COMP 210, FALL 2000
Lecture 37: More Vectors

Reminders:

- Final homework Friday (or Monday).
- Exam 3 handed out Friday, due the following Wednesday. Review session in class on Friday.

Review

Last class, we introduced the notion of a vector, along with some functions to manipulate vectors. In particular, we saw **vector**, **vector-ref**, **build-vector**, and **vector-set!** which, taken together, give us the tools to create and manipulate vectors. We talked about using vectors in an application where we needed constant time access to any element of the data structure, ***and*** we knew, in advance, the number of elements in the data structure. Vectors differ from structures in that we can compute the name of a vector element, while a structure element must be named explicitly and uniquely.

What about Linear Algebra?

Consider another example. Vectors are common in mathematics. A vector is a k-tuple that specifies some point in a vector space. Two important operations on vectors in linear algebra (vectors are shown in ***bold italic*** typeface) are

scalar arithmetic : s * ***v*** or s + ***i***

vector arithmetic : ***v*** * ***w*** or ***v*** + ***w***

Scalar arithmetic adjusts each element in the vector by the given number using the given operation. Vector arithmetic combines two vectors pairwise according to the given operation. Let's write programs to perform scalar and vector arithmetic:

```
(define (scalar* a-num a-vec)
  (build-vector (vector-length a-vec)
                (lambda (i) (* s (vector-ref a-vec i)))))
```

Or, more generally:

```
(define (scalar-arith a-num a-vec an-op)
  (build-vector (vector-length a-vec)
          (lambda (i) (an-op s (vector-ref a-vec i)))))
```

For vector arithmetic, we need:

```
(define (vector-arith vec1 vec2 an-op)
        (build-vector (vector-length vec1)
                (lambda (i) (an-op (vector-ref vec1 i) (vector-ref vec2 i)))))
```

Of course, for real linear algebra, we need multi-dimensional arrays. In Scheme, we can build those out of vectors of vectors of vectors …

Another example

Remember our Quicksort program? It picked a ***pivot*** element and then partitioned the list into a list of elements smaller than the pivot, a list equal to the pivot, and a list larger than the pivot, and recurred on the smaller and larger lists. In our code, we picked the first element in the list as the pivot.

What happened with that code if the elements of the list were already in order (either ascending or descending)? The pivot elements were ineffective, since they split the list into an empty list, the set of repeats of the first element, and the rest of the list. It turns out that choosing the pivot element carefully is the key to avoiding pathological worst case behavior in Quicksort. When it is sorting a list, picking a better pivot is expensive. We could take the average of the first three elements, but that doesn't help a lot with the ordered input. What we would like to do is to average a couple of different elements–-the first, the last, and the middle element; or two elements chosen at random from the set. If we use an array to hold the set of elements being sorted, we can do this without significantly increasing the cost.

One Final Example?

Say that you suddenly found yourself with 6,000,000 ballots to tally in the presidential election. Say that you distrusted someone making tally marks on a sheet of paper, and wanted to write a Scheme program to tally the results. Say that you had 10 candidates, because you were in Southern Florida, where there are many accredited parties and candidates. How might you do this tally?

Assume that the votes come in as a list of symbol

```
  (list ‘Gore “Bush ‘Gore ‘Bush ‘Gore “Bush ‘Nader ‘Buchanan ‘Nobody)
```

We could build a list of pairs–-name and total, and write a simple pair of functions. The first one, **GetVotes,** would walk the list and call the second one, **RecordVote**, on each symbol in the list. **RecordVote** could walk the list of pairs and use **set-structure!** to change the count for the appropriate name. This would take time proportional to the length of the list of votes (6 million votes) times the number of candidates (10 candidates). If this was a recount, we could speed things up by putting the two candidates who received most of the votes at the front of the list; if we messed up and put them at the end, it would be a disaster.

As several people suggested for the ranking example (the International Tiddlywinks Federation), we could build a binary search tree whose nodes held the pairs. This would reduce the search times to $\log_2 10 = 4$ probes on average from $10/2 = 5$ on average.

Could we use an array to improve this situation? Yes, but only if we had a quick (*i.e.*, in constant time) way to map each of the symbols into a small integer–- say between zero and nine. Assume that we have a function ***h(s)*** that consumes a symbol ***s*** and produces an integer between zero and nine that is uniquely associated with that integer. (Mathematically, we call ***h*** a perfect hash function, but that can wait for COMP 314.)

Given ***h***, we can implement **RecordVote** quite efficiently. It becomes

```
(define Votes (build-vector 10 (lambda(n) 0))

(define (RecordVote asym)
    (local  [(define index  h(asym))]
        (vector-set!  Votes index (add1 (vector-ref Votes index)))
      ))
```

This uses the vector **Votes** and the function ***h*** to make the process a factor of five to ten more efficient.

Summary

When should we use a structure, a list, and a vector?

(define-struct fee (a b c))	creates a value with three components
(list `a `b `c)	creates a list with three components, but we can add arbitrary additional elements (We can also use **build-list** …)
(build-list 10 (lamda(n) n))	
> (list 0 1 2 3 4 5 6 7 8 9)	
(list-ref (list 0 1 2 3 4 5 6 7 8 9) 7)	
> 6	*requires 7 recursions….*
(vector `a `b `c)	creates a vector with three components, and we cannot add any more elements …
(build-vector 10 (lambda(n) n)	
> (vector 0 1 2 3 4 5 6 7 8 9)	
(vector-ref (vector 0 1 2 3 4 5 6 7 8 9) 7)	

> 6 *requires no recursion (is instantaneous)*

Both **build-list** and **build-vector** create arbitrarily long aggregate objects, but the one returned by **build-vector** has a ***fixed*** length, while the one returned by **build-list** can be extended.

If a program inspects or processes all of its elements, use a list. If it inspects or processes elements in a seemingly random order (an order dictated by external numbers), use an array.