• Binary Search Tree Data Structure

- For dynamic set, where keys are from totally ordered set
- Can support search, min, max, pred, succ, insert, delete
- Binary search tree lets these operations be done fast

Overview

⇒ Binary Search Tree (Chapter 12)
- Querying a Binary Search Tree
- Insertion and Deletion
Binary Tree

• Uses binary tree structure of Chapter 10
  - parent, left child, right child, key

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

• Code to manually building 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.right = Node(7); top.right.parent = top
top.right.right = Node(8); top.right.right.parent = top.right
```

Binary Search Tree Property

**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.
Inorder Tree Walk

- Can print an ordered list of keys by doing an *inorder* tree walk
  - Versus *pre-order* or *post-order*
  - Similar to infix 5+2, prefix +(5,2) and postfix (5,2)+
  - Print left tree, print key, print right tree

class Node:

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

top.InOrderWalk()
```

Versus Textbook Code

- 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)
  - Why is the placement of the ‘if’ different?

```python
def InOrderWalk(node):
    if node is not None:
        InOrderWalk(node.left)
        print self.key
        InOrderWalk(node.right)```
Overview

• Binary Search Tree (Chapter 12)
  ⇒ Querying a Binary Search Tree
• Insertion and Deletion

Search

Textbook version (as a function):

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:

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
Static Methods

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

```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)
```

`Node.ClassSearch(top, 5)`

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
```
Min and Max

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

Succ and Pred

• Must go through each point once, even if duplicates

• Textbook:
  def Succ(self):
    x = self
    if x.right is not None:
        return x.right.min()
    y = x.parent
    while y is not None and x == y.right:
        x = y
        y = y.parent
    return y

• If x.right is not None
  - All nodes under x.right guaranteed to be ≤ anything going up the tree

• If x.right is None
  - Need to find first node above x, that we are a left ancestor of
Overview

- Binary Search Tree (Chapter 12)
- Querying a Binary Search Tree
  ⇒ Insertion and Deletion

Insertion

- Insert $z$ while keeping the binary search structure
- Turns out that we can always insert by adding it as a new leaf
- Let $a, b$ be in tree, $b = \text{succ}(a)$, and $a.\text{key} \leq z.\text{key} \leq b.\text{key}$
  - If $a.\text{right}$ is null, add $z$ at $a.\text{right}$
    + $z$ will then come right after $a$ in an intree-walk since $z$ has no left child
  - If $a.\text{right}$ is not null
    + $b$ must be in the $a.\text{right}$ branch, and must be leftmost node in the branch
    + so $b.\text{left}$ will be empty
    + add $z$ at $b.\text{left}$
    + So we can add it to $b.\text{left}$
Proof of Correctness

- Have we preserved the binary search structure?
  - Equivalent to asking if we do an inorder walk of the tree, will keys be in right order
  - Case 1: we inserted z at a.right
  - Case 2: we inserted z at b.left
- Rather than search for a and b nodes
  - We will search for an empty node to insert z into
  - Similar to our search code

Code

- Better to do this on a tree (to allow inserting into an empty tree)

```python
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
```
Deletion

- Binary tree property: make sure you don’t change InorderTreeWalk
- Everything below q is either ≥ or ≤ to q
  + No need to worry about q
- Case d:
  + Simpler solution, but makes longer tree

Code: Transplant

• Must be a tree method since we might be deleting the root node
• Transplant: replaces subtree at u with that v

```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
```
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