Fall 2024 – CC-213 Data Structure and Algorithms

Week 01 – Introduction to Data Structure and Algorithms

1 Introduction to Data Structure and Algorithms

The Data Structures and Algorithms (DSA) course is fundamental to mastering efficient programming practices. This course is designed to teach students how to effectively organize, store, and manipulate data, providing them with the tools needed to solve real-world programming problems. It covers a range of essential data structures such as arrays, linked lists, stacks, queues, hash tables, trees, and graphs. As students progress, they'll explore more advanced structures like heaps and AVL trees, learning both their implementation and the pros and cons of each.

In addition to understanding data structures, the course focuses heavily on algorithms—the step-by-step processes used to solve problems. Topics like sorting, searching, and recursion form the core of this section. A key part of the course is analyzing the time and space complexity of these algorithms, typically using Big O notation. This analysis helps students evaluate how efficient different algorithms are in terms of memory and processing time, which is critical for developing optimized solutions in real-world applications.

Mastering DSA is essential for anyone pursuing a career in computer science or software development. Beyond understanding how data is organized and managed, DSA empowers students to write code that performs efficiently. These skills are crucial for building systems that scale well, such as search engines, databases, and operating systems. In today's fast-paced technological landscape, where performance and resource efficiency are key, having a strong grasp of DSA can make all the difference.

Additionally, technical interviews in the software industry frequently focus on DSA-related challenges. Mastery of these concepts not only boosts job prospects but also enhances problem-solving skills. Students will develop a mindset for breaking down complex problems into simpler parts, enabling them to choose the best data structures and algorithms for different situations, improving their overall coding proficiency.

This course also builds naturally on the principles learned in Object-Oriented Programming (OOP). While OOP teaches how to structure code using objects and classes, DSA enhances that by teaching how to manage and process data efficiently within that structure. When combined, OOP and DSA allow students to create highly scalable, maintainable, and optimized software systems.

Summary

Data Structures and Algorithms equips students with precision tools, much like a craftsman perfecting their trade, to tackle the complexity of real-world programming. This course lays the foundation for becoming proficient in solving challenging problems, both in academic settings and professional environments. Through continuous practice, students will refine their ability to write optimized, efficient code—building the critical thinking skills needed for advanced software development.

1.1Searching

Searching is one of the most fundamental operations performed on data structures. It involves finding the position of a specific element within a dataset, such as an array or list. Two common searching algorithms are **Linear Search** and **Binary Search**, each of which operates differently and has its own advantages and disadvantages in terms of efficiency.

1.1.1 Linear Search

Linear Search is the simplest searching algorithm. It works by checking each element of the array one by one until the desired element is found or the end of the array is reached.

1.1.1.1 Algorithm:

- Start from the first element of the array.
- Compare the current element with the target element.
- If the current element matches the target, return its index.
- If the end of the array is reached without finding the element, return -1.

1.1.1.2 C++ Code:

```
int linearSearch(int arr[], int n, int target) {
   for (int i = 0; i < n; i++) {
      if (arr[i] == target) {
         return i; // Return the index of the target
      }
   }
   return -1; // Target not found
}
```

1.1.1.3 Dry Run:

- Array: {2, 4, 6, 8, 10}
- Target: 8
- Step-by-step:
  1. Compare 2 with 8 → no match.
  2. Compare 4 with 8 → no match.
  3. Compare 6 with 8 → no match.
  4. Compare 8 with 8 → match found at index 3.

1.1.1.4 Step Count:

- Best case: 1 comparison (element is at the first position).
- Worst case: n comparisons (element is at the last position or not in the array).

1.1.1.5 Time Complexity:

- Best case: **O(1)** (first element is the target).
- Worst case: **O(n)** (target is at the end or not present).

1.1.1.6 Rate of Growth:

- Linear Search grows linearly with the size of the array, meaning if the array size doubles, the time taken roughly doubles.

1.1.2 Binary Search

Binary Search is a much more efficient algorithm, but it can only be applied to sorted arrays. It works by repeatedly dividing the search interval in half and discarding the half in which the target cannot exist.

