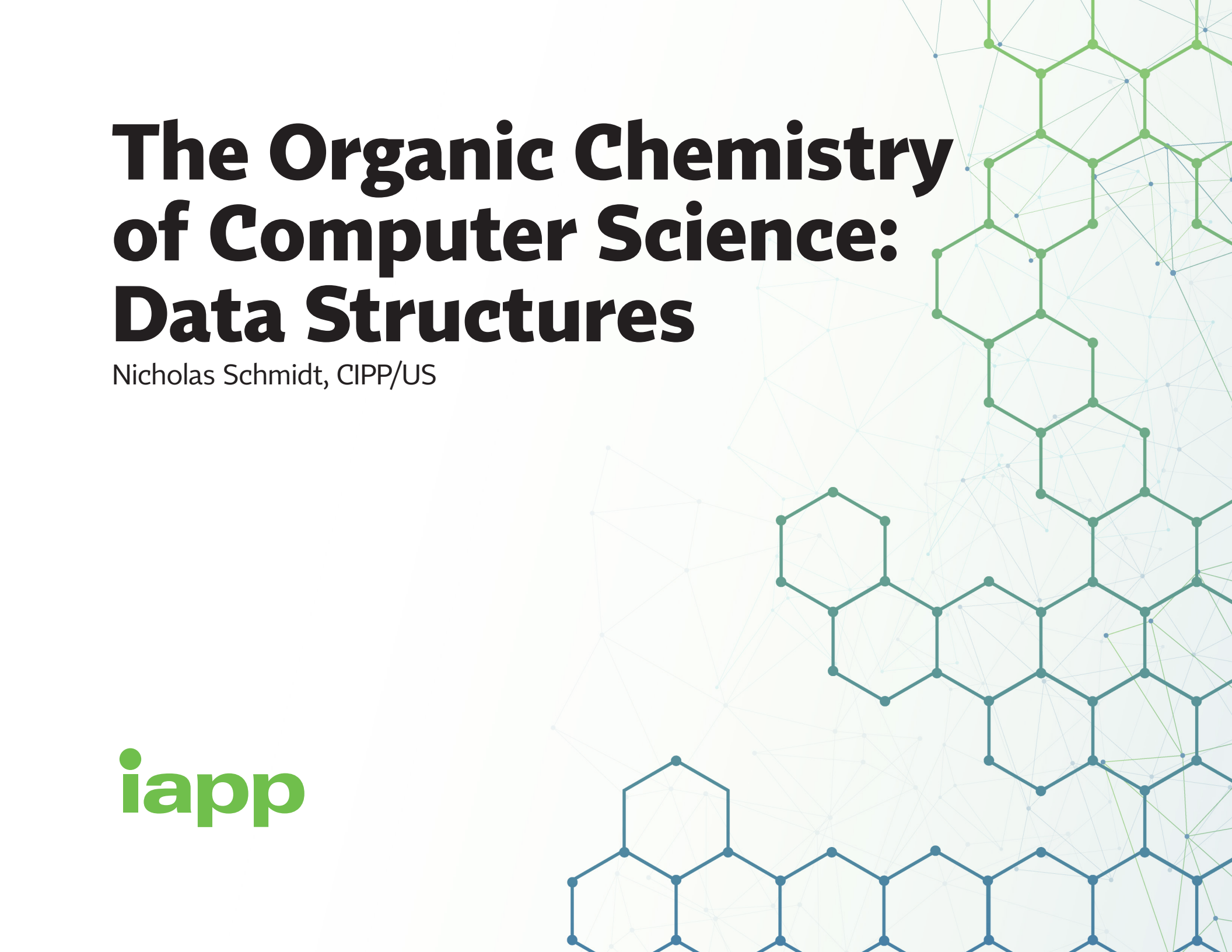
The Organic Chemistry of Computer Science: Data Structures

Nicholas Schmidt, CIPP/US



The Organic Chemistry of Computer Science: Data Structures

A course in data structures, the organization and movement of data within a program, holds the same place in a computer science curriculum that organic chemistry holds in a pre-medicine curriculum, a difficult and dreaded course widely derided as a "GPA-killer." However, this difficulty exists for a good reason, it is in the data structures course that students truly learn to "think like a programmer." Just as an understanding of organic chemistry is fundamental to modern medicine, an understanding of data structures is fundamental to understanding technology. It is the design of the data structures, not the quality of the code implementing them, which will determine if a program functions well and holds together over the long-term.

In this fourth installment on a series of white papers for privacy pros, we'll present a summary of this complicated but crucial topic, focused on the subject's privacy professionals need to understand in order to fully grasp other technical topics which will be introduced in future white papers.

Why are data structures so important?

The fundamental power of data is in its ability to connect to other data. The address "10 Downing Street, London" has no meaning unless you know it is the Prime Minister's address. The number "1" is meaningless unless you realize it is Notre Dame's proper college football ranking. This is especially true in a statistical database, large databases of information about specific members of a limited or general population for the purposes of drawing statistical inferences about the population.

What data structures do inside of a program is what databases do on a hard disk: connect, group, and link data. Data structures are the "bones" programs are created around and, like bones, they are designed for the task they are intended to perform. For example, a decision tree allows the user to move through different data records in succession, picking the next record based on the needs of the moment while a queue requires users to follow a pre-determined route through the data. It is incredibly useful

in AI but would cause a riot if used to process orders at a bar. The opposite is true of a queue, it is the ideal way to handle work in a sequential matter but would make a stilted AI.

