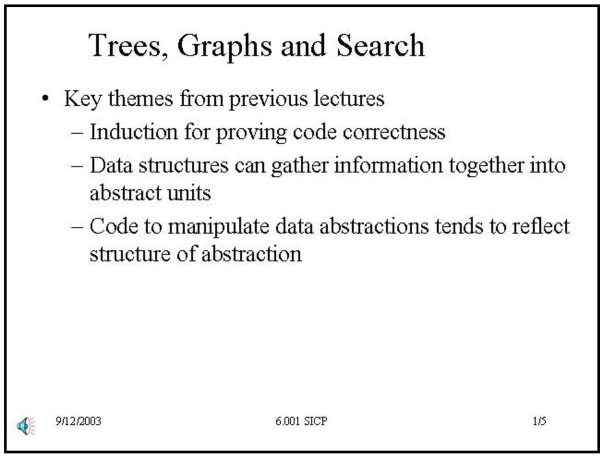
6.001 Notes: Section 31.1

Slide 31.1.1



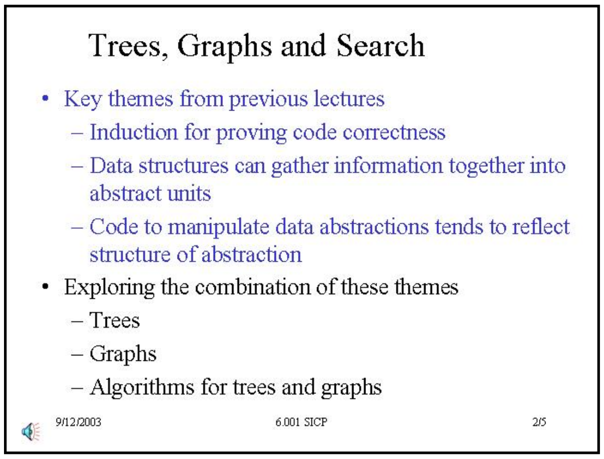
Trees, Graphs and Search

- Key themes from previous lectures
  - Induction for proving code correctness
  - Data structures can gather information together into abstract units
  - Code to manipulate data abstractions tends to reflect structure of abstraction

9/12/2003 6.001 SICP 1/5

In previous lectures we have seen a number of important themes, which relate to designing code for complex systems. One was the idea of proof by induction, meaning that we could reason formally about whether and why our code would work correctly based on an inductive proof. This specifically said that if we could prove (using basic axioms) that the code correctly handled a base case, and if we could prove that assuming the code ran correctly for all cases whose input was less than some given size, then it ran correctly for input of that given size, then we could conclude that it ran correctly on all correct inputs.
A second was the idea that we can gather related information together into higher-level data units, which we could abstract into treating as simple elements. Lists were a good example. And we saw that so long as the constructor and selectors of the abstraction obeyed a contract, we could suppress the details of the abstraction from its use.
The third was that code written to manipulate data abstractions frequently had a structure that reflected the underlying structure of the abstraction: often we would use selectors to extract out subparts, manipulate those parts, and then use the constructor to create a new version of the abstraction.

Slide 31.1.2



Trees, Graphs and Search

- Key themes from previous lectures
  - Induction for proving code correctness
  - Data structures can gather information together into abstract units
  - Code to manipulate data abstractions tends to reflect structure of abstraction
- Exploring the combination of these themes
  - Trees
  - Graphs
  - Algorithms for trees and graphs

9/12/2003 6.001 SICP 2/5

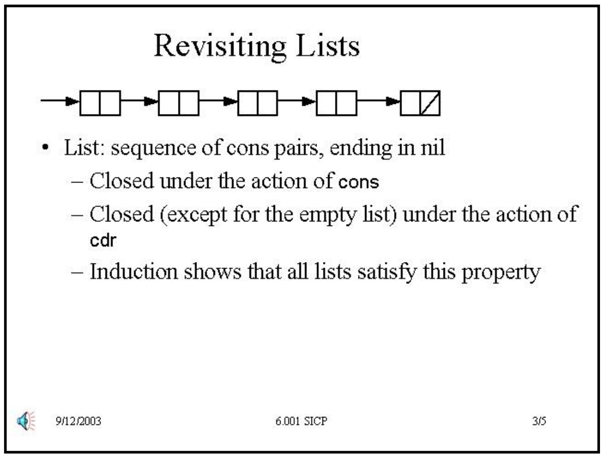
Today we are going to explore these themes in more detail. We are going to use them to build more complex data structures, and are particularly going to see how we can use inductive reasoning to design procedures to manipulate such structures. Specifically, we are going to look at two very useful data abstractions: trees and graphs. And we are going to see how procedures that search for information in large collections can be nicely designed to interact with such structures.

Slide 31.1.3

Let's start by revisiting lists. Remember that we said a list was simply a sequence of cons cells, connected by cdr's, ending in an empty list (also known as nil).

Remember that we said lists had the property of closure. If we were to cons anything onto the beginning of the list, we would get a new sequence of cons cells, ending in the empty list, hence a list. And with the exception of the empty list, taking the cdr of a list results in a sequence of cons cells, ending in the empty list, hence a list.

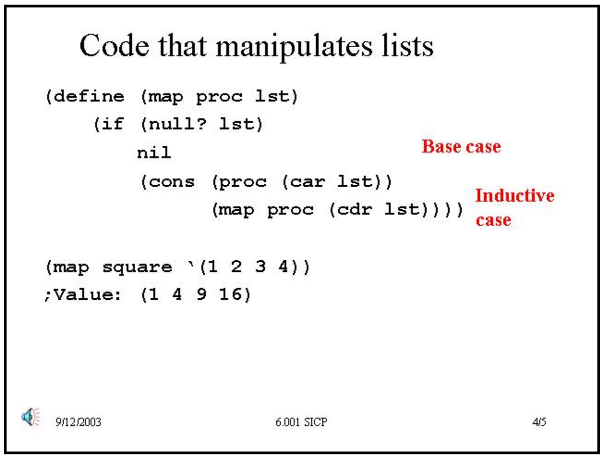
Note that this really has an inductive flavor to it. The base case is an empty list. Given that every collection of cons cells of size less than n, ending in an empty list, is a list; then clearly consing a new element onto such a list yields a list, and thus by induction, every collection of cons cells ending in the empty list is a list. We should be able to use this idea to reason about procedures that manipulate lists.



Revisiting Lists

- List: sequence of cons pairs, ending in nil
  - Closed under the action of cons
  - Closed (except for the empty list) under the action of cdr
  - Induction shows that all lists satisfy this property

9/12/2003 6.001 SICP 3/5



Code that manipulates lists

```
(define (map proc lst)
    (if (null? lst)
        nil
        (cons (proc (car lst))
              (map proc (cdr lst)))))

(map square '(1 2 3 4))
;Value: (1 4 9 16)
```

Base case

Inductive case

9/12/2003 6.001 SICP 4/5

Slide 31.1.4

Here is one of our standard procedures for manipulating lists: our old friend `map`. Intuitively, we know how this code should work, but can we establish formally that it does what we expect?

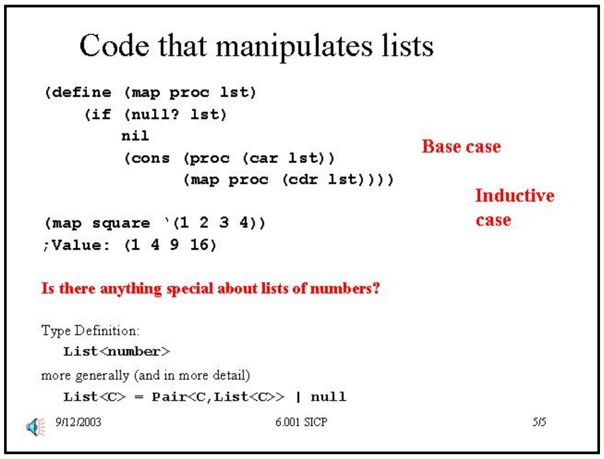
Sure. We just use our notion of induction here, both on the data structure and on the code itself. For the base case, we have the base case data structure for a list, namely an empty list. Thus, the code clearly returns the right structure. Now, assume that `map` correctly returns a list in which each element of the input list has had `proc` applied to it, and that the order of the elements is preserved, for any list of size smaller than the current list. Then we know, given `lst`, that by induction on the data structure `(cdr lst)` is a list. By induction on the procedure `map` we know this willl return a list of the same size, with each element replaced by the application of `proc` to that element. We can then process the first element of the list, and by induction on the data structure, `cons` will return a new list of the appropriate size that also satisfies the conditions of `map`. Thus, by induction, we know that `map` will correctly perform as expected.

Slide 31.1.5

Now, is there anything explicit in the code that says this applies only to lists of numbers?

Of course not. It could be lists of symbols, or lists of strings. Or lists of anything. That can be seen by looking at the type definition for a list.

As with pairs, we will represent a list by the symbol **List** followed by angle brackets containing a type definition for the elements of the list. Since lists are created out of pairs, we can be more explicit about this definition, by noting that a **List** of some type is either a **Pair** whose first element is of that type, and whose second element is a **List** of the same type, or (which is indicated by the vertical bar) the **list** is the special empty list.



Code that manipulates lists

```
(define (map proc lst)
    (if (null? lst)
        nil
        (cons (proc (car lst))
              (map proc (cdr lst)))))

(map square '(1 2 3 4))
;Value: (1 4 9 16)
```

Base case

Inductive case

Is there anything special about lists of numbers?

Type Definition:

```
List<number>
```

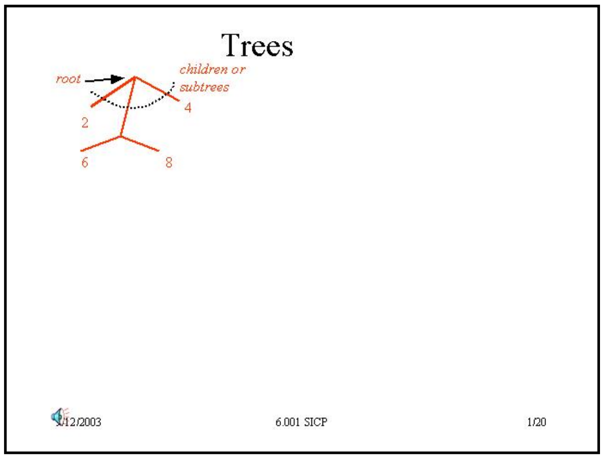
more generally (and in more detail)

```
List<C> = Pair<C,List<C>> | null
```

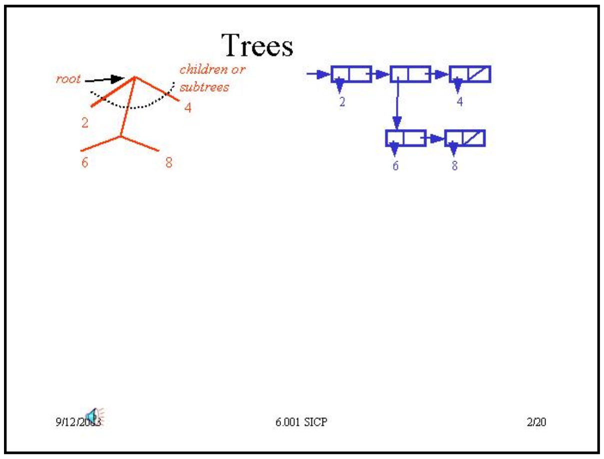
9/12/2003 6.001 SICP 5/5

Notice the nice recursive definition of a list, with the closure property that the cdr of a list is itself a list.
Notice that nothing in the type definition of a list says that the elements hanging off the cars of the list have to be numbers. In fact, our definition uses the arbitrary type C. This means that those structures could be anything, including other lists.
This leads nicely to a more general data structure called a **tree**, which lets us capture much more complex kinds of relationships and structures. So what does a tree look like, and are there common procedures associated with the manipulation of trees?

6.001 Notes: Section 31.2

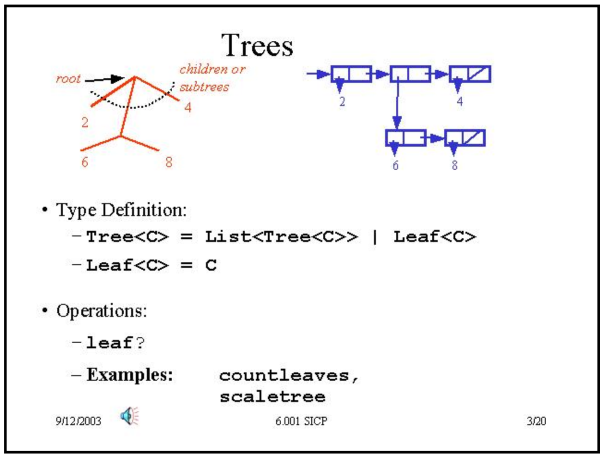


Slide 31.2.1
Here is the conceptual idea behind a tree. A tree has a root, or starting point. At the root, and indeed at every other point in the tree, there are a set of branches that connect different parts of the tree together. Each branch is said to contain a child, or sub-tree, and that sub-tree could itself be a tree, or could simply terminate in a leaf. In the example shown here, all the leaves are numbers, but of course they could be other objects such as strings.
Note that trees have a nice recursive property, much like lists, in which taking the element hanging off a branch gives us another tree. This suggests that procedures designed to manipulate trees should have a very similar recursive structure.



Slide 31.2.2
To implement a tree, we can just build it out of lists. Each level of the tree will be a list. The branches at a level will be the cars of the associated list. Those branches may themselves be sub-trees, that is, lists of lists. In the example shown, the first top-level branch is just the leaf, 2, but the second top-level branch is a tree, with two branches, 6 and 8. Thus, we have lists of lists as our implementation of a tree.

