Data Structure & Algorithm By – Coding Ninjas

Module 0
Time & Space Comolexity
1.Time Complexity
2.Space Complexity

Module 1
Arrays
1.Prefix and Suffix Sum
2.Kadane's Algorithm
3.Dutch National Flag Algorithm
4.Searching and Sorting
5.Mixed Problems

Module 2
Strings
1.Introduction to Strings
2.Mixed Problems in Strings

Module 3
Basic Algorithms
1.Two Pointers Technique
2.Sliding Window Technique

Module 4
Multidimensional Arrays
1.Multidimensional Arrays
2.Traversal Based Problems
3.Rotation Based Problems

Module 5

Recursion & Backtracking
1.Recursion
2.Divide & Conquer
3.Backtracking
4.Mixed Problems

Module 6
Sorting
1.Sorting Notes
2.Problems Based on Sorting

Module 7
Binary Search
1.Binary Search Notes
2.Binary Search on Matrix
3.Binary Search on Answer

Module 8
Linked Lists
1.Linked List Reversal
2.Sorting in Linked List
3.Slow and Fast Pointers
4.Modify Linked List
5.Mixed Problems

Module 9
Stacks & Queues
1.Stacks
2.Queues
3.Implementation-based problems
4.Application-based problems

Module 10
Trees
1.Introduction to Trees
2.Trees traversals
3.Construction Of Trees
4.Tree Views
5.Standard Problems
6.Mixed Problems

Module 11
Binary Search Trees
1.Introduction to BST
2.Construction of BST
3.Conversion Based Problems
4.Modification in BST Problems
5.Standard Problems in BST
6.Mixed Problems in BST

Module 12

Priority Queues & Heaps

1.Priority Queues & Heaps Notes
2.Implementation and Conversion Based Problems
3.K Based Problems
4.Application Based Problems

Module 13
Graphs

1.Introduction To Graphs
2.Graph Traversals
3.Minimum Spanning Trees
4.Shortest/Minimum path algorithms
5.Topological Sort
6.Problems related to graph algorithms
7.Problems related to graphs in matrix
8.Mixed Problems

Module 14
Dynamic Programming

1.Dynamic Programming Notes
2.DP with Arrays Based Problems
3.DP with Strings Based Problems
4.DP with Maths Based Problems
5.DP with Trees Based Problems
6.Breaking and Partitioning Based Problems
7.Counting Based Problems
8.Standard Problems
9.Mixed Problems

Module 15
Tries

1.Introduction to Tries
2.Application Based Problems

Module 16
Bit Manipulation

1.Bit Manipulation
2.Application Based Problems
3.Mixed Problems

Module 17
Greedy

1.Greedy

2.Minimize/Maximize Problems
3.Sorting based Problems
4.Standard Problems
5.Mixed Problems

Module 18
Miscellaneous Topics
1.Hashmaps
2.Circular Queues
3.Deques
4.Doubly Linked Lists
5.Circular Linked Lists
6.String Algorithms

What is Running Time Analysis?

It is the process of determining how processing time of a problem increases as the size of the problem (input size) increases. Input size is the number of elements in the input, and depending on the problem type, the input may be of different types. Example:

- Size of an array
- Polynomial degree
- Number of elements in matrix
- Number of bits in the binary representation of the input
- Vertices and edges in the graph

How to Compare Algorithms?

To compare algorithms, let us define a few objective measures

Execution Times? Not a good measure as execution time are specific to a particular computer

Number of statements executed? Not a good measure as the number of statements varies with programming languages as well as with the style of individual programmer

Ideal Solution? Let us assume that we express the running time of a given algorithm as a function of input size n(i.e., f(n)) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc. We measure the total number of basic operations (additions, subtractions, increments, multiplications, divisions,modulo etc.) performed as a function of input size.

What is the Rate of Growth?

The rate at which running time increases as a function of input is called rate of growth.

Let us assume that you go to a shop to buy a car and a bicycle. If your friend sees you over there and asks you what you are buying, then in general you say buying a car. This is because the cost of the car is high compared to the cost of the bicycle (approximately the cost of the bicycle to the cost of the car).

Total cost = cost of car + cost of bicycle

Total cost ~ cost of car (approximation)

Similarly,

For a given function, ignore the low order terms that are relatively insignificant.

$n^4 + 2n^3 + 100n + 500 \sim n^4$ (for large value of input size, n)

Commonly used rate of growths

Time Complexity	Name
1	Constant
logn	Logarithmic
n	Linear
nlogn	Linear Logarithmic
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n!	Factorial

Types of analysis

To analyse the given algorithm, we need to know with which inputs does the algorithm take less time and with which inputs the algorithm takes a long time. We have already seen that an algorithm can be represented in the form of an expression. That means we represent the algorithm with multiple expressions:one for the case where it takes less time and another for the case where it takes more time.

Three types of analysis are generally performed:

• **Worst-Case Analysis:** The worst-case consists of the input for which the algorithm takes the longest time to complete its execution.

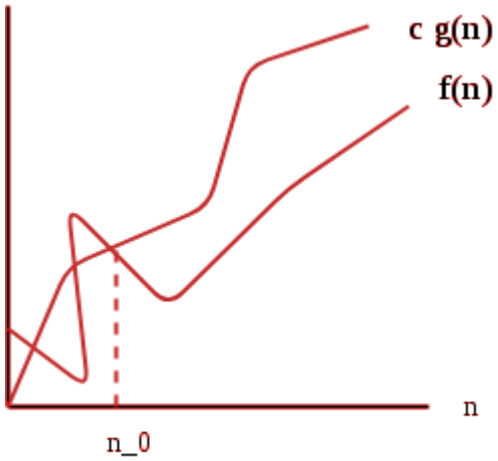
• **Best Case Analysis:** The best case consists of the input for which the algorithm takes the least time to complete its execution.

• **Average case:** The average case gives an idea about the average running time of the given algorithm.

Asymptotic Notations

We aim to identify upper and lower bounds by doing worst case, average case and best case analysis. To represent the upper and lower bounds, we need some kind of syntax, and that is the subject of the following discussion. Let us assume that the given algorithm is represented in the form of function f(n).

- **Big-O Notation**
  This notation gives the **tight upper bound** of the given function. Generally, it is represented as f(n)=O(g(n)). That means at larger values of N the upper bound of a f(n) is g(n).
  For example: if f(n) = $n^4 + 2n^3 + 100^n + 500$ is the given algorithm then n^4 is g(n). That means g(n) gives the maximum rate of growth for f(n) at larger values of n.



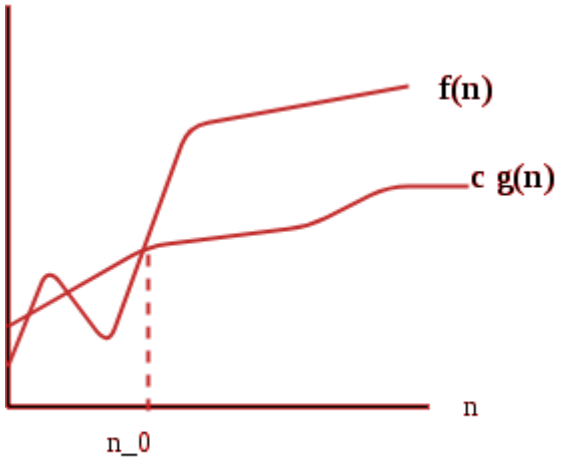
f(n) = O(g(n))

O-notation is defined as
O(g(n)) = {f(n): there exists positive constants c and n_0 such that 0 <= f(n) <= cg(n) for all n > n_0 }. g(n) is an asymptotic tight upper bound for f(n). Our objective is to give the smallest rate of growth g(n) which is greater than or equal to the given algorithm's rate of growth.

Generally we discard lower values of n. That means the rate of growth at lower values of n is not important. In the figure, n_0 is the point from which we need to consider the rate of growth for a given algorithm. Below n_0 the rate of growth could be different. n_0 is called the threshold for the given function.

- **Omega-Ω Notation**
  Similar to the O discussion, this notation gives the **tighter lower bound** of the given algorithm and we represent it as f(n) = Ω(g(n)). That means, at larger values of n, the tighter lower bound of f(n) is g(n).

  For example, if f(n) = $100n^2+10n+50$, g(n) is $\Omega(n^2)$.

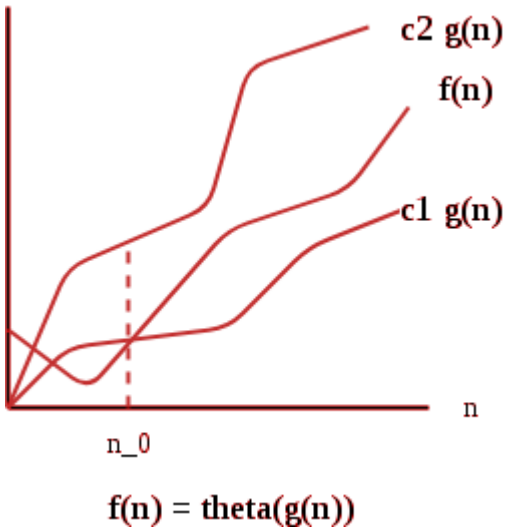


f(n) = Omega(g(n))

Ω-notation is defined as
Ω(g(n)) = {f(n): there exists positive constants c and n_0 such that 0 <= cg(n) <= f(n) for all n > n_0 }.
g(n) is an asymptotic lower bound for f(n). Our objective is to give the largest rate of growth g(n) which is less than or equal to the given algorithms' rate of growth.

- **Theta-Θ Notation**
  This notation decides whether the upper and the lower bound of the given function(algorithm) are the same. The average running time of an algorithm is always **between the lower bound and the upper bound**. If the upper bound(O) and the lower bound(Ω) give the same result, then the Θ notation will also have the same rate of growth. As an example, let us assume that f(n) = 10n+ n is the expression. Then, its tight upper bound g(n) is O(n). The rate of growth in the best case is g(n) = O(n).

  In this case, the rate of growth in the best case and worst case are the same. As a result the average case will also be the same. For a given function(algorithm), if the rates of growth for O and Ω are not same, then the rate of growth for Θ case may not be the same.



Θ-notation is defined as
Θ(g(n)) = {f(n): there exists positive constants c and n_0 such that 0 <= c_1g(n) <= f(n) <= c_2g(n) for all n > n_0 }. g(n) is an asymptotic tight bound for f(n). Θ(g(n)) is the set of functions with the same order of growth as g(n).

Note: In the analysis, we generally focus on the upper bound(O) because knowing the lower bound(Ω) of an algorithm is of no practical importance, and we use the Θ notation if the upper bound(O) and lower bound(Ω) are the same .

Guidelines for Asymptotic Notations

There are some general rules to help us determine the running time of an algorithm.

- Loops: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations .

```
//   executes n times
for i from 1 to n
      m = m + 2;       //   constant time c
      i++;
```

Total time = (a constant c) x n = cn = O(n)

- Nested loops: Analyze from the inside out. Total running time is the product of the sizes of all the loops.

```
//   outer loop executed n times
for i from 1 to n {
     //   inner loop executed n time
     i++;
     for j from 1 to n {
          k = k+1;    //   constant time
          j++
     }
}
```

Total time = c x n x n = cn^2 = $O(n^2)$

- Consecutive Statements: Add the time complexities of each statement.

```
x = x + 1;                 //  constant time
//   executed n times
for i from 1 to n {
      m = m + 2;       //   constant time c
      i++;
}
//   outer loop executed n time
for i from 1 to n {
     //   inner loop executed n time
     for j from 1 to n {
          k = k + 1;   //   constant time
          j++;
      }
}
```

Total time = $c_0 + c_1n + c_2n^2 = O(n^2)$

- If-then-else statements: Worst case running time: the test, plus either the then part or the else part (whichever is the largest)

```
//   condition check: constant time
if(length() == 0)
     return false;
else {
     // else part: (constant + constant) * n
     for n = 0 to n< length {
          //   another if: constant + constant (no else part)
          if( !list[n].equals(otherList.list[n]) )
               return false;
     }
}
```

Total time = $c_0 + (c_1 + c_2) * n = O(n)$

- Logarithmic Complexity: Algorithm is O(logn) if it takes a constant time to cut the problem size by a fraction(usually by 1/2).

```
for i from 1 to n
    i = i * 2
```

If we observe carefully, the value of i is doubling every time. Initially i=1, in the next step i=2, and in subsequent steps i=4,8 and so on. Let us assume that the loop is executing some k times. At the kth step $2^k = n$, and at (k+1)th step we come out of the loop.

Taking logarithm on both sides

$log(2^k) = logn$

$klog2 = logn$

$k = logn$

Total time = O(logn)

Recurrence relations and Master's Theorem

Recurrence Relation: A recurrence relation is an equation that recursively defines a sequence where the next term of the sequence is a function of the previous terms. In other words we express the n^{th} term of the sequence(F_n) as a function of the previous terms F_i(i<n).
For example: Fibonacci Series: $F_n = F_{n-1} + F_{n-2}$, where $F_0 = 0$ and $F_1 = 1$.
In Fibonacci Series the n^{th} term can be expressed as the sum of previous 2 terms. The base case for n <= 1 is also defined above. So the second term is $F_2 = F_0 + F_1 = 1$, $F_3 = 2$, $F_4 = 3$ and so on.

There are various algorithms whose running time can be described in terms of a recurrence relation, and solving the recurrence relation gives us the time complexity of the algorithm.
For example: Let us consider the recursive version of binary search to find the position of an element in a sorted array.
In **Binary search** we are given a sorted array of elements and we aim to find the target element. We do this by comparing the target with the middle element and compress the search space to half of its original size in every pass. If the target element is greater than the middle element we move our left pointer to the middle position else we move the right pointer to the middle position.

```
/*
      array from leftidx(0) to rightidx(arr.length-1) is considered
*/
function binarySearch(arr, leftidx , rightidx , target)
       //   base case : element not found
       if  leftidx > rightidx
                return -1

        middle = (leftidx + rightidx) / 2
        if  arr[middle] equals target
                return middle

        else if arr[middle] > target
                return binarySearch(arr , leftidx , middle - 1 , target)

        else
                return  binarySearch(arr, middle + 1, rightidx, target)
```

The recurrence relation for the above recursive function can be defined as -
Let T(n) denote the time taken by an algorithm to execute for input size 'n'. Hence, **T(n) = T(n/2) + 1** where T(1) = 1.

How to solve for T(n) ?
Solving the above recurrence relation and in general the recurrence relations for the algorithms following the divide and conquer paradigm can be easily done using the Master's Theorem.

Master's Theorem:
Master's theorem can be used to solve the recurrence relations of the type:
T(n) = a.T(n/b) + f(n) a>=1, b>1

Here,

- **n:** The size of the problem.
- **a:** The number of subproblems in the recursion.
- **n/b:** The size of each subproblem.(It is assumed that the size of all the subproblems are the same)
- **f(n):** The cost of work done outside the recursive calls, which basically includes the cost of dividing the problem and merging the solutions of the subproblems.

Note: Here 'a' and 'b' are constants and f(n) is asymptotically a positive function. In other words, for sufficiently large input size 'n', f(n) > 0 and T(n) is a monotonically increasing function.

The solution of recurrence relations of the form T(n) = a.T(n/b) + f(n) as given by Master's theorem where the whole idea is based upon the comparison of f(n) and n^($\log_b$a) and determining which of them is the dominating factor -

- **Case 1:** If f(n) = O(n^($\log_b$a - ε)), for some ε > 0, then T(n) = Θ(n^($\log_b$a)).
  This case can be interpreted as the worst case of f(n) is n^($\log_b$a - ε), which is less than n^($\log_b$a) so n^($\log_b$a) takes more time and dominates.
- **Case 2:** If f(n) = Θ(n^($\log_b$a)), then T(n) = Θ(n^($\log_b$a) * logn).
  If f(n) is also Θ(n^($\log_b$a)), then the time taken will be Θ(n^($\log_b$a)*logn).
- **Case 3:** If f(n) = Ω(n^($\log_b$a + ε)), for some ε > 0, and if a.f(n/b) <= c.f(n) for some c < 1 and all sufficiently large 'n', then T(n) = Θ(f(n)).
  Since the best case of f(n) is n^($\log_b$a - ε), so the best case of f(n) is greater than n^($\log_b$a), hence f(n) dominates.

Coming back to our problem of solving for T(n) for binary search, where T(n) = T(n/2) + 1, here the recurrence relation satisfies all the conditions of the master's theorem, where a = 1 and b = 2.

=> Calculating $\log_b a = \log_2 1 = 0$, so n^($\log_b$a) = n^(0) = 1.
=> Since f(n) = 1 = n^($\log_b$a)
=> Case 2: f(n) = Θ(n^($\log_b$a))
=> Hence, T(n) = Θ(logn)

Limitations of Master's Theorem

- We cannot use Master's theorem if T(n) is not monotone, for example T(n) = sin(n).
- f(n) must be a polynomial
- If a is not a constant, for example a = 2*n, or b cannot be expressed as a constant, for example T(n) = T(√n), then the master's theorem cannot be applied.

Let us now look at a few examples of Master theorem applications

a) $T(n) = 8T(n/2) + 1000n^2$

   Here,
   a = 8
   b = 2
   $f(n) = 1000n^2 = \Theta(n^2)$
   $\log_b a = \log_2 8 = 3 > 2$
   ie. f(n) < n^$\log_b$a-ϵ , where, ϵ is a constant.
   Case 1 implies here.
   Thus, $T(n) = f(n) = \Theta(n^{\log_b(a)}) = \Theta(n^3)$

b) $T(n) = 2T(n/2) + \Theta(n)$

Here,
a = 2
b = 2
f(n) = n
$\log_b a = \log_2 2 = 1 = 1$
ie. f(n) = n^$\log_b a$
Case 2 implies here.
Thus, $T(n) = f(n) = \Theta(n^{\log_2(2)} \log n) = \Theta(n \log n)$

c) $T(n) = 3T(n/2) + n^2$

Here,
a = 3
b = 2
$f(n) = n^2$
$\log_b a = \log_2 3 \approx 1.58 < 2$
ie. f(n) > n^$\log_b a+\epsilon$, where, ϵ is a constant.
Case 3 implies here.
Thus, $T(n) = f(n) = \Theta(n^2)$

What is Space Complexity?

Space complexity is a measure of the total amount of memory including the space of input values with respect to the input size, that an algorithm needs to run and produce the result.

Auxiliary Space and Space complexity

- **Auxiliary Space:** It is the temporary or extra space used by an algorithm apart from the input size in order to solve a problem.
- **Space complexity:** It is the total space used by an algorithm in order to solve a problem including the input size. It includes both auxiliary space and space taken by the input size.

  Space complexity = Auxiliary Space + Space taken by the input size

Calculating Space Complexity

The calculation of space complexity is necessary for determining the algorithm's efficiency. However, the space complexity also depends on the programming language, the compiler used, and the machine on which it is executed.

- Let us consider a program for calculating the sum of two numbers:

```
// Function to print the sum of two integers
function SumOfTwoIntegers()
   // Reading integers num1 and num2
   read(num1);
   read(num2);

   // Calculating the sum of two integers
   sum = num1 + num2;
   print("The sum of two integers");
   print(sum);
```

  We have declared three variables 'num1', 'num2',' 'sum', considering them of data type 'int' let the space occupied by 'int' data type be 4 bytes, hence the total space consumed is 4*3 = 12 bytes.

  Hence we can say that the space complexity of the above program is O(1) as 12 is a constant.

- Now, let us consider a program for calculating the sum of the numbers in an array:

```
// Function to calculate the sum of elements of the array
```

```
function sum()
   // Reading n, the size of the array
   read(n);
   // Declaring array A of size n
   A[n];

   // Reading the values of the array A
   for i = 0 to n - 1
      read(A[i]);

   // Calculating the sum of elements of array A
   for i = 0 to n - 1
      sum = sum + A[i];

   print("The sum of elemtents of array is");
   print(sum);
```

We have declared variables ‘n’, ‘sum’ and array of size ‘n’, considering them of datatype ‘int’, let the space occupied by ‘int’ be 4 bytes, hence the total space consumed is 4+4+n*4 bytes. Note that the auxiliary space is O(1) as only the sum variable is declared apart from the input array.

Since, the space consumed is a linear function of ‘n’, the size of the array so the space complexity of the above program is O(n), where n is the size of the array.

- Let’s consider a recursive version of the above program of calculating the sum of the numbers in an array

```
// Recursive function to calculate the sum of elements of array
function calc(A, index)
   if(index < 0) return 0;
   return A[index] + calc(A, index - 1);

// Function to calculate the sum of elements of array
function sum()
   // Reading n, the size of the array
   read(n);
   // Declaring array A of size n
   A[n];

   // Reading the values of the array A
   for i = 0 to n - 1
      read(A[i]);
```

```
// Calculating the sum of elements of array A through recursive function calc()
sum = calc(A, n - 1);

print("The sum of elemtents of array is");
print(sum);
```

The above program is a recursive version of the program to find the sum of numbers in an array. However, due to the recursive calls, we need to consider the space consumption due to function call stack too. The function call stack is responsible for keeping track of the function calls. The function call stack is made up of stack frames - one for each method call.

A stack frame consists of -

- Local variables
- Arguments passed to the function
- Information about caller's stack frame
- The return address of the function

In the above example, the maximum depth of the recursive call goes from n-1 to 0 as -

```
calc(n-1)
   |
calc(n-2)
   |
calc(n-3)
   .
   .
   .
calc(0)
```

Hence, the maximum depth of the recursion is all the way up to n, as we make n recursive calls which is definitely an auxiliary space for our program. So, the overall space complexity is O(n) + O(n) => O(n), where one term is the space consumed due to input size and the other is the auxiliary space due to function call stack.

NOTE: The notations used above for representing the space complexities of the programs have been discussed in detail in the tutorial for time complexity analysis.

The trade-off between Time and Space Complexity

The best algorithm to solve a particular problem is no doubt the one that requires less memory space and takes less time to execute. However, designing such an algorithm is not a trivial task, there can be more than one algorithm to solve a given problem one may require less memory space while the other may require less time to solve the problem.

So, it is quite common to observe a tradeoff between time and space consumed while designing an algorithm, where one needs to be sacrificed for the other. So, if space is a constraint, one might choose an algorithm that takes less space at the cost of more CPU time and vice-versa. Hence, we must choose an algorithm according to the requirements and the environment in which it needs to be executed.

Data Structure & Algorithm Notes by Coding Ninjas

Introduction to Arrays Notes

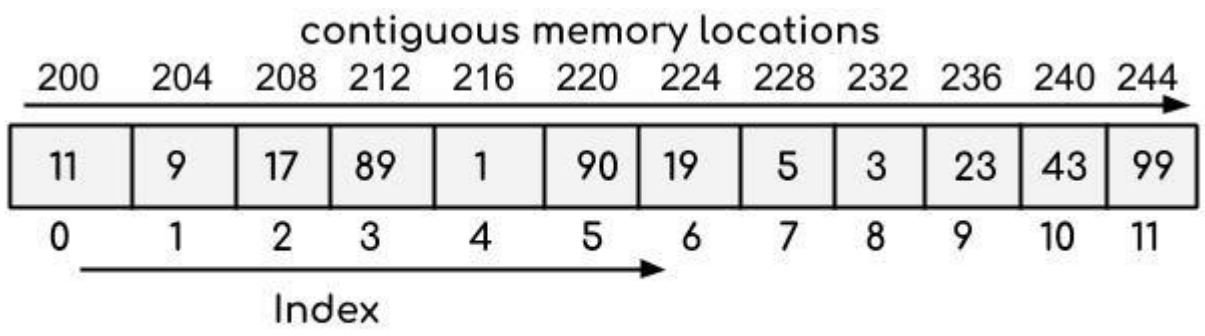
#1.Introduction To Arrays

An array is defined as a fixed-size collection of elements of the same data type stored in contiguous memory locations. It is the simplest data structure where each element of the array can be accessed by using its index.

Properties of arrays

- **Each element of the array is of the same data type and same size. For example: For an array of integers with the int data type, each element of the array will occupy 4 bytes.**
- **Elements of the array are stored in contiguous memory locations. For example :**

200 is the starting address (base address) assigned to the first element of the array and each element of the array is of integer data type occupying 4 bytes in memory.



