CSE 143

Trees

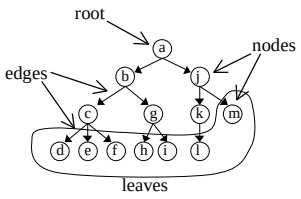
[Chapter 10]

Everyday Examples of Trees

- Family tree
- Company organization chart
- Table of contents
- Any hierarchical organization of information, representing components in terms of their subcomponents

NB: Some of these are not true trees in the CS definition of the term

A Tree

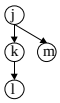


Tree Terminology

- Empty tree: tree with no nodes
- *Child* of a node u
  - Any node reachable from u by 1 edge pointing away from u
  - Nodes can have zero, one, or more children
  - *Leaves* have no children
- If `b` is a child of `a`, then `a` is the *parent* of `b`
  - All nodes except root have exactly one parent
  - *Root* has no parent

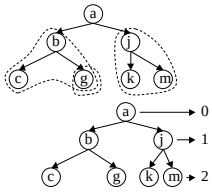
Tree Terminology (2)

- *Descendant* of a node (recursive definition)
  - P is a descendant of P
  - If C is child of P, and P is a descendant of A, then C is a descendant of A
  - Example: `k` and `m` are descendants of `j`, `l` is descendant of `j`, `k`, and `l`
- *Ancestor* of a node
  - If D is a descendant of A, then A is an ancestor of D
  - Example: `j`, `k`, and `l` are ancestors of `l`



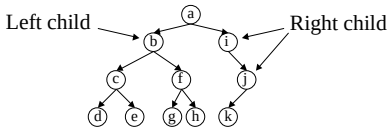
Tree Terminology (3)

- Subtree
  - Any node of a tree, with all of its descendants
- Depth (recursive definition)
  - Depth of root node is 0
  - Depth of any node other than root is one greater than depth of its parent
- Height
  - Height of a tree is maximum of all depths of its leaves
- Warning: Definitions vary
  - Some define depth of the root node as 1.



Binary Trees

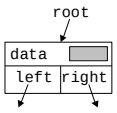
- A *binary tree* is a tree each of whose nodes has exactly zero, one, or two children
- Two children are called the left child and right child



Binary Tree Data Structure

- Usually use a variant of a Node:

```
struct BTreeNode {
  int data;
  BTreeNode *left;
  BTreeNode *right;
};
```



- Keep a `root` pointer to the root node
  - Empty tree has a `NULL` root
- Note the recursive data structure
- As the data structure is recursive, algorithms often are recursive as well

Example: Counting Nodes

- Base case: Empty tree has zero nodes
- Recursive case: Nonempty tree has root + nodes in left subtree + nodes in right subtree

```
int CountNodes(BTreeNode *root)
{
  if ( root == NULL )
    return 0;  // base
  else
    return 1 + CountNodes(root->left)
             + CountNodes(root->right);
}
```


Finding the Height

- Base case: Empty tree has height `-1`
- Recursive case: Nonempty tree has height `1` more than maximum height of left and right subtrees

```
int Height(BTreeNode *root) {
  if ( root == NULL )
    return -1;
  else
    return 1 + max(Height(root->left),
                   Height(root->right));
}
```


Analyses

- What is running time of these algorithms?
  - Base case: `O(1)` for both
  - Recursive case: `O(N)` for both, where `N` is the number of nodes in tree
- How to write an iterative version?
  - Try it

Recursive Tree Searching

- How to tell if a data item is in a binary tree?

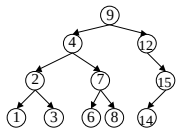
```
bool Find(BTreeNode *root, int item) {
  if ( root == NULL )
    return false;
  else if ( root->data == item )
    return true;
  else
    return ( Find(root->left, item) ||
             Find(root->right, item) );
}
```

- What is the running time of this algorithm?
  - Worst case: Has to visit every node in the tree, `O(N)`
  - Can we do better?

