BSTs and Heaps

1 Tree Traversals

1.1 Introduction

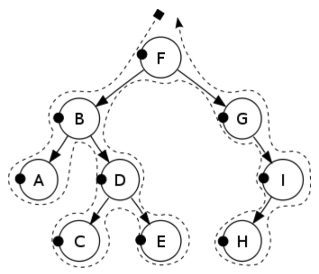
Tree traversals are useful when we want to visit all the nodes in a tree. The order in which we 'visit' these nodes can vary depending on how we implement our traversal. Feel free to imagine 'visiting' as printing a node instead– the point is that we want establish what order we see each node in.

Why is this useful? Oftentimes, our application will require a certain order of 'visits'. For instance, in-order traversal over a BST gives back the values in sorted order. Post-order traversal is also used to make a topological sort, a type of sort on directed acyclic graphs that we will learn later.

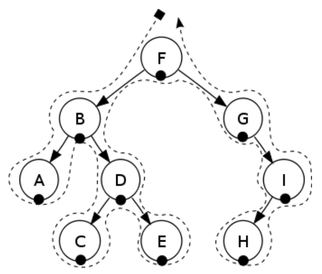
1.2 Pre-, In-, Post- Order

Consider how you may implement such a traversal. In a recursive strategy, you may realize that at any node n, you either choose to visit n first, then visit its children in some order, OR you may choose to visit its children first, then n itself.

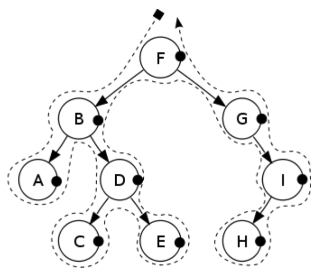
There are three main tree traversals based off this idea: pre-order, in-order, and post-order.
Here are graphics from Wikipediathat show the visit order for these three traversals:



(a) Pre-order Visit:
F, B, A, D, C, E, G, I, H



(b) In-order Visit:
A, B, C, D, E, F, G, H, I



(c) Post-order Visit:
A, C, E, D, B, H, I, G, F

It may be helpful to note that a node is visited when the line intersects with its corresponding 'dot'. Another detail you may notice is the order of recursion when 'visiting':
Pre-order: self, left-child, right-child
In-order: left-child, self, right-child
Post-order: left-child, right-child, self

2 Binary Search Trees

2.1 Introduction

A BST is a tree that satisfies the rule: for every node, its left child has a smaller value than the node, and its right child has a greater value than the node. If we have a BST of items with values attached to them, this rule becomes rather useful when we talk about searching for some item in our tree.
Here's an example: When searching for some item, you need only compare the value of the item you want with the value of the root. If the item's value is greater than the root's value, explore the right subtree. If the item's value is lesser than the root's value, explore the left subtree. You can imagine such a recursive strategy for searching in our tree.

Thus, we have $O(\log n)$ search, **if** the tree is bushy. In the worst case, the tree is not bushy and is instead spindly (imagine if every node had a right child but no left child). Then, it is essentially a linked list, and search takes $O(n)$.

2.2 Delete Operation

You can find the python code for BST deletion on Wikipedia. Here is a simplified version of what DELETE, called on some node n, does:

1. If n has no children, just remove n from the tree
2. If n has one child, replace n with its child
3. If n has two children, find either the next-smallest item before n, or the next-largest item after n. Let's call this node p. Replace the value at n with the value of p. Now, call DELETE on the node p (note the difference between the value of a node and the node itself)

*You may ask yourself: how do I find the next-smallest item (the in-order predecessor) before n, or the next-largest item (the in-order successor) after n? For the next-smallest item, you can continuously call .RIGHT() on the left subtree of n. For the next-largest item, you can continuously call .LEFT() on the right subtree of n. Draw an example to convince yourself (or examine the in-order traversal graph from the Tree Traversals section)

3 Min Heaps

A handy visualizer

3.1 Introduction

Consider the problem of implementing a priority queue. Remember the analogy of a priority queue as a kind of Emergency Room line. In an ER line, you may have several people who come in at different times; however, we are only concerned with the highest priority case at any time.

