**Binary Search Tree Review**

* Data structure designed for **O(LogN) search**
* Can be viewed as a **recursive data structure (subtrees)**
* Have overall ordering: **values(L) < root < values(R)**
* Insert new nodes as leaves
  + We want to make it as balanced as possible via. rotation
* Delete from anywhere

**Rebalancing Trees**

An approach to balanced trees

* Insert into leaves as for simple BST, then periodically rebalance the tree

Questions: How frequently / when / how to rebalance?

* E.g. after X amount of insertions?

How to rebalance a BST?

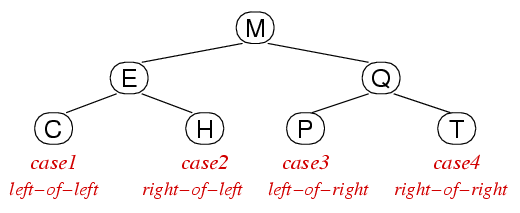
* **Move the median item to root (via. Rotations)**
  + Median / Midpoint item will always be the root – Roughly half of the nodes on the LEFT, half on the RIGHT
  + How do we find the median item?
    - We index into the tree and go to **N/2th item**
  + Move the Median Item to the root node via. rebalancing

**Rebalancing Trees – Analysis**

* Visit every node 🡪 O(n)
* Cost is not feasible to rebalance after each insertion
* When to rebalance?
  + After every k insertions
  + Whenever “imbalance” exceeds threshold

**Splay Trees**

* Another kind of “balanced tree”
* Splay tree insertion modifies **insertion-at-root** method:
  + Considers **Parent-Child-Grandchild** (three level analysis)
  + Performs double-rotations based on **P-C-G** orientations
* The idea: **appropriate double-rotations** **improve tree balance**
* Splay tree implementations also do **rotations-in-search**  
  (Modify tree when you’re doing a search – You move the item you just found to the top of the tree)  
  (Reasoning: studies shown that an item that you search for is likely to be searched for again)
  + Can provide **similar effects to periodic rebalance**
  + Improve balance, but **makes search more expensive**



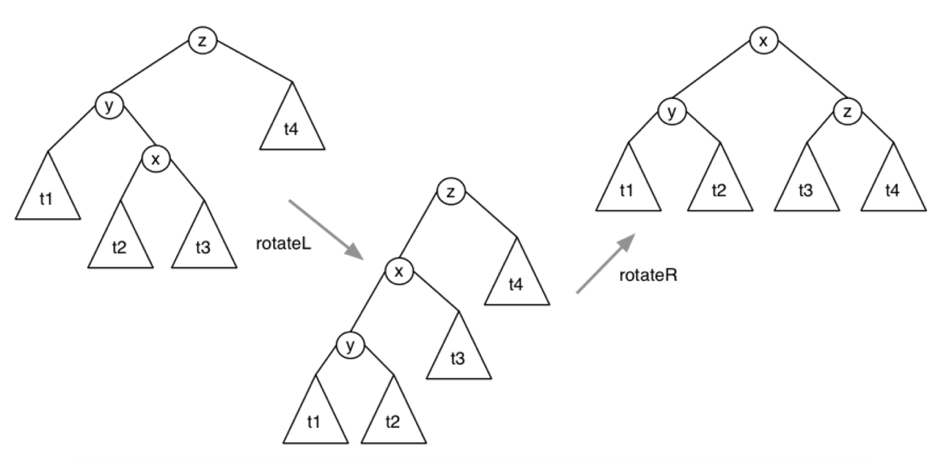
Case 1: Grandchild = left-child of left-child

Case 2: Grandchild = right-child of left-child

Case 3: Grandchild = left-child of right-child

Case 4: Grandchild = right-child of right-child

**Double-Rotation for Right-Child of Left-Child** (#1: Rotate X🡪LEFT then #2: Rotate X🡪RIGHT)



Gives good overall cost:

* Splay has higher insertion cost because of rotations
* Rotations potentially improve balance
* Potentially higher search costs (rotations)
* **Overall search cost is lower**(assuming recently searched items are often searched again)

Need empirical analysis to determine how much better.