Prefix and Suffix Sum Notes

Prefix and Suffix Sum

Prefix Sum:

Given an array, 'A' of size N, its prefix sum array is an array of the same size N such that the ith element of the prefix sum array 'Prefix' is the sum of all elements of the given array till ith index from the beginning, i.e Prefix[i] = A[0] + A[1] + A[2] + … + A[i].

For Example: Given A[] = [3, 4, -1, 2, 5], the prefix sum array P[] is given as -
P[0] = 3, P[1] = 7, P[2] = 6, P[3] = 8, P[4] = 13
i.e. P[] = [3, 7, 6, 8, 13]

Applications:

1. **Useful for answering efficiently range sum/xor queries, provided the array elements do not change over which the prefix sum/xor array is calculated.**
2. **Product of elements in a given range.**
3. **Useful for calculating maximum sum subarray and many more...**

Suffix Sum:

Given an array 'A' of size N, its suffix sum array is an array of the same size N such that the ith element of the suffix sum array 'Suffix' is the sum of all elements of the given array till ith index from the end, i.e Suffix[i] = A[i] + A[i+1] + A[i+2] + … + A[N-1] - (0-Based indexing).

For Example: Given A[] = [3, 4, -1, 2, 5], the suffix sum array S[] is given as -
S[0] = 13, S[1] = 10, S[2] = 6, S[3] = 7, S[5] = 5.
i.e. S[] = [13, 10, 6, 7, 5]

Suffix sum array can serve the same applications as prefix sum array, as it works in a similar manner to prefix sum array.

Kadane's Algorithm Notes

Kadane's Algorithm

Problem Statement
Given an array of N integers a1,a2,a3,....., aN find the maximum subarray(non-empty) sum of the given array.

NOTE: An array B is a subarray of an array A if B can be obtained from A by deleting several (possibly, zero, or all) elements from the beginning and several (possibly, zero or all) elements from the end. In particular, an array is a subarray of itself.

For example:
Array A[] = [-1, 2, -2, 5, 7, -3, 1]

Maximum subarray sum -> 12
Subarray(0-Based indexed) from index 1 to 4 -> [2, -2, 5, 7] and subarray(0-Based indexed) from index 3 to 4 -> [5, 7] have sum 12.

Kadane's Algorithm
The idea of Kadane's algorithm is to maintain a maximum subarray sum ending at every index 'i' of the given array and update the maximum sum obtained by comparing it with the maximum sum of the subarray ending at every index 'i'.

At any given index 'i' of the array, we can either:

- **Append the element at index 'i' to the maximum sum subarray(so just add the element at index 'i' to the maximum you've found so far).**
- **Start a new subarray starting from index 'i'.**

Appending an element at index 'i' to the maximum sum subarray obtained so far is beneficial if the sum till index 'i-1' is non-negative, otherwise it is better to start a new subarray starting from index 'i' and update the maximum sum obtained accordingly.

For Example: Consider the given array A[] = [1, -2, -3, 4, -1, 2, 1]. Element denotes the current element at index 'i', MaxSum is the maximum sum obtained so far till index 'i', Sum denotes the current sum obtained.

```
Initialize Sum to 0, MaxSum to INT_MIN
for i = 0
        A[i] = 1,
        Sum = Sum + A[i] = 1
        MaxSum = max(MaxSum,Sum) = 1

for i = 1
        A[i] = -2
        Sum = Sum + A[i] = -1
        MaxSum = max(MaxSum,Sum) = 1
```

Since Sum is negative, there is no point in appending the current sum obtained to the next element, so Sum = 0 i.e It is better to start a new subarray from the next index.

```
for i = 2
        A[i] = -3
        Sum = Sum + A[i] = -3
        MaxSum = max(MaxSum,Sum) = 1
```

Again, since Sum is negative, there is no point in appending the current sum obtained to the next element, so Sum = 0 i.e It is better to start a new subarray from the next index.

```
for i = 3
        A[i] = 4
        Sum = Sum + A[i] = 4
        MaxSum = max(MaxSum,Sum) = 4

for i = 4
        A[i] = -1
        Sum  = Sum + A[i] = 3
        MaxSum = max(MaxSum,Sum) = 4
```

Even if the element at A[i] is negative, the overall current sum is non-negative, so we retain the current sum to look for possible better options on appending the next elements. We have already updated the MaxSum for the subarray ending at index 3.

```
for i = 5
        A[i] = 2
        Sum = Sum + A[i] = 5
        MaxSum = max(MaxSum,Sum) = 5

for i = 6
        A[i] = 1
        Sum  = Sum + A[i] = 6
        MaxSum = max(MaxSum,Sum) = 6
```

Pseudocode:

```
function Kadane(arr, N)
```

```
        //      Initializing curSum to 0 and maxSum to min value, denoting an empty subarray
        curSum = 0
        maxSum = INT_MIN

        for idx = 0 to N-1
                curSum = curSum + arr[idx]
                //      Taking the max of maxSum and the curSum of the subarray
                maxSum = max(maxSum,curSum)

                //      Checking if the curSum becomes negative
                if curSum < 0
                        curSum = 0

        return maxSum
```

Time complexity: O(N), where N is the number of elements in the array, as we traverse the array once to get the maximum subarray sum.
Space complexity: O(1), as no extra space is required.

Dutch National Flag Algorithm Notes

Dutch National Flag Algorithm

Problem Statement
Given an array consisting of only 0s, 1s and 2s, sort the array.

Naive Approach:
Simply sort the array with the help of sorting algorithms like Merge Sort, Quick Sort. This gives a time complexity of O(N*logN), where N is the number of elements in the array.

Two-Pass Algorithm:
The solution involves iterating through the original array and counting the number of 0s, 1s, and 2s, and just overwriting the original array in a second pass. The only disadvantage is that we need to traverse the array twice to get a sorted array.

Steps:

- **Traverse the array once and keep track of the count of 0s, 1s and 2s encountered.**
- **Now traverse the array again and overwrite the array starting from the beginning, first with 0s, then 1s, and finally all 2s.**

Pseudocode:

```
/*
        array of size N from 0 to N-1 is considered
*/
function sort012(arr, N)

        //      Initialize the cnt0, cnt1 and cnt2 variables to 0.
        cnt0 = 0
        cnt1 = 0
        cnt2 = 0

        //      Count the number of 0s, 1s and 2s
        for idx = 0 to N-1
                if arr[idx] == 0
                        cnt0 += 1
                else if arr[idx] == 1
                        cnt1 += 1
                else
                        cnt2 += 1

        //      Now overwrite the array from the beginning
        for idx = 0 to N-1
                if cnt0 > 0
                        arr[idx] = 0
                        cnt0 -= 1
                else if cnt1 > 0
                        arr[idx] = 1
                        cnt1 -= 1
                else
                        arr[idx] = 2
```

```
                        cnt2 -= 1
```

Time complexity: O(N), where N is the number of elements in the array, as we traverse the array twice only.
Space complexity: O(1), as no extra space is required.

Dutch National Flag algorithm or Three-way partitioning

The Dutch National Flag algorithm or three-way partitioning algorithm allows sorting the array consisting of 0s, 1s, and 2s in a single traversal only and in constant space.

Steps:

- **Maintain three indices low = 0, mid = 0, and high = N-1, where N is the number of elements in the array.**
    1. **The range from 0 to low denotes the range containing 0s.**
    2. **The range from low to mid denotes the range containing 1s.**
    3. **The range from mid to high denotes the range containing any of 0s, 1s, or 2s.**
    4. **The range from high to N-1 denotes the range containing 2s.**
- **The mid pointer denotes the current element, traverses the array while mid<=high i.e we have exhausted the search space for the range which can contain any of 0s, 1s, or 2s.**
    1. **If A[mid] == 0, swap A[mid] and A[low] and increment low and mid pointers by 1.**
    2. **If A[mid] == 1, increment the mid pointer by 1.**
    3. **If A[mid] == 2, swap A[high] and A[mid] and increment mid by 1 and decrement high by 1.**

The resulting array will be a sorted array containing 0s, 1s, and 2s.

Pseudocode:

```
/*
        array of size N from 0 to N-1 is considered
*/
function DNF(arr, N)

        //      Initializing low, mid and high pointers
        low = 0
        mid = 0
        high = N-1

        while mid <= high
                /*
                Check if arr[mid] == 0, swap arr[low] and arr[mid], increment mid and
                low pointers
                */
                if arr[mid] == 0
                        swap(arr[mid],arr[low])
                        low += 1
                        mid += 1

                //      Check if arr[mid] == 1, increment mid pointer
                else if arr[mid] == 1
                        mid += 1

                /*
                Check if arr[mid] == 2, swap arr[mid] and arr[high], decrement high pointer
                */
                else if arr[mid] == 2
                        swap(arr[mid],arr[high])
                        high -= 1
```

Time complexity: O(N), where N is the number of elements in the array, as we sort the array in a single traversal only.
Space complexity: O(1), as no extra space is required.

Searching and Sorting Notes

Searching and Sorting

Searching

Searching means to find out whether a particular element is present in the given array/list. For instance, when you visit a simple google page and type anything that you want to know/ask about, basically you are searching that topic in google's huge database for which google is using some technique in order to provide the desired result to you.

There are basically two types of searching techniques:

- **Linear search**
- **Binary search**

Linear Search

It is a simple sequential search over all the elements of the array, and each element is checked for a match, if a match is found return the element otherwise the search continues until we reach the end of the array.

Pseudocode:

```
/*
        array from leftidx(0) to rightidx(arr.length-1) is considered
*/
function linearSearch(arr, leftidx , rightidx , target)

        //   Search for the target from the beginning of arr

        for idx = 0 to arr.length-1
                if arr[idx] == target
                        return idx

        //   target is not found
        return -1
```

Time complexity: O(N), as we traverse the array only once to check for a match for the target element.
Space complexity: O(1), as no extra space is required.

Binary Search

Search in a sorted array by repeatedly dividing the array into two halves and searching in one of the halves.
Suppose you want to find a particular element in the sorted array, then following this technique, you have to traverse all the array elements for searching one element but guess if you only have to search at most half of the array indices for performing the same operation. This can be achieved through binary search.

Now, let's look at what binary searching is.
Let us consider the array:

0	1	2	3	4
1	2	3	4	5

Given an array of size 5 with elements inside the boxes and indices above them. Our target element is 2.

Steps:

- **Find the middle index of the array.**
- **Now, we will compare the middle element with the target element. In case they are equal then we will simply return the middle index.**
- **In case they are not equal, then we will check if the target element is less than or greater than the middle element.**

1. **In case, the target element is less than the middle element, it means that the target element, if present in the given array, will be on the left side of the middle element as the array is sorted.**
2. **Otherwise, the target element will be on the right side of the middle element.**

- **This helps us discard half of the length of the array each time and we reduce our search space to half of the current search space.**

In the example above, the middle element is 3 at index 2, and the target element is 2 which is greater than the middle element, so we will move towards the left part. Now marking start = 0, and end = n/2-1 = 1, now middle = (start + end)/2 = 0. Now comparing the 0-th index element with 2, we find that 2 > 1, hence we will be moving towards the right. Updating the start = 1 and end = 1, middle becomes 1, comparing the 1-st index of the array with the target element, we find they are equal, meaning from here we will simply return 1 (the index of the element).

Advantages of Binary search:

- **This searching technique is fast and easier to implement.**
- **Requires no extra space.**
- **Reduces time complexity of the program to a greater extent i.e O(logN), where N is the number of the elements in the array, provided the given array is already sorted.**

Pseudocode:

```
/*
        array of size N from 0 to N-1 is considered
*/
function binarySearch(arr, N, target)

        //   Initializing lo and hi pointers
        lo = 0
        hi = N-1

        //   Searching for the target element until lo<=hi
        while lo <= hi

                //   Finding the mid of search space from lo to hi
                mid = lo + (hi-lo)/2

                //   If the target element is present at mid
                if arr[mid] == target
                        return mid

                /*
                If the target element is less than arr[mid], then if the target is
                present, it must be in the left half.
                */

                if target < arr[mid]
                        hi = mid-1

                //   Otherwise if the target is present, it must be in the right half
                else
                        lo = mid+1

        //   If the target is not found return -1
        return -1
```

Time complexity: O(logN), where N is the number of elements in the array, given the array is sorted. Since we search for the target element is one of the halves every time, reducing our search space to half of the current search space.

Since we go on searching for the target until our search space reduces to 1, so

Iteration 1- Initial search space: N
Iteration 2 - Search space: N/2
Iteration 3 - Search space: N/4

Let after 'k' iterations search space reduces to 1

So, N/(2k) = 1
=> N = 2k

Taking Log2 on both sides:
=> k = log2N
Hence, the maximum number of iterations 'k' comes out to be log2N.

Space complexity: O(1), as no extra space is required.

Sorting

Sorting means an arrangement of elements in an ordered sequence either in increasing(ascending) order or decreasing(descending) order. Sorting is very important and many software and programs use this. The major difference is the amount of space and time they consume while being performed in the program.

For a detailed explanation of types of sorting algorithms that are generally used, please refer to the topic Sorting Algorithms.

Strings Notes

Introduction to strings

A string is a data type in programming that is defined as a sequence of characters, it is implemented as an array of bytes (or words) that stores a sequence of elements, typically characters using some character encoding.

Depending on the programming language, strings can be of two types:

- Mutable strings: Mutable strings are strings that can be modified. However, it depends on the declaration and the programming language.
- Immutable strings: Immutable strings are strings that cannot be modified, the string is unique. Making any changes to the string involves creating a copy of the original string and deallocating it.

Operations on strings

Common string operations include finding the length, copying, concatenating, replacing, counting the occurrences of characters in a string. Such operations on strings can be performed easily with built-in functions provided by any programming language.

Some of the standard operations involving strings:

- Access characters: Retrieving characters of the string. Similar to arrays, strings follow 0-based indexing. For example, S = "apple", 'a' is the character at 0th index (S[0]), 'e' is the character at 4th index of the string S (S[4]).
- Concatenation: Joining characters end to end, combining strings.For example: S1 = "Hello ", S2 = "World.", the concatenation of strings S1 and S2 represented by S3 is S3 = S1+S2 => "Hello World." , while S3 = S2+S1 => "World.Hello ".
- Substring: A contiguous sequence of characters in a string. For example, Substrings of the string "boy" are: "", "b", "o", "y", "bo", "oy", "boy" (an empty string is also a substring).
    - Prefix: A prefix is any leading contiguous part of the string. For example, S = "garden" - "", "g", "ga", "gar", "gard", "garde", "garden" are all prefixes of the string S.
    - Suffix: A suffix is any trailing contiguous part of the string. For example, S = "garden" - "", "n", "en", "den", "rden", "arden", "garden" are all suffixes of the string S.

NOTE: A string of length 'N' has (N * (N + 1)) / 2 substrings.

Applications of strings

- String matching algorithms, which involve searching for a pattern in a given text have various applications in the real-world.
- String matching algorithms contribute to efficiently implementing Spell checkers, Spam filters, Intrusion detection systems, plagiarism detection, bioinformatics, digital forensics, etc.

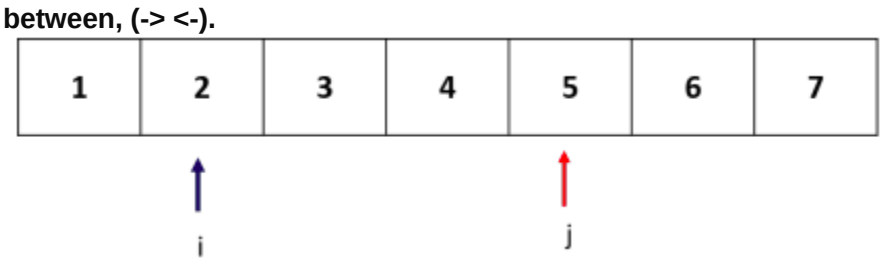
Two Pointers Technique Notes

Two Pointers

Two pointers is an algorithmic technique where, as the name suggests, we maintain two pointers to keep track of two indices in the array. The primary condition for using the Two Pointers technique is monotonicity. When we can prove that based on a certain condition the pointer is either going to move left or right and by how much, two pointers provide an efficient way to solve problems.

1. Equi-directional: Both start from the beginning: we have fixed and variable sliding window algorithms.

2. Opposite-directional: One at the start and the other at the end, they move close to each other and meet somewhere in between, (-> <-).



Equi Directional Two Pointers Technique

Here, we maintain a sliding window of flexible length whose endpoints are stored in our moving pointers. We keep one pointer always behind or at the other and use the other one to expand the window size.

- MINIMUM SUBARRAY WITH PRODUCT AT LEAST P

Problem Statement: Given an array A consisting of natural numbers, and a number P, find the length of the minimum length subarray whose product is >= P.

Example: Array : 1 2 3 4 5 6

P: 20

Output: 2

Explanation: The subarray 5-6 is the minimum length subarray with product >=20.

Approach:

We know that the product of the subarray is monotonically increasing as the size of the subarray increases. Therefore, we place a 'window' with left and right as i and j at the first item first. The steps are as follows:

- Get the optimal subarray starting from current i, 0: Then we first move the j pointer to include enough items that product[0:j+1]>=P, this is the process of getting the optimal subarray that starts with 0. And assume j stops at ed
- Get the optimal subarray that ends with current j, e0: we shrink the window size by moving the i pointer forward so that we can get the optimal subarray that ends with current j and the optimal subarray starts from s0.

- **Now, we find the optimal solution for subproblem [0:i,0:j](the start point in range [0, i] and endpoint in range [0,j]. Start from next i and j, and repeat steps 1 and 2.**

Fig.3 A Subarray with product >=20.

The code looks as follows:

```
function minSubarray(A, P)

        //   store the length of the array
        n = A.length

        /*
          maintain 2 pointers one for the left end and the other for the right end
          of the subarray
         */

        int i = 0, j = 0

        //   We maintain the Product of the current window in the variable curP
        curP = 1

        //   final answer
        ans  = infinity

        //   move the left pointer rightwards
        while(i < n)
        /*
           while we do not reach the required product keep expanding the
           window
         */
                while(j < n and curP <P)
                        curP *= A[j]
                        j++
                //   we update the ans with the window length
                if(curP >= P)
                ans = min(ans, j  - i + 1)
                //   move the left pointer and update its contribution to the product
                curP /= A[i]
                i++

        return ans
```

The above process is a standard flexible window size algorithm, and it is a complete search that searches all the possible result space. Both j and i pointer move at most n, it makes the total operations to be at most 2*n, which we get time complexity as O(n).

Opposite - directional Two Pointers Technique

We start with 2 pointers where one is usually placed at the beginning of the array and the other one is placed at the end of the array.

Two-pointers are usually used for searching a pair in the array. There are cases where the data is organized in a way that we can search all the result space by placing two pointers each at the start and rear of the array and move them to each other and eventually meet and terminate the search process. The search target should help us decide which pointer to move at that step. This way, each item in the array is guaranteed to be visited at most one time by one of the two pointers, thus making the time complexity be O(n). Binary search uses the technique of two pointers too, the left and right pointer together decide the current searching space, but it erases half searching space at each step instead.

- **TWO SUM**

Problem Statement: Given a sorted array and a number S, find the indices of any two elements which sum up to exactly S, or report that such a pair does not exist.

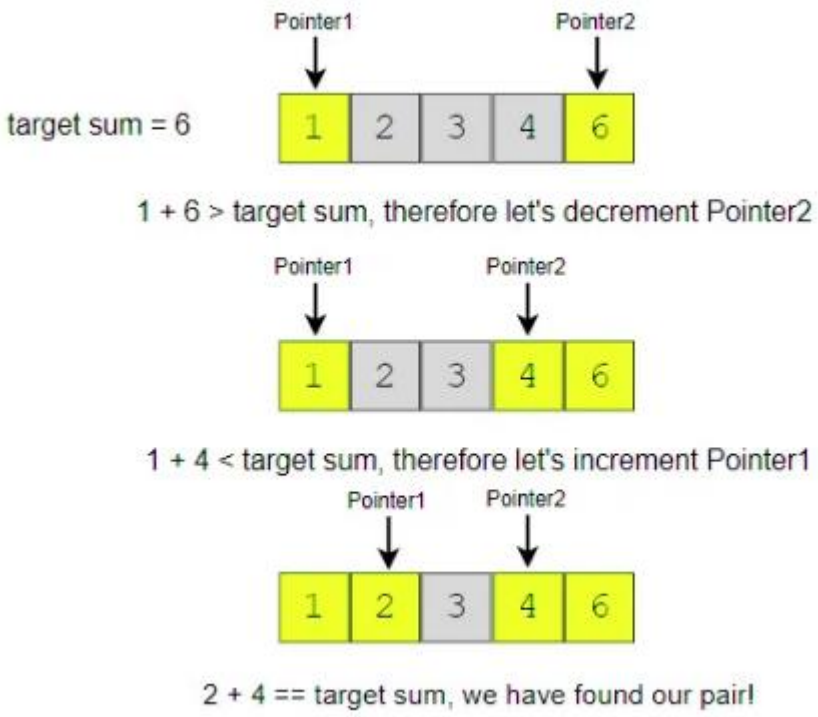
Example: Array: 1 2 3 4 5 6

Target Sum: 6

Output: 1, 3

Explanation: Elements at indices 1 and 3 (i.e 2 and 4 respectively sum upto the target sum = 6).

Approach:



Due to the fact that the array is sorted which means in the array [s,s1 ..., e1, e], the sum of any two integers is in the range of [s+s1, e1+e]. By placing two pointers, v1 and v2, that start from s and e, we started the search space from the middle of the possible range. [s+s1, s+e, e1+e]. Compare the target t with the sum of the two pointers v1 and v2:

1. S == v1 + v2: found
2. v1 + v2 < S: we need to move to the right side of the space from v1, i.e we increase v1 to get a larger value.
3. v1 + v2 > S: we need to move to the left side of the space from v2, i.e we decrease v2 to get a smaller value.
The code looks as follows:

```
function twoSum(A, S):
        //   store the length of the array
        n = A.length


        // maintain 2 pointers one for the left end and the other for the right end of the subarray.


        int i = 0, j = n - 1
        //   final answer pair
        ans  = {-1 , -1}
        //   while the left pointer is less than the right pointer
        while(i <= j)
                //   calculate the current Sum of 2 pointers
                curSum = A[i] + A[j]
                if(curSum == S)
                        //   we have found a valid pair so return the indices
                        ans = {i, j}
                        return ans
                else if (curSum > S)
                        //   move the right pointer left as the sum is greater
                        j -= 1
                else
                        //   move the left pointer right as the sum is smaller
                        i += 1
        return ans
```

Since both the pointers move to the right and left at most n times the total increment and decrement operators are bounded by 2*n. Hence the Time complexity is O(n). The space complexity is O(1) since we are using constant space to maintain the pointers.
Sliding Window Notes

Sliding Window

The sliding window technique is useful for solving problems in arrays or strings. Generally, it is considered as a technique that could reduce the time complexity from $O(n^2)$ to O(n).