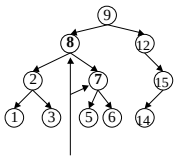
Binary Search Trees [Section 13.3]

- Properties:
  - No duplicates
  - Prereq: The "greater-than" and "less-than" relations are well-defined for the data values.
- Sorting constraints: for every node v
  - All data in left subtree of v < data in root
  - All data in right subtree of v > data in root
- A binary tree with these constraints is called a *binary search tree* (BST)
- Given a set of values, there could be many possible BSTs

Examples and Non-Examples



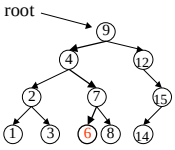
A Binary Search Tree



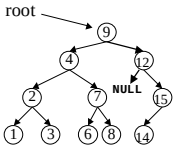
Not a Binary Search Tree

Finding an item in a BST

`Find(root, 6)`



`Find(root, 10)`



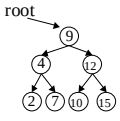
Finding an item in a BST (2)

If we have a binary search tree, then Find can be done as:

```
bool Find(BTreeNode *root, int item)  {
  if ( root == NULL )
    return false;
  else if (item == root->data)
    return true;
  else if (item < root->data)
    return Find(root->left, item);
  else
    return Find(root->right, item);
}
```

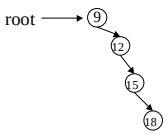

Running time of `Find`

- Best case: `O(1)`, item is at root
- Worst case: `O(h)`, where h is *height* of tree
- Leads to a question:
  - What is the height of a binary search tree with `N` nodes?
- "Perfect" tree (2^d nodes at each depth `d`) is best case:
  - $N = 2^{h+1} - 1$
  - $h = \log_2(N+1) - 1 = O(\log N)$
  - logarithmic running time



Running time of `Find` (2)

- What if tree isn't balanced?
- Worst case is *degenerate* tree
  - Height = `N-1`, `N` is number of nodes
- Running time of `Find`, worst-case, is `O(N)`



Tables Revisited

- Using a List or Array to implement Table can be inefficient for searching
- Could use binary search trees to implement a table (dictionary)
- Must support `Insert` and `Delete` in addition to `Find`
- Must maintain BST ordering constraint

Inserting in a BST

To insert a new key:

- Base case:
  - If tree is empty, create new node for item
  - If root holds key, return (no duplicate keys allowed)
- Recursive case: If key < root's value, recursively add to left subtree, otherwise to right subtree

Example

Add 8, 10, 5, 1, 7, 11 to an empty BST, in that order:

Inserting in a BST (2)

```
// Add data to tree root and return ptr to tree
BTreeNode * Insert(BTreeNode *root, int data) {
  if ( root == NULL ) {
    BTreeNode *tmp = new BTreeNode;
    tmp->left = tmp->right = NULL;
    tmp->data = data;
    return tmp;
  }
  if (data < root->data )
    root->left = Insert(root->left, data);
  else if (data > root->data )
    root->right = Insert(root->right, data);
  return root;
}
```


Example (2)

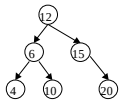
- What if we change the order in which the numbers are added?
- Add 1, 5, 7, 8, 10, 11 to a BST, in that order (following the algorithm):

Complexity of `Insert`

- Base case: `O(1)`
- How many recursive calls?
  - For each node added, takes `O(H)`, where `H` is the height of the tree
- Again, what is height of tree?
  - Balanced trees yields best-case height of `O(log N)` for `N` nodes
  - Degenerate trees yield worst-case height of `O(N)` for `N` nodes
  - For random insertions, expected height is `O(log N)` -- true, but not simple to prove

Deleting an Item from a BST

- Simple strategy: lazy deletion (just mark the node as "deleted")
- The hard way. Must deal with 3 cases
  - 1. The deleted item has no children (easy)
  - 2. The deleted item has 1 child (harder)
  - 3. The deleted item has 2 children (hardest)



Deletion (2): Algorithm

- First find the node (call it N) to delete.
- If N is a leaf, just delete it.
- If N has just one child, have N's parent bypass it and point to N's child.
- If N has two children:
  - Replace N's key with the *smallest key K of the right subtree*
  - (Recursively) delete the node that had key K (this node is now useless)
  - Note: The smallest key always lives at the leftmost "corner" of a subtree (why?)

