Recursion

Presentation Subtitle

Brad Miller David Ranum[1]

[1]Department of Computer Science
Luther College

12/19/2005

Outline

Outline

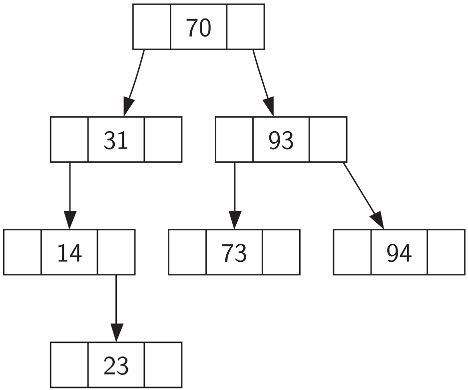
1. Binary Search Trees
   - Search Tree Operations
   - Search Tree Implementation
   - Search Tree Analysis

- `BinaryTree()` Create a new, empty binary tree.
- `put(key,val)` Add a new key-value pair to the tree.
- `get(key)` Given a key, return the value stored in the tree or `None` otherwise.
- `delete_key(key)` Delete the key-value pair from the tree.
- `length()` Return the number of key-value pairs stored in the tree.
- `has_key(key)` Return `True` if the given key is in the dictionary.
- `operators` We can use the above methods to overload the [] operators for both assignment and lookup. In addition, we can use `has_key` to override the `in` operator.

Outline

1. Binary Search Trees
   - Search Tree Operations
   - Search Tree Implementation
   - Search Tree Analysis

A Simple Binary Search Tree



The Binary Search Tree Outer Class I

```
class BinarySearchTree:
    def __init__(self):
        self.root = None
        self.size = 0

    def put(self,key,val):
        if self.root:
            self.root.put(key,val)
        else:
            self.root = TreeNode(key,val)
        self.size = self.size + 1

    def __setitem__(self,k,v):
        self.put(k,v)

    def get(self,key):
        if self.root:
```

The Binary Search Tree Outer Class II

```
            return self.root.get(key)
        else:
            return None

    def __getitem__(self,key):
        return self.get(key)

    def has_key(self,key):
        if self.root.get(key):
            return True
        else:
            return False

    def length(self):
        return self.size

    def __len__(self):
```

The Binary Search Tree Outer Class III

```
            return self.size

    def delete_key(self,key):
            if self.size > 1:
                self.root.delete_key(key)
                self.size = self.size-1
            elif self.root.key == key:
                self.root = None
                self.size = self.size - 1
            else:
                print 'error, bad key'
```

Constructor for a TreeNode

```
def __init__(self,key,val,parent=None,
                                left=None,right=None):
     self.key = key
     self.payload = val
     self.leftChild = left
     self.rightChild = right
     self.parent = parent
```

Inserting a new node

- Starting at the root of the tree, search the binary tree comparing the new key to the key in the current node. If the new key is less than the current node, search the left subtree. If the new key is greater than the current node, search the right subtree.
- When there is no left (or right) child to search, we have found the position in the tree where the new node should be installed.
- To add a node to the tree, create a new `BinarySearchTree` object and insert the object at the point discovered in the previous step.

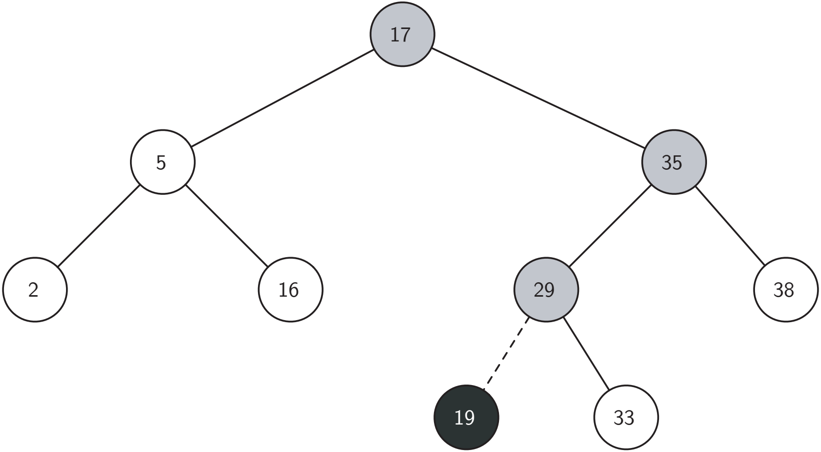
Insert a New Node in a Binary Search Tree

```
def put(self,key,val):
    if key < self.key:
        if self.leftChild:
            self.leftChild.put(key,val)
        else:
            self.leftChild = TreeNode(key,val,self)
    else:
        if self.rightChild:
            self.rightChild.put(key,val)
        else:
            self.rightChild = TreeNode(key,val,self)
```

