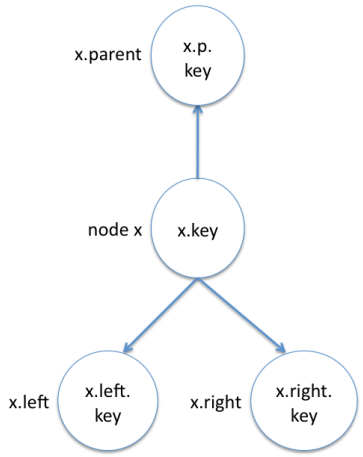
Binary Search Trees

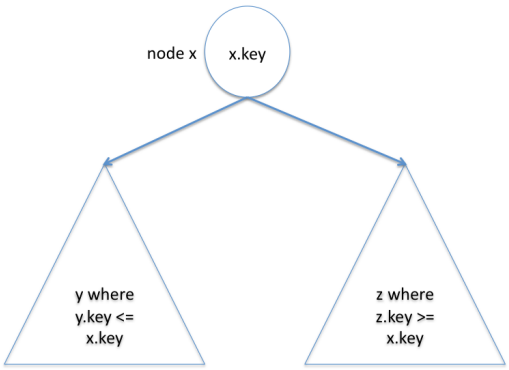
A binary search tree is a data structure that allows for key lookup, insertion, deletion, predecessor, and successor queries. It is a binary tree, meaning every node x of the tree has at most two child nodes, a left child and a right child. Each node of the tree holds the following information:

- x.key - Value stored in node x.
- x.left- Pointer to the left child of node x. NIL if x has no left child.
- x.right - Pointer to the right child of node x. NIL if x has no right child.
- x.parent - Pointer to the parent node of node x. NIL if x has no parent, i.e. x is the root of the tree.



Binary search tree has the following invariants:

- For each node x, every key found in the left subtree of x is less than or equal to the key found in x.
- For each node x, every key found in the right subtree of x is greater than or equal to the key found in x.



BST Operations

There are operations of a binary search tree that take advantage of the properties above to search for keys. There are other operations that manipulate the tree to insert new keys or remove old ones while maintaining these two invariants.

In the lecture, we saw `find(k)`, `insert(x)`, `find_min()` and `find_max()`. They all have $O(h)$ running time where h is the height of the tree. Today, we will look at two more operations. First, we will review `insert()`.

`insert()`

Description: We insert a new node x into the binary search tree by traversing the tree downwards (while following the BST invariant) until we find an empty location.

```
1  def insert(self, x):
2      """Inserts a new node x into the BST rooted at self."""
3      if x.key < self.key:
4          if self.left is not None:
5              self.left.insert(x)
6          else:
7              self.left = x
8              x.parent = self
9      else:
10         if self.right is not None:
11             self.right.insert(x)
12         else:
13             self.right = x
14             x.parent = self
```

Note that the runtime of $insert$ is upper bounded by the height of the tree.

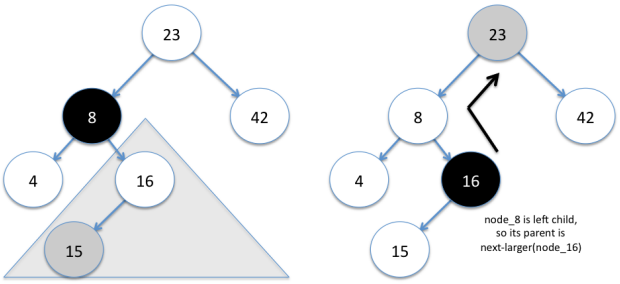
`next_larger()` and `next_smaller()`

Description: Returns the node that contains the next larger (the successor) or next smaller (the predecessor) key in the binary search tree in relation to the key at node x.

Case 1: x has a right sub-tree where all keys are larger than x.key. The next larger key will be the minimum key of x's right sub-tree.

Case 2: x has no right sub-tree. We can find the next larger key by traversing up x's ancestry until we reach a node that's a left child. That node's parent will contain the next larger key.

```
1  def next_larger(self):
2      # Case 1:
3      if self.right is not None:
4          return self.right.find_min()
5      # Case 2:
6      current = self
7      while current.parent is not None and current is current.parent.right:
8          current = current.parent
9      return current.parent
```



Case 1: next-larger(node_8) = node_15 Case 2: next-larger(node_16) = node_23

Analysis: In the worst case, `next_larger` goes through the longest branch of the tree if x is the root. Since `find_min` can take $O(h)$ time, `next_larger` could also take $O(h)$ time where h is the height of the tree.

