Algorithms and Datastructures

Balanced Trees (AVL-Trees, (a,b)-Trees, Red-Black-Trees)

Prof. Dr. Rolf Backofen

Bioinformatics Group / Department of Computer Science

Algorithms and Datastructures, January 2017

Structure

Structure

Structure

Binary search tree:

Binary search tree:

- With `BinarySearchTree` we could perform an `lookup` or `insert` in $O(d)$, with d being the depth of the tree

Balanced Trees
Motivation

Binary search tree:

- With `BinarySearchTree` we could perform an `lookup` or `insert` in $O(d)$, with d being the `depth` of the tree
- Best case: $d = O(\log n)$

Balanced Trees

Motivation

Binary search tree:

- With `BinarySearchTree` we could perform an `lookup` or `insert` in $O(d)$, with d being the depth of the tree
- Best case: $d = O(\log n)$
  - If the keys are inserted randomly

Balanced Trees

Motivation

Binary search tree:

- With `BinarySearchTree` we could perform an `lookup` or `insert` in $O(d)$, with d being the depth of the tree
- Best case: $d = O(\log n)$
  - If the keys are inserted randomly
- Worst case: $d = O(n)$

Balanced Trees

Motivation

Binary search tree:

- With `BinarySearchTree` we could perform an `lookup` or `insert` in $O(d)$, with d being the depth of the tree
- Best case: $d = O(\log n)$
  - If the keys are inserted randomly
- Worst case: $d = O(n)$
  - if the keys are inserted in ascending / descending order $(20, 19, 18, \dots)$

Gnarley trees:

Balanced Trees

Motivation

Gnarley trees:

- http://people.ksp.sk/~kuko/bak

Balanced Trees

Motivation

Gnarley trees:

- http://people.ksp.sk/~kuko/bak

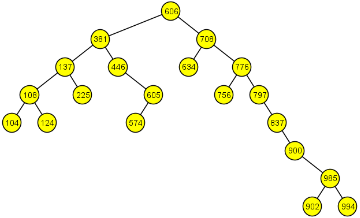


Figure: Binary search tree with random insert [Gna]

Balanced Trees

Motivation

Gnarley trees:

- http://people.ksp.sk/~kuko/bak

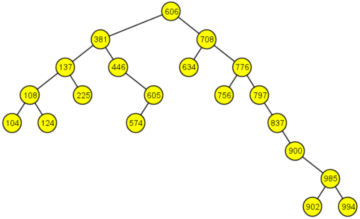


Figure: Binary search tree with random insert [Gna]

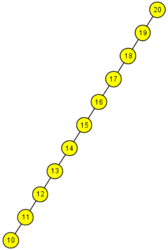


Figure: Binary search tree with descending insert [Gna]

Balanced trees:

Balanced Trees

Motivation

Balanced trees:

- We do not want to rely on certain properties of our key set

Balanced trees:

- We do not want to rely on certain properties of our key set
- We explicitly want a depth of $O(\log n)$

Balanced trees:

- We do not want to rely on certain properties of our key set
- We explicitly want a depth of $O(\log n)$
- We rebalance the tree from time to time

Balanced Trees

Motivation

How do we get a depth of $O(\log n)$?

Balanced Trees

Motivation

How do we get a depth of $O(\log n)$?

- **AVL-Tree:**

How do we get a depth of $O(\log n)$?

- **AVL-Tree:**
  - Binary tree with 2 children per node

Balanced Trees

Motivation

How do we get a depth of $O(\log n)$?

- **AVL-Tree:**
  - Binary tree with 2 children per node
  - Balancing via "rotation"

Balanced Trees
Motivation

How do we get a depth of $O(\log n)$?

- **AVL-Tree:**
  - Binary tree with 2 children per node
  - Balancing via "rotation"
- **(a,b)-Tree** or **B-Tree:**

Balanced Trees
Motivation

How do we get a depth of $O(\log n)$?

- **AVL-Tree:**
  - Binary tree with 2 children per node
  - Balancing via "rotation"
- **(a,b)-Tree** or **B-Tree:**
  - Node have between a and b children

Balanced Trees
Motivation

How do we get a depth of $O(\log n)$?

- **AVL-Tree:**
  - Binary tree with 2 children per node
  - Balancing via "rotation"
- **(a,b)-Tree** or **B-Tree:**
  - Node have between a and b children
  - Balancing through splitting and merging nodes

Balanced Trees

Motivation

How do we get a depth of $O(\log n)$?

- **AVL-Tree:**
  - Binary tree with 2 children per node
  - Balancing via “rotation”
- **(a,b)-Tree** or **B-Tree:**
  - Node have between a and b children
  - Balancing through splitting and merging nodes
  - Used in data bases and file systems

Balanced Trees
Motivation

How do we get a depth of $O(\log n)$?

- **AVL-Tree:**
  - Binary tree with 2 children per node
  - Balancing via "rotation"
- **(a,b)-Tree** or **B-Tree:**
  - Node have between a and b children
  - Balancing through splitting and merging nodes
  - Used in data bases and file systems
- **Red-Black-Tree:**

Balanced Trees
Motivation

How do we get a depth of $O(\log n)$?

- **AVL-Tree:**
  - Binary tree with 2 children per node
  - Balancing via "rotation"
- **(a,b)-Tree** or **B-Tree:**
  - Node have between a and b children
  - Balancing through splitting and merging nodes
  - Used in data bases and file systems
- **Red-Black-Tree:**
  - Binary tree with "black" and "red" nodes

Balanced Trees

Motivation

How do we get a depth of $O(\log n)$?

- **AVL-Tree:**
  - Binary tree with 2 children per node
  - Balancing via "rotation"
- **(a,b)-Tree** or **B-Tree:**
  - Node have between a and b children
  - Balancing through splitting and merging nodes
  - Used in data bases and file systems
- **Red-Black-Tree:**
  - Binary tree with "black" and "red" nodes
  - Balancing through "rotation" and "recoloring"

How do we get a depth of $O(\log n)$?

- **AVL-Tree:**
  - Binary tree with 2 children per node
  - Balancing via "rotation"
- **(a,b)-Tree** or **B-Tree:**
  - Node have between a and b children
  - Balancing through splitting and merging nodes
  - Used in data bases and file systems
- **Red-Black-Tree:**
  - Binary tree with "black" and "red" nodes
  - Balancing through "rotation" and "recoloring"
  - Can be interpreted as (2, 4)-tree

Balanced Trees

Motivation

How do we get a depth of $O(\log n)$?

- **AVL-Tree:**
  - Binary tree with 2 children per node
  - Balancing via "rotation"
- **(a,b)-Tree** or **B-Tree:**
  - Node have between a and b children
  - Balancing through splitting and merging nodes
  - Used in data bases and file systems
- **Red-Black-Tree:**
  - Binary tree with "black" and "red" nodes
  - Balancing through "rotation" and "recoloring"
  - Can be interpreted as (2, 4)-tree
  - Used in C++ std::map, Java SortedMap

Structure

AVL-Tree:

AVL-Tree:

- Gregory Maximovich **A**delson-**V**elskii, Yevgeniy Mikhailovlovich **L**andis (1963)

AVL-Tree:

- Gregory Maximovich **A**delson-**V**elskii, Yevgeniy Mikhailovlovich **L**andis (1963)
- Search tree with modified `insert` and `remove` operations while satisfying a depth condition

AVL-Tree:

- Gregory Maximovich **A**delson-**V**elskii, Yevgeniy Mikhailovlovich **L**andis (1963)
- Search tree with modified `insert` and `remove` operations while satisfying a `depth` condition
- Prevents degeneration of the search tree

AVL-Tree:

- Gregory Maximovich **A**delson-**V**elskii, Yevgeniy Mikhailovlovich **L**andis (1963)
- Search tree with modified `insert` and `remove` operations while satisfying a depth condition
- Prevents degeneration of the search tree
- Height difference of left and right subtree is at maximum one

AVL-Tree:

- Gregory Maximovich **A**delson-**V**elskii, Yevgeniy Mikhailovlovich **L**andis (1963)
- Search tree with modified `insert` and `remove` operations while satisfying a depth condition
- Prevents degeneration of the search tree
- Height difference of left and right subtree is at maximum one
- With that the height of the search tree is always $O(\log n)$

AVL-Tree:

- Gregory Maximovich **A**delson-**V**elskii, Yevgeniy Mikhailovlovich **L**andis (1963)
- Search tree with modified `insert` and `remove` operations while satisfying a depth condition
- Prevents degeneration of the search tree
- Height difference of left and right subtree is at maximum one
- With that the height of the search tree is always $O(\log n)$
- We can perform all basic operations in $O(\log n)$

Balanced Trees

AVL-Tree

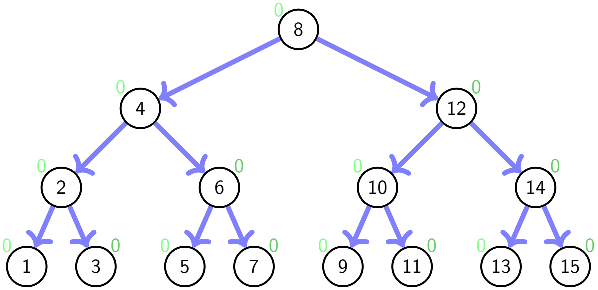


Figure: Example of an AVL-Tree

Balanced Trees

AVL-Tree

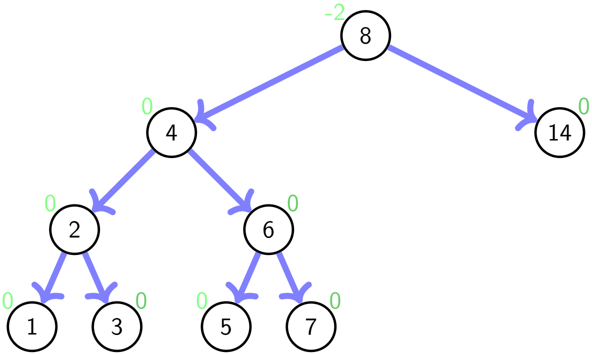


Figure: **Not** an AVL-Tree

Balanced Trees
AVL-Tree

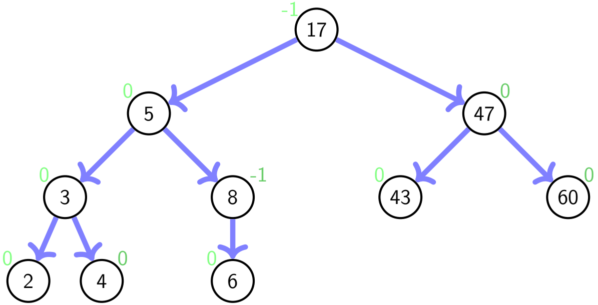


Figure: Another example of an AVL-Tree

Balanced Trees

AVL-Tree - Rebalancing

Rotation:

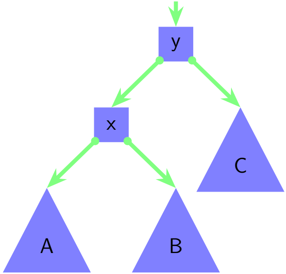


Figure: Before rotating

⇒

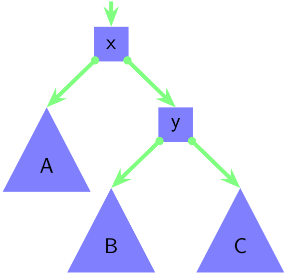


Figure: After rotating

Balanced Trees

AVL-Tree - Rebalancing

Rotation:

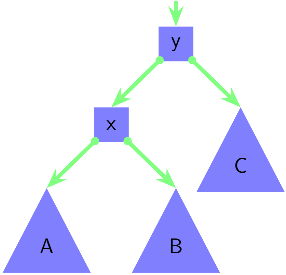


Figure: Before rotating

$\Rightarrow$

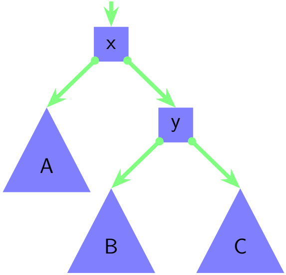


Figure: After rotating

- Central operation of rebalancing

Balanced Trees

AVL-Tree - Rebalancing

Rotation:

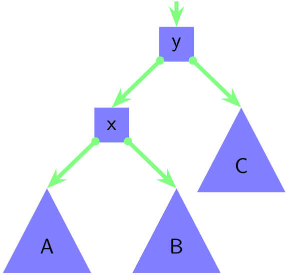


Figure: Before rotating

$\Rightarrow$

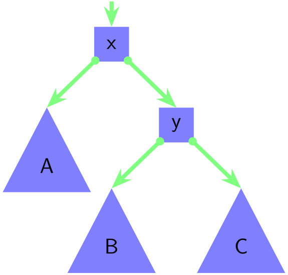


Figure: After rotating

- Central operation of rebalancing
- After rotation to the right:

Balanced Trees

AVL-Tree - Rebalancing

Rotation:

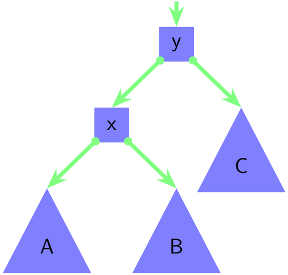


Figure: Before rotating

$\Rightarrow$

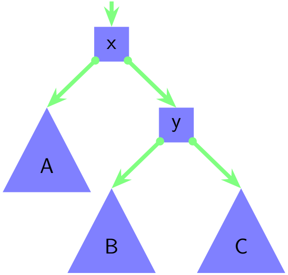


Figure: After rotating

- Central operation of rebalancing
- After rotation to the right:
  - Subtree A is a layer higher and subtree C a layer lower

Balanced Trees

AVL-Tree - Rebalancing

Rotation:

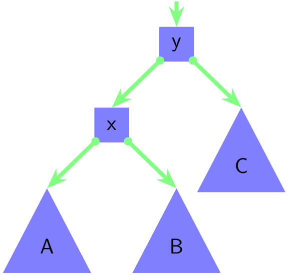


Figure: Before rotating

⇒

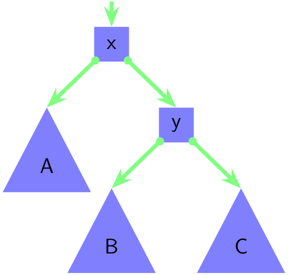


Figure: After rotating

- Central operation of rebalancing
- After rotation to the right:
  - Subtree A is a layer higher and subtree C a layer lower
  - The parent child relations between nodes x and y have been swapped

AVL-Tree:

Balanced Trees

AVL-Tree - Rebalancing

AVL-Tree:

- If a height difference of ± 2 occurs on an `insert` or `remove` operation the tree is rebalanced

Balanced Trees

AVL-Tree - Rebalancing

AVL-Tree:

- If a height difference of ± 2 occurs on an `insert` or `remove` operation the tree is rebalanced
- Many different cases of rebalancing

Balanced Trees

AVL-Tree - Rebalancing

AVL-Tree:

- If a height difference of ± 2 occurs on an `insert` or `remove` operation the tree is rebalanced
- Many different cases of rebalancing
- **Example:** `insert` of $1, 2, 3, \ldots$

Balanced Trees

AVL-Tree:

- If a height difference of ± 2 occurs on an `insert` or `remove` operation the tree is rebalanced
- Many different cases of rebalancing
- **Example:** `insert` of $1, 2, 3, \ldots$
- http://people.ksp.sk/~kuko/bak

Balanced Trees

AVL-Tree - Rebalancing

AVL-Tree:

- If a height difference of ± 2 occurs on an `insert` or `remove` operation the tree is rebalanced
- Many different cases of rebalancing
- **Example:** `insert` of $1, 2, 3, \ldots$
- http://people.ksp.sk/~kuko/bak

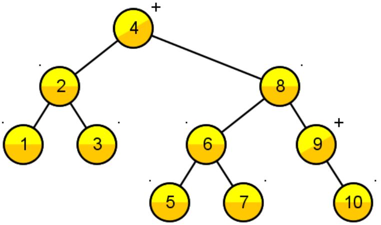


Figure: Inserting $1, \ldots, 10$ into an AVL-tree [Gna]

Summary:

Summary:

- Historical the first search tree providing guaranteed `insert`, `remove` and `lookup` in $O(\log n)$

Summary:

- Historical the first search tree providing guaranteed `insert`, `remove` and `lookup` in $O(\log n)$
- However not amortized update costs of $O(1)$

Summary:

- Historical the first search tree providing guaranteed `insert`, `remove` and `lookup` in $O(\log n)$
- However not amortized update costs of $O(1)$
- Additional memory costs: We have to save a height difference for every node

