CS Foundation Study guide

By Alex

10/17/23

Contents

Data Structure & Algorithms

1. **Arrays:**
- ***Explanation:*** Arrays are ordered collections of elements, typically with fixed size, where each element is accessed using an index.
- ***Use Case:*** Arrays are commonly used for storing and accessing data in a linear or sequential manner. For example, you might use an array to store a list of integers, such as an array of grades for a class.

2. **Linked List:**
- ***Explanation:*** A linked list is a linear data structure consisting of nodes, where each node holds data and a reference to the next node.
- ***Use Case:*** Linked lists are useful when you need dynamic sizing or efficient insertions and deletions. For instance, they can be used to implement a music playlist where songs can be added or removed easily.

3. **Stacks:**
- ***Explanation:*** Stacks are linear data structures following the Last-In-First-Out (LIFO) principle, where the last element added is the first one to be removed.
- ***Use Case:*** Stacks are employed in tasks like function call management in programming, where you need to keep track of function calls and their return addresses.

4. **Queues:**
- ***Explanation:*** Queues are linear data structures following the First-In-First-Out (FIFO) principle, where the first element added is the first one to be removed.
- ***Use Case:*** Queues are commonly used for tasks like managing task scheduling in an operating system, where processes are executed in the order they arrive.

5. ***Trees:***
- ***Explanation:*** Trees are hierarchical data structures consisting of nodes with parent-child relationships, used for organizing and efficiently searching data.
- ***Use Case:*** Trees are widely used in data structures like Binary Search Trees (BSTs) to store and retrieve data efficiently, such as dictionary or phonebook applications.

6. **Hash Tables:**
- ***Explanation:*** Hash tables are data structures that use a hash function to map keys to specific locations, enabling fast retrieval and insertion of data.
- ***Use Case:*** Hash tables are employed in implementing dictionaries, databases, and caches to ensure fast data access, like searching for words in a dictionary.

7. **Graphs:**
- ***Explanation:*** Graphs are structures with nodes (vertices) connected by edges, used for modeling relationships and connections between entities.
- ***Use Case:*** Graphs are essential in applications like social networks, where users and their connections can be represented, or in routing algorithms for finding the shortest path between locations on a map.

8. **Heaps:**
- ***Explanation:*** Heaps are specialized tree structures that ensure the highest (or lowest) priority element is always at the root.
- ***Use Case:*** Heaps are employed in priority queues, such as in job scheduling, where tasks with higher priority should be executed first.

9. **Sets:**
- ***Explanation:*** Sets are collections of distinct elements, meaning no duplicate values are allowed.
- ***Use Case:*** Sets can be used in situations where you need to maintain a unique collection of items, such as tracking unique visitor IPs in a web application.

10. **Trie:**
- ***Explanation:*** A trie is a tree-like data structure used for efficient storage and retrieval of strings.
- ***Use Case:*** Tries are commonly used in dictionary and autocomplete applications, where you want fast and efficient string searching and word completion.

1. **Sorting Algorithms:**
- ***Bubble Sort:*** Simple and intuitive, but not very efficient.
- ***Insertion Sort:*** Efficient for small data sets.
- Selection Sort: Not very efficient, but simple to implement.
- ***Merge Sort:*** Efficient and stable, commonly used in practice.
- ***Quick Sort:*** Efficient in practice, commonly used for large data sets.

2. **Searching Algorithms:**
- ***Linear Search:*** Simple and intuitive, but not very efficient for large datasets.
- ***Binary Search:*** Highly efficient for sorted arrays.

3. **Recursion:**
- A programming technique in which a function calls itself in order to solve a problem.
- ***Base Case(s):*** This case prevent the function from infinitely calling itself, which would lead to a stack overflow. Without a base case, a recursive function will keep calling itself indefinitely.
- ***Recursive Case:*** In this case, the function breaks down a problem into one or more smaller instances of the same problem, and then calls itself with these smaller instances. The results from the smaller instances are combined to solve the original problem.

4. **Graph Algorithms:**
- ***Breadth-First Search (BFS):*** Used for exploring a graph layer by layer.
- ***Depth-First Search (DFS):*** Used for exploring a graph branch by branch.
- ***Dijkstra's Algorithm:*** Finds the shortest path in weighted graphs.
- ***Bellman-Ford Algorithm:*** Finds the shortest path in weighted graphs with negative edge weights (but no negative cycles).
- ***Floyd-Warshall Algorithm:*** Finds all shortest paths in a weighted graph.

5. **Dynamic Programming:**
- ***Fibonacci Sequence:*** Understanding dynamic programming through the Fibonacci sequence is a good starting point.
- ***Knapsack Problem:*** Solving this problem illustrates the power of dynamic programming for optimization.

6. **Greedy Algorithms:**
- ***Fractional Knapsack Problem:*** Introduction to the greedy approach.
- ***Prim's Algorithm:*** Finds a minimum spanning tree in a weighted graph.
- ***Kruskal's Algorithm:*** Finds a minimum spanning tree in a weighted graph.

7. **Divide and Conquer:**
- ***Closest Pair of Points:*** Illustrates the divide-and-conquer paradigm.

8. **Bit Manipulation:**
- Understanding bitwise operations and their applications.

9. **Hashing:**
- Understanding hash functions and their applications in data retrieval.

10. **Backtracking:**
- ***N-Queens Problem:*** A classic example of backtracking.

Object-Oriented Programming (OOP)

1. **Definition:**
   - OOP, or Object-Oriented Programming, is a programming paradigm or style that organizes code based on the concept of "objects". It's a way of structuring code to model real-world entities, allowing for more organized, modular, and reusable code.
2. **Classes and Objects:**
   - ***Classes:*** Classes are blueprints or templates that define the structure and behavior of objects. They encapsulate data and methods that operate on that data.
   - ***Objects:*** Objects are instances of classes. They are tangible entities created from class definitions. Objects hold their unique data and can perform actions as defined in the class.
   - ***Use Case:*** Classes are used to define the attributes and behaviors of objects. Objects, in turn, are created from these classes and represent real-world entities in your software, allowing you to model and manipulate data in a structured manner.
3. **Encapsulation:**
   - ***Encapsulation*** is the principle of bundling the data (attributes) and methods (functions) that operate on that data into a single unit, the class. It also involves controlling access to the data by defining access modifiers (e.g., public, private, protected).
   - ***Use Case:*** Encapsulation helps in data hiding and ensures that the internal state of an object can only be modified through well-defined methods, enhancing data security and reducing the risk of unintended data manipulation.
4. **Inheritance:**
   - ***Inheritance*** is a mechanism that allows one class (subclass or derived class) to inherit the properties and behaviors of another class (superclass or base class).
   - ***Use Case:*** Inheritance enables code reuse and the creation of specialized classes that inherit common attributes and methods from a base class. It's particularly useful for modeling relationships between objects in the real world, such as "is-a" relationships.
5. **Polymorphism:**
   - ***Polymorphism*** allows objects of different classes to be treated as objects of a common superclass. It also allows a single method or function to operate on different types of objects.
   - ***Use Case:*** Polymorphism makes code more flexible and extensible. It enables you to write generic code that can work with a variety of objects, as long as they adhere to a common interface. This is especially useful for writing code that handles multiple object types without the need for extensive conditionals.
6. **Abstraction:**
   - ***Abstraction*** is the process of simplifying complex systems by modeling classes based on their essential characteristics while ignoring unnecessary details.
   - ***Use Case:*** Abstraction helps in managing complexity by focusing on relevant aspects of an object while hiding the irrelevant details. It allows for the creation of clear and concise class definitions, making it easier to design, understand, and maintain code.

