Nested Data Structures

One of the most surprising things about programming can be how many different ways there are to code up the same informal idea. The space of choices can usually be divided between *data structures*, or the ways of storing data, and *algorithms*, or the ways of working with the data structures to compute certain results. In this lecture, we explore (informally) a variety of different implementations of one high-level data model and some operations on it. In the process, we learn some general techniques for encoding data, including via multiple levels of data structures *nested* inside of other structures.

Our motto throughout will be that choosing a data structure is a careful balancing between concerns of *how easy the code is to write* and *how well the code performs.* Performance can be broken out into dimensions like speed and memory usage, but today we will only focus on speed as the dimension to contrast with ease of coding.

Preface: What Is an Object?

One of the most prevalent terms in programming is *object*. We learned about *object-oriented programming* in previous lectures. In this lecture, we'll avoid Python's explicit object-orientation features to show that the idea of an *object* is more fundamental.

For this lecture, an object is *a value with a small, fixed set of fields holding further values*. For instance, an object standing for a student might contain a field for the student ID (a positive integer) and a field for the student's name (a string of text). An object standing for an academic major might include a course number (a positive integer), a name (a string of text), and a roster of participating students (a list of student objects). The concept meshes nicely with informal ways of describing all sorts of real-world ideas.

For us, objects are *mutable*, meaning that we can change the values of an object's fields *after* the object is first created. Other pieces of code may be holding *references* to that same object, and they will be able to see the field values changing. One place that references are stashed are in the fields of other objects. The chance for multiple contexts (be they object fields or other spots) to reference the same object is called *aliasing*. Aliasing can lead to remarkably tricky-to-debug mistakes in code. However, in our first application of the motto for this lecture,

aliasing can pay off in supporting faster algorithms.

OK, so let's code up some objects, which for us will be dictionaries whose keys are some fixed set of constant strings. For instance, here is an example following the informal objects we sketched above, of students and majors that contain them.

```
student1 = {“id”: 1, “name”: “Alice”}
student2 = {“id”: 2, “name”: “Bob”}
major1 = {“num”: 42, “name”: “Blah Blah Studies”,
     “students”: [student1, student2]}
```

We read and write object fields like so:

```
name1 = student1[“name”]
student1[“name”] = name1 + “, Jr.”
```

A Flat Representation of Data

For the rest of the lecture, we'll explore the choices for representing one data model, suitable for the record-keeping of a subject much like 6.009. The full set of data consists of:

- A set of students, each with an ID and a name
- A set of psets, each with an ID and a point total
- A set of grade entries, each with a student ID, pset ID, and point total

We can represent the data as a set of *tables*, much as they might be laid out in a spreadsheet. Some example data, which we also use in the lecture code:

Students:

ID#	Name
1	Alice
2	Bob
3	Charlie

Psets:

ID#	Points Available
1	10
2	20

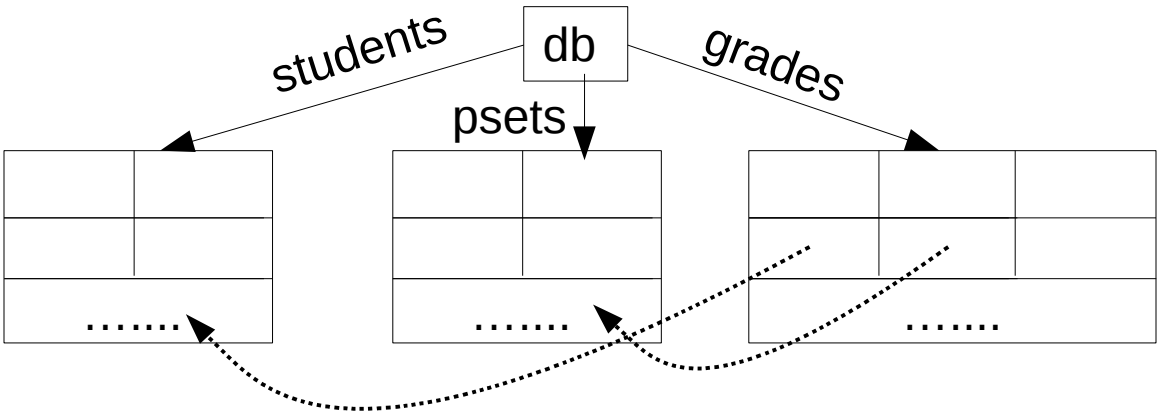
ID#	Points Available
3	30

Grades:

Student ID	Pset ID	Points Earned
1	1	10
1	2	18
1	3	25
2	3	15
3	3	10

Though the terminological details aren't important for us in 6.009, we'll mention that this way of thinking about data is called the *relational data model*. In that perspective, where we lay out data as *rows* in tables (or relations), some columns of some tables are called *keys*, because they can be used as unique identifiers for their rows. In our example here, the ID field serves as a key in both the student and pset tables (though it's a different set of IDs for each; both a student and a pset have ID 1, yet we are not talking about a student who was bitten by a radioactive pset and gained pset powers).

Our first realization of this data model in code comes by taking the relational model quite literally, storing the data set as an object with one field per table above. Each table is represented as a list of objects. Here's a graphical representation of this choice:

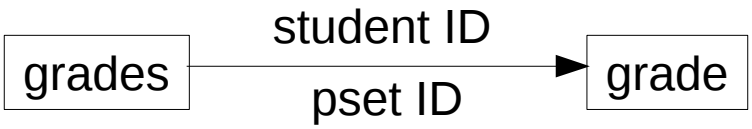


Using Dictionaries Over Keys

With the list-based representation, some simple operations can be rather slow. For instance, looking up a student's name by ID may take

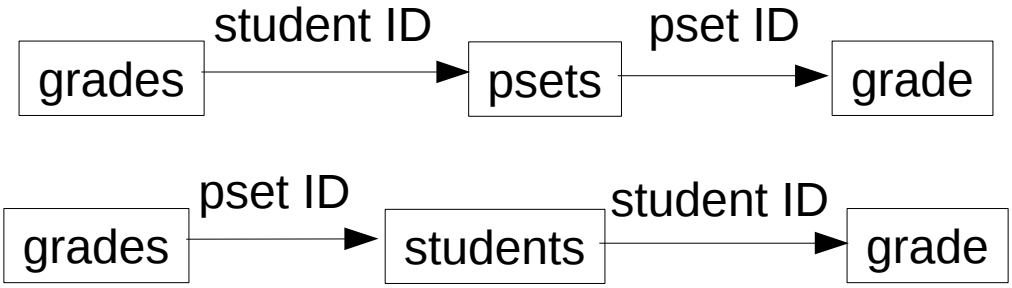
time proportional to the total number of students. The problem is that we implement the lookup by iterating over all students, checking for a matching ID. In an earlier lecture, we already saw the idea of instead using a dictionary keyed on the field that we use in lookups.

There are still some interesting choices to make after deciding to encode each table as a dictionary. For the grades table, we have at least 3 reasonable choices for dictionary organization. Perhaps the most obvious is to make grades a dictionary keyed on pairs of student IDs and pset IDs:



With this representation, it is very fast to look up the grade of a particular student on a particular pset. However, we are in trouble if we want to, say, look up all grades of one student or all grades on one pset, since the basic dictionary lookup operation only works if we commit to a pair of both IDs.

The other two dictionary choices use nested dictionaries, visualized like:



Which version of these is faster for which kind of listing operation (grades for one student vs. grades on one pset)?