Slide 31.2.3



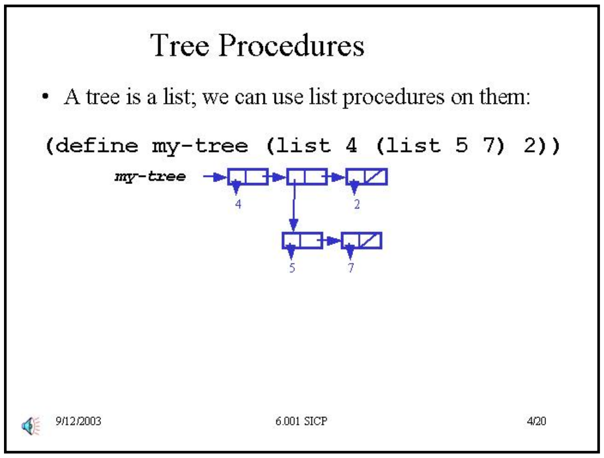
Trees

root
children or subtrees
2
4
6
8
2
4
6
8

- Type Definition:
  - `Tree<C> = List<Tree<C>> | Leaf<C>`
  - `Leaf<C> = C`
- Operations:
  - `leaf?`
  - **Examples:** `countleaves, scaletree`

9/12/2003 6.001 SICP 3/20

So what can we say about a tree? First, we can define its type. By analogy to a list, we have a tree of type C defined as either a list of trees of type C, reflecting the fact that there could be arbitrary branches, or that a tree is just a leaf of type C. Note the recursive definition of the tree structure. And a leaf of some type is just an element of that type.
In fact, as defined here, it would appear that a tree of type C would always have leaves of the same type, e.g. numbers or strings. Of course, we could generalize this to allow variations of types of elements in the leaves of a tree.
Associated with a tree, we expect to see some standard operations. For example, we should have a predicate, `leaf?` that tests where an element is a leaf or if it is a more complex sub-tree. In addition, we would like to have procedures that generalize operations on lists, such as counting the number of leaves (or basic elements) of a tree. We expect that these procedures should be more general than the lists versions, to reflect the fact that elements of a tree may themselves be complex things, like trees.



Tree Procedures

- A tree is a list; we can use list procedures on them:

```
(define my-tree (list 4 (list 5 7) 2))
```

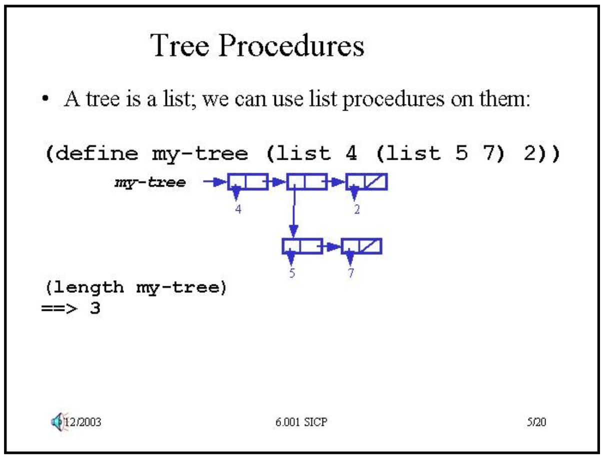
my-tree
4
2
5
7

9/12/2003 6.001 SICP 4/20

Slide 31.2.4

Given trees, built as lists of lists, what kinds of procedures can we create to manipulate them? First, because a tree is implemented as a list, in principle we could use list operations on the tree, although that starts to infringe on the data abstraction barrier that separates the use of an abstraction from its implementation.
Here is a simple example. I have defined a test tree, and given it a name. I have also shown the box-and-pointer diagram that represents the actual implementation of this tree.

Slide 31.2.5



Tree Procedures

- A tree is a list; we can use list procedures on them:

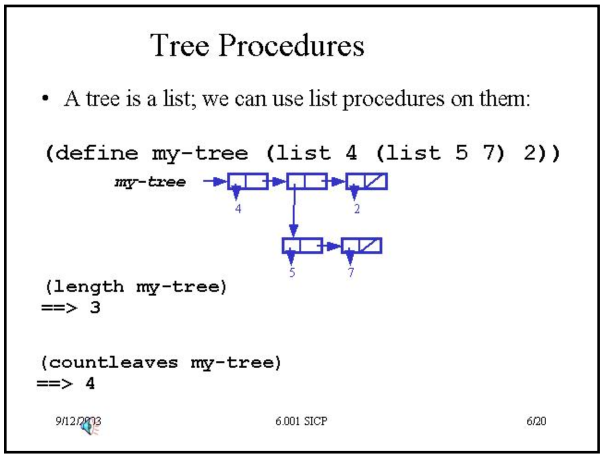
```
(define my-tree (list 4 (list 5 7) 2))
```

my-tree
4
2
5
7

```
(length my-tree)
==> 3
```

12/2003 6.001 SICP 5/20

Suppose I ask for the length of this structure. Ideally, `length` should be applied to lists, but because I have chosen to represent a tree as a list of list, this procedure can be applied here. For this example, it returns the value 3, which probably isn't what you expected. After all, we would like to think of a tree in terms of the number of leaves, not the number of top-level branches.
Why did this happen? Well, recall that `length` applies to a list, and it simple counts the number of cons pairs in that list, without ever looking at what is hanging off those pairs. In this case, this data structure is a list of three elements, some of which happen to be lists as well.
So if I want the number of top-level branches, this does the right thing, but suppose I really want to know how many leaves are in the tree?



Tree Procedures

- A tree is a list; we can use list procedures on them:

```
(define my-tree (list 4 (list 5 7) 2))
```

my-tree
4
2
5
7

```
(length my-tree)
==> 3

(countleaves my-tree)
==> 4
```

Slide 31.2.6

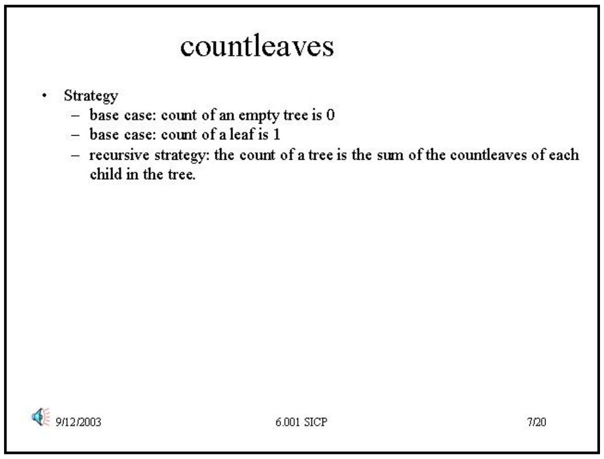
What I would like is another procedure, `count-leaves` that would in this case return the value 4 to indicate there are four basic elements in this tree. I need a procedure that directly takes advantage of the data structure of the tree to accomplish this.

Slide 31.2.7

So let's think about this, using the general idea that procedures should reflect the structure of the data abstraction to which they apply.

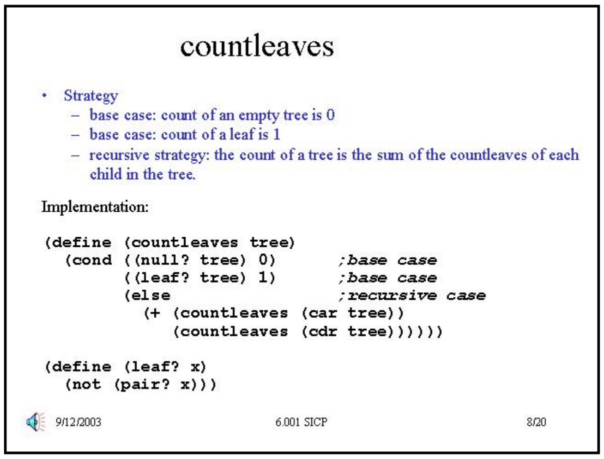
To count the number of leaves in a tree, we can devise a nice recursive strategy. The base case says if the tree is empty, then there are no leaves, obviously. However, we also have a second base case here, which is different than what we have seen before. This second base case says that if the tree is just a single leave, then the value to return is 1.

For the recursive strategy we need to be careful. When we were dealing with lists, the recursive strategy said "do something to the car of the list" and then add that into whatever we are doing applied again to the rest of the list. Remember that a tree could have another tree as each branch, or child, of the tree, so we have to count the leaves in each child, then add all of those up.



countleaves

- Strategy
  - base case: count of an empty tree is 0
  - base case: count of a leaf is 1
  - recursive strategy: the count of a tree is the sum of the countleaves of each child in the tree.



countleaves

- Strategy
  - base case: count of an empty tree is 0
  - base case: count of a leaf is 1
  - recursive strategy: the count of a tree is the sum of the countleaves of each child in the tree.

Implementation:

```
(define (countleaves tree)
  (cond ((null? tree) 0)        ;base case
        ((leaf? tree) 1)        ;base case
        (else                   ;recursive case
          (+ (countleaves (car tree))
             (countleaves (cdr tree))))))

(define (leaf? x)
  (not (pair? x)))
```

Slide 31.2.8

So let's implement that idea. My definition for `count-leaves` has one argument, a tree. It starts with the two base cases. If the tree is empty, using the fact that the tree is implemented as a list, then return 0. If the tree is a leaf, which we can find by testing that it is not a pair (since that would mean it was a list, and hence a subtree), then return 1.

Otherwise, I add up the number of leaves in the first branch of this tree, and combine that with the number of leaves in the tree formed by pruning that first branch off.

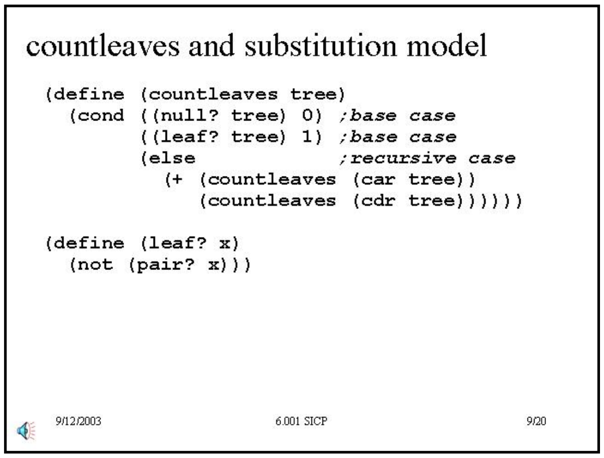
Aha! This is a different form than what we saw earlier. Remember for lists, we would have just done something to the `car` of the tree, and then combined that with a recursive call to `count-leaves` on the `cdr` of the tree. But since the `car` of a tree could itself be a tree, and not just an element, we have to recursively apply the procedure to both the `car` and the `cdr` of the tree.

So what kind of order of growth should we expect for such a procedure? Notice that there are two calls to the procedure in the recursive case, and that should remind you of a type of procedure we saw before, namely **exponential**.

Step back for a second. Notice how again the definition of this procedure reflects the structure of the associated data abstraction to which it will be applied. Compare this to what we saw with lists.

This message we will repeat many times during the term. The structure of the procedure tends to go hand-in-hand with the structure of the data abstraction it is designed to manipulate.

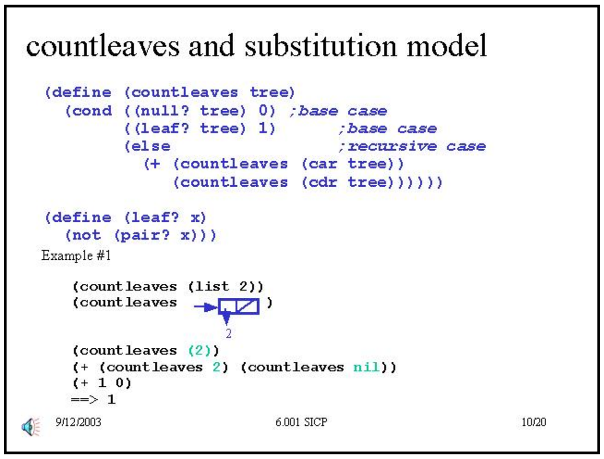
Slide 31.2.9
Because this is a new kind of data structure, and because we are creating new kinds of procedures to manipulate them, let's look at this a bit more carefully. Here again is the code for `count-leaves`, with the two base cases, and the double recursion down the first branch and the rest of the branches of the tree.



countleaves and substitution model

```
(define (countleaves tree)
  (cond ((null? tree) 0)  ;base case
        ((leaf? tree) 1)  ;base case
        (else             ;recursive case
          (+ (countleaves (car tree))
             (countleaves (cdr tree))))))

(define (leaf? x)
  (not (pair? x)))
```

9/12/2003 6.001 SICP 9/20



countleaves and substitution model

```
(define (countleaves tree)
  (cond ((null? tree) 0)  ;base case
        ((leaf? tree) 1)        ;base case
        (else                   ;recursive case
          (+ (countleaves (car tree))
             (countleaves (cdr tree))))))

(define (leaf? x)
  (not (pair? x)))
```

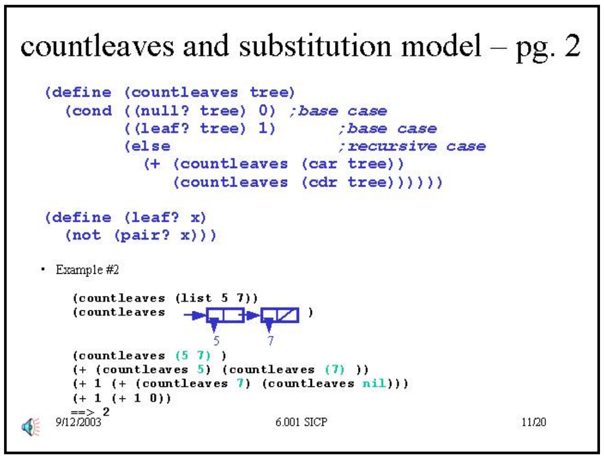
Example #1

```
(countleaves (list 2))
(countleaves           )

(countleaves (2))
(+ (countleaves 2) (countleaves nil))
(+ 1 0)
==> 1
```

9/12/2003 6.001 SICP 10/20

Slide 31.2.10
Let's use our substitution model to see how this procedure evolves, and we will do this in stages. Here is the simplest case. Let's apply `count-leaves` to a tree of one leaf. To be very careful, I am going to replace `(list 2)` with the actual box-and-pointer structure that is its value. `Count-leaves` first checks the base cases, but neither applies. Thus it reduces to adding `count-leaves` of the first subtree, or the `car` of the box-and-pointer structure to `count-leaves` of the rest of the branches, or the `cdr` or the box-and-pointer structure.
Note that the first recursive call will catch the second base case, while the second recursive call will catch the first base case, and eventually this reduces to the correct value for the number of leaves in this tree, as seen in the original box and pointer diagram, which had exactly one element in it.