Traversals

Sometimes we need to iterate over our stored data in sorted or reverse-sorted order. We call this performing an in-order and reverse-order traversal, respectively. We will now try implementing this traversal given a correctly constructed BST.

Let's assume that we need to call a function $visit(x)$ on each element of our BST, in sorted order. A naive way to iterate over our data would be to call the successor function n times, running $visit(x)$ on each returned node:

```
1 def naive_in_order_traversal(node, visit):
2     x = node.min()
3     while x is not None:
4         visit(x)
5         x = node.successor(x.key)
```

By naive analysis, this algorithm will take time $O(nh)$, where h is the height of the tree. In the optimal case, the height of the tree will be $O(\log n)$ and we will have a runtime of $O(n \log n)$. Of course, even using a simple sorted array we could perform this traversal in $O(n)$ time. Let's see if we can match this with BSTs:

```
1 def in_order_traversal(self, visit):
2     self.left.in_order_traversal(visit) if self.left is not None
3     visit(self)
4     self.right.in_order_traversal(visit) if self.right is not None
```

This recursive definition may seem too simple at first. Let's first examine its correctness, then analyze the runtime.

Correctness

Let's consider two nodes in the BST, x and y. We know visit will be called at some point on every node in the BST because $in\_order\_traversal$ reaches all left and right children. Suppose

$x.key < y.key$. The key to correctness is that $in\_order\_traversal$ visits all left children of a node before visiting all right children. So we have three cases: either (1) x is in the left subtree of y, (2) y is in the right subtree of x, or (3) x and y are not in each others' subtrees. The correctness of (1) and (2) follow directly from the recursion order of $in\_order\_traversal$ (visit the left subtree then self, then right subtree). For (3), by the binary search property, there must be some node z such that $x.key < z.key < y.key$, where x is in the left subtree of z and y is in the right subtree of z. Correctness now follows in (3) from the execution of $in\_order\_traversal$ on z.

Runtime

For reasoning about the runtime of $in\_order\_traversal$, we note that every node is a direct child of only one parent. Thus $in\_order\_traversal$ is called exactly once on each node. Since the amount of work done in each level of recursion is constant, we have a total runtime of $O(n)$.

In fact, by a more careful analysis we can similarly show that the first algorithm (finding minimum and then consecutively call successor) runs in $O(n)$ [too].

Tree Height

Many important operations of BSTs ($insert$, $delete$, $traversal$) take time proportional to the height of the tree. The height of a BST is dependent on the order of inputs. What would be a worst-case input order and what would be the height of the resulting BST? (strictly increasing/decreasing, height n) How about a best case order and height? (ranks [$\frac{n}{2}$, $\frac{n}{4}$ $\frac{3n}{4}$, $\frac{n}{8}$ $\frac{3n}{8}$ $\frac{5n}{8}$ $\frac{7n}{8}$, ...], height $\lg n$).

`delete()`

Description: Removes the node x from the binary search tree, making the necessary adjustments to the binary search tree to maintain its invariants. (Note that this operation removes a specified node from the tree. If you wanted to delete a key k from the tree, you would have to first call `find(k)` to find the node with key k and then call `delete` to remove that node.)

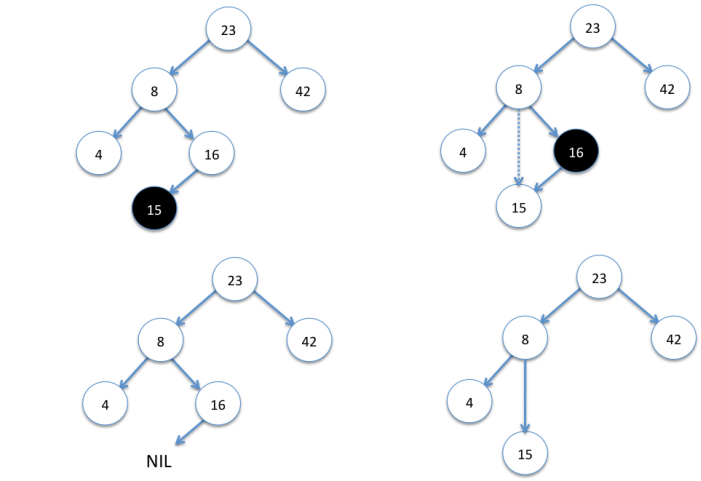
Case 1: x has no children. Just delete it (i.e. change its parent node so that it doesn't point to x).