Sliding Window Technique is a method for finding subarrays in an array that satisfy given conditions. We do this via maintaining a subset of items as our window and resize and move that window within the larger list until we find a solution.
There are two types of sliding window:

1. **Fixed window length k: the length of the window is fixed and it asks you to find something in the window such as the maximum sum of all windows, the maximum or a median number of each window. Usually, we need some kind of**

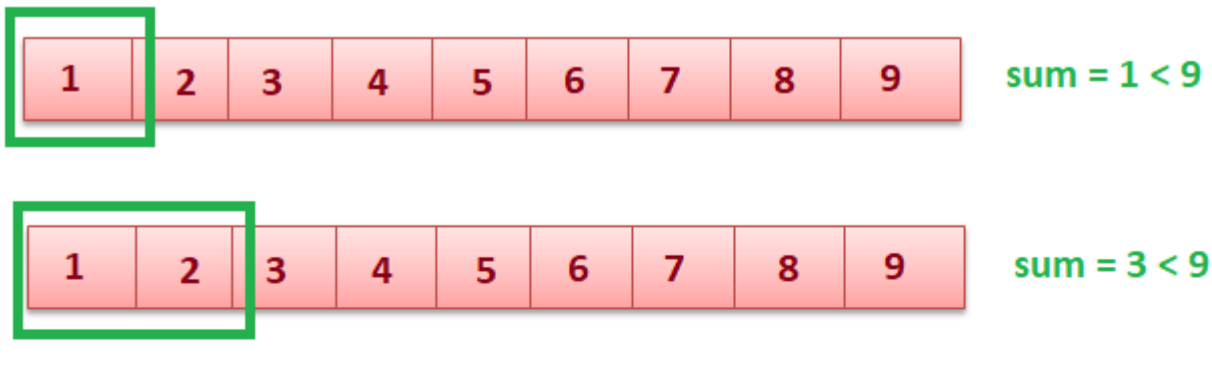
variables to maintain the state of the window, some are as simple as an integer or it could be as complicated as some advanced data structure such as list, queue, or deque.

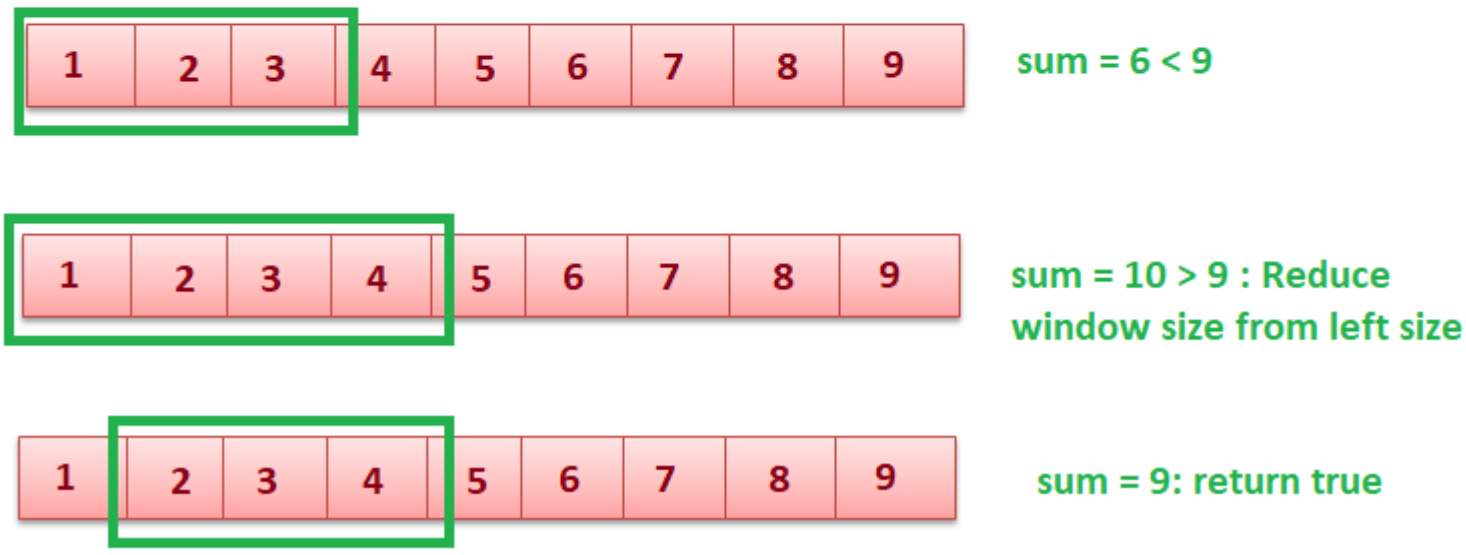
2. Two pointers + criteria: the window size is not fixed, usually it asks you to find the subarray that meets the criteria, for example, given an array of integers, find the number of subarrays whose sum is equal to a target value.

Let's understand it by an example:

- **EXAMPLE: Given an array of positive integers, find a subarray that sums up to target. Let the array be [1, 2, 3, 4, 5, 6, 7, 8, 9] and target be 9.**

We will start with window size = 1, then keep increasing the window size until the sum of elements inside the window is greater than or equal to the target. If the sum equals to target return true else decrease the window size from the left and reduce the sum till it is less than or equal to the target.






```
function countSubarrays(arr, target)
          count =0 ;
          curr_sum = arr[0], start = 0, i = 1

          //   Pick a starting point
          while i <= arr.length
                    //   If curr_sum exceeds the target, then remove the starting elements
                    while curr_sum > target AND start < i - 1
                              curr_sum = curr_sum - arr[start]
                              start += 1

                    //   If curr_sum becomes equal to target,  then return true
                    if curr_sum == target
                               p = i - 1
                               print("subarray found between start and p")
                               return 1

                    //   Add this element to curr_sum
                    if i < n
                              curr_sum = curr_sum + arr[i]
                    i += 1
          print("no subarray found")
          return 0
```

Time Complexity: O(n), only one traversal of array is needed.
Multidimensional Arrays Notes

Multidimensional Arrays

Introduction to multidimensional arrays

A multidimensional array is an array with more than one dimension. In a multidimensional array, each element is another array with a smaller number of dimensions.

Properties of multidimensional arrays

- **Just like single-dimensional arrays, the elements of a 2d array are also stored in contiguous memory locations, where each element of the array occupies a fixed memory size (for integers it is 4).**
- **An example of a 2D array of size 3 * 5:-**

	0	1	2	3	4
0	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
1	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
2	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

Accessing array elements

- **The elements of the array are accessed by using their index. The row index of a 2D array of size N * M ranges from 0 to N - 1. Similarly, the column index ranges from 0 to M - 1. For example: Accessing element at row index 5 and column index 7: Array[5][7] -> this is the element at the 6th row and 8th column in the array.**
- **Every array is identified by its base address i.e the location of the first element of the first row of the array in memory. So, basically, the base address helps in identifying the address of all the elements of the array.**
- **Since the elements of an array are stored in contiguous memory locations, the address of any element can be accessed from the base address itself. For example, 200 is the base address of the entire array. If the number of columns in the array is equal to 10 then the address of the element stored at the index Array[5][7] is equal to 200 + (5*10 + 7) * (size of(int)) = 428**

Applications of Multidimensional arrays

- **Multidimensional arrays are used to store the data in a tabular form. For example, storing the roll number and marks of a student can be easily done using multidimensional arrays. Another common usage is to store the images in 3D arrays.**
- **In dynamic programming questions, multidimensional arrays are used which are used to represent the states of the problem.**
- **Apart from these, they also have applications in many standard algorithmic problems like:**
  - **Matrix Multiplication**
  - **Adjacency matrix representation in graphs**
  - **Grid search problems**

Time Complexity of various operations

- **Accessing elements: Since elements in an array are stored at contiguous memory locations, they can be accessed very efficiently (random access) in O(1) time using indices.**
- **Finding elements: Finding an element in an array takes O(N * M) time in the worst case, where N is the number of rows of the array and M is the number of columns of the array, as you may need to traverse the entire array.**

Recursion Notes

Introduction to Recursion

Any function which calls itself is called recursion. A recursive method solves a problem by calling a copy of itself to work on a smaller problem. Each time a function calls itself with a slightly simpler version of the original problem. This sequence of smaller problems must eventually converge on a base case.

Working of recursion

We can define the steps of the recursive approach by summarizing the above three steps:

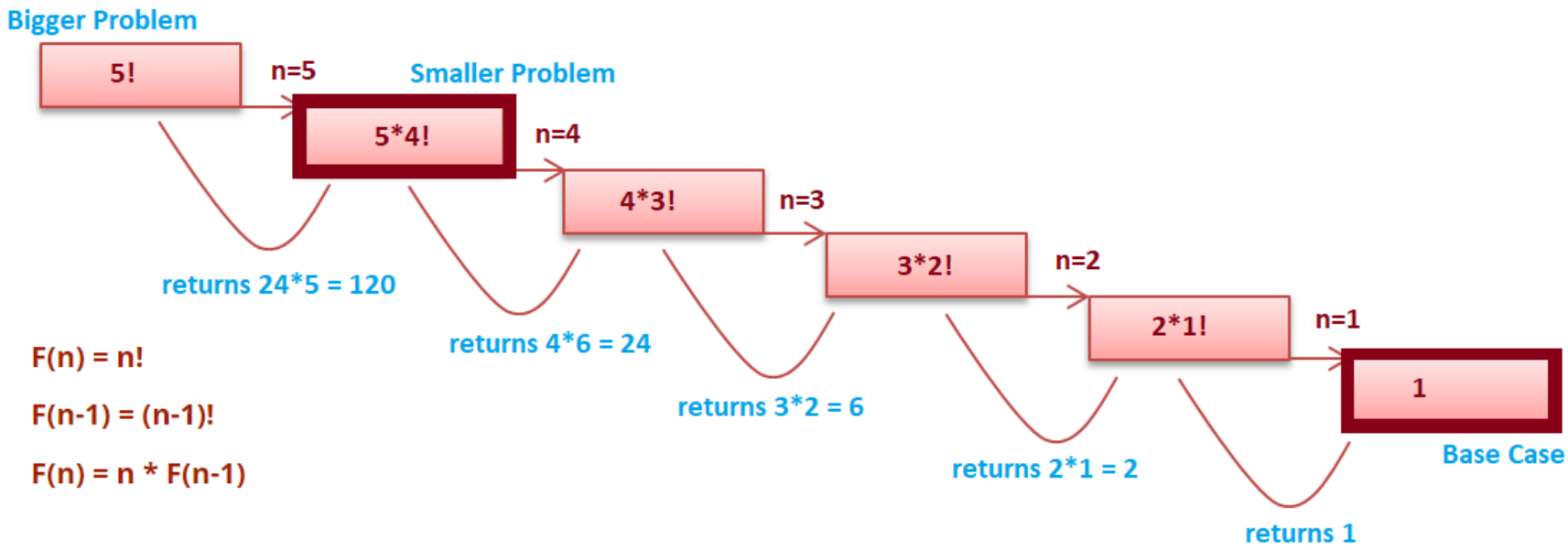
- **Base case: A recursive function must have a terminating condition at which the process will stop calling itself. Such a case is known as the base case. In the absence of a base case, it will keep calling itself and get stuck in an infinite loop. Soon, the recursion depth* will be exceeded and it will throw an error.**
- **Recursive call (Smaller problem): The recursive function will invoke itself on a smaller version of the main problem. We need to be careful while writing this step as it is crucial to correctly figure out what your smaller problem is.**
- **Self-work : Generally, we perform a calculation step in each recursive call. We can achieve this calculation step before or after the recursive call depending upon the nature of the problem.**

Note*: Recursion uses an in-built stack that stores recursive calls. Hence, the number of recursive calls must be as small as possible to avoid memory-overflow. If the number of recursion calls exceeded the maximum permissible amount, the recursion depth* will be exceeded. This condition is called stack overflow.

Now, let us see how to solve a few common problems using Recursion.

Problem Statement - Find Factorial of a number n.

Factorial of any number n is defined as n! = n * (n-1) * (n-2) * … *1. Ex: 5! = 5 * 4 * 3 * 2 * 1 = 120;
Let n = 5 ;



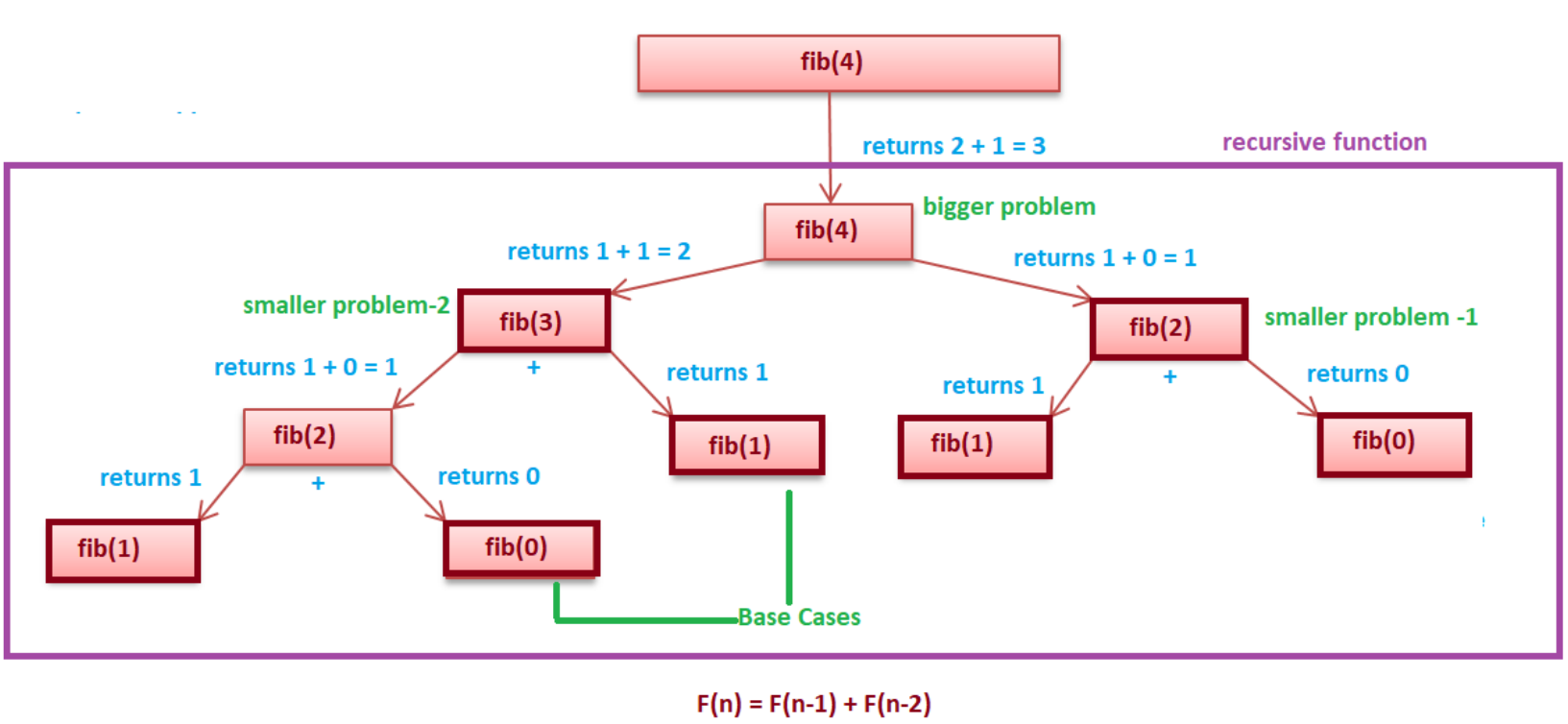
In recursion, the idea is that we represent a problem in terms of smaller problems. We know that 5! = 5 * 4!. Let's assume that recursion will give us an answer of 4!. Now to get the solution to our problem will become 4 * (the answer of the recursive call).

Similarly, when we give a recursive call for 4!; recursion will give us an answer of 3!. Since the same work is done in all these steps we write only one function and give it a call recursively. Now, what if there is no base case? Let's say 1! Will give a call to 0!; 0! will give a call to -1! (doesn't exist) and so on. Soon the function call stack will be full of method calls and give an error Stack Overflow. To avoid this we need a base case. So in the base case, we put our own solution to one of the smaller problems.

```
function factorial(n)
        //   base case
        if n equals 0
                    return 1
         //   getting answer of the smaller problem
         recursionResult = factorial(n-1)
         //  self work
         ans = n * recursionResult
         return ans
```

Problem Statement - Find nth fibonacci.

We know that Fibonacci numbers are defined as follows
fibo(n) = n for n <= 1
fibo(n) = fibo (n - 1) + fibo (n - 2) otherwise
Let n = 4



As you can see from the above fig and recursive equation that the bigger problem is dependent on 2 smaller problems. Depending upon the question, the bigger problem can depend on N number of smaller problems.

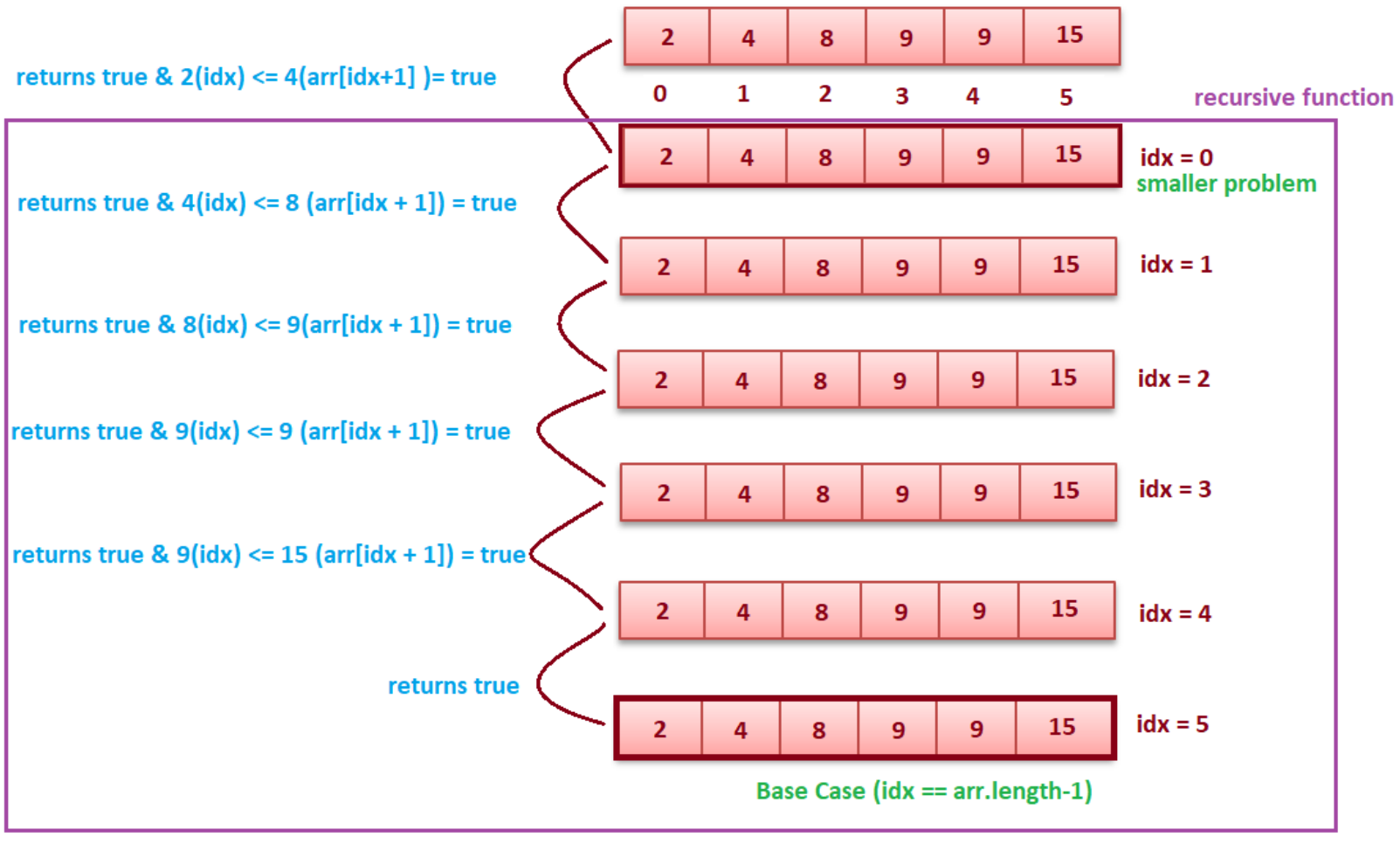
```
function fibonacci(n)
```

```
        //   base case
        if n equals 1 OR 0
                    return n
         //   getting answer of the smaller problem
         recursionResult1 = fibonacci(n - 1)
         recursionResult2 = fibonacci(n - 2)
         //  self work
         ans = recursionResult1 + recursionResult2
         return ans
```

Problem Statement - Check if an array is sorted

For example:

- **If the array is {2, 4, 8, 9, 9, 15}, then the output should be true.**
- **If the array is {5, 8, 2, 9, 3}, then the output should be false.**




```
function isArraySorted(arr, idx)     //   0 is passed in idx
         //   base case
         if idx equals arr.length - 1
                    return true
          //   getting answer of the smaller problem
          recursionResult = isArraySorted(arr, idx+1)

          //   self work
          ans = recursionResult & arr[idx] <= arr[idx+1]
          return ans
```

Divide & Conquer Notes

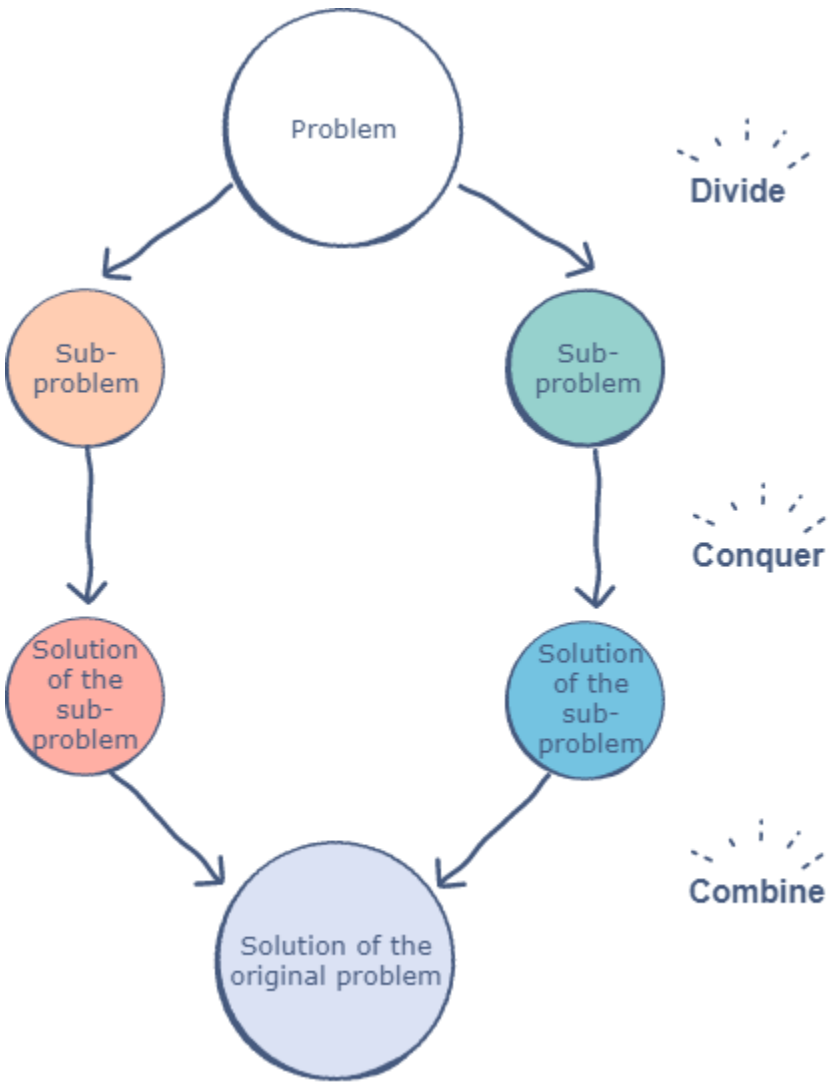
Divide and Conquer

Divide and conquer is an important algorithm design technique based on recursion.The divide and conquer strategy solves the problem by:

- **Divide: breaking the problem into smaller subproblems that are themselves smaller instances of the same type of problem.**
- **Conquer: Conquer the subproblems by solving them recursively.**
- **Combine: Combine the solutions to the subproblems into the solution for the original given problem.**

Divide and Conquer Visualization

Assume that n is the size of the original problem. As described above we can see that the problem is divided into subproblems with each of size n/b (for some constant b). We solve the subproblem recursively and combine the solutions to get the solution for the original problem.




```
DivideAndConquer( P ){
        if(small (P))
                //   P is very small so that the solution is obvious
                return solution( n );

        Divide the problem P into k subproblems P1, P2, P3, …, Pk
        return (
                Combine (
                        DivideAndConquer( P1 ),
                        DivideAndConquer( P2 ),
                        ...
                        DivideAndConquer( Pk )
                )
        )
}
```

Example: Given an array nums of size n, return the majority element. The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

If we know the majority element in the left and right halves of an array, we can determine which is the global majority element in linear time.