1. **Example:**
   - Think of OOP as a way to write computer programs by modeling objects in the real world. Everything around us can be thought of as an object - a car, a book, a person, etc. Each of these objects has certain characteristics (like color, size, or speed) and can perform actions (like driving, reading, or walking). In OOP, we create these objects in our code to represent things in the real world.
2. **Objects and Classes:**
   - An object is like a thing or item from the real world. It has both data (attributes) and things it can do (methods).
   - A class is like a blueprint or a template for creating objects. It defines what an object of that class should look like and what it can do.
3. **Encapsulation:**
   - Encapsulation is like putting things in a box. In OOP, we hide the details of how an object works and only show the parts that are important to us. It's like driving a car without needing to know how the engine works. We just need to know how to use the pedals and the steering wheel.
4. **Inheritance:**
   - Inheritance is a way to create new objects by reusing the attributes and methods of existing objects. It's like saying, "I want a new object that's like this one but with some changes." For example, you can create a "sports car" object that inherits from the "car" object but might have some extra features.
5. **Polymorphism:**
   - Polymorphism allows different objects to be used in a similar way. It's like having a remote control that works for many different TV brands. You don't need to know the inner workings of each TV; you just use the remote to change channels or adjust the volume.
6. **Abstraction:**
   - Abstraction is like looking at the bigger picture and ignoring the tiny details. In OOP, you focus on what an object does rather than how it does it. For example, when using a smartphone, you don't need to know how every app is coded; you just use the apps for their specific purposes.

Big O notation (Time complexity)

Definition: Time complexity, often expressed using Big O notation, is a way to analyze and describe the efficiency of an algorithm or a piece of code in terms of how its runtime or resource usage (typically, but not limited to, time) scales with the size of the input. In other words, time complexity helps us understand how the runtime of an algorithm scales with the size of the problem it's solving. The key components used in time complexity analysis are:

- ***Basic Operations:*** These are the atomic operations that an algorithm performs. In time complexity analysis, you typically assume that each basic operation takes a constant amount of time, denoted as O(1).
-
- ***Input Size:*** This is the size of the input data that an algorithm operates on. It's often represented by the variable "n."
-
- ***Counting Operations:*** In Big O notation, you count the number of basic operations executed by the algorithm as a function of the input size "n."
-
- ***Asymptotic Analysis:*** Big O notation describes the upper bound of the growth rate of the algorithm's time complexity as the input size becomes very large. It helps you understand how the algorithm's efficiency scales with large inputs.
-
- ***Big O Notation (O()):*** This is the notation used to express the upper bound of an algorithm's time complexity. For example, O(1) represents constant time complexity, O(log n) represents logarithmic time complexity, O(n) represents linear time complexity, and so on. The "O" stands for "order of," and the term inside the parentheses describes how the runtime scales with input size.
-
- ***Best, Average, and Worst-Case Time Complexity:*** Algorithms may have different time complexities in different scenarios. For instance, an algorithm might have a best-case time complexity (when everything goes smoothly), an average-case time complexity (a typical scenario), and a worst-case time complexity (the most unfavorable scenario). Big O notation is often used to describe the worst-case time complexity, as it provides an upper bound that guarantees performance in all cases.

Big O notation examples:

1. **Constant Time Complexity - O(1):**
   - This represents algorithms where the number of operations remains the same regardless of input size. In other words, the algorithm's runtime doesn't depend on the size of the data it's processing.
   - ***Example:*** Accessing an element in an array by its index. No matter how large the array is, it takes a constant amount of time to access any element because the index directly points to the desired element.
2. **Logarithmic Time Complexity - O(log n):**
   - Logarithmic time complexity is common in algorithms that repeatedly divide the input into smaller portions. As the input size grows, the number of operations increases slowly.
   - ***Example:*** Binary search, where you divide a sorted list in half with each comparison. It takes logarithmic time to find an element in a sorted list, making it very efficient for large datasets.
3. **Linear Time Complexity - O(n):**
   - Linear time complexity means the number of operations increases linearly with the size of the input data. If you double the input size, the number of operations roughly doubles.
   - ***Example:*** A simple linear search through an unsorted list. You may need to examine every element in the list to find a match, so the time it takes is proportional to the size of the list.
4. **Quadratic Time Complexity - O(n^2):**
   - Quadratic time complexity represents an algorithm that performs a constant number of operations for each element in the input data. As a result, the number of operations grows quadratically with the input size.
   - ***Example:*** Inefficient sorting algorithms like bubblesort or selection sort. For each element in the list, these algorithms compare it to every other element, leading to a quadratic number of comparisons for larger lists.
5. **Exponential Time Complexity - O(2^n):**
   - Exponential time complexity is very inefficient. It means that the number of operations doubles with each additional element in the input, making it impractical for large datasets.
   - ***Example:*** Some brute-force algorithms, such as solving the traveling salesman problem by considering every possible route. The number of possible routes grows exponentially with the number of locations to visit.
6. **Factorial Time Complexity - O(n!):**
   - Factorial time complexity is even more inefficient than exponential. It means the number of operations grows with the factorial of the input size, making it extremely slow for any non-trivial input.
   - ***Example:*** Permutations of a set of elements. Calculating all possible permutations becomes infeasible very quickly as the input size increases.
7. **Linearithmic Time Complexity - O(n log n):**
   - Linearithmic time complexity is common in algorithms that efficiently process data. It's faster than quadratic and exponential time complexities but slower than linear time complexity.
   - ***Example:*** Many efficient sorting algorithms like merge sort and heapsort have a time complexity of O(n log n). They divide the input data into smaller parts and combine them in a way that is more efficient than quadratic sorting algorithms like bubblesort.

Python (Basic)

1. Python Basics:

1.1. Comments:
- # This is a single-line comment
- """ This is a
- multi-line comment
- """

1.2. Print:
- print("Hello, World!")

1.3. Variables:
- my_variable = 42

1.4. Data Types:
- int, float, str, bool
- Lists, Tuples, Dictionaries, Sets

1.5. Type Conversion:
- int("42")
- float("3.14")
- str(42)

2. Control Flow:

2.1. Conditional Statements:
- if condition:
- # Code to run if condition is True
- elif another_condition:
- # Code to run if another_condition is True
- else:
- # Code to run if all conditions are False

2.2. For Loop:
- for item in iterable:
- # Code to run for each item

2.3. While Loop:
- while condition:
- # Code to run while condition is True

3. Functions:

3.1. Function Definition:
- def my_function(arg1, arg2):
- # Function code
- return result

3.2. Function Call:
- result = my_function(value1, value2)

4. Lists:

4.1. Create a List:
- my_list = [1, 2, 3]

4.2. Access Elements:
- element = my_list[0] # Indexing (0-based)

4.3. List Methods:
- append(), extend(), insert(), remove(), pop(), index(), count(), sort(), reverse()

1. Strings:

1.1. Create a String:
- my_string = "Hello, World!"

1.2. String Concatenation:
- new_string = string1 + string2

1.3. String Methods:
- upper(), lower(), strip(), replace(), split(), join()

2. Dictionaries:

2.1. Create a Dictionary:
- my_dict = {"key1": "value1", "key2": "value2"}

2.2. Access Values:
- value = my_dict["key1"]

2.3. Dictionary Methods:
- keys(), values(), items(), get()

3. Tuples and Sets:

3.1. Create a Tuple:
- my_tuple = (1, 2, 3)

3.2. Create a Set:
- my_set = {1, 2, 3}
- Tuples and Sets are similar to lists but are immutable (tuples) and unordered without duplicate elements (sets).

