Outline

- The Lame Speech (Where & Why)
- Binary Search Trees
  - Principles
  - Algorithms & Python Code
- Augmenting Binary Search Trees
  - Rank computation
Orientation

• Binary Search Trees (BSTs)
  • Time/op: $O(\lg(N))$ avg, $O(N)$ max

• Balanced BSTs
  • Time/op: $O(\lg(N))$ guaranteed

• Hash Tables
  • Time/op: $O(1)$ avg, $O(N)$ worst
Motivation: Web Sites

• Many millions of DAILY visitors, billions of queries (searches)
• Run on SQL databases
• SQL indexes are mainly
  • Tree indexes
  • Hash indexes
BST Invariants

- Binary rooted tree
- All left descendants have keys < node’s key
- All right descendants have keys > node’s key
BST Conclusions

- No key shows up twice
- Each subtree contains all and only the keys with values in an interval
  - left subtree: upper bound
  - right subtree: lower bound
BST Subtree Intervals

> 3

< 8
Invariants Rock!
You can mess up a BST infinitely; as long as you maintain the invariants, it works.
Algorithms & Python

‘cause you need to know how to build this
BST Design

• BSTnode
  • attributes: key, children (left & right)
  • methods: insert, find (subtree rooted at)

• BST
  • attributes: root of the tree
  • methods: same as above
class BSTnode(object):
    def __init__(self, parent, t):
        self.key = t
        self.parent = parent
        self.left = None
        self.right = None

    def find(self, t):
        if t == self.key:
            return self
        elif t < self.key:
            if self.left is None:
                return None
            else:
                return self.left.find(t)
        else:
            if self.right is None:
                return None
            else:
                return self.right.find(t)
Insert 16
Insert 16
Insert 9
Insert 6.1
Insert 6.1
class BSTnode(object):
    def __init__(self, parent, t):
        self.key = t
        self.parent = parent
        self.left = None
        self.right = None

    def insert(self, t):
        if t < self.key:
            if self.left is None:
                self.left = BSTnode(self, t)
            else:
                self.left.insert(t)
        else:
            if self.right is None:
                self.right = BSTnode(self, t)
            else:
                self.right.insert(t)
The BST Wrapper

class BST(object):
    def __init__(self):
        self.root = None

    def insert(self, t):
        if self.root is None:
            self.root = BSTNode(None, t)
        else:
            self.root.insert(t)

    def find(self, t):
        if self.root is None:
            return None
        else:
            return self.root.find(t)
Successor of 8
Successor of 8
Successor of 10
Successor of 10
Successor of 10
Successor of 10
Successor of 7
Successor of 7

Diagram showing the successor of 7 in a tree structure.
Successor of 7
Successor of 7
Successor of 14
Successor of 14
class BSTnode(object):
    def __init__(self, parent, t):
        self.key = t
        self.parent = parent
        self.left = None
        self.right = None

    def minimum(self):
        current = self
        while current.left is not None:
            current = current.left
        return current

    def successor(self):
        if self.right is not None:
            return self.right.minimum()
        current = self
        while current.parent is not None and current.parent.right is current:
            current = current.parent
        return current.parent
Delete 4
Delete 10
Delete 10
Delete 10
Delete 3
Delete 3
Delete 3
class BSTnode(object):
    def delete(self):
        if self.left is None or self.right is None:
            if self is self.parent.left:
                self.parent.left = self.left or self.right
            if self.parent.left is not None:
                self.parent.left.parent = self.parent
            else:
                self.parent.right = self.left or self.right
            if self.parent.right is not None:
                self.parent.right.parent = self.parent
            return self
        else:
            s = self.successor()
            self.key, s.key = s.key, self.key
            return s.delete()
Delete 8
Delete 8
Delete 8

BST: Dude, where’s self.root ?!
Delete 8, Take 2
Delete 8, Take 2
Delete 8, Take 2
Delete 8, Take 2
Delete 8, Take 2

