Structure

Balanced Trees

Motivation
AVL-Trees
(a,b)-Trees
   Introduction
   Runtime Complexity
Red-Black Trees
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: $d \in O(\log n)$, keys are inserted randomly
- Worst case: $d \in O(n)$, keys are inserted in ascending / descending order (20, 19, 18, …)
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:

- 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 $O(\log n)$?

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

- **(a,b)-Tree or B-Tree:**
  - Node has between $a$ and $b$ children
  - Balancing through splitting and merging nodes
  - Used in databases 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` and Java `SortedListMap`
AVL-Tree:

- Gregory Maximovich Adelson-Velskii, Yevgeniy Mikhailovlovich Landis (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 $O(\log n)$
Figure: Example of an AVL-Tree
Figure: Not an AVL-Tree
Balanced Trees

AVL-Tree

Figure: Another example of an AVL-Tree
Rotation:

- **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
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, ...

![AVL Tree Example](image)

**Figure:** Inserting 1, …, 10 into an AVL-tree [Gna]
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
(a,b)-Trees

Introduction

(a,b)-Tree:
- Also known as b-tree (b for “balanced”)
- Used in databases 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:

- 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 $b \geq 2a - 1$
(2,4)-Tree:

- (2,4)-tree with depth of 3
- Each node has between 2 and 4 children (1 to 3 elements)
**Not an (2,4)-Tree:**

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

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

*Figure: (3,5)-Tree [Gna]*
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:** As a result node can overflow by one element (Degree $b+1$)
- Then we **split** the node
Inserting an element: (insert)

- If the degree is higher than $b + 1$ we split the node.
- This results in a node with $\lceil \frac{b-1}{2} \rceil$ elements, a node with $\lfloor \frac{b-1}{2} \rfloor$ elements and one element for the parent node.
- That's why we have the limit $b \geq 2a - 1$.

Figure: Splitting a 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 $b + 1$
- We split the parent nodes the same way
- If we split the root node we create a new parent root node
  (The tree is now one level deeper)
Removing an element: (remove)

- Search the element in $O(\log n)$ time
- **Case 1:** The element is contained by a leaf
  - Remove element
- **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 $a - 1$) ⇒ We rebalance the tree
Removing an element: (remove)

- **Attention:** The leaf might be too small (degree of $a - 1$) ⇒ 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

![Diagram showing the removal of an element and rebalancing of the tree.](image)
Removing an element: \(\text{remove}\)

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

- **Case b:** We **merge** the node with its right or left neighbour

![Diagram of merge two nodes](image)

**Figure:** Merge two nodes
Removing an element: \text{(remove)}

- Now the parent node can be of degree $a - 1$
- We \text{merge} parent nodes the same way
- If the root has only a single child
  - Remove the root
  - Define sole child as new root
  - The tree shrinks by one level
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
  \[ n \geq a^{d-1} \text{ and } d \leq 1 + \log_a n = O(\log_a n) \]

**In detail:**

- **lookup** always takes $\Theta(d)$
- **insert** and **remove** often require only $O(1)$ time
- **Worst case:** split or merge all nodes on path up to the root
- Therefore instead of $b \geq 2a - 1$ we need $b \geq 2a$
Counter example (2,3)-Tree:
- Before executing delete(11)

Figure: Normal (2,3)-Tree
Counter example (2,3)-Tree:

- Executing `delete(11)`

![Figure: (2,3)-Tree - Delete step 1](image)
Counter example (2,3)-Tree:

- Executing `delete(11)`

**Figure:** (2,3)-Tree - Delete step 2
(a,b)-Trees
Runtime Complexity - Counter example for (2,3)-Tree

Counter example (2,3)-Tree:
- Executing `delete(11)`

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

- Executed `delete(11)`

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

- Executing \texttt{insert(11)}

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

- Executing `insert(11)`

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

- Executing `insert(11)`

Figure: (2,3)-Tree - Insert step 3
Counter example (2,3)-Tree:
- Executed \textit{insert}(11)

Figure: (2,3)-Tree - Insert step 4
Counter example (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 \( O(\log n) \)
- We need \( b \geq 2a \) instead of \( b \geq 2a - 1 \)

Figure: (2,3)-Tree
(2,4)-Trees:

- If all nodes have 2 children we have to merge 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 an insert operation.
- If all nodes have 3 children it takes some time to reach one of the previous two states.

⇒ **Nodes of degree 3 are stable**
Neither an insert nor a remove operation trigger rebalancing operations.
(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 $O(1)$
(a,b)-Trees
Runtime Complexity - (2,4)-Tree

Terminology:

- We analyze a sequence of \( n \) operations
- Let \( \Phi_i \) be the potential of the tree after the \( i \)-th operation
- \( \Phi_i = \) the number of stable nodes with degree 3
- Empty tree has 0 nodes: \( \Phi = 0 \)
(a,b)-Trees
Runtime Complexity - (2,4)-Tree

Example:
- Nodes of degree 3 are highlighted

Figure: Tree with potential $\Phi = 4$
Terminology:

- Let $c_i$ be the costs = runtime of the $i$-th operation
- We will show:
  - Each operation can at most destroy one stable node
  - For each cost incurring step the operation creates an additional stable node
- The costs for operation $i$ are coupled to the difference of the potential levels
  \[ c_i \leq A \cdot (\Phi_i - \Phi_{i-1}) + B, \quad A > 0 \text{ and } B > A \]

Number of gained stable nodes (degree 3) $\geq -1$

- Each operation has an amortized cost of $O(1)$ summing up to $O(n)$ in total
Case 1: *i*-th operation is an *insert* operation on a full node

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
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

- **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 successor in \( O(1) \)

**Figure:** Tree with doubly linked list
Case 2: *i*-th operation is an `remove` operation

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

![Figure: Case 2.1.1: Borrow an element](image-url)
Case 2: \(i\)-th operation is an remove operation

- **Case 2.1**: Borrow a node
  - Creates no additional operations
  - Case 2.1.2: Potential is lowered by one

---

**Figure**: Case 2.1.2: Borrow an element
Case 2: \textit{i-th} operation is an \texttt{remove} operation

- Case 2.2: Merging two node

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 node of degree 2, which can borrow from a neighbour

\textbf{Figure:} Merging two nodes
**Case 2:** *i*-th operation is an `remove` operation

- **Case 2.2:** Merging two nodes

![Figure: Merging two nodes](image)

- 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:

- We know:

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

- With that we can conclude:

\[ \sum_{i=0}^{n} c_i \in O(n) \]
Proof:

\[
\sum_{i=0}^{n} c_i \leq A \cdot (\Phi_1 - \Phi_0) + B + A \cdot (\Phi_2 - \Phi_1) + B + \cdots + A \cdot (\Phi_n - \Phi_{n-1}) + B
\]

\[
\leq c_1 + c_2 + \cdots + c_n
\]

\[
= A \cdot (\Phi_n - \Phi_0) + B \cdot n \quad \text{telescope sum}
\]

\[
= A \cdot \Phi_n + B \cdot n \quad \text{we start with an empty tree}
\]

\[
< A \cdot n + B \cdot n \in O(n) \quad \text{number of degree 3 nodes}
\]

\[
< \text{number of nodes}
\]
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
Figure: Example of an red-black-tree [Gna]
General


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%934_tree

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

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