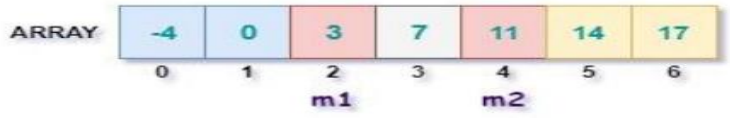
DAA TUTORIAL

Q1) If the Binary Search algorithm is modified to work as Ternary Search Algorithm where in the input will be divided into three almost equal parts and two mid pointers 'm1' and 'm2' will be used to mark the start of second and third list. (As shown in the diagram below).



Answer:

The Ternary Search Algorithm is a modification of the Binary Search Algorithm that divides the input array into three almost equal parts and uses two mid-pointers m1 and m2 to mark the start of the second and third lists. The algorithm works as follows:

Initialize the low and high indices of the search interval to the first and last indices of the array, respectively.

While the low index is less than or equal to the high index, repeat steps 3-6. Calculate the two mid-pointers as follows:

m1 = low + (high - low) / 3

m2 = high - (high - low) / 3

If the target element is equal to the element at m1 or m2, return the index of the element. If the target element is less than the element at m1, update the high index to m1 - 1.

If the target element is greater than the element at m2, update the low index to m2 + 1. If the target element is not found, return -1.

Let's apply the Ternary Search Algorithm to the given array (-4, 0, 3, 7, 11, 14, 17) to search for the Element 11.

Set low = 0 and high = 6 low <= high, so continue Calculate m1 and m2:

m1 = 0 + (6 - 0) / 3 = 2

m2 = 6 - (6 - 0) / 3 = 4

11 > array[m1] and 11 < array[m2], so update low = m1 + 1 and high = m2 - 1:

low = 2 + 1 = 3

high = 4 - 1 = 3

low <= high, so continue

Calculate m1 and m2:

m1 = 3 + (3 - 3) / 3 = 3

m2 = 3 - (3 - 3) / 3 = 3

11 == array[m2], so return 3.

Therefore, the Ternary Search Algorithm returns the correct index 3 for the element 11 in the array (- 4, 0, 3, 7, 11, 14, 17). The time complexity of the Ternary Search Algorithm is O(log3 n), which is slightly better than the time complexity of the Binary Search Algorithm (O(log2 n)).

Q2) List the time complexity of Push and Pop operations. If the implementation of stack is done using queue, then discuss the time analysis of Push and Pop operations.

Answer:

The time complexity of Push and Pop operations in a stack depends on the underlying data structure used to implement it.

If the stack is implemented using an array, the time complexity of both Push and Pop operations is O(1) (constant time) on average. However, in the worst-case scenario, if the array needs to be resized due to lack of space, the time complexity of Push can become O(n) (linear time), where n is the current size of the array.

If the stack is implemented using a linked list, the time complexity of both Push and Pop operations is O(1) (constant time) on average, as there is no need for resizing.

If the stack is implemented using a queue, then Push and Pop operations become more expensive. For Push operation, we need to enqueue the element at the end of the queue, which takes O(1) time. However, for Pop operation, we need to dequeue all the elements from the front of the queue until we reach the last element, which takes O(n) time, where n is the number of elements currently in the queue. Therefore, the time complexity of Pop operation in a stack implemented using a queue is O(n).

To improve the time complexity of Pop operation, we can use two queues to implement the stack. In this case, when we push an element, we enqueue it to the first queue. To pop an element, we dequeue all the elements from the first queue except the last one, and enqueue them to the second queue. Then we dequeue the last element from the first queue, which is the top element of the stack. Finally, we swap the two queues, so that the second queue becomes the first queue and vice versa. This way, the time complexity of both Push and Pop operations becomes O(1) (constant time) on average, as required for a stack.

Q3. List the time complexity of Enqueue and Dequeue operations. If the implementation of queue is done using stack, then discuss the time analysis of Enqueue and Dequeue operations.

Answer:

The time complexity of Enqueue and Dequeue operations in a queue depends on the underlying data structure used to implement it.

If the queue is implemented using an array, the time complexity of Enqueue operation is O(1) (constant time) on average, as we can simply add the element to the end of the array. However, in the worst-case scenario, if the array needs to be resized due to lack of space, the time complexity of Enqueue operation can become O(n) (linear time), where n is the current size of the array. The time complexity of Dequeue operation is also O(1) (constant time) on average, as we can simply remove the element from the front of the array. However, if we need to shift all the elements to the left after removing the front element, the time complexity of Dequeue operation becomes O(n) (linear time), where n is the number of elements in the array.

If the queue is implemented using a linked list, the time complexity of both Enqueue and Dequeue operations is O(1) (constant time) on average, as there is no need for resizing or shifting.

