Lecture 6: Balanced Search Trees

Michael Dinitz

September 16, 2021
601.433/633 Introduction to Algorithms

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* n keys/objects

Approach 1: Sorted array

- Lookup:

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

- Lookup: $\mathbf{O(\log n)}$

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

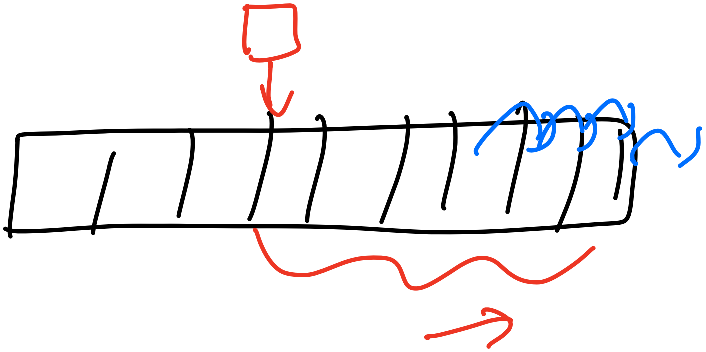
- Lookup: $\mathbf{O(\log n)}$
- Insert:

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

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



Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

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

Approach 2: Unsorted (linked) list

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

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

Approach 2: Unsorted (linked) list

- Insert:

Obvious Approaches

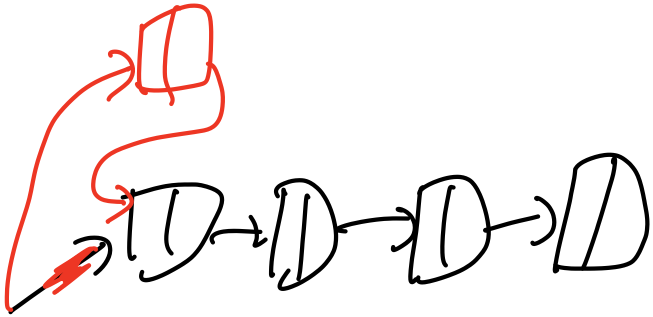
Reminder: all running times for *worst case*

Approach 1: Sorted array

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

Approach 2: Unsorted (linked) list

- Insert: $\mathbf{O(1)}$



Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

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

Approach 2: Unsorted (linked) list

- Insert: $\mathbf{O(1)}$
- Lookup:

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

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

Approach 2: Unsorted (linked) list

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

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

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

Approach 2: Unsorted (linked) list

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

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

Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array

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

Approach 2: Unsorted (linked) list

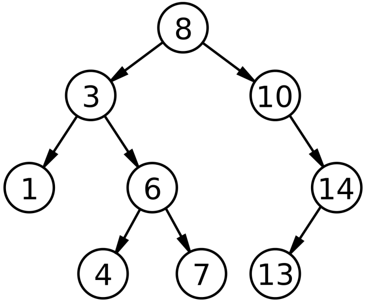
- Insert: $\mathbf{O(1)}$
- Lookup: $\mathbf{\Omega(n)}$

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

Binary Search Tree Review

Binary search tree:

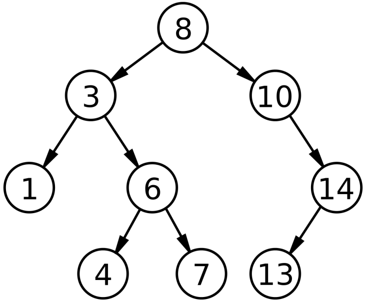
- All nodes have at most **2** children
- Each node stores (key, object) pair
- All descendants to left have smaller keys
- All descendants to the right have larger keys



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**):

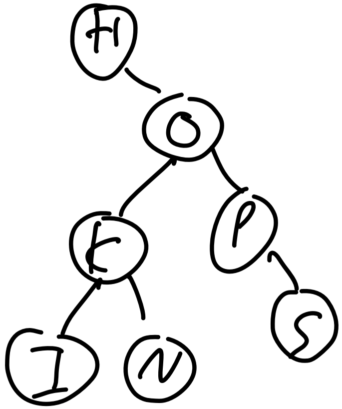
- If tree empty, put **x** at root
- Else if **x** < **root.key** recursively insert into left child
- Else (if **x** > **root.key**) recursively insert into right child

