Lecture 6: Balanced Search Trees

Michael Dinitz

September 12, 2024

601.433/633 Introduction to Algorithms

Announcements

- HW2 due now, HW3 released
- Regrade policy: 72 hours from when grades released
  - Don't abuse this!
  - If too many of your regrade requests do not result in positive changes, will ban you from regrade requests
  - Grading can go down!

Introduction

Today, and next few weeks: data structures.

- Since “Data Structures” a prereq, focus on advanced structures and on interesting analysis

Introduction

Today, and next few weeks: data structures.

- Since “Data Structures” a prereq, focus on advanced structures and on interesting analysis

Today and later: data structures for *dictionaries*

Introduction

Today, and next few weeks: data structures.

- Since "Data Structures" a prereq, focus on advanced structures and on interesting analysis

Today and later: data structures for *dictionaries*

Definition

A *dictionary data structure* is a data structure supporting the following operations:

- **insert(key,object)**: insert the (key, object) pair.
- **lookup(key)**: return the associated object
- **delete(key)**: remove the key and its object from the data structure. We may or may not care about this operation.

Obvious Approaches

Reminder: all running times for *worst case*

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

- Lookup:

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

- Lookup: $O(\log n)$

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

- Lookup: $O(\log n)$
- Insert:

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

- Lookup: $O(\log n)$
- Insert: $\Omega(n)$

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

- Lookup: $O(\log n)$
- Insert: $\Omega(n)$

Approach 2: Unsorted (linked) list

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

- Lookup: $O(\log n)$
- Insert: $\Omega(n)$

Approach 2: Unsorted (linked) list

- Insert:

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

- Lookup: $O(\log n)$
- Insert: $\Omega(n)$

Approach 2: Unsorted (linked) list

- Insert: $O(1)$

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

- Lookup: $O(\log n)$
- Insert: $\Omega(n)$

Approach 2: Unsorted (linked) list

- Insert: $O(1)$
- Lookup:

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

- Lookup: $O(\log n)$
- Insert: $\Omega(n)$

Approach 2: Unsorted (linked) list

- Insert: $O(1)$
- Lookup: $\Omega(n)$

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

- Lookup: $O(\log n)$
- Insert: $\Omega(n)$

Approach 2: Unsorted (linked) list

- Insert: $O(1)$
- Lookup: $\Omega(n)$

Goal: $O(\log n)$ for both.

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

- Lookup: $O(\log n)$
- Insert: $\Omega(n)$

Approach 2: Unsorted (linked) list

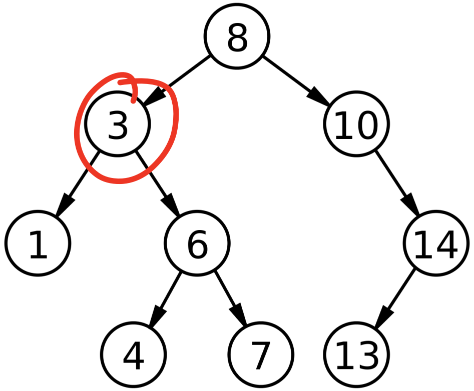
- Insert: $O(1)$
- Lookup: $\Omega(n)$

Goal: $O(\log n)$ for both.
Approach today: search trees

Binary Search Tree Review

Binary search tree:

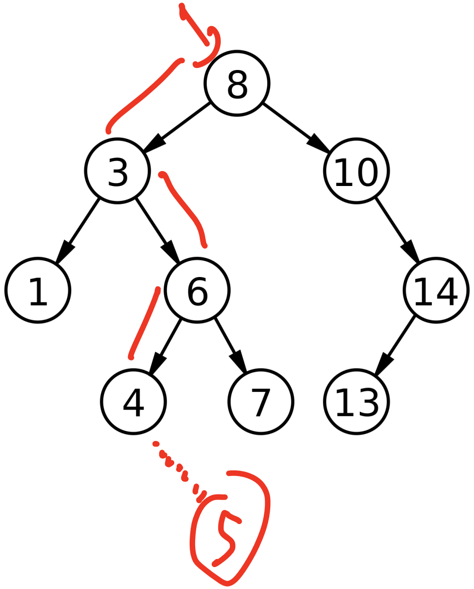
- All nodes have at most **2** children
- Each node stores (key, object) pair
- All descendants to left have smaller keys
- All descendants to the right have larger keys



Binary Search Tree Review

Binary search tree:

- All nodes have at most **2** children
- Each node stores (key, object) pair
- All descendants to left have smaller keys
- All descendants to the right have larger keys



Lookup: follow path from root!

Dictionary Operations in Simple Binary Search Tree

insert(x):

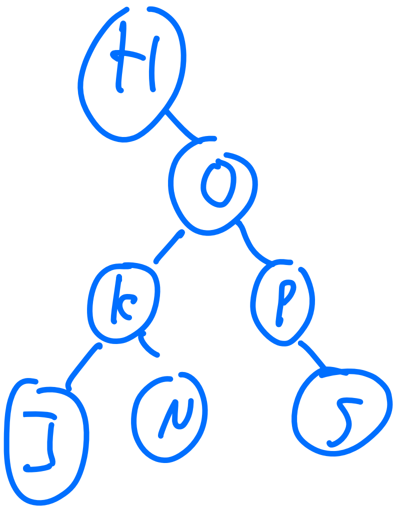
- If tree empty, put x at root
- Else if $x < \textit{root.key}$ recursively insert into left child
- Else (if $x > \textit{root.key}$) recursively insert into right child

Dictionary Operations in Simple Binary Search Tree

insert(x):

- If tree empty, put x at root
- Else if $x <$ ***root.key*** recursively insert into left child
- Else (if $x >$ ***root.key***) recursively insert into right child

Example: `H O P K I N S`



Simply Binary Search Tree: Analysis

Pluses: easy to implement

Simply Binary Search Tree: Analysis

Pluses: easy to implement

(Worst-case) Running time:

Simply Binary Search Tree: Analysis

Pluses: easy to implement

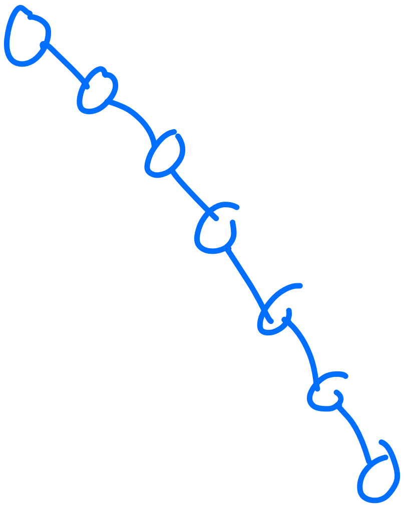
(Worst-case) Running time: if depth $\boldsymbol{d}$, then $\boldsymbol{\Theta(d)}$

Simply Binary Search Tree: Analysis

Pluses: easy to implement

(Worst-case) Running time: if depth $\boldsymbol{d}$, then $\boldsymbol{\Theta(d)}$

- If very unbalanced $\boldsymbol{d}$ could be $\boldsymbol{\Omega(n)}$!