Summary:

- Historical the first search tree providing guaranteed `insert`, `remove` and `lookup` in $O(\log n)$
- However not amortized update costs of $O(1)$
- Additional memory costs: We have to save a height difference for every node
- Better (and easier) to implement are (a,b)-trees

Structure

(a,b)-Tree:

(a,b)-Tree:

- Also known as **b-tree** (b for "balanced")

(a,b)-Tree:

- Also known as **b-tree** (b for "balanced")
- Used in data bases and file systems

(a,b)-Trees

Introduction

(a,b)-Tree:

- Also known as **b-tree** (b for "balanced")
- Used in data bases and file systems

Idea:

(a,b)-Tree:

- Also known as **b-tree** (b for "balanced")
- Used in data bases and file systems

Idea:

- Save a varying number of elements per node

(a,b)-Tree:

- Also known as **b-tree** (b for “balanced”)
- Used in data bases and file systems

Idea:

- Save a varying number of elements per node
- So we have space for elements on an `insert` and balance operation

(a,b)-Tree:

(a,b)-Tree:

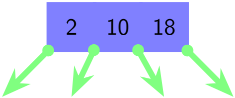
- All leaves have the same depth

(a,b)-Tree:

- All leaves have the same depth
- Each inner node has $\geq a$ and $\leq b$ nodes
  (Only the root node may have less nodes)

(a,b)-Tree:

- All leaves have the same depth
- Each inner node has $\geq a$ and $\leq b$ nodes
  (Only the root node may have less nodes)

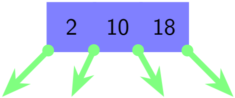


(a,b)-Trees

Introduction

(a,b)-Tree:

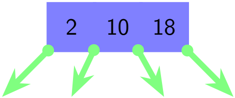
- All leaves have the same depth
- Each inner node has $\geq a$ and $\leq b$ nodes
  (Only the root node may have less nodes)



- Each node with n children is called "node of degree n" and holds $n - 1$ sorted elements

(a,b)-Tree:

- All leaves have the same depth
- Each inner node has $\geq a$ and $\leq b$ nodes
(Only the root node may have less nodes)



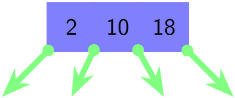
- Each node with n children is called "node of degree n" and holds $n - 1$ sorted elements
- Subtrees are located "between" the elements

(a,b)-Trees

Introduction

(a,b)-Tree:

- All leaves have the same depth
- Each inner node has $\geq a$ and $\leq b$ nodes
  (Only the root node may have less nodes)



- Each node with n children is called "node of degree n" and holds $n - 1$ sorted elements
- Subtrees are located "between" the elements
- We require: $a \geq 2$ and $b \geq 2a - 1$

(a,b)-Trees

Introduction

(2,4)-Tree:

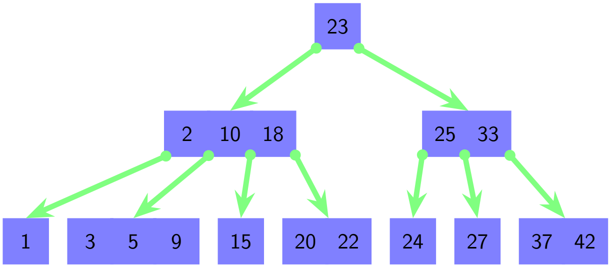


Figure: Example of an (2,4)-tree

(a,b)-Trees
Introduction

(2,4)-Tree:

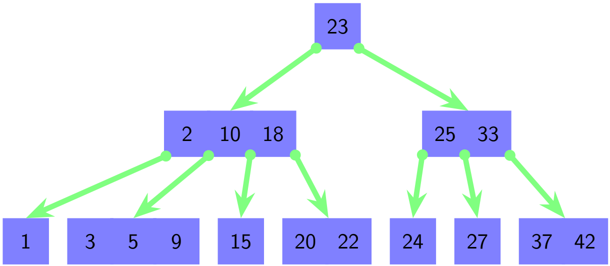


Figure: Example of an (2,4)-tree

- (2,4)-tree with depth of 3

(a,b)-Trees
Introduction

(2,4)-Tree:

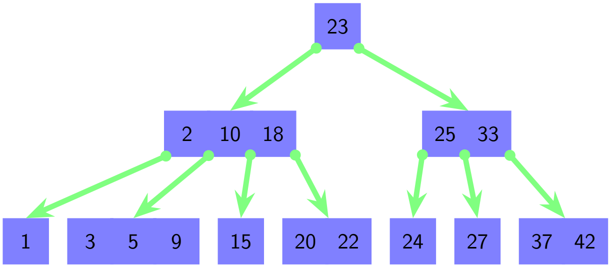


Figure: Example of an (2,4)-tree

- (2,4)-tree with depth of 3
- Each node has between 2 and 4 children (1 to 3 elements)

(a,b)-Trees

Introduction

Not an (2,4)-Tree:

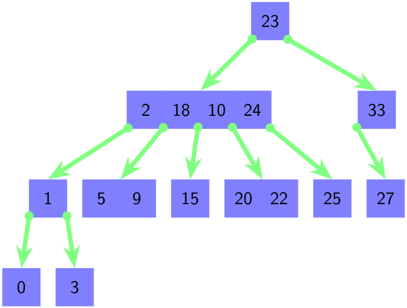


Figure: **Not** an (2,4)-tree

Not an (2,4)-Tree:

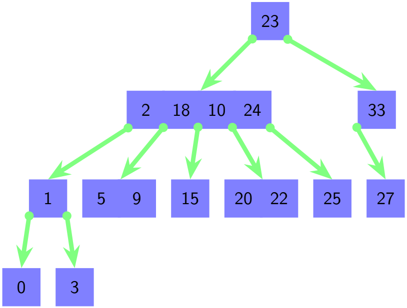


Figure: **Not** an (2,4)-tree

- Invalid sorting

(a,b)-Trees

Introduction

Not an (2,4)-Tree:

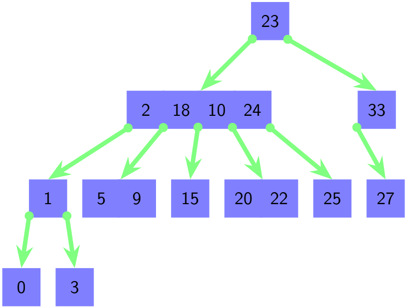


Figure: **Not** an (2,4)-tree

- Invalid sorting
- Degree of node too large / too small

(a,b)-Trees

Introduction

Not an (2,4)-Tree:

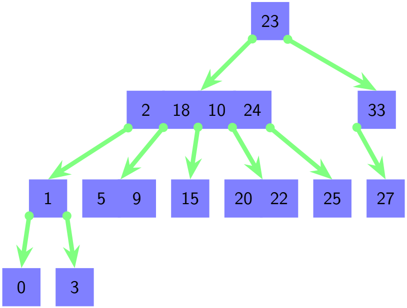


Figure: **Not** an (2,4)-tree

- Invalid sorting
- Degree of node too large / too small
- Leaves on different levels

Searching an element: (`lookup`)

(a,b)-Trees

Implementation - Lookup

Searching an element: (`lookup`)

- The same algorithm as in BinarySearchTree

(a,b)-Trees

Implementation - Lookup

Searching an element: (`lookup`)

- The same algorithm as in BinarySearchTree
- Searching from the root downwards

(a,b)-Trees

Implementation - Lookup

Searching an element: (`lookup`)

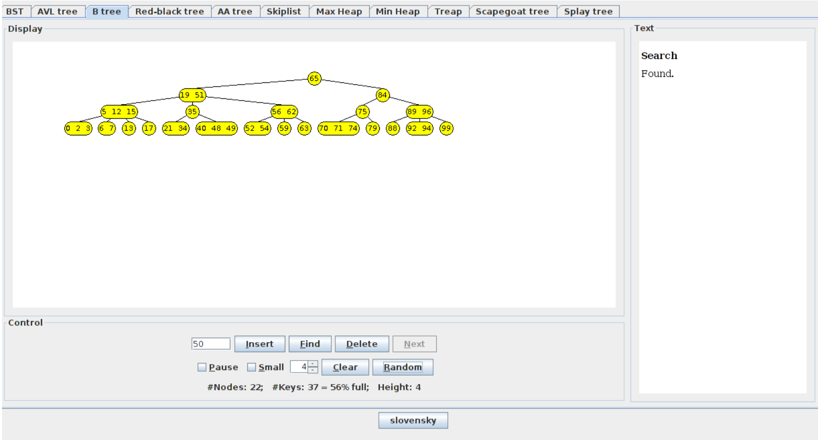
- The same algorithm as in BinarySearchTree
- Searching from the root downwards
- The keys at each node set the path

