


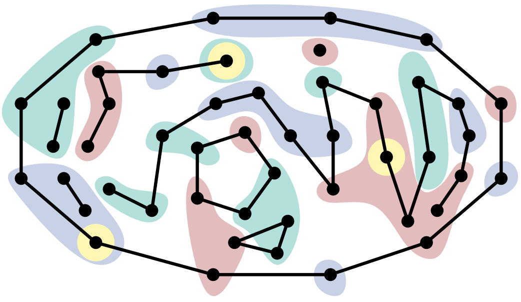












Organization

Team

Max
Lecture
(first part)

Stefan
Lecture
(second part)

Hans-Peter
Exercise

Thursday 11:30

Tuesday 8:00 (every other week)

Organization

Karlsruhe Institute of Technology

Team

Max
Lecture
(first part)

Stefan
Lecture
(second part)

Hans-Peter
Exercise

Thursday 11:30

Tuesday 8:00 (every other week)

Except this week!

Organization

Team

Max
Lecture
(first part)

Stefan
Lecture
(second part)

Hans-Peter
Exercise

Thursday 11:30

Tuesday 8:00 (every other week)

Website scale.iti.kit.edu/teaching/2023ws/randalg

Questions? Ilias, Discord, Matrix?

Organization

Team

Max
Lecture
(first part)

Stefan
Lecture
(second part)

Hans-Peter
Exercise

Thursday 11:30

Tuesday 8:00 (every other week)

Assumed Background

- Algorithms and data structures
- Probability theory

Website scale.iti.kit.edu/teaching/2023ws/randalg

Questions? Ilias, Discord, Matrix?

Organization

Team

Max
Lecture
(first part)

Stefan
Lecture
(second part)

Hans-Peter
Exercise

Thursday 11:30

Tuesday 8:00 (every other week)

Assumed Background

- Algorithms and data structures
- Probability theory

Website scale.iti.kit.edu/teaching/2023ws/randalg

Questions? Ilias, Discord, Matrix?

Sheets

- Every week, hand in on the Thursday before the next exercise

Organization

Team

Max
Lecture
(first part)

Stefan
Lecture
(second part)

Hans-Peter
Exercise

Thursday 11:30

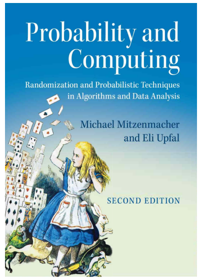
Tuesday 8:00 (every other week)

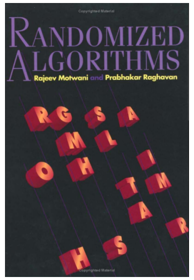
Assumed Background

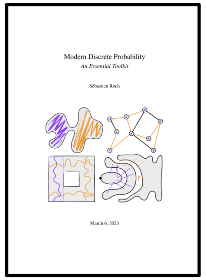
- Algorithms and data structures
- Probability theory

Material

- Slides
- Previous script
- *Probability and Computing*
- *Randomized Algorithms*
- *Modern Discrete Probability*







Website scale.iti.kit.edu/teaching/2023ws/randalg

Questions? Ilias, Discord, Matrix?

Sheets

- Every week, hand in on the Thursday before the next exercise

Organization

Team

Max
Lecture
(first part)

Stefan
Lecture
(second part)

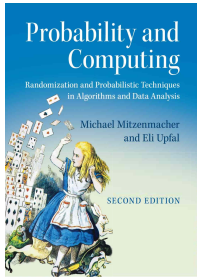
Hans-Peter
Exercise

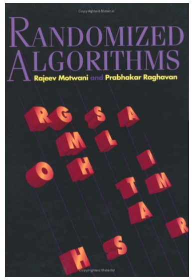
Thursday 11:30

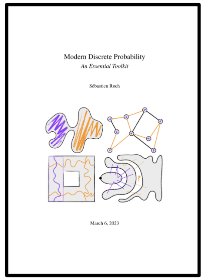
Tuesday 8:00 (every other week)

Assumed Background
- Algorithms and data structures
- Probability theory

Material
- Slides
- Previous script
- *Probability and Computing*
- *Randomized Algorithms*
- *Modern Discrete Probability*







Website
scale.iti.kit.edu/teaching/2023ws/randalg

Questions?
Ilias, Discord, Matrix?

Sheets
- Every week, hand in on the Thursday before the next exercise

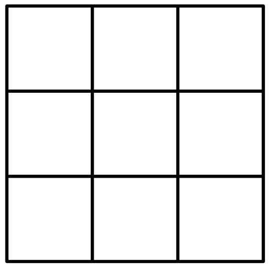
Exam
- Oral
- Requirment: sheets handed in regularly

Power of Randomness: Let's Play a Game

Tic-Tac-Toe

- Players take turns placing ○ and ✕ in 3×3 grid
- First to get three in a line wins

Power of Randomness: Let’s Play a Game



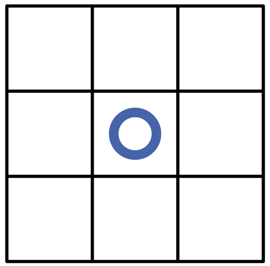
Tic-Tac-Toe

- Players take turns placing O and X in 3×3 grid
- First to get three in a line wins

Power of Randomness: Let's Play a Game

Tic-Tac-Toe

- Players take turns placing ○ and ✕ in 3×3 grid
- First to get three in a line wins

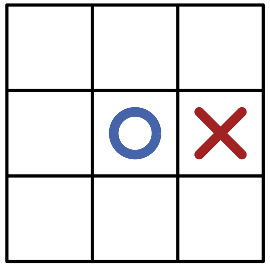


Power of Randomness: Let's Play a Game

Karlsruhe Institute of Technology

Tic-Tac-Toe

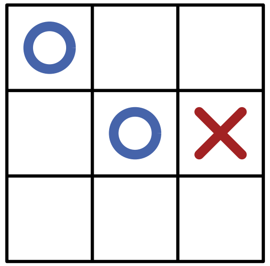
- Players take turns placing O and X in 3×3 grid
- First to get three in a line wins



Power of Randomness: Let's Play a Game

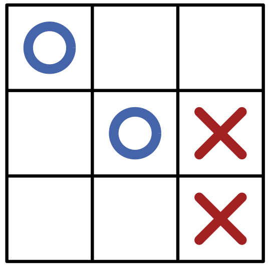
Tic-Tac-Toe

- Players take turns placing O and X in 3×3 grid
- First to get three in a line wins



Power of Randomness: Let's Play a Game

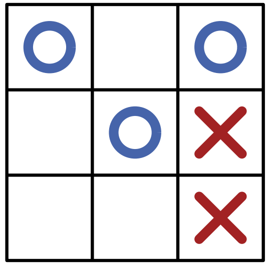
Tic-Tac-Toe



- Players take turns placing O and X in 3×3 grid
- First to get three in a line wins

Power of Randomness: Let's Play a Game

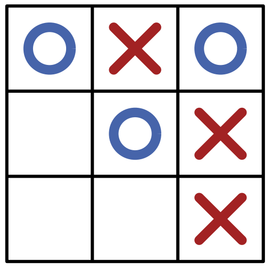
Tic-Tac-Toe



- Players take turns placing O and X in 3×3 grid
- First to get three in a line wins

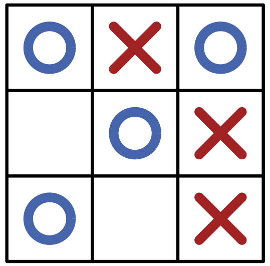
Power of Randomness: Let's Play a Game

Tic-Tac-Toe



- Players take turns placing O and X in 3×3 grid
- First to get three in a line wins

Power of Randomness: Let’s Play a Game



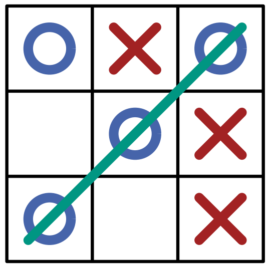
Tic-Tac-Toe

- Players take turns placing O and X in 3×3 grid
- First to get three in a line wins

Power of Randomness: Let's Play a Game

Tic-Tac-Toe

- Players take turns placing O and X in 3×3 grid
- First to get three in a line wins

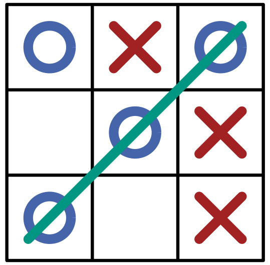


Power of Randomness: Let’s Play a Game

Tic-Tac-Toe

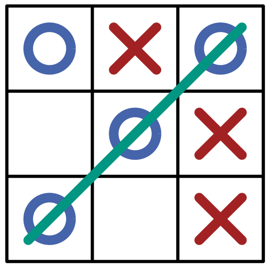
- Players take turns placing ○ and ✕ in 3×3 grid
- First to get three in a line wins

Can Player 2 win the game?



Power of Randomness: Let's Play a Game

Karlsruhe Institute of Technology



Tic-Tac-Toe

- Players take turns placing O and X in 3×3 grid
- First to get three in a line wins

Can Player 2 win the game?

Tree of Moves

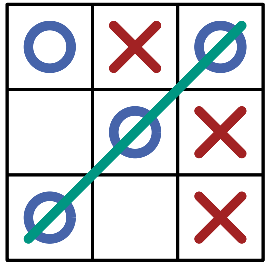
- Each node is a board configuration
- A parent-child relation represents a valid move

Power of Randomness: Let’s Play a Game

Tic-Tac-Toe

- Players take turns placing ○ and ✕ in 3×3 grid
- First to get three in a line wins

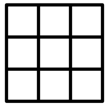
Can Player 2 win the game?



Tree of Moves

- Each node is a board configuration
- A parent-child relation represents a valid move

(initial configuration)



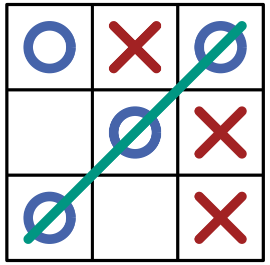
c_0

Power of Randomness: Let's Play a Game

Tic-Tac-Toe

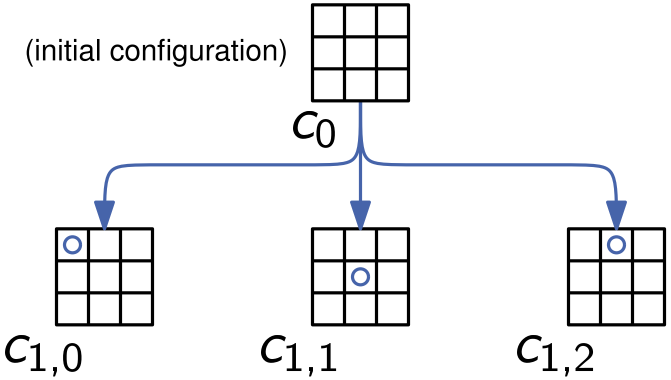
- Players take turns placing O and X in 3×3 grid
- First to get three in a line wins

Can Player 2 win the game?



Tree of Moves

- Each node is a board configuration
- A parent-child relation represents a valid move

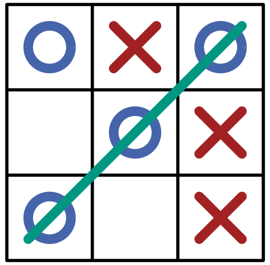


Power of Randomness: Let's Play a Game

Tic-Tac-Toe

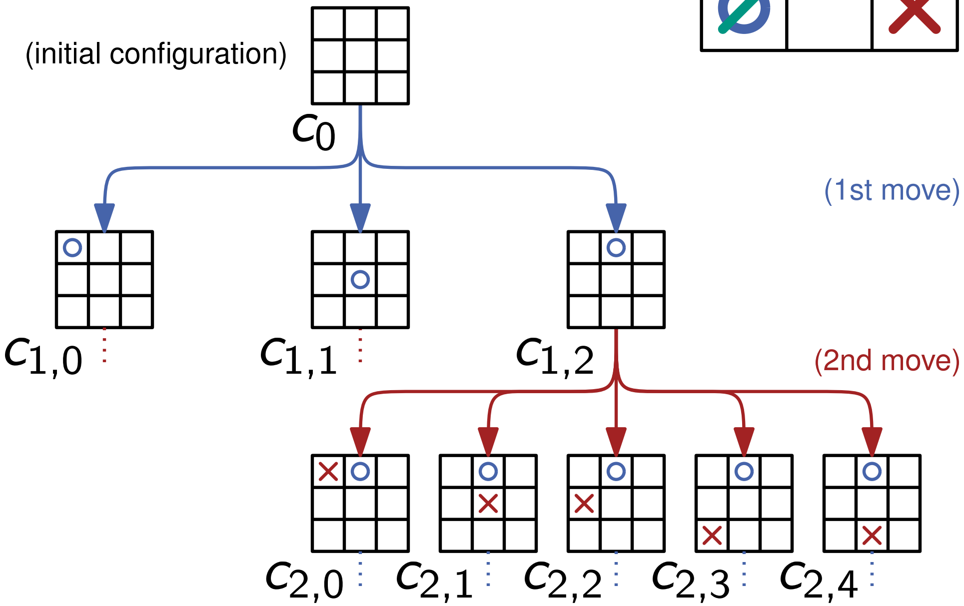
- Players take turns placing ○ and ✕ in 3×3 grid
- First to get three in a line wins

Can Player 2 win the game?



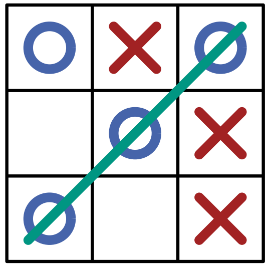
Tree of Moves

- Each node is a board configuration
- A parent-child relation represents a valid move



Institute of Theoretical Informatics, Algorithm Engineering & Scalable Algorithms

Power of Randomness: Let's Play a Game



Tic-Tac-Toe

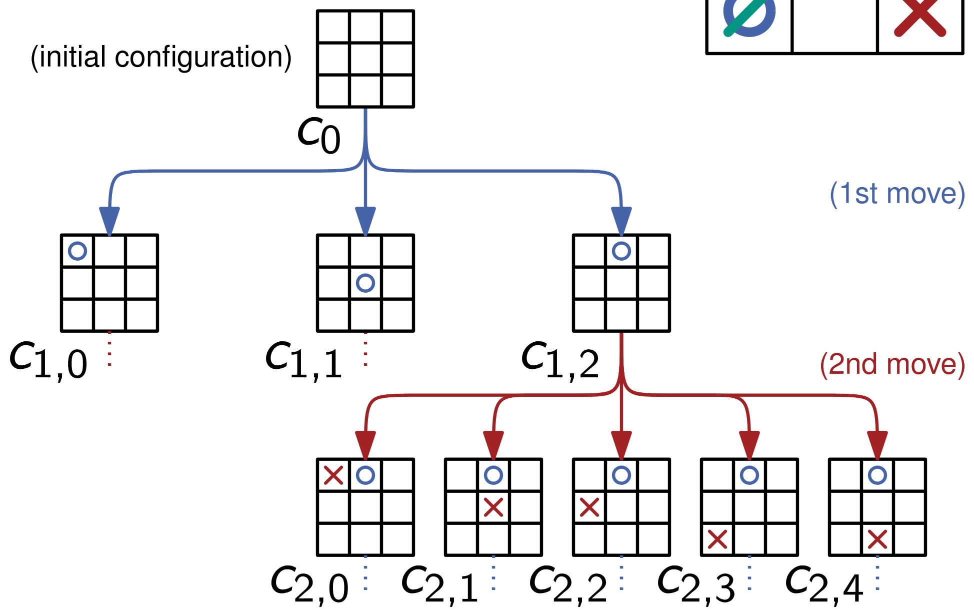
- Players take turns placing O and X in 3×3 grid
- First to get three in a line wins

Can Player 2 win the game?

Tree of Moves

- Each node is a board configuration
- A parent-child relation represents a valid move
- Label a config 1 if Player 2 can win, 0 o.w.

What label do we put on the root?

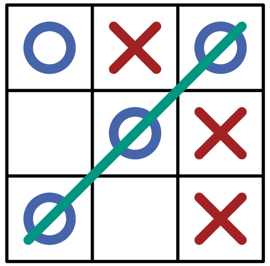


Power of Randomness: Let’s Play a Game

Tic-Tac-Toe

- Players take turns placing O and X in 3×3 grid
- First to get three in a line wins

Can Player 2 win the game?

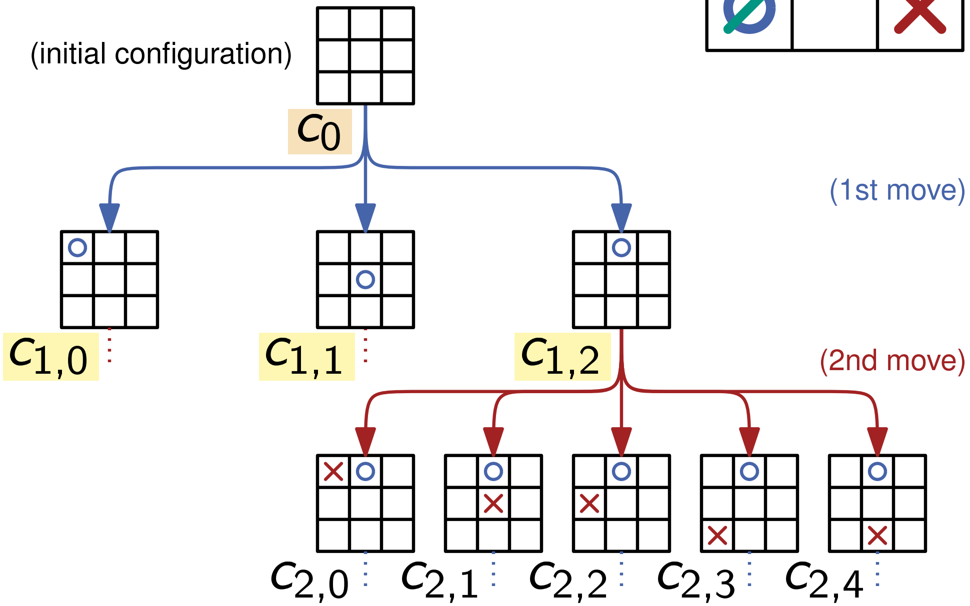


Tree of Moves

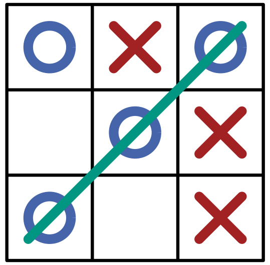
- Each node is a board configuration
- A parent-child relation represents a valid move
- Label a config 1 if Player 2 can win, 0 o.w.

What label do we put on the root?

- $c_0 = 1$ if there exists *no* i such that $c_{1,i} = 0$



Power of Randomness: Let's Play a Game



Tic-Tac-Toe

- Players take turns placing O and X in 3×3 grid
- First to get three in a line wins

Can Player 2 win the game?

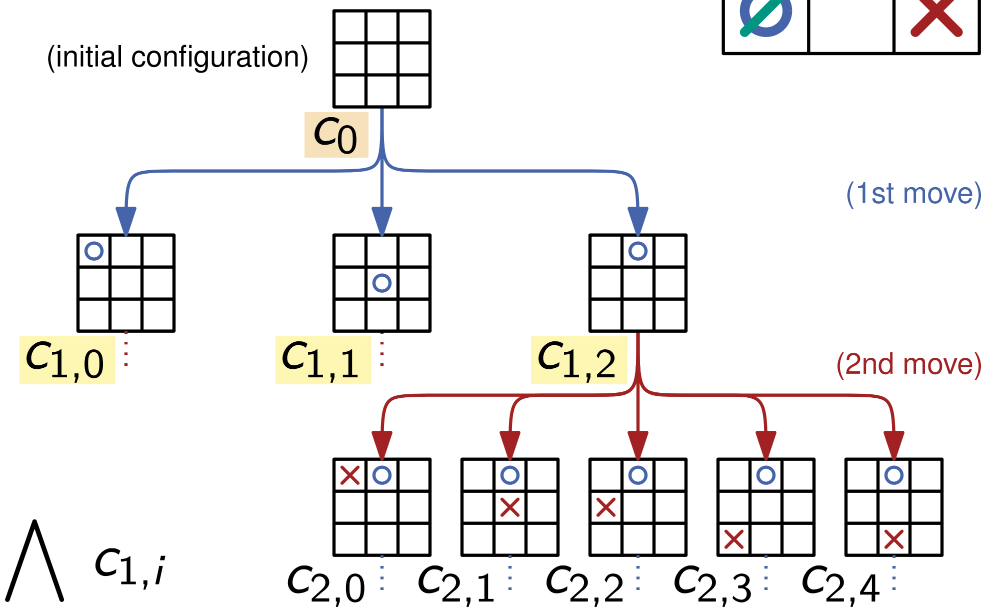
Tree of Moves

- Each node is a board configuration
- A parent-child relation represents a valid move
- Label a config 1 if Player 2 can win, 0 o.w.

What label do we put on the root?

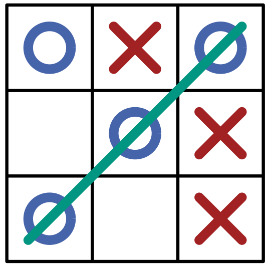
- $c_0 = 1$ if there exists *no* i such that $c_{1,i} = 0$
  or equivalently, if for *all* i we have $c_{1,i} = 1$

$$c_0 = \bigwedge_{i \in [2]} c_{1,i}$$



Power of Randomness: Let's Play a Game

Karlsruhe Institute of Technology



Tic-Tac-Toe

- Players take turns placing O and X in 3×3 grid
- First to get three in a line wins

Can Player 2 win the game?

