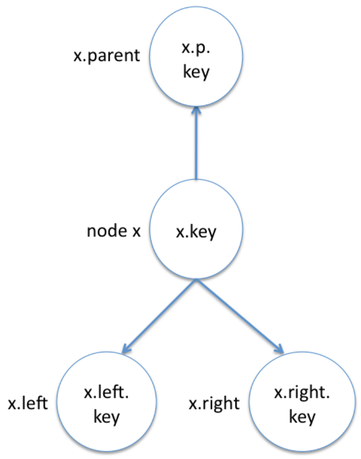
Binary Search Trees

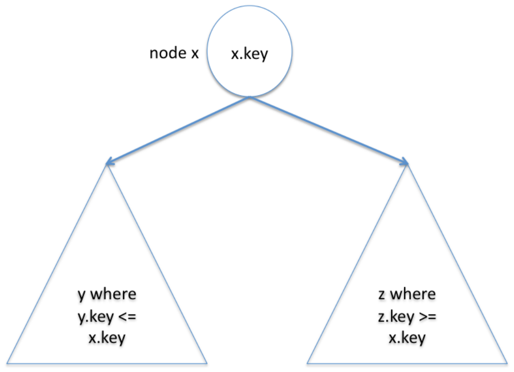
A binary search tree is a data structure that allows for key lookup, insertion, deletion, predecessor, and successor queries. It is a binary tree, meaning every node x of the tree has at most two child nodes, a left child and a right child. Each node of the tree holds the following information:

- key[x] - Value stored in node x.
- left[x]- Pointer to the left child of node x. NIL if x has no left child.
- right[x] - Pointer to the right child of node x. NIL if x has no right child.
- p[x] - 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.

```
def insert(self, x):
    """Inserts a new node x into the BST rooted at self."""
    if x.key < self.key:
        if self.left is not None:
            self.left.insert(x)
        else:
            self.left = x
            x.parent = self
    else:
        if self.right is not None:
            self.right.insert(x)
        else:
            self.right = x
            x.parent = self
```

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

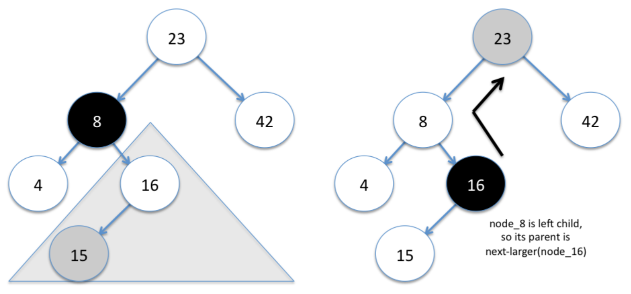
`successor()` and `predecessor()`

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 successor will be the minimum key of x's right sub-tree.

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

```
def successor(self):
    # Case 1:
    if self.right is not None:
        return self.right.find_min()
    # Case 2:
    current = self
    while current.parent is not None and current is current.parent.right:
        current = current.parent
    return current.parent
```



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

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

```
def naive_in_order_traversal(node, visit):
    x = node.min()
    while x is not None:
        visit(x)
        x = node.successor(x.key)
```

Unfortunately, this naive 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:

```
def in_order_traversal(self, visit):
    self.left.in_order_traversal(visit) if self.left is not None
    visit(self)
    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 eachothers' 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 $O(n)$.

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 $\lg n$). Next lecture we will learn how to keep a BST at height $O(\lg n)$ without increasing the asymptotic runtime.

`delete()`

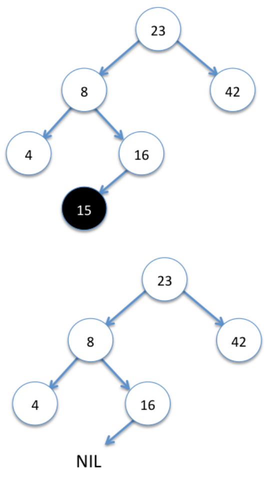
Description: Removes the node x from the binary search tree, making the necessary adjustments to the binary search tree to maintain its invariants. (Note that this operation removes a specified node from the tree. If you wanted to delete a key k from the tree, you would have to first call `find(k)` to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

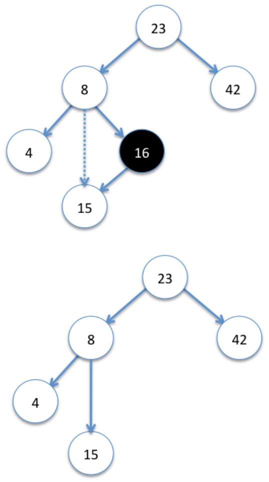
Case 2: x has one child. Splice out x by linking x's parent to x's child.

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

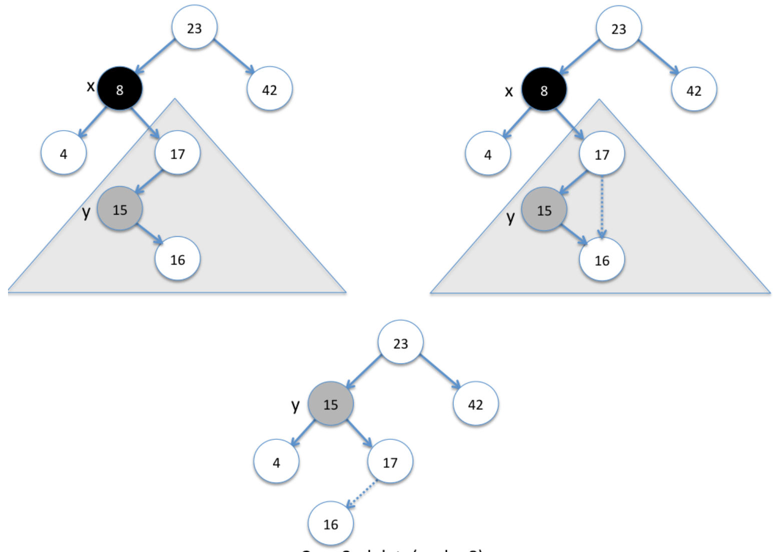
```python
def delete(self):
    """Deletes and returns this node from the BST."""
    # Case 1 & 2:
    if self.left is None or self.right is None:
        if self is self.parent.left:
            self.parent.left = self.left or self.right
            if self.parent.left is not None:
                self.parent.left.parent = self.parent
        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.successor()
        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 `successor`, which takes $O(h)$ time. At worst case, `delete` takes $O(h)$ time where h is the height of the tree.