**Worst case = O(n)**

**AVL Trees**

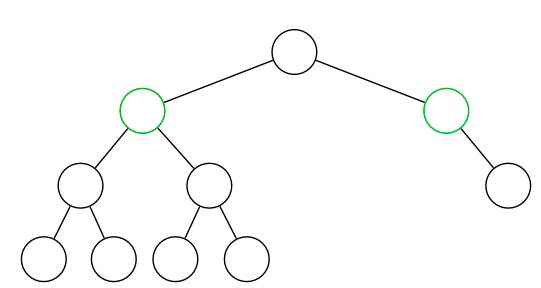
* Approach: AVL trees repairs balance as soon as imbalance is noticed.
  + Repairs are done locally (relative root node), not by the overall tree structure
* **Height / Depth of a tree is what makes searching expensive, not # of nodes**
  + Determining height / depth of tree is expensive
* Check if **LHS DEPTH > RHS DEPTH** or **RHS DEPTH > LHS DEPTH**
* Repaired by **Rotation**
  + LHS subtree too deep = rotateR
  + RHS subtree too deep = rotate

Current version of insertAVL() is inefficient, as it computes **depth()** recursively on same branch.

Can fix this by storing height of subtree in each node.

Analysis:

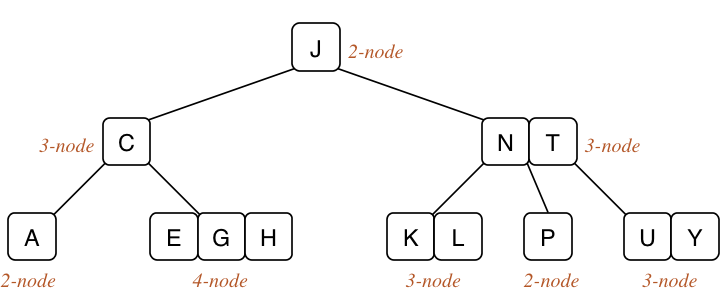
* Trees are **height-balanced**: subtree depths differ by + / - 1
  + For a more efficient operation, store the height in each node (rather than recursively checking)
* Average / worst-case search performance = **O(logN)**
* May not be **weight-balanced**.
  + Subtree sizes may differ. E.g. see tree below
  + However, this does not really matter since we are optimising for search cost and we’ve kept the length of the search path relatively small.



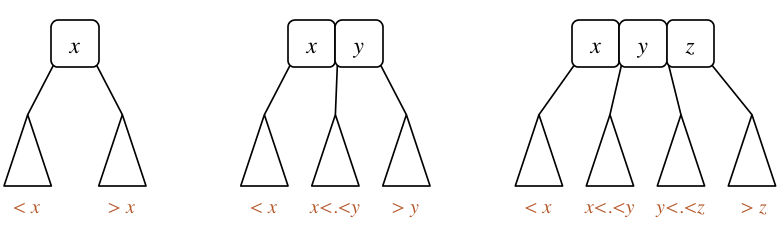
**2-3-4 Trees**

2-3-4 trees have three kind of nodes:

* **2-nodes**, with two children (same as normal BSTs)
* **3-nodes**, two items and three children
* **4-nodes**, three items and four children



2-3-4 trees are ordered similarly to BSTs **(in SEARCH order)**



**Binary search tree:** follow path down to the root and add something down at the bottom and use rotations to move it up.

**For 2-3-4 trees:** All leaves are at the same level and you add items in. If **leaves become** **full / node becomes full**, you **push values upwards** from the leaves. The leaves may stay at the same level while the root goes higher and higher up.

