Comparison

AVL trees and splay trees achieve O(log n) height by keeping the subtrees from getting too uneven.

2-4 trees and red-black trees achieve O(log n) height by constraining the maximum depth of any element.

- AVL trees, 2-4 trees, and red-black trees all have worst-case O(log n) operations
- splay trees have amortized O(log n) operations, with worst-case O(n) behavior

Comparison

- frequently-accessed elements are near the root in splay trees, resulting in faster access
  - advantageous in applications where there is *locality of reference* – repeated access of related storage locations
- AVL trees are more tightly balanced than red-black trees, so faster retrieval but slower insertion and removal
  - “tightly balanced” → smaller height
  - AVL trees are good for applications where trees are built once but searched often

Comparison

- splay trees have the simplest implementation
  - just BST + restructuring operation – no additional information to store/maintain
- red-black trees are more commonly used than 2-4 trees
  - easily built on top of binary trees – just need to store a color bit
  - simpler to implement than 2-4 trees
- 'find' may restructure a splay tree
  - from a design perspective, having read-only operations change structure is undesirable
  - a consequence is that splay trees are not thread-safe for concurrent finds without extra bookkeeping

Designing Data Structures

Studying balanced search trees reveals two tactics:

- it can be effective to add additional properties to the organization of the elements stored in order to improve runtime of an operation
  - e.g. AVL trees, 2-4 trees, red-black trees
- it can be effective (though harder to analyze) to piggyback local optimizations on other operations
  - e.g. splay trees

In both cases, it is essential that the additional work does not overwhelm the savings gained.