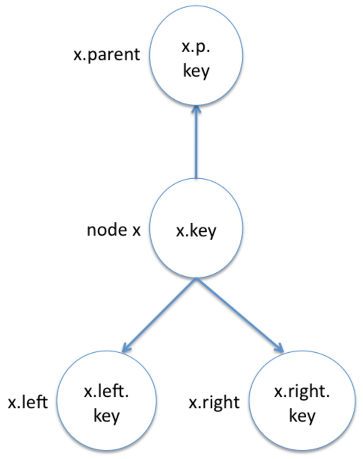
Binary Search Tree

A binary search tree is a data structure that allows for key lookup, insertion, and deletion. It is a binary tree, meaning every node 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:

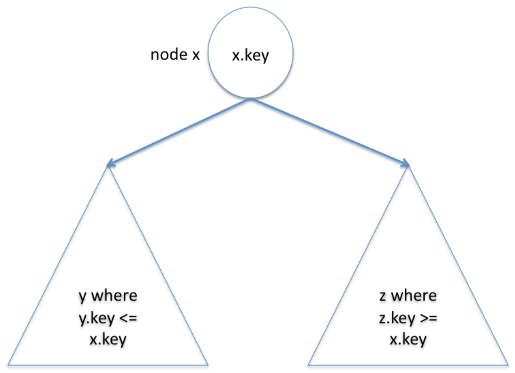
- x.key - Value stored in node x
- x.left- Pointer to the left child of node x. NIL if x has no left child
- x.right - Pointer to the right child of node x. NIL if x has no right child
- x.parent - Pointer to the parent node of node x. NIL if x has no parent, i.e. x is the root of the tree



Later on this week, we will learn about binary search trees that holds data in addition to the four listed above but for now we will focus on the vanilla binary search tree.

A binary search tree has two simple properties:

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



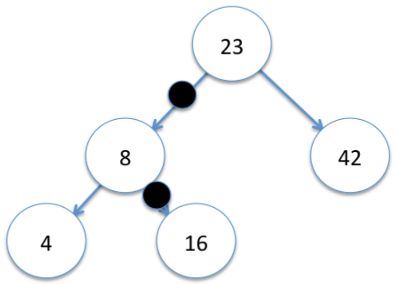
BST Operations

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 key or remove old ones while maintaining these two properties.

`find(x, k)`

Description: Find key k in a binary search tree rooted at x. Return the node that contains k if it exists or NIL if it is not in the tree

```
find(x, k)
  while x != NIL and k != x.key
    if k < x.key
      x = x.left
    else
      x = x.right
  return x
```

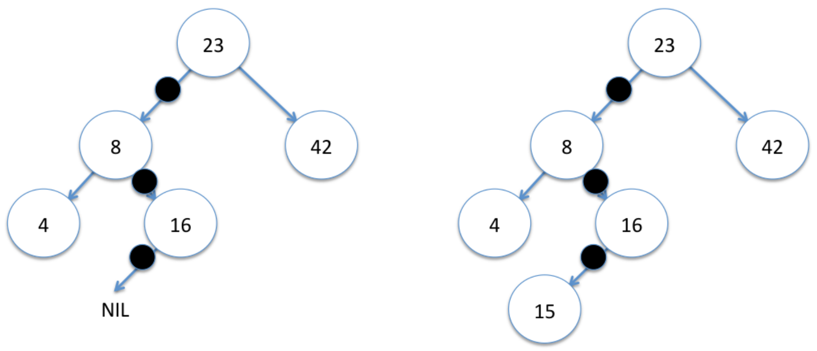


Analysis: At worst case, `find` goes down the longest branch of the tree. In this case, `find` takes $O(h)$ time where h is the height of the tree

`insert(x, k)`

Description: Insert key k into the binary search tree T

```
insert(T, k)
  z.key = k                      //z is the node to be inserted
  z.parent = NIL
  x = root(T)
  while x != NIL                 //find where to insert z
    z.parent = x
    if z.key < x.key
      x = x.left
    else
      x = x.right
  if z.parent = NIL              //in the case that T was an empty tree
    root(T) = z                  //set z to be the root
  else if z.key < z.parent.key //otherwise insert z
    z.parent.left = z
  else
    z.parent.right = z
```



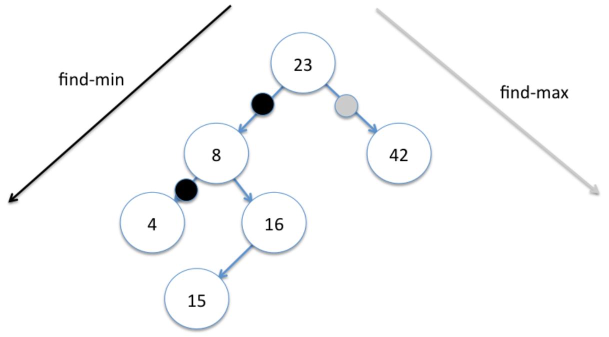
Analysis: At worst case, `insert` goes down the longest branch of the tree to find where to insert and then makes constant time operations to actually make the insertion. In this case, `insert` takes $O(h)$ time where h is the height of the tree

