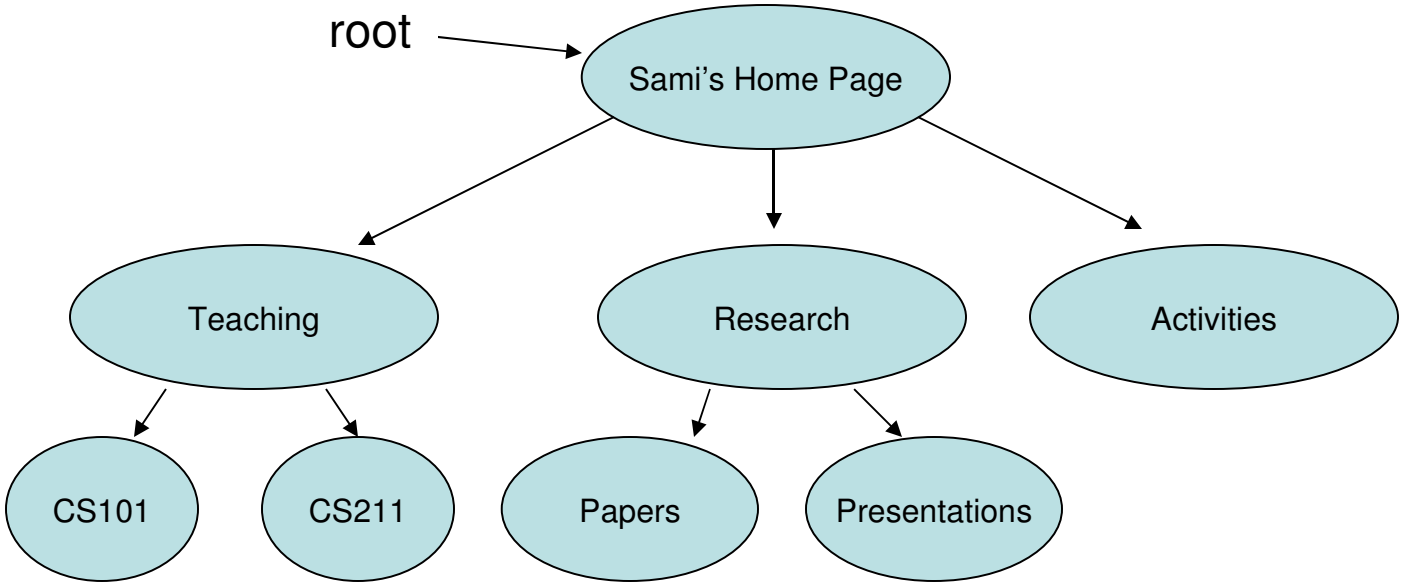
Trees

Why a tree?

- Faster than linear data structures
- More natural fit for some kinds of data
  - Examples?

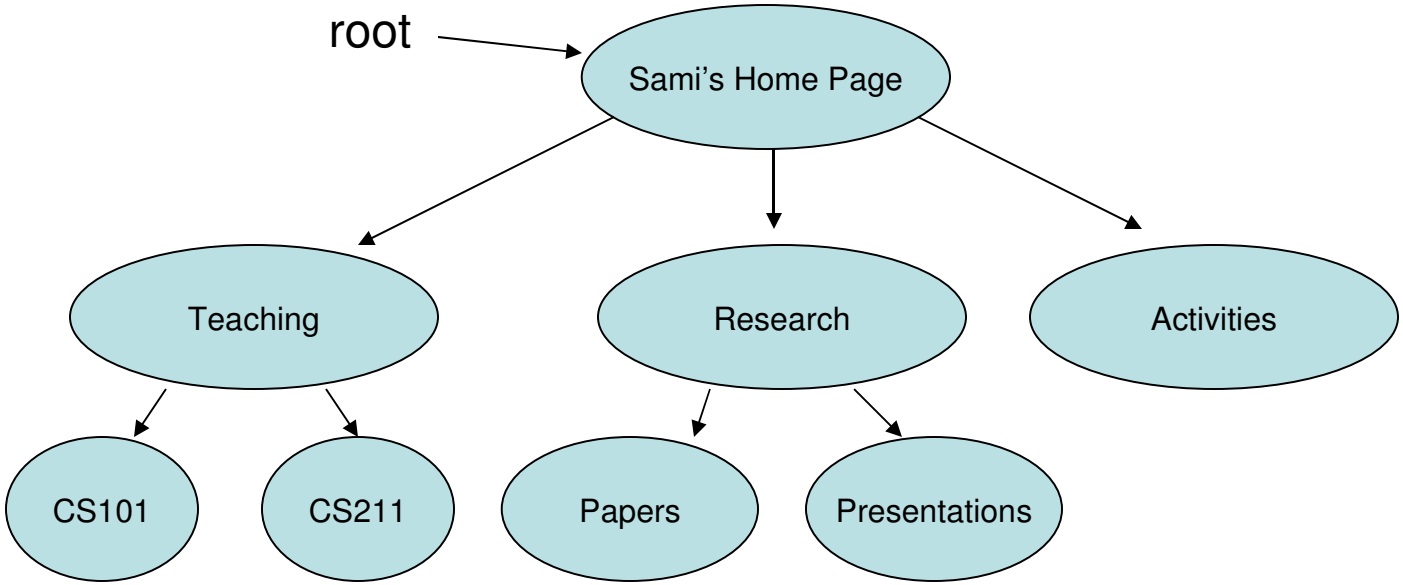
Example Tree



Terminology

- Root
- Parent
- Child
- Sibling
- External node
- Internal node
- Subtree
- Ancestor
- Descendant

