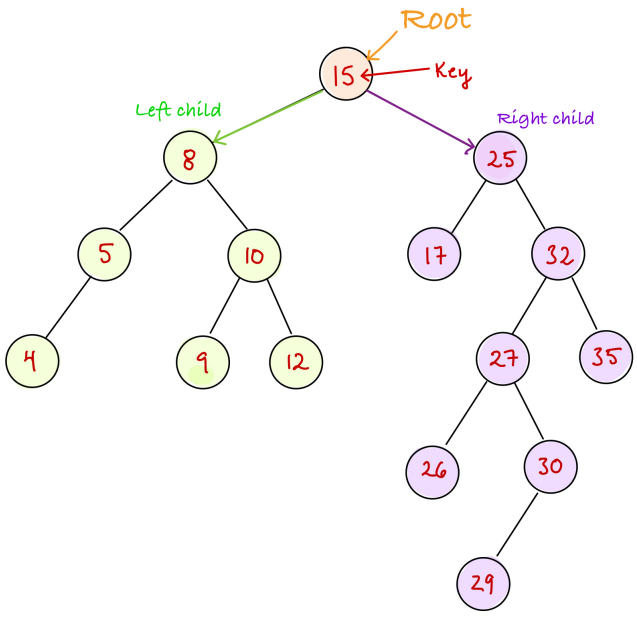
Binary Search Trees

In this lecture we define and analyze one of the most well-known structures in computer science: the *binary search tree*. This famous structure allows us to maintain a set of n elements in such a way that they are *esssentially* sorted: to output the sorted elements takes only $O(n)$ time. Furthermore, we can carry out updates and queries to this set fairly quickly. For example, we may want to insert a new element, or quickly find a maximum, or delete an element. The operations on the binary search tree are *dynamic*, meaning that each operation does not change the underlying definition of the BST. We shall see in this lecture how quickly these operations can be done in the best, worst, and expected case. In the following lecture we look at some variations of the binary search tree that have a better guaranteed worst-case time.

1 The Binary Search tree

We defined a **Binary tree** in our section on heaps. As a quick recall, the binary tree is simply a tree where each node in the tree has zero, one, or two children. A **binary search tree** is a binary tree with the following properties:

- The tree has a specific node called the **root**, which is the only node that has no parent node.
- The tree is *ordered*, meaning that the children are labelled as either the **left child** or the **right child**.
- The nodes of the tree store a *key*
- The keys in the **left** subtree of a node are all **less** than the node's key.
- The keys in the **right** subtree of a node are all **larger** than the node's key



In the example above, the nodes in light green are all smaller than the root. The nodes in light purple are all larger than the root. The node with key 8 is the *left child* of the root. The node with key 25 is the *right child* of the root. Note that this property is true for every other node in the tree.

A binary search tree can be used for any type of *total order*, which is simply a set for which we can compare any two elements. For example, natural numbers could be used as the keys of a binary search tree since we can compare the size of any two numbers with $\leq$. We could also create a binary search tree using a set of english words, where words are compared based on their alphabetical order.

1.1 Implementation

Typically binary search trees are implemented in such a way that each node of the tree is an *object*. The object may have several attributes, however those that are relevant to the structure of the BST are:

- $node.key$: the key that is used the in binary search tree property
- $node.left$: a pointer to the left child
- $node.right$: a pointer to the right child
- Optional: $node.parent$: a pointer to the parent

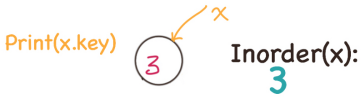
Most algorithms that we see here could be implemented without the use of a parent pointer. The decision to use one or not use one simply depends on the particular implementation.

1.2 How do we output the elements in sorted order?

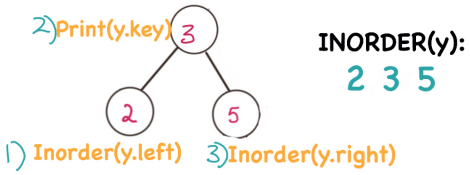
One of the most powerful aspects of a binary search tree is that it maintains the data in such a way that sorting the keys takes time $\Theta(n)$. The algorithm that carries this out is called a **inorder traversal**, which is a recursive algorithm that calls itself on the left subtree, then prints the key of the current node, and then calls itself on the right subtree. The algorithm below takes as input the root node x of a BST and outputs the keys in sorted order.

```
INORDER(x)
    If x ≠ NIL
        INORDER(x.left)
        print(x.key)
        INORDER(x.right)
```

