Data Structures in C

Prof. Georg Feil

Trees

Winter 2018

Acknowledgement

- These lecture slides are based on slides by Professor Simon Hood
- Additional sources are cited separately

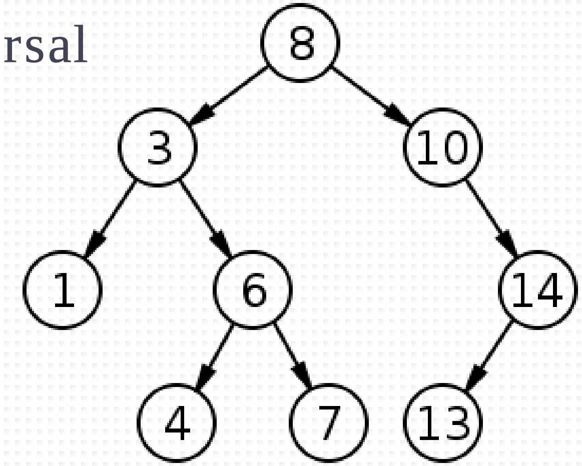
Reading Assignment (required)

- Read Data Structures (recommended textbook)
  - Chapter 9 sections 9.1 – 9.5

Note the textbook does a few things
we might consider poor style, for example
one-letter variable names and using int
for Boolean values.

Trees

- Next we'll learn about a data structure called a tree
  - General trees
  - Pre-order, post-order, and in-order traversal
  - Binary Trees
    - Building
    - Traversing
    - Counting Nodes



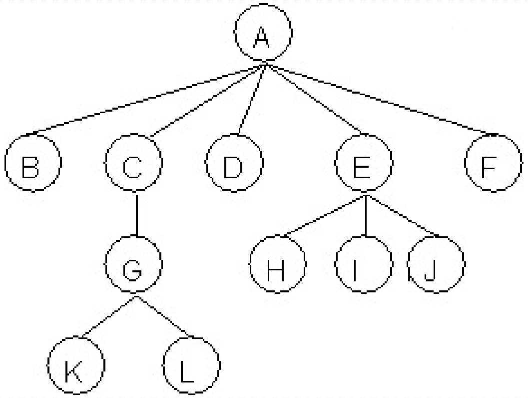
- In principle a tree is not that different from a linked list
  - Imagine a linked list with more than one next pointer

Tree definition

- Formally, a tree is a set of nodes such that:

1. There is one specially designated node called the root of the tree
2. The remaining nodes are partitioned into m >= 0 disjoint sets T1, T2, …, Tm
3. Each of these sets (T1, T2, etc.) is also a tree

Tree definition



In other words…

- There is a root node
- All other nodes branch from the root node
- The branches can, in turn, branch again
- By convention, trees are drawn with the root at the top
- Normally each node stores some information
  - The tree shown above has letters of the alphabet at each node

Tree terminology

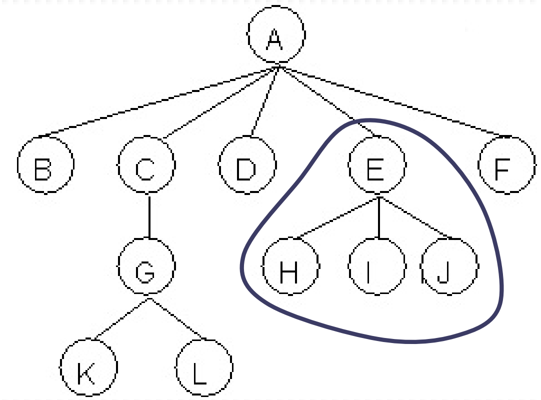
- We use the terms parent, child, and sibling to refer to the nodes of a tree, but you may also see it written as branches, leaves, levels, and others
- For our structure to be a “tree” each node can only have one line leading in – though they can have any number leading out
  - Except for the root, which has no lines leading into it
- We’ll discuss graphs soon
  - A graph is more general than a tree, so many of the rules go away
  - A node of a graph can have two or more parent nodes, for example

