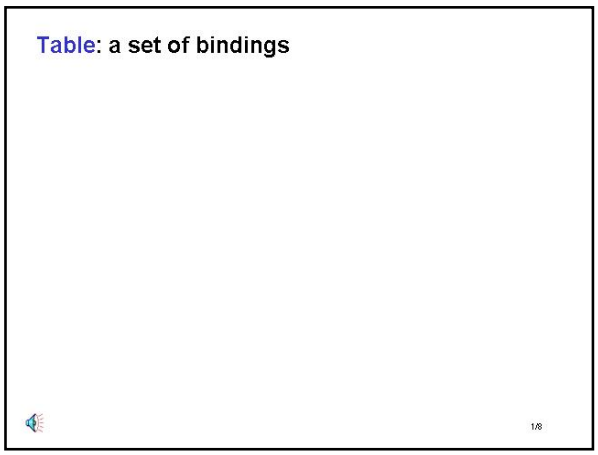
6.001 Notes: Section 10.1

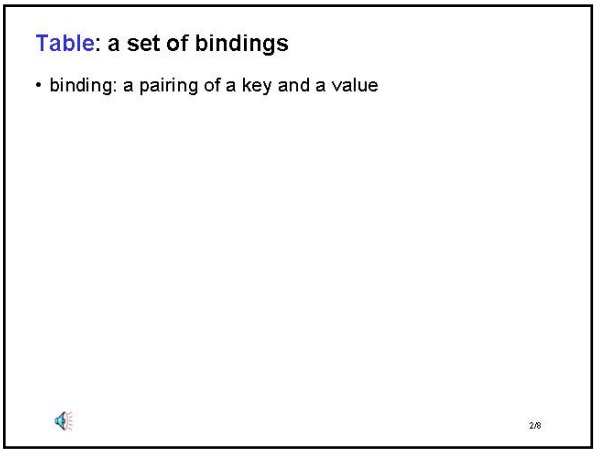
Slide 10.1.1



Over the past few lectures, we have introduced the concepts of data abstractions, types and aggregate structures. In this lecture, we want to explore the idea of pulling those pieces together to explore data abstractions in more depth. To do this we will examine a couple of particular examples, but we want you to step away from these details to see how in designing any data structure we have a tradeoff to consider. That tradeoff is between constructing concrete structures that are very efficient but open to misuse versus abstract structures that nicely separate the implementation from its use but which may suffer in efficiency as a consequence.

This may sound a bit odd. The typical instinct for a less experienced programmer is to opt for efficiency. But in fact, if one is writing code for a large system that will be used over extended periods of time and will involve components from a large number of coders, then robustness, modularity and ease of use and maintenance are typically much more important that speed.

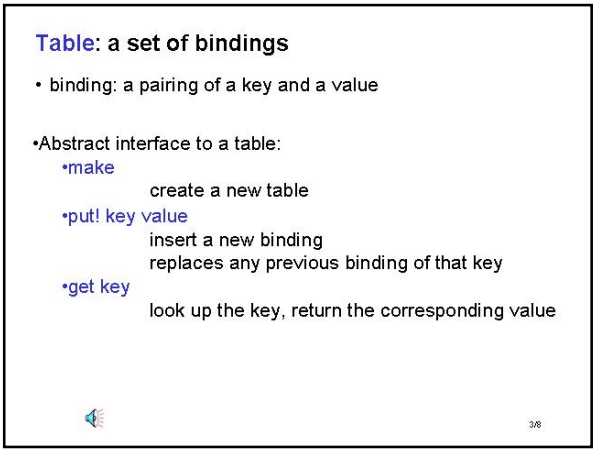
In this lecture we are going to do an initial exploration of these ideas. We are going to examine three different kinds of data structures that move from concrete to abstract, culminating in the creation of a table data abstraction. We will also see how design choices influence the implementation of an abstract data structure, with different kinds of performance. And we will see that the fundamental issue in designing abstract data types is using this methodology to hide information, both in the **types** and in the **code**.



Slide 10.1.2

So here is where we are headed. We would like to build an abstract data structure of a table. Conceptually a table is just a collection of bindings, and we are free to consider different ways of structuring that collection. What's a binding? It is just a pairing of a **key** and a **value**, or in other words, the key tells us the entry into the table, or how to find the binding in the table, and the value is the thing actually associated with that key.

Slide 10.1.3



We can define the behavior that we want a table to have, without having to consider the specifics of the implementation. In particular, we want the following abstract interface between the user of a table and the actual table itself.

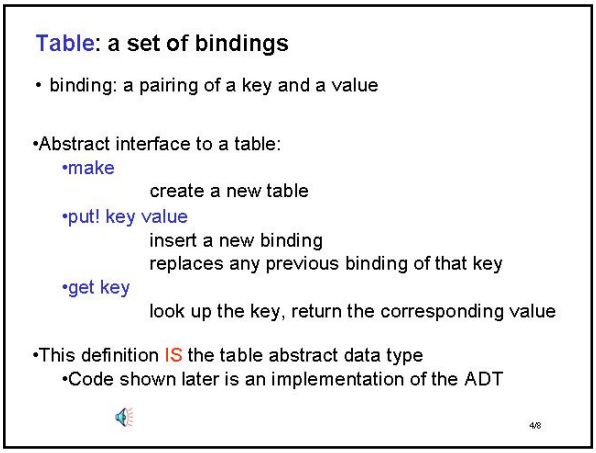
First, we need a constructor for building a table. We will just call that constructor, `make`. We ignore the details of how `make` works. We can just use `make` as a black box to construct a table for us.

We need a way of **inserting** a binding into a table. We will assume that inserting a binding replaces any previous binding for that key in the table. Note that we don't specify whether that old binding is actually removed from the table, or only if the old binding will no longer be found by procedures that search the table.

Why is this relevant? Well, notice that there is likely to be a tradeoff between efficiency in space (whether we keep old bindings around) versus efficiency in time (how quickly we can add new bindings and whether keeping old bindings in the table will affect the speed of searching the table for bindings).

Note that at this stage of our design, we have simply specified that `put!` should take a **key** and a **value**, create a binding of those two pieces of information, and install that binding into the table in such a way that searching the table for that key will find this binding and not any previous ones.

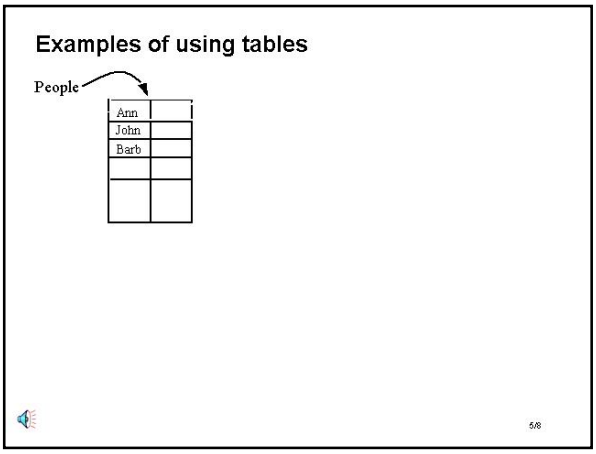
We will also need a way of getting values out of the table. Thus, we will have another abstract interface, called `get`, that takes as input a **key**, looks up the key in the table, and returns the corresponding **value**. Note again that we have said nothing about how to implement this operation, only what behave we expect as a user of tables.



Slide 10.1.4

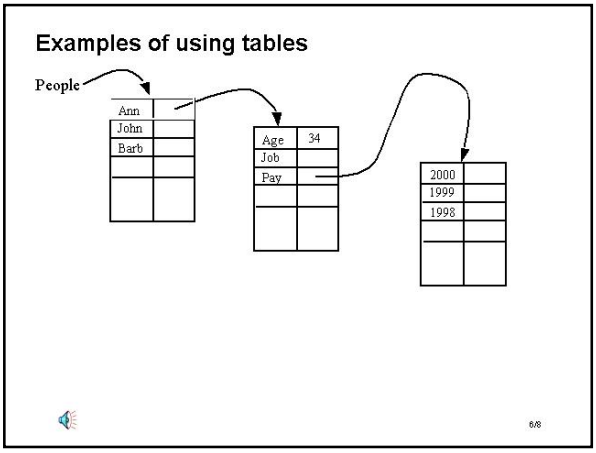
This is a **very important point**. The definition we have just provided of an interface to a table really does **define** the abstract data type of a table. From the user's perspective, using these interface abstractions is sufficient to allow them to use the table. The details (the code we are about to show) will be an implementation of this abstract data type, but there can be many alternatives that also satisfy the contract inherent in this description of the behavior of bindings in a table.

Slide 10.1.5



Before we start exploring different kinds of data structures that will lead to the implementation of a table, let's first think about why tables might be of value to a user.

An obvious example is keeping track of personnel information. For example, suppose you have just started a new "dot com" company. You will want to keep track of your employees, and a table is an obvious way to store data about them. You might start with information keyed by the names of the employees.

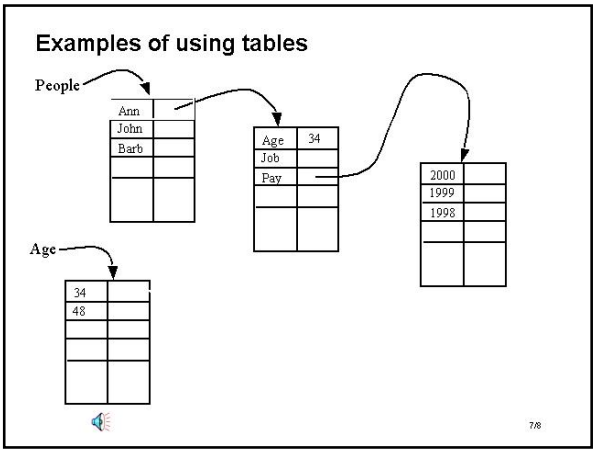


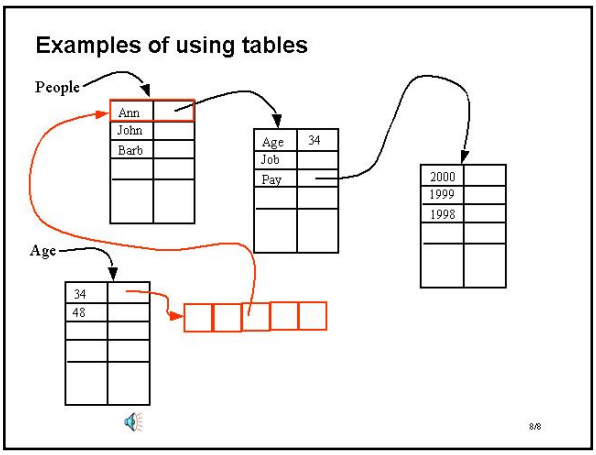
Slide 10.1.6

Associated with each **name** key, we might have another table that lists information about that person: their age, their salary, and so on. And associated with each of those pieces of information might be other tables, for example, the salary information for previous years.

Slide 10.1.7

Not only do our table data abstractions need to be flexible enough to allow for other tables to be entries in a table, we may have other ways of gathering information together in a table. For example, we might have another table that lists employees by age.





Slide 10.1.8

Clearly we can build a table where the keys are the ages of the employees. Here, however, the value associated with a key should be a list whose elements point to the entries in other tables that capture all the data about each employee.