2-3-4 trees cost analysis

* Cost is determined by **depth d**
* 2-3-4 trees are always balanced. Worst case = **O(logN)**
* Worst case for depth: **all nodes are 2-nodes, d = log2N**
  + Same as BSTs
* Best case for depth: **all nodes are 4-nodes, d = log4N**

1. Start with empty tree and store 3 items:AMT

**2. Insert C**

- Too full, so split the nodes up

- A 🡪 LHS subtree / T 🡪 RHS subtree

- M 🡪 New root node

- Add C into appropriate location

**3. Insert H**

- H < M 🡪 Insert into LHS subtree (not full yet)

**4. Insert J**

- J < M 🡪 Do a split on LHS subtree

- Split = promote mid value (C) to parent node  
- Restructure H 🡪 MID Subtree  
- J belongs btwn CM 🡪 Insert into MID Subtree

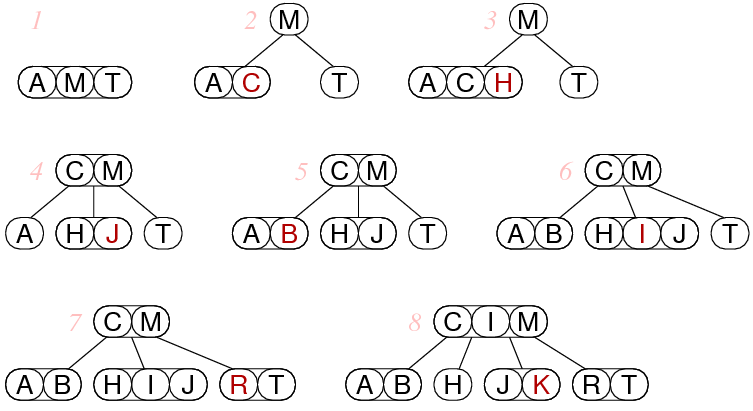
**8. Insert K**

- J btwn CM 🡪 Do a split on MID subree

- Promote mid vale ( I ) to parent  
- J 🡪 insert between IM

* + Balanced tree with branching factor 4

Building a 2-3-4 tree



If **S** is added, it will go between R-T 🡪 RST

If **V** is added, RST is already full.

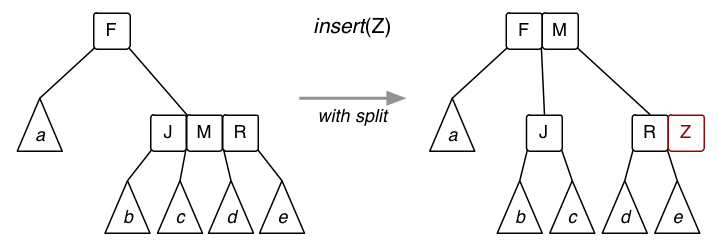
* S is promoted into a node that is full.
* I is promoted to a parent node.
* C becomes LHS subtree to I
* M becomes RHS subtree to I
* Insert S into node with M 🡪 MS
* Insert V 🡪 RTV

**In general 2-3-4 trees would have better search cost as nodes would “fit together” hence reducing depth (which is the main factor for search cost)**

General Summary of 2-3-4 Trees

* Each node contains **1 <= n <= 3 Items** and **n+1 subtrees**
* Same ordering as BST (keys in LHS subtree < smallest key in the 2-3-4 root node)
* New values are inserted at leaves. All leaves are at the same level.
* Trees grows upwards from leaves via **split-promote** (rather than grows from leaves + moves up via. Rotations)

**Insertion into a 4-node (requires a split)**



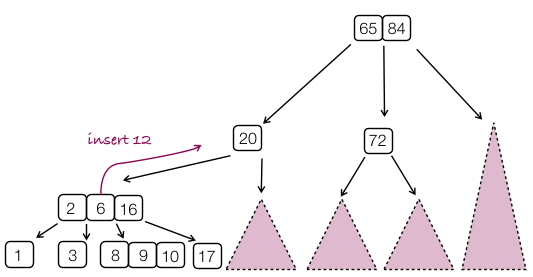
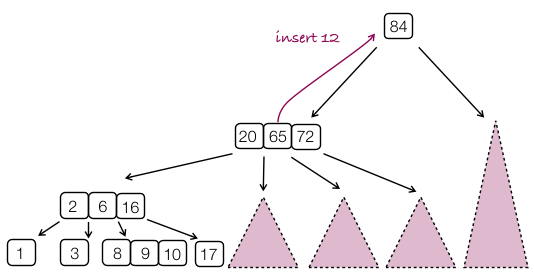
Doing a chain of splits upwards to root:

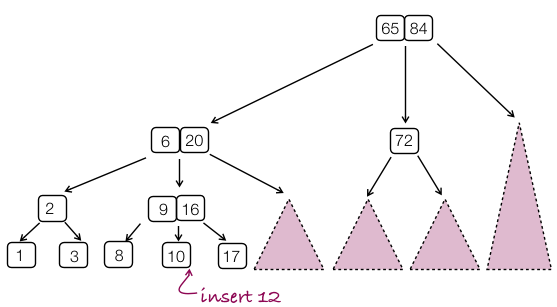
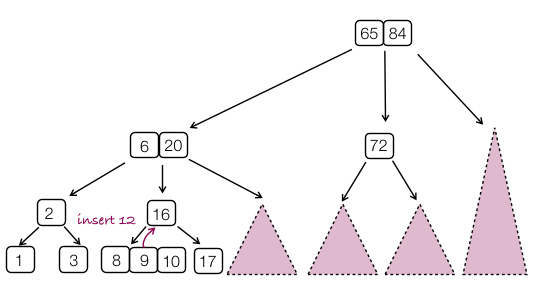
* Is starting from insertion into a leaf 4-node
* Is not the best approach to insertion

Alternative approach to bottom 🡪 up:

* Split 4-nodes attached to 2 or 3 nodes as we go down
* By the time we reach the bottom leaves, there is guaranteed room for insertion

**TOP DOWN SPLITTING STRATEGY**





**2-3-4 Tree TOP-DOWN APPROACH analysis**

**insertion** (into tree of depth d) = O(d) comparisons

* Multiple comparisons in each of d 2-3-4 nodes
* Along with occasional splitting to shift values between nodes

**Search** (in tree of depth d) = O(d) comparisons

* Multiple comparisons in each of d 2-3-4 nodes

Depth of 2-3-4 tree with N nodes = **log4N < d < log2N**

* Worst case log2N similar to BST

Note that all paths in a 2-3-4 tree have same length d

**Variations on 2-3-4 trees**

Variation #1: why stop at 4? Why not 2-3-4-5? Or M-trees?

* Allow nodes to hold up to M-1 items and at least M/2
* If each node is a **disk-page**, then we have a **B-tree (databases)**
  + **I.e. thousands of items in a given node**
  + **M > 100/200/400**

Variation #2: don’t use “variable” (2-3-4) sized nodes

* Use standard BST nodes, augmented with one extra piece of data
* Implement similar strategy as 2-3-4 trees 🡪 red black trees

**Red Black Trees**

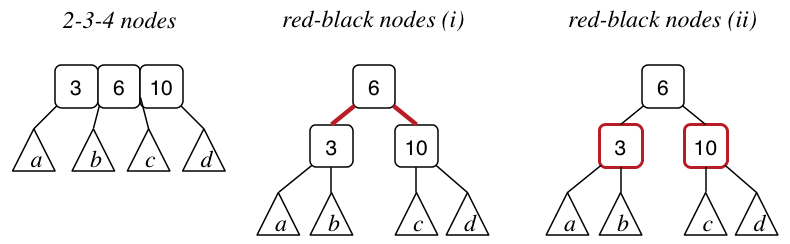
A representation of 2-3-4 trees using BST nodes.

Definition of a red-black tree.

* A BST in which each node is marked **red** or **black**
* No two red nodes appear consecutively on any path
* A red node corresponds to a 2-3-4 sibling of its parent
* A black node corresponds to a 2-3-4 child of its parent