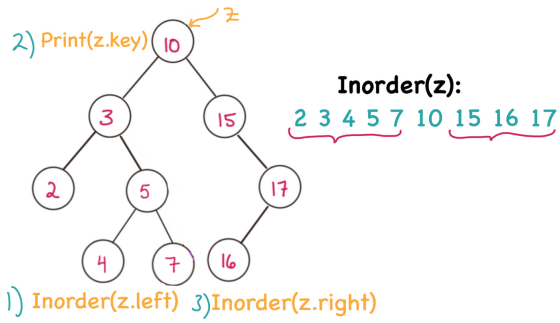
Given a single node x with no children, INORDER(x) prints the key value in x. Both the left and right children of x are NIL and thus the recursive calls to the children of x will simply terminate.



Given a node y with two children, INORDER(y) makes a call to INORDER($y.left$) which prints the key 2, then prints $y.key$, and then makes a call to INORDER($y.right$) which prints the key 5.



Given a node z with left and right subtrees, INORDER(z) makes a call to INORDER($z.left$) which will print all the keys in the left subtree in increasing order, then print $z.key$, and then will call INORDER($z.right$) which prints all the keys in the right subtree in increasing order.



Time Complexity: $\Theta(n)$

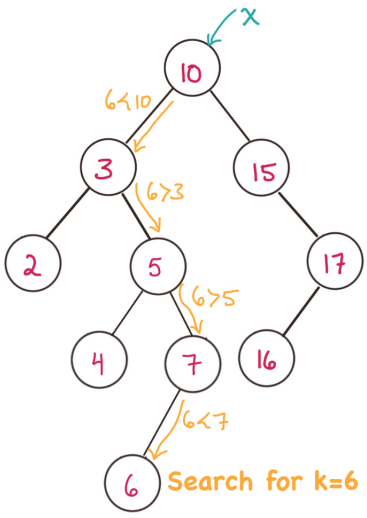
The inorder traversal is a very simple recursive algorithm, and we can use our techniques from the section on recursion to determine its runtime. If the node x is NIL, then the algorithm returns in constant time. However, if the node x is the root of a tree of size n, then the runtime consists of the time it takes to complete on the left and right subtrees, and also the constant time it takes to print the current node:

$$T(n) = T(|x.left|) + T(|x.right|) + c$$

and we can easily show by induction that this is $\Theta(n)$. This exercise is included in the problem set.

2 Searching a Binary Search Tree

To search for a particular key k in the binary search tree, we exploit the ordered nature of the children and simply trace along the path that would lead to our particular key. For each node x it encounters on the path, it compares $x.key$ to k. If $k = x.key$, then we have found the correct node, and we return x. If k is smaller than $x.key$, the search continues to the left. Otherwise the search continues to the right. An example is shown below:



The algorithm itself could be written recursively or iteratively (with a simple while loop). In both cases, the search traces down a path of the tree. Recall that the *height* of the tree is the longest path in the tree, thus the runtime of BST-search is $O(h)$, where h is the height of the tree. We shall see later what this height is, in the best, worst, and average case. In the pseudo-code below, the searching algorithm is implemented iteratively. TREE-SEARCH(x,k) takes as input a pointer x to the root of the tree, and a target key k.

```
TREE-SEARCH(x,k)
        While x ≠ NIL and k ≠ x.key
           if k < x.key
              x = x.left
           else x = x.right
        return x
```
Runtime: $O(h)$ where h is the height of the tree.

2.1 Maximums and Minimums:

Unlike heaps, the binary search tree holds the maximum (and minimum) in a unique spot in the tree.