4. File Handling:

4.1. Open and Read a File:
- with open("file.txt", "r") as file:
- contents = file.read()

4.2. Write to a File:
- with open("file.txt", "w") as file:
- file.write("Hello, File!")

5. Exception Handling:

5.1. Try and Except:
- try:
- # Code that may raise an exception
- except ExceptionType:
- # Code to handle the exception

6. Libraries:

6.1. Import a Library:
- import library_name

6.2. Common Libraries:
- math, random, os, datetime, json, requests

Python (DSA)

1. Arrays:

1.1. Creating an Array:

```
my_array = [1, 2, 3, 4, 5]
```

1.2. Accessing Elements:

```
element = my_array[2]  # Accesses the element at index 2 (3rd element).
```

2. Lists:

2.1. Creating a List:

```
my_list = [1, 2, 3, 4, 5]
```

2.2. List Operations:

```
Append: my_list.append(6)
Remove: my_list.remove(3)
Sort: my_list.sort()
```

3. Stacks:

3.1. Using a Stack (Using Lists):

```
stack = []
stack.append(1)  # Push
popped_element = stack.pop()  # Pop
```

4. Queues:

4.1. Using a Queue (Using Lists):

```
from collections import deque
queue = deque()
queue.append(1)  # Enqueue
dequeued_element = queue.popleft()  # Dequeue
```

5. Linked Lists:

5.1. Defining a Node:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

5.2. Creating a Linked List:

```
node1 = Node(1)
node2 = Node(2)
node1.next = node2
```

1. Trees:

1.1. Creating a Binary Tree (Using Classes):

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
root = Node(1)
root.left = Node(2)
root.right = Node(3)
```

2. Sorting Algorithms:

2.1. Bubble Sort:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

2.2. Quick Sort:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

3. Searching Algorithms:

3.1. Binary Search:

```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

4. Hash Tables (Dictionaries):

4.1. Creating a Dictionary:

```
my_dict = {'key1': 'value1', 'key2': 'value2'}
Accessing Values:
value = my_dict['key1']
```

5. Recursion:

5.1. Recursive Factorial:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Python (OOP)

1. Classes and Objects:

1.1. Creating a Class:

```
class MyClass:
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2
    def method1(self):
        # Method code
my_object = MyClass("Value1", "Value2")
```

1.2. Accessing Attributes and Methods:

```
value1 = my_object.attribute1
my_object.method1()
```

2. Encapsulation:

2.1. Private Attributes and Methods (Name Mangling):

```
class MyClass:
    def __init__(self):
        self.__private_attribute = "I'm private"
    def __private_method(self):
        # Private method code
```

3. Inheritance:

3.1. Creating Subclasses:

```
class ParentClass:
    def __init__(self):
        self.parent_attribute = "Parent Attribute"
class ChildClass(ParentClass):
    def __init__(self):
        super().__init__()  # Call the parent class constructor
        self.child_attribute = "Child Attribute"
```

4. Polymorphism:

4.1. Method Overriding:

```
class ParentClass:
    def speak(self):
        return "Parent speaks."
class ChildClass(ParentClass):
    def speak(self):
        return "Child speaks."
```

4.2. Polymorphic Function:

```
def speak_and_print(obj):
    print(obj.speak())
parent = ParentClass()
child = ChildClass()
speak_and_print(parent)  # Calls ParentClass's speak method
speak_and_print(child)   # Calls ChildClass's speak method
```

1. Abstraction:

1.1. Abstract Base Classes:

```
from abc import ABC, abstractmethod
class MyAbstractClass(ABC):
    @abstractmethod
    def my_abstract_method(self):
        pass
```

2. Method Overloading:

2.1. Using Default Arguments:

```
class MyClass:
    def my_method(self, arg1, arg2="default"):
        # Method code
```

3. Class Variables:

3.1. Creating Class Variables:

```
class MyClass:
    class_variable = "I'm a class variable"
    def __init__(self, instance_variable):
        self.instance_variable = instance_variable
```

3.2. Accessing Class Variables:

```
MyClass.class_variable
```

4. Static Methods:

4.1. Creating Static Methods:

```
class MyClass:
    @staticmethod
    def my_static_method(arg1, arg2):
        # Static method code
```

- Accessing Static Methods:

```
MyClass.my_static_method(value1, value2)
```