Binary Search Trees

Richard Hua

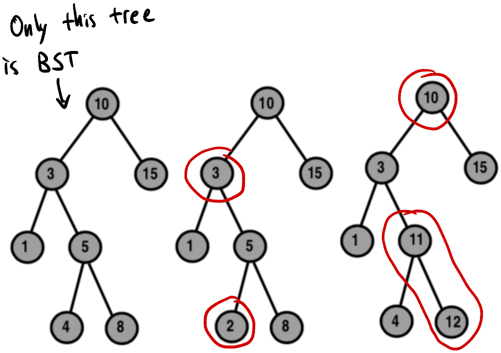
Semester 2, 2021

BST

We will now use BST as an explicit data structure to implement the table ADT.
BST implementation allows efficient search, insert and remove operation.
Additionally BST implementation also support operations such as sorting all keys, finding min (max) key etc.
A lot more complicated then binary heaps.
Support arbitrary node removal (unlike heaps where only the root can be removed).

Example

Left = smaller
Right = Bigger



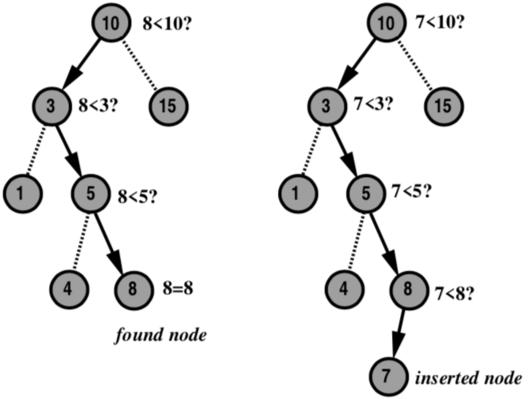
'11' and '12' in left subtree of '10'

The '2' is not in the right subtree, but the root of the right subtree should be bigger.

Continued

The search operation is similar to usual binary search (as we have seen yesterday).
Starts at the root, moves either left or right along the tree, depending on the result of comparing the search key to the key in the node.

Example

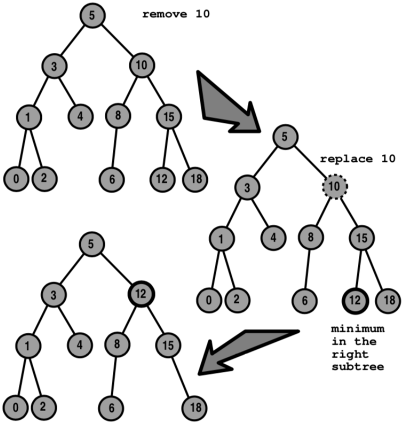

10
8<10?
3
8<3?
15
1
5
8<5?
4
8
8=8
found node
10
7<10?
3
7<3?
15
1
5
7<5?
4
8
7<8?
7
inserted node

Node removal

Node deletion is the most complicated operation since we do want to disconnect the tree after we delete a given node.
The tree has to retain the ordering condition but should not needlessly increase its height after we reconnect everything.
We can delete an arbitrary node v from a BST, there are three cases in general:

- If v is a leaf, we just delete it.
- If v has only one child, we use its only child to replace v itself.
- If v has two children, then we need to replace v by the node with the smallest key in the right subtree of v.

Example


remove 10
5
3
10
1
4
8
15
0
2
6
12
18
replace 10
minimum in the right subtree

Analysis

Lemma

The search, retrieval, update, insert and remove operations in a BST all take time in $O(h)$ in the worst, where h is the height of the tree.

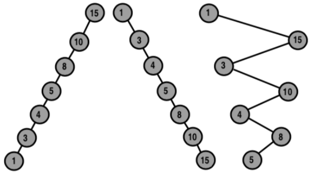
Proof.

