Binary Search Tree Traversals

September 18, 2018

Depth traversals
Pattern of writing data at current node (C) recursing right (R) and left (L), in different orders.
Pre-order: CLR
In-order: LCR (outputs data in sorted order)
Post-order: LRC
Each should be implemented recursively and has complexity O(n).

Bredth traversals
Level order: nodes are arranged by depth. Not recursive.
1. Add root to a queue
2. Iteratively remove the next node from the queue
3. Record the removed node data
3. Enqueue the removed node's children

Common errors (homework)
Using compareTo with $-1, 0, 1$ instead of $< 0, 0, > 0$.
When removing a node with 2 children, calling remove on the successor.
Forgetting to "reinforce" the root (it should be root = add(root,data)).
Doing depth traversal iteratively..
Doing level traversal recursively.

Heaps

Two properties to structure:

1. Shape: heaps are complete (each level is filled, except for the last one, which is filled left-to-right). Heaps are not BSTs.

2. Order: min. heap (every node's data is less than the one of its children, and the smallest item is the root). Max. heap is the opposite (every node's data is greater than the one of its children, and the largest item is the root).

Implementation

Usually as an array, because complete trees do not have gaps.
We leave the 0 index blank, so that we can easily access children and parents.
For a node at index i, its parents would be at $\frac{i}{2}$ (floor, or integer division). Its left child would be at $2i$, and its right child, at $2i + 1$.
This structure requires less memory because we are not storing relations between parents and children.

Operations with heaps

* Adding
1. Add a node at the end of the last level (that is, size + 1), to maintain shape property
2. Perform heapify/upheap to bring back the order property:
Compare the node with its parents. If the order property is violated, swap the node with its parent.
Continue swapping until the order property is not violated.

* Accessing the smallest/largest
O(1), and it always corresponds to what is most efficient for the heap type (max/min).

* Removing
1. Save the root data (as we will only remove root)
2. Move the data from the last node to the root that was just removed, to maintain shape property
3. Perform heapify down (in min heap, swap with smalled child)
Continue swapping until the order property is not violated.

* Building a heap
Given an array of data with index 0 null, make a heap.
Option 1 would be to call add() on each data, but this would be O(n log n).
Option 2: Build Heap.
Start at index size/2 (the last index that has a child)
Call heapify down from index size/2 to index 1.
Complexity: O(n).

	add	remove	upheap	downheap	buildheap
average/worst	log n	log n	log n	log n	n

Priority queues

ADT backed by a heap. A wrapper around a heap, with the same efficiencies.
enqueue: add; dequeue: remove