Approach: If we know the majority element in the left and right halves, we can find the global majority element. Recurse on left and right halves of the array. The array on which the work is being performed can be denoted by lo and hi. If the left and right halves have the same majority element, then for that slice of the array, the majority element is the one obtained from left and right halves. And if they don't, we count the occurrences of the majority elements of the left and right halves to obtain which half's answer is globally correct. The overall answer for the array is thus the majority element between indices 0 and n.

```
majorityElement(int[] nums) {
        return majorityElementRec(nums, 0, nums.length-1)
 }

countTheMajorityElementInRange(nums, num,  lo, hi) {
        count = 0
        i = lo
        while i < hi
            if (nums[i] == num)
                count += 1

            i += 1

        return count
 }
```

```
majorityElementRec(nums, lo, hi) {
        //    base case; the only element in an array of size 1 is the majority element
        if (lo == hi) {
            return nums[lo]
        }

        //   recurse on left and right halves of this slice.
        mid = (hi-lo)/2 + lo
        leftAns= majorityElementRec(nums, lo, mid)
        rightAns = majorityElementRec(nums, mid+1, hi)

        //   if the two halves agree on the majority element, return it.
        if leftAns equals rightAns
               return leftAns


        //   otherwise, count each element and return the "winner".
        leftCount = countTheMajorityElementInRange(nums, leftAns, lo, hi)
        rightCount =countTheMajorityElementInRange(nums, rightAns, lo, hi)

        if(leftCount > rightCount)
              return leftCount
         return rightCount
 }
```

Time Complexity: O(logn).

Advantages of Divide and Conquer

- **Solving difficult problems: Divide and conquer is a powerful method for solving difficult problems. As an example, consider the Tower of Hanoi problem. This requires breaking the problems into subproblems, solving the trivial cases and combining the subproblems to solve the original problem. Dividing the problem into subproblems so that the subproblem can be combined again is a major difficulty in designing a new algorithm. For many such problems divide and conquer provides a simple solution**
- **Parallelism: Divide and conquer allows us to solve the problems independently, this allows for execution in multi-processor machines, especially shared memory systems where the communication of data between processes does not need to be planned in advance because different subproblems can be executed on different processors.**
- **Memory Access: Divide and conquer algorithm naturally tend to make efficient use of memory caches. This is because once a subproblem is small, all its subproblems can be solved within the cache without accessing the slower main memory.**

Applications of Divide and Conquer

- **Binary Search**
- **Merge Sort and Quicksort**
- **Median Finding**
- **Min and Max finding**
- **Matrix Multiplication**
- **Closest Pair problem**

Backtracking Notes

Introduction to Backtracking

Backtracking is a famous algorithmic-technique for solving/resolving problems recursively by trying to build a solution incrementally. Solving one piece at a time, and removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to as the time elapsed till reaching any level of the search tree) is the process of backtracking.

In other words, Backtracking can be termed as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

There are generally three types of problems in backtracking -

- **Decision Problems: In these types of problems, we search for any feasible solution.**
- **Optimization Problems: In these types of problems, we search for the best possible solution.**
- **Enumerations Problems: In these types of problems, we find all feasible solutions.**

Backtracking and recursion

Backtracking is based on recursion, where the algorithm makes an effort to build a solution to a computational problem

incrementally. Whenever the algorithm needs to choose between multiple possible alternatives, it simply tries all possible options for the solution recursively step-by-step. The usual scenario is that you are faced with a number of options, and you must choose one of these. After you make your choice you will get a new set of options; just what set of options you get depends on what choice you made. This procedure is repeated over and over until you reach a final state. If you made a good sequence of choices, your final state is a goal state; if you didn't, it isn't.

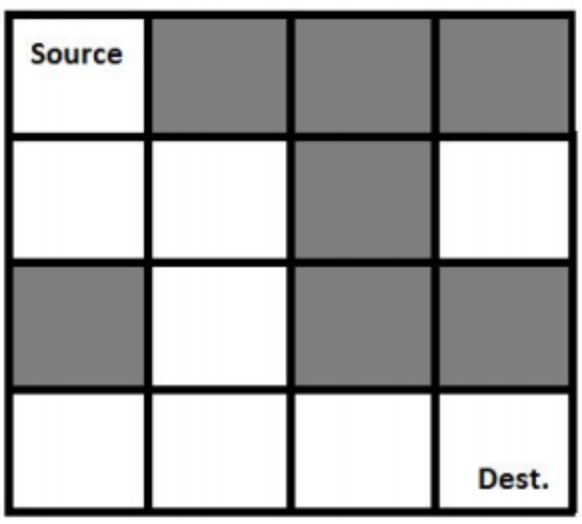
In recursion, the function calls itself until it reaches a base case. In backtracking, we use recursion in order to explore all the possibilities until we get the desired solution for the problem. Backtracking works right after the recursive step, i.e if the recursive step results in a solution that is not desired, it retraces back and looks for other possible options.

Implementation

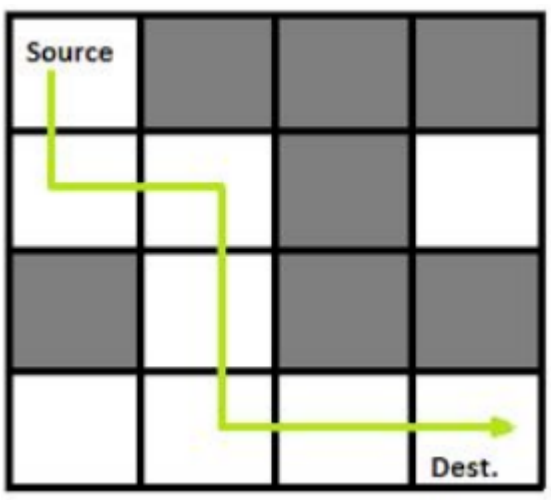
Rat Maze problem

Given a 2D-grid, for simplicity let us assume that the grid is a square matrix of size N, having some cells as free and some as blocked. Given source S and destination D, we need to find whether there exists a path from source to destination in the maze, by traversing through free cells only.

Let us assume that source S is at the top-left corner of the maze and destination D is at the bottom-right corner of the maze, and the movements allowed are to either move to the right or to down.



The black-colored cells represent the blocked cells and white-colored cells represent the free cells.
The possible path from source to destination is -



NOTE: There may be multiple possible paths from source to destination.
We will use backtracking to find whether there exists a path from source to destination in the given maze.

Steps:

- **If the destination point i.e N is reached, return true.**
- **Else**
  - **Check if the current position is a valid position i.e the position is within the bounds of the maze as well as it is a free position.**
  - **Mark the current position as 1, denoting that the position can lead to a possible solution.**
  - **Move forward, and recursively check if this move leads to reach to the destination.**
  - **If the above move fails to reach the destination, then move downward, and recursively check if this move leads to reach the destination.**
  - **If none of the above moves leads to the destination, then mark the current position as 0, denoting that it is not possible to reach the destination from this position.**

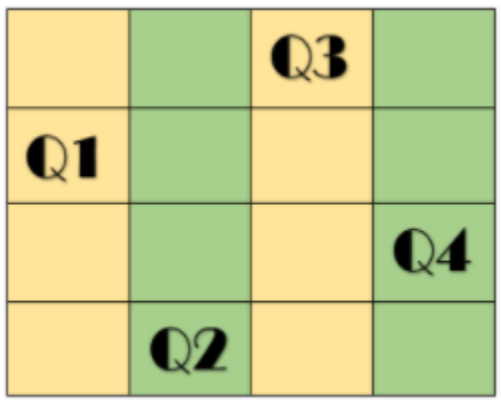
Pseudocode:

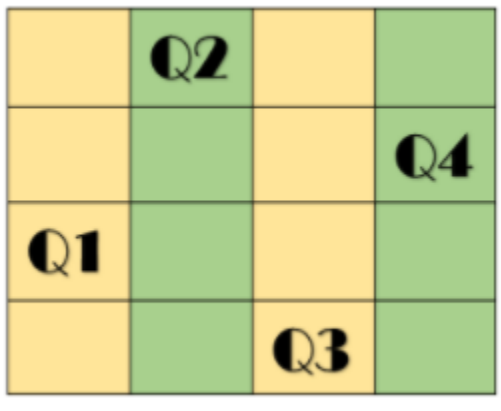
```
function isValid(x, y, N)
```

```
        //Check if the position is within the bounds of the maze and the position does not contain a
blocked cell.

        if(x <= N and y <= N and x, y is not blocked)
                return true
        else
                return false

function RatMaze(maze[][], x, y, N)

        /*
        x, y is the current position of the rat in the maze.
        Check if the current position is a valid position.
        */

        if isValid(x, y, N)
                mark[x][y] = 1
        else
                return false

        /*
        If the current position is the bottom-right corner, i.e N, then we have found a solution
        */

        if x equals N and y equals N
                mark[x][y] = 1
                return true

        /*
        Otherwise, try moving forward or downward to look for other possible solutions.
        */

        bool found = RatMaze(maze,x+1,y,N) OR RatMaze(maze,x,y+1,N)

        /*
        If a solution is found from the current position by moving forward or downward, then return true,
otherwise mark the current position as 0, as it is not possible to
        reach the end of the maze.
        */

        if(found)
                return true
        else
                mark[x][y] = 0
                return false
```

N-Queens Problem

Given an NxN chessboard, we need to arrange N queens on the board in such a way that no two queens attack each other. A queen can attack horizontally, vertically, or diagonally.
Below is a diagram showing how 4 queens can be placed on the chessboard of size 4x4, such that no two queens attack each other.





In the above diagram, the 4 queens are represented by Q1, Q2, Q3, Q4 and it shows the possible valid arrangements of the 4 queens on the chessboard.

NOTE: There can be multiple valid arrangements for the queens on the chessboard.
We will use backtracking to find a valid arrangement of the queens on the given chessboard. We place the first queen arbitrarily anywhere within the board, and then place the next queen in a position that is not attacked by any other queens placed so far, if no such position is present we backtrack and change the position of the previous queens. A solution is found if we are able to place all the queens on the chessboard.

Steps:

- **Place a queen arbitrarily at any position on the chessboard.**
- **Check if this position is safe, i.e it is not attacked by any other queens.**
- **If the position is not safe, then look for other positions on the board, and if no such position is found, then return false as we cannot place any more queens.**
- **If the position is safe, then recursively check for Q-1 queens, if the function returns true, in other words, all queens were placed successfully on the board, then return true.**
- **NOTE: Q denoting the number of queens to be placed, N is passed as the value in the function call, representing the value of Q initially, as we need to place N queens on the board.**

Pseudocode:

```
function isValid(x,y,board[][],N)

        /*
        Check if the position at x,y is not attacked by any other
        queen.
        Check if no position is marked 1 in the same row.
        Check if no position is marked 1 in the same column.
        Check if no position is marked 1 in the diagonals.
        */

        for i = 0 to N - 1
                if board[x][i] equals 1 or board[i][y] equals 1
                        return false

        tempx = x
        tempy = y

        while tempx >= 0 and tempy < N
                if board[tempx][tempy] equals 1
                        return false
                tempx -= 1
                tempy += 1

        tempx = x
        tempy = y

        while tempx < N and tempy >= 0
                if board[tempx][tempy] equals 1
                        return false
                tempx += 1
                tempy -= 1

        tempx = x
        tempy = y

        while tempx >= 0 and tempy >= 0
                if board[tempx][tempy] equals 1
                        return false
                tempx -= 1
                tempy -= 1

        tempx = x
        tempy = y

        while tempx < N and tempy < N
                if board[tempx][tempy] equals 1
                        return false
                tempx += 1
                tempy += 1
```

```
        //   The position is a safe position, return true
        return true

function N-Queens(Q, board[][], N)

        //   Q represents the number of queens to be placed on the board.

        //   Base Case, when all queens have been placed
        if Q equals 0
                return true

        /*
        For each possible position on the board, check if the position is safe i.e it is not attacked by any
other queen placed so far.
        */

        for i = 0 to N - 1
                for j = 0 to N - 1
                        bool can = isValid(i, j, board, N)

                        /*
                        If the position is safe, then mark the position on the board as 1, and check
recursively for Q-1 queens if they can be placed successfully.
                        */

                        if isValid is true
                                board[i][j] = 1
                                bool solve = N - Queens(Q - 1,board,N)

                        /*
                        The remaining queens can be placed successfully, return true, otherwise, unmark the
position on the board, and check for other possible options.
                        */

                                if solve is true
                                        return true
                                else
                                        board[i][j] = 0
                        else
                                continue
        /*
        Since there was no possible option to place a queen on the board, so return false.
        */

        return false
```

Applications of backtracking

- Backtracking is useful in solving puzzles such as the Eight queen puzzle, Crosswords, Verbal arithmetic, Sudoku, and Peg Solitaire.
- The technique is also useful in solving combinatorial optimization problems such as parsing and the knapsack.
- The backtracking search algorithm is used in load frequency control of multi-area interconnected power systems.

Advantages of backtracking

- It is a step-by-step representation of a solution to a given problem, which is very easy to understand.
- It is easy to first develop an algorithm, and then convert it into a flowchart and into a computer program.
- It is very easy to implement and contains fewer lines of code, almost all of them being generally few lines of recursive function code.

Disadvantages of backtracking

- More optimal algorithms for the given problem may exist.
- Very time inefficient in a lot of cases when the branching factor is large.
- Can lead to large space complexities because of the recursive function call stack.

Introduction to Sorting

Sorting

What is Sorting?

Sorting means an arrangement of elements in an ordered sequence either in increasing(ascending) order or decreasing(descending) order. Sorting is very important and many software and programs use this. The major difference is the amount of space and time they consume while being performed in the program.

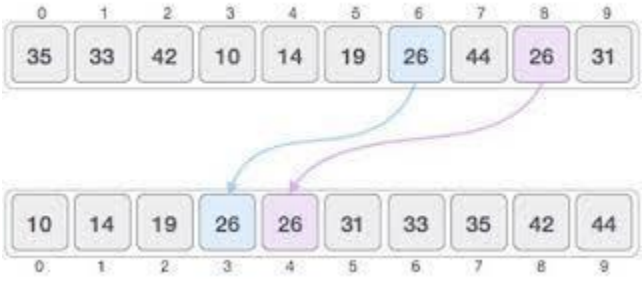
Different languages have their own in-built functions for sorting, which are basically hybrid sorts that are a combination of different basic sorting algorithms. For example, C++ uses introsort, which runs quicksort and switches to heapsort when the recursion gets too deep. This way, you get the fast performance of quicksort in practice while guaranteeing a worst-case time complexity of O(N*logN), where N is the number of elements.

We will be focusing mainly upon the following sorting algorithms:

- **Selection Sort**
- **Bubble Sort**
- **Insertion Sort**
- **Merge Sort**
- **Quick Sort**
- **Counting Sort**

The sorting algorithms are further classified on the following basis:

- **In-place sorting and out-place sorting**
  - **In-place sorting: In-place sorting does not require any extra space except the input space excluding the constant space which is used for variables or iterators. It also doesn’t include the space used for stack in the recursive algorithms. For Example, Bubble Sort, Selection Sort, Insertion Sort, Quicksort, etc.**
  - **Out-place sorting: Out-place sorting requires extra space to sort the elements. For Example, Merge Sort, Counting sort, etc.**
- **Stable and Unstable sorting**
  - **Stable sorting: A sorting algorithm when two objects with equal keys appear in the same order in the sorted output as they appear in the unsorted input. For Example, Insertion Sort, Merge Sort, Bubble Sort, Counting sort, etc.**
  - **Unstable sorting: A sorting algorithm is called unstable sorting if there are two or more objects with equal keys which don’t appear in the same order before and after sorting. For Example, Quick Sort, Heap Sort, Selection Sort, etc.**



Above is an example of stable sorting here number 26 is appearing two times at positions 6 and 8 respectively in an unsorted array and their order of occurrence is preserved in the sorted array as well and sorted array i.e. element 26 (blue) at position 6 in unsorted array appears first in sorted array followed by another 26 (red) at position 8.

- **Adaptive and Non-adaptive sorting**
  - **Adaptive sorting: When the order of occurrence of elements in an array affects the time complexity of a sorting algorithm, then such an algorithm is known as an adaptive sorting algorithm. For Example, Bubble sort, Insertion Sort, Quick Sort, etc.**
  - **Non-adaptive sorting: When the order of occurrence of elements in an array does not affect the time complexity of a sorting algorithm, then such an algorithm is known as a non-adaptive sorting algorithm. For Example, Selection Sort, Heap Sort, Merge Sort, etc.**

Selection Sort

Selection Sort

Algorithm:

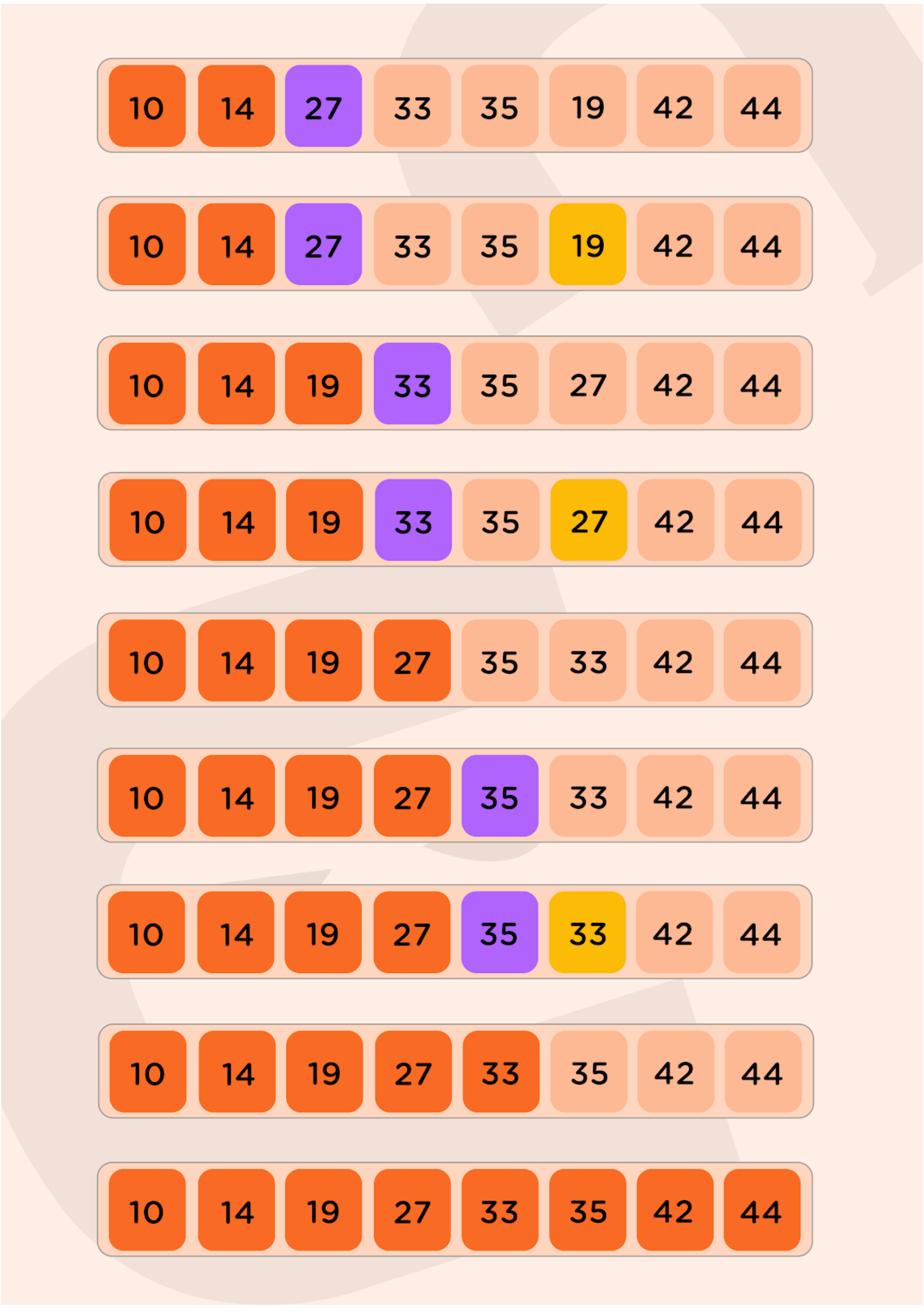
Steps: (sorting in increasing order)

- **First-of-all, we will find the smallest element of the array and swap it with index 0.**
- **Similarly, we will find the second smallest and swap that with the element at index 1 and so on…**
- **Ultimately, we will be getting a sorted array in increasing order only.**

Let us look at the example for better understanding:

Consider the following depicted array as an example. You want to sort this array in increasing order.

Following is the pictorial diagram for a better explanation of how it works:



This is how we obtain the sorted array at last.

Pseudocode:

```
/*
        array of size N from 0 to N-1 is considered
*/
function selectionSort(arr, N)

        for idx = 0 to N-2
                //   Initialize minimum value to be present at idx
                minIdx = idx
                for jdx = idx+1 to N-1
                        //   If a new minimum is found, set minIdx to jdx
                        if arr[jdx] < arr[minIdx]
                                minIdx = jdx

                //   Finally swap the minimum found from idx to N-1 with idx
                swap(arr[idx], arr[minIdx])
```

Time complexity: O(N^2), in the worst case.
As to find the minimum element from the array of 'N' elements, we require 'N-1' comparisons, then after putting the minimum element in the correct position, we repeat the same for the unsorted array of the remaining 'N-1' elements, performing 'N-2' comparisons and so on.

So, total number of comparisons : N-1 + N-2 + N-3 + … + 1 = (N*(N-1))/2 and total number of exchanges(swapping): N-1
So, time complexity becomes O(N^2).

Space complexity: O(1), as no extra space is required.
Bubble Sort

Bubble Sort

Algorithm:

In selection sort, the elements from the start get placed at the correct position first and then the further elements, but in the bubble sort, the elements start to place correctly from the end.

In this technique, we just compare the two adjacent elements of the array and then sort them manually by swapping if not sorted. Similarly, we will compare the next two elements (one from the previous position and the corresponding next) of the array and sort them manually. This way the elements from the last get placed in their correct position. This is the difference between selection sort and bubble sort.

