Balanced Binary Search Trees

Ralph McDougall

9 February 2019

What is a BBST?

- In graph theory, a tree is a connected, undirected graph that does not contain any cycles
- A binary tree is a tree with one “root” node and where each node has at most two “child” nodes
- A binary search tree is a binary tree where each node is assigned some value and the property holds that for each node *V*, all nodes in the left subtree of *V* have a value less than that of *V* and all nodes in the right subtree of *V* have a value greater than that of *V*
- A balanced binary search tree is a binary search tree where all leaf nodes are as close as possible to the root

Why is a binary search tree useful?

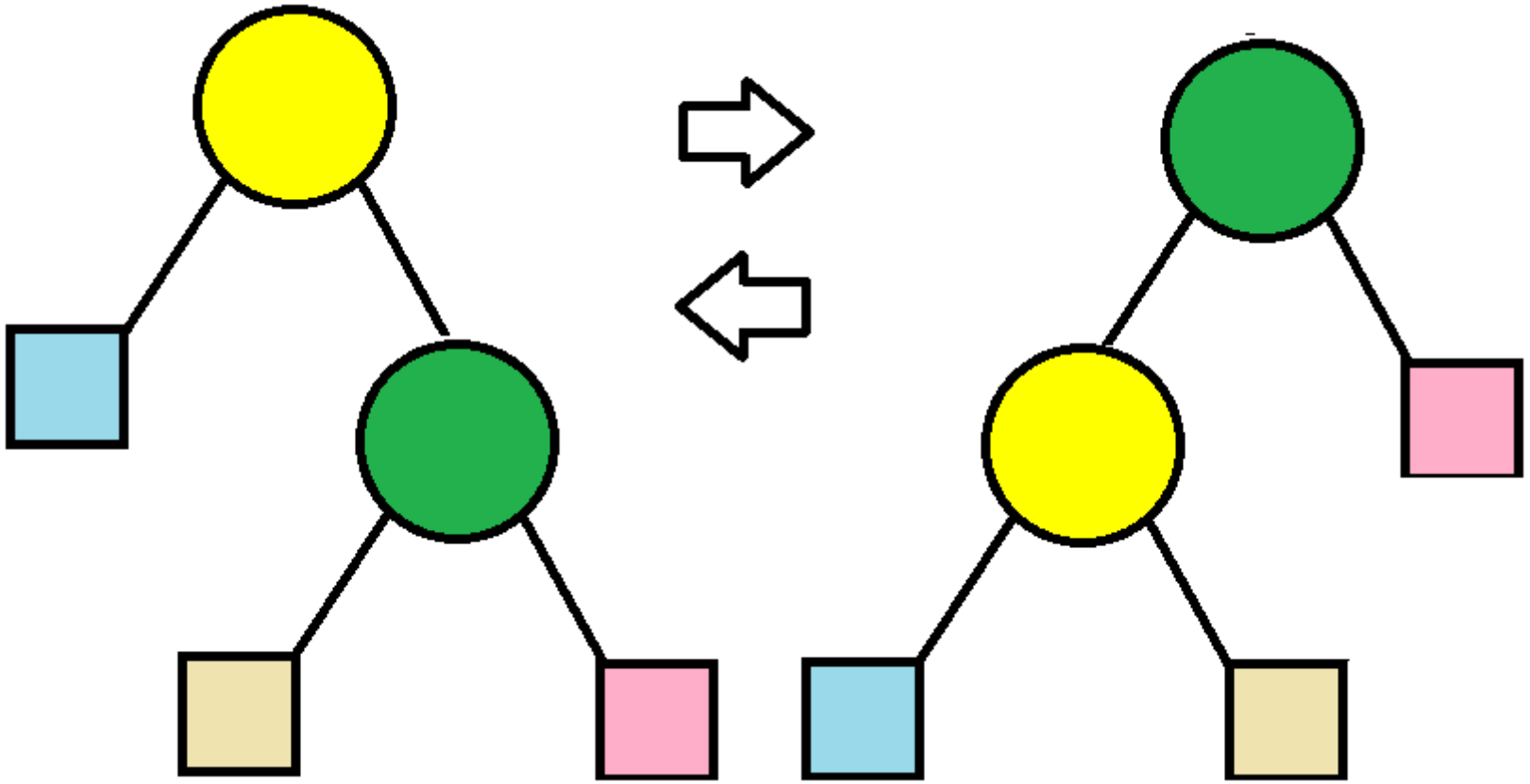
- If you want to query whether a given value is in the tree, you can easily find where it would be if it were in the tree
- If the value *A* is less than the value of some node *V*, then you know that *A* must lie in the left subtree of *V* and vice versa
- This makes querying whether or not a value is in a dataset significantly faster than naively checking every value in the dataset
- Since one would expect the maximum height of the tree to be log*N* where *N* is the number of nodes present, BSTs allow for O(log*N*) queries

Why is a balanced binary search tree useful?