(a,b)-Trees

Implementation - Lookup

Searching an element: (`lookup`)

- The same algorithm as in BinarySearchTree
- Searching from the root downwards
- The keys at each node set the path



Inserting an element: (`insert`)

Inserting an element: (`insert`)

- Search the position to insert the key into

Inserting an element: (`insert`)

- Search the position to insert the key into
- This position will always be an leaf

Inserting an element: (`insert`)

- Search the position to insert the key into
- This position will always be an leaf
- Insert the element into the tree

Inserting an element: (`insert`)

- Search the position to insert the key into
- This position will always be an leaf
- Insert the element into the tree
- Attention: Nodes can have one element too many (Degree $b + 1$)

Inserting an element: (`insert`)

- Search the position to insert the key into
- This position will always be an leaf
- Insert the element into the tree
- Attention: Nodes can have one element too many (Degree $b+1$)
- Then we **split** the node

(a,b)-Trees

Implementation - Insert

Inserting an element: (`insert`)

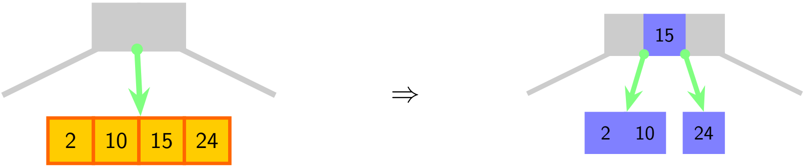


Figure: Splitting a node

(a,b)-Trees

Implementation - Insert

Inserting an element: (`insert`)

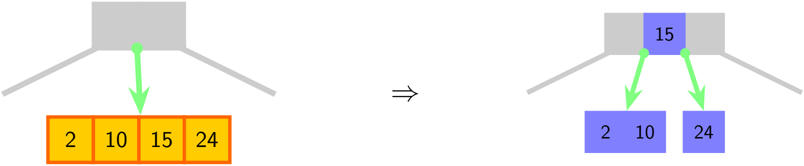


Figure: Splitting a node

- If the degree is higher than $b + 1$ we split the node

(a,b)-Trees

Implementation - Insert

Inserting an element: (insert)

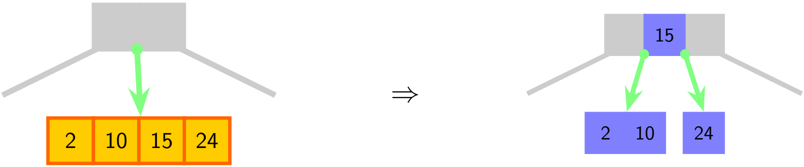


Figure: Splitting a node

- If the degree is higher than $b + 1$ we split the node
  - This results in a node with $\text{ceil}\left(\frac{b-1}{2}\right)$ elements, a element for the parent node, and a node with $\text{floor}\left(\frac{b-1}{2}\right)$ elements

(a,b)-Trees

Implementation - Insert

Inserting an element: (`insert`)

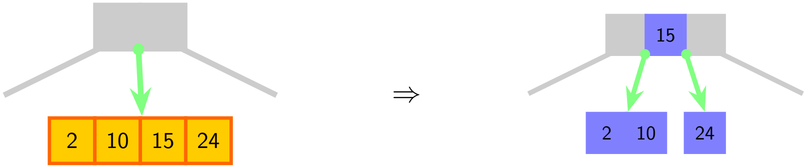


Figure: Splitting a node

- If the degree is higher than $b+1$ we split the node
  - This results in a node with $\mathrm{ceil}\left(\frac{b-1}{2}\right)$ elements, a element for the parent node, and a node with $\mathrm{floor}\left(\frac{b-1}{2}\right)$ elements
  - Thats why we have the limit $b \geq 2a-1$

Inserting an element: (`insert`)

Inserting an element: (`insert`)

- If the degree is higher than $b + 1$ we split the node

Inserting an element: (`insert`)

- If the degree is higher than $b + 1$ we split the node
- Now the parent node can be of a higher degree than $b + 1$

Inserting an element: (`insert`)

- If the degree is higher than $b + 1$ we split the node
- Now the parent node can be of a higher degree than $b + 1$
- We split the parent nodes the same way

Inserting an element: (`insert`)

- If the degree is higher than $b+1$ we split the node
- Now the parent node can be of a higher degree than $b+1$
- We split the parent nodes the same way
- If the node to split is the root we split it and create a new root node
  (The tree is now one level deeper)

Removing an element: (`remove`)

Removing an element: (`remove`)

- Search the element in $O(\log n)$ time

Removing an element: (`remove`)

- Search the element in $O(\log n)$ time
- **Case 1:** The element is contained by a leaf, remove it

(a,b)-Trees
Implementation - Remove

Removing an element: (`remove`)

- Search the element in $O(\log n)$ time
- **Case 1:** The element is contained by a leaf, remove it
- **Case 2:** The element is contained by an inner node

Removing an element: (`remove`)

- Search the element in $O(\log n)$ time
- **Case 1:** The element is contained by a leaf, remove it
- **Case 2:** The element is contained by an inner node
  - Search the successor in the right subtree

Removing an element: (`remove`)

- Search the element in $O(\log n)$ time
- **Case 1:** The element is contained by a leaf, remove it
- **Case 2:** The element is contained by an inner node
  - Search the successor in the right subtree
  - The successor is always contained by a leaf

Removing an element: (`remove`)

- Search the element in $O(\log n)$ time
- **Case 1:** The element is contained by a leaf, remove it
- **Case 2:** The element is contained by an inner node
  - Search the successor in the right subtree
  - The successor is always contained by a leaf
  - Replace the element with its successor and delete the successor from the leaf

Removing an element: (`remove`)

- Search the element in $O(\log n)$ time
- **Case 1:** The element is contained by a leaf, remove it
- **Case 2:** The element is contained by an inner node
  - Search the successor in the right subtree
  - The successor is always contained by a leaf
  - Replace the element with its successor and delete the successor from the leaf
- Attention: The leaf might be too small (degree of $a - 1$)
  $\Rightarrow$ We rebalance the tree

Removing an element: (`remove`)

Removing an element: (`remove`)

- Attention: The leaf might be too small (degree of $a - 1$)
  $\Rightarrow$ We rebalance the tree

Removing an element: (`remove`)

- Attention: The leaf might be too small (degree of $a - 1$)
  $\Rightarrow$ We rebalance the tree
  - **Case a:** If the left or right neighbour node has a degree greater than a we **borrow** one element from this node

Removing an element: (`remove`)

- Attention: The leaf might be too small (degree of $a - 1$)
  $\Rightarrow$ We rebalance the tree
  - **Case a:** If the left or right neighbour node has a degree greater than a we **borrow** one element from this node

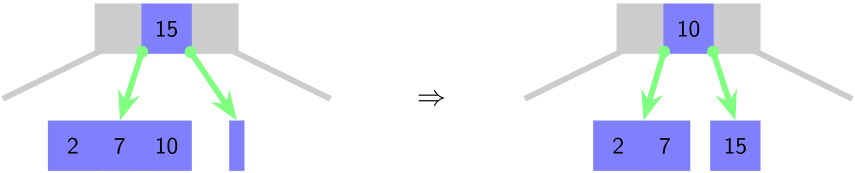


Figure: Borrowing an element

Removing an element: (`remove`)

(a,b)-Trees

Implementation - Remove

Removing an element: (`remove`)

- Attention: The leaf might be too small (degree of $a - 1$)
  $\Rightarrow$ We rebalance the tree

(a,b)-Trees

Implementation - Remove

Removing an element: (`remove`)

- Attention: The leaf might be too small (degree of $a-1$)
  $\Rightarrow$ We rebalance the tree
  - **Case b:** We combine the node with its right or left neighbour

Removing an element: (`remove`)

- Attention: The leaf might be too small (degree of $a-1$)
  $\Rightarrow$ We rebalance the tree
  - **Case b:** We combine the node with its right or left neighbour

