

# DSA QB

## ChatGPT 3.5

**You**

1. Write an algorithm for selection sort.

**ChatGPT**

Selection Sort Algorithm:

1. Start with an unsorted array of elements.
2. Iterate through the array from the first element to the second-to-last element.
3. For each iteration, assume the current element is the minimum.
4. Compare the current element with all the following elements in the array to find the minimum.
5. If a smaller element is found, update the index of the minimum element.
6. Swap the current element with the minimum element found.
7. Continue this process until the array is completely sorted.
8. Repeat steps 2-7 until all elements are in their correct positions.

Pseudocode:

less

```
function selectionSort(arr):
    n = length of arr
    for i from 0 to n - 2:
        minIndex = i
        for j from i + 1 to n - 1:
            if arr[j] < arr[minIndex]:
                minIndex = j
        if minIndex != i:
            swap(arr[i], arr[minIndex])
    return arr
```

This algorithm sorts an array by repeatedly finding the minimum element from the unsorted part and putting it at the beginning. It has a time complexity of  $O(n^2)$  for average and worst-case scenarios, where  $n$  is the number of elements in the array.

**You**

2. Write algorithm for bubble sort method.

**ChatGPT**

Bubble Sort Algorithm:

1. Start with an unsorted array of elements.

2. Iterate through the array from the first element to the second-to-last element.
3. For each iteration, compare each element with its adjacent element.
4. If the current element is greater than the next element, swap them.
5. Continue this process until the end of the array.
6. After completing one iteration, the largest element will be bubbled up to its correct position at the end of the array.
7. Repeat steps 2-6 for the remaining elements, excluding the last element that is already in its correct position.
8. Continue this process until the entire array is sorted.

Pseudocode:

vbnet

```
function bubbleSort(arr):
    n = length of arr
    for i from 0 to n - 1:
        for j from 0 to n - i - 2:
            if arr[j] > arr[j + 1]:
                swap(arr[j], arr[j + 1])
    return arr
```

This algorithm repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. It has a time complexity of  $O(n^2)$  for average and worst-case scenarios, where  $n$  is the number of elements in the array.

 **You**

3. Apply quick sort for the following data: 19, 70, 25, 110, 212, 28, 14, 73, 10, 69

 **ChatGPT**

To apply the quick sort algorithm to the given data, we'll follow these steps:

1. Choose a pivot element from the data set.
2. Partition the data into two sub-arrays: elements less than the pivot and elements greater than the pivot.
3. Recursively apply the quick sort algorithm to the sub-arrays.
4. Combine the sorted sub-arrays to get the final sorted array.

Let's apply quick sort to the given data:

Data: 19, 70, 25, 110, 212, 28, 14, 73, 10, 69

1. Choose a pivot. For simplicity, we'll choose the first element (19) as the pivot.
2. Partition the data:
  - Lesser elements than the pivot: 14, 10
  - Greater elements than the pivot: 70, 25, 110, 212, 28, 73, 69
3. Recursively apply quick sort to the two sub-arrays:
  - Lesser elements: 10, 14
  - Greater elements: 25, 28, 69, 70, 73, 110, 212
4. Combine the sorted sub-arrays: 10, 14, 19, 25, 28, 69, 70, 73, 110, 212

The sorted array using quick sort is: 10, 14, 19, 25, 28, 69, 70, 73, 110, 212.

**22 You**

4. Write an algorithm of radix sort and apply radix sort on the following data: 1121, 32, 564, 123, 1, 450, 788

**ChatGPT**

Radix Sort Algorithm:

1. Find the maximum number in the array to determine the number of digits in the maximum number.
2. Iterate through each digit place (units, tens, hundreds, etc.).
3. Sort the elements based on the digit place from the least significant digit (units place) to the most significant digit (highest place value).
4. Combine the sorted arrays after sorting each digit place.
5. Repeat steps 2-4 until all digits have been considered.

Pseudocode:

lua

```
function getMax(arr):
    max = arr[0]
    for i from 1 to length of arr - 1:
        if arr[i] > max:
            max = arr[i]
    return max

function countingSort(arr, exp):
    n = length of arr
    output = new array of size n
    count = new array of size 10 initialized to 0

    for i from 0 to n - 1:
        index = (arr[i] / exp) % 10
        count[index]++

    for i from 1 to 9:
        count[i] += count[i - 1]

    for i from n - 1 down to 0:
        index = (arr[i] / exp) % 10
        output[count[index] - 1] = arr[i]
        count[index]--

    for i from 0 to n - 1:
        arr[i] = output[i]

function radixSort(arr):
    max = getMax(arr)
    exp = 1
    while max / exp > 0:
        countingSort(arr, exp)
        exp *= 10
    return arr
```

Now, let's apply Radix Sort to the given data:

Data: 1121, 32, 564, 123, 1, 450, 788

1. Find the maximum number in the array. The maximum number is 1121, so the maximum number of digits is 4.
2. Sort based on units place (least significant digit):
  - 1121, 32, 123, 1, 450, 564, 788
3. Sort based on tens place:
  - 1, 1121, 32, 123, 450, 564, 788
4. Sort based on hundreds place:
  - 1, 32, 123, 450, 564, 788, 1121
5. Sort based on thousands place (not applicable in this case as no number has a thousands place).
6. The array is now sorted: 1, 32, 123, 450, 564, 788, 1121.

## You

5. Sort the given data using bubble sort. Data: 100,90,80,70,60,50,40

## ChatGPT

To sort the given data using the bubble sort algorithm:

Data: 100, 90, 80, 70, 60, 50, 40

Bubble Sort Algorithm:

1. Start with an unsorted array of elements.
2. Repeat the following steps until no swaps are needed:
  - a. Iterate through the array from the first element to the second-to-last element.
  - b. Compare each element with its adjacent element.
  - c. If the current element is greater than the next element, swap them.
3. After completing one iteration, the largest element will be bubbled up to its correct position at the end of the array.
4. Repeat steps 2-3 until the entire array is sorted.

Let's apply the bubble sort algorithm to the given data:

Iteration 1:

- Compare and swap: 100, 90, 80, 70, 60, 50, 40 (no swaps needed)

Iteration 2:

- Compare and swap: 90, 100, 80, 70, 60, 50, 40 (swap 90 and 100)
- Compare and swap: 90, 80, 100, 70, 60, 50, 40 (swap 80 and 100)

Iteration 3:

- Compare and swap: 90, 80, 70, 100, 60, 50, 40 (swap 70 and 100)
- Compare and swap: 90, 80, 70, 60, 100, 50, 40 (swap 60 and 100)
- Compare and swap: 90, 80, 70, 60, 50, 100, 40 (swap 50 and 100)

Iteration 4:

- Compare and swap: 80, 90, 70, 60, 50, 100, 40 (swap 80 and 90)

- Compare and swap: 80, 70, 90, 60, 50, 100, 40 (swap 70 and 90)  
Iteration 5:
- Compare and swap: 80, 70, 60, 90, 50, 100, 40 (swap 60 and 90)
- Compare and swap: 80, 70, 60, 50, 90, 100, 40 (swap 50 and 90)  
Iteration 6:
- Compare and swap: 70, 80, 60, 50, 90, 100, 40 (swap 70 and 80)  
Iteration 7:
- Compare and swap: 70, 60, 80, 50, 90, 100, 40 (swap 60 and 80)
- Compare and swap: 70, 60, 50, 80, 90, 100, 40 (swap 50 and 80)  
Iteration 8:
- Compare and swap: 60, 70, 50, 80, 90, 100, 40 (swap 60 and 70)  
Iteration 9:
- Compare and swap: 60, 50, 70, 80, 90, 100, 40 (swap 50 and 70)  
Iteration 10:
- Compare and swap: 50, 60, 70, 80, 90, 100, 40 (swap 50 and 60)  
Iteration 11:
- Compare and swap: 50, 60, 70, 80, 90, 40, 100 (swap 40 and 100)  
Iteration 12:
- Compare and swap: 50, 60, 70, 80, 40, 90, 100 (swap 40 and 90)  
Iteration 13:
- Compare and swap: 50, 60, 70, 40, 80, 90, 100 (swap 40 and 80)  
Iteration 14:
- Compare and swap: 50, 60, 40, 70, 80, 90, 100 (swap 40 and 70)  
Iteration 15:
- Compare and swap: 50, 40, 60, 70, 80, 90, 100 (swap 40 and 60)  
Iteration 16:
- Compare and swap: 40, 50, 60, 70, 80, 90, 100 (swap 40 and 50)

After 16 iterations, the array is sorted: 40, 50, 60, 70, 80, 90, 100.

## 22 You

6. Demonstrate the sorting process of insertion sort on the list: 20, 710, 315, 87, 143, 379, 147, 189, 517, 29 and represent its worst case time complexity.

## ChatGPT

Insertion Sort Algorithm:

1. Start with an unsorted array of elements.
2. Iterate through the array from the second element to the last element.
3. For each iteration, consider the current element and compare it with the elements to its left.
4. Move the current element to its correct position by shifting elements to the right until the correct position is found.
5. Continue this process until all elements are sorted.

Let's demonstrate the sorting process of insertion sort on the given list:

Initial list: 20, 710, 315, 87, 143, 379, 147, 189, 517, 29

1. Consider the second element (710):
  - Compare 710 with the first element (20), no shifting needed.
2. Consider the third element (315):
  - Compare 315 with 710, no shifting needed.
  - Compare 315 with 20, insert 315 before 710.
3. Consider the fourth element (87):
  - Compare 87 with 710, shift 710 to the right.
  - Compare 87 with 315, shift 315 to the right.
  - Compare 87 with 20, insert 87 before 315.
4. Consider the fifth element (143):
  - Compare 143 with 710, shift 710 to the right.
  - Compare 143 with 315, no shifting needed.
  - Compare 143 with 87, shift 87 to the right.
  - Compare 143 with 20, insert 143 before 315.
5. Continue this process for the remaining elements until the list is fully sorted.

Sorted list: 20, 29, 87, 143, 147, 189, 315, 379, 517, 710

#### Worst Case Time Complexity of Insertion Sort:

In the worst-case scenario, when the array is in reverse order, each element will have to be compared and shifted all the way to the beginning of the array. Thus, the time complexity of insertion sort in the worst case is  $O(n^2)$ , where  $n$  is the number of elements in the array.

#### You

7. What is the best case and worst case time complexity of the merge sort? Demonstrate the sorting process of merge sort on the data: 135, 708, 123, 193, 277, 490, 87, 655, 480, 68.

#### ChatGPT

#### Merge Sort Time Complexity:

1. Best Case: The best-case time complexity of merge sort is  $O(n \log n)$ . This occurs when the input array is already sorted or nearly sorted.
2. Worst Case: The worst-case time complexity of merge sort is also  $O(n \log n)$ . This occurs regardless of the initial order of elements in the input array.

Now, let's demonstrate the sorting process of merge sort on the given data: 135, 708, 123, 193, 277, 490, 87, 655, 480, 68.

#### Merge Sort Algorithm:

1. Divide the array into halves recursively until each sub-array contains only one element.
2. Merge the divided arrays back together in sorted order.

Here's the step-by-step sorting process:

Step 1: Divide the array into halves recursively.

CSS

```
[135, 708, 123, 193, 277] [490, 87, 655, 480, 68]
```

Step 2: Divide each sub-array further.

CSS

```
[135, 708] [123, 193, 277] [490, 87] [655, 480, 68]
```

Step 3: Continue dividing until each sub-array contains only one element.

CSS

```
[135] [708] [123] [193] [277] [490] [87] [655] [480] [68]
```

Step 4: Merge the divided arrays back together in sorted order.

CSS

```
[135, 708] [123, 193, 277] [87, 490] [68, 480, 655]
```

Step 5: Merge the remaining divided arrays.

CSS

```
[123, 135, 193, 277, 708] [68, 87, 480, 490, 655]
```

Step 6: Merge the final two sorted arrays.

Csharp

```
[68, 87, 123, 135, 193, 277, 480, 490, 655, 708]
```

The sorted array using merge sort is: 68, 87, 123, 135, 193, 277, 480, 490, 655, 708.

## 22 You

8. Explain sequential searching with example and algorithm. Analyze it for worst case time complexity. Represent its time complexity using Big-oh (O) notation.



## ChatGPT

Sequential Search, also known as Linear Search, is a simple search algorithm that sequentially checks each element of the array or list until a match is found or until all elements have been checked. It does not require the array or list to be sorted beforehand.

Example:

Consider an array of integers: [12, 45, 7, 23, 56, 89, 34]. Let's say we want to search for the element 23.

1. Start from the beginning of the array.
2. Compare each element with the target element (23 in this case) until a match is found or until the end of the array is reached.
3. If a match is found, return the index of the element (index 3 in this case).
4. If no match is found after checking all elements, return a message indicating that the element is not found.

Algorithm:

vbnet

```
function sequentialSearch(arr, target):
    for i from 0 to length of arr - 1:
        if arr[i] equals target:
            return i
    return "Element not found"
```

Worst Case Time Complexity Analysis:

In the worst case scenario, the element being searched for is not present in the array, or it is the last element in the array. In this case, the algorithm will have to iterate through all the elements in the array to determine that the element is not present.

The worst-case time complexity of sequential search is  $O(n)$ , where  $n$  is the number of elements in the array. This is because the algorithm may have to iterate through all  $n$  elements in the array before determining that the target element is not present.

Using Big-Oh ( $O$ ) notation, we represent the worst-case time complexity of sequential search as  $O(n)$ , indicating that the time it takes to execute the algorithm grows linearly with the size of the input array.

22 You

9. Compare sequential searching with binary searching in detail.



ChatGPT

Sequential Search vs. Binary Search:

Sequential Search:

1. **Algorithm:** Sequential search (or linear search) sequentially checks each element in the array until the target element is found or until the end of the array is reached.
2. **Efficiency:** It is simple to implement and works efficiently for small arrays or unsorted lists.
3. **Time Complexity:** The worst-case time complexity of sequential search is  $O(n)$ , where  $n$  is the number of elements in the array. This means the time it takes to search grows linearly with the size of the input.
4. **Suitability:** It is suitable for unsorted arrays or lists and when the location of the target element is not known in advance.

Binary Search:



1. **Algorithm:** Binary search requires the array to be sorted beforehand. It repeatedly divides the search interval in half until the target element is found or until the interval becomes empty.
2. **Efficiency:** It is highly efficient for large sorted arrays or lists.
3. **Time Complexity:** The time complexity of binary search is  $O(\log n)$ , where  $n$  is the number of elements in the array. This is because at each step, the search interval is divided in half, leading to a logarithmic time complexity.
4. **Suitability:** It is suitable for sorted arrays or lists and when the location of the target element is likely to be in the middle of the array.

Comparison:

1. **Efficiency:** Binary search is generally more efficient than sequential search for large datasets due to its logarithmic time complexity. However, for small datasets or unsorted lists, sequential search may be more efficient.
2. **Preprocessing:** Binary search requires the array to be sorted beforehand, which may incur additional preprocessing time. Sequential search does not require any preprocessing.
3. **Search Interval:** Binary search divides the search interval in half at each step, while sequential search checks elements one by one.
4. **Suitability:** Binary search is suitable for sorted arrays or lists, whereas sequential search can be used for both sorted and unsorted arrays or lists.
5. **Space Complexity:** Both algorithms have a space complexity of  $O(1)$  as they do not require additional space proportional to the size of the input.

In summary, binary search is more efficient for large sorted datasets, while sequential search is simpler and suitable for small or unsorted datasets. The choice between the two depends on factors such as the size of the dataset, whether it is sorted, and the efficiency requirements of the search operation.

## You

10. Explain binary searching algorithm with example.

## ChatGPT

Binary Search Algorithm:

Binary search is an efficient search algorithm used to find the position of a target value within a sorted array or list. It works by repeatedly dividing the search interval in half until the target element is found or until the interval becomes empty.

1. **Initialize pointers:** Define two pointers, `left` and `right`, which represent the boundaries of the search interval. Initially, `left` points to the first element of the array, and `right` points to the last element.
2. **Find the middle element:** Calculate the middle index of the search interval using the formula `mid = (left + right) / 2`.
3. **Compare with the target element:** Compare the middle element of the array with the target element:

- If the middle element is equal to the target element, return its index (search successful).
- If the middle element is greater than the target element, update `right` to `mid - 1` to search in the left half of the array.
- If the middle element is less than the target element, update `left` to `mid + 1` to search in the right half of the array.

4. **Repeat:** Repeat steps 2 and 3 until the target element is found or until the interval becomes empty (`left` is greater than `right`).

5. **Return not found:** If the target element is not found after the search interval becomes empty, return a message indicating that the element is not present in the array.

Example:

Consider an array of integers: [10, 20, 30, 40, 50, 60, 70, 80, 90].

Let's search for the element 60 using binary search:

1. Initialize `left = 0` and `right = 8` (indices of the first and last elements).
2. Calculate the middle index: `mid = (0 + 8) / 2 = 4`.
3. Compare the middle element (50) with the target element (60).
  - Since 50 is less than 60, update `left = mid + 1 = 5`.
4. Repeat the process:
  - Calculate the new middle index: `mid = (5 + 8) / 2 = 6`.
  - Compare the middle element (70) with the target element (60).
    - Since 70 is greater than 60, update `right = mid - 1 = 5`.
  - Calculate the new middle index: `mid = (5 + 5) / 2 = 5`.
  - Compare the middle element (60) with the target element (60).
    - Since they are equal, return the index 5 (search successful).

In this example, the element 60 is found at index 5 using binary search.

22 You

11. Search the number 500 from the given data using binary search technique. Illustrate the searching process. 100, 140, 200, 395, 410, 452, 491, 500, 607



ChatGPT

To search for the number 500 using the binary search technique, we will follow these steps:

1. Initialize pointers `left` and `right` to define the search interval.
2. Calculate the middle index of the search interval.
3. Compare the middle element with the target element.
4. Update the search interval based on the comparison result.
5. Repeat steps 2-4 until the target element is found or the search interval becomes empty.

Given data: 100, 140, 200, 395, 410, 452, 491, 500, 607

1. Initialize `left = 0` and `right = 8` (indices of the first and last elements).
2. Calculate the middle index: `mid = (0 + 8) / 2 = 4`.

3. Compare the middle element (410) with the target element (500).
  - Since 410 is less than 500, update `left = mid + 1 = 5`.
4. Calculate the new middle index: `mid = (5 + 8) / 2 = 6`.
5. Compare the middle element (491) with the target element (500).
  - Since 491 is less than 500, update `left = mid + 1 = 7`.
6. Calculate the new middle index: `mid = (7 + 8) / 2 = 7`.
7. Compare the middle element (500) with the target element (500).
  - Since they are equal, return the index 7 (search successful).

Illustration of the searching process:

sql

Step 1: Search Interval [0, 8]

```

-----
100 140 200 395 410 452 491 500 607
  ^                               ^
  |                               |
left                             right
mid = 4
Compare 410 with 500 (410 < 500), update left = 5
  
```

Step 2: Search Interval [5, 8]

```

-----
100 140 200 395 410 452 491 500 607
                        ^       ^
                        |       |
                     left    right
mid = 6
Compare 491 with 500 (491 < 500), update left = 7
  
```

Step 3: Search Interval [7, 8]

```

-----
100 140 200 395 410 452 491 500 607
                        ^       ^
                        |       |
                     left    right
mid = 7
Compare 500 with 500 (500 = 500), found at index 7
  
```

In this example, the number 500 is found at index 7 using binary search.

You

12. List out different hash methods and explain any three.

ChatGPT

Hashing is a technique used to map data of arbitrary size to fixed-size values. There are various hash methods, each with its own characteristics and applications. Here are some commonly used hash methods:

1. Division Method
2. Multiplication Method
3. Folding Method
4. Mid-Square Method
5. Digit Extraction Method

6. Universal Hashing
7. Cryptographic Hashing (e.g., MD5, SHA-1, SHA-256)

Now, let's explain three of these hash methods in detail:

### 1. Division Method:

- In the division method, the key is divided by the size of the hash table, and the remainder is used as the hash value.
- The formula to calculate the hash value is: `hash_value = key % table_size`.
- Example: Suppose we have a hash table of size 10, and we want to hash the key 27. Using the division method, the hash value would be `27 % 10 = 7`.

### 2. Multiplication Method:

- In the multiplication method, the key is multiplied by a constant (typically a fraction between 0 and 1), and the fractional part of the result is used as the hash value.
- The formula to calculate the hash value is: `hash_value = floor(table_size * (key * A % 1))`.
- A common choice for the constant A is the golden ratio `(sqrt(5) - 1) / 2`.
- Example: Let's hash the key 27 using a hash table of size 10. With  $A = (\sqrt{5} - 1) / 2$ , the hash value would be `floor(10 * (27 * 0.61803398875 % 1)) = 6`.

### 3. Folding Method:

- The folding method involves dividing the key into equal-sized parts, which are then added together to obtain the hash value.
- If the key has more digits than the hash table size, the resulting sum may need to be reduced by taking the modulo of the sum with the size of the hash table.
- Example: Suppose we have a hash table of size 10, and we want to hash the key 123456. We divide the key into two parts (12 and 34) and add them together to get 46. The hash value would be `46 % 10 = 6`.

Each of these hash methods has its advantages and disadvantages, and the choice of method depends on factors such as the nature of the data, the size of the hash table, and the desired distribution of hash values.

## 22 You

13. State and explain collision resolution techniques in hashing with example.

## ChatGPT

Collision resolution techniques are used in hashing to handle situations where two or more keys hash to the same index in the hash table. There are several collision resolution techniques, each with its own way of resolving collisions. Here are some common collision resolution techniques:

1. Separate Chaining
2. Open Addressing
  - a. Linear Probing

- b. Quadratic Probing
- c. Double Hashing

Let's explain each of these collision resolution techniques with examples:

### 1. Separate Chaining:

- In separate chaining, each bucket in the hash table is associated with a linked list or another data structure. When a collision occurs, the colliding keys are stored in the same bucket, but in separate linked lists.
- Example: Suppose we have a hash table with 10 buckets, and we use separate chaining to resolve collisions. If two keys, "apple" and "banana," hash to the same index (let's say index 3), they will be stored in separate linked lists within the same bucket. The structure might look like this:

rust

```
Index 3: -> "apple" -> "banana"
```

### 2. Open Addressing:

- In open addressing, when a collision occurs, an alternative location within the same hash table is sought to store the colliding key. This process involves probing, where additional locations are searched until an empty slot is found.

#### a. Linear Probing:

- In linear probing, when a collision occurs at index  $i$ , the algorithm searches for the next available slot by linearly probing the subsequent indices until an empty slot is found.
- Example: Suppose we have a hash table with 10 buckets, and we use linear probing to resolve collisions. If two keys, "apple" and "banana," hash to the same index (let's say index 3), linear probing will search for the next available slot. If index 4 is empty, "banana" will be stored at index 4.

#### b. Quadratic Probing:

- In quadratic probing, instead of linearly probing the subsequent indices, the algorithm probes at positions that follow a quadratic function until an empty slot is found.
- Example: Similar to linear probing, but instead of probing  $i+1, i+2, i+3, \dots$ , quadratic probing would probe at positions  $i+1^2, i+2^2, i+3^2, \dots$  until an empty slot is found.

#### c. Double Hashing:

- In double hashing, the algorithm uses a second hash function to determine the interval between probe sequences. This helps in avoiding clustering and distributing keys more uniformly.
- Example: Suppose we have a hash table with 10 buckets, and we use double hashing to resolve collisions. If two keys, "apple" and "banana," hash to the same index (let's say index 3), double hashing will use a second hash function to determine the interval between probe sequences, ensuring that the keys are stored at different locations.

Each collision resolution technique has its advantages and disadvantages, and the choice of technique depends on factors such as the nature of the data, the size of the hash table, and the

desired performance characteristics.

22 You

14. What is hashing? What are the qualities of a good hash function? Explain any two hash functions in detail.

ChatGPT

Hashing is a technique used in computer science to map data of arbitrary size to fixed-size values, typically for efficient storage, retrieval, and manipulation of data in data structures like hash tables. In hashing, a hash function is applied to a given input (or "key") to produce a unique hash value, which is used as an index or address to store or retrieve the corresponding data.

Qualities of a Good Hash Function:

1. **Uniformity:** A good hash function should distribute keys uniformly across the hash table, minimizing collisions and ensuring that each bucket in the hash table has approximately the same number of keys hashed to it.
2. **Deterministic:** The hash function should always produce the same hash value for the same input key. This ensures consistency in hashing operations.
3. **Efficiency:** The hash function should be computationally efficient to calculate, ensuring fast insertion, retrieval, and deletion operations in the hash table.
4. **Avalanche Effect:** A small change in the input key should result in a significant change in the hash value, causing a cascade effect (avalanche) throughout the hash table. This property enhances the security and reliability of hash functions.
5. **Minimal Collisions:** While collisions (two keys mapping to the same hash value) are inevitable, a good hash function should minimize collisions by evenly distributing keys across the hash table.

Two commonly used hash functions are:

#### 1. Division Method:

- In the division method, the key is divided by the size of the hash table, and the remainder is used as the hash value.
- The formula to calculate the hash value is: `hash_value = key % table_size`.
- Example: Suppose we have a hash table of size 10, and we want to hash the key 27. Using the division method, the hash value would be `27 % 10 = 7`.

#### 2. Multiplication Method:

- In the multiplication method, the key is multiplied by a constant (typically a fraction between 0 and 1), and the fractional part of the result is used as the hash value.
- The formula to calculate the hash value is: `hash_value = floor(table_size * (key * A % 1))`.

- A common choice for the constant A is the golden ratio  $\frac{\sqrt{5} - 1}{2}$ .
- Example: Let's hash the key 27 using a hash table of size 10. With  $A = (\sqrt{5} - 1) / 2$ , the hash value would be  $\text{floor}(10 * (27 * 0.61803398875 \% 1)) = 6$ .

These hash functions are simple and effective for many applications, providing good distribution and minimal collisions when implemented properly. However, there are many other hash functions tailored for specific use cases and requirements.

## You

15. Consider the hash table of size 10 and the result of inserting keys {5,18,55,78,35,15} using the hash function ( $f(\text{key}) = \text{key} \% 10$ ) using linear probing strategy.

## ChatGPT

To insert keys {5, 18, 55, 78, 35, 15} into a hash table of size 10 using the hash function  $f(\text{key}) = \text{key} \% 10$  with linear probing strategy, we'll follow these steps:

1. Hash each key using the given hash function.
2. If the hash index is already occupied, probe linearly until an empty slot is found.
3. Insert the key into the first available slot.

Let's illustrate this process step by step:

### 1. Insert 5:

- Hash value:  $f(5) = 5 \% 10 = 5$
- Index 5 is empty, so insert 5 at index 5.

### 2. Insert 18:

- Hash value:  $f(18) = 18 \% 10 = 8$
- Index 8 is empty, so insert 18 at index 8.

### 3. Insert 55:

- Hash value:  $f(55) = 55 \% 10 = 5$
- Index 5 is already occupied by 5.
- Linear probing: Start probing linearly from the next index.
- Index 6 is empty, so insert 55 at index 6.

### 4. Insert 78:

- Hash value:  $f(78) = 78 \% 10 = 8$
- Index 8 is already occupied by 18.
- Linear probing: Continue probing linearly.
- Index 9 is empty, so insert 78 at index 9.

### 5. Insert 35:

- Hash value:  $f(35) = 35 \% 10 = 5$

- Index 5 is already occupied by 5 and 55.
- Linear probing: Continue probing linearly.
- Index 6 is already occupied by 55.
- Linear probing: Continue probing linearly.
- Index 7 is empty, so insert 35 at index 7.

## 6. Insert 15:

- Hash value:  $f(15) = 15\%10 = 5$
- Index 5 is already occupied by 5, 55, and 35.
- Linear probing: Continue probing linearly.
- Index 6 is already occupied by 55.
- Linear probing: Continue probing linearly.
- Index 7 is already occupied by 35.
- Linear probing: Continue probing linearly.
- Index 8 is already occupied by 18.
- Linear probing: Continue probing linearly.
- Index 9 is already occupied by 78.
- Linear probing: Wrap around to the beginning of the table.
- Index 0 is empty, so insert 15 at index 0.

Now, the hash table after inserting keys {5, 18, 55, 78, 35, 15} using linear probing strategy would look like this:

markdown

Index:	0	1	2	3	4	5	6	7	8	9
Key:	15			35		5	55	78	18	

Note: This is a simplified example for illustrative purposes. In practice, hash tables may use more sophisticated probing techniques and handle collisions differently.

22 You

16. Explain Asymptotic Notations in detail.

 ChatGPT

Asymptotic notations are mathematical tools used in computer science to describe the time complexity and space complexity of algorithms. They provide a concise way to represent the behavior of algorithms as the size of the input approaches infinity. Asymptotic notations are particularly useful for analyzing the efficiency of algorithms and comparing their performance.

There are three main asymptotic notations commonly used:

1. Big O notation ( $O$ )
2. Big Omega notation ( $\Omega$ )
3. Big Theta notation ( $\Theta$ )

Let's explain each of these notations in detail:



## 1. Big O notation (O):

- Big O notation represents the upper bound or worst-case time complexity of an algorithm. It describes the maximum amount of time an algorithm takes to execute for a given input size.
- Formally, a function  $f(n)$  is said to be  $O(g(n))$  if there exist constants  $c$  and  $n_0$  such that  $0 \leq f(n) \leq c \cdot g(n)$  for all  $n \geq n_0$ .
- Intuitively,  $f(n)$  is  $O(g(n))$  if its growth rate is bounded above by the growth rate of  $g(n)$  for sufficiently large values of  $n$ .
- Example: If the time complexity of an algorithm is  $O(n^2)$ , it means that the worst-case time taken by the algorithm grows quadratically with the size of the input.

## 2. Big Omega notation ( $\Omega$ ):

- Big Omega notation represents the lower bound or best-case time complexity of an algorithm. It describes the minimum amount of time an algorithm takes to execute for a given input size.
- Formally, a function  $f(n)$  is said to be  $\Omega(g(n))$  if there exist constants  $c$  and  $n_0$  such that  $0 \leq c \cdot g(n) \leq f(n)$  for all  $n \geq n_0$ .
- Intuitively,  $f(n)$  is  $\Omega(g(n))$  if its growth rate is bounded below by the growth rate of  $g(n)$  for sufficiently large values of  $n$ .
- Example: If the time complexity of an algorithm is  $\Omega(n)$ , it means that the best-case time taken by the algorithm grows linearly with the size of the input.

## 3. Big Theta notation ( $\Theta$ ):

- Big Theta notation represents both the upper and lower bounds or tight bounds of an algorithm's time complexity. It describes the exact behavior of an algorithm for a given input size.
- Formally, a function  $f(n)$  is said to be  $\Theta(g(n))$  if there exist constants  $c_1$ ,  $c_2$ , and  $n_0$  such that  $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$  for all  $n \geq n_0$ .
- Intuitively,  $f(n)$  is  $\Theta(g(n))$  if its growth rate is bounded both above and below by the growth rate of  $g(n)$  for sufficiently large values of  $n$ .
- Example: If the time complexity of an algorithm is  $\Theta(n)$ , it means that the algorithm's time taken grows linearly with the size of the input, and this growth is not faster or slower than linear.

In summary, asymptotic notations provide a powerful framework for analyzing and comparing the efficiency of algorithms by focusing on their growth rates as the input size increases. They help in understanding how algorithms behave in practice and making informed decisions about algorithm selection and optimization.

 **You**

17. Define time complexity and space complexity of algorithm.

 **ChatGPT**

Time complexity and space complexity are two important metrics used to analyze the efficiency of algorithms:

### 1. Time Complexity:

- Time complexity of an algorithm is a measure of the amount of time taken by an algorithm to execute as a function of the length of the input. It describes how the runtime of an algorithm increases with the size of the input.
- Time complexity is typically expressed using asymptotic notation, such as Big O notation ( $O$ ), which provides an upper bound on the growth rate of the algorithm's runtime.
- Example: If the time complexity of an algorithm is  $O(n^2)$ , it means that the runtime of the algorithm grows quadratically with the size of the input. As the input size doubles, the runtime quadruples.

### 2. Space Complexity:

- Space complexity of an algorithm is a measure of the amount of memory space required by an algorithm to execute as a function of the length of the input. It describes how the memory usage of an algorithm increases with the size of the input.
- Space complexity is also typically expressed using asymptotic notation, such as Big O notation ( $O$ ), which provides an upper bound on the growth rate of the algorithm's memory usage.
- Example: If the space complexity of an algorithm is  $O(n)$ , it means that the algorithm's memory usage grows linearly with the size of the input. As the input size doubles, the memory usage also doubles.

In summary, time complexity and space complexity are important metrics for understanding the efficiency and resource requirements of algorithms. They help in comparing different algorithms, identifying bottlenecks, and making informed decisions about algorithm design and optimization.

### You

18. Explain the Big-Oh computation for each of the following control structures. (a) If-then-else (b) for loop (c) Inner outer for loop

### ChatGPT

Big O notation provides a way to represent the upper bound or worst-case time complexity of algorithms. Let's compute the Big O notation for each of the following control structures:

#### (a) If-then-else:

The time complexity of an if-then-else statement depends on the complexity of the statements inside each branch and the probability of each branch being executed.

- **Best Case:** The best-case time complexity occurs when the condition of the if statement evaluates to false, and only the statements inside the if or else branch are executed. In this case, the time complexity is the time complexity of the executed statements.

- **Worst Case:** The worst-case time complexity occurs when the condition of the if statement evaluates to true, and both the if and else branches are executed. In this case, the time complexity is the maximum of the time complexities of the statements inside the if and else branches.

Example:

python

```
if condition:
    # Statements inside if branch
else:
    # Statements inside else branch
```

- **Best Case Time Complexity:** The time complexity of the executed statements inside the if or else branch.
- **Worst Case Time Complexity:** The maximum of the time complexities of the statements inside the if and else branches.

#### (b) For Loop:

The time complexity of a for loop depends on the number of iterations and the complexity of the statements inside the loop.

- **Best Case:** The best-case time complexity occurs when the loop is not executed at all, typically when the loop condition is false initially. In this case, the time complexity is constant.
- **Worst Case:** The worst-case time complexity occurs when the loop is executed the maximum number of times, typically when the loop condition is true for all iterations. In this case, the time complexity is the product of the number of iterations and the time complexity of the statements inside the loop.

Example:

python

```
for i in range(n):
    # Statements inside the loop
```

- **Best Case Time Complexity:**  $O(1)$  if the loop is not executed.
- **Worst Case Time Complexity:**  $O(n)$ , where  $n$  is the number of iterations.

#### (c) Inner-Outer For Loop:

The time complexity of nested for loops depends on the number of iterations of each loop and the complexity of the statements inside the innermost loop.

- **Best Case:** The best-case time complexity occurs when the innermost loop is not executed at all. In this case, the time complexity is constant.

- **Worst Case:** The worst-case time complexity occurs when all loops are executed the maximum number of times. In this case, the time complexity is the product of the number of iterations of each loop and the time complexity of the statements inside the innermost loop.

Example:

python

```
for i in range(n):
    for j in range(m):
        # Statements inside the innermost loop
```

- **Best Case Time Complexity:**  $O(1)$  if the innermost loop is not executed.
- **Worst Case Time Complexity:**  $O(n * m)$ , where  $n$  is the number of iterations of the outer loop and  $m$  is the number of iterations of the inner loop.

22 You

19. Explain Divide and Conquer approach with suitable example.



ChatGPT

The Divide and Conquer approach is a problem-solving strategy that involves breaking down a problem into smaller, more manageable subproblems, solving each subproblem independently, and then combining the solutions to the subproblems to solve the original problem. This technique is often used to solve complex problems efficiently by reducing them to simpler instances.

The Divide and Conquer approach typically follows three steps:

1. **Divide:** Break the problem into smaller subproblems. This step usually involves dividing the problem into two or more smaller instances of the same problem.
2. **Conquer:** Solve each subproblem recursively. If the subproblems are small enough, solve them directly using a base case or simple algorithm.
3. **Combine:** Merge the solutions of the subproblems to obtain the solution to the original problem.

A suitable example to illustrate the Divide and Conquer approach is the "Merge Sort" algorithm for sorting a list of elements.

### Merge Sort Algorithm:

Merge Sort is a classic example of the Divide and Conquer approach used for sorting arrays or lists efficiently.

1. **Divide:** Divide the array into two halves.
2. **Conquer:** Recursively sort the two halves.
3. **Combine:** Merge the sorted halves to produce the final sorted array.

Example:

Let's consider an unsorted array: [38, 27, 43, 3, 9, 82, 10].

1. **Divide:** Divide the array into two halves: [38, 27, 43, 3] and [9, 82, 10].
2. **Conquer:** Recursively sort each half:
  - For the first half: [38, 27, 43, 3]
    - Divide: [38, 27] and [43, 3]
    - Conquer: Recursively sort each half: [27, 38] and [3, 43]
    - Combine: Merge the sorted halves: [3, 27, 38, 43]
  - For the second half: [9, 82, 10]
    - Divide: [9] and [82, 10]
    - Conquer: Recursively sort each half: [9] and [10, 82]
    - Combine: Merge the sorted halves: [9, 10, 82]
3. **Combine:** Merge the two sorted halves: [3, 9, 10, 27, 38, 43, 82].

Now, the original unsorted array [38, 27, 43, 3, 9, 82, 10] has been sorted using the Divide and Conquer approach with the Merge Sort algorithm.

Merge Sort has a time complexity of  $O(n \log n)$  in the worst-case scenario, making it one of the most efficient sorting algorithms for large datasets.

 **You**

20. Discuss maximum sub array sum problem with example.

 **ChatGPT**

The maximum subarray sum problem is a classic algorithmic problem that seeks to find the contiguous subarray within a one-dimensional array of numbers that has the largest sum. This problem has applications in various fields, including data analysis, finance, and algorithm design.

Given an array of integers, the maximum subarray sum problem can be stated as follows:

**Input:** An array  $A$  of  $n$  integers  $a_1, a_2, \dots, a_n$ .

**Output:** The maximum sum of a contiguous subarray within the array  $A$ .

Example:

Consider the array: [-2, 1, -3, 4, -1, 2, 1, -5, 4]

To solve the maximum subarray sum problem, we can use the Kadane's algorithm, which is an efficient algorithm with a time complexity of  $O(n)$ .

Kadane's algorithm works by iterating through the array and maintaining two variables: *max\_ending\_here* and *max\_so\_far*.

1. **max\_ending\_here:** Represents the maximum sum of a subarray ending at the current index.
2. **max\_so\_far:** Represents the maximum sum of any subarray seen so far.

The algorithm iterates through the array, updating *max\_ending\_here* and *max\_so\_far* at each index:

- At each index, *max\_ending\_here* is updated as the maximum of:
  - The current element  $a_i$
  - The sum of the current element and the maximum sum of a subarray ending at the previous index  $max\_ending\_here + a_i$
- *max\_so\_far* is updated as the maximum of:
  - The current value of *max\_ending\_here*
  - The current value of *max\_so\_far*

After iterating through the array, *max\_so\_far* will hold the maximum subarray sum.

Let's apply Kadane's algorithm to the example array:

- **Iteration 1:** [-2]
  - $max\_ending\_here = -2$
  - $max\_so\_far = -2$
- **Iteration 2:** [1]
  - $max\_ending\_here = 1$  (max ending here)
  - $max\_so\_far = 1$
- **Iteration 3:** [-3]
  - $max\_ending\_here = -2$  (max ending here)
  - $max\_so\_far = 1$
- **Iteration 4:** [4]
  - $max\_ending\_here = 4$  (max ending here)
  - $max\_so\_far = 4$
- **Iteration 5:** [-1]
  - $max\_ending\_here = 3$  (max ending here)
  - $max\_so\_far = 4$
- **Iteration 6:** [2]
  - $max\_ending\_here = 5$  (max ending here)
  - $max\_so\_far = 5$
- **Iteration 7:** [1]
  - $max\_ending\_here = 6$  (max ending here)
  - $max\_so\_far = 6$
- **Iteration 8:** [-5]
  - $max\_ending\_here = 1$  (max ending here)
  - $max\_so\_far = 6$
- **Iteration 9:** [4]
  - $max\_ending\_here = 5$  (max ending here)
  - $max\_so\_far = 6$

After iterating through the array, the maximum subarray sum is 6, which corresponds to the subarray [4, -1, 2, 1].

21. Discuss red – black tree.

### ChatGPT

A red-black tree is a balanced binary search tree that maintains balance through a set of rules enforced during insertions and deletions. It is named "red-black" because each node in the tree is assigned a color, either red or black, and the tree must satisfy several properties to ensure balance.

Properties of a red-black tree:

1. Every node is either red or black.
2. The root node is black.
3. Red nodes cannot have red children (i.e., no two adjacent red nodes can appear in a row).
4. Every path from a node to its descendant NULL nodes must have the same number of black nodes. This property ensures that the longest path from the root to a leaf node is no more than twice the length of the shortest path, maintaining the balance of the tree.

Red-black trees achieve a balance between the advantages of a height-balanced tree (such as AVL tree) and the simplicity of a binary search tree. The maximum height of a red-black tree with  $n$  nodes is  $2 \log_2(n + 1)$ , which ensures efficient search, insertion, and deletion operations with a worst-case time complexity of  $O(\log n)$ .

Operations on red-black trees:

1. **Insertion:** After inserting a new node into the tree, the tree may violate the red-black tree properties. To restore balance, various cases are considered, including rotations and color changes. The insertion algorithm ensures that the tree remains balanced while inserting the new node.
2. **Deletion:** Similar to insertion, deleting a node from the tree may violate the red-black tree properties. The deletion algorithm handles cases where deleting a node affects the balance of the tree, ensuring that the resulting tree remains balanced.
3. **Search:** Searching for a key in a red-black tree follows the same process as in a binary search tree, starting from the root and recursively searching the appropriate subtree based on the key's value.

Red-black trees are widely used in various applications, including database indexing, associative arrays, and compiler symbol tables, due to their efficient performance and balanced structure. However, their implementation can be more complex compared to simpler binary search trees, such as binary search trees and AVL trees, due to the additional rules for maintaining balance.

### You

22. Write the properties of red – black tree.

### ChatGPT

The properties of a red-black tree are as follows:

1. **Node Color:** Every node in the tree is either red or black.
2. **Root Property:** The root node is always black.
3. **Red Property:** Red nodes cannot have red children. In other words, no two adjacent nodes in the tree can be red.
4. **Black-Height Property:** Every path from a node to its descendant NULL nodes (i.e., the number of black nodes along the path) must have the same number of black nodes. This property ensures that the longest path from the root to a leaf node is no more than twice the length of the shortest path, maintaining the balance of the tree.

These properties ensure that a red-black tree remains balanced after insertions and deletions, guaranteeing a maximum height of  $2 \log_2(n + 1)$ , where  $n$  is the number of nodes in the tree. As a result, red-black trees provide efficient search, insertion, and deletion operations with a worst-case time complexity of  $O(\log n)$ .

 **You**

23. Explain insertion and deletion to/from red-black tree.

 **ChatGPT**

Insertion and deletion in a red-black tree are operations that maintain the balance and properties of the tree while adding or removing nodes. These operations involve adjusting the colors of nodes and performing rotations to ensure that the red-black tree properties are preserved.

#### Insertion:

1. **Insertion as a Binary Search Tree:** Start by inserting the new node into the red-black tree following the rules of a binary search tree (BST). The new node is initially colored red.
2. **Fix Violations:** After insertion, if the red-black tree properties are violated, fix them by performing a series of rotations and color changes. There are several cases to consider:
  - a. If the parent of the newly inserted node is black, then no violation occurs, and the tree remains balanced.
  - b. If the parent of the newly inserted node is red, there are three sub-cases to consider:
    - Case 1: If the uncle (sibling of the parent's parent) is red, then recolor the parent, uncle, and grandparent appropriately to maintain balance.
    - Case 2: If the uncle is black and the newly inserted node is a right child, perform a left rotation at the parent and treat the parent as the new node.
    - Case 3: If the uncle is black and the newly inserted node is a left child, perform a right rotation at the grandparent, recolor the parent and grandparent, and perform a left rotation at the grandparent.



3. **Adjust Root Color:** Ensure that the root node remains black after all adjustments.

## Deletion:

1. **Deletion as a Binary Search Tree:** Start by deleting the node from the red-black tree following the rules of a binary search tree (BST). If the node being deleted has two children, replace it with its successor (the smallest node in its right subtree) or predecessor (the largest node in its left subtree).
2. **Fix Violations:** After deletion, if the red-black tree properties are violated, fix them by performing a series of rotations and color changes. There are several cases to consider:
  - a. If the sibling of the deleted node is red, perform a rotation to change the sibling's color to black and the parent's color to red, then continue with the appropriate cases.
  - b. If the sibling of the deleted node is black and has at least one red child, perform rotations and color changes to balance the tree.
3. **Adjust Root Color:** Ensure that the root node remains black after all adjustments.

Insertion and deletion in red-black trees are complex operations due to the need to maintain the balance and properties of the tree. However, with careful implementation following the specified cases, these operations can be performed efficiently while preserving the benefits of a balanced binary search tree.

## You

24. Explain rotations of red-black tree.

## ChatGPT

Rotations are fundamental operations in red-black trees that help maintain the balance and properties of the tree. There are two types of rotations: left rotation and right rotation. These rotations are used during insertion, deletion, and balancing operations to ensure that the red-black tree properties are preserved.

### 1. Left Rotation:

Left rotation is used to move a node down in the tree while promoting one of its children. It restructures the tree to ensure that the left child of the current node becomes its parent, and the current node becomes the right child of its former left child.

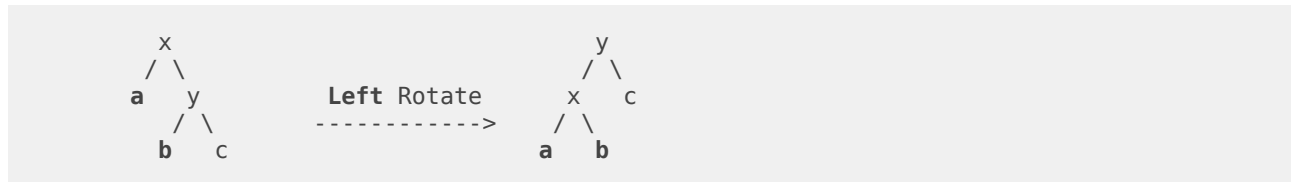
The steps to perform a left rotation are as follows:

- a. Let  $x$  be the node to rotate.
- b. Set  $y$  as the right child of  $x$ .
- c. Set the left child of  $x$  to be the right child of  $y$ .

- d. If  $y$  has a left child, update its parent to be  $x$ .
- e. Set the parent of  $y$  to be the parent of  $x$ .
- f. If the parent of  $x$  is null, set  $y$  to be the new root of the tree. Otherwise, if  $x$  is a left child, set  $y$  as the left child of the parent of  $x$ , otherwise set  $y$  as the right child of the parent of  $x$ .

Example:

CSS



## 2. Right Rotation:

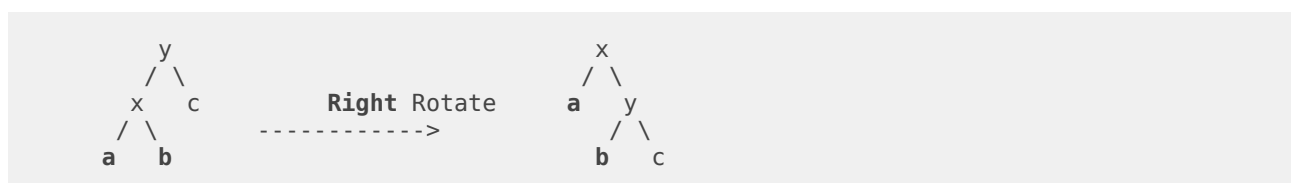
Right rotation is the mirror operation of left rotation. It is used to move a node down in the tree while promoting one of its children. It restructures the tree to ensure that the right child of the current node becomes its parent, and the current node becomes the left child of its former right child.

The steps to perform a right rotation are as follows:

- a. Let  $y$  be the node to rotate.
- b. Set  $x$  as the left child of  $y$ .
- c. Set the right child of  $y$  to be the left child of  $x$ .
- d. If  $x$  has a right child, update its parent to be  $y$ .
- e. Set the parent of  $x$  to be the parent of  $y$ .
- f. If the parent of  $y$  is null, set  $x$  to be the new root of the tree. Otherwise, if  $y$  is a right child, set  $x$  as the right child of the parent of  $y$ , otherwise set  $x$  as the left child of the parent of  $y$ .

Example:

CSS



Rotations are essential for maintaining the balance and properties of red-black trees. They are used in conjunction with color changes to ensure that the red-black tree remains balanced after insertions and deletions.

22 You

25. List out elements of greedy strategy.

 ChatGPT

The greedy strategy is a problem-solving approach that makes the locally optimal choice at each step with the hope of finding a globally optimal solution. While the greedy strategy does not guarantee an optimal solution for every problem, it can be useful for solving certain types of problems efficiently. The key elements of the greedy strategy include:

1. **Greedy Choice Property:** At each step, make the locally optimal choice that seems the best at the moment. This means selecting the option that appears to be the best without considering the consequences of that choice on future steps.
2. **Optimal Substructure:** The problem can be solved by making a series of choices, with each choice leading to a subproblem that can be solved optimally. In other words, the problem can be broken down into smaller subproblems, and the optimal solution to the overall problem can be constructed from the optimal solutions to the subproblems.
3. **Greedy Algorithm:** Develop an algorithm that implements the greedy strategy by making locally optimal choices at each step. The algorithm iteratively selects the best available option without reconsidering previous choices.
4. **Proof of Correctness:** Prove that the greedy algorithm produces an optimal solution by demonstrating that the locally optimal choices lead to a globally optimal solution. This often involves showing that the solution satisfies the optimal substructure property.
5. **Applicability:** Identify problems where the greedy strategy is applicable and likely to produce an optimal or near-optimal solution. Greedy algorithms are most effective when the problem exhibits both the greedy choice property and optimal substructure.
6. **Performance Analysis:** Analyze the time and space complexity of the greedy algorithm to understand its efficiency. Greedy algorithms often have a linear or polynomial time complexity, making them suitable for solving large-scale problems efficiently.

By following these elements, the greedy strategy can be effectively applied to various optimization problems, such as scheduling, shortest path, and minimum spanning tree problems, to find approximate solutions quickly. However, it's important to note that the greedy strategy may not always yield the globally optimal solution and may require careful consideration of problem constraints and characteristics.

22 You

26. Explain greedy algorithms with example.

 ChatGPT

Greedy algorithms are problem-solving strategies that make locally optimal choices at each step with the aim of finding a globally optimal solution. These algorithms are particularly useful for optimization problems where making the best decision at each step leads to an optimal overall solution. Greedy algorithms are straightforward to implement and often have efficient time complexity, making them suitable for solving large-scale problems. However, it's important to note that greedy algorithms may not always produce the globally optimal solution, as they do not consider the consequences of their choices on future steps.

Let's illustrate the concept of greedy algorithms with an example:

### Example: Fractional Knapsack Problem

**Problem Statement:** You are given a set of items, each with a weight  $w_i$  and a value  $v_i$ . You have a knapsack that can carry a maximum weight  $W$ . Your goal is to select a subset of items to maximize the total value in the knapsack, subject to the constraint that the total weight of selected items does not exceed  $W$ . Moreover, you are allowed to take fractional amounts of items (i.e., you can take a fraction of an item if it maximizes the total value).

**Solution using Greedy Algorithm:**

1. **Calculate Value-to-Weight Ratio:** Compute the value-to-weight ratio  $\frac{v_i}{w_i}$  for each item.
2. **Sort Items by Ratio:** Sort the items in descending order of their value-to-weight ratios. This ensures that the items with the highest ratio (i.e., the best value for their weight) are considered first.
3. **Fill Knapsack Greedily:** Start with an empty knapsack. Iterate through the sorted list of items, adding items to the knapsack greedily until the knapsack is full. For each item, add as much of it as possible (up to its entire weight), and update the remaining capacity of the knapsack.
4. **Calculate Total Value:** Calculate the total value of the items in the knapsack.

**Example:**

Consider the following set of items:

- Item 1: Weight = 10, Value = 60
- Item 2: Weight = 20, Value = 100
- Item 3: Weight = 30, Value = 120

And a knapsack with a maximum weight capacity of 50.

1. Compute the value-to-weight ratio for each item:

- Item 1:  $\frac{60}{10} = 6$
- Item 2:  $\frac{100}{20} = 5$
- Item 3:  $\frac{120}{30} = 4$

2. Sort items by ratio (descending order):

- Item 1: Ratio = 6
- Item 2: Ratio = 5

- Item 3: Ratio = 4

### 3. Fill the knapsack greedily:

- Take Item 1 completely (weight = 10, value = 60)
- Take 20/30 of Item 2 (weight = 20, value =  $100 * 20/30 = 66.67$ )
- Total weight =  $10 + 20 = 30$  (remaining capacity = 20)

### 4. Total value = $60 + 66.67 = 126.67$

In this example, the greedy algorithm selects items based on their value-to-weight ratios, resulting in a total value of 126.67. While this solution may not be globally optimal, it demonstrates how the greedy strategy can be applied to solve optimization problems efficiently.

## 22 You

27. How to augment a data structure?

## ChatGPT

Augmenting a data structure involves enhancing its functionality or adding additional information to it in order to support specific operations or queries efficiently. This process typically involves modifying the underlying data structure and associated algorithms to accommodate the new requirements. Augmentation can improve the performance or enable new functionalities without significantly altering the original structure.

Here are the general steps to augment a data structure:

1. **Identify Requirements:** Determine the specific requirements or functionalities that need to be added to the data structure. This may involve analyzing the operations or queries that are frequently performed on the data structure and identifying areas for improvement.
2. **Choose the Data Structure:** Select a suitable data structure that can accommodate the new requirements efficiently. Depending on the nature of the problem and the desired functionalities, different data structures may be appropriate.
3. **Define Additional Information:** Determine the additional information or metadata that needs to be stored along with the existing data in the data structure to support the new functionalities. This could include extra fields, pointers, counters, or any other relevant information.
4. **Modify Data Structure Operations:** Modify the operations or algorithms associated with the data structure to incorporate the new functionalities. This may involve updating existing operations or adding new ones to handle the augmented data structure appropriately.
5. **Update Insertion and Deletion Procedures:** If necessary, update the procedures for inserting and deleting elements from the data structure to ensure that the additional information is maintained correctly.

6. **Ensure Correctness and Efficiency:** Test the augmented data structure rigorously to ensure that it behaves correctly and efficiently under various scenarios. This may involve conducting unit tests, stress tests, and performance evaluations to validate the modifications.
7. **Documentation and Maintenance:** Document the changes made to the data structure, including the rationale behind the augmentation, the modified operations, and any additional information stored. Ensure that the documentation is comprehensive and understandable to facilitate future maintenance and updates.
8. **Iterative Improvement:** Continuously monitor and evaluate the augmented data structure in practice. If necessary, further refine or optimize the implementation based on performance feedback and evolving requirements.

Examples of augmentation of data structures include:

- Adding parent pointers to nodes in a tree data structure to support efficient traversal or update operations.
- Maintaining additional counters or metadata in a priority queue to support operations such as finding the minimum or maximum element.
- Storing extra information in a graph data structure to facilitate shortest path calculations or cycle detection.

Overall, augmenting a data structure involves a careful balance between functionality, efficiency, and maintainability to meet the specific requirements of the problem at hand.

22 **You**

28. Explain dynamic programming in detail.

 **ChatGPT**

Dynamic programming is a powerful problem-solving technique used to solve problems by breaking them down into simpler subproblems and solving each subproblem only once. It is particularly useful for optimization problems, where the goal is to find the best solution among many possible solutions. Dynamic programming is based on the idea of memoization, which involves storing the solutions to subproblems to avoid redundant calculations.

Here are the key components and steps involved in dynamic programming:

1. **Identify Overlapping Subproblems:** The first step in applying dynamic programming is to identify the presence of overlapping subproblems in the given problem. Overlapping subproblems occur when the same subproblem is solved multiple times in the process of solving the larger problem.
2. **Formulate Recursive Solution:** Once overlapping subproblems are identified, formulate a recursive solution to the problem. Break down the problem into smaller subproblems and express the solution to the larger problem in terms of solutions to its subproblems.

3. **Memoization or Tabulation:** Implement memoization or tabulation to store the solutions to subproblems. Memoization involves storing the results of solved subproblems in a data structure (such as an array or hash table) so that they can be reused when needed. Tabulation involves building a table bottom-up, starting with the smallest subproblems and iteratively solving larger subproblems based on the solutions to smaller subproblems.
4. **Optimal Substructure:** Ensure that the problem exhibits optimal substructure, meaning that the optimal solution to the larger problem can be constructed from the optimal solutions to its subproblems. This property allows dynamic programming to work effectively by combining solutions to subproblems to solve the larger problem optimally.
5. **Reconstruct Solution (Optional):** After solving the larger problem using dynamic programming, reconstruct the solution if needed. This involves tracing back through the memoized solutions to determine the optimal sequence of decisions or actions that led to the optimal solution.

Dynamic programming can be further classified into two approaches:

- **Top-Down Approach (Memoization):** In this approach, the recursive solution is implemented with memoization to store the solutions to subproblems. The memoization table is typically implemented using recursion with memoization or by using data structures like arrays or hash tables.
- **Bottom-Up Approach (Tabulation):** In this approach, the solutions to subproblems are computed iteratively starting from the smallest subproblems and building up to the larger subproblems. This approach is usually more efficient than the top-down approach as it avoids the overhead of function calls and recursion.

Dynamic programming is widely used to solve a variety of problems, including optimization problems, shortest path problems, sequence alignment problems, and more. It offers significant performance improvements over naive recursive solutions by avoiding redundant calculations and exploiting overlapping subproblems.

## 22 You

29. Write a Dijkstra's algorithm for finding shortest path. (Practice for all the examples discussed in classroom)

## ChatGPT

Sure, here's an implementation of Dijkstra's algorithm in Python:

python

```
import heapq

def dijkstra(graph, start):
    # Initialize distances to all nodes as infinity
    distances = {node: float('inf') for node in graph}
    # Initialize distance to start node as 0
    distances[start] = 0
```

```
# Priority queue to store nodes with their distances
priority_queue = [(0, start)]

while priority_queue:
    # Pop node with the smallest distance from priority queue
    current_distance, current_node = heapq.heappop(priority_queue)

    # If the current distance is greater than the distance in the distances dictionary,
    # it means we've already found a shorter path to this node, so we skip it
    if current_distance > distances[current_node]:
        continue

    # Explore neighbors of the current node
    for neighbor, weight in graph[current_node].items():
        distance = current_distance + weight
        # If the distance to the neighbor through the current node is shorter than the
        # previously calculated distance, update it and add the neighbor to the priority
        if distance < distances[neighbor]:
            distances[neighbor] = distance
            heapq.heappush(priority_queue, (distance, neighbor))

return distances

# Example graph
graph = {
    'A': {'B': 5, 'C': 3},
    'B': {'A': 5, 'C': 2, 'D': 1},
    'C': {'A': 3, 'B': 2, 'D': 4, 'E': 5},
    'D': {'B': 1, 'C': 4, 'E': 6, 'F': 7},
    'E': {'C': 5, 'D': 6, 'F': 8},
    'F': {'D': 7, 'E': 8}
}

# Test the algorithm
start_node = 'A'
shortest_distances = dijkstra(graph, start_node)
print("Shortest distances from node", start_node + ":")
for node, distance in shortest_distances.items():
    print("Node:", node, "Distance:", distance)
```

This implementation of Dijkstra's algorithm uses a priority queue (implemented using `heapq` module) to efficiently explore nodes with the smallest distance first. It iterates over each node and its neighbors, updating the distances dictionary with the shortest distances found so far. Finally, it returns the shortest distances from the start node to all other nodes in the graph.

## 22 You

30. Explain each rotation with example of AVL tree.

## ChatGPT

In an AVL tree, rotations are key operations used to maintain the balance factor property of the tree, ensuring that the tree remains balanced after insertions and deletions. There are four types of rotations: left rotation, right rotation, left-right rotation (also known as double rotation), and right-left rotation (also known as double rotation). Each rotation is designed to restore balance to the tree while preserving the binary search tree property.

### 1. Left Rotation:

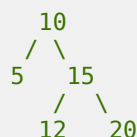


In a left rotation, a node becomes the left child of its right child. This operation is performed when a node's right subtree is taller by two levels than its left subtree. A left rotation is used to move a node down in the tree and balance the tree's height.

Example:

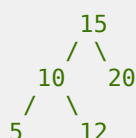
Consider the following AVL tree:

markdown



Performing a left rotation on node 10 results in:

markdown



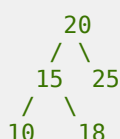
## 2. Right Rotation:

In a right rotation, a node becomes the right child of its left child. This operation is performed when a node's left subtree is taller by two levels than its right subtree. A right rotation is used to move a node down in the tree and balance the tree's height.

Example:

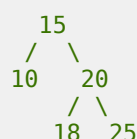
Consider the following AVL tree:

markdown



Performing a right rotation on node 20 results in:

markdown



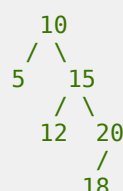
## 3. Left-Right Rotation (Double Rotation):

A left-right rotation is a combination of a left rotation followed by a right rotation. This operation is performed when a node's left child has a right subtree that is taller than its left subtree. A left-right rotation is used to balance the tree while preserving the binary search tree property.

Example:

Consider the following AVL tree:

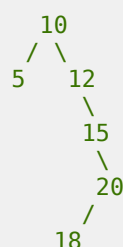
markdown



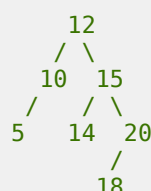
Performing a left-right rotation on node 10 involves:

markdown

- Left rotation on node 15:



- Right rotation on node 10:



#### 4. Right-Left Rotation (Double Rotation):

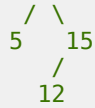
A right-left rotation is a combination of a right rotation followed by a left rotation. This operation is performed when a node's right child has a left subtree that is taller than its right subtree. A right-left rotation is used to balance the tree while preserving the binary search tree property.

Example:

Consider the following AVL tree:

markdown





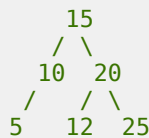
Performing a right-left rotation on node 20 involves:

markdown

- Right rotation on node 10:



- Left rotation on node 20:



These rotations play a crucial role in maintaining the balance factor property of AVL trees, ensuring that the height of the tree remains logarithmic and providing efficient search, insertion, and deletion operations with a time complexity of  $O(\log n)$ .

