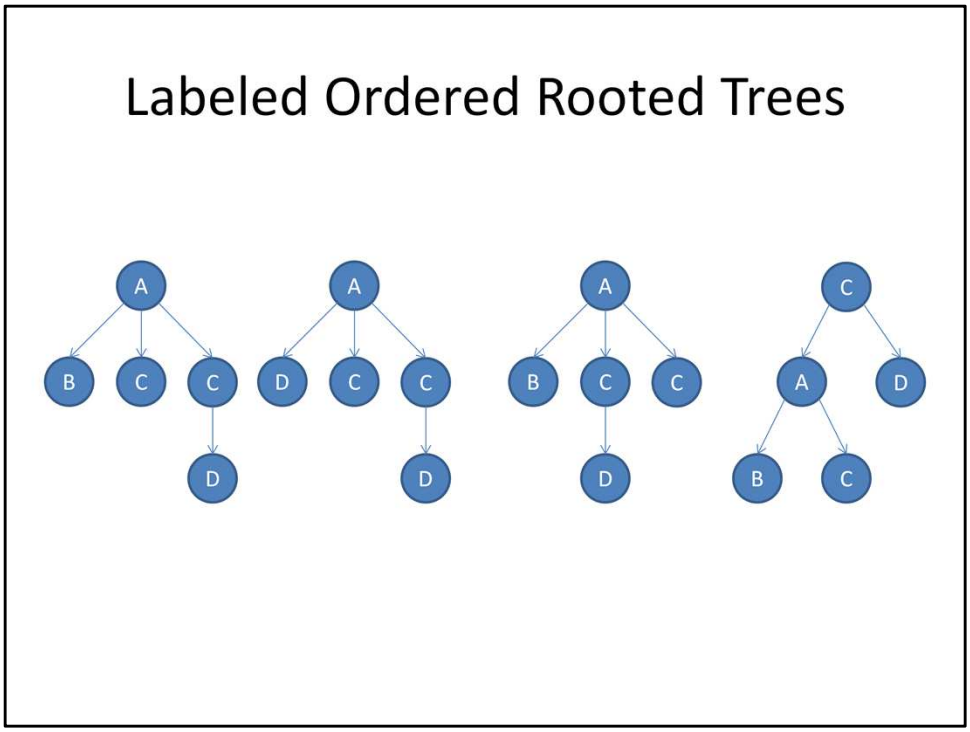
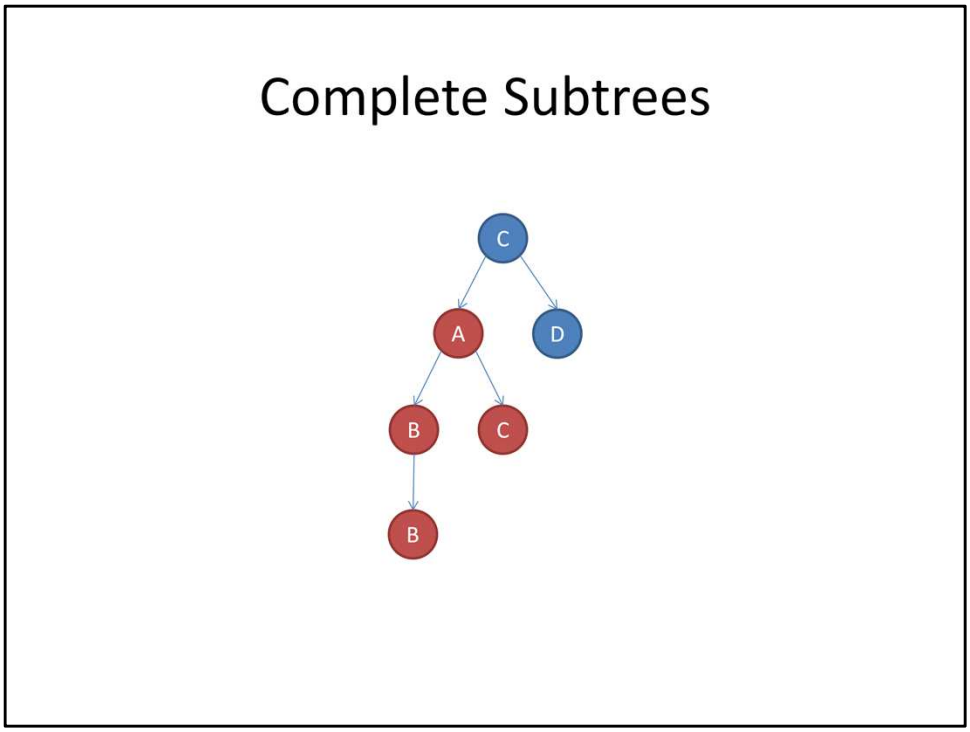
CS161L: Implementation of Algorithms

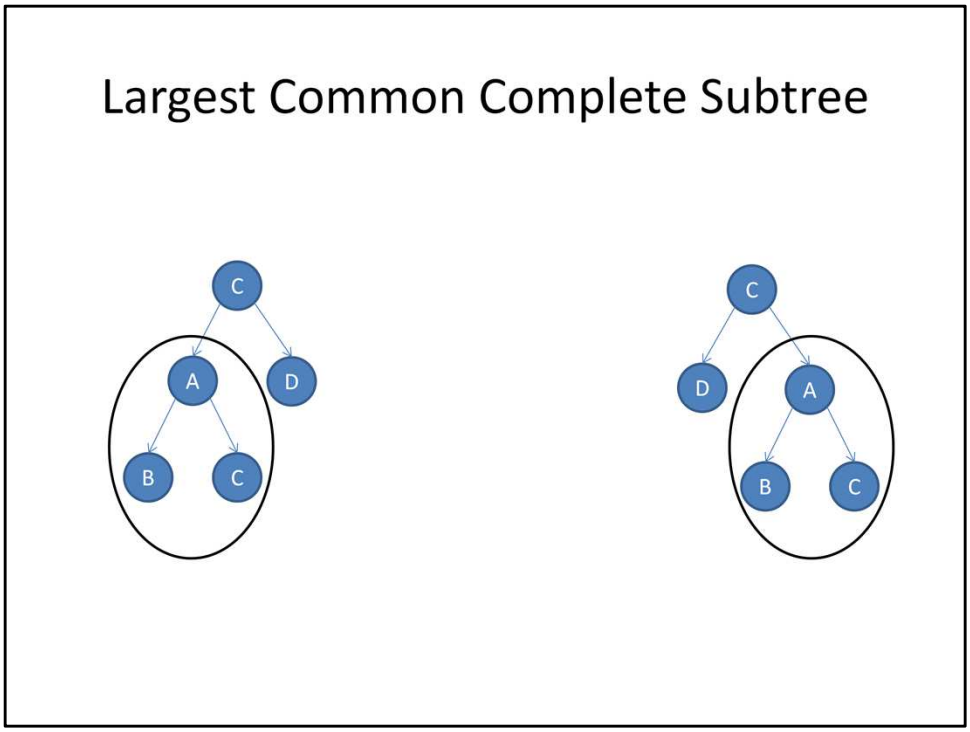
Friday, May 2, 2014



You may have noticed that the first section of this week's problem is devoted to defining exactly what it is we're working with. For what it's worth, it's a lot simpler to explain with a couple pictures. All of those extra words we used to describe the tree just tell us which trees to treat as equal or not equal. In this picture, none of these trees are equal to each other. Let's focus on the first tree, and go over why the second through fourth trees are different from the first tree. In the second tree, the leftmost node is labeled a D instead of a B, so the trees aren't the same, even though their structure matches, which is why we say the tree is labeled. In the third tree, even though the unordered set of children of the root is the same, the second and third children of the root are switched in order, so the trees aren't the same, which is why we say the tree is ordered. In the fourth tree, the same nodes are adjacent to each other as in the first tree if we were to treat edges as undirected, but they're not if we treat them as directed, which is why we say the tree is rooted.

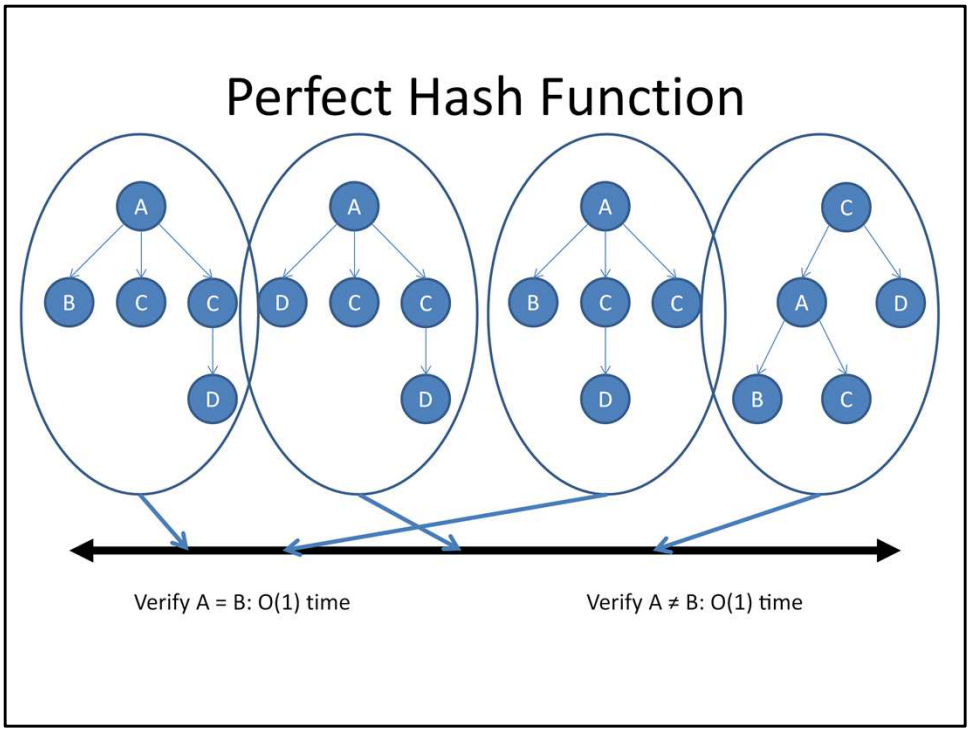


Now let's go over what a complete subtree is. A complete subtree is what you get if you pick a node, say this A here, and keep it and all of its descendants. Notice that for each node we pick, we get a different complete subtree, so if there are n nodes in our tree, there are n different complete subtrees, no matter how the tree is structured.

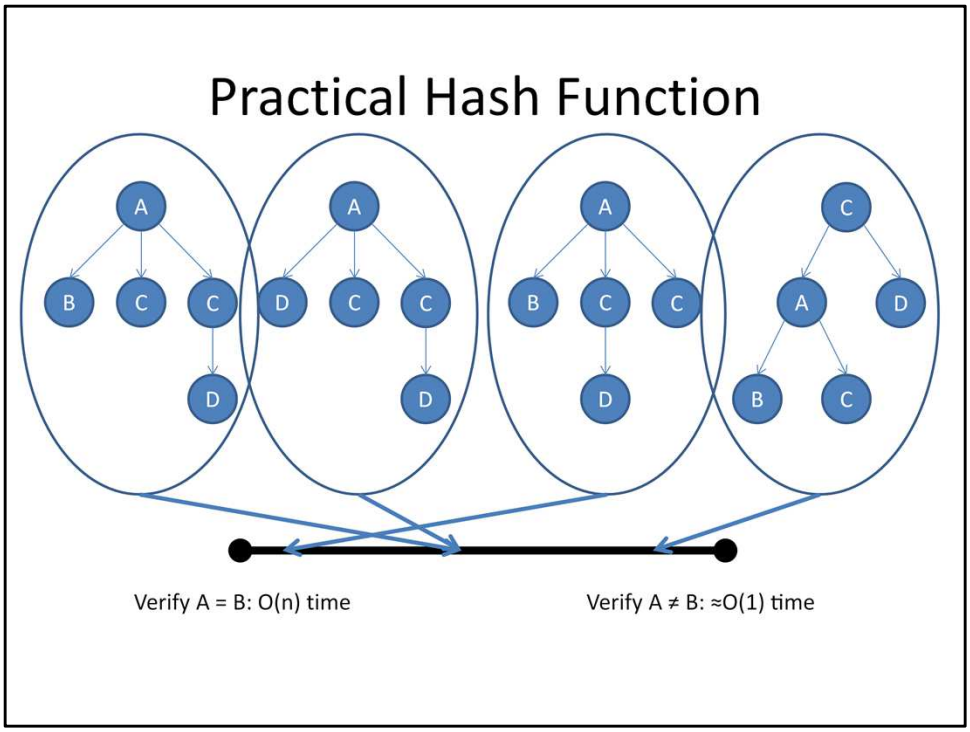


So what problem are we trying to solve this week? Well, we're given two trees, and we want to find the largest complete subtree they have in common. In this case here, we've circled what the largest common complete subtree is. This problem wasn't covered in the main 161 lecture, so I'd like to take a moment to justify why we would care about a problem like this. In fact, this came out of some research I've been working on over the past year. You see, we can represent the computer programs we write as trees that reflect the structures of programming we obey, kind of like how we can diagram sentences in natural language. In fact, compilers do this in the process of turning high-level code into assembly code. Then a complete subtree of such a tree corresponds to some piece of the code, say the condition of an "if" statement. If we want to find similar regions between two different pieces of code, for example to see whether two programming submissions might have involved plagiarism, we can parse the code to make these trees, and then find the subtrees they have in common. All right, so how do we actually solve this problem? Well, we just said that a tree with n nodes has n complete subtrees, so naively we could take all pairs consisting of one subtree from the first tree and one subtree from the second tree, see whether they match, and then take the biggest matching pair. This would take us cubic time, since there are n^2 pairs of subtrees, and checking to see whether two subtrees match takes linear time. If we think about it a little more, we can get this down to quadratic time by observing that in order for two subtrees to match, they have to be the same size, and the complete subtrees that are of the same size in a given tree are all disjoint. But we can do even better: We're going to go over how

to solve this problem in LINEAR time, which you know has to be optimal because no matter what you need to at least look at both trees. And as promised, the trick we're going to use is hashing.



How does hashing help us? Well, imagine we had some way of hashing each and every possible tree to a unique integer. In that case, we could test whether two trees were equal simply by checking whether those hashes matched. If we could do that check in constant time, then we could dump the hashes of all complete subtrees of one tree into a hashtable, and then do a lookup for each hash of a complete subtree of the other tree, all of which would take linear time. Unfortunately, this story sounds a little off. There are an infinite number of trees, and while there are an infinite number of integers as well, we can only actually get away with constant time comparison of the numbers if we restrict ourselves to a finite set of integers, say a 32-bit int.



So what we're going to do instead is we're going to hash all trees to 32-bit integers. This means we can't get a one-to-one mapping, but if we do it right, the chances of a given pair of trees hashing to the same number will be very small. In this case, if two trees are identical, they'll still match to the same number, but we'll need to actually verify equality by checking the trees themselves to make sure it wasn't a collision. This means it takes linear time to do an equality test if the answer is yes. On the other hand, if two trees are different, the vast majority of the time they'll have different hashes, in which case we can definitively return no in constant time. Once in a while the hashes will collide, so it'll take linear time to find the difference in the trees, but the chances of this happening are so small that in expectation, it still takes only constant time to do an equality test if the answer is no.

Practical Hash Function

$$H(X) = \sum_{i:x_i \in X} p^i H(x_i)$$

$$H(Tree)$$
$$= H(root) + pH(size) + p^2 H(children)$$
$$= label + p \cdot size + p^2 H(children)$$

So how do we make such a hash function? Well, there's a strategy that's adopted pretty often in Java; in fact, if you have Eclipse autogenerate a hash function for a class based on its ivars, it will generate one of this form. Note that we ARE using a deterministic hash function here; we're not going to worry about the scenario that requires us to randomly select from a family of hash functions. What this function does to hash a class is it takes each ivar in the class, recursively hashes it, and multiplies each such hash with a different power of some prime. In Java, you'll often see the prime 31 used for this purpose. The choice of prime isn't too important, as long as you don't choose something like 2. The reason for this is we're just going to happily allow the computation to overflow the 32-bit int and wrap around, and as long as we choose an odd prime, the overflow won't hurt us much. In any case, in our example, we're gonna have 3 instance variables that matter in our representation of a tree: the root node label, the size of the tree, and the list of children of the root. Note that Java has a builtin hash function for lists that applies this function to each element in the list, so if we define our hash function appropriately for a tree node, hashing the list of children will invoke our hash function in the next generation as we expect.

Hashing All Complete Subtrees

- Recursively compute and store sizes of all subtrees
  - In each node, store size of complete subtree rooted there
- Do the same for hashes!

Now this hash function for a tree takes linear time, since it has to hash every node in the tree. However, in the process of doing so, we get the hashes of all of the complete subtrees, so if we save our results, we can get the hashes for all complete subtrees in linear time. How would we do this? Well, let's look at an easier example, that of computing the sizes of all the subtrees. We'll want to do this first anyway since we're going to use it as a part of our hash. The size of a tree is equal to 1 plus the sum of the sizes of all the subtrees that are children of the root. So when we compute this, for each node, we store the size of the complete subtree rooted there before we pass it back to the parent that asked for our size. We'll do the same thing with hashes. When our parent asks us for our hash, we'll compute it, store a copy for ourselves at the node, and then pass the answer back up. From then on, if anyone asks us for our hash, we just return this stored copy.

Java Hashing

- int hashCode() and boolean equals(Object o)
  - a.hashCode() must equal b.hashCode() if a.equals(b) (but converse doesn't have to hold)
- hashCode:
  - actually compute the hash only once, and have hashCode return the precomputed value
- equals:
  - check precomputed hashCodes first for O(1) false, then do a deep check for O(n) true

Now how does hashing work in Java? Well, every Object has two functions that you can override, namely, hashCode and equals. You HAVE to use these methods, because they're what the Java standard library expects to exist for you to be able to put your Objects into a hashtable. hashCode returns an int that's equal to your hash value, and equals returns whether you're equal to the Object passed in. They need to be defined in a manner consistent with each other. What does that mean? If two objects are equal to each other, then their hashCodes HAVE TO be the same. The converse isn't necessarily true; hashCodes can match even if the objects are different. For what it's worth, that means always returning 0 is a consistent hashCode with any definition of equals; it's just not a very good one. Now as we mentioned on the last slide, when you implement hashCode, you should only actually compute the hash the first time hashCode is called, and for all subsequent calls, you should just return the stored value. Then, when you implement equals, you should first check the hashCodes of the two objects to see whether they agree, because if they don't, you can immediately return false. It's only when the hashCodes agree that you have to do a deep equality check, namely, check to see that the labels, sizes, and lists of children are themselves equal. By the way, notice that equals takes in an Object as an argument; you'll need to cast that to a Node before you can access its fields.

Java HashSet<T>

```
Node[] tree = new Node[n];
// initialize tree, including hashes
HashSet<Node> set = new HashSet<Node>();

// add to set if not already present
// returns true if not already present, false otherwise
// warning: if already present, will cost a deep equals
set.add(tree[0]);

// check to see if inside set
// warning: if this returns true, will cost a deep equals
// warning: input type not checked at compile time
if (set.contains(tree[0])) {
  ...
}
```

Once you've implemented hashCode and equals for a class, you can put instances of that class into a Java HashSet. The things you'll care about today are making a HashSet, adding to a HashSet, and checking for whether a HashSet contains a particular object. All of these functions are intuitively named. You should be careful about using contains, though; make sure you're always passing in an object of the right class. Even though HashSets are generic, because they existed before generics were introduced to Java, the contains function is required to take any Object, not just the type you specified. That means the compiler won't check to make sure you passed in the appropriate type for the HashSet. Also as a reminder, I've included warnings about what these calls will cost you in terms of runtime. Basically, if there isn't a match, only hashCode will be invoked, which will cost you constant time, but if there IS a match, there will be an equals call, which will cost you time linear in the size of the subtree you just passed in.

Largest Common Complete Subtree

- Given trees A and B, want to find LCCS(A, B)
- Put all complete subtrees of A into a HashSet
  - but be careful of duplicates!

OK, so now let's go back to that idea we had when we assumed we had a perfect hash function, and finish it up for our practical hash function, with hash caching. First we precompute all the sizes and hashes of all the subtrees of both trees. Then we take all the complete subtrees of the first tree and put them into a HashSet by adding the root Node, and then recursively adding all subtrees of all its children. We do want to be careful, though; if we ever call add on a Node that's already there, we just paid linear time. However, if we just tried to add a Node that's already in our set, then all subtrees of THAT Node also have to already be in our set (since we recursively added all subtrees of the first instance), so we DON'T have to recursively add all subtrees of the children of our current Node. Thankfully, there's an easy way to check whether our add hit a duplicate; the add method returns false if the thing we tried to add already exists in the set. Then, we look at our second tree, starting at the root. If the root Node exists in the set, then we can return its size as our answer, as the whole tree matches some subtree in the first tree; otherwise, we recursively check each of the children of the root to find the largest common subtree under each child, and then return the largest of all of those.

Runtime Analysis

- Computing size + hashes: linear time
- Equals calls are expensive
  - returning false is constant, returning true is linear
- But every time we do a linear call on a subtree rooted at x, we do NO further calls on the descendants of x!
- Each node only participates in one equals call
  - total time: linear!

So how do we analyze the runtime of this algorithm? Well, the precomputation of the sizes and the hashes is linear, since we visit each node exactly once. From then on, we make a linear number of calls for sizes, hashes, and equality, all of which are constant EXCEPT for the last one. In fact, equals calls can be expensive, since returning true means we cost linear time, and we make a linear number of equals calls. It might look like we still are taking quadratic time, but if we look carefully, we can see that the equals calls taken all together don't actually cost that much. The reason for that is whenever we pay linear time for a subtree, say, when we added it as a duplicate when we were building up the HashSet, we then skip over all descendants of the root of that subtree. That means that we can think of each node in our tree as only costing us once; either it cost us because we called equals directly on it, or it cost us because we called equals on ONE ancestor of it, but never both. That means that the total amount of work that all the equals calls cost us is only linear in the number of nodes. This kind of analysis is called amortized analysis; next week in 161 lecture we'll go over how to argue a runtime like this formally, but this informal argument you saw here is the basic idea behind it.