Insertion algorithm: avoids worst case O(n) behaviour

Search algorithm: standard BST search



Representing 4-nodes in red-black trees

**RED NODES** = sibling of its parent node

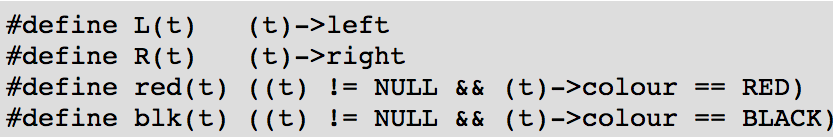
**BLACK NODES** = root

**Red-Black Insertion**

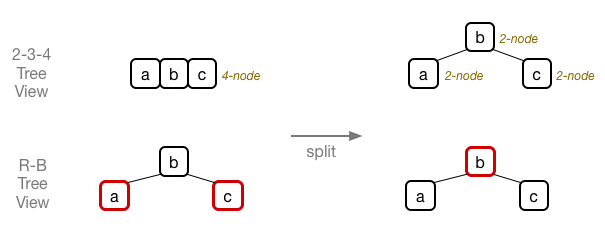
* Need to recall direction of last branch (L or R)
* Need to recall whether parent link is RED or BLACK
* Splitting / promoting implemented by rotateL / rotateR
* Several cases to consider depending on colour / direction combinations

Some initial components of the algo:

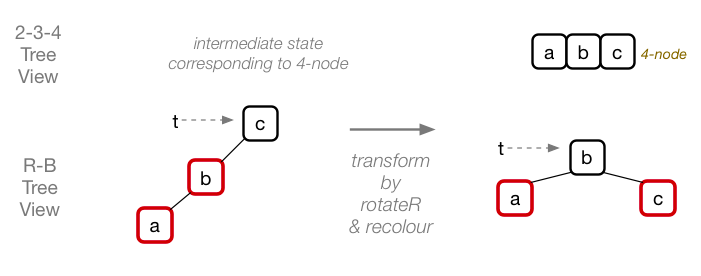
Assume some macros:



Splitting a 4-node in a red-black tree



Checking tree after insertion:



**Red Black Cost Analysis**

* Tree is well-balanced: **worst case search = O(log2N)**
* Insertion affects nodes down one path. Max rotations is **2d** (d = depth)
* Only disadvantage is **complexity of insertion / deletion code**
* Note: red-black trees were popularised by Sedgewick

**Performance Measurement**

Why do we care about performance?

* **Good performance** = less hardware, happy users
* **Bad performance** = more hardware, unhappy users

Even efficient systems have unhappy users if interface is poorly designed.

Generally:

* Performance = execution time
* Other measures: memory/ disk space, network traffic, disk i/o

Strategy for developing efficient programs:

* Design the program well (choice of a good algorithm)
* Implement the program well (write good code)
* Test the program well
* Only after you’re sure its working, **measure performance**
* If (and only if) performance is inadequate, **find the “hot spots”**
* **Tune the code** to fix these
* REPEAT THE **MEASURE 🡪 ANALYSE 🡪 TUNE CYCLE** UNTIL PERFORMANCE IS OKAY

Complexity / estimates give some idea of performance in advance. However:

* Maybe an algo is Log2N but there is a massive constant sitting in there that we have not taken account of, thus end up being slow
* We may forget about a frequently occurring / expensive operation
  + E.g. we assume there is average/worst/best case so it will average out to be average case, but in fact worst case happens instead.
* Best way to evaluate performance: **measure program execution**
* Performance analysis can be:
  + **Coarse-grained** = overview of performance characteristics
  + **Fine-grained** = detailed description of performance

**Coarse-grained performance analysis**

* Devise a range of representation inputs
* Measure execution time of program on each input
* Can conveniently be combined with testing  
  (but we only care about timing if correct result produced)

80/20 rule generally holds: **80% of the execution time is spend in 10% of the code**

* Most of the code has little impact on overall performance
* Small regions of the code are bottlenecks = **hotspots**
* To significantly improve performance: make bottlenecks faster

Need a method for locating **hot spots** (the expensive 20% of code)

* An **execution profile** for a program is:
  + The total cost of performance each **code block**
  + For one execution of the program
* Cost may be measured via.
  + A count of the number of times the block is executed
  + The total execution time spent within that block

Profiles are typically collected at function level (e.g. code block = function)

**Gprof (on Linux)** is a command that that displays execution profiles

* Must compile with the **–pg** flag
* Executing program creates an extra **gmon.out** file
* **Gprof** reads **gmon.out** and prints it on stdout.