ψ
3
1
4
6
4
7
Delete 8, Take 2
Delete 8, Take 2

BST: Ah, there’s self.root!
Delete 8, Take 2

BST: Ah, there’s self.root!
Delete 8, Take 2

BST: Ah, there’s self.root!
class BST(object):
    def __init__(self):
        self.root = None

    def delete(self, t):
        node = self.find(t)
        if node is self.root:
            pseudoroot = BSTnode(None, 0)
            pseudoroot.left = self.root
            self.root.parent = pseudoroot
            deleted = self.root.delete()
            self.root = pseudoroot.left
            self.root.parent = None
            return deleted

        if node is not None:
            return node.delete()
Augmenting BSTs

‘cause you don’t wanna reinvent the wheel for every new feature
Case Study: Rank

- Want to implement a data structure with the following operations
  - given a set $S$ (initially empty)
  - $\text{insert}(x)$: add $x$ to $S$
  - $\text{delete}(x)$: remove $x$ from $S$
  - $\text{rank}(x)$: # of $y \in S$ such that $y \leq x$
Implementing Rank

• Remember that BSTs will kick ass when we learn how to balance them
• Remember that BSTs are good with order relationships
Implementing Rank

- Remember that BSTs will kick ass when we learn how to balance them
- Remember that BSTs are good with order relationships
class BSTnode(object):

    def __init__(self, parent, t):
        self.key = t
        self.parent = parent
        self.left = None
        self.right = None

    def find(self, t):
        if t == self.key:
            return self
        elif t < self.key:
            if self.left is None:
                return None
            else:
                return self.left.find(t)
        else:
            if self.right is None:
                return None
            else:
                return self.right.find(t)
```python
class BSTnode(object):
    def __init__(self, parent, t):
        self.key = t
        self.parent = parent
        self.left = None
        self.right = None
        self.size = 1

    def find(self, t):
        if t == self.key:
            return self
        elif t < self.key:
            if self.left is None:
                return None
            else:
                return self.left.find(t)
        else:
            if self.right is None:
                return None
            else:
                return self.right.find(t)
```

BST Search + Size
class BSTnode(object):
    def __init__(self, parent, t):
        self.key = t
        self.parent = parent
        self.left = None
        self.right = None

    def insert(self, t):
        if t < self.key:
            if self.left is None:
                self.left = BSTnode(self, t)
            else:
                self.left.insert(t)
        else:
            if self.right is None:
                self.right = BSTnode(self, t)
            else:
                self.right.insert(t)
class BSTnode(object):
    def __init__(self, parent, t):
        self.key = t
        self.parent = parent
        self.left = None
        self.right = None
        self.size = 1

    def insert(self, t):
        self.size += 1
        if t < self.key:
            if self.left is None:
                self.left = BSTnode(self, t)
            else:
                self.left.insert(t)
        else:
            if self.right is None:
                self.right = BSTnode(self, t)
            else:
                self.right.insert(t)
class BST(object):
    def __init__(self):
        self.root = None

    def insert(self, t):
        if self.root is None:
            self.root = BSTnode(None, t)
        else:
            self.root.insert(t)

    def find(self, t):
        if self.root is None:
            return None
        else:
            return self.root.find(t)
class BST(object):
    def __init__(self):
        self.root = None

    def insert(self, t):
        if self.root is None:
            self.root = BSTnode(None, t)
        else:
            self.root.insert(t)

    def find(self, t):
        if self.root is None:
            return None
        else:
            return self.root.find(t)
class BSTnode(object):
    def __init__(self, parent, t):
        self.key = t
        self.parent = parent
        self.left = None
        self.right = None

    def minimum(self):
        current = self
        while current.left is not None:
            current = current.left
        return current

    def successor(self):
        if self.right is not None:
            return self.right.minimum()
        current = self
        while current.parent is not None and current.parent.right is current:
            current = current.parent
        return current.parent
class BSTnode(object):
    def __init__(self, parent, t):
        self.key = t
        self.parent = parent
        self.left = None
        self.right = None
        self.size = 1

