# Ground-Up Computer Science

Chapter 4

(January 22, 2022)

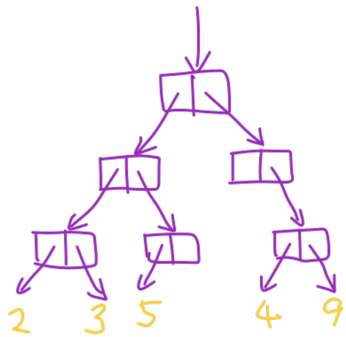
Yin Wang

# 4 Trees

A: Today we make a small extension on *lists*, and we can have *trees*.

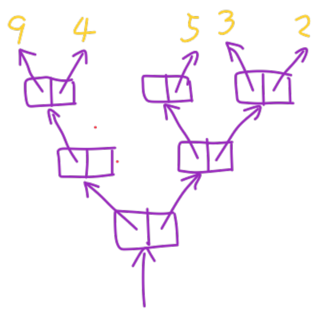
B: What are *trees*?

A: For example, here is a tree. It is built with pairs.




B: Pairs again. They are really powerful. This does look like a tree, except that it grows downwards.

A: We could draw the tree structure in the other direction, then it will look exactly like a tree.




B: Indeed, then why do we draw it upside-down?

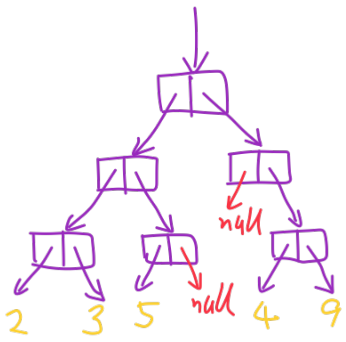
A: Computer science is different from art. We often don't know how tall the tree is. If we draw it as a usual tree, we won't know where to put the root. So we usually draw the root first and let the tree "grow" downwards. This is more convenient.

B: I see.

A: Can you see how similar they are, the tree data structure and the real tree? They both have *branches* and *leaves*. The pairs are the branching points, and the numbers are leaves. We call the pairs *internal nodes* because they are in the middle of the tree.

B: It seems that some pairs have only one branch growing from them?

A: Yes. Where I haven't drawn branches, you may think there is a `null`. We use `null` to mean that there is no left or right branch.




B: I have seen `null` in the previous lesson when we learned lists. We use `null` to mean the empty list. It seems that `null` has a similar meaning in trees too.

A: Yes, they are very similar. In lists, `null` means an *empty list*. In trees, `null` means an *empty tree*. `null` signifies the end of a list or a tree.

B: So the two `null`s have different meanings. Is it possible that we get confused because both data structures have `null` in them?

A: It is possible. We often use the same `null` to mean different things in computer programming. Some people have complained about this.

B: Can we do something different?

A: The important thing of this lesson is the tree data structure, so I will try not to use too much abstraction. But to be clear about the meaning, we can use a variable `emptyTree` instead of using `null` directly. We just define `emptyTree` to be `null`.

```
var emptyTree = null;
```

B: This makes me feel better, although for the computer they are the same.

A: Yes, the purpose is on the human side. Let's see how lists and trees are related. Lists are constructed with pairs whose `second` parts are also lists. The `first` parts are considered to be data (members). When we do recursion on lists, the recursive calls only go into the `second` parts. This is why lists have a *linear* shape. If we relax these restrictions, we have trees.

B: Can we be more clear about the restrictions?

A: Let me rephrase that. There are two restrictions on lists.

1. The `second` parts can only be lists.
2. Recursive calls only apply on `second` parts and don't go into the `first` parts.

B: If we remove these restrictions, does that mean

1. The `second` parts can also contain members.
2. Recursive calls may go into `first` parts. If `first` parts are pairs, we look deeper into them.

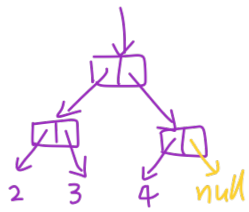
A: Correct. The difference between a list and a tree is partly about the structure, and partly about how we process them.

B: I sort of see this second point. Even some lists contain pairs in their `first` parts, they are still lists, just because we don't treat them as trees. For example, `pair(pair(2, 3), pair(4, null))`.

