Topic 14

Tree (Part II)

資料結構與程式設計

Data Structure and Programming

12/25/ 2019

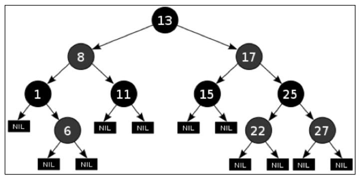
What we have learned in “Tree Part I”...

- Definitions and properties of Tree
- Tree implementation
- Tree traversal
- Binary tree (BT)
  - Complete/Full binary tree
- Binary search tree (BST)
- Balanced binary search tree (BBST)
  - AVL

In “Tree Part II”…

- We will cover more types of BBST
  - Red-Black Tree
  - B-Tree
    - 2-3 Tree
    - 2-3-4 Tree
  - Splay Tree

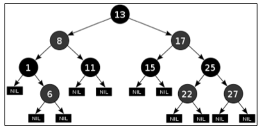
RB Tree – Definition



- A red-black tree is a binary search tree with the following properties

1. A node is either red or **black**.
2. The root is **black**
3. All leaves are black (i.e. All leaves are same color as the root.)
4. Every red node must have two **black** child nodes.
5. Every path from a given node to any of its descendant leaves contains the same number of **black** nodes.

RB Tree – Properties



- ◆ Height-balanced:
  - The path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf
  - ➔ Proof?
  - This ensures the worst case of insert, delete, find operations to be O(log n)

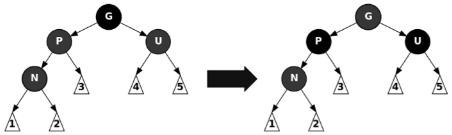
Insert() operation

- ◆ (As in BST) First, insert to the proper leaf
  - Replace the leaf (null node) with a red node with two null leaves.
- ◆ Then, what happens to the 5 properties in p4?
  - 1 ~ 3 holds? // What if new node is root?
  - 4 (red has two black children)?
  - 5 (every path has same #blacks)?

Insert() operation

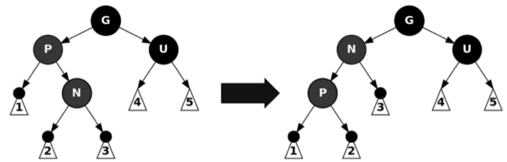
Let N be the new node

1. If N is root ➔ change color
2. If N's parent is black ➔ done
3. If both N's parent and uncle are red
   - Repaint parent (P) and uncle (U) to black
   - Repaint grandparent (G) to red
   - Let N = G, go to 1



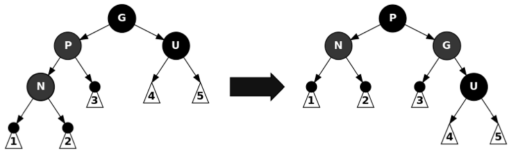
Insert() operation

4. If parent (P) is red, but uncle (U) is black, and N-P-G is zagged
   - Perform a rotation to switch N and P
   - Continue to 5



Insert() operation

5. If parent (P) is red, but uncle (U) is black, and N-P-G is in a line
   - Perform a rotation to switch P and G
   - Done!



➔ At most 2 rotations

➔ Complexity = O(h) = O(log n)

Delete() operation

◆ Remember, we categorize the delete() of BST to 3 cases:

1. Leaf case ➔ properties may be violated
2. One-child case ➔ properties may be violated
3. Two-children case
   - Replaced N with its successor (replace value)
   - Remove successor (becomes case 1 or 2 as successor must have at most one child)

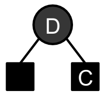
➔ In the following, we focus on case 1 & 2

Delete() operation

◆ Let D be the node to be deleted, and C be its selected child (i.e. the non-leaf child if D has one non-leaf child)

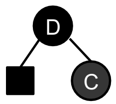
◆ Simple case 1: D is red
- C must be black (property 4)
- C must be leaf (property 5)

➔ Just delete D and replace it with a leaf



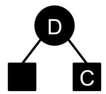
◆ Simple case 2: D is black and C is red
- C's sibling must be black
- Delete D may violate properties 4 and 5

➔ Just replace D with C and repaint it black

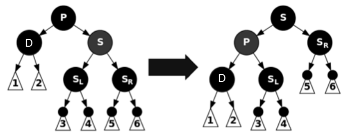


Delete() operation

◆ The only complicate case is that both D and C are black
- C must be a leaf as N has at most one non-leaf child
- Let S be the sibling (if exists) of D

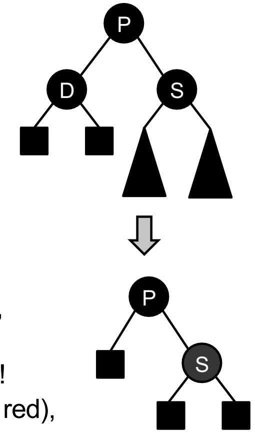


1. If D is root ➔ delete it; done.
2. If S is red
   - Reverse the colors of D's parent (P) and S
   - Rotate P-S
   - Deleting D will violate property 5 ➔ continue to step 4



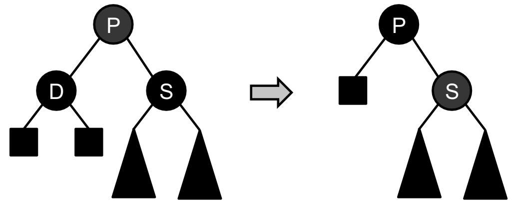
Delete() operation

3. If S is black, P is black
   - All the internal nodes between S and the leaves must be red.
   - From property 4, there must be at most one red node between S and leaves.
   - If both children of S are black,
   - → Both children must be leaves.
   - → Simply repaint S to red. Done!
   - Otherwise (some child of S is red), go to 5 or 6.



