CHAPTER 3

Algorithm and Data Structure

Now that we've discussed computer hardware and how to program it to achieve desired purpose, we will discuss how to make programs efficient by leveraging well-known algorithms and data structures for managing logic and data, respectively.

What Is an Algorithm

The dictionary defines an algorithm as a step-by-step procedure for solving a problem or accomplishing some end. In other words, an algorithm is a technique that can be used and communicated to accomplish your goal. Algorithms are not unique to computers. You probably use algorithms every day. The mathematical technique of carrying the one or borrowing from the tens place for addition and subtraction is an algorithm that humans can learn. There is usually more than one algorithm to accomplish your goal. For instance, one algorithm for division is to count the number of times you subtract the divisor from the dividend; this count is the quotient. This is different than finding the largest number the divisor can be multiplied by to be less than the most significant bits of the dividend and then subtracting that value from the dividend to get a new dividend, which is the method most of us learned in school.

© Paul D. Crutcher, Neeraj Kumar Singh, and Peter Tiegs 2021
P. D. Crutcher et al., *Essential Computer Science*,
https://doi.org/10.1007/978-1-4842-7107-0_3

Algorithms can be encoded in any programming language for computers. It should be noted that algorithms for humans are not necessarily optimal for computers to accomplish the same end. This is also true for different computing architectures; an algorithm for a general-purpose CPU will not be the best algorithm for a GPU (Graphics Processing Unit), or quantum computer. In the next section, we will examine how to evaluate algorithms and what trade-offs are made to find the right algorithm for what you need to accomplish.

Good and *Not So Good* Algorithm

Knowing that there are likely multiple algorithms for accomplishing what you want to do, how do we judge what is a good algorithm? What are the factors that we look at? Can we use math to compare algorithms?

One thing that should not be overlooked, but is hard to compare critically, is the readability of a particular algorithm. Most of the software that you write professionally will be viewed and likely maintained by others. Choosing an algorithm that can be read, and more easily maintained, to learn what goal you originally set out to accomplish can be a better choice than the most efficient algorithm. Choosing well-known algorithms can help readability, because there can be plenty of documentation about those algorithms and they can be recognized. Of course, explicitly stating the goal you are seeking to accomplish in source code comments can help.

Time/Space Complexity

One of the main areas where we make trade-offs when selecting or creating algorithms is between the amount of memory, or space, that the algorithm takes and the amount of time it takes to finish.

Asymptotic Notation

Asymptotic notation is a method of writing the complexity of an algorithm in time based on the number of inputs into the algorithm. We cannot simply say that because algorithm 1 will take 7 seconds and algorithm 2 will take 5 seconds, algorithm 2 is better. Asymptotic notation helps by eliminating differences in individual computing machines and programming languages. Taking a closer look at those times, we need to specify the number "n" for the number of items that the algorithm will process to have a realistic measure of its performance to compare against other implementations. For simplicity, let us say n = 100. For algorithm 1, let us say the time it takes to run is 3 + .04n; similarly algorithm 2 takes 0.0005(n^2) seconds to run. As we can see in the graph (Figure 3-1), there is a crossover point in the number of items where algorithm 1 outperforms algorithm 2.