- Optimal BSTs are efficient because of the property that their height is log*N*
- However, it is not guaranteed that the height is always log*N*. If new values are inserted into the tree, one would add them as children of some leaf nodes
- This can lead to the height of the tree becoming very large and thus reducing the runtime to O(*N*)
- A balanced binary search tree introduces extra conditions that must be satisfied at all times when adding values to ensure that the height stays as small as possible

Tree rotations

- In order to restructure the tree, BBSTs use “tree rotations”



Tree Rotations (continued)

- Tree rotations help to shorten the maximum distance from the root to a leaf node
- Different BBSTs use different heuristics to determine where tree rotations should take place

Types of BBSTs

- AVL tree
- Splay tree
- Red-Black tree

AVL Tree

- An AVL Tree maintains the property that the difference between a node's left subtree height and right subtree height is at most 1
- This means that the AVL tree always has as small of a height as possible
- Since all queries are worst case O(*h*) where *h* is the height of the tree, AVL trees allow for guaranteed O(log*N*) queries

AVL Tree (continued)

- AVL Insertions:
  - Insert a node like you would for a normal BST
  - Walk from the leaf node to the root
  - Suppose the nodes visited are V_1, V_2, V_3, …, V_k in that order
  - If V_i does not satisfy the AVL property anymore, perform some tree rotations with V_i, V_{i-1} and V_{i-2}
  - There are 4 cases to consider that affect what tree rotations should be performed (left-left, left-right, right-left, right-right)

AVL Tree (continued some more)

- Deletion:
  - If you want to remove a node from the tree, perform a standard BST deletion
  - Walk up from where the deleted node was to the root and perform rebalancing in the same manner as was done for insertion

AVL Tree Performance

- Queries are guaranteed to be O(log*N*)
- Insertions are guaranteed to be O(log*N*)
- Deletions are guaranteed to be O(log*N*)
- There can be quite a large constant overhead as a lot of rotations may need to be performed during insertion and deletion
- An AVL Tree is thus not the ideal choice if there are a lot more insertions and deletions than queries

Splay Tree

- A splay tree utilises the 80-20 rule
- 80% of the queries are to 20% of the elements in a set in a lot of cases
- Thus, splay trees aim to keep the most recently queried values close to the root

Splay Trees 2: Electric Boogaloo

- Querying a splay tree is initially identical to querying a normal BST
- Once the query is completed, tree rotations are performed repeatedly until the last node visited is the new root node

Splay Tree review

- Query is O(log*N*) (albeit with a fairly large constant)
- Insertion and deletion are O(log*N*) too
- Splay trees are very useful because they are faster than Red-Black trees and AVL Trees in most modern situations
- Splay trees are used in the gcc compiler, the implementation of the Unix malloc and for Linux loadable modules

Red-Black Tree

- Red-Black Tree property:
  - Every node is either red or black
  - The root of the tree is always black
  - If a node is red, it’s children must be black
  - Every path from a node to all of its descendant leaf nodes has the same number of black nodes

Red-Black Tree: The Empire Strikes Back

- For insertion, Red-Black Trees use tree rotations and recolourings
- When a node *V* is added to the tree (standard BST insertion), mark it as red
- If *V*'s parent and uncle are red, make them both black and make *V*'s grandparent red. Repeat this from *V*'s grandparent.
- If *V*'s parent is red and the uncle is black, there are 4 cases of rotations with *V*, the parent and the grandparent. Each case has a specific tree rotation and recolouring that needs to be performed. These aren't too difficult to figure out.

Red-Black Tree review

- All updates and queries are O(log*N*)
- Red-Black Trees utilise fewer tree rotations than other BBSTs, making the faster on average for lots of insertions
- Red-Black Trees should be used when there is a high ratio of insertions to queries

The problem with BBSTs

- All insertions and queries can be performed in O(log*N*) with all of the trees that have been covered
- Which tree you want to use depends on the constraints of the scenario
- A problem arises: the people that create test data often create pathological test cases that are designed to break commonly used data structures
- Test-case authors can predict how these trees will look, so they know what cases test them to their limits

The Solution

- How can the test-case authors predict how your tree will look if your program doesn’t even know how the tree will look?
- Random numbers come to the rescue!


AVL Tree
Red-Black Tree
Splay Tree
This is brilliant.
Treap
AVL Tree
Red-Black Tree
Splay Tree
But I like this.

Treap

- A Treap is the amalgamation of a tree and a heap
- (By this point in the lecture, you should hopefully know what a tree is)
- A heap is a data structure with the property that all of a node's children have a value less than it (or larger than it for a min-heap)
- Queries are performed identically to any BST

You can never have enough Treaps

- Insertion:
  - Each node that gets inserted is assigned a random priority
  - The node gets inserted in the tree according to the heap property on the priorities
  - The value of the node is used to decide whether it should be inserted into the left or the right subtree
  - Assuming the priorities assigned are true random numbers, the treap will remain reasonably well balanced at all times

Treap Review

- All queries and insertions are O(log*N*)
- No matter what test data is given, the treap should always be balanced allowing for very fast queries
- It is a good idea to seed your random number generator since your submissions should always run identically if the same input data is given

TL;DR

- Treaps are cool