**What is a Balanced Binary Search Tree?**

* Binary search trees can be **balanced**
  + Subtrees hold nearly equal number of nodes
  + Subtrees are with nearly the same height

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Structure** | **Worst case** | | | **Average case** | |
| **Search** | **Insert** | **Delete** | **Search Hit** | **Insert** |
| **BST** | N | N | N | 1.39 lg N | 1.39 lg N |
| **2-3 Tree** | **c** lg N | **c** lg N | **c** lg N | **c** lg N | **c** lg N |
| **Red-Black** | 2 lg N | 2 lg N | 2 lg N | lg N | lg N |
| **AVL Tree** | 1.44 lg N | 1.44 lg N | 1.44 lg N | lg N | lg N |

**What are B-Trees?**

* [**B-trees**](https://en.wikipedia.org/wiki/B-tree)are generalization of the concept of ordered binary search trees – see the [**visualization**](https://www.cs.usfca.edu/~galles/visualization/BTree.html)
  + B-tree of order **b** has between **b** and **2\*b** keys in a node and between **b+1** and **2\*b+1** child nodes
  + The keys in each node are ordered increasingly
  + All keys in a child node have values between their left and right parent keys
* B-trees can be efficiently stored on the hard disk

**B-Trees vs. Other Balanced Search Trees**

* B-Trees hold a **range of child nodes**, not single one
  + B-trees do not need re-balancing so frequently
* B-Trees are good for **database indexes**
  + Because a single node is stored in a single cluster of the hard drive
  + Minimize the number of disk operations (which are very slow)
* B-Trees are almost perfectly balanced
  + The count of nodes from the root to any **null** node is the same

**2-3 Trees**

**Definition**

* A 2-3 search tree can contain:
  + Empty node (**null**)
  + 2-node with **1 key** and **2 links** (children)
  + 3-node with **2 keys** and **3 links** (children)
* As usual for BSTs, all items to the left are smaller, all items to the right are larger.

**2-3 Tree Properties**

* Unlike standard BSTs, 2-3 trees **grow from the bottom**
* The **number of links** from the root to any **null** node is the same
* Transformations are **local**
* Nearly **perfectly balanced**
* Inserting **10 nodes** will result with height of the tree **2**
  + For normal BSTs the height can be **9** in the worst case

**AVL Trees**

* [AVL tree](https://en.wikipedia.org/wiki/AVL_tree) is a self-balancing binary-search tree ([visualization](https://www.cs.usfca.edu/~galles/visualization/AVLtree.html))
  + Height of two subtrees can **differ by at most 1**

**AVL Tree Rebalancing**

* Height difference is measured by a balance factor (BF)
* **BF(Tree)** = **Height(Left)** – **Height(Right)**
* BF of any node is in the range **[-1, 1]**
* If BF becomes **-2** or **2** 🡪 rebalance
* Rebalancing is done by retracing
* Start from inserted node's parent  
  and go up to root
* Perform **rotations** to restore balance

**Left Rotation**

* Set (x) to be parent of (y)
* Set Right Child of(y) to be Left Child of (x)

temp = node.left

node.left = temp.right

temp.right = node

**Right Rotation**

* Set (y) to be child of (x)
* Set Right Child of(x) to be Left Child of (y)

temp = node.right

node.right = temp.left

temp.left = node

**AVL Tree Insertion Algorithm**

* Insert like in ordinary BST
* Retrace up to root
  + Modify balance / height
  + If balance factor ∉ [-1,1]  
     🡪 rebalance

**Double Right Rotation**

* Rotate Right (node) with negatively balanced Left Child

**Red-Black Tree**

The height of any red–black BST on n keys (regardless of the order of insertion) is guaranteed to be between log2n and 2log2n

**Why Yet Another Balanced BST?**

* We want operations to happen at:
  + O(log(n)) not O(h) where h in worst case is n
* AVL vs Red-Black trees:
  + The AVL trees are more balanced that causes more rotations during insertion and deletion
  + if your application involves many frequent insertions and deletions, then Red Black trees should   
    be preferred

**Representing 3-Nodes from 2-3 Tree**

* We will represent 3-nodes with a left-leaning **red** nodes
* Nodes with values between the 2 nodes will be to the right of the **red** node

**Red-Black Tree Properties**

* All **leaves** are **black**
* The **root** is **black**
* No node **has two** **red** links connected to it
* Every path from a given **node** to **its descendant leaf** nodes contains the same number of **black** nodes
* **Red** links **lean** left

**Left Rotation**

* Orient a right-leaning red link to lean left

Y X

/ \ -> / \

X Y

/ \ / \

**Right Rotation**

* Orient a left-leaning red link to lean right (temporarily)

X Y

/ \ => / \

Y X

/ \ / \

**Insertion Algorithm**

* **Locate** the node position
* Create new **red** node
* **Add** the new node to the tree
* **Balance** the tree if needed

**Insertion**

* Insert into 2-node:
  + Smaller element
  + Larger element – needs left rotation
* Insertion Into 3-Node
  + The element is **larger** than both keys

13 13

/ \ => / \

11 41 11 41

Flipping the colors **increases** the **tree height**, which maintains the 1-1 correspondence to 2-3 trees

