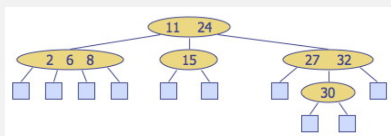
Multiway Search Trees

A *multiway search tree* allows more than one value per node.

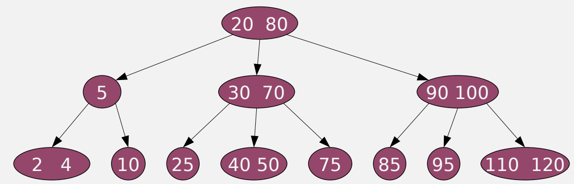
- each node has up to m-1 values, in sorted order
- a node with k values has k+1 children (which may be empty)
- *i*th subtree of a node $[v_1, \ldots, v_k]$ only contains values in the range $v_i \leq v < v_{i+1}$
  - $0 \leq i \leq k$
  - $v_o = -\infty$, $v_{k+1} = \infty$



2-4 Trees

A *2-4 tree* is a multiway search tree where

- all leaves are at the same depth
- each node has 1, 2, or 3 keys and (# keys)+1 children



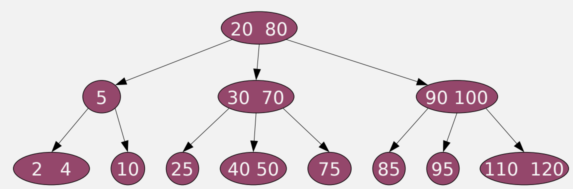
Height of 2-4 Trees

Does this ensure logarithmic height?
→ Yes!

Observe.

- the 2-4 tree with the fewest keys for its height has 1 key per node (complete binary tree)
  - level i has 2^i keys and the whole tree has $n = 2^h-1$ keys
    → h = O(log n)
- the 2-4 tree with the most keys for its height has 3 keys per node
  - level i has 3×4^i keys and the whole tree has $n = 4^h-1$ keys
    → h = O(log n)

Operations on 2-4 Trees



Searching in a multiway tree is similar to searching in a binary tree –
if the target element is not one of the keys in the current node, continue the search with the appropriate child.

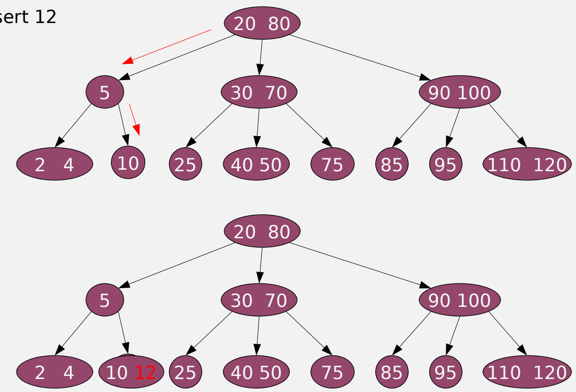
Operations on 2-4 Trees

For insert and remove, we use the same approach as with AVL trees:

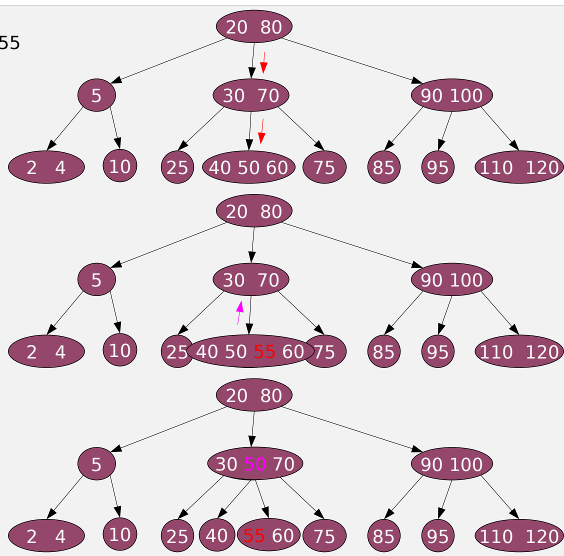
- insert/remove as dictated by the structural and ordering rules
  - new elements are always inserted at a leaf
  - elements can only be removed from a leaf – first swap with next larger (or smaller) as needed
- fix up the broken node size property as needed
  - if insertion creates an overflow –
    - split the node and promote a middle item to the proper place in the parent
    - repeat until there are no more overflows, creating a new root if necessary
  - if removal creates an underflow –
    - if there's a sibling with at least two keys, transfer one (via the parent)
    - otherwise, merge – move a key from the parent, merging the node with a sibling
    - repeat until there are no more underflows, removing the root if necessary

