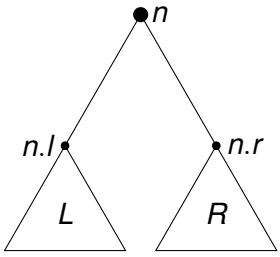
Cosc 201
Algorithms and Data Structures
Lecture 12 (1/4/2026)
BSTs traversals and balance

Brendan McCane
brendan.mccane@otago.ac.nz
And
Michael Albert

Traversing a BST



There are three different traversals we commonly consider in a BST. They are all based on the picture to the left.

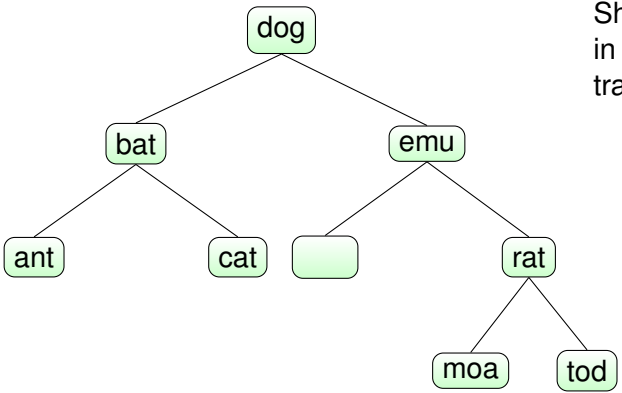
We need to "visit" n, visit all the nodes in L and visit all the nodes in R. The three different traversals represent three different choices of what order to make those visits in.

Preorder
Visit n
Traverse L
Traverse R

Inorder
Traverse L
Visit n
Traverse R

Postorder
Traverse L
Traverse R
Visit n

Traversal example



Showing the keys of the nodes in the order visited by the three traversals:

Preorder	dog, bat, ant, cat, emu, rat, moa, tod.
Inorder	ant, bat, cat, dog, emu, moa, rat, tod.
Postorder	ant, cat, bat, moa, tod, rat, emu, dog.

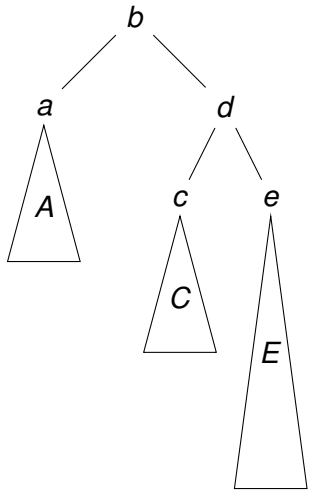
Long branches are a problem

- The performance bounds for all of the BST operations (except traversal) are linear in the length of the longest branch.
- Ideally, we'd like our BST's to look more like heaps (wide and shallow) than long spindly branches.
- Unlike heaps though, the structure of a BST is out of our control – it depends on the order in which elements are added.
- In particular if they are added in their natural order we'll get one great big long branch and the BST performance will degrade to that of a linked list.
- A BST in which there are no "long" branches is called *balanced*.
- So, are there mechanisms by which we can ensure balance? Can they be implemented with limited additional overhead?

Is balancing even possible?

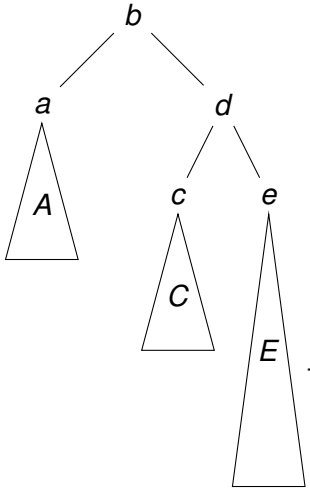
- In a global way, yes.
  - Perform an in-order traversal of the whole tree and save the results to a (sorted) array.
  - Now repeatedly bisect the array - use the middle element as the root
  - Recursively build the left and right subtrees from the part before the middle and the part after.
  - But, this hardly meets the “limited additional overhead” criterion!
- We need some local operation that helps to restore balance.
- The name of this operation is rotation.
- So, what’s that?