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

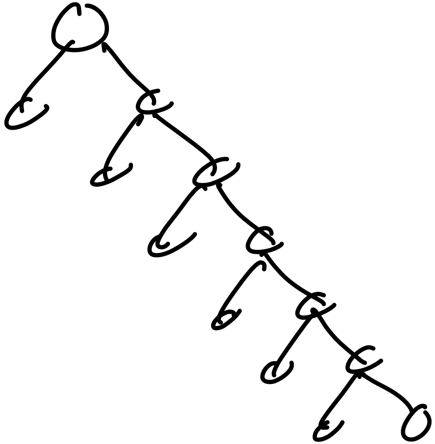
(Worst-case) Running time: if depth $\mathbf{d}$, then $\boldsymbol{\Theta}(\mathbf{d})$

Simply Binary Search Tree: Analysis

Pluses: easy to implement

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

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



Simply Binary Search Tree: Analysis

Pluses: easy to implement

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

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

Want to make tree *balanced*.

Simply Binary Search Tree: Analysis

Pluses: easy to implement

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

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

Want to make tree *balanced*.

Rest of today:

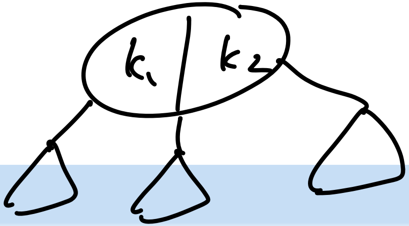
- B-trees: perfect balance, not binary
- Red-black trees: approximate balance, binary
- Turn out to be related!

B-Trees

B-tree Definition

Parameter $\mathbf{t} \geq \mathbf{2}$.

B-tree Definition



Parameter $\mathbf{t} \geq 2$.

Definition (B-tree with parameter $\mathbf{t}$)

1. Each node has between $\mathbf{t}-1$ and $2\mathbf{t}-1$ keys in it (except the root has between 1 and $2\mathbf{t}-1$ 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 $\mathbf{v}$ is a node with keys $[\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_k]$ and the children are $[\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{k+1}]$, then the tree rooted at $\mathbf{v}_i$ contains only keys that are at least $\mathbf{a}_{i-1}$ and at most $\mathbf{a}_i$ (except the the edge cases: the tree rooted at $\mathbf{v}_1$ has keys less than $\mathbf{a}_1$, and the tree rooted at $\mathbf{v}_{k+1}$ has keys at least $\mathbf{a}_k$).
3. All leaves are at the same depth.

B-tree Definition

Parameter $\mathbf{t} \geq \mathbf{2}$.

Definition (B-tree with parameter $\mathbf{t}$)

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

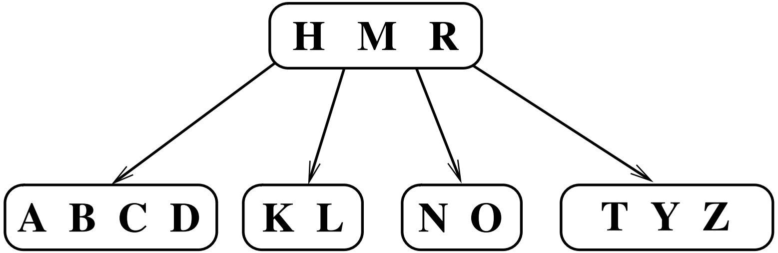
When $\mathbf{t = 2}$ known as a *2-3-4 tree*, since # children either 2, 3, or 4

B-tree: Example

t = **3**:

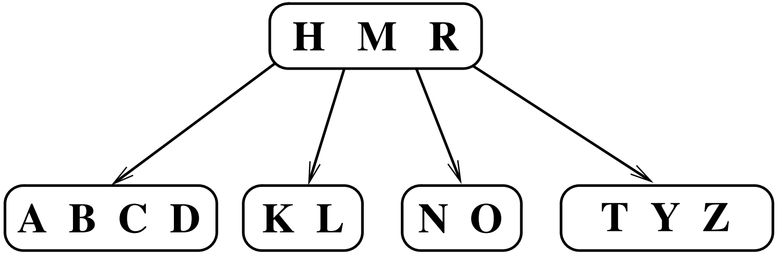
- Root has between **1** and **5** keys, non-roots have between **2** and **5** keys
- Non-leaves have between **3** and **6** children (root can have fewer).

and non-root

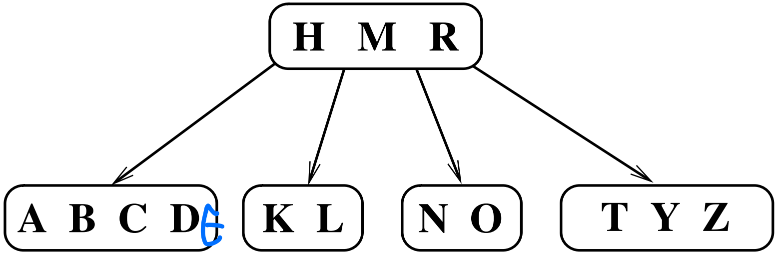


Lookups

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



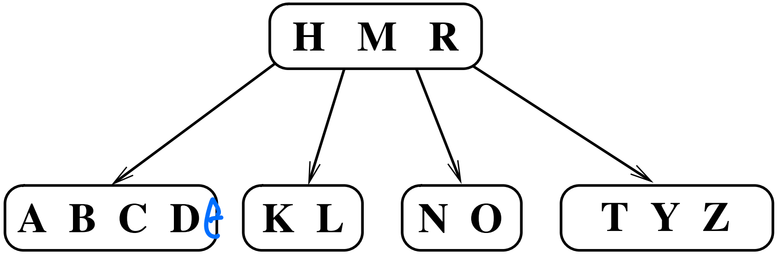
Insert(**x**)



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

- Example: insert **E**

Insert(x)

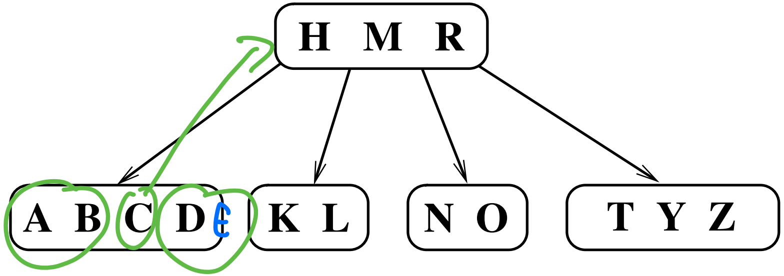


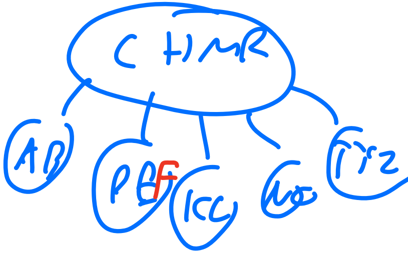
Obvious approach: do a lookup, put **x** in leaf where it should be.

- Example: insert **E**

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

Insert($\mathbf{x}$)





Obvious approach: do a lookup, put $\mathbf{x}$ in leaf where it should be.

- Example: insert **E**

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

Split:

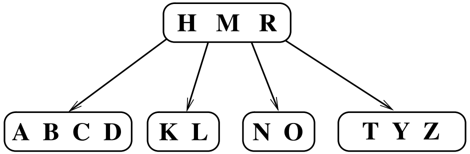
- Only used on *full* nodes (nodes with $\mathbf{2t-1}$ keys) whose parents are *not* full.
- Pull median of its keys up to its parent
- Split remaining $\mathbf{2t-2}$ keys into two nodes of $\mathbf{t-1}$ keys each. Reconnect appropriately.