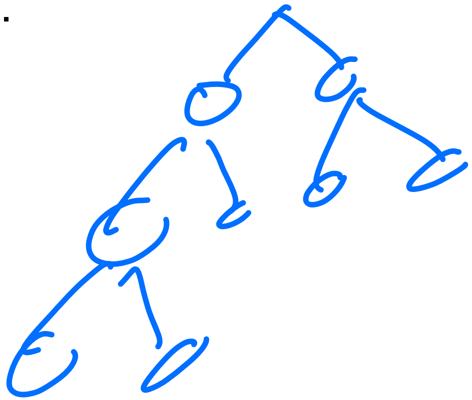
Simply Binary Search Tree: Analysis

Pluses: easy to implement

(Worst-case) Running time: if depth $\boldsymbol{d}$, then $\boldsymbol{\Theta(d)}$

- If very unbalanced $\boldsymbol{d}$ could be $\boldsymbol{\Omega(n)}$!

Want to make tree *balanced*.



Simply Binary Search Tree: Analysis

Pluses: easy to implement

(Worst-case) Running time: if depth d, then $\Theta(d)$

- If very unbalanced d could be $\Omega(n)$!

Want to make tree *balanced*.

Rest of today:

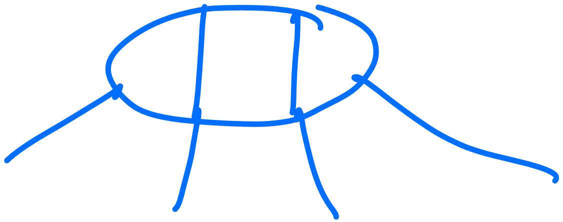
- B-trees: perfect balance, not binary
- Red-black trees: approximate balance, binary
- Turn out to be related!

B-Trees

B-tree Definition

Parameter $t \geq 2$.

B-tree Definition



Parameter $t \geq 2$.

Definition (B-tree with parameter t)

1. Each node has between $t-1$ and $2t-1$ keys in it (except the root has between 1 and $2t-1$ keys). Keys in a node are stored in a sorted array.
2. Each non-leaf has degree (number of children) equal to the number of keys in it plus 1. If v is a node with keys $[a_1, a_2, \ldots, a_k]$ and the children are $[v_1, v_2, \ldots, v_{k+1}]$, then the tree rooted at v_i contains only keys that are at least a_{i-1} and at most a_i (except the the edge cases: the tree rooted at v_1 has keys less than a_1, and the tree rooted at v_{k+1} has keys at least a_k).
3. All leaves are at the same depth.

B-tree Definition

Parameter $t \geq 2$.

Definition (B-tree with parameter t)

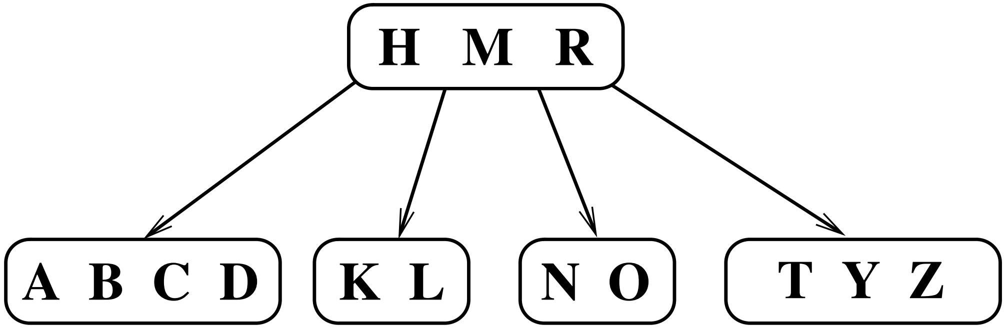
1. Each node has between $t-1$ and $2t-1$ keys in it (except the root has between 1 and $2t-1$ keys). Keys in a node are stored in a sorted array.
2. Each non-leaf has degree (number of children) equal to the number of keys in it plus 1. If v is a node with keys $[a_1, a_2, \ldots, a_k]$ and the children are $[v_1, v_2, \ldots, v_{k+1}]$, then the tree rooted at v_i contains only keys that are at least a_{i-1} and at most a_i (except the the edge cases: the tree rooted at v_1 has keys less than a_1, and the tree rooted at v_{k+1} has keys at least a_k).
3. All leaves are at the same depth.

When $t = 2$ known as a *2-3-4 tree*, since # children either 2, 3, or 4

B-tree: Example

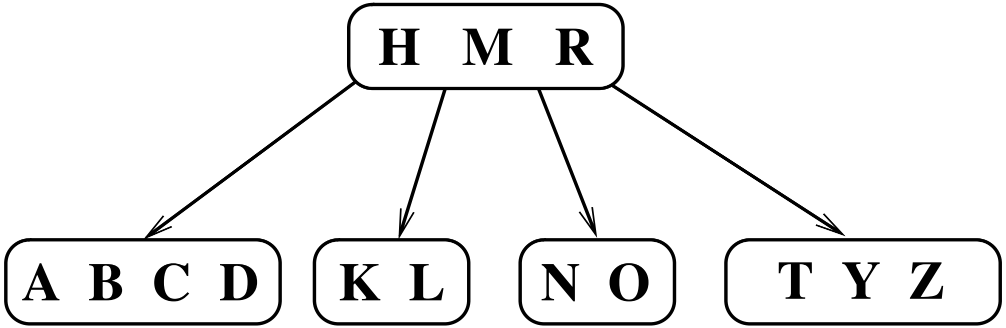
$t = 3$:

- Root has between **1** and **5** keys, non-roots have between **2** and **5** keys
- Non-leaves have between **3** and **6** children (root can have fewer).