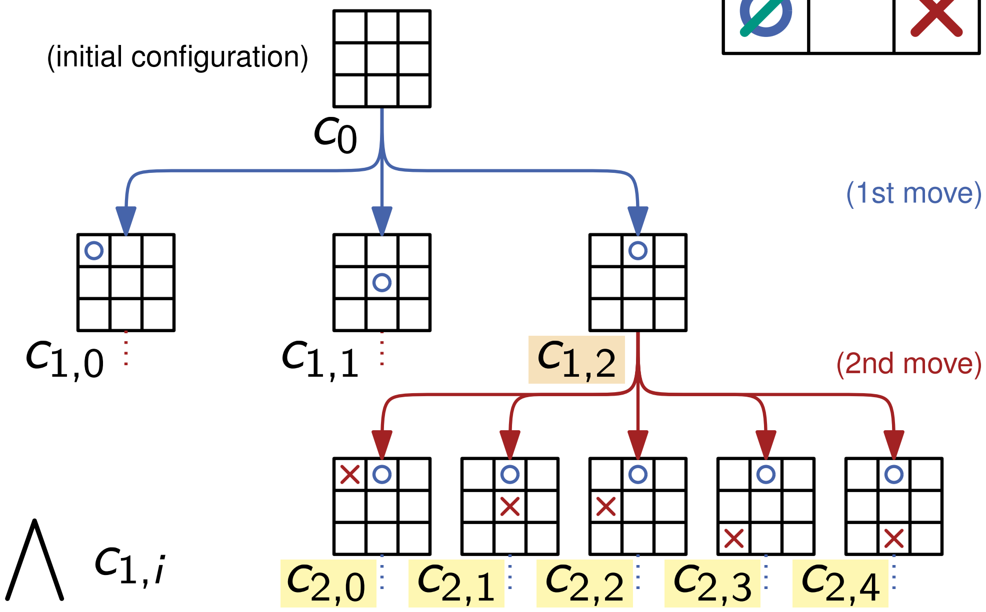
Tree of Moves

- Each node is a board configuration
- A parent-child relation represents a valid move
- Label a config 1 if Player 2 can win, 0 o.w.

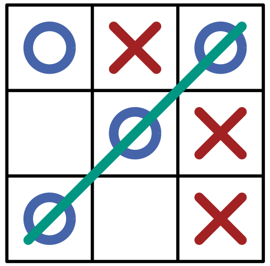
What label do we put on the root?

- $c_0 = 1$ if there exists *no* i such that $c_{1,i} = 0$
  or equivalently, if for *all* i we have $c_{1,i} = 1$
- $c_{1,2} = 1$ if there exists an i such that $c_{2,i} = 1$

$$c_0 = \bigwedge_{i \in [2]} c_{1,i}$$



Power of Randomness: Let's Play a Game



Tic-Tac-Toe

- Players take turns placing ○ and ✕ in 3×3 grid
- First to get three in a line wins

Can Player 2 win the game?

Tree of Moves

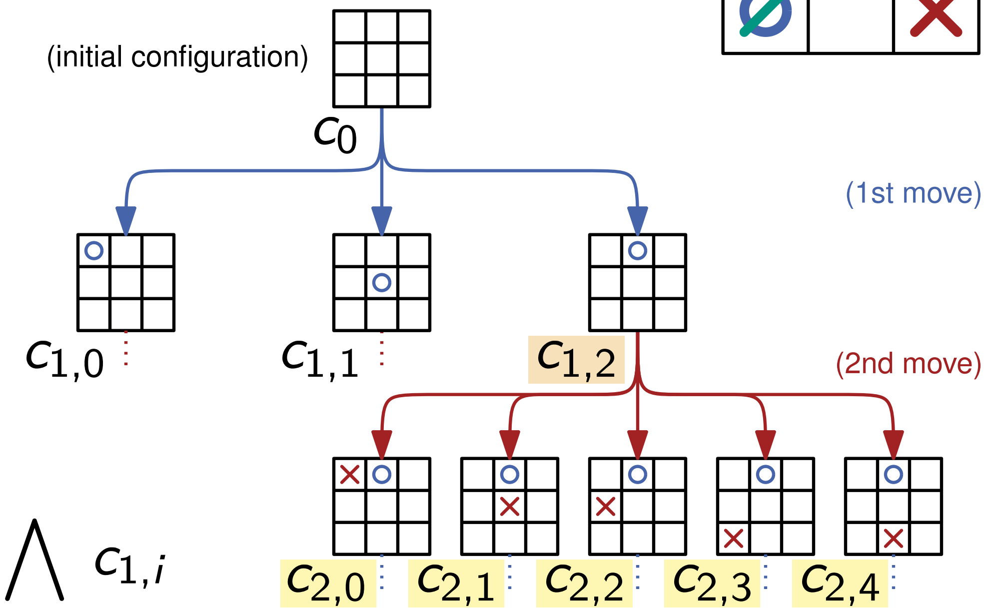
- Each node is a board configuration
- A parent-child relation represents a valid move
- Label a config 1 if Player 2 can win, 0 o.w.

What label do we put on the root?

- $c_0 = 1$ if there exists *no* i such that $c_{1,i} = 0$
  or equivalently, if for *all* i we have $c_{1,i} = 1$
- $c_{1,2} = 1$ if there exists an i such that $c_{2,i} = 1$

$$c_0 = \bigwedge_{i \in [2]} c_{1,i}$$

$$c_0 = \bigvee_{i \in [4]} c_{2,i}$$



AND/OR-Trees

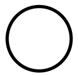
Structure

- Node types: $\wedge$-nodes, $\vee$-nodes, and leaves

AND/OR-Trees

Structure

- Node types: $\wedge$-nodes, $\vee$-nodes, and leaves
- The root is a leaf



AND/OR-Trees

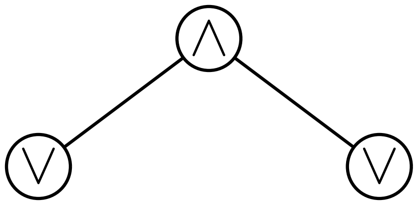
Structure

- Node types: $\wedge$-nodes, $\vee$-nodes, and leaves
- The root is a leaf or an $\wedge$-node

AND/OR-Trees

Structure

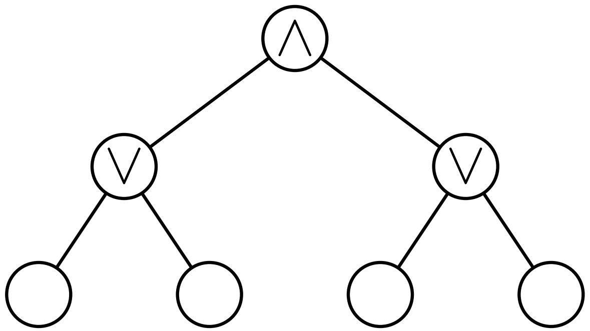
- Node types: $\wedge$-nodes, $\vee$-nodes, and leaves
- The root is a leaf or an $\wedge$-node
- $\wedge$-nodes have only $\vee$-nodes as children



AND/OR-Trees

Structure

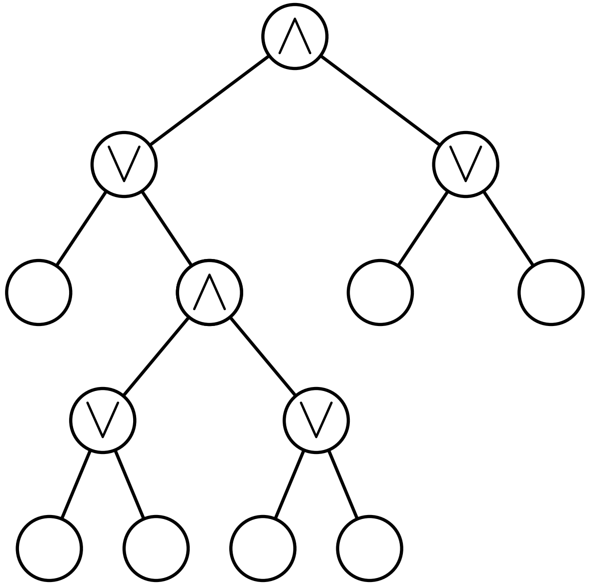
- Node types: $\wedge$-nodes, $\vee$-nodes, and leaves
- The root is a leaf or an $\wedge$-node
- $\wedge$-nodes have only $\vee$-nodes as children
- $\vee$-nodes have only AND/OR-trees as children



AND/OR-Trees

Structure

- Node types: $\wedge$-nodes, $\vee$-nodes, and leaves
- The root is a leaf or an $\wedge$-node
- $\wedge$-nodes have only $\vee$-nodes as children
- $\vee$-nodes have only AND/OR-trees as children



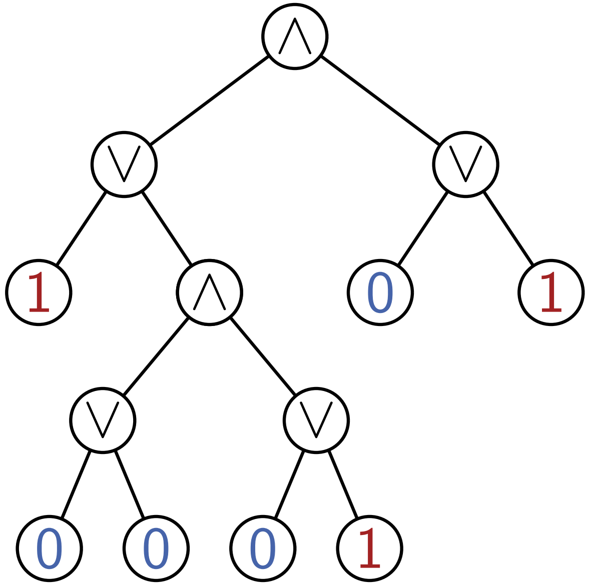
AND/OR-Trees

Structure

- Node types: $\wedge$-nodes, $\vee$-nodes, and leaves
- The root is a leaf or an $\wedge$-node
- $\wedge$-nodes have only $\vee$-nodes as children
- $\vee$-nodes have only AND/OR-trees as children

Evaluation

- Leaves contain boolean values



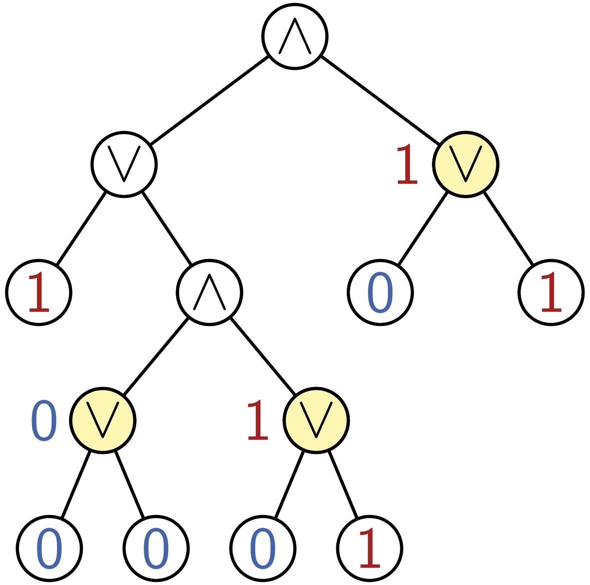
AND/OR-Trees

Structure

- Node types: $\wedge$-nodes, $\vee$-nodes, and leaves
- The root is a leaf or an $\wedge$-node
- $\wedge$-nodes have only $\vee$-nodes as children
- $\vee$-nodes have only AND/OR-trees as children

Evaluation

- Leaves contain boolean values
- Inner nodes evaluate to . . .
  - the disjunction of their children, for $\vee$-nodes



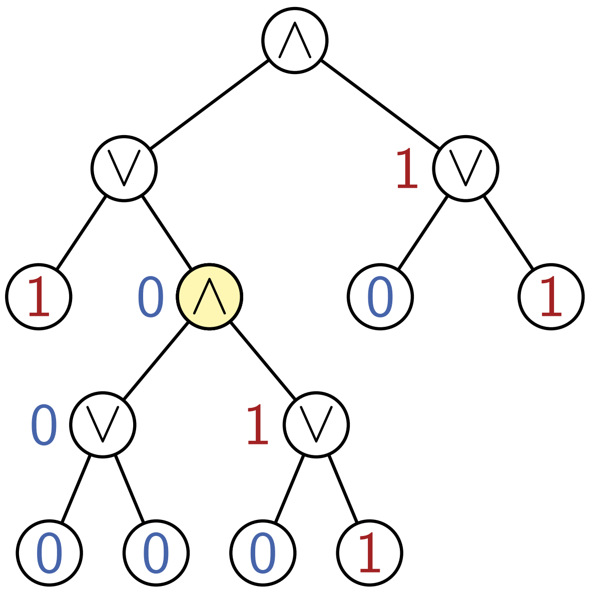
AND/OR-Trees

Structure

- Node types: $\wedge$-nodes, $\vee$-nodes, and leaves
- The root is a leaf or an $\wedge$-node
- $\wedge$-nodes have only $\vee$-nodes as children
- $\vee$-nodes have only AND/OR-trees as children

Evaluation

- Leaves contain boolean values
- Inner nodes evaluate to ...
  - the disjunction of their children, for $\vee$-nodes
  - the conjunction of their children, for $\wedge$-nodes



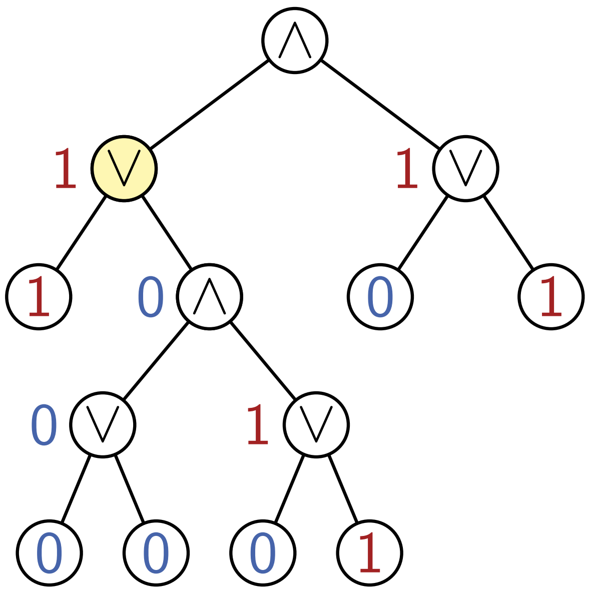
AND/OR-Trees

Structure

- Node types: ∧-nodes, ∨-nodes, and leaves
- The root is a leaf or an ∧-node
- ∧-nodes have only ∨-nodes as children
- ∨-nodes have only AND/OR-trees as children

Evaluation

- Leaves contain boolean values
- Inner nodes evaluate to . . .
  - the disjunction of their children, for ∨-nodes
  - the conjunction of their children, for ∧-nodes



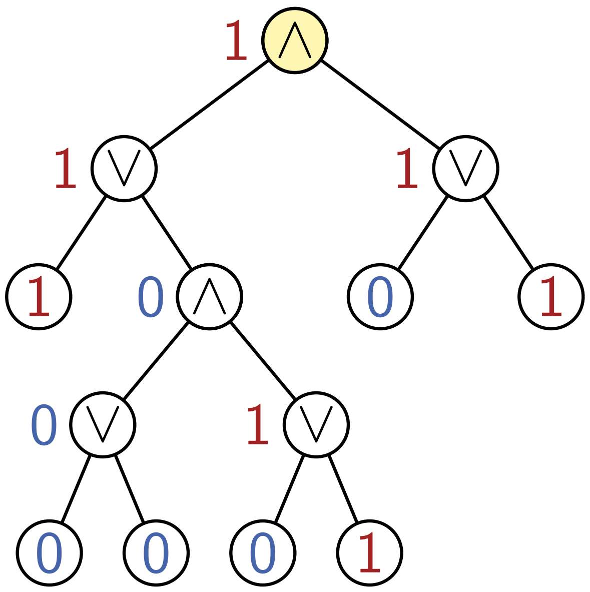
AND/OR-Trees

Structure

- Node types: $\wedge$-nodes, $\vee$-nodes, and leaves
- The root is a leaf or an $\wedge$-node
- $\wedge$-nodes have only $\vee$-nodes as children
- $\vee$-nodes have only AND/OR-trees as children

Evaluation

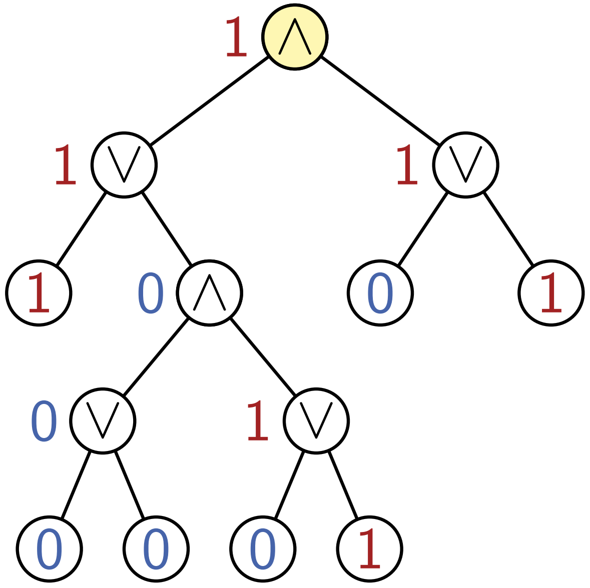
- Leaves contain boolean values
- Inner nodes evaluate to . . .
  - the disjunction of their children, for $\vee$-nodes
  - the conjunction of their children, for $\wedge$-nodes



AND/OR-Trees

Structure

- Node types: $\wedge$-nodes, $\vee$-nodes, and leaves
- The root is a leaf or an $\wedge$-node
- $\wedge$-nodes have only $\vee$-nodes as children
- $\vee$-nodes have only AND/OR-trees as children



Evaluation

- Leaves contain boolean values
- Inner nodes evaluate to . . .
  - the disjunction of their children, for $\vee$-nodes
  - the conjunction of their children, for $\wedge$-nodes

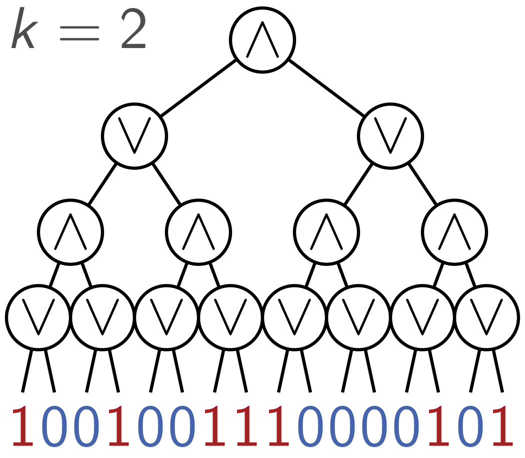
Example Complexities

- Tic-Tac-Toe: 31896 (non-symmetric) games (leaves)
- Checkers: approx. 10^{40} leaves
- Chess: approx. 10^{123} leaves
- Go (19×19): approx. 10^{360} leaves

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

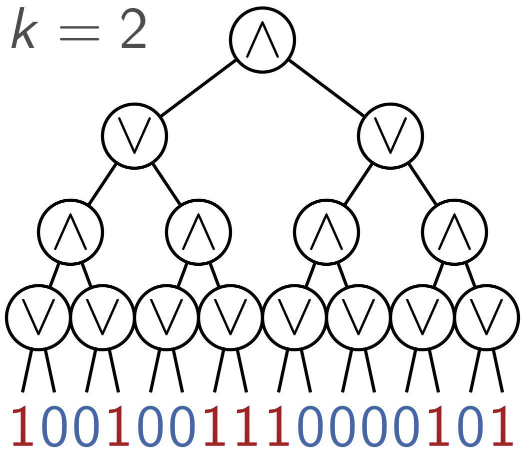


Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

  $\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

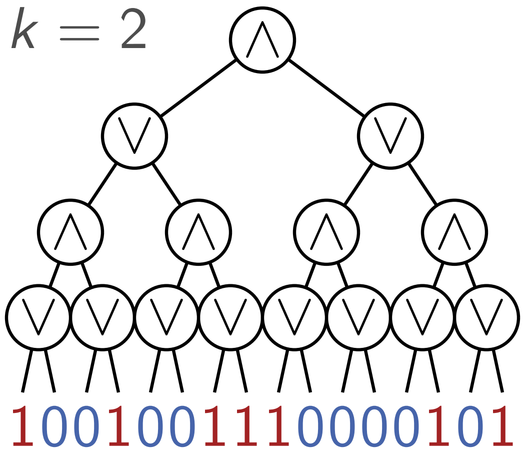


Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely



A Simple Deterministic Algorithm

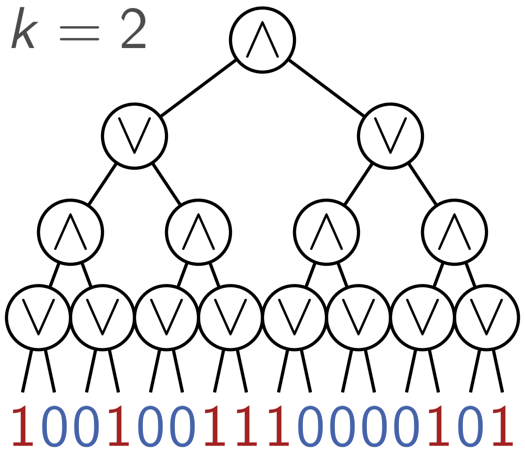
- Compute all nodes bottom up

Deterministic Evaluation

Simplifying Assumption
- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm
- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ



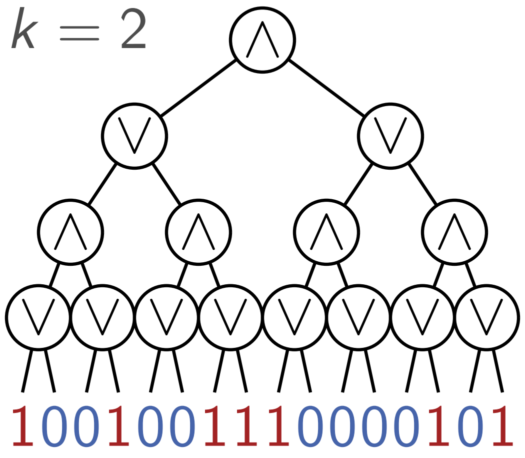
Deterministic Evaluation

Karlsruhe Institute of Technology

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely



A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

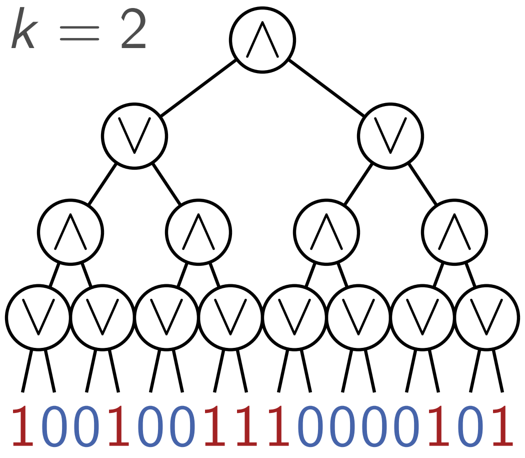
$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

Can we do better?