With that in mind, we want a priority queue of items with associated values. More items could be added at any time, and the priority of any item could change at any time, but we are only concerned with being able to 'pop' out the item with the highest priority at any time. There are many ways to implement this behavior, but a particularly efficient solution (and one that Java uses for its own priority queue) is to use a binary min/max heap.

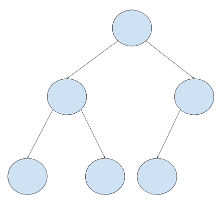
Why is this useful? Be aware that you'll actually be using a heap priority queue for two well-known algorithms, DIJKSTRA'S shortest paths algorithm and PRIM'S minimum spanning tree algorithm. There are many other uses for a priority queue (ER line perhaps?).

3.2 Invariants

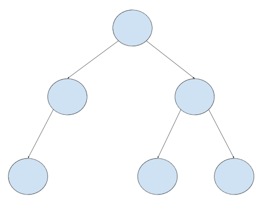
Let's focus on a min heap for now (we can imagine that a max heap will be defined similarly, just with the opposite comparisons). Items with smaller values have higher priorities. Our min heap should look like a binary tree, where the smallest value node (AKA, highest priority node) is the root of the tree. However, this is not quite like a BST! In BSTs, the left child of a node must be smaller than the node, and the right child of a node must be larger than the node. For min heaps, we just need to guarantee that any child of a node must be greater than it. Specifically, we must satisfy these two invariants:

1. The tree must be maximally balanced
    - This basically means that you must fill up the tree level by level, from left to right. You can't skip a level before its filled.

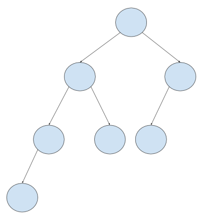
(a) A maximally balanced tree



(b) Not maximally balanced



(c) Not maximally balanced

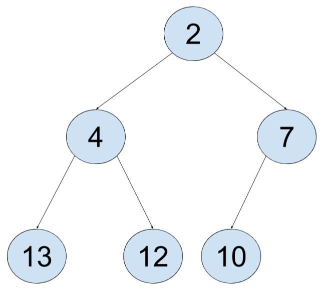


2. Every node is smaller than its descendants
    - You can see that a result of this invariant is that the min node **is** the root

3.3 Array-based Representation

Now let's get into how you INSERT a value into the heap, as well as DELETE-MIN. First, let's review two ways of representing a tree in code. The first is most intuitive, and probably the type you've all worked with: you have a NODE object with a value, and the NODE object has pointers to its left child and right child. The second representation is through an array, and it is what we'll use now, for speedy access reasons that become important for our heap operations.

Remember our level-by-level, left-to-right order of filling the tree? Well, imagine that is how we store values in our array as well! Here is an example of a valid min-heap, and its corresponding array:



0	1	2	3	4	5	6	7
–	2	4	7	13	12	10	–

The 7th index is empty since there is no value there. However, the 0th index, too, is empty, and it's always kept that way by convention. Why is that? Well, first realize that one of the reasons this array implementation is convenient is not only because we can access any node in constant time, but also because from any index in the array, we can still figure out what the parent and left/right child of that index is! For a node at index n, we can find them through these simple formulas:

- The parent of that node is at index $\frac{n}{2}$
- The left child of that node is at index $2n$
- The right child of that node is at index $2n + 1$

Thus, we keep the 0th index empty so that our formula for querying the parent/child can still be valid.

3.4 Operations

- **Insertion**
  Takes one argument: item to be added
  Goal: add item into heap

  1. Put the item you're adding in the next available spot in the bottom row (AKA, the first available entry in the array, not counting index 0)
  2. Bubble up: swap the item you added with its parent until it is not smaller than its parent.

- **Delete-Min**
  Takes no argument.
  Goal: Remove min element

  1. Replace the item at the root with the item in the last bottom spot (AKA, the last entry in the array)
  2. *Bubble down* the new root until it is smaller than both its children. If you reach a point where you can either bubble down through the left or right child, you must choose the smaller of the two.

  You can imagine that, for a POP operation, you'll DELETE-MIN, but also return the item that had previously been at the root.