- The **maximum** is the right-most node
- The **minimum** is the left-most node

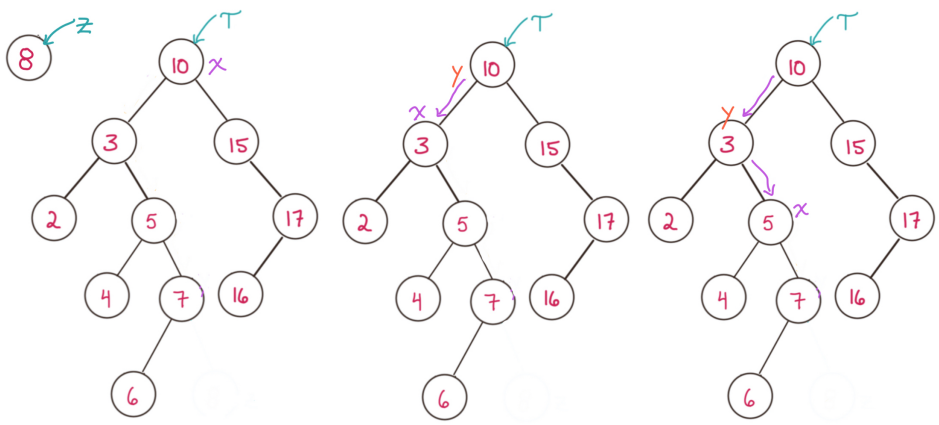
In the example above, the right-most node is a 17 and the left-most node is a 2. In order to find the right-most node for example, we would simply search along the path from the root following only the right child until there are no more right children. The last node we encounter must be the maximum. The same idea holds for finding the minimum. Again, since this procedure follows along a path in the tree, then it runs in worst-case time $O(h)$ where h is the height of the tree. We see the pseudo-code of the Max/Min finding algorithm in the practice problems for this week.

3 Inserts and Deletes in a BST

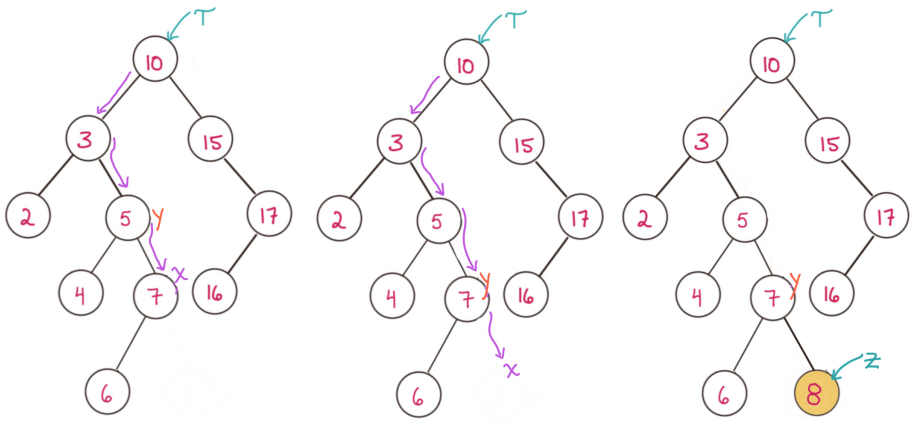
When a new key is to be inserted in the binary search tree, we must chose a spot for the key that maintains the binary search tree property. We shall show in this section that the insert operation is rather easy, and the delete operation requires only a few adjustments of the tree structure.

3.1 Insert

Suppose we wish to insert a new node z into the tree T. Assume that z holds a key in $z.key$. If the tree is empty, then we simply insert z at the root and we are done, and *return* z as the new root. However, if T is not empty, then we must *search* for the insert position of z. This is much like a regular tree search, using the key value $z.key$, except that we use the addition of a *trailing pointer*, which is a pointer that follows the path to the insertion position, but remains at the position of the parent of x. The diagram below illustrates this search down the tree until it terminates when x is NIL. The pointer x is used for the search, and the trailing pointer is y.



Once the search terminates (when x is NIL), the new node z is inserted as the child of y. It is inserted as a right child if it larger than y and otherwise as a left child:



As with the TREE-SEARCH algorithm, the worst-case runtime for an insert is $O(h)$, and therefore it depends on the height of the tree. The pseudo-code below takes as input a pointer, T to the root of the tree. If the pointer T is null, the tree is empty. Therefore the algorithm *returns* z as the new root. Otherwise, the algorithm searches for the position of insert, and returns the original pointer T which now points to a tree containing the new node z.