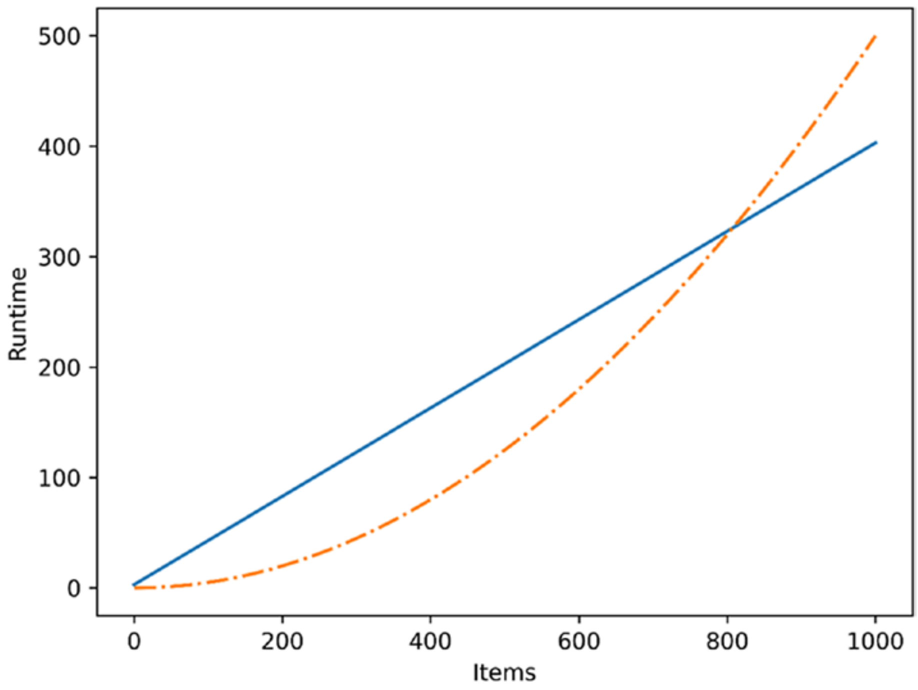


Figure 3-1. *Runtime Comparison Example*

These numbers are on the same computer. If we do analysis on an older computer, we find that algorithm 1 takes 5 + 0.4n or 45 seconds and algorithm 2 takes .005(n^2) or 50 seconds. We will simplify our algorithm by removing constants from the time to allow for differences in computing machines and programming languages. This is called Big-Oh notation as the function for the time an algorithm runs asymptotically approaches the highest degree of the polynomial of n. We will write analysis O(n) for algorithm 1 and O(n^2) for algorithm 2.

With Big-Oh expressions, we generally want to consider "tightness" of the upper bound. While it is correct to say that an algorithm with a time function 3 + 0.4n is O(n), it is a stronger statement to simply say that this algorithm is O(n).

Big-Oh notation is a consistent method for comparing and discussing algorithms across multiple computing machines and programing languages. Table 3-1 is a table of Big-Oh expressions and their informal names.

Table 3-1. *Big-Oh Common Names*

Big-Oh	Name
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	n log n
O(n^2)	Quadratic
O(n^3)	Cubic
O(2^n)	Exponential

Fundamental Data Structures and Algorithms

Now that we have examined what an algorithm is and how we can compare them, we will look at common data structures that hold our data. We will also look at common algorithmic techniques using these data structures.

Store (Data Structure)

There are several structures that can store data. Each of these structures has different advantages, and algorithms may be able to utilize different data structures more efficiently than others.

Stack

A stack is a data structure that reflects the metaphor of a stack of plates. When using a stack, an algorithm operates only on the "top" item in the stack. When that item is operated on, it is removed or "popped" off the stack. A data item may also be "pushed" onto a stack. Because data is only operated on or removed from the "top" of the stack, a stack is sometimes referred to as a FILO (First In, Last Out) or LIFO (Last In, First Out). See Figure 3-2.

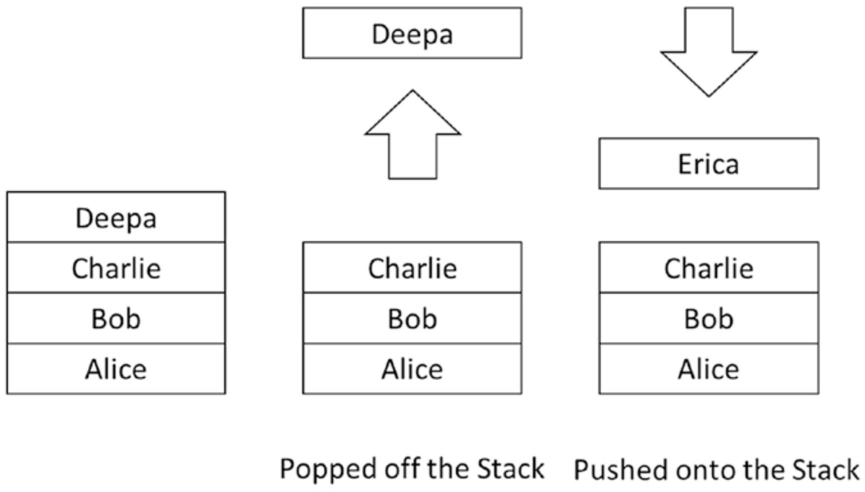


Figure 3-2. *Stack Example*

Queue

A queue is another data structure. As you can imagine, a queue also acts like a line to an event. Data items in a queue are added at the "back" of the queue and processed at the "front" of the queue. Queues can vary in length, allowing them to be used as a buffer. Queues are also referred to as FIFOs (First In, First Out). See Figure 3-3.

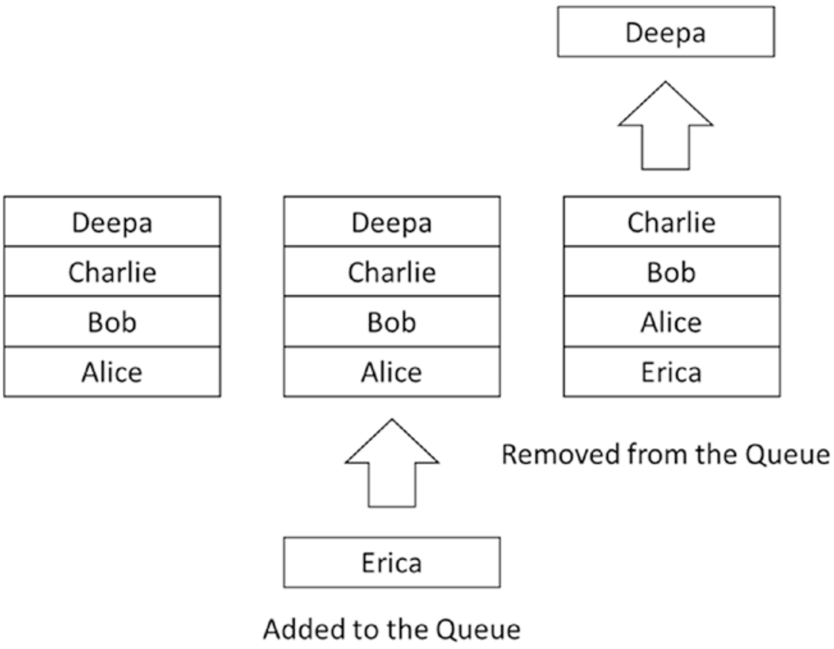


Figure 3-3. *Queue Example*

Tree

A tree is another data structure that allows for multiple branches. Data items or nodes are attached to the trunk, which has one or more items attached to it as branches. Each branch can have one or more branches attached to it. Nodes without branches attached to them are referred to as leaf nodes, or simply leaves. See Figure 3-4.

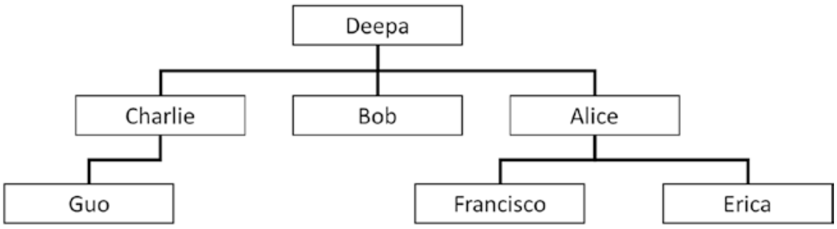


Figure 3-4. *Tree Example*

Graph

A graph is a data structure where nodes or data items are connected via edges to other nodes. The edges may contain data about the relationship to the nodes. A directed graph is a graph data structure where all the edges have a common direction. A tree can be thought of as a directed graph. See Figure 3-5.

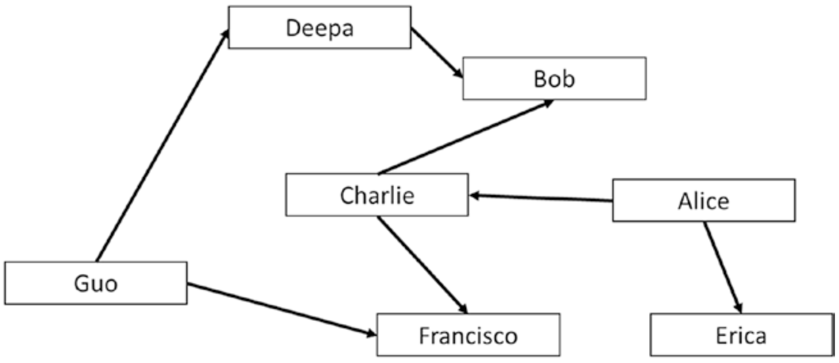


Figure 3-5. *Graph Example*

Linked List

A linked list is another data structure where each node or data item is linked to (connected with one or two) other data items in a chain. A doubly linked list is a list where each node contains a link to both the next node and the previous node. Data items can be inserted into a linked list by connecting to the new data item. Some of the other data structures such as the queue and the stack can be implemented as linked lists. See Figure 3-6.

