1 | Review

1.1 | Binary Search

- Looking at the median of the array: if bigger, we only have to search half. If smaller, we only have to search the smaller half
- Each step, we narrow our search space down to half the remaining size
- We can find an element in it O(log n) time instead of O(n) time

1.2 | Insertion Sort

- Created sorted sub-arrays at each iteration
- Each iteration involves comparing every element and swapping every element

1.3 | "TriSort"

- Try to make MergeSort faster by dividing more into three segments
- Turns out, its not actually faster

2 | Optimizations

2.1 | Optimizing Insertion Sort

What if we used binary search as an comparative? We know that the first (pre-index pointer) subarray is already sorted. Hence, we can use binary search to drop the comparison down to just binary searching on the sorted space.

The problem… Doing this requires insertion of a value to the middle of the list—an O(n) operation that requires rightward shifting to work.

Therefore, we realized that the array data structure is kind of terrible if we are constantly shifting. Something better to do? A linked list.

2.2 | A linked List

At each point, story the value and a pointer to the next value. We can first convert our list to a link list, and then sort, so that the swaps are pretty good. But then! Actually finding the middle is pretty crap. Visa, versa

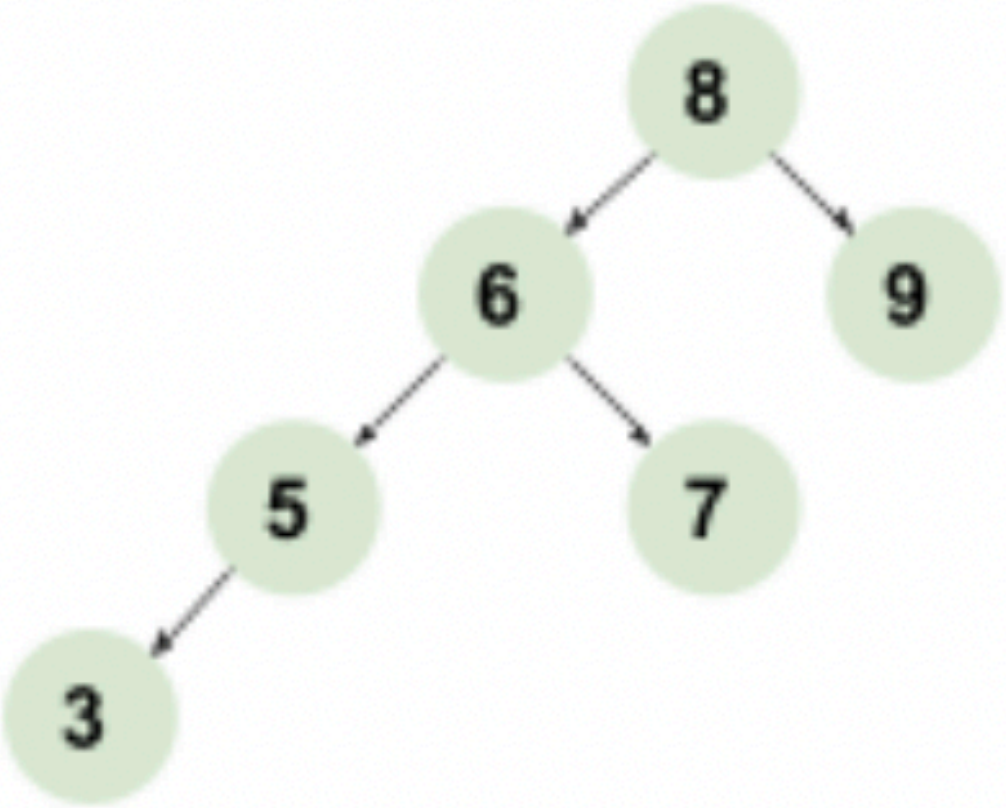
- Arrays: fast comparison (O(log n)), slow insert(O(n))
- Linked list, slow comparison (O(n)), fast insert (O(1))

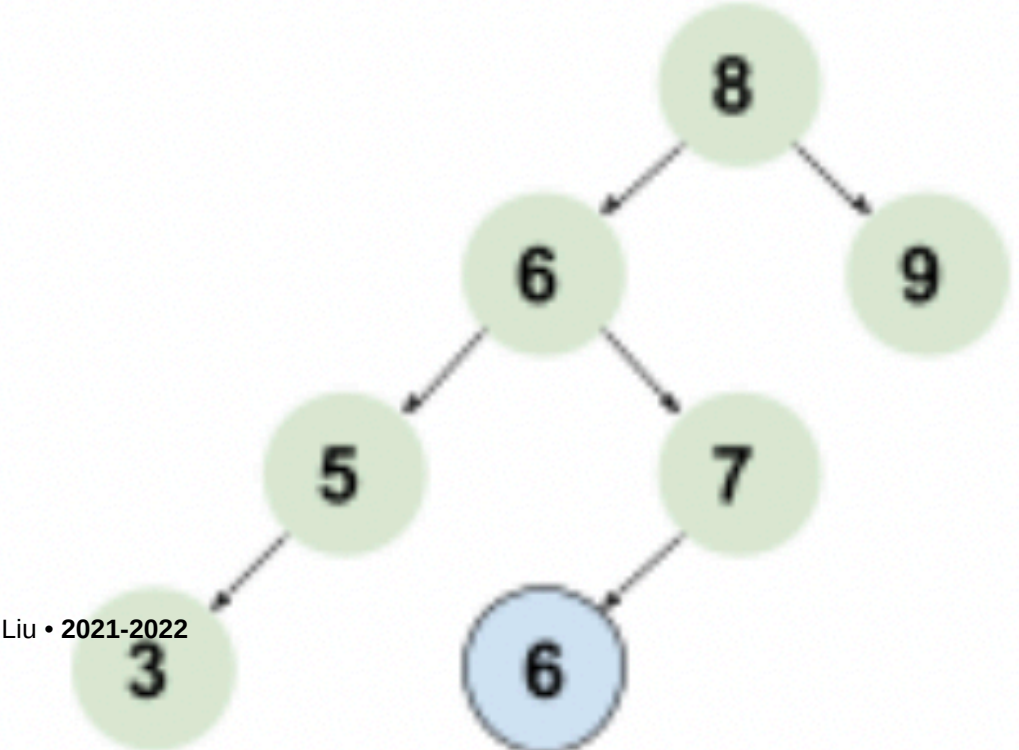
So… We need something better.