Example Tree

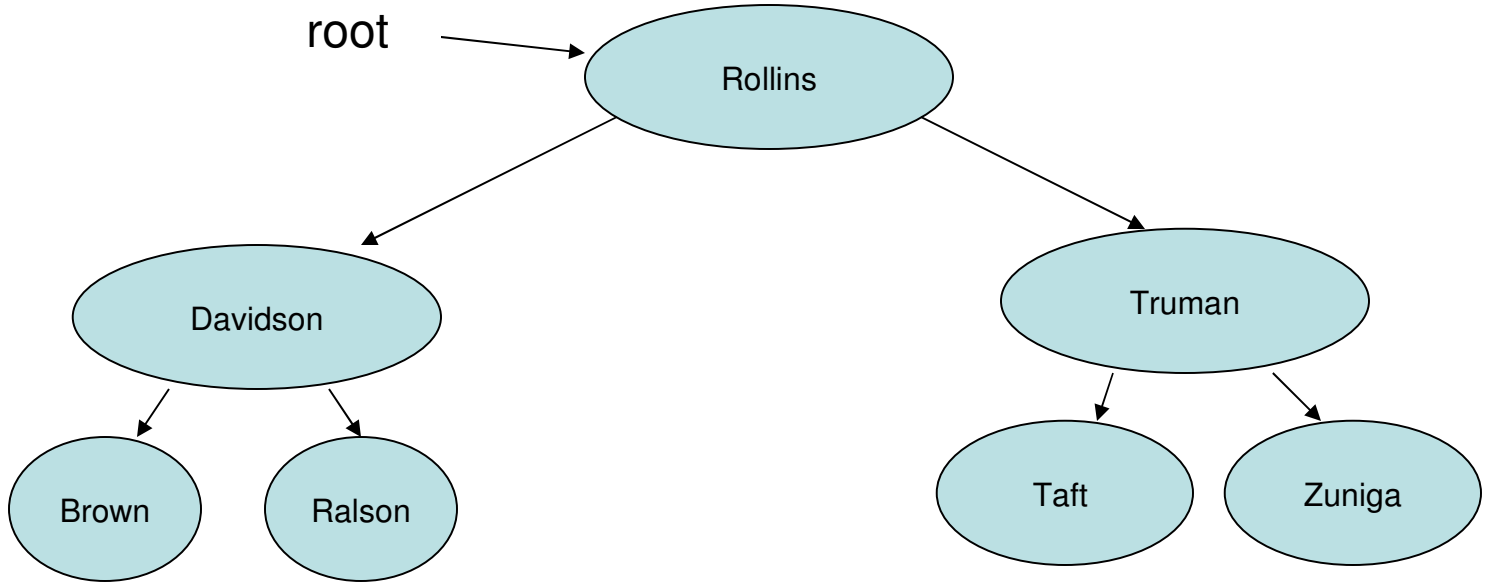


Root?
Parent – papers, activities
Children – cs101, research
Sibling - teaching

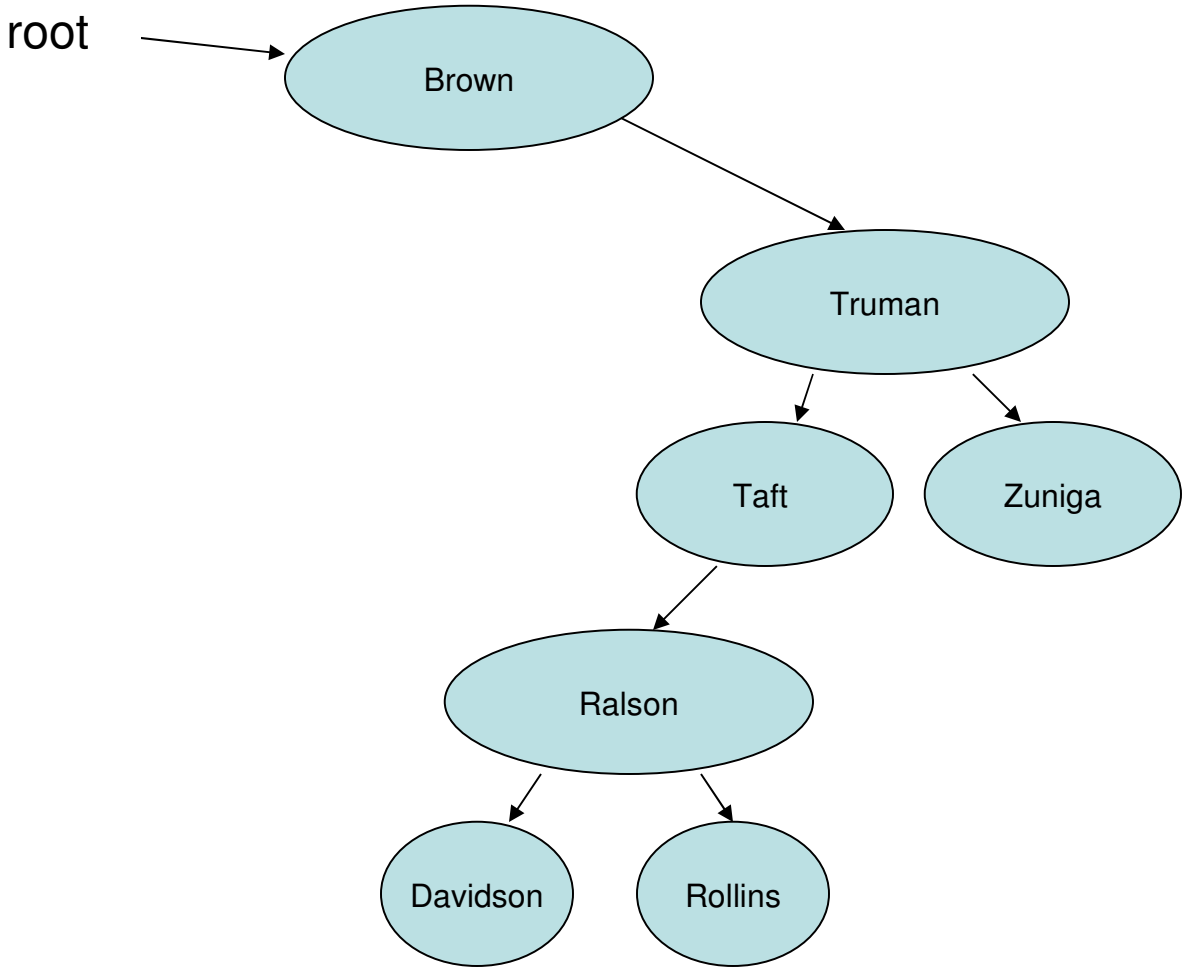
External nodes
Internal nodes
Subtree – left subtree of research?
Ancestor – papers ancestor of activities?
Descendant – papers descendant of home?

Ordered Trees