Consider the following depicted array as an example. You want to sort this array in increasing order.
Arr[] = [50,25,5,20,10]

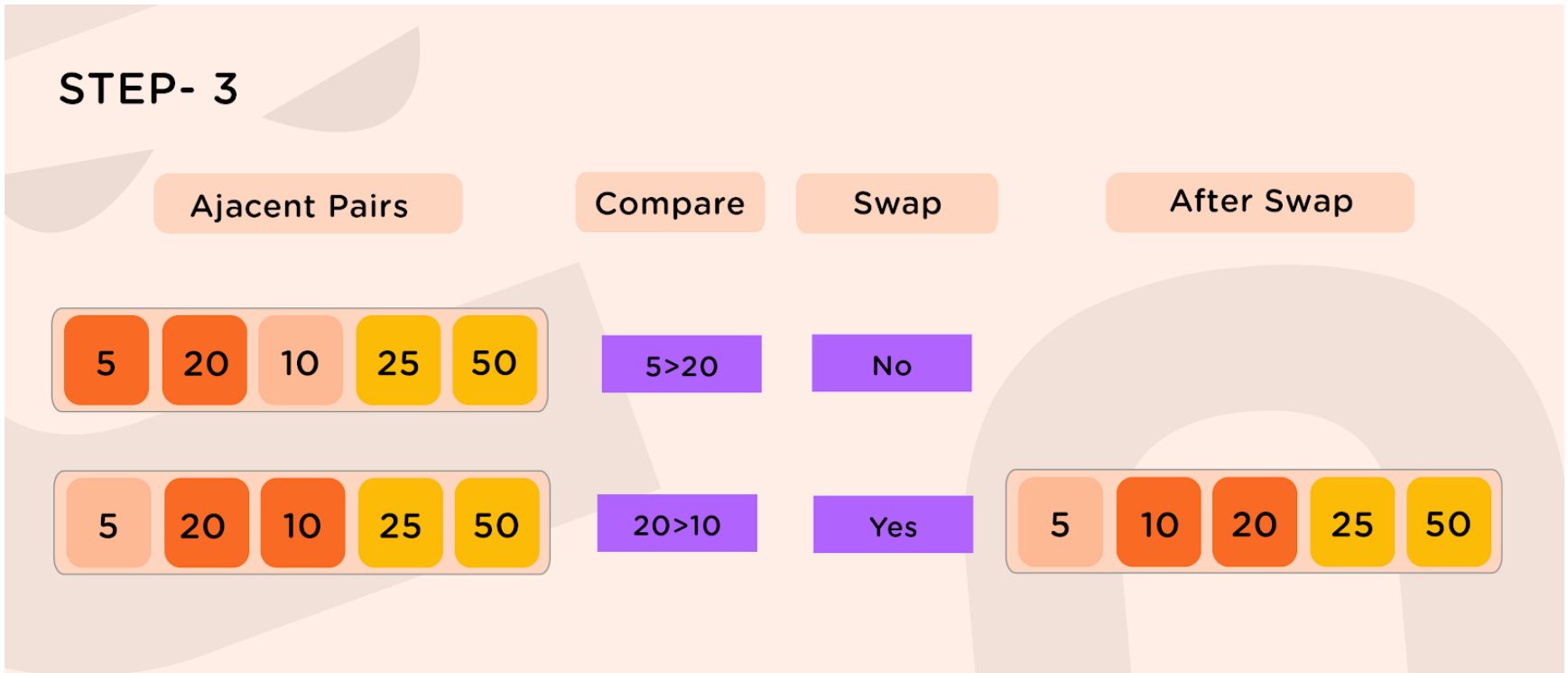
Following is the pictorial diagram for a better explanation of how it works:

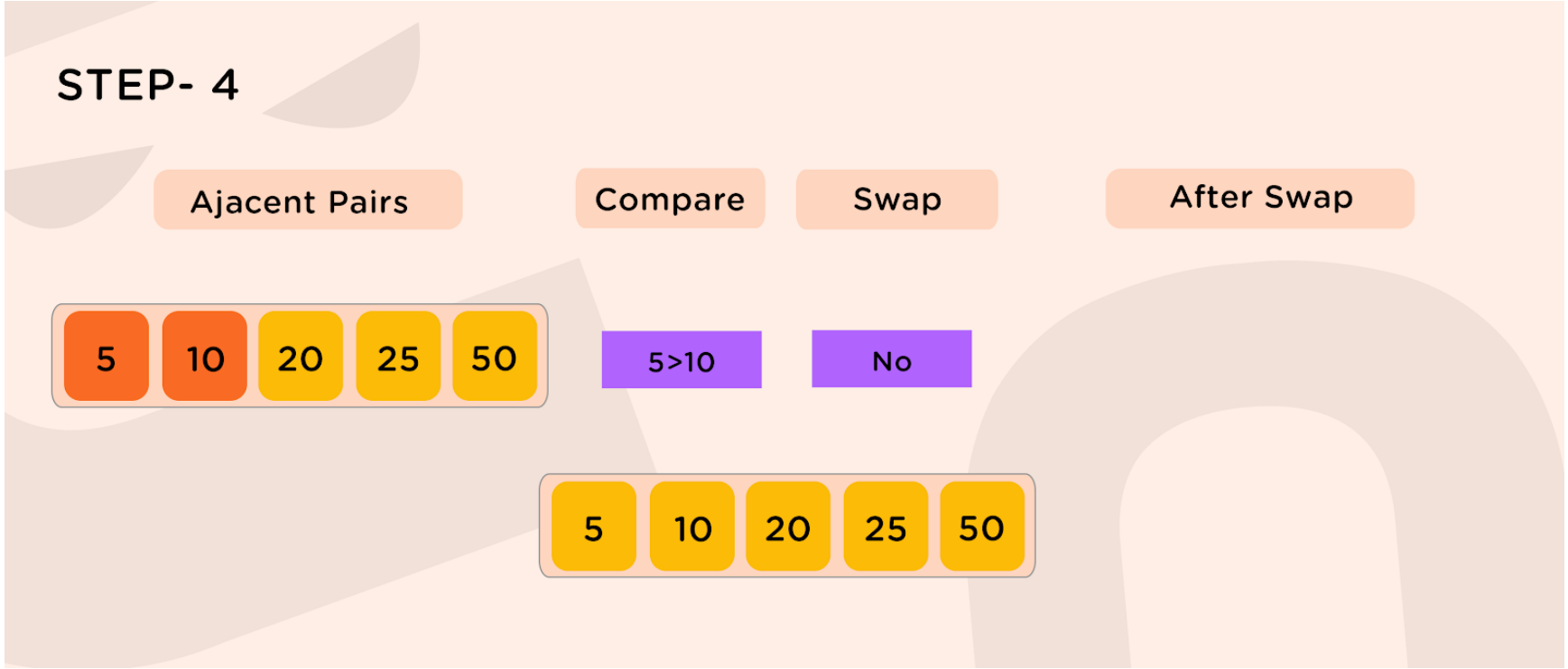
STEP- 1

Ajacent Pairs	Compare	Swap	After Swap
50 25 5 20 10	50>25	Yes	25 50 5 20 10
25 50 5 20 10	50>5	Yes	25 5 50 20 10
25 5 50 20 10	50>20	Yes	25 5 20 50 10
25 5 20 50 10	50>10	Yes	25 5 20 10 50

STEP- 2

Ajacent Pairs	Compare	Swap	After Swap
25 5 20 10 50	25>5	Yes	5 25 20 10 50
5 25 20 10 50	25>20	Yes	5 20 25 10 50
5 20 25 10 50	25>10	Yes	5 20 10 25 50





Pseudocode:

```
/*
        array of size N from 0 to N-1 is considered
*/
function bubbleSort(arr, N)

        for idx = 0 to N-2
                //   Last idx elements are already sorted
                for jdx = 0 to N-idx-2
                        if arr[jdx] > arr[jdx+1]
                                swap(arr[jdx],arr[jdx+1])
```

Time complexity: O(N^2), in the worst case.
For every element, we iterate over the entire array each time giving an overall time complexity of O(N^2).

Space complexity: O(1), as no extra space is required.
Insertion Sort

Insertion Sort

Algorithm:

Insertion Sort works similar to how we sort a hand of playing cards.

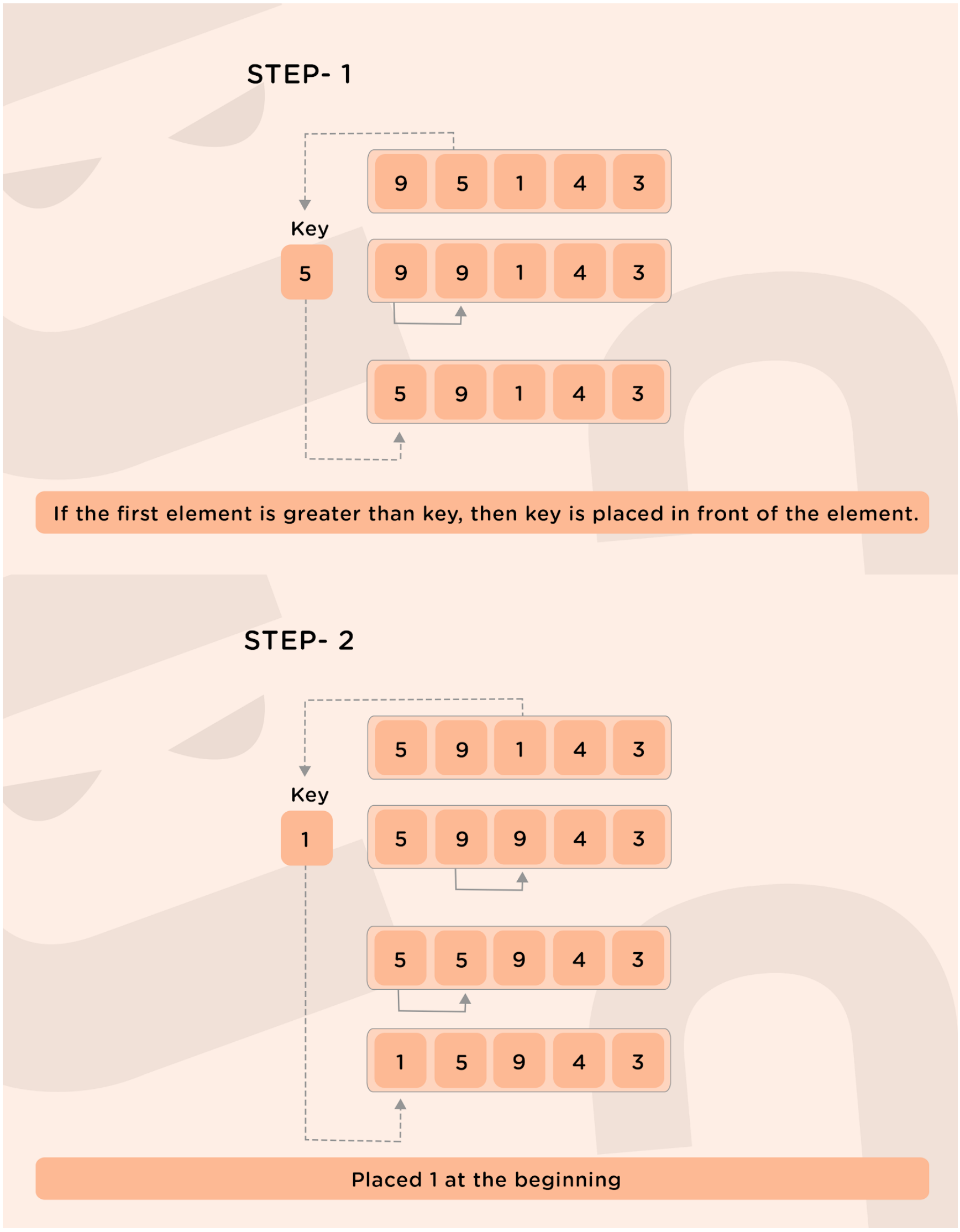
Imagine that you are playing a card game. You're holding the cards in your hand, and these cards are sorted. The dealer hands you exactly one new card. You have to put it into the correct place so that the cards you're holding are still sorted. In selection sort, each element that you add to the sorted subarray is no smaller than the elements already in the sorted subarray.
But in our card example, the new card could be smaller than some of the cards you're already holding, and so you go down the line, comparing the new card against each card in your hand until you find the place to put it. You insert the new card in the right place, and once again, your hand holds fully sorted cards. Then the dealer gives you another card, and you repeat the same procedure. Then another card, and another card, and so on, until the dealer stops giving you cards. This is the idea behind

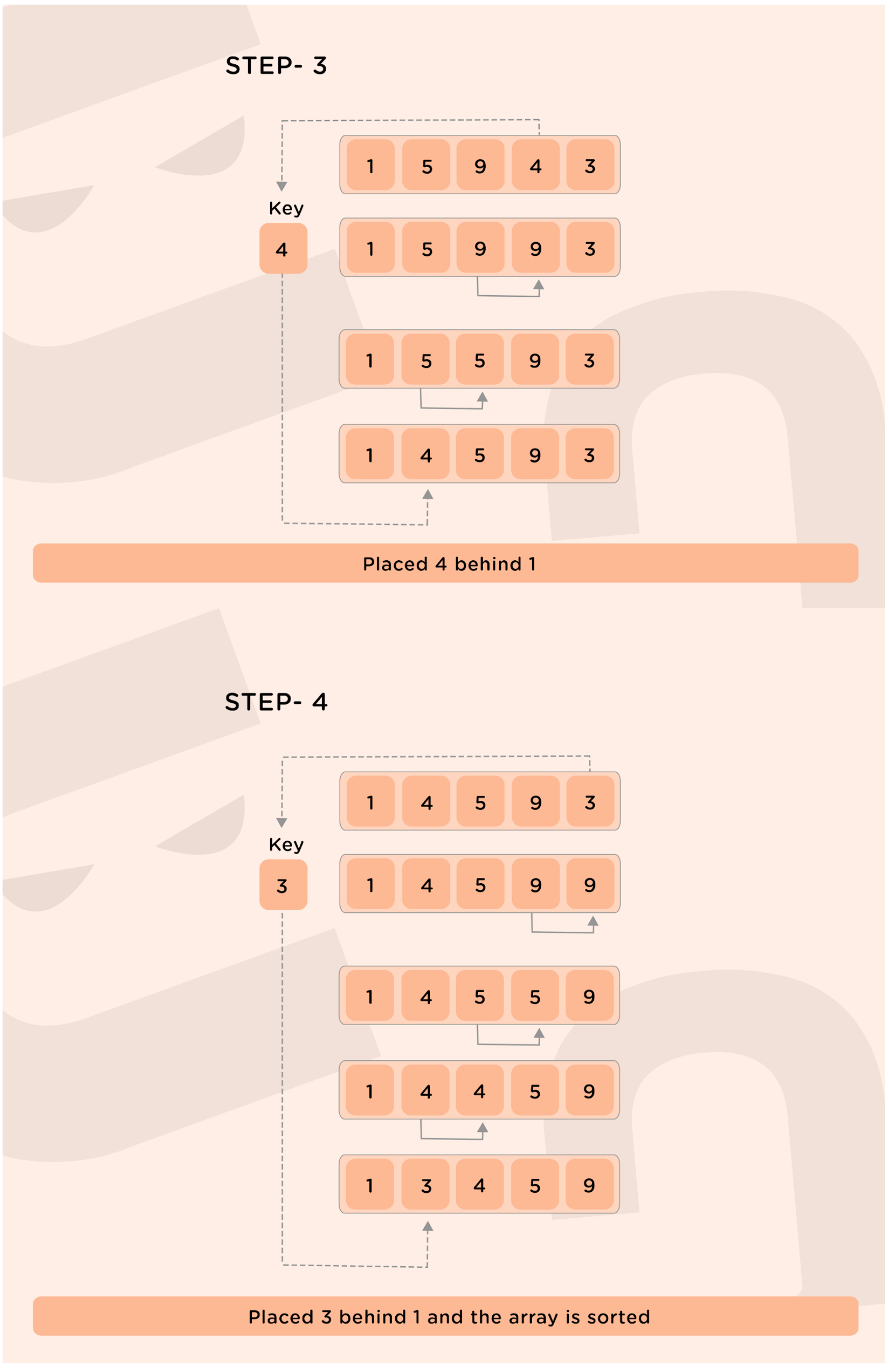
the insertion sort. Loop over positions in the array, starting with index 1. Each new position is like the new card handed to you by the dealer, and you need to insert it into the correct place in the sorted subarray to the left of that position.

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Consider the following depicted array as an example. You want to sort this array in increasing order.
Arr[] = [9,4,5,1,3]

Following is the pictorial diagram for a better explanation of how it works:




STEP- 3
Key
4
1 5 9 4 3
1 5 9 9 3
1 5 5 9 3
1 4 5 9 3
Placed 4 behind 1
STEP- 4
Key
3
1 4 5 9 3
1 4 5 9 9
1 4 5 5 9
1 4 4 5 9
1 3 4 5 9
Placed 3 behind 1 and the array is sorted

Pseudocode:

```
/*
```

```
        array of size N from 0 to N-1 is considered
*/
function insertionSort(arr, N)

        for idx = 1 to N-1
                cur = arr[idx]
                //   Finding the appropriate position for the element from 0 to idx-1.
                jdx = idx-1
                while arr[jdx] > cur and jdx >=0
                        //   Creating space for the element at idx
                        arr[jdx] = arr[jdx+1]
                        jdx -= 1

                //   Placing the element at idx, making the array sorted from 0 to idx
                arr[jdx+1] = cur
```

Time complexity: O(N^2), in the worst case.
As for finding the correct position for the ith element, we need i iterations from 0 to i-1th index, so the total number of comparisons = 1+2+3+ … + N-1 = (N*(N-1))/2 and the total number of exchanges in the worst case: N-1.
So, the overall complexity becomes O(N^2).

Space complexity: O(1), as no extra space is required.
Merge Sort

Merge Sort

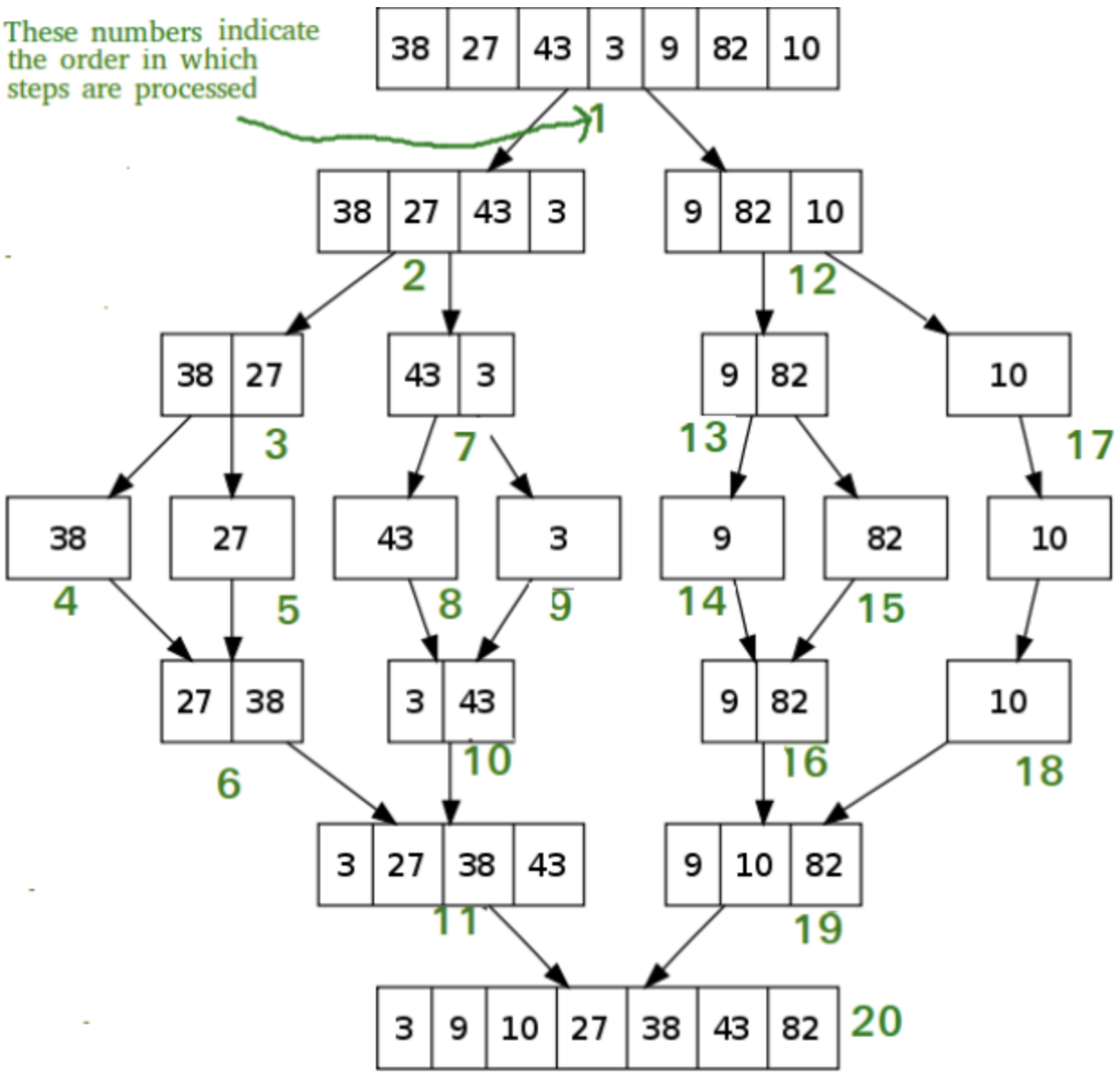
Algorithm:

Merge sort requires dividing a given list into equal halves until it can no longer be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines/merges the smaller sorted lists keeping the new list sorted too.

- **If it is only one element in the list it is already sorted, return.**
- **Divide the list recursively into two halves until it can’t be divided further.**
- **Merge the smaller lists into a new list in sorted order.**

It has just one disadvantage that it creates a copy of the array and then works on that copy.

The following diagram shows the complete merge sort process for an example array [38,27,43,3,9,82,10]. If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



Pseudocode:

```
/*
        array from leftidx(0) to rightidx(arr.length-1) is considered
*/
function mergeSort(arr, leftidx , rightidx )

          //   base case : only 1 element to be sorted
          if  leftidx == rightidx
                      return

           middle = (leftidx + rightidx) / 2

           //   smaller problems
           mergerSort(arr, leftidx, middle)
           mergeSort(arr, middle + 1, rightidx)

           //   selfwork
           merge(leftidx , middle, rightidx)


function merge(arr, leftidx , middle , rightidx)

          leftlo = leftidx
          rightlo = middle+1
          idx = 0
          //   create an array temp of size (rightidx - leftidx + 1)

          while leftlo <= middle AND rightlo <= rightidx
                    if   arr[leftlo] < arr[rightlo]
                                temp[idx] = arr[leftlo]
                                leftlo++
                                idx++
                    else
                                temp[idx] = arr[rightlo]
                                rightlo++
                                idx++
```

```
        while leftlo <= middle
                    temp[idx] = arr[leftlo]
                    leftlo++
                    idx++


        while rightlo <= rightidx
                    temp[idx] = arr[rightlo]
                    rightlo++
                    idx++


         //   copy temp to original array
         count = 0
        while count < temp.length AND leftidx <= rightidx
                    arr[leftidx] = temp[count]
                    leftidx++
                    count++
```

Time complexity: O(N*logN), in the worst case.

N elements are divided recursively into 2 parts, this forms a tree with nodes as divided parts of the array representing the subproblems. The height of the tree will be log2N and at each level of the tree, the computation cost of all the subproblems will be N. At each level the merge operation will take O(N) time. So the overall complexity comes out to be O(N*logN).