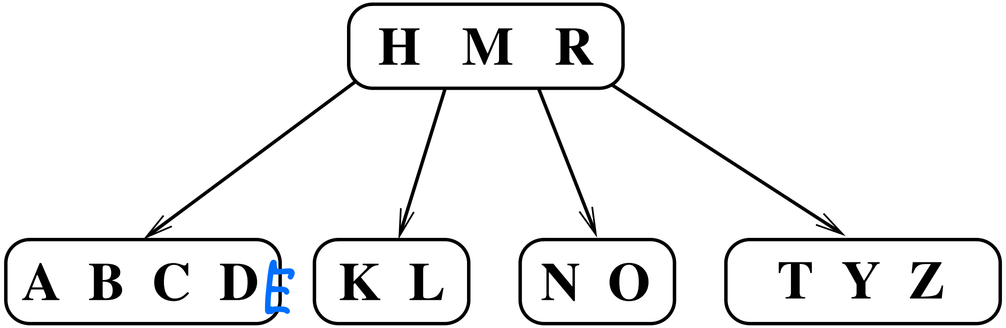


Lookups

Binary search in array at root. Finished if find item, else get pointer to appropriate child, recurse.



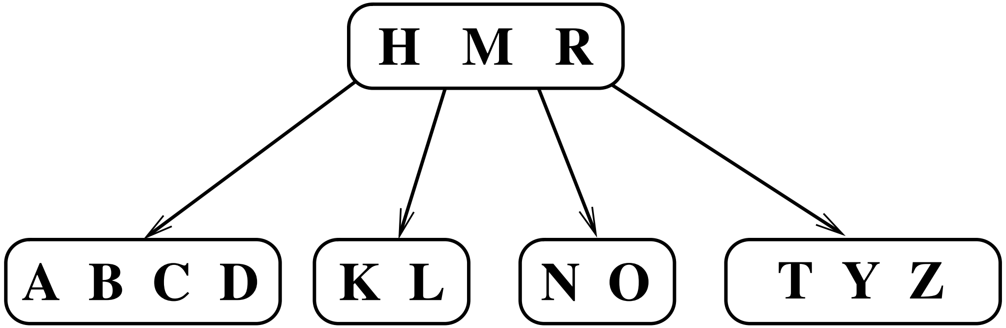
Insert(x)



Obvious approach: do a lookup, put x in leaf where it should be.

- Example: insert E

Insert(x)

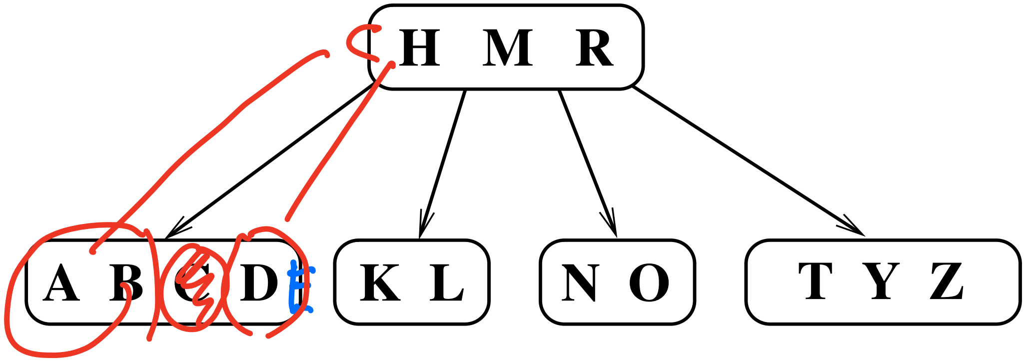


Obvious approach: do a lookup, put x in leaf where it should be.

- Example: insert E

Problem: What if leaf is *full* (already has $2t - 1$ keys)?

Insert(x)



Obvious approach: do a lookup, put x in leaf where it should be.

- Example: insert E

Problem: What if leaf is *full* (already has $2t - 1$ keys)?

Split:

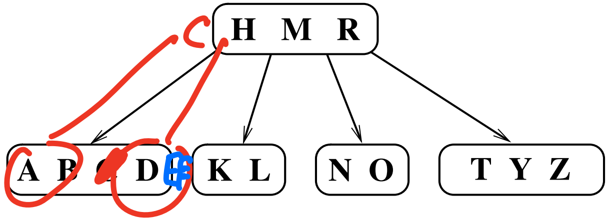
- Only used on *full* nodes (nodes with $2t - 1$ keys) whose parents are *not* full.
- Pull median of its keys up to its parent
- Split remaining $2t - 2$ keys into two nodes of $t - 1$ keys each. Reconnect appropriately.

Insert (continued)

Insert: do a lookup and insert at leaf, but when we encounter a full node on way down, split it.

Insert (continued)

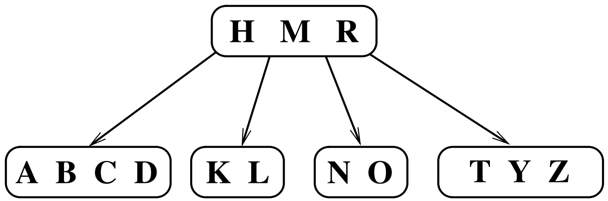
Insert: do a lookup and insert at leaf, but when we encounter a full node on way down, split it.



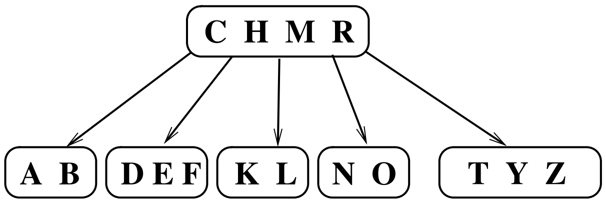
Insert ***E*, *F*** into example.

Insert (continued)

Insert: do a lookup and insert at leaf, but when we encounter a full node on way down, split it.