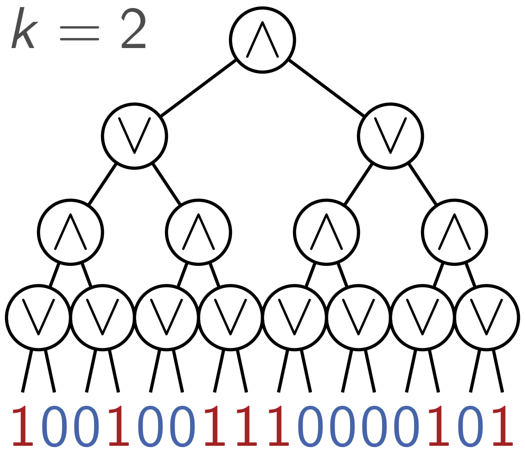


Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely



A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

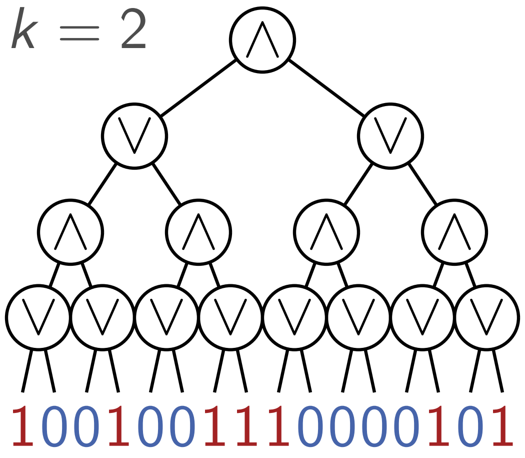
Can we do better? **NO!**

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely



A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

Can we do better? **NO!**

Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Karlsruhe Institute of Technology

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

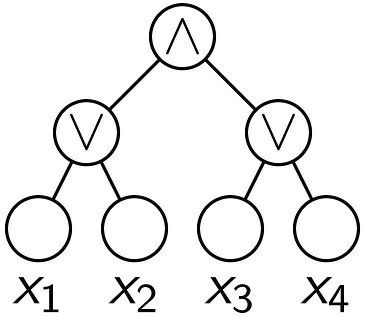
$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

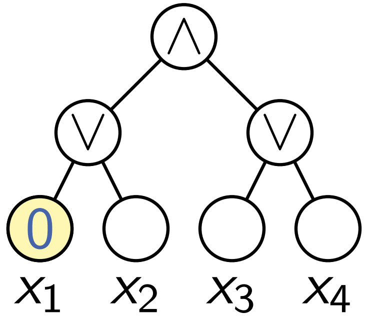
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \to x_1$
- Set $x_1 := 0$



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Karlsruhe Institute of Technology

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

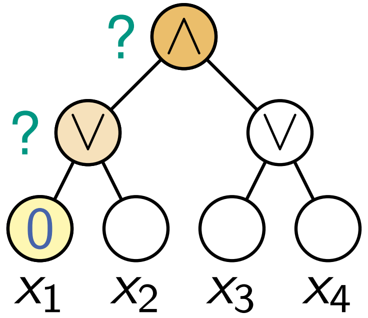
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \to x_1$
- Set $x_1 := 0$ (value of parent and root *not* determined, yet)



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

  $\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

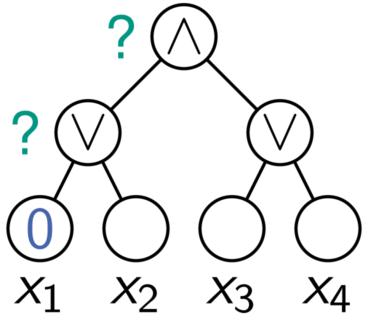
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \rightarrow x_1$
- Set $x_1 := 0$ (value of parent and root *not* determined, yet)
- A needs to visit another leaf



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

  $\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

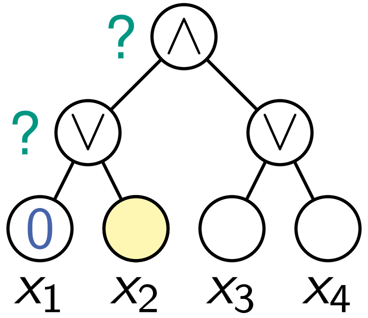
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \to x_1$
- Set $x_1 := 0$ (value of parent and root *not* determined, yet)
- A needs to visit another leaf
- Case 1: $A \to x_2$



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

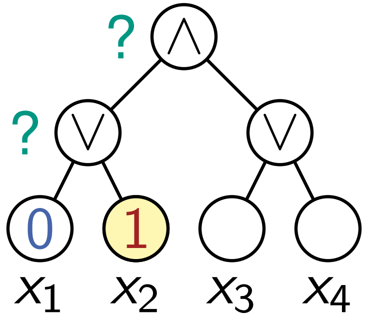
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \to x_1$
- Set $x_1 := 0$ (value of parent and root *not* determined, yet)
- A needs to visit another leaf
- Case 1: $A \to x_2$
  - $x_1 := 1$



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Karlsruhe Institute of Technology

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

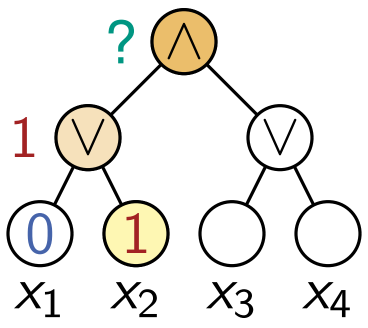
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \rightarrow x_1$
- Set $x_1 := 0$ (value of parent and root *not* determined, yet)
- A needs to visit another leaf
- Case 1: $A \rightarrow x_2$
  - $x_1 := 1$ (value of parent determined, but not of root)



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

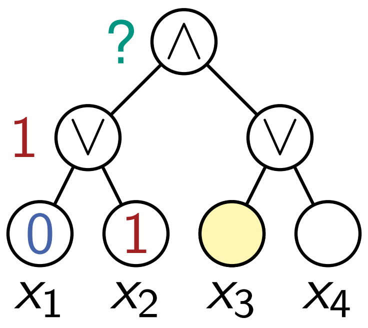
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \rightarrow x_1$
- Set $x_1 := 0$ (value of parent and root *not* determined, yet)
- A needs to visit another leaf
- Case 1: $A \rightarrow x_2$
  - $x_1 := 1$ (value of parent determined, but not of root)
  - w.l.o.g. $A \rightarrow x_3$



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Karlsruhe Institute of Technology

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

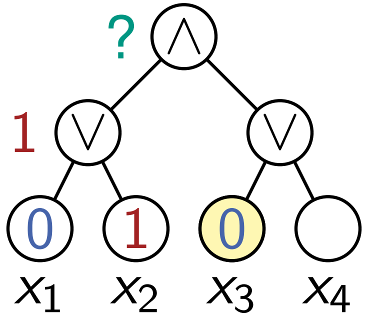
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \rightarrow x_1$
- Set $x_1 := 0$ (value of parent and root *not* determined, yet)
- A needs to visit another leaf
- Case 1: $A \rightarrow x_2$
  - $x_1 := 1$ (value of parent determined, but not of root)
  - w.l.o.g. $A \rightarrow x_3$
  - $x_3 := 0$



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

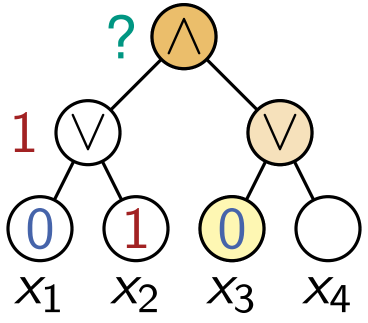
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \to x_1$
- Set $x_1 := 0$ (value of parent and root *not* determined, yet)
- A needs to visit another leaf
- Case 1: $A \to x_2$
  - $x_1 := 1$ (value of parent determined, but not of root)
  - w.l.o.g. $A \to x_3$
  - $x_3 := 0$ (value of parent and root *not* determined, yet)



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

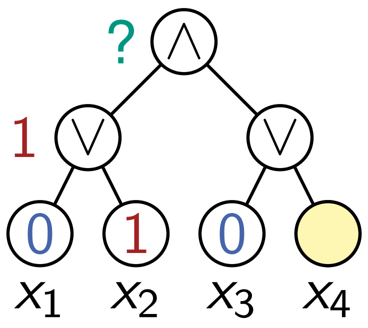
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \to x_1$
- Set $x_1 := 0$ (value of parent and root *not* determined, yet)
- A needs to visit another leaf
- Case 1: $A \to x_2$
  - $x_1 := 1$ (value of parent determined, but not of root)
  - w.l.o.g. $A \to x_3$
  - $x_3 := 0$ (value of parent and root *not* determined, yet)



$\Rightarrow A \to x_4$

$\Rightarrow$ output is x_4 ✓

Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

  $\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

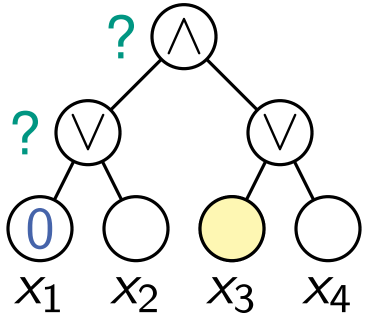
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \to x_1$
- Set $x_1 := 0$ (value of parent and root *not* determined, yet)
- A needs to visit another leaf
- Case 2: $A \to x_3$



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

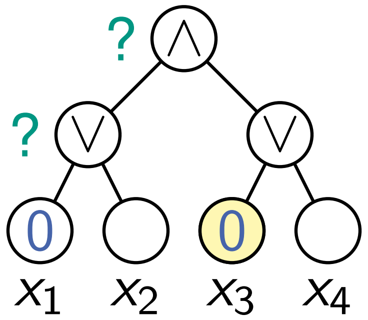
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \to x_1$
- Set $x_1 := 0$ (value of parent and root *not* determined, yet)
- A needs to visit another leaf
- Case 2: $A \to x_3$
  - $x_3 := 0$



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

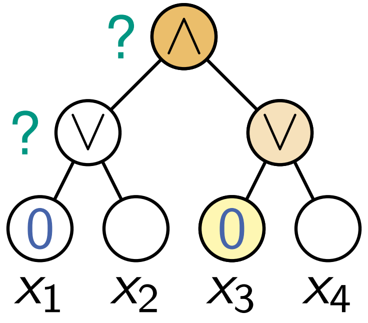
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \to x_1$
- Set $x_1 := 0$ (value of parent and root *not* determined, yet)
- A needs to visit another leaf
- Case 2: $A \to x_3$
  - $x_3 := 0$ (value of parent and root *not* determined, yet)



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Karlsruhe Institute of Technology

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

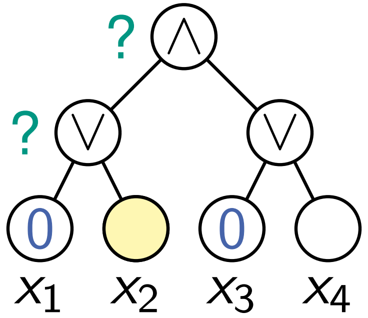
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \to x_1$
- Set $x_1 := 0$ (value of parent and root *not* determined, yet)
- A needs to visit another leaf
- Case 2: $A \to x_3$
  - $x_3 := 0$ (value of parent and root *not* determined, yet)
  - w.l.o.g. $A \to x_2$



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

  $\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

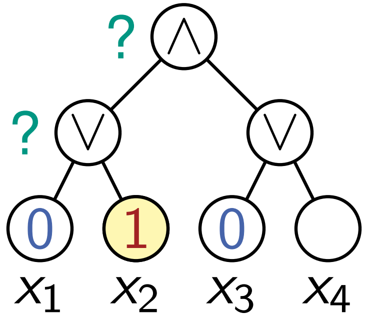
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \to x_1$
- Set $x_1 := 0$ (value of parent and root *not* determined, yet)
- A needs to visit another leaf
- Case 2: $A \to x_3$
  - $x_3 := 0$ (value of parent and root *not* determined, yet)
  - w.l.o.g. $A \to x_2$
  - $x_2 := 1$



> **Theorem**: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

  $\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

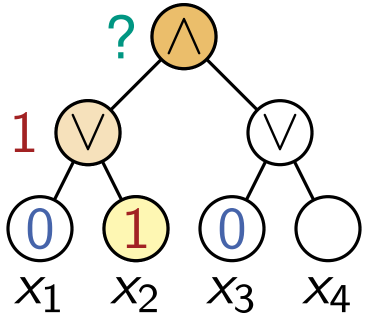
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \to x_1$
- Set $x_1 := 0$ (value of parent and root *not* determined, yet)
- A needs to visit another leaf
- Case 2: $A \to x_3$
  - $x_3 := 0$ (value of parent and root *not* determined, yet)
  - w.l.o.g. $A \to x_2$
  - $x_2 := 1$ (value of parent determined, but not of root)



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

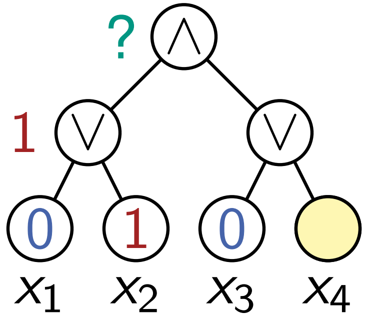
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Base: $k = 1$

- A visits ≥ 1 leaf: w.l.o.g. $A \to x_1$
- Set $x_1 := 0$ (value of parent and root *not* determined, yet)
- A needs to visit another leaf
- Case 2: $A \to x_3$
  - $x_3 := 0$ (value of parent and root *not* determined, yet)
  - w.l.o.g. $A \to x_2$
  - $x_2 := 1$ (value of parent determined, but not of root)



$\Rightarrow A \to x_4$

$\Rightarrow$ output is x_4 ✓

Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

  $\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

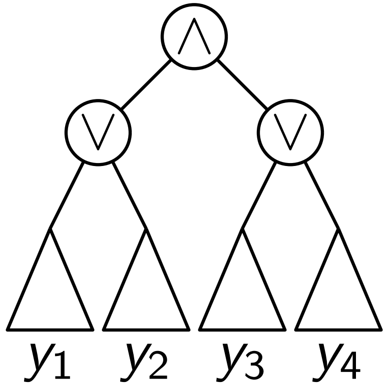
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Step: $k - 1 \to k$

- Consider tree of depth $2k$ as a tree of depth 2 with trees $y_1, \ldots, y_4$ (of depth $2(k-1)$) as "leaves"



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

  $\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

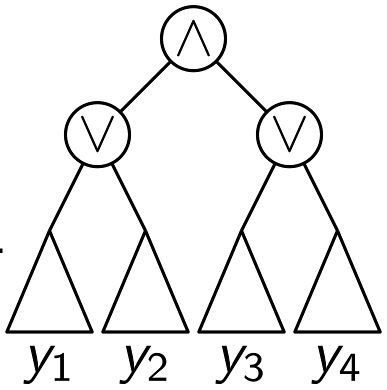
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Step: $k - 1 \to k$

- Consider tree of depth $2k$ as a tree of depth 2 with trees $y_1, \dots, y_4$ (of depth $2(k-1)$) as “leaves”
- Analogous to the base, we can enforce that A needs to look at all y_i



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \dots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

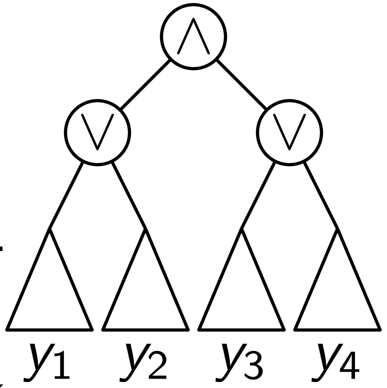
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Step: $k - 1 \rightarrow k$

- Consider tree of depth $2k$ as a tree of depth 2 with trees $y_1, \ldots, y_4$ (of depth $2(k-1)$) as "leaves"
- Analogous to the base, we can enforce that A needs to look at all y_i
- By induction, we can force A to look at all leaves in each y_i



Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Deterministic Evaluation

Simplifying Assumption

- Each inner node has two children
- All leaves have the same depth $2k$

$\Rightarrow$ A bit-string of length $n = 4^k$ encodes the input completely

A Simple Deterministic Algorithm

- Compute all nodes bottom up
- Running time on layer ℓ: 2^ℓ

$$\sum_{\ell=0}^{2k} 2^\ell = 2^{2k+1} - 1 = \Theta(4^k) = \Theta(n)$$

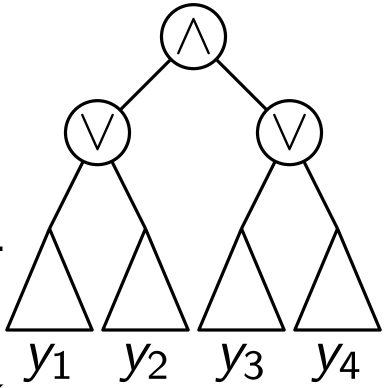
Can we do better? **NO!**

Proof via Induction

- Idea: We are an adversary who knows A and constructs an input (...on the fly, while the algorithm is running. Since A is deterministic this does not make a difference.)

Step: $k-1 \to k$

- Consider tree of depth $2k$ as a tree of depth 2 with trees $y_1, \ldots, y_4$ (of depth $2(k-1)$) as "leaves"
- Analogous to the base, we can enforce that A needs to look at all y_i
- By induction, we can force A to look at all leaves in each y_i



$\Rightarrow$ A looks at all leaves ✓

Theorem: Let A be any deterministic AND/OR-tree-algorithm. For $k \geq 1$ there exists an input $x_1, \ldots, x_{4^k}$ s.t. A visits all 4^k leaves and the output is the value of the last one visited.

Randomized Evaluation

Karlsruhe Institute of Technology

Idea

- We can evaluate an $\wedge$-node to 0 if we find *one* 0-child
- We can evaluate an $\vee$-node to 1 if we find *one* 1-child

} while ignoring the other child!

Randomized Evaluation

Idea

- We can evaluate an $\wedge$-node to 0 if we find *one* 0-child
- We can evaluate an $\vee$-node to 1 if we find *one* 1-child

} while ignoring the other child!

Algorithm

evalAndNode(v)
- **if** v is leaf **then**
  - **return value**(v)
- $c :=$ **uniformSample**(v.**children**)
- **if evalOrNode**(c) $= 0$ **then**
  - **return** 0
- $c' :=$ the other child
- **return evalOrNode**(c')

Randomized Evaluation

Idea

- We can evaluate an $\wedge$-node to 0 if we find *one* 0-child
- We can evaluate an $\vee$-node to 1 if we find *one* 1-child

} while ignoring the other child!

Algorithm

```
evalAndNode(v)
  if v is leaf then
    return value(v)
  c := uniformSample(v.children)
  if evalOrNode(c) = 0 then
    return 0
  c' := the other child
  return evalOrNode(c')
```

Here each of the two children is selected with equal probability $1/2$.

Randomized Evaluation

Idea

- We can evaluate an $\wedge$-node to 0 if we find *one* 0-child
- We can evaluate an $\vee$-node to 1 if we find *one* 1-child

} while ignoring the other child!

Algorithm

Here each of the two children is selected with equal probability $1/2$.

$\vee$-nodes are not leaves in our setting

```
evalAndNode(v)
    if v is leaf then
        return value(v)
    c := uniformSample(v.children)
    if evalOrNode(c) = 0 then
        return 0
    c' := the other child
    return evalOrNode(c')

evalOrNode(v)
    c := uniformSample(v.children)
    if evalAndNode(c) = 1 then
        return 1
    c' := the other child
    return evalAndNode(c')
```

Randomized Evaluation

Karlsruhe Institute of Technology

Idea

- We can evaluate an $\wedge$-node to 0 if we find *one* 0-child
- We can evaluate an $\vee$-node to 1 if we find *one* 1-child

} while ignoring the other child!

Algorithm

```
evalAndNode(v)
  if v is leaf then
    return value(v)
  c := uniformSample(v.children)
  if evalOrNode(c) = 0 then
    return 0
  c' := the other child
  return evalOrNode(c')
```

Here each of the two children is selected with equal probability $1/2$.

```
evalOrNode(v)
  c := uniformSample(v.children)
  if evalAndNode(c) = 1 then
    return 1
  c' := the other child
  return evalAndNode(c')
```

$\vee$-nodes are not leaves in our setting

- Execute as evalAndNode(r) for root-node r

Randomized Evaluation

Karlsruhe Institute of Technology

Idea

- We can evaluate an $\wedge$-node to 0 if we find *one* 0-child
- We can evaluate an $\vee$-node to 1 if we find *one* 1-child

} while ignoring the other child!

Algorithm

```
evalAndNode(v)
  if v is leaf then
    return value(v)
  c := uniformSample(v.children)
  if evalOrNode(c) = 0 then
    return 0
  c' := the other child
  return evalOrNode(c')
```

Here each of the two children is selected with equal probability $1/2$.

```
evalOrNode(v)
  c := uniformSample(v.children)
  if evalAndNode(c) = 1 then
    return 1
  c' := the other child
  return evalAndNode(c')
```

$\vee$-nodes are not leaves in our setting

- Execute as evalAndNode(r) for root-node r

How long does that take?

Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child

Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$.

Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Randomized Evaluation – Running Time

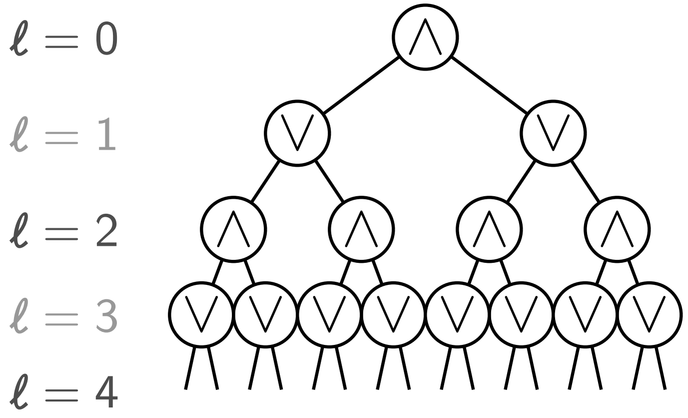
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction

(that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

- Expected number of nodes evaluated on *even* layer $\ell = 2i$ is at most 3^i



Randomized Evaluation – Running Time

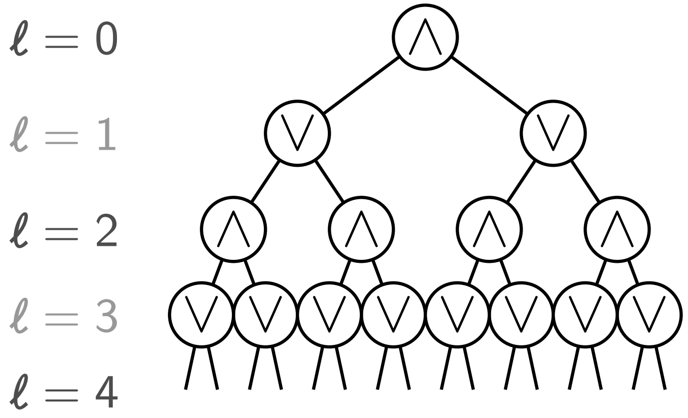
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction

(that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

- Expected number of nodes evaluated on *even* layer $\ell = 2i$ is at most 3^i
- Expected number of nodes evaluated on *odd* layer ℓ is at most that of the layer beneath



Randomized Evaluation – Running Time

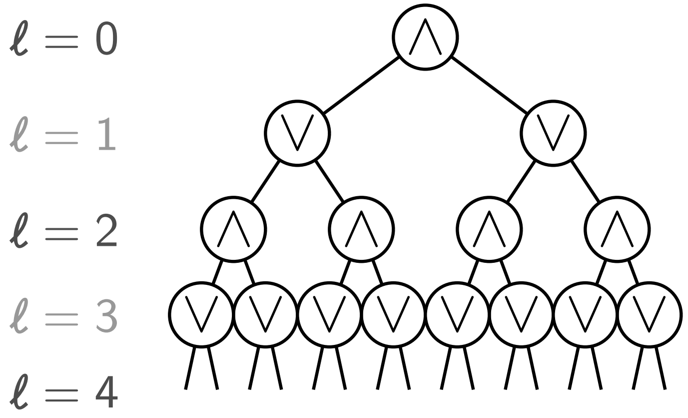
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction

(that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

- Expected number of nodes evaluated on *even* layer $\ell = 2i$ is at most 3^i
- Expected number of nodes evaluated on *odd* layer ℓ is at most that of the layer beneath
- Expected number of total evaluated nodes is at most



Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

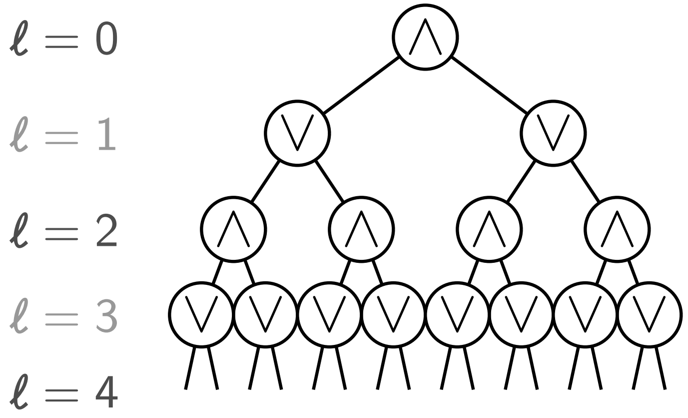
Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction

(that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

- Expected number of nodes evaluated on *even* layer $\ell = 2i$ is at most 3^i
- Expected number of nodes evaluated on *odd* layer ℓ is at most that of the layer beneath
- Expected number of total evaluated nodes is at most

$$\underbrace{\overbrace{3^0}^{\ell=0}}_{i=0} + \overbrace{3^1}^{\ell=1} + \underbrace{\overbrace{3^1}^{\ell=2}}_{i=1} + \overbrace{3^2}^{\ell=3} + \underbrace{\overbrace{3^2}^{\ell=4}}_{i=2} + \cdots + \underbrace{\overbrace{3^k}^{\ell=2k}}_{i=k}$$



Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

Proof via Induction

(that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

- Expected number of nodes evaluated on *even* layer $\ell = 2i$ is at most 3^i
- Expected number of nodes evaluated on *odd* layer ℓ is at most that of the layer beneath
- Expected number of total evaluated nodes is at most

$$\underbrace{\overbrace{3^0}^{\ell=0}}_{i=0} + \overbrace{3^1}^{\ell=1} + \underbrace{\overbrace{3^1}^{\ell=2}}_{i=1} + \overbrace{3^2}^{\ell=3} + \underbrace{\overbrace{3^2}^{\ell=4}}_{i=2} + \dots + \underbrace{\overbrace{3^k}^{\ell=2k}}_{i=k} \leq \sum_{i=0}^{k} 2 \cdot 3^i = \Theta(3^k)$$

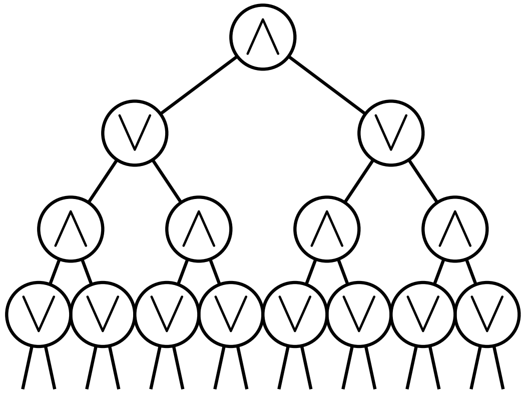
$\ell = 0$

$\ell = 1$

$\ell = 2$

$\ell = 3$

$\ell = 4$



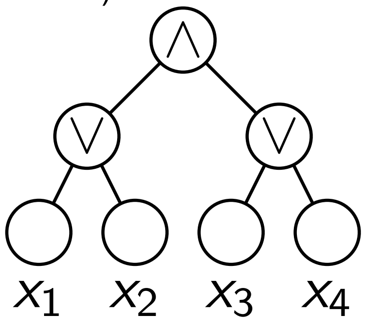
Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Base: $k = 1$



Randomized Evaluation – Running Time

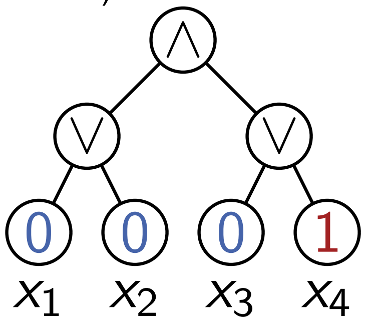
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Base: $k = 1$

- Case analysis over all bit-strings x_1, x_2, x_3, x_4, example 0001



Randomized Evaluation – Running Time

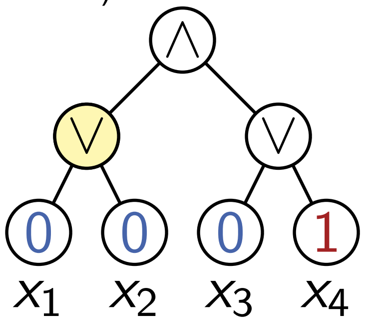
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Base: $k = 1$

- Case analysis over all bit-strings x_1, x_2, x_3, x_4, example 0001
- Let X_L be number of leaves visited when going left first



Randomized Evaluation – Running Time

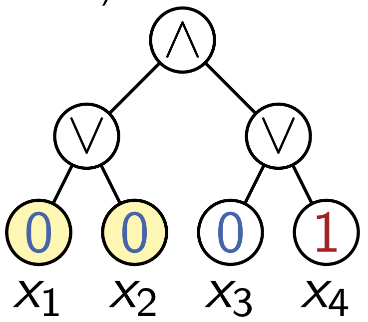
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Base: $k = 1$

- Case analysis over all bit-strings x_1, x_2, x_3, x_4, example 0001
- Let X_L be number of leaves visited when going left first
  - Independent of leaf choice, need to look at other too: $X_L = 2$



Randomized Evaluation – Running Time

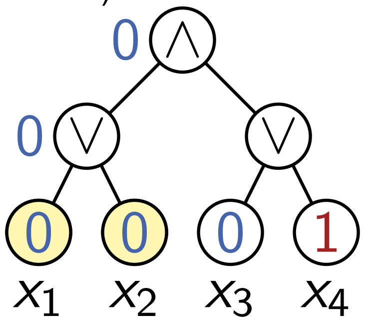
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Base: $k = 1$

- Case analysis over all bit-strings x_1, x_2, x_3, x_4, example 0001
- Let X_L be number of leaves visited when going left first
  - Independent of leaf choice, need to look at other too: $X_L = 2$
  - When left $\vee$-node is checked, root value is determined



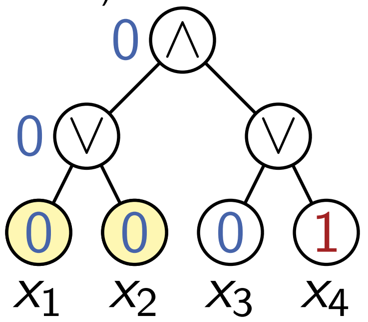
Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Base: $k = 1$



- Case analysis over all bit-strings x_1, x_2, x_3, x_4, example 0001
- Let X_L be number of leaves visited when going left first
  - Independent of leaf choice, need to look at other too: $X_L = 2$
  - When left $\vee$-node is checked, root value is determined

$\mathbb{E}[X_L] = 2$

Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

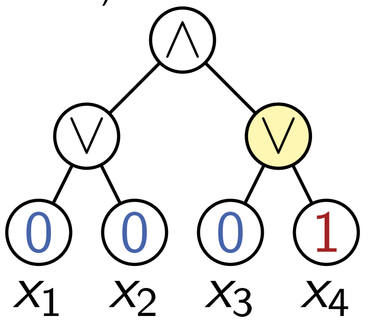
Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Base: $k = 1$

- Case analysis over all bit-strings x_1, x_2, x_3, x_4, example 0001
- Let X_L be number of leaves visited when going left first
  - Independent of leaf choice, need to look at other too: $X_L = 2$
  - When left $\vee$-node is checked, root value is determined
- Let X_R be number of leaves visited when going right first

$\mathbb{E}[X_L] = 2$



Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

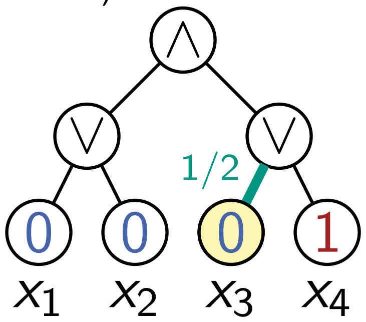
Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Base: $k = 1$

- Case analysis over all bit-strings x_1, x_2, x_3, x_4, example 0001
- Let X_L be number of leaves visited when going left first
  - Independent of leaf choice, need to look at other too: $X_L = 2$
  - When left $\vee$-node is checked, root value is determined

$\mathbb{E}[X_L] = 2$

- Let X_R be number of leaves visited when going right first
  - $\Pr[\text{RE} \to x_3] = 1/2$



Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

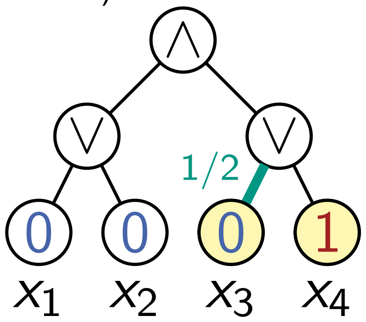
Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Base: $k = 1$

- Case analysis over all bit-strings x_1, x_2, x_3, x_4, example 0001
- Let X_L be number of leaves visited when going left first
  - Independent of leaf choice, need to look at other too: $X_L = 2$
  - When left $\vee$-node is checked, root value is determined
- Let X_R be number of leaves visited when going right first
  - $\Pr[\text{RE} \to x_3] = 1/2$ → visit x_4

$\mathbb{E}[X_L] = 2$



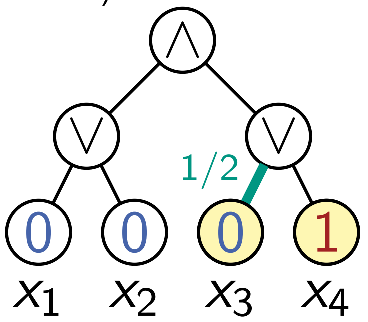
Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Base: $k = 1$



- Case analysis over all bit-strings x_1, x_2, x_3, x_4, example 0001
- Let X_L be number of leaves visited when going left first
  - Independent of leaf choice, need to look at other too: $X_L = 2$
  - When left $\vee$-node is checked, root value is determined $\mathbb{E}[X_L] = 2$
- Let X_R be number of leaves visited when going right first
  - $\Pr[\text{RE} \to x_3] = 1/2$ → visit x_4 → $X_R = 2$

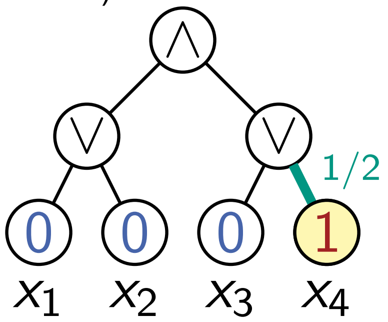
Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Base: $k = 1$



- Case analysis over all bit-strings x_1, x_2, x_3, x_4, example 0001
- Let X_L be number of leaves visited when going left first
  - Independent of leaf choice, need to look at other too: $X_L = 2$
  - When left $\vee$-node is checked, root value is determined

$\mathbb{E}[X_L] = 2$

- Let X_R be number of leaves visited when going right first
  - $\Pr[\text{RE} \rightarrow x_3] = 1/2$ → visit x_4 → $X_R = 2$
  - $\Pr[\text{RE} \rightarrow x_4] = 1/2$

Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

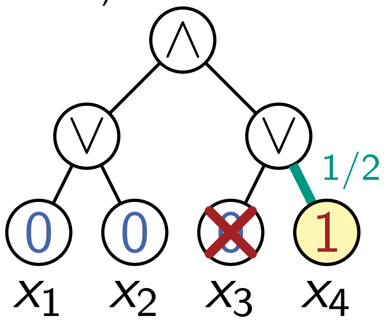
Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Base: $k = 1$

- Case analysis over all bit-strings x_1, x_2, x_3, x_4, example 0001
- Let X_L be number of leaves visited when going left first
  - Independent of leaf choice, need to look at other too: $X_L = 2$
  - When left $\vee$-node is checked, root value is determined

$\mathbb{E}[X_L] = 2$

- Let X_R be number of leaves visited when going right first
  - $\Pr[\text{RE} \to x_3] = 1/2$ → visit x_4 → $X_R = 2$
  - $\Pr[\text{RE} \to x_4] = 1/2$ → do *not* visit x_3



Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

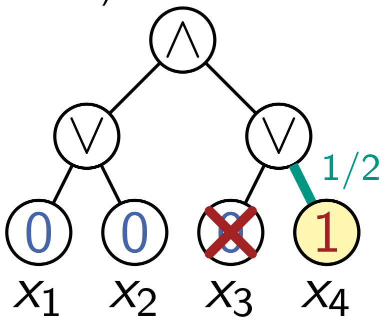
Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Base: $k = 1$

- Case analysis over all bit-strings x_1, x_2, x_3, x_4, example 0001
- Let X_L be number of leaves visited when going left first
  - Independent of leaf choice, need to look at other too: $X_L = 2$
  - When left $\vee$-node is checked, root value is determined

$\mathbb{E}[X_L] = 2$

- Let X_R be number of leaves visited when going right first
  - $\Pr[\text{RE} \to x_3] = 1/2$ → visit x_4 → $X_R = 2$
  - $\Pr[\text{RE} \to x_4] = 1/2$ → do *not* visit x_3 → $X_R = 1$



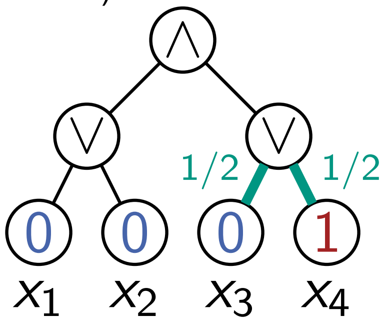
Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Base: $k = 1$



- Case analysis over all bit-strings x_1, x_2, x_3, x_4, example 0001
- Let X_L be number of leaves visited when going left first
  - Independent of leaf choice, need to look at other too: $X_L = 2$
  - When left $\vee$-node is checked, root value is determined
- Let X_R be number of leaves visited when going right first
  - $\Pr[\text{RE} \to x_3] = 1/2$ → visit x_4 → $X_R = 2$
  - $\Pr[\text{RE} \to x_4] = 1/2$ → do *not* visit x_3 → $X_R = 1$

$\mathbb{E}[X_L] = 2$

$\mathbb{E}[X_R] = 2 + \frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 1 = \frac{7}{2}$

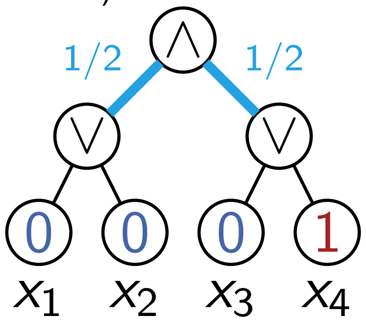
Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Base: $k = 1$



- Case analysis over all bit-strings x_1, x_2, x_3, x_4, example 0001
- Let X_L be number of leaves visited when going left first
  - Independent of leaf choice, need to look at other too: $X_L = 2$
  - When left $\vee$-node is checked, root value is determined
- Let X_R be number of leaves visited when going right first
  - $\Pr[\text{RE} \to x_3] = 1/2$ → visit x_4 → $X_R = 2$
  - $\Pr[\text{RE} \to x_4] = 1/2$ → do *not* visit x_3 → $X_R = 1$

$\mathbb{E}[X_L] = 2$

$\mathbb{E}[X_R] = 2 + \frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 1 = \frac{7}{2}$

- First left/right with prob $1/2$

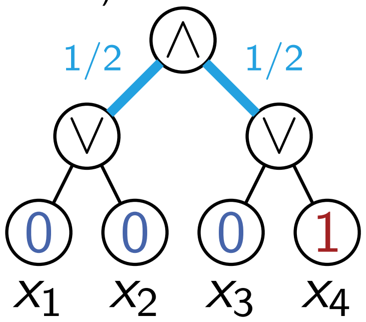
Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Base: $k = 1$



- Case analysis over all bit-strings x_1, x_2, x_3, x_4, example 0001
- Let X_L be number of leaves visited when going left first
  - Independent of leaf choice, need to look at other too: $X_L = 2$
  - When left $\vee$-node is checked, root value is determined
- Let X_R be number of leaves visited when going right first
  - $\Pr[\text{RE} \to x_3] = 1/2$ → visit x_4 → $X_R = 2$
  - $\Pr[\text{RE} \to x_4] = 1/2$ → do *not* visit x_3 → $X_R = 1$

$\mathbb{E}[X_L] = 2$

$\mathbb{E}[X_R] = 2 + \frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 1 = \frac{7}{2}$

- First left/right with prob $1/2$

$\mathbb{E}[X] = \frac{1}{2} \cdot 2 + \frac{1}{2} \cdot \frac{7}{2} = \frac{11}{4} \leq 3$ ✓

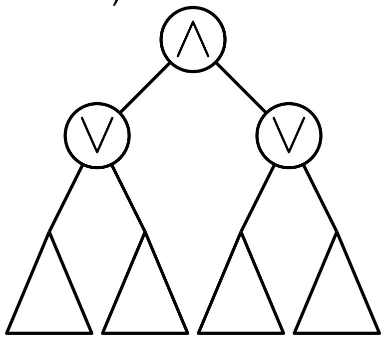
Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k - 1 \to k$



Randomized Evaluation – Running Time

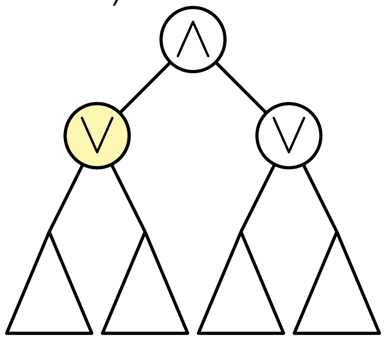
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k - 1 \to k$

- Let Y be *trees* visited in $\vee$-node



Randomized Evaluation – Running Time

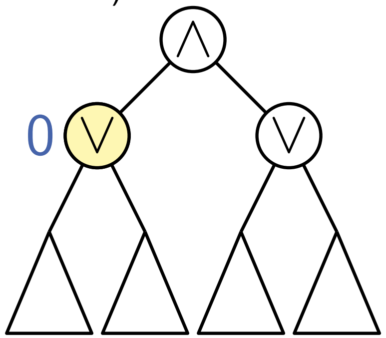
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$

- Let Y be *trees* visited in $\vee$-node
- $\vee$-Case 0: node evaluates to 0



Randomized Evaluation – Running Time

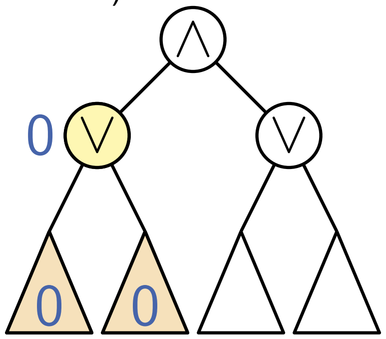
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$