One important thing to remember is that the smart design of data structures is one of the primary vectors for inference control, or control over how users can access information. When dealing with a data structure for sensitive data, it is important to ask questions such as "how can this data structure be used?" and "is it possible to exploit this data structure in an undesirable way?" Although it is not always possible to design data structures that wholly prevent access to sensitive information and there are other methods of inference control, they are one of the best methods of inference control. If the bones of the program do not even allow a sensitive inference, it is overwhelmingly difficult to draw.

Common building blocks of programs and data structures

In traditional programming, a program follows a sequential, though sometimes looping, set of instructions, then exits. Most programs have variables, identifying names that point to a location in memory that holds a value, which may change as the program executes. More complex programs may also have functions, or separately-defined sets of instructions outside the main sequence of commands that can be referenced by it, essentially inserting the repeating instructions into the main sequence. Functions can call themselves, this is known as recursion. Variables can be local, limited to a specific function, or global, shared by the entire program. This short description fully encompasses most of the early programming languages, including classic Fortran, classic Perl, C, and Pascal.

Every variable has a data type. The data type is the nature of information contained within the variable. There are many different kinds of data types, but all data types ultimately whittle down to some combination of the primitive data types: characters (one letter, numeral, or other symbol), numbers (integers and various implementations of decimals), booleans (true/false), and references or pointers (locations of other information in memory). The following three program examples illustrate the concepts above:

Program 1 (in C):

Data Type (integer) → `int x = 0;`

Function (shows x on screen) → `printf (x);`

Variable → `x`

Operator (special function) → `x+1`

```
int x = 0;
printf (x);
x = x+1;
printf(x);
```

Output of Program 1:

```
0
1
```

Program 2 (Pseudocode):

```
Do Thing 1
Do Thing 2
Function()
Do Thing 1
Function()
Do Thing 5
```

Definition of `Function()`:

```
Do Thing 3
Do Thing 4
Go back to Main Program
```

Actual Sequence of Events in Progam 2:

```
Thing 1
Thing 2
Thing 3
Thing 4
Thing 1
Thing 3
Thing 4
Thing 5
```

In `Function()`

Program 3 (Pseudocode):

```
Do Thing 1
Function()
Do Thing 4
```

Definition of `Function()`:

```
Thing 2
If third time
   finish function early
Function()
Thing 3
```

Recursion → `Function()`

Actual Sequence of Events in Progam 3:

```
Thing 1
Thing 2
If third time (no)
Thing 2
If third time (no)
Thing 2
If third time (yes)
finish function early
Thing 3
Thing 3
Thing 4
```

1st time in `Function()`

3rd time in `Function()`

2nd time in `Function()`

Object-oriented programming is a design technique that allows users to create custom complex data types called objects. The code defining and implementing an object is called a class and an individual occurrence of an object is called an instance. The class defines a list of contained variables and objects, alternatively called properties, fields or attributes. They also contain custom functions, called methods, which can be called either within the instance or by other code outside of it. Together, properties and methods are called class members. Classes can be built on top of other classes using inheritance, where a child class inherits a parent class's members. Refer to the following example.

The apple object is defined by the `apple` class. Here's a specific instance of apple:

Methods:

```
eat
cut_up
throw
feed_to_horse
make_pie
dip_in_peanut_butter
decore
make_jam
make_apple_sauce
```

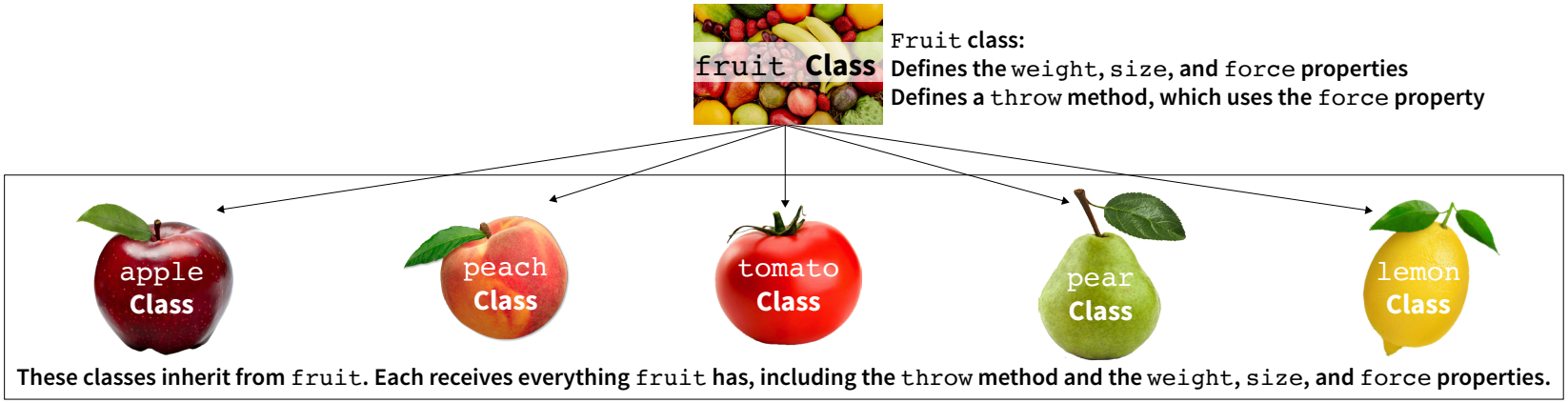
Properties/Fields/Attributes of Apple:

```
color = red
weight = 100 grams
size = 7.5 cm
force = 1 N
leaf = true
stem = true
worm = false
type = "Honeycrisp"
spoils = 11/5/18
```

The `apple` class inherits from the `fruit` class.