- Linear relationship between child nodes
- Binary tree – max two children per node
  - Left child, right child



Another Ordered Binary Tree

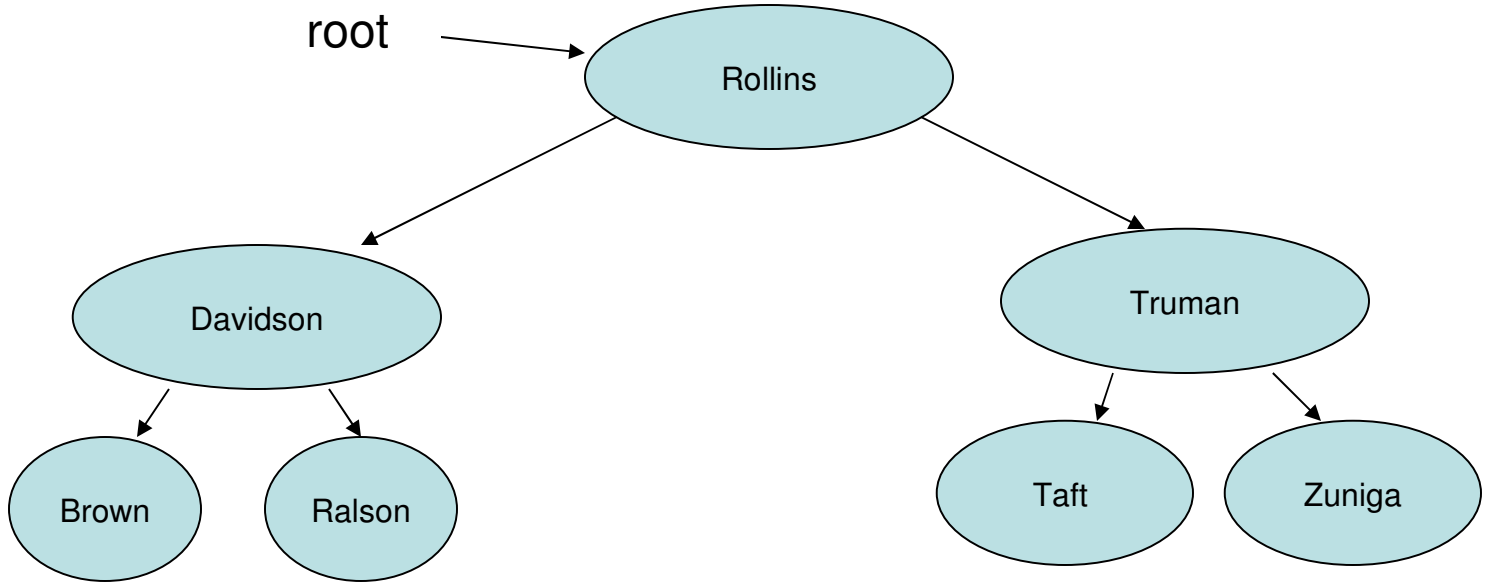


Tree Traversal

- Pre-order traversal
  - Visit node, traverse left subtree, traverse right subtree
- Post-order traversal
  - Traverse left subtree, traverse right subtree, visit node