Insert (continued)

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

Insert (continued)

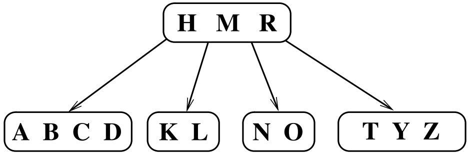
Insert: do a lookup and insert at leaf, but when we encounter a full node on way down, split it.



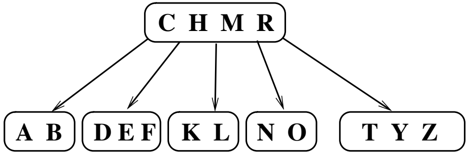
Insert **E, F** into example.

Insert (continued)

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

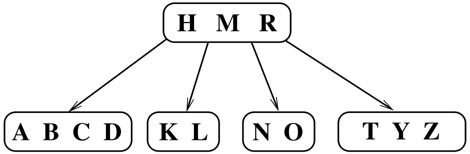


Insert **E, F** into example.

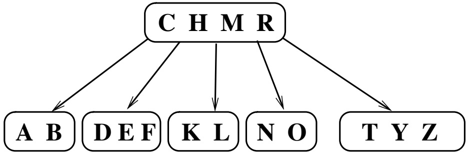


Insert (continued)

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

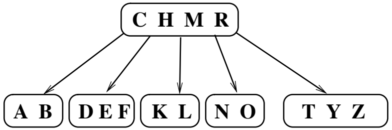


Insert **E, F** into example.

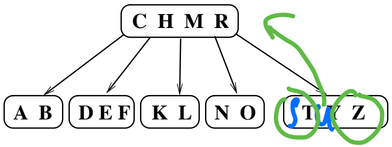


Note: since split *on the way down*, when a node is split, its parent is not full!

Example continued

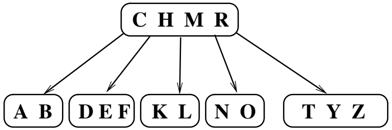


Example continued

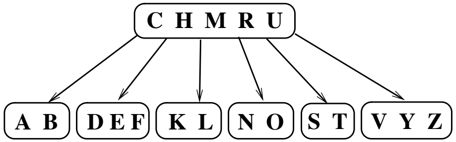


Insert **S, U, V**:

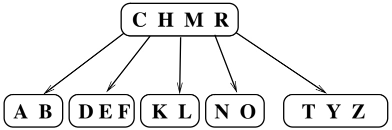
Example continued



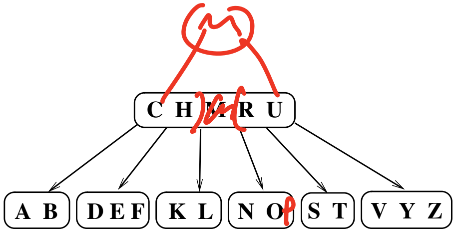
Insert **S, U, V**:



Example continued

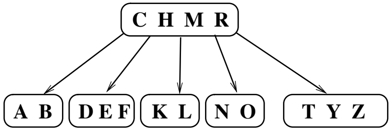


Insert **S, U, V**:

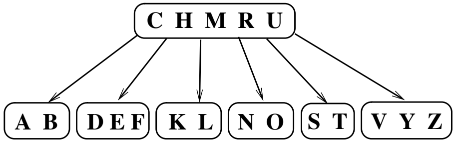


Insert **P**:

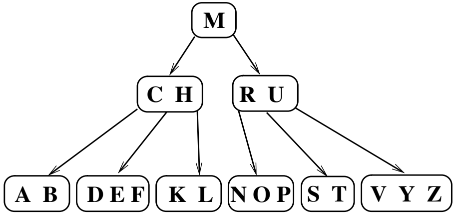
Example continued



Insert **S, U, V**:



Insert **P**:



Insert: Correctness sketch

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

Insert: Correctness sketch

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

Third property (all leaves at same depth):

Insert: Correctness sketch

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

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

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 $\mathbf{t-1}$ and $\mathbf{2t-1}$ keys):

Insert: Correctness sketch

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

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

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

- No split:

Insert: Correctness sketch

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

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

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

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

Insert: Correctness sketch

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

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

First property (all non-leaves other than root have between $\mathbf{t-1}$ and $\mathbf{2t-1}$ 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 $\mathbf{x}$.