Inheritance is where the real power of objects becomes clear. It is possible for a child class to redefine a method while allowing it to use the same name. This is called overriding. Because the child's overridden method and parent's original method are the same, they can be called in the same way without any need to account for the underlying difference between them. This is called polymorphism, and a powerful technique for handling complex differences between classes quickly and easily. Interfaces, or blank classes meant to be inherited and implemented by override, also use polymorphism to allow the same code to work with wide varieties of different objects, saving significant implementation work.

Some objects are designed for the specific purpose of storing groups of other objects. These are called containers or collections and are how data structures are usually implemented in code. Most modern programming languages contain several container implementations. Although it is also possible for objects to connect to one another using reference variables, containers provide a helpful wrapping that makes the collection useful.



Inheritance and polymorphism present privacy concerns for software companies that want to make a code base public. A malicious programmer could inherit from a class and then, in the child class, draw on sensitive inherited properties and methods they shouldn't have access to, like the underlying database. Additionally, the programmer could override a crucial function to behave unexpectedly in a way that damages the entire system. These exploits are prevented by the use of access modifiers, which designate which other entities can call the method. The four most common access modifiers are:

Public — Public members are fully heritable and can be called by any other piece of code using the class.

Internal — Internal members (and the little brother to the other modifiers) can only be inherited or called by code that has been packaged and compiled together with the class definition in a package called an assembly.

Protected — Protected members can be inherited/overridden but can only be called by other code within that instance or the inherited instance.

Private — Private members cannot be inherited/overridden and can only be called by code within the same instance of the object.

Object Oriented Programming has almost entirely replaced traditional, non-object programming paradigms for complex tasks and application development. Most modern programming languages support it, including C++, Python, modern Perl, and Javascript. The two most popular "heavy duty" development languages, Java and C#, use it exclusively.

Time complexity and the Bachmann-Landau notations

To pick the right data structure for a task, one must understand the data structure's power in accomplishing that task. In computer science, power is a function of time complexity, or the measure of how the processing time required by an algorithm (set of instructions) grows as the amount of data it must process grows. Time complexity is most commonly represented one by one of the Bachmann-Landau notations. They are sometimes also called the asymptotic notations because they describe the behavior of an algorithm as the amount of data it must process approaches infinity.

There are three major Bachmann-Landau notations (*Editor's note: The definitions have been streamlined for simplicity.*):

Big O notation represents how the processing time will grow in the worse-case scenario. For example in sorting, Big O notation represents the amount of time to sort a list that has already been sorted into the opposite order. Big O is the most important notation and is usually the only one reported. Sometimes Big O is incorrectly used to report Big Ω or Big Θ results and the term is often used as a metonym for the entire family of Bachmann-Landau notations.

Big Ω notation represents how the processing time will grow in the best-case scenario. For example in sorting, Big Ω notation represents the amount of time to "sort" a list which has already been sorted into the desired order.

Big Θ notation is used to indicate that an algorithm has the same Big O and Big Ω.

These notations can be derived using mathematical limits and , which represents the number of records the algorithm must process. Rather than fully resolving the limit, report the order of the final term before deriving the numerical result. Consider the following examples:

Looping algorithm	Time requirements
`Preparation work`	5 minutes constant time
`Loop over each datum{`	Loop n times
`  Loop over each datum{`	Loop n times
`  Do work`	2 minutes constant time
`  }`	
`}`	
`Other work`	10 minutes constant time
`Loop over each datum{`	Loop n times
`  Transformation work`	4 minutes constant time
`}`	
`Other work`	2 minutes constant time

Big O notation

$$f(n)=5+n*n*2+10+n*4+2=2n^2+4n+17$$

The limit of $f(n)$ as it approaches infinity, or $\lim_{n\to\infty} 2n^2+4n+17$, will behave most like $\lim_{n\to\infty} 2n^2$ since that term of the polynomial will grow the fastest.

Therefore, we say $\left(\lim_{n\to\infty} 2n^2+4n+17 \approx \lim_{n\to\infty} 2n^2\right)=\infty$
Because Big O is an approximation, we drop the coefficient and report that the algorithm is $O(n^2)$.

Constant algorithm	Time requirements
`Some work`	3 minutes constant time
`Some other work`	4 minutes constant time
`Some more work`	10 minutes constant time

Big O notation

(Editor's note: The following solution has been streamlined for simplicity.)

$$f(n)=3+4+10=17$$
$$\lim_{n\to\infty} 17=17$$

Because Big O is about approximation, we report constant time like this as $O(1)$ or $\Theta(1)$.

Recursive algorithm	Time requirements
`Function (n) {`	
`  Loop n times{`	Loop n times
`  Function(n-1)`	Repeat the function $(n-1)$ times, looping once
`  }`	less time each time, therefore
`}`	$(n-1)*(n-2)*\ldots*2*1$ times.

Big O notation

$$f(n)=n*((n-1)*(n-2)*\ldots*2*1)=n(n-1)!=n!$$
$$\lim_{n\to\infty} n!=\infty$$

The Big O notation is $O(n!)$.

