COMP20007 Design of Algorithms

Binary Search Trees and their Extensions

Daniel Beck

Lecture 14

Semester 1, 2020

Dictionaries

Dictionaries

- Abstract Data Structure

Dictionaries

- Abstract Data Structure
- Collection of *(key, value)* pairs

Dictionaries

- Abstract Data Structure
- Collection of *(key, value)* pairs
  - Values are usually *records*, such as my videogames or the Unimelb students.

Dictionaries

- Abstract Data Structure
- Collection of *(key, value)* pairs
  - Values are usually *records*, such as my videogames or the Unimelb students.
  - Keys are (unique) identifiers, such as the name of a game or the student ID.

Dictionaries

- Abstract Data Structure
- Collection of *(key, value)* pairs
  - Values are usually *records*, such as my videogames or the Unimelb students.
  - Keys are (unique) identifiers, such as the name of a game or the student ID.
- Required operations:
  - *Search* for a value (given a key)
  - *Insert* a new pair
  - *Delete* an existent pair (given a key)

Dictionaries - Implementations

Dictionaries - Implementations

Unsorted array / Linked list

- Search: $\Theta(n)$ comparisons;

Dictionaries - Implementations

Unsorted array / Linked list

- Search: $\Theta(n)$ comparisons;

Sorted array

- Search: $\Theta(\log n)$ comparisons;

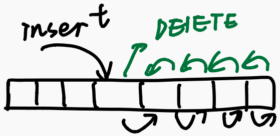
Dictionaries - Implementations

Unsorted array / Linked list

- Search: $\Theta(n)$ comparisons;

Sorted array

- Search: $\Theta(\log n)$ comparisons;
- Insert/Delete: $\Theta(n)$ record swaps;



Dictionaries - Implementations

Unsorted array / Linked list

- Search: $\Theta(n)$ comparisons;

Sorted array

- Search: $\Theta(\log n)$ comparisons;
- Insert/Delete: $\Theta(n)$ record swaps;

This lecture: a better data structure

Dictionaries - Implementations

Unsorted array / Linked list

- Search: $\Theta(n)$ comparisons;

Sorted array

- Search: $\Theta(\log n)$ comparisons;
- Insert/Delete: $\Theta(n)$ record swaps;

This lecture: a better data structure

- Search: $\Theta(\log n)$ comparisons;
- Insert/Delete: $\Theta(\log n)$ record swaps;

Binary Search Tree

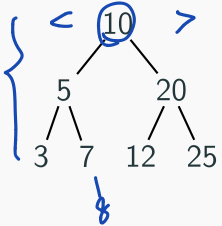
$\boxed{?}$. $\boxed{?}$

Delete

$\theta(\log n) + \theta(\log n)$ bring all subtree up

$\in \theta(\log n)$

$h = \log n$



search

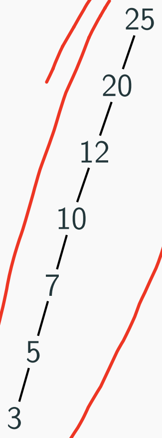
$\theta(\log n)$

Average case

insert $\theta(\log n) + \theta(1) \in \theta(\log n)$

search

Binary Search Tree - Worst Case



Linked List

$\Theta(n)$

for all operations

→ Highly unbalanced

⇓

change to balanced

BST - How to avoid degeneracy?

avoid it to be a linked list

Two options:

BST - How to avoid degeneracy?

Two options:

- Self-balancing
  - **AVL trees**
  - Red-black trees
  - Splay trees

BST - How to avoid degeneracy?

Two options:

- Self-balancing
  - **AVL trees**
  - Red-black trees
  - Splay trees
- Change the representation → NODES to have >1 elements
  - **2-3 trees**
  - 2-3-4 trees
  - B-trees

AVL trees

AVL trees

- Named after Adelson-Velsky and Landis.

