Lecture 9

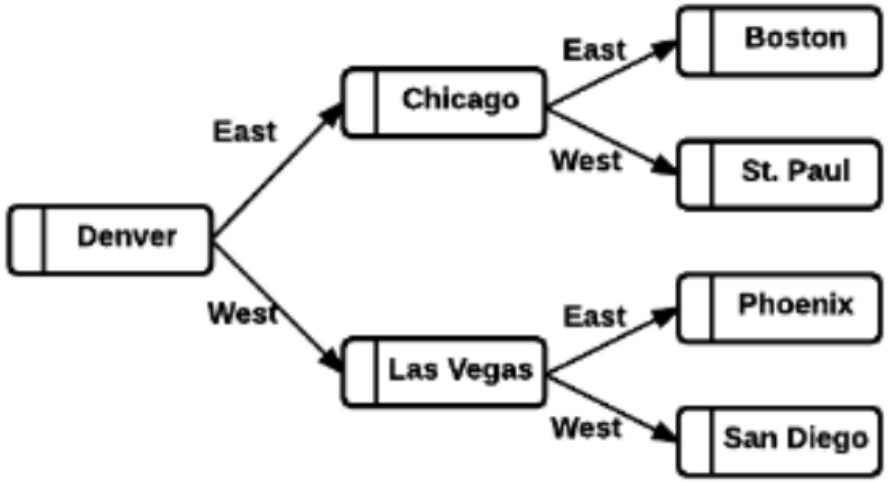
Christopher Godley

CSCI 2270 Data Structures

June 12th, 2018

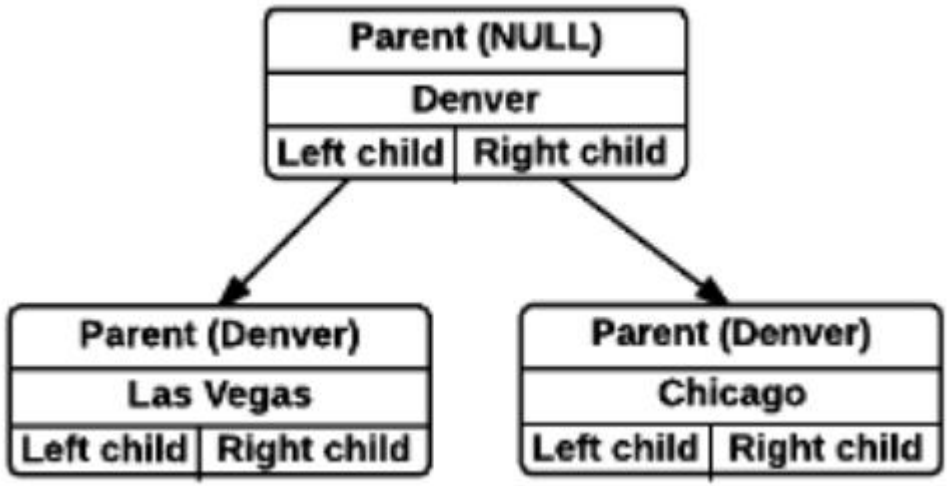
Binary Trees

- Imagine a transportation network where each city connects to only two other cities
- Starting from the **root**, you have two **children** cities to travel to



Binary Trees

- In the example below, each city has a
  - **Parent** pointer
  - City name
  - **Left child** pointer
  - **Right child** pointer



Binary Trees

- Singly Linked Lists:
  - Next pointer
- Doubly Linked Lists:
  - Next pointer
  - Previous pointer
- Binary Trees
  - Parent pointer
  - Left child
  - Right child

Binary Trees

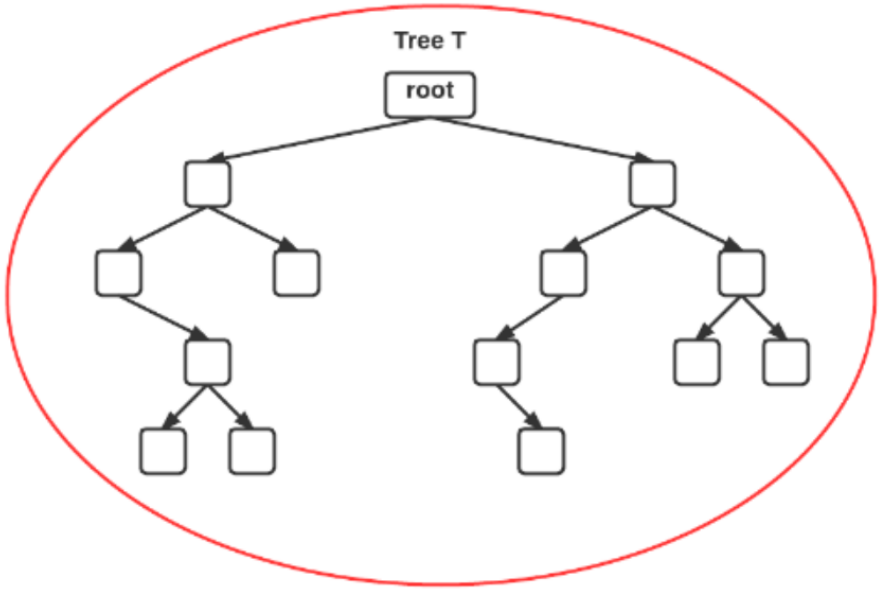
- Parent Node
  - Each node in the tree has a **parent**
  - Each node in the tree is a **parent** for *at most* two children
- Root Node
  - Topmost node
  - **Parent** pointer of **root** is **NULL**

