Algorithms and Data Structures for Data Science

Trees

CS 277
Brad Solomon

February 26, 2024
Exam 1 next week

Multiple Choice / Fill in the blank exam

Covers content through Monday February 19th

See website for details
Learning Objectives

Build an understanding of the tree ADT

See the implementation details of a binary tree

Practice recursion in the context of trees
There are many types of trees.

KDTree

BTree

SBT

\[ a_1: 0.4 \]
\[ a_2: 0.35 \]
\[ a_3: 0.2 \]
\[ a_4: 0.05 \]
(Binary) Tree Recursion

A binary tree is a tree $T$ such that:

$T = \text{None}$

or

$T = \text{treeNode}(v, T_L, T_R)$
Visualizing trees
Which of the following are binary trees?

A

B

C

Yes!

Yes!

At most 2 children

Yes!
Tree ADT

Properties
  1. Root Node

Functions
  1. Traverse (visit all nodes in an order)
  2. Add or insert data
  3. Get node / search / lookup (across a node)
  4. Remove data
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
Recursion Practice: build_random_tree()

```python
def build_random_tree(size, seed=None):
    random.seed(seed)
    keys = list(range(size))
    random.shuffle(keys)
    root = random_tree_helper(keys)
    return root
```

Ex: build_random_tree(3, 1)

Ex: build_random_tree(3, 1001)
Recursion Practice: build_random_tree()

```python
def build_random_tree(size, seed=None):
    random.seed(seed)
    keys = list(range(size))
    random.shuffle(keys)
    root = random_tree_helper(keys)
    return root

def random_tree_helper(keyList):
    # Base Case
    if len(keyList) == 0:
        return None
    if len(keyList) == 1:
        return treeNode(keyList[0])
    # Reduction Step
    node = treeNode(keyList.pop(0))
    # Combining Step
    partition = random.randint(0, len(keyList))
    leftList = keyList[:partition]
    rightList = keyList[partition:]
    node.left = random_tree_helper(leftList)
    node.right = random_tree_helper(rightList)
    return node
```

Ex: build_random_tree(3, 1)

```
1
  2
  |
  0
```

Ex: build_random_tree(3, 1001)

```
1
  2
  |
  0
```
Binary Tree Insert

If I want to insert a value into my tree, what information do I need?

Ex: I want to insert the value ‘13’.

We need to know parent of 13 and the child "direction".

Steps:
1) Make new Node (13)
Binary Tree Insert

Different implementations will have very different insert strategies!

In our case, we need to know the following:

1. The exact insert location
   - Parent
   - Direction
2. The value we want to insert
Binary Tree Insert

**Choice:** What happens if a node already exists at our target location?

Insert ($X, 2, "Rohn")

Choice (1): Add $X$ and put current node as child

Choice (2): Delete old branch

(3): Replace value

Let's code up our choice! What is the Big O?
**Binary Tree Insert**

**Choice:** What happens if a node already exists at our target location?

\[ \text{Insert}(X, 2, 'R\text{\small{ight}}') \]

1. Make new tree node (\( O(1) \))
2. Save old branch as 'tmp' (\( O(1) \))
3. Add new tN as child (\( O(1) \))
4. Add tmp as child of tN (\( O(1) \))

Lets code up our choice! What is the Big O?
Binary Tree Insert Big O

Binary Tree insert is similar to linked list insert.

If we are given the *previous* node (here, the parent node), its $O(1)$.

But the act of *finding* a node by value is more complicated (traversal)
Binary Tree Remove

Removing a tree from a binary tree looks deceptively simple...

Ex: I want to remove the value ‘4’.

We need pointer of removed node!
Binary Tree Remove

**Choice:** How do we adjust our tree given a removed node?

If the node being removed has 0 children:

```python
2 info, Parent & direction

Parent & direction = None
```

```python
remove(5, 'left')
```

```plaintext
O(1)
```
Binary Tree Remove

When we remove, we have to be careful not to delete a tree branch!

Ex: I want to remove the value ’8’.
Binary Tree Remove

**Choice:** How do we adjust our tree given a removed node?

If the node being removed has 1 child:

\[
N_8 = \text{Node}(11) \_ \text{left has one child}
\]
\[
N_8 \_ \text{right exists!}
\]
Binary Tree Remove

When we remove, we have to be careful not to delete a tree branch!

Ex: I want to remove the value ’11’.

Choice (1): Assign order rules for every combination.

Choice (2): Swap removed value with leaf, then remove it.
Binary Tree Remove

When we remove, we have to be careful not to delete a tree branch!

Ex: I want to remove the value ’11’.

Choose (1): Assign order rules for every combination

Choose (2): Swap removed value with leaf

Then remove it.
Binary Tree Remove

**Choice:** How do we adjust our tree given a removed node?

If the node being removed has 2 children:

1) Descend and find a leaf $\in O(n)$

2) Swap value between leaf and node $\in O(1)$

3) Remove new node w/ target value $\in O(1)$ leaf; 0 child case
Binary Tree Remove Big O

What is the Big O of our removal algorithm on a binary tree?

0 child: $O(1)$  
1 child: $O(1)$  
2 child: $O(n)$

Worst case: $O(n)$
Tree Traversal

A traversal of a tree $T$ is an ordered way of visiting every node once.

At each node:
1) Look at Node
2) Recurse left
3) Recurse right

$+ - a / b c * d e$
Tree Traversal

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: $+ - / ^ ( $
In-order Traversal

In-order:
Post-order Traversal

```
+  
-  
/  
*  
*  
*  
*  
*  
*  
*  

Post-order:
```
Tree Traversals

Let's practice our traversals!

Pre-order:

In-order:

Post-order:
Traversal vs Search

Traversal

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 $i+1$
Traversals 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:

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

$\forall n \in T_R, \ n.\text{val} > T.\text{val}$