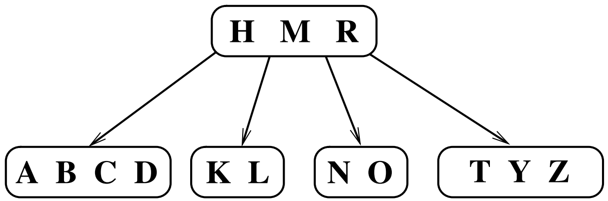


Insert $\boldsymbol{E}, \boldsymbol{F}$ into example.

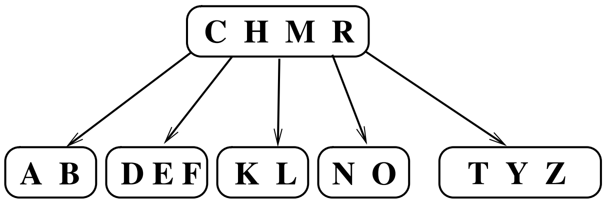


Insert (continued)

Insert: do a lookup and insert at leaf, but when we encounter a full node on way down, split it.

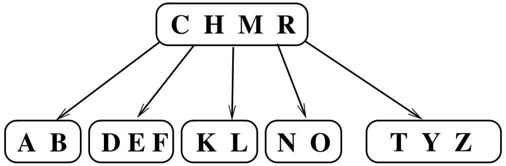


Insert $\boldsymbol{E}, \boldsymbol{F}$ into example.

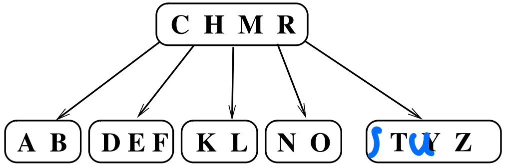


Note: since split *on the way down*, when a node is split, its parent is not full!

Example continued

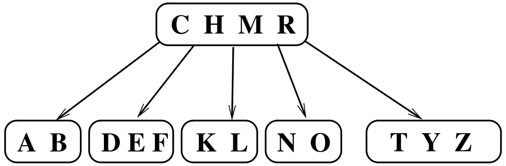


Example continued

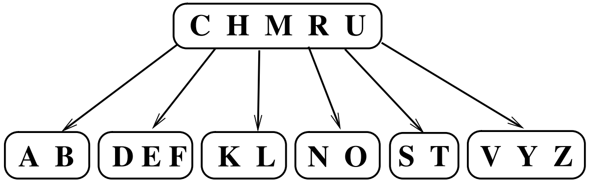


Insert $\boldsymbol{S}, \boldsymbol{U}, \boldsymbol{V}$:

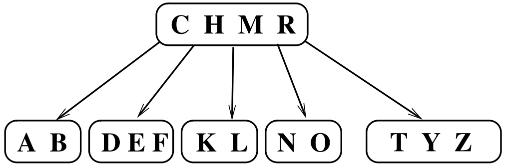
Example continued



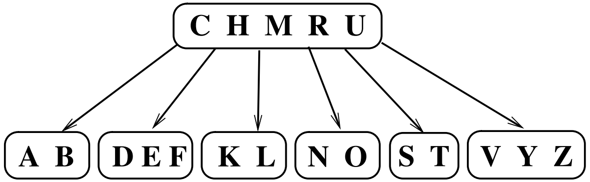
Insert $\boldsymbol{S}, \boldsymbol{U}, \boldsymbol{V}$:



Example continued

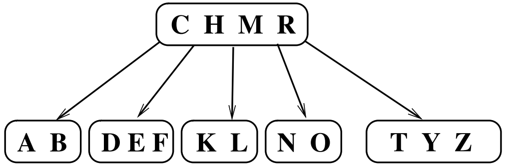


Insert ***S*, *U*, *V***:

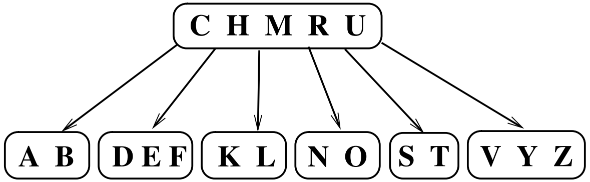


Insert ***P***:

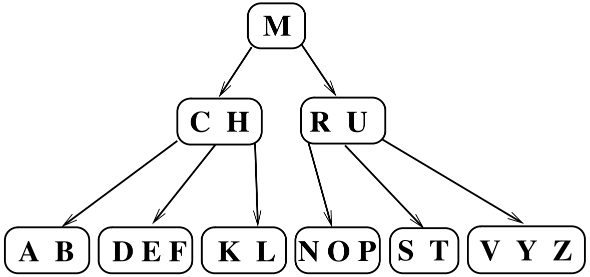
Example continued



Insert $\boldsymbol{S}, \boldsymbol{U}, \boldsymbol{V}$:



Insert $\boldsymbol{P}$:



Insert: Correctness sketch

Induction. Start with a valid B-tree, insert x.

Insert: Correctness sketch

Induction. Start with a valid B-tree, insert $\boldsymbol{x}$.

Third property (all leaves at same depth):

Insert: Correctness sketch

Induction. Start with a valid B-tree, insert $\boldsymbol{x}$.

Third property (all leaves at same depth): Tree grows up. ✓

Insert: Correctness sketch

Induction. Start with a valid B-tree, insert $\boldsymbol{x}$.

Third property (all leaves at same depth): Tree grows up. ✓

First property (all non-leaves other than root have between $t - 1$ and $2t - 1$ keys):

Insert: Correctness sketch

Induction. Start with a valid B-tree, insert x.

Third property (all leaves at same depth): Tree grows up. ✓

First property (all non-leaves other than root have between $t - 1$ and $2t - 1$ keys):

- No split:

Insert: Correctness sketch

Induction. Start with a valid B-tree, insert $\boldsymbol{x}$.

Third property (all leaves at same depth): Tree grows up. ✓

First property (all non-leaves other than root have between $\boldsymbol{t-1}$ and $\boldsymbol{2t-1}$ keys):

- No split: only leaf changes, was not full (or would have split)

Insert: Correctness sketch

Induction. Start with a valid B-tree, insert x.

Third property (all leaves at same depth): Tree grows up. ✓

First property (all non-leaves other than root have between $t-1$ and $2t-1$ keys):

- No split: only leaf changes, was not full (or would have split)
- Split:

Insert: Correctness sketch

Induction. Start with a valid B-tree, insert $\boldsymbol{x}$.