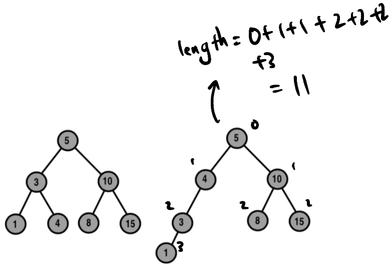
The running time of these operations is proportional to the number of nodes visited. For the find and insert operations, this equals 1 plus the depth of the node, but for the remove operation it equals the depth of the node plus at most the height of the subtrees of the node. Overall, each case is $O(h)$. □

Continued

For well-balanced trees, all operations take $O(\log n)$ time. Since the height is bounded by $O(\log n)$ (recall the height of a complete binary tree).
Repeated insertions and deletions may destroy the balance, and in practice BSTs may be heavily unbalanced.

Example





The worst-case time complexity for search is $\Theta(n)$, and it's even worse than sequential search. Extra overhead in storing the tree structure, creating/deleting tree arc etc..

Continued

The shape of the BST we get depends on insertion order, we wish to know the expected height/depth etc. of the resulting BST with random insertion order. Let's define some notations to make our analysis easier.

Definition

The **total internal path length**, $S_T(n)$, of a binary tree T with n nodes is the sum of the depths of all its nodes.

The average node depth is obviously S_T/n, and this is the average time complexity of a successful search in T.
If we wish to know the average performance of a random binary tree with n nodes, we can compute the sum $S_T(n)$ for all BST of size n and then take the average.
There are $n!$ different insertions orders (some of them will result in the same tree), so the average is $S(n) = \sum S_T(n)/n!$ assuming each permutation happens equally likely.

Continued

Lemma

Suppose that a BST is created by n random insertions starting from an empty tree. Then the expect time for successful and unsuccessful search is $\Theta(\log n)$*. The same is true for update, retrieval, insertion and deletion.*

Proof.

We basically need to show that $S(n) \in \Theta(n \log n)$.
It is obvious that $S(1) = 0$. Furthermore, any $n-$node tree, $n > 1$, contains a left subtree with i nodes, a root at height 0, and a right subtree with $(n - i - 1)$ nodes where $0 \leqslant i \leqslant n - 1$, each value of i being by assumption equally probable.

Continued

Proof.

For a fixed i, $S(i)$ is the average total internal path length in the left subtree with respect to its own root and $S(n-i-1)$ is the analogous total path length in the right subtree. The root of the tree adds 1 to the path length of each other node. Because there are $n-1$ such nodes, the following recurrence holds: $S(n) = (n-1) + S(i) + S(n-i-1)$ for $0 \leqslant i \leqslant n-1$. After summing these recurrences for all $i = 0, \cdots, n-1$ and taking the average, we get $S(n) = (n-1) + \frac{2}{n} \sum_{i=0}^{n-1} S(i)$. And $S(n) \in \Theta(n \log n)$. □

Note that deletion requires that the expect height of a random BST to be $\Theta(\log n)$, which is harder to prove. See the textbook for more details.

Continued

We have learned that in practice, for random input, all BST operations have expected running time of $\Theta(\log n)$.
Worst-case behaviour is still unacceptable in many situations.
Have to do something about it.

Exercise

Exercise

Show how to find the median or an arbitrary order statistic in a BST.

Self-balancing BSTs

These type of BSTs have internal mechanism to balance itself after node insertion/deletion.
Worst-case complexity is $O(\log n)$ which means $S(n)$ for these trees is fairly to the optimal value.
Insertion and deletion is more complicated.

AVL trees

Definition

An **AVL tree** is a BST with the additional balancing requirement that, for any node in the tree, the heights of its left and right subtrees can differ by at most 1.

This condition ensures AVL trees to have height $\Theta(\log n)$. Complete binary trees have too rigid a balance condition which is difficult to maintain when nodes insertion/deletion.

AVL tree analysis

Lemma

The height of an AVL tree with n nodes is $\Theta(\log n)$.