Here is a list of a table of the most commonly encountered Big O values:

Big O	Meaning	Description
$O(1)$	Constant time	The algorithm always takes the same time, regardless of the amount of data it is handling.
$O(n)$	Linear	Every new datum an algorithm must handle will add the same amount of time.
$O(n^2)$	Quadratic	The amount of time added to an algorithm increases linearly with every new datum.
$O(n^3)$	Cubic	The amount of time added to an algorithm increases quadratically with every new datum.
$O(n^x)$	Polynomial (generally)	The amount of time added to an algorithm increases by $O(n^{x-1})$ with every new datum.
$O(\log n)$	Logarithmic	The amount of time added to an algorithm by each new datum decreases exponentially.
$O(e^n)$	Exponential	The amount of time added to an algorithm by each datum increases exponentially.
$O(n!)$	Factorial	The amount of time added to an algorithm increases at a factorial rate.

A primer on common data structures

All data structures necessarily utilize or adapt an abstract data type, which describes the structure mathematically. When well-implemented, all data structures of the same abstract data type have the same time complexity for the same operations. The development of new abstract data types is one of the purposes of discrete mathematics, the study of non-divisible numbers, graphs, and logic. Lawyers and MBAs no doubt have fond memories of this area of math from the logic games section of the LSAT or the integrated reasoning section of the GMAT.

Most programming languages offer generalized, container implementations of each of the ADTs as part of their standard object libraries. These implementations can be inherited from and adapted to suit the specific needs of your organization. ADTs break down into three broad types:

Associative data structures — These data structures store data in a way where each datum has a relationship with a "key" or identifier, such that when the identifier is given to the data structure, the data structure will return the linked value. The datum's position in the structure or relation to other data is irrelevant. They can be visually depicted in an unordered list or look-up table.

Linear data structures — These data structures carry data in a way where data are placed in an ordered sequence. They can be visually depicted in an ordered list or list of lists/matrix.

Graph data structures — These are data structures where data interconnect with one another, such that it is possible to "navigate through" the data structure. The connections between data matter more than the sequential position. They can only be fully depicted visually by chart or graph.

Here is a list of common data structures, some of which riff off the same abstract data types:

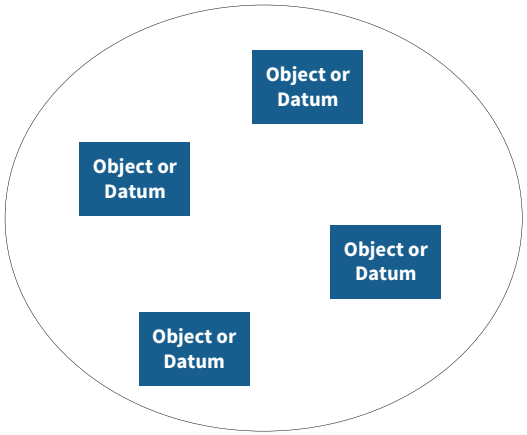
Associative data structures

Set

A collection of unique elements with no apparent organization other than their grouping together. Technically speaking, a set is an associative array where an element is also its own key. Primarily used for membership checking and similar functions, not for data storage.

Typical behavior:

Find/Retrieve an element	$O(n)$	$\Omega(1)$
Insert an element	$\Theta(1)$	
Remove an element	$O(n)$	$\Omega(1)$
Memory footprint	$\Theta(n)$	



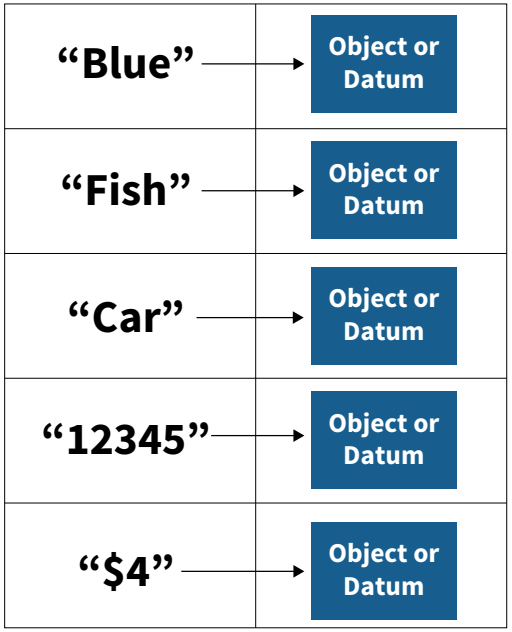
Associative array (aka hash/map/symbol table/dictionary)

A collection of elements, unique or not, where each element is identified by a separate value, called a key. Associative arrays are usually regular arrays where the index is determined by a hash function rather than sequentially.

Typical behavior (assuming no collisions):

Retrieve an element	$\Theta(1)$	
Find an element	$O(n)$	$\Omega(1)$
Insert an element	$\Theta(1)$	
Remove an element	$\Theta(1)$	
Memory footprint	$\Theta(1)$	

Even though time and memory are constant, they may be very large due to the complexity of hashing functions and the need for lots of space to avoid collisions (where two keys have the same hash).