Slide 31.2.11
Now let's try a bit more complicated tree, as shown. Notice that this tree has the property that its first branch is just a leaf, but its second branch, defined as everything but the first branch, is itself a tree.
`Count-leaves` thus applies to the box-and-pointer structure shown. In this case, the recursive call will add the number of leaves in the first branch, which is just the leaf, 5, since that is what the `car` returns, to the number of leaves in the tree, `(7)` since that is what the `cdr` of the original structure returns.
Thus, the first recursive call will trigger the second base case, returning 1 for the single leaf there. The second recursive call will reduce to the case we saw on the last slide, recognizing this as another tree. Thus it unwinds the tree one more level, then returning 1 for the leaf in the first branch, and 0 for the empty tree in the second branch.



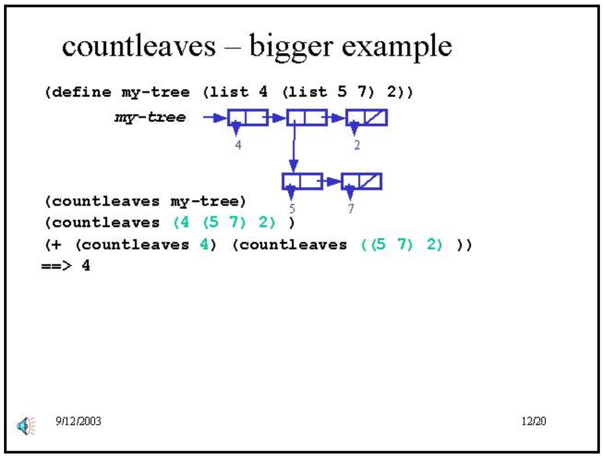
countleaves and substitution model – pg. 2

```
(define (countleaves tree)
  (cond ((null? tree) 0) ;base case
        ((leaf? tree) 1)       ;base case
        (else                  ;recursive case
          (+ (countleaves (car tree))
             (countleaves (cdr tree))))))

(define (leaf? x)
  (not (pair? x)))
```

- Example #2

```
(countleaves (list 5 7))
(countleaves                 )

(countleaves (5 7) )
(+ (countleaves 5) (countleaves (7) ))
(+ 1 (+ (countleaves 7) (countleaves nil)))
(+ 1 (+ 1 0))
==> 2
```

9/12/2003 6.001 SICP 11/20



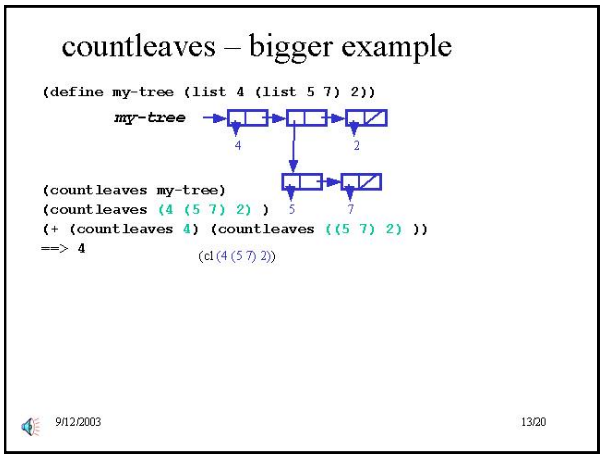
countleaves – bigger example

```
(define my-tree (list 4 (list 5 7) 2))
       my-tree

(countleaves my-tree)
(countleaves (4 (5 7) 2) )
(+ (countleaves 4) (countleaves ((5 7) 2) ))
==> 4
```

9/12/2003 12/20

Slide 31.2.12
Now let's try a full-fledged tree, as shown here. In fact, `count-leaves` will recursively unwind the tree, ending up with the correct value of 4, but let's see if we can quickly trace out why.

Slide 31.2.13
To do this, let's trace the calls to `count-leaves`, which for convenience I have abbreviated as `c-l`. Also notice that I have abbreviated the tree by its associated list structure, which I represent in blue to indicate that this represents a list, not a procedure call.
Recursively, we know what this does. It applies `count-leaves` to the first element of the tree, and to the rest of the tree, and adds the result.

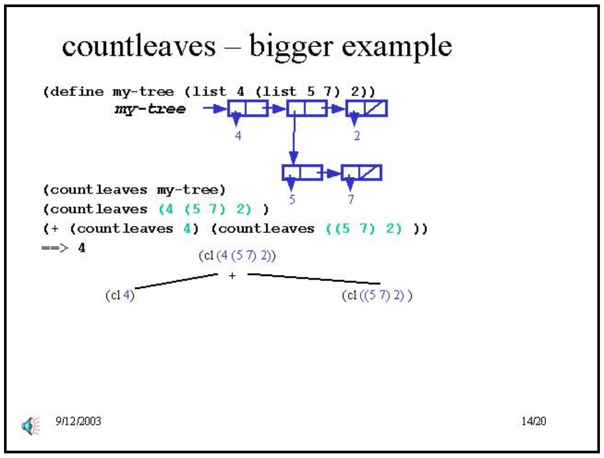


countleaves – bigger example

```
(define my-tree (list 4 (list 5 7) 2))
       my-tree

(countleaves my-tree)
(countleaves (4 (5 7) 2) )
(+ (countleaves 4) (countleaves ((5 7) 2) ))
==> 4
```

4
2
5
7
(cl (4 (5 7) 2))

9/12/2003 13/20



countleaves – bigger example

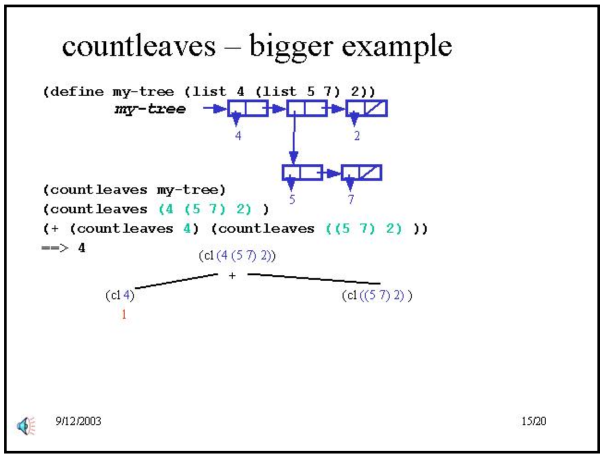
```
(define my-tree (list 4 (list 5 7) 2))
       my-tree

(countleaves my-tree)
(countleaves (4 (5 7) 2) )
(+ (countleaves 4) (countleaves ((5 7) 2) ))
==> 4
```

4
2
5
7
(cl (4 (5 7) 2))
+
(cl 4)
(cl ((5 7) 2))

9/12/2003 14/20

Slide 31.2.14
So here is that recursive decomposition. Note how we have stripped out the two subtrees, as shown.

Slide 31.2.15
Well, doing`count-leaves` down the first sub-tree is easy. By the second base case, this is just the value 1.

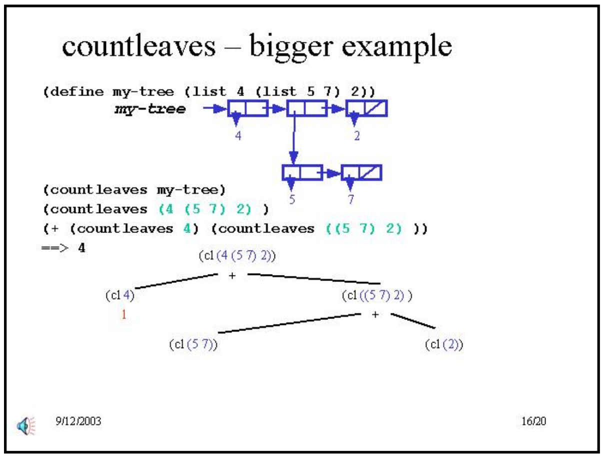


countleaves – bigger example

```
(define my-tree (list 4 (list 5 7) 2))
       my-tree

(countleaves my-tree)
(countleaves (4 (5 7) 2) )
(+ (countleaves 4) (countleaves ((5 7) 2) ))
==> 4
```

4
2
5
7
(cl (4 (5 7) 2))
+
(cl 4)
1
(cl ((5 7) 2))

9/12/2003 15/20



countleaves – bigger example

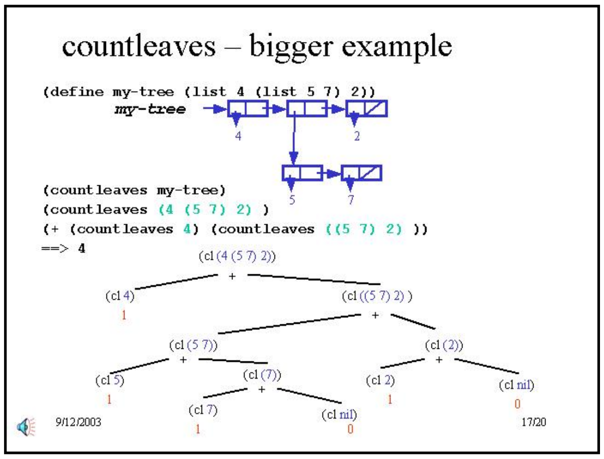
```
(define my-tree (list 4 (list 5 7) 2))
(countleaves my-tree)
(countleaves (4 (5 7) 2) )
(+ (countleaves 4) (countleaves ((5 7) 2) ))
==> 4
```

Slide 31.2.16

And of course the recursive call down the rest of three does not trigger a base case, since the argument is a tree. Instead, we add`count-leaves` of the first element (notice this is just the first element of that list, even though it is itself a tree), to`count-leaves` of the rest of the tree (i.e. the`cdr` of the argument).

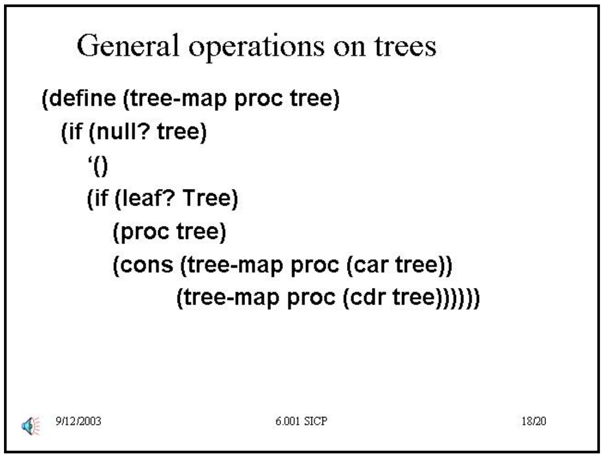
Slide 31.2.17

And now you get the idea. At each level of the tree, we are calling`count-leaves` on the first element of tree, and adding that to whatever we get by call`count-leaves` on the remaining branches of the tree, which is itself a tree. These calls keep unwinding until they either reach a leaf (with a return value of 1) or an empty tree (with a return value of 0), and all these values then get added up to return the final value.



countleaves – bigger example

```
(define my-tree (list 4 (list 5 7) 2))
(countleaves my-tree)
(countleaves (4 (5 7) 2) )
(+ (countleaves 4) (countleaves ((5 7) 2) ))
==> 4
```



General operations on trees

```
(define (tree-map proc tree)
  (if (null? tree)
     '()
     (if (leaf? Tree)
        (proc tree)
        (cons (tree-map proc (car tree))
                 (tree-map proc (cdr tree))))))
```

9/12/2003
6.001 SICP
18/20

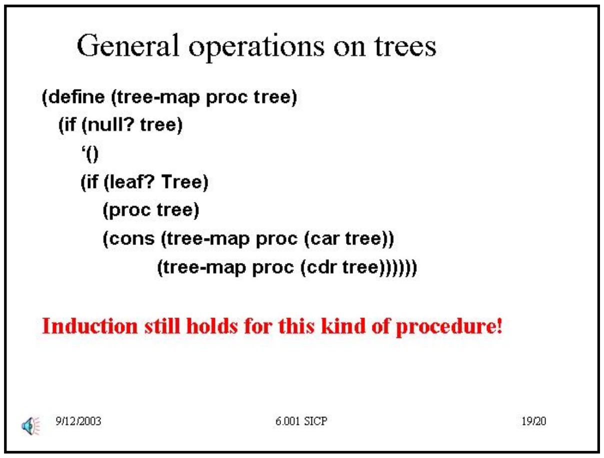
Slide 31.2.18

Of course, not only can we write procedures that directly manipulate trees, we can capture general patterns. Just as we had the notion of`map` for lists, we have a similar idea for trees, shown here.

Here we need to separate two different base cases: the empty tree, and a leaf (or isolated element) of a tree. For the empty tree, we just return an empty tree. For a leaf, we simply apply the procedure.

In the general case, we have to be careful about our data structure. A tree is a list, each of whose elements might itself be a tree. So we can split a tree into its`car` and its `cdr`, each of which is a tree. We must then map our procedure down each of these subpieces, and then glue them back together. This is different than mapping down a list, where we could just directly cons the processed first element onto the remainder of the list.

Slide 31.2.19
Note that induction holds in this case as well. For each of the two base cases, we get back an appropriate tree, either an empty tree or a single leaf. In the inductive case, we know that both the `car` and the `cdr` of the tree are trees of smaller size, so we can assume that `tree-map` correctly returns a processed tree. Then we know that `cons` will glue each of these pieces back into the larger tree, and hence by induction this code will correctly process trees of all sizes.

