1 Expression trees

Reading: Chapter 24, second half.

1. Write a program which takes two expression trees and checks if they represent the same expression. Assume here that the operators + and / are non commutative and non associative.

2. Same as above, but assume that + is commutative. This basically means that two trees represent the same expression if the second can be obtained by exchanging left and right subtrees in the first.

   The standard way of doing this is: convert the given tree to a *canonical form.* A canonical form is simply a unique way of writing something.

   A simple canonical form for trees can be defined as follows. First define the inorder signature of a tree. This is simply the textual input which would generate the tree. Now we know that several inputs would generate the same expression (though not the same tree), for example, ((a+b)+(c+d)) and ((d+c)+(a+b)). Consider an expression A. It can be represented by several trees T1, T2, ... because of the commutativity of +. Each of these trees will have inorder signatures S1, S2, ... respectively. Suppose Si is the lexicographically smallest among these signatures. Then we will say that Si is the canonical representation of A.

   So the idea of the algorithm is: given two trees, generate their canonical representations and check if they are the same.

3. Solve all problems at the end of Chapter 24.

2 Hash Tables

1. Consider a hash table with 100 entries and a universe $\{0, \ldots, 123456\}$. What is the largest possible list size that can arise for any table entry?

2. Suppose I have a hash table of size 2^t and a universe of size 2^u. Let A be a $t \times u$ matrix, in which each entry is independently set to 0 or 1 with equal probability. We can think of keys as u bit vectors, and table indices as t bit vectors. For any key x, let $h(x) = Ax$, where the matrix vector multiplication uses addition modulo 2. Thus $h(x)$ is a t bit vector, and thus can be used to index into the table.

   Suppose x, y are any keys. Show that the probability that x, y have a collision is $1/2^t$.

3. Consider the setting of the previous problem. Suppose n arbitrary keys are hashed into the $m = 2^t$ sized table. Let i be any table index. What is the expected size of the list associated with $T[i]$?

The point of the last two exercises is: even if you publish that you are going to use a random matrix as a hash function (but the matrix is generated only during execution), no one can force your table to have bad behavior. In other words you can now claim that your table will work well on the average, where the probability space is the space of all choices for the matrix A. This is better than saying that hashing works well *typically* or hashing works well if the keys are randomly generated – which you cannot really guarantee in practice.