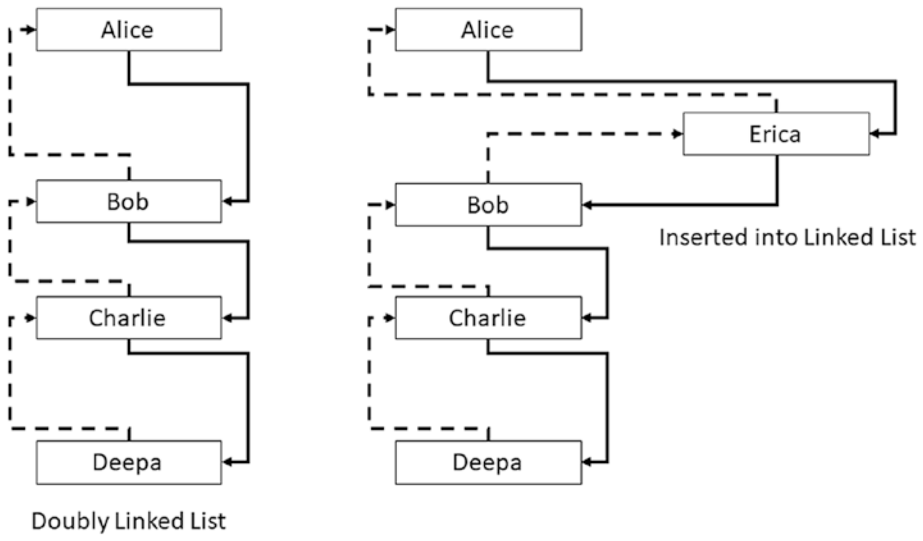
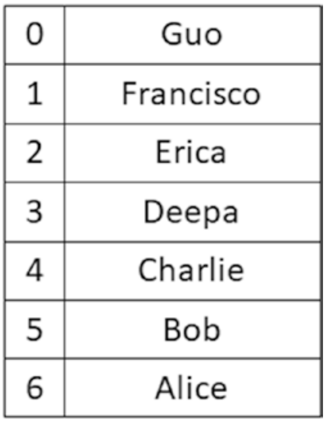


Figure 3-6. *Doubly Linked List Example*

Array

An array is a fixed-size set of data, where each data node is referred to by a coordinate system. In a single-dimensional array, this value is called the index and typically starts at zero for the first node. In a two-dimensional array, or grid, a node has two coordinates like x and y; and in a three-dimensional array, like a cube, it has three, like x, y, and z. Arrays can have more dimensions than three if needed. Data in an array can be accessed from any position in the array at any time. A sparse array is an array that does not have meaningful data in every position. See Figure 3-7.

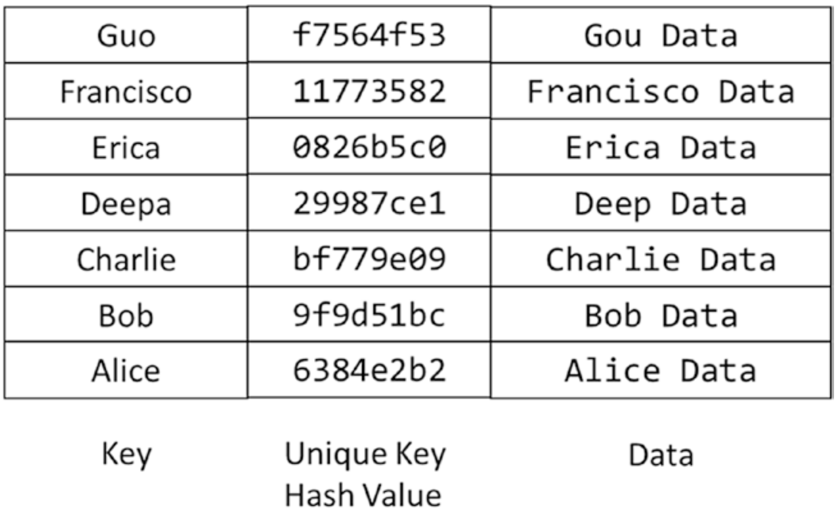


0	Guo
1	Francisco
2	Erica
3	Deepa
4	Charlie
5	Bob
6	Alice

Figure 3-7. *Array Example*

Dictionary

One more data structure is a dictionary, sometimes referred to as a hash table. Similar to an array, in a dictionary, the data nodes are referred to by a key or index. Unlike an array, this index is not integer values. Instead, a hashing algorithm is run to find a unique value for each data node, and that is used as the key to look up the data node. Like an array, data can be accessed from any node in the hash table at any time. See Figure 3-8.



Guo	f7564f53	Gou Data
Francisco	11773582	Francisco Data
Erica	0826b5c0	Erica Data
Deepa	29987ce1	Deep Data
Charlie	bf779e09	Charlie Data
Bob	9f9d51bc	Bob Data
Alice	6384e2b2	Alice Data
Key	Unique Key Hash Value	Data

Figure 3-8. *Dictionary Example*

Making Use of the Data: Searching, Sorting

Two of the most common things to do with the data in these data structures are to search through the data for a specific item and to sort the data in some fashion. There are different sorting and searching algorithms that can be used on the data. Sorting is often done as part of searching as it can be easier to find an item with the data structure sorted. Depending on the type of data structure, different algorithms will perform better or worse.

The first sorting algorithm that we will look at is the bubble sort (Listing 3-1). In this algorithm, the items are sorted into order with the priority items “bubbling” to the top of the data structure. If we have a linked list, call it `I`, we will start with the first item (`i[0]`) in the list and compare it to the next item (`i[1]`). We then compare `i[0]` and `i[1]`; if `i[1]` is before `i[0]`, then we swap `i[0]` with `i[1]`. Then we proceed to compare the new `i[1]` with `i[2]`; if `i[2]` needs to swap with `i[1]`, then we swap. If the items are in the right order, we do not swap but proceed to the next item to compare.