Overloading __setitem__

```
def __setitem__(self,k,v):
    self.put(k,v)
```

Inserting a Node with Key = 19



Find the Value Stored with a Key

```
def get(self,key):
    if key == self.key:
        return self.payload
    elif key < self.key:
        if self.leftChild:
            return self.leftChild.get(key)
        else:
            return None
    elif key > self.key:
        if self.rightChild:
            return self.rightChild.get(key)
        else:
            return None
    else:
        print 'error: this line should never be executed'

def __getitem__(self,key):
    return self.get(key)
```

A Simple `length` Method I

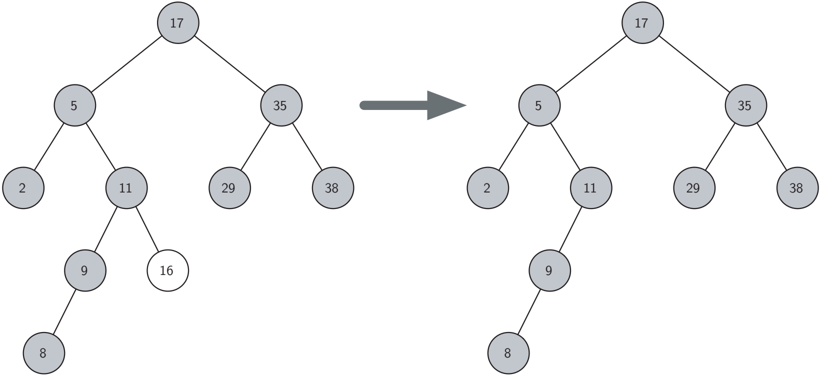
```python

    def length(self):
        return self.size

    def __len__(self):
        return self.size
```

1. The node to be deleted has no children (see Figure 13).
2. The node to be deleted has only one child (see Figure 15).
3. The node to be deleted has two children (see Figure 16).

Case 1: Deleting a Node with No Children

```
if not (self.leftChild or self.rightChild):
    print "removing a node with no childrend"
    if self == self.parent.leftChild:
        self.parent.leftChild = None
    else:
        self.parent.rightChild = None
```

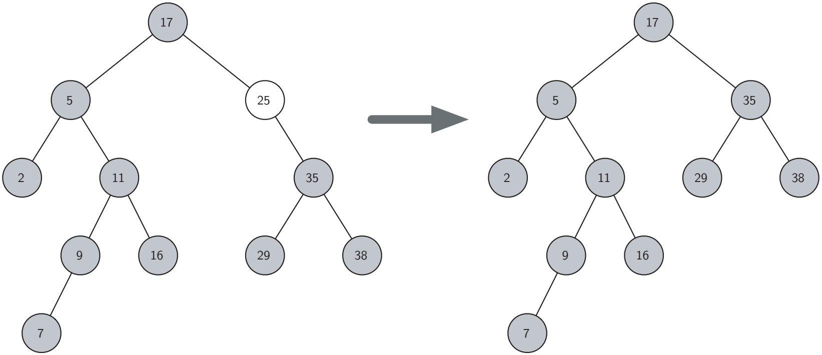
Deleting Node 16, a Node Without Children