    def minimum(self):
        current = self
        while current.left is not None:
            current = current.left
        return current

    def successor(self):
        if self.right is not None:
            return self.right.minimum()
        current = self
        while current.parent is not None and current.parent.right is current:
            current = current.parent
        return current.parent
class BSTnode(object):
    def delete(self):
        if self.left is None or self.right is None:
            if self is self.parent.left:
                self.parent.left = self.left or self.right
            else:
                self.parent.right = self.left or self.right

        else:
            s = self.successor()
            self.key, s.key = s.key, self.key

        return s.delete()
class BSTnode(object):
    def delete(self):
        if self.left is None or self.right is None:
            if self is self.parent.left:
                self.parent.left = self.left or self.right
                if self.parent.left is not None:
                    self.parent.left.parent = self.parent
            else:
                self.parent.right = self.left or self.right
                if self.parent.right is not None:
                    self.parent.right.parent = self.parent
            current = self.parent
        while current is not None:
            current.size -= 1
            current = current.parent
        return self
    else:
        s = self.successor()
        self.key, s.key = s.key, self.key
        return s.delete()
(again) Deletion Hack

class BST(object):
    def __init__(self):
        self.root = None

    def delete(self, t):
        node = self.find(t)
        if node is self.root:
            pseudoroot = BSTNode(None, 0)
            pseudoroot.left = self.root
            self.root.parent = pseudoroot
            deleted = self.root.delete()
            self.root = pseudoroot.left
            self.root.parent = None
            return deleted
        if node is not None:
            return node.delete()
class BST(object):
    def __init__(self):
        self.root = None

    def delete(self, t):
        node = self.find(t)
        if node is self.root:
            pseudoroot = BSTnode(None, 0)
            pseudoroot.left = self.root
            self.root.parent = pseudoroot
            deleted = self.root.delete()
            self.root = pseudoroot.left
            self.root.parent = None
            return deleted

        if node is not None:
            return node.delete()
Rank of 7
Rank of 7

Diagram showing a tree structure with numbers: 8, 3, 1, 10, 14, 6, 7, 4, and 13. The diagram includes an annotation '+2' at the node 3.
Rank of 7
Rank of 7

Diagram of ranks:
- Rank 8 is connected to ranks 3 and 10.
- Rank 3 is connected to ranks 1 and 6.
- Rank 6 is connected to ranks 4 and 7.
- Rank 10 is connected to rank 14.
- Rank 14 is connected to rank 13.

Ranks are connected by arrows indicating their relationship.
```python
class BSTnode(object):
    def __init__(self, parent, t):
        self.key = t
        self.parent = parent
        self.left = None
        self.right = None
        self.size = 1

    def find(self, t):
        if t == self.key:
            return self
        elif t < self.key:
            if self.left is None:
                return None
            else:
                return self.left.find(t)
        else:
            if self.right is None:
                return None
            else:
                return self.right.find(t)
```
Finally, Rank!

```python
class BSTnode(object):
    def __init__(self, parent, t):
        self.key = t
        self.parent = parent
        self.left = None
        self.right = None
        self.size = 1

    def rank(self, t):
        left_size = 0 if self.left is None else self.left.size
        if t == self.key:
            return left_size + 1
        elif t < self.key:
            if self.left is None:
                return 0
            else:
                return self.left.rank(t)
        else:
            if self.right is None:
                return left_size + 1
            else:
                return self.right.rank(t) + left_size + 1
```
And we’re done!

- costan@mit.edu
- (617) 230-9694, no voicemail
- AIM: victorcostan
- Google Talk: costan@gmail.com
- 32G-8th Floor
v.Next

• Use a better name than ‘wrapper’ for BST
• Explain rank by example not by math
• This cannot be covered in 1 recitation