Exam comments:
If it looks like you have a correct answer, and for some reason we marked it wrong, by all means, tell us. However, begin with your TA.

Please don't haggle over partial credit. The TAs and I have decided in advance how much partial credit to give for various sorts of mistakes, and we try to be consistent across the whole class.

Ensembles of Data and Data Structures:

Computers are often most useful manipulating large amounts of data, but the way the data is organized can have a significant effect on how easy and efficient it is to manipulate the data. So different ways of organizing the data are called data structures, and much of what we study in CS is data structures (beginning with CS140).

What are some of the considerations in data structures?

(1) How is the data going to be accessed? Is it sequential access? Or "random access"? For sequential access, can you go in both directions or only one? With sequential access, it's most efficient to access whatever comes next in order. And the further away it is, the less efficient it is. Random access means that it's equally efficient to access any element of the data structure. The efficient way to process the data in a sequential data structure is in order from front to back. In a random-access structure, it is equally efficient to process it in any order (in this sense "random").

So when you are selecting or designing a data structure for some application, you have to think about how you will want to access it (e.g., sequential or random), and

pick the appropriate one.

(2) Is the data only readable, or can you modify it?

(3) Is the structure fixed (static) or can it be changed (dynamic)? In particular, can the data structure grow or shrink in size. (Sometimes we don't know in advance how big the data structure should be.)

(4) Does all data have to be of the same type? For example, in a given data structure, does it have to be all doubles? Or can can you have doubles, ints, strings, etc. all in one data structure. I.e., is the data structure homogeneous or heterogeneous. Think of various kinds of forms that a business might have to process (orders, bills, payments, etc.). Do we have one stack containing all different kinds of forms, or a different stack for each kind?

(5) Does the order of the data matter? I.e., is it more like a sequential list or row of things, or like a bag of things?

These are some of the main considerations.

Which data structures you pick is an engineering decision, in which you the various pros and cons of a data structure and try to pick the best ones for what you are trying to do.

An important CS rule of thumb: *"Pick the right data structure, and the algorithm will design itself."*

What this means is that if you make a good choice of data structure, it will be very easy to design an efficient algorithm for the task. So you need to think carefully

about how to structure the data to make what we want to do as simple and efficient as possible.

Where do you get data structures?

(1) There are some built into the programming language (in the case of C++, there are C-arrays and structs).

(2) There are standard libraries, that have been implemented by expert programmers, so that they are efficient and reliable to use. In C++ there is the STL (Standard Template Library).

(3) You may buy it or get it somewhere else.

(4) You may invent and implement your own new data structure.

Examples: A stack (or LIFO) is a last-in first-out data structure.

A queue (or FIFO) is a first-in first-out data structure.

Vectors are a random-access, ordered homogeneous data structure.

You know about 2D and 3D vectors in math, such as (3, 4), or (3, 4, -10). Notice that order matters: (3, 4) and (4, 3) are two different vectors. Suppose V = (3, 4, -10), we can talk about the vector as a whole by the name V. We can also talk about the elements of the vector, e.g., $V_1 = 3$, and $V_3 = -10$.

The idea in C++ is very similar, but:

(1) Vectors can be of any dimension (e.g., 365 for a vector of the rainfall on each day of the year).

(2) C++ uses 0-origin indexing (whereas in math we usually 1-origin). So if V is the vector (3, 4, -10), then V[0] == 3, V[1] == 4, V[2] == -10.

(3) C++ allows vectors of any type of data, e.g., doubles, ints, strings, bools, and even other vectors. But the vector data structure is homogeneous, so all the elements of any one vector must be the same type. This is so the compiler can calculate the location of vector elements efficiently and give you the random access property. Accessing any particular vector takes "constant time"; that is, time independent of the position of the element in the vector.