22 You

31. Explain BFS and DFS algorithms for graph traversal.



ChatGPT

Breadth-First Search (BFS) and Depth-First Search (DFS) are two fundamental algorithms used for traversing graphs and exploring or searching for nodes or vertices. Both algorithms are applicable to both directed and undirected graphs.

### 1. Breadth-First Search (BFS):

Breadth-First Search explores a graph level by level, starting from a selected source vertex. It explores all the neighbor vertices at the present level before moving to the vertices at the next level. BFS uses a queue data structure to keep track of vertices to be explored.

Algorithm:

1. Start from a selected source vertex and mark it as visited.
2. Enqueue the source vertex into a queue.
3. While the queue is not empty:
  - Dequeue a vertex from the queue.
  - Visit the dequeued vertex and process it.
  - Enqueue all its unvisited neighbor vertices into the queue and mark them as visited.

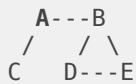
4. Repeat step 3 until the queue is empty.

BFS is particularly useful for finding the shortest path between two vertices in an unweighted graph.

Example:

Consider the following undirected graph:

CSS



Starting from vertex A, BFS traversal would visit the vertices in the order: A, B, C, D, E.

## 2. Depth-First Search (DFS):

Depth-First Search explores a graph by going as deep as possible along each branch before backtracking. It starts from a selected source vertex and explores as far as possible along each branch before backtracking. DFS uses a stack data structure (or recursion) to keep track of vertices to be explored.

Algorithm:

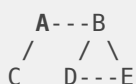
1. Start from a selected source vertex and mark it as visited.
2. Visit the source vertex and process it.
3. Recursively visit all unvisited neighbor vertices of the source vertex.
4. Repeat steps 2 and 3 for each unvisited neighbor vertex, until all vertices are visited.

DFS is useful for topological sorting, finding connected components, and detecting cycles in graphs.

Example:

Consider the following undirected graph:

CSS



Starting from vertex A, DFS traversal would visit the vertices in the order: A, B, D, E, C.

Both BFS and DFS are essential graph traversal algorithms and have various applications in graph theory, computer networks, social network analysis, and more. The choice between BFS and DFS depends on the specific requirements of the problem and the characteristics of the graph being traversed.

31. Determine BFS and DFS traversals for graph. (Practice for all the examples discussed in classroom)

### ChatGPT

Sure, let's determine both BFS and DFS traversals for the following example graph discussed in the classroom:

mathematica



This is a simple directed graph with vertices A, B, C, D, E, and F.

#### **BFS (Breadth-First Search):**

Starting from vertex A:

1. Visit A and enqueue it.
2. Dequeue A and visit its adjacent vertices B and C.
3. Enqueue B and C.
4. Dequeue B and visit its adjacent vertex D and E.
5. Enqueue D and E.
6. Dequeue C and visit its adjacent vertex F.
7. Enqueue F.
8. Continue until the queue is empty.

The BFS traversal order is: A, B, C, D, E, F.

#### **DFS (Depth-First Search):**

Starting from vertex A:

1. Visit A and mark it as visited.
2. Recursively visit its adjacent vertices B and C.
3. Visit B and recursively visit its adjacent vertices D and E.
4. Visit D and E.
5. Backtrack to B and visit C.
6. Visit C and recursively visit its adjacent vertex F.
7. Visit F.
8. Continue until all vertices are visited.

The DFS traversal order is: A, B, D, E, C, F.

These traversals demonstrate the different exploration patterns of BFS and DFS algorithms. BFS explores vertices level by level, while DFS explores vertices along each branch before backtracking.

32. Construction of AVL tree for data: 9, 8, 7, 6, 5, 4, 3, 2, 1 (Practice for all the examples discussed in classroom).

### ChatGPT

To construct an AVL tree for the given data: 9, 8, 7, 6, 5, 4, 3, 2, 1, we'll insert each element into the AVL tree one by one while ensuring that the tree remains balanced after each insertion. AVL trees are self-balancing binary search trees, and they maintain a balance factor (height difference between the left and right subtrees) of -1, 0, or 1 for each node.

Here's how we can construct the AVL tree:

1. **Insert 9:** Insert 9 as the root of the AVL tree.

markdown

```

  9

```

2. **Insert 8:** Insert 8 as the left child of 9.

markdown

```

  9
 /
8

```

3. **Insert 7:** Insert 7 as the left child of 8.

markdown

```

  9
 /
8
/
7

```

4. **Insert 6:** Perform a right rotation at node 8, then insert 6 as the left child of 7.

markdown

```

  9
 /
7
/ \
6  8

```

5. **Insert 5:** Insert 5 as the left child of 6.

markdown

```

    7
   / \
  6   9
 / \
5  8

```

6. **Insert 4:** Perform a right rotation at node 7, then insert 4 as the left child of 5.

markdown



7. **Insert 3:** Perform a right rotation at node 5, then insert 3 as the left child of 4.

markdown



8. **Insert 2:** Perform a left rotation at node 5, then insert 2 as the left child of 3.

markdown



9. **Insert 1:** Perform a left rotation at node 3, then insert 1 as the left child of 2.

markdown



After inserting all the elements, we've constructed the AVL tree for the given data: 9, 8, 7, 6, 5, 4, 3, 2, 1. The tree is now balanced, and the balance factor of each node is within the acceptable range (-1, 0, 1).

## 22 You

33. Construct Binary Search Tree for the given data: 45,15,79,90,10,55,12,20,50 . Also determine pre-order, post-order and in-order traversal for the tree.



To construct a Binary Search Tree (BST) for the given data: 45, 15, 79, 90, 10, 55, 12, 20, 50, we'll insert each element into the BST one by one while ensuring that the BST property is maintained, which states that for any node, the value of nodes in its left subtree is less than its value, and the value of nodes in its right subtree is greater than its value.

Here's how we can construct the BST:

1. **Insert 45:** Insert 45 as the root of the BST.

markdown

```
45
```

2. **Insert 15:** Insert 15 as the left child of 45.

markdown

```
  45
 /
15
```

3. **Insert 79:** Insert 79 as the right child of 45.

markdown

```
  45
 /  \
15   79
```

4. **Insert 90:** Insert 90 as the right child of 79.

markdown

```
  45
 /  \
15   79
     \
     90
```

5. **Insert 10:** Insert 10 as the left child of 15.

markdown

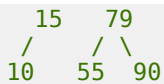
```
  45
 /  \
15   79
 /
10
```

6. **Insert 55:** Insert 55 as the right child of 79.

markdown

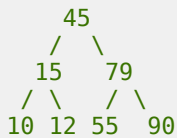
```
  45
 /  \
```





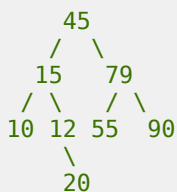
7. **Insert 12:** Insert 12 as the right child of 15.

markdown



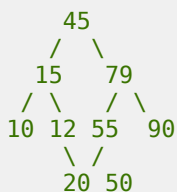
8. **Insert 20:** Insert 20 as the right child of 12.

markdown



9. **Insert 50:** Insert 50 as the left child of 55.

markdown



Now that we have constructed the BST, let's determine the pre-order, post-order, and in-order traversals for the tree:

- **Pre-order Traversal:** Root, Left, Right  
Pre-order Traversal: 45, 15, 10, 12, 20, 79, 55, 50, 90
- **In-order Traversal:** Left, Root, Right  
In-order Traversal: 10, 12, 15, 20, 45, 50, 55, 79, 90
- **Post-order Traversal:** Left, Right, Root  
Post-order Traversal: 10, 12, 20, 15, 50, 55, 90, 79, 45

These traversals provide different ways of visiting the nodes in the BST, each with its own application and use case.

## 22 You

34. Construct a binary tree for given expressions.

(a)  $a*(b+c)+(d*e)/f+g+h$

(b)  $((a*x+b)*x+e)*(x+f)$

### ChatGPT

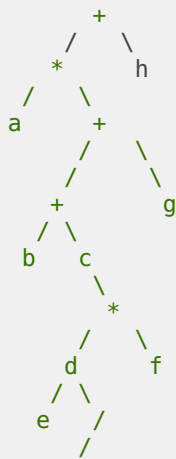
To construct binary trees for the given expressions, we'll use the concept of expression trees where operands are leaf nodes and operators are internal nodes. We'll construct the trees based on the precedence and associativity of operators.

Let's construct the binary trees for the given expressions:

(a)  $a(b+c)+(de)/f+g+h**$

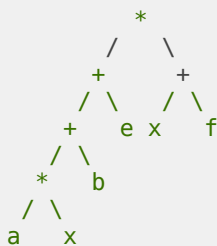
To construct the expression tree for this expression, we follow the order of operations (parentheses, then multiplication/division, then addition/subtraction).

markdown



(b)  $((a*x+b)x+e)(x+f)$

markdown



These are the binary trees constructed for the given expressions. In these trees, each internal node represents an operator, and each leaf node represents an operand. The trees are constructed in such a way that the order of operations is preserved.

### 22 You

35. Explain graph representation methods with the suitable examples.



Graphs can be represented in various ways, each with its own advantages and disadvantages depending on the specific application and requirements. The most commonly used methods for graph representation are:

### 1. Adjacency Matrix:

- An adjacency matrix is a 2D array where the elements indicate whether pairs of vertices are adjacent or not in the graph.
- If the graph is undirected, the matrix is symmetric across the diagonal.
- If the graph is weighted, the matrix contains the weights of the edges.
- In an unweighted graph, a 1 indicates the presence of an edge, and a 0 indicates absence.
- In a weighted graph, the weight of the edge is stored at the corresponding position.

Example:

Consider the following undirected graph:

lua



The adjacency matrix representation would be:

```

      0  1  2  3
0  0  1  1  0
1  1  0  1  1
2  1  1  0  1
3  0  1  1  0

```

### 2. Adjacency List:

- An adjacency list represents each vertex in the graph and lists all the vertices adjacent to it.
- It can be implemented using an array of linked lists, where each element of the array represents a vertex, and the linked list contains the adjacent vertices.
- Alternatively, it can be implemented using a dictionary, where the keys represent vertices, and the values represent lists of adjacent vertices.

Example:

Consider the same undirected graph as before.

The adjacency list representation would be:

makefile

```

0: [1, 2]
1: [0, 2, 3]

```

```
2: [0, 1, 3]
3: [1, 2]
```

### 3. Edge List:

- An edge list simply lists all the edges in the graph.
- It consists of a list of tuples, where each tuple represents an edge and contains the pair of vertices connected by the edge.
- If the graph is weighted, the tuples may also contain the weight of the edge.

Example:

Consider the same undirected graph as before.