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

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

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

Insert: Correctness sketch

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

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

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

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

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

Insert: Correctness sketch

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

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

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

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

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

B-tree running time

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

B-tree running time

Suppose **n** keys, depth $\mathbf{d} \leq \mathbf{O}(\log_t \mathbf{n})$ B/c all nodes have deg. $\geq t$

B-tree running time

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

Lookup:

- Binary search on array in each node we pass through

B-tree running time

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

Lookup:

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

B-tree running time

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

Lookup:

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

B-tree running time

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

Lookup:

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

Insert:

B-tree running time

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

Lookup:

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

Insert:

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

B-tree running time

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

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

Lookup:

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

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

Lookup:

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

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: $\mathbf{O}(\mathbf{t})$

B-tree running time

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

Lookup:

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

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: $\mathbf{O(t)}$
  - Splitting time:

B-tree running time

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

Lookup:

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

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: $\mathbf{O}(\mathbf{t})$
  - Splitting time: $\mathbf{O}(\mathbf{t})$ per split

B-tree running time

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

Lookup:

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

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: $\mathbf{O}(\mathbf{t})$
  - Splitting time: $\mathbf{O}(\mathbf{t})$ per split $\Longrightarrow$ $\mathbf{O}(\mathbf{td}) = \mathbf{O}(\mathbf{t} \log_t \mathbf{n})$ total

B-tree running time

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

Lookup:

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

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: $\mathbf{O}(\mathbf{t})$
    - Splitting time: $\mathbf{O}(\mathbf{t})$ per split $\implies$ $\mathbf{O}(\mathbf{td}) = \mathbf{O}(\mathbf{t} \log_t \mathbf{n})$ total
- $\mathbf{O}(\mathbf{t} \log_t \mathbf{n}) = \mathbf{O}(\frac{\mathbf{t}}{\log \mathbf{t}} \log \mathbf{n})$ 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 $\mathbf{O(\log n)}$

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 $\mathbf{O(\log n)}$

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 $\mathbf{O(\log n)}$

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

- Degree 2: good!

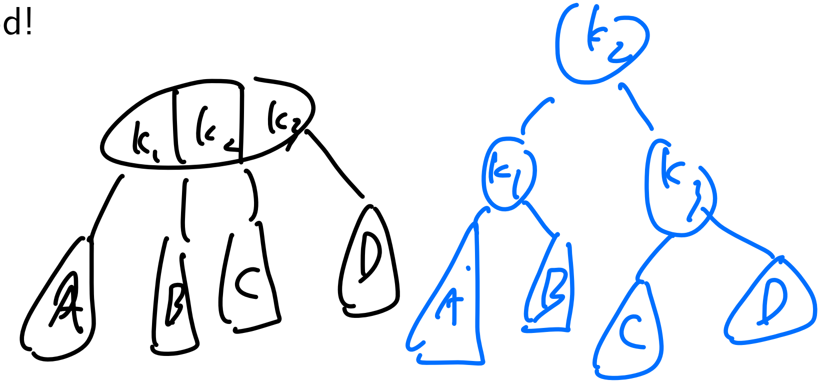
2-3-4 trees to binary

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 $\mathbf{O(\log n)}$

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

- Degree 2: good!
- Degree 4:



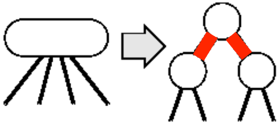
2-3-4 trees to binary

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 $\mathbf{O(\log n)}$

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

- Degree 2: good!
- Degree 4:



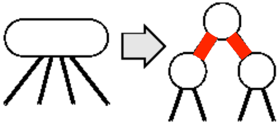
2-3-4 trees to binary

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

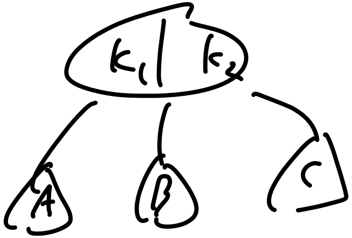
- *No*: can't have perfect balance!
- Just need depth $\mathbf{O(\log n)}$