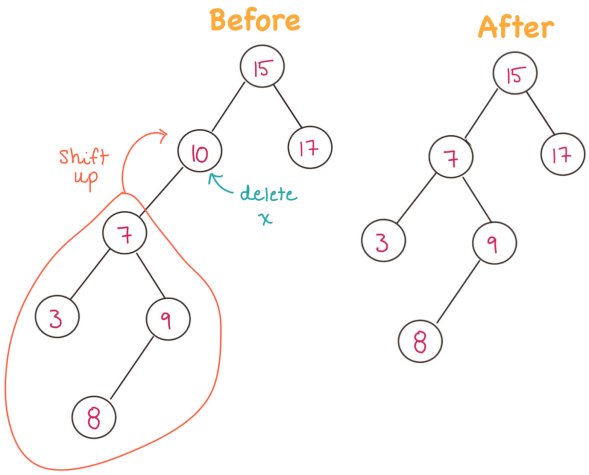
TREE-INSERT(T,z)
if $T = NIL$ return z
else $x = T$
While $x \neq NIL$
y = x
if $z.key < x.key$
x = x.left
else x = x.right
z.parent = y
if $z.key < y.key$
$y.left = z$
else $y.right = z$
return T

Runtime: $O(h)$ where h is the height of the tree.

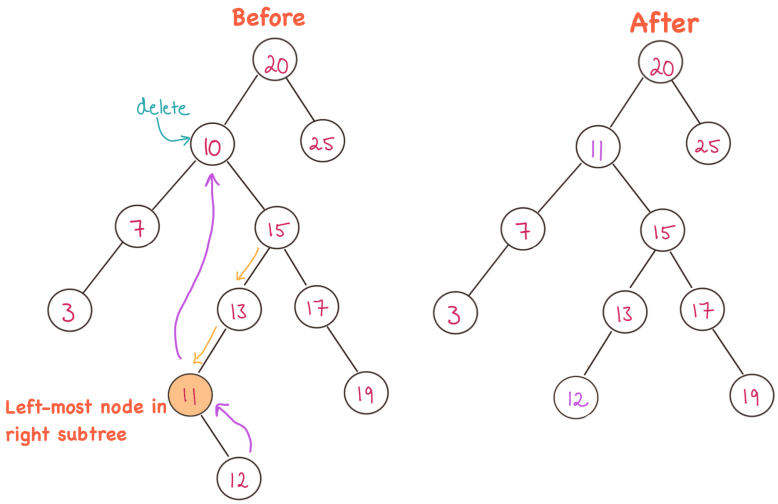
3.2 Delete

The deletion of a node x in the tree is slightly more delicate. We can't simply chop x out of the tree. Instead, the tree may need to be restructured in order to maintain the binary search tree property. There are three distinct cases that arise, the first two being quite simple to handle:

- The node x is actually a leaf. In this case, we simply remove it from the BST.
- If the node x has only one child, then we simply shift up the entire subtree belonging to that child and have it take the place of x:



- If x has two children, then we find the *successor* of x which is in the *right subtree of* x. The successor can be found easily be searching for the *left-most* node in the right subtree. The successor is removed and is used to replace x in the tree. If the successor has a right subtree, it is shifted up to take its place.

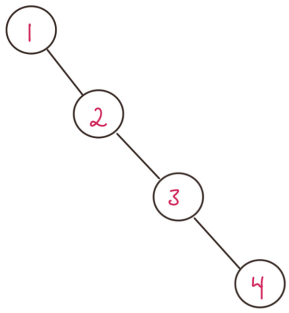


Again, this delete procedure makes at most one traversal down the tree, and thus runs in time $O(h)$. The exact pseudo-code for the deletion procedure is studied in our practice problems for the week.

4 The height of a BST

Most of the operations we have looked at so far depend on the height of the binary search tree. We saw in our section on heaps that *complete* binary trees have height $\Theta(\log n)$. Since the heaps we studied in that section were *complete* binary trees, we had a definite bound on the height of the heap, and could use that in our analysis of how long the heap operations took. Binary search trees on the other hand are not necessarily complete trees. Furthermore, the operations we did on heaps did not change the height. For binary search trees unfortunately, each time we carry out an insert or a delete, we alter the structure and possibly the height of the tree.

We begin by looking at how we could initially build a tree and what the resulting height would be. If we build a binary search tree by inserting elements one at a time into a tree, we could end up with a BST which is extremely unbalanced. For example, suppose we inserted the keys $1, 2, 3, 4$ one at a time in increasing order. The resulting tree would be:



Thus it is possible to insert keys in such a way that the tree height is $O(n)$. In this case, our operations above (Insert, Delete, etc) would run in time $O(n)$. However, if the keys were inserted in **random** order, then inserting the keys in increasing (or decreasing) order would be rather unlikely. It turns out that indeed, by inserting the keys into the tree in random order we have the following famous result:

Random BST height:
A binary search tree built by inserting n distinct keys in random order has expected height $O(\log n)$.

