The Art of Data Structures
Trees

Alan Beadle
CSC 162: The Art of Data Structures

UNIVERSITY of ROCHESTER
Agenda

• To understand what a tree data structure is and how it is used

• To see how trees can be used to implement a map data structure

• To implement trees using a list

• To implement trees using classes and references
Trees
Trees

Properties

• Hierarchical
• Child nodes are all independent
• Path to leaf nodes are unique
• Subtrees
Trees

Example: Animal Taxonomy
Trees

Example: UNIX File System
Trees

Example: HTML Markup Elements

```
html
  head
    meta
title
  body
    ul
      li
      li
    h1
    h2
      a
```
Trees

Vocabulary

• **Node**  A node is a fundamental part of a tree. It can have a name, which we call the "key"

• **Edge**  An edge connects two nodes to show that there is a relationship between them (incoming/outgoing)

• **Root**  The root of the tree is the only node in the tree that has no incoming edges
Trees
Vocabulary

- **Path**  A path is an ordered list of nodes that are connected by edges

- **Children**  The set of nodes c that have incoming edges from the same node to are said to be the children of that node

- **Parent**  A node is the parent of all the nodes it connects to with outgoing edges
Trees
Vocabulary

- **Sibling**  Nodes in the tree that are children of the same parent are said to be siblings

- **Subtree**  A subtree is a set of nodes and edges comprised of a parent and all the descendants of that parent

- **Leaf Node**  A leaf node is a node that has no children
Trees

Vocabulary

• **Level**  The level of a node \( n \) is the number of edges on the path from the root node to \( n \)

• **Height**  The height of a tree is equal to the maximum level of any node in the tree
Trees

*Defined as Nodes and Edges*

- One node of the tree is designated as the root node
- Every node $n$, except the root node, is connected by an edge from exactly one other node $p$, where $p$ is the parent of $n$
- A unique path traverses from the root to each node
- If each node in the tree has a maximum of two children, we say that the tree is a *binary tree*
Trees

A Tree with a Set of Nodes and Edges
Trees
Defined Recursively

• A tree is either empty or consists of a root and zero or more subtrees, each of which is also a tree

• The root of each subtree is connected to the root of the parent tree by an edge
Trees

A Recursive Definition of a Tree
**Trees**

**Specification**

- `binary_tree()/BinaryTree()` creates a new instance of a binary tree using a procedural or OO method
- `get_left_child()` returns the binary tree corresponding to the left child of the current node
- `get_right_child()` returns the binary tree corresponding to the right child of the current node
Trees

Specification (cont.)

- `set_root_val(val)` stores the object in parameter `val` in the current node
- `get_root_val()` returns the object stored in the current node
- `insert_left(val)` creates a new binary tree and installs it as the left child of the current node
- `insert_right(val)` creates a new binary tree and installs it as the right child of the current node
Tree Implementation

Representing a Tree as a List of Lists
Tree Implementation

Representing a Tree as a List of Lists

mytree = [‘a’,  #root
          [‘b’,  #left subt
           [‘d’, [], []],
           [‘e’, [], []]],
          [‘c’,  #right subt
           [‘f’, [], []]],
          []]
Tree Implementation

Procedural Implementation

# This is an example of a binary tree data structure created
# with python lists as the underlying data structure.

def binary_tree(r):
    return [r, [], []]
def insert_left(root, new_branch):
    t = root.pop(1)

    if len(t) > 1:
        root.insert(1, [new_branch, t, []])
    else:
        root.insert(1, [new_branch, [], []])

    return root
def insert_right(root, new_branch):
    t = root.pop(2)

    if len(t) > 1:
        root.insert(2, [new_branch, [], t])
    else:
        root.insert(2, [new_branch, [], []])

    return root
def get_root_val(root):
    return root[0]

def set_root_val(root, new_val):
    root[0] = new_val

def get_left_child(root):
    return root[1]

def get_right_child(root):
    return root[2]
Tree Implementation

Procedural Implementation
(Usage 1)

```python
r = binary_tree(3)
insert_left(r,4)
insert_left(r,5)
insert_right(r,6)
insert_right(r,7)
l = get_left_child(r)
print(l)

set_root_val(l,9)
print(r)
print(get_right_child(get_right_child(r)))
```
Tree Implementation

Procedural Implementation

(Usage 2)

b = binary_tree('a')

# Build up the left side of this tree
insert_left(b, 'b')
insert_right(get_left_child(b), 'd')

# Build up the right side of this tree
insert_right(b, 'c')
insert_left(get_right_child(b), 'e')
insert_right(get_right_child(b), 'f')

print(b)
Tree Implementation

Using an Object-Oriented Approach
Tree Implementation
Using a Nodes and References Approach
Tree Implementation
Using a Nodes and References Approach

# This is an example of a BinaryTree data structure
# built as a class
# This example will only work if the rootObj passed into the
# class is a python primitive data type.

class BinaryTree:
    def __init__(self, root_obj):
        self.key = root_obj
        self.left_child = None
        self.right_child = None

    def insert_left(self, new_node):
        if self.left_child == None:
            self.left_child = BinaryTree(new_node)
        else:
            t = BinaryTree(new_node)
            t.left_child = self.left_child
            self.left_child = t
def insert_right(self, new_node):
    if self.right_child == None:
        self.right_child = BinaryTree(new_node)
    else:
        t = BinaryTree(new_node)
        t.right_child = self.right_child
        self.right_child = t

def get_right_child(self):
    return self.right_child

def get_left_child(self):
    return self.left_child

def set_root_val(self, obj):
    self.key = obj

def get_root_val(self):
    return self.key
r = BinaryTree('a')
print(r.get_root_val())
print(r.get_left_child())

r.insert_left('b')
print(r.get_left_child())
print(r.get_left_child().get_root_val())

r.insert_right('c')
print(r.get_right_child())
print(r.get_right_child().get_root_val())

r.get_right_child().set_root_val('hello')
print(r.get_right_child().get_root_val())
Binary Tree Applications
Binary Tree Applications