Third property (all leaves at same depth): Tree grows up. ✓

First property (all non-leaves other than root have between $\boldsymbol{t-1}$ and $\boldsymbol{2t-1}$ keys):

- No split: only leaf changes, was not full (or would have split)
- Split: Parent was not full. New nodes have exactly $\boldsymbol{t-1}$ keys.

Insert: Correctness sketch

Induction. Start with a valid B-tree, insert x.

Third property (all leaves at same depth): Tree grows up. ✓

First property (all non-leaves other than root have between $t-1$ and $2t-1$ keys):

- No split: only leaf changes, was not full (or would have split)
- Split: Parent was not full. New nodes have exactly $t-1$ keys.

Second property (correct degrees, subtrees have keys in correct ranges):

Insert: Correctness sketch

Induction. Start with a valid B-tree, insert x.

Third property (all leaves at same depth): Tree grows up. ✓

First property (all non-leaves other than root have between $t - 1$ and $2t - 1$ keys):

- No split: only leaf changes, was not full (or would have split)
- Split: Parent was not full. New nodes have exactly $t - 1$ keys.

Second property (correct degrees, subtrees have keys in correct ranges): Hooked nodes up correctly after split. ✓

B-tree running time

Suppose $\boldsymbol{n}$ keys, depth $\boldsymbol{d}$

B-tree running time

Suppose n keys, depth $d \leq O(\log_t n)$

B-tree running time

Suppose n keys, depth $d \leq O(\log_t n)$

Lookup:

- Binary search on array in each node we pass through

B-tree running time

Suppose n keys, depth $d \leq O(\log_t n)$

Lookup:

- Binary search on array in each node we pass through $\implies O(\log t)$ time per node.

B-tree running time

Suppose $\boldsymbol{n}$ keys, depth $\boldsymbol{d} \leq \boldsymbol{O}(\log_t \boldsymbol{n})$

Lookup:

- Binary search on array in each node we pass through $\implies \boldsymbol{O}(\log \boldsymbol{t})$ time per node.
- Total time $\boldsymbol{O}(\boldsymbol{d} \times \log \boldsymbol{t}) = \boldsymbol{O}(\log_t \boldsymbol{n} \times \log \boldsymbol{t}) = \boldsymbol{O}(\frac{\log \boldsymbol{n}}{\log \boldsymbol{t}} \times \log \boldsymbol{t}) = \boldsymbol{O}(\log \boldsymbol{n})$

B-tree running time

Suppose n keys, depth $d \leq O(\log_t n)$

Lookup:

- Binary search on array in each node we pass through $\implies O(\log t)$ time per node.
- Total time $O(d \times \log t) = O(\log_t n \times \log t) = O(\frac{\log n}{\log t} \times \log t) = O(\log n)$

Insert:

B-tree running time

Suppose n keys, depth $d \leq O(\log_t n)$

Lookup:

- Binary search on array in each node we pass through $\implies O(\log t)$ time per node.
- Total time $O(d \times \log t) = O(\log_t n \times \log t) = O(\frac{\log n}{\log t} \times \log t) = O(\log n)$

Insert:

- Same as insert, but need to split on the way down & insert into leaf

B-tree running time

Suppose n keys, depth $d \leq O(\log_t n)$

Lookup:

- Binary search on array in each node we pass through $\implies O(\log t)$ time per node.
- Total time $O(d \times \log t) = O(\log_t n \times \log t) = O(\frac{\log n}{\log t} \times \log t) = O(\log n)$

Insert:

- Same as insert, but need to split on the way down & insert into leaf
- Total: lookup time + splitting time + time to insert into leaf

B-tree running time

Suppose n keys, depth $d \le O(\log_t n)$

Lookup:

- Binary search on array in each node we pass through $\Longrightarrow O(\log t)$ time per node.
- Total time $O(d \times \log t) = O(\log_t n \times \log t) = O(\frac{\log n}{\log t} \times \log t) = O(\log n)$

Insert:

- Same as insert, but need to split on the way down & insert into leaf
- Total: lookup time + splitting time + time to insert into leaf
    - Insert into leaf:

B-tree running time

Suppose n keys, depth $d \leq O(\log_t n)$

Lookup:

- Binary search on array in each node we pass through $\implies O(\log t)$ time per node.
- Total time $O(d \times \log t) = O(\log_t n \times \log t) = O(\frac{\log n}{\log t} \times \log t) = O(\log n)$

Insert:

- Same as insert, but need to split on the way down & insert into leaf
- Total: lookup time + splitting time + time to insert into leaf
  - Insert into leaf: $O(t)$

B-tree running time

Suppose n keys, depth $d \leq O(\log_t n)$

Lookup:

- Binary search on array in each node we pass through $\implies O(\log t)$ time per node.
- Total time $O(d \times \log t) = O(\log_t n \times \log t) = O(\frac{\log n}{\log t} \times \log t) = O(\log n)$

Insert:

- Same as insert, but need to split on the way down & insert into leaf
- Total: lookup time + splitting time + time to insert into leaf
  - Insert into leaf: $O(t)$
  - Splitting time:

B-tree running time

Suppose n keys, depth $d \leq O(\log_t n)$

Lookup:

- Binary search on array in each node we pass through $\implies$ $O(\log t)$ time per node.
- Total time $O(d \times \log t) = O(\log_t n \times \log t) = O(\frac{\log n}{\log t} \times \log t) = O(\log n)$

Insert:

- Same as insert, but need to split on the way down & insert into leaf
- Total: lookup time + splitting time + time to insert into leaf
  - Insert into leaf: $O(t)$
  - Splitting time: $O(t)$ per split

B-tree running time

Suppose n keys, depth $d \le O(\log_t n)$

Lookup:

- Binary search on array in each node we pass through $\implies$ $O(\log t)$ time per node.
- Total time $O(d \times \log t) = O(\log_t n \times \log t) = O(\frac{\log n}{\log t} \times \log t) = O(\log n)$

Insert:

- Same as insert, but need to split on the way down & insert into leaf
- Total: lookup time + splitting time + time to insert into leaf
  - Insert into leaf: $O(t)$
  - Splitting time: $O(t)$ per split $\implies$ $O(td) = O(t \log_t n)$ total