Insert

insert 12

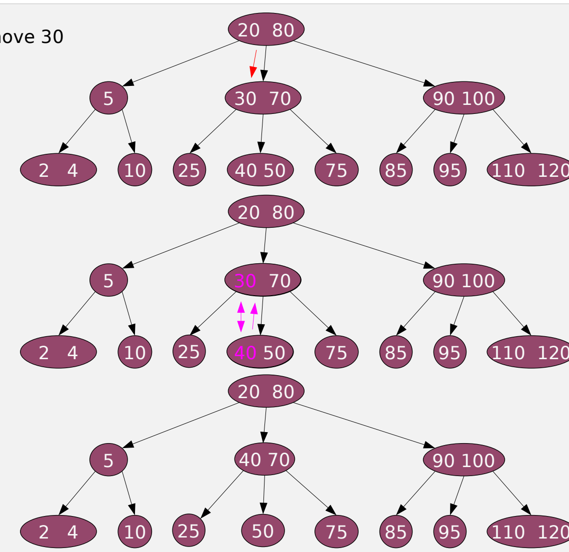


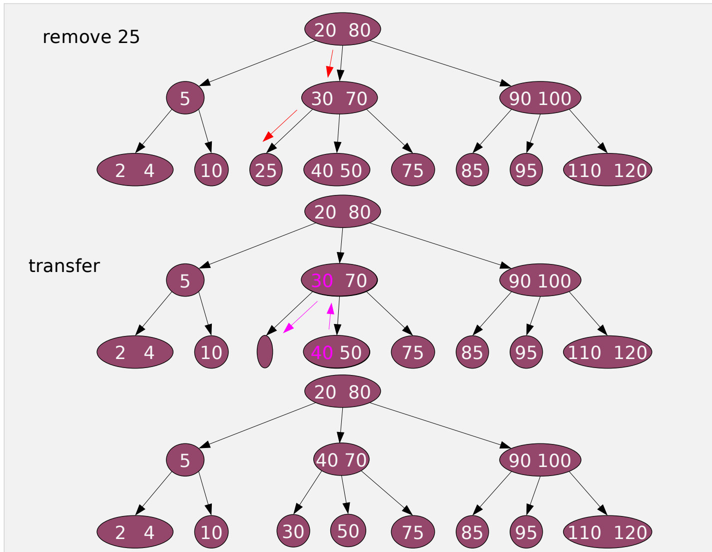
insert 55

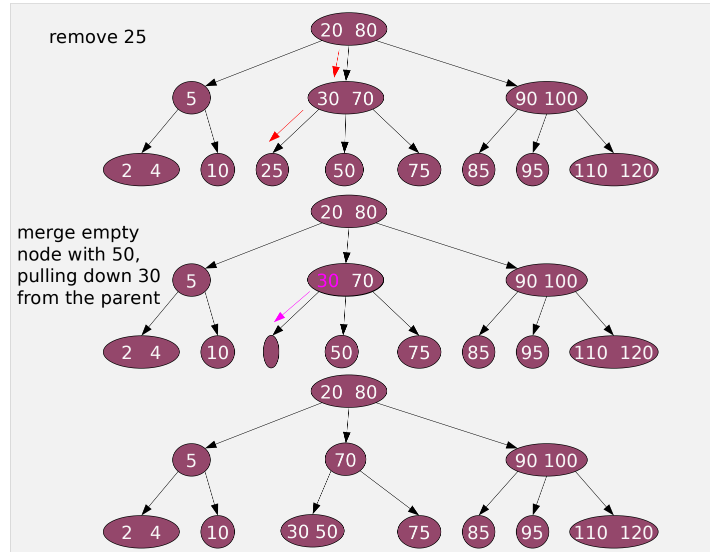


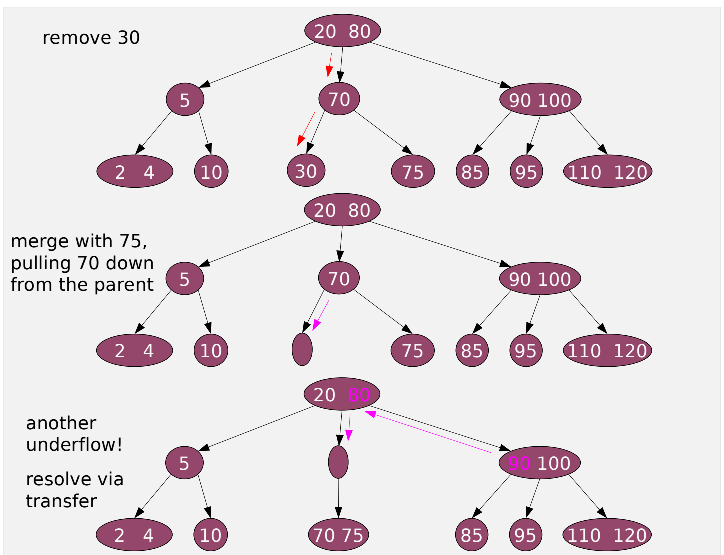
remove 30

swap



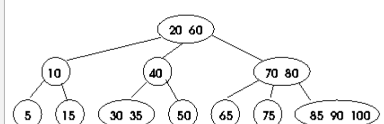






Insert the elements 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 into an initially-empty 2-4 tree. If an overflow occurs, promote the leftmost of the two middle elements. Draw the state of the tree after each insertion.

