Lecture 6: Linked Structures

In this lecture, we learn about linked structures, a family of data structures that can be visualized with diagrams of boxes with arrows between them. That style is a natural match with recursion, which we have been studying so far mostly in the context of search problems. Linked structures are related to the concept of pointers in some non-Python languages, and recursion and pointers together have been described as the defining difficult concepts of programming or computer science\(^1\). If nothing else, practicing these concepts will prepare you for the interview questions that the big-name software companies like to use! It doesn't hurt that these ideas also form the bedrock for most kinds of algorithmically challenging coding.

A Puzzle with Unintended Sharing in Python Lists

The code for this lecture begins with a program demonstrating a problem that some students ran into in a lab near the beginning of the semester in 6.009. Here's an easy way to initialize a list of lists, where each element of the overall list is an empty list.

\[
\text{ls} = [[]] * n
\]

For the example of \(n = 3\), the programmer might be thinking of this code as creating a list that we draw like:

In other words, the programmer is expecting that each element of the main list is referencing a separate empty list. Unfortunately, when we modify all of the constituent lists in most any way, we discover how the naïve model is misguided! The reason is that the real effect of the

code above is to create one empty list and then stash its references in all cells of the main list. In other words, the true situation looks like:

A variety of different short code snippets could lead to the first drawing instead. One of them uses a list comprehension:

```python
ls = [[] for i in range(n)]
```

The important thing here is that the `[]` expression gets reevaluated for every value of `i`, creating a new list!

The possibility for several arrows (called pointers or references in different contexts) to go to the same box is called aliasing, and it's a persistent source of complexity in programming. You might be wondering why such a confusing feature should be tolerated in a programming language. It turns out that aliasing can enable some satisfyingly elegant programming styles, including one that we focus on in the rest of the lecture, which allows us to maintain several closely related copies of a data structure, reusing space across them. The important thing is that we will avoid in-place updates and leave each data node alone after we create it, following part of the philosophy of functional programming.

Introducing Linked Lists

A Python list, called an array in most programming languages, can look like:

```
  0   1   2
  A   B   C
```

A linked list uses pointers instead, so that the same example looks like:
What advantages follow from this representation? Well, one disadvantage is that it takes longer to look up an element of the list by position. With arrays (Python lists), we can look up the 1000th element of `myList` like `myList[999]`. With linked lists, we need to follow every arrow from the beginning node (called the `root`) onward, until hitting element 1000!

However, linked lists (and related data structures) often allow much faster creation of new values based on old ones. If we want to add a new element to the beginning of the example list, the array way works in three steps:

1. Create new
2. Copy values
3. Add new ones

The copying step will take time proportional to the length of the array. In contrast, with linked lists, we can build a new derived list in a single step that works immediately, for any addition of a new element to the beginning of a list.
We create a new list node that points to the original list root. There was no need to change the original list. In fact, the original list is still usable! Linked structures make it very easy to build up a sequence of versions of a data structure, where versions share most of their nodes. In this way, we not only speed up the creation of new versions, but we also save plenty of memory vs. a naïve array-based solution.

The lecture code gives examples of a number of classic operations on linked lists, using both loops (“iterative solutions”) and recursion. From basic dictionary-based versions, we also step up to wrapping functionality inside Python classes.

Representing an Archive of Course Schedules: Basic Methods

The rest of the examples in the lecture code are based on course (subject) schedules for the fictitious Institute of 6.009. There is likely to be much similarity between schedules across semesters, so we can share plenty of memory contents across many different semesters' schedules. The basic functionality is a mapping from subject numbers to titles.

One of the simplest implementations in Python uses Python lists (arrays), creating a completely distinct schedule for each semester, with no sharing, because we just copy all old entries that we reuse.

An implementation with more sharing uses linked lists. The simplest variant here allows the schedule entries to occupy arbitrary positions in the list. For instance, using small integers as subject numbers, we might wind up with a list like this one:
To look up a subject in the list, we start at the beginning and follow all the links, checking for subject-number equality until we find the entry or reach the end of the list. A disadvantage of this representation is that, when a subject does not exist, we must always traverse the list fully before we can be sure. However, adding a new entry is a simple “cons” operation like before, so we reuse all of the space for the old entries, and the new list can be built more or less instantly, regardless of the schedule size!

We can keep much of that benefit and dodge the first complaint by maintaining the list in sorted order, more like this:

```
num  title  next  num  title  next  num  title  next
2    ...    1    ...    3    ...    None
```

Now, in some rough sense, lookups take “about half as long,” on average. Unfortunately, adding a new element now requires more work, to maintain sorting. We often need to recreate copies of existing list cells; see the lecture code for details.

**Binary Search Trees (a true classic data structure)**

We can do even better than cutting lookup time by “about half.” Instead we can get it to be “about the logarithm of the schedule size,” while still maintaining sharing across versions. (We're being fuzzy about timing details here, saving the specifics for 6.006 and its follow-up courses.)

Our new approach is *binary search trees*, which look like this:
The top node of the tree we call the root. Here we have a good opportunity to introduce one of the oddities of programming terminology: in the world of computer science, trees grow upside down, with their roots on top! Each node may have some children, connected with arrows going down. Each node is also labeled with the key associated with it. Notice that, reading left to right, we encounter the keys in sorted order.

The basic idea of binary search trees is that each node is implicitly associated with a range of keys that could possibly be found there. We use the same interval notation that you probably know from calculus class. In any kind of lookup, we can skip whole subtrees whose intervals couldn't possibly contain the key we care about! That's the key to speeding up all operations, as compared to sorted lists.

The root of the tree is associated with interval [2, 7], containing all keys from the tree. The root includes key 5. All keys less than 5 must be in its left child, and all keys greater than 5 must be in its right child. We illustrate this rule in tagging nodes with their intervals. “Zigging left” pulls down the high end of the interval to equal the value we've skipped, and “zagging right” pushes up the low end of the interval to equal the value we've skipped.

When searching for a key, we recursively walk down the tree. If the key on the node matches the one we want, we're done. Otherwise, we know exactly which child tree to search next, because their intervals have no keys in common! We get to skip entirely the irrelevant subtree.

The lecture code shows the details of lookup and adding an element in binary search trees. It also works through a small challenge, arising
when we step through a sorted list in order, adding each element to a binary search tree. It turns out that we then get a degenerate tree that is just as slow as a sorted linked list! Another recursive function comes to our rescue.

As a final example, the lecture code uses binary search trees in a small rewrite of last lecture's Sudoku solver with implication tracking. We avoid the need to track and undo implications, by saving all past versions of the board along the current path of function calls. Binary search trees achieve good sharing among versions, so we save space over a similar implementation that just copies the two-dimensional arrays used in the original Sudoku code.