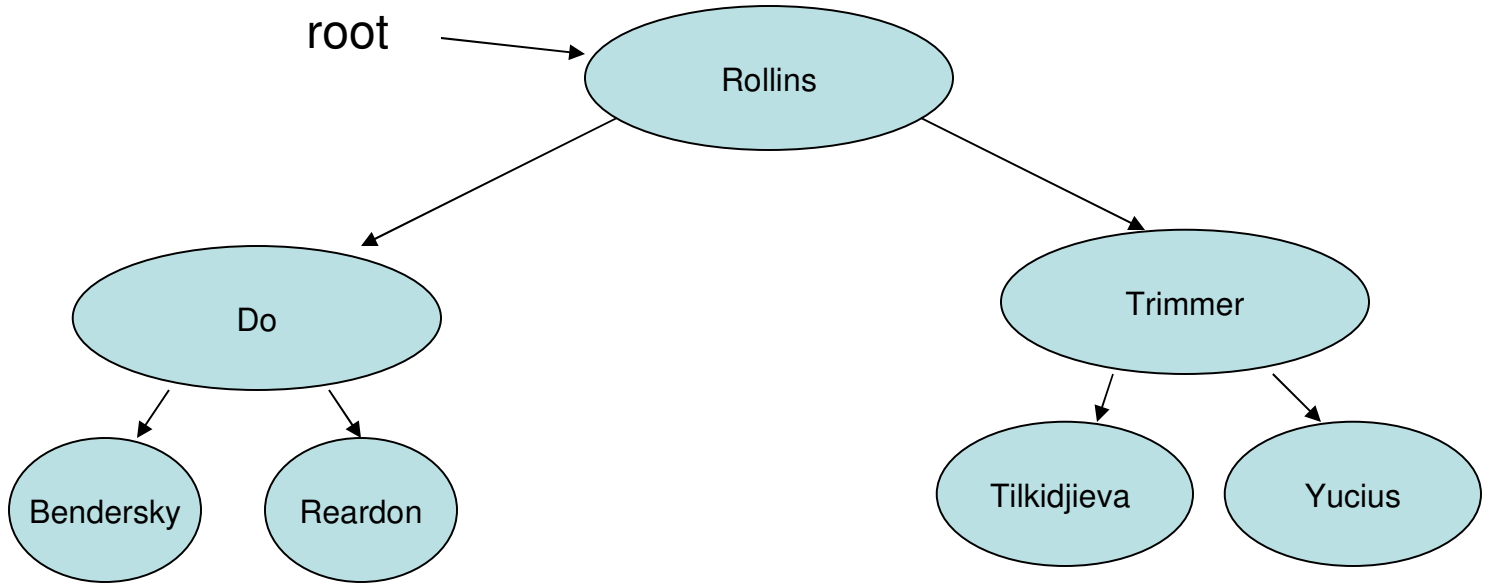
Example

- Pre-order
- Post-order



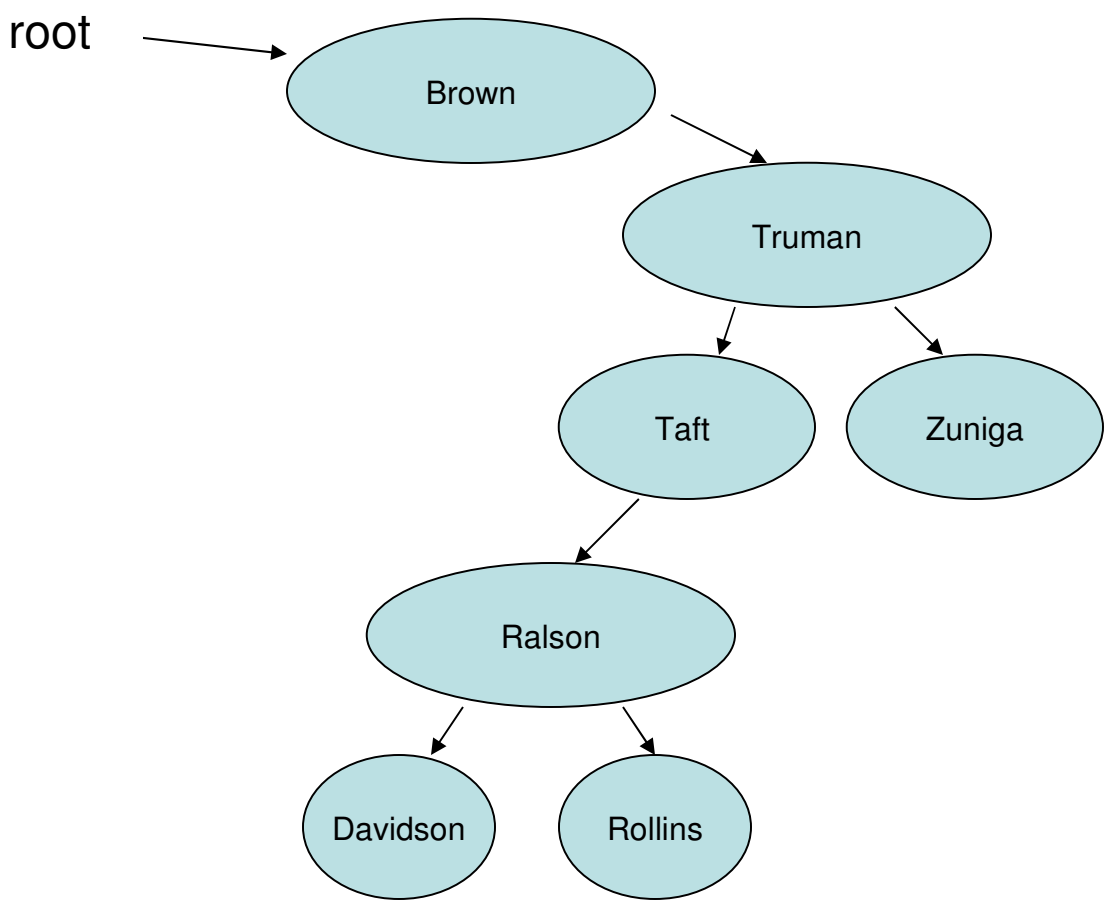
Example

- Pre – Rollins, Davidson, Brown, Ralson, Truman, Taft, Zuniga