Redundancy for Speed

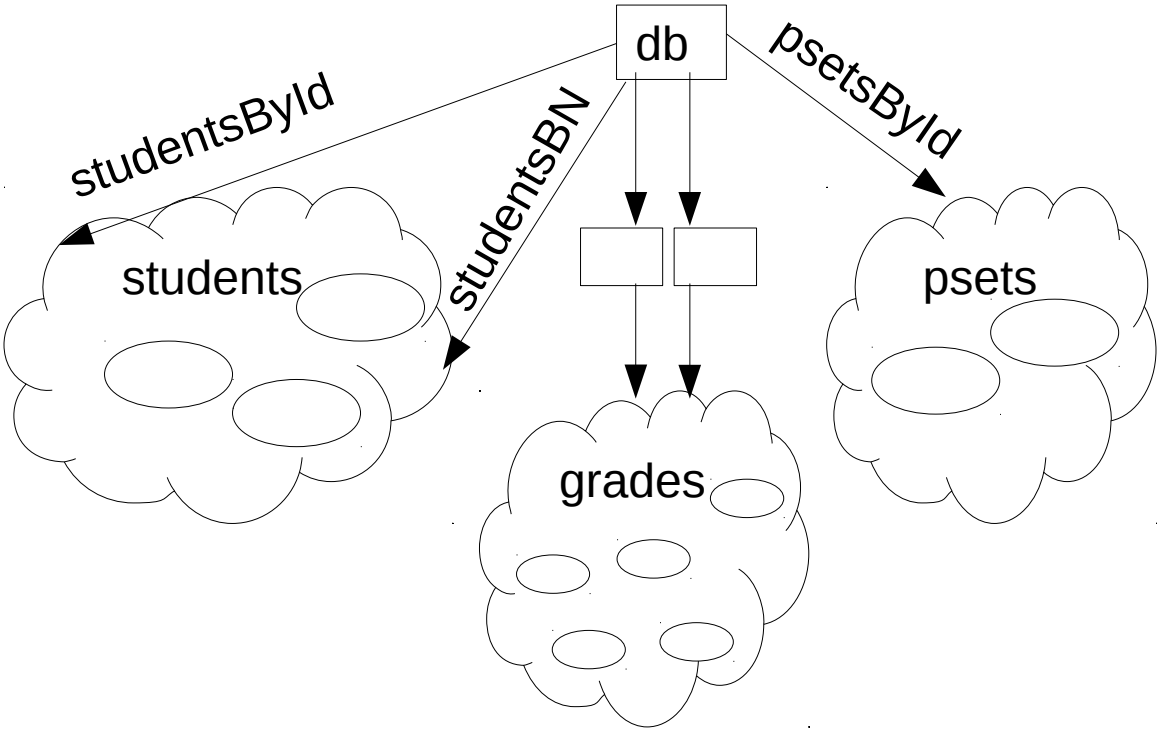
We are free to encode data in a way that is *redundant*, where, say, we could remove some fields of an object, in a way where the contents of those fields could be reconstructed systematically from the contents of remaining fields. Why bother keeping redundant information? It's often easier to code most operations in less redundant representations. However, some operations can be much *faster* with redundant data structures around.

In our class-grades example, the next representation employs two tricks to speed up operations:

1. Store dictionaries for *both directions* of the mapping between student IDs and student names.
2. Provide *both of the nested dictionary representations of grades* that we considered above, so that we can support either direction efficiently.

Exposing Mutable Objects

Another change we can make is, instead of having our operations return only primitive types, we can make them return objects, exposing aliasing strategically. Otherwise, we keep the same structure as before.