Tree terminology

- The degree of a node is the number of subtrees that node has
  - We can think of it as the number of lines leaving the node
- The depth of a node is the number of branches which must be traversed on the path from the root to the node
  - How many lines to follow to reach that point
- The height of a tree is the number of levels in the tree
  - The height is one more than the tree's largest depth
- the depth (also called level) is concerned with the lines, while the height is concerned with the nodes

Tree terminology

- A subtree is a part of the tree starting at a node that's not the root
  - That node is the root of the subtree



- If the relative order of the subtrees is important, the tree is called an ordered tree
- On the other hand if the order is unimportant, the tree is called oriented
- The most important type of tree for us is the Binary Tree

Binary Trees

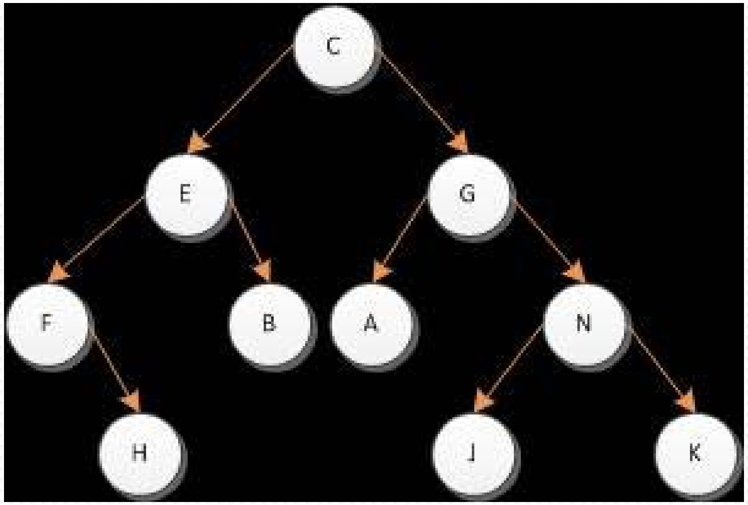
- A binary tree is a tree that has at most two children per node (a left and a right)
- Note that the order of the left child and right child will be important
- If there are two children for every node except the terminals ("leaves"), and the terminals have no children, the tree is complete

Traversing a binary tree

- Traversing a tree means to "visit" all the nodes in some order
- There are three common ways to traverse a binary tree
  - Pre-order
  - In-order
  - Post-order
- We can use different traversals to interpret the tree's data in a different order, or to search for a specific data item in a different way

Pre-order

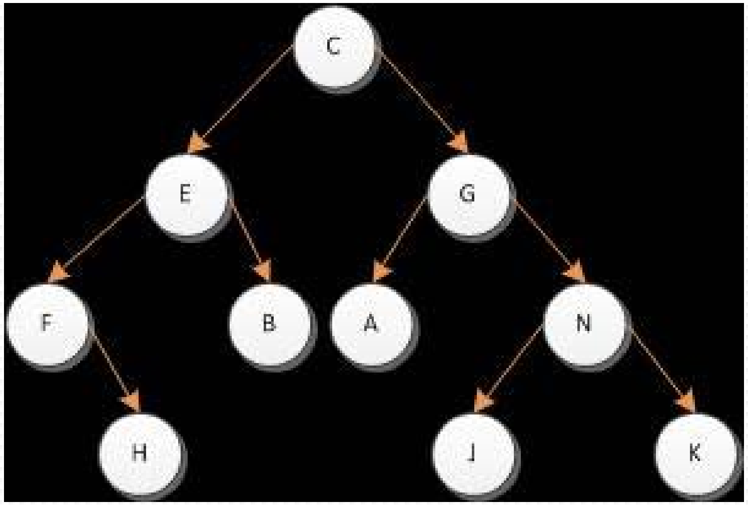
- Suppose we have the following tree



- Pre-order traversal says visit the root, then the left subtree (in pre-order), then the right subtree (in pre-order)
- The pre-order sequence is C E F H B G A N J K