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

- Degree 2: good!
- Degree 4:



- Degree 3:



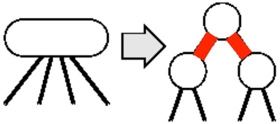
2-3-4 trees to binary

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

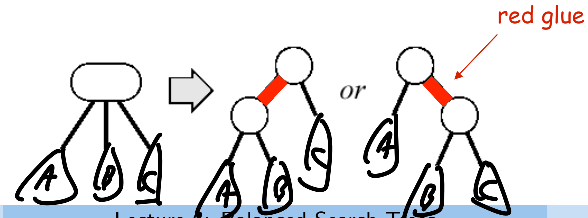
- *No*: can't have perfect balance!
- Just need depth $\mathbf{O(\log n)}$

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

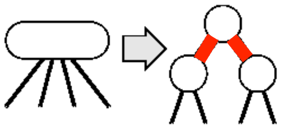
- Degree 2: good!
- Degree 4:

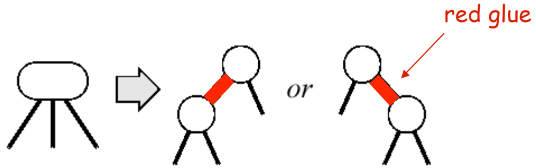


- Degree 3:

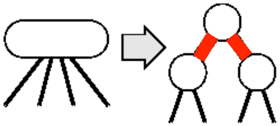


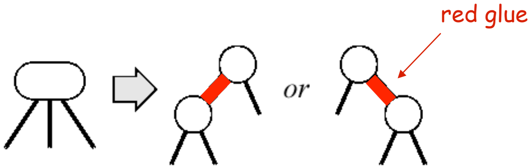
Important Properties





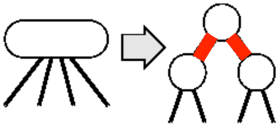
Important Properties

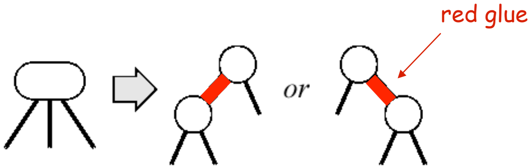




1. Never have two red edges in a row.
   - Red edge is "internal", never have more than one "internal" edge in a row.

Important Properties

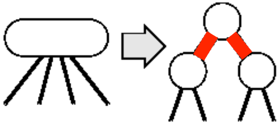


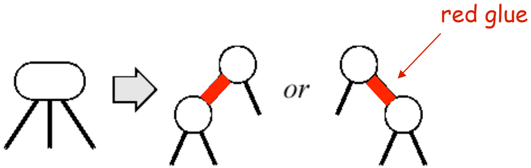


1. Never have two red edges in a row.
   - Red edge is "internal", never have more than one "internal" edge in a row.

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



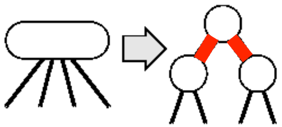


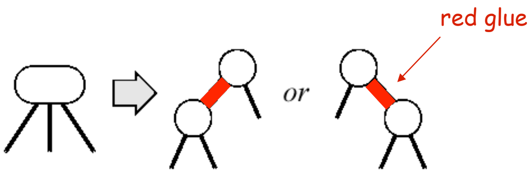
1. Never have two red edges in a row.
   - Red edge is "internal", never have more than one "internal" edge in a row.

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 $\leq$ **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

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

$\Longrightarrow$ depth $\leq \mathbf{O(\log n)}$

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.

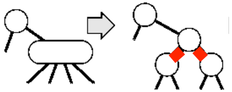
Easy cases:

Insert

Want to insert while preserving two properties.

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

Easy cases:

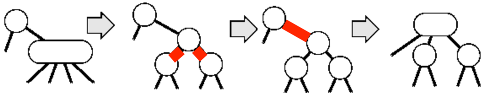


Insert

Want to insert while preserving two properties.