Associative data structures (continued)

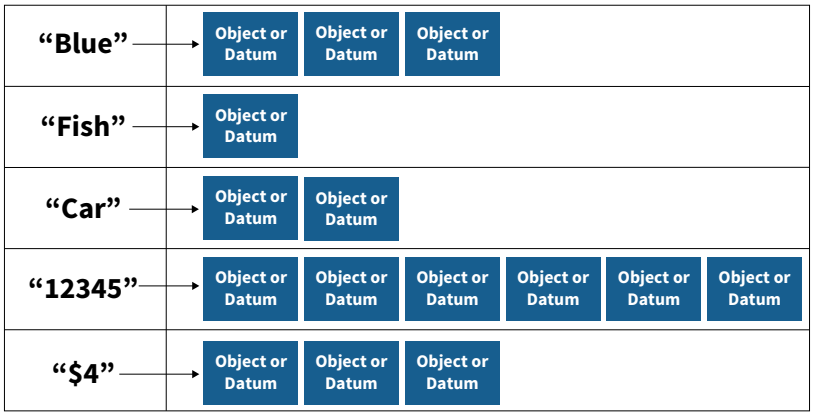
Multimap/Multidict/MultiHash

An associative array where each key points to a linear data structure (usually an array) that can contain multiple elements.

Typical behavior (assuming no collisions and an array):

Retrieve an element	$\Theta(1)$	
Find an element	$O(n)$	$\Omega(1)$
Insert an element	$O(n)$	$\Omega(1)$
Remove an element	$O(n)$	$\Omega(1)$
Memory footprint	$O(n)$	$\Omega(1)$

Time and memory may be very large due to the complexity of hashing functions and the need for lots of space to avoid collisions (where two keys have the same hash).



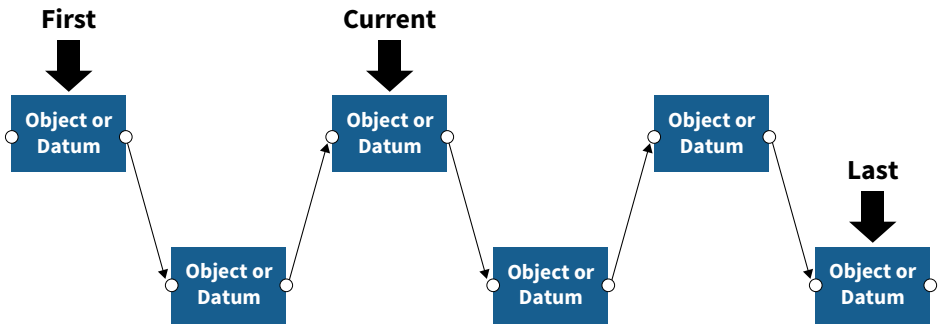
Linear data structures

Linked list

A collection of elements where each element holds a reference to the next element in the sequential order. The linked list container holds a reference to the first element in the sequence, and occasionally to the last or currently focused-on elements. Elements may be in different memory locations, making linked lists flexible. Linked lists are the base linear data structure.

Typical behavior (assuming only a "first" pointer):

Retrieve an element	$O(n)$	$\Omega(1)$
Find/Remove an element	$O(n)$	$\Omega(1)$
Insert an element	$O(n)$	$\Omega(1)$
Memory footprint	$\Theta(n)$	



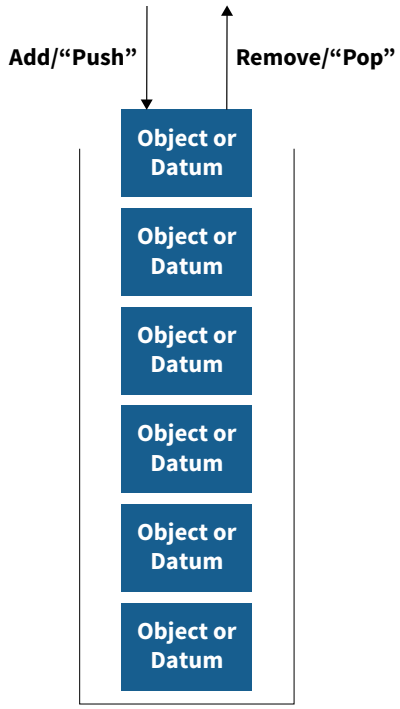
Linear data structures (continued)

Stack

A list of elements in the order they were added to the collection. Elements are taken off the list in last in, first out order.

Typical behavior:

Retrieve/Delete top element	$\Theta(1)$	
Find/Remove specific element	$O(n)$	$\Omega(1)$
Insert an element	$\Theta(1)$	
Memory footprint	$\Theta(n)$	

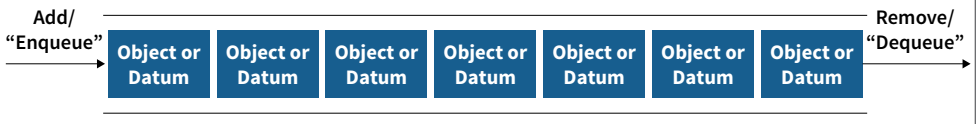


Queue

A list of elements in the order they were added to the collection. Elements are taken off the list in first in, first out order.