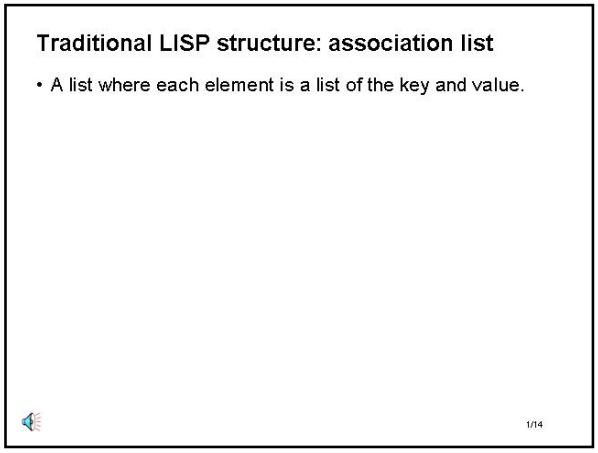
So our tables need to be very flexible in terms of what information they can store.

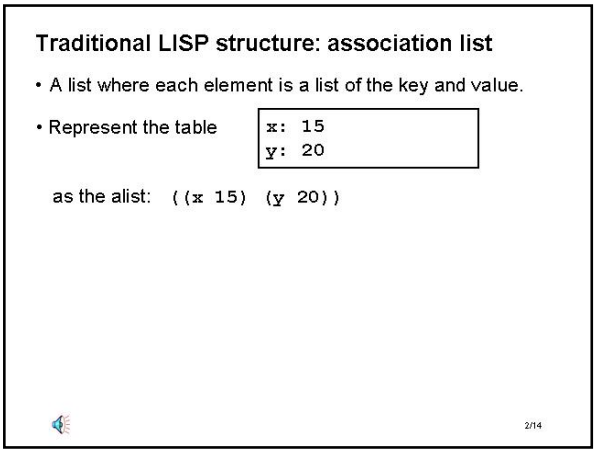
6.001 Notes: Section 10.2

Slide 10.2.1

Let's start by looking at the idea of pairing keys and values, as this is going to be the heart of a table implementation. Our goal is to see how different choices of implementation influence the behavior of the table.

A traditional structure for doing this is called an **a-list** or an association list. This is a list in which each element is itself a list of a key and a value, or in other words, it is a list of lists, all of which are two elements long.



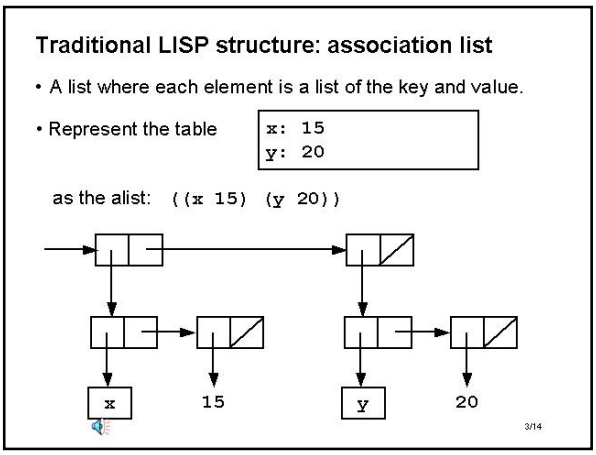


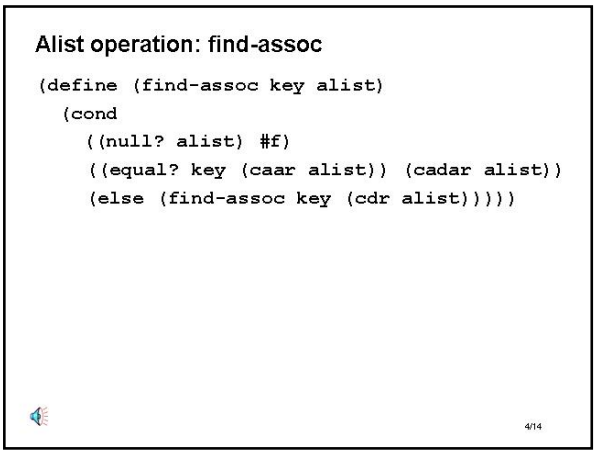
Slide 10.2.2

So for example, if we want to represent this table (and notice that this is an abstract version of a table, that is, a binding of **x** to **15** and **y** to **20**) as an association list, we can do so as the illustrated list of lists. Each of the inner lists is a two-element list of a key and a value.

Slide 10.2.3

Or to do this a little more concretely, we could represent this particular table with the illustrated box-and-pointer diagram. This illustrates the nice structure of this system, in which each element of the top level list is a binding, and each of the bindings is just a two-element list of a name and a binding.





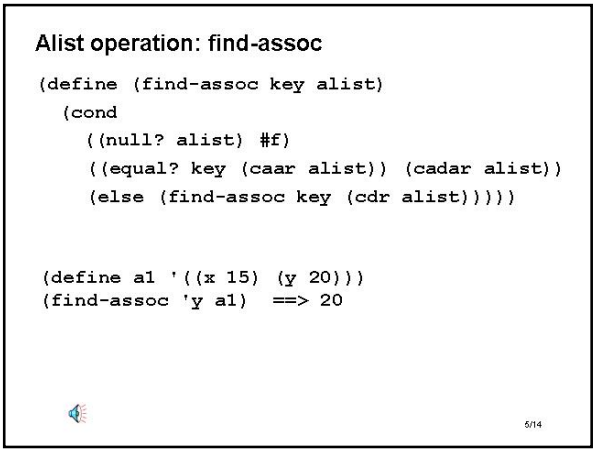
Slide 10.2.4

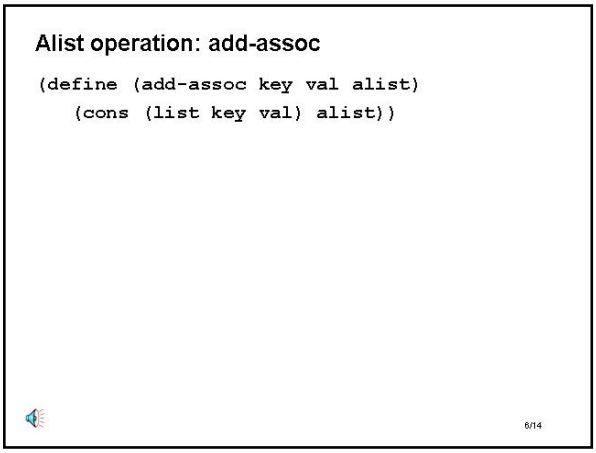
If we were to make the design choice to use a-lists as our representation for a table, then we just need to add a couple of operations. First, we will need a way to find an entry for a key in the a-list and returning the associated value.

Here is one way to do this. Notice how it walks down the list, checking the **key** against the first element of the first element of the list (this is what `caar` extracts, which you can check by looking back at the box-and-pointer structure). If we get to the end of the list, we return `false` to indicate the key is not present. Notice how we use `equal?` to check the equality of the key and the element. And notice that when we do find a matching key, we use `cadar` to extract the second element of the first element, which is exactly the value associated with this key.

Slide 10.2.5

As an example, let's give a name to our simple alist (noticd the use of the quoted list structure to create this list of lists). If we then evaluate `(find-assoc 'y a1)` (note the use of the ' to get the symbol `y` rather than any value associated with it) we get out the expected value of `20`. Trace through the code and the box-and-pointer structure to convince yourself that this is correct.





Slide 10.2.6

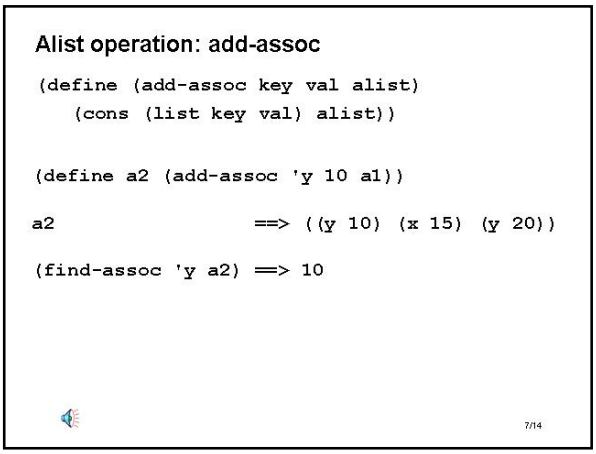
Adding a new entry into this a-list is pretty easy. We'll just "cons" it onto the front of the existing list. All we have to do is take the key and value to be associated together, put them into a list, then put that list at the front of the existing a-list. Since the a-list is a list, consing something onto the front of the list, by the closure property, gives us a new list.

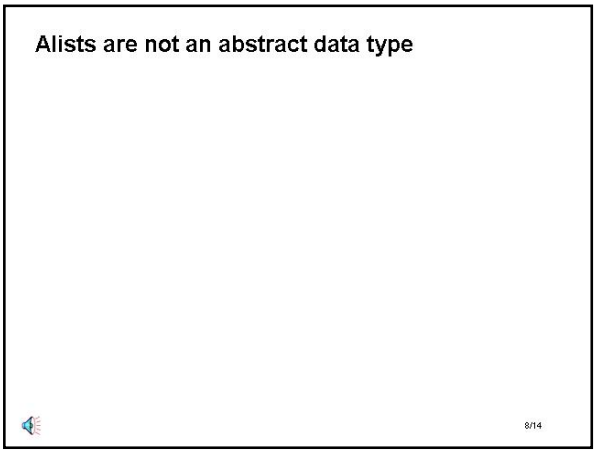
Note that in this implementation, we don't actually remove any prior binding of a key from the representation. We simply shadow the old binding with the new one, meaning that the procedure for looking up a binding will always find the new one first, since it is at the beginning of the list. So what does this do to the efficiency of `find-assoc`? Clearly the more things we add, the more things we have to search through to find a binding different from those added.

Slide 10.2.7

Nonetheless, this is a nice implementation. Thus, we can add a new binding to our old table, giving it a new name (which we need to do because we need a way to refer to the table). Now, `a2` is a list of 3 elements, with the new binding of `y` at the front and our original binding further down the list.

Looking up the binding of `y` in this a-list will give us the value `10` that is, the new binding value.



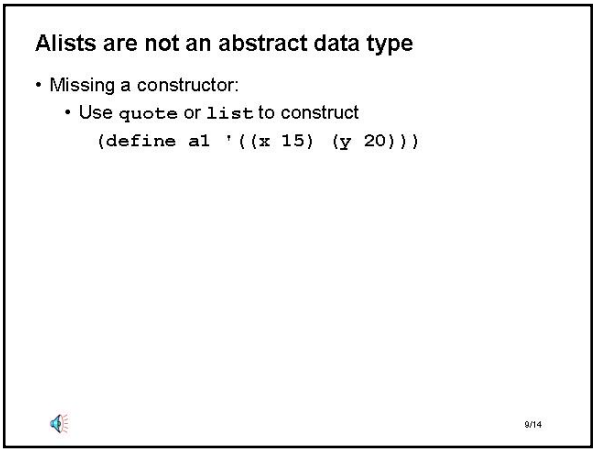


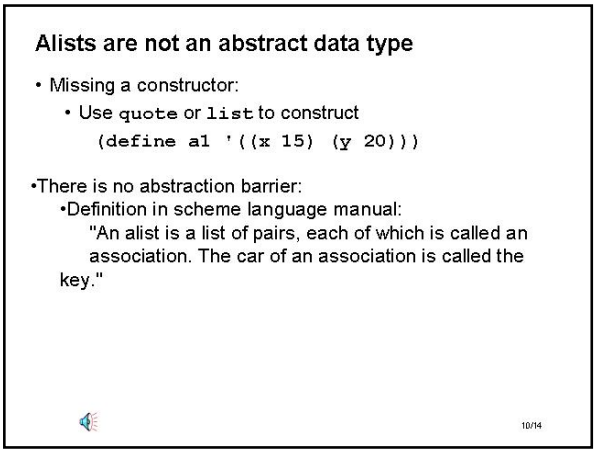
Slide 10.2.8

This seems like a reasonable implementation, but let's think about this structure. In particular, this is **not** an abstract data type. And here is why: ...

Slide 10.2.9

...first, we have no constructor. We are just building a list using list operations or quotes. This may seem like a neat efficient hack, but it has an important impact.



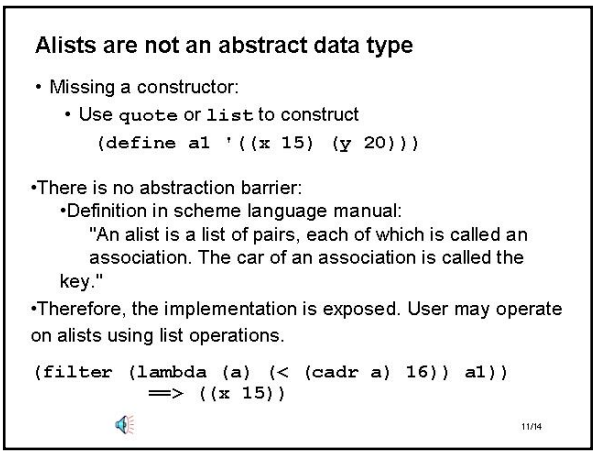


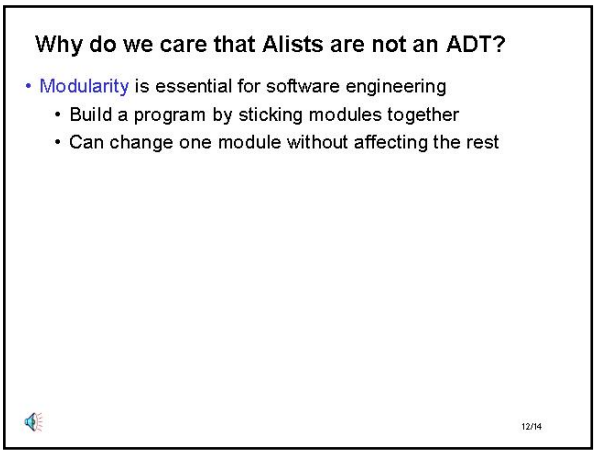
Slide 10.2.10

In particular, there is no abstraction barrier. That means there is no way of separating out the implementation and manipulating of a-lists from the use of the a-list as a table. In fact, a-lists are designed that way and the definition from the Scheme manual clearly states this. A-lists are intended to just be exposed lists.

Slide 10.2.11

As a consequence, the implementation of the table is exposed, meaning that the user can operate on the a-list just using list operations, rather than being required to use operations designed for tables. The example shown seems very convenient, using filter directly on the list, but this is a dangerous thing to do.



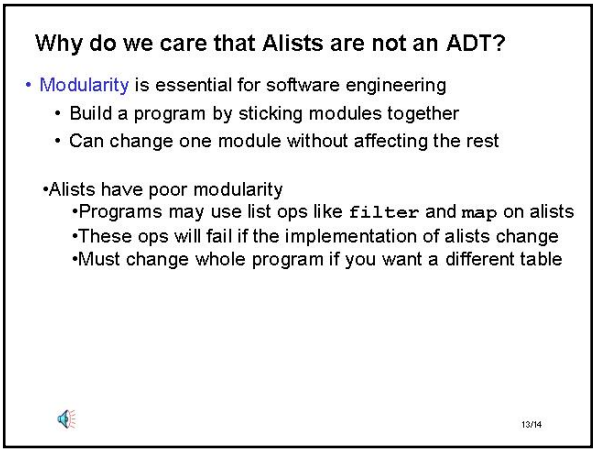


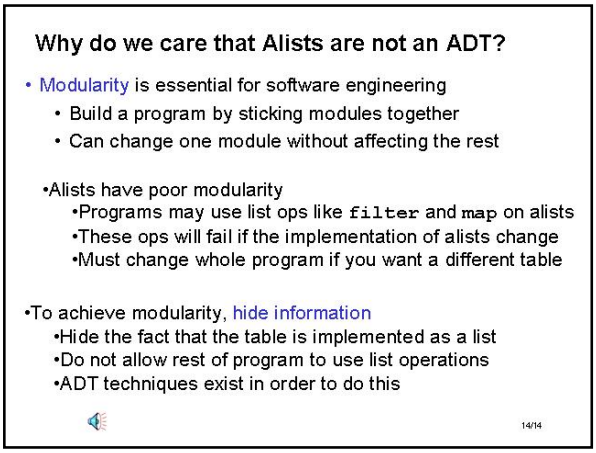
Slide 10.2.12

So why should we care that a-lists are not an abstract data type? The primary reason is the loss of modularity. Good software design always involves programs that are created from units that are easily glued together, are easily replaced with other small sized units, and can be treated as black box abstractions. If done right, we can easily change one module without having to change anything else in the system.

Slide 10.2.13

And why does that matter here? Because a-lists have poor modularity. Since we have exposed the "guts" of the implementation of tables to the outside, this means that other users can write operations using `filter` and `map` directly on the a-lists. There is no way of separating the use of the table from the implementation of the table. If we later decide to change the implementation of the table, we will be in trouble because these operations are defined based on an assumption about how things are implemented, and changing the representation will require that we find and change all the operations that manipulate tables.





Slide 10.2.14

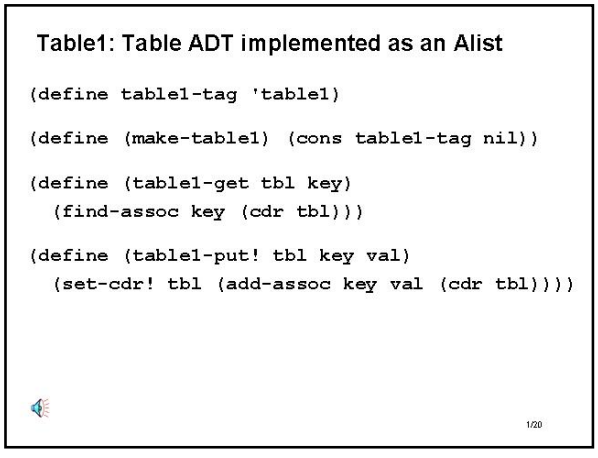
To achieve modularity, we need to hide information. In particular, we want to hide the fact that the table is implemented as an a-list from the use of the table itself.

This has two parts. The first part is that we will create abstractions for getting at the parts of the data structure (i.e. constructors and selectors) and we will insist that anything outside of the abstraction cannot use list operations but **must** go through the abstract selectors and constructors.

6.001 Notes: Section 10.3

Slide 10.3.1

So let's build on this idea. We know that we should be able to use a-lists to represent the **internal** structure of the table, but we need to accomplish this hiding of information, that is, the separation of the internal aspects of the table from the outside world.

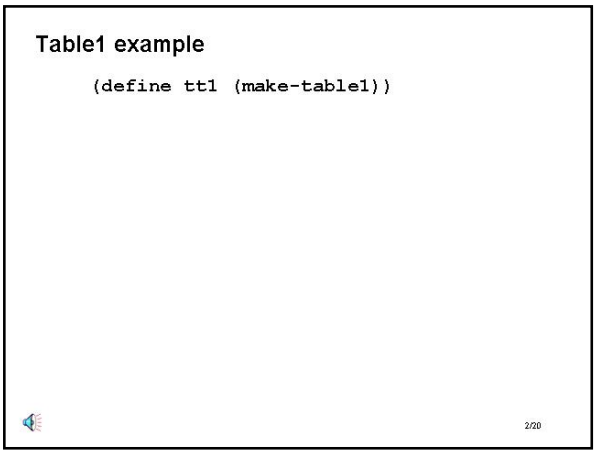


Here is how we will do that. First, we will need a constructor, `make-table1` (we give it this name in order to distinguish different versions of our table abstraction). This simply puts a **tag** at the front of a table structure. Notice how we have separately given a name to that tag, a point we will come back to shortly.

Now we can create a **get** operation on this table by: given a table, remove the tag to get the actual table implementation, and then use the procedure designed for a-lists to extract that actual entry in the table. This builds on the choice of an a-list as the actual internal representation of the table.

To put something new into the table, we can again extract the internal representation minus the tag, use the a-list operations to add a new value to this internal representation. This returns a new a-list as a value, so we can then glue the tag back onto the front of the table. We do this using an unusual operation, `set-cdr!` which we will discuss in detail in a few lectures. Think of this operation as taking the box-and-pointer structure pointed to by the value of the first argument, finding the **cdr** of that structure, and changing it to point to the value of the second argument. Don't worry about the details of this new operation. We simply will use it create a new version of a tagged table, while

preserving the association between a name for a table and the actual representation.

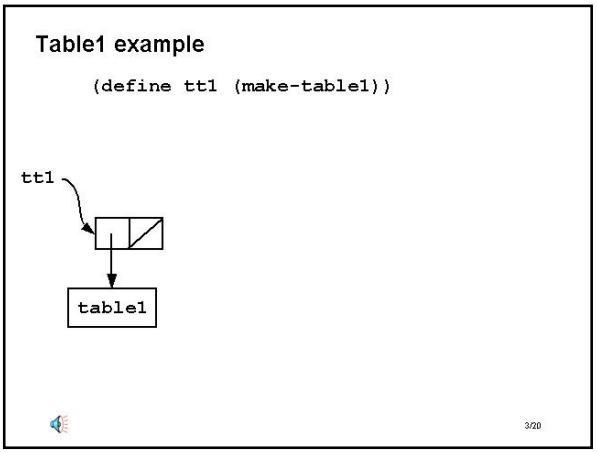


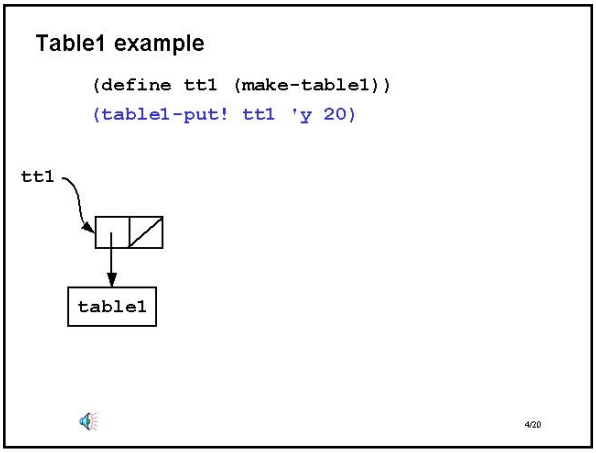
Slide 10.3.2

So let's see how this works. First, we will create a table with name `tt1` by using our constructor.

