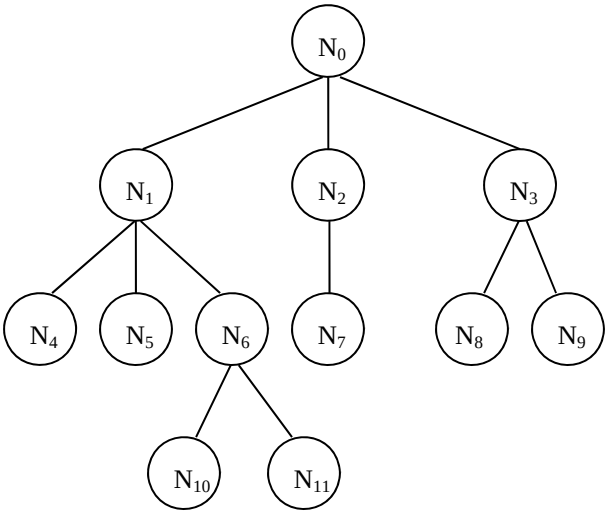
Lecture Notes 15 - Trees
CSS 501 – Data Structures and Object-Oriented Programming

Reading:

Carrano 5th Edition, Chapter 10.1-10.2
Carrano 6th Edition, Chapter 15.1-15.2

Introduction to trees

The data structures we have seen so far to implement abstract data types (arrays, linked lists) maintain the objects in some order. They were **linear**, in the sense that, for every item, there is an item before it and an item after it (unless it is at the beginning or end.) The **tree** data structure also stores objects, but it is not linear. The structure branches like a tree (hence the name). Tree structures in computer science are **usually drawn upside-down** from what a tree usually looks like in nature. Example tree:



These have **many applications**, some of which we will discuss. The most obvious is storing hierarchical structures, such as genealogies, file structures, and organizational structures.

The **root** (which corresponds to a single data element) is at the top, and the **branches** are arranged below the root and always proceed **downward (never sideways). Each branch connects two nodes** (also called **vertices**). Each node stores (usually) one data element of the tree. The branches between nodes are sometimes called ***edges***. Nodes that don't have any further branches attached below them are called ***leaves*** (and they are typically at the bottom of this type of tree).

If there is a branch (or edge) linking node *p* to node *c* and *p* is above *c*, then *p* is said to be the **parent** of *c*, and *c* is the **child** of *p*. If *c* and *d* are both children of *p*, then they are said to be **siblings**. This can be generalized to the concept of **ancestor** and **descendant** by allowing multiple links (always in the same direction – up or down) between nodes. The root is an ancestor of all of the other nodes in the tree. Every node together with its descendants forms another tree (called a **subtree**).

The ***depth*** of a node in a tree is the number of branches to get to the root. In other words, the root is at level zero. Each other node has a level that is one greater than the level of its parent. The **height** of a tree is the maximum level of any node in the tree. According to this definition, an empty tree would have height -1. An alternative definition would place the root at level 1 and adjust every other level accordingly. You should be able to work with either definition.

Trees can be defined using a **recursive definition**.

T is a tree if:

T is empty, or

T has a node and zero or more non-empty subtrees (each of which is also a tree.)

The important implications are:

- There are no *cycles* in the tree. You can't get from a node back to the same node by following branches unless you follow the same branch twice.
- Each child has a single parent.

General tree implementation

Trees are implemented with nodes that store a data value and pointers to children and/or sibling nodes. While this is not necessary, we will use pointers to the node data in the class. This has advantages and disadvantages. Some advantages:

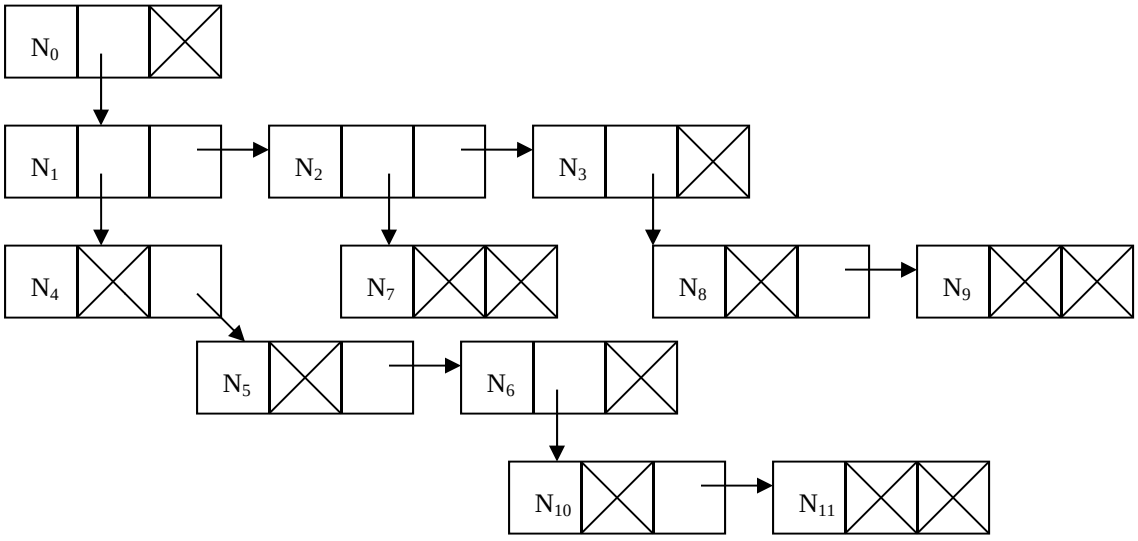
- You don't need necessarily need to copy the data to store it in the tree.
- Pointers are necessary with abstract class items (as we will see when we discuss inheritance.)
- You can use inheritance instead of templates to implement a general ADT.
- This technique is commonly used in industry.

One disadvantage is the need to deal with additional dynamic memory, including confusion over "ownership" of the memory. Memory leaks are never good! Another disadvantage is the possibility of privacy leaks (since someone else has a pointer to the data stored). However, we will use this technique for data structures in the remainder of this course, so it will be important to make sure that memory is always managed appropriately.

In a **general tree**, each node can have **any number of children**. We could use linked-lists to store the children for each node, but there is a better way that stores two pointers for each node that is called the **first-child, next-sibling** representation.

```
struct GeneralTreeNode {
        Object *item;
        GeneralTreeNode *firstChild;
        GeneralTreeNode *nextSibling;
};
```

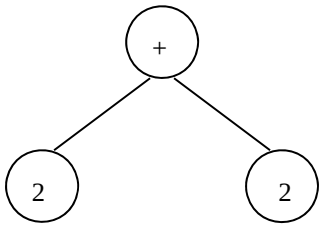
Using this representation, our previous tree would look like:



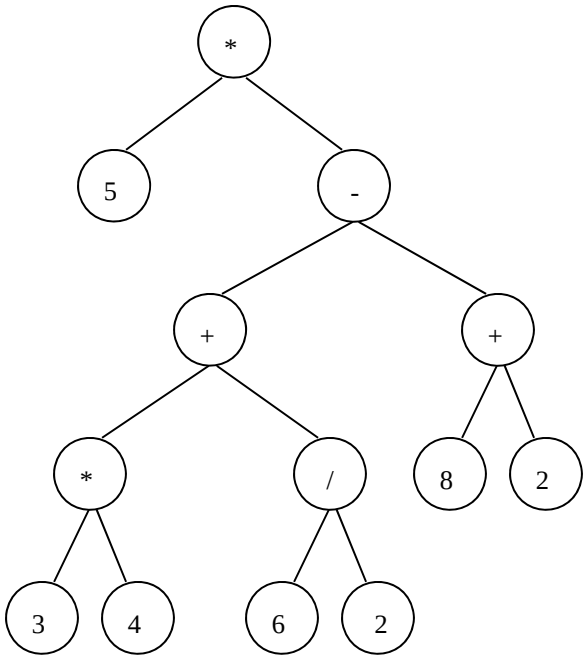
Binary trees

Usually, we will be concerned with binary trees. In this case, each node has two subtrees, either of which can be empty.