1.1.2.1 Algorithm:

- Start by checking the middle element of the array.
- If the middle element is the target, return its index.
- If the target is smaller than the middle element, repeat the search on the left half.
- If the target is larger, repeat the search on the right half.
- Continue dividing until the target is found or the interval is empty.

1.1.2.2 C++ Code:

```
int binarySearch(int arr[], int n, int target) {
  int low = 0, high = n - 1;
  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (arr[mid] == target) {
      return mid; // Return the index of the target
    } else if (arr[mid] < target) {
      low = mid + 1; // Search the right half
    } else {
      high = mid - 1; // Search the left half
    }
  }
  return -1; // Target not found
}
```

1.1.2.3 Dry Run:

- Array: {2, 4, 6, 8, 10}
- Target: 8
- Step-by-step:

1. low = 0, high = 4, mid = 2. Compare arr[2] (6) with 8 → 8 is greater, so search in the right half.
2. low = 3, high = 4, mid = 3. Compare arr[3] (8) with 8 → match found at index 3.

1.1.2.4 Step Count:

- In each step, the size of the array is halved. For an array of size n, the number of steps required is approximately $\log_2(n)$.

1.1.2.5 Step Time Equation:

- At each step, one comparison is made, and the size of the search space is halved.
- The time complexity is proportional to the number of steps taken, which is $\log_2(n)$.

1.1.2.6 Time Complexity:

- Best case: **O(1)** (middle element is the target).
- Worst case: **O(log n)** (requires halving the array multiple times).

1.1.2.7 Rate of Growth:

- Binary Search grows logarithmically. As the size of the array increases, the number of required steps grows much more slowly compared to Linear Search. For example, doubling the size of the array only adds one extra step.

1.1.3 Comparing Linear and Binary Search:

1.1.3.1 Efficiency:

- Linear Search: O(n) in the worst case, meaning it scales linearly with the size of the input.
- Binary Search: O(log n), much faster for larger datasets but requires the array to be sorted.

1.1.3.2 Use Cases:

- Use Linear Search for small, unsorted datasets.
- Use Binary Search for large, sorted datasets, where efficiency is important.

1.1.4 Rate of Growth Comparison:

When comparing **Linear Search** and **Binary Search**, their time complexities highlight significant differences in performance, especially for large datasets.

- **Linear Search**: O(n – as the input size grows, the time required grows proportionally.
- **Binary Search**: $O(\log_2 n)$ – time grows logarithmically, meaning it scales much better for larger datasets.

For example: For n = 1,000,000, Linear Search may require up to 1,000,000 steps, while Binary Search requires only about 20 steps.

This efficiency gain makes Binary Search a much more powerful algorithm for searching in large, sorted datasets.

1.1.5 Summary:

Both **Linear Search** and **Binary Search** are fundamental algorithms, but their use depends on the context. Linear Search is simple and works on unsorted arrays, but it can be slow for large datasets. Binary Search, while much faster due to its logarithmic time complexity, requires the data to be sorted. Understanding and analyzing these algorithms is crucial for optimizing data processing, and the concepts of **step count**, **step time equations**, and **time complexity** using Big O notation will serve as the foundation for analyzing more advanced algorithms throughout the course.

1.1.6 Step Counts for Linear and Binary Search a working example

Let's calculate the best, average, and worst-case step counts for **Linear Search** and **Binary Search** when searching for an element in arrays of sizes 10, 100, and 1000.

1.1.6.1 Linear Search:

- **Best-case**: The target element is found at the first position.
- **Average-case**: The target element is somewhere in the middle of the array.
- **Worst-case**: The target element is at the last position or not present in the array.

For an array of size n, here are the step counts:

Array Size (n)	Best Case (1st position)	Average Case (middle position)	Worst Case (last position)
10	1	5	10
100	1	50	100
1000	1	500	1000

1.1.6.2 Binary Search:

- **Best-case**: The target element is found at the mid-point in the first iteration.
- **Average-case**: The target element is found after a few iterations, requiring fewer comparisons.
- **Worst-case**: The target element is found or not found after all divisions are completed.

For **Binary Search**, the number of steps is proportional to $\log_2(n)$, where n is the size of the array.