In-order

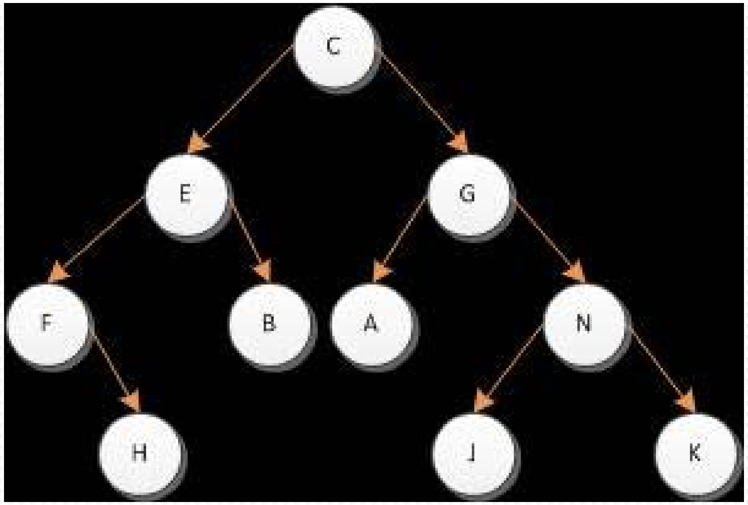
- Given the same tree, we can traverse it using in-order



- In-order traversal says visit the left subtree (in-order), root, then the right subtree (in-order)
- The in-order sequence is F H E B C A G J N K

Post-order

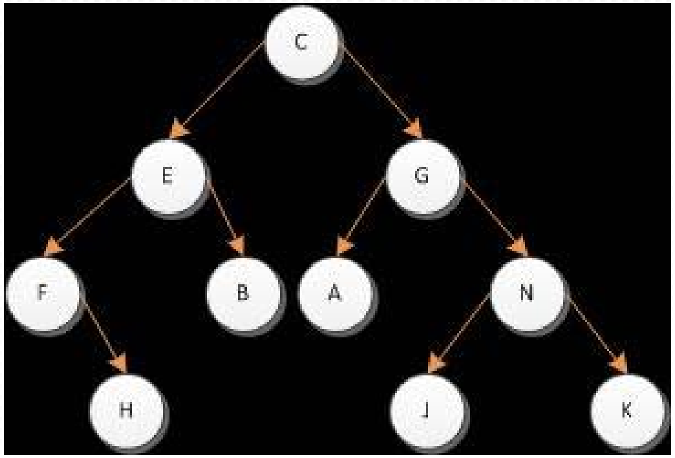
- We can also traverse the tree using post-order



- Post-order traversal says visit the left subtree (post-order), then the right subtree (post-order), then the root
- The post-order sequence is H F B E A J K N G C

Other types of traversal

- Pre-order, in-order, and post-order are all considered types of depth-first traversal
  - Because the traversal tends to move "down" the tree before it moves across
- There is also breadth-first traversal, illustrated below

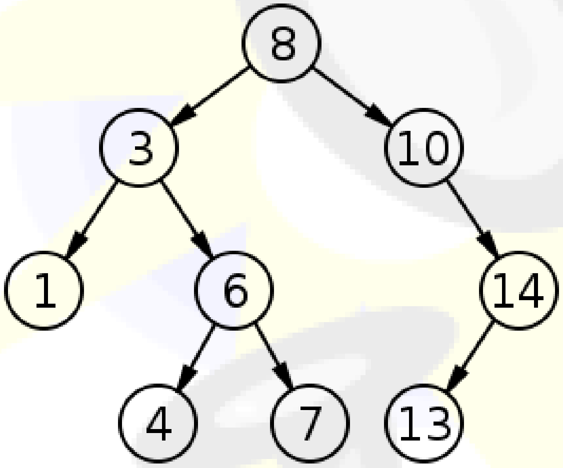


Breadth-first sequence:

C E G F B A N H J K

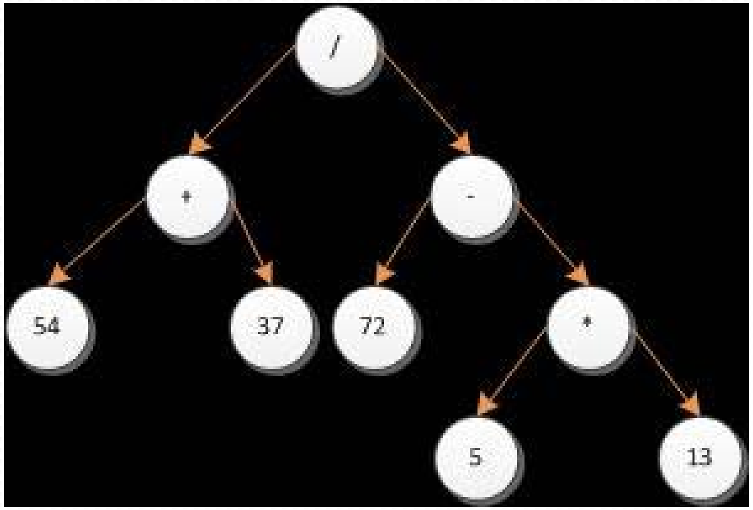
Exercise 1

- For the binary tree shown below, write out the tree's data using pre-order, in-order, and post-order



Math using trees

- If we want to calculate (54 + 37) / (72 –(5 * 13)) we can represent this as a tree



- The pre-order traversal is / + 54 37 –72 * 5 13
- The in-order traversal is 54 + 37 / 72 –5 * 13
- The post-order traversal is 54 37 + 72 5 13 * -/

Math using trees

- Interestingly, a computer can perform the mathematical operations in post-order more easily than our regular in-order!

→ Create an empty stack

→ Using post-order traversal, push numbers (operands) onto the stack

→ If an operator is reached in post-order traversal, pop off the top two numbers, perform the operation on them, and push the result back on the stack

- This is also called a postfix expression, and is used by some calculators

Exercise 2

- Draw a stack
- Use the post-order traversal described on the previous slide to calculate (54 + 37) / (72 – (5 * 13))
- The post-order traversal is 54 37 + 72 5 13 * – /

Binary Tree Implementation

Binary tree implementation

- We can represent a binary tree using a structure that's very similar to a linked list node structure

```
typedef struct treenode {
    NodeData data;  // struct inside a struct
    struct treenode* left;
    struct treenode* right;
} TreeNode;
```

- Notice our data is stored in another struct—this will allow us to recycle our code by changing NodeData as needed
- Also notice there are two "next" pointers (that's why it's a binary tree)

Binary tree implementation

- We might decide we just want our tree to store an int

```
typedef struct  {
    int num;
} NodeData;
```

- … or, we could store a word and associated frequency of that word

```
typedef struct{
    char word[MAX_WORD_SIZE];
    int freq;
} NodeData;
```

Binary tree implementation

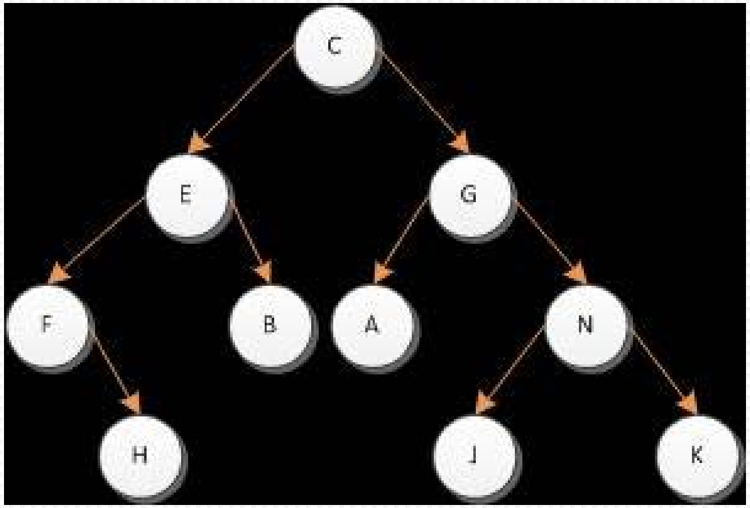
- And, of course, we need to know the root of the tree!
- Once we have access to the root, we have access to all the nodes via the left and right pointers

```
typedef struct {
    TreeNode* root;
} BinaryTree;
```