- Post – Brown, Ralson, Davidson, Taft, Zuniga, Truman, Rollins



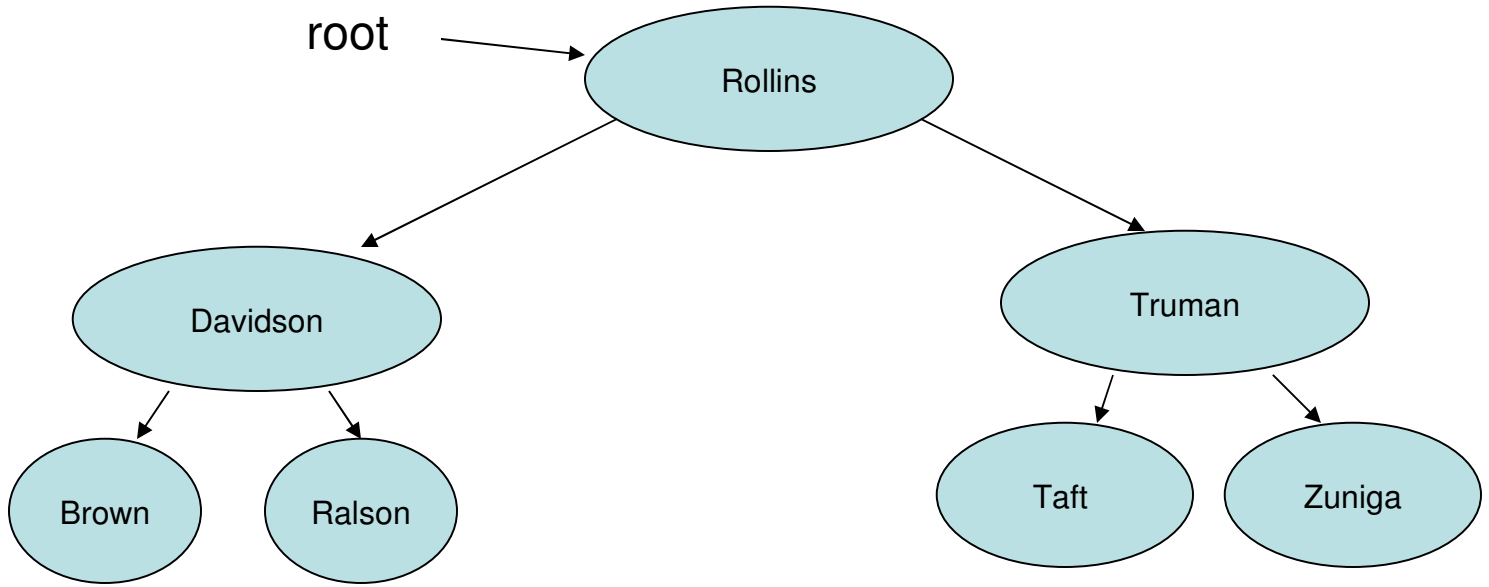
Another Example

- Pre – Brown, Truman, Taft, Ralson, Davidson, Rollins, Zuniga
- Post – Davidson, Rollins, Ralson, Taft, Zuniga, Truman, Brown



