CSCI 2270 - Data Structures and Algorithms
Instructor-Hoenigman/Jacobson
Recitation 7

In this recitation, you will be learning about Binary Trees and Binary Search trees, Recursion, and how to search these trees accordingly.

Before moving on to trees, here's a quick recap of stacks and queues:

A **queue** is a data structure that stores a collection of elements and restricts which element can be accessed at any time. Queues follow the **first-in-first-out (FIFO)** order - the first element added to the queue is the first element removed from the queue, much like the line at the grocery store.

A **stack** is an abstract data structure that stores a collection of elements and restricts which element can be accessed at any time. Stacks work on a **last in first out principle (LIFO)**: the last element added to the stack is the first item removed from the stack, much like a stack of cafeteria plates. Elements are added to the top of the stack, and the element on the top is the only element that can be removed. Since they are stacked on top of one another, you can't access one further down the stack until the others on top of it are removed.

Binary Trees

A binary tree is made of nodes, where each node contains a left reference (child), a right reference (child), and a data element. Since each node in the tree can be connected to at most two nodes through the left and right reference, it is called a binary tree. The topmost node in the tree is called the root.

Binary trees are similar to linked lists in that the nodes in the tree can be created dynamically and then linked together to create a structure that can be easily modified to support dynamic data. The next and previous pointers of a doubly linked list are replaced with parent and left and right child pointers in binary trees to make it possible to represent a hierarchical structure in a data set, which is one advantage that trees have over linked lists. Trees can also be searched and modified with minimal computational effort. These advantages mean that binary trees are extremely useful and frequently used over other data structures.

Pointers in a doubly linked list:
• next
• previous

Pointers in a binary tree:
• parent
• left child

- right child

Node Properties of Binary Trees:

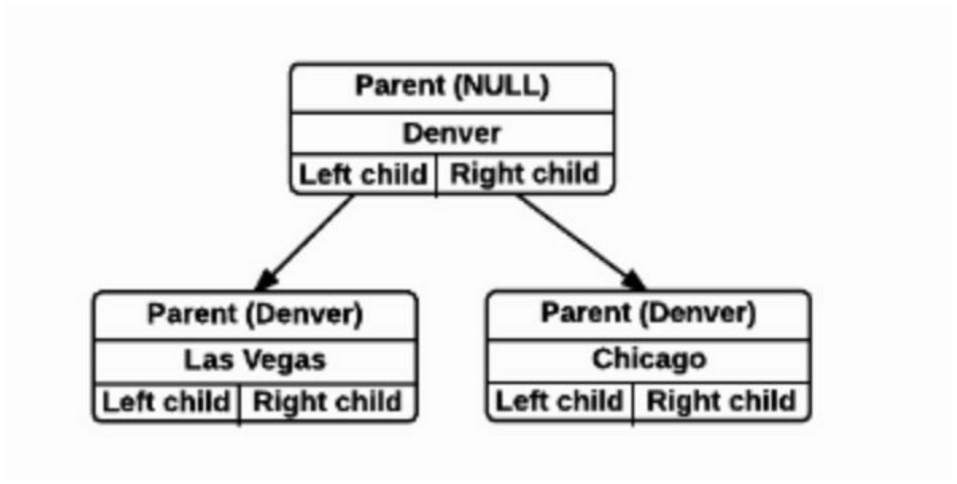
All nodes in the tree can be a parent or a child to other nodes (except for the root, it can never be a child to another node). There are general properties that all nodes exhibit, as well as properties that nodes exhibit as parent nodes and as the root node of the tree.

Root node properties
- The topmost node in the tree is called the root.
- The parent of the root is NULL.

Parent node properties
- Each node in the tree has a parent.
- Each node in the tree is a parent node for at most two children, a left and a right child.

Node properties
- Each node in the tree has a key that identifies it.
- If a node doesn't have a left child, then its left child property is NULL.
- If a node doesn't have a right child, then its right child property is NULL.
- If a node doesn't have a left or a right child, then it is a leaf node.

Example:



Nodes in a binary tree with properties for the parent, key, left child, and right child. The root of the tree is Denver, and Denver has a left child, Las Vegas, and a right child, Chicago. The parent of Las Vegas and Chicago is Denver. The key for each node is the name of the city.

Implementation of a Binary Tree node

A binary tree node in C/C++ can be built with a struct, just like a node in a linked list, where the members of the struct include the key, a pointer to the parent node, pointers to the leftChild and rightChild nodes, and any additional data that the program needs to store to operate successfully.

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

Recursion[1]

Recursion is the process of a function calling itself, and it is frequently used to evaluate structures that can be defined by self-similarity, such as trees. A recursive call to a function evaluates a smaller and smaller instance of the structure until the smallest case is reached.

Recursion is typically used on problems where the structure of the data is also recursive, such as a file system on a computer. The directory structure is defined recursively in terms of smaller and smaller directory structures. At the top level, there are directories and files. Within the directories, there are other directories and files, and within those directories there can be other directories, and so on. Searching through the file system reveals a repeating pattern down to a level where there are only files. In a recursive search algorithm, if the search gets to this level and doesn't find the specified file, then the algorithm returns that the file is not found in the file system.

