Sidebar: Data Structures

A **data structure** is a collection of algorithms for storing and retrieving information. The operations that store information are called **updates**, and the operations that retrieve information are called **queries**. For example, a sorted array supports the following operations:

1. Queries: MIN(), MAX(), SEARCH(x)
2. Updates: INSERT(x), DELETE(x)

The salient property of a data structure is its **representation invariant** (**RI**), which specifies how information is stored. Formally, the representation invariant is a predicate which must always be true for the data structure to function propertly. The query operations offered by the data structure are guaranteed to produce the correct result, as long as the representation invariant **holds** (is true). Update operations are guaranteed to **preserve** the representation invariant (if the RI holds before the update, it will also hold after the update).

For example, a sorted array's representation invariant is that it stores keys in an array, and the array must always be sorted. SEARCH is implemented using the binary search algorithm, which takes $O(\log(N))$ time. SEARCH is guaranteed to be correct if the RI holds (the array is sorted). On the other hand, INSERT must preserve the RI and its running time is $O(N)$. INSERT's worst-case input is a key that is smaller than all the keys in the array, as it will require shifting all the elements in the array to the right.

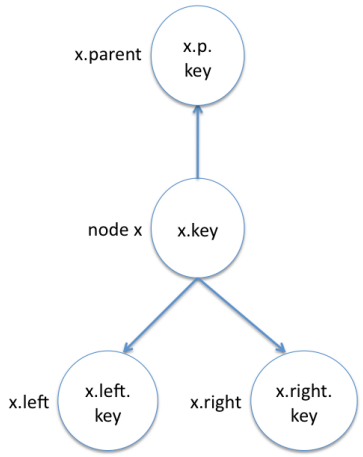
When implementing (and debugging) a data structure, it is useful to write a `check_ri` method that checks whether the representation invariant is met, and raises an exception if that is not the case. While debugging the data structure, every operation that modifies the data structure would call `check_ri` right before completing. This helps you find a bug in your implementation as soon as it happens, as opposed to having to track it down based on incorrect query results. Because it is only used during debugging, `check_ri` can be slower than the data structure's main operations. For example, checking a binary heap's representation invariant takes $O(N)$ time, whereas the usual query (MIN) and update operations (INSERT, UPDATE, EXTRACT-MIN) take $O(\log(N))$ time.

When building a software system, you should stop and think for a bit about the data structures that can perform each task efficiently. Once you have some data structures in mind, you can design the API (interface) between the module implementing the task and the rest of the system, in a way that would allow the module to be implemented using any of the efficient data structures. Once the API is in place, you should initially choose the data structure with the simplest implementation, to minimize development time. If you need to optimize your system later, you will be able to switch in a more efficient data structure easily, because you thought of that possibility in advance, when designing the module's API.

Binary Search Tree

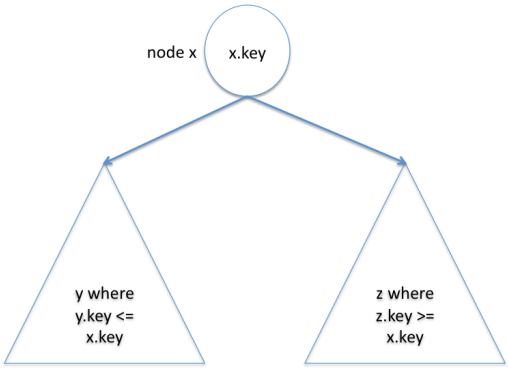
A binary search tree is a data structure that allows for key lookup, insertion, and deletion. It is a binary tree, meaning every node x of the tree has at most two child nodes, a left child and a right child. Each node of the tree holds the following information:

- 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.

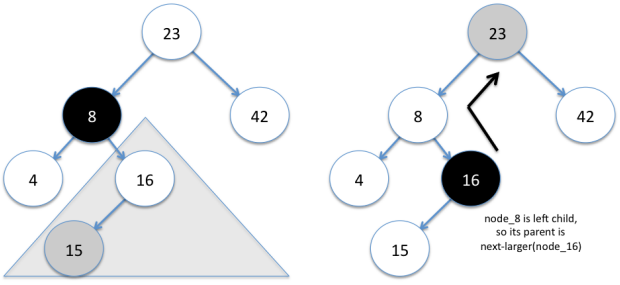
`next_larger()` and `next_smaller()`

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.

`delete()`

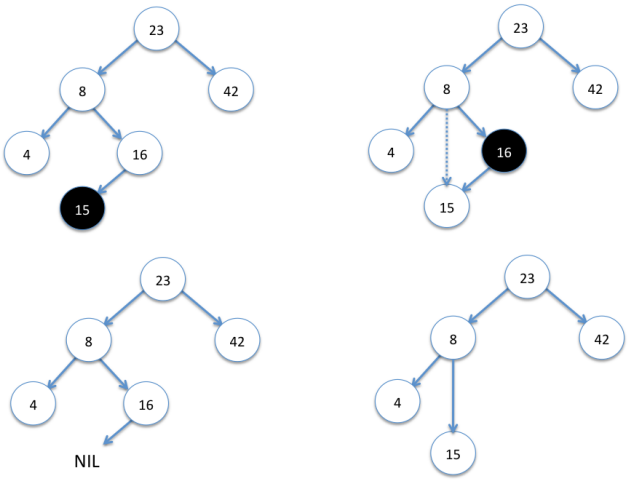
Description: Removes the node x from the binary search tree, making the necessary adjustments to the binary search tree to maintain its invariants. (Note that this operation removes a specified node from the tree. If you wanted to delete a key k from the tree, you would have to first call `find(k)` to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

