CS 106X
Lecture 18: Trees

Wednesday, February 22, 2017

Programming Abstractions (Accelerated)
Winter 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Section 16.1
Today's Topics

• Logistics
  • Midterm Tomorrow!
  • Midterm will cover up to and including Linked Lists

• Introduction to Trees
Trees

We have already seen trees in the class in the form of decision trees!
Trees

You've coded trees for recursive assignments!

Random expansion from sentence.txt grammar for symbol "<s>"
Trees Can Describe Hierarchies

- World
  - USA
    - California
      - SF
      - LA
    - Ohio
      - Kent
  - Honduras
    - Tegucigalpa
  - China
    - Hunan
      - Changsha
    - Shanghai
Trees Can Describe Websites (HTML)
Trees Can Describe Programs

// Example student solution
function run() {
    // move then loop
    move();
    // the condition is fixed
    while (notFinished()) {
        if (isPathClear()) {
            move();
        } else {
            turnLeft();
        }
    } // redundant
    move();
}

* This is a figure in an academic paper written by a recent CS106 student!
What is a Tree (in Computer Science)?

- A tree is a collection of nodes, which can be empty. If it is not empty, there is a “root” node, \( r \), and zero or more non-empty subtrees, \( T_1, T_2, \ldots, T_k \), whose roots are connected by a directed edge from \( r \).
What is a Tree (in Computer Science)?

• A tree is a collection of nodes, which can be empty. If it is not empty, there is a “root” node, $r$, and zero or more non-empty subtrees, $T_1, T_2, \ldots, T_k$, whose roots are connected by a directed edge from $r$. 

Note: there are $N$ nodes and $N-1$ edges
What is a Tree (in Computer Science)?

- A tree is a collection of **nodes**, which can be empty. If it is not empty, there is a “root” node, \( r \), and **zero or more non-empty subtrees**, \( T_1, T_2, \ldots, T_k \), whose roots are connected by a directed edge from \( r \).
What is a Tree (in Computer Science)?

- A tree is a collection of nodes, which can be empty. If it is not empty, there is a “root” node, \( r \), and zero or more non-empty subtrees, \( T_1, T_2, \ldots, T_k \), whose roots are connected by a directed edge from \( r \).

**Tree Terminology**

- Nodes with the same parent are siblings.
We can define a path from a parent to its children.

The path A-E-J-O has a length of three (the number of edges).
The **depth** of a node is the length from the root. The depth of node J is 2. The depth of the root is 0.

The **height** of a node is the longest path from the node to a leaf. The height of node F is 1. The height of all leaves is 0.
The height of a tree is the height of the root (in this case, the height of the tree is 3.)
Tree Terminology

Trees can have only one parent, and cannot have cycles
Trees can have only one parent, and cannot have cycles.
Trees can have only one parent, and cannot have cycles.

Node A has two parents.
Trees can have only one parent, and cannot have cycles
Trees can have only one parent, and cannot have cycles.

Not a tree: the red edges make a cycle.
How can we build trees programmatically?
How can we build trees programmatically?

Binary Tree:

```
  value
  /
```

```
How can we build trees programmatically?

Binary Tree:

Linked List:

value
How can we build trees programmatically?

Binary Tree:

Linked List
How can we build trees programmatically?

Binary Tree:

value

value

value

Linked List

value
How can we build trees programmatically?

Binary Tree:

value

Linked List

value
The Most Important Slide

Binary Tree:

```c
struct Tree {
    string value;
    Tree *left;
    Tree *right;
};
```
We Can Have Ternary Trees (or any number, $n$)

Ternary Tree:

```
struct Tree {
    string value;
    Tree *left;
    Tree *middle;
    Tree *right;
};
```
We Can Have Ternary Trees (or any number, \( n \))

N-ary Tree:

```cpp
struct Tree {
    string value;
    Vector<Tree *> children;
};
```
Trees can be defined as either structs or classes.

```cpp
struct Tree {
    string value;
    Tree * left;
    Tree * right;
};

class Tree {
private:
    string value;
    Vector<Tree *> children;
};
```
Let's write some code to "traverse" the tree

```c
struct Tree {
    string value;
    Tree * left;
    Tree * right;
};
```

There are multiple ways to traverse the nodes in a binary tree:

1. Pre-order
2. In-order
3. Post-order
4. Level-order
Let's write some code to "traverse" the tree

```
struct Tree {
    string value;
    Tree * left;
    Tree * right;
};
```

There are multiple ways to traverse the nodes in a binary tree:

1. Pre-order
2. In-order
3. Post-order
4. Level-order

1. Do something
2. Go left
3. Go right

Image of a tree with nodes labeled "this", "is", "correctly", "written", and "sentence."
Let's write some code to "traverse" the tree

```c
struct Tree {
    string value;
    Tree * left;
    Tree * right;
};
```

There are multiple ways to traverse the nodes in a binary tree:

1. Pre-order
2. In-order
3. Post-order
4. Level-order

1. Go left
2. Do something
3. Go right
Let's write some code to "traverse" the tree

```c
struct Tree {
    string value;
    Tree * left;
    Tree * right;
};
```

There are multiple ways to traverse the nodes in a binary tree:

1. Pre-order
2. In-order
3. Post-order
4. Level-order

1. Go left
2. Go right
3. Do something
Let's write some code to "traverse" the tree

```c
struct Tree {
    string value;
    Tree * left;
    Tree * right;
};
```

There are multiple ways to traverse the nodes in a binary tree:

1. Pre-order
2. In-order
3. Post-order
4. Level-order

Hmm...can we do this recursively?
We want to print the levels: 0, 1, 2 from left-to-right order
struct Tree {
    string value;
    Tree * left;
    Tree * right;
};

There are multiple ways to traverse the nodes in a binary tree:

1. Pre-order
2. In-order
3. Post-order
4. Level-order

Not easy recursively...let's use a queue!

1. Enqueue root
2. While queue is not empty:
   a. dequeue node
   b. do something with node
   c. enqueue left child of node if it exists
   d. enqueue right child of node if it exists

should look familiar...word ladder?
struct Tree {
    string value;
    Tree * left;
    Tree * right;
};

void preOrder(Tree * tree) {
    if(tree == NULL) return;
    cout<< tree->value <<" ";
    preOrder(tree->left);
    preOrder(tree->right);
}

void inOrder(Tree * tree) {
    if(tree == NULL) return;
    inOrder(tree->left);
    cout<< tree->value <<" ";
    inOrder(tree->right);
}

void postOrder(Tree * tree) {
    if(tree == NULL) return;
    postOrder(tree->left);
    postOrder(tree->right);
    cout<< tree->value << " ";
}

void levelOrder(Tree *tree) {
    Queue<Tree *>treeQueue;
    treeQueue.enqueue(tree);
    while (!treeQueue.isEmpty()) {
        Tree *node = treeQueue.dequeue();
        cout << node->value << " ";
        if (node->left != NULL) {
            treeQueue.enqueue(node->left);
        }
        if (node->right != NULL) {
            treeQueue.enqueue(node->right);
        }
    }
}
References and Advanced Reading

• References:

• Advanced Reading:
  
  • Great set of tree-type questions: