Ground-Up Computer Science

Chapter 6

(May 1, 2021)

Yin Wang

6 Lookup Tables

A: We are done with the calculator.

B: Can we start writing the interpreter now?

A: Not yet. We have an important data structure missing. Once we have it, we will be ready to write the interpreter.

B: What is the data structure?

A: It is called a *lookup table*. Actually we have two such structures. One of them is a list, the other is a tree. They fulfill the same purpose, so they are exchangeable.

B: What does a *lookup table* do?

A: A *lookup table* is like a restaurant menu. You look it up for the price of an item. What is the price of steak?

pizza	128
cake	46
pasta	68
steak	258
salad	45
beer	35

B: 258.

A: Have you got the idea? We call `"steak"` the *key*, and `258` its *value*. You lookup the key for its value.

B: I see. What has this to do with interpreters? The keys feel like variables.

A: You are ahead of me. In the interpreter we use lookup tables to find values of variables. But for demo purposes we just use them to make restaurant menus here.

B: Foods are good examples.

A: Lookup table is a very useful data structure. It is the core idea behind *databases*. Database people usually call it *key-value store*.

B: I always thought databases are complicated monsters.

A: If the product looks simple, they can't ask for a huge price for it.

B: That is funny!

A: Let's start building our first lookup table. Look at the menu. It has multiple rows. Each row has two columns, the key and its value. We can use known data structures to represent the rows and columns.

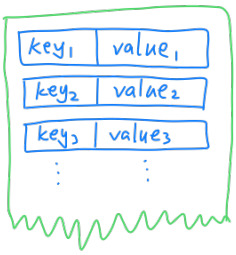
B: I guess each *key-value pair* can be represented as a *pair*?

A: That is reasonable. You just need another structure to contain the pairs.

B: I can use a list.

A: Why do you use a list?

B: Because the table has a linear shape in the vertical direction. It looks like a list.



A: Yes, you can use a list. Another reason to use the list is because the list is a *recursive data structure*, so it can contain arbitrary number of rows.

B: I haven't thought of this. What does "arbitrary number" mean?

A: "Arbitrary number" means that the number is not a predetermined constant, for example 6. You may have different menus and they may have different lengths, but we have only one definition for the lookup table data structure.

B: I see. One definition for multiple lengths. That is why we have the `length` function for lists, because each list may have a different length.

A: There is one more thing that may help our understanding here. We can define a variable `emptyTable`, whose value is `null`. It is much like `emptyTree`. Use `emptyTable` for the end of the table instead of `null`. This may improve understandability of code.

B: Okay. I'll keep that in mind.

A: Now you may construct the menu with the "pairs inside a list" structure. Write the code for constructing the menu. Run it and display the table with `pairToString`. You should be able to see the menu.

B: Don't we define an abstract type for the lookup table?

A: This structure is so simple, so we skip that part this time. We just need to understand the idea. For actual engineering projects, it is usually better to define an abstract data type even for the simplest structures.

B: *(Write your answer for constructing the menu and send it to the teacher.)*

A: Let's call this `menu1`. Create a variable for it. Now, write a function `lookupTable` which finds the value for a given key. It should be a recursive function on lists.

```
function lookupTable(key, table)
{
  // TODO
}
```

After you have this function, you may test it with `menu1`. Just lookup some keys and make sure you get correct values.

B: *(Write your answer and send it to the teacher.)*

A: Very good. We need another function `addTable`, which will construct a *new* table based on an existing table. It will put a new key-value pair on top of the old table, thus creating a new table.

```
function addTable(key, value, oldTable)
{
  // TODO
}
```

You can now open Exercise Set 6. There are some more tests in there. Make sure they all pass and understand their implications.

B: *(Write your answer and send it to the teacher.)*

A: Well done. Do you have any questions?

B: I see that you emphasized that `addTable` creates a *new* table. Why is that?

A: Because `addTable` creates a new table. It doesn't modify `oldTable`. This is like all other list functions you have written so far. They never change their input lists but just create new lists. The new lists may reference old lists, so we don't need to copy a lot of data.

B: Since the old table is not modified, can we still find the old value of the key in the old table?

A: Yes, you can find the old value because the old key-value pair is never deleted. There is a test for this. Please study the test carefully and understand why it behaves that way.

B: I have studied it. Indeed, when I lookup the same key in new and old tables, `lookupTable` gives me different values. How is this useful?

A: This can be quite useful. Have you noticed that the old data is automatically "backed up" whenever you update it?

B: Yes, that is interesting. It behaves like a *version control system* such as Git.

(You may ignore the following conversations if you haven't heard of Git or blockchains.)

A: Actually Git uses the same idea, except that Git uses a tree, not a list. We have already built similar trees. The tree functions in this course don't change their inputs either.

B: Good to know. Are there other things built with this idea?

A: Have you heard of *blockchains*?

B: Yes. That is a hot topic recent years, but I have never understood how blockchains work.

A: After this lesson you may have ideas how to build those things. They are just slightly more complex.

B: That is so nice. New doors opened for me.

(Ignorable parts end here.)

A: Now we are done with the linear lookup table, built as lists. Let's proceed to something more advanced. It is called a *binary search tree* (*BST* for short). It serves the same purpose as a lookup table.

B: How is the *BST* more advanced?

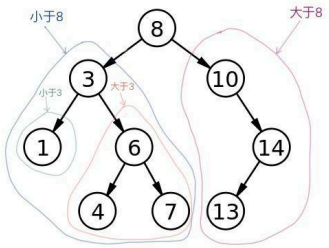
A: It is usually more efficient, so you can find the values quicker using less energy.