Binary Trees

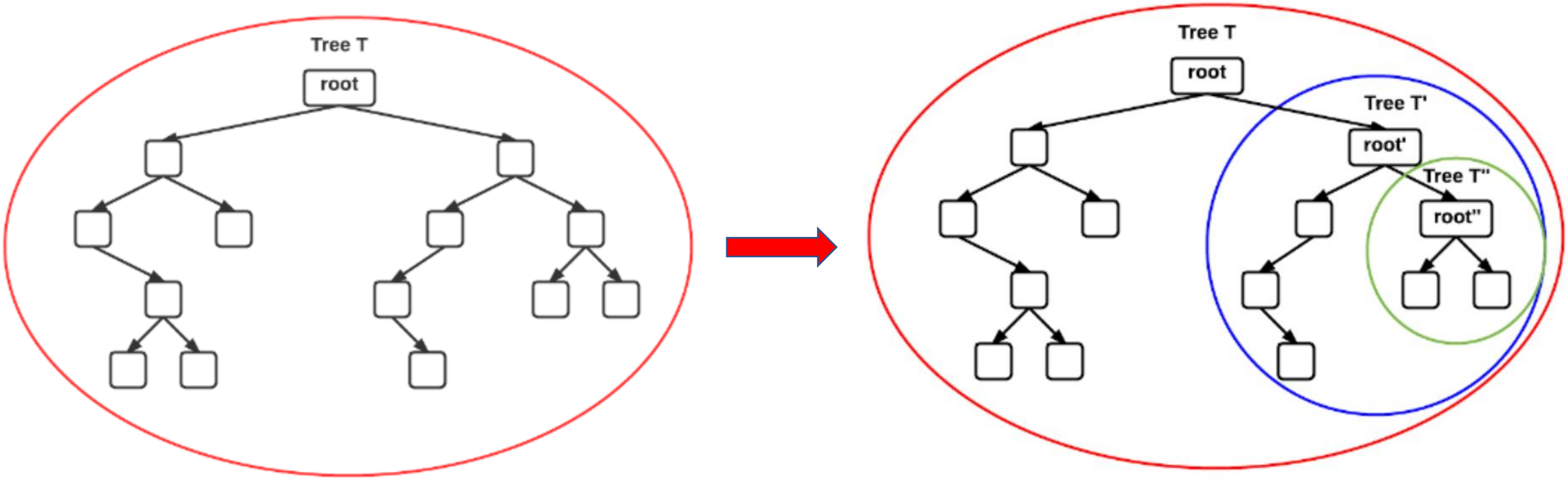
- Node Properties
  - Each node has a **key** that identifies it
  - If a node doesn’t have a left child, it’s **left child** is NULL
  - If a node doesn’t have a right child, it’s **right child** is NULL
  - If a node has no children, it’s called a **leaf**

Binary Trees

- Any binary tree may be split or separated into smaller sub-trees
- We refer to this characteristic as **self-similarity**
  - This characteristic can be taken advantage of for elegant ways to search the tree as smaller sub-trees