Slide 10.3.3

Here is what that looks like. The name `tt1` points to a list whose car is the symbol `table1` (our tag) and whose cdr is just the empty list (i.e. there is nothing in our a-list yet).



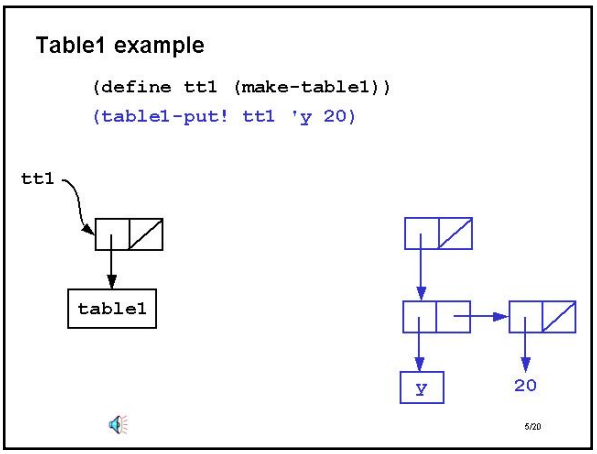


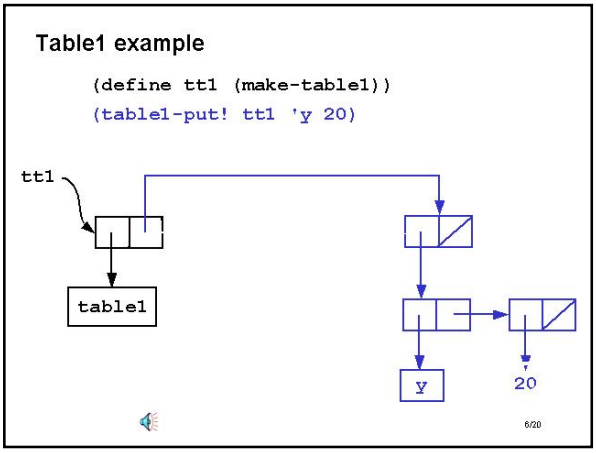
Slide 10.3.4

Now, let's put a new binding of a key and a value into our table, for example, the binding of **y** and **20**.

Slide 10.3.5

What happens when we use this operation to insert something? Remember that `table-put!` uses `add-assoc`, which extracts the a-list from the tagged representation, (in this case an empty list), and then creates a binding of the arguments (remember that is a list of two elements) and finally "conses" that onto the front of the a-list. Since this is the empty list, it creates a top level list of one element, as shown.



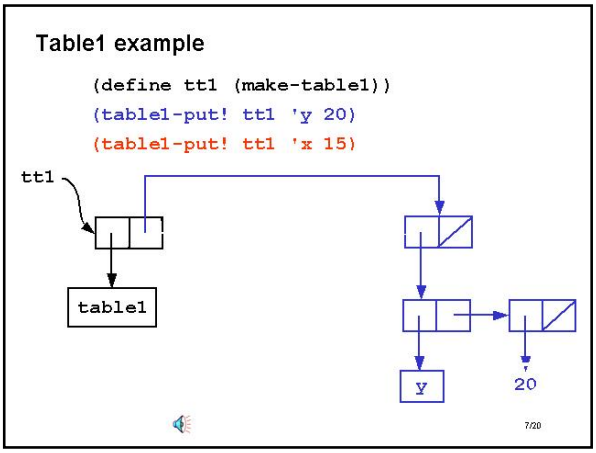


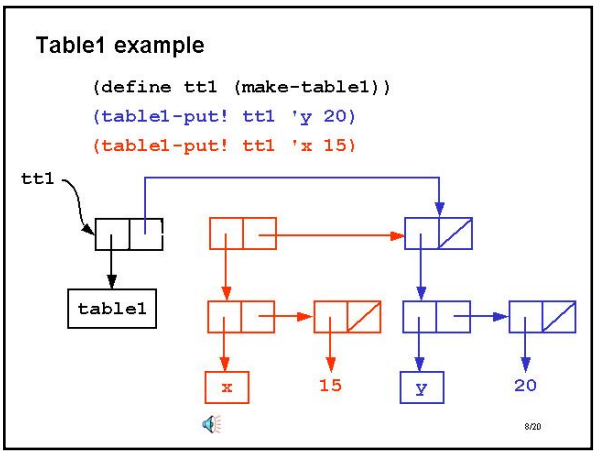
Slide 10.3.6

And then we use this new operation, `set-cdr!` to take the cons pair pointed to by `tt1`, and mutate or change it's cdr to point to this new structure, that is to the value returned by `add-assoc`. Notice what this gives us. We now have a new table. It has a tag on the front, and it has an a-list at the back, which currently has one binding within it.

Slide 10.3.7

Let's add another binding, `x` and `15`, to this same table.



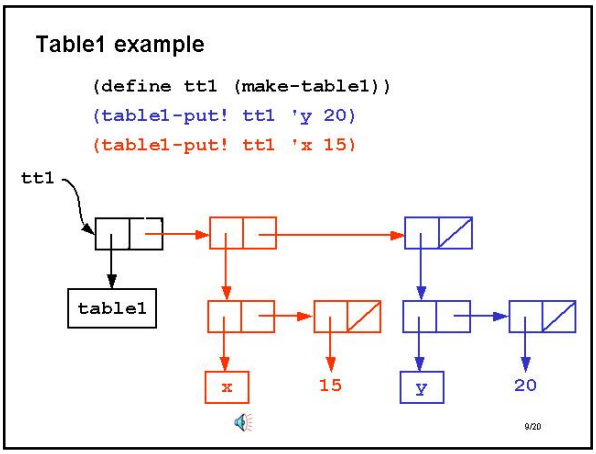


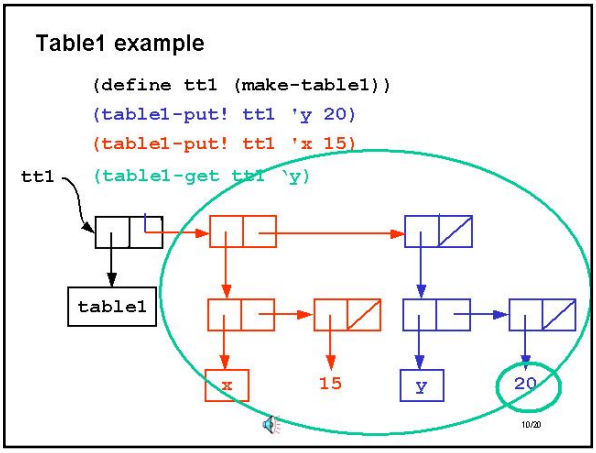
Slide 10.3.8

Looking back at the code, you can see what this should do. `Add-assoc` first takes the internal representation for the table (i.e. removes the tag) getting access to the structure shown in blue. It then creates a new binding, a pairing of `x` and `15`, creating a 2 element list, and then "conses" this onto the front of the existing a-list. This is shown in red. This new structure is our new a-list, and it has within it two bindings, each a separate 2 element list.

Slide 10.3.9

Then we once more use the `set-cdr!` operation to take the pair pointed to by `tt1` and change its `cdr` part to point to the value returned by `add-assoc`, i.e., this new structure. This gives us a new a-list associated with this table.



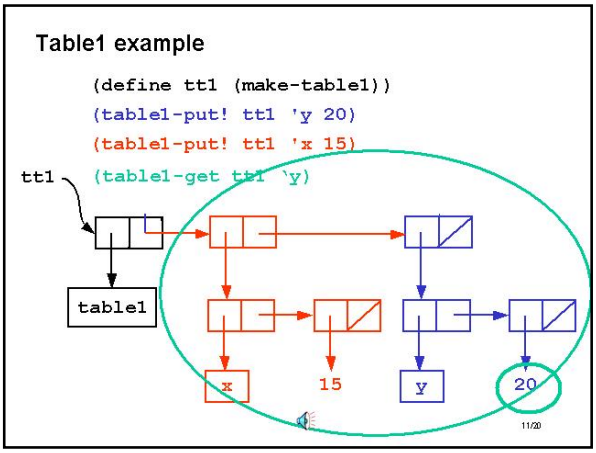


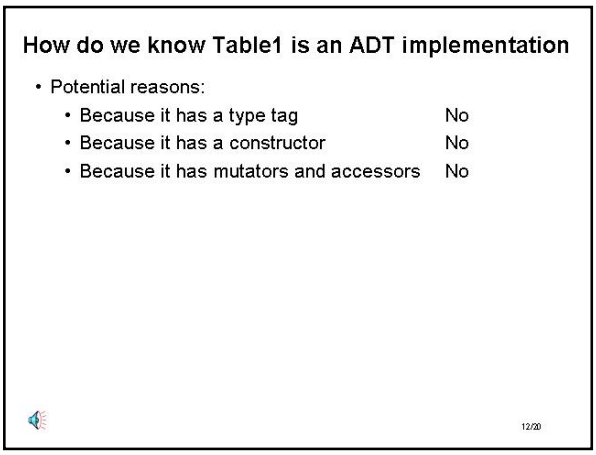
Slide 10.3.10

If we **get** a value out of this table, say the pairing associated with the key `y`, then `table-get` does a very similar thing. It removes the tag, getting the pointer to the actual a-list (shown in light blue), and then uses `find-assoc`, the operator associated with the a-list, to find and return the value associated with this key.

Slide 10.3.11

So this gives us a simple implementation of a table. It has a tag at the front to identify it, it has some constructors and selectors to get at the internal representation of the table, which in this case is an a-list, pairing up keys and values.





Slide 10.3.12

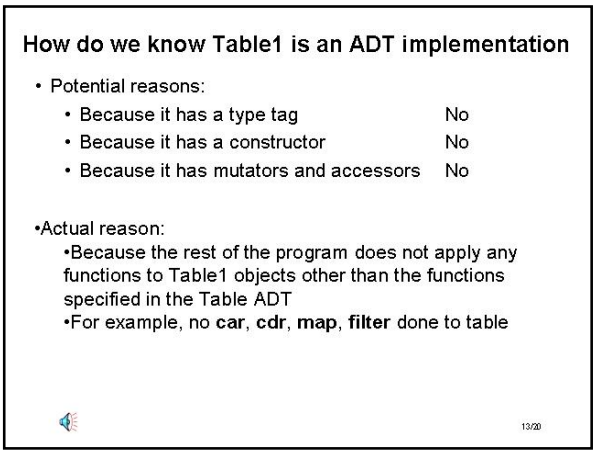
So what is it that makes this an abstract data type? Is it the fact that that it has a tag? While this is careful programming, it does not suffice to create an ADT.