Case 2: x has one child. Splice out x by linking x's parent to x's child.

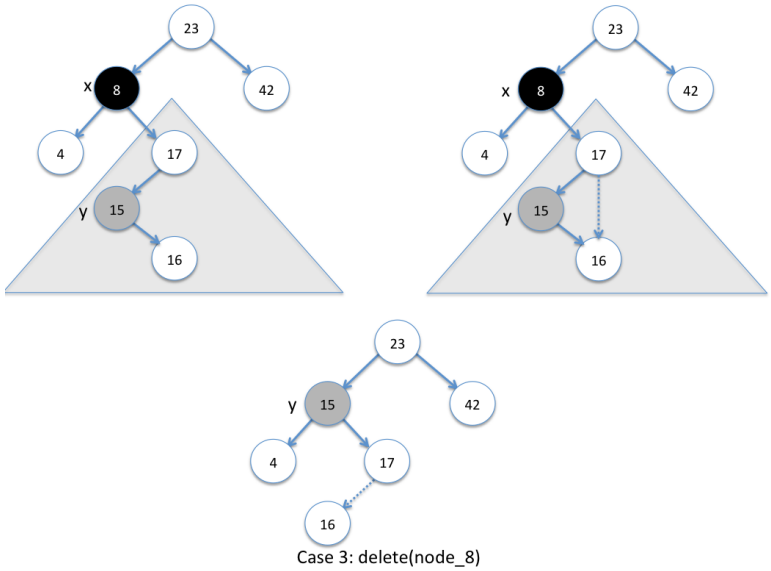
Case 3: x has two children. Splice out x's successor and replace x with x's successor.

```
1 def delete(self):
2     """Deletes and returns this node from the BST."""
3     # Case 1 & 2:
4     if self.left is None or self.right is None:
5         if self is self.parent.left:
6             self.parent.left = self.left or self.right
7             if self.parent.left is not None:
8                 self.parent.left.parent = self.parent
9         else:
```

```
            self.parent.right = self.left or self.right
            if self.parent.right is not None:
                self.parent.right.parent = self.parent
        return self
    # Case 3:
    else:
        s = self.next_larger()
        self.key, s.key = s.key, self.key
        return s.delete()
```



Case 1: delete(node_15) Case 2: delete(node_16)



Case 3: delete(node_8)

Analysis: In case 3, `delete` calls `next_larger`, which takes $O(h)$ time. At worst case, `delete` takes $O(h)$ time where h is the height of the tree.

AVL Trees

Recall the operations (e.g. `find`, `insert`, `delete`) of a binary search tree. The runtime of these operations were all $O(h)$ where h represents the height of the tree, defined as the length of the longest branch. In the worst case, all the nodes of a tree could be on the same branch. In this case, $h = n$, so the runtime of these binary search tree operations are $O(n)$. However, we can maintain a much better upper bound on the height of the tree if we make efforts to balance the tree and even out the length of all branches. An AVL trees is a binary search tree that balances itself every time an element is inserted or deleted. In addition to the invariants of a BST, **each node of an AVL tree has the invariant property that the heights of the sub-tree rooted at its children differ by at most one**, i.e.:

$$\left|\texttt{height(node.left)} - \texttt{height(node.right)}\right| \leq 1 \tag{1}$$

Height Augmentation

In AVL trees, we augment each node to keep track of the node's height.

```
1  def height(node):
2      if node is None:
3          return -1
4      else:
5          return node.height
6
7  def update_height(node):
8      node.height = max(height(node.left), height(node.right)) + 1
```

Every time we insert or delete a node, we need to update the height all the way up the ancestry until the height of a node doesn't change.