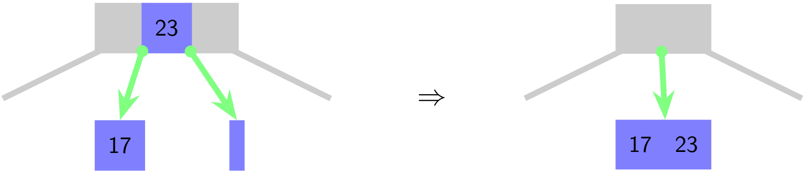


Figure: Combining two nodes

Removing an element: (`remove`)

Removing an element: (`remove`)

- Now the parent node can be of degree $a - 1$

Removing an element: (`remove`)

- Now the parent node can be of degree $a - 1$
- We combine parent nodes the same way

Removing an element: (`remove`)

- Now the parent node can be of degree $a - 1$
- We combine parent nodes the same way
- If the root has only one child left we take the child as new root (The tree shrinks one level)

Runtime complexity of `lookup`, `insert` and `remove`:

Runtime complexity of `lookup`, `insert` and `remove`:

- All operations in $O(d)$ with d being the depth of the tree

Runtime complexity of `lookup`, `insert` and `remove`:

- All operations in $O(d)$ with d being the depth of the tree
- Each node (except the root) has more than a children
  $\Rightarrow n \geq a^{d-1}$ and $d \leq 1 + \log_a n = O(log_a n)$

(a,b)-Trees

Runtime Complexity

Runtime complexity of `lookup`, `insert` **and** `remove`**:**

- All operations in $O(d)$ with d being the depth of the tree
- Each node (except the root) has more than a children
  $\Rightarrow n \geq a^{d-1}$ and $d \leq 1 + \log_a n = O(log_a n)$
- If we look closer:

(a,b)-Trees
Runtime Complexity

Runtime complexity of `lookup`, `insert` and `remove`:

- All operations in $O(d)$ with d being the depth of the tree
- Each node (except the root) has more than a children
  $\Rightarrow n \geq a^{d-1}$ and $d \leq 1 + \log_a n = O(log_a n)$
- If we look closer:
  - `lookup` always takes $\Theta(d)$

Runtime complexity of `lookup`, `insert` and `remove`:

- All operations in $O(d)$ with d being the depth of the tree
- Each node (except the root) has more than a children
  $\Rightarrow n \geq a^{d-1}$ and $d \leq 1 + \log_a n = O(log_a n)$
- If we look closer:
  - `lookup` always takes $\Theta(d)$
  - `insert` and `remove` often require only $O(1)$ time

(a,b)-Trees

Runtime Complexity

Runtime complexity of `lookup`, `insert` and `remove`:

- All operations in $O(d)$ with d being the depth of the tree
- Each node (except the root) has more than a children
  $\Rightarrow n \geq a^{d-1}$ and $d \leq 1 + \log_a n = O(log_a n)$
- If we look closer:
  - `lookup` always takes $\Theta(d)$
  - `insert` and `remove` often require only $O(1)$ time
  - Only in the worst case we have to split or combine all nodes on a path up to the root

Runtime complexity of `lookup`, `insert` and `remove`:

- All operations in $O(d)$ with d being the depth of the tree
- Each node (except the root) has more than a children
  $\Rightarrow n \geq a^{d-1}$ and $d \leq 1 + \log_a n = O(log_a n)$
- If we look closer:
  - `lookup` always takes $\Theta(d)$
  - `insert` and `remove` often require only $O(1)$ time
  - Only in the worst case we have to split or combine all nodes on a path up to the root
  - We want to analyse in detail

Runtime complexity of `lookup`, `insert` and `remove`:

- All operations in $O(d)$ with d being the depth of the tree
- Each node (except the root) has more than a children
  $\Rightarrow n \geq a^{d-1}$ and $d \leq 1 + \log_a n = O(log_a n)$
- If we look closer:
  - `lookup` always takes $\Theta(d)$
  - `insert` and `remove` often require only $O(1)$ time
  - Only in the worst case we have to split or combine all nodes on a path up to the root
  - We want to analyse in detail
  - Therefore instead of $b \geq 2\,a - 1$ we need $b \geq 2\,a$.

Runtime complexity of `lookup`, `insert` and `remove`:

- All operations in $O(d)$ with d being the depth of the tree
- Each node (except the root) has more than a children
  $\Rightarrow$ $n \geq a^{d-1}$ and $d \leq 1 + \log_a n = O(log_a n)$
- If we look closer:
  - `lookup` always takes $\Theta(d)$
  - `insert` and `remove` often require only $O(1)$ time
  - Only in the worst case we have to split or combine all nodes on a path up to the root
  - We want to analyse in detail
  - Therefore instead of $b \geq 2\,a - 1$ we need $b \geq 2\,a$.
  - Here is a counter-example for (2,3)-trees, analysis of (2,4)-trees

(a,b)-Trees

Runtime Complexity - Counter-example for (2,3)-Tree

(2,3)-Tree:

(a,b)-Trees

Runtime Complexity - Counter-example for (2,3)-Tree

(2,3)-Tree:

- Before executing `delete(11)`

(a,b)-Trees

Runtime Complexity - Counter-example for (2,3)-Tree

(2,3)-Tree:

- Before executing `delete(11)`

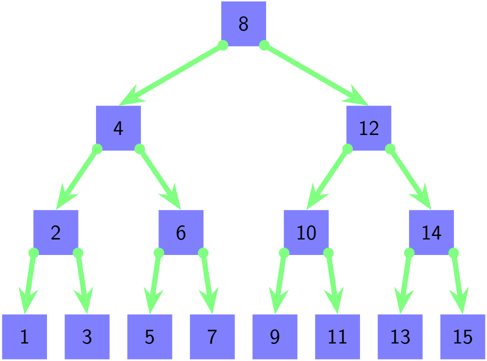


Figure: Normal (2,3)-Tree

(a,b)-Trees

Runtime Complexity - Counter example for (2,3)-Tree

(2,3)-Tree:

- Executing `delete(11)`

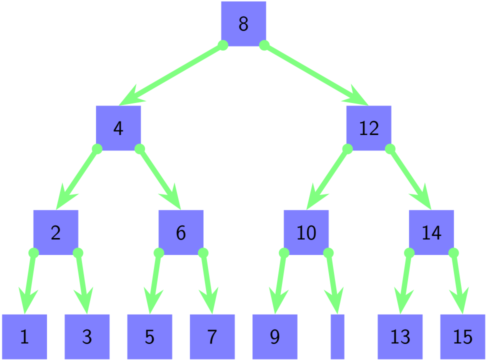


Figure: (2,3)-Tree - Delete step 1

(a,b)-Trees

Runtime Complexity - Counter example for (2,3)-Tree

(2,3)-Tree:

- Executing `delete(11)`

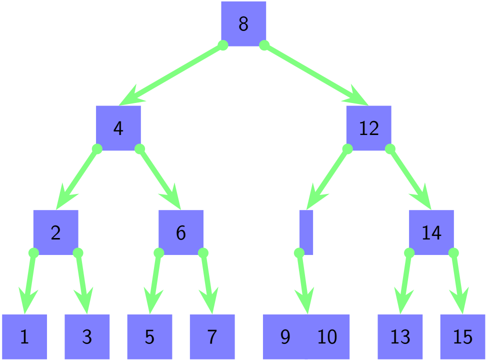


Figure: (2,3)-Tree - Delete step 2

(a,b)-Trees

Runtime Complexity - Counter example for (2,3)-Tree

(2,3)-Tree:

- Executing `delete(11)`

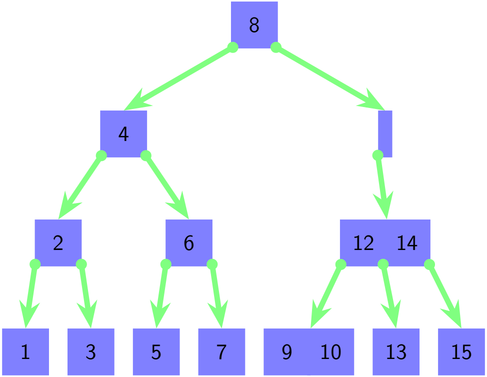


Figure: (2,3)-Tree - Delete step 3

(a,b)-Trees

Runtime Complexity - Counter example for (2,3)-Tree

(2,3)-Tree:

- Executed `delete(11)`

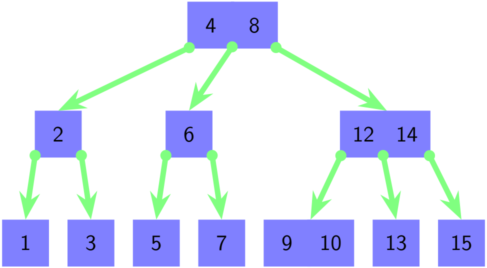


Figure: (2,3)-Tree - Delete step 4

(2,3)-Tree:

(2,3)-Tree:

- Executing `insert(11)`

(a,b)-Trees

Runtime Complexity - Counter example for (2,3)-Tree

(2,3)-Tree:

- Executing `insert(11)`

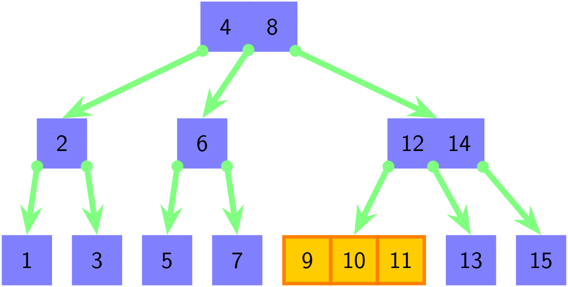


Figure: (2,3)-Tree - Insert step 1

(a,b)-Trees

Runtime Complexity - Counter example for (2,3)-Tree

(2,3)-Tree:

- Executing `insert(11)`

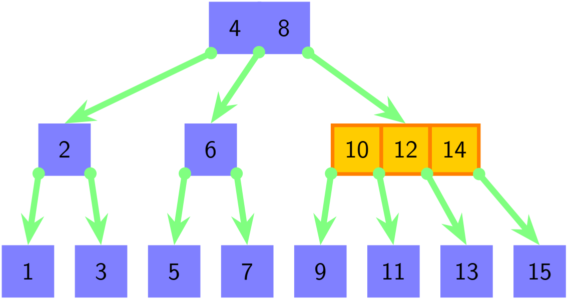


Figure: (2,3)-Tree - Insert step 2

(a,b)-Trees

Runtime Complexity - Counter example for (2,3)-Tree

(2,3)-Tree:

- Executing `insert(11)`

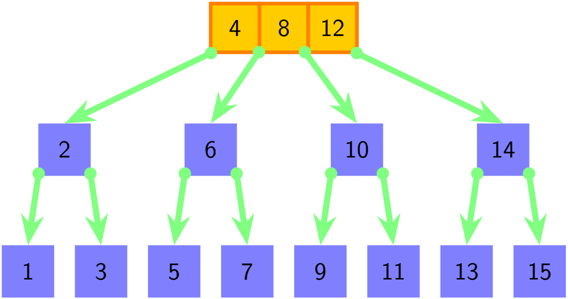


Figure: (2,3)-Tree - Insert step 3

(a,b)-Trees

Runtime Complexity - Counter example for (2,3)-Tree

(2,3)-Tree:

- Executed `insert(11)`

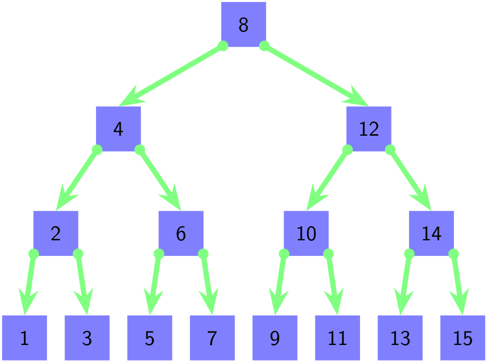


Figure: (2,3)-Tree - Insert step 4

(a,b)-Trees

Runtime Complexity - Counter example for (2,3)-Tree

(2,3)-Tree:

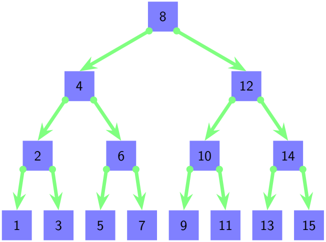


Figure: (2,3)-Tree

(a,b)-Trees

Runtime Complexity - Counter example for (2,3)-Tree

(2,3)-Tree:

- We are exactly where we started

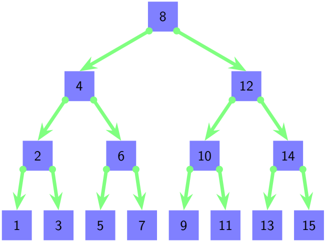


Figure: (2,3)-Tree

(a,b)-Trees

Runtime Complexity - Counter example for (2,3)-Tree

(2,3)-Tree:

- We are exactly where we started
- If $b = 2a - 1$ then we can create a sequence of `insert` and `remove` operations where each operation costs $O(\log n)$

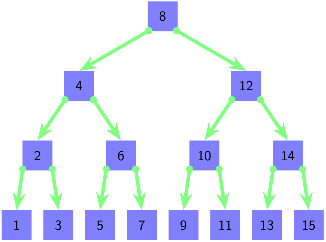


Figure: (2,3)-Tree

(a,b)-Trees

Runtime Complexity - Counter example for (2,3)-Tree

(2,3)-Tree:

- We are exactly where we started
- If $b = 2\,a - 1$ then we can create a sequence of `insert` and `remove` operations where each operation costs $O(\log n)$
- We need $b \geq 2\,a$ instead of $b \geq 2\,a - 1$

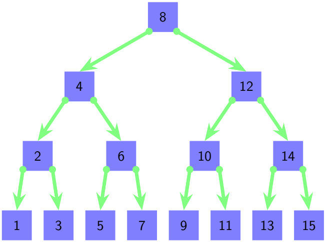


Figure: (2,3)-Tree

(2,4)-Tree:

(2,4)-Tree:

- If all nodes have 2 children we have to combine the nodes up to the root on a `remove` operation

(2,4)-Tree:

- If all nodes have 2 children we have to combine the nodes up to the root on a `remove` operation
- If all nodes have 4 children we have to split the nodes up to the root on a `insert` operation

(2,4)-Tree:

- If all nodes have 2 children we have to combine the nodes up to the root on a `remove` operation
- If all nodes have 4 children we have to split the nodes up to the root on a `insert` operation
- If all nodes have 3 children it takes some time to reach one of the previous two states

(2,4)-Tree:

- If all nodes have 2 children we have to combine the nodes up to the root on a `remove` operation
- If all nodes have 4 children we have to split the nodes up to the root on a `insert` operation
- If all nodes have 3 children it takes some time to reach one of the previous two states

$\Rightarrow$ **Nodes of degree 3 are harmless**
Neither an `insert` nor a `remove` operation trigger rebalancing operations

(2,4)-Tree:

(2,4)-Tree:

- **Idea:**

(2,4)-Tree:

- **Idea:**
  - After an expensive operation the tree is in a stable state

(2,4)-Tree:

- **Idea:**
  - After an expensive operation the tree is in a stable state
  - It takes some time until the next expensive operation occurs

(2,4)-Tree:

- **Idea:**
  - After an expensive operation the tree is in a stable state
  - It takes some time until the next expensive operation occurs
- Like with dynamic arrays:

(2,4)-Tree:

- **Idea:**
  - After an expensive operation the tree is in a stable state
  - It takes some time until the next expensive operation occurs
- Like with dynamic arrays:
  - Reallocation is expensive but it takes some time until the next expensive operation occurs

(2,4)-Tree:

- **Idea:**
  - After an expensive operation the tree is in a stable state
  - It takes some time until the next expensive operation occurs
- Like with dynamic arrays:
  - Reallocation is expensive but it takes some time until the next expensive operation occurs
  - If we overallocate clever we have an amortized runtime of $O(1)$

Terminology:

Terminology:

- We analyze a sequence of n operations

Terminology:

- We analyze a sequence of n operations
- Let Φ_i be the potential of the tree after the i-*th* operation

Terminology:

- We analyze a sequence of n operations
- Let Φ_i be the potential of the tree after the i-*th* operation
- $=$ is the number of nodes with degree 3

Example:

Example:

- Nodes of degree 3 are highlighted

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Example:

- Nodes of degree 3 are highlighted

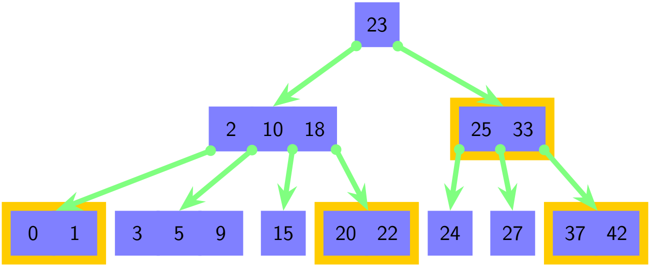


Figure: Tree with potential $\Phi = 4$

Terminology:

Terminology:

- Let c_i be the costs $=$ runtime of the i-th operation

Terminology:

- Let c_i be the costs = runtime of the i-th operation
- We will show:

Terminology:

- Let c_i be the costs = runtime of the i-th operation
- We will show:
  - Each operation can maximally destroy one harmless node

Terminology:

- Let c_i be the costs = runtime of the i-th operation
- We will show:
  - Each operation can maximally destroy one harmless node
  - For each further step, that incurs cost, the operation creates a further harmless node

Terminology:

- Let c_i be the costs = runtime of the i-th operation
- We will show:
  - Each operation can maximally destroy one harmless node
  - For each further step, that incurs cost, the operation creates a further harmless node
- The costs for operation i are coupled to the difference of the potential levels

$$c_i \leq A \cdot \underbrace{(\Phi_i - \Phi_{i-1})} + B, \qquad A > 0 \text{ and } B > A$$

Number of harmless (degree 3) nodes at operation i. Can be -1, but not smaller than -1

Terminology:

- Let c_i be the costs = runtime of the i-th operation
- We will show:
  - Each operation can maximally destroy one harmless node
  - For each further step, that incurs cost, the operation creates a further harmless node
- The costs for operation i are coupled to the difference of the potential levels

$$c_i \leq A \cdot \underbrace{(\Phi_i - \Phi_{i-1})} + B, \qquad A > 0 \text{ and } B > A$$

Number of harmless (degree 3) nodes at operation i. Can be -1, but not smaller than -1

- With that each operation has an amortitzed cost of $O(1)$

Case 1: *i-th* operation is an `insert` operation on a full node

Case 1: *i-th* operation is an `insert` operation on a full node

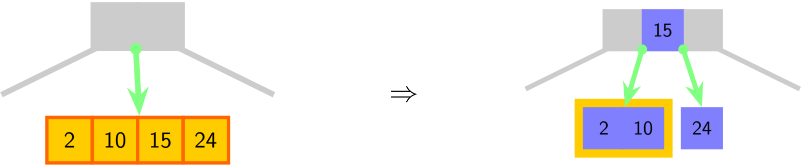


Figure: Splitting a node on `insert`

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 1: *i-th* operation is an `insert` operation on a full node

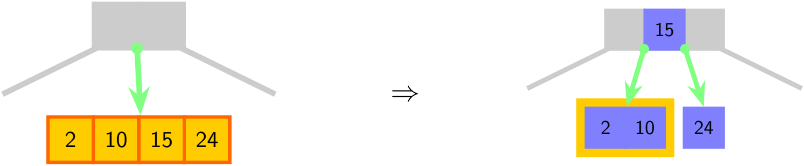


Figure: Splitting a node on `insert`

- Each splitted node creates a node of degree 3

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 1: *i-th* operation is an `insert` operation on a full node

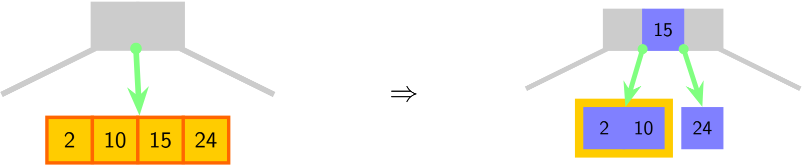


Figure: Splitting a node on `insert`

- Each splitted node creates a node of degree 3
- The parent node receives an element from the splitted node

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 1: *i-th* operation is an `insert` operation on a full node

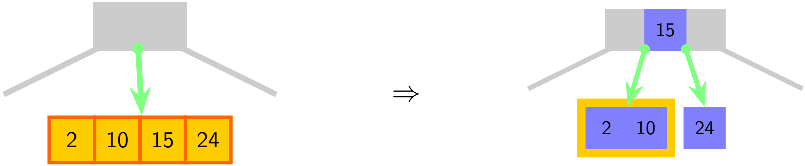


Figure: Splitting a node on `insert`

- Each splitted node creates a node of degree 3
- The parent node receives an element from the splitted node
- If the parent node is also full we have to split it too

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 1: *i-th* operation is an `insert` operation on a full node

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 1: i-*th* operation is an `insert` operation on a full node

- Let m be the number of nodes split

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 1: i-*th* operation is an `insert` operation on a full node

- Let m be the number of nodes split
- The potential rises by m

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 1: i-*th* operation is an `insert` operation on a full node

- Let m be the number of nodes split
- The potential rises by m
- If the "stop-node" is of degree 3 then the potential goes down by one

Case 1: *i-th* operation is an `insert` operation on a full node

- Let m be the number of nodes split
- The potential rises by m
- If the "stop-node" is of degree 3 then the potential goes down by one

$$\Phi_i \geq \Phi_{i-1} + m - 1$$
$$\Rightarrow m \leq \Phi_i - \Phi_{i-1} + 1$$

Case 1: i-*th* operation is an `insert` operation on a full node

- Let m be the number of nodes split
- The potential rises by m
- If the "stop-node" is of degree 3 then the potential goes down by one

$$\Phi_i \geq \Phi_{i-1} + m - 1$$
$$\Rightarrow m \leq \Phi_i - \Phi_{i-1} + 1$$

Costs: $c_i \leq A \cdot m + B$

$$\Rightarrow c_i \leq A \cdot (\Phi_i - \Phi_{i-1} + 1) + B$$
$$c_i \leq A \cdot (\Phi_i - \Phi_{i-1}) + \underbrace{A + B}_{B'}$$

Case 2: *i-th* operation is an `remove` operation

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 2: *i-th* operation is an `remove` operation

- **Case 2.1:** Inner node

Case 2: *i-th* operation is an `remove` operation

- **Case 2.1:** Inner node
  - Searching the successor in a tree is $O(d) = O(\log n)$

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 2: i-*th* operation is an `remove` operation

- **Case 2.1:** Inner node
  - Searching the successor in a tree is $O(d) = O(\log n)$
  - Normally the tree is coupled with a doubly linked list
    $\Rightarrow$ We can find the succcessor in $O(1)$

Case 2: *i-th* operation is an `remove` operation

- **Case 2.1:** Inner node
  - Searching the successor in a tree is $O(d) = O(\log n)$
  - Normally the tree is coupled with a doubly linked list
    $\Rightarrow$ We can find the succcessor in $O(1)$

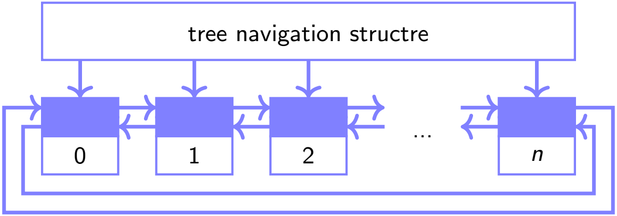


Figure: Tree with doubly linked list

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 2: i-*th* operation is an `remove` operation

Case 2: i-*th* operation is an `remove` operation

- **Case 2.1:** Borrowing a node

Case 2: *i-th* operation is an `remove` operation

- **Case 2.1:** Borrowing a node
  - Creates no additional operations

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 2: i-*th* operation is an `remove` operation

- **Case 2.1:** Borrowing a node
  - Creates no additional operations
  - Case 2.1.1: Potential rises by one

Case 2: *i-th* operation is an `remove` operation

- **Case 2.1:** Borrowing a node
  - Creates no additional operations
  - Case 2.1.1: Potential rises by one

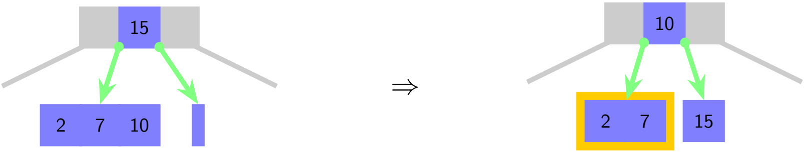


Figure: Borrowing an element case 2.1.1

Case 2: i-*th* operation is an `remove` operation

Case 2: *i-th* operation is an `remove` operation

- **Case 2.1:** Borrowing a node

Case 2: *i-th* operation is an `remove` operation

- **Case 2.1:** Borrowing a node
  - Creates no additional operations

Case 2: *i-th* operation is an `remove` operation

- **Case 2.1:** Borrowing a node
  - Creates no additional operations
  - Case 2.1.2: Potential lowers by one