B-tree running time

Suppose n keys, depth $d \le O(\log_t n)$

Lookup:

- Binary search on array in each node we pass through $\implies O(\log t)$ time per node.
- Total time $O(d \times \log t) = O(\log_t n \times \log t) = O(\frac{\log n}{\log t} \times \log t) = O(\log n)$

Insert:

- Same as insert, but need to split on the way down & insert into leaf
- Total: lookup time + splitting time + time to insert into leaf
  - Insert into leaf: $O(t)$
  - Splitting time: $O(t)$ per split $\implies O(td) = O(t \log_t n)$ total
- $O(t \log_t n) = O(\frac{t}{\log t} \log n)$ total

B-tree notes

Used a lot in databases

- Large t: shallow trees. Fits well with memory hierarchy

B-tree notes

Used a lot in databases

- Large t: shallow trees. Fits well with memory hierarchy

$t = 2$:

- 2-3-4 tree
- Can be implemented as *binary* tree using *red-black trees*

Red-Black Trees

Red-Black Trees: Intro

B-Trees great, but binary is nice: lookups very simple!
Want *binary* balanced tree.

Red-Black Trees: Intro

B-Trees great, but binary is nice: lookups very simple!
Want *binary* balanced tree.

- Classical and super important data structure question
- Many solutions!

Red-Black Trees: Intro

B-Trees great, but binary is nice: lookups very simple!
Want *binary* balanced tree.

- Classical and super important data structure question
- Many solutions!

Most famous: *red-black trees*

- Default in Linux kernel, used to optimize Java HashMap, ...
- Today: Quick overview, connection to 2-3-4 trees.
- *Not* traditional or practical point of view on red-black trees. See book!

2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?

2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?

- *No*: can't have perfect balance!

2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?

- *No*: can't have perfect balance!
- Just need depth $O(\log n)$

2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?

- *No*: can't have perfect balance!
- Just need depth $\boldsymbol{O(\log n)}$

Nodes in 2-3-4 tree have degree 2, 3, or 4

2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?

- *No*: can't have perfect balance!
- Just need depth $O(\log n)$

Nodes in 2-3-4 tree have degree 2, 3, or 4

- Degree 2: good!

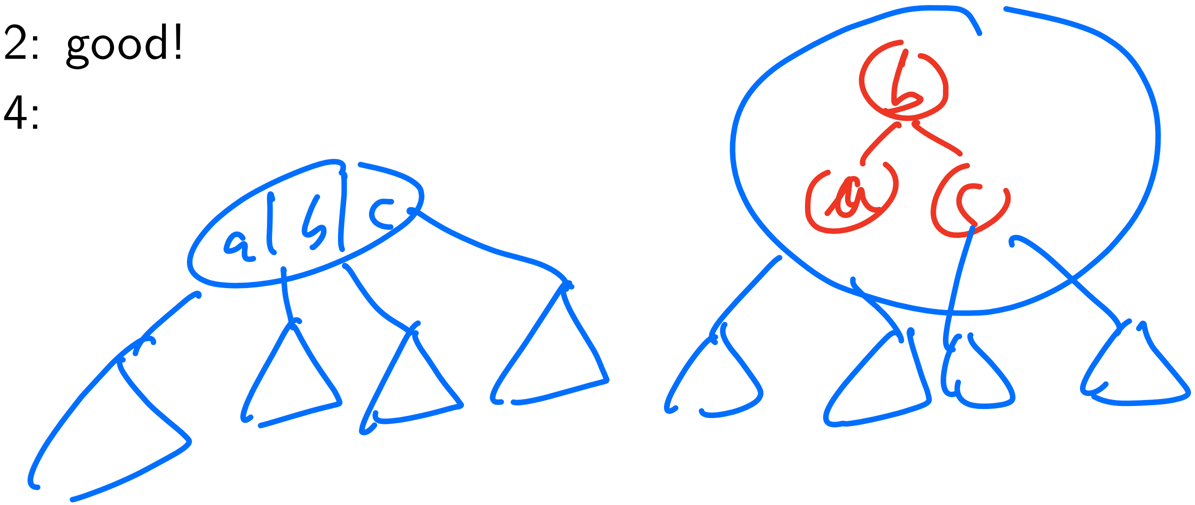
2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?

- *No*: can't have perfect balance!
- Just need depth $O(\log n)$

Nodes in 2-3-4 tree have degree 2, 3, or 4

- Degree 2: good!
- Degree 4:



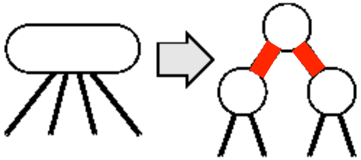
2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?

- *No*: can't have perfect balance!
- Just need depth $O(\log n)$

Nodes in 2-3-4 tree have degree 2, 3, or 4

- Degree 2: good!
- Degree 4:



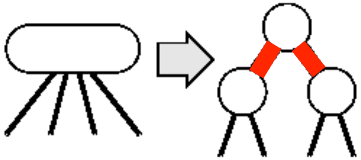
2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?

- *No*: can't have perfect balance!
- Just need depth $O(\log n)$

Nodes in 2-3-4 tree have degree 2, 3, or 4

- Degree 2: good!
- Degree 4:



- Degree 3:

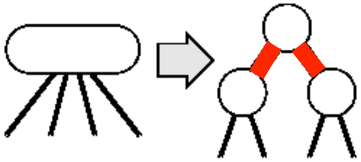
2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?