Listing 3-1. Bubble Sort Algorithm in Python

```
def bubble(NUMBER_LIST):
    print(NUMBER_LIST)#Display the unsorted list
    swap_counter = 0 #Set a counter for the number of swaps
    for idx in range(0, len(NUMBER_LIST)):#Loop through list
        pos = idx #Set the item to compare
        swap_pos = pos - 1 #Set the item to swap if needed
        #Loop through the items to compare
        while swap_pos >= 0: #Loop through the unsorted list
            #Check to see if you need to swap
            if NUMBER_LIST[swap_pos] > NUMBER_LIST[pos]:
                #Swap positions
                NUMBER_LIST[pos], NUMBER_LIST[swap_pos] =
                NUMBER_LIST[swap_pos], NUMBER_LIST[pos]
                #Increment the swap counter to show the work
                swap_counter = swap_counter +1
            print(NUMBER_LIST) # Display the current list
            #Move to the next swap item
            swap_pos = swap_pos -1
            #Move to the next item to compare
            pos = pos -1

    #Display the number of swaps
    print("SWAPS:", swap_counter)
```

Python console output

```
>>> bubble.bubble([90,87,82,43,3,5])
[90, 87, 82, 43, 3, 5]
[87, 90, 82, 43, 3, 5]
[87, 82, 90, 43, 3, 5]
[82, 87, 90, 43, 3, 5]
```

```
[82, 87, 43, 90, 3, 5]
[82, 43, 87, 90, 3, 5]
[43, 82, 87, 90, 3, 5]
[43, 82, 87, 3, 90, 5]
[43, 82, 3, 87, 90, 5]
[43, 3, 82, 87, 90, 5]
[3, 43, 82, 87, 90, 5]
[3, 43, 82, 87, 5, 90]
[3, 43, 82, 5, 87, 90]
[3, 43, 5, 82, 87, 90]
[3, 5, 43, 82, 87, 90]
[3, 5, 43, 82, 87, 90]
SWAPS: 14
```

If we do a Big-Oh analysis of this, then we can see this is O(n^2), with the worst case being having to compare every element with every other element.

Selection sort is the next sorting algorithm we will look at (Listing 3-2). In this algorithm, we will compare the first item to the rest of the items and select the smallest item and swap those items. We then proceed with the next item and select the next smallest item and swap them. We proceed until we have iterated through each item in the array. Selection sort is also O(n^2).

Listing 3-2. Selection Sort Algorithm in Python

```
def selection(number_list):
 print(number_list)#Display the unsorted list
 iter_count = 0 #set a counter for the iterations
  #Loop through the each item on the list
 for i in range(0, len(number_list)):
   min_index = i #Set the current min value in the list
   #Loop through the remaining unsorted list
   for j in range(i+1, len(number_list)):
```

```
            #Compare the current item with the current minimum
            if number_list[j] < number_list[min_index]:
                #If the current item is smaller
                #make it the new minimum
                min_index = j
            #Swap the new minimum with the
            #current value in the list
            number_list[i], number_list[min_index] =
            number_list[min_index], number_list[i]
            #Increment the count of swaps
            iter_count = iter_count +1
        print(number_list): #Display the current list
    #Display the number of iterations
    print("Iterations: ", iter_count)
```

Python console output

```
>>> selection.selection([90, 87, 82, 43, 3, 5])
[90, 87, 82, 43, 3, 5]
[5, 90, 87, 82, 43, 3]
[5, 3, 90, 87, 82, 43]
[5, 3, 43, 90, 87, 82]
[5, 3, 43, 82, 90, 87]
[5, 3, 43, 82, 87, 90]
[5, 3, 43, 82, 87, 90]
Iterations:  15
```

Problem Solving Techniques

We have examined how we analyze and compare algorithms. And we have looked at how we can structure our data. Now we will look at common techniques for solving problems.

Recursion

A recursive algorithm is an algorithm where the function calls itself. Recursive functions, or methods, can be very efficient and easy to understand. The following is an example of a very simple recursive algorithm (Listing 3-3) to calculate the Fibonacci sequence. In the Fibonacci sequence, the current value is defined as the sum of the previous two values F(N) = F(N – 1) + F(N – 2). Also the first two values F(1) and F(0) are predefined to 1 and 0, respectively. For example, to calculate the value of F(3), we need to first calculate the F(2) and F(1). To calculate F(2), we need to calculate F(1) and F(0).

F(1) is 1 and F(0) is 0 so that makes F(2) = 1 + 0 or 1. To finish calculating F(3), we add F(2) + F(1) or 1 + 1. Therefore, F(3) is 2.