Array Size (n)	Best Case (mid-point in 1st iteration)	Average Case (logarithmic reduction)	Worst Case (full log2 reductions)
10	1	$\log_2(10) \approx 3.3$	$\lceil \log_2 10 \rceil = 4$
100	1	$\log_2(100) \approx 6.6$	$\lceil \log_2(100) \rceil = 7$
1000	1	$\log_2(1000) \approx 9.97$	$\lceil \log_2(1000) \rceil = = 10$

2 Algorithm Performance Analysis vs. Measurement

Algorithm performance analysis and **algorithm performance measurement** are two essential concepts in understanding how efficiently an algorithm operates. Both help evaluate how well an algorithm solves a problem, but they differ in their approach, timing, and purpose.

2.1 Performance Analysis (Theoretical Analysis)

Definition: Algorithm performance analysis refers to the **theoretical evaluation** of an algorithm's efficiency. It is done by analyzing the algorithm without actually running it, focusing on its **time complexity** and **space complexity** using mathematical techniques.

- **Time complexity**: Measures how the runtime of the algorithm increases with input size n.
- **Space complexity**: Measures how the memory usage grows with the input size.

Performance analysis is usually expressed in **Big O notation**, which categorizes algorithms based on their growth rates (e.g., O(n), O(n^2), O(log n), etc.).

Example:

- For **linear search**, the time complexity is O(n) because, in the worst case, we may need to search through the entire array.
- For **binary search**, the time complexity is O(log n) because it divides the search space in half with each step.

Advantages:

- Can be applied before implementation.
- Provides insight into how the algorithm scales with input size.
- Helps compare the efficiency of multiple algorithms.

2.1.1 Performance Analysis of Algorithms: Linear Search and Binary Search

Performance analysis of algorithms is done through theoretical evaluation of their efficiency. This involves counting the number of steps required to execute an algorithm for an input of size n. We will look at two common search algorithms — **linear search** and **binary search** — and evaluate their performance through step counts, step count equations, and Big O notation.

2.1.1.1 Linear Search Algorithm

Algorithm Description: Linear Search is a simple algorithm that checks each element of an array sequentially until the target element is found or the array ends. It works with **unsorted arrays**.

```
int linearSearch(int arr[], int n, int target) {
  for (int i = 0; i < n; i++) {        // Loop iterates 'n+1' times (i<n) and 'n' time (i++) in the worst case
    if (arr[i] == target) {            // 1 comparison per iteration
      return i;                  // 1 return statement (in best case, once)
    }
  }
  return -1;                      // 1 return if target not found (in worst case)
```

}

Step-by-Step Breakdown: For each iteration of the loop:

1. **Loop condition check** (i < n): 1 step per iteration and 1 for loop termination in worst case.
2. **Loop condition increment** (i++): 1 step per iteration
3. **Array element comparison** (arr[i] == target): 1 step per iteration.
4. **Return statement** if the target is found: 1 step (executed only once when target is found).
5. **Return -1** if the target is not found: 1 step (executed once, only in the worst case).

- **Best Case**: The target is found at the first position (i=0).
    - **Steps executed**: 1 loop condition check, 1 comparison, 1 return.
    - Total = 2 steps (loop + comparison) + 1 step for return = **3 steps**.
- **Worst Case**: The target is not found in the array or is at the last position (i=n−1).
    - **Steps executed**: The loop runs n times, with 1 comparison per iteration, and the function returns after the loop finishes.
    - Total = n + 1 loop condition checks + n loop increments + n comparisons + 1 return statement = (n+1) + (n) + (n) + 1 **steps = 3n + 2 steps**.

Step Count Equation: For **linear search**, the total number of steps in the worst case is: T (n)=3n+2

- 3n + 1: One step each for loop condition, increment and inner comparison, per iteration.
- +1: For the final return statement after all iterations are completed.

Big O Notation: The Big O notation provides the upper bound of the time complexity. The dominant term in the step count equation is n, meaning the performance grows **linearly** with the input size.

Time Complexity (Worst Case)=O(n): Linear search has **O(n)** time complexity in the worst case because the time required grows directly with the size of the input array.