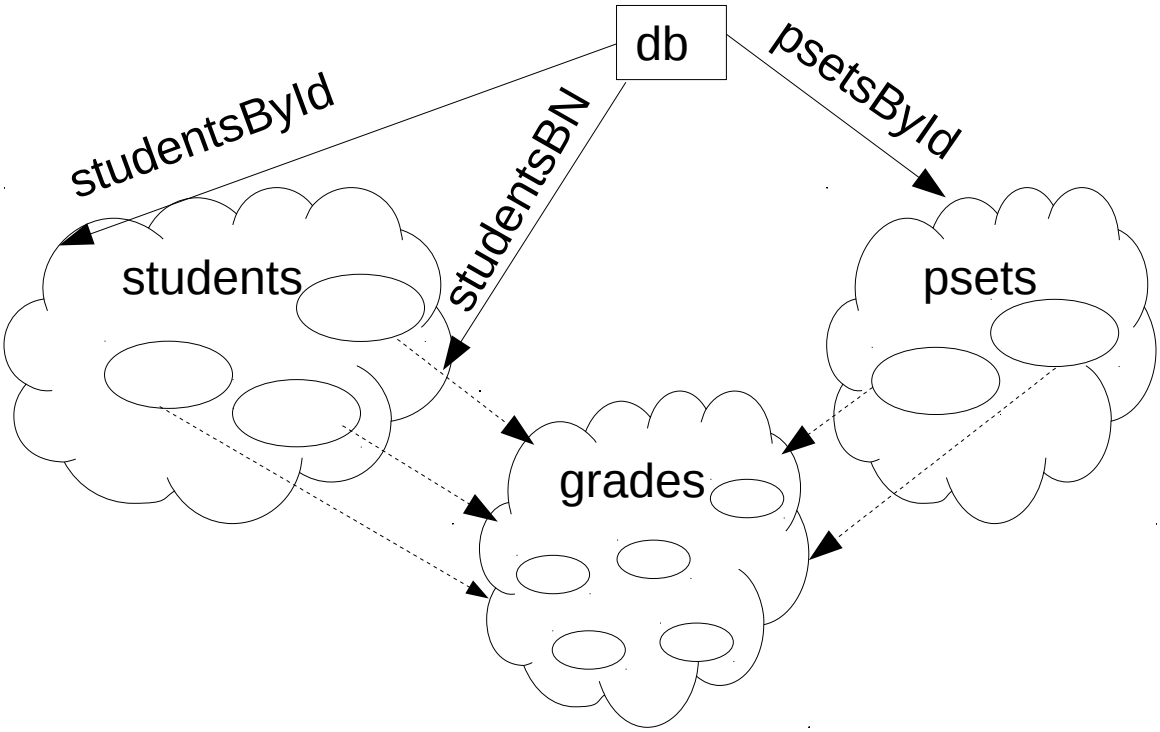


This representation strategy gets more interesting as we combine it with another one next.

Nesting Dictionaries Within Objects

We can cut down on extra dictionary lookups by giving each student *its own dictionary of grades, keyed by pset ID*. We can also do the symmetrical thing for psets, giving each one its own dictionary of

grades keyed by student ID. Now the picture looks more like this, where dashed arrows represent references via nested dictionaries:



We haven't achieved representation perfection yet, as several reasonable operations take longer than they really should. For instance, consider computing the average score on a pset. We need to loop over all entries in that pset's grades dictionary, taking time proportional to the number of students. What if we're running in a hit MOOC taking the world by storm, with a billion students?

Caching

A simple trick is to store, in each pset, the sum of all grades recorded for that pset so far. Now computing the average is as simple as dividing the sum by the appropriate number, which runs in an amount of time independent of the number of students. We have to do a little extra work each time we record a grade, but that work really is just a little, also taking time independent of the number of students.

Let's consider one more challenge in supporting fast operations. What if we want to list all of a student's grades sorted by pset number, or what if we want to list all grades on a pset in sorted order? More interestingly, what if we want to support both operations with approximately the fastest possible code? That code should run in time

proportional to the output of each operation; we shouldn't spend time on grades that aren't relevant to the question being asked.