If the queue is implemented using a stack, then Enqueue and Dequeue operations become more expensive. To enqueue an element, we need to push it onto one stack, then pop all the elements from the second stack and push them onto the first stack, before finally pushing the new element onto the first stack. This takes O(n) time, where n is the number of elements in the queue, as we need to pop and push each element twice. Therefore, the time complexity of Enqueue operation in a queue implemented using two stacks is O(n).

To dequeue an element, we simply need to pop the top element from the first stack, which takes O(1) time. However, if the second stack is empty, we need to transfer all the elements from the first stack to the second stack before we can pop the top element. This takes O(n) time, where n is the number of elements in the queue, as we need to pop and push each element once. Therefore, the time complexity of Dequeue operation in a queue implemented using two stacks is also O(n).

To improve the time complexity of Enqueue operation, we can use two stacks to implement the queue in such a way that Enqueue operation takes O(1) time on average. In this case, we push the new element onto the first stack. When we need to dequeue an element, we pop it from the second stack if it is not empty, otherwise we transfer all the elements from the first stack to the second stack in reverse order before popping the top element. This way, the time complexity of Enqueue operation remains O(1) on average, and the time complexity of Dequeue operation becomes O(1) on average if we assume that a sequence of n Enqueue and Dequeue operations takes O(n) time to execute.

Q4) Matrix multiplication for 2x2 and 4x4 Matrix with its time complexity analysis a. Naïve Matrix multiplication and b. Strassen's matrix multiplication

Answer:

Naïve Matrix Multiplication:

For two matrices A and B of size n x n, the naive matrix multiplication algorithm involves computing each entry in the resulting matrix C by summing the products of corresponding entries in rows of A and columns of B. The time complexity of this algorithm is O(n^3), since we need to perform n^2 multiplications and n^2 additions for each of the n^2 entries in the resulting matrix.

For a 2x2 matrix multiplication, the time complexity is O(2^3) = O(8), since we need to perform 8 multiplications and 4 additions.

For a 4x4 matrix multiplication, the time complexity is O(4^3) = O(64), since we need to perform 64 multiplications and 16 additions.

Strassen's Matrix Multiplication:

Strassen's algorithm is a more efficient algorithm for matrix multiplication than the naive approach, especially for large matrices. It involves recursively dividing the matrices into smaller sub-matrices and performing a combination of matrix additions and subtractions to reduce the number of multiplications required.

For a 2x2 matrix multiplication, Strassen's algorithm involves performing 7 multiplications and 18 additions/subtractions, resulting in a time complexity of O(2^log2(7)) = O(2^2.81) ≈ O(7.6).

For a 4x4 matrix multiplication, Strassen's algorithm involves performing 49 multiplications and 156 additions/subtractions, resulting in a time complexity of O(4^log2(49)) = O(4^2.81) ≈ O(205.4).

However, there is a practical limit to how much Strassen's algorithm can be faster than the naive algorithm, due to the overhead involved in the recursive sub-matrix calculations and additional arithmetic operations required. The crossover point, where the Strassen's algorithm becomes more efficient than the naive algorithm, is typically for matrices larger than 64x64 or 128x128. For smaller matrices, the naive algorithm is generally faster.

Q5) Solve the given recurrence relations (2nd order) using characteristic equation.

a.) Fn=10 Fn-1 – 25 Fn-2 where F0=3 and F1=17

b.) Fn=2 Fn-1 – 2 Fn-2 where F0=1 and F1=3

Answer:

a.) Fn=10 Fn-1 – 25 Fn-2 where F0=3 and F1=17

To solve this recurrence relation using the characteristic equation, we first assume a solution of the form:

Fn = ar^n

where a and r are constants. Substituting this into the recurrence relation, we get:

ar^n = 10ar^(n-1) - 25ar^(n-2)

Dividing both sides by ar^(n-2), we get:

r^2 - 10r + 25 = 0

This is the characteristic equation. We solve for r by factoring:

(r-5)^2 = 0

r = 5 (a repeated root)

So, the general solution is:

Fn = (c1 + c2n)5^n

Using the initial conditions F0 = 3 and F1 = 17, we can solve for c1 and c2:

c1 = 3

c2 = (F1 - c1)/5 = (17-3)/5 = 2.8

Thus, the solution to the recurrence relation is:

Fn = 3(5^n) + 2.8n(5^n)

b.) Fn=2 Fn-1 – 2 Fn-2 where F0=1 and F1=3

Following the same procedure as above, we assume a solution of the form:

Fn = ar^n

where a and r are constants. Substituting this into the recurrence relation, we get:

ar^n = 2ar^(n-1) - 2ar^(n-2)