Listing 3-3. Recursive Fibonacci Algorithm

```
def fibonacci(value):
    if value == 0:#Set F(0) to 0
        retval = value
    elif value == 1:#Set F(1) to 1
        retval = value
    else: #Otherwise calculate the value of F(N)
        #Recursively call the fibonacci function on the
        #previous value. Then call fibonacci function on the
        #value before that.
        #Set the current value to the sum of those two values
        retval = fibonacci(value-1) + fibonacci(value-2)
    return retval

def fibonacci_list(max):
    for i in range(0, max):
       #Display the current Fibonacci value
       print(fibonacci(i))
```

Python console output

```
>>> fibonacci.fibonacci_list(5)
0
1
1
2
3
```

Divide and Conquer

Divide and conquer is a technique where the data is divided and each smaller portion is operated on.

The merge sort algorithm (Listing 3-4) is a good example of both recursion and divide and conquer algorithms. The basic part of the merge sort algorithm splits a list into two separate equal halves. Those halves are then sorted. Once you have two sorted halves, you simply compare the first items in each list and add the smaller to the next position in a new list. To get each half sorted, you can call the merge sort algorithm on each half.

Listing 3-4. Merge Sort Divide and Conquer Algorithm in Python

```
def merge(number_list):
  #Check if the list is longer than one element
  if len(number_list) > 1:
        #Find the middle of the list
        half_idx = int(len(number_list)/2)
        #Create a list with front half of the list
        list_a = number_list[:half_idx]
        #Create a list with the back half of the list
        list_b = number_list[half_idx:]
        #Recursively call this merge function
        #to sort the first half
        sorted_a = merge(list_a)
        #Recursively call this merge function
        #to sort the second half
        sorted_b = merge(list_b)
        #Init an empty list to insert the sorted values
        sorted_list = []
        #Set a flag to indicate both lists are inserted
        done = False
        while not done: #Iterate on the lists until done
            #Compare the first item of each list
            if sorted_a[0] < sorted_b[0]:
                #When the first list item is smaller
                # insert into the sorted list
                sorted_list.append(sorted_a.pop(0))
            else:
                #When the second list item is smaller
                # insert into the sorted list
                sorted_list.append(sorted_b.pop(0))
            if len(sorted_a) == 0:
                #When the first list is empty add the
                # remainder of the second list to the
                # sorted list
                sorted_list = sorted_list + sorted_b
                #Set the done flag to end the loop
                done = True
            elif len(sorted_b) == 0:
                #When the first list is empty add the
                # remainder of the second list to the
                # sorted list
                sorted_list = sorted_list + sorted_a
                #Set the done flag to end the loop
```

```
                  done = True
          print(sorted_list)
      else:# If the list is only one element it is sorted
          sorted_list = number_list


     return(sorted_list)
```

Python console output

```
>>> merge.merge([90, 87, 82,43,3,5])
[82, 87]
[82, 87, 90]
[3, 5]
[3, 5, 43]
[3, 5, 43, 82, 87, 90]
[3, 5, 43, 82, 87, 90]
```

Brute Force

A brute force algorithm is just as it sounds, doing the most obvious thing with the data operating on each data item individually. In some situations, especially with smaller data sets, this can be the quickest way to solve the problems, but in general, this is a costly way O()) to perform a function.

Greedy Algorithms

A greedy algorithm is an algorithm that makes a locally optimal decision. This can, in some cases, lead to locally optimized implementations vs. the best globally optimized solution. Greedy algorithms include the Huffman coding algorithm for data compression and the Dijkstra algorithm for search in a tree.

Class of Problems

Many algorithms can be solved in polynomial time where the Big-Oh expression can be written as a polynomial. These are considered tractable problems. There is also the set of problems that cannot be solved in polynomial time. These are considered intractable. However, within the set of intractable problems are a set of problems that can verify possible answers in polynomial time. These are referred to as nondeterministic polynomial, or NP, problems. Finding a prime number is an example of this type of problem.

NP-Complete and NP-Hard Problems

Within the set of NP problems are the set of problems no one knows how to solve in less than exponential time known as NP-complete.

One common example of an NP-complete problem is the traveling salesman problem, where we want to find the shortest path for a salesman to navigate a set of cities connected by routes of different lengths. Checking the length of a route and comparing it to other routes is polynomial, but finding the shortest route requires going through all possible combinations.

In addition to NP problems are another set of problems that are defined as NP-hard. These problems are as hard as or harder than any NP problems. This set of problems are called NP-hard problems. If these problems are found to be solvable in polynomial time, that would imply that all NP problems are actually solvable in polynomial time. This is not believed to be the case.

Databases

So far in this chapter, we have looked at data structures and algorithms that have been operating on data in system memory (e.g., RAM). Now we will look at database systems that can persistently store and recover the data. A database is simply an organized set of data that is stored apart from the program that will utilize that data.

