Binary Search Tree Data Structure

- Can support search, min, max, pred, succ, insert and delete
- And we care about the ordering
- For dynamic set where keys are from totally ordered set

Overview

- Inversion and Deletion
- Querying a Binary Search Tree
- Binary Search Tree (Chapter 12)
Binary Search Tree Property:

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key ≤ x.key. If y is a node in the right subtree of x, then y.key ≥ x.key.

Binary Tree

- Uses binary tree structure of Chapter 10

Code to manually building a tree

```python
class Node:
    def __init__(self, k):
        self.key = k
        self.left = None
        self.right = None
        self.parent = None
```

Example of manually constructing a tree:

```python
top = Node(6)
top.left = Node(5); top.left.parent = top
    top.left.left = Node(2); top.left.left.parent = top.left
    top.left.right = Node(5); top.left.right.parent = top.left
    top.right = Node(7); top.right.parent = top
    top.right.right = Node(8); top.right.right.parent = top.right
```

Diagram representation of the tree.

- Parent, left child, right child, key
Here is our method:

class Node:
...
    def InOrderWalk(self):
        if self.left is not None:
            self.left.InOrderWalk()
        print self.key
        if self.right is not None:
            self.right.InOrderWalk()

Here is textbook code (function):

def InOrderWalk(node):
    if node is not None:
        InOrderWalk(node.left)
    print node.key
    InOrderWalk(node.right)

... class Node:

Here is our method:

### Inorder Tree Walk

- Can print an ordered list of keys by doing an *inorder* tree walk

- Why is the placement of the "if" different?

- *Inorder* tree walk

  - Similar to infix 5+2, prefix +(5,2) and postfix (5,2)+

- *Pre-order* or *post-order*

- Can print an ordered list of keys by doing an *inorder* tree walk
Search

Textbook Version (as a function):

```python
def TreeSearch(x, k):
    if x is None or k == x.key:
        return x
    if k < x.key:
        return TreeSearch(x.left, k)
    else:
        return TreeSearch(x.right, k)
```

As a method:

```python
def Search(self, k):
    if k == self.key:
        return self
    if k < self.key and self.left is not None:
        return self.left.Search(k)
    if k > self.key and self.right is not None:
        return self.right.Search(k)
    return None
```

Overview

• Binary Search Tree (Chapter 12)

• Querying a Binary Search Tree

• Insertion and Deletion

Binary Search Tree (Chapter 12)
Iterative Search

- Can write this as an iterative routine
- Removes overhead of subroutine calls
- But some compilers can remove tail-end recursion

```python
def IterativeSearch(self, k):
    x = self
    while x is not None and k != x.key:
        if k < x.key:
            x = x.left
        else:
            x = x.right
    return x
```

Static Methods

- Third option for specifying code
- Have a method that is not associated with an object
- But still associated with the class
- Removing method is not associated with an object

```python
class Node:
    ...
    @staticmethod
    def ClassSearch(x, k):
        if x is None or k == x.key:
            return x
        if k < x.key:
            return Node.ClassSearch(x.left, k)
        else:
            return Node.ClassSearch(x.right, k)
```

```python
Node.ClassSearch(top, 5)
```
**Succ and Pred**

- Must go through each point once; even if duplicates

```python
class Node:
    def Succ(self):
        if self.right is not None:
            return self.right.min()
        y = self.parent
        while y is not None and self == y.right:
            x = y
            y = y.parent
        return y
```

- If x.right is None
  - All nodes under x.right guaranteed to be $\leq$ anything going up the tree

- If x.right is not None
  ```python
  def Succ(self):
      return x
      x = x.right
      while x.left is not None:
          x = x.left
      return x
  ```

**Min and Max**

```python
def Min(self):
    x = self
    while x.left is not None:
        x = x.left
    return x

def Max(self):
    x = self
    while x.right is not None:
        x = x.right
    return x
```
Insertion

Insert $z$ while keeping the binary search structure

- If $a$'s right is null, add $z$ at $a$. $z$ will then come right after $a$ in an in-order walk since $z$ has no left child
- If $a$'s right is null, add $z$ at $a$'s right

Let $a, q$ be in tree, $q = \text{succ}(a)$, and $a$'s key $\leq$ $z$'s key

Turns out that we can always insert by adding it as a new leaf

Insert $z$ while keeping the binary search structure

- So we can add it to $b$'s right
- $z$'s right will be empty
- $b$ must be in the $a$'s right branch, and must be leftmost node in the branch

- If $a$'s right is not null
- If $a$'s right is null, add $z$ at $a$'s right
- $z$ will then come right after $a$ in an in-order walk since $z$ has no left child

Overview

- Binary Search Tree (Chapter 12)
- Querying a Binary Search Tree
- Binary Search Tree (Chapter 12)
class Tree:

def __init__(self):
    self.root = None

def Insert(self, z):
    y = None
    x = self.root
    while x is not None:
        y = x
        if z.key < x.key:
            x = x.left
        else:
            x = x.right
    z.parent = y
    if y is None:
        self.root = z
    elif z.key < y.key:
        y.left = z
    else:
        y.right = z

Proof of Correctness

• Have we preserved the binary search structure?
  - Similar to our search code
  - We will search for an empty node to insert into
  - Rather than search for a and b nodes
  - Case 1: we inserted at left
  - Case 2: we inserted at right
  - Regardless of which of the two inserting the next node will be in

• Better to do this on a tree (to allow inserting into an empty tree)
**Transplant** replaces subtree at u with that v.

**Must be a tree method since we might be deleting the root node**

**Code: Transplant**

```python
def Transplant(self, u, v):
    if u.parent == None:
        self.root = v
    elif u == u.parent.left:
        u.parent.left = v
    else:
        u.parent.right = v

    if v is not None:
        v.parent = u.parent
```

**Deletion**

- **Case a:**
  - b
  - No need to worry about b
  - Everything below g is either ≤ or ≥ to g
  - Change InorderTreeWalk
  - Binary tree property: make sure you don't

- **+ Simpler solution, but makes longer tree**
def Delete(self, z):
    if z.left is None:
        self.Transplant(z, z.right)
    elif z.right is None:
        self.Transplant(z, z.left)
    else:
        y = z.right.Min()
        if y.parent != z:
            self.Transplant(y, y.right)
        y.right = z.right
        y.right.parent = y
        self.Transplant(z, y)
        y.left = z.left
        y.left.parent = y