Dividing both sides by ar^(n-2), we get:

r^2 - 2r + 2 = 0

This is the characteristic equation. We solve for r using the quadratic formula:

r = (2 ± sqrt(2^2 - 4(1)(2))) / 2 = 1 ± i

So, the general solution is:

Fn = c1(1+i)^n + c2(1-i)^n

Using the initial conditions F0 = 1 and F1 = 3, we can solve for c1 and c2:

c1 = (F1 - (1-i)F0) / (2i) = (3 - (1-i)(1)) / (2i) = -i c2 = (1-i)F0 - c1 = (1-i) - (-i) = 1

Thus, the solution to the recurrence relation is:

Fn = (-i(1+i)^n + (1-i)(1-i)^n) / 2

Q6) Solve the following hashing questions:

a) Given the following input (4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199) and the hash function x mod 10. Which hashes to the same value?

b) Find the hash function value using the multiplication method. Given that: {Key: value} pairs: {1234: "Ram", 5678: "Shyam"}, Size of the table: 100 , A = 0.56

Answer:

a.)The hash function is x mod 10, which means that the input is hashed to one of the 10 possible values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. To find which inputs hash to the same value, we simply apply the hash function to each input and group them by their resulting value:

0:

1: 1471

2: 4322

3: 1334, 9679

4: 1989, 4199

5:

6: 6171, 6173

7:

8:

9:

We can see that the inputs 1334 and 9679 hash to the same value (3), and the inputs 6171 and 6173 hash to the same value (6).

b.)The multiplication method for hashing involves multiplying the key by a constant A (0 < A < 1), taking the fractional part of the product, and multiplying it by the size of the hash table. The hash function value is the integer part of this product.

Given that the size of the table is 100 and A = 0.56, we can find the hash function values for the keys 1234 and 5678:

For key 1234:

h = floor(100 * (0.56 * 1234 % 1)) = floor(100 * 0.24624) = 24

For key 5678:

h = floor(100 * (0.56 * 5678 % 1)) = floor(100 * 0.95808) = 95

Therefore, the hash function values for the keys 1234 and 5678 are 24 and 95, respectively.

Q7) What is the difference between 'divide and conquer' and 'dynamic programming'? Give suitable example to justify your answer and explain above two with context of bottom-up and top-down approach.

Answer:

Both divide and conquer and dynamic programming are algorithmic techniques used to solve complex problems.

Divide and conquer is a recursive approach that breaks down a problem into smaller sub-problems of the same type until they become simple enough to be solved directly. The solutions to the sub- problems are then combined to solve the original problem. For example, the classic example of divide and conquer is the merge sort algorithm, which breaks down an unsorted array into two halves, sorts each half recursively, and merges the two sorted halves to produce a sorted array.

Dynamic programming, on the other hand, is a technique that solves a problem by breaking it down into overlapping sub-problems and solving each sub-problem only once. The solutions to the sub- problems are stored in a table or array, so that they can be used to solve larger sub-problems and eventually the original problem. For example, the Fibonacci sequence can be solved using dynamic programming by storing the values of previously computed Fibonacci numbers and reusing them to compute the next numbers.

The main difference between the two approaches is that divide and conquer divides the problem into non-overlapping sub-problems, while dynamic programming divides the problem into overlapping sub-problems. In other words, divide and conquer solves the sub-problems independently and combines them at the end, while dynamic programming solves the sub-problems in a specific order to avoid redundancy.

Both divide and conquer and dynamic programming can be implemented using either a top-down or bottom-up approach. The top-down approach starts with the original problem and recursively breaks it down into smaller sub-problems until the base case is reached. The solutions to the sub-problems are then combined to solve the original problem. This approach is often used in divide and conquer. The bottom-up approach, on the other hand, starts with the smallest sub-problems and iteratively solves larger sub-problems until the original problem is solved. This approach is often used in dynamic programming.

For example, let's consider the problem of finding the shortest path between two nodes in a graph. The divide and conquer approach would break down the problem into smaller sub-problems by dividing the graph into smaller sub-graphs and finding the shortest path between each pair of nodes in each sub-graph. The solutions to the sub-problems would then be combined to find the shortest path between the two original nodes. The dynamic programming approach, on the other hand, would break down the problem into smaller sub-problems by computing the shortest path between each pair of nodes in the graph and storing the solutions in a table. The solution to the original problem would then be retrieved from the table. The top-down approach for divide and conquer would start with the original graph and recursively divide it into smaller sub-graphs until the base case is reached. The

bottom-up approach for dynamic programming would start with the smallest sub-problems (pairs of adjacent nodes) and iteratively solve larger sub-problems (paths between nodes) until the original problem is solved.