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

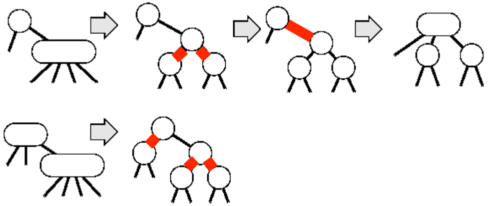
Easy cases:



Insert

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

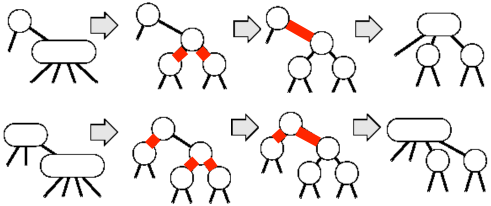
Easy cases:



Insert

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

Easy cases:

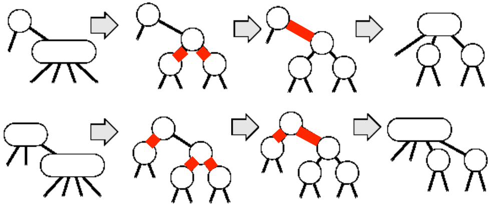


Insert

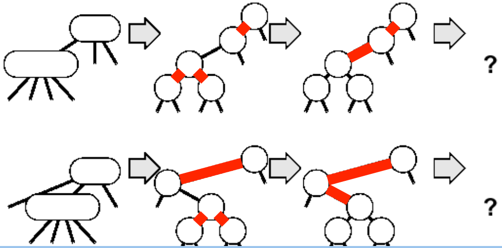
Want to insert while preserving two properties.

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

Easy cases:



Harder cases:

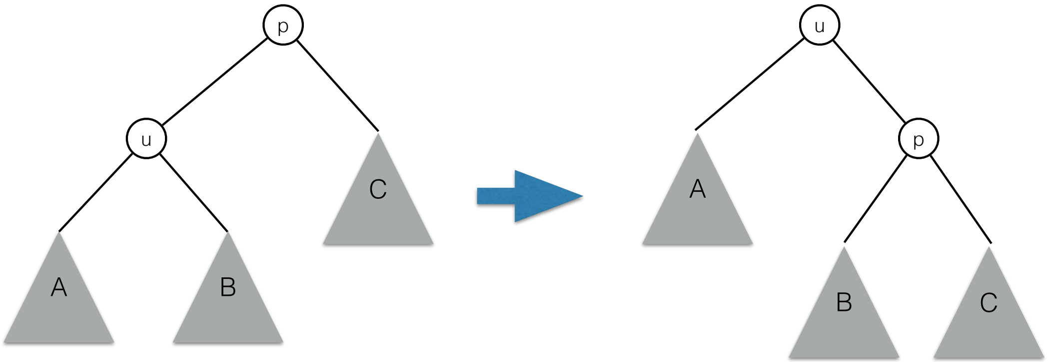


Tree Rotations

Used in many different tree constructions.

Tree Rotations

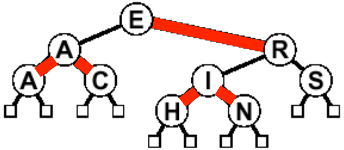
Used in many different tree constructions.



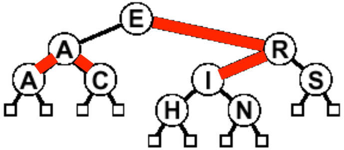
Using Rotations

Can use rotations to “fix” hard cases. Example:

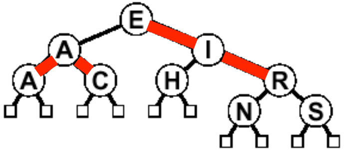
inserting G



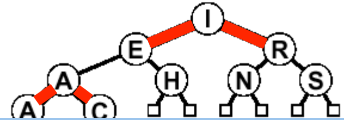
change colors



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 $\mathbf{O(\log n)}$ lookup time
- Insert time (basically) same as 2-3-4 tree, so also $\mathbf{O(\log n)}$.
- See book for direct approach (not through 2-3-4 trees).