One application for a binary tree is to represent a mathematical expression, for example, when a compiler is parsing a program. Each expression containing binary (and unary) operators can be represented as a binary tree. Each operator (+, -, *, /, etc.) is a node of the tree with children representing the operands (which may be expression trees themselves.) A simple expression tree for 2+2 would be:



Here is the tree that represents the expression: 5 * ((3 * 4) + (6 / 2) – (8 + 2)).



Binary trees - pointer implementation

For a binary tree, nodes look like this:

```
struct BinaryTreeNode {
        Object *item;
        BinaryTreeNode *leftChild;
        BinaryTreeNode *rightChild;
};
```

This is straightforward to implement and the diagram looks just like the tree. Your first programming assignment in CSS 502 will include a binary tree implementation.

Tree traversal

Let's say that you have a tree that has already been built and you want to print out all of the elements in the tree. What order should they be printed in? There are **three orders** that are commonly used when traversing a tree:

- **preorder** – each node is visited (printed, in this case) before either of the children, then the left subtree and right subtree are traversed (in that order)
- **inorder** – the left subtree is traversed, then the node is visited, then the right subtree is traversed
- **postorder** – each node is visited after both the left subtree and the right subtree have been printed (in that order)

The most commonly used is inorder. However, preorder and postorder are useful in certain circumstances, as well.

The **general form** a traversal algorithm looks like this:

```
traverse(BinaryTreeNode *node)
{
        if (node == NULL) return;
        preorderVisit(node->item);              // Use this for preorder traversal.
        traverse(node->leftChild);
        inorderVisit(node->item);               // Use this for inorder traversal.
        traverse(node->rightChild);
        postorderVisit(node->item);             // Use this for postorder traversal.
}
```

What is the **efficiency** of traversing a tree with n nodes? Each node is visited exactly once and the amount of work done per visit is usually O(1), so the total traversal is O(n).

For an expression tree, the three types of traversal correspond to outputting the expression in prefix, infix, and postfix notation. Prefix and postfix notation (also known as Polish and reverse Polish notation) are useful, since they never require parentheses to denote the order of the operations. Traversing our tree for the expression above yields:

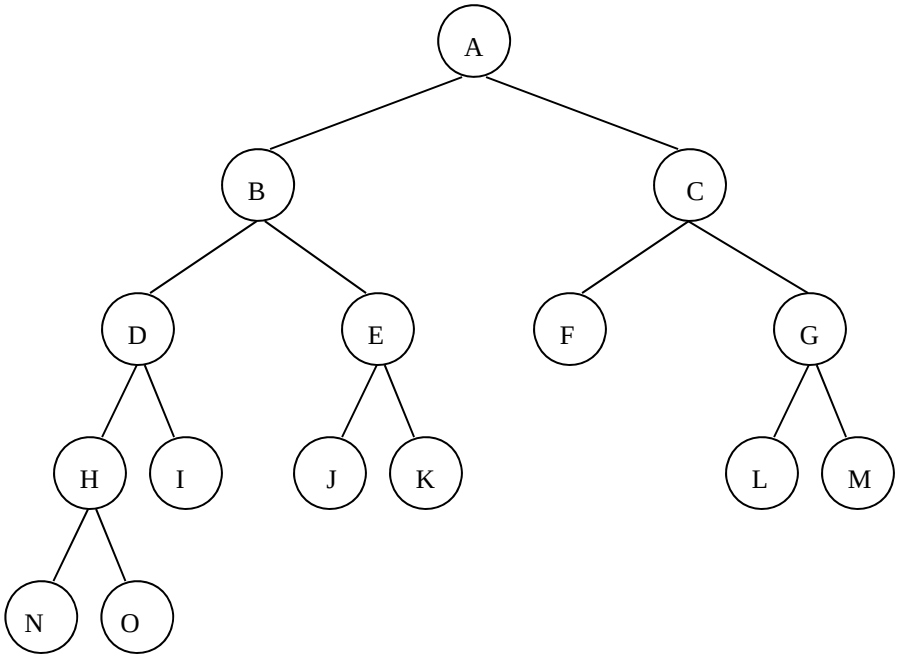
Prefix: * 5 - + * 3 4 / 6 2 + 8 2
Postfix: 5 3 4 * 6 2 / + 8 2 + - *
Infix: (5 * (((3 * 4) + (6 / 2)) – (8 + 2)))

Infix notation is ugly, since it needs all of the parentheses to denote the order of operations! Outputting the infix notation actually requires us to visit the operator nodes in prefix, postfix, and infix order. (The prefix and postfix visits output the parentheses.)