Space complexity: O(N), as extra space is required to sort the array.
Quick Sort

Quick Sort

Algorithm:

Based on the Divide-and-Conquer approach, the quicksort algorithm can be explained as:

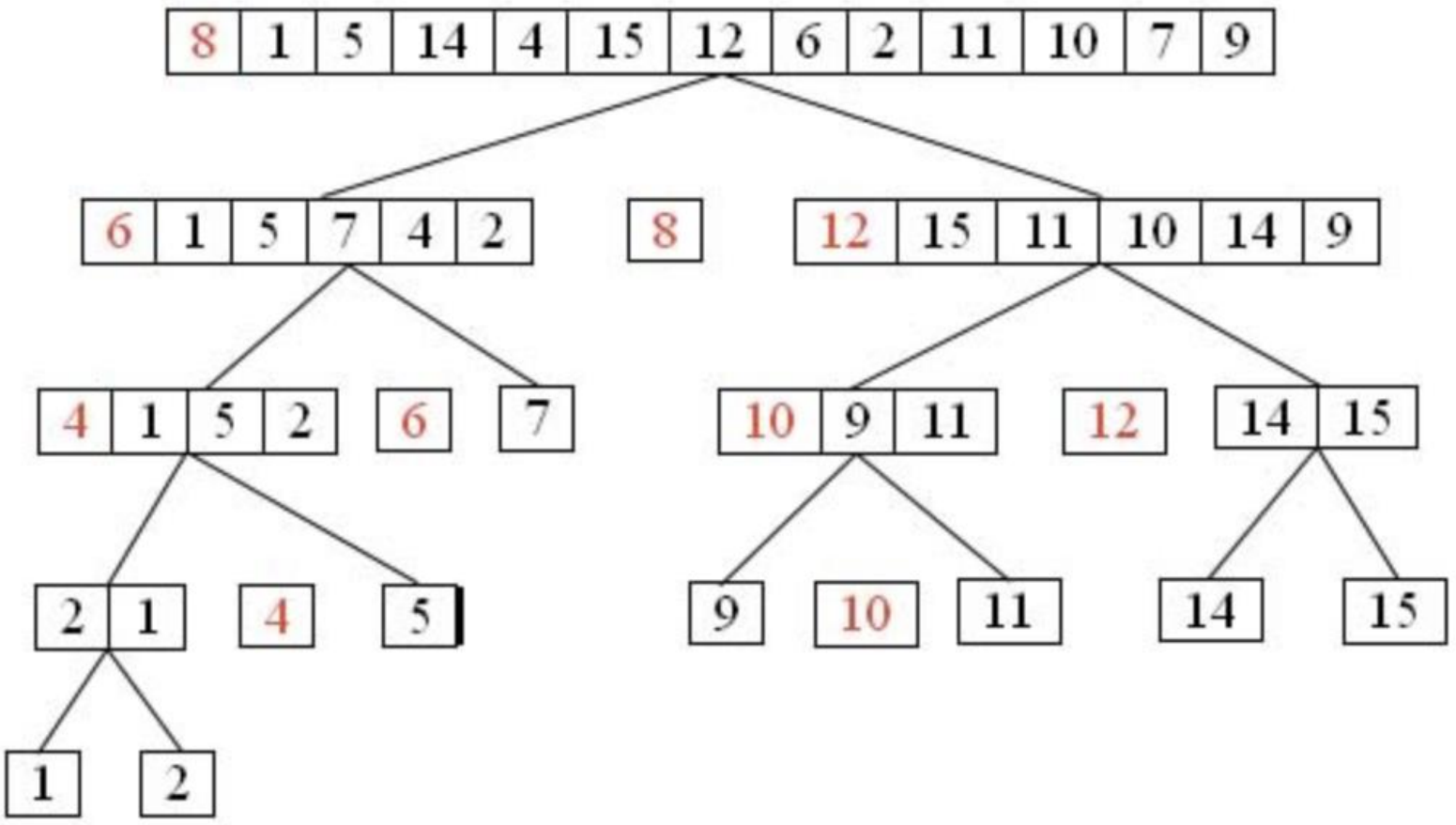
- **Divide: The array is divided into subparts taking pivot as the partitioning point. The elements smaller than the pivot are placed to the left of the pivot and the elements greater than the pivot are placed to the right side.**
- **Conquer: The left and right sub-parts are again partitioned using the by selecting pivot elements for them. This can be achieved by recursively passing the subparts into the algorithm.**
- **Combine: This part does not play a significant role in quicksort. The array is already sorted at the end of the conquer step.**

The advantage of quicksort over merge sort is that it does not require any extra space to sort the given array.

There are many ways to pick a pivot element:

- **Always pick the first element as the pivot.**
- **Always pick the last element as the pivot.**
- **Pick a random element as the pivot.**
- **Pick the middle element as the pivot.**

The following diagram shows the complete quick sort process, by considering the first element as the pivot element, for an example array [8,1,5,14,4,15,12,6,2,11,10,7,9].



1	2	4	5	6	7	8	9	10	11	12	14	15

- **In step 1, 8 is taken as the pivot.**
- **In step 2, 6 and 12 are taken as pivots.**
- **In step 3, 4 and 10 are taken as pivots.**
- **We keep dividing the list about pivots till there are only 2 elements left in a sublist.**

Pseudocode:

```
/*
        array from lo(0) to hi(arr.length-1) is considered
*/
function quickSort(arr, lo , hi )

       if (lo >= hi)
        return

       pivot = arr[lo]

       //   partitioning
       left = lo
       right = hi

       while left <= right

        //   move left to a problem
        while  arr[left] < pivot
               left++


        //   move right to a problem
        while  arr[right] > pivot
               right--


        //   problem solve : swap
        if  left <= right
               temp = arr[left]
               arr[left] = arr[right]
```

```
                arr[right] = temp
                left++
                right--


        //   smaller problems
        quickSort(arr, lo, right)
        quickSort(arr, left, hi)
```

Time complexity: O(N^2), in the worst case.
But as this is a randomized algorithm, its time complexity fluctuates between O(N2) and O(N*log N) and mostly it comes out to be O(N*logN).

Space complexity: O(1), as no extra space is required.
Counting Sort

Counting Sort

Algorithm:

Counting sort is a sorting algorithm that sorts the elements of the array by obtaining the frequencies of each unique element in the array. The frequency of each such unique element of the array is maintained with the help of an auxiliary array and sorting is done by mapping the count(frequency) as an index of the auxiliary array.

Let us say, we need to sort an array A[] of size N, we define the auxiliary array Aux[] of size max(A[i]) where 0<=i<=N-1 i.e the size of the auxiliary array should be equal to the - maximum element of the array A[]+1.

Steps:

- **Initialize the auxiliary array Aux[] as 0 as it is responsible for keeping the count of the elements of the array.**
- **Now traverse the array A[], and store the count of occurrence of the elements of the array at the appropriate index of the auxiliary array Aux[], in other words, increment Aux[A[i]] by 1, for each i from 0 to N-1.**
- **Now store the cumulative sum of the elements of the Aux[], i.e Aux[i] = Aux[i] + Aux[i-1] for each i from 1 to size(Aux[]) -1. This will help in determining the index of the element of the given array A[] when sorted.**

The above approach works because we are basically keeping track of the positions of the element represented by the index of the auxiliary array Aux[] will be present when A[] is sorted.

For example:
Let A[] = [1, 3, 1, 3, 2]

Defining an auxiliary array Aux[] of size 4.
Aux[] = [0, 2, 1, 2]
Index 0 represents the number of elements in the array having value 0.
Index 1 represents the number of elements in the array having value 1.
Index 2 represents the number of elements in the array having value 2.
Index 3 represents the number of elements in the array having value 3.

Doing a cumulative sum(prefix sum) of auxiliary array Aux[]:
Aux[] = [0, 2, 3, 5]

Now as we iterate over the array A[] again, as we encounter an element A[i], the auxiliary array tells the number of A[i]'s remaining to be explored and because of the prefix sum elements at lower indices will always be present before the elements at higher indices

- **Now, iterate over the array A[] again, and for an element, A[i], obtain the index of the element from Aux[A[i]] and place the element A[i] in the output array at index Aux[A[i]] and decrement Aux[A[i]] by 1. The resulting output array will be the required sorted array.**
- **Note:**
    - **The limitation of the algorithm is that it is not a comparison-based sorting algorithm, with space proportional to the maximum element of the array. So if the maximum element of the array is quite large, then counting sort may not work because of memory limits.**
    - **Counting sort is efficient if the range of input data is not much larger than the number of elements in the array. For example, The number of elements to be sorted are 5 but lie within the range of 0 to 125, so this essentially makes our algorithm to be of the order of 5^3. This performance can become even worse if the order of difference between the number of elements to be sorted and their range is high.**
    - **Counting sort can be extended to work for negative numbers too.**

Pseudocode:

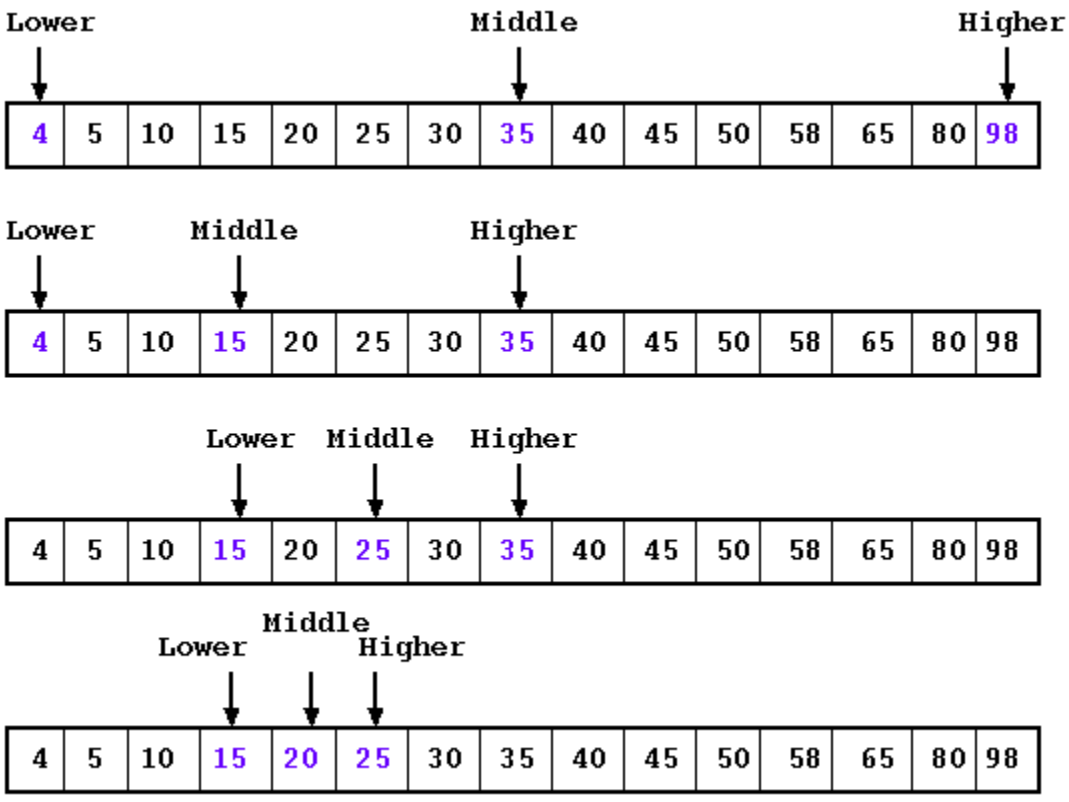
```
/*
        array of size N from 0 to N-1 is considered
*/
function countingSort(arr, N)

        /*
                Obtaining the max element of the array arr to get the size of the auxiliary array.
        */
        aux_size = 0
        for idx = 0 to N-1
                if aux_size < arr[i]
                        aux_size = arr[i]
        /*
                Declaring an auxiliary array of size aux_size and an output array of
                size N
        */
        aux[aux_size+1]
        output[N]

        //   Initializing aux array to 0
        for idx = 0 to aux_size
                aux[idx] = 0

        //   Storing the frequencies of the elements of the array arr
        for idx = 0 to N-1
                aux[arr[i]] = aux[arr[i]]+1

        //   Cumulative sum (prefix sum) on auxiliary array
        for idx = 1 to aux_size
                aux[idx] = aux[idx] + aux[idx-1]

        //   Building the output array
        for idx = 0 to N-1
                output[aux[arr[idx]] - 1] = arr[idx]
                aux[arr[idx]] = aux[arr[idx]] - 1

        return output
```

Time complexity: O(N+K), in the worst case.
Traversing the given array of size N and an auxiliary array of size K i.e the range of the input.

Space complexity: O(N+K)
We need an extra auxiliary array of size K (range of input) and an output array of size N.
Binary Search Notes

Introduction

Binary Search is an algorithmic technique where we make two partitions of the problem and then throw away one instance of it based on a certain condition. The primary condition for using the Binary Search technique is monotonicity. When we can prove that the value of a boolean condition will be true for some time and then become false for the rest of the search space, or vice versa, it often implies that Binary Search can be used there.



Binary Search in Sorted Array

Problem Statement: You are given an array of n elements which is sorted. You are also given an element target which you need to find out whether it is present in the array or not.
We will first discuss the naive algorithm of Linear search and then we will exploit the property that the given array is sorted using Binary Search.

Linear Search

It is a simple sequential search over all the elements of the array, and each element is checked for a match. If a match is found, return the element, otherwise, the search continues until we reach the end of the array.
Pseudocode:

```
/*
        array from leftidx(0) to rightidx(arr.length-1) is considered
*/
function linearSearch(arr, leftidx , rightidx , target)

        // Search for the target from the beginning of arr
        for idx = 0 to arr.length-1
                if arr[idx] == target
                        return idx
        //   target is not found
        return -1
```

Time complexity: O(N), as we traverse the array only once to check for a match for the target element.
Space complexity: O(1), as no extra space is required.

Binary Search

Suppose you want to find a particular element in the sorted array, then following this technique, you have to traverse all the array elements for searching one element but guess if you only have to search at most half of the array indices for performing the same operation. This can be achieved through binary search.
Here consider the monotonic condition: Uptill certain index the value of all elements will be <= x and after that, the value of all elements will be > x. So condition <= is monotonic in our case. Hence we can apply Binary search!
Let us consider the array:

0	1	2	3	4
1	2	3	4	5

Given an array of size 5 with elements inside the boxes and indices above them. Our target element is 2.
Steps:

1. **Find the middle index of the array.**
2. **Now, we will compare the middle element with the target element. In case they are equal then we will simply return the middle index.**
3. **In case they are not equal, then we will check if the target element is less than or greater than the middle element.**

- **In case, the target element is less than the middle element, it means that the target element, if present in the given array, will be on the left side of the middle element as the array is sorted.**

- **Otherwise, the target element will be on the right side of the middle element.**

1. **This helps us discard half of the length of the array each time and we reduce our search space to half of the current search space.**

In the example above, the middle element is 3 at index 2, and the target element is 2 which is less than the middle element, so we will move towards the left part. Now marking start = 0, and end = n/2-1 = 1, now middle = (start + end)/2 = 0. Now comparing the 0-th index element with 2, we find that 2 > 1, hence we will be moving towards the right. Updating the start = 1 and end = 1, the middle becomes 1, comparing the 1-st index of the array with the target element, we find they are equal, meaning from here we will simply return 1 (the index of the element).

Recursive Pseudocode:

```
/*
        array of size N from 0 to N-1 is considered
*/
function binarySearch(arr, N, target, lo , hi)

        // base condition if element is not found
        if lo > hi
                return -1
        // Finding the mid of search space from lo to hi
        mid = lo + (hi-lo)/2
        //  If the target element is present at mid
        if arr[mid] == target
                return mid
        /*
                If the target element is less than arr[mid], then if the
                target is present, it must be in the left half.
        */
        if target < arr[mid]
                return binarySearch(arr, N, target, lo, mid - 1)
        // Otherwise if the target is present, it must be in the right half
        else
                return binarySearch(arr, N, target, mid + 1, hi)
```

Iterative Pseudocode:

```
/*
        array of size N from 0 to N-1 is considered
*/
function binarySearch(arr, N, target)

        // Initializing lo and hi pointers
        lo = 0
        hi = N-1
        // Searching for the target element until lo<=hi
        while lo <= hi
                // Finding the mid of search space from lo to hi
                mid = lo + (hi-lo)/2
                //   If the target element is present at mid
                if arr[mid] == target
                        return mid
                /*
                        If the target element is less than arr[mid], then if the
                        target is present, it must be in the left half.
                */
                if target < arr[mid]
                        hi = mid-1

                else
                        // Otherwise if the target is present, it must
                        // be in the right half
                        lo = mid+1

        // If the target is not found return -1
        return -1
```

Time complexity: O(logN), where N is the number of elements in the array, given the array is sorted. Since we search for the target element in one of the halves every time, reducing our search space to half of the current search space.
Since we go on searching for the target until our search space reduces to 1, so
Iteration 1- Initial search space: N

Iteration 2 - Search space: N/2
Iteration 3 - Search space: N/4
Let after ‘k’ iterations search space reduces to 1
So, N/(2k) = 1
=> N = 2k
Taking Log2 on both sides:
=> k = log2N
Hence, the maximum number of iterations ‘k’ comes out to be log2N.

Space complexity: O(1), as no extra space is required.

Generic Binary Search

It is easier to think about binary search when solving a more general problem than finding an element in a sorted list. It is quite a handy algorithm when you are trying to solve a problem that has the following properties. The answer to the problem is a number. You don’t know the answer, but given the answer, you can code the algorithm to check if the number is an answer to the problem in a relatively straightforward way. So, essentially you have a predicate function ok(x) that returns true when x is a potential solution. Your task is to find the minimal x so that ok(x) returns true.

There is one crucial property that needs to be satisfied to apply a binary search algorithm. The predicate has to be *monotonic*. It means that if x is the lowest number making the predicate true, then it is also true for all the larger values of x. For example, the following table shows the values of a monophonic predicate over integers in the range from 1 to 6:

```
 x  :   1 2 3 4 5 6
ok(x) :  F F T T T T
```

In a binary search algorithm, we introduce two variables. One variable keeps the index of the left side of the currently searched range and is commonly shortened to (l), while the second one keeps the index of the right side (r). The algorithm maintains the following *invariant* throughout its operation*:*

- **ok(l) is false**
- **ok(r) is true**

Binary search is a primitive example of a *divide and conquer* algorithm. It repeatedly halves the range from l to r while maintaining this invariant until l and r only differ by one, thus exactly pinpointing the indices at which monotonic predicate turns from false to true:

```
 x : 1 2 3 4 5 6
ok(x) : F F T T T T
        ^ ^
        l  r    // at the end
```

So the whole solution is to run the following loop:

```
while (r - l > 1)
        // compute the midpoint of l..r interval
        m = (l + r) / 2
        if (ok(m))
        // Maintain invariant: ok(r) is true
                r = m
        else
        // Maintain invariant: ok(l) is false
                l = m
```

If you keep the loop invariant in mind when writing this code it is easy to see what exactly needs to be put into each of the branches of the if statement. No chance for any kind of off-by-one errors.

Advantages of Binary search:

1. **This searching technique is fast and easier to implement.**
2. **Requires no extra space.**
3. **Reduces time complexity of the program to a greater extent i.e O(logN), where N is the number of the elements in the search space.**

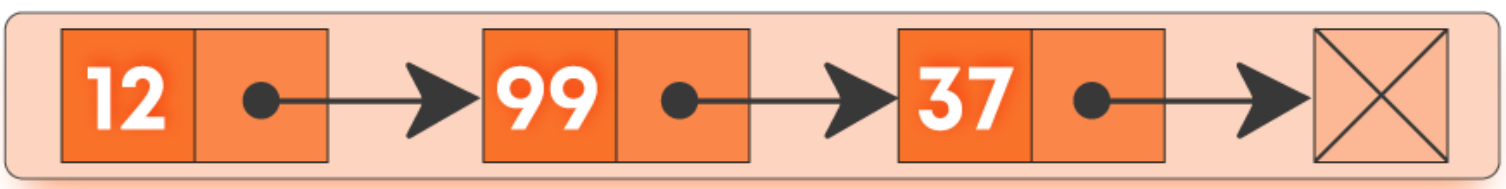
Disadvantages of Binary search:

1. **Binary Search can be only applied to monotonic problem space. For example, if the array is not sorted, we need to go back to the linear search or sort the array, both of which are very expensive operations**
2. **Binary search can only be applied to data structures that allow direct access to elements. If we do not have direct access then we can not apply binary search on it eg. we can not apply binary search to Linked List.**

Linked List Reversal Notes

Introduction to linked lists

A linked list is a collection of nodes in non-contiguous memory locations where every node contains some data and a pointer to the next node of the same data type. In other words, the node stores the address of the next node in the sequence. A singly linked list allows traversal of data only in one way.



Following are the terms used in Linked Lists :

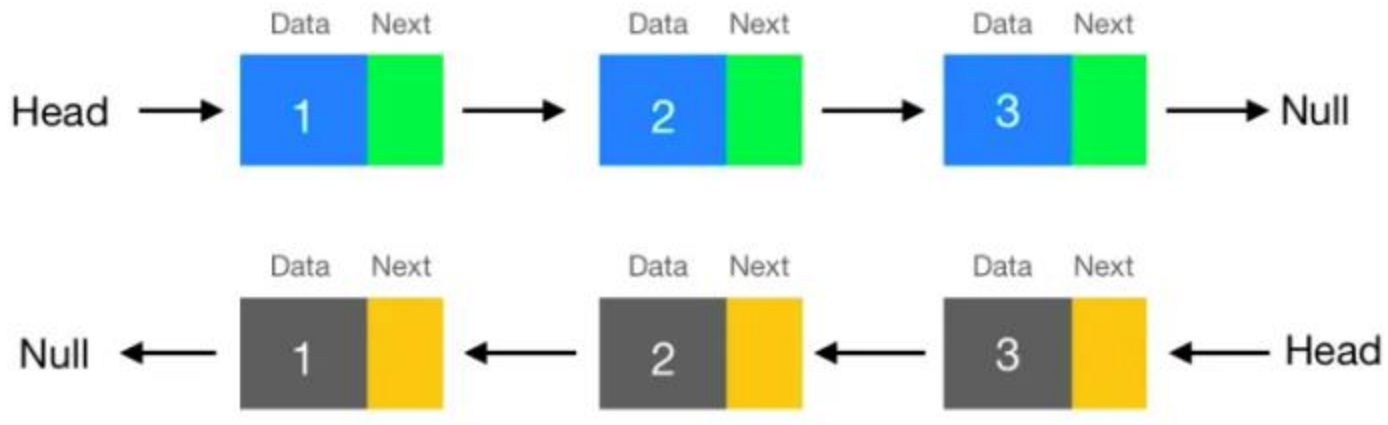
- **Node: A node in a singly linked list contains two fields -**
  - **Data field which stores the data at the node**
  - **A pointer that contains the address of the next node in the sequence.**
- **Head: The first node in a linked list is called the head. The head is always used as a reference to traverse the list.**
- **Tail: The last node in a linked list is called the tail. It always contains a pointer to NULL (since the next node is NULL), denoting the end of a linked list.**

Properties of Linked Lists

- **A linked list is a dynamic data structure, which means the list can grow or shrink easily as the nodes are stored in memory in a non-contiguous fashion.**
- **The size of a linked list is limited to the size of memory, and the size need not be declared in advance.**
- **Note: We must never lose the address of the head pointer as it references the starting address of the linked list and if lost, would lead to loss of the list.**

Linked List Reversal

We first look at the standard algorithm for reversing a Linked list. Given a Linked List we want to reverse it and then return the pointer to the first node of the reversed list.



Recursive Approach

Algorithm:

1. **We divide the linked list of N nodes into two parts. i.e head and rest of the Linked List with (N-1) nodes.**
2. **Now recursively reverse the (N-1) nodes of Linked List and return the head of this part i.e rest. After the reversal, the next node of the head will be the last node of the reversed Linked List and the head will be pointing to this node.**
3. **But for the complete reversal of the Linked List, the head should be the last node. So, we do the following:**
    1. **head.next.next = head, where head.next is the last node of the reverse Linked List.**
    2. **head.next = NULL**
    4. **Return the head pointer of the reversed Linked List i.e. return rest.**

```
function reverseLL(head)
        // Base condition
        if head is null or head.next is null
        //      Return the last node.
                return head

        // Reverse the rest of Linked List
        rest = reverseLinkedList(head->next)
        // Changing the reference of next node next to itself

        head->next->next = head
        // Assign current node next to NULL.
        head->next = NULL

        // Return the reverse Linked List.
        return rest
```

Time Complexity: O(N), where N is the number of nodes in the Linked List. In the worst case, we are traversing the whole Linked List O(N) using recursion, Hence, the overall complexity will be O(N).

Space Complexity: O(N), where N is the total number of nodes in the Linked List. In the worst case, O(N) extra space is required for the recursion stack.

Iterative Approach

Algorithm:

1. **Initially, we will take three-pointers, current that points to the head of Linked List, prev, and nextNode, both pointing to null.**
2. **Then we will iterate over the linked list until the current is not equal to NULL and do the following update in every step of the iteration:**
    1. **nextNode = current.next**
    2. **current.next = prev**
    3. **prev = current**
    4. **current = nextNode**
3. **Now return the prev pointer which is now the head of reverse Linked List.**

```
function reverseLL(head)
        // Creating node for remembering the previous node in the Linked List.
        prev = NULL
        // Creating temporary node.
        current = head
        while current is not null
                nextNode = current->next
                current->next = prev
                prev = current
                current = nextNode
```

```
        // Return reverse Linked List.
        return prev
```

Time Complexity: O(N), where N is the number of nodes in the linked list.
In the worst-case, we are iterating the whole linked list O(N). Hence, the overall complexity will be O(N).