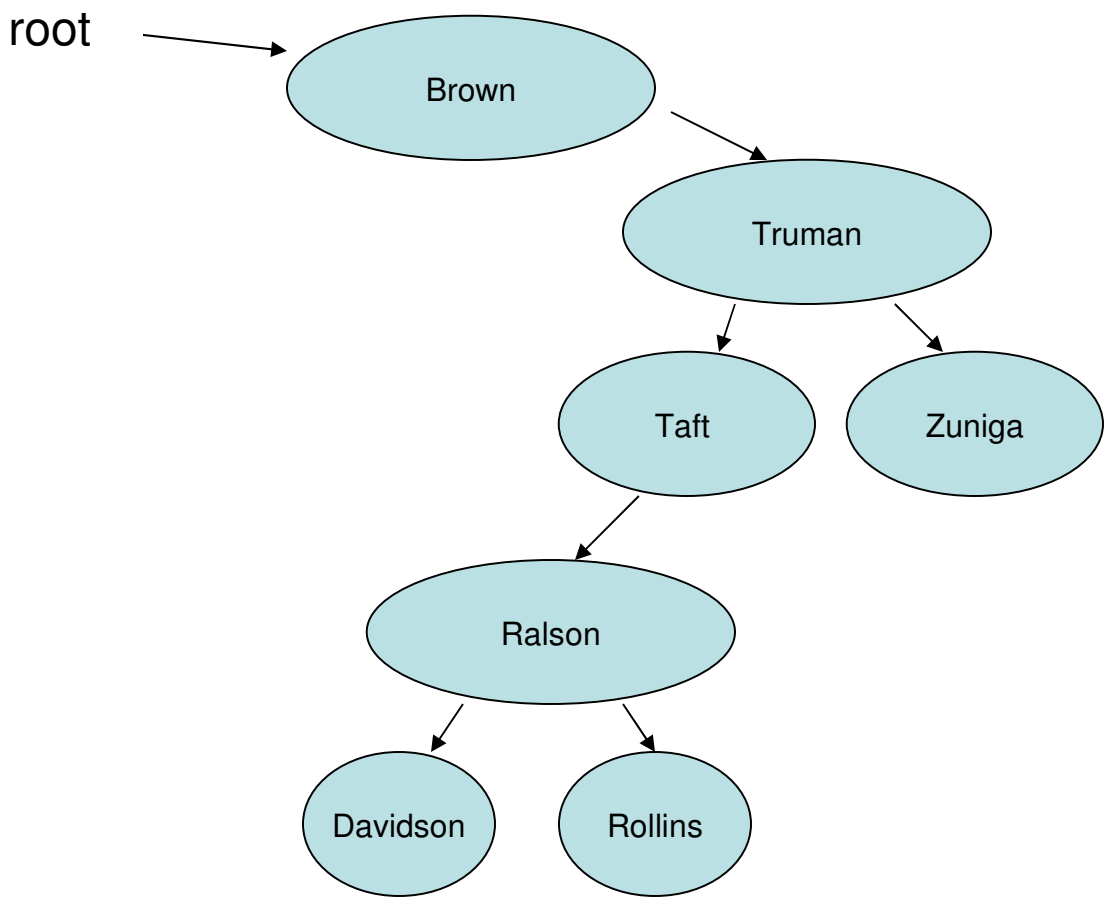
In-order Traversal

- Traverse left subtree, visit node, traverse right subtree
  - Brown, Davidson, Ralson, Rollins, Taft, Truman, Zuniga

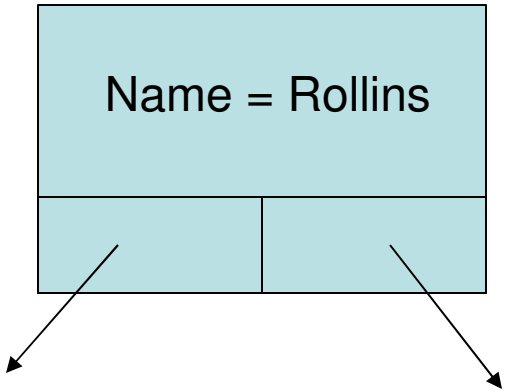


Another Example

- In-order – Brown, Davidson, Ralson, Rollins, Taft, Truman, Zuniga

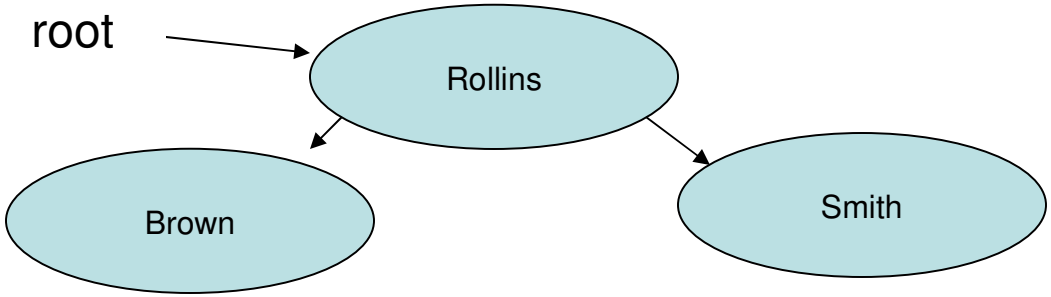


Implementation – TreeNode



- Data members?
- Functions?