The formal proof of this fact is found in CLRS 12.4. We do not show it here. There are some interesting "'light" proofs that you may find here and there, many of them not technically correct, but they often give you the basic idea. However the actual proof regards levels of probability that are too involved.

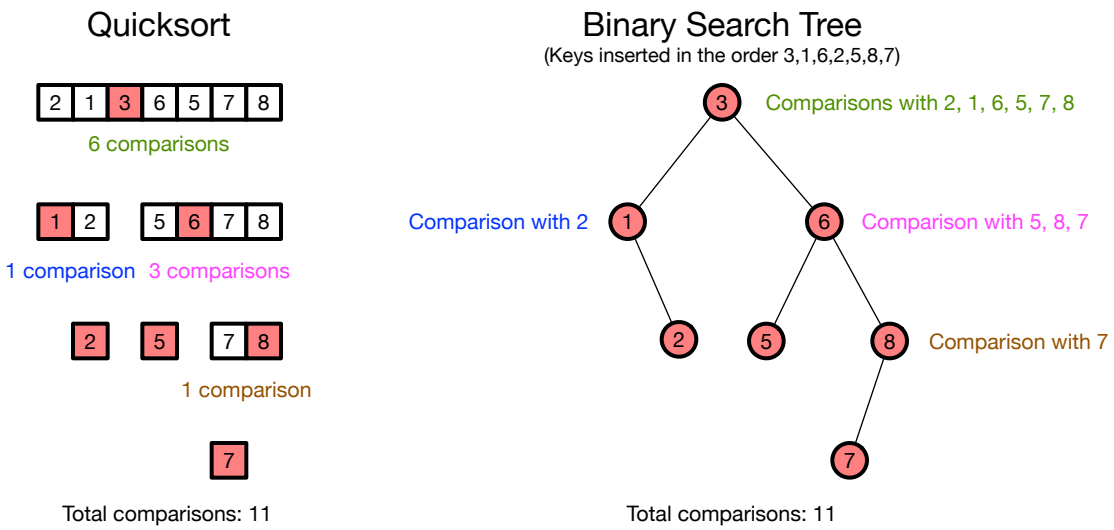
4.1 A Random BST and Quicksort

In this last section, we study the **runtime** of building a random binary search tree. Instead of developing the runtime from scratch, we use an interesting trick. We will establish the following fact:

Given: An array of size n.
Fact: The number of comparisons made by Randomzied Quicksort is the **same** as the number of comparisons made when building the Random BST, when the random pivots selected by Quicksort corresponds to the random nodes inserted in the BST.

We establish the above fact using the example below. In this example, we compare Quicksort and the construction of a binary search tree. Both algorithms will run on the same input: $3, 6, 5, 2, 7, 8, 1$, and the random pivot selected at each iteration by Quicksort will be the *same* random element inserted next into the BST. We notice something very interesting: that the number of comparisons made by Quicksort is *identical* to the number of comparisons made during the construction of the BST. In fact, each time two elements are compared during the PARTITION step of Quicksort, this corresponds exactly to the *same* comparison made during the BST construction.

On the left of the figure we see the steps of Quicksort. Each time a pivot is selected, we note the number of comparisons made during the partitioning. On the right of the figure, we construct a BST by inserting the elements in the order determined by the random pivots. Notice that this results in the exact same number comparisons being made.



Therefore if Quicksort runs in expected time $O(n \log n)$ when the pivots are randomly selected, then the construction of a binary search tree by randomly selected the input order is *also* $O(n \log n)$.

> Expected time to build a random binary search tree:
> Given a set of n distinct keys, if we insert them into an initially empty binary search tree in *random* order, the total worst-case runtime of building the tree is $O(n \log n)$.

5 Conclusion:

Thanks to the above result, we know we can build a random BST in expected time $O(n \log n)$. The resulting tree is *balanced*: its height is expected to be $O(\log n)$. This may seem like wonderful news, because we have constructed a BST that has a very efficient height, making our $O(h)$ operations run in time $O(\log n)$. However, the tree height *changes* as we perform insertions and deletions. Indeed, a randomly build BST may start out with height $O(\log n)$, but as we insert and delete, we may eventually end up back at the worst-case scenario of a tree of height $O(n)$. In the next section, we look at examples of *balanced trees*, those for which we can maintain a height of $O(\log n)$ even as we perform our dynamic operations.