`find-min(x)` and `find-max(x)`

Description: Return the node with the minimum or maximum key of the binary search tree rooted at node x

```
find-min(x)
  while x.left != NIL
```

```
    x = x.left
  return x
```



Analysis: At worst case, `find-min` goes down the longest branch of the tree before finding the minimum element. In this case, `find-min` takes $O(h)$ time where h is the height of the tree

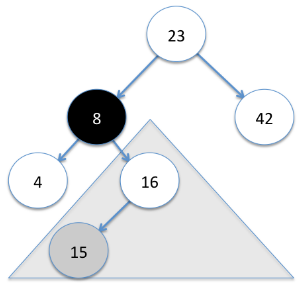
`next-larger(x)` and **`next-smaller(x)`**

Description: Return 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

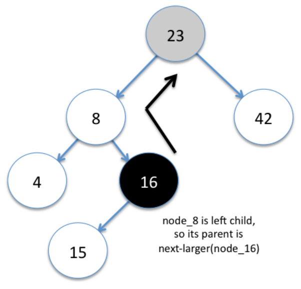
Case 1: x has a right sub-tree where all keys are larger than x.key. The next larger key will be the minimum key of x's right sub-tree

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

```
next-larger(x)
  if x.right != NIL                  //case 1
    return find-min(x.right)
  y = x.parent
  while y != NIL and x = y.right    //case 2
    x = y
    y = y.parent
  return y
```



Case 1: next-larger(node_8) = node_15



Case 2: next-larger(node_16) = node_23

Analysis: At 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

`delete(x)`

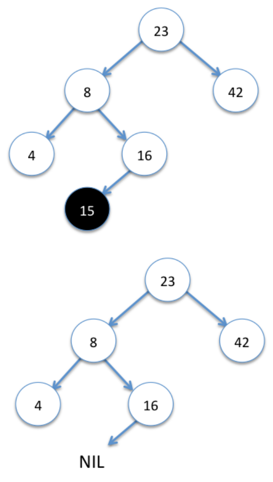
Description: Remove the node x from the binary search tree, making the necessary adjustments to the binary search tree to maintain its properties. (Note that this operation removes a specified node from the tree. If you wanted to delete a key k from the tree, you would have to first call `find(k)` to 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 parent node so that it doesn't point to x)

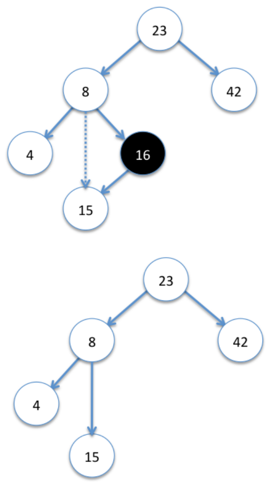
Case 2: x has one child. Splice out x by linking x's parent to x's child

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

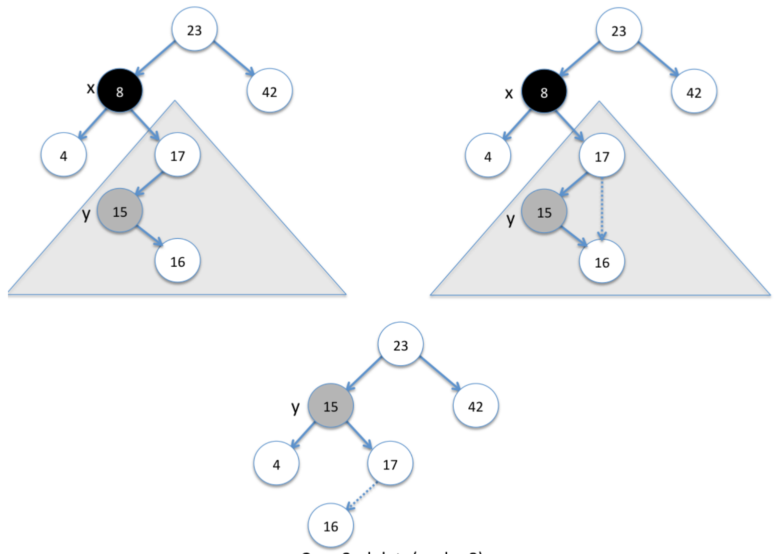
```
delete(x)
  if x.left = NIL and x.right = NIL   //case 1
    if x.parent.left = x
      x.parent.left = NIL
    else
      x.parent.right = NIL
  else if x.left = NIL                //case 2a
    connect x.parent to x.right
  else if x.right = NIL               //case 2b
    connect x.parent to x.left
  else                                //case 3
    y = next-larger(x)
    connect y.parent to y.right
    replace x with y
```



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

inorder-tree-walk(x)

Description: Print out the keys in the binary search tree rooted at node x in sorted order

```
inorder-tree-walk(x)
  if x != NIL
    inorder-tree-walk(x.left)
    print x.key
    inorder-tree-walk(x.right)
```

Analysis: `inorder-tree-walk` goes through every node and traverses to each node's left and right children. Overall, `inorder-tree-walk` prints n keys and traverses $2n$ times, resulting in $O(n)$ runtime