Proof.

Due to the possibly different heights of its subtrees, an AVL tree of height h may contain fewer than $2^{h+1} - 1$ nodes. We will show that it contains at least c^h nodes for some constant $c > 1$, so that the maximum height of a tree with n nodes is $\log_c n \in \Theta(\log n)$. Let S_h be the size of the smallest AVL tree of height h. It is obvious that $S_0 = 1$ (the root only) and $S_1 = 2$ (root plus one child). The smallest AVL tree of height h has subtrees of height $h-1$ and $h-2$, because at least one subtree is of height $h-1$ and the second height can differ by at most 1 by the AVL balance condition.

Continued

Proof.

These subtrees must also be the smallest AVL trees for their height, so that $S_h = S_{h-1} + S_{h-2} + 1$ which is another recurrence we have solved before (see Lecture 8, note that the initial condition is slightly different). If we solve it, we get $S_h \approx \frac{\phi^{h+3}}{\sqrt{5}} - 1$ where ϕ is the golden ratio again. Therefore, for each AVL tree with n nodes, $n \geqslant \frac{\phi^{h+3}}{\sqrt{5}} - 1$, rearrange it and we get $h \leqslant 1.44 \lg(n+1) - 1.33$. □

We omitted some details here, the complete proof is on page 83 of the textbook.

Note that the height of a complete binary tree is at most $\lg(n)$. So an AVL tree is at most 44% taller than the optimal height.

The average height is even better. Although we do not know the theoretical value (it's quite difficult to analyze), empirical estimates show that it's close to $\lg n$.

Continued

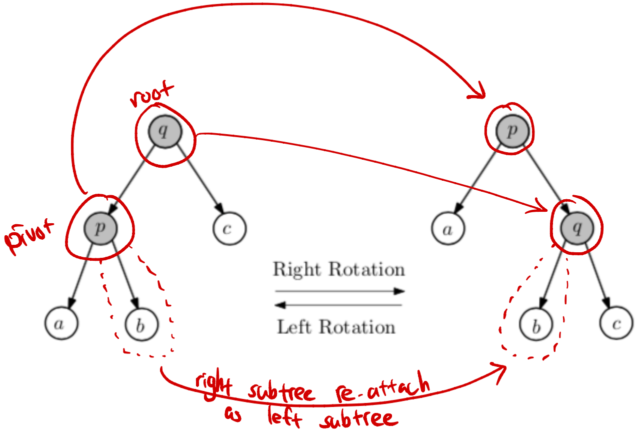
All basic operation in AVL trees have $O(\log n)$ worst-case running time.

Insertion and deletion is more complicated since they may destroy the AVL property.

Need to do something to fix the issue (if it happens).

Do insertion/deletion as in normal BSTs. Check if the AVL condition is violated. If it is, use (a sequence of) **rotations** to fix it.

AVL tree rotation



AVL tree rotation

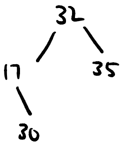
Basic procedure:

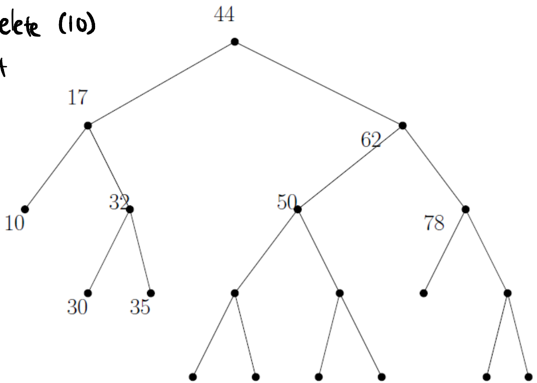
- Identify which node has the AVL condition broken.
- The node with the broken AVL condition will be the root of the rotation.
- The root has two children, the heavier (one with a bigger height value) of the two will be the pivot of the rotation.
- Go to the parent of the root and repeat.