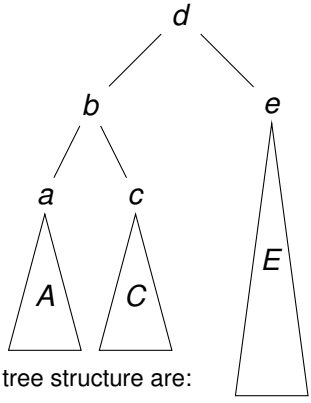
The bad situation (and its fix)



- The *height* of a node in a BST is the length of a longest chain from it to a leaf. The *tree's height* is the height of its root.
- Time complexity of operations in a BST is directly related to height.
- We want to keep height small, i.e., have the tree wide and bushy.
- Imagine that there is a (much) longer chain in E than elsewhere.
- To fix the problem, the idea is to *rotate d* upwards.
- In this view, that makes it the root.

Side by side





The internal changes to the tree structure are:

- Change *b*'s right child to *c*.
- Change *d*'s left child to *b*.
- Change *d*'s parent to *b*'s current parent and make *d* the appropriate child of that parent.
- Change *b*'s parent to *d*.

What to do and when to do it?

- That's the big question!
- The basic idea is to modify the `add` and `delete` operations of the BST to be (somewhat) *self-balancing*.
- That requires a rule for when a tree is "balanced enough" and strategies for fixing problems when the rule is violated.
- I'm going to (briefly!) discuss three options:
  - AVL-trees (historically important)
  - Red-black trees (the form used in Java's `TreeMap`)
  - Treaps (a link between heaps and trees that uses the power of randomisation)
- Another important structure for the same purpose is the B-tree but this is *not* a binary tree.

AVL trees

- In an AVL-tree each node maintains some extra information: the difference between the height of its right subtree and that of its left subtree.
- Balance is maintained by ensuring that *at every node* this value always belongs to $\{-1, 0, 1\}$.
- What's the least possible number of nodes in an AVL tree of height k?
- Basically, it's Fibonacci again! (which grows exponentially)
- The size of an AVL-tree is exponential in its height, and therefore the height is logarithmic in the size.
- Operations are much like the basic ones for BST's but then we need to check (and fix) any excess imbalance along a single path from the affected leaf node back up to the root.
- For insertions, at most three rotations are required, for deletions the worst case is $O(\log n)$.

Red-black trees

- In a red-black tree each node is either red or black.
- There are rules:
  - The root node is black (optional)
  - All `null` nodes are considered black.
  - A red node may not have a red child.
  - Every path from a node to a descendant `null` node contains the same number of black nodes.
- These guarantee that the longest path from root to `null` (which could alternate red and black) is at most twice as long as the shortest path (which could be all black).
- That in turn implies that the height is logarithmic in the size since the tree must be complete to the depth of half the height.
- Operations that modify the tree require (worst-case) $O(\log n)$ recolourings (and on average a constant number) and not more than three rotations.

Treaps

- If we knew that items were to be added to a BST in random order then getting a badly unbalanced situation would be possible, but highly unlikely.
- There's a relationship with quicksort here – if quicksort is applied to a list in random order (or with randomly chosen pivots) then it's very (very) likely to run in $O(n \log n)$ time.
- A *treap* (portmanteau of tree and heap) is designed to achieve this even if the elements are **not** added in random order.
- The idea is when we add an element, we give it a random priority. Then, after doing normal BST insertion we perform a series of rotations to fix the heap-ordering issues.
- The net effect is that the elements look as if they were inserted in descending order of priority. Since the priorities were randomly chosen, that means that at any time we see a BST which "thinks" that its elements were added in random order.

Where's the code?

- A very good question!
- Manipulating the underlying structure of a linked data structure can be tricky.
- It's an interesting exercise, but can be tedious. Try treaps if you're interested.
- In practice, in Java use a `TreeSet` (for sets) or `TreeMap` (for maps).