CS 261
Binary Tree Traversals
Binary Search Trees
by
Tim Budd
Sinisa Todorovic
Tree Traversals

• How to examine every node in a tree

• A list is a simple linear structure:
  – can be traversed either forward or backward

• What order do we visit nodes in a tree?

• Most common traversal orders:
  – Pre-order
  – In-order
  – Post-order
Binary Tree Traversals

• All traversal algorithms have to:
  – Process node
  – Process left subtree
  – Process right subtree

Traversal order is determined by the order these operations are done
Binary Tree Traversals

- 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:
  \[ \text{Node, left, right} \rightarrow \text{Pre-order} \]

- Subtrees are not usually analyzed from right to left.
Pre-Order Traversal

- **Process order** → Node, Left subtree, Right subtree

```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`
Post-Order Traversal

- **Process order** → Left subtree, Right subtree, Node

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

Example result: a a m s l t e e r e p
In-Order Traversal

• **Process order** → Left subtree, Node, Right subtree

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

Example result: a sample tree
Traversals

• Computational complexity:
  – Each traversal requires constant work at each node (not including recursive calls)
  – Order not important
  – Iterating over all $n$ elements in a tree requires $O(n)$ time
Traversals

• Problems with tree traversals:
  – If internal: ties (task dependent) **process** method to the tree implementation
  – If external: exposes internal structure (access to nodes) → Not good information hiding
  – Recursive function **can’t** return single element at a time
  – Solution → Iterator (more on this later)
Binary Search Trees
Binary Search Tree

• Binary trees where every node value is:
  – Greater than all its left descendants
  – Less than or equal to all its right descendants
  – In-order traversal returns elements in sorted order

• If tree is reasonably full (well balanced), searching for an element is $O(\log n)$
Binary Search Tree: Example
Bag Implementation: **Contains**

- Start at root
- At each node, compare value to node value:
  - Return true if match
  - If value is less than node value, go to left child (and repeat)
  - If value is greater than node value, go to right child (and repeat)
  - If node is null, return false
Bag Implementation: Contains

• Traverses a path from the root to the leaf
• Therefore, if tree is *reasonably full*, execution time is \( O(??) \)
**BST: Contains/Find Example**

Object to find → Agnes
Binary Search Tree (BST): Implementation

```c
struct BSTree {
    struct Node *root;
    int cnt;
};

void initBSTree(struct BSTree *tree);
void addBSTree(struct BSTree *tree, TYPE val);
int containsBSTree(struct BSTree *tree, TYPE val);
void removeBSTree(struct BSTree *tree, TYPE val);
int sizeBSTree(struct BSTree *tree);
```
struct Node {
    TYPE val;
    struct Node *left; /* Left child */
    struct Node *right; /* Right child */
};
BST: Add Example

Before first call to `add`
Node _addNode(Node current, TYPE val){
    if current is null
        return new Node with val
    else if val < Node.val
        leftchild = addNode(leftchild, val)
    else
        rightchild = addNode(rightchild, val)
    return current node
}
Add: Calls Utility Routine

```c
void add(struct BSTree *tree, TYPE val) {
    tree->root = _addNode(tree->root, val);
    tree->cnt++;
}
```

What is the complexity? $O(??)$
How would you remove Abigail? Audrey? Angela?
Who fills the hole in the tree?

• Answer:

  the leftmost child of the right child
  (smallest element in right subtree)

• Try this on a few values
BST: Remove Example

Before call to remove

Replace with: leftmost(right)
BST: Remove Example

After call to `remove`
Special Case

• What if you don’t have the right child?

• Try removing “Audrey”
  – Simply, return the left child
Useful Routines

TYPE _leftmost(struct Node *cur)
{
    ... /*Returns value of leftmost child of current node*/
}

struct Node *_removeLeftmost(struct Node *cur)
{
    ... /*Returns tree with leftmost child removed*/
}
Remove

Node removeNode(Node current, TYPE val){
    if val == current.val
        if rightchild is NULL
            return leftchild;
        else
            replace val with
            the leftmost child of the rightchild;
            rightchild = removeLeftmost(right);
    else if val < current.value
        leftchild = removeNode(leftchild, val);
    else
        rightchild = removeNode(rightchild, val);
    return current
}
Complexity

• Running down a path from root to leaf

• What is the time complexity? \( O( ?? ) \)