Example

Perfect AVL tree

Example: if we delete (10) then the entire left subtree will look:





Continued

Single rotation is a constant time operation, we only need to change a fixed number of pointer references.

We potentially need to traverse the entire height of the tree so overall time complexity is $O(\log n)$.

Note that there is a special case sometimes, where a single rotation cannot fix the subtree.

Also note that we are basically assuming that we can check whether the AVL condition is violated at a given node in constant time. This can be achieved by storing the left/right subtree height at each node. The difference between the subtree heights is called the **balance factor**.

Overall, AVL trees have better (guaranteed near optimal) time complexity, but is computation heavy and requires more memory space to implement.

Red-black trees

An even more relaxed efficient balanced search tree. More often used in practice (e.g. Java TreeMap class, C++ map). Can be defined slightly differently.

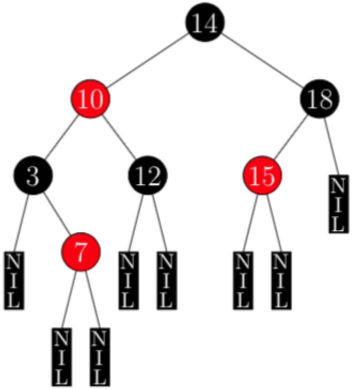
Definition

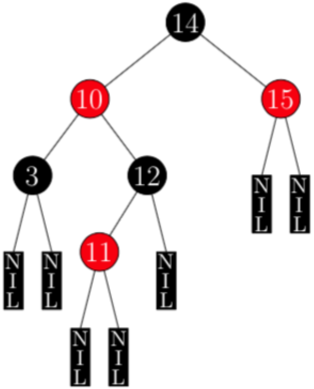
A **red-black tree** is a BST such that every node is colored either red or black, and every non-leaf node has two children. In addition, it satisfies the following properties:

- the root is black;
- all leaves (external nodes) are black;
- all children of a red node must be black;
- every path from a node to a leaf must contain the same number of black nodes. This number is called the **black-height** of the node.

Note that the root condition and the external nodes are not

Example





Continued

Lemma

If every path from the root to a leaf contains b black nodes, then the tree contains at least $2^b - 1$ black nodes (not counting the external nodes).

Proof.

The statement holds for $b = 1$ (in this case the tree contains either the black root only or the black root and one or two red children). In line with the induction hypothesis, let the statement hold for all red-black trees with b black nodes in every path.

Continued

Proof.

If a tree contains $b+1$ black nodes in every path and the root has two black children, then the tree contains two subtrees with b black nodes just under the root and has in total at least $1+2(2^b-1)=2^{b+1}-1$ black nodes. If the root has a red child, the latter has only black children, so that the total number of the black nodes can become even larger. $\square$

Continued

How do we insert nodes into red-black trees?
The hardest property to maintain is the black-height condition.
Adding red nodes doesn't violate this in anyway.
Basic strategy:

- New nodes are always colored red initially.
- If it doesn't violate any conditions then we are done.
- If the parent of the new node is also red, then we do something to fix it.

What to do exactly is quite complicated, so we won't discussed the details here. In short, depending on the situation, you need to do a combination of rotations and re-coloring to fix all the problems.

Continued

Each path cannot contain two consecutive red nodes and increase more than twice after all the red nodes are inserted. Therefore, the height of a red-black tree is at most $2\lceil\lg n\rceil$, and so search is $O(\log n)$.

There is no precise analysis of the average-case performance. Empirical results show that there are about $\lg n$ comparisons per search on the average and fewer than $2\lg n + 2$ comparisons in the worst case.

After insertion/deletion, fixing the tree requires $O(1)$ rotations (unlike AVL trees) and $O(\log n)$ color changes in the worst case.

Note that you can encode the color of a node with a single bit, unlike AVL trees.