Persistence and Volume

We separate data out from the software into a database for various reasons. One reason is the persistence of data. If you have software that doesn't, somehow, "save" its resulting data, that data would not be available after the software is run, as it was only in system memory, which will be reused by other programs once your program is done. This storage, or persistence, of data also provides some other advantages. It allows multiple different software applications to access the same data. Many database systems allow for multiple applications to access the data concurrently.

The other reason to store the data separate from the software is that it allows the software to operate on much larger volumes of data than can be contained in the RAM. A database system can provide parts of the data to the software at a time so that software can work on this smaller sets of data.

Fundamental Requirements: ACID

As the volume of data gets larger, and there is more concurrent access (from multiple concurrently running applications) to the data, a database must make sure that it meets the requirements of ACID (Atomicity, Consistency, Isolation, and Durability).

Atomicity means that an update happens to the database as a single, atomic event, so there are no partial updates. Say, for instance, I have a simple database of a name, street address, and zip code. And I need to update a record because someone moved to a new city. A nonatomic update might be to update the zip code without updating the street address, followed by an update of the street address. This would lead to a point in time where the data in the database is incorrect (only partially updated). In contrast, an atomic update, or commit, would update the record with both the new street address and zip code at the same time, so the database is never incorrect.

Consistency means that in the event of a failure, for instance, an update failure, the database stays consistent with a known good state; this is usually the previous state of the database. For example, in our previous example, we may want to update all the names to make sure they are capitalized. If there is a failure after the third record is updated, then the transaction will roll back to the previous state, where none of the names are capitalized.

Isolation means that if there are multiple concurrent updates to the database, each transaction must not be intermixed with any other transaction. The two previous examples for updating one record (a person moved) and updating all the records to make sure that names are capitalized must be isolated. In this case, all the names get updated first, and then the one record is updated with a new street address and zip code. This is important for data consistency and durability. If we needed to roll back a transaction and both sets of changes were intermixed, we would not be able to clearly go back to a known good state.

Durability is like consistency; it means that in the event of a failure of the underlying database system, when the database system restarts, it is able to pick up where it left off and complete the transaction. For example, in the previous example, say that after the third record gets updated, the operating system forces a reboot. When the operating system comes back up, the database system must complete the transaction starting at exactly the fourth record.

Brief History of Database System Evolution

In 1970 Edgar F. Codd wrote a paper describing relational database systems. Prior to the publication of Codd's paper, companies had started to develop database systems based on other models, but by the late 1970s, the relational database model had become prevalent. IBM produced the first prototype relational database with SQL in 1976. The Oracle Database was the first commercial database that implemented the model and featured SQL, the Structured Query Language. Oracle was released in 1977, prior to IBM's release of SQL/DS in 1981, despite IBM having a head start. Also, in 1981, dBase II, considered the first relational database for PCs, was released for personal computers. Oracle became the primary database used in the enterprise as well as the Internet until the release of the open source database MySQL in 1995. On the PC side, many solutions were released over the next decade with Microsoft Access becoming the de facto standard relational database on the PC in 1993.

Most Prominent Current Database Systems

Today, Oracle remains one of the most prominent relational database systems. In addition, the open source community has brought several solutions to prominent usage. MySQL still is in use but is joined by PostgreSQL and SQLite as to the very common open source relational database solutions. On the commercial side, Microsoft SQL Server has also risen to prominence in its usages.

Relational Data and SQL

Relational data is based on set theory and the relationships between sets. Sets can be combined in a union. This means a new set is formed that contains all the data elements that are in the sets combined. A new set, for instance, may be formed from the differences of sets; this would be a set of all of the data elements that are unique between the sets. Furthermore, another set can be formed from the intersection of two sets. This is where a new set is formed from all the elements that are common between the two sets. See Figure 3-9.

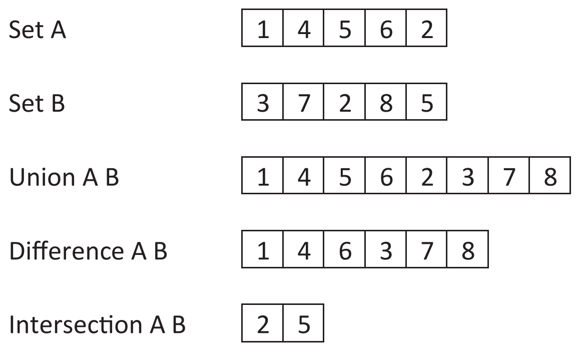


Figure 3-9. *Set Operations Example*

SQL is a standard language to describe these relationships between sets of data to extract meaningful data from a relational database. For example (Figure 3-10), a SQL statement `SELECT (id, name, zipcode) FROM people_table WHERE (zipcode IS '97124')` forms a set containing the value 97124 and then intersects that data with the set of zip codes in the table. This new intersected set of records will have the same set of fields as the original table but only contain the values for those that match the zip code 97124.