Implementation – Tree



- Data Members
- Functions
  - Pre/post/in-order print

Implementation – Pre-order

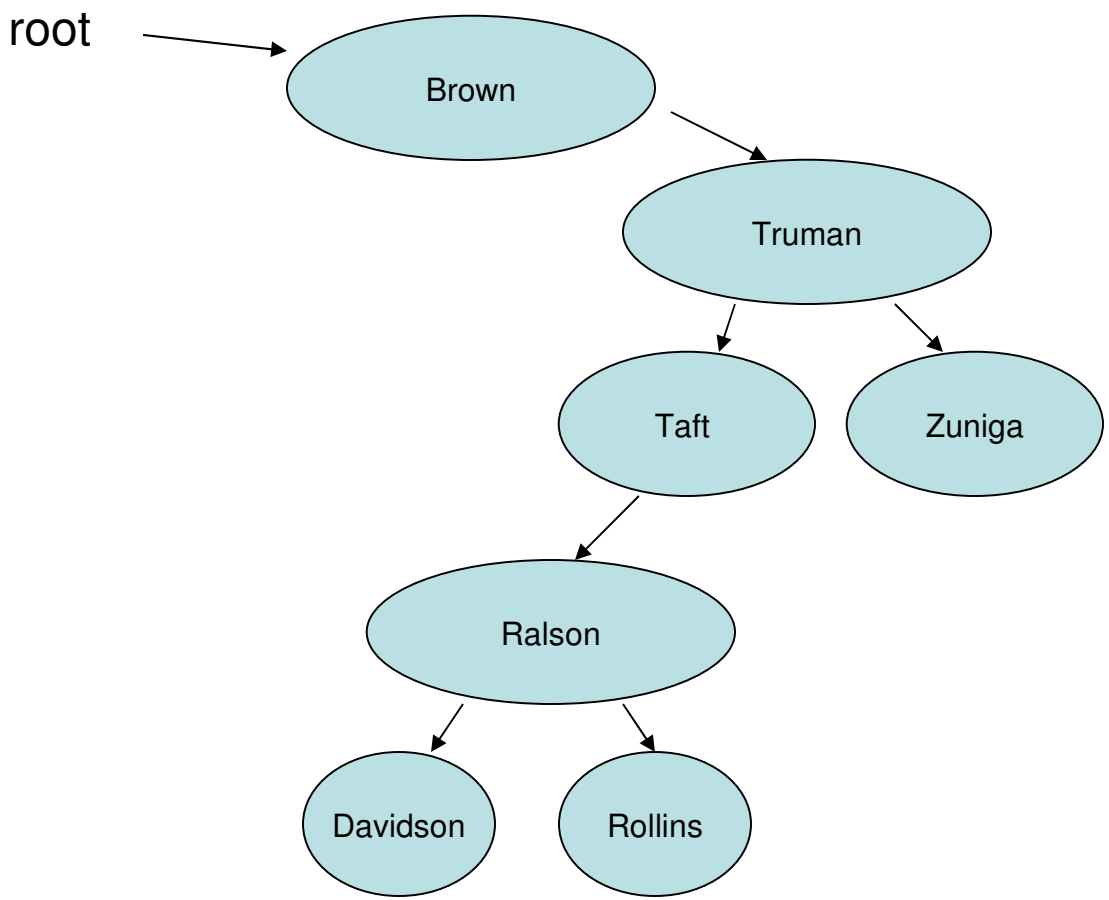
```
void preOrderPrint(TreeNode* curnode) {
    o.print();
    if(curnode->getLeftChild() != NULL)
        preOrderPrint(curnode->getLeftChild());
    if(curnode->getRightChild() != NULL)
        preOrderPrint(curnode->getRightChild());

}

Tree* t = …;
t->preOrderPrint(t->getRoot());
```

BSTs

- Elements in left subtree nodes are before (are less than) element in current node
- Element in current node is before (less than) elements in right subtree

find Operation

- Algorithm for finding element in BST



find Algorithm

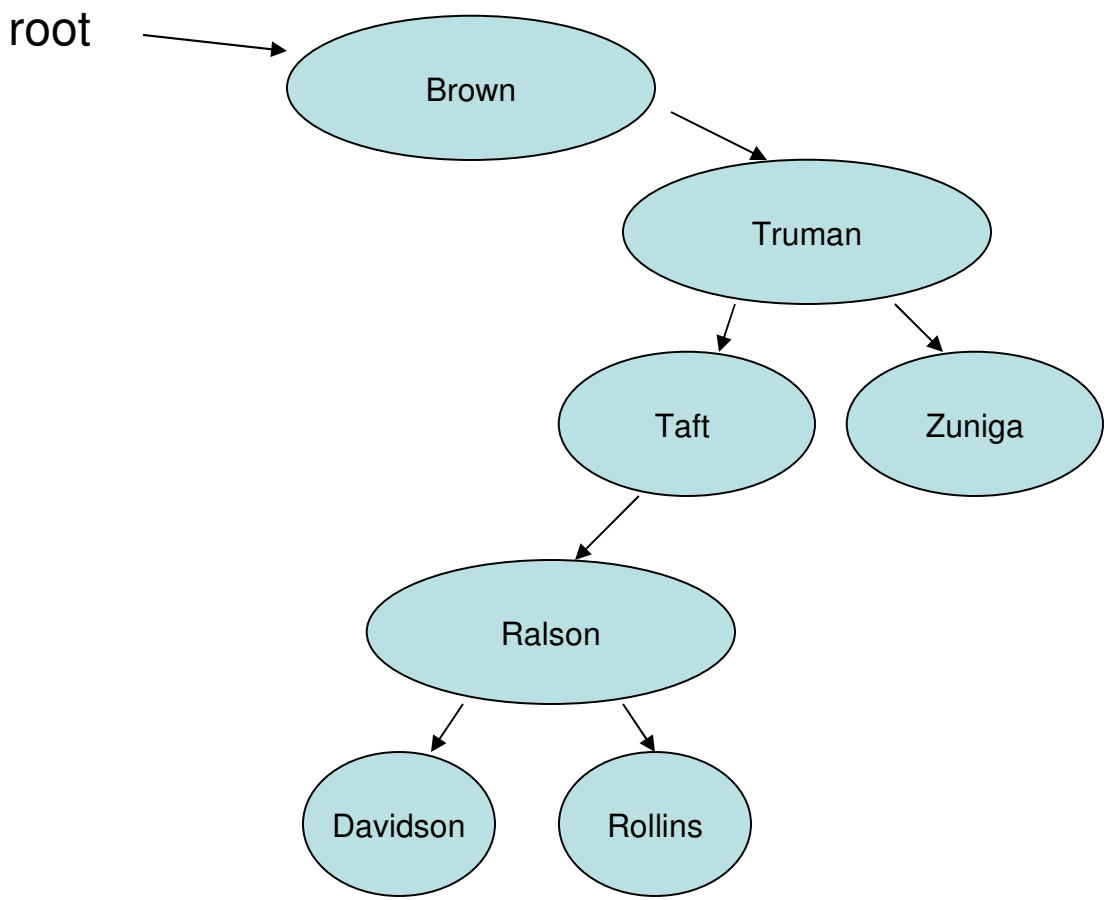
```
if current node is null
    return not found
else if target is in current node
    return found
else if target is before current node
    return find(left child)
else
    return find(right child)
```