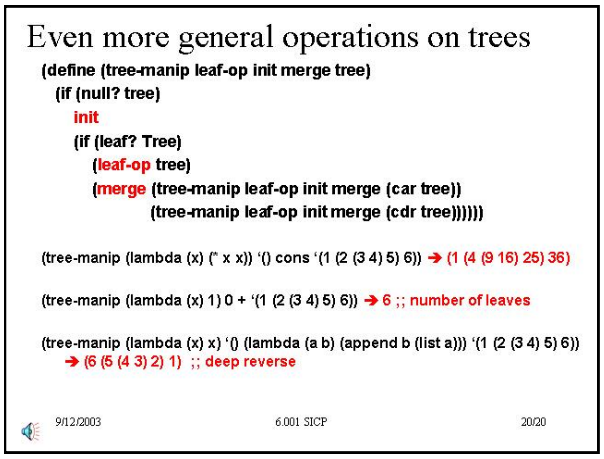


General operations on trees

```
(define (tree-map proc tree)
  (if (null? tree)
      '()
      (if (leaf? Tree)
          (proc tree)
          (cons (tree-map proc (car tree))
                (tree-map proc (cdr tree))))))
```

Induction still holds for this kind of procedure!

9/12/2003 6.001 SICP 19/20



Even more general operations on trees

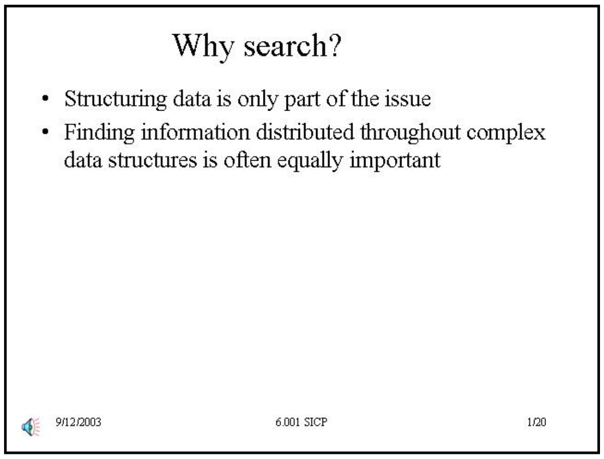
```
(define (tree-manip leaf-op init merge tree)
  (if (null? tree)
      init
      (if (leaf? Tree)
          (leaf-op tree)
          (merge (tree-manip leaf-op init merge (car tree))
                 (tree-manip leaf-op init merge (cdr tree))))))

(tree-manip (lambda (x) (* x x)) '() cons '(1 (2 (3 4) 5) 6)) ➔ (1 (4 (9 16) 25) 36)

(tree-manip (lambda (x) 1) 0 + '(1 (2 (3 4) 5) 6)) ➔ 6 ;; number of leaves

(tree-manip (lambda (x) x) '() (lambda (a b) (append b (list a))) '(1 (2 (3 4) 5) 6))
   ➔ (6 (5 (4 3) 2) 1)  ;; deep reverse
```

9/12/2003 6.001 SICP 20/20

Slide 31.2.20
We can capture higher order operations on trees, beyond just mapping. For example, this code allows us to specify what operation we want to apply to each leaf of the tree, and how we want to glue pieces together.
Using this we can map procedures onto each element of the tree; or we can count the number of leaves in a tree; or we can reverse the elements of the tree, both at the top level and at each subtree.

6.001 Notes: Section 31.3

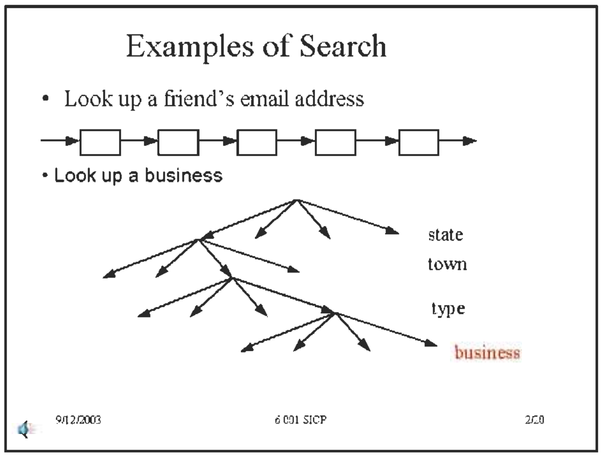
Slide 31.3.1
Now, let's turn to a different issue. Given that we can have fairly complex data structures: trees, each of whose subtree is a complex tree, the issue of how to manipulate that information becomes more complex. In the examples we just completed, we were mostly considering procedures that manipulated trees to return trees. But in many cases, we may want to decide if information is contained in a tree, or to decide how to add data to a tree in a structured way. That is, trees can be used to collect related data together in a structured way, and we need to think about how to handle that interaction.
Thus, the issue of how to collect data into complex structures is really only half of the problem; we also want to be able to retrieve information from that structure.



Why search?

- Structuring data is only part of the issue
- Finding information distributed throughout complex data structures is often equally important

9/12/2003 6.001 SICP 1/20



Examples of Search

- Look up a friend's email address
- Look up a business

9/12/2003 6.001 SICP 2/20

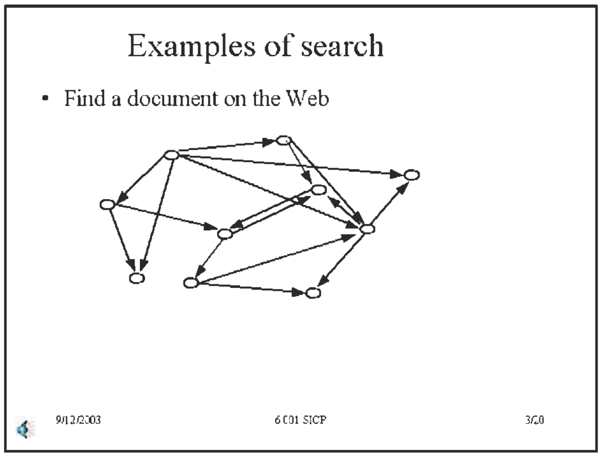
Slide 31.3.2

So we may want to find information in a complex structure by searching through that structure; or we may want to know if the information is simply missing from the structure. This means we want to search through data structures in efficient ways. What does this mean; to search for information in a data structure? Well, if the problem is small in size, we can imagine just storing the data in a linear structure like a list. For example, looking up an email address of a friend might only require looking at a few tens of addresses, so just searching down a list makes sense. Here each element of the list might be a small data structure that contains names, addresses, and other information; and search would involve walking down the list checking each entry against a name, and then returning the full entry when we find the right one.

But suppose I want to find the address for some business, perhaps based on the name of the business. Given that there are hundreds of thousands of businesses, I don't want to have to look through a linear list. Better would be first to restrict myself to a particular state, and then to a town, and then to a kind of business, and then look through a much smaller list. In this case, one can conceptually see why using a tree would be a much better way of structuring the information. I would only need to search at the top level through a set of branches to find the right state, thereby excluding huge portions of the data set. I could then search through a set of towns, and so on.

Slide 31.3.3

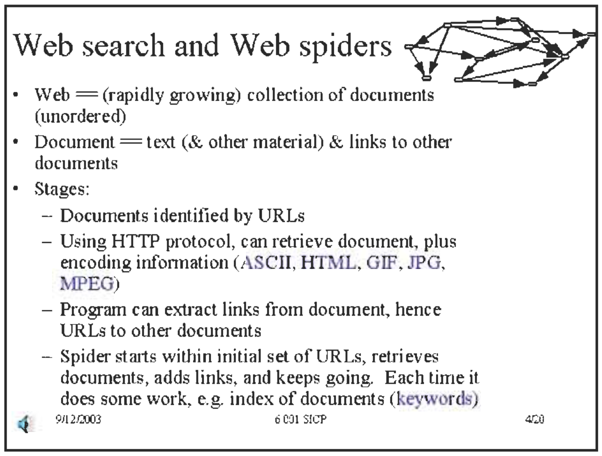
Let's broaden this even further. Suppose I want to find a document on the World Wide Web. How do I structure that vast amount of data in a manner that makes retrieval efficient?



Examples of search

- Find a document on the Web

9/12/2003 6.001 SICP 3/20



Web search and Web spiders

- Web == (rapidly growing) collection of documents (unordered)
- Document == text (& other material) & links to other documents
- Stages:
  - Documents identified by URLs
  - Using HTTP protocol, can retrieve document, plus encoding information (ASCII, HTML, GIF, JPG, MPEG)
  - Program can extract links from document, hence URLs to other documents
  - Spider starts within initial set of URLs, retrieves documents, adds links, and keeps going. Each time it does some work, e.g. index of documents (keywords)

9/12/2003 6.001 SICP 4/20

Slide 31.3.4

Well, at a conceptual level, what is the Web?

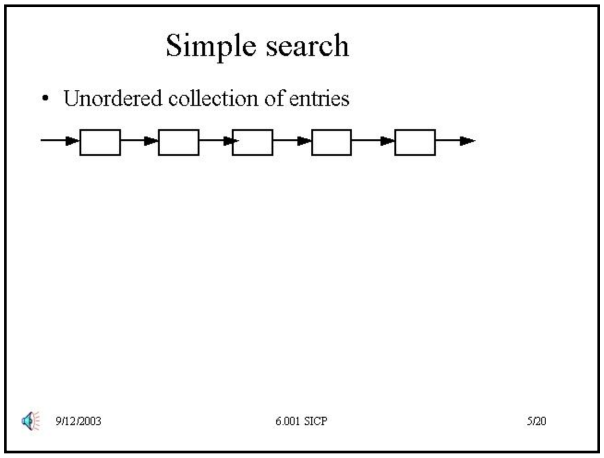
Basically, it is a very rapidly growing collection of documents, that are totally unordered (that is, there is no coherent organization collecting them together). A document consists of some text (or other information) that may be of interest to us, plus links to other documents.

To search through the Web, one starts with a particular document, which is identified by its URL (or Uniform Resource Locator). Using some common protocol for decoding the URL, one can retrieve the file containing the document, as well as information about how the document or file is encoded (is this in basic characters, is it a web page, is it an image, or a movie, and so on).

Of particular interest are those links contained within the document that may point to other documents. So typically, a search through the Web involves invoking a spider, that starts with some initial set of documents (identified by their URL's), retrieves the actual documents, processes them to find links within those documents,

then finding these new documents, and so on. Each time the spider retrieves a document, it might decide to record some information about the document, such as a set of keywords describing the content, which it will gather together to create an index into the data.
One can visualize how this creates a web, in which documents may be linked to multiple other documents, and may be linked from many other documents, creating an intricate data structure.

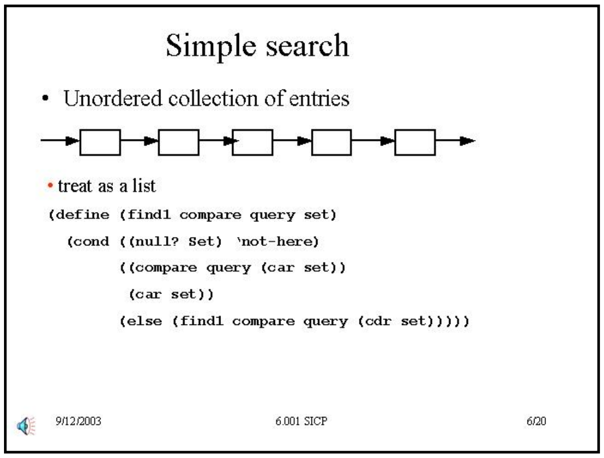
Slide 31.3.5
Now, suppose we want to search for some information. We have several options for how to store the information, and maintain it. Let's consider some of the tradeoffs.
First, suppose we just decide to collect data in an unordered list.



Simple search

- Unordered collection of entries

9/12/2003 6.001 SICP 5/20



Simple search

- Unordered collection of entries

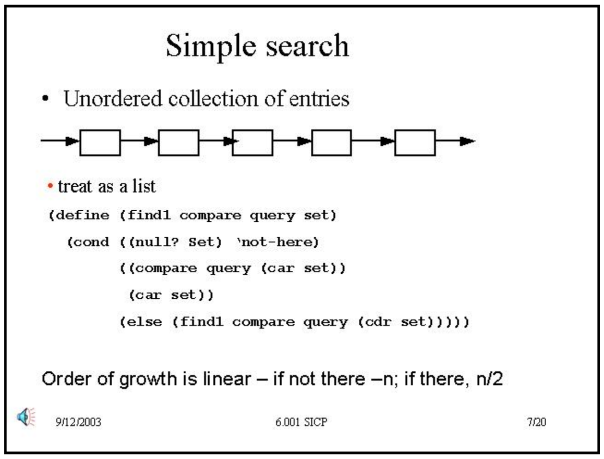
- treat as a list

```
(define (find1 compare query set)
  (cond ((null? Set)  'not-here)
        ((compare query (car set))
         (car set))
        (else (find1 compare query (cdr set)))))
```

9/12/2003 6.001 SICP 6/20

Slide 31.3.6
In this case, determining if a piece of information is present (or if you like, retrieving data based on some key word) simply looks like our standard list processing application. We walk down each element in the list in turn, checking to see if the desired information is there.
Note that what is stored in each element of the list is left open. For example, each element might be some other structure, and the comparison might involve checking to see if some keyword is present that structure. Thus, we leave open how we search each specific entry, but the top-level search is simply a linear traversal an unordered list.

Slide 31.3.7
In this case, we know how efficient (or not) this is. If the thing we are seeking is not present, we have to check every element of the list. If it is present, on average we have to look at half the elements of the list. Either way, the order of growth is linear in the size of the list. Clearly for large collections this is not efficient.