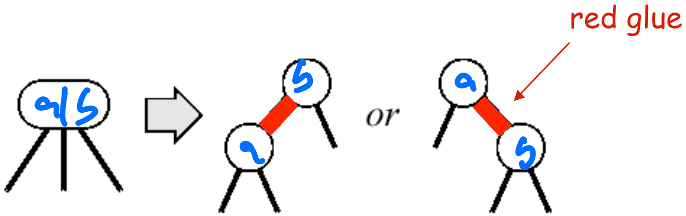
- *No*: can't have perfect balance!
- Just need depth $O(\log n)$

Nodes in 2-3-4 tree have degree 2, 3, or 4

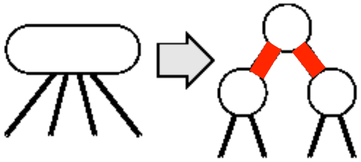
- Degree 2: good!
- Degree 4:

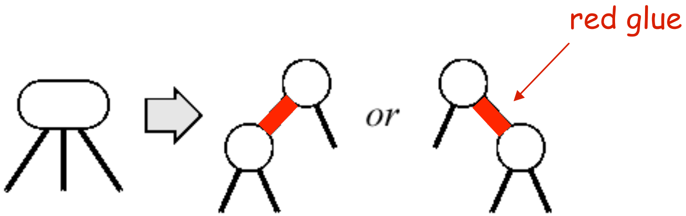


- Degree 3:

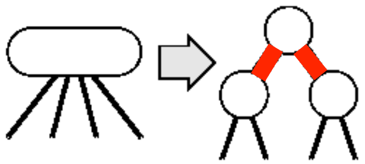


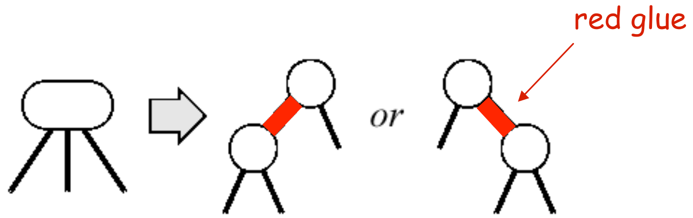
Important Properties





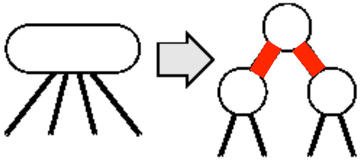
Important Properties

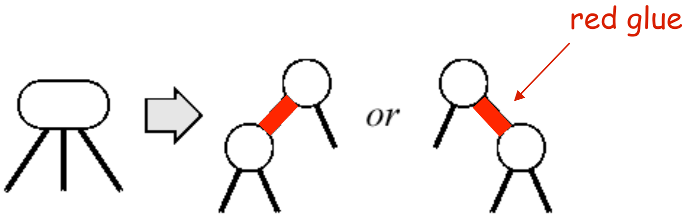




1. Never have two red edges in a row.
    - Red edge is "internal", never have more than one "internal" edge in a row.

Important Properties

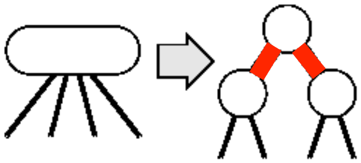


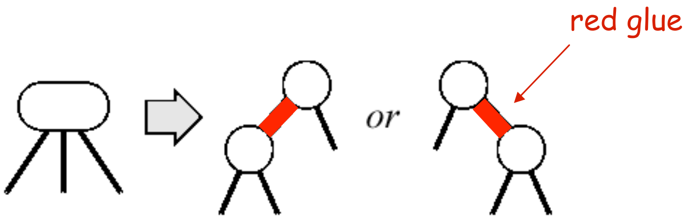


1. Never have two red edges in a row.
   - Red edge is "internal", never have more than one "internal" edge in a row.

2. Every leaf has same number of *black* edges on path to root (*black-depth*)
   - Each black edge is a 2-3-4 tree edge
   - All leaves in 2-3-4 tree at same distance from root

Important Properties



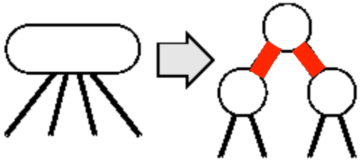


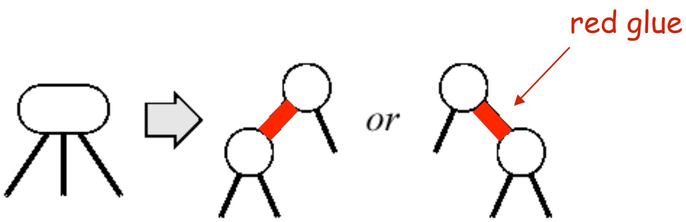
1. Never have two red edges in a row.
   - Red edge is "internal", never have more than one "internal" edge in a row.

2. Every leaf has same number of *black* edges on path to root (*black-depth*)
   - Each black edge is a 2-3-4 tree edge
   - All leaves in 2-3-4 tree at same distance from root

$\Longrightarrow$ path from root to deepest leaf $\leq \mathbf{2} \times$ path to shallowest leaf

Important Properties





1. Never have two red edges in a row.
   - Red edge is "internal", never have more than one "internal" edge in a row.

2. Every leaf has same number of *black* edges on path to root (*black-depth*)
   - Each black edge is a 2-3-4 tree edge
   - All leaves in 2-3-4 tree at same distance from root

$\implies$ path from root to deepest leaf $\leq \mathbf{2} \times$ path to shallowest leaf

$\implies$ depth $\leq \boldsymbol{O}(\log n)$

Insert

Want to insert while preserving two properties.

Insert

Want to insert while preserving two properties.

2-3-4 trees: split full nodes on way down.

Insert

Want to insert while preserving two properties.
2-3-4 trees: split full nodes on way down.

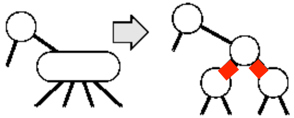
Easy cases:

Insert

Want to insert while preserving two properties.

2-3-4 trees: split full nodes on way down.

Easy cases:

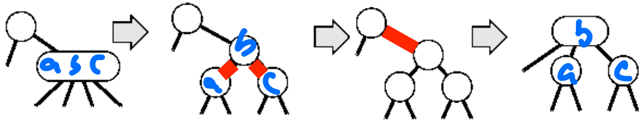


Insert

Want to insert while preserving two properties.

2-3-4 trees: split full nodes on way down.

Easy cases:

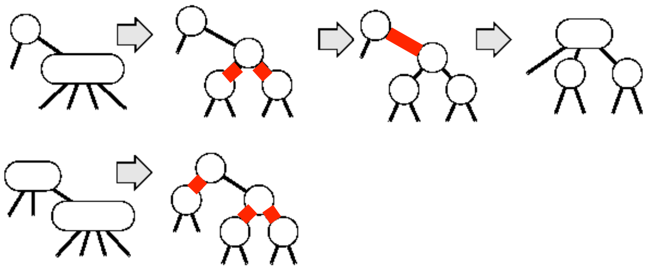


Insert

Want to insert while preserving two properties.