Space Complexity: O(1), as we are using constant extra space.
Now we move to an application-based problem that uses the concept of reversal in linked lists.
Sorting in Linked List Notes

Sorting in Linked Lists

We have the problem of sorting a Linked List. We will discuss 2 standard algorithms to sort the linked List - Bubble sort and Merge Sort

Bubble Sort Algorithm

We swap the adjacent nodes of the linked list until the list becomes sorted. We do this in the standard bubble sort way repeating the swapping procedure n - 1 times, wherein each iteration we go through the list once and swap every pair of adjacent elements that do not follow the sorting order.

Algorithm:

1. **Repeat the step n - 1 times.**
2. **Make a pass through the list and keep swapping the adjacent elements if node.data > node.next.data.**

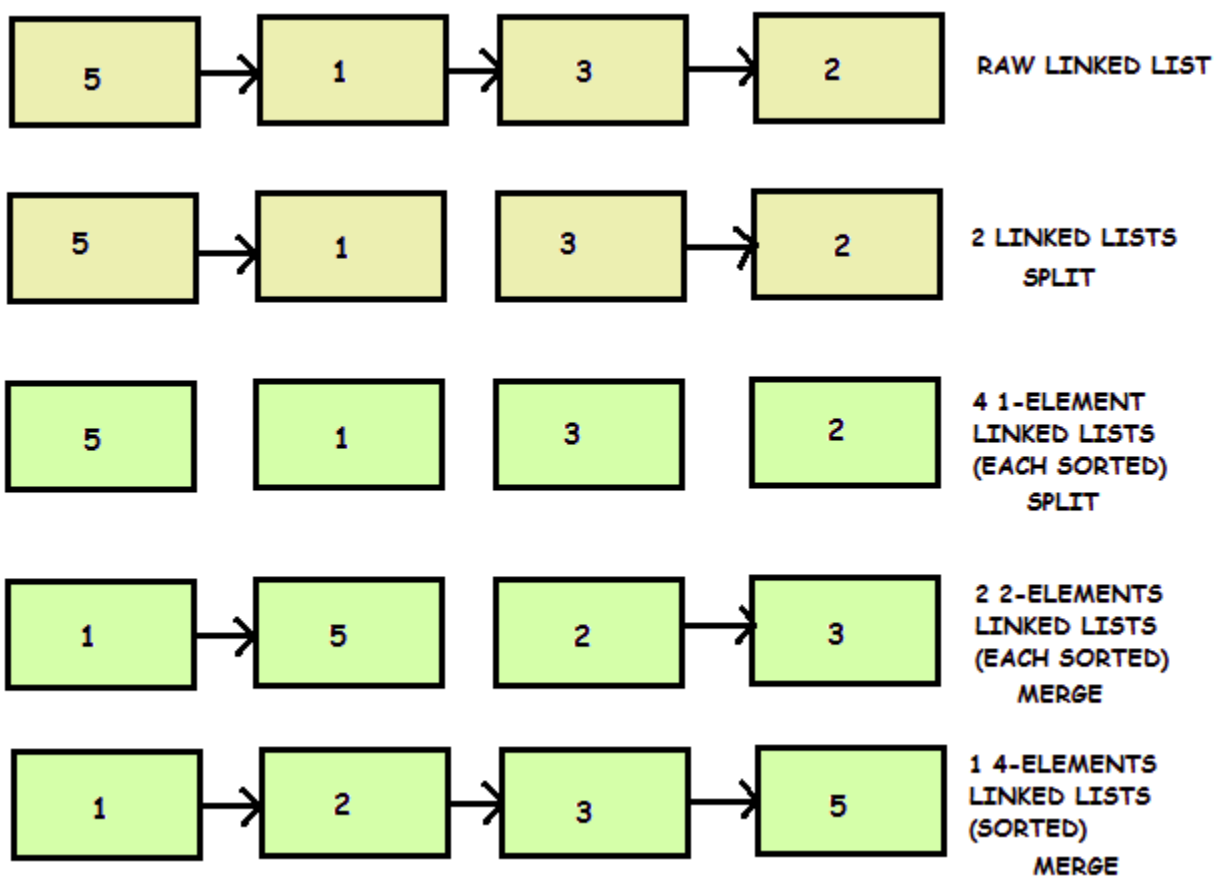
```
function bubbleSort(head)
        //n is the size of the linked list
        n = head.length
        for each i from 0 to n - 1
                h = head
                for each j from 0 to n  - i - 1
                        p1 = h
                        p2 = p1->next
                        if (p1->data > p2->data)
                                // update the link after swapping
                                swap(p1, p2)
                        h = h->next

        return head
```

Time Complexity: O(N*N), Where N is the number of nodes in the linked list. We make a pass over the linked list n - 1 times and in each of the iterations we make n comparisons.

Space Complexity: O(1). We use constant additional space for swapping the elements.

Merge Sort Algorithm

Merge Sort is a Divide and Conquer algorithm. It divides the input into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted subarrays into one.



Algorithm:

1. **If the list contains only one node, return the head of the list.**
2. **Else, divide the list into two sublists. For this, we will take pointers 'mid' and 'tail' which will initially point to the head node. We will change 'mid' and 'tail' as follows until 'tail' becomes NULL:**
   mid = mid->next
   tail = tail->next->next
3. **The above operation will ensure that mid will point to the middle node of the list. 'mid' will be the head of the second sublist, so we will change the 'next' value of the node which is before mid to NULL.**
4. **Call the same algorithm for the two sublists.**
5. **Merge the two sublists and return the head of the merged list. For this, we will take a pointer to a node, 'mergeList', which will be initially pointing to NULL. If the head of the first sublist has a value less than the head of the second sublist then we will point the 'mergeList' to the head of the first sublist and change the head to its next value. Else, we will point the 'mergeList' to the head of the second sub list. In the end, if any of the sublists becomes empty and the other sublist is non-empty, then we will add all nodes of the non-empty sublist in the 'mergeList'.**

The code looks as follows:

```
function merge(firstHead, secondHead)

        mergeList = NULL, cur = NULL
        // Placing the nodes in sorted order and simultaneously
        // merging the two lists
        while (firstHead is not null && secondHead is not null)
                if (firstHead->data < secondHead->data)
                        if (mergeList is null)
                                mergeList = firstHead
                                cur = firstHead
                        else
                                cur->next = firstHead
                                cur = cur->next
                                firstHead = firstHead->next
                else
                        if (mergeList is null)
                                mergeList = secondHead
                                cur = secondHead
                        else
                                cur->next = secondHead
                                cur = cur->next
                secondHead = secondHead->next

        // If any of the list is left, append it at the end of
        // currently merged list
        if (firstHead is not null)
                cur->next = firstHead
        if (secondHead is not null)
                cur->next = secondHead
```

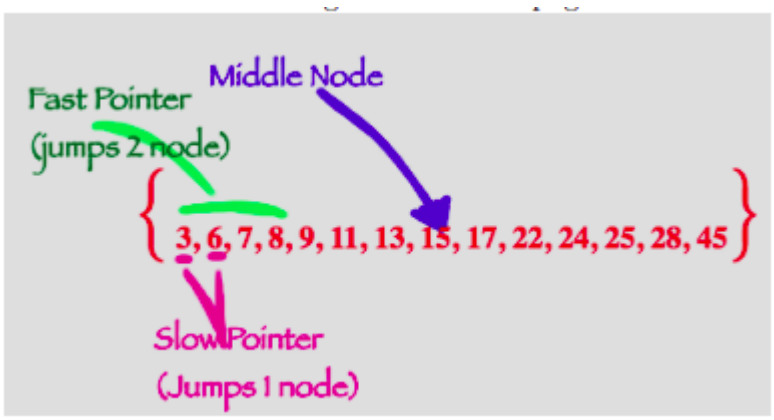
```
        return mergeList

function MergeSort(head)
        if (head is null or head->next is null)
                return head
        // dividing list into two sublists
        mid = head , tail = head
        while (tail is not null && tail->next is not null)
                prev = mid
                mid = mid->next
                tail = tail->next->next

        prev->next = NULL
        firstHead = head
        secondHead = mid
        // calling the recursive function on sub-lists
        firstHead = MergeSort(firstHead)
        secondHead = MergeSort(secondHead)
        head = merge(firstHead, secondHead)
        return head
```

Time Complexity: O(N*log2(N)), Where N is the number of nodes in the linked list.
The algorithm used (Merge Sort) is divide and conquer. So, for each traversal, we divide the list into 2 parts, thus there are log2(N) levels and the final complexity is O(N * log2(N)).

Space Complexity: O(log2(N)), where N is the number of nodes in the linked list
Since the algorithm is recursive and there are log2(N) levels, it requires O(log2(N)) stack space.
Slow and Fast Pointer Notes

Slow and Fast Pointer Technique

We maintain 2 pointers that move in the same direction, but one of the pointers makes larger strides as compared to the other. We make use of the fact that when the slower pointer points to the nth element, the faster pointer has made 2*n steps. This has applications in problems like finding the middle node of the linked list and detecting cycles.



Let’s try to see how this works by the use of an Example.

Problem Statement: Given a linked list, find the middle node.

Example: Linked-List : 1->2->3->4->5

Output: 3

Approach:
We place two pointers simultaneously at the head node, each one moves at different paces, the slow pointer moves one step, and the fast moves two steps instead. When the fast pointer reaches the end, the slow pointer will stop in the middle. For the loop, we only need to check on the faster pointer, make sure fast pointer and fast.next is not NULL so that we can successfully visit the fast.next.next. When the length is odd, the fast pointer will point at the end node, because fast.next is NULL, when it is even, the fast pointer will point at None node, it terminates because fast is None.

```
function findMiddle(head)
        // If head is null just return null
        if head is null
                return head

        // If the Linked List has just 1 element that element is the middle
        if head->next is null
                return head
```

```
        fast = head
        slow = head

        // moving the fast and slow pointer until the fast pointer reaches the end
        while fast is not null and fast->next is not null
                fast = fast->next->next
                slow = slow->next

        return slow
```

Time Complexity: O(n) Since the loop runs at most n/2 times to reach the middle node, the time complexity turns out to be O(n), where ‘n’ is the length of the linked list.

Space Complexity: O(1), Since we are using constant space to store the fast and slow pointers
Stacks Notes

Introduction to stacks

- **Stacks are simple data structures that allow us to store and retrieve data sequentially.**
- **A stack is a linear data structure like arrays and linked lists.**
- **It is an abstract data type (ADT).**
- **In a stack, the order in which the data arrives is essential. It follows the LIFO order of data insertion/abstraction. LIFO stands for Last In First Out.**
- **Consider the example of a pile of books:**

Here, unless the book at the topmost position is removed from the pile, we can’t have access to the second book from the top and similarly, the books below the second one. When we apply the same technique to the data in our program then, this pile-type structure is said to be a stack.

Like deletion/removal, we can only insert the book at the top of the pile rather than at any other position. This means that the object/data that made its entry at the last would be one to come out first, hence known as LIFO.

Various operations on stacks

In a stack, insertion and deletion are done at one end, called top.

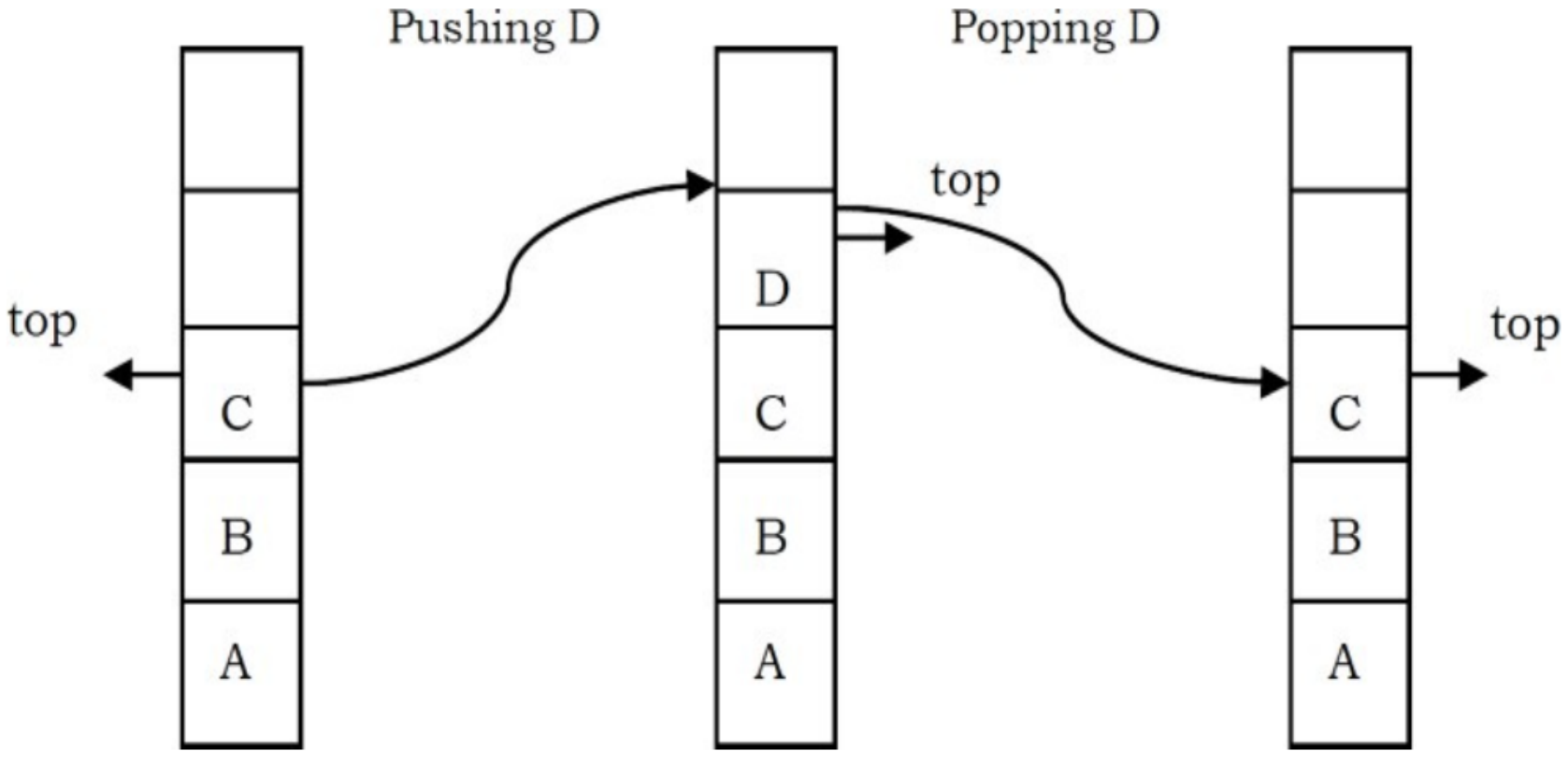
- **Insertion: This is known as a push operation.**
- **Deletion: This is known as a pop operation.**

Main stack operations

• push (data): Insert data onto the stack.
• pop(): Removes and returns the last inserted element from the stack.

Auxiliary stack operations

• top(): Returns the last inserted element without removing it.
• int size(): Returns the number of elements stored in the stack.



• **boolean isEmpty(): Indicates whether any elements are stored in the stack or not i.e. whether the stack is empty or not.**

Implementation of Stacks

Stacks can be implemented using arrays, linked lists, or queues. The underlying algorithm for implementing operations of the stack remains the same.

- **Push operation**

```
function push(data, stack)
        //   data : the data to be inserted into the stack.
        if stack is full
                return null
        /*
         top : It refers to the position (index in arrays) of the last
         element into the stack
        */
        top = top + 1
        stack[top] = data
```

- **Pop operation**

```
function pop(stack)
        if stack is empty
                return null

        //   Retrieving data of the element to be popped.
        data = stack[top]
        //   Decrementing top
        top = top - 1
        return data
```

- **Top operation**

```
function top(stack)
        if stack is empty
                return null
        else
                return stack[top]
```

- **isEmpty operation**

```
function isEmpty(stack)
        if top is null
                return true
        else
                return false
```

Time Complexity of various operations

Let ‘n’ be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

Operations	Time Complexity
Push(data)	O(1)
Pop()	O(1)
int top()	O(1)
boolean isEmpty()	O(1)
int size()	O(1)
boolean isFull()	O(1)

Exceptions

- **Attempting the execution of an operation may sometimes cause an error condition, called an exception.**
- **Exceptions are said to be "thrown" by an operation that cannot be executed.**
- **Attempting the execution of pop() on an empty stack throws an exception called Stack Underflow.**
- **Trying to push an element in a full-stack throws an exception called Stack Overflow.**

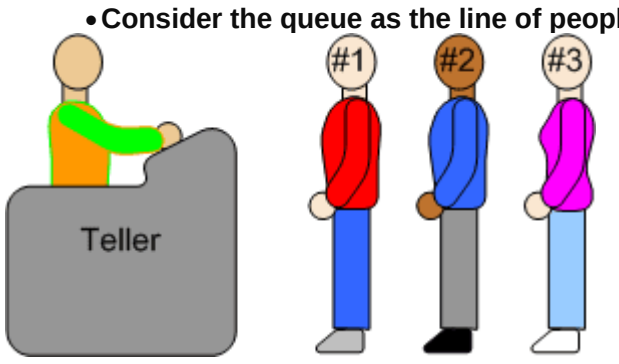
Applications of stacks

- **Stacks are useful when we need dynamic addition and deletion of elements in our data structure. Since stacks require O(1) time complexity for all the operations, we can achieve our tasks very efficiently.**
- **It also has applications in various popular algorithmic problems like:-**
  - **Tower of Hanoi**
  - **Balancing Parenthesis**
  - **Infix to postfix**
  - **Backtracking problems**
- **It is useful in many Graph Algorithms like topological sorting and finding strongly connected components.**
- **Apart from all these it also has very practical applications like:**
  - **Undo and Redo operations in editors**
  - **Forward and backward buttons in browsers**
  - **Allocation of memory by an operating system while executing a process.**

Queues Notes

Introduction to queues

- **Queues are simple data structures in which insertions are done at one end and deletions are done at the other end.**
- **It is a linear data structure as arrays.**
- **It is an abstract data type (ADT).**
- **The first element to be inserted is the first element to be deleted. It follows either FIFO (First in First Out) or LILO (Last In Last Out).**
- **Consider the queue as the line of people.**

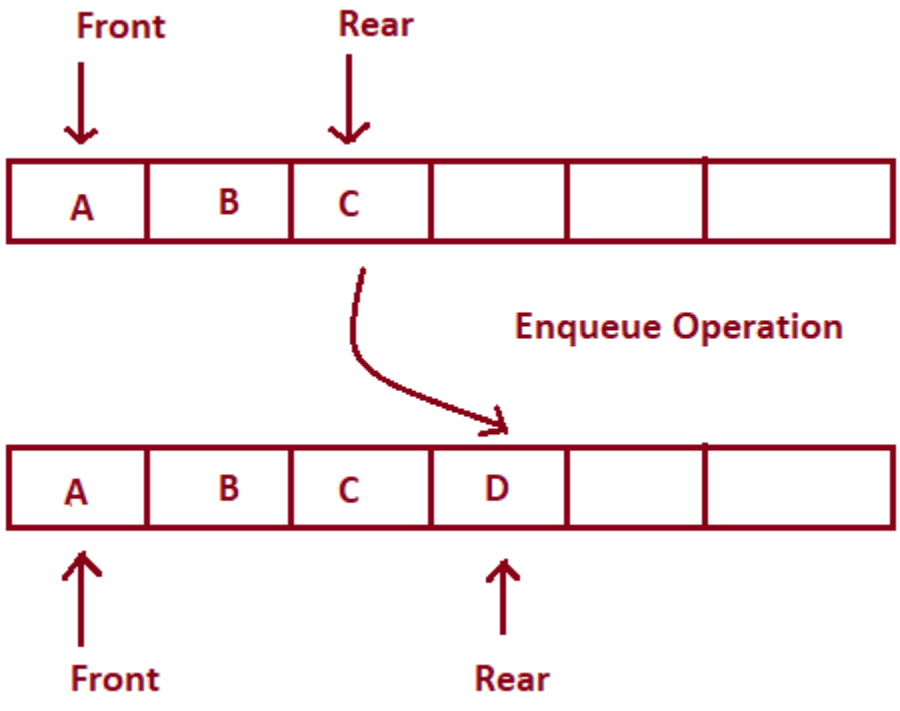


The above picture shows that when we enter the queue/line, we stand at the end of the line and the person who is at the front of the line is the one who will be served first.

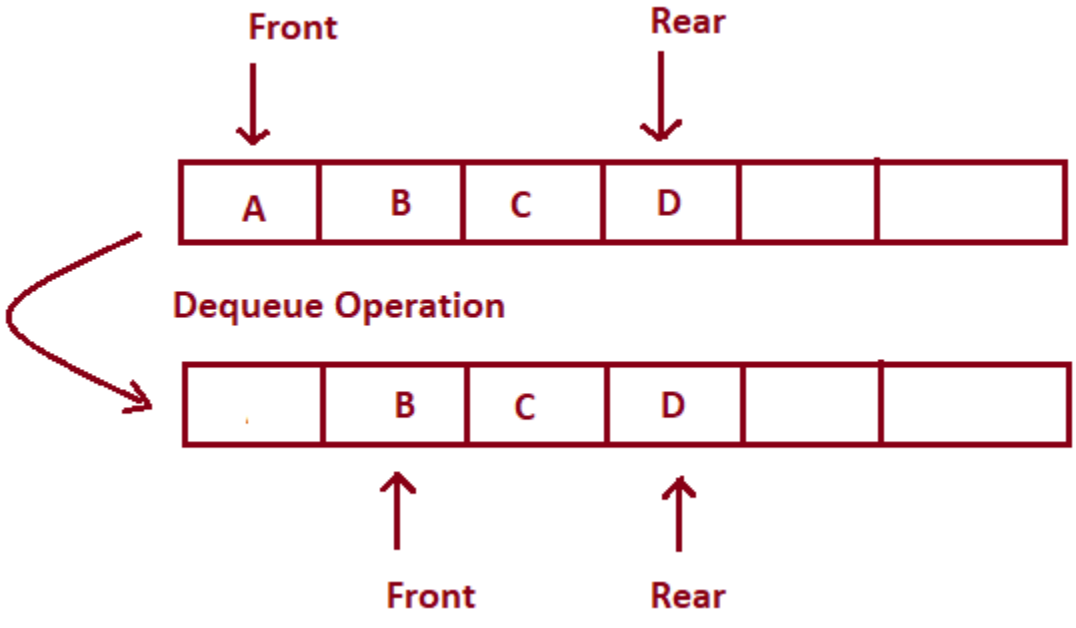
After the first person is served, he will exit, and as a result, the next person will come at the head of the line. Similarly, the head of the queue will keep exiting the line as we move towards the front/head of the line. Finally, we reach the front. we are served and we exit. This behavior is useful when we need to maintain the order of arrival.

Various operations on queues

In queues, insertion is done at one end (rear) and deletion is done at other end (front) :



Enqueue Operation



Dequeue Operation

- **Insertion: Adding elements at the rear side. The concept of inserting an element into the queue is called Enqueue. Enqueueing an element, when the queue is full, is called overflow.**
- **Deletion: Deleting elements at the front side. The concept of deleting an element from the queue is called Dequeue. Deleting an element from the queue when the queue is empty is called underflow.**

Main Queue Operations:

- **enqueue(data) : Insert data in the queue (at rear).**
- **dequeue() : Deletes and returns the first element of the queue (at front).**

Auxiliary Queue Operations:

- **front() : returns the first element of the queue.**
- **int size() : returns number of elements in the queue.**
- **boolean isEmpty() : returns whether the queue is empty or not.**

How can queues be implemented?

Queues can be implemented using arrays, linked lists or stacks. The basic algorithm for implementing a queue remains the same. We will use array-based implementation here. We maintain two variables for all the operations: front, rear.