AVL trees

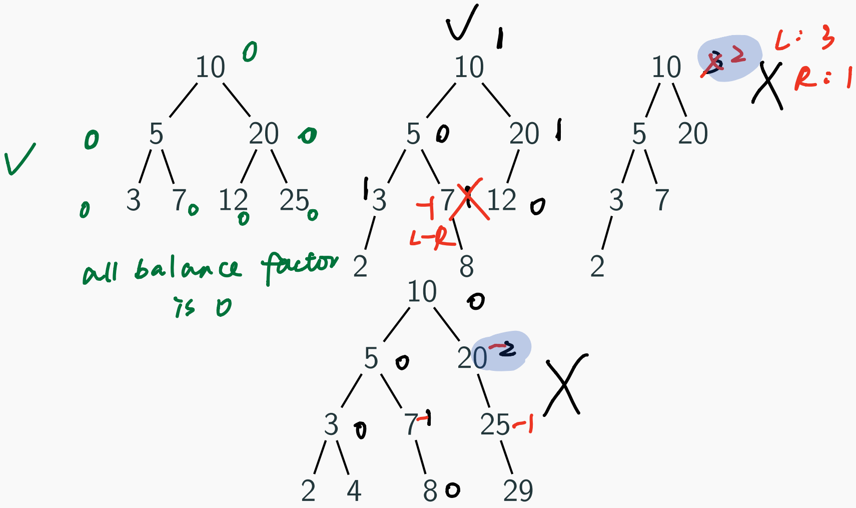
- Named after Adelson-Velsky and Landis.
- A BST where each node has a *balance factor*: the difference in height between the left and right subtrees.

AVL trees

- Named after Adelson-Velsky and Landis.
- A BST where each node has a *balance factor*: the difference in height between the left and right subtrees.
- When the balance factor becomes 2 or -2, *rotate* the tree to adjust them.

AVL Trees: Examples and Counter-Examples

Which of these are AVL trees?



AVL Trees - Rotations

- Search is done as in BSTs.

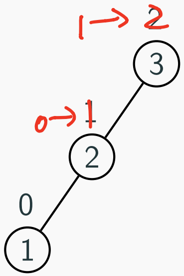
AVL Trees - Rotations

- Search is done as in BSTs.
- Insertion and Deletion also done as in BSTs, with additional steps at the end.

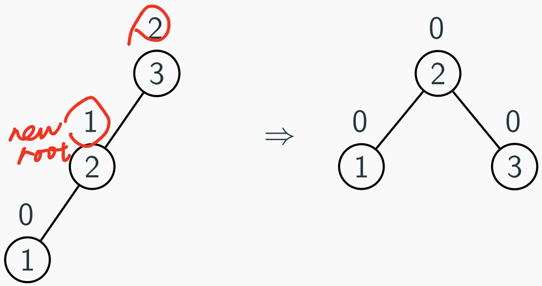
AVL Trees - Rotations

- Search is done as in BSTs.
- Insertion and Deletion also done as in BSTs, with additional steps at the end.
  - Update balance factors.
  - If the tree becomes unbalanced, perform *rotations* to rebalance it.