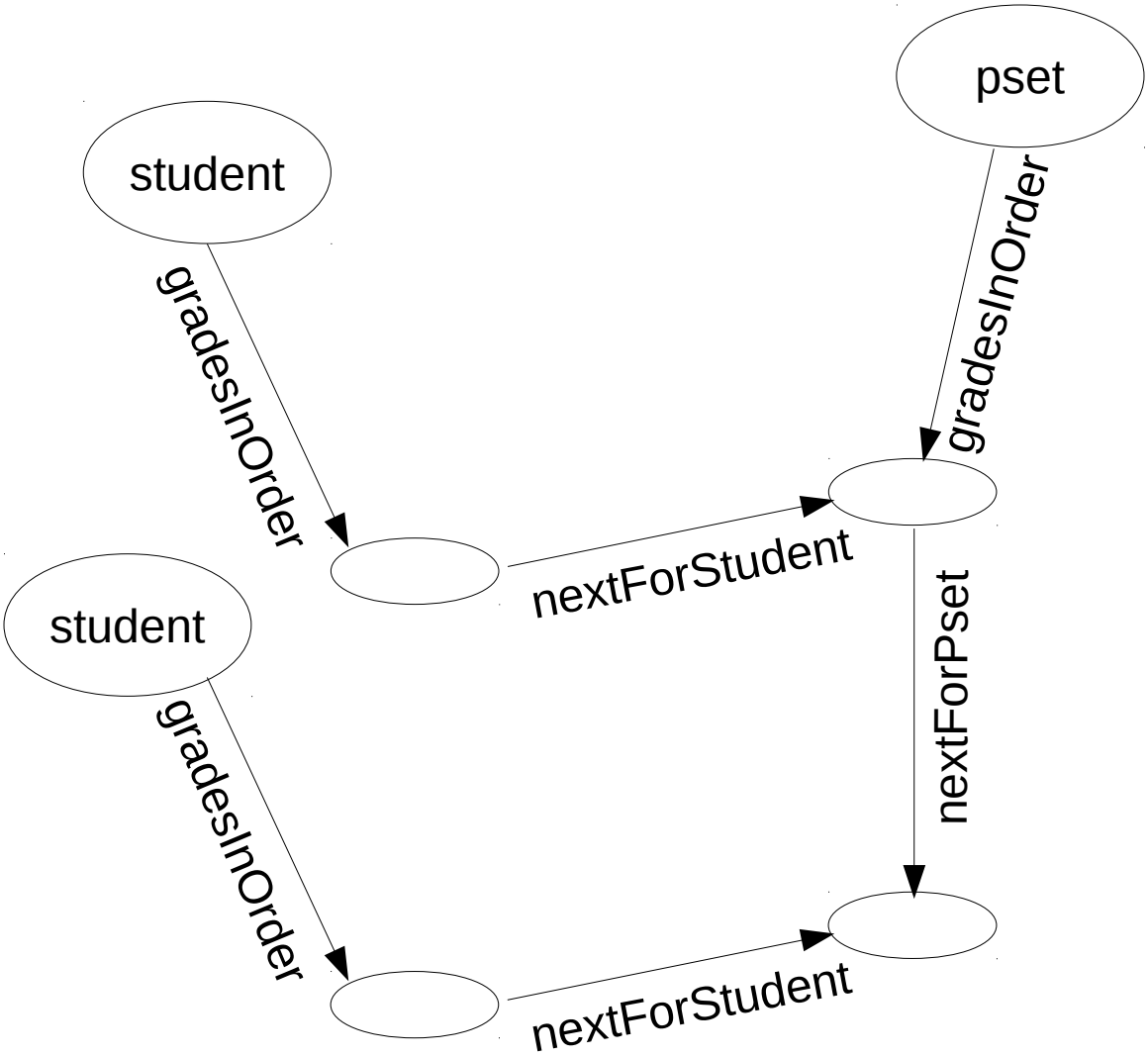
Just to make this interesting, also imagine that each grade record involves a variety of supporting data fields, so that a single grade object is rather large. We'd rather avoid duplicating the grades across nested data structures.

Putting one object in multiple linked lists

Admittedly, this last representation trick is relatively rarely the right choice. We give it here primarily as an example of how it's possible to employ so many different ideas in encoding a single data model! Usually this kind of trick would only be employed when it is essential to use as little memory as possible, which is definitely not the case for the 6.009 labs.

The idea is to make each grade object *a member of two different linked lists*: one for the grades of its student and the other for the grades of its pset. Each list is maintained in *its appropriate sorted order*. Since those orders are different, each grade has two “next” pointers, one for each list it belongs to.

Here's a sketch of how the multi-list representation works:



Now it's easy to follow each linked list in order, to get the results of each of our two challenge operations. What gets significantly trickier is recording a new grade: we need to update two linked lists that share the new grade object.

Summary of Rules of Thumb

In trading off coding simplicity vs. running time of operations, we have implicitly been using a table like this one, telling us how much each kind of basic operation costs.

Operation	Cost
Accessing field of object	Instant
Accessing key of dictionary	Almost instant

Operation	Cost
Iterating through full (Python) list	Proportional to list length
Iterating through full dictionary	Proportional to dictionary size
Iterating through full linked list	Proportional to list length (only counts nodes that we visit!)

Those are awfully vague descriptions, because we are trying to avoid diving into the 6.006 material on precise performance analysis!

README.txt has a summary of all the implementations and their structure and efficiency characteristics.