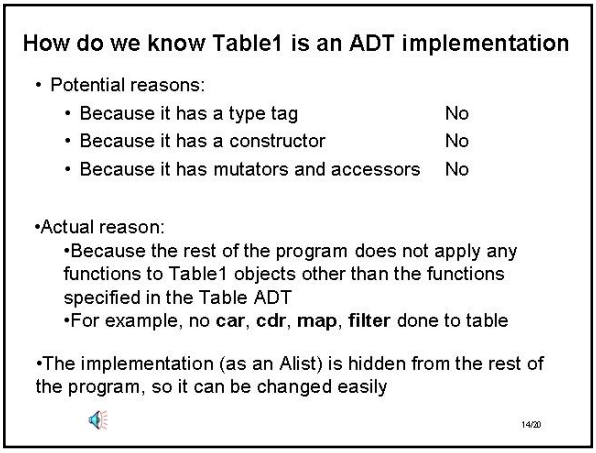
Is it the fact that it has a constructor? While having a distinct interface is good programming, this also does not suffice to create an ADT.

Similarly, the fact that it has accessors or selectors, and mutators (the `set-cdr!`) is useful, but not sufficient to define an ADT. In fact, a-lists also had these properties, but we argued that they were not ADT's.

Slide 10.3.13

In fact, the key issue is **the isolation of the abstraction from its users**. We can't use any list processors, like `car, cdr, map` or `filter` directly on the table.



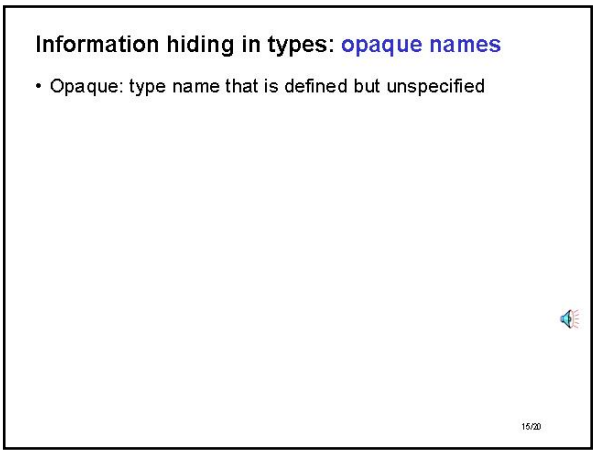


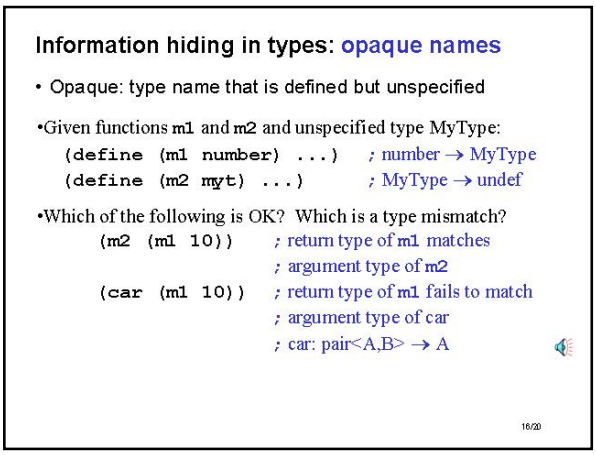
Slide 10.3.14

This is because the representation for table is not an exposed a-list. We have hidden the internal representation for the table from the users or consumers of tables. As a consequence, this abstraction barrier would allow us to freely change our internal representation of a table, without requiring a user to make any changes to their code that uses tables. This is the **key** point. We have hidden away the a-list from the user. If we change our choice of representation, nothing will have to change on the outside.

Slide 10.3.15

Because this is a key point, we want to generalize it. By creating a distinct data structure type for tables, we have in essence hidden information behind a name. We have made that name **opaque** to the user.





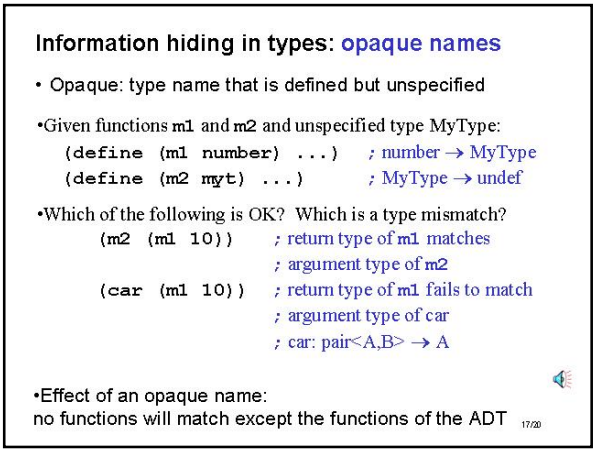
Slide 10.3.16

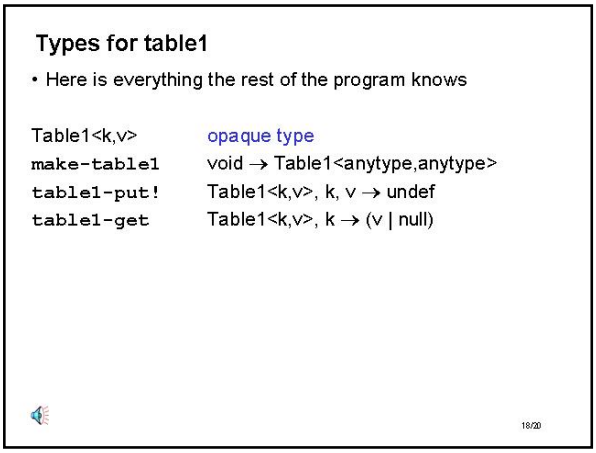
What does this mean? Well, it says that by creating a new type in this way, its name hides the details. For example, suppose I give you a new data type, and simply tell you that it exists, but not anything about its details. For example, lets call it `MyType`. Further, lets suppose we have two procedures with the indicated type contracts. `M1` maps a number to one of these objects of type `MyType`, and `m2` takes one of these types of objects as input, and does something to it. Given **just** that information, we can ask the following questions.

Which of the two indicated expressions is acceptable? Clearly the first expression is fine. The type of object returned by `m1` is exactly what is expected by `m2`. Even though we know nothing about the type, the contract (or the opaqueness of the name) allows us to remove those details from consideration, and just consider the overall behavior of the procedure. On the other hand, the second expression should "bomb", because here the type of object returned by `m1` is not appropriate for a procedure like `car`. We have no way of knowing if the returned value is a pair or not.

Slide 10.3.17

In essence, the opaque name and the infrastructure hidden behind it, guarantee clean performance for this particular data type. As a consequence, we can change that infrastructure, without affecting anything on the other side of that opaque name.





Slide 10.3.18

So, for tables here is everything that a user needs to know.

First is the opaque type name, a thing called `Table1` that has inside of it two arguments of types k and v. Or said another way, this type is created out of pairings of k's and v's, but we haven't said anything about how this is done.

Our constructor for this type takes no arguments as input, and creates an instance of this type of object. Note that the elements paired within this table can be of any type.

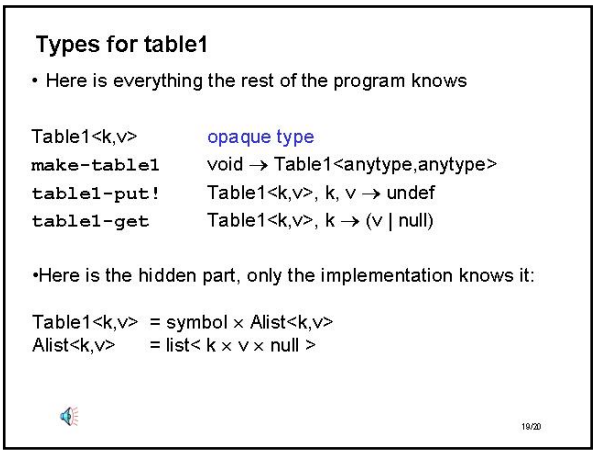
Our operation for putting things into the table takes a table object, and a k and a v as input. It doesn't return any value, since its role is to change the bindings within the table object.

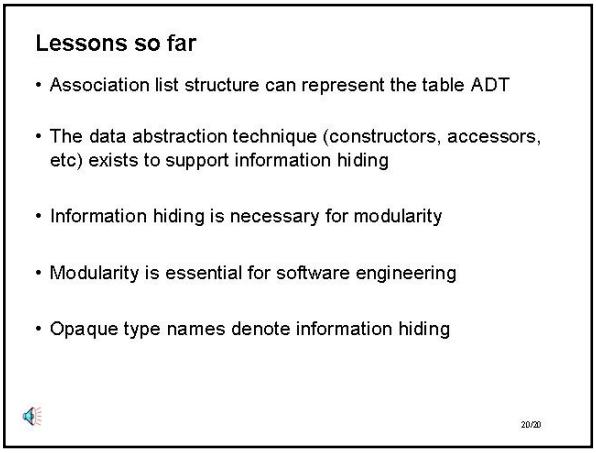
And our accessor takes one of these abstract table objects, and a k and either returns the associated v or the empty list to signal that no binding was found.

Note again, this is everything that a user of tables needs to know. And **nothing** is said about a-lists or cons pairs or anything else. This simply defines an interaction between the abstract data type and the elements within it.

Slide 10.3.19

Hiding below that abstraction barrier, or if you like behind that opaque name, is the actual implementation. Here we have made a choice. Specifically, we have decided that a table will be a pairing of a symbol (for our tag) and an a-list (for the actual bindings). Note that we can alsor provide a formal specification of an a-list as a list of elements each of which is a 2 element list, as shown.

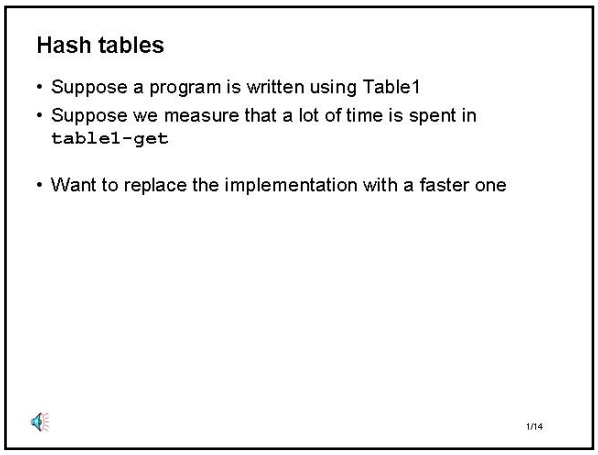


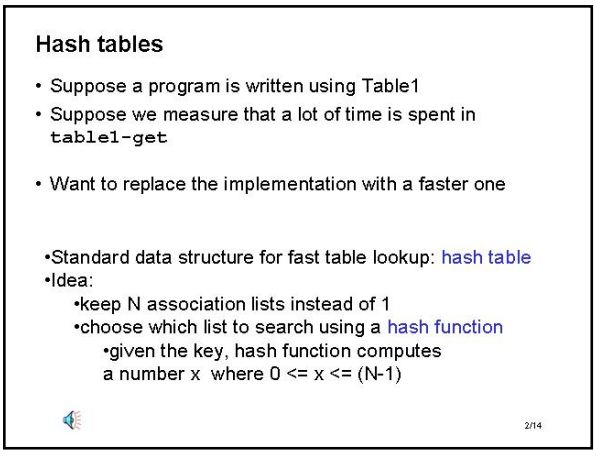


Slide 10.3.20
Here is a summary of the key messages from this part of the lecture.