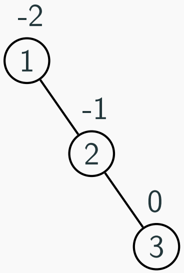
AVL Trees: R-Rotation



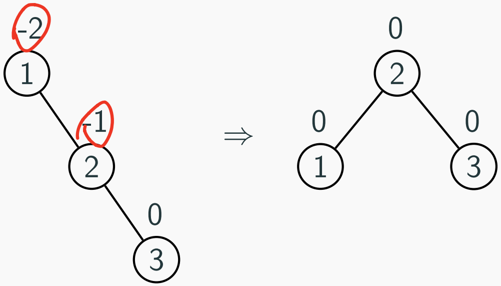
AVL Trees: R-Rotation



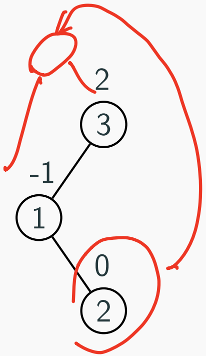
AVL Trees: L-Rotation



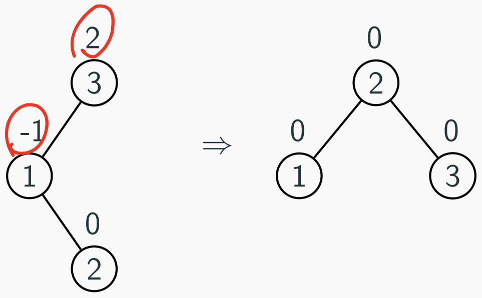
AVL Trees: L-Rotation



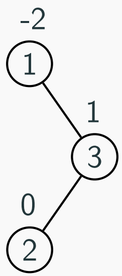
AVL Trees: LR-Rotation



AVL Trees: LR-Rotation

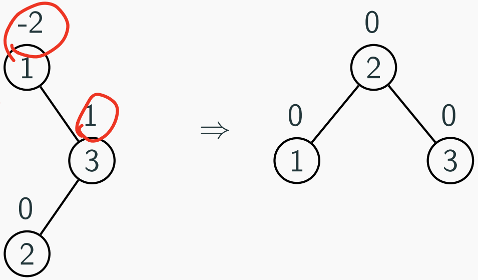


AVL Trees: RL-Rotation



AVL Trees: RL-Rotation

the main while choosing the rotation is to look at balance factor

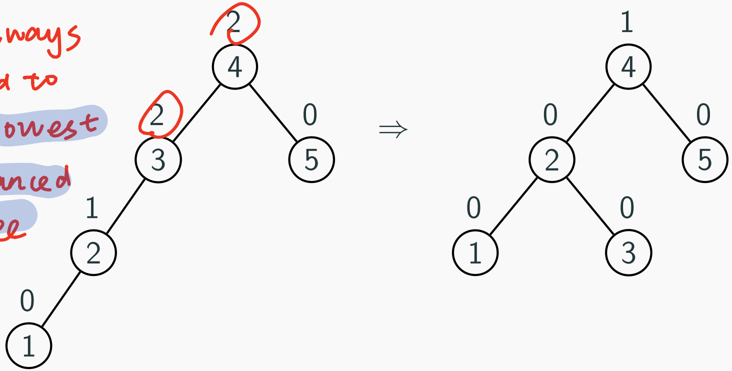


AVL Trees: Where to Perform the Rotation

Along an unbalanced path, we may have several

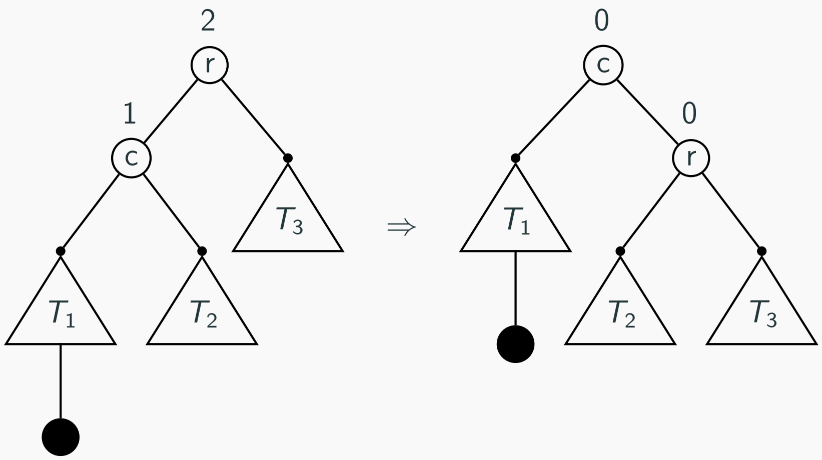
nodes with balance factor 2 (or -2):