* + The element is **smaller** than both keys

13 11 flip 11

/ right / \ => / \

11 => 9 13 colors 9 13

/ rotation

9

* + The element is **between** the 2 keys

50 50 30 flip 30

/ right / => / \ => / \

20 => 30 20 50 colors 20 50

\ left /

30 20

**Keeping Black Root**

* Insert on a single node (root):

30 flip 30 30

/ \ => / \ = > / \

20 50 colors 20 50 20 50

* Each time the root switches colors, the height of the tree is increased

**AA Tree**

**Why AA Trees**

* **Red-Black** vs **AA** trees:
* The implementation and number of rotation cases in Red-Black Trees is **complex**. AA trees **simplifies**   
  the algorithm
* It eliminates **half** of the restructuring process by eliminating half of the **rotation** cases, which is   
  easier to code
* It **simplifies** the deletion process by removing   
  multiple cases

**AA Tree**

* Utilizes the concept of **levels**
* **Level** - the **number of left links** on the path to a **null** node

17 Level 3

-------/------\----------

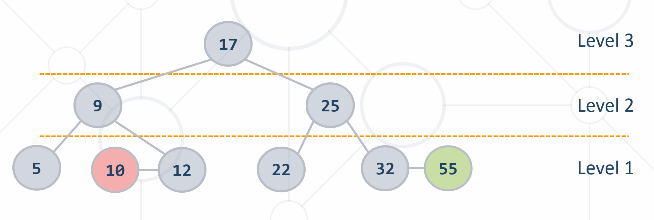
9 25 Level 2

---/---\------/---\------

5 12 22 32 Level 1

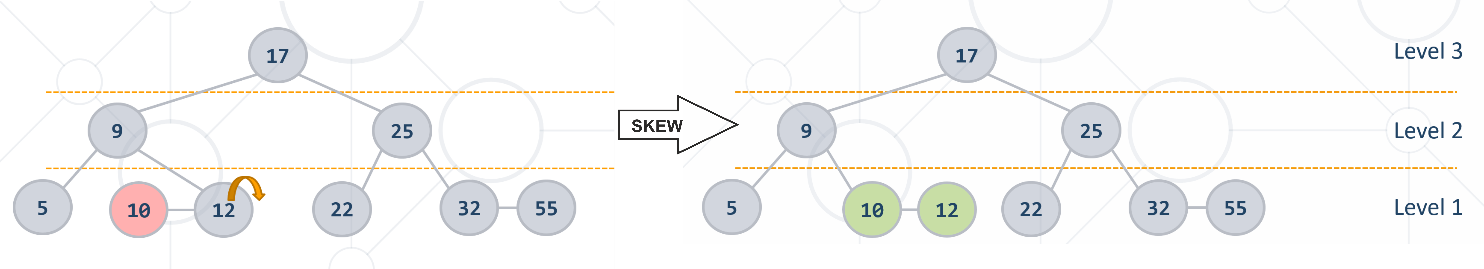
/ \ / \ / \ / \

* AA tree invariants
  + The **level** of every **leaf node** is **one**
  + Every **left child** has **level one less** than its **parent**
  + Every **right child** has **level equal** to or **one less** than its **parent**
  + **Right grandchildren** have **levels less** than their **grandparents**
  + Every node of level greater than one **has two children**
* **Right** horizontal links **are possible**
* **Left** horizontal links **are not allowed**



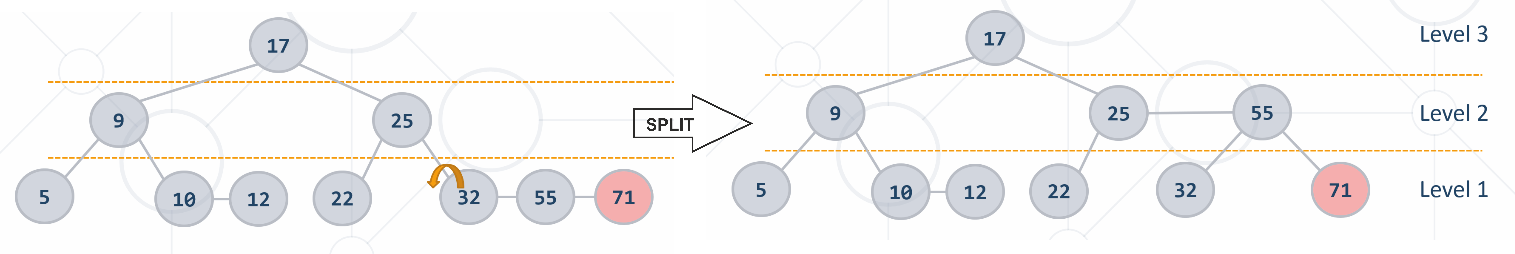
**Skew**

* Skew operation is a single **right** rotation
* Skew when an **insertion** or **deletion** creates a horizontal **left** **link**



**Split**

* Split operation is a **single left rotation**
* Split when an insertion or deletion **two consecutive right horizontal links**

