

Data Structures

These lessons have covered many different data structures and algorithms. Using the proper data structure to hold your data can make a big difference in the performance of your application. If you choose poorly, the app will be slow, and the code will look clunky. Imagine sorting a large list of structures stored in an array. There is more work for the processor to do, and much more work for the programmer to write all the details. If you had chosen to build a tree there would be no sorting necessary, since that occurs when you build it. If you had chosen a linked list there would be no data moving at all, only the pointers would be modified.

Compare using a linked list to hold your queue, as compared to an array for the same job. With the linked list you simply add to the tail and remove from the head. If you use an array, you will need to move all the data for each step. You could make your array circular, but, then you are limiting how many members can be in the queue.

Examine your data and the requirements of your application. You want to move as little data as possible, no matter which data structure you use. It is just faster that way. Will you be searching and sorting your data frequently? Then choose a data structure which makes that simple and fast. If you require a permanent, unchangeable list use an array for its small size and efficiency. Does your algorithm use recursion? Then keep track of how much stack space it uses. If you are not careful you will run out of room and errors will arise.

Consult the web or other references about the data structure you want to implement. Check the big O analysis notes on it. Determine the time and space used for each of the operations you will run on your data structure – inserting, deleting, traversing, searching, and sorting. Think of the various cases your operations may encounter. Is the data already sorted or close to it? Then some sorting algorithms will be quite slow. Sorting algorithms often perform best on randomly distributed data. Study all the sorting methods in these lessons, and other methods you have found in your references. Some of them are quite sensitive to input ordering while others are not.

You also need to examine the memory use of your data structure and the algorithms you are using to manipulate it. Is your data structure stored in heap or stack memory? Using the latter may be efficient, but it limits you to smaller data sets. Recursion makes writing algorithms simpler, but it has hidden problems. At each recursion a new stack frame is built. As you recurse ever deeper, more and more stack frames eat up your stack memory. Be mindful of this; you may even need to build tools to monitor how much stack memory is remaining. When you run out your application will crash in unexpected ways. This may not be easy to debug unless you remember how you solved your problem, and the memory use it entails. Write notes for your own sanity.

Structures

- Arrays
 - Heaps
- Linked lists
 - Single link
 - Queues
 - Doubly linked
 - Stacks
 - Arrays of linked lists 1D
 - Bins of linked lists 2D
- Trees
 - Binary trees 1D
 - Quad trees 2D
 - Oct trees 3D
 - kd-trees
- Graphs
- Hash Tables

Algorithms

- Search
- Sort
- Depth first
- Breadth first

Operations

- Inserting
- Deleting
- Walking
- Searching
- Sorting

Arrays are good for finding the middle of a list. Using an index can be useful for binary searches. Arrays are not good for data insertion or deletion.

Linked lists are good for insertion or deletion, but not for finding the middle. Singly linked lists are good for queues, but not for stacks. For those a doubly-linked list is more appropriate.

Trees are good for keeping data sorted as it is added. Insertions are easy. Deletions are not. Searching is fairly rapid, depending on the number of branches from each node.

Graphs are good for showing connections, and the costs of those connections. Insertions and deletions are both difficult.

Quad trees, Oct trees, and kd-trees are useful for $1/\text{distance}$ or $1/\text{distance}^2$ scenarios, where local effects are much stronger than more distant ones.

Binning data structures help find neighbors when the forces are local, not global in effect.