CIS 313: Intermediate Data Structure

week of Feb 18

seventh week of the term
red-black trees

1. every node is either red or black
2. the root is black
3. every leaf (null) is black
4. if a node is red, both of its children are black
5. for each node, all simple paths from the node to descendant leaves contain the same number of black nodes
example insertions

after insertion of 1, 2, 3, 4, 5, 6 into empty RB tree

let’s continue with 7, 8, ...
insert 7

single left rotate
insert 8

color shift

color shift

green single left rotate (at 2)
insert 9
insert 10
insert 10 (cont’d)

note: 4 as root gets colored black at the end

color shift
RB Deletion

• BST Deletion Revisited: delete $z$
  • If $z$ has no children, then just remove it
  • If $z$ has only one child, then splice out $z$
  • If $z$ has two children, then:
    • Find its successor $y$
    • Splice out $y$
    • Replace $z$’s value with $y$’s value

-> so the physical node deleted is $z$ in the first two cases and $y$ in the third case
RB Deletion

• Delete z as in BST
• If z has two children, when replace z’s value with the successor’s value, keep z’s color (don’t change z’s color)
• Let y be the node being removed or spliced out in this procedure (y would be either z or successor of z, thus y has at most one child)
• If y is red, no violation of the red-black properties, done
• If y is black, some violations might arise and we need to restore the red-black properties
RB Deletion: $y$ is black

• Let $x$ be the child of $y$ before it was spliced out
  So $x$ is either nil (a leaf) or the only non-nil child of $y$
Restoring RB Properties

• The RB-DELETE-FIXUP routine in the text, applied to x
• If x is red, so easy, just change its color to black and done
• If x is black:
  • Transform the tree and move x up, until:
    • x points to a red node, or
    • x is the root
  • At each step:
    • need to consider 8 cases; four when x is a left child and four when x is a right child.
    • due to the symmetry, just consider the 4 cases when x is a left child here
  • REMEMBER: set the color of x to black in the end
Restoring RB Properties: x is black and is a left child

**Case 1:**
- x’s sibling w is red:
  - Left rotate B, switch colors of B and D

**Case 2:**
- x’s sibling w is black; both w’s children are black:
  - Move x up, change w’s color to red

**Case 3:**
- x’s sibling w is black; w’s right child is black, left child is red:
  - Right rotate on w (D), switch colors of C and D

**Case 4:**
- x’s sibling w is black; w’s right child is red:
  - Left rotate on B, switch colors of B and D, change E’s color to black

The nodes with c or c’ can be either red or black.
example: delete 4

replace 4 with it’s successor 5, remove 5’s node (y), x is a nil node (child of 5) and it’s black
delete 4 (cont’d)

case 1: left rotate 8, switch colors of 6 and 8
case 2: change color of 7 to red, move up \( x \) to 6

\( x \) is now red and we are done with the moving up of \( x \)

FINAL STEP: change color of \( x \) to black
delete 4 (done)

NEXT: delete 1 from this
y is 1 and x is the nil child of 1 (black)
deleting 1 from previous

case 2: change color of 3 to red and move up x to 2
delete 1 (cont’d)

case 2 again: change color of 8 to red, move x up to 5

done since x is at root (we’ve reduced the black height of the tree)
remove 2

y is 2, x is child of y (3), x is red, so just need to change color of x (3) to black

NEXT: delete 3 from this
y is 3, x is the nil node (child of y)
x is black
removing 3 from previous

case 1: left rotate 8, switch colors of 5 and 8
remove 3 (cont’d)

set up as case 4:

motivated by a “transfer” in 2-3-4 tree

case 4: left rotate B (5), switch colors of 5 and 6, change color of 7 to black

Done!
remember: all cases come with mirror image

- here x is right child of parent
- the left child of w is red
- fix-up can be completed with a right rotation and color changes
- note that the blue nodes (B and C) can be either red or black
B-trees

• very important data structure in computer science
• database indexing, hard disk referencing, MongoDB, ...
• balanced, multi-way search tree
• many slight variations, we will use definition in CLRS text
• idea is that nodes are large and fit into a disk block (minimum amount of data that’s pulled off a hard drive)
• node size parameters (here called t) depend on disk speeds, block sizes, etc.
B-tree specifications

• fixed parameter t, called *minimum degree*
• nodes have between t-1 and 2t-1 keys
• so therefore they have between t and 2t children
• root is exception: it may have as few as 1 key (2 children)
• all null pointers have the same depth (distance from root)
• a 2-3-4 tree is a B-tree with minimum degree t=2
different texts: things to look for

• top-down versus bottom-up insertion
  • CLRS does top-down, split full nodes during search
  • unlike how it does RB trees
  • bottom-up more common in practice, less wasted space

• ties to left or right
  • no duplicates here
  • need to know for B+ trees, which have all keys at an additional “leaf level”

• left/right bias: if middle key not well defined (when splitting a node with even number of keys)