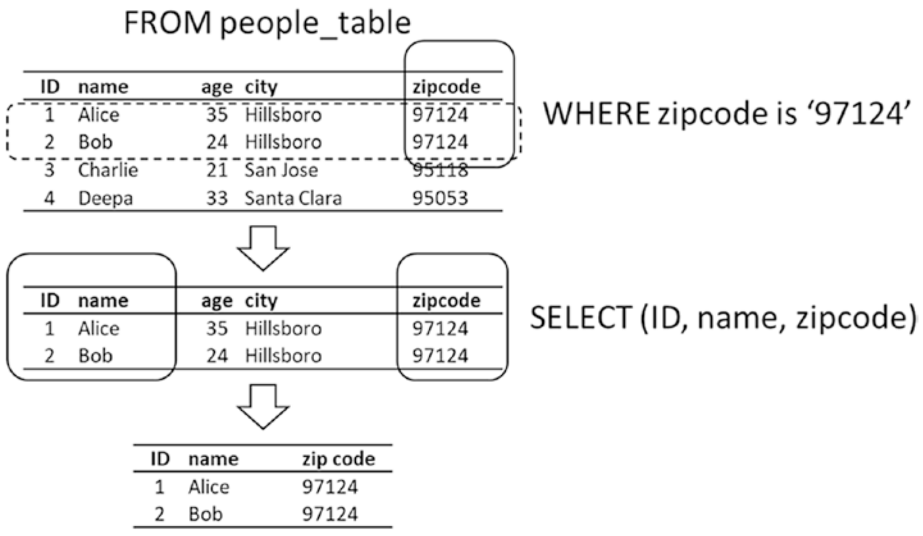


Figure 3-10. *SQL Statement Actions*

SQL syntax allows for a rich group of set relationships described in a machine-translatable language that approximates natural language.

Structured Data/Unstructured Data

Relational databases mostly have structured data, data that is organized into rows and columns. This structured organization makes it easy to interact with the data using SQL and the set relations. The definition of this structure is called a schema. As you can imagine, however, much of the data that we have in the world is not so easily structured. Unstructured data is data that cannot easily be organized into rows and columns, such as natural language text. This rise in unstructured data has also led to an increase in databases that do not follow the same constraints of relational databases.

NoSQL

NoSQL or Not Only SQL is a collective name of a growing set of databases that apply different data structures besides tables of rows and columns used in relational databases.

With the rise of the Internet and service-oriented architectures, one of the key points of integrating the data from multiple applications shifted from the relational database and SQL to service access. This allowed developers to create a closer mapping of the data structures used in the application to the data stored in the database. Now developers could have a much more natural connection between the data that is being stored and the data that is being used.

Examples of NoSQL Databases

We will look at some common examples of NoSQL databases.

Graph DB: Neo4j

Neo4j is a native graph database where the data is stored and its relationship to other data is also stored. A record is stored as a node in a graph data structure, and additional relationship records are stored with information about how various nodes are related (connected) to each other.

Neo4j can be schema-less with nodes having different fields as needed. Neo4j also has its own query language called Cypher.

Column Family DB: Bigtable and Cassandra

Bigtable is a proprietary wide-column family database from Google. Bigtable is designed to specifically handle exceptionally large sets of data.

Like Bigtable, Cassandra is an open source column family database from Apache. A column family database organizes the data into rows and columns. A column is the primary data entity. A column is made up of a name and a value with the name acting as a key in a key-value pair. A row is an arbitrary group of columns with a row key. A column family is a group of rows with some column keys in common. Cassandra is a schema-free database in that rows do not have to have the same columns. Cassandra also has its own query language CQL.

Document DB: CouchDB and MongoDB

CouchDB is a document database from an open source project that is part of the Apache group. Each piece of data is considered a document with its own set of fields.

MongoDB is another open source project that is a document database. It stores records as JSON (JavaScript Object Notation) documents. Each document can have its own set of attributes so it can be schema-free. Both CouchDB and MongoDB have their own mechanisms for querying the data.

Summary

As we have seen throughout this chapter, there are many considerations when working with data. The selection algorithm, data structures, and database for persistent storage should be chosen thoughtfully so that the software can be developed in the most effective way.

References and Further Reading

- Thomas Cormen. *Introduction to Algorithms, Third Edition*. MIT Press, 2009
- Avi Silberschatz. *Database System Concepts*. McGraw-Hill Education, 2010

- Alfred V. Aho and Jeffery D. Ullman. *Foundations of Computer Science*. Computer Science Press, 1992
- Mukesh Negi. *Fundamentals of Database Management System*. BPB Publications, 2019
- Pramod Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 2013