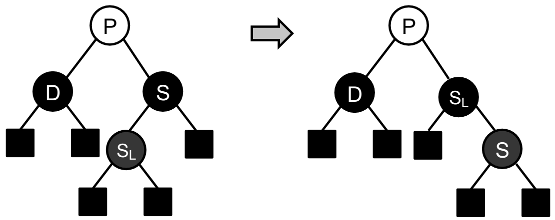
Delete() operation

4. If S is black, P is red
   - If any child of S is red, continue to 5 or 6.
   - Otherwise (both children are leaves), just exchange the colors of P and S. done!



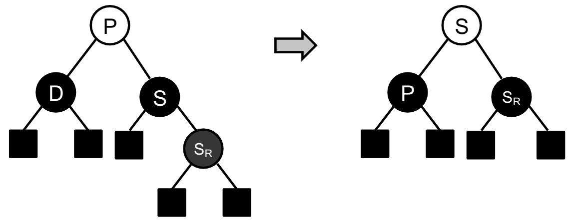
Delete() operation

5. If S is black, its left child S_L is red, right child is black (so it is a leaf), and D is the left child of P
   - Rotate S_L – S, and continue to 6.



Delete() operation

6. If S is black, its right child S_R is red, left child is black (so it is a leaf), and D is the left child of P
   - Rotate P – S, and rotate S – S_R. Done!

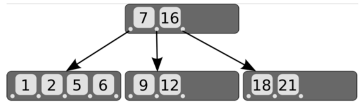


Comparison to AVL tree

- RB tree
  - Offer guaranteed O(log n) for most operations ➔ Best for time-sensitive, robustness-required applications
- AVL tree
  - The heights of left and right sub-trees diff at most 1
  - Generally more rigidly balanced than RB tree
  - Faster in find()
  - More complex insert/delete operations ➔ slower in insert() and delete()
  - Best for applications that DO NOT often alter structure after construction

B Tree – Definition

- A generalization of BST that each node can have more than two children



  - Number of children is usually bounded by a range (e.g. "2-3 Tree" means the number of children is no less than 2, and no more than 3)
  - A B Tree of order m ➔ max #children of a node = m
  - Each node contains a number of keys, which are as separation values dividing its children. A node has at most k children if the tree is of order k+1
  - All the leaf nodes must be at the same level

B Tree – Properties

- A B-tree of order *m* is a tree which satisfies the following properties:
  - Every node has at most *m* children.
  - Every non-leaf node (except root) has at least $\lceil m/2 \rceil$ children.
  - The root has at least two children if it is not a leaf node.
  - A non-leaf node with *k* children contains *k*−1 keys.
  - All leaves appear in the same level
- The root node's #children has the same upper limit as internal nodes, but has no lower limit. (why?)

B Tree – Properties (cont'd)

- A B-tree is usually designed to have #keys between (k, 2k)
  - #children between (k+1, 2k+1)
  - When inserting a key to a node with 2k keys, this node will split into two nodes with k keys and one node with the middle key. This middle key will then be added to its parent.
- B Tree is especially useful when the time to access the data greatly exceeds the time to process the data
  - e.g. Database, file system ➔ The time to access data from disk is far more than the time to process in memory

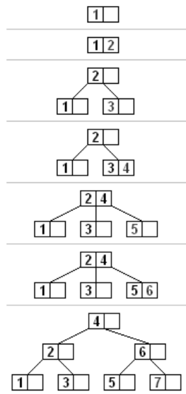
B Tree – find()

- [Note] Usually in HD, each access fetches a "block" of data, say 100, at once
  ➔ We can design a B Tree of order 100
- [Assume] Each disk block access takes ~10ms, and the time to locate the data in a block can be ignored ➔ find() is O(h)
- Given a set with 1M data. Each find() for BST takes $\log_{100}$1M * 10ms = 30ms
  - If the data are stored in a regular BBST, then it takes $\log_2$1M ~= 20 disk accesses (~10ms each) and 20 comparisons (time can be ignored) ➔ 0.2s

21

B Tree – insert()

- Insert(d)
  1. Find a leaf node that d should be placed. Insert d and ensure data in the leaf node is ordered
  2. If no overfloat, done, else pick the medium data and insert it to its parent
  3. Repeat 2 until no overfloat. (Note) When the root is split, the height is increased.



22

B Tree – delete(d)

- Locate and remove d in the tree, restructure the tree if needed.
  1. (For internal node) If d is the separation value for its children, find its successor (from a leaf node) to replace it.
  2. If a node n has the minimum #keys, find its sibling that can spare a data to perform rotation
  3. If all other nodes has the minimum #keys, merge n with its neighbor and the separate key in its parent node
  4. Repeat 2 & 3 until all nodes have more than minimum #keys, or root is reached

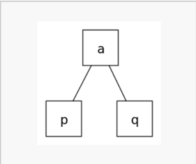
23

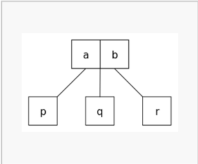
Discussion

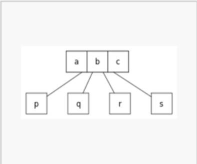
- All operations in a B Tree has $O(\log_K N)$, where K is the order of the tree
  - Assume locating data in a block is relatively fast and its time can be ignored
- In practice, insertion and deletion can be slow when it needs to shift/move data in/among blocks
  ➔ [Solution] Spare some free space in a block for insertion, and mark a node "deleted" instead of remove it from the tree

24

2-3-4 Tree







2-node 3-node 4-node