Algorithms and Data Structures for Data Science
Binary Search Tree

CS 277
Brad Solomon

February 28, 2024
Learning Objectives

Review understanding of Binary Trees

Practice tree traversals

Introduce the dictionary ADT

Extend ADT to Binary Search Trees

Practice recursion in the context of trees
A **binary tree** is a tree $T$ such that:

$T = \text{None}$

or

$T = \text{treeNode}(\text{val}, T_L, T_R)$
Tree ADT

**Constructor:** Build a new (empty) tree

**Insert:** Add an object into tree

**Remove:** Remove a specific object from tree

**Traverse:** Visit every node in tree (all objects)

**Search:** Find a specific object in the tree
Binary Tree Insert and Remove

Last class we implemented insert and remove where we are given the parent node and direction (allowing us to reach the node of interest)

Insert was a lot like what previous data structure:

Remove has one bad case, which was:
A **traversal** of a tree $T$ is an ordered way of visiting every node once.
Pre-order Traversal

```
def preorderTraversal(node):
    if node:
        print(node.val)
        preorderTraversal(node.left)
        preorderTraversal(node.right)
```

Pre-order:

```
+  
-  *
  /
  d e
  b c
```
Pre-order Traversal Visualized

def preorderTraversal(node):
    if node:
        print(node.val)
        preorderTraversal(node.left)
        preorderTraversal(node.right)

def preorderTraversal(node):
    if node:
        print(node.val)
        preorderTraversal(node.left)
        preorderTraversal(node.right)

def preorderTraversal(node):
    if node:
        print(node.val)
        preorderTraversal(node.left)
        preorderTraversal(node.right)

def preorderTraversal(node):
    if node:
        print(node.val)
        preorderTraversal(node.left)
        preorderTraversal(node.right)
In-order Traversal

In-order:

```
In-order: a / b * c + d e
```
Post-order Traversal

Post-order:

```
  +
 /-
|  
|   *
|    d
|     e
|     
|     b
c
```

```
a
```

```
b
```
Tree Traversals

Let's practice our traversals!

Pre-order:

In-order:

Post-order:
Traversals vs Search

Traversals

Search
Searching a Binary Tree

There are two main approaches to searching a binary tree:
Depth First Search

Explore as far along one path as possible before backtracking
Breadth First Search

Fully explore depth $i$ before exploring depth $i+1$
Traversal vs Search II

Pre-order, in-order, and post-order are three ways of doing which search?

**Pre-order:** + - a / b c * d e

**In-order:** a - b / c + d * e

**Post-order:** a b c / - d e * +
Level-Order Traversal

A tricky recursive implementation but an easier queue implementation!

Level-order:
What search algorithm is best?

The average ‘branch factor’ for a game of chess is ~31. If you were searching a decision tree for chess, which search algorithm would you use?
Improved search on a binary tree

5 3 6 7 1 4

1 3 4 5 6 7
Binary Search Tree (BST)

A **BST** is a binary tree $T = \text{treeNode}(\text{val}, T_L, T_r)$ such that:

\[ \forall n \in T_L, \ n.\text{val} < T.\text{val} \]

\[ \forall n \in T_R, \ n.\text{val} > T.\text{val} \]
Improved search on a binary tree

5 3 6 7 1 4

1 3 4 5 6 7

5

3 6

7 1 4

5

3 6

1 4 7

5

3 6

1 4 7
A BST is a binary tree \( T = \text{treeNode}(val, T_L, T_r) \) such that:

\[
\forall n \in T_L, \ n.val < T.val
\]

\[
\forall n \in T_R, \ n.val > T.val
\]
Dictionary ADT

Data is often organized into key/value pairs:

Word ➔ Definition
Course Number ➔ Lecture/Lab Schedule
Node ➔ Edges
Flight Number ➔ Arrival Information
URL ➔ HTML Page
Average Image Color ➔ File Location of Image
Dictionary as a Binary Search Tree

class bstNode:
    def __init__(self, key, val, left=None, right=None):
        self.key = key
        self.val = val
        self.left = left
        self.right = right
Binary Search Tree ADT

**Constructor:** Build a new (empty) tree

**Insert:** Add an object into tree

**Remove:** Remove a specific object from tree

**Traverse:** Visit every node in tree (all objects)

**Search:** Find a specific object in the tree
BST In-Order Traversal
BST Insert

Base Case:

Recursive Step:

Combining:
BST Insert

insert(33)
# inside class bst
    def insert(self, key, val):
        self.root = self.insert_helper(self.root, key, val)
    def insert_helper(self, node, key, val):
BST Insert

What binary would be formed by inserting the following sequence of integers: [3, 7, 2, 1, 4, 8, 0]
BST Find

Base Case:

Recursive Step:

Combining:
BST Find

find(66)
BST Find

find(9)
# inside class bst

def find(self, key):

def find_helper(self, node, key):
BST Remove

remove(40)
BST Remove

```
remove(25)
```
BST Remove

remove(13)
def remove(self, key):
    self.root = self.remove_helper(self.root, key)

def remove_helper(self, node, key):
BST Remove

What will the tree structure look like if we remove node 16 using IOS?
## BST Analysis – Running Time

<table>
<thead>
<tr>
<th>Operation</th>
<th>BST Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td></td>
</tr>
<tr>
<td>traverse</td>
<td></td>
</tr>
</tbody>
</table>
Limiting the height of a tree
Option A: Correcting bad insert order

The height of a BST depends on the order in which the data was inserted.

**Insert Order:** [1, 3, 2, 4, 5, 6, 7]

**Insert Order:** [4, 2, 3, 6, 7, 1, 5]
AVL-Tree: A self-balancing binary search tree

Rather than fixing an insertion order, just correct the tree as needed!
When would we use a tree?

Pretend for a moment that we always have an optimal BST.

What is the running time of \texttt{find}?

What is the running time of \texttt{insert}?

What is the running time of \texttt{remove}?

Is there a data structure with a \textit{better} running time for all of these?
Real World Use Case: Nearest neighbor search
Real World Use Case: Nearest neighbor search

```
[[ 45  218  0], [223  147  243], [116  57  223], [187  9  9], [238  208  236]]
[[216  190  15], [193  64  80], [184  35  215], [ 95 152 180], [128  36  41]]
[[101  128  53], [224  122  191], [237  212  74], [ 35  98  227], [214  66  167]]
[[188  3  211], [217  142  33], [210  229  167], [208  57  22], [  3 213  235]]
[[ 11  172  37], [225  191  57], [184 130  34], [136  33  51], [ 26 220  67]]
```
Real World Use Case: Nearest neighbor search
Real World Use Case: Nearest neighbor search
Real World Use Case: Nearest neighbor search

Given an input image, how can we find the closest match from a collection of other images?