Graphs New

Tree Traversals & Iterators
Goals

• Euler Tours
• Recursive Implementation
• Tree Sort Algorithm
Binary Tree Traversals

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

```
80
  50  100
   10  60  90  125
    55 60  150
     150

152
```
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
  4. Node, right, left
  5. Right, node, left
  6. Right, left, node

  Most common

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:
  
  – 1\textsuperscript{st} visit: left side of node
  – 2\textsuperscript{nd} visit: bottom side of node
  – 3\textsuperscript{rd} visit: right side of node

• Traversal order depends on when node **processed**:
  
  – Pre-order: 1\textsuperscript{st} visit
  – In-order: 2\textsuperscript{nd} visit
  – Post-order: 3\textsuperscript{rd} 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
• All traversal algorithms have to:
  – Process a node (i.e. do something with the value)
  – Process left subtree
  – Process right subtree
Recursive Traversal: Preorder

- **Process order** → Node, Left subtree, Right subtree (preorder)

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

Example result: p s a m a e l r t e e
Euler Tour: General Recursive Implementation

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

void beforeLeft (Node n) { printf("\(\)"); }
void inBetween (Node n) { printf("%s\n", n.value); }
void afterRight (Node n) { printf("\)\n"); }
```
Traversal Application 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)
BSP Tree
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/AVL 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
Binary Tree Iterator
void inorder(struct Node *node) {
    if (node != 0) {
        inorder(node->left);
        process (node->val);
        inorder(node->right);
    }
}
Goals

• In-Order traversal that supports the Iterator Interface (HasNext, Next)
  – Concepts
  – Implementation
Can we build an iterator for a tree utilizing a data structure that we DO know how to iterate over?
Simple Iterator

• Simple iterator \(\rightarrow\) recursively traverse tree, placing all node values into a linked list, then use a linked list iterator

• Problem: duplicates data, uses twice as much space

• Can we do better?
Exercise

What is being stored in the process stack?

```c
void inorder(struct Node *node) {
    if (node != 0){
        inorder(node->left);
        process (node->val);
        inorder(node->right);
    }
}
```
Exercise

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

When Process Abigail:
- Robert
- Gabby
- Abigail
are all unfinished!

When Process Dave:
- Robert
- Gabby
- Abigail
- Dave
are all unfinished!

Process Stack represents a path to the leftmost unprocessed node!!

InOrder Robert
- InOrder Gabby
- InOrder Null
- Process Abigail
- InOrder Dave
- InOrder Null
- Process Dave
- InOrder Null
- Process Gabby
- InOrder Kate
• Simulate recursion using a stack

• ‘Stack’ the path as we traverse down to the leftmost element (smallest in BST)

• Useful routine:

```c
void _slideLeft(struct Stack *stk, struct Node *n)
    while (n != 0) {
        pushStack(stk, n);
        n = n->left;
    }
```

Binary Tree In-Order Iterator

Main Idea

- **Next** returns the top of the stack (e.g. the next element you’ll go UNDER in Euler Tour)

- **HasNext**
  
  - Returns true if there are elements left (on stack) to iterate
  
  - Sets up the subsequent call to ‘Next()’ by making sure the leftmost node (smallest unprocessed node in BST) element is on top of the stack. It does this by calling _slideLeft on the node’s right child
BST In-Order Iterator: Algorithm

Initialize: create an empty stack
hasNext:
  if stack is empty (first time) perform slide left on root
  otherwise
    let n be top of stack
    pop n
    slide left on right child of n
return true if stack is not empty (false otherwise)
next:
  return value of node on top of stack (but don’t pop node)
In-Order Iterator: Simulation

- On stack (lowest node at top).
- Not yet visited.
- Enumerated (order indicated).

Initialized in hasNext() slideLeft

next, hasNext pop

next, hasNext pop, slideLeft

next, hasNext pop

next, hasNext pop
In-Order Iterator: Simulation

- On stack (lowest node at top).
- Not yet visited.
- Enumerated (order indicated).

1. Next, hasNext
2. Pop
3. Next, hasNext
4. Next, hasNext
5. Pop
6. Next, hasNext
7. Pop, slideLeft
8. Next, hasNext
9. Next, hasNext
10. Next, hasNext

Trees and nodes are marked with symbols indicating their status or action.
Complexity?

- Each node goes on the stack exactly one time
- Each node is popped off the stack exactly one time
- $O(N)$
Other Traversals

• Pre-order and post-order traversals also use a stack
• See Chapter 10 discussion
Haven’t seen this traversal yet:

– Traverse nodes a level at a time from left to right
– Start with root level and then traverse its children and then their children and so on

– Implementation?

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

Complete Worksheet #30: BST Iterator