2.1.1.2 Binary Search Algorithm

Algorithm Description: Binary Search is a more efficient algorithm, but it only works with **sorted arrays**. The idea is to repeatedly divide the search space in half by comparing the target element to the middle element of the array.

```
int binarySearch(int arr[], int n, int target) {
  int low = 0, high = n - 1;            // 2 assignments (low and high)
  while (low <= high) {                 // Loop continues until low > high
    int mid = low + (high - low) / 2;   // 3 steps: subtraction, division, and addition
    if (arr[mid] == target) {           // 1 comparison per iteration
      return mid;                       // 1 return if target found
```

```
    } else if (arr[mid] < target) {      // 1 comparison, 1 assignment (if true)
      low = mid + 1;                     // 1 assignment
    } else {
      high = mid - 1;                    // 1 assignment
    }
  }
  return -1;                             // 1 return if target not found
}
```

Step-by-Step Breakdown: For each iteration of the loop:

1. **Loop condition check** (low <= high): 1 step per iteration.
2. **Middle index calculation** (mid = low + (high - low) / 2): 3 steps (subtraction, division, and addition).
3. **Comparison** (arr[mid] == target): 1 step per iteration.
4. **Low or high adjustment** (low = mid + 1 or high = mid - 1): 1 assignment step per iteration.

- **Best Case**: The target is found in the first comparison.
    - **Steps executed**: Initial assignments (2 steps), 1 loop iteration (5 steps), and return (1 step).
    - Total = 2 (initial assignments) + 5 (loop body) + 1 (return) = **8 steps**.
- **Worst Case**: The target is not found, or the algorithm searches through the entire array.
    - For each iteration, we reduce the search space by half, so the loop runs approximately $\log_2 n$ times.
    - Each iteration consists of:
        - 1 step for the loop check.
        - 3 steps for the middle index calculation.
        - 1 step for comparison.
        - 1 step for updating low or high.
    - Total steps per iteration: 1+3+1+1=6 steps.
    - Total = $6 \cdot \log_2 n + 2$ steps in the worst case.

Step Count Equation:

For **binary search**, the total number of steps in the worst case is: $T(n) = 6 \cdot \log_2 n + 2$

- 6: Steps per iteration (loop check, index calculation, comparison, and assignment).
- +2: Initial assignments before the loop starts.

Comparison of Step Count Equations

Algorithm	Best Case Step Count	Worst Case Step Count
Linear Search	3	3n+2
Binary Search	8	$6 \cdot \log_2 n + 2$

Big O Notation from Step Count Equations:

- **Linear Search**: From the equation $T(n)=3n+2$, we focus on the dominant term 3n, which grows linearly with n. Big O notation: **O(n)** (linear time complexity).
- **Binary Search**: From the equation $T(n)=6 \cdot \log_2 n+2$, we focus on the dominant term $6 \cdot \log_2 n$, which grows logarithmically. Big O notation: **$O(\log_2 n)$** (logarithmic time complexity).

2.1.1.3 Comparison of Linear Search and Binary Search Performance

Algorithm	Best Case Time Complexity	Average Case Time Complexity	Worst Case Time Complexity
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)

- **Linear Search**: In the best case, linear search finds the target immediately with O(1) time complexity, but in the worst case, it has O(n), meaning it must go through the entire array. Linear search is simple but less efficient, particularly for large data sets, with a worst-case time complexity of O(n).
- **Binary Search**: In the best case, binary search also has O(1), but its worst-case performance is O(log n), meaning the search space reduces exponentially with each step. Binary search, on the other hand, is much more efficient with a worst-case time complexity of O(log n), but it requires a sorted array to function.

By analyzing the step count equations and using Big O notation, we can clearly understand how different algorithms scale and how they behave as the input size increases, which is essential for making informed decisions when choosing the right algorithm for a particular problem.

2.2 Performance Measurement (Empirical Analysis)

Definition: Algorithm performance measurement refers to the **empirical evaluation** of an algorithm by running it on a machine and measuring **real-world metrics** such as runtime, memory usage, and CPU usage. It is often done using **profiling tools** or benchmarking.

Performance measurement provides concrete data on how an algorithm behaves in practice with specific hardware, software, and datasets.

Advantages:

- Reflects real-world behavior, including hardware and software limitations.

- Can reveal practical inefficiencies like cache misses, I/O bottlenecks, and network delays.
- Useful in identifying factors that affect algorithm performance outside of pure complexity.

