Balanced Search Trees

Two approaches for O(log n) height so far –

- AVL trees keep sibling subtree heights similar
- 2-4 trees fix the depth of leaves
  - made possible by flexibility in the number of elements per node

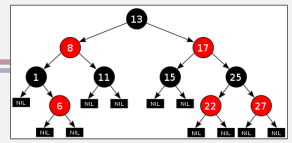
Red-black trees have the same idea as 2-4 trees, but allow flexibility in the depth of leaves rather than the number of elements per node.

- 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
- easier implementation – based on binary trees

Splay trees rely on local optimizations to keep things from getting too unbalanced.

- trade worst-case performance for simpler implementation

Red-Black Trees



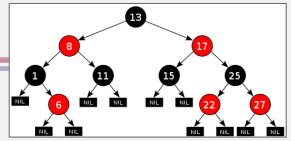
A red-black tree is a BST + coloring rules:

- 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.

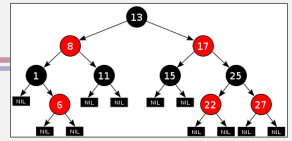
- O(log n) height
  - longest root-to-leaf path (alternating red and black nodes) is no more than twice as long as the shortest (all black nodes)
- O(log n) insert/remove
  - O(log n) to perform insert/remove
  - O(log n) color changes and at most three restructurings to restore properties

Red-Black Trees – Find



- find
  - red-black trees are BSTs, so just do BST find

Red-Black Trees – Insert



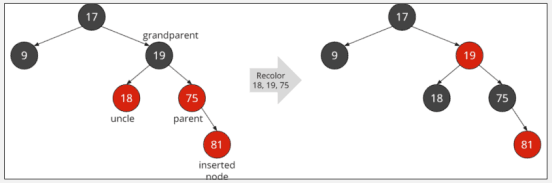
- do BST insert, coloring the new node red
  - to maintain black depth of leaves
- fix up problems
  - coloring rules are violated if the new node is the root (root must be black) or its parent is red (red node must have only black children)

Insert Cases

- new node is the root
  - recolor it black instead

Insert Cases

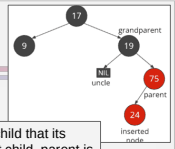
- parent is red and sibling of parent is red
  - flip color of parent, sibling of parent, and grandparent (red → black, black → red)



  - grandparent is now red – repeat repair process with it

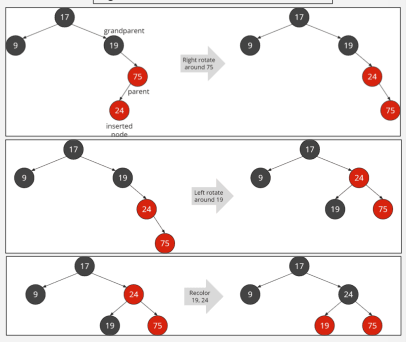
Insert Cases

- parent is red and sibling of parent is black and inserted node is inner grandchild



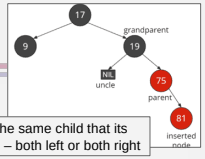
node is the opposite child that its parent is – node is left child, parent is right or vice versa

  - rotate at the parent in the opposite direction of the inserted node
  - rotate at the grandparent in the opposite direction of that
  - color the inserted node black and the original grandparent red
  - repair is complete



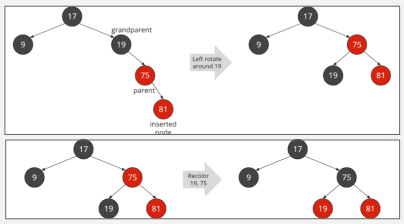
Insert Cases

- parent is red and sibling of parent is black and inserted node is outer grandchild

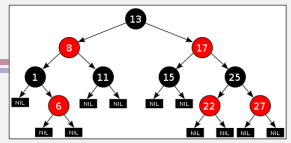


node is the same child that its parent is – both left or both right

  - rotate at the grandparent in the opposite direction of the children
  - color former parent black and former grandparent red
  - repair is complete



Red-Black Trees – Delete



- do BST delete
- fix up problems
  - coloring rules are violated if the deleted node is black and the tree has more than one node (changes the black depth, may result in red parent with red children)

