CS261 Data Structures

Tree Traversals

Goals

- Euler Tours
- Recursive Implementation
- Tree Sort Algorithm

Binary Tree Traversals

- What order do we ***enumerate*** nodes in a tree?

Binary Tree Traversals

- All traversal algorithms have to:
  - Process a node (i.e. do something with the value)
  - Process left subtree
  - Process right subtree

  Traversal order determined by the order these operations are done

- Six possible traversal orders:
  1. Node, left, right → Pre-order
  2. Left, node, right → In-order
  3. Left, right, node → Post-order

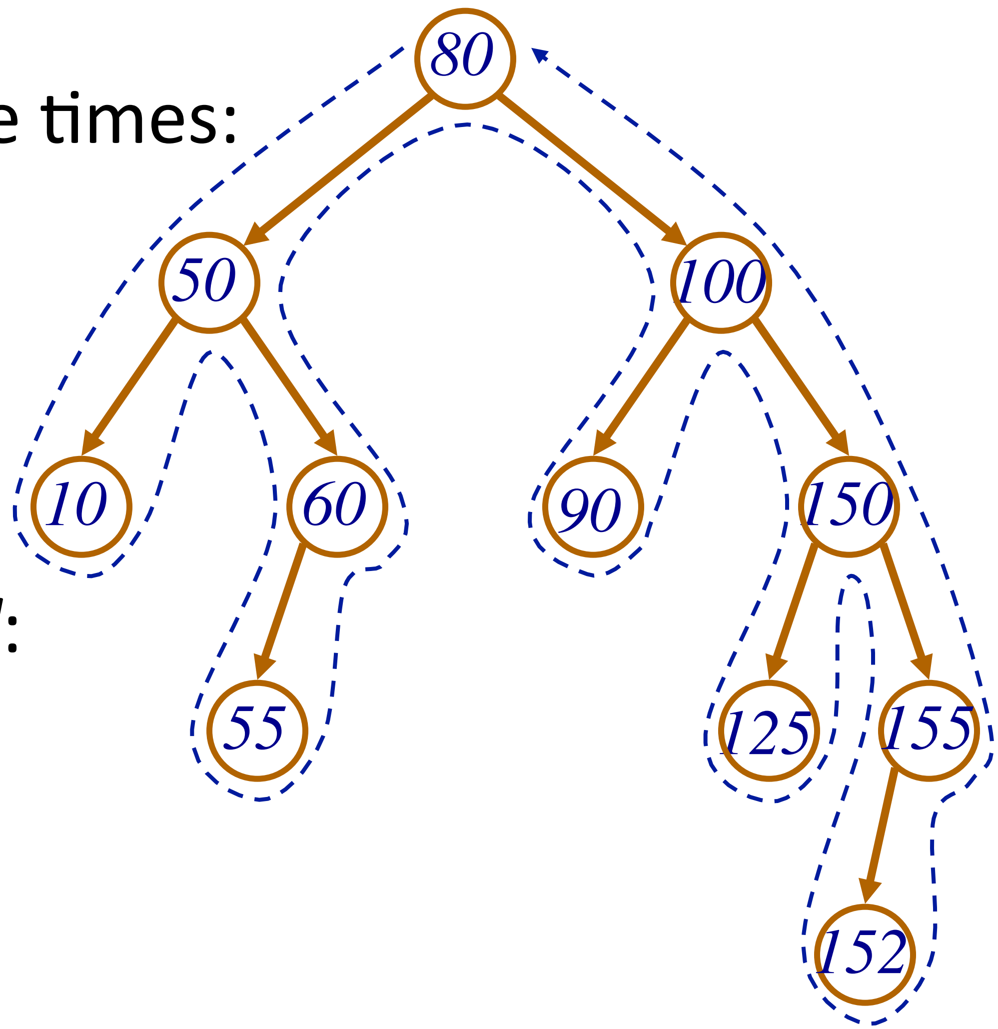
  Most common

  4. Node, right, left
  5. Right, node, left