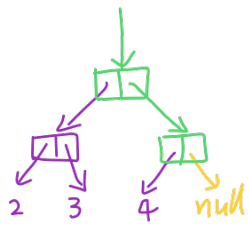
A: Right. Members of lists can be pairs, but if we don't consider those pairs as the structure of the data, then they are still lists.

B: So the difference between lists and trees somewhat lies in how we *look* at a structure, a subjective matter.

A: Indeed this may depend on how you look at the structure. For example, the following tree can also be thought of as a list.




B: That is interesting. If I just follow the `second` parts, I will reach `null`. This would make this tree also a list whose first member is `pair(2, 3)` and the second member `4`.




A: Yes, as long as we don't have recursive calls into the `first` part, we have a list.

B: So some trees can be thought of as lists. Can lists be thought of as trees too?

A: Actually every list can be thought of as a tree.

B: Every one of them? Let me think... Indeed, lists are just trees where the left branches are all leaves, and the last pair has no right branch.

*(Think about this and discuss with the teach if you have questions.)*

A: Every list is also a tree, but not every tree is a list. So we can have picture like this:


B: In math, this means that lists are a *subset* of trees.

A: The trees presented in this lesson are quite restricted, because the *internal nodes* don't contain data members. Have you noticed that?

B: Indeed, there aren't any number (data) in the middle of the trees that we have seen so far. All numbers are at the leaves. The internal nodes only points to internal nodes or leaves. Can we also have data in the internal nodes?

A: Yes we can, but we will wait until the next lesson. This simpler tree structure can help you understand the recursion pattern on them. We only need a small extension to our way of doing recursion on lists.

B: Nice. I like baby steps.

A: I'm glad you understand this. Don't go too fast. Now we can start writing our first recursive function on trees.

B: Good.

A: Actually you have already seen such a function. The `pairToString` function you used in last lesson is a recursive function on trees. We will take a look at it first.

B: Okay. I have the code here.

```
function pairToString(x)
{
  if (!isPair(x))
  {
    return String(x);
  }
  else
  {
    return "("
      + pairToString(first(x))
      + ", "
      + pairToString(second(x))
      + ")";
  }
}
```

A: Take a look at the code. How many *recursive calls* are there?

B: Two of them. One of them is recursion on `first(x)`, the other on `second(x)`.

A: Good. Take a look at the list functions you wrote for the previous lesson, have you ever did recursion on the `first` parts?

B: No. All the recursive calls are on the `second` parts.

A: This is because the `first` parts are considered to be data and not structure, so you don't do recursion on them.

B: I see.

A: Now you may understand `pairToString` using this difference.

B: This is easy. I can see that it does recursive calls on `first` and `second` parts, converts them into strings, inserts a comma in between, and put parentheses around the whole thing.

A: How about the base case? Can you figure out how we arrive at the base case, using our three-step method of thinking about recursion?

B: I figure out the base case by looking at the recursive calls. There are two recursive calls `pairToString(first(x))` and `pairToString(second(x))`. Both of them eventually reach a leaf node.

A: Right. Leaf nodes are not pairs, so we use the condition `!isPair(x)` to distinguish them. This case also includes `emptyTree`.

```
if (!isPair(x))
{
  return String(x);
}
```

B: For the base case's return value, I guess `String(x)` is JavaScript's way of converting values into a strings?

A: Yes, but `String(x)` can only meaningfully convert basic data recognizable by JavaScript, that is, numbers, booleans, strings etc.

B: Can't I use `String(x)` to convert pairs into strings?

A: If you use `String(x)` on a pair, you will get something like "[function]" or "select => select(a, b)", just as if you use `console.log` on the pair. Pair is our own *custom defined* data structure, so JavaScript has no idea what is in there. It only knows that it is a function. So we have to convert pairs to strings ourselves.

B: Got it. We have to write a function like `pairToString`, which is a recursive function on trees.

A: Right. This is why we need `pairToString`. We will need to write these "toString" functions for any custom defined data structure, if we ever want to display them.

B: I see.

A: This is pretty much all you need to do the exercises for trees, because trees are just simple extension to lists.

B: Nice. Are there a lot of exercises?

A: Almost the same amount as lists. For almost every list function, there is a tree function which does a very similar thing.

B: That is good amount of exercise. Can I relate the functions together?

A: Yes. I gave similar names to them. For example, when there is `listEqual`, you have `treeEqual`.

B: Nice.

A: Here are the exercises. Exercise Set 4. (Exercise omitted for the sample)

B: Thank you!