Overview

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

Binary Search Tree Data Structure

• For dynamic set where keys are from totally ordered set
  - And we care about the ordering
• Can support search, min, max, pred, succ, insert and delete
  - Binary search tree lets these operations be done fast
Binary Tree

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

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

• Code to manually build a tree

    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

* Do we need a Tree class?

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 \( y.key \leq x.key \). If y is a node in the right subtree of x, then \( y.key \geq x.key \).
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

```python
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()

top.InOrderWalk()
```

* Can it be used to print subtrees?

Versus Textbook Code

- Here is our method:

```python
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)

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

* Why is the placement of the ‘if’ different?
Overview

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

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
```

* Why did we shorten the name of the method?
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)

* How are static methods invoked?

Iterative Search

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

```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
```

* Why introduce a new variable as opposed to using self?
* Does self’s value change as x’s value is changing?
* What is the difference between ‘is not None’ rather than ‘!= None’?
Succ and Pred

- Must go through each point once, even if duplicates

Textbook:

```python
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

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
```
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 inter-tree-walk since \( z \) has no left child
  - If \( a.\text{right} \) is not null
    + \( b \) must be in \( a.\text{right} \) branch,
      and must be leftmost node in 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

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

class Tree:
    def __init__(self):
        self.root = None
    def Insert(self,new):
        y = None
        x = self.root
        while x is not None:
            y = x
            if new.key < x.key:
                x = x.left
            else:
                x = x.right
        new.parent = y
        if y is None:
            self.root = new
        elif new.key < y.key:
            y.left = new
        else:
            y.right = new
Deletion - Simple Cases (a-c)

- Want to remove node z
- Binary tree property: make sure you don’t change InorderTreeWalk
- Everything below q is either all \( \geq \) or \( \leq \) q
  + No need to worry about who is q’s new child

Deletion - Complex Case - d

- Move r into z’s spot
- Find minimum node under r, call it y
- Make l into y’s left child

* Will the new tree be less tall than the original tree?
A better version of Case d

- Is new tree guaranteed to be no higher than original tree?
- Might it even be shorter?

Code: Transplant

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