Binary Trees

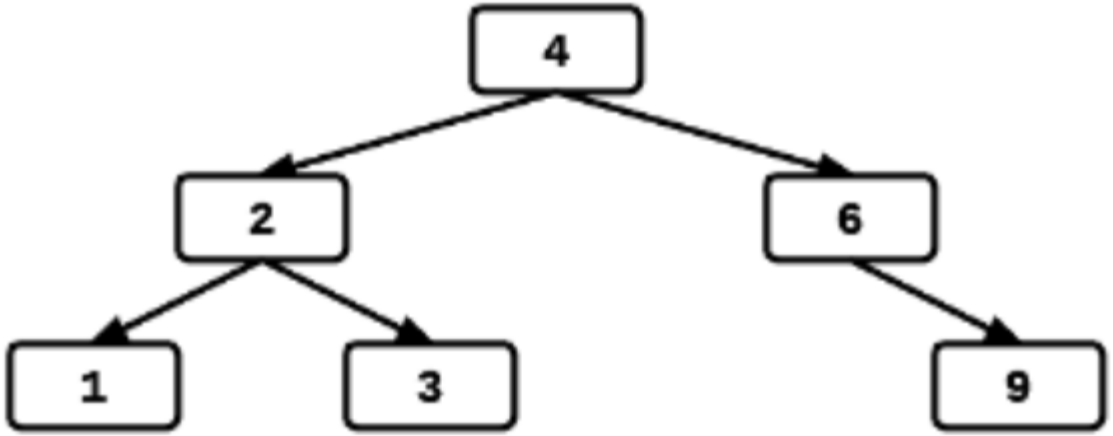


Binary Search Trees

- A binary search tree is a special tree in which the data is ordered
- Imagine a tree containing numbers
  - Each child left of a parent node must contain values less than the parent
  - Each child right of the parent must contain values greater than the parent

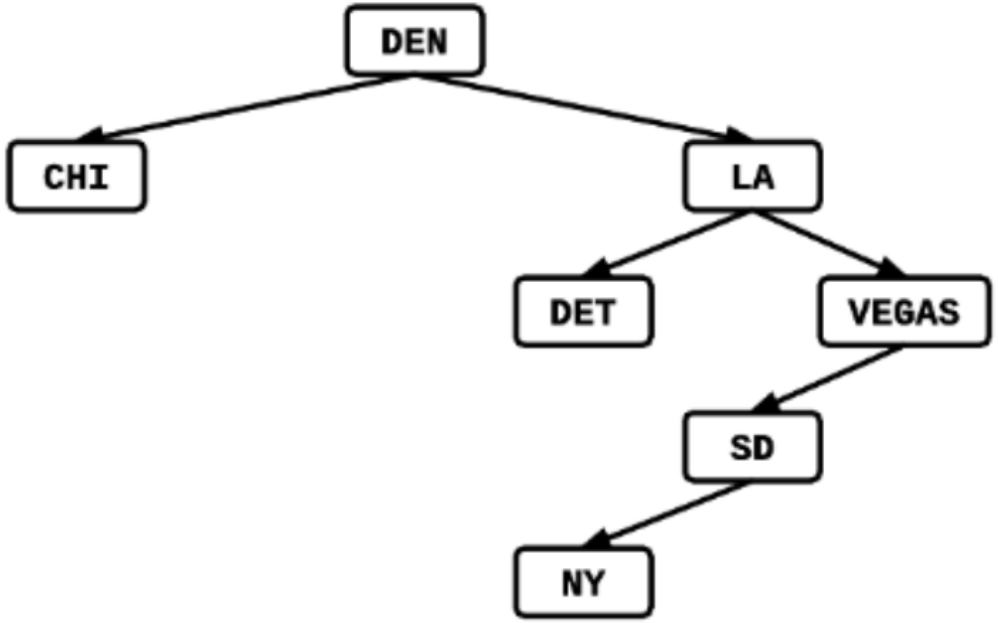
Binary Search Trees

- Let’s build a tree from the following:
  - Each element is entered in the order it is observed
  - The elements are <4, 2, 6, 9, 1, 3>



Binary Search Trees

- What if we build a BST using the following
  - Each element is entered in the order it is observed
  - The elements are <DEN, LA, CHI, VEGAS, SD, DET, NY>



Binary Search Tree: ADT

BinarySearchTree

private:

root

searchRecursive(node, value)

public:

Init()

insert(value)

search(value)

traverseAndPrint()

delete(value)

deleteTree()

```
struct node{
        int key
        node* parent
        node* leftChild
        node* rightChild
}
```

Binary Search Tree: Traversal

- We can use recursion to implement an in-order tree traversal

```
printNode(node)
        if (node->leftChild != NULL)
                printNode(node->leftChild)
        print(node->key)
        if (node->rightChild != NULL)
                printNode(node->rightChild)
```

Binary Search Trees: Delete

- Deleting a node may require replacing the deleted node with one of it’s children
- The following three cases must be considered:
  - The node has no children
  - The node has one child
  - The node has two children

Binary Search Trees: Delete

- Node has no children
  - Update the parent to point to NULL
  - Delete the node
- Node has one child
  - Update the parent to point to the child of the node
  - Delete the node

Binary Search Trees: Delete

- The node has two children:
  - We'll need to replace the deleted node with the minimum value of it's right sub-tree
  - For that we'll use another function to help us

Binary Search Trees: treeMinimum(node)

- Used to find the minimum value in a binary search tree or any of its sub-trees.

```
treeMinimum(node)
      while (node->leftChild != NULL)
            node = node->leftChild
      return node
```

- How could we find the maximum value?