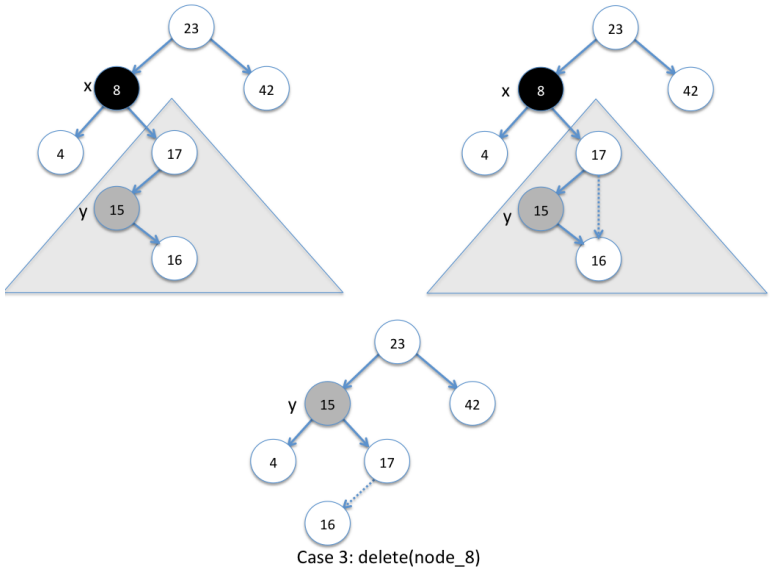
Case 2: x has one child. Splice out x by linking x's parent to x's child.

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

```
1
2  def delete(self):
3      """Deletes and returns this node from the BST."""
4      # Case 1 & 2:
5      if self.left is None or self.right is None:
6          if self is self.parent.left:
7              self.parent.left = self.left or self.right
8              if self.parent.left is not None:
9                  self.parent.left.parent = self.parent
10         else:
11             self.parent.right = self.left or self.right
12             if self.parent.right is not None:
13                 self.parent.right.parent = self.parent
14         return self
15     # Case 3:
16     else:
17         s = self.next_larger()
18         self.key, s.key = s.key, self.key
19         return s.delete()
```



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



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.

Augmented BSTs

The BST data structure can be easily augmented to implement new features without reinventing the wheel. In the lecutre, we have seen an example of augmenting BST to find the rank of a node. Here, we will look at another example.

The `find_min()` and `find_max()` operations of a vanilla BST takes $O(h)$ time. This is not as good as a heap where these operations are constant time. Can we augment BST to do better?

For every node x, we can add a field x`.min` to keep track of the node with the minimum key in the subtree rooted at x. So `find_min()` just returns `root.min` which is constant time. However, to maintain the invariant of x`.min`, we need to change `insert(x)` and `delete()` as well.

For insertion, if the key of the node to be inserted is smaller than the current node's mininum, we need to update the current node's minimum. Line 7 and 8 are the only addition and the cost is constant time. So insertion is still $O(h)$.

```
1  def insert(self, node):
2      if node is None:
3          return
4      if node.key < self.key:
5          # Updates the min of this node if the inserted node has a smaller
6          # key.
7          if node.key < self.min.key:
8              self.min = node
9          if self.left is None:
10             node.parent = self
11             self.left = node
```

```
12          else:
13              self.left.insert(node)
14      else:
15          if self.right is None:
16              node.parent = self
17              self.right = node
18          else:
19              self.right.insert(node)
```

For deletion, whenever a node's left child is changed, this node's minimum maybe changed too. This is because the minimum is always in the left subtree, and in the case that there's no left subtree, the minimum is the node itself.

Let x be the node to be deleted and x is a left child of its parent, then there are 2 cases:

Case 1: after removing x, x.parent has a new left child, then x`.parent.min = `x`.parent.left.min`.

Case 2: after remving x, x.parent has no left child, then x`.parent.min = `x`.parent`.

Are we done? No. We need to propogate this `min` update up to all the parents until we reach a node that is a right child because in this case its `min` does not affect its parent. In the worst case, the propagation takes $O(h)$ time, so deletion is still $O(h)$.

```
1  def delete(self):
2      if self.left is None or self.right is None:
3          if self is self.parent.left:
4              self.parent.left = self.left or self.right
5              # Case: 1
6              if self.parent.left is not None:
7                  self.parent.left.parent = self.parent
8                  self.parent.min = self.parent.left.min
9              # Case: 2
10             else:
11                 self.parent.min = self.parent
12             # Propagates the changes upwards.
13             current  = self.parent
14             while current.parent is not None and current is current.parent.left:
15                 current.parent.min = current.min
16                 current = current.parent
17         else:
18             self.parent.right = self.left or self.right
19             if self.parent.right is not None:
20                 self.parent.right.parent = self.parent
21         return self
22     else:
23         s = self.next_larger()
24         self.key, s.key = s.key, self.key
25         return s.delete()
```