- Let Y be *trees* visited in $\vee$-node
- $\vee$-Case 0: node evaluates to 0
  - both sub-trees evaluate to 0



Randomized Evaluation – Running Time

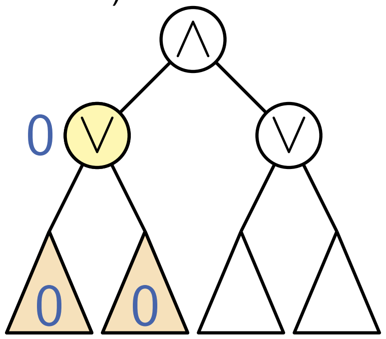
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k - 1 \to k$

- Let Y be *trees* visited in $\vee$-node
- $\vee$-Case 0: node evaluates to 0
  - both sub-trees evaluate to 0 ⟶ $Y = 2$



Randomized Evaluation – Running Time

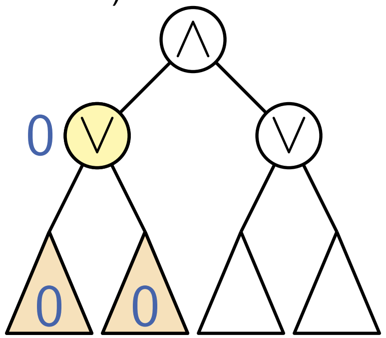
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$

- Let Y be *trees* visited in $\vee$-node
- $\vee$-Case 0: node evaluates to 0 $\longrightarrow$ $\mathbb{E}[Y] = 2$
  - both sub-trees evaluate to 0 $\longrightarrow$ $Y = 2$



Randomized Evaluation – Running Time

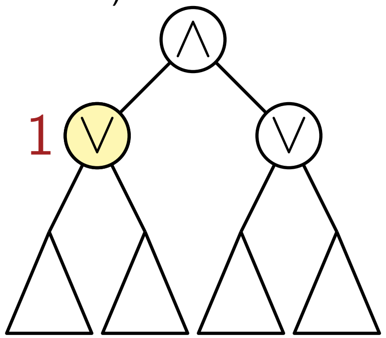
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \rightarrow k$

- Let Y be *trees* visited in $\vee$-node
- $\vee$-Case 0: node evaluates to 0 ⟶ $\mathbb{E}[Y] = 2$
  - both sub-trees evaluate to 0 ⟶ $Y = 2$
- $\vee$-Case 1: node evaluates to 1



Randomized Evaluation – Running Time

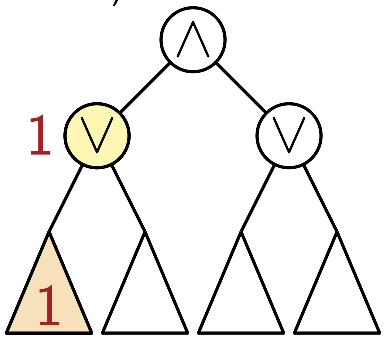
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$

- Let Y be *trees* visited in $\vee$-node
- $\vee$-Case 0: node evaluates to 0 ⟶ $\mathbb{E}[Y] = 2$
  - both sub-trees evaluate to 0 ⟶ $Y = 2$
- $\vee$-Case 1: node evaluates to 1
  - at least one sub-tree evaluates to 1



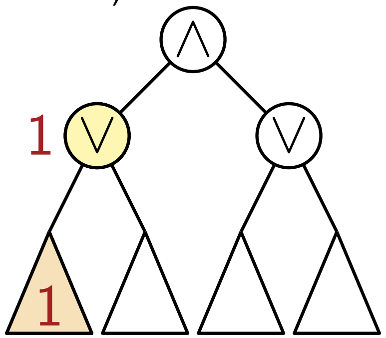
Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$



- Let Y be *trees* visited in $\vee$-node
- $\vee$-Case 0: node evaluates to 0 $\longrightarrow \mathbb{E}[Y] = 2$
  - both sub-trees evaluate to 0 $\longrightarrow Y = 2$
- $\vee$-Case 1: node evaluates to 1
  - at least one sub-tree evaluates to 1
  - with prob $p \geq 1/2$ (only!) this tree is visited first $\longrightarrow Y = 1$

Randomized Evaluation – Running Time

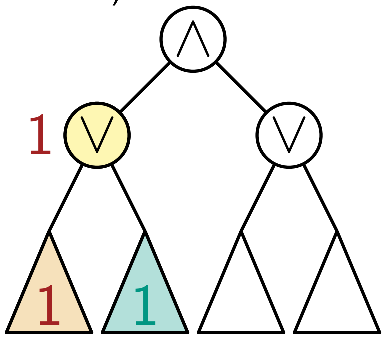
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k - 1 \to k$

- Let Y be *trees* visited in $\vee$-node
- $\vee$-Case 0: node evaluates to 0 $\longrightarrow \mathbb{E}[Y] = 2$
  - both sub-trees evaluate to 0 $\longrightarrow Y = 2$
- $\vee$-Case 1: node evaluates to 1
  - at least one sub-tree evaluates to 1
  - with prob $p \geq 1/2$ (only!) this tree is visited first $\longrightarrow Y = 1$



Randomized Evaluation – Running Time

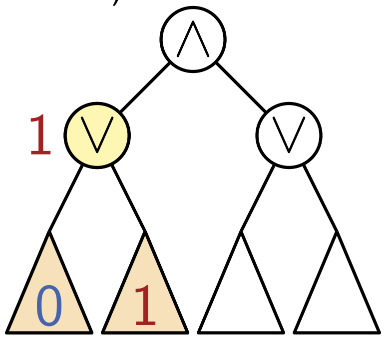
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$

- Let Y be *trees* visited in $\vee$-node
- $\vee$-Case 0: node evaluates to 0 $\longrightarrow \mathbb{E}[Y] = 2$
  - both sub-trees evaluate to 0 $\longrightarrow Y = 2$
- $\vee$-Case 1: node evaluates to 1
  - at least one sub-tree evaluates to 1
  - with prob $p \geq 1/2$ (only!) this tree is visited first $\longrightarrow Y = 1$
  - with prob $1-p \leq 1/2$ both sub-trees are visited $\longrightarrow Y = 2$



Randomized Evaluation – Running Time

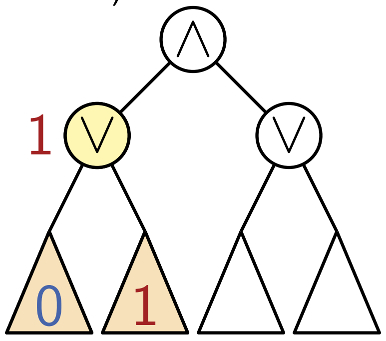
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$

- Let Y be *trees* visited in $\vee$-node
- $\vee$-Case 0: node evaluates to 0 $\longrightarrow \mathbb{E}[Y] = 2$
  - both sub-trees evaluate to 0 $\longrightarrow Y = 2$
- $\vee$-Case 1: node evaluates to 1 $\longrightarrow \mathbb{E}[Y] = p \cdot 1 + (1-p) \cdot 2 = 2 - p \leq \frac{3}{2}$
  - at least one sub-tree evaluates to 1
  - with prob $p \geq 1/2$ (only!) this tree is visited first $\longrightarrow Y = 1$
  - with prob $1-p \leq 1/2$ both sub-trees are visited $\longrightarrow Y = 2$



Randomized Evaluation – Running Time

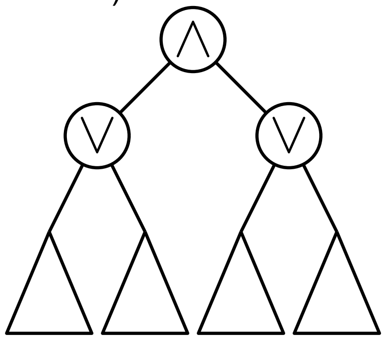
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k - 1 \rightarrow k$

- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$



Randomized Evaluation – Running Time

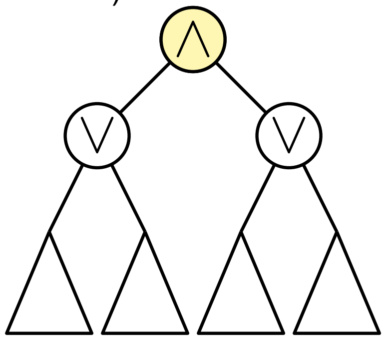
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k - 1 \to k$

- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$
- Let Z be trees visited in $\wedge$-node



Randomized Evaluation – Running Time

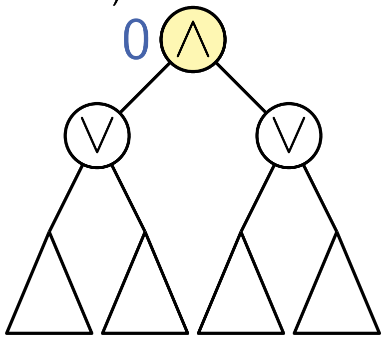
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k - 1 \to k$

- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$
- Let Z be trees visited in $\wedge$-node
- $\wedge$-Case 0: node evaluates to 0



Randomized Evaluation – Running Time

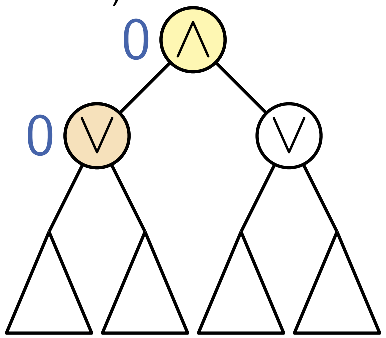
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$

- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$
- Let Z be trees visited in $\wedge$-node
- $\wedge$-Case 0: node evaluates to 0
  - at least one $\vee$-node evaluates to 0



Randomized Evaluation – Running Time

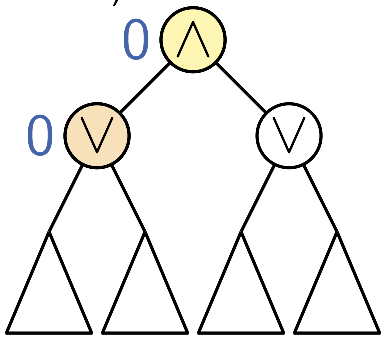
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$

- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$
- Let Z be trees visited in $\wedge$-node
- $\wedge$-Case 0: node evaluates to 0
  - at least one $\vee$-node evaluates to 0
  - with prob $p \geq 1/2$ (only!) this node is visited first



Randomized Evaluation – Running Time

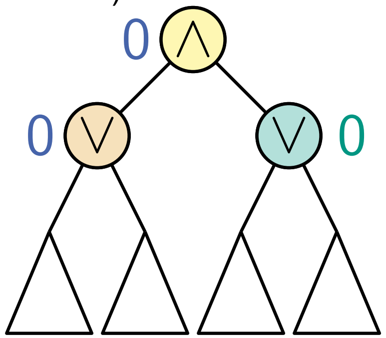
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$

- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$
- Let Z be trees visited in $\wedge$-node
- $\wedge$-Case 0: node evaluates to 0
  - at least one $\vee$-node evaluates to 0
  - with prob $p \geq 1/2$ (only!) this node is visited first



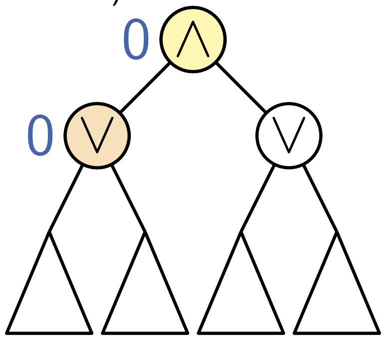
Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$



- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$
- Let Z be trees visited in $\wedge$-node
- $\wedge$-Case 0: node evaluates to 0 → $\mathbb{E}[Z] = p \cdot 2 + \cdots$
  - at least one $\vee$-node evaluates to 0
  - with prob $p \geq 1/2$ (only!) this node is visited first

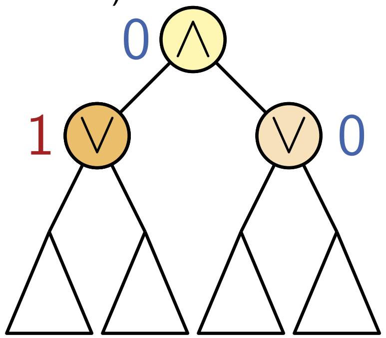
Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$



- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$
- Let Z be trees visited in $\wedge$-node
- $\wedge$-Case 0: node evaluates to 0 → $\mathbb{E}[Z] = p \cdot 2 + \cdots$
  - at least one $\vee$-node evaluates to 0
  - with prob $p \geq 1/2$ (only!) this node is visited first
  - with prob $1 - p \leq 1/2$ both $\vee$-nodes are visited

Randomized Evaluation – Running Time

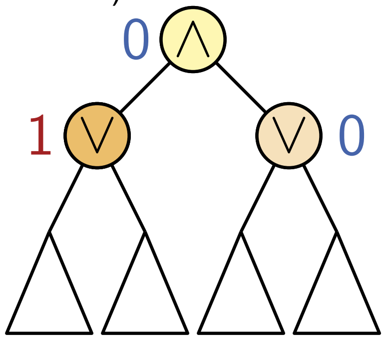
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$

- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$
- Let Z be trees visited in $\wedge$-node
- $\wedge$-Case 0: node evaluates to 0 → $\mathbb{E}[Z] = p \cdot 2 + (1-p) \cdot (2 + \frac{3}{2})$
  - at least one $\vee$-node evaluates to 0
  - with prob $p \geq 1/2$ (only!) this node is visited first
  - with prob $1-p \leq 1/2$ both $\vee$-nodes are visited



Randomized Evaluation – Running Time

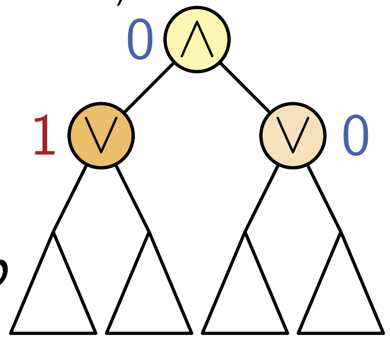
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$

- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$
- Let Z be trees visited in $\wedge$-node
- $\wedge$-Case 0: node evaluates to 0 → $\mathbb{E}[Z] = p \cdot 2 + (1-p) \cdot (2 + \frac{3}{2}) = \frac{7}{2} - \frac{3}{2}p$
  - at least one $\vee$-node evaluates to 0
  - with prob $p \geq 1/2$ (only!) this node is visited first
  - with prob $1 - p \leq 1/2$ both $\vee$-nodes are visited



Randomized Evaluation – Running Time

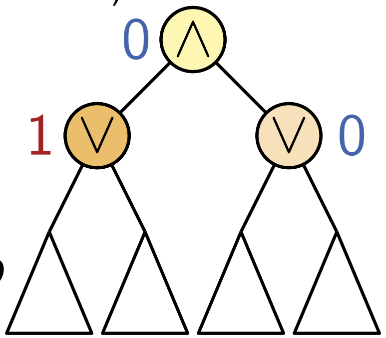
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k - 1 \to k$

- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$
- Let Z be trees visited in $\wedge$-node
- $\wedge$-Case 0: node evaluates to 0 → $\mathbb{E}[Z] = p \cdot 2 + (1-p) \cdot (2 + \frac{3}{2}) = \frac{7}{2} - \frac{3}{2}p$ $\leq \frac{11}{4} \leq 3$
  - at least one $\vee$-node evaluates to 0
  - with prob $p \geq 1/2$ (only!) this node is visited first
  - with prob $1 - p \leq 1/2$ both $\vee$-nodes are visited



Randomized Evaluation – Running Time

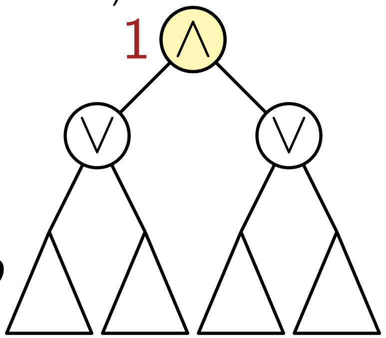
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$

- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$
- Let Z be trees visited in $\wedge$-node
- $\wedge$-Case 0: node evaluates to 0 → $\mathbb{E}[Z] = p \cdot 2 + (1-p) \cdot (2 + \frac{3}{2}) = \frac{7}{2} - \frac{3}{2}p$ $\leq \frac{11}{4} \leq 3$
  - at least one $\vee$-node evaluates to 0
  - with prob $p \geq 1/2$ (only!) this node is visited first
  - with prob $1 - p \leq 1/2$ both $\vee$-nodes are visited
- $\wedge$-Case 1: node evaluates to 1



Randomized Evaluation – Running Time

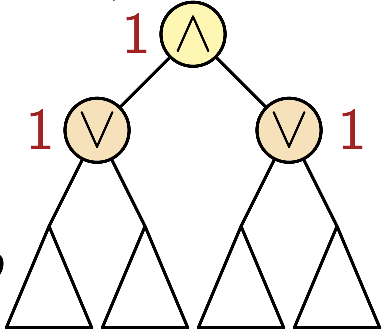
- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$

- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$
- Let Z be trees visited in $\wedge$-node
- $\wedge$-Case 0: node evaluates to 0 → $\mathbb{E}[Z] = p \cdot 2 + (1-p) \cdot (2 + \frac{3}{2}) = \frac{7}{2} - \frac{3}{2}p$ $\leq \frac{11}{4} \leq 3$
  - at least one $\vee$-node evaluates to 0
  - with prob $p \geq 1/2$ (only!) this node is visited first
  - with prob $1-p \leq 1/2$ both $\vee$-nodes are visited
- $\wedge$-Case 1: node evaluates to 1
  - both $\vee$-nodes evaluate to 1



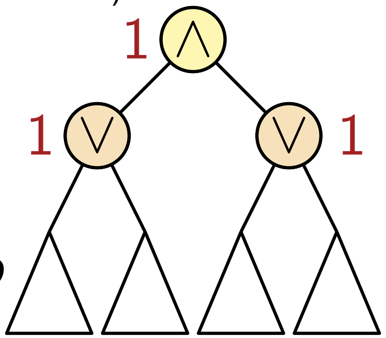
Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

> **Theorem**: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$



- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$
- Let Z be trees visited in $\wedge$-node
- $\wedge$-Case 0: node evaluates to 0 → $\mathbb{E}[Z] = p \cdot 2 + (1-p) \cdot (2 + \frac{3}{2}) = \frac{7}{2} - \frac{3}{2}p \leq \frac{11}{4} \leq 3$
  - at least one $\vee$-node evaluates to 0
  - with prob $p \geq 1/2$ (only!) this node is visited first
  - with prob $1 - p \leq 1/2$ both $\vee$-nodes are visited
- $\wedge$-Case 1: node evaluates to 1 → $\mathbb{E}[Z] = 2 \cdot \frac{3}{2} = 3$
  - both $\vee$-nodes evaluate to 1

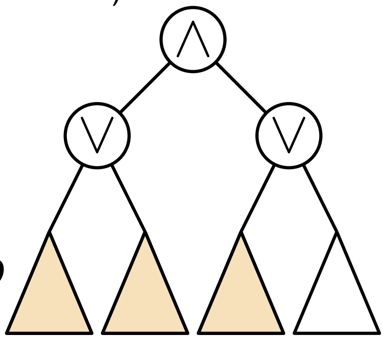
Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$



- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$
- Let Z be trees visited in $\wedge$-node
- $\wedge$-Case 0: node evaluates to 0 → $\mathbb{E}[Z] = p \cdot 2 + (1-p) \cdot (2 + \frac{3}{2}) = \frac{7}{2} - \frac{3}{2}p \leq \frac{11}{4} \leq 3$
  - at least one $\vee$-node evaluates to 0
  - with prob $p \geq 1/2$ (only!) this node is visited first
  - with prob $1-p \leq 1/2$ both $\vee$-nodes are visited
- $\wedge$-Case 1: node evaluates to 1 → $\mathbb{E}[Z] = 2 \cdot \frac{3}{2} = 3$
  - both $\vee$-nodes evaluate to 1

- Both cases: visit ≤ 3 trees in exp.

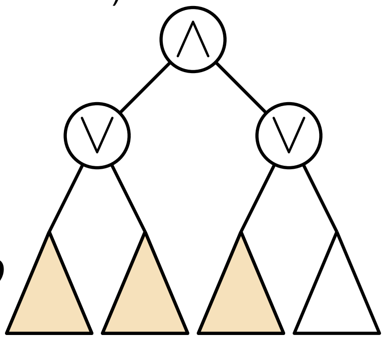
Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \ldots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\ldots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$



- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$
- Let Z be trees visited in $\wedge$-node
- $\wedge$-Case 0: node evaluates to 0 → $\mathbb{E}[Z] = p \cdot 2 + (1-p) \cdot (2 + \frac{3}{2}) = \frac{7}{2} - \frac{3}{2}p$ $\leq \frac{11}{4} \leq 3$
  - at least one $\vee$-node evaluates to 0
  - with prob $p \geq 1/2$ (only!) this node is visited first
  - with prob $1-p \leq 1/2$ both $\vee$-nodes are visited
- $\wedge$-Case 1: node evaluates to 1 → $\mathbb{E}[Z] = 2 \cdot \frac{3}{2} = 3$
  - both $\vee$-nodes evaluate to 1

- Both cases: visit ≤ 3 trees in exp.
- Induction: exp. leaves per tree $\leq 3^{k-1}$

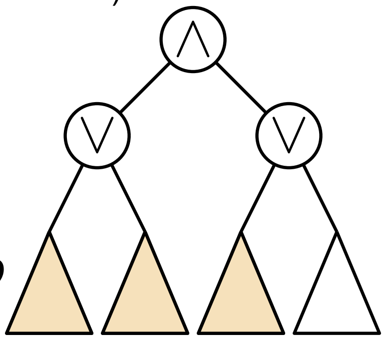
Randomized Evaluation – Running Time

- Depends on how *lucky* we are, i.e., how often we can avoid checking the other child
- The running time is a *random variable*, we cannot deduce a specific value in advance

Theorem: On *every* input $x_1, \dots x_{4^k}$ the **Randomized Evaluation** algorithm (RE) has an *expected running time* of $O(n^{\log_4(3)})$. $\approx O(n^{0.792\dots})$ is sublinear!