Deletion (3): Finding the Node

- This is the "easy" part:

```
BTreeNode* delItem(int item, BTreeNode* t) {
   if (t != NULL) {
     if (item == t->data)
       t = delNode(t);
     else if (item > t->data)
       t->right = delItem(item, t->right);
     else
       t->left = delItem(item, t->left);
   }
   return t;
}
```


Deletion (4): Deleting the Node

```
BTreeNode* delNode(BTreeNode* t) {
 if (t->left && t->right) {        // 2 children
    t->data = findMin(t->right);
    t->right = delItem(t->data, t->right);
    return t;
 }
 else {                            // 0 or 1 child
    BTreeNode* rval = NULL;
    if (t->left)              // left child only
      rval = t->left;
    else if (t->right)        // right child only
      rval = t->right;
    delete t;
    return rval;
 }
}
```


Deletion (5): Finding Min

- All that remains is to figure out how to find the minimum value in a BST
- Remember, the minimum element lives at the leftmost "corner" of a BST

```
// PRECONDITION:  must be called on non-NULL pointer

int findMin(BTreeNode* t)
{
  assert(t != NULL);
  while (t->left != NULL)
    t = t->left;
  return t->data;
}
```


Balanced Search Trees

- BST operations are dependent on tree height
  - `O( log N)` for `N` nodes if tree is balanced
  - `O(N)` if tree is not
- Can we ensure tree is always balanced?
  - Yes: `Insert` and `Delete` can be modified to reorganize tree if it gets unbalanced
  - Exact details more complicated
  - Results in `O( log N)` dictionary operations, even in worst case

Tree Traversal

- As with lists, would like to be able to iterate through all nodes in a tree
  - How to have equivalent to `Start()`, `IsEnd()`, `NextElement()`?
- What order should nodes be visited in?
  - Top-down?
  - Left-to-right?
  - Bottom-up?
  - How to deal with recursive nature of trees?
- Visiting nodes of a tree is called *tree traversal*

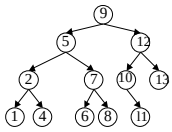
Types of Traversal

- Preorder traversal:
  - Visit the node *first*
  - Recursively do preorder traversal on its children, in some order
- Postorder traversal:
  - Recursively do postorder traversals of children, in some order
  - Visit the node *last*

Inorder (for Binary Trees)

- With a binary tree, each node has at most two children, called `left` and `right`
- Inorder traversal:
  - Recursively do inorder traversal of left child
  - Then visit the node
  - Then recursively do inorder traversal of right child
- For preorder and postorder traversals, typically traverse left child before right one

Example of Tree Traversal



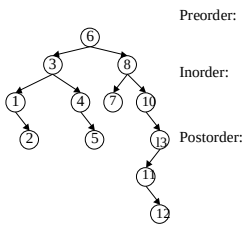
Preorder:

Inorder:

Postorder:

Another Example

What about this tree?



Preorder:

Inorder:

Postorder:

Example Traversal: Printing

```
// print the nodes of tree with given root
void Preorder (BTreeNode *root)
{
  if ( root == NULL )
    return;
  // Visit the node first
  cout << root->data << "  ";
  Preorder (root->left);
  Preorder (root->right);
}
```

Example Traversal: Printing

```
void printInOrder(BTreeNode* t) {
  if (t != NULL) {
    printInOrder(t->left);
    cout << t->data << " ";
    printInOrder(t->right);
  }
}
void printPreOrder(BTreeNode* t) {
  if (t != NULL) {
    cout << t->data << " ";
    printInOrder(t->left);
    printInOrder(t->right);
  }
}
```

- Why might these traversals be useful?