6.001 Notes: Section 10.4

Slide 10.4.1
Now let's look at how the data abstraction allows us to alter the internal representation of tables, without affecting any consumer of tables. This, after all, has been our key point all along.
To motivate this, suppose we build a table using our existing abstraction, but then observe that most of the time spent in using the table is actually spend in the `get` operation. In other words, as you might expect, we spend most of our time retriveing information from the table, rather than putting things into the table. If this is the case, it would be nice if we could devise an implementation in which `get` operations are very efficient, while preserving all the contracts of the table.



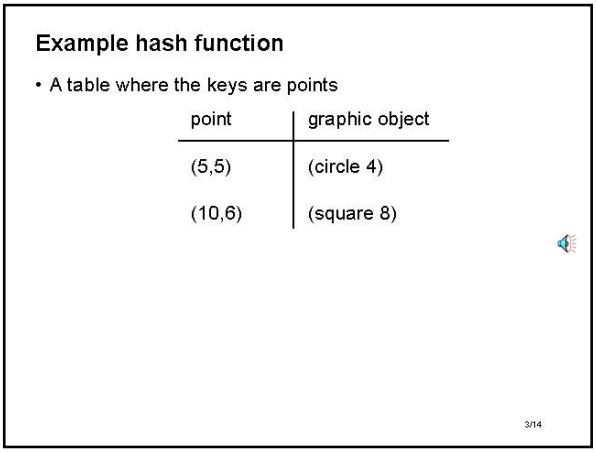


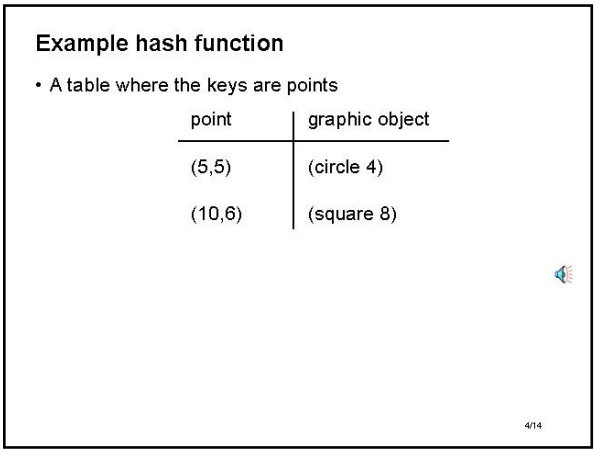
Slide 10.4.2
In fact, there is a standard data structure for fast table lookup, called a **hash table**. The idea behind a hash table is to keep a bunch of association lists, rather than a single one. We chose which association list to use, based on a function called a **hash function**. That hash function takes our key as input, and computes a number between 0 and the number of a-lists that we are using, and that number will tell us which a-list to use to look up a value.
You should already begin to see why this idea should lead to a faster implementation. Rather than having to search one big a-list to find a pairing, we should only have to search a much smaller a-list. This should be faster, so long as the cost of computing the hash function is not too large.

Slide 10.4.3

What does a hash function look like? As noted, it should take as input a key and compute a number between 0 and N, for some fixed N.

As an example, suppose we have a table where the keys are points, i.e. **x** and **y** coordinates of points in the plane, such as the points we might use in a graphics display. Associated with each point will be some graphic object that passes through that point.





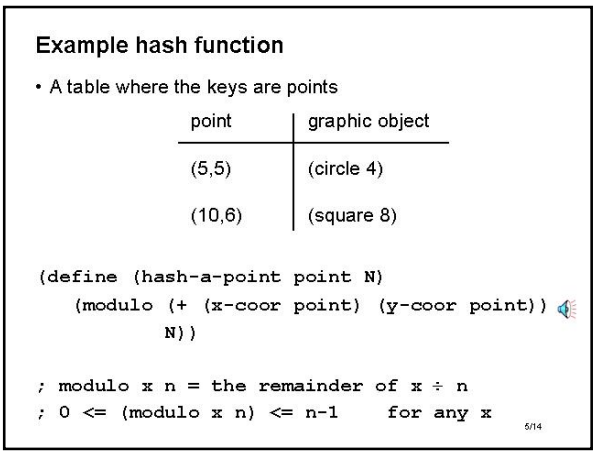
Slide 10.4.4

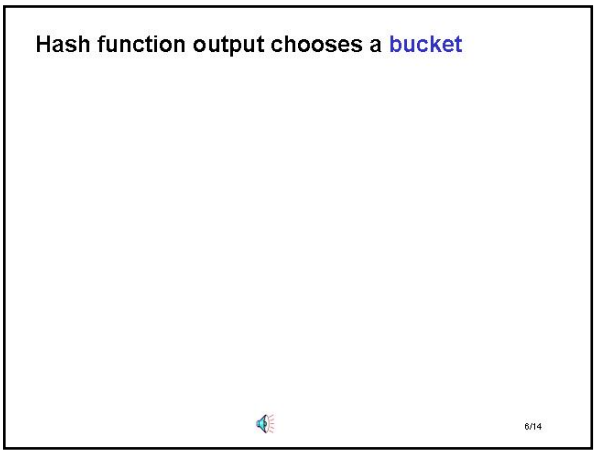
For example, if in my graphics system includes I want to move some object on the screen, I would like to know which points are covered by that object, as I am going to have to redraw those points. A nice way to do this is to take a point and find all the objects that cover that point, so I want to use the point as my key. So how could we design a hash function that takes points as input, and returns a number between 0 and N.

Slide 10.4.5

Here is a particular example. Given a point (represented by some data abstraction) and a specified number N, we can add the x and y coordinates of that point and find the sum's remainder modulo N. Remember that this means finding the remainder of the number after dividing it by N.

Note that a **good** hash function is one that evenly distributes its keys among the values between 0 and N. For this case, if the points are uniformly distributed in the plane, the result will be a nearly uniform distribution of output values of the hash function.



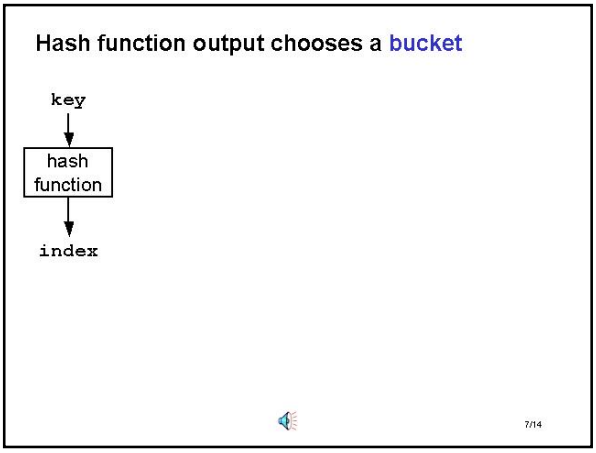


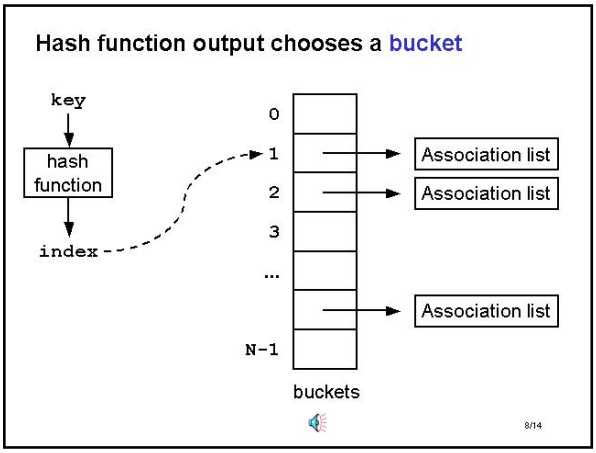
Slide 10.4.6

So a hash function basically chooses a bucket (out of a fixed set of buckets) into which to put an object or from which to extract an object.

Slide 10.4.7

The underlying idea is that given a key, we first apply the hash function, which computes an **index** (a value between `0` and `N-1`, inclusive).



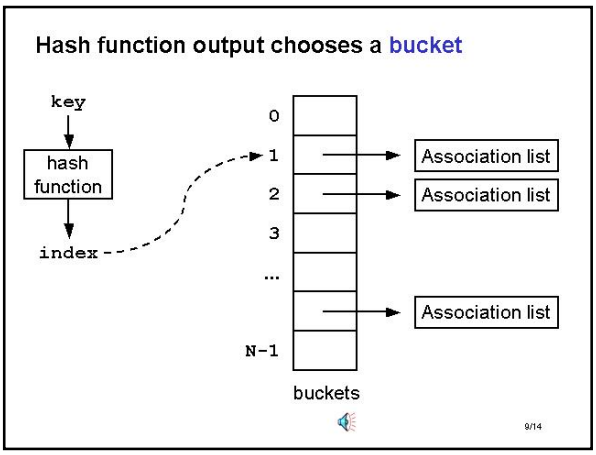


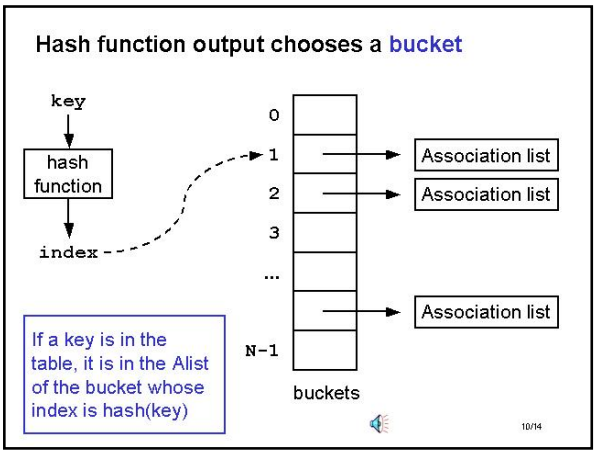
Slide 10.4.8

That index then tells us which bucket, or which set, of things in which to look. As we said, if the hash function is well designed, then on average we will have to do `1/N` of the work that we would normally have performed in the straightforward implementation. This assumes, of course, that computing the hash function is not very expensive. Otherwise we would lose the benefit gained by reducing the actual search through the table.

Slide 10.4.9

Note that we haven't yet said anything about how the buckets are organized. While this has the appearance of a list, we are free to choose other representations, as we will see shortly.



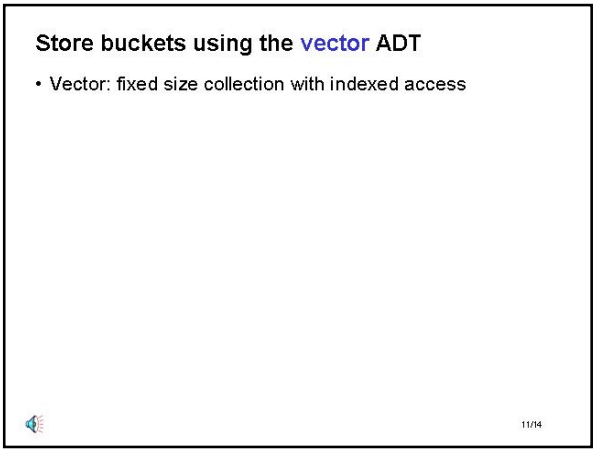


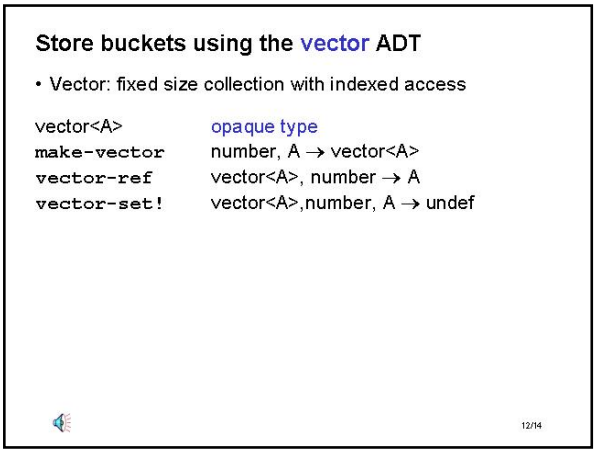
Slide 10.4.10

The main idea is that if the key is in the table, then it is going to be in the association list whose bucket is indexed by applying the hash function to the key.