we always applied to the lowest unbalanced tree



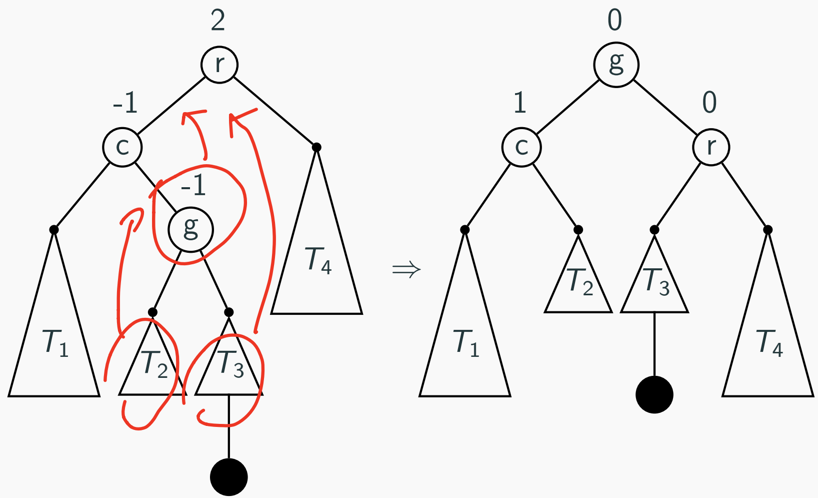
It is always the lowest unbalanced subtree that is re-balanced.

AVL Trees: The Single Rotation, Generally



This shows an R-rotation; an L-rotation is similar.

AVL Trees: The Double Rotation, Generally



This shows an LR-rotation; an RL-rotation is similar.

Properties of AVL Trees

- Rotations ensure that an AVL tree is always balanced.

Properties of AVL Trees

- Rotations ensure that an AVL tree is always balanced.
- An AVL tree with n nodes has depth $\Theta(\log n)$.

Properties of AVL Trees

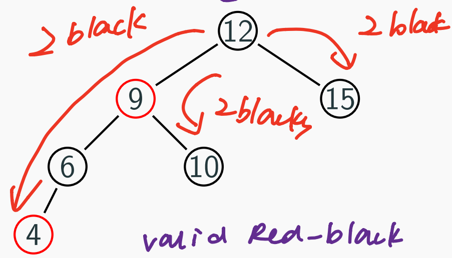
- Rotations ensure that an AVL tree is always balanced.
- An AVL tree with n nodes has depth $\Theta(\log n)$.
- This ensures all three operations are $\Theta(\log n)$.

Red-black Trees

- A red-black tree is another self-balancing BST.
- Its nodes are coloured red or black so that:

not a valid VAL tree

2 balance factor



1. No red node has a red child.
2. Every path from the root to the leaves has the same number of black nodes.

complexity operation

valid Red-black tree

A worst-case red-black tree (the longest path is twice as long as the shortest path).

Self-balancing trees - In practice

AVL trees vs. red-black trees

Self-balancing trees - In practice

AVL trees vs. red-black trees

- AVL trees are "more balanced" but require more frequent rotations.
  - Better if searches are more frequent than insertions/deletions.

more rotation required

when we require less dynamic

more static structure

Self-balancing trees - In practice

AVL trees vs. red-black trees

- AVL trees are “more balanced” but require more frequent rotations.
    - Better if searches are more frequent than insertions/deletions.
- Red-black trees are “less balanced” but require less rotations.
    - Better if insertions/deletions are more frequent than searches.

Self-balancing trees - In practice

AVL trees vs. red-black trees

- AVL trees are "more balanced" but require more frequent rotations.
  - Better if searches are more frequent than insertions/deletions.
- Red-black trees are "less balanced" but require less rotations.
  - Better if insertions/deletions are more frequent than searches.

Key property: rotations keep trees in a shape that guarantees $\Theta(\log n)$ operations.

Representational Changes

Representational Changes

- Goal is the same: keep the tree balanced.