The edge list representation would be:

CSS

```
[(0, 1), (0, 2), (1, 2), (1, 3), (2, 3)]
```

Each representation method has its own advantages and disadvantages:

- Adjacency matrices are efficient for dense graphs but can be inefficient for sparse graphs due to the wasted space.
- Adjacency lists are efficient for sparse graphs and can be more memory-efficient than adjacency matrices.
- Edge lists are simple and space-efficient, but accessing adjacency information may be slower than adjacency matrices or lists.

The choice of representation depends on factors such as the size and density of the graph, memory constraints, and the specific operations to be performed on the graph.

22 You

36. Explain in term of tree: Height, Degree of tree, Leaves, Internal and External nodes.

 ChatGPT

In terms of trees, several fundamental concepts describe the structure and characteristics of a tree. Let's explore these concepts:

#### 1. Height of a Tree:

- The height of a tree is the length of the longest path from the root to a leaf node.
- It represents the maximum number of edges on the longest path from the root to a leaf.
- The height of a tree with only one node (the root) is 0.
- The height of a tree is a measure of its depth or how "deep" the tree is.

#### 2. Degree of a Tree:

- The degree of a tree is the maximum number of children any node in the tree has.

- For example, in a binary tree, the maximum degree is 2 because each node can have at most two children.
- In a k-ary tree, the maximum degree is k.

### 3. Leaves:

- Leaves, also known as external nodes or terminal nodes, are the nodes in a tree that have no children.
- In other words, they are the nodes at the "end" of the tree branches.
- Leaves are the nodes with degree 0.

### 4. Internal Nodes:

- Internal nodes are the nodes in a tree that have at least one child.
- They are the non-leaf nodes or the nodes that are not at the "end" of the tree branches.
- Internal nodes have a degree greater than 0.

### 5. External Nodes:

- External nodes, also known as leaves, are the nodes in a tree that have no children.
- They are the nodes at the "end" of the tree branches.
- External nodes have a degree of 0.

Let's illustrate these concepts with an example:

mathematica



- Height of the tree: 3 (The longest path from the root to a leaf node is A -> B -> E -> F)
- Degree of the tree: 3 (The maximum number of children any node has is 3, for node A)
- Leaves: E, F, G, H, I (The nodes with no children)
- Internal nodes: A, B, C, D, H (The nodes with at least one child)
- External nodes: E, F, G, H, I (The nodes with no children)

Understanding these concepts is fundamental when analyzing and working with trees, as they provide insights into the structure and properties of the tree.

## 22 You

37. Define the term: Tree, Graph, Acyclic graph, leaf node, complete binary tree.

## ChatGPT

Certainly! Let's define each of the terms:

### 1. Tree:

- A tree is a hierarchical data structure composed of nodes connected by edges.
- It consists of a root node, which has zero or more child nodes, with each child node having its own subtree.
- In a tree, there is exactly one path between any pair of nodes.
- A tree does not contain any cycles (i.e., there are no loops).

## 2. Graph:

- A graph is a collection of vertices (nodes) and edges that connect pairs of vertices.
- Graphs can be either directed or undirected.
- In a directed graph, edges have a direction, indicating a one-way connection between vertices.
- In an undirected graph, edges have no direction, indicating a two-way connection between vertices.

## 3. Acyclic Graph:

- An acyclic graph, also known as a directed acyclic graph (DAG), is a directed graph that contains no directed cycles.
- In other words, it is a graph where it is impossible to traverse continuously through the graph and return to the starting node.

## 4. Leaf Node:

- A leaf node, also known as a terminal node or external node, is a node in a tree data structure that has no children.
- In other words, it is a node with a degree of zero, meaning it does not have any outgoing edges.

## 5. Complete Binary Tree:

- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- In a complete binary tree, if a node has a right child, it must also have a left child.
- Complete binary trees are typically used in heap data structures, where elements are stored in an array, and they are also used in some binary tree-based sorting algorithms.

Understanding these terms is essential for working with trees and graphs, as they provide the foundational concepts necessary for designing, analyzing, and implementing algorithms and data structures related to trees and graphs.

22 You

38. What is binary tree traversal? What are the various traversal methods? Explain any two with suitable example.



ChatGPT

Binary tree traversal refers to the process of visiting and processing all the nodes in a binary tree in a specific order. There are three main methods of binary tree traversal:

### 1. In-order traversal:

- In an in-order traversal, nodes are visited in the order: left subtree, current node, right subtree.
- In other words, for each node, first, its left subtree is traversed, then the node itself is processed, and finally, its right subtree is traversed.
- In-order traversal is commonly used to produce a sorted sequence of the elements in a binary search tree.

### 2. Pre-order traversal:

- In a pre-order traversal, nodes are visited in the order: current node, left subtree, right subtree.
- In other words, for each node, first, the node itself is processed, then its left subtree is traversed, and finally, its right subtree is traversed.
- Pre-order traversal is useful for creating a copy of the tree or evaluating expressions in prefix notation.

### 3. Post-order traversal:

- In a post-order traversal, nodes are visited in the order: left subtree, right subtree, current node.
- In other words, for each node, first, its left subtree is traversed, then its right subtree is traversed, and finally, the node itself is processed.
- Post-order traversal is commonly used in expression trees to generate postfix notation for arithmetic expressions.

Let's illustrate in-order and pre-order traversal methods with suitable examples:

#### Example Binary Tree:

mathematica



#### In-order Traversal:

mathematica

In-order Traversal: D, B, E, A, F, C, G

Explanation: In in-order traversal, the nodes are visited in the order: left subtree, current node, right subtree. So, starting from the root node A, we first traverse the left subtree (B), then the current node (A), and finally, the right subtree (C).

#### Pre-order Traversal:

mathematica

Pre-order Traversal: A, B, D, E, C, F, G

Explanation: In pre-order traversal, the nodes are visited in the order: current node, left subtree, right subtree. So, starting from the root node A, we first process the current node (A), then traverse the left subtree (B), and finally, the right subtree (C).

These examples demonstrate how in-order and pre-order traversal methods visit the nodes of a binary tree in different orders, providing different perspectives on the structure of the tree.

22 You

39. Construct a binary tree having the following traversal sequences:

In-order traversal = 1, 10, 11, 12, 13, 14, 15, 17, 18, 21

Post-order traversal = 1, 11, 12, 10, 14, 18, 21, 17, 15, 13



ChatGPT

To construct a binary tree given the in-order and post-order traversal sequences, we can follow these steps:

1. The last element in the post-order traversal sequence is always the root of the tree.
2. Find the root node in the in-order traversal sequence.
3. Elements to the left of the root in the in-order traversal sequence represent the left subtree, and elements to the right represent the right subtree.
4. Recursively construct the left and right subtrees using the sub-sequences of in-order and post-order traversals.

Let's construct the binary tree:

In-order traversal = 1, 10, 11, 12, 13, 14, 15, 17, 18, 21

Post-order traversal = 1, 11, 12, 10, 14, 18, 21, 17, 15, 13

1. The last element in the post-order traversal is the root: 13
2. Find the root node (13) in the in-order traversal sequence: 1, 10, 11, 12, 13, 14, 15, 17, 18, 21
  - Elements to the left (1, 10, 11, 12) form the left subtree.
  - Elements to the right (14, 15, 17, 18, 21) form the right subtree.

css

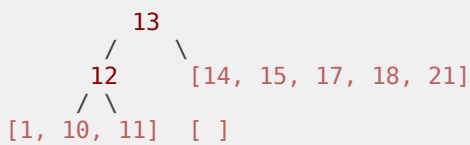
```

      13
     /  \
  [1, 10, 11, 12]  [14, 15, 17, 18, 21]
  
```

3. For the left subtree:
  - The root of the left subtree is the last element in the left subsequence of the post-order traversal: 12
  - Find the root node (12) in the left subsequence of the in-order traversal sequence: 1, 10, 11, 12
    - Elements to the left (1, 10, 11) form the left subtree.
    - Element 12 is the root.
    - No elements to the right.

css

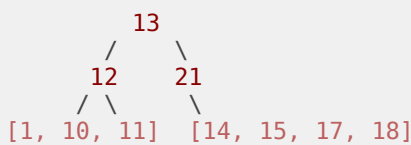




#### 4. For the right subtree:

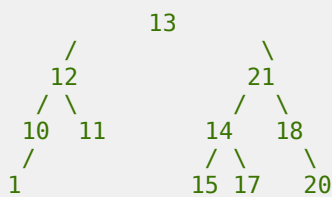
- The root of the right subtree is the last element in the right subsequence of the post-order traversal: 21
- Find the root node (21) in the right subsequence of the in-order traversal sequence: 14, 15, 17, 18, 21
  - Elements to the left (14, 15, 17, 18) form the left subtree.
  - Element 21 is the root.
  - No elements to the right.

CSS



#### 5. Continue recursively until all elements are processed.

markdown



This binary tree has the given in-order and post-order traversal sequences.

#### 22 You

40. Construct a binary tree having the following traversal sequences:

Pre-order traversal = A B C D E F G H I

In-order traversal = B C A E D G H F I



#### ChatGPT

To construct a binary tree given the pre-order and in-order traversal sequences, we can follow these steps:

1. The first element in the pre-order traversal sequence is always the root of the tree.
2. Find the root node in the in-order traversal sequence.
3. Elements to the left of the root in the in-order traversal sequence represent the left subtree, and elements to the right represent the right subtree.

4. Recursively construct the left and right subtrees using the sub-sequences of in-order and pre-order traversals.

Let's construct the binary tree:

Pre-order traversal = A B C D E F G H I

In-order traversal = B C A E D G H F I

1. The first element in the pre-order traversal is the root: A
2. Find the root node (A) in the in-order traversal sequence: B C A E D G H F I
  - Elements to the left (B, C) form the left subtree.
  - Elements to the right (E, D, G, H, F, I) form the right subtree.

mathematica



3. For the left subtree:
  - The root of the left subtree is the second element in the pre-order traversal: B
  - Find the root node (B) in the left subsequence of the in-order traversal sequence: B, C
    - Element B is the root.
    - No elements to the left.
    - No elements to the right.

CSS



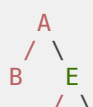
4. For the right subtree:
  - The root of the right subtree is the fifth element in the pre-order traversal: E
  - Find the root node (E) in the right subsequence of the in-order traversal sequence: E, D, G, H, F, I
    - Elements to the left (D, G, H, F) form the left subtree.
    - Element E is the root.
    - Element I is to the right.

CSS



5. Continue recursively until all elements are processed.

mathematica





This binary tree has the given pre-order and in-order traversal sequences.

ChatGPT can make mistakes. Consider checking important information.