  6. Right, left, node

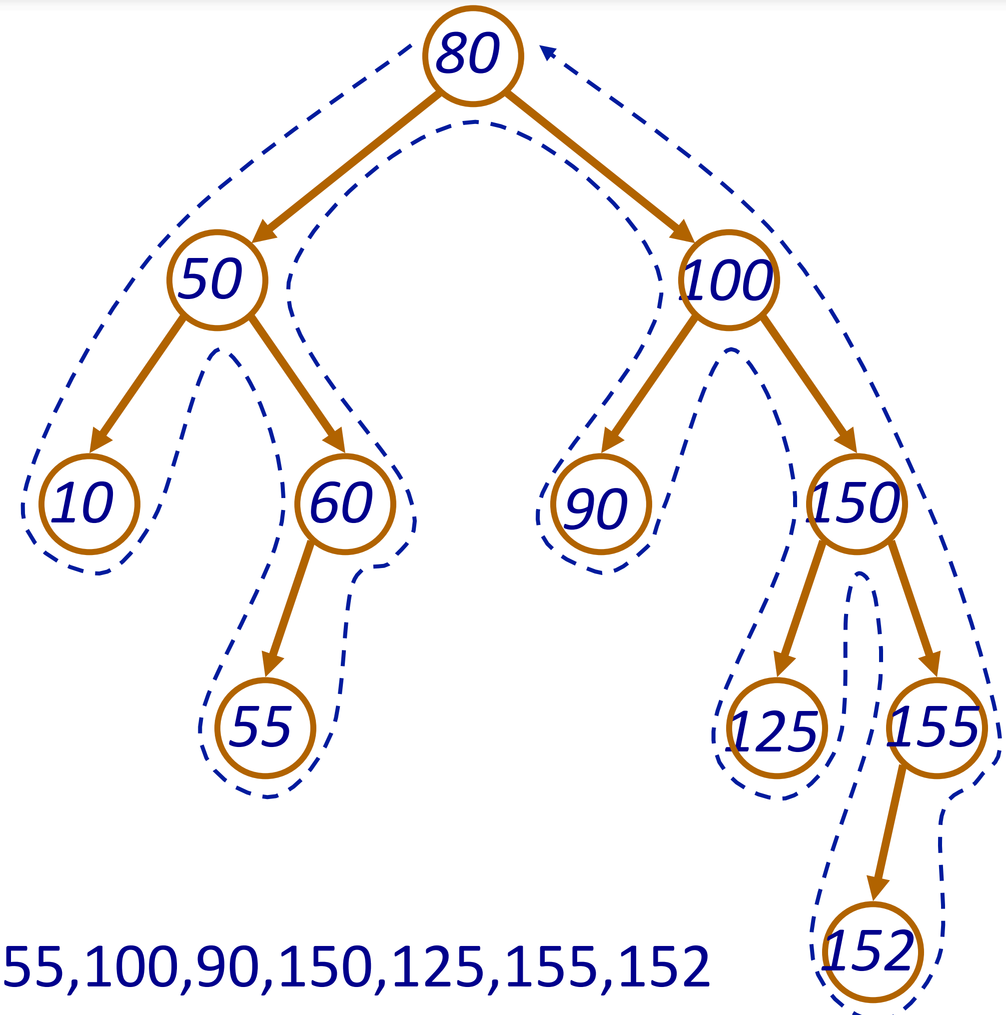
  Subtrees are ***not*** usually analyzed from right to left.

Binary Tree Traversals: Euler Tour

- An Euler Tour "walks" around the tree's perimeter, without crossing edges
- Each node is ***visited*** three times:
  - 1st visit: left side of node
  - 2nd visit: bottom side of node
  - 3rd visit: right side of node
- Traversal order depends on when node ***processed***:
  - Pre-order: 1st visit
  - In-order: 2nd visit
  - Post-order: 3rd visit



Example



Pre: 80,50,10,60,55,100,90,150,125,155,152

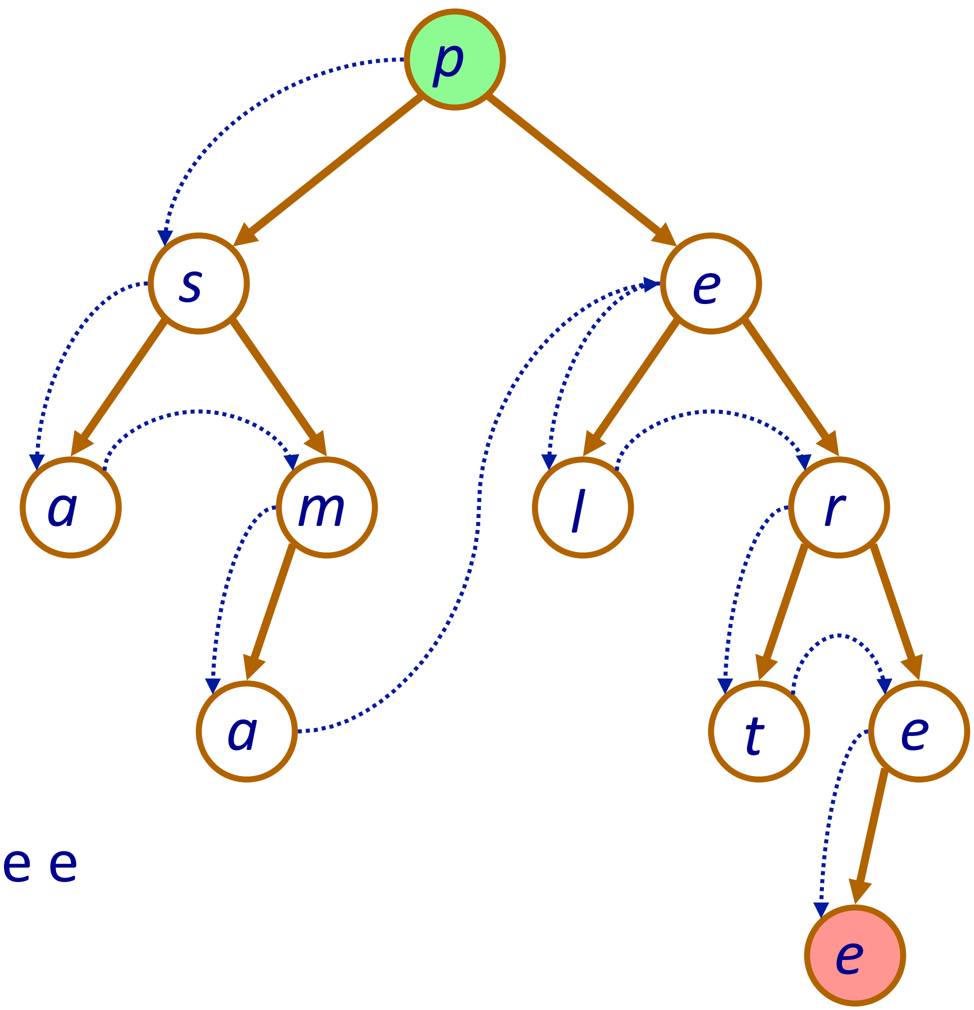
Post: 10,55,60,50,90,125,152,155,150,100,80