Representational Changes

- Goal is the same: keep the tree balanced.
- But instead of relying on rotations, we will allow *multiple elements per node* and *multiple children per node*.

Representational Changes

- Goal is the same: keep the tree balanced.
- But instead of relying on rotations, we will allow *multiple elements per node* and *multiple children per node*.
- 2–3 trees: contains only 2-nodes and 3-nodes.

Representational Changes

- Goal is the same: keep the tree balanced.
- But instead of relying on rotations, we will allow *multiple elements per node* and *multiple children per node*.
- 2–3 trees: contains only 2-nodes and 3-nodes.
  - A 2-node contains one element and at most two children (as in BSTs)
  - A 3-node contains two elements and at most three children.

Representational Changes

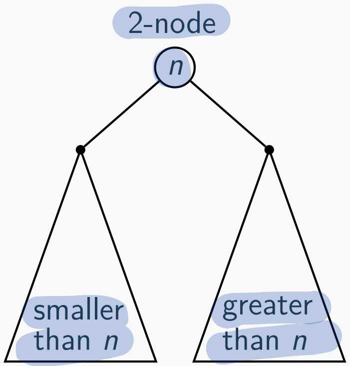
- Goal is the same: keep the tree balanced.
- But instead of relying on rotations, we will allow *multiple elements per node* and *multiple children per node*.
- 2–3 trees: contains only 2-nodes and 3-nodes.
    - A 2-node contains one element and at most two children (as in BSTs)
    - A 3-node contains two elements and at most three children.
- Easy way to keep the tree balanced.

Representational Changes

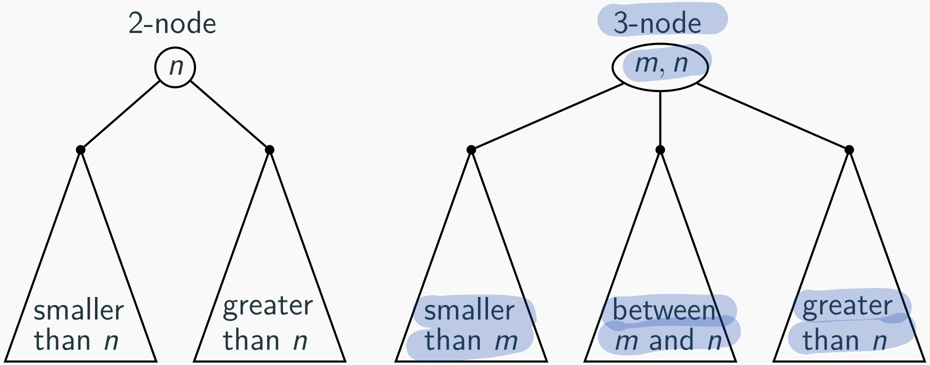
- Goal is the same: keep the tree balanced.
- But instead of relying on rotations, we will allow *multiple elements per node* and *multiple children per node*.
- 2–3 trees: contains only 2-nodes and 3-nodes.
  - A 2-node contains one element and at most two children (as in BSTs)
  - A 3-node contains two elements and at most three children.
- Easy way to keep the tree balanced.
- Can be extended in many ways: 2–3–4 trees, B-trees, etc.

2-Nodes and 3-Nodes

2-Nodes and 3-Nodes



2-Nodes and 3-Nodes



Insertion in a 2–3 Tree

Insertion in a 2–3 Tree

- As in a BST, pretend that we are searching for k.

Insertion in a 2–3 Tree

- As in a BST, pretend that we are searching for k.
- If the leaf node is a 2-node, insert k, becoming a 3-node.

Insertion in a 2–3 Tree

- As in a BST, pretend that we are searching for k.
- If the leaf node is a 2-node, insert k, becoming a 3-node.
- Otherwise, momentarily form a node with three elements:
  - In sorted order, call them k_1, k_2, and k_3.