Simple search

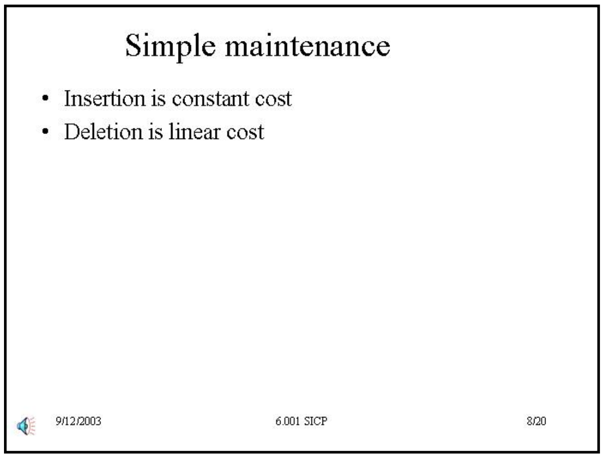
- Unordered collection of entries

- treat as a list

```
(define (find1 compare query set)
  (cond ((null? Set)  'not-here)
        ((compare query (car set))
         (car set))
        (else (find1 compare query (cdr set)))))
```

Order of growth is linear – if not there –n; if there, n/2

9/12/2003 6.001 SICP 7/20



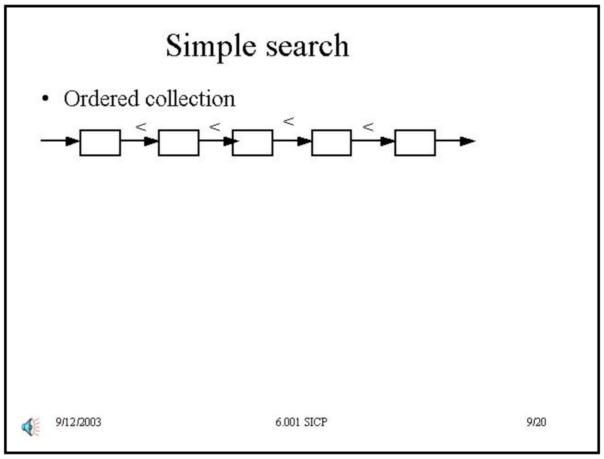
Simple maintenance

- Insertion is constant cost
- Deletion is linear cost

9/12/2003 6.001 SICP 8/20

Slide 31.3.8
On the other hand, if we want to add information to the set, this is easy. We just `cons` the new element onto the list, which requires only constant effort. Deleting an element from the list simply requires walking down the list until we find it, then removing it from the list and appending together the remaining elements. This is linear in the size of the list.

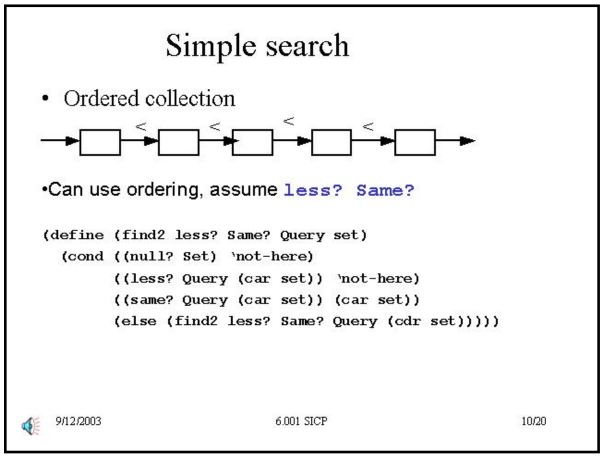
Slide 31.3.9
So can we do better? Well, let's assume that we have some ordering on the collection. We can leave this unspecified, but think of this like the lexicographic ordering of a dictionary. Does this help?



Simple search

- Ordered collection

9/12/2003 6.001 SICP 9/20



Simple search

- Ordered collection

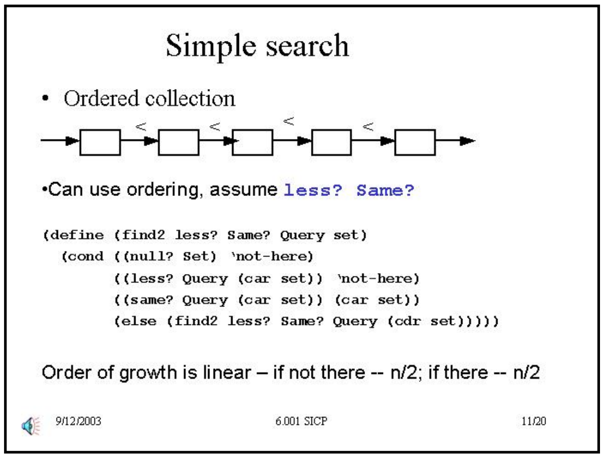
•Can use ordering, assume `less? Same?`

```
(define (find2 less? Same? Query set)
  (cond ((null? Set) 'not-here)
        ((less? Query (car set)) 'not-here)
        ((same? Query (car set)) (car set))
        (else (find2 less? Same? Query (cdr set)))))
```

9/12/2003 6.001 SICP 10/20

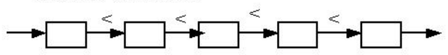
Slide 31.3.10
In terms of finding a piece of information, we can now use a bit more knowledge. We will still need to have a way of deciding if the thing we want is present (using some comparison procedure called `same?`). But we can also assume that there is some way of using the ordering information, for example, `less?` would be a general procedure that would tell us if the thing we were seeking should have appeared before this point in the list. By way of example, assume that each element of the list is a name. Then `same?` might be symbolic equality, and `less?` might be a way of testing whether one name comes before another in an alphabetical list.

Slide 31.3.11
Does this help us? Well, it may speed things up in actual run time, but it doesn't help in the long run. If the thing we are seeking is not there, on average we will have to search through half the list to determine that it is absent. If it is present, then on average we will also have to search through half the list. Thus, this speeds things up a bit, but the order of growth is still linear in the size of the list.



Simple search

- Ordered collection

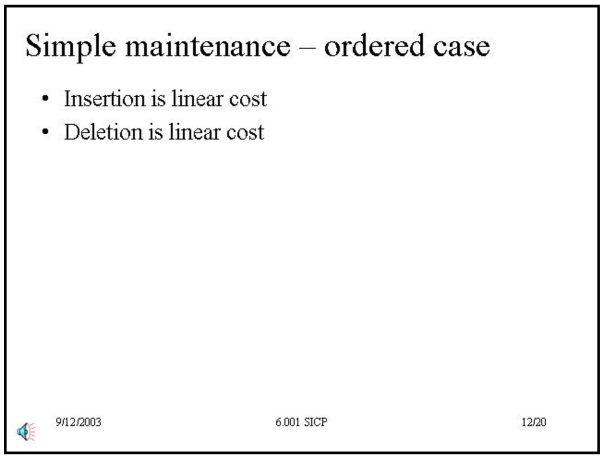


•Can use ordering, assume `less?` `Same?`

```
(define (find2 less? Same? Query set)
  (cond ((null? Set) 'not-here)
        ((less? Query (car set)) 'not-here)
        ((same? Query (car set)) (car set))
        (else (find2 less? Same? Query (cdr set)))))
```

Order of growth is linear – if not there -- n/2; if there -- n/2

9/12/2003 6.001 SICP 11/20



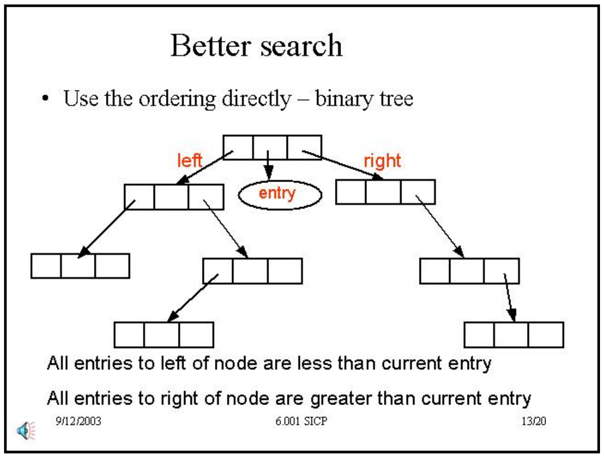
Simple maintenance – ordered case

- Insertion is linear cost
- Deletion is linear cost

9/12/2003 6.001 SICP 12/20

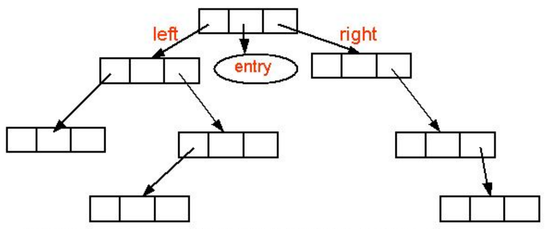
Slide 31.3.12
Here, if we want to add information to the set, we need to walk down the list to find the right place to insert it. This is also linear in effort. Deleting an element from the list also requires walking down the list until we find it, then removing it from the list and appending together the remaining elements. Again, this is linear in the size of the list. So we see that there is a bit more effort in maintaining an ordered list, compared to an unordered one, but we gain a little bit in efficiency.

Slide 31.3.13
This seems a bit surprising. Intuitively, if there is an order to the information, we ought to be able to use that ordering to be more efficient.
In fact we can; we simply need to use an appropriate data structure. One nice one is a particular kind of tree, called a **binary tree**. It is called this because at each level of the tree, there are at most two branches: one to the left and one to the right.
Conceptually, it is nice to think of each node of the tree as containing an entry (or a collection of information) and two branches. The tree has the property that everything lying down the left branch is less than the entry at that point, and everything lying down the right branch is greater than the entry at that point.



Better search

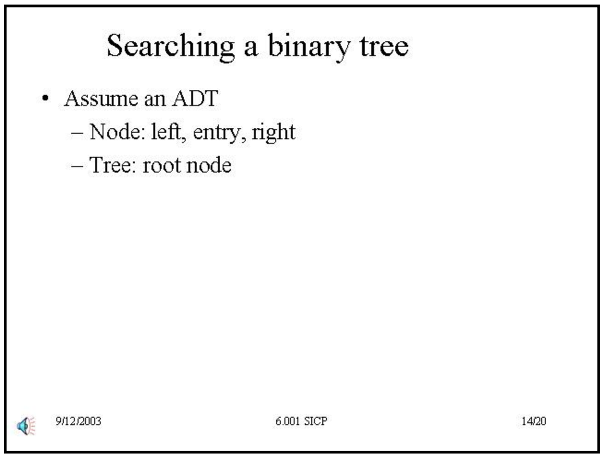
- Use the ordering directly – binary tree



All entries to left of node are less than current entry

All entries to right of node are greater than current entry

9/12/2003 6.001 SICP 13/20



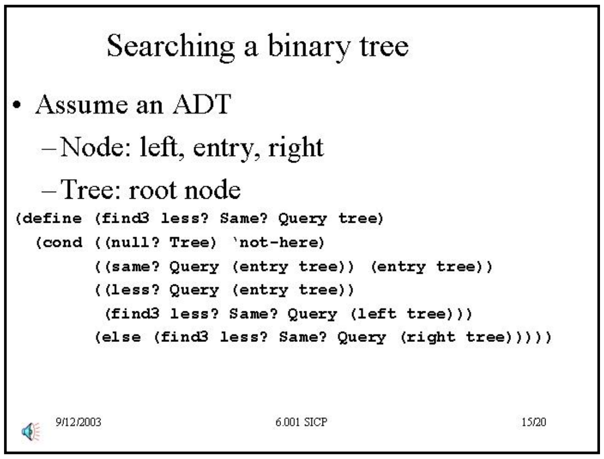
Searching a binary tree

- Assume an ADT
  - Node: left, entry, right
  - Tree: root node

9/12/2003 6.001 SICP 14/20

Slide 31.3.14
What does it take to search in this case? First, lets just assume that we have some abstract data type for representing a binary tree. Each node in the tree is a collection of three data structures, a left subtree, a right subtree, and an entry (of course each of these structures might itself be a complex data structure). A tree then just consists of some initial (or root) node, which contains pointers to the appropriate subtrees.

Slide 31.3.15
To search through the tree, we can use the ordering information in an efficient manner. The key difference is that if the information is not at the current location, we can use the ordering to decide which subtree to search, thus cutting in half the amount of data we need to search through at each stage. You can see this in the code.

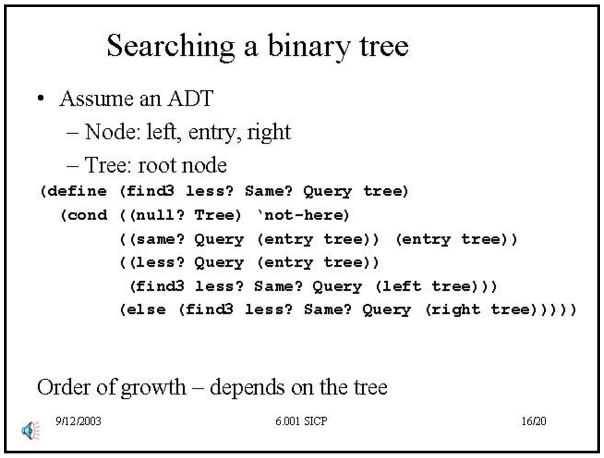


Searching a binary tree

- Assume an ADT
  - Node: left, entry, right
  - Tree: root node

```
(define (find3 less? Same? Query tree)
  (cond ((null? Tree) 'not-here)
        ((same? Query (entry tree)) (entry tree))
        ((less? Query (entry tree))
         (find3 less? Same? Query (left tree)))
        (else (find3 less? Same? Query (right tree)))))
```

9/12/2003 6.001 SICP 15/20



Searching a binary tree

- Assume an ADT
  - Node: left, entry, right
  - Tree: root node

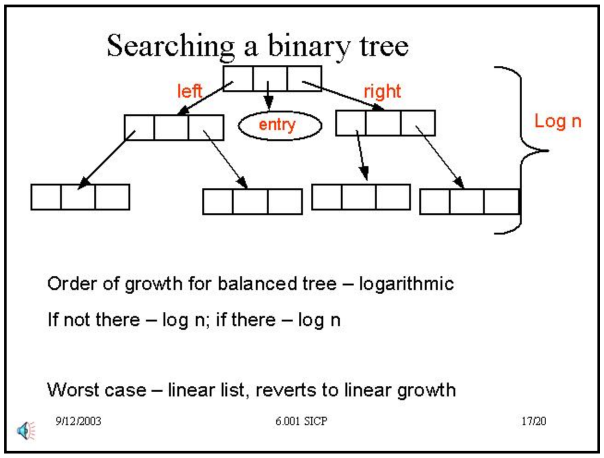
```
(define (find3 less? Same? Query tree)
  (cond ((null? Tree) 'not-here)
        ((same? Query (entry tree)) (entry tree))
        ((less? Query (entry tree))
         (find3 less? Same? Query (left tree)))
        (else (find3 less? Same? Query (right tree)))))
```

Order of growth – depends on the tree

9/12/2003 6.001 SICP 16/20

Slide 31.3.16
So what is the order of growth in this case? Well, technically it depends on the structure of the tree. For example, if each left branch were empty, then the tree ends up looking like a list, and the search would be linear.