- Even though this structure is really just a node pointer we like to think of it as representing the whole tree

Building a binary tree

- To create the binary tree shown below, we can supply data as a list of characters or strings
- C E F @ H @ @ B @ @ G A @ @ N J @ @ K @ @
- The @ represents an empty child node, we'll replace it with NULL in our structure



Building a binary tree

- Note that we'll use the following definition of NodeData

```
typedef struct{
    char word[MAX_WORD_SIZE];
} NodeData;
```

- ... and that each node of our tree stores a NodeData structure as a variable (structure field) called data

Building a binary tree

- This recursive function builds a tree from a text file containing a list of data specified by a FILE pointer (handle)

```
TreeNode* buildTree(FILE* in) {
    char str[MAX_WORD_SIZE];
    int cnt = fscanf(in, "%s", str);
    if (cnt != 1 || strcmp(str, "@") == 0)
        return NULL;
    TreeNode* p = (TreeNode*) malloc(sizeof(TreeNode));
    strcpy(p->data.word, str);  // data not a ptr, use .
    p->left = buildTree(in);
    p->right = buildTree(in);
    return p;
}
```

Building a binary tree

- We can now create a binary tree, bt, with the following code

```
FILE* in = fopen("btree.txt", "r");
BinaryTree bt;
bt.root= buildTree(in);
```

- With our tree built, let's see how we can traverse it
- Let's define a function called visit() that does the action to perform at each node (e.g. print the value in the node)

```
void visit(TreeNode* node) {
    printf("%s ", node->data.word);
}
```

Traversing a binary tree

- We can visit each node in our tree using code like shown below
  - This is pre-order traversal

```
void preOrder(TreeNode* node) {
    if (node != NULL) {
        visit(node); // Process node, e.g print
        preOrder(node->left);
        preOrder(node->right);
    }
}
```

Exercise 3 – Building a Binary Tree

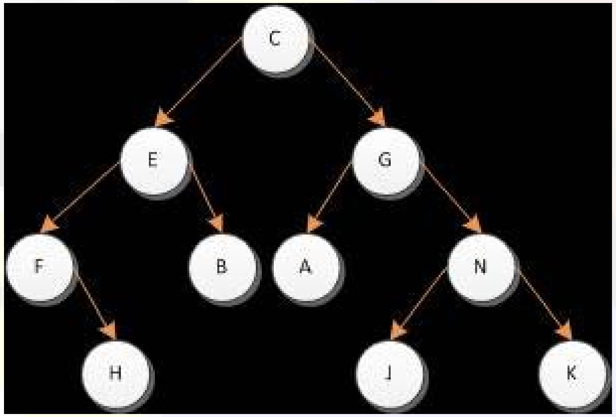
- Implement the binary tree described in the preceding slides
  - Use a separate C module (.c file) for your tree implementation
- Your main function should build a tree using the input data on the next slide
- After building, traverse the tree to print it out using the preOrder() function

Exercise 3 – Building a Binary Tree (cont'd)

- Use the following data as your input file. Notice each word starts with the same letter as our example tree.

Cat
Elephant
Fox
@
Hyena
@
@
Bullfrog
@
@
Giraffe

Aardvark
@
@
Narwhal
Jaguar
@
@
Kangaroo
@
@



Exercise 4

- Starting with your program from Exercise 3, add functions inOrder() and postOrder() that traverse the tree in the other two common orders
  - Hint: Just change the order of the lines from preOrder()
- Call all three traversal functions and examine the output to see the differences

Counting tree nodes

- We can count all the nodes in a binary tree with the following recursive function

```
int numNodes(TreeNode* root) {
    if (root == NULL)
        return 0;
    return 1 + numNodes(root->left)
             + numNodes(root->right);
}
```

Exercise 5

- Implement the function to count nodes in your program from Exercise 4
  - Call it from the main function and print out the result
- Note that you can also use the existing preOrder/inOrder/postOrder functions to count nodes, just by changing the visit() function
  - Will the three functions then give you different numbers, or the same number?
- Write a function called visit2() that counts nodes and use it to count the tree nodes in your program