Insertion in a 2–3 Tree

- As in a BST, pretend that we are searching for k.
- If the leaf node is a 2-node, insert k, becoming a 3-node.
- Otherwise, momentarily form a node with three elements:
  - In sorted order, call them k_1, k_2, and k_3.
  - Split the node, so that k_1 and k_3 form their own individual 2-nodes, and k_2 is promoted to the parent node.

Insertion in a 2–3 Tree

- As in a BST, pretend that we are searching for k.
- If the leaf node is a 2-node, insert k, becoming a 3-node.
- Otherwise, momentarily form a node with three elements:
  - In sorted order, call them k_1, k_2, and k_3.
  - Split the node, so that k_1 and k_3 form their own individual 2-nodes, and k_2 is promoted to the parent node.
  - If the parent node was a 3-node, repeat.

Insertion in a 2–3 Tree

- As in a BST, pretend that we are searching for k.
- If the leaf node is a 2-node, insert k, becoming a 3-node.
- Otherwise, momentarily form a node with three elements:
  - In sorted order, call them k_1, k_2, and k_3.
  - Split the node, so that k_1 and k_3 form their own individual 2-nodes, and k_2 is promoted to the parent node.
  - If the parent node was a 3-node, repeat.

Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7

Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7

2 nodes   3 nodes

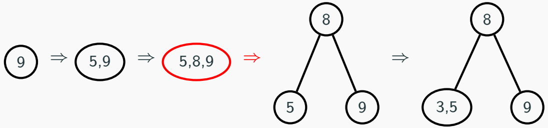
9 $\Rightarrow$ 5,9

Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7

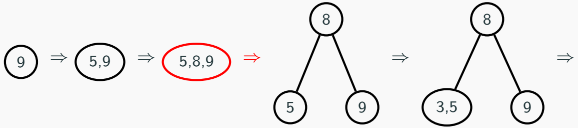
9 $\Rightarrow$ 5,9 $\Rightarrow$ 5,8,9

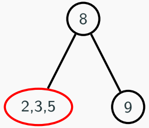
Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7

Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7

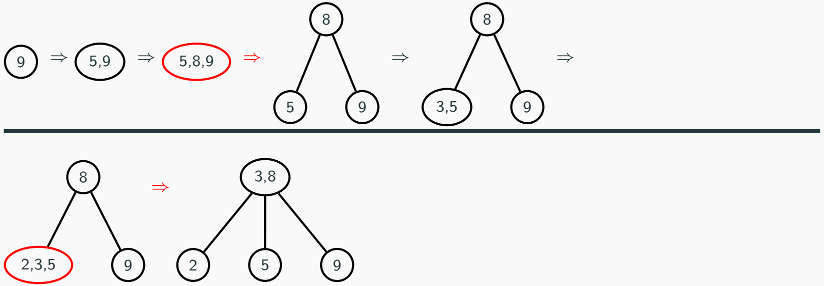


Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7

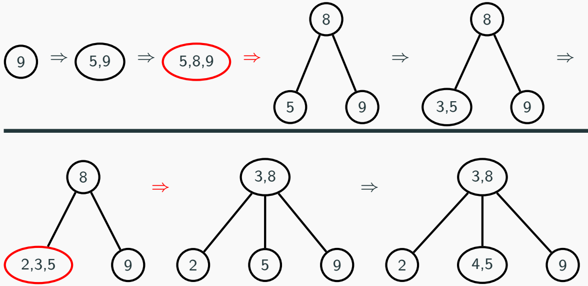




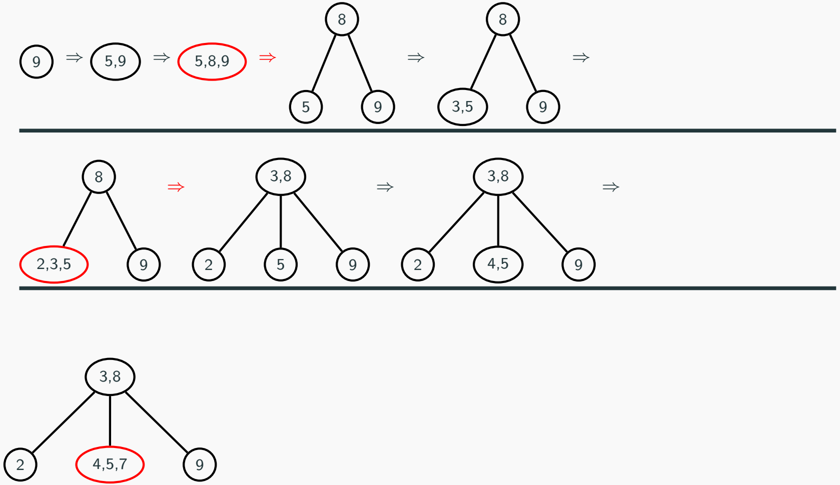
Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7



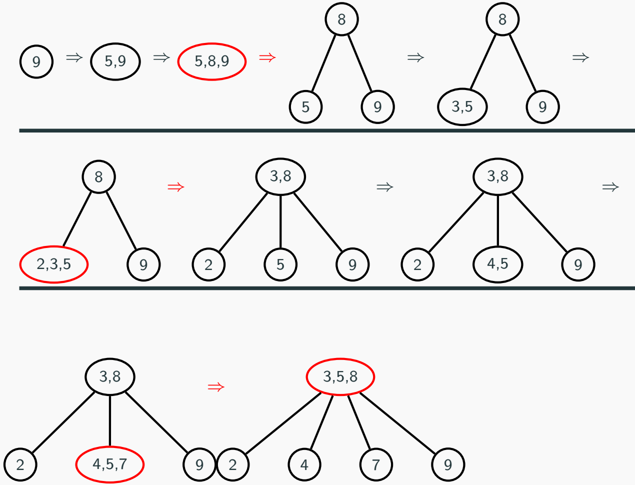
Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7



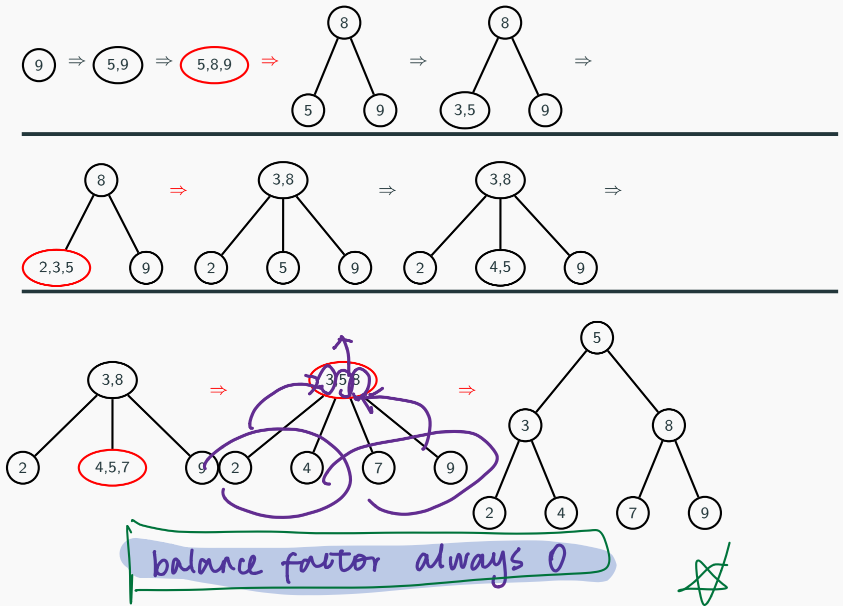
Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7



Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7



Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7



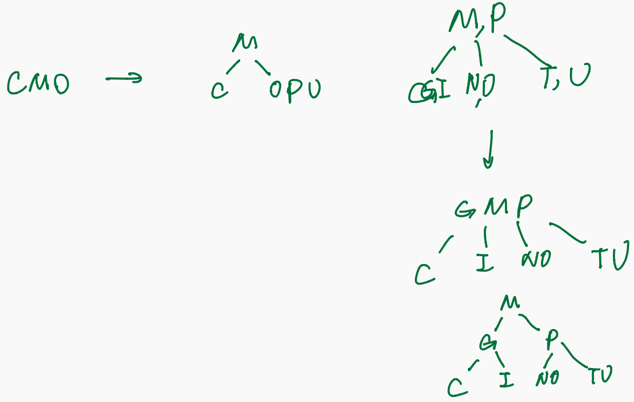
Exercise: 2–3 Tree Construction

Build the 2–3 tree that results from inserting these keys, in the given order, into an initially empty tree:

C, O, M, P, U, T, I, N, G

Extensions

- 2–3–4 trees: includes 4-nodes.



Extensions

- 2–3–4 trees: includes 4-nodes.
- B-trees: a generalisation. (2–3 trees are B-trees of order 3)

Extensions

- 2–3–4 trees: includes 4-nodes.
- B-trees: a generalisation. (2–3 trees are B-trees of order 3)
- B^+-trees: internal nodes *only contain keys*, values are all in the leaves (plus a bunch of optimisations)

Extensions

- 2–3–4 trees: includes 4-nodes.
- B-trees: a generalisation. (2–3 trees are B-trees of order 3)
- B^+-trees: internal nodes *only contain keys*, values are all in the leaves (plus a bunch of optimisations)

Key property: balance is achieved by allowing multiple elements per node.

BSTs - Summary

element
record

- Dictionaries store *(key, value)* pairs.

BSTs - Summary

- Dictionaries store *(key, value)* pairs.
- BSTs provide $\Theta(\log n)$ search, insert and delete operations.

BSTs - Summary

- Dictionaries store *(key, value)* pairs.
- BSTs provide $\Theta(\log n)$ search, insert and delete operations.
- Standard BSTs can degrade to linked lists and $\Theta(n)$ worst case performance.

BSTs - Summary

- Dictionaries store *(key, value)* pairs.
- BSTs provide $\Theta(\log n)$ search, insert and delete operations.
- Standard BSTs can degrade to linked lists and $\Theta(n)$ worst case performance.
  - Self-balancing: AVL trees
  - Change of representation: 2–3 trees

BSTs - In practice

- Self-balancing trees tend to be better when the dictionary is fully in memory.

BSTs - In practice

- Self-balancing trees tend to be better when the dictionary is fully in memory.
    - C++ *maps*: implemented via Red-black trees.

insertion is the
most frequent operation

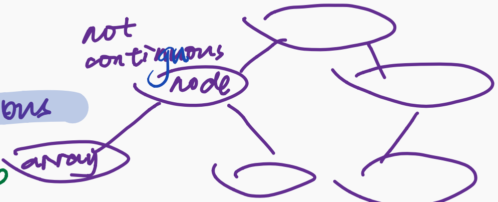
BSTs - In practice

- Self-balancing trees tend to be better when the dictionary is fully in memory.
  - C++ *maps*: implemented via Red-black trees.
- Multiple elements per node are a better choice when secondary memory is involved.

hard drive

BSTs - In practice

- Self-balancing trees tend to be better when the dictionary is fully in memory.
  - C++ *maps*: implemented via Red-black trees.
- Multiple elements per node are a better choice when secondary memory is involved.
  - B-trees (and its variants) are widely used in SQL databases (PostgreSQL, SQLite) and filesystems (Ext4, Reiser4).



contiguous

not much jump

BSTs - In practice

- Self-balancing trees tend to be better when the dictionary is fully in memory.
  - C++ *maps*: implemented via Red-black trees.
- Multiple elements per node are a better choice when secondary memory is involved.
  - B-trees (and its invariants) are widely used in SQL databases (PostgreSQL, SQLite) and filesystems (Ext4, Reiser4).

Next week: C++ maps use BSTs. What about Python *dicts*, do they also use BSTs? (spoiler: no)