Slide 31.3.17
But in general, if we believe that inserting and deleting elements in a tree will result in a structure that is **balanced**, meaning that typically there are as many elements in a left subtree as a right one, then we get much better efficiency. This is similar to some procedures we saw earlier in the term: if there are **n** elements stored in the tree, then there are **log n** levels in the tree. Since each stage of the search involves moving down one level in the tree, this is a **logarithmic** search, both when the information is present and when it is not.



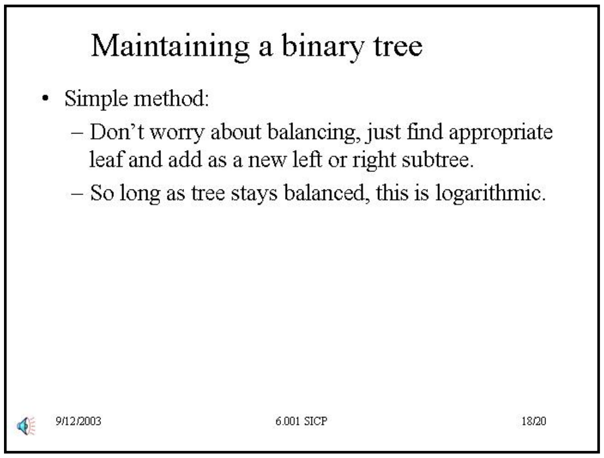
Searching a binary tree

Order of growth for balanced tree – logarithmic

If not there – log n; if there – log n

Worst case – linear list, reverts to linear growth

9/12/2003 6.001 SICP 17/20



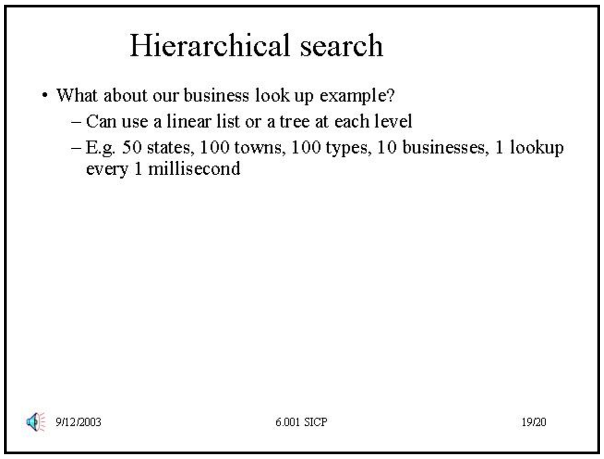
Maintaining a binary tree

- Simple method:
  - Don't worry about balancing, just find appropriate leaf and add as a new left or right subtree.
  - So long as tree stays balanced, this is logarithmic.

9/12/2003 6.001 SICP 18/20

Slide 31.3.18
Maintenance in this case is also efficient. If we don't worry about maintaining tree balance, we simply need to walk down the tree to find the right leaf at which to add, then insert as a new left or right subtree. Assuming that the tree stays reasonably balanced, this is just **logarithmic**. In practice, one may have to do some work to rebalance a tree, a topic that you will see in a later course.

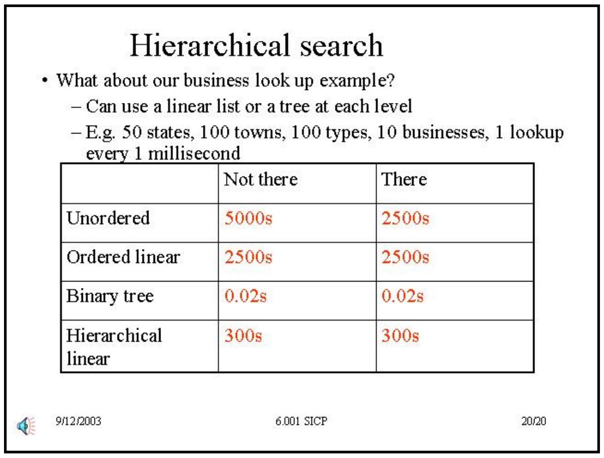
Slide 31.3.19
So, we have now seen several different ways of structuring ordered information. How different are they really? Well, suppose we go back to our example of looking up business information. For each level of this representation, we could choose to represent that information as a list or a tree. Let's assume that we have the number of states, towns, types, and businesses shown, and that each test takes a millisecond. How big a difference do the different choices make?



Hierarchical search

- What about our business look up example?
  - Can use a linear list or a tree at each level
  - E.g. 50 states, 100 towns, 100 types, 10 businesses, 1 lookup every 1 millisecond

9/12/2003 6.001 SICP 19/20



Hierarchical search

- What about our business look up example?
  - Can use a linear list or a tree at each level
  - E.g. 50 states, 100 towns, 100 types, 10 businesses, 1 lookup every 1 millisecond

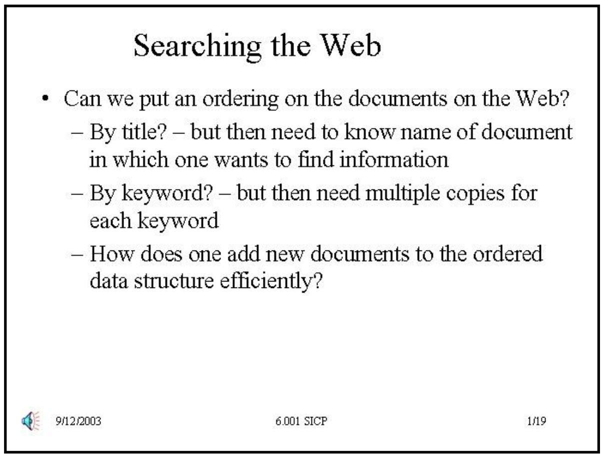
	Not there	There
Unordered	5000s	2500s
Ordered linear	2500s	2500s
Binary tree	0.02s	0.02s
Hierarchical linear	300s	300s

9/12/2003 6.001 SICP 20/20

Slide 31.3.20
Well, here is a chart of how long it will take on average to get information, when it is present, or to determine that the information is not present. Clearly an unordered list of all of the data is least efficient. Using a single ordered list (for example sorted by name of business) is a bit more efficient but not much.
Check out the binary tree, however. Clearly this is much more efficient!
One could imagine combining structures, for example, using a tree to represent each level of the overall structure, but representing the information at each level in an ordered list. While this improves over a pure list, it is still not as efficient as a tree.
Thus we see that different data structures really do have an impact on the efficiency of information retrieval and maintenance.

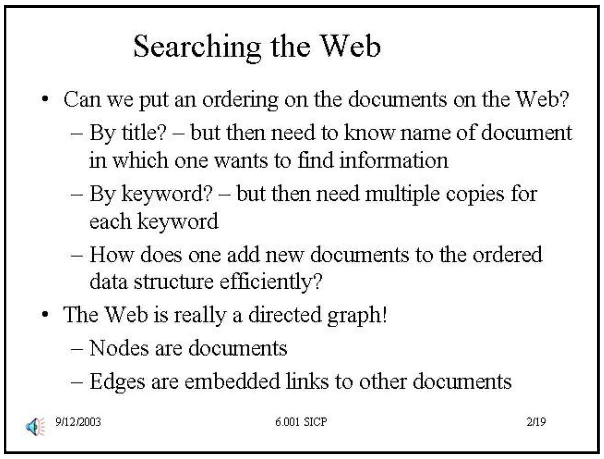
6.001 Notes: Section 31.4

Slide 31.4.1
Now, what about our idea of searching the Web to find information? We have seen that there are very different efficiencies, depending on what data structure we use. And we can see from the last slide, that these differences get more pronounced as the size of the data collection increases, which will certainly be the case for the Web.
The first question is how (or even whether) we can put an ordering on the documents on the Web. Based on our earlier example of an ordered list, we might opt to order Web documents by title, but then in order to execute a search we need to know the title of the document for which we are looking. Clearly that doesn't make sense, since we might want to search for documents that contain a particular keyword.
Well then, how about ordering the documents by keyword? This would be possible, except that a document might contain several key words, in which case we would need multiple copies of the document in our data structure, once for each keyword. This will clearly increase the size of the problem, and slow down our search.
Even if we go with keyword ordering, what about adding new documents to the data structure? Since insertion is linear in the size of the data structure, and since we would have to insert multiple copies into our data structure, this is going to be very slow.
So can we do better?


Searching the Web

- Can we put an ordering on the documents on the Web?
  - By title? – but then need to know name of document in which one wants to find information
  - By keyword? – but then need multiple copies for each keyword
  - How does one add new documents to the ordered data structure efficiently?

9/12/2003 6.001 SICP 1/19

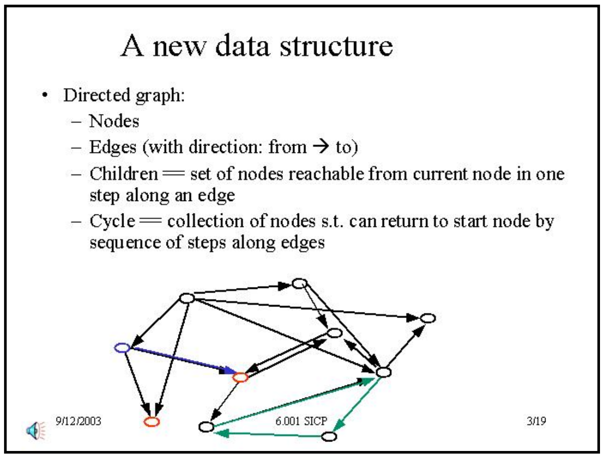

Searching the Web

- Can we put an ordering on the documents on the Web?
  - By title? – but then need to know name of document in which one wants to find information
  - By keyword? – but then need multiple copies for each keyword
  - How does one add new documents to the ordered data structure efficiently?
- The Web is really a directed graph!
  - Nodes are documents
  - Edges are embedded links to other documents

9/12/2003 6.001 SICP 2/19

Slide 31.4.2
Well, in reality the Web is a different kind of beast. It is really a structure called a **directed graph**. A **graph** is an extension of a tree. It consists of a set of **nodes**, which in the case of the Web are our documents. And it consists of a set of connections between the nodes, called **edges**, which in the case of the Web are the embedded links to other documents.
The general idea of a graph is that one can move from one node to another by following a link. In this case, the links are **directed**, meaning that one can only follow the link from the current node through the edge to the target node, and there may not be a return link from the target node. This makes sense since a document might have an embedded link to a target document, without the target document containing an embedded link back to the original one.

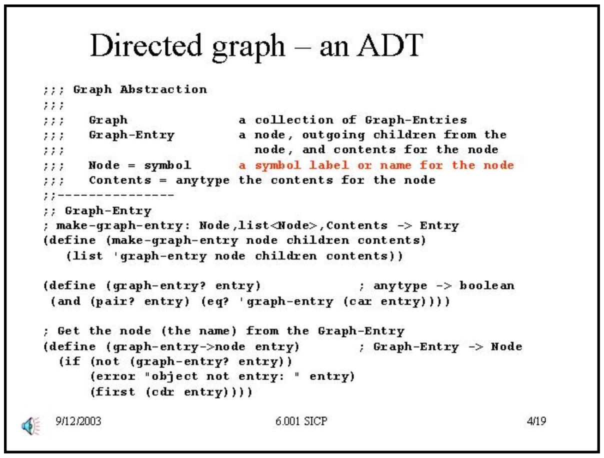
Slide 31.4.3
So we want to create a new data structure, our **directed graph**. As noted, it will consist of a set of nodes, and a set of edges, which connect two nodes in a particular direction. For example, the blue edge illustrates a directed edge between two nodes. Given these two components of a directed graph, it is useful to extract the **children** of a node, which are the set of nodes reachable in one step along an edge from a given node. For example, the blue node has two children, both drawn in red. While one can consider a tree as a kind of directed graph, in general directed graphs can be more complex. Specifically, they might contain a **cycle**, that is, a sequence of nodes connected by directed edges such that one can return to a starting node by a


A new data structure

- Directed graph:
  - Nodes
  - Edges (with direction: from → to)
  - Children == set of nodes reachable from current node in one step along an edge
  - Cycle == collection of nodes s.t. can return to start node by sequence of steps along edges

9/12/2003 6.001 SICP 3/19

stepping along the edges. An example is shown in green. This will be important when we get around to designing our search algorithm, since we don't want to get trapped going around in circles through the same set of nodes.