Remove the elements 35, 80, 75, 40 from the 2-4 tree shown. If swaps are needed, swap the element with its successor. Draw the state of the tree after each removal.



2-4 Trees Running Time

- time for initial insert – O(log n)
- time to fix up one overflow – O(1)
- number of overflows to fix – O(log n)

→ total time for insert – O(log n)

- time for initial remove – O(log n)
- time to fix up one underflow – O(1)
- number of underflows to fix – O(log n)

→ total time for remove – O(log n)

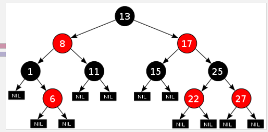
Balanced Search Trees

2-4 trees achieve O(log n) height by fixing the maximum depth of any element. This is made possible by allowing flexibility in the number of elements per node.

Red-black trees have the same idea – logarithmic max depth – but achieve it through flexibility in height rather than in the number of elements per node.

- structurally equivalent to 2-4 trees
  - can create an instance of the other with elements in the same order
  - can map operations on one into operations on the other

Red-Black Trees



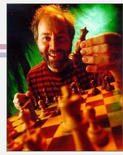
A red-black tree is a BST + coloring rules:

- the root and the (null) leaves are black
- every red node has two black child nodes
- every path from a node to any of its descendant leaves contains the same number of black nodes

Properties.

- O(log n) height
  - longest root-to-leaf path (alternating red and black nodes) is no more than twice as long as the shortest (all black nodes)
- O(log n) insert/remove
  - O(log n) to perform insert/remove
  - O(log n) color changes and at most three restructurings to restore properties

Splay Trees




- invented by Daniel Sleator and Robert Tarjan in 1985

A *splay tree* is a BST + a restructuring operation:

- after each find/insert/remove, that node (or its parent) is brought to the root through *splaying*

Observation.

- frequently-accessed nodes are near the root

Does this ensure O(log n) height?

- on average, yes
- worst case is O(n) – but the worst case is unlikely

Splaying

- *x* is the node being splayed
- *p* is the parent of *x*
- *g* is the parent of *p* (i.e. the grandparent of *x*)

Case 1: zig – applies when *p* is the root

- x is rotated to the root

(the right-handed case is similar)

http://en.wikipedia.org/wiki/Splay_tree

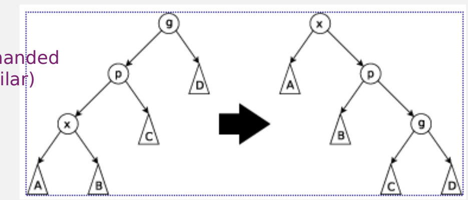
Splaying

- *x* is the node being splayed
- *p* is the parent of *x*
- *g* is the parent of *p* (i.e. the grandparent of *x*)

Case 2: zig-zig – applies when *p* is not the root, and *x* and *p* are both either right children or left children

- p is rotated into g's position, then x is rotated into p's position

(the right-handed case is similar)



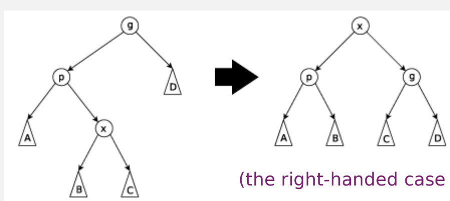
http://en.wikipedia.org/wiki/Splay_tree

Splaying

- *x* is the node being splayed
- *p* is the parent of *x*
- *g* is the parent of *p* (i.e. the grandparent of *x*)

Case 3: zig-zag – applies when *p* is not the root, and one of *x* and *p* is a right child and the other is a left child

- x is rotated into p's position, then x is rotated into g's position



(the right-handed case is similar)

http://en.wikipedia.org/wiki/Splay_tree

Performance

- all operations are O(height) to perform the operation + O(height) splay steps
  - each zig-zig or zig-zag raises *x* two levels, each zig (done at most one per splay) raises *x* one level

  → O(log n) amortized

- worst-case performance
  - splay trees perform as well as optimum static balanced BSTs on sequences of at least *n* accesses (up to a constant factor)
    - "static" = no restructuring of tree after construction
    - "optimal" = tree providing smallest possible time for a series of accesses
  - it is conjectured that splay trees perform as well as optimum dynamic balanced BSTs on sequences of at least *n* accesses (up to a constant factor)
    - "dynamic" = tree can be restructured after construction (e.g. AVL trees, red-black trees)

Splay Trees Takeaways

- another form of restructuring operation
- randomized or heuristic approaches can result in good performance in practice because worst case scenarios are rare
- amortized analysis
  - based on performance over a series of operations