find Complexity

- Worst case
- Best case
- Average case

insert Operation

- Algorithm for inserting element in BST

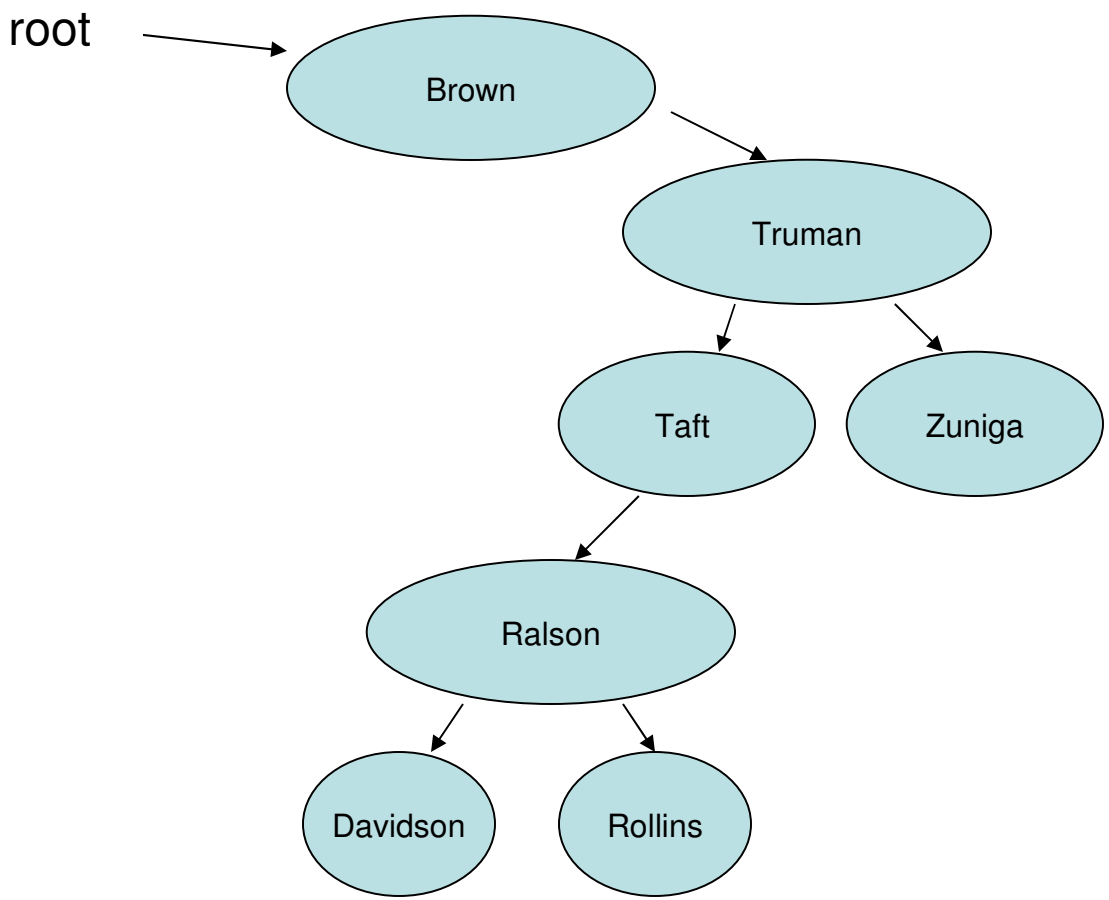


insert Algorithm

```
if new_elt is before current and current left child is null
    insert as left child
else if new_elt is after current and current right child is null
    insert as right child
else if new_elt is before current
    insert in left subtree
else
    insert in right subtree
```

remove Operation

- Algorithm for removing element in BST



remove Algorithm

elt = find node to remove
if elt left subtree is null
    replace elt with right subtree
else if elt right subtree is null
    replace with left subtree
else
    find successor of elt
    (go right once and then left until you hit null)
    replace elt with successor
    call remove on successor