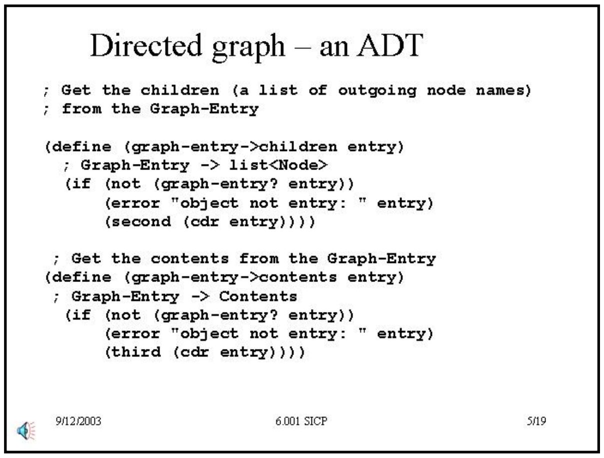
Directed graph – an ADT

```
;;; Graph Abstraction
;;;
;;;   Graph             a collection of Graph-Entries
;;;   Graph-Entry       a node, outgoing children from the
;;;                       node, and contents for the node
;;;   Node = symbol     a symbol label or name for the node
;;;   Contents = anytype the contents for the node
;;---------------
;; Graph-Entry
; make-graph-entry: Node,list<Node>,Contents -> Entry
(define (make-graph-entry node children contents)
   (list 'graph-entry node children contents))

(define (graph-entry? entry)            ; anytype -> boolean
 (and (pair? entry) (eq? 'graph-entry (car entry))))

; Get the node (the name) from the Graph-Entry
(define (graph-entry->node entry)       ; Graph-Entry -> Node
  (if (not (graph-entry? entry))
      (error "object not entry: " entry)
      (first (cdr entry))))
```

9/12/2003 6.001 SICP 4/19

Slide 31.4.4
So here is a Scheme implementation of a graph as an **abstract data type**. A graph consists of a set of **graph entries**. Each of these consists of a node, the set of children reachable from that node (which inherently defines the set of edges from the node), and the contents of the node. Think of the node as an abstract representation for a document on the Web, which we might represent by a name or symbolic label. The node also contains some real contents, such as the actual text of a Web document, or an image, or whatever information is stored at that site.
You can see a particular implementation of a graph entry, which uses a tag to label the entity as a graph entry, and a way of selecting the node (or name) from one of these entries.

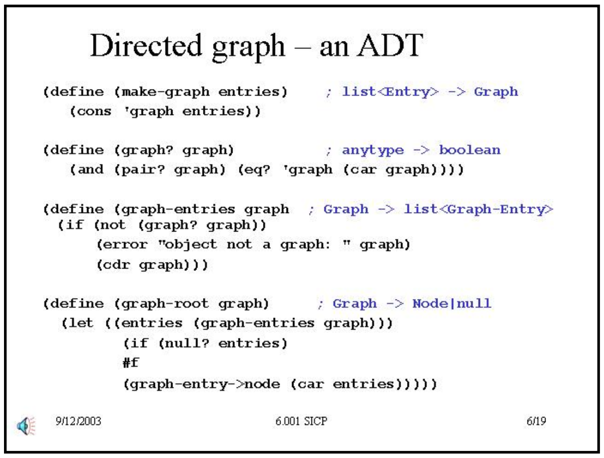
Slide 31.4.5
Just to complete the implementation, here are selectors for the children, and the contents of a graph entry.



Directed graph – an ADT

```
; Get the children (a list of outgoing node names)
; from the Graph-Entry

(define (graph-entry->children entry)
  ; Graph-Entry -> list<Node>
  (if (not (graph-entry? entry))
      (error "object not entry: " entry)
      (second (cdr entry))))

 ; Get the contents from the Graph-Entry
(define (graph-entry->contents entry)
 ; Graph-Entry -> Contents
  (if (not (graph-entry? entry))
      (error "object not entry: " entry)
      (third (cdr entry))))
```

9/12/2003 6.001 SICP 5/19



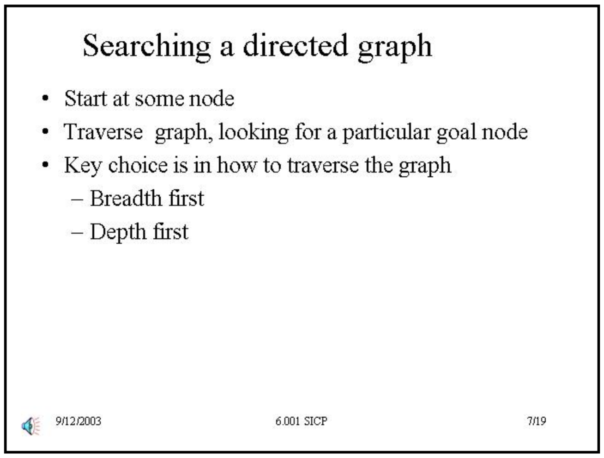
Directed graph – an ADT

```
(define (make-graph entries)     ; list<Entry> -> Graph
   (cons 'graph entries))

(define (graph? graph)           ; anytype -> boolean
   (and (pair? graph) (eq? 'graph (car graph))))

(define (graph-entries graph   ; Graph -> list<Graph-Entry>
  (if (not (graph? graph))
      (error "object not a graph: " graph)
      (cdr graph)))

(define (graph-root graph)      ; Graph -> Node|null
  (let ((entries (graph-entries graph)))
        (if (null? entries)
        #f
        (graph-entry->node (car entries)))))
```

9/12/2003 6.001 SICP 6/19

Slide 31.4.6
The **graph** data structure is then simply a labeled collection of entries, which we will store as a list. We can then use the label to identify a structure as a graph, and we have an appropriate selector to get out the entries.
Finally, we need a starting point of the graph, which we call the **root**. We will simply take this to be the first entry in the list of entries of the graph.

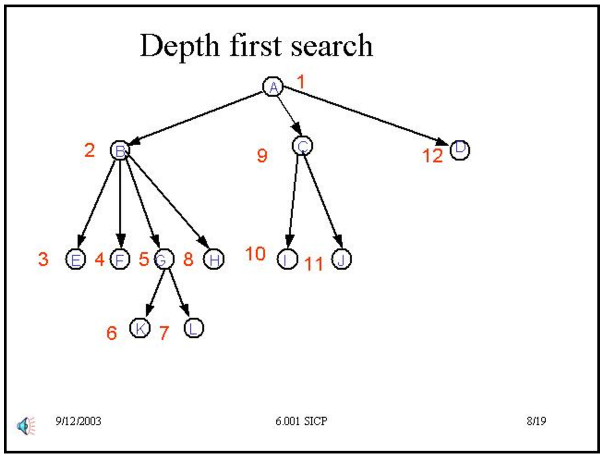
Slide 31.4.7
Now, our goal is to think about how we would search in a graph data structure. The basic idea is simple: we start at some node (say the root). We then walk along edges in the graph (which are defined by the set of children of the node), looking for a particular goal node. In the case of Web search, this might be looking for an entry that contains a particular key word. We simply want to walk along edges of the graph until we find what we are seeking, or until we have explored the entire graph. Of course, this is a very generic description, and the key open question is how to decide which children to explore and in what order. There are lots of options here: two primary ones are to use **breadth first** search, or **depth first** search. Let's briefly explore each of these ideas.



Searching a directed graph

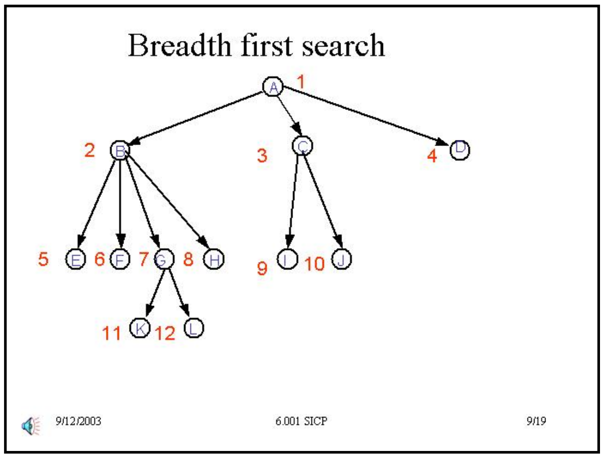
- Start at some node
- Traverse graph, looking for a particular goal node
- Key choice is in how to traverse the graph
  - Breadth first
  - Depth first

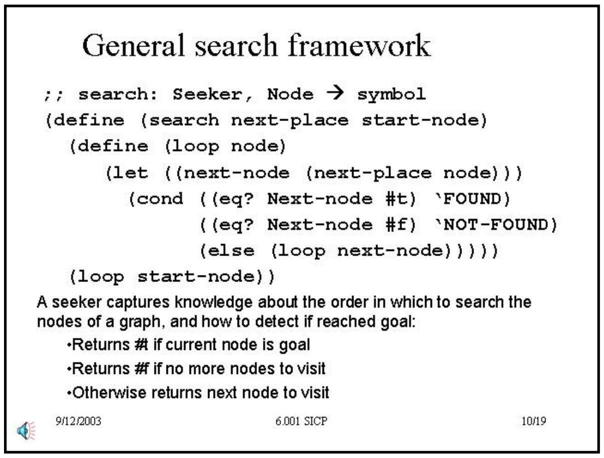
9/12/2003 6.001 SICP 7/19



Slide 31.4.8
Here is an example graph. We have labeled the nodes **A** through **L** and we have drawn the edges connecting the nodes. You can see, for example, that node **A** has three children, **B, C** and **D**, and node **B** has four children, and so on.
In depth first search, we start at the root node, and examine it to see if it is our goal. If not, we then take the set of children of the root node and move to the first element of that set. Thus if **A** is our root node, we would next go to node **B**. Note that we clearly have some choice in how we order the children of a node, which could influence the effectiveness of the search.
If node **B** is not what we are seeking, we repeat the process. That is, we take the set of children of this node, and move next to the first element of that set. You can see that we are exploring this structure **depth first**, that is, we are always moving down the structure to the first child of the node.
Eventually, however, we will reach a node that has no children. In that case, we want to **backtrack** (that is retrace our steps back to the previous parent node) and go to the next child node. Thus, using the ordering shown in red, after we examine node **E**, we will backtrack to node **B** and move to the next child node, which is node **F**, and so on. Thus the nodes are visited in the order shown, always exploring down the structure first.

Slide 31.4.9
An alternative is to explore the structure in a **breadth first** manner. This means that when we get a set of children of a node, we will first examine each child in that set, before moving on to the children of those children. Thus, in our example, you can see that we use a different ordering of the nodes, now exploring across the breadth of the child structure before moving down to the next level.





General search framework

```
;; search: Seeker, Node → symbol
(define (search next-place start-node)
  (define (loop node)
     (let ((next-node (next-place node)))
       (cond ((eq? Next-node #t) 'FOUND)
             ((eq? Next-node #f) 'NOT-FOUND)
             (else (loop next-node)))))
  (loop start-node))
```

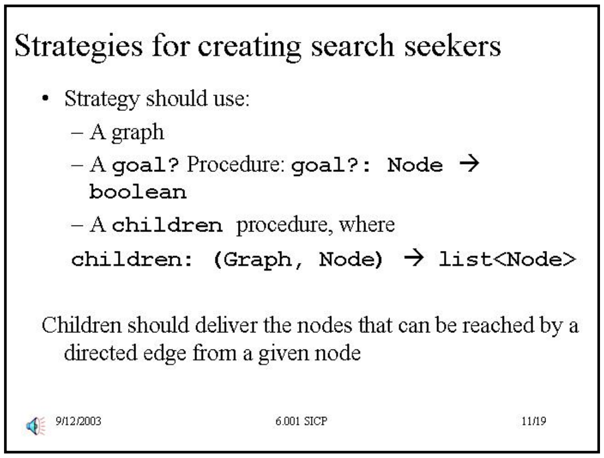
A seeker captures knowledge about the order in which to search the nodes of a graph, and how to detect if reached goal:

- Returns #t if current node is goal
- Returns #f if no more nodes to visit
- Otherwise returns next node to visit

9/12/2003 6.001 SICP 10/19

Slide 31.4.10
So how could we create an algorithm to capture these different options for searching a graph structure?
Clearly we need a starting place in the graph, and most importantly, we need a procedure that captures information about the order in which to search the nodes, and how to tell if we have reached our desired node. We call this a **seeker**, which we will write in a second. Given a seeker, and a starting point, you can see that our search process simply loops by applying our seeker to the current node. If this node is what we are looking for, we can stop. If it is not what we are looking for, then the seeker will provide us with a new node (or if there are no more, it will tell us so that we can stop the search). The trick will be to write a seeker that has the described characteristics.

Slide 31.4.11
So how do we capture different search patterns in seekers? We will create a higher order procedure that provides us with the seeker procedure. This **strategy** should use a graph, a procedure that tells us whether a current node in the graph is what we are seeking (for example, does it contain a specific key word?), and a procedure that provides us with the set of children of a particular node.
This is probably best seen by considering some explicit examples.