Another Example

- Use a postorder traversal to return a whole tree to the heap.

```
void deleteTree(BTreeNode* t) {
  if (t != NULL) {
    deleteTree(t->left);
    deleteTree(t->right);
    delete t;
  }
}
```

- Would inorder or preorder work as well?

Analysis of Tree Traversal

- How many recursive calls?
  - Two for every node in tree (plus one initial call);
  - `O(N)` in total for `N` nodes
- How much time per call?
  - Depends on complexity `O(V)` of the visit
  - For printing and most other types of traversal, visit is `O(1)` time
- Multiply to get total
  - `O(N)*O(V) = O(N*V)`
- Does tree shape matter?

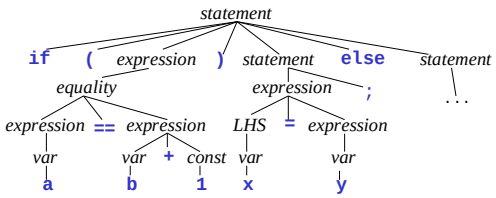
Expression Trees [Section 13.5]

- Programs have a hierarchical structure
  - All statements have a fixed form
  - Statements can be ordered and nested almost arbitrarily (nested `if-then-else`)
- Can use a structure known as a *syntax tree* to represent programs
  - Trees capture hierarchical structure

A Syntax Tree

Consider the C++ statement:

```
if ( a == b + 1 ) x = y; else ...
```



Syntax Trees

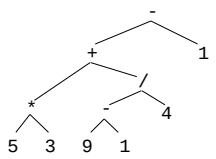
- Compilers usually use syntax trees when compiling programs
  - Can apply simple rules to check program for syntax errors (the "grammar" in Appendix K of textbook)
  - Easier for compiler to translate and optimize than text file
- Process of building a syntax tree is called *parsing*

Binary Expression Trees

- A *binary expression tree* is a syntax tree used to represent meaning of a mathematical expression
  - Normal mathematical operators like `+`, `-`, `*`, `/`
- Structure of tree defines result
- Easy to evaluate expressions from their binary expression tree

Example

```
5 * 3 + (9 - 1) / 4 - 1
```



Traversing Expression Trees

- Traverse in preorder for prefix notation

  `- + * 5 3 / - 9 1 4 1`
- Traverse in inorder for infix notation

  `5 * 3 + 9 - 1 / 4 - 1`
  - Note that operator precedence may be wrong without adding parentheses at every step

  `(((5*3) + ((9 - 1) / 4)) - 1)`
- Traverse in postorder for postfix notation

  `5 3 * 9 1 - 4 / + 1 -`

Evaluating Expressions

- Easy to evaluate an expression from its binary expression tree
  - Use a postorder traversal
  - Recursively evaluate the left and right subtrees and store those values
  - Apply operator at root to stored values
- Much like using a stack to evaluate postfix notation

Trees Summary

- Tree as new hierarchical data structure
  - Recursive definition and recursive data structure
- Tree parts and terminology
  - Made up of nodes
  - Root node, leaf nodes
  - Children, parents, ancestors, descendants
  - Depth of node, height of tree
  - Subtrees

Trees Summary (2)

- Binary Trees
  - Either 0, 1, or 2 children at any node
  - Recursive functions to manipulate them
- Binary Search Trees
  - Binary Trees with ordering invariant
  - Recursive BST search
  - Recursive `Insert`, `Delete` functions
  - `O(H)` operations, where `H` is height of tree
  - `O(log N)` for `N` nodes in balanced case
  - `O(N)` in worst case

Trees Summary (3)

- Tree Traversals
  - Preorder traversal
  - Postorder traversal
- Binary Tree Traversals
  - Inorder traversal
- Expression and Syntax Trees

3/10/99 520