Typical behavior:

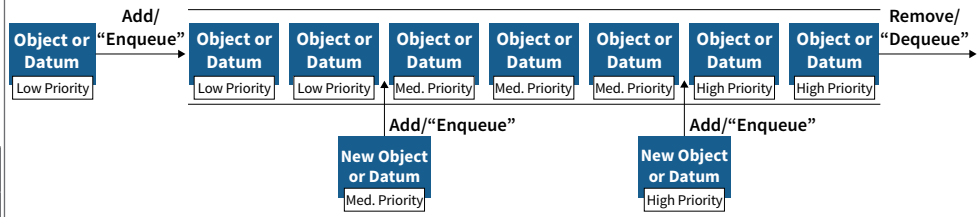
Retrieve/Remove top element	$\Theta(1)$	
Find/Remove specific element	$O(n)$	$\Omega(1)$
Insert an element	$\Theta(1)$	
Memory footprint	$\Theta(n)$	



Linear data structures (continued)

Priority Queue

A list of elements, each with an independent priority, in the order they were added to the collection. Elements are removed from the queue by priority, with ties being broken in favor of the first added element. Priority queues are usually implemented as heaps (below). It is also possible to create a priority stack.

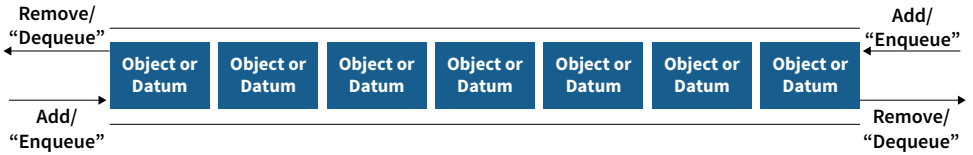


Typical behavior (implemented as a binary heap):

Retrieve highest priority element	$\Theta(1)$	
Remove highest priority element	$O(\log n)$	$\Omega(1)$
Find specific element	$O(n)$	$\Omega(1)$
Insert an element	$O(\log n)$	$\Omega(1)$
Reorder by priority	$O(n \log n)$	$\Omega(n)$
Memory footprint	$\Theta(n)$	

Double-ended queue

A combination of a stack and a queue, where elements can be added or removed from either side. Double-ended priority queues are also possible.



Typical behavior:

Retrieve first/last elements	$\Theta(1)$	
Remove first/last elements	$\Theta(1)$	
Find specific element	$O(n)$	$\Omega(1)$
Insert an element	$\Theta(1)$	
Memory footprint	$\Theta(n)$	

Linear data structures (continued)

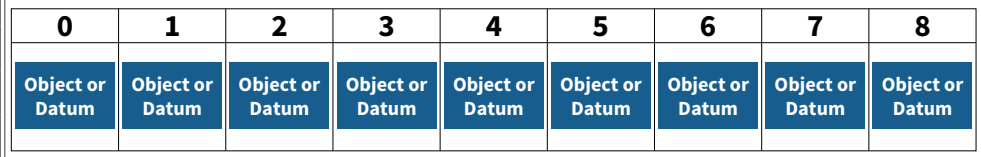
Array/Tuple/Vector/List

A list of elements, with each element assigned a number, or index, signifying its location within the array. Arrays can be of fixed or dynamic size. An array of characters is called a string and is the most common way to hold text.

Typical behavior (implemented as a binary heap):

Retrieve any element	$\Theta(1)$	
Add/Remove an element	$O(n)$	$\Omega(1)$
Find specific element	$O(n)$	$\Omega(1)$
Sort (insertion algorithm)	$O(n^2)$	$\Omega(n)$
Sort (bubble algorithm)	$O(n^2)$	$\Omega(n)$
Sort (merge algorithm)	$O(n \log n)$	
Memory footprint	$\Theta(n)$	

The general sorting algorithms given can be hybridized for better results, other popular algorithms, such as quicksort, also exist.



Stream

A pointer to a file on the hard drive or other input source, such as a text box. This pointer slowly and sequentially advances over the file, reporting each piece of data (usually though not always a character) until it reaches a special character that tells it to terminate.

Typical behavior:

Retrieve next character	$\Theta(1)$	
Add/Remove a character	Not possible	
Search for character(s)	$O(n)$	$\Omega(1)$
Memory footprint	$\Theta(1)$	