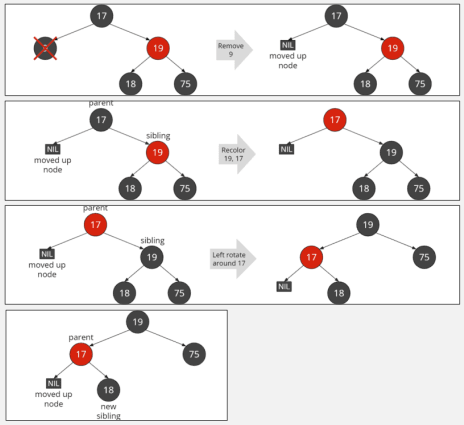
Delete Cases

- deleted node is the root
  - this can only happen if the tree had only one or two (internal) nodes before the deletion
    - if root had two children, the root node would not have been deleted (can only remove above leaf)
  - if the tree had only one internal node, no fix is needed – a black leaf is moved up

  - if the tree had only two internal nodes, the child must be red – it becomes the new root
    - color it black (not shown)

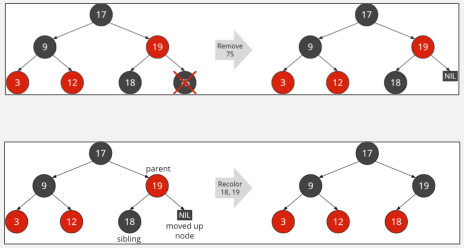
Delete Cases

- sibling of deleted node is red
  - swap colors of sibling and parent (red→black, black→ red)
  - rotate around parent in direction of deleted node
  - sibling is now black – continue repair with those cases



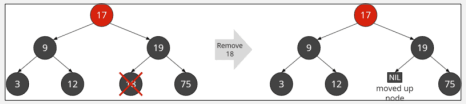
Delete Cases

- sibling is black and has two black children, parent is red
  - swap colors of sibling, parent (black→red, red→black)
  - repair is done



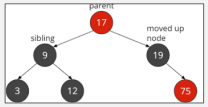
Delete Cases

- sibling is black and has two black children, parent is black



  - color sibling red

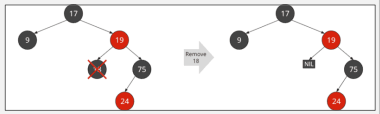
  - continue repair with parent of deleted node (19)



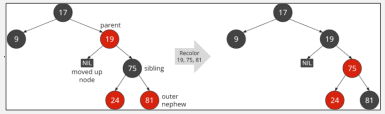
Delete Cases

- sibling is black and has at least one red child, outer nephew is red

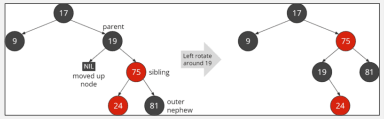
outer nephew is the child of the sibling that is opposite the deleted node – if deleted node is left child, outer nephew is right child and vice versa



  - recolor the sibling in the color of its parent, recolor parent and outer nephew black



  - rotate parent in the direction of the deleted node
    - if deleted node is left child, rotate parent left
    - if deleted node is right child, rotate parent right

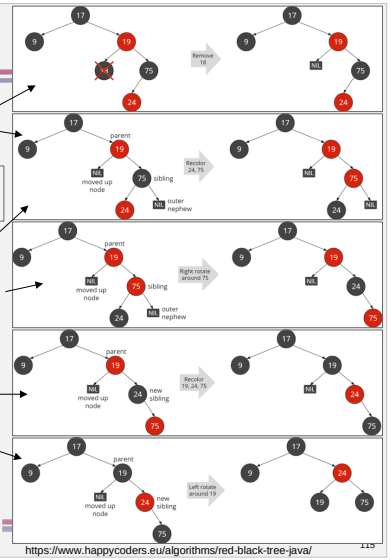


  - repair is complete

Delete Cases

- sibling is black and has at least one red child, outer nephew is black

outer nephew is the child of the sibling that is opposite the deleted node – if deleted node is left child, outer nephew is right child and vice versa



  - swap colors of sibling and inner nephew (black→red, red→black)
  - rotate sibling away from the deleted node
    - if deleted node is left child, rotate sibling right
    - if deleted node is right child, rotate sibling left
  - recolor the sibling in the color of its parent, recolor parent and outer nephew black
  - rotate parent in the direction of the deleted node
    - if deleted node is left child, rotate parent left
    - if deleted node is right child, rotate parent right
  - repair is complete