Trees

Binary Tree

A binary tree is a rooted tree in which:

- Every node has at most 2 children
- The children of a node are ordered from left to right

The depth of a binary tree is the length of the longest path from the root node to a leaf node.

BinaryTreeNode

BinaryTreeNode:

- element
- leftchild
- rightchild

Note that this is the equivalent of a singly-linked list because the arrows only go in one direction.

Computing the Height of a Node

height(node):

- = 0 if the node is a leaf
- = 1 + max(height(left), height(right))

Code

```
def height(node):
    if node = None:
        return 0
    elif node.leftchild == None and node.rightchild == None:
        return 0
    else:
        return 1 + max(height(node.leftchild), height(node.rightchild))
```

It's a recursive function.

Preorder Traversal

So called because it does the parent before the children.

To visit a node:

- Read the element then visit the children.

Code

```
def preorder_print(node):
    if node:
        print(node.element)
        preorder_print(node.leftchild)
        preorder_print(node.rightchild)

def preorder_str(node):
    if node:
        outputstr = str(node.element)
        outputstr += preorder_str(node.leftchild)
        outputstr += preorder_str(node.rightchild)
        return outputstr
    else:
        return ''
```

Inorder Traversal

(Doing the parent in between the children.)

To visit a node:

- Visit the left child, then the parent's element, then the right child.

Typically we put brackets around each subtree.

Code

```
def inorder_str(node):
    if node:
        if node.leftchild or node.rightchild:
            outstr = '(' + inorder_str(node.leftchild)
            outstr += node.element
            outstr += inorder_str(node.rightchild)
            […]
```

Post-order Traversal

Visit the children and then the parent.

Binary Search Tree

A binary search tree is a representation of an ordererd sequence of elements where:

- All left **descendants** of a node have values less than the node's value.
- All right **descendants** of a node have values greater than the node's value.

Note that the arrows do not give the order of the sequence, just the tree structure. To get the order of the elements in the sequence, do an inorder traversal.

Searching a Binary Search Tree

- Look at the top node.
    - If the number we're looking for is less than that value, go left.
    - If it's greater than that value, go right.
- Repeat until you find the node or run out of children.

If h is the height of the root, then this search is O(h) in the worst case.

Note

This is not technically binary search yet, because we don't know that the root node is the midpoint, and there could be a really big subtree in one corner.

Exercises

- How would you implement the preorder traversal without using recursion?
    - This is important because with large trees, the function calls will take up a lot of memory, which may cause trouble on limited systems.
- How would you add a new item x into a binary search tree?

Solutions

Preorder Traversal Without Recursion

We use a stack. We put the right child on the stack before the left child because you want to do it later than the left child.

```
st = Stack()
st.push(node)
while st is not empty:
    n = st.pop()
    print(n.element)
    if n.rightchild is not None:
        st.push(n.rightchild)
    if n.leftchild is not None:
        st.push(n.leftchild)
```

Manipulating Binary Search Trees

Adding a Node to a Binary Search Tree

Requirement: maintain the order properly Aim: minimise the work

When asked to add, if we allow only one copy of each element, then:

1. We check that the element is not already there
2. Unless it is, we add it

Pseudocode

```
add(node, i):
    if i < current element
        if no left child
            add i as new left child
        else
            add(node.left, i)
    else if i > current element
        if no right child
            add i as new right child
        else
            add(node.right, i)
    else
        #do nothing - already there
```

Complexity

To find the node or its natural position is O(height of the tree).

To add the node at that position is a constant number of operations.

If the tree is nicely structured, then the height of the tree will be a good bit less than the number of elements in the tree, which is why we use linked structures instead of e.g. the inbuilt Python list,

which would be O(n) if you have to add something at the start.

Removing a Node from a Binary Search Tree

Requirement: maintain the order Aim: minimise the work

We handle this by breaking it down into different cases:

- The node we want to remove is a leaf node
- The node we want to remove has only one child
- The node we want to remove has two children
- The node we want to remove has no parent (is the root node)

Since for this we need to access the parent of a node we want to remove, we now make it a doubly-linked structure, where each node links to its parent as well as to its children.

Case 1: Removing a Leaf Node

- Find the node we want to remove
- Update the parent's child reference
- Set the node's parent to None
- Remember the element
- Set the node's element to None
- Return the element

Case 2: Removing a Root and Leaf Node from a BST

Trivial case where the tree only has one node. Put a check in so that we only update the parent's child reference if the node has a parent.

Case 3: Removing a Semi-Leaf from a BST

A semi-leaf is a node with only one child.

- Change the references to that the semi-leaf's child is now the correct child of the semi-leaf's parent
- Clean up

Case 4: Removing an Internal Node from a BST

- Replace the node with the biggest element less than it.[1]
- Move that element up to our current node
- Delete the node we have copied

To find the biggest node less than a given node, we go to the left child, and then keep going right until we can't anymore.

Since we've hit a dead end, we know the new node is a leaf or a semi-leaf. We've already written the pseudocode to do that process.

Complexity

To find the node is O(height of the tree).

- Worst case is when it is a leaf on the longest branch

To remove is it O(height of the tree).

- To find the biggest item less than it, it's O(height of the node).
- To replace the node, it's a constant number of operations.

Note

Implementing node removal in a BST is tricky.

Maintaining the parent references makes it easier to keep track of where you are in the tree, and of which node needs to be updated.

But there are now more referenecs to update, and more special cases to handle.

Careful and exhaustive testing is important, and we'll probably get it wrong when we first implement it, so we must not be disheartened.

---

1. We could also replace it with the smallest element greater than it.↩