2.2.1 Example:

Following is a C++ program to empirically measure the run-time performance of both **linear search** and **binary search** on a computer. It generates random arrays and measures the execution time of both algorithms using the chrono library for high-precision timing.

```
#include <iostream>
#include <vector>
#include <algorithm> // For sort()
#include <chrono>    // For high-resolution clock
#include <cstdlib>  // For rand() and srand()
#include <ctime>    // For time()
using namespace std;
using namespace std::chrono;

// Function to generate a random array of size n
vector<int> generateRandomArray(int n) {
   vector<int> arr(n);
   for (int i = 0; i < n; i++) {
      arr[i] = rand() % 100000;  // Random numbers between 0 and 99,999
   }
   return arr;
}

// Linear Search function
int linearSearch(const vector<int>& arr, int target) {
   for (int i = 0; i < arr.size(); i++) {
      if (arr[i] == target) {
         return i;
      }
   }
   return -1;
}

// Binary Search function
int binarySearch(const vector<int>& arr, int target) {
   int low = 0, high = arr.size() - 1;
   while (low <= high) {
      int mid = low + (high - low) / 2;
      if (arr[mid] == target) {
         return mid;
      } else if (arr[mid] < target) {
         low = mid + 1;
      } else {
         high = mid - 1;
```

```
        }
    }
    return -1;
}

// Main function to measure performance of both algorithms
int main() {
    srand(time(0)); // Seed for random number generator

    int n; // Size of the array
    cout << "Enter the size of the array: ";
    cin >> n;

    // Generate a random array
    vector<int> arr = generateRandomArray(n);

    // Target element to search
    int target = rand() % 100000;

    // Measure the time for Linear Search
    auto start_linear = high_resolution_clock::now();
    int result_linear = linearSearch(arr, target);
    auto stop_linear = high_resolution_clock::now();
    auto duration_linear = duration_cast<microseconds>(stop_linear - start_linear);
    cout << "Linear Search Time: " << duration_linear.count() << " microseconds" << endl;

    // Sort the array for Binary Search (required for binary search to work)
    sort(arr.begin(), arr.end());

    // Measure the time for Binary Search
    auto start_binary = high_resolution_clock::now();
    int result_binary = binarySearch(arr, target);
    auto stop_binary = high_resolution_clock::now();
    auto duration_binary = duration_cast<microseconds>(stop_binary - start_binary);
    cout << "Binary Search Time: " << duration_binary.count() << " microseconds" << endl;

    // Output search results (optional)
    if (result_linear != -1) {
        cout << "Linear Search: Target found at index " << result_linear << endl;
    } else {
        cout << "Linear Search: Target not found" << endl;
    }

    if (result_binary != -1) {
        cout << "Binary Search: Target found at index " << result_binary << endl;
    } else {
        cout << "Binary Search: Target not found" << endl;
    }
```

```
    return 0;
}
```

2.3 Key Differences

Aspect	Performance Analysis (Theoretical)	Performance Measurement (Empirical)
Nature	Theoretical, based on mathematical analysis	Empirical, based on real-world execution
Tools Used	Mathematical tools, Big O notation	Profiling tools, benchmarking
When Applied	Before implementing or running the algorithm	After implementing and running the algorithm
Focus	Time and space complexity, asymptotic behavior	Actual runtime, memory usage, CPU usage, hardware effects
Hardware/Software Impact	Ignores hardware and software factors	Takes hardware and software factors into account
Goal	To determine general efficiency and scalability	To measure real-world performance
Example	Binary Search has O(log n) time complexity	Binary Search takes 0.5 ms on a specific dataset

2.4 Example of Both Approaches:

- **Performance Analysis**: For **Quick Sort**, the worst-case time complexity is O(n^2) (when the pivot is poorly chosen), and the average-case time complexity is O(n log n).
- **Performance Measurement**: After implementing Quick Sort, you may find that on a real dataset of 1 million elements, the actual runtime is 2 seconds on a particular machine, and it performs better with optimized hardware or cache.

In summary, **performance analysis** provides theoretical insight into how an algorithm behaves as input size grows, while **performance measurement** gives a real-world evaluation of the algorithm's performance in specific conditions. Both are crucial in designing and selecting efficient algorithms.