Case 2: *i-th* operation is an `remove` operation

- **Case 2.1:** Borrowing a node
  - Creates no additional operations
  - Case 2.1.2: Potential lowers by one

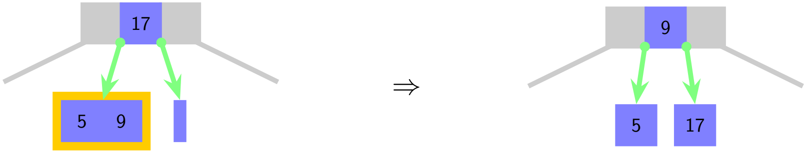


Figure: Borrowing an element case 2.1.2

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 2: i-*th* operation is an `remove` operation

Case 2: *i-th* operation is an `remove` operation

- **Case 2.2:** Merging a node

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 2: *i-th* operation is an `remove` operation

- **Case 2.2:** Merging a node

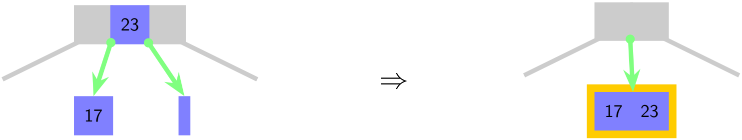


Figure: Merging two nodes

- Potential rises by one

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 2: *i-th* operation is an `remove` operation

- **Case 2.2:** Merging a node

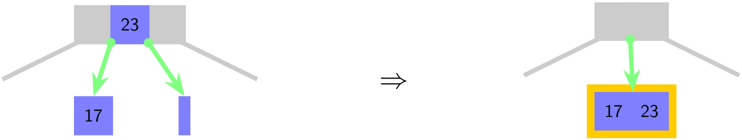


Figure: Merging two nodes

- Potential rises by one
- Parent node has one element less after the operation

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 2: *i-th* operation is an `remove` operation

- **Case 2.2:** Merging a node

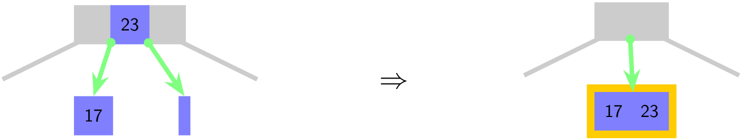


Figure: Merging two nodes

- Potential rises by one
- Parent node has one element less after the operation
- This operation propagates upwards until a node of degree > 2 or a degree 2 node, which can borrow from a neighbour

Case 2: i-*th* operation is an `remove` operation

- **Case 2.2:** Merging a node

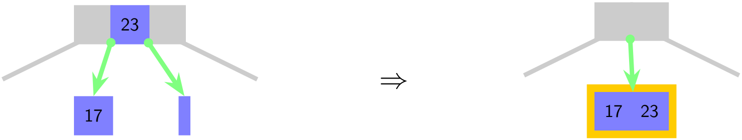


Figure: Merging two nodes

- Potential rises by one
- Parent node has one element less after the operation
- This operation propagates upwards until a node of degree > 2 or a degree 2 node, which can borrow from a neighbour
- The potential rises by m

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 2: i-*th* operation is an `remove` operation

- **Case 2.2:** Merging a node

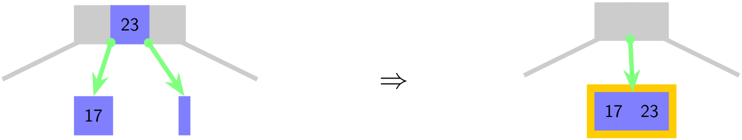


Figure: Merging two nodes

- Potential rises by one
- Parent node has one element less after the operation
- This operation propagates upwards until a node of degree > 2 or a degree 2 node, which can borrow from a neighbour
- The potential rises by m
- If the "stop-node" is of degree 2 then the potential eventually goes down by one

(a,b)-Trees

Runtime Complexity - (2,4)-Tree

Case 2: *i-th* operation is an `remove` operation

- **Case 2.2:** Merging a node

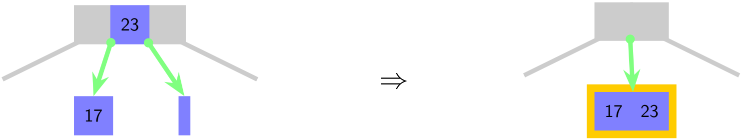


Figure: Merging two nodes

- Potential rises by one
- Parent node has one element less after the operation
- This operation propagates upwards until a node of degree > 2 or a degree 2 node, which can borrow from a neighbour
- The potential rises by m
- If the "stop-node" is of degree 2 then the potential eventually goes down by one
- Same costs as `insert`

Lemma:

Lemma:

- We know:

$$c_i \leq A \cdot (\Phi_i - \Phi_{i-1}) + B, \qquad A > 0 \text{ and } B > A$$

Lemma:

- We know:

$$c_i \leq A \cdot (\Phi_i - \Phi_{i-1}) + B, \qquad A > 0 \text{ and } B > A$$

- With that we can conclude:

$$\sum_{i=0}^{n} c_i = O(n)$$

Proof:

$$\sum_{i=0}^{n} c_i \leq \underbrace{A \cdot (\Phi_1 - \Phi_0) + B}_{\leq c_1} + \underbrace{A \cdot (\Phi_2 - \Phi_1) + B}_{\leq c_1} + \cdots + \underbrace{A \cdot (\Phi_n - \Phi_{n-1})}_{\leq c_n}$$

$= A \cdot (\Phi_n - \Phi_0) + B \cdot n$ | telescope sum

$= A \cdot \Phi_n + B \cdot n$ | we start with an empty tree

$< A \cdot n + B \cdot n = O(n)$ | number of degree 3 nodes

¡ number of nodes

Structure

Red-Black Tree:

Red-Black Tree:

- Binary tree with red and black nodes

Red-Black Tree:

- Binary tree with red and black nodes
- Number of black nodes on path to leaves is equal

Red-Black-Trees

Introduction

Red-Black Tree:

- Binary tree with red and black nodes
- Number of black nodes on path to leaves is equal
- Can be interpreted as (2,4)-tree (also named 2-3-4-tree)

Red-Black Tree:

- Binary tree with red and black nodes
- Number of black nodes on path to leaves is equal
- Can be interpreted as (2,4)-tree (also named 2-3-4-tree)
- Each (2,4)-tree-node is a small red-black-tree with a black root node

Red-Black-Trees

Introduction

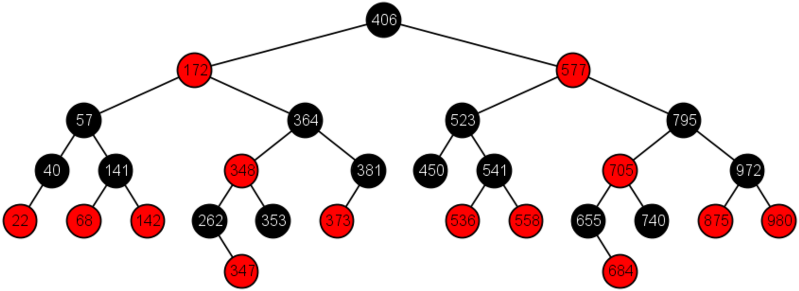


Figure: Example of an red-black-tree [Gna]

- **General**

[CRL01] Thomas H. Cormen, Ronald L. Rivest, and Charles E. Leiserson.
Introduction to Algorithms.
MIT Press, Cambridge, Mass, 2001.

[MS08] Kurt Mehlhorn and Peter Sanders.
Algorithms and data structures, 2008.
https://people.mpi-inf.mpg.de/~mehlhorn/ftp/Mehlhorn-Sanders-Toolbox.pdf.

- **Gnarley Trees**

[Gna] Gnarley Trees
https://people.ksp.sk/~kuko/gnarley-trees/

- **AVL-Tree**

[Wik] AVL tree
https://en.wikipedia.org/wiki/AVL_tree

- **(a,b)-Tree**

[Wika] 2-3-4 tree
https://en.wikipedia.org/wiki/2%E2%80%933%E2%80%934_tree

[Wikb] (a,b)-tree
https://en.wikipedia.org/wiki/(a,b)-tree

- **Red-Black-Tree**

[Wik] Red-black tree
https://en.wikipedia.org/wiki/Red%E2%80%93black_tree