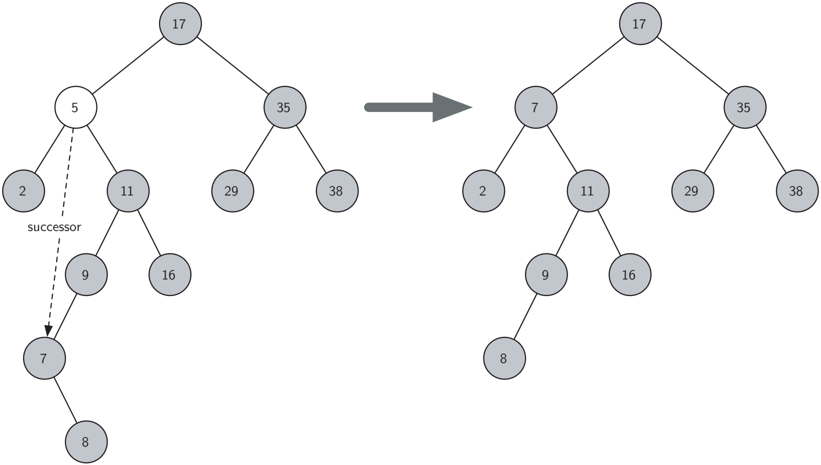
Case 2: Deleting a Node with One Child

```
elif (self.leftChild or self.rightChild) \
       and (not (self.leftChild and self.rightChild)):
    print "removing a node with one child"
    if self.leftChild:
        if self == self.parent.leftChild:
            self.parent.leftChild = self.leftChild
        else:
            self.parent.rightChild = self.leftChild
    else:
        if self == self.parent.leftChild:
            self.parent.leftChild = self.rightChild
        else:
            self.parent.rightChild = self.rightChild
```

Deleting Node 29, a Node That Has a Single Child



Deleting Node 5, a Node with Two Children



Case 3: Delete a Node with Two Children

```python
else:
    succ = self.findSuccessor()
    succ.spliceOut()
    if self == self.parent.leftChild:
        self.parent.leftChild = succ
    else:
        self.parent.rightChild = succ
    succ.leftChild = self.leftChild
    succ.rightChild = self.rightChild
```

1. If the node has a right child, then the successor is the smallest key in the right subtree.
2. If the node has no right child and is the immediate left child of its parent, then the parent is the successor.
3. If the node is the immediate right child of its parent, and itself has no right child, then the successor to this node is the successor of its parent, excluding this node.

Finding the Successor

```
def findSuccessor(self):
    succ = None
    if self.rightChild:
        succ = self.rightChild.findMin()
    else:
        if self.parent.leftChild == self:
            succ = self.parent
        else:
            self.parent.rightChild = None
            succ = self.parent.findSuccessor()
            self.parent.rightChild = self
    return succ
```

Finding the minimum child

```
    def findMin(self):
        n = self
        while n.leftChild:
            n = n.leftChild
        print 'found min, key = ', n.key
        return n
```

Helper Method to Splice Out a Node

```
def spliceOut(self):
    if (not self.leftChild and not self.rightChild):
        if self == self.parent.leftChild:
            self.parent.leftChild = None
        else:
            self.parent.rightchild = None
    elif (self.leftChild or self.rightChild):
        if self.leftChild:
            if self == self.parent.leftChild:
                self.parent.leftChild = self.leftChild
            else:
                self.parent.rightChild = self.leftChild
        else:
            if self == self.parent.leftChild:
                self.parent.leftChild = self.rightChild
            else:
                self.parent.rightChild = self.rightChild
```

Code for Deleting a Key I

```
def delete_key(self,key):
    if self.key == key:   # do the removal
        if not (self.leftChild or self.rightChild):
            if self == self.parent.leftChild:
                self.parent.leftChild = None
            else:
                self.parent.rightChild = None
        elif (self.leftChild or self.rightChild) and \
              (not (self.leftChild and self.rightChild)):
            if self.leftChild:
                if self == self.parent.leftChild:
                    self.parent.leftChild = self.leftChild
                else:
                    self.parent.rightChild = self.leftChild
            else:
                if self == self.parent.leftChild:
                    self.parent.leftChild = self.rightChild
```

Code for Deleting a Key II

```
                else:
                    self.parent.rightChild = self.rightChild
            else:  # replace self with successor
                succ = self.findSuccessor()
                succ.spliceOut()
                if self == self.parent.leftChild:
                    self.parent.leftChild = succ
                else:
                    self.parent.rightChild = succ
                succ.leftChild = self.leftChild
                succ.rightChild = self.rightChild
        else: # continue looking
            if key < self.key:
                if self.leftChild:
                    self.leftChild.delete_key(key)
                else:
            else:
```

Code for Deleting a Key III

```
        if self.rightChild:
            self.rightChild.delete_key(key)
        else:
            print "trying to remove a non-existant node"
```

An Iterator for a Binary Search Tree

```
def __iter__(self):
    if self:
        if self.leftChild:
            for elem in self.leftChild:
                yield elem
        yield self.key
        if self.rightChild:
            for elem in self.rightChild:
                yield elem
```

Outline

1. Binary Search Trees
   - Search Tree Operations
   - Search Tree Implementation
   - Search Tree Analysis

A Skewed Binary Search Tree