Proof via Induction (that the number X of visited leaves at depth $2k$ is $\leq 3^k = 3^{\log_4(n)} = n^{\log_4(3)}$ in expectation)

Step: $k-1 \to k$



- Let Y be *trees* visited in $\vee$-node → Case 0: $\mathbb{E}[Y] = 2$ Case 1: $\mathbb{E}[Y] \leq \frac{3}{2}$
- Let Z be trees visited in $\wedge$-node
- $\wedge$-Case 0: node evaluates to 0 → $\mathbb{E}[Z] = p \cdot 2 + (1-p) \cdot (2 + \frac{3}{2}) = \frac{7}{2} - \frac{3}{2}p$ $\leq \frac{11}{4} \leq 3$
  - at least one $\vee$-node evaluates to 0
  - with prob $p \geq 1/2$ (only!) this node is visited first
  - with prob $1 - p \leq 1/2$ both $\vee$-nodes are visited
- $\wedge$-Case 1: node evaluates to 1 → $\mathbb{E}[Z] = 2 \cdot \frac{3}{2} = 3$
  - both $\vee$-nodes evaluate to 1

- Both cases: visit ≤ 3 trees in exp.
- Induction: exp. leaves per tree $\leq 3^{k-1}$

$\mathbb{E}[X] \leq 3 \cdot 3^{k-1} = 3^k$ ✓

Power of Randomness: Average-Case Analysis

Binary Search Trees

- Goal: in a sequence of elements, quickly determine whether a given element is contained

Power of Randomness: Average-Case Analysis

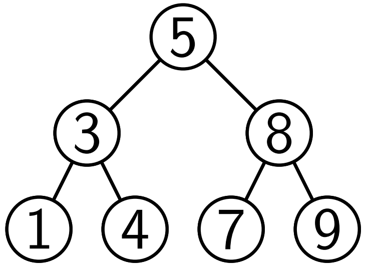
Binary Search Trees

- Goal: in a sequence of elements, quickly determine whether a given element is contained
- Example: $(1, 3, 4, 5, 7, 8, 9)$ Find: 4

Power of Randomness: Average-Case Analysis

Binary Search Trees

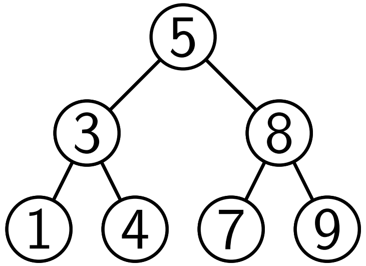
- Goal: in a sequence of elements, quickly determine whether a given element is contained
- Example: $(1, 3, 4, 5, 7, 8, 9)$ Find: 4
- Idea: elements in left sub-tree are smaller, elements in right sub-tree are larger



Power of Randomness: Average-Case Analysis

Binary Search Trees

- Goal: in a sequence of elements, quickly determine whether a given element is contained
- Example: $(1, 3, 4, 5, 7, 8, 9)$ Find: 4
- Idea: elements in left sub-tree are smaller, elements in right sub-tree are larger



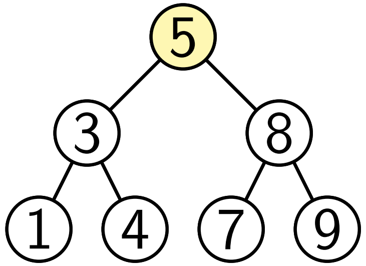
Query

- Element equal to node? O.w. recurse in left/right child when element is smaller/larger

Power of Randomness: Average-Case Analysis

Binary Search Trees

- Goal: in a sequence of elements, quickly determine whether a given element is contained
- Example: $(1, 3, 4, 5, 7, 8, 9)$ Find: 4
- Idea: elements in left sub-tree are smaller, elements in right sub-tree are larger



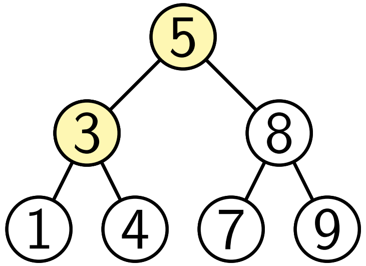
Query

- Element equal to node? O.w. recurse in left/right child when element is smaller/larger

Power of Randomness: Average-Case Analysis

Binary Search Trees

- Goal: in a sequence of elements, quickly determine whether a given element is contained
- Example: $(1, 3, 4, 5, 7, 8, 9)$ Find: 4
- Idea: elements in left sub-tree are smaller, elements in right sub-tree are larger



Query

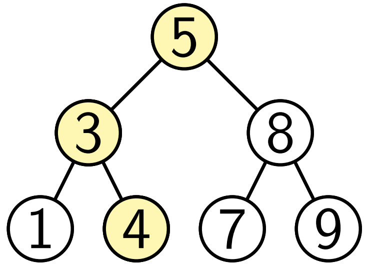
- Element equal to node? O.w. recurse in left/right child when element is smaller/larger

Power of Randomness: Average-Case Analysis

Karlsruhe Institute of Technology

Binary Search Trees

- Goal: in a sequence of elements, quickly determine whether a given element is contained
- Example: $(1, 3, 4, 5, 7, 8, 9)$ Find: 4
- Idea: elements in left sub-tree are smaller, elements in right sub-tree are larger



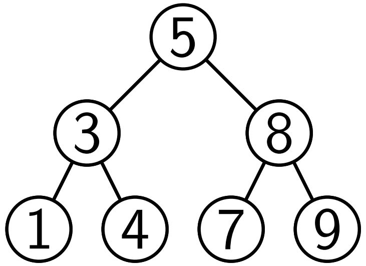
Query

- Element equal to node? O.w. recurse in left/right child when element is smaller/larger

Power of Randomness: Average-Case Analysis

Binary Search Trees

- Goal: in a sequence of elements, quickly determine whether a given element is contained
- Example: $(1, 3, 4, 5, 7, 8, 9)$ Find: 4
- Idea: elements in left sub-tree are smaller, elements in right sub-tree are larger



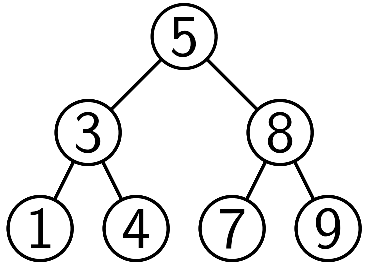
Query

- Element equal to node? O.w. recurse in left/right child when element is smaller/larger
- Running time: linear in the depth of the tree

Power of Randomness: Average-Case Analysis

Binary Search Trees

- Goal: in a sequence of elements, quickly determine whether a given element is contained
- Example: $(1,3,4,5,7,8,9)$ Find: 4
- Idea: elements in left sub-tree are smaller, elements in right sub-tree are larger



Query

- Element equal to node? O.w. recurse in left/right child when element is smaller/larger
- Running time: linear in the depth of the tree

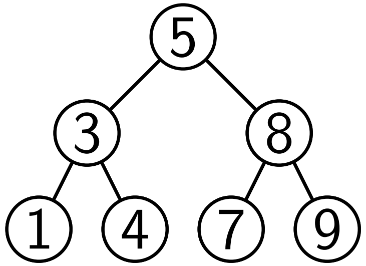
Maintenance

- Setting: elements appended over time, but never deleted

Power of Randomness: Average-Case Analysis

Binary Search Trees

- Goal: in a sequence of elements, quickly determine whether a given element is contained
- Example: $(1, 3, 4, 5, 7, 8, 9)$ Find: 4
- Idea: elements in left sub-tree are smaller, elements in right sub-tree are larger



Query

- Element equal to node? O.w. recurse in left/right child when element is smaller/larger
- Running time: linear in the depth of the tree

Maintenance

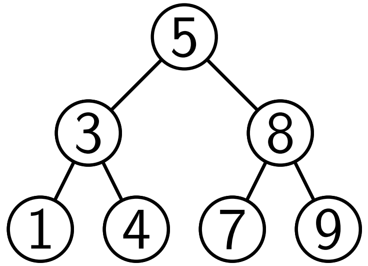
- Setting: elements appended over time, but never deleted
- How can we maintain the search-tree property as new elements arrive?

Power of Randomness: Average-Case Analysis

Karlsruhe Institute of Technology

Binary Search Trees

- Goal: in a sequence of elements, quickly determine whether a given element is contained
- Example: $(1, 3, 4, 5, 7, 8, 9)$ Find: 4
- Idea: elements in left sub-tree are smaller, elements in right sub-tree are larger



Query

- Element equal to node? O.w. recurse in left/right child when element is smaller/larger
- Running time: linear in the depth of the tree

Maintenance

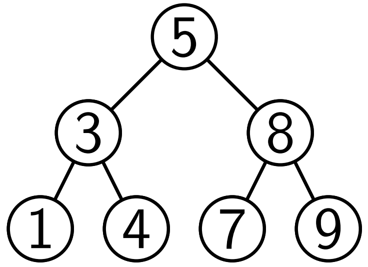
- Setting: elements appended over time, but never deleted
- How can we maintain the search-tree property as new elements arrive?

  Red-Black-Trees (a, b)*-Trees* *AVL-Trees*
- Complicated mechanisms that update the tree structure after an insertion

Power of Randomness: Average-Case Analysis

Binary Search Trees

- Goal: in a sequence of elements, quickly determine whether a given element is contained
- Example: $(1, 3, 4, 5, 7, 8, 9)$ Find: 4
- Idea: elements in left sub-tree are smaller, elements in right sub-tree are larger



Query

- Element equal to node? O.w. recurse in left/right child when element is smaller/larger
- Running time: linear in the depth of the tree

Maintenance

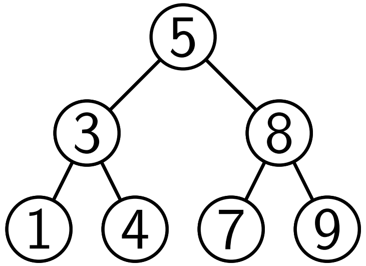
- Setting: elements appended over time, but never deleted
- How can we maintain the search-tree property as new elements arrive?

  Red-Black-Trees (a, b)*-Trees* *AVL-Trees*
- Complicated mechanisms that update the tree structure after an insertion
- Ensure that the depth is logarithmic in the number of nodes

Power of Randomness: Average-Case Analysis

Binary Search Trees

- Goal: in a sequence of elements, quickly determine whether a given element is contained
- Example: $(1, 3, 4, 5, 7, 8, 9)$ Find: 4
- Idea: elements in left sub-tree are smaller, elements in right sub-tree are larger



Query

- Element equal to node? O.w. recurse in left/right child when element is smaller/larger
- Running time: linear in the depth of the tree

Maintenance

- Setting: elements appended over time, but never deleted
- How can we maintain the search-tree property as new elements arrive?

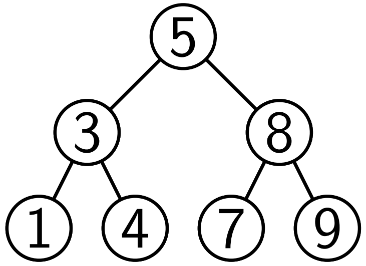
  Red-Black-Trees (a, b)*-Trees* *AVL-Trees*
- Complicated mechanisms that update the tree structure after an insertion
- Ensure that the depth is logarithmic in the number of nodes

Is all that necessary?

Keep it Simple

Simple Insert Strategy

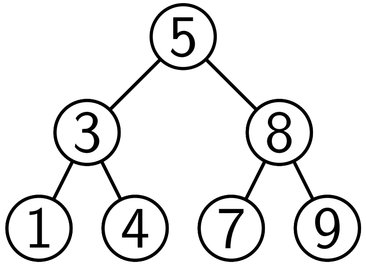
- Place a new element where it belongs. ✓



Keep it Simple

Simple Insert Strategy

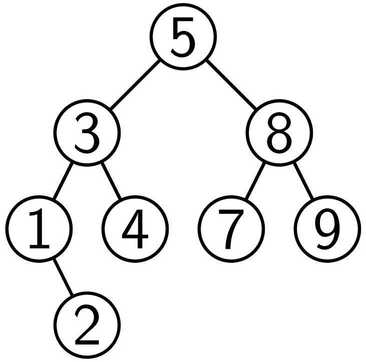
- Place a new element where it belongs. ✓
- Example: Insert 2



Keep it Simple

Simple Insert Strategy

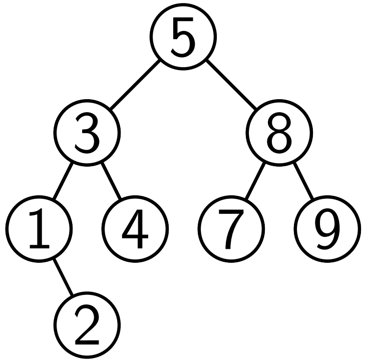
- Place a new element where it belongs. ✓
- Example: Insert 2



Keep it Simple

Simple Insert Strategy

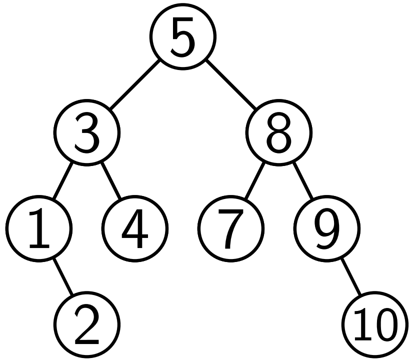
- Place a new element where it belongs. ✓
- Example: Insert 2 , 10



Keep it Simple

Simple Insert Strategy

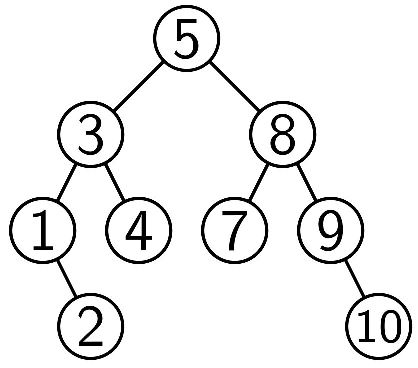
- Place a new element where it belongs. ✓
- Example: Insert 2 , 10



Keep it Simple

Simple Insert Strategy

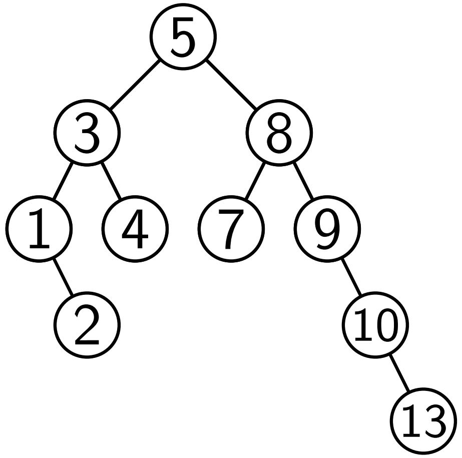
- Place a new element where it belongs. ✓
- Example: Insert 2 , 10 , 13



Keep it Simple

Simple Insert Strategy

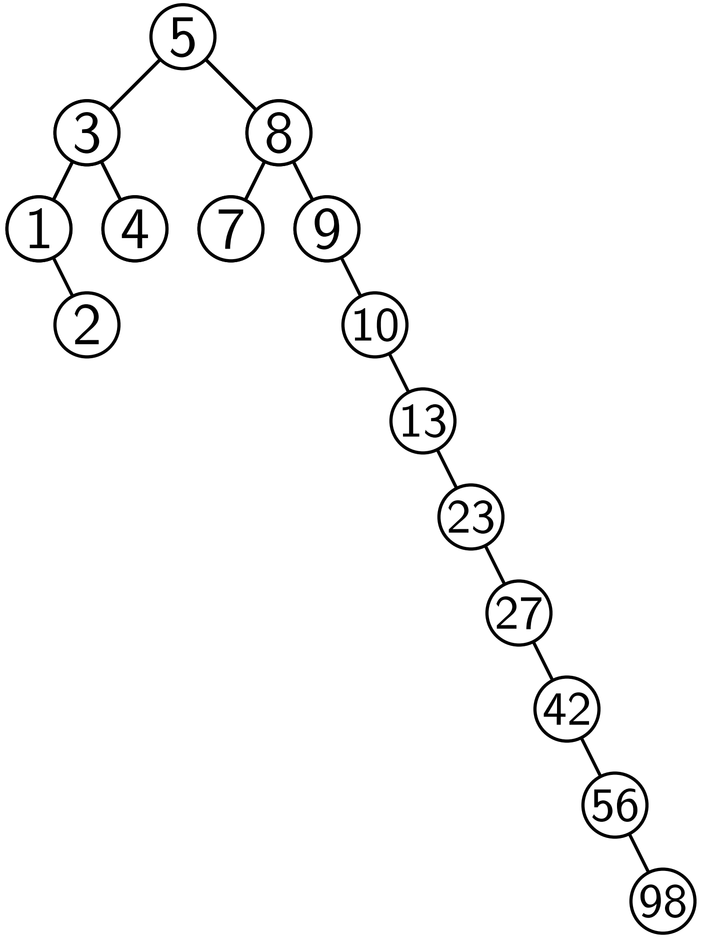
- Place a new element where it belongs. ✓
- Example: Insert 2 , 10 , 13



Keep it Simple

Simple Insert Strategy

- Place a new element where it belongs. ✓
- Example: Insert 2 , 10 , 13 , 23, 27, 42, 56, 98



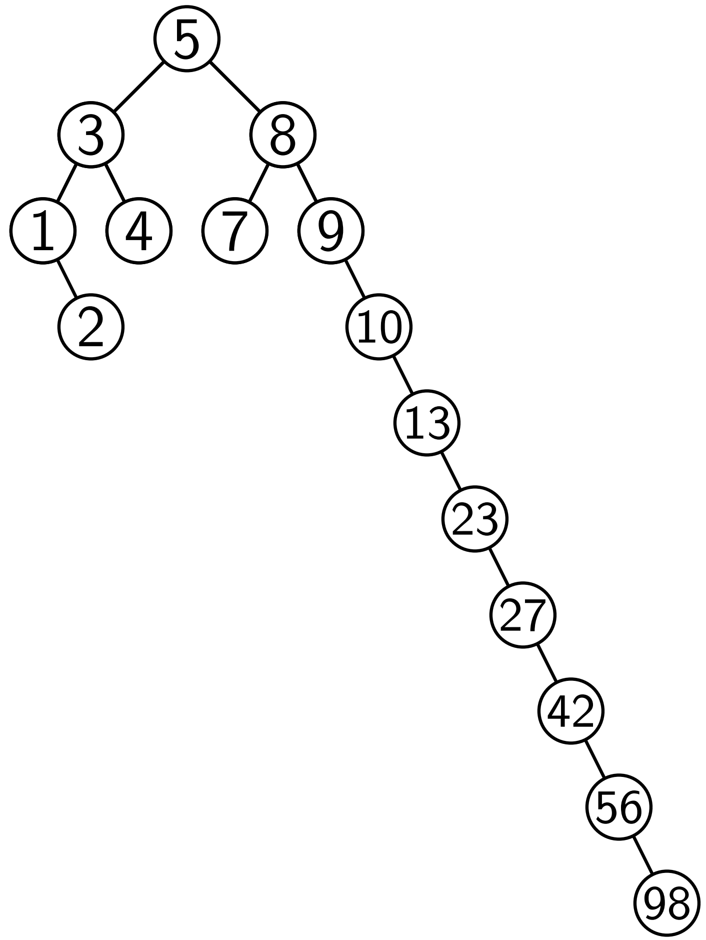
Keep it Simple

Simple Insert Strategy

- Place a new element where it belongs. ✓
- Example: Insert 2 , 10 , 13 , 23, 27, 42, 56, 98

Problem

- If elements come in sorted order, tree is unbalanced



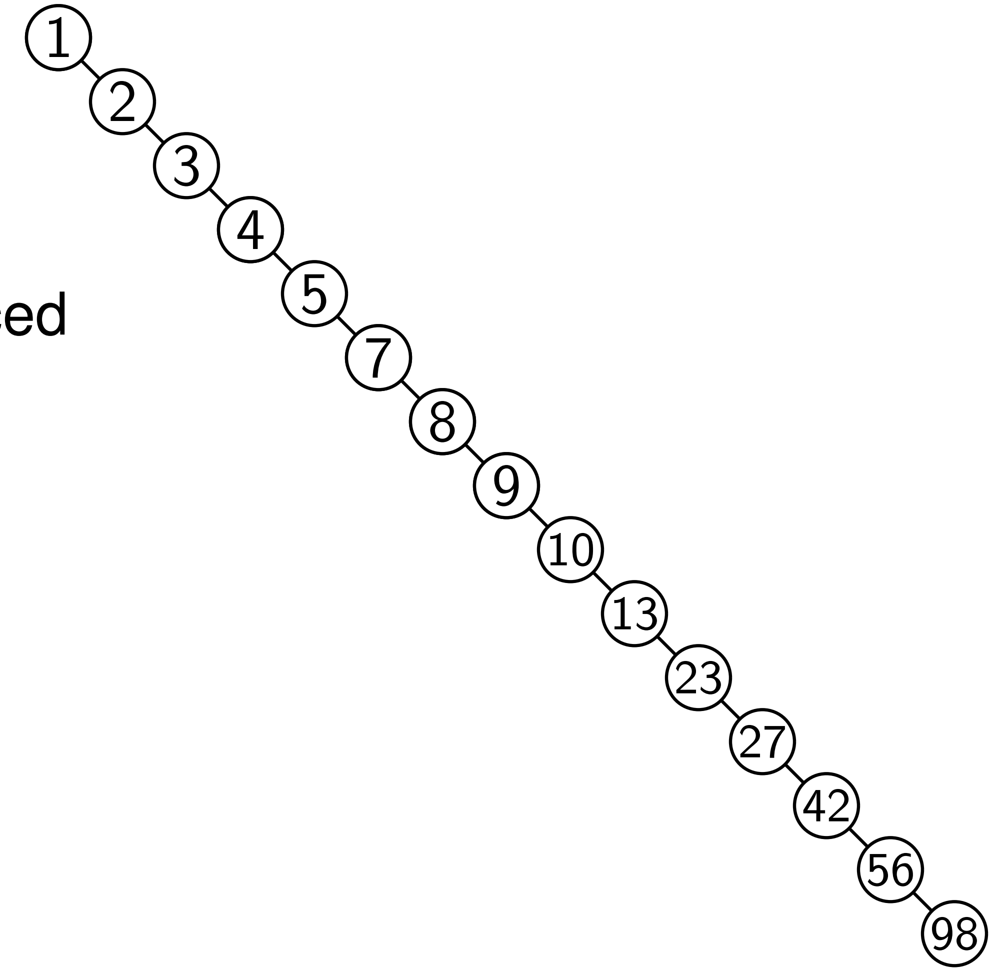
Keep it Simple

Simple Insert Strategy

- Place a new element where it belongs. ✓
- Example: Insert 2 , 10 , 13 , 23, 27, 42, 56, 98