A Parse Tree for a Simple Sentence

Sentence

Noun Phrase

Proper Noun

Homer

Verb Phrase

Verb

Hit

Noun Phrase

Proper Noun

Bart
Binary Tree Applications

Parse Tree for \(((7+3)\ast(5-2))\)
Binary Tree Applications

Simplified parse tree for \(((7 + 3) \ast (5 - 2))\)
Binary Tree Applications

*Simplified parse tree for* \((7 + 3) \ast (5 - 2))\)

- How to build a parse tree from a fully parenthesized mathematical expression
- How to evaluate the expression stored in a parse tree
- How to recover the original mathematical expression from a parse tree
1. If the current token is a "(" , add a new node as the left child of the current node, and descend to the left child

2. If the current token is in the list ['+', '-', '/', '*'], set the root value of the current node to the operator represented by the current token. Add a new node as the right child of the current node and descend to the right child
Binary Tree Applications

Simplified parse tree for \(((7 + 3) \ast (5 - 2))\)

3. If the current token is a number, set the root value of the current node to the number and return to the parent.

4. If the current token is a ")\)\)\), go to the parent of the current node.
Binary Tree Applications

Tracing Parse Tree Construction

\((3 + (4\times5))\)

(a) (b) (c) (d)

(e) (f) (g) (h)
Binary Tree Applications

Tracing Parse Tree Construction

1. Create an empty tree

2. Read "(" as the first token. By rule 1, create a new node as the left child of the root. Make the current node this new child.

3. Read "3" as the next token. By rule 3, set the root value of the current node to "3" and go back up the tree to the parent.
Binary Tree Applications

Tracing Parse Tree Construction

4. Read "+" as the next token. By rule 2, set the root value of the current node to "+", and add a new node as the right child. The new right child becomes the current node.

5. Read a "(" as the next token. By rule 1, create a new node as the left child of the current node. The new left child becomes the current node.
6. Read a "4" as the next token. By rule 3, set the value of the current node to "4". Make the parent of "4" the current node.

7. Read "*" as the next token. By rule 2, set the root value of the current node to "*" and create a new right child. The new right child becomes the current node.
Binary Tree Applications

Code to Create a Parse Tree

def build_parse_tree(fpexp):
    fplist = fpexp.split()
    p_stack = Stack()
    e_tree = BinaryTree('')

    p_stack.push(e_tree)
    current_tree = e_tree

    for i in fplist:
        if i == '(':
            current_tree.insert_left('')
            p_stack.push(current_tree)
            current_tree = current_tree.get_left_child()
            elif i not in ['+', '-', '*', '/', ')']:
            current_tree.set_root_val(int(i))
            parent = p_stack.pop()
            current_tree = parent

    # Code continues on next slide...
elif i in ["+", "-", "*", "/":
    # Create right child and descend
    current_tree.set_root_val(i)
    current_tree.insert_right('')
    p_stack.push(current_tree)
    current_tree = current_tree.get_right_child()
elif i == ")":
    current_tree = p_stack.pop()
else:
    raise ValueError("invalid expression given!")

return e_tree

pt = buildParseTree("( ( 10 + 5 ) * 3 )")
def evaluate(parse_tree):
    opers = {
        '+': operator.add,
        '-': operator.sub,
        '*': operator.mul,
        '/': operator.truediv
    }

    left_c = parse_tree.get_left_child()
    right_c = parse_tree.get_right_child()

    if left_c and right_c:
        fn = opers[parse_tree.get_root_val()]
        return fn(evaluate(left_c), evaluate(right_c))
    else:
        return parse_tree.get_root_val()
Binary Tree Applications

Representing a Book As a Tree
def preorder(tree):
    if tree:
        print(tree.get_root_val())
        preorder(tree.get_left_child())
        preorder(tree.get_right_child())
Binary Tree Applications

*Postorder Traversal Algorithm I*

def postorder(tree):
    if tree:
        postorder(tree.get_left_child())
        postorder(tree.get_right_child())
        print(tree.get_root_val())
def evaluate_post(tree):
    opers = {
        '+': operator.add,
        '-': operator.sub,
        '*': operator.mul,
        '/': operator.truediv
    }
    res1 = None
    res2 = None
    if tree:
        res1 = evaluate_post(tree.get_left_child())
        res2 = evaluate_post(tree.get_right_child())
        if res1 and res2:
            fn = opers[tree.get_root_val()]
            return fn(res1, res2)
        else:
            return tree.get_root_val()
Binary Tree Applications

Inorder Traversal Algorithm I

def inorder(tree):
    if tree:
        inorder(tree.get_left_child())
        print(tree.get_root_val())
        inorder(tree.get_right_child())
def printexp(tree):
    s_val = ""
    if tree:
        s_val = '(' + printexp(tree.get_left_child())
        s_val += str(tree.get_root_val())
        s_val += printexp(tree.get_right_child()) + ')
    return s_val
Recursive Function to Evaluate a Binary Parse Tree

```python
in_string = "(( 10 + 5 ) * 3 )"
print(in_string)
pt = build_parse_tree(in_string)

print(evaluate_post(pt))
print(preorder(pt))
print(postorder(pt))
print(inorder(pt))

print(printexp(pt))
```
Questions?