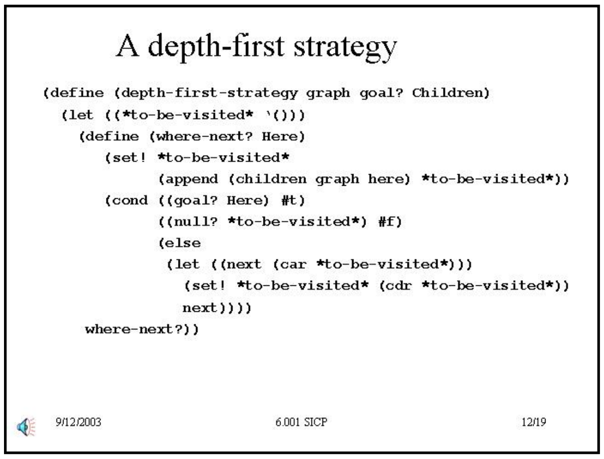


Strategies for creating search seekers

- Strategy should use:
  - A graph
  - A `goal?` Procedure: `goal?: Node → boolean`
  - A `children` procedure, where
  `children: (Graph, Node) → list<Node>`

Children should deliver the nodes that can be reached by a directed edge from a given node

9/12/2003 6.001 SICP 11/19



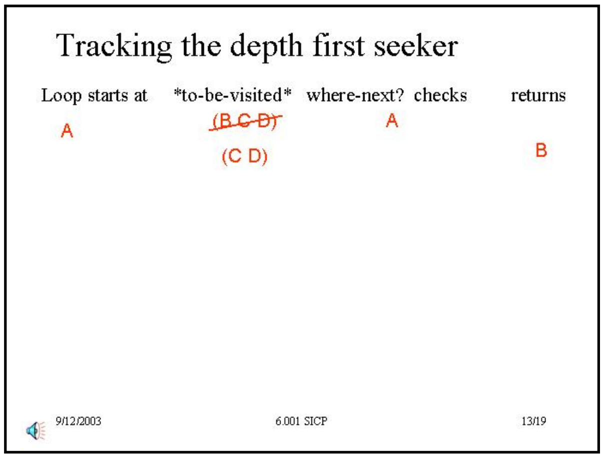
A depth-first strategy

```
(define (depth-first-strategy graph goal? Children)
  (let ((*to-be-visited* '()))
    (define (where-next? Here)
       (set! *to-be-visited*
             (append (children graph here) *to-be-visited*))
       (cond ((goal? Here) #t)
             ((null? *to-be-visited*) #f)
             (else
              (let ((next (car *to-be-visited*)))
                (set! *to-be-visited* (cdr *to-be-visited*))
                next))))
    where-next?))
```

9/12/2003 6.001 SICP 12/19

Slide 31.4.12
Here is a strategy for depth first search. Let's look at it carefully. At the top level, this strategy will take a graph, a way of deciding if we have reached our goal, and a way of getting the children of a node out of the graph (which is just some data abstraction manipulations). The strategy will return a procedure to be used in deciding if we have reached the goal, and if not, what node to examine next. Note that this procedure (called `where-next?`) has some internal state associated with it, specifically `*to-be-visited*` which is the set of nodes still to be examined.
Now, suppose we apply `where-next?` to our current node. Note what it does. It first adds the children of this node to the set of things to be explored, but it does it by adding them to the front of the list of unexplored nodes. It then applies its `goal?` procedure to see if our current location is the right one. If it is, we return true, and stop. If it isn't and there are no things left in our list of unvisited nodes, we return false, and stop. Otherwise, we take the first node out of the unexplored list, modify the list by removing that node from it, and then return that node. This will update the internal state associated with this procedure, while returning the current location to our search algorithm.

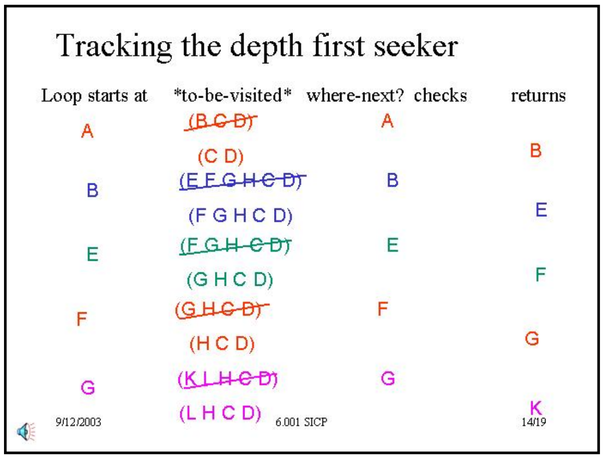
Slide 31.4.13
Suppose we apply this depth first seeker to our example graph, using the search algorithm. Let's assume that our root node is node **A**. Then we can see that the list of nodes to be visited will first be set to the children of **A**. The seeker next checks to see if **A** is our goal. Let's assume it is not. In that case, we remove the first child from the list, updating the list, and we return that first child as the next node to examine.



Tracking the depth first seeker

Loop starts at	*to-be-visited*	where-next? checks	returns
A	~~(B C D)~~ (C D)	A	B

9/12/2003 6.001 SICP 13/19



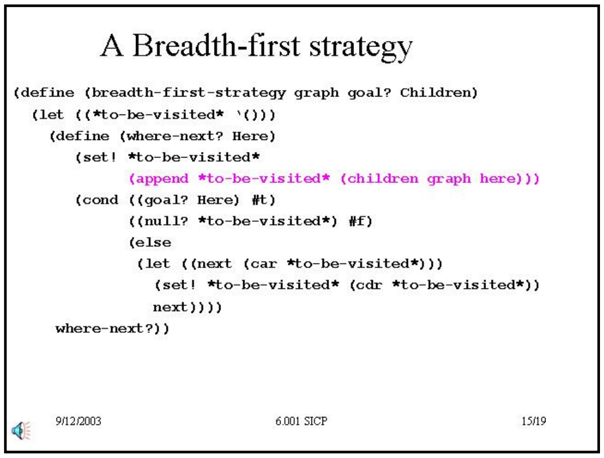
Tracking the depth first seeker

Loop starts at	*to-be-visited*	where-next? checks	returns
A	~~(B C D)~~ (C D)	A	B
B	~~(E F G H C D)~~ (F G H C D)	B	E
E	~~(F G H C D)~~ (G H C D)	E	F
F	~~(G H C D)~~ (H C D)	F	G
G	~~(K L H C D)~~ (L H C D)	G	K

9/12/2003 6.001 SICP 14/19

Slide 31.4.14
Now, let's move on to node **B**. Here we do the same thing. We first add the children of **B** to the front of our list of places to visit. Notice how this will enforce that we go depth first, that is, that we examine these children before we come back to the remaining children of the first node. We then check to see if **B** is our goal. If it is not, we remove the first child from the list, update the list, and return that child.
You can trace through the next few steps in the search, as shown on the slide.

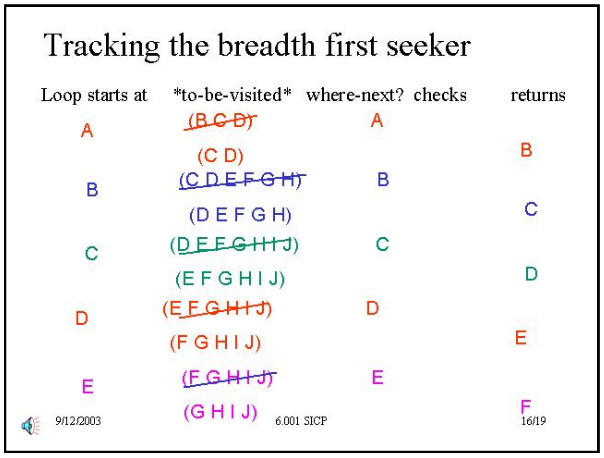
Slide 31.4.15
What about breadth first search? Well, a very small change in the code accomplishes this change in search strategy. When we update the list of nodes to be visited, we simply put the children of the current node at the **end** of the list, rather than at the **beginning**. Although this seems like a very small change, it has a big impact on the actual process.



A Breadth-first strategy

```
(define (breadth-first-strategy graph goal? Children)
  (let ((*to-be-visited* '()))
    (define (where-next? Here)
      (set! *to-be-visited*
            (append *to-be-visited* (children graph here)))
      (cond ((goal? Here) #t)
            ((null? *to-be-visited*) #f)
            (else
             (let ((next (car *to-be-visited*)))
               (set! *to-be-visited* (cdr *to-be-visited*))
               next))))
    where-next?))
```

9/12/2003 6.001 SICP 15/19



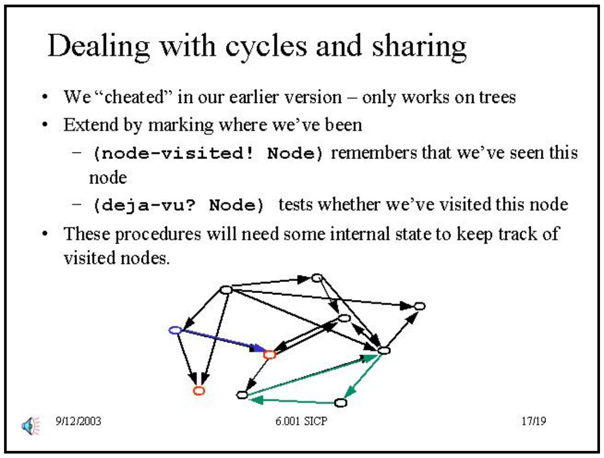
Tracking the breadth first seeker

Loop starts at	*to-be-visited*	where-next? checks	returns
A	~~(B C D)~~ (C D)	A	B
B	~~(C D E F G H)~~ (D E F G H)	B	C
C	~~(D E F G H I J)~~ (E F G H I J)	C	D
D	~~(E F G H I J)~~ (F G H I J)	D	E
E	~~(F G H I J)~~ (G H I J)	E	F

9/12/2003 6.001 SICP 16/19

Slide 31.4.16
Here is the trace of the process in this case. While the first stage is the same, note how the set of children is different in this case, and this simple change causes us to explore the graph in a very different order.

Slide 31.4.17

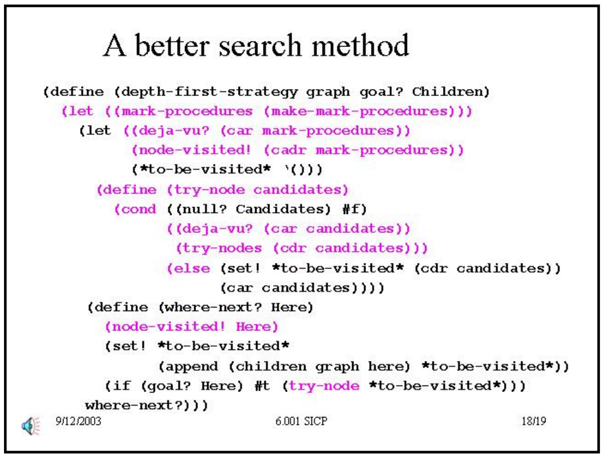


Dealing with cycles and sharing

- We "cheated" in our earlier version – only works on trees
- Extend by marking where we've been
  - `(node-visited! Node)` remembers that we've seen this node
  - `(deja-vu? Node)` tests whether we've visited this node
- These procedures will need some internal state to keep track of visited nodes.

9/12/2003 6.001 SICP 17/19

Note, however, that we cheated. Our example graph was really a tree, and in fact, the code as we have described it will only work on trees. If there is a cycle in the graph, we will get stuck going around in a circle, revisiting nodes that we have already seen, since the children of one of the nodes will include a previously visited node.
We are going to skip the details, but the idea behind fixing this is intuitively simple. We need to add a little bit of state information, allowing us to mark nodes that we have already visited. This way, we can create a procedure that tests whether a node has been seen previously, and if it has we do not add that node to the list of nodes to visit. Clearly, so long as we have a way of maintaining that bit of internal state in each node, we can accomplish this.



A better search method

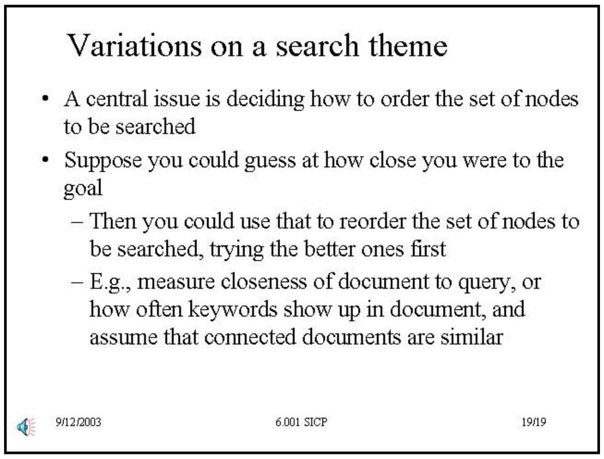
```
(define (depth-first-strategy graph goal? Children)
  (let ((mark-procedures (make-mark-procedures)))
    (let ((deja-vu? (car mark-procedures))
          (node-visited! (cadr mark-procedures))
          (*to-be-visited* '()))
      (define (try-node candidates)
        (cond ((null? Candidates) #f)
              ((deja-vu? (car candidates))
               (try-nodes (cdr candidates)))
              (else (set! *to-be-visited* (cdr candidates))
                    (car candidates))))
     (define (where-next? Here)
       (node-visited! Here)
       (set! *to-be-visited*
             (append (children graph here) *to-be-visited*))
       (if (goal? Here) #t (try-node *to-be-visited*)))
     where-next?)))
```

9/12/2003 6.001 SICP 18/19

Slide 31.4.18

Don't worry about the details of this code, but here is one way of incorporating those ideas. Here we add some additional state information to our seeker, by creating two internal procedures that mark nodes as visited and test whether a node has been visited.
We then modify which node we select to take this information into account. In particular, we first mark the node as visited, and then update the list of things to visit. If we are in the right spot, we return true, as before. Otherwise, we run through a loop in which we examine the children in order, but now if the next child is marked as having been visited, we move on to the next choice, until we reach an unvisited child.
This then allows us to deal with graphs that contain cycles, without getting trapped.

Slide 31.4.19



Variations on a search theme

- A central issue is deciding how to order the set of nodes to be searched
- Suppose you could guess at how close you were to the goal
  - Then you could use that to reorder the set of nodes to be searched, trying the better ones first
  - E.g., measure closeness of document to query, or how often keywords show up in document, and assume that connected documents are similar

9/12/2003 6.001 SICP 19/19

Of course, there are other things you could try, as variations on the search. As you can see, deciding how to order the set of nodes has a potentially large impact on how the graph is searched. Thus, if you had better ways of deciding how close you were to your goal node, you could use that to reorder the set of nodes to be visited, rather than just blindly adding them to the list. As you will see in future courses, this is a rich area of exploration with many variations on this idea of searching graph structures.
The message to take away from this lecture is that there are a variety of data structures for collecting related information. Associated with them are procedures for manipulating, maintaining, and searching that information. And, depending on which data structure, and which set of procedures we choose, we may get very different performance in terms of efficiency and ease of maintenance. Part of your job is to learn how to associate data structures with kinds of problems, in order to create efficient, robust and maintainable code.