Data Structures:

A data structure is way of arranging data to make processing the data more convenient or more efficient.

So whether something is a good data structure or not depends on what you intend to do with the data. Different data structures are good for different purposes. Data structures are tools.

There is an aphorism in CS: "Pick the right data structure and the algorithm will design itself."
What this means is that if you pick the right data structure, an efficient algorithm will be relatively easy to design.

So, you need to know what different data structures are good for. Know their characteristics.

There are a number of data structures in common use, and you will learn about them in later courses. You can also invent your own data structures.

There are three fundamental ways you can be creative in CS:
- invent new data structures
- invent new algorithms
- invent new combinations of data structures and algorithms.

Often in OOP the data structures and algorithms might be combined together into a class (called an Abstract Data Type, discussed in ch. 19).

Linked Lists:

Linked lists are very important data structures, but they also illustrate how data structures can be better or worse for different purposes.

In a linked list each element is linked (by a pointer) to the next element, and the elements do not have be contiguous or consecutive in memory.

Every data structure makes tradeoffs: some things are easier or more efficient, other things are harder or less efficient. What are the pros and cons of vectors and linked lists?

Vectors (or Arrays):

* efficient "random access," because you can compute (very quickly, in constant time) the address of any element of the vector.

*inefficient insertion or deletion, because I may have to move a lot of the vector's elements (copying, not constant time).

Linked Lists:

* inefficient random access. They are sequential (as opposed to random access), because I have to traverse the linked list sequentially to get to the element I want.

* efficient insertion and deletion, because I just need to change a couple of pointers (constant time).

Linked structures are generally good when you need to do a lot of insertions and/or

deletions.

If you try to traverse a circularly linked list, you will end up in an infinite loop.

Is it possible to write a program that will tell if your program (or any program) will go into an infinite loop? **No!**

One of the important contributions of Turing (in the 1930s)was to prove "The Undecidability of the Halting Problem." (It is a fundamental result, like the impossibility of a perpetual motion machine in physics; you will learn about it in later CS courses.)

A decision procedure for the Halting Problem would be a program that would take as input any program P and its input X and eventually give you a definite Yes or No about whether P halts when run on input X. Turing showed that no such decision procedure is possible. He showed how the existence of a decision procedure would allow construction of a program that halts if and only if it doesn't halt, which of course is impossible, and so the decision procedure cannot exist.

He did this by a technique analogous to the Liar Paradox: "This is a lie." He said, in effect, you give me a function for the decision procedure, and I will construct a paradoxical program Q that has the following behavior: Q asks the decision procedure if Q halts, and if it says Yes, then Q goes into an infinite loop. If it says No, the Q halts immediately.

Rice showed that the undecidability of the Halting Problem implies the undecidability of many other interesting questions. He did this by showing that if you could write programs to decide these other questions, then you could also solve the Halting Problem. Since you can't decide the Halting Problem, you can't decide

these other ones either.

These results establish fundamental limits on the power of any possible computer.