2-3-4 trees: split full nodes on way down.

Easy cases:

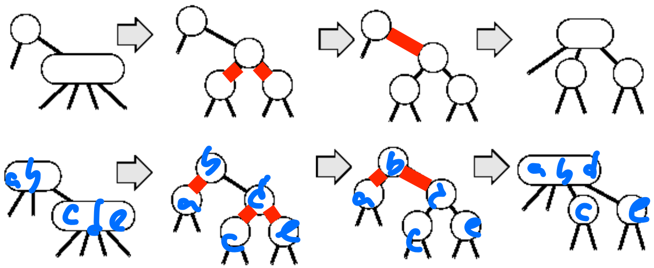


Insert

Want to insert while preserving two properties.

2-3-4 trees: split full nodes on way down.

Easy cases:

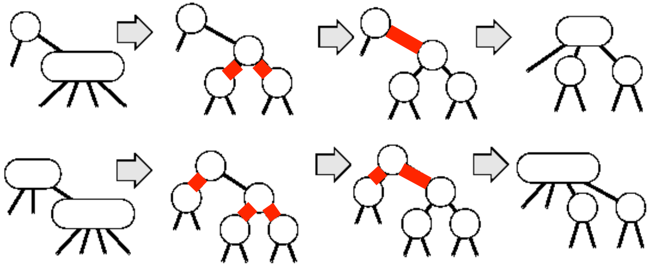


Insert

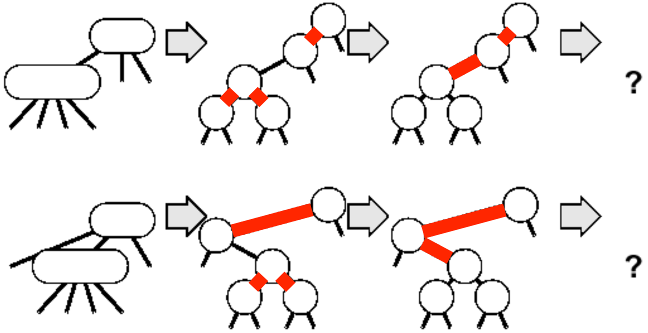
Want to insert while preserving two properties.

2-3-4 trees: split full nodes on way down.

Easy cases:



Harder cases:

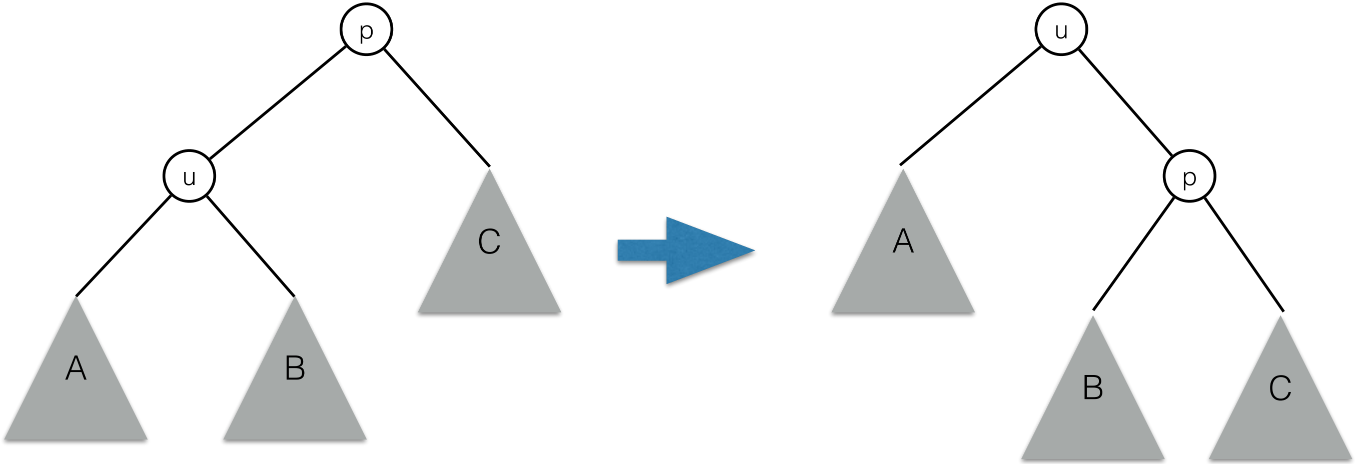


Tree Rotations

Used in many different tree constructions.

Tree Rotations

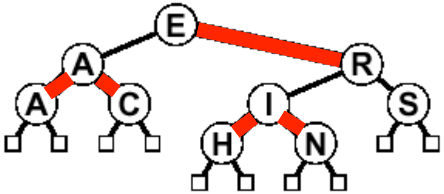
Used in many different tree constructions.



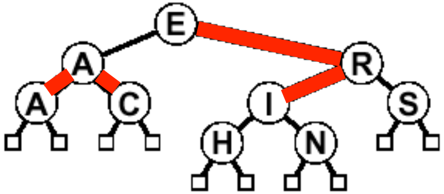
Using Rotations

Can use rotations to "fix" hard cases. Example:

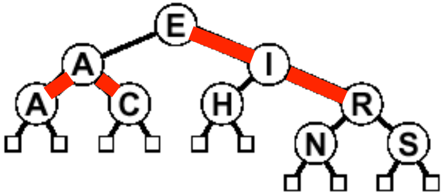
inserting G



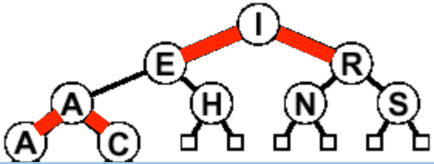
change colors



right rotate R →



left rotate E →



End

A few more complications to deal with – see lecture notes, textbook.

End

A few more complications to deal with – see lecture notes, textbook.

Main points:

- Red-Black trees can be thought of as a binary implementation of 2-3-4 trees
- Approximately balanced, so $\boldsymbol{O(\log n)}$ lookup time
- Insert time (basically) same as 2-3-4 tree, so also $\boldsymbol{O(\log n)}$.
- See book for direct approach (not through 2-3-4 trees).