In: 10,50,55,60,80,90,100,125,150,152,155

Recursive Traversal

- Process order → Node, Left subtree, Right subtree

```
void preorder(struct Node *node) {
  if (node != 0){
    process (node->val);
    preorder(node->left);
    preorder(node->rght);
  }
}
```



Example result: p s a m a e l r t e e

Euler Tour: General Recursive Implementation

```
void EulerTour(struct Node *node) {
     if(node != 0)
     {
         beforeLeft(node);
         EulerTour(node->left);
          inBetween(node)
         EulerTour(node->rght);
         afterRight(node);
      }
 }
```

void beforeLeft (Node n) { printf("("); }

void inBetween (Node n) { printf("%s\n", n.value); }

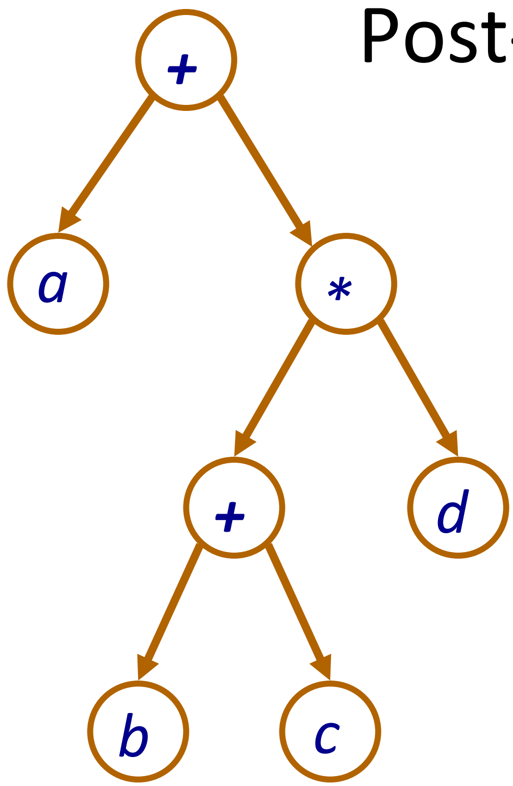
void afterRight (Node n) { printf(")"); }

Traversal Example – Expression Tree

Pre-order: + a * + b c d (Polish notation)

In-order: (a + ((b + c) * d)) (parenthesis added)

Post-order: a b c + d * + (reverse Polish notation)



Complexity

- Computational complexity:
  - Each traversal requires constant work at each node (not including recursive calls)
  - each node is processed a max of 3 times (in the general case): still constant work
  - recursive call made once on each node
  - Iterating over all n elements in a tree requires $O(n)$ time

Problems

- Problems with traversal code:
  - If external (ie. user written): exposes internal structure (access to nodes) → Not good information hiding
  - Can make it internal (see our PrintTree in AVL.c), and require that the user pass a function pointer for the 'process'
  - Recursive function can't return single element at a time. Can't support a typical looping structure.
  - Solution → Iterator (more on this later)

Tree Sort

- An AVL tree can easily sort a collection of values:
  1. Copy the values of the data into the tree: $O(n \log n)$
  2. Copy them out using an in-order traversal: $O(n)$

  In-order on a BST produces elements in sorted order!!

- As fast as QuickSort

- Does not degrade for already sorted data

- However, requires extra storage to maintain both the original data buffer (e.g., a `DynArr`) and the tree structure

Your Turn

- Complete Worksheet32: Tree Sort