Rules for recursive algorithms

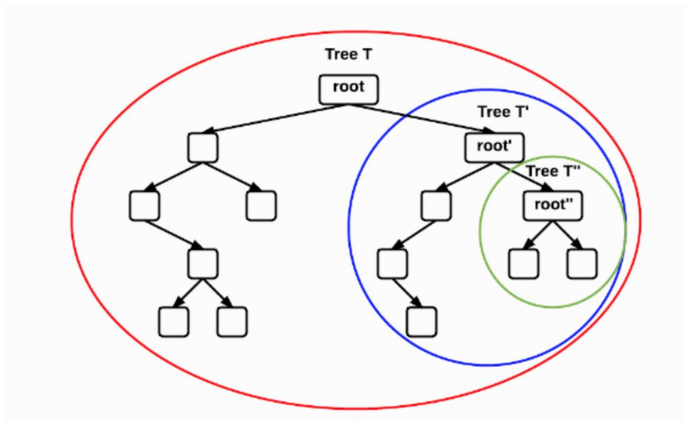
There are two rules that define the structure of any recursive algorithm. The algorithm needs to include:

• A base case. This is the smallest unit of the problem that can be defined. Once the base case is reached, the algorithm should return without additional recursive calls. The base case is defined by the structure of the data. In a file system, the base case is an individual file. Traversal of a directory structure down to the base case means going down to a level where there are only files and no additional directories. In a BT, the base case is an individual node with no children. Smaller and smaller sub-trees can be evaluated until a sub-tree is reached that is a single node.

• A set of rules that can reduce all cases down to the base case. The base case is the exit strategy for a recursive algorithm. If the algorithm never reaches the base case, then it will never exit.

[1] Excerpt From: Rhonda Hoenigman. “Visualizing Data Structures.” iBooks.

For example, a function can be called multiple times to evaluate each of the trees or subtrees.



Traversing a Binary Tree

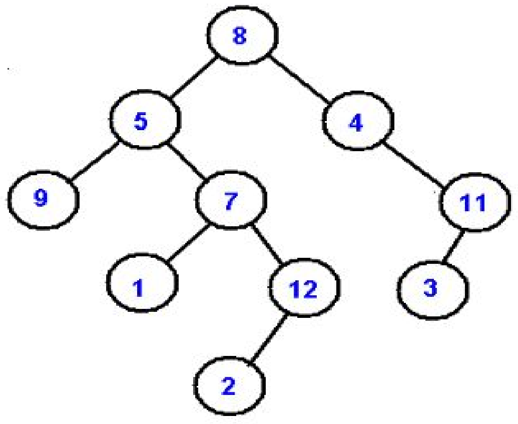
A traversal is a process that visits all the nodes in the tree. In general the traversal algorithms can be classified as:

- depth-first traversal
- breadth-first traversal

There are three different types of depth-first traversals, :

- PreOrder traversal - parent first and then left and right children; (PARENT, LEFT, RIGHT)
- InOrder traversal - left child, then the parent and the right child; (LEFT, PARENT, RIGHT)
- PostOrder traversal -left child, then the right child and then the parent; (LEFT, RIGHT, PARENT)

There is only one kind of breadth-first traversal -- the level order traversal. This traversal algorithm visits nodes by levels from top to bottom and from left to right.



Consider the following tree:

PreOrder - 8, 5, 9, 7,1, 12, 2, 4, 11, 3
InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8

LevelOrder - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2

Searching In a Binary Tree:

Using inorder traversal

We search for the node recursively but in an inorder fashion.
Order follows - LEFT, PARENT, RIGHT

```
node* searchInorder(int value, node* current)
        if(current->left != NULL)
                searchInorder(value, current->left)
        if(current->key == value)
                return current;
        if(currentnode->right != NULL)
                searchInorder(value, current->right)
```

Using preorder traversal

We search for the node recursively but in an inorder fashion.
Order follows- PARENT, LEFT ,RIGHT

```
node* searchPreorder(int value, node* current)
        if(current->key == value)
                return current
        if(current->left != NULL)
                searchPreorder(value, current->left)
        if(currentnode->right != NULL)
                searchPreorder(value, current->right)
```

Using postorder traversal

We search for the node recursively but in an inorder fashion.
Order follows-LEFT, RIGHT, PARENT

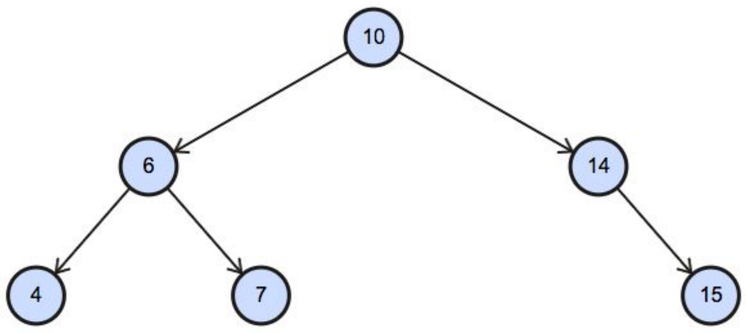
```
node* searchPostorder(int value, node* current)
        if(current->left != NULL)
                searchPostorder(value, current->left)
        if(currentnode->right!=NULL)
                searchPostorder(value, current->right)
        if(current->key == value)
                return current
```

In class work

Use what you learned in recitation to write the nodes visited after each function:

- node* searchPreorder(int value, node* current)
- node* searchPostorder(int value, node*current)
- node* searchInorder(int value, node* current)

on the root of following binary search tree



Show to your TA the visited nodes in the form of arrays, i. e. write down the arrays:

- int inOrderArray[6];
- int postOrderArray[6];
- int preOrderArray[6];

Recitation 7 Conceptual Question

There is a link on Moodle to a Recitation 7 Programming exercise. In the quiz, you are asked to select the correct answer for a multiple choice question. You will walk through pseudocode for printing a Binary tree in an Inorder traversal. You have until Sunday at 5pm to complete the exercise.

Note: You will only be able to attempt the quiz 2 times maximum. All other attempts will be discarded.