Problem

- If elements come in sorted order, tree is unbalanced
- Worst case: linear running time for single query



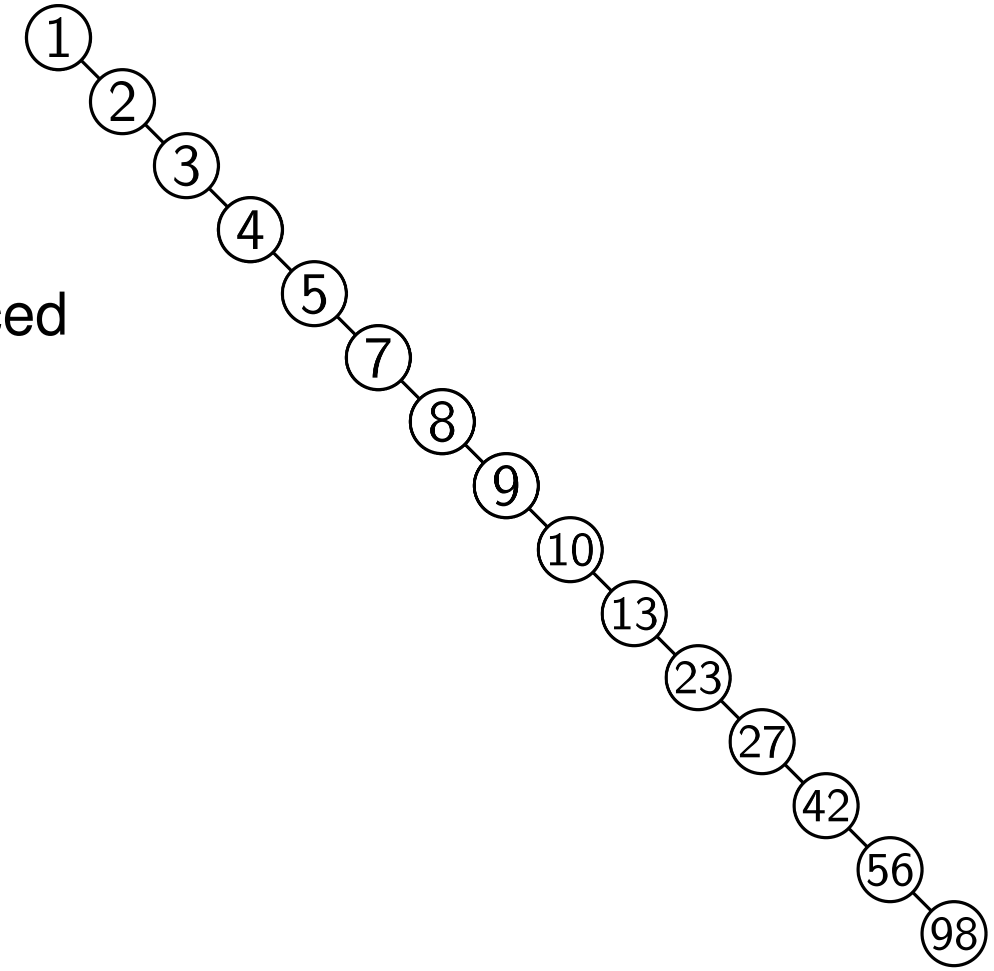
Keep it Simple

Simple Insert Strategy

- Place a new element where it belongs. ✓
- Example: Insert 2 , 10 , 13 , 23, 27, 42, 56, 98

Problem ?

- If elements come in sorted order, tree is unbalanced
- Worst case: linear running time for single query
- Is that *actually* a problem?



Keep it Simple

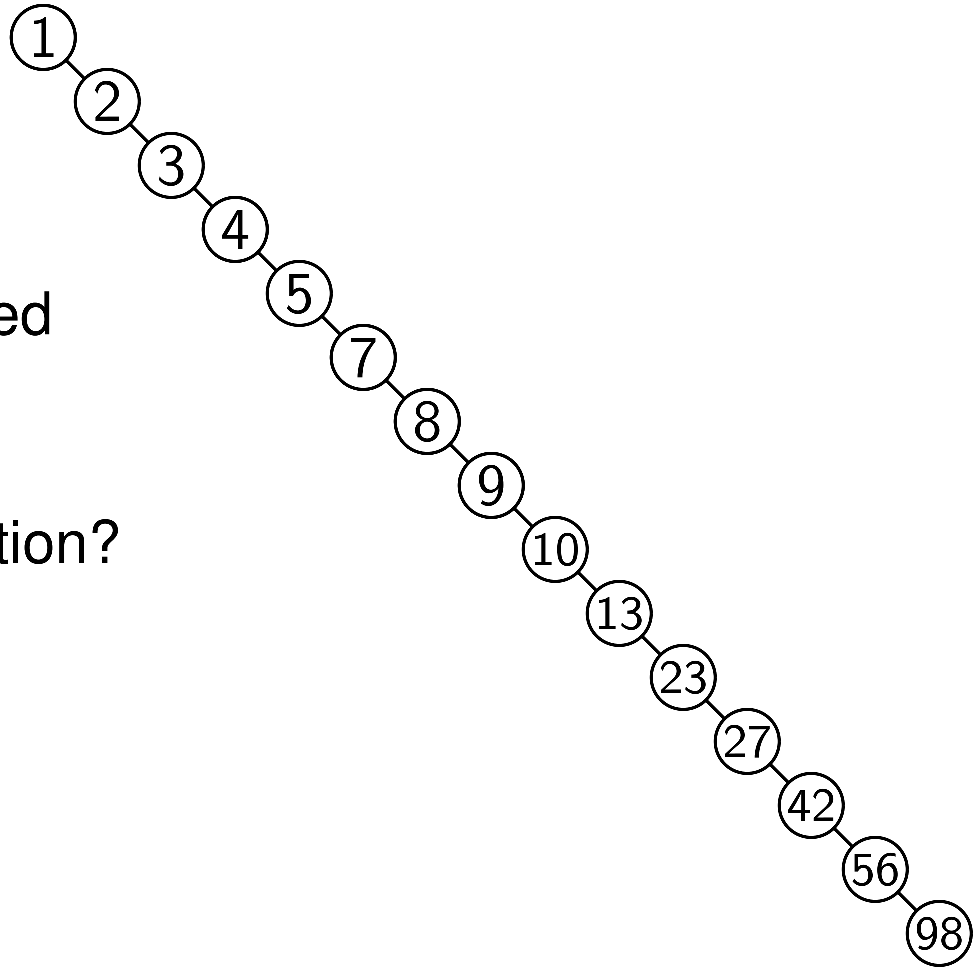
Karlsruhe Institute of Technology

Simple Insert Strategy

- Place a new element where it belongs. ✓
- Example: Insert 2 , 10 , 13 , 23, 27, 42, 56, 98

Problem ?

- If elements come in sorted order, tree is unbalanced
- Worst case: linear running time for single query
- Is that *actually* a problem?
- Is it *likely* that this happens in a real-world application?



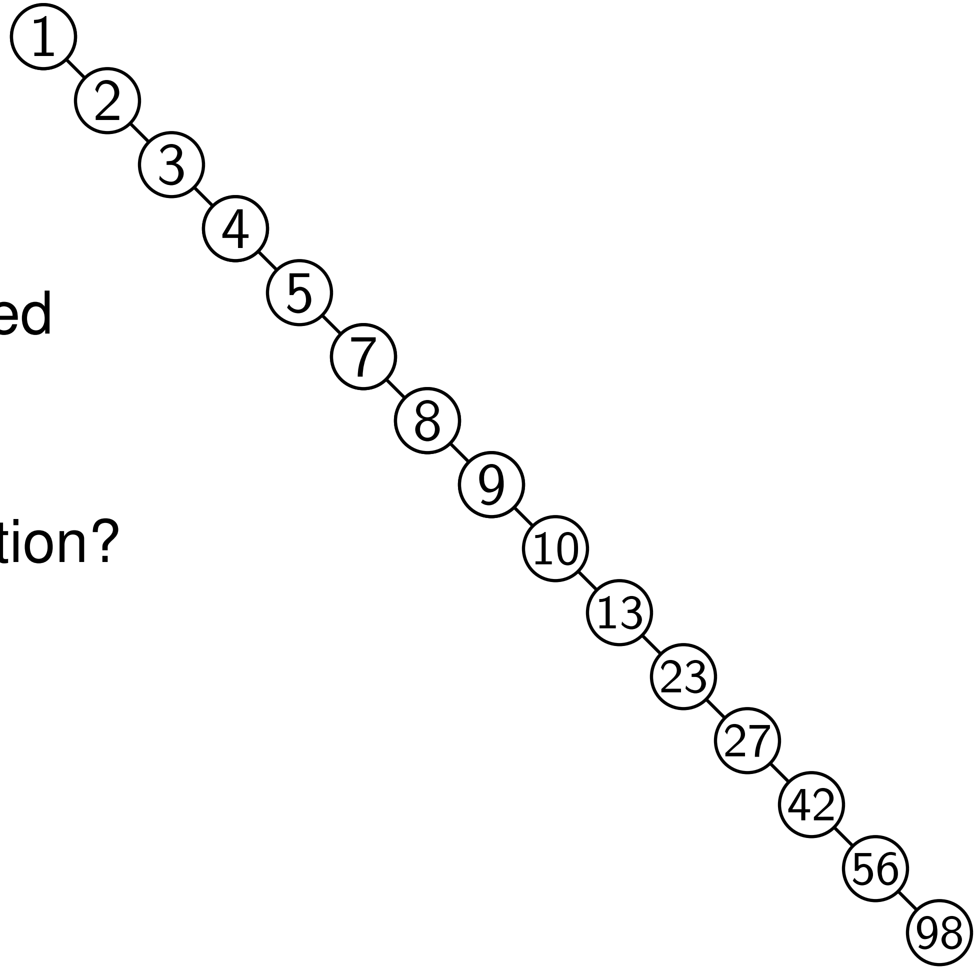
Keep it Simple

Simple Insert Strategy

- Place a new element where it belongs. ✓
- Example: Insert 2 , 10 , 13 , 23, 27, 42, 56, 98

Problem ?

- If elements come in sorted order, tree is unbalanced
- Worst case: linear running time for single query
- Is that *actually* a problem?
- Is it *likely* that this happens in a real-world application?
- Only 1 sequence yields this tree



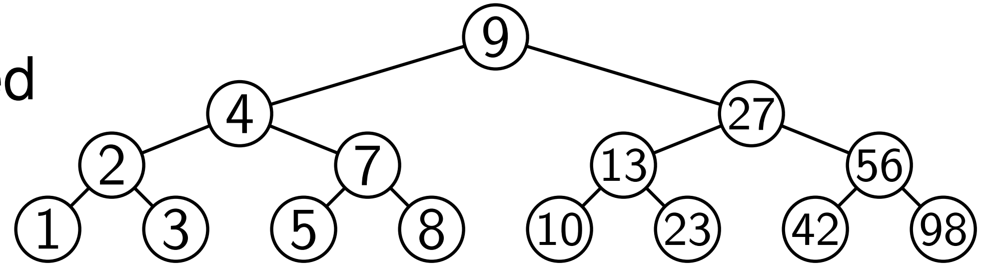
Keep it Simple

Simple Insert Strategy

- Place a new element where it belongs. ✓
- Example: Insert 2 , 10 , 13 , 23, 27, 42, 56, 98

Problem ?

- If elements come in sorted order, tree is unbalanced
- Worst case: linear running time for single query
- Is that *actually* a problem?
- Is it *likely* that this happens in a real-world application?
- Only 1 sequence yields this tree , 21964800 sequences yield a perfectly balanced tree



https://oeis.org/A056971

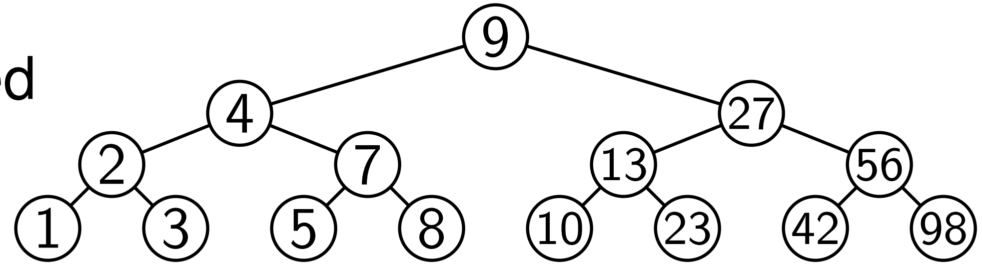
Keep it Simple

Simple Insert Strategy

- Place a new element where it belongs. ✓
- Example: Insert 2 , 10 , 13 , 23, 27, 42, 56, 98

Problem ?

- If elements come in sorted order, tree is unbalanced
- Worst case: linear running time for single query
- Is that *actually* a problem?



- Is it *likely* that this happens in a real-world application?
- Only 1 sequence yields this tree , 21964800 sequences yield a perfectly balanced tree

https://oeis.org/A056971

Average-Case Analysis

- Model real world via probability distribution over possible inputs, which is

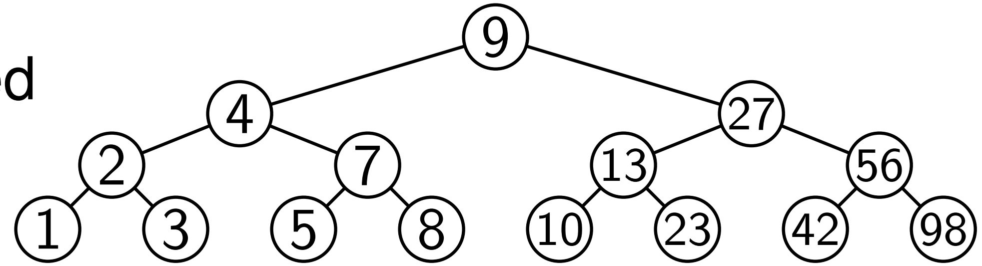
Keep it Simple

Simple Insert Strategy

- Place a new element where it belongs. ✓
- Example: Insert 2 , 10 , 13 , 23, 27, 42, 56, 98

Problem ?

- If elements come in sorted order, tree is unbalanced
- Worst case: linear running time for single query
- Is that *actually* a problem?
- Is it *likely* that this happens in a real-world application?
- Only 1 sequence yields this tree , 21964800 sequences yield a perfectly balanced tree



https://oeis.org/A056971

Average-Case Analysis

- Model real world via probability distribution over possible inputs, which is
  - simple (so that we can analyze it)

Keep it Simple

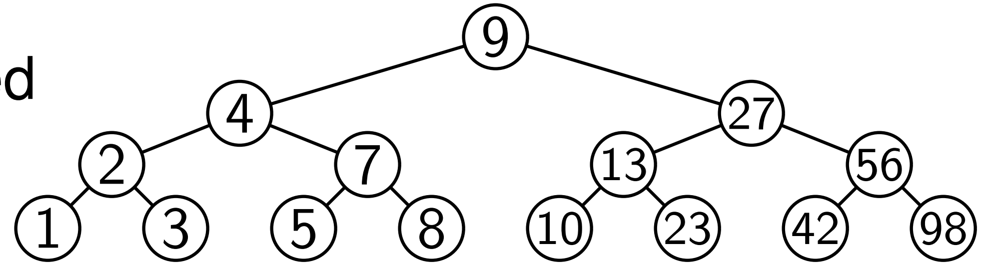
Karlsruhe Institute of Technology

Simple Insert Strategy

- Place a new element where it belongs. ✓
- Example: Insert 2 , 10 , 13 , 23, 27, 42, 56, 98

Problem ?

- If elements come in sorted order, tree is unbalanced
- Worst case: linear running time for single query
- Is that *actually* a problem?
- Is it *likely* that this happens in a real-world application?
- Only 1 sequence yields this tree , 21964800 sequences yield a perfectly balanced tree



https://oeis.org/A056971

Average-Case Analysis

- Model real world via probability distribution over possible inputs, which is
  - simple (so that we can analyze it)
  - realistic (so that we can make useful predictions about the real world)

Keep it Simple

Karlsruhe Institute of Technology

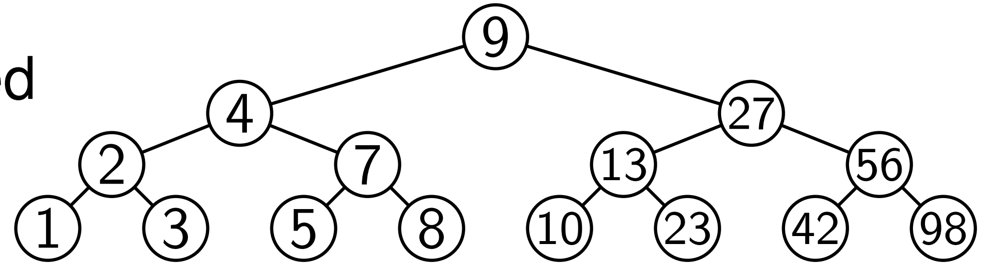
Simple Insert Strategy

- Place a new element where it belongs. ✓
- Example: Insert 2 , 10 , 13 , 23, 27, 42, 56, 98

Problem ?

- If elements come in sorted order, tree is unbalanced
- Worst case: linear running time for single query
- Is that *actually* a problem?
- Is it *likely* that this happens in a real-world application?
- Only 1 sequence yields this tree , 21964800 sequences yield a perfectly balanced tree

https://oeis.org/A056971



Average-Case Analysis

- Model real world via probability distribution over possible inputs, which is
  - simple (so that we can analyze it)
  - realistic (so that we can make useful predictions about the real world)

In the following: uniform random permutation of the numbers

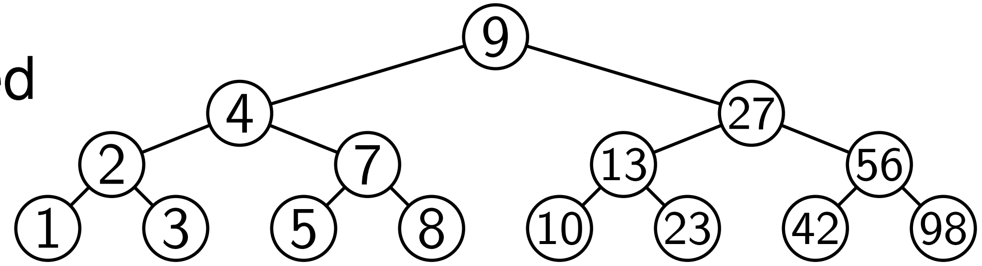
Keep it Simple

Simple Insert Strategy

- Place a new element where it belongs. ✓
- Example: Insert 2 , 10 , 13 , 23, 27, 42, 56, 98

Problem ?

- If elements come in sorted order, tree is unbalanced
- Worst case: linear running time for single query
- Is that *actually* a problem?
- Is it *likely* that this happens in a real-world application?
- Only 1 sequence yields this tree , 21964800 sequences yield a perfectly balanced tree



https://oeis.org/A056971

Average-Case Analysis

- Model real world via probability distribution over possible inputs, which is
  - simple (so that we can analyze it) ✓
  - realistic (so that we can make useful predictions about the real world) Not so clear...

In the following: uniform random permutation of the numbers

Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \ldots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \ldots, n$, as we are only interested in the order

Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \ldots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \ldots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \ldots, v\}$ in S.

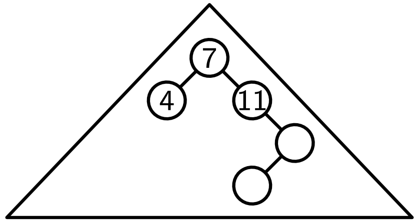
Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \ldots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \ldots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \ldots, v\}$ in S.

$$M = \{1, 2, 3, 4, \ldots, u, u+1, \ldots, v, \ldots, n\}$$
$$S = (7, 11, 4, \ldots, u, \ldots, v, \ldots, u+1, \ldots, 1)$$



Simple Insert Strategy: Analysis

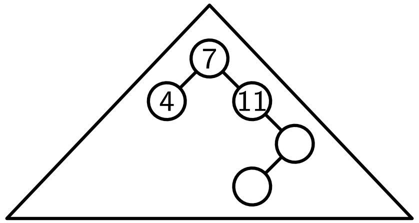
Theorem: Let S be a permutation of $M = \{1, 2, ..., n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, ..., n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, ..., v\}$ in S.

$M = \{1, 2, 3, 4, ..., u, u+1, ..., v, ..., n\}$

$S = (7, 11, 4, ..., u, ..., v, ..., u+1, ..., 1)$



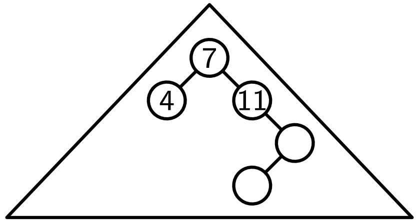
Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \dots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \dots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \dots, v\}$ in S.

$$M = \{1, 2, 3, 4, \dots, u, u+1, \dots, v, \dots, n\}$$
$$S = (7, 11, 4, \dots, u, \dots, v, \dots, u+1, \dots, 1)$$



Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \ldots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

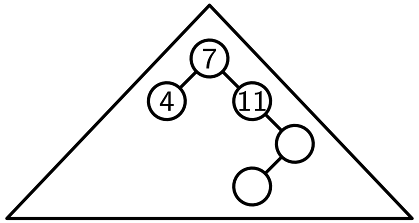
- w.l.o.g. we can assume the elements to be $1, \ldots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \ldots, v\}$ in S.

