Algorithms in Systems Engineering
ISE 172

Lecture 14

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - Section 6.4
- References
  - CLRS Chapter 22
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

Data Structures for Storing Trees

- As with lists, there are two primary data structures for storing a tree.
  - An array/list implementation (not very object-oriented)
  - An implementation similar to a linked list (object-oriented)
- In the list-based implementation, we store the children of each node as a list.
- In the linked list implementation, we have a "node" object (as in a linked list) that explicity stores the children and parent objects.

List of Lists Implementation

- In this implementation, the tree is stored as a recursively defined list of lists (of lists of lists of...).
- We can think of each node as a tuple of it's label and a list of its children.
- The list of children is itself a list of tuples and this is how the tree is built up.

Quick and Dirty Tree

```
>>> LABEL = 0
>>> CHILDREN = 1
>>> tree = (0, [])
>>> tree[CHILDREN].append((1, []))
>>> tree[CHILDREN].append((2, []))
>>> tree[CHILDREN][0][CHILDREN].append((3, []))
```

Note that we could do this a little more cleanly with dictionaries, but it would waste a lot of memory.

```
def traverse(tree):
    print tree[LABEL]
    for child in tree[CHILDREN}:
        traverse(child)
```

Object-oriented Implementation (Node Class)

To make the implementation a bit more object-oriented, we can add a node class with explicit pointers to parent and a list of children.

```
Class Node:
    def __init__(self, name, parent, **attrs)
        self.name = name
        self.parent = parent
        self.children = []
        for a in attr:
            self.attr[a] = attr[a]

    def get_children(self, name):
        return self.children
    def get_parent(self, name):
        return self.parent
```

Object-oriented Implementation (Tree Class)

To make the implementation a bit more object-oriented, we can add a node class with explicit pointers to parent and a list of children.

```
class Tree:
    def __init__(self, root)
        self.root = root
    def add_node(self, key, data, parent):
        parent.get_children.append(Node(key, data, parent))
    def __contains__(self, key)
        if key == self.root:
            return True
        for child in self.get_children(self.rooot):
            if key in Tree(child):
                return True
```

Object-oriented Implementation (Iterator)

To make the implementation a bit more object-oriented, we can add a node class with explicit pointers to parent and a list of children.

```
class Tree:
    def __iter__(self)
        return self.forward()

    def forward(self):
        yield self.root
        for child in self.get_children(current):
            Tree(child).forward()
```

Data Structures for Storing Binary Trees

- For binary trees, we can be a little bit smarter.
- Since we know that there will be at most two children, we can store the whole tree in a single array.
  - The root is stored in position 0.
  - The children of the node in position i are stored in positions $2i + 1$ and $2i + 2$.
  - This determines a unique storage location for every node in the tree and makes it easy to find a node's parent and children.
  - Using an array, the basic operations can be performed very efficiently.
- If the tree is unbalanced or dynamic, this may waste a lot of memory.
- With Python lists, since we can't insert into specific slots past the end of the list, we may also have to do a lot of excess initialization.