Multiway Search Trees

- non-binary search trees
- d-node
  - d number of children
  - has d-1 keys/node
  - child keys sorted between parent keys
    - keys in first child < parent’s key_1
    - keys in second child > key_1, < key_2
    - etc.
- e.g., B-trees invented by Bayer, McCreight (1972)
(2,4)-Trees

• aka 2-4 Trees, 2-3-4 Trees
• can have 2-, 3-, 4- nodes
• all external nodes at same depth
  • guarantees height is $O(\log n)$
  • sketch of proof in notes
(2,4)-Tree Insertion

• Search (similar to BST search)
• if 2-, 3- node, just insert
• if 4- node, overflow to 5- node
• to fix overflow
  • push $2^{nd}$ (or $3^{rd}$) key to parent
  • split into 2-, 3- nodes (or 3-, 2-nodes)
  • check parent for overflow, may cascade to root!
(2,4)-Tree Deletion

- search
- if not at leaf, swap with beforehand leaf value
- delete item from leaf
- if leaf empty, check siblings
  - if 3-, 4- node sibling
    - push key up to parent, then pull key down from parent
  - if 2- node sibling
    - fuse with sibling, pull key from parent
- could have cascading underflows/fusions
(2,4)-Tree demo

- 2-3-4 trees are B trees with max degree = 4
- won’t implement because they are logically equivalent to red-black trees
Where We Stand

• $O(\log n)$ complexities, but...

• AVL trees
  • cascading rebalancings after deletion

• Splay trees
  • specific use cases

• $(2,4)$-Trees
  • cascading splits/fuses, non-binary
Red-Black Tree

• Invented by Guibas, Sedgewick (1978)
• binary tree version of (2-4)-Tree
  • back to simple BST searching
• rebalances, but not all the time
  • some rebalancing just changes “color”
Red-Black Tree Properties

• Root Property
  • Root is always black

• Red Property
  • Any children of a red node are black

• Depth Property
  • All nodes with 0 or 1 children have the same black depth
  • black depth is number of black ancestors

• Red-Black to (2,4) conversion steps in notes
  • (would probably never actually do, just for logical completeness)
Red-Black Mechanics

• Description longer than the code!
• Webnotes/textbook
• Best description I’ve come across is in
  • https://mitpress.mit.edu/books/introduction-algorithms

• red_black_tree.py
The Red-Black Tree Advantage

• Guaranteed limits on restructurings!
• Insertion – never more than 1
• Deletion – never more than 2

So while all the self-balancing trees have $O(\log n)$ complexities, in practice red-black will probably be faster.
Maps!

- Know the runtime complexities for the 5 essential map behaviors of all the concrete classes
- Be able to compare the tradeoffs of all the concrete classes
Reading the Sample Code

• Digression on factory method design pattern in webnotes
• Most of that is hidden, trust the API
C++ STL and Java SDK

- `std::vector`/`ArrayList`
  - dynamic array ~ Python list
  - recommended default data structure in C++

- `std::map`/`SortedMap`
  - balanced binary search tree
    - (usually a red-black tree)
  - when needing key lookups and sorted data

- `std::unordered_map`/`HashMap`
  - hash table ~ Python dictionary
  - when needing key lookups but don’t care about ordering