Slide 10.4.11

To actually build a hash table, we need an efficient way to gather the buckets together. We are still going to represent the contents of each bucket as an a-list, but since we have an ordered sequence of buckets, we would like an efficient way of structure them. For this, we will introduce another common ADT, a vector. This is a fixed size collection, where access is determined by an index or number. This is a bit like a list, but with two important distinctions. First, the size of the vector is fixed at construction time. And second, we can retrieve the contents of an entry in the vector in constant time, whereas for a list, retrieval was linear in the size of the list.



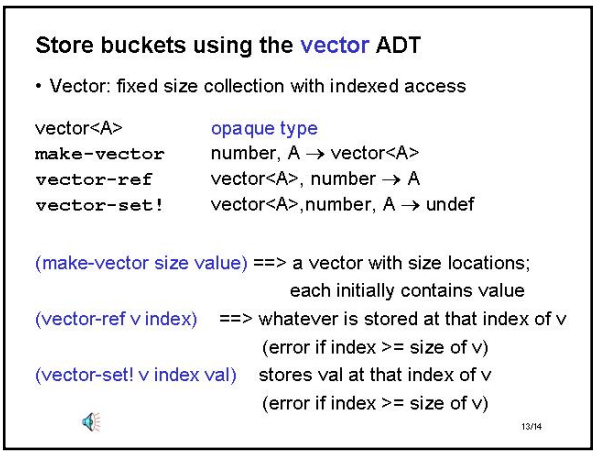


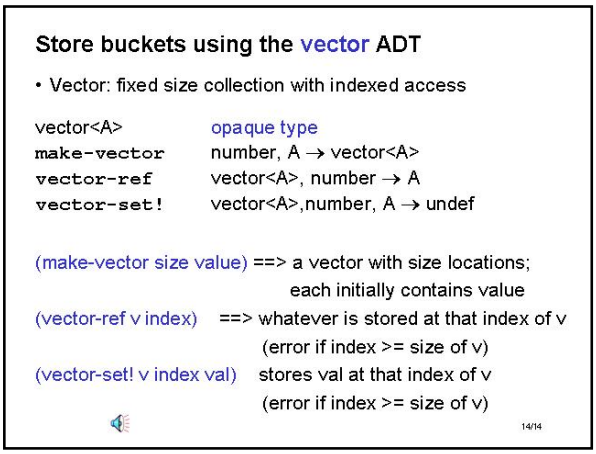
Slide 10.4.12

With that idea in mind, here is the contract for a vector. As before, the type is opaque in order to hide the details. We will just denote the type of the elements are some arbitrary type, A. Associated with this abstract data type is a constructor, and two operations on vectors: one that gets an element of the vector, and one that changes the contents of a vector at an index.

Slide 10.4.13

Examples of the kinds of operations we would expect to see on vectors are shown at the bottom of the slide. Note the error conditions if we try to go beyond the bounds of the data structure.



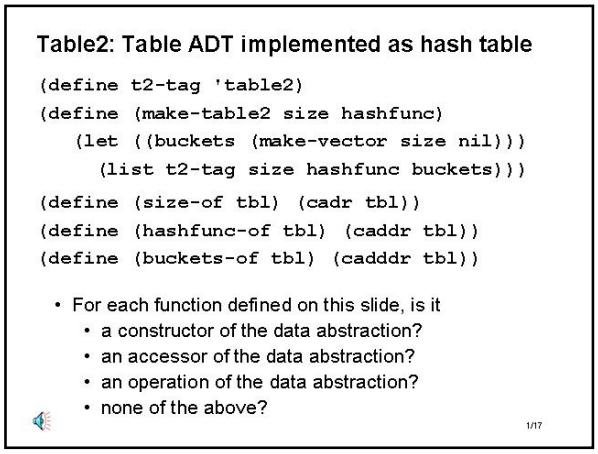


Slide 10.4.14

This then defines our new data type, a hash table. We have built an implementation of it on top of another common data type, a vector. We expect the hash table to provide a different kind of behavior as it stores and retrieves objects. We now want to turn to the question of how we can use hash tables to change the implementation of our more general tables.

6.001 Notes: Section 10.5

Slide 10.5.1

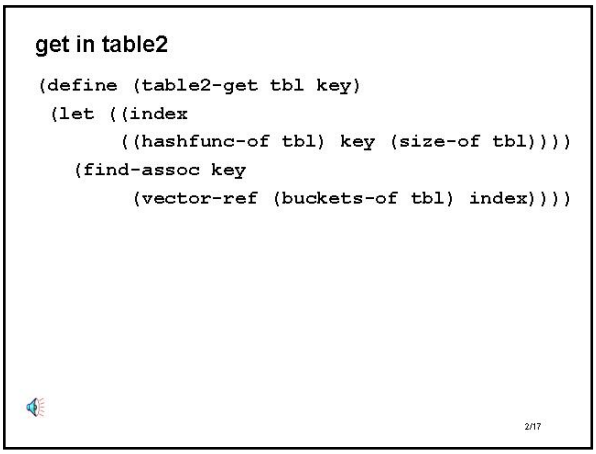


Given our abstract data type of a hash table, let's see how we can use it to change the behavior of our more general table abstraction. Our goal is to see how we can replace the original implementation with a more efficient one, without requiring any changes on the part of procedures that use tables as an abstraction.

The main change is in how we construct a table. As before, we will use a tag to identify the table, but now when we construct the table, we will specify two things: the hash function we are going to use within the implementation, and how big a hash function we want (i.e. the range of values of the hash function or equivalently the number of buckets in our representation).

Notice what we do here. We list together the tag, the size, the actual hash function, and the set of buckets stored as a vector. Of course, associated with this will be the appropriate set of selectors for getting out the various pieces of the structure.

Something to think about, by the way, is for each procedure on this slide, which kind of procedure is it?



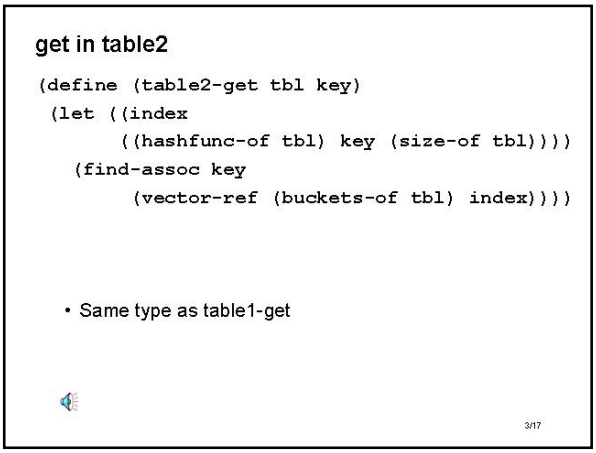
Slide 10.5.2

With this change, how do we implement `get`? By contract, we need the same inputs to the procedure, a **table** and a **key**. Here, we will get the hash function out of the table abstraction, and apply it to the key and the size of the hash function (also extracted from the abstraction). The result will tell use in which of the buckets to look. The `let` form gives us a temporary name for that `index`. Then, just as before, we can use our association list operation to find the pairing with the key, now only lookking in the association list stored at that index in the set of buckets.

Slide 10.5.3

This seems like a lot of code, but stop and look at it. Note that we are simply using the selectors of our data abstractions to get the pieces needed to look in the right place for our binding.

Also notice that this procedure has **exactly** the same type as the original `get`, which is must have if we are going to preserve the isolation of use of the abstraction from the implementation of the abstraction.



put! in table2

```
(define (table2-put! tbl key val)
  (let ((index
         ((hashfunc-of tbl) key (size-of tbl)))
        (buckets (buckets-of tbl)))
    (vector-set! buckets index
                 (add-assoc key val
                   (vector-ref buckets index)))))
```

- Same type as table1-put!

Slide 10.5.4

To put something into this new table, we do basically the same thing in reverse. Notice how the type is still preserved, as it must be. The difference is that inside the procedure we use the selectors of the abstraction to get out the hash function, find the right assocation list and then add the new binding to that list, and install that modified list back into the right bucket.

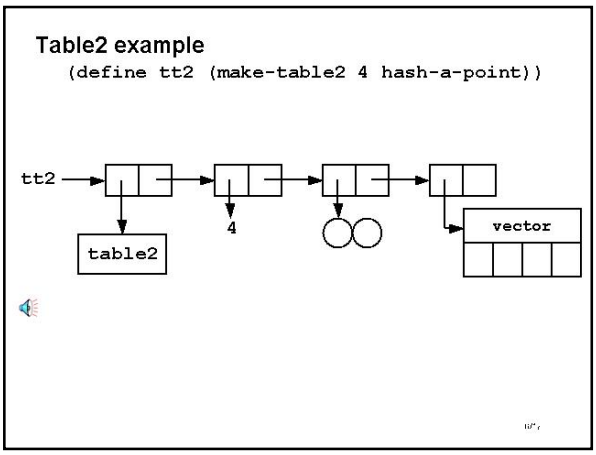
Slide 10.5.5

So let's look at a little example, to see how the behavior of the accessors of a table has changed in terms of efficiency but not in terms of user interface. Let's create a new table using our constructor. As noted, we need to specify both a size and a hash function (we will use the one we defined earlier for points in the plane as an example).

Table2 example

```
(define tt2 (make-table2 4 hash-a-point))
```

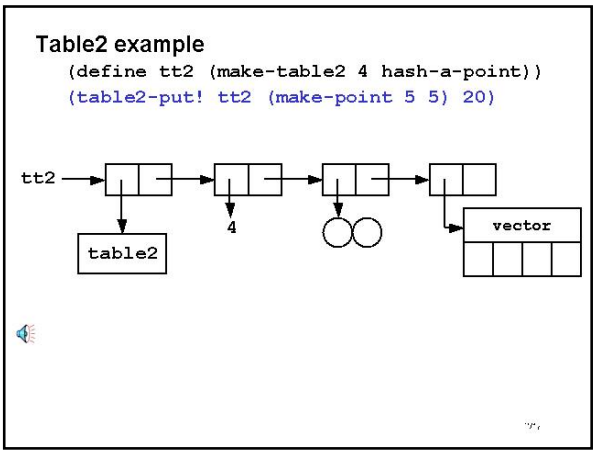
Slide 10.5.6

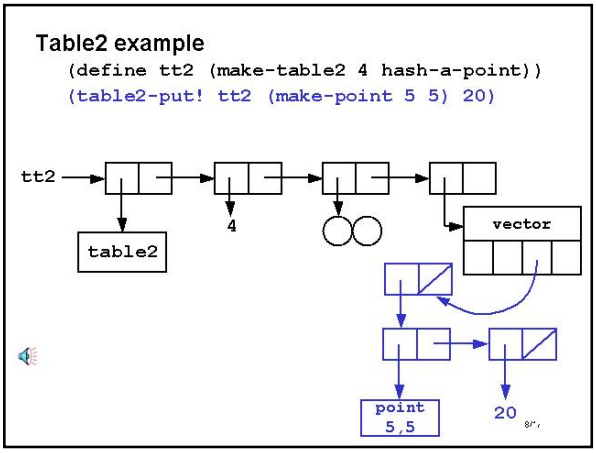


And here is a representation of the structure that is created by this evaluation. `Tt2` points to a list, containing the symbolic tag, the size, the procedure that computes the hash function, and a pointer to an initially empty vector of the specified size.