B: Nice. How do we use less energy?

A: Do less work.

B: Okay.

A: A BST looks like this. The numbers in the circles are keys. I haven't shown the values but they are also in the nodes.



B: How is this different from a usual tree with numbers in internal nodes?

A: Look at the tree. It has this property.

For every key k in the internal node, every key in the left subtree is less than k, and every key in the right subtree is greater than k.

B: I can see that. This is true for every internal node. 8, 3, 10, 6, 14.

A: Yes, every internal node must have this property, otherwise it will not work.

B: I didn't know that keys can be numbers too. We used strings for the menu.

A: Keys can be any value that can be *compared*. Do you remember that you used the `==` operator in `lookupTable` function for the lookup table? `==` is a *comparison operator*.

B: Yes. I didn't think about this carefully though.

A: Keys can't be something that you can't compare.

B: For example? What can't be compared?

A: For example, the beauty in two paintings, the value in two persons.

B: Oh, that makes sense.

A: For BST, we need to not only use `==`, but also `<` and `>`.

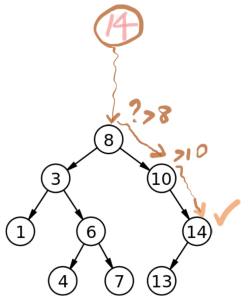
B: How do we use them?

A: Here is how you look for the key *k*. If k is equal to the key of the current node, then you have found it. Otherwise if *k* is less than the current key, then you look for *k* in the left subtree, otherwise you look for *k* in the right subtree. Is that clear?

B: I'm not sure. Maybe we can do an exercise?

A: Okay. Try looking for the key 14 in the previous BST.

B: Since 14 is greater than 8, I look for 14 in the right subtree, and I see 10. Since 14 is greater than 10, I go right again. Then I have found it. The process looks like this picture.



A: Good. Drawing pictures is a good habit. One picture is worth a thousand words.

B: It seems we only visited three nodes of the tree. We always go down from the root, never switch to the other branch.

A: Right. This is why BST is more efficient than linear lookup table.

B: But for the linear lookup table, we may also skip some nodes. If we find the key before reaching the end, we just return it without looking further.

A: That is the case. But for BSTs, the number of visits can be much smaller. Can you see why?

B: I have no idea right now.

A: If the tree is balanced, that is, every internal node has two subtrees, what is the *depth* of a tree with 3 nodes?

B: *(Draw a picture showing the answer, and send it to your teacher.)*

A: What is the depth of a balanced tree with 7 nodes?

B: *(Draw a picture showing the answer, and send it to your teacher.)*

A: What is the depth of a balanced tree with 15 nodes?

B: *(Draw a picture showing the answer, and send it to your teacher.)*

A: Do you see the pattern? What is the depth of a balanced tree with *N* nodes?

B: (Answer the question with a formula containing N, and send it to your teacher.)

A: Very good. What is the average number of visits if you look for a key in a linear lookup table with *N* keys in it?

B: (Answer the question with a formula containing N, and send it to your teacher.)

A: Compare those two formulas, you can see a big difference. The logarithm function is very slow growing, so the BST has the potential of skipping a lot more nodes because you just need to travel the tree's depth.

B: I can see that from the picture. I just go from the root of the tree to a leaf. Sometimes I don't even need to go all the way to a leaf because the key appears in an internal node.

A: This is why we often use trees in computer science. It is probably the most used data structure.

B: Really?

A: There are more complex data structures, but they are seldom used, so trees are considered most useful.

B: Good to know that.

A: Balanced trees are very efficient, but for learning purpose, we will not write balanced trees because they have quite some complexity. Our simple BSTs may not be balanced, but they are often quite efficient too.

B: Okay. I will be happy enough with the basic BST.

A: Similar to `emptyTable`, we have a variable `emptyBST`. Its value is also `null`. Instead of using `null` directly, we use this variable.

```
var emptyBST = null;
```

B: I see. This is just to avoid confusion.

A: Yes. Now we can start to build BSTs. First we write a constructor function for the internal nodes. This node structure is very much like `binop`, except that it contains a key and a value, instead of an operator.

B: So the internal node has four parts -- key, value, left subtree and right subtree?

A: Correct. Four parts. Now write this constructor function. It is called `bst`. It should be easy because you have already written so many constructors.

```
function bst(key, value, left, right)
{
  // TODO
}
```

B: *(Write your answer and send it to the teacher.)*

A: Now we need another function `addBST` which constructs a new BST with a new key-value pair. `addBST` is very much like `addTable`. In fact they have exactly the same parameter pattern, except the last parameter means a different structure.

```
function addBST(key, value, node)
{
  // TODO
}
```

B: I see. `addTable` extends a table, and `addBST` extends a BST, but they serve the same purpose, that is, extending the structure to have one more key-value pair.

A: Because of this similarity, we may even exchange them freely with the use of abstraction. You may see this in the interpreter exercises.

B: Looking forward to the interpreter lesson.

A: Using `addBST`, you should be able to conveniently construct the previous restaurant menu, starting from `emptyBST`. You may construct it now, call it `bstMenu1`. Remember to use `emptyBST` instead of `null`.

pizza	128
cake	46
pasta	68
steak	258
salad	45
beer	35

B: *(Write your answer and send it to the teacher.)*

A: Next function to write is `lookupBST`. It just implements the lookup process that you have described before. `lookupBST` should be a recursive function.

```
function lookupBST(key, node)
{
  // TODO
}
```

B: *(Write your answer and send it to the teacher.)*

A: Now we are done with lookup tables and BSTs. Next lesson we can write an interpreter.

B: Great!

(Exercise omitted for the sample.)