- Before an element in $M_{u,v}$ is added, all elements are smaller/larger

$M = \{1, 2, 3, 4, \ldots, u, u+1, \ldots, v, \ldots, n\}$

$S = (7, 11, 4, \ldots, u, \ldots, v, \ldots, u+1, \ldots, 1)$



Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \dots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

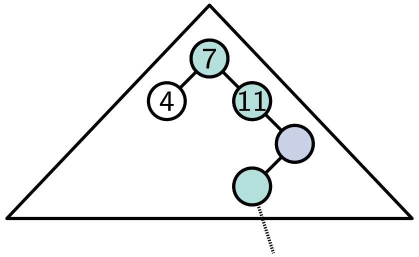
- w.l.o.g. we can assume the elements to be $1, \dots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \dots, v\}$ in S.

- Before an element in $M_{u,v}$ is added, all elements are smaller/larger
- All paths that would lead to $x \in M_{u,v}$ are identical

$M = \{1, 2, 3, 4, \dots, u, u + 1, \dots, v, \dots, n\}$

$S = (7, 11, 4, \dots, u, \dots, v, \dots, u + 1, \dots, 1)$



Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \dots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

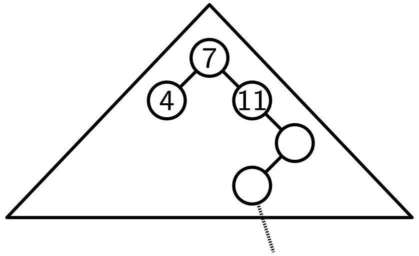
- w.l.o.g. we can assume the elements to be $1, \dots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \dots, v\}$ in S.

- Before an element in $M_{u,v}$ is added, all elements are smaller/larger
- All paths that would lead to $x \in M_{u,v}$ are identical
- Let $u' \in M_{u,v}$ be the *first* element from $M_{u,v}$ to appear in S

$M = \{1, 2, 3, 4, ..., u, u + 1, ..., v, ..., n\}$

$S = (7, 11, 4, ..., u, ..., v, ..., u + 1, ..., 1)$



Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \ldots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

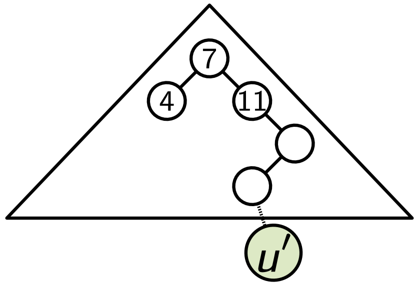
- w.l.o.g. we can assume the elements to be $1, \ldots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \ldots, v\}$ in S.

- Before an element in $M_{u,v}$ is added, all elements are smaller/larger
- All paths that would lead to $x \in M_{u,v}$ are identical
- Let $u' \in M_{u,v}$ be the *first* element from $M_{u,v}$ to appear in S
- From then on, u' is on the path that would lead to v

$M = \{1, 2, 3, 4, \ldots, u, u+1, \ldots, v, \ldots, n\}$

$S = (7, 11, 4, \ldots, u, \ldots, v, \ldots, u+1, \ldots, 1)$



Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \ldots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

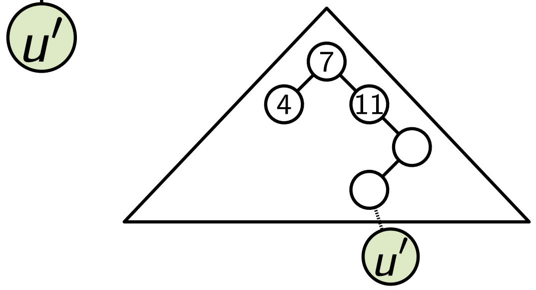
- w.l.o.g. we can assume the elements to be $1, \ldots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \ldots, v\}$ in S.

- Before an element in $M_{u,v}$ is added, all elements are smaller/larger
- All paths that would lead to $x \in M_{u,v}$ are identical
- Let $u' \in M_{u,v}$ be the *first* element from $M_{u,v}$ to appear in S
- From then on, u' is on the path that would lead to v
- Case 1: $u' = u$: u is on path ✓

$M = \{1, 2, 3, 4, \ldots, u, u+1, \ldots, v, \ldots, n\}$

$S = (7, 11, 4, \ldots, u, \ldots, v, \ldots, u+1, \ldots, 1)$



Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \dots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

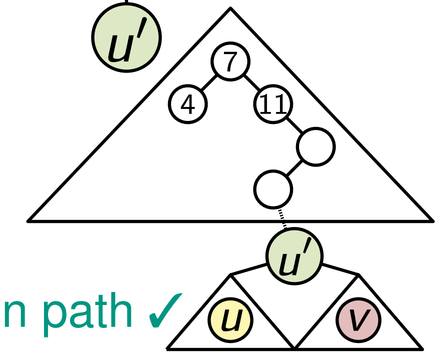
- w.l.o.g. we can assume the elements to be $1, \dots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \dots, v\}$ in S.

- Before an element in $M_{u,v}$ is added, all elements are smaller/larger
- All paths that would lead to $x \in M_{u,v}$ are identical
- Let $u' \in M_{u,v}$ be the *first* element from $M_{u,v}$ to appear in S
- From then on, u' is on the path that would lead to v
- Case 1: $u' = u$: u is on path ✓
- Case 2: $u' \neq u$: ($u < u'$) & u is in left sub-tree of u' but v is in right u not on path ✓

$M = \{1, 2, 3, 4, ..., u, u+1, ..., v, ..., n\}$

$S = (7, 11, 4, ..., u, ..., v, ..., u+1, ..., 1)$



Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \dots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \dots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \dots, v\}$ in S.

$u > v$

$M_{v,u} = \{v, \dots, u\}$ (for symmetry reasons)

Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \dots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \dots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \dots, v\}$ in S. $u > v$

$M_{v,u} = \{v, \dots, u\}$ (for symmetry reasons)

- Let $S_{u,v}$ be the subsequence of S containing the elements in $M_{u,v}$

$$M_{u,v} = \{u, u+1, u+2, v\}$$
$$S = (\dots, u, \dots, u+2, \dots, v, \dots, u+1, \dots)$$
$$S_{u,v} = (u, u+2, v, u+1)$$

Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \dots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \dots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \dots, v\}$ in S. $u > v$

$M_{v,u} = \{v, \dots, u\}$ (for symmetry reasons)

- Let $S_{u,v}$ be the subsequence of S containing the elements in $M_{u,v}$
- Then $S_{u,v}$ is a uniform random permutation of $M_{u,v}$

$$M_{u,v} = \{u, u+1, u+2, v\}$$
$$S = (\dots, u, \dots, u+2, \dots, v, \dots, u+1, \dots)$$
$$S_{u,v} = (u, u+2, v, u+1)$$

Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, ..., n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, ..., n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, ..., v\}$ in S.

$u > v$

$M_{v,u} = \{v, ..., u\}$ (for symmetry reasons)

- Let $S_{u,v}$ be the subsequence of S containing the elements in $M_{u,v}$
- Then $S_{u,v}$ is a uniform random permutation of $M_{u,v}$
- The probability that u is first in $S_{u,v}$ is
  $\Pr[\text{“}u \text{ first in } S_{u,v}\text{”}] = 1/|M_{u,v}| = 1/(v - u + 1)$

$$M_{u,v} = \{u, u+1, u+2, v\}$$
$$S = (..., u, ..., u+2, ..., v, ..., u+1, ...)$$
$$S_{u,v} = (u, u+2, v, u+1)$$

Simple Insert Strategy: Analysis

Karlsruhe Institute of Technology

Theorem: Let S be a permutation of $M = \{1, 2, \ldots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \ldots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \ldots, v\}$ in S.

$u > v$

$M_{v,u} = \{v, \ldots, u\}$ (for symmetry reasons)

- Let $S_{u,v}$ be the subsequence of S containing the elements in $M_{u,v}$
- Then $S_{u,v}$ is a uniform random permutation of $M_{u,v}$
- The probability that u is first in $S_{u,v}$ is
  $\Pr[\text{“}u \text{ first in } S_{u,v}\text{”}] = 1/|M_{u,v}| = 1/(v - u + 1)$
- Analogous for $S_{v,u}$
  $\Pr[\text{“}u \text{ first in } S_{v,u}\text{”}] = 1/(u - v + 1)$

$$M_{u,v} = \{u, u+1, u+2, v\}$$
$$S = (\ldots, u, \ldots, u+2, \ldots, v, \ldots, u+1, \ldots)$$
$$S_{u,v} = (u, u+2, v, u+1)$$

Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \dots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \dots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \dots, v\}$ in S. $u > v$

$$\Pr[\text{“}u \text{ on path to } v\text{”}] = \begin{cases} 1/(v-u+1), & \text{if } u < v \\ 1/(u-v+1), & \text{if } v < u \end{cases}$$

Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \ldots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \ldots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \ldots, v\}$ in S. $u > v$

- Let X_u be the indicator random variable with

$$X_u = \begin{cases} 1, & \text{if } u \text{ is on the path to } v \\ 0, & \text{otherwhise} \end{cases}$$

$$\Pr[\text{“}u \text{ on path to } v\text{”}] = \begin{cases} 1/(v-u+1), & \text{if } u < v \\ 1/(u-v+1), & \text{if } v < u \end{cases}$$

Simple Insert Strategy: Analysis

Karlsruhe Institute of Technology

Theorem: Let S be a permutation of $M = \{1, 2, \ldots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \ldots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \ldots, v\}$ in S. $u > v$

- Let X_u be the indicator random variable with

$$\Pr[\text{“}u \text{ on path to } v\text{”}] = \begin{cases} 1/(v-u+1), & \text{if } u < v \\ 1/(u-v+1), & \text{if } v < u \end{cases}$$

$$X_u = \begin{cases} 1, & \text{if } u \text{ is on the path to } v \\ 0, & \text{otherwhise} \end{cases}$$

- Then the length of the path to v is $\ell = \sum_{u \in \{1,\ldots,n\} \setminus \{v\}} X_u$

Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \dots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \dots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \dots, v\}$ in S. $u > v$

- Let X_u be the indicator random variable with

$$X_u = \begin{cases} 1, & \text{if } u \text{ is on the path to } v \\ 0, & \text{otherwhise} \end{cases}$$

$$\Pr[\text{"}u \text{ on path to } v\text{"}] = \begin{cases} 1/(v-u+1), & \text{if } u < v \\ 1/(u-v+1), & \text{if } v < u \end{cases}$$

- Then the length of the path to v is $\ell = \sum_{u \in \{1,\dots,n\} \setminus \{v\}} X_u$

$$\mathbb{E}[\ell] = \mathbb{E}\left[\sum_{u=1}^{v-1} X_u + \sum_{u=v+1}^{n} X_u\right]$$

Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \ldots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \ldots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \ldots, v\}$ in S. $u > v$

- Let X_u be the indicator random variable with

$$X_u = \begin{cases} 1, & \text{if } u \text{ is on the path to } v \\ 0, & \text{otherwise} \end{cases}$$

$$\Pr[\text{“}u \text{ on path to } v\text{”}] = \begin{cases} 1/(v-u+1), & \text{if } u < v \\ 1/(u-v+1), & \text{if } v < u \end{cases}$$

- Then the length of the path to v is $\ell = \sum_{u \in \{1,\ldots,n\} \setminus \{v\}} X_u$

$$\mathbb{E}[\ell] = \mathbb{E}\left[\sum_{u=1}^{v-1} X_u + \sum_{u=v+1}^{n} X_u\right] = \sum_{u=1}^{v-1} \mathbb{E}[X_u] + \sum_{u=v+1}^{n} \mathbb{E}[X_u]$$

Simple Insert Strategy: Analysis

Karlsruhe Institute of Technology

Theorem: Let S be a permutation of $M = \{1, 2, \ldots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \ldots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \ldots, v\}$ in S. $u > v$

- Let X_u be the indicator random variable with

$$X_u = \begin{cases} 1, & \text{if } u \text{ is on the path to } v \\ 0, & \text{otherwise} \end{cases}$$

$$\mathbb{E}[X_u] = \Pr[X_u = 1]$$

$$\Pr[\text{“}u \text{ on path to } v\text{”}] = \begin{cases} 1/(v - u + 1), & \text{if } u < v \\ 1/(u - v + 1), & \text{if } v < u \end{cases}$$

- Then the length of the path to v is $\ell = \sum_{u \in \{1,\ldots,n\} \setminus \{v\}} X_u$

$$\mathbb{E}[\ell] = \mathbb{E}\left[\sum_{u=1}^{v-1} X_u + \sum_{u=v+1}^{n} X_u\right] = \sum_{u=1}^{v-1} \mathbb{E}[X_u] + \sum_{u=v+1}^{n} \mathbb{E}[X_u]$$

Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \ldots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \ldots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \ldots, v\}$ in S. $u > v$

- Let X_u be the indicator random variable with

$$X_u = \begin{cases} 1, & \text{if } u \text{ is on the path to } v \\ 0, & \text{otherwhise} \end{cases}$$

$$\mathbb{E}[X_u] = \Pr[X_u = 1]$$

$$\Pr[\text{“}u \text{ on path to } v\text{”}] = \begin{cases} 1/(v - u + 1), & \text{if } u < v \\ 1/(u - v + 1), & \text{if } v < u \end{cases}$$

- Then the length of the path to v is $\ell = \sum_{u \in \{1,\ldots,n\} \setminus \{v\}} X_u$

$$\mathbb{E}[\ell] = \mathbb{E}\left[\sum_{u=1}^{v-1} X_u + \sum_{u=v+1}^{n} X_u\right] = \sum_{u=1}^{v-1} \frac{1}{v - u + 1} + \sum_{u=v+1}^{n} \frac{1}{u - v + 1}$$

Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \dots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \dots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \dots, v\}$ in S. $u > v$

- Let X_u be the indicator random variable with

$$X_u = \begin{cases} 1, & \text{if } u \text{ is on the path to } v \\ 0, & \text{otherwise} \end{cases}$$

$$\mathbb{E}[X_u] = \Pr[X_u = 1]$$

$$\Pr[\text{“}u \text{ on path to } v\text{”}] = \begin{cases} 1/(v-u+1), & \text{if } u < v \\ 1/(u-v+1), & \text{if } v < u \end{cases}$$

- Then the length of the path to v is $\ell = \sum_{u \in \{1,\dots,n\} \setminus \{v\}} X_u$

$$\mathbb{E}[\ell] = \mathbb{E}\left[\sum_{u=1}^{v-1} X_u + \sum_{u=v+1}^{n} X_u\right] = \sum_{u=1}^{v-1} \frac{1}{v-u+1} + \sum_{u=v+1}^{n} \frac{1}{u-v+1} = \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{v} + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n-v+1}$$

Simple Insert Strategy: Analysis

Theorem: Let S be a permutation of $M = \{1, 2, \dots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \dots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \dots, v\}$ in S. $u > v$

- Let X_u be the indicator random variable with

$$X_u = \begin{cases} 1, & \text{if } u \text{ is on the path to } v \\ 0, & \text{otherwise} \end{cases}$$

$$\mathbb{E}[X_u] = \Pr[X_u = 1]$$

$$\Pr[\text{“}u \text{ on path to } v\text{”}] = \begin{cases} 1/(v-u+1), & \text{if } u < v \\ 1/(u-v+1), & \text{if } v < u \end{cases}$$

- Then the length of the path to v is $\ell = \sum_{u \in \{1,\dots,n\} \setminus \{v\}} X_u$

Harmonic number:
$H_n = \sum_{i=1}^{n} \frac{1}{i} \in O(\log(n))$

$$\mathbb{E}[\ell] = \mathbb{E}\left[\sum_{u=1}^{v-1} X_u + \sum_{u=v+1}^{n} X_u\right] = \sum_{u=1}^{v-1} \frac{1}{v-u+1} + \sum_{u=v+1}^{n} \frac{1}{u-v+1} = \underbrace{\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{v}}_{H_v - 1} + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n-v+1}$$

Simple Insert Strategy: Analysis

Karlsruhe Institute of Technology

> **Theorem**: Let S be a permutation of $M = \{1, 2, \ldots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \ldots, n$, as we are only interested in the order

> **Observation**: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \ldots, v\}$ in S. $u > v$

- Let X_u be the indicator random variable with

$$X_u = \begin{cases} 1, & \text{if } u \text{ is on the path to } v \\ 0, & \text{otherwhise} \end{cases}$$

> $\mathbb{E}[X_u] = \Pr[X_u = 1]$

$$\Pr[\text{“}u \text{ on path to } v\text{”}] = \begin{cases} 1/(v-u+1), & \text{if } u < v \\ 1/(u-v+1), & \text{if } v < u \end{cases}$$

- Then the length of the path to v is $\ell = \sum_{u \in \{1,\ldots,n\} \setminus \{v\}} X_u$

> Harmonic number:
> $H_n = \sum_{i=1}^{n} \frac{1}{i} \in O(\log(n))$

$$\mathbb{E}[\ell] = \mathbb{E}\left[\sum_{u=1}^{v-1} X_u + \sum_{u=v+1}^{n} X_u\right] = \sum_{u=1}^{v-1} \frac{1}{v-u+1} + \sum_{u=v+1}^{n} \frac{1}{u-v+1} = \underbrace{\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{v}}_{H_v - 1} + \underbrace{\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-v+1}}_{H_{n-v+1} - 1}$$

Simple Insert Strategy: Analysis

Karlsruhe Institute of Technology

Theorem: Let S be a permutation of $M = \{1, 2, \ldots, n\}$ chosen uniformly at random. Then, the expected depth of a binary search tree with the Simple Insert Strategy is $O(\log(n))$.

- w.l.o.g. we can assume the elements to be $1, \ldots, n$, as we are only interested in the order

Observation: Let T be a binary search tree with the Simple Insert Strategy and let $v \in T$ be an element. Then the path from v to the root contains a node $u < v$, if and only if u is the first among $M_{u,v} = \{u, \ldots, v\}$ in S. $u > v$

- Let X_u be the indicator random variable with

$$X_u = \begin{cases} 1, & \text{if } u \text{ is on the path to } v \\ 0, & \text{otherwhise} \end{cases}$$

$$\mathbb{E}[X_u] = \Pr[X_u = 1]$$

$$\Pr[\text{“}u \text{ on path to } v\text{”}] = \begin{cases} 1/(v-u+1), & \text{if } u < v \\ 1/(u-v+1), & \text{if } v < u \end{cases}$$

- Then the length of the path to v is $\ell = \sum_{u \in \{1,\ldots,n\} \setminus \{v\}} X_u$

Harmonic number:
$H_n = \sum_{i=1}^{n} \frac{1}{i} \in O(\log(n))$

$$\mathbb{E}[\ell] = \mathbb{E}\left[\sum_{u=1}^{v-1} X_u + \sum_{u=v+1}^{n} X_u\right] = \sum_{u=1}^{v-1} \frac{1}{v-u+1} + \sum_{u=v+1}^{n} \frac{1}{u-v+1} = \underbrace{\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{v}}_{H_v - 1} + \underbrace{\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-v+1}}_{H_{n-v+1} - 1} \in O(\log(n))\ ✓$$

Conclusion

Organizational

- Homepage: scale.iti.kit.edu/teaching/2023ws/randalg
- A place for questions will be linked on the website

Conclusion

Organizational

- Homepage: scale.iti.kit.edu/teaching/2023ws/randalg
- A place for questions will be linked on the website

Randomized Algorithms

- Often simpler/faster than deterministic ones (sometimes the only possible way)
- At the cost of certainty (may be slow, may be wrong)

Conclusion

Organizational

- Homepage: scale.iti.kit.edu/teaching/2023ws/randalg
- A place for questions will be linked on the website

Randomized Algorithms

- Often simpler/faster than deterministic ones (sometimes the only possible way)
- At the cost of certainty (may be slow, may be wrong)

  Quicksort (expected $O(n \log(n))$ but $O(n^2)$ worst case)

Conclusion

Karlsruhe Institute of Technology

Organizational

- Homepage: scale.iti.kit.edu/teaching/2023ws/randalg
- A place for questions will be linked on the website

Randomized Algorithms

- Often simpler/faster than deterministic ones (sometimes the only possible way)
- At the cost of certainty (may be slow, may be wrong)

  Quicksort (expected $O(n\log(n))$ but $O(n^2)$ worst case) Next week!

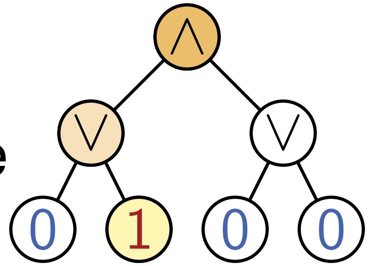
Conclusion

Organizational

- Homepage: scale.iti.kit.edu/teaching/2023ws/randalg
- A place for questions will be linked on the website

Randomized Algorithms

- Often simpler/faster than deterministic ones (sometimes the only possible way)
- At the cost of certainty (may be slow, may be wrong)
  - Quicksort (expected $O(n\log(n))$ but $O(n^2)$ worst case)
  - Next week!
- Example: AND/OR-Trees, expected running time sublinear in the input size



Conclusion

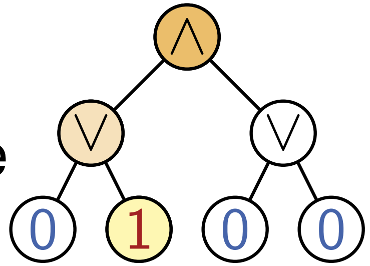
Karlsruhe Institute of Technology

Organizational

- Homepage: scale.iti.kit.edu/teaching/2023ws/randalg
- A place for questions will be linked on the website

Randomized Algorithms

- Often simpler/faster than deterministic ones (sometimes the only possible way)
- At the cost of certainty (may be slow, may be wrong)
  Quicksort (expected $O(n\log(n))$ but $O(n^2)$ worst case) Next week!
- Example: AND/OR-Trees, expected running time sublinear in the input size



Average-Case Analysis

- Model real world using probability distributions over inputs
- If worst case is unlikely, expect good running times
- Example: Binary search-trees with simple insert strategy have same expected depth as complicated deterministic data structures