Graph data structures

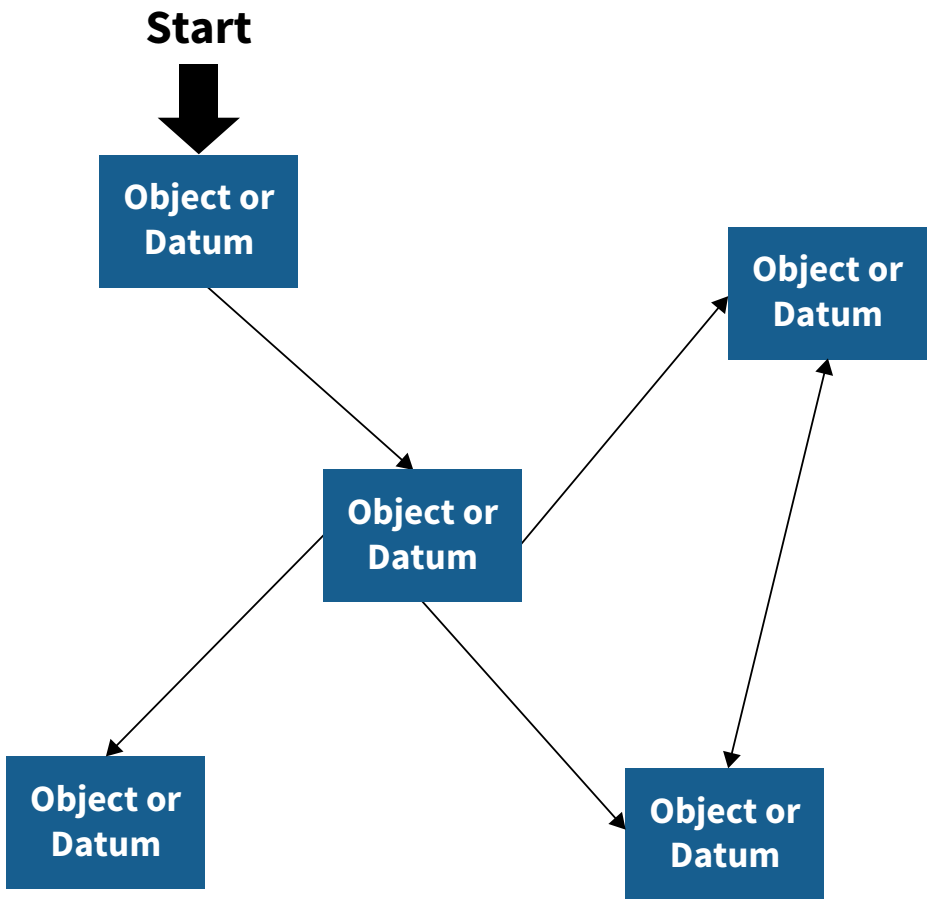
Graph

A collection of elements where each element (node) holds a reference (edge) to one or more other elements, allowing the structure to be traversed. The container holds a pointer to one or more starting nodes where the structure can be entered. Edges may be directional, weighted to indicate distance, or be objects instead of just pointers. Unlike a linked list, there is no sequential ordering of the elements. Graph nodes are sometimes placed into a linear data structure to facilitate searching.

Typical behavior (assuming no linear data structure):

Add an element	$\Theta(1)$	
Remove an element	$\Theta(E)$	
Add an edge	$\Theta(1)$	
Remove edge	$O(n)$	$\Omega(1)$
Memory footprint	$\Theta(n+E)$	

E signifies the number of edges. Finding a specific element efficiently also requires finding the shortest path to it and varies on whether the edges are directional or weighted.



Graph data structures (continued)

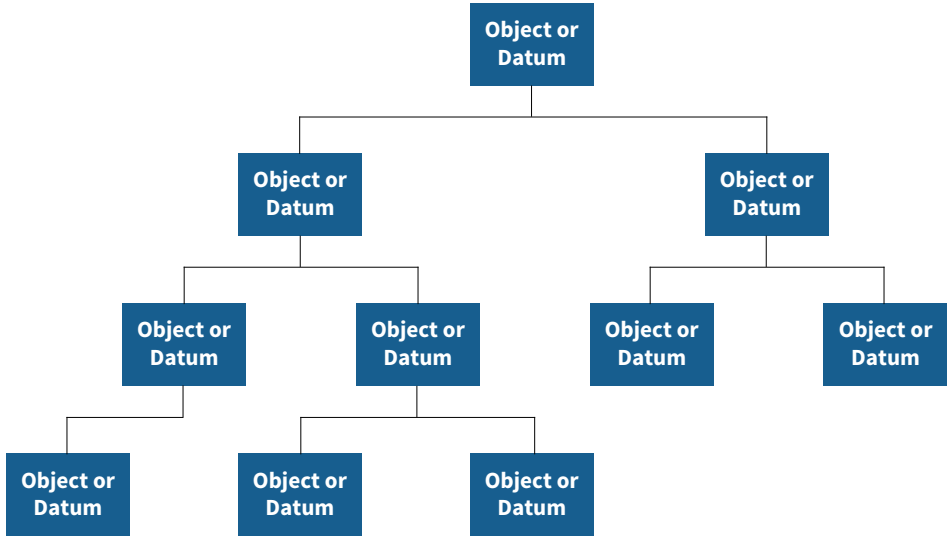
(Decision) tree

A graph where the nodes (leaves) branch out and away from the starting leaf such that every leaf is connected to the top leaf but can only be reached from it through one path. The number of "child" leaves a "parent" can link to is called the branching factor. There is no limit but two (binary trees) are the most common.

Typical behavior (assuming tree is unsorted):

Find a leaf	$O(n)$	$\Omega(1)$
Go to a lowest-level leaf	$O(n)$	$\Omega(1)$
Add a leaf	$O(n)$	$\Omega(1)$
Remove a leaf	$O(n)$	$\Omega(1)$
Memory footprint	$\Theta(n)$	

If only searching the tree up to a certain distance from the top and no further, the search complexity is $O(b^d)$ not $O(n)$, where b is the branching factor and d is the number of levels searched.



Graph data structures (continued)

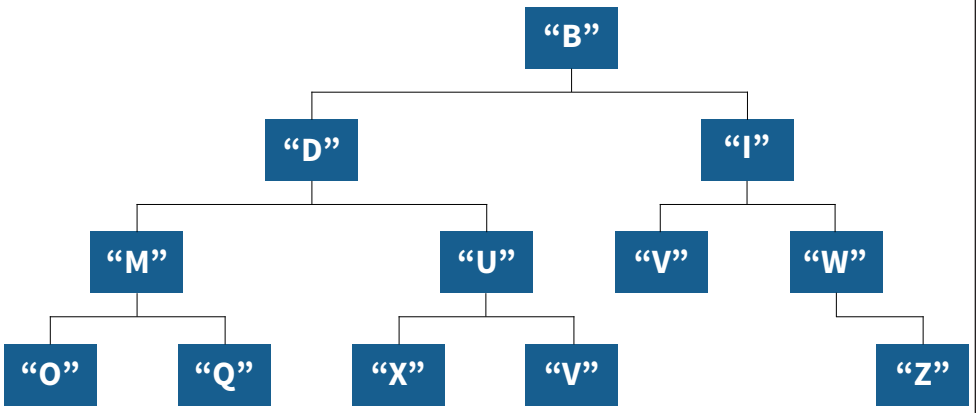
Heap

A heap is a special kind of sorted tree where the parent leaves are sorted higher than the child leaves. Heaps are one of the best ways to implement a priority queue and are sometimes used as a sorting algorithm for arrays.