Slide 10.5.7

Now, let's manipulate this structure. Let's insert into the table a point and a paired value.



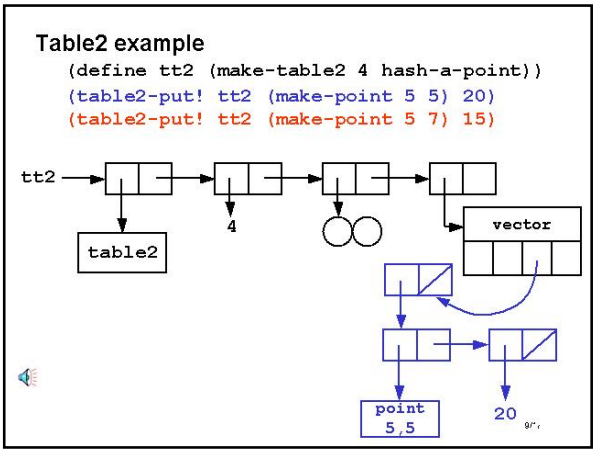


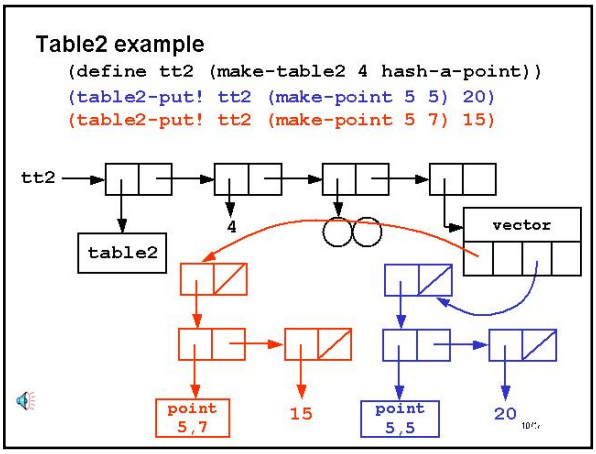
Slide 10.5.8

If you look back at the code, you will see that this operation does the following. First, it gets the hash function and the size, and applies those two things to this key to determine which bucket of the vector to use. It then takes the a-list stored at that location, and glues to the front of that list a new binding of the input point and input value. We expect this to lead to better efficiency, since the objects will be distributed among 4 (in this case) different a-lists.

Slide 10.5.9

Let's do the same thing with another pairing, inserting this into the table...



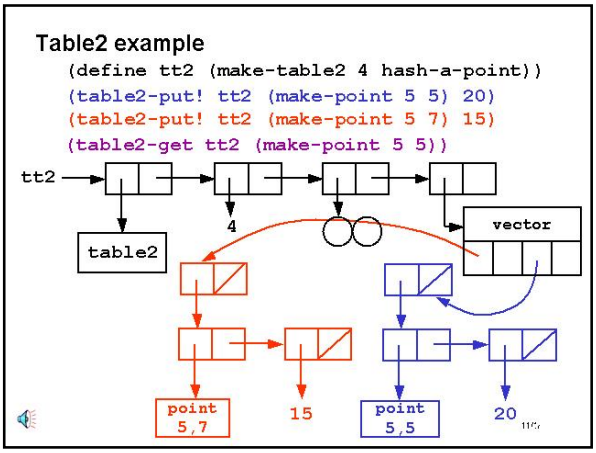


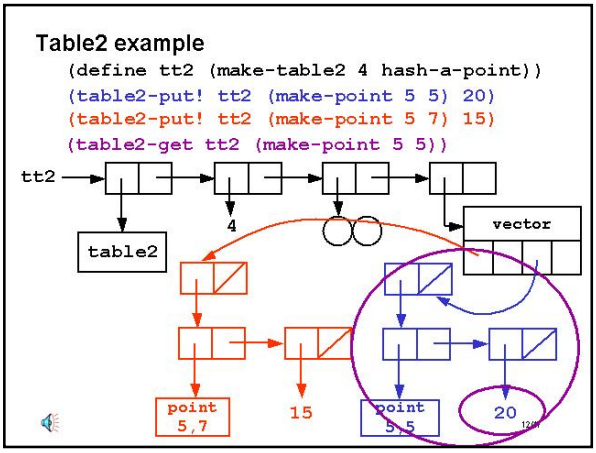
Slide 10.5.10

... and in exactly the same way, this will create a new association list hanging off of one of the buckets of the vector.

Slide 10.5.11

Now remember that our reason for making this change was to get a table that was more efficient in supporting retrieval of information. Let's see what happens if we retrieve a value from the table, using the example shown in purple.





Slide 10.5.12

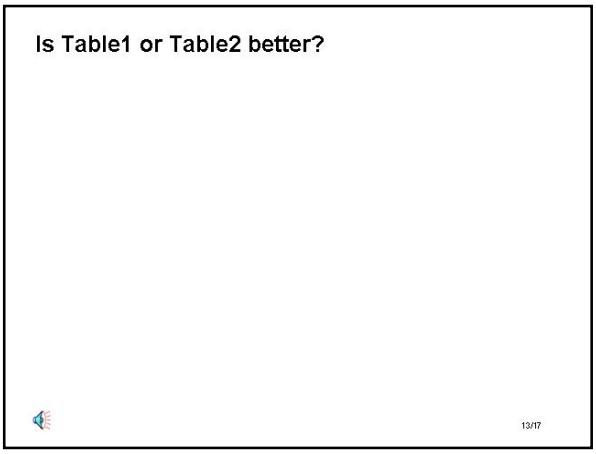
If you look back at the code, you will see that `get` for this kind of table takes the key, the size, and the hash function, and computes an index. In this case, we will extract the entry at the third bucket of the vector. Note that for a vector we can go straight to that bucket (in constant time) and retrieve the a-list stored there. Then, we used normal a-list operations to find the value paired with the key in that list. This will return the avlue `20`.

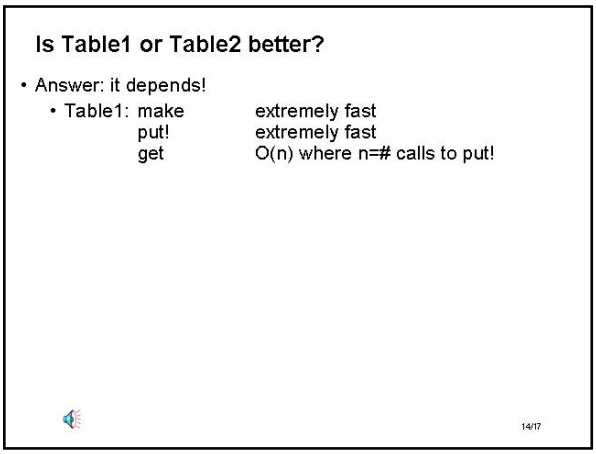
A key thing to note is that we are now searching down a much smaller a-list, and we expect in this case to do only 1/4 the amount of searching of the original case.

A second key thing to note is that from the point of view of the user none of this detail is visible. Only the change in efficiency will be noticed.

Slide 10.5.13

So what have we accomplished? We have built an abstract data type, a table, and have shown two different implementations of that ADT. The type contract for the ADT is the same in both cases, so the user can treat each implementation identically. At the same time, our motivation for having different implementations was to support different efficiencies. So which version is better?





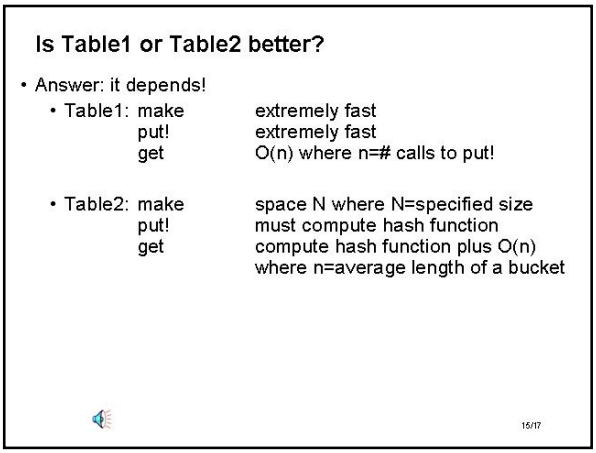
Slide 10.5.14

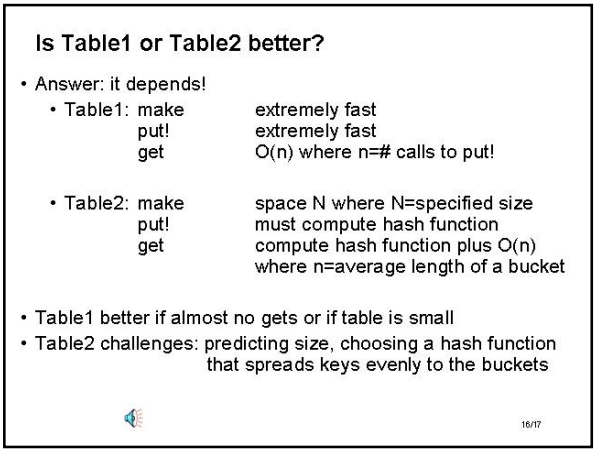
I have actually lead you down a garden path, since I originally implied that the second version would be better. In fact, the question of which version is better depends on what use you intend to make of the table.

For example, table1 has a very fast constructor and `put!` operation. But its `get` operation is order `O(n)`, where `n` is the number of calls to `put!`. This is because it will have to walk down a long association list to find the binding, where the lenght is the number of things put into the list.

Slide 10.5.15

Table2, on the other hand, takes a different trade off. Its constructor is going to use more space because it has to build a vector of the specified size. Note that its `put!` operation must compute a hash function, and thus the speed will depend on the speed of that function. The `get` operation also has to compute the hash function, then search down a smaller size list.





Slide 10.5.16

This then says that table1 will be better if we are doing very few `gets` or if the table is very small. Table2 will typically be better, but we have to be sure that we can accurately predict the size of vector needed and find a hash function that uniformly distributes keys over that range in an efficient manner.

Slide 10.5.17

So why go to all this work? First, we have seen a convenient new data structure. More importantly, we have seen how we can separate the use of an ADT from its implementation. The lesson to take away is how using opaque names and hiding of information lets us accomplish this separation.