2.3 | **Binary Trees**

- Every element is held in a node
- Each node has two pointers: left and right
- All elements in the left are smaller
- All elements in the right sub-tree are equal or larger

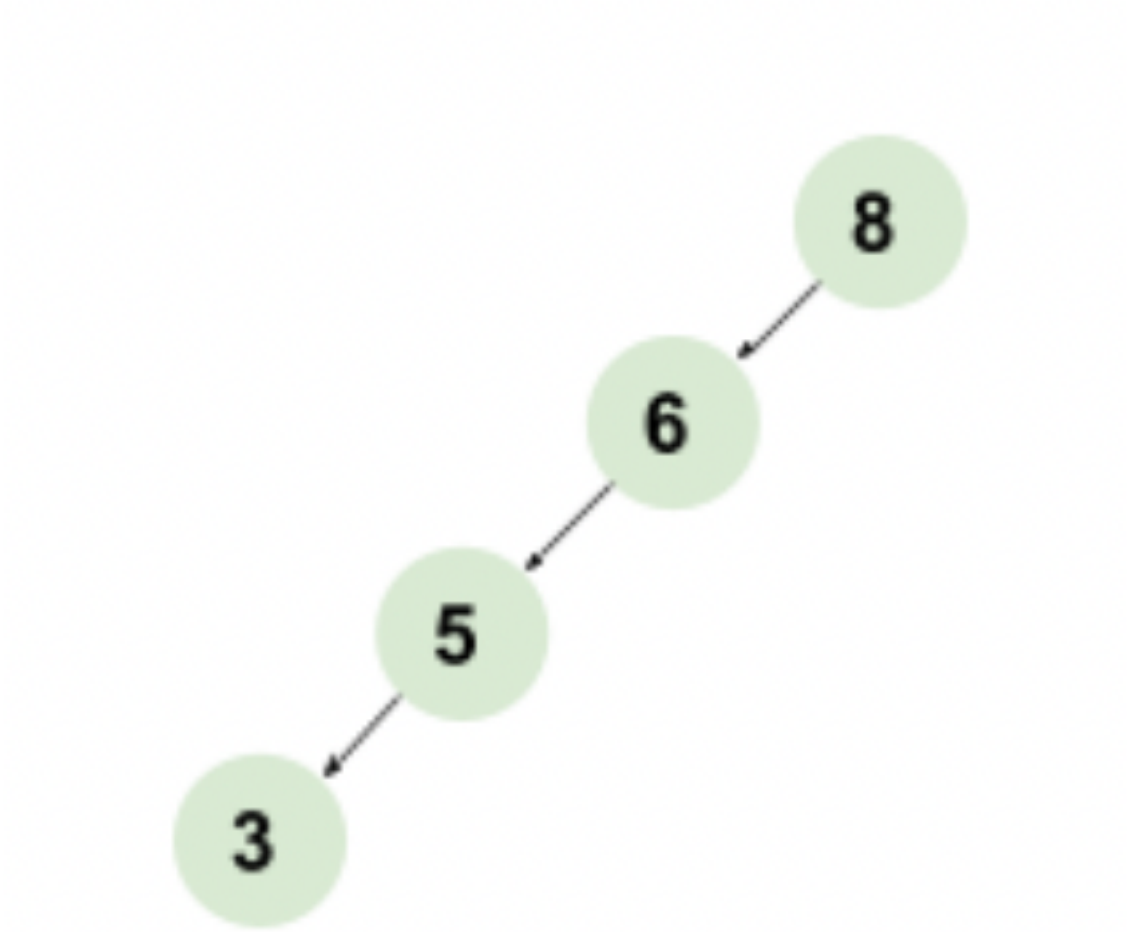
2.3.1 | **Insertions**





Compare downward continuously to look for an empty pointer; move to the left if lesser than current node, move to the right if greater or equal. Each comparison reduces the search space by half, so it takes $log(n)$ time.

But… This assumes that this tree is balanced correctly: what if you just kept adding to an edge, etc.? That would be pretty darn bad: you would just get a linked list back.



So what if…. When we insert the element, we inserted the element in a way that the tree is always balanced? We just have to think about something to do that's relative fast in each step to actually ensure the balancing of the tree.

There is actually like many many a way of creating a self-balancing tree:

- AVL tree
- BB(alpha) tree
- Red Black trees
- etc. etc.