Typical behavior (binary heap):

Retrieve the top leaf	$\Theta(1)$	
Remove the top leaf	$\Theta(\log n)$	
Insert a leaf	$O(\log n)$	$\Omega(1)$
Combine two heaps	$\Theta(n)$	
Heapsort algorithm	$O(n \log n)$	$\Omega(1)$
Memory footprint	$\Theta(n)$	

Heapsort is $\Theta(n \log n)$ if leaf values must be unique.



Type: Binary Heap of characters where P<C

Test your knowledge

You are the data privacy officer for Legend, a medical database company. You've been asked to help review some change requests by management. See if you can select the best data structure from the list above for each of these business requirements:

1. The hospital billing system should prioritize newly due bills first in its job queue since they have the highest likelihood of being paid. The system should only recommend older bills when there are no new bills available.

2. Many customers have recently asked Legend to design a special system their personnel can use to quickly bring up a patient's medical records in an emergency using just their hospital ID number. Management considers speed of retrieval to be the most important concern.

3. A route optimizer for physicians that suggests the shortest route through the hospital that will allow them to visit each patient in their care.

4. A new patient sign-in system for urgent care clinics. It should prioritize patients by order of arrival but bump critically ill patients to the front of the line.

5. A management system for floor nurses that will allow them to quickly pull up the vitals for the patients in each room in their area of responsibility.

As part of Legend's push toward privacy by design, you are required to review major technical designs and evaluate if the proposed design presents any privacy or compliance concerns. Do the following proposals present or facilitate infringements of protected health information? If so, how?

1. A priority queue of patient records as part of a workflow designed to help determine which person on the transplant list is best to receive an available organ.

2. A graph of a patient's family members and their medical records. Designed to facilitate access to the records of different patients suffering from the same hereditary disease for physician comparison.

3. A publicly available API health app developer can use to hook in a patient's medical records. The API works by exposing several public interfaces but no full classes. The only other public methods belong to querying objects and use the interfaces to pass requests to other objects that query the database.

4. A universal protected access modifier will be used on the above-mentioned querying objects.

5. A crucial container, which is used to run database access functions, allows the insertion of user-created objects alongside Legend-created objects as long as the objects use an interface that allows them to pass the queries correctly.

Answers to Part 1

1. The proper data structure for this task is a stack. Thanks to its last-in-first-out design, it will prioritize the most recent bills first.

2. The best data structure for this task on paper is an associative array of points to each patient's records, with hospital ID numbers acting as keys. However, because hashing functions can sometimes take a long time to process, something like a decision tree based upon the patient's ID number (assuming the numbers follow a discernable system) may be better. It will all depend upon the speed of the hashing function. Size may also be a concern if the patient population is large.

3. The best data structure for this is a graph, specifically a graph with weighted edges where the weights signify travel distance or time and the nodes each represent a patient. This task is called the travelling salesman problem and designing an algorithm to solve it efficiently is one of the hardest problems in theoretical computer science. Trying all possible routes is one of the few computing tasks that is , which is considered unworkably complex for anything more than a very small number of nodes. The best proven approach is the Held-Karp algorithm, which is slightly better at , but still too complex. Currently workable solutions only evaluate a subset of routes chosen by estimation rules.

4. The best data structure for this is a priority queue with two priorities, non-critical and critical. That way, both critically ill and non-critical patients will be prioritized in the order they came in, but the critically ill patients will be called before the non-critical patients.

5. The best data structure for this is an array, with the index of each patient's vitals corresponding to their room number or another easily-convertible datum based on the patient's room.

Answers to Part 2

1. No, although presence and location within a queue does convey protected health information, such as the seriousness of a patient's case and their time on the list, remember that the queue data structure is designed to feed information in a specific order. Therefore, it is conceivable that anyone with access to the evaluation system will be authorized to view every patient record on it, so there is no risk of a PHI breach.

2. Yes, this structure allows anyone with access to one patient's records (i.e. one node) to see connections of other patient records, and thus discover the identities of other patients with the same disease, even if they can't directly access the records. Remember from chapter one: nullity or lack of nullity also conveys information. This might be solved by anonymizing the identities of the other patients in the display of the structure by only referring to them by their relationship (for example, "parent" or "sibling"). This isn't an ideal solution, but it is better.

3. No, this structure only allows users to use the interfaces in their own classes so that they can properly communicate with the API. Access to the actual database isn't possible, that is handled in other objects which presumably have numerous security checks.

4. Yes, the protected modifier will allow those objects to be inherited, which may give a malicious programmer access to the database. This is a data breach waiting to happen.

5. Yes, most likely. Whether or not this is a true security problem will depend upon the protections within the container and how it is used by the other objects. Generally, it is good practice to process user-created objects separately from internally-created objects, and to heavily circumscribe the processing of user-created objects compared to internally-created objects, since internal object require little security safeguards while external objects require a great deal.