AVL Insertion, Deletion and Rebalance

We can insert a node into or delete a node from a AVL tree like we do in a BST. But after this, the height invariant (1) of the AVL tree may not be satisfied any more.

For insertion, there are 2 cases where the invariant will be violated:

1. The left child of node x is heavier than the right child. Inserting into the left child may imbalance the AVL tree.

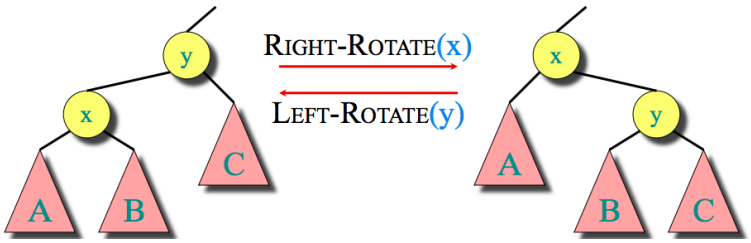
2. The right child of node x is heavier than the left child. Inserting into the right child may imbalance the AVL tree.

For deletion, the cases are analogous.

So we need to reblance the tree to maintain the invariant, starting from the node inserted or the parent of the deleted node, and continue up.

There are two operations needed to help balance an AVL tree: a left rotation and a right rotation. Rotations simply re-arrange the nodes of a tree to shift around the heights while maintaining the

order of its elements. Making a rotation requires re-assigning `left`, `right`, and `parent` of a few nodes, and **updating their heights**, but nothing more than that. Rotations are $O(1)$ time operations.




```
1  def rebalance(self, node):
2      while node is not None:
3          update_height(node)
4          if height(node.left) >= 2 + height(node.right):
5              if height(node.left.left) >= height(node.left.right):
6                  self.right_rotate(node.left)
7              else:
8                  self.left_rotate(node.left.right)
9                  self.right_rotate(node.left)
10         elif height(node.right) >= 2 + height(node.left):
11             if height(node.right.right) >= height(node.right.left):
12                 self.left_rotate(node.right)
13             else:
14                 self.right_rotate(node.right.left)
15                 self.left_rotate(node.right)
16         node = node.parent
```

Note that `rebalance` includes `update_height` as well.

Selection with AVL Trees

Suppose you have an AVL tree and you want to define the function SELECT(k) which finds and returns the k^{th} smallest element in the tree. The naive approach is to find the minimum element, and then call successor $k-1$ times. This has a worse-case running time of $O(n)$. We can do better by augmenting the AVL tree.

Augmenting with subtree size

Define the 'size' of each node as the total number of nodes in that node's subtree, including that node. If each node keeps track of its size, then you can easily determine the k^{th} element by walking down the tree in time $O(\lg n)$.

```
1  def select(self, node, k):
2
3      if node.left == None:
4        left_size = 0
5      else:
6        left_size = node.left.size
7
8      if k-1 == left_size:
9        return node
10     elif k-1 < left_size:
11       return select(node.left, k)
12     else:
13       return select(node.right, k - left_size - 1)
```

Maintaining Size

To maintain the size of each node in an AVL tree, we must slightly alter how we insert and delete from the tree. When we insert a new node, we climb down the tree starting from the root and end up stopping where we insert our new node. We can maintain size by incrementing the size of each node we see along the way. This is enough for a normal BST, but if we have an AVL tree, we must also make corrections when we call rotate. We do a similar thing when we delete a node, but with decrementing size.

AVL Sort

You can easily sort a list of n elements by creating an AVL tree, inserting each element into the AVL tree, then doing an in-order traversal of the AVL tree. Note that once you have constructed an AVL tree, the in-order traversal takes $O(n)$ time. Unfortunately, to actually construct the AVL tree takes $O(n \lg n)$ time, so using an AVL to sort a list takes $O(n \lg n)$ time, just like merge-sort and heap-sort.

It's interesting to note that the time-consuming part of AVL-sort is constructing the AVL tree, and doing a traversal to get the sorted list is quick once the tree is constructed. It is the opposite case for heaps: to construct a heap is fast, but to get the sorted list from a heap takes $O(n \lg n)$ time. It may help to think of the elements in a binary search tree as more organized than the elements in a heap: because BSTs have more restrictions on the ordering of their elements, they take longer to construct, but are also more powerful.