Binary trees - array implementation

It is also possible to implement a binary tree using an array, although this is less common, since it requires knowing the size of the tree and wastes space if the tree is not **complete**. In a complete tree, there are no NULL child pointers, except at the lowest level of the tree. With this definition a complete binary tree with height n has 2^{n+1}-1 nodes. More generally, we might allow a complete tree to have empty children at the next to last level of the tree as long as there are no non-empty children to the right of empty children at this level.

The basic idea in the array implementation of a tree is to store the root at index 0 and the children of a node at position i at positions $2i+1$ and $2i+2$. Here is an example tree:



Here is the array representation:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	B	C	D	E	F	G	H	I	J	K			L	M	N	O

Trees and recursion

Most tree algorithms are implemented using recursion. It is possible to implement them using a stack, but recursion is much easier.

Let's start with something easy: counting the number of leaves in a tree.

```
int countLeaves(BinaryTreeNode *root)
{
        if (root == NULL) return 0;
        if (root->leftChild == NULL && root->rightChild == NULL) return 1;
        return countLeaves(root->leftChild) + countLeaves(root->rightChild);
}
```

This is essentially a preorder traversal of the tree, but the addition of the leaves in the two subtrees happens in postorder. It is useful to trace the **execution tree** for functions like this.

If this were part of a tree class, we would need to use a helper function, since we wouldn't want our operation to have any parameters:

```
int BinaryTree::countLeaves()
{
        return countLeaves(root);
}
```

In your code, you should keep these together in the implementation (.cpp) file. They don't go together in the header (.h) file, since one is public and one is private.

Another example is to count the number of nodes at a particular level of the tree. Here our helper function will need some additional parameters. For the next two, we'll assume we aren't writing as part of a class.

```
int countNodesAtLevel(BinaryTreeNode *root, int level)
{
	if (root == NULL) return 0;
	if (level == 0) return 1;
	return countNodesAtLevel(root->leftChild, level - 1) +
		countNodesAtLevel(root->rightChild, level - 1);
}
```

Finally, let's find the sibling (if it exists) of a particular object in the tree. We'll return NULL if there is no sibling.

```
const Object *sibling(BinaryTreeNode *root, const Object &key)
{
	if (root == NULL) return NULL;
	if (root->leftChild != NULL && *root->leftChild->item == key)
	{
		if (root->rightChild == NULL) return NULL;
		return root->rightChild->item;
	}
	if (root->rightChild != NULL && *root->rightChild->item == key)
	{
		if (root->leftChild == NULL) return NULL;
		return root->leftChild->item;
	}

	const Object *tryBranch = sibling(root->leftChild, key);
	if (tryBranch == NULL) tryBranch = sibling(root->rightChild, key);
	return tryBranch;
}
```

Try finding the depth of a node in the tree. Return 0 for the root, -1 for an object not found. Use the following prototype

```
int depth(BinaryTreeNode *root, const Object &key);
```

One solution:

```
int depth(BinaryTreeNode *root, const Object &key)
{
	if (root == NULL) return = -1;
	if (*root->item == key) return 0;

	int tryBranch = depth(root->leftChild, key);
	if (tryBranch == -1) tryBranch = depth(root->rightChild, key);
	if (tryBranch == -1) return -1;
	else return 1 + tryBranch;
}
```

Practice problems (optional):

Carrano 5th Edition, Chapter 10: Exercise: #2
Carrano 6th Edition, Chapter 15: Exercise: #4

Write a method to sum the elements in a binary tree.
Write a method to count the number of nodes with exactly one child.