****

**Hash Tables**

* **Hash Function -** Given a key of any type, convert it to an integer
* **Hash Table -** A [hash table](https://en.wikipedia.org/wiki/Hash_table) is an array that holds a set of **{key, value} pairs.**

The process of mapping a key to a position in a table is called **hashing**

**Hash Functions and Hashing**

* A hash table has **m** slots, indexed from **0** to **m-1**
* A hash function converts **keys** into array indices

**Hashing Functions**

* Perfect hashing function (PHF)
  + **h(k)**: one-to-one mapping of each key **k** to an integer in the range **[0, m-1]**
  + The PHF maps each key to a **distinct** integer within some manageable range
* Finding a perfect hashing function is impossible in most cases
* Good hashing function
  + **Consistent** - equal keys must produce the same hash value
  + **Efficient** - efficient to compute the hash
  + **Uniform** - should uniformly distribute the keys

**Collisions in a Hash Table**

* A **collision** comes when **different key**s have the **same hash value**
  + **h(k1) = h(k2) for k1 ≠ k2**
* When the number of collisions is sufficiently small, the hash tables work quite well (fast)
* Several **collisions resolution strategies** exist
  + **Chaining** collided keys (+ values) in a list
  + Using **other slots** in the table (open addressing)
  + Cuckoo hashing
  + Many other

**Collision Resolution: Open Addressing**

* **Open addressing** as collision resolution strategy means to take another slot in the hash-table in case of collision, e.g.
  + **Linear probing**: take the next empty slot just after the collision
    - **h(key, i) = h(key) + i**
    - where **i** is the attempt number: 0, 1, 2, …
    - **h(key) + 1, h(key) + 2, h(key) + 3**, etc.
  + **Quadratic probing**: the **ith** next slot is calculated by a quadratic polynomial (**c1** and **c2** are some constants)
    - **h(key, i) = h(key) + c1\*i + c2\*i2**
    - **h(key) + 12, h(key) + 22, h(key) + 32**, etc.
    - **Re-hashing**: use separate (second) hash-function for collisions
    - **h(key, i) = h1(key) + i\*h2(key)**

**Hash Table Performance**

* The hash-table performance depends on the probability  
  of collisions
  + **Less collisions** 🡪 **faster** add / find / delete operations
  + **Collisions resolution** algorithm
  + **Fill factor** (used buckets / all buckets)
* **Add** / **Find** / **Delete** take just few primitive operations
  + Speed does not depend on the size of the hash-table
  + Amortized complexity **O(1)** – constant time
* Example:
  + Finding an element in a **hash-table** holding **1 000 000 elements**  
    takes average just **1-2 steps**
  + Finding an element in an **array** holding **1 000 000 elements**  
    takes average **500 000 steps**

**How Big the Hash-Table Should Be?**

* The **load factor** (fill factor) **= used cells / all cells**
  + How much the hash table is filled, e.g. 65%
* Smaller fill factor leads to less collisions (faster average   
  seek time)
* Recommended fill factors:
  + When **chaining** is used as collision resolution 🡪 less than **75%**
  + When **open addressing** is used 🡪 less than **50%**

**Sets and Bags**

* The abstract data type (ADT) "**set**" keeps a set of elements with no duplicates
* Sets with duplicates are also known as ADT "**bag**"
* Set specific operations:
  + **unionWith(set)**
  + **intersectWith(set)**
  + **exceptWith(set)**
  + **symmetricExceptWith(set)**

**HashSet<T>**

* **HashSet<T>** implements ADT **set** by hash table
  + Elements are in no particular order
* All major operations are fast: **Add** / **Delete** / **Contains**

**TreeSet<T>**

* **TreeSet<T>** implements ADT **set** by balanced search tree (red-black tree)
  + Elements are sorted in increasing order

**Comparasion Methods**

* **Map<Key, Value>** relies on
  + **Object.equals()** – for comparing the keys
  + **Object.hashCode()** – for calculating the hash codes of the keys
* **TreeMap<Key, Value>** relies on **Comparable<Key>** for ordering the keys

**Maps (Dictionaries)**

* The abstract data type (ADT) "**dictionary**" maps key to values
  + Also known as "**map**" or "**associative array**"
  + Holds a set of **{key, value} pairs**
* Many implementations
* Hash table, balanced tree, list, array, ...

**Map <Key, Value>**

* Major operations:
  + **add(key, value)** – adds an element by key + value
  + **remove(key)** – removes a value by key
  + **get(key)** – returns the value by key
  + **keys** – returns a collection of all keys (in order of entry)
  + **values** – returns a collection of all values (in order of entry)
  + **containsKey(key)** – checks if given key exists in the dictionary
  + **containsValue(value)** – checks whether the dictionary contains given value
  + Warning: slow operation – **O(n)**

**TreeMap<Key, Value>**

* **TreeMap<Key, Value>** implements the ADT "dictionary" as self-balancing search tree
  + Elements are arranged in the tree ordered by key
  + Traversing the tree returns the elements in increasing order
  + **add** / **find** / **delete** perform **log N** operations
* Use **TreeMap<Key, Value>** when you need the elements sorted by key
  + Otherwise use **Map<Key, Value>** – it has better performance