- **Enqueue Operation**

```
function enqueue(data)
        if queue is full
        return "Full Queue Exception"

    queue[rear] = data
    rear++
```

- **Dequeue Operation**

```
function  dequeue()
        if queue is empty
        return "Empty Queue Exception"

    temp = queue[front]
    front++
    return temp
```

- **getFront() Operation**

```
function getFront()
        if queue is empty
        return "Empty Queue Exception"
    temp = queue[front]
    return temp
```

Time Complexity of various operations

Let 'n' be the number of elements in the queue. The complexities of queue operations with this representation can be given as:

Operations	Time Complexity
enqueue(data)	O(1)
dequeue	O(1)
int getFront()	O(1)
boolean isEmpty()	O(1)
int size()	O(1)

Applications of queues

Real-Life Applications

- **Scheduling of jobs in the order of arrival by the operating system**
- **Multiprogramming**
- **Waiting time of customers at call centers**
- **Asynchronous data transfer**

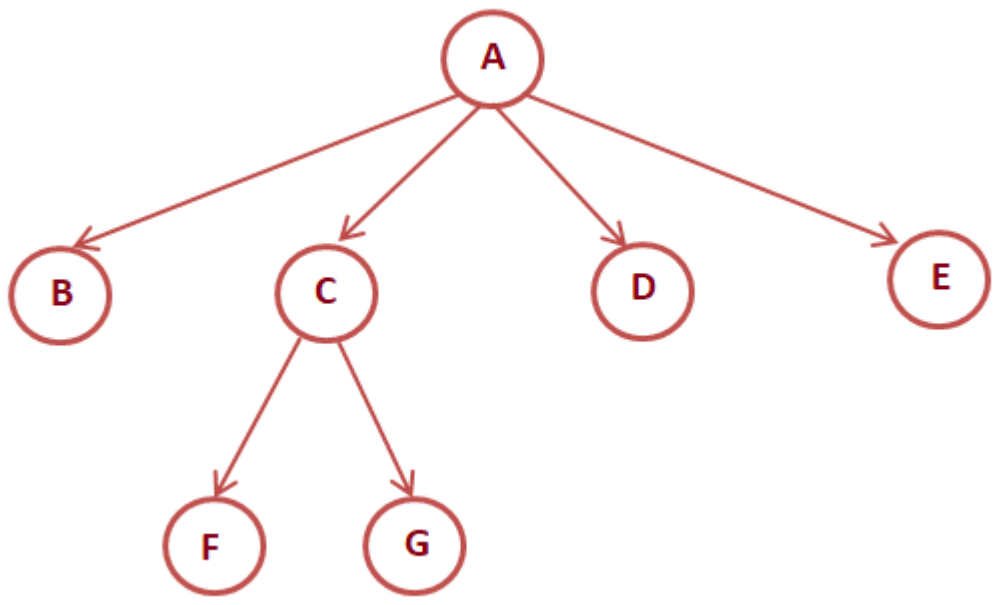
Application in solving DSA problems

- **Queues are useful when we need dynamic addition and deletion of elements in our data structure. Since queues require O(1) time complexity for all the operations, we can achieve our tasks very efficiently.**
- **It is useful in many Graph Algorithms like breadth-first search, Dijkstra's algorithm, and prim's algorithm.**

Introduction to Trees Notes

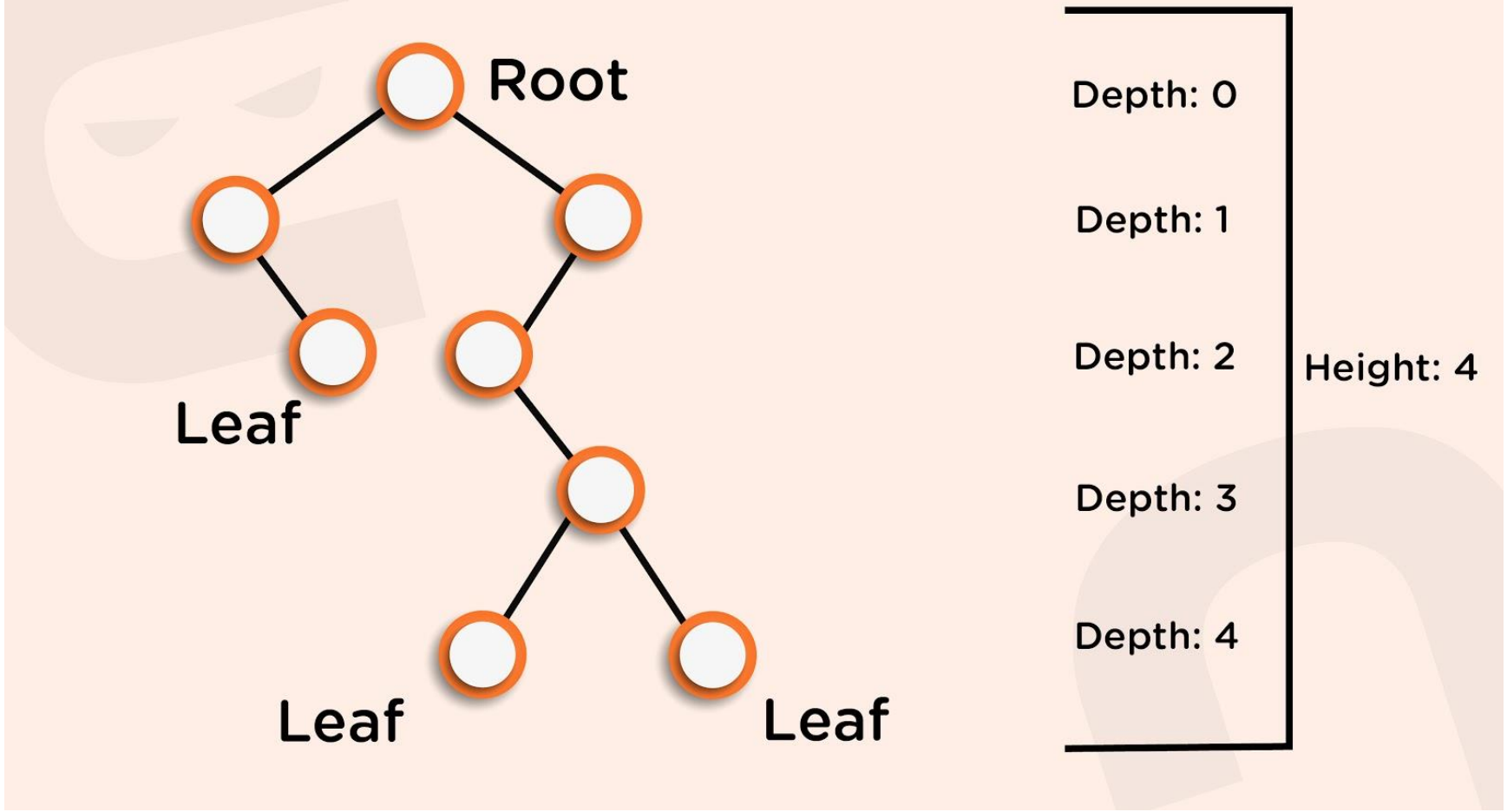
Introduction to Trees

A tree is a data structure similar to linked lists but instead of each node pointing to just one next node in a linear fashion, each node points to a number of nodes. The tree is a non-linear data structure. A tree structure is a way of representing a hierarchical nature of a structure in a graphical form.



Properties of Trees

- **Root: The root of the tree is the node with no parents. There can be at most one root node in the tree (Eg: A is the root node in the above example).**
- **Parent: For a given node, its immediate predecessor is known as its parent node. In other words, nodes having 1 or more children are parent nodes. (Eg: A is the parent node of B, C, D, E, and C is the parent node of F, G)**
- **Edge: An edge refers to the link from the parent node to the child node.**
- **Leaf: Nodes with no children are called leaf nodes (Eg: B, F, G, D, E are leaf nodes in the above example).**
- **Siblings: Children with the same parent are called siblings. (Eg: B, C, D, E are siblings, and F, G are siblings).**
- **A node x is an ancestor of node y if there exists a path from the root to node y such that x appears on the path. Node y is called the descendant of node x. (Eg: A is an ancestor of node F, G)**
- **Depth: The depth of a node in the tree is the length of the path from the root to the node. The depth of the tree is the maximum depth among all the nodes in the tree.**

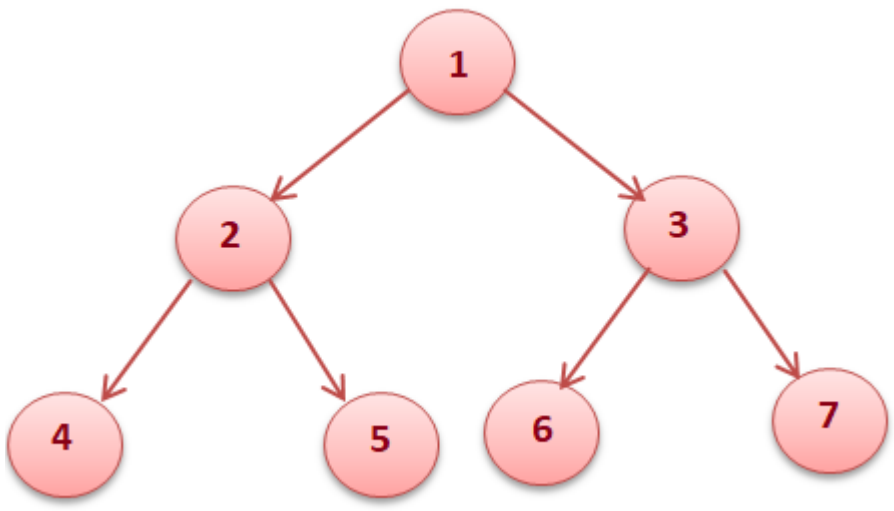


- **Height: The height of the node is the length of the path from that node to the deepest node in the tree. The height of the tree is the length of the path from the root node to the deepest node in the tree. (Eg: the height of the tree in the above example is four (count the edges, not the nodes)).**
- **A tree with only one node has zero height.**
- **For a given tree, depth and height returns the same value but may be different for individual nodes.**
- **Skew Trees: If every node in a tree has only one child then we call such a tree a skew tree. If every node has only a left child, we call them left skew trees. If every node has only the right child, we call them right skew trees.**

Tree Traversal Notes

Binary Tree Traversals

The process of visiting all the nodes of the tree is called tree traversal. Each node is processed only once but may be visited more than once. We will be considering the below tree for all the traversals.
D: Current node
L: Left Subtree
R: Right Subtree



- **PreOrder Traversal (DLR)**

In preorder traversal, each node is processed before processing its subtrees.
Like in the above example, 1 is processed first, then the left subtree, and this is followed by the right subtree. Therefore processing must return to the right subtree after finishing the processing of the left subtree. To move to the right subtree after processing the left subtree we have to maintain the root information.

Steps for preOrder traversal

- **Visit the root.**
- **Traverse the left subtree in Preorder.**
- **Traverse the right subtree in Preorder.**

```
function preOrderTraversal(root)

        if root is null
                return
```

```
   //   process current node
        print "root.data"
        //   calling function recursively for left and right subtrees
        preOrderTraversal(root.left)
        preOrderTraversal(root.right)
```

PreOrder Output of above tree : 1 2 4 5 3 6 7

- **InOrder Traversal (LDR)**

In inorder traversal, the root is visited between the subtrees.

Steps for InOrderTraversal

- **Traverse the left subtree in Inorder.**
- **Visit the root.**
- **Traverse the right subtree in Inorder.**

```
function inOrderTraversal(root)
        if root is null
             return

        //   calling function recursively for left subtree
        inOrderTraversal(root.left)
        //   process current node
        print "root.data"
        //   calling function recursively for right subtree
        inOrderTraversal(root.right)
```

InOrder Output of above tree : 4 2 5 1 6 3 7

- **PostOrder Traversal(LDR)**

In postorder traversal, the root is visited after the left and right subtrees respectively.

Steps for postOrder traversal

- **Traverse the left subtree in postOrder.**
- **Traverse the right subtree in postOrder.**
- **Visit the root.**

```
function postOrderTraversal(root)
        if root is null
             return

        //   calling function recursively for left and right subtrees
        postOrderTraversal(root.left)
        postOrderTraversal(root.right)
        //   process current node
        print "root.data"
```

postOrder output of above tree : 4 5 2 6 7 3 1

- **LevelOrder Traversal**

In level order traversal, each node is visited level-wise in increasing order of levels from left to right.

Steps for levelOrder traversal

- **Visit the root.**
- **While traversing the level l, add all the elements at (l + 1) in the queue**
- **Go to the next level and visit all the nodes at that level.**
- **Repeat this until all levels are completed.**

```
function levelOrderTraversal()
        if root is null
             return

        //   Create queue of type Node and add root to the queue
        queue.add(root).

        while the queue is not empty
             //   remove the front item from the queue
             removedNode= queue.remove()
             //   process the removedNode
            print "removeNode.data"
```

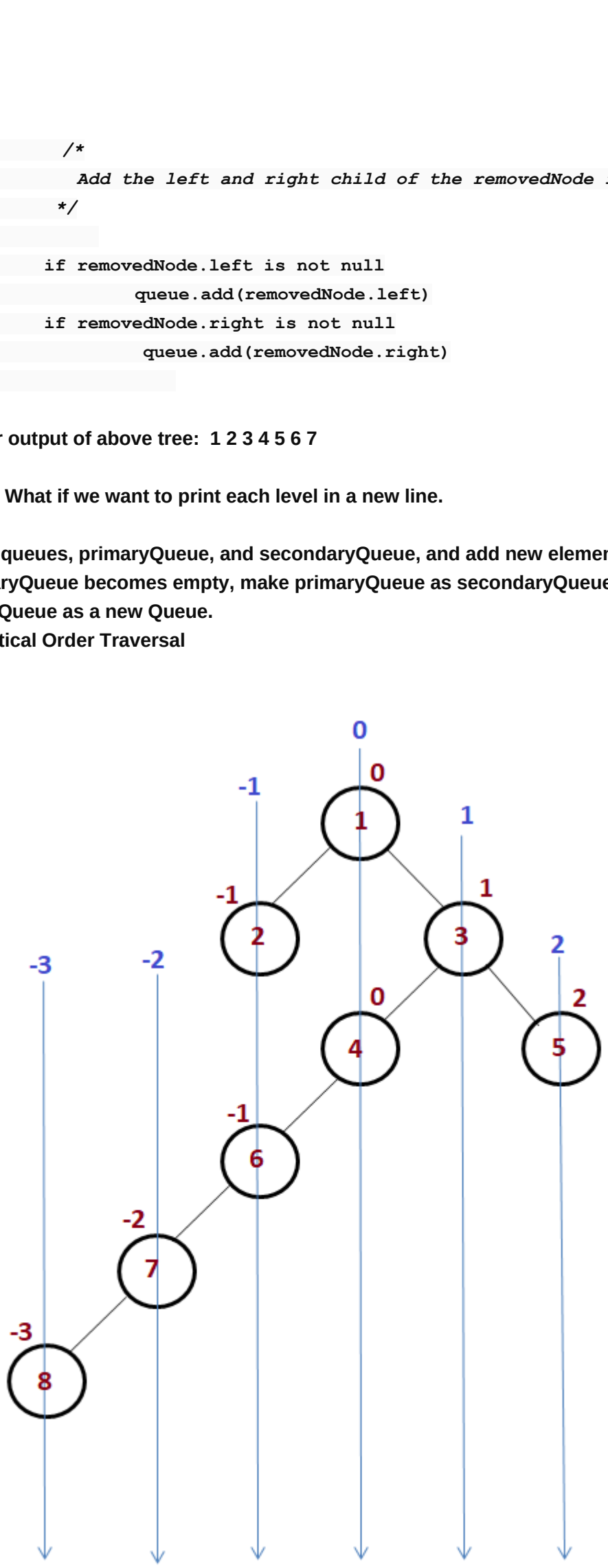
```
                /*
                  Add the left and right child of the removedNode if they are not null
                */

              if removedNode.left is not null
                      queue.add(removedNode.left)
              if removedNode.right is not null
                       queue.add(removedNode.right)
    end
```

LevelOrder output of above tree: 1 2 3 4 5 6 7

Follow Up: What if we want to print each level in a new line.

Maintain 2 queues, primaryQueue, and secondaryQueue, and add new elements for the next level in the secondaryQueue. When your primaryQueue becomes empty, make primaryQueue as secondaryQueue, print a new line, and reinitialize the secondaryQueue as a new Queue.

- **Vertical Order Traversal**



Output: 8 7 2 6 1 4 3 2

Note: If two nodes are at the same horizontal distance like 2 and 6 then 2 must be printed before 6 i.e mentioning the order

Steps for vertical Order traversal

- **Create a TreeMap of the vertical level Vs list integers(in that particular vertical level)**
- **Maintain another variable with each node know as the vertical level of that node**
- **Visit the root. Add it to the queue.**
- **Remove the node from the queue say removedNode and add the node's value in treemap across the removedNode.verticallevel.**
- **While traversing level l, add left and right child of the removed node to queue and treemap setting the leftNode's and rightNode's vertical level equal to removedNode.verticalLevel-1 and removedNode.verticalLevel+1 respectively.**
- **Go to the next level and visit all the nodes at that level.**

- **Repeat this until all levels are completed.**
- **After this traversal, iterate over the treemap, get the list across the current key of the treemap and print the list.**

```
function verticalOrderTraversal()
        if root is null
              return

        //   Create queue of type Node and add root to the queue
        queue.add(root).

        while the queue is not empty
            //   remove the front item from the queue
              removedNode= queue.remove()
              //   process the removedNode, add to the list in treemap across that level
              levelItems = map.get(root.verticalLevel)
              levelItems.add(removedNode.data)

              /*
               Add the left and right child of the removedNode if they are not null
              */

              if removedNode.left is not null
                    removedNode.left.verticalLevel = removedNode.verticalLevel-1
                    queue.add(removedNode.left)
             if removedNode.right is not null
                    removedNode.right.verticalLevel = removedNode.verticalLevel+1
                    queue.add(removedNode.right)
   end

   //   Iterate over the treemap
   for(every Key in map)
             print(map.get(key))
```


Construction Of Trees

If we are given two traversal sequences, can we construct the binary tree uniquely?

It depends on what traversals are given. If one of the traversal methods is inorder then the tree can be constructed uniquely, otherwise not.

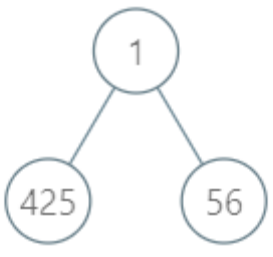
Construction of a tree from Preorder traversal and inorder traversal

Let’s understand it by an example

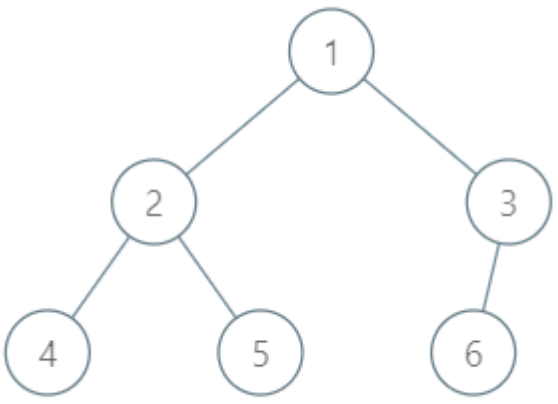
Preorder traversal: 1 2 4 5 3 6
Inorder Traversal: 4 2 5 1 6 3

We know that in preOrder traversal root node comes first, therefore the root of the tree will be 1. Later on, searching 1 in Inorder traversal we will get to know the nodes in the left and the right subtree of the root node. So to optimize searching we must create a HashMap to keep track of numbers Vs indexes.

After the first step, the tree will look like



Repeating the same steps we will get the tree as shown below



Algorithm

constructTree()

- **Initialize preIndex = 0 to iterate over the preorder traversal.**
- **Pick an element from preOrder traversal and increment the preIndex to pick the element in the next recursive call.**
- **Create a newNode and set newNode's data as the picked element.**
- **Find the picked element in inorder traversal and let the index be idx.**
- **Call constructTree for elements before idx and make the constructTree as a left subtree of newNode.**
- **Call constructTree for elements after idx and make the constructTree as the right subtree of newNode.**
- **return newNode.**

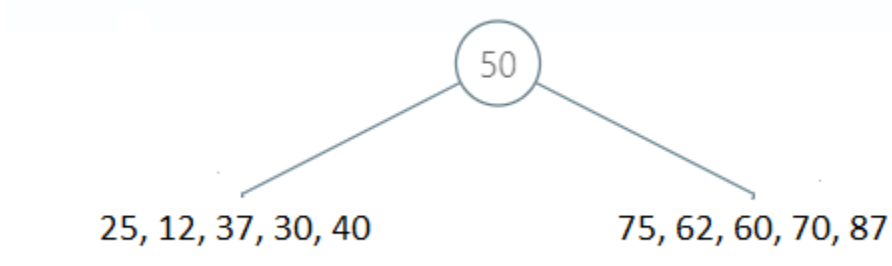
Construction of a tree from PostOrder traversal and Inorder traversal

Given traversals:

Postorder: 12, 30, 40, 37, 25, 60, 70, 62, 87, 75, 50
Inorder: 12, 25, 30, 37, 40, 50, 60, 62, 70, 75, 87

- **The last element in the postorder traversal is the root of the tree.**
- **So, here 50 will be the root of the tree.**
- **We will find the index of 50 in the inorder traversal. The index found is 5. Let this index be denoted by 'pos'.**
- **All the elements to the left of this index (from 0 to 4 index) will be in the left subtree of 50.**
- **And all the elements to the right of this index (from 6 to 10) will be in the right subtree of 50.**

Now the structure of the tree is:



Now, we will divide the postorder and inorder array into two parts. One is for the left subtree and the other is for the right subtree.

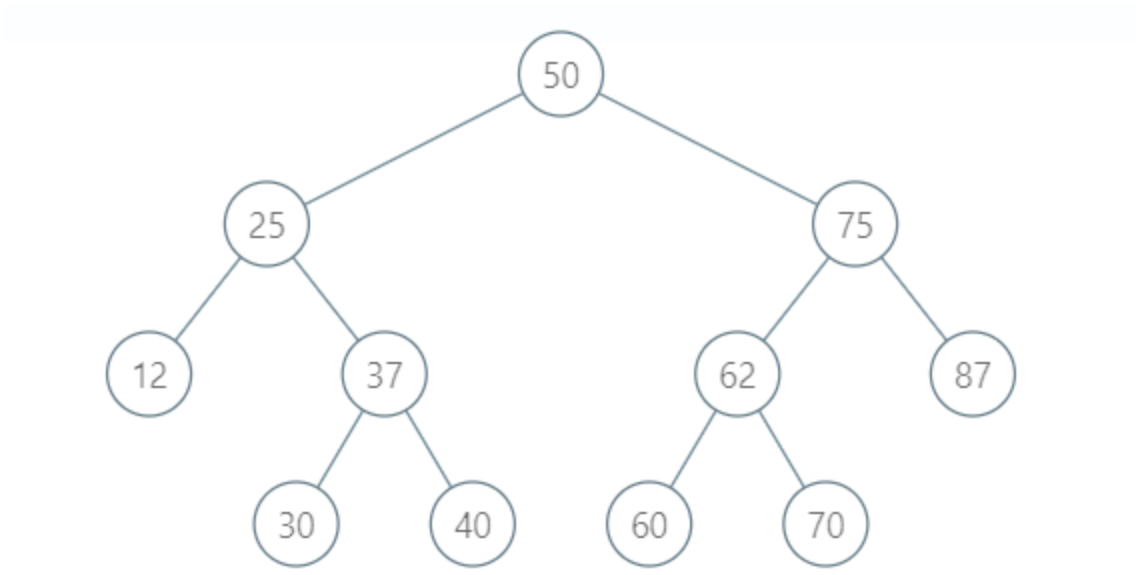
Let psi: starting index for the preorder array
pei: ending index for the preorder array
isi: starting index of the inorder array
iei: ending index of the preorder array
clc: Number of elements in the left subtree

Clearly, clc = pos - isi;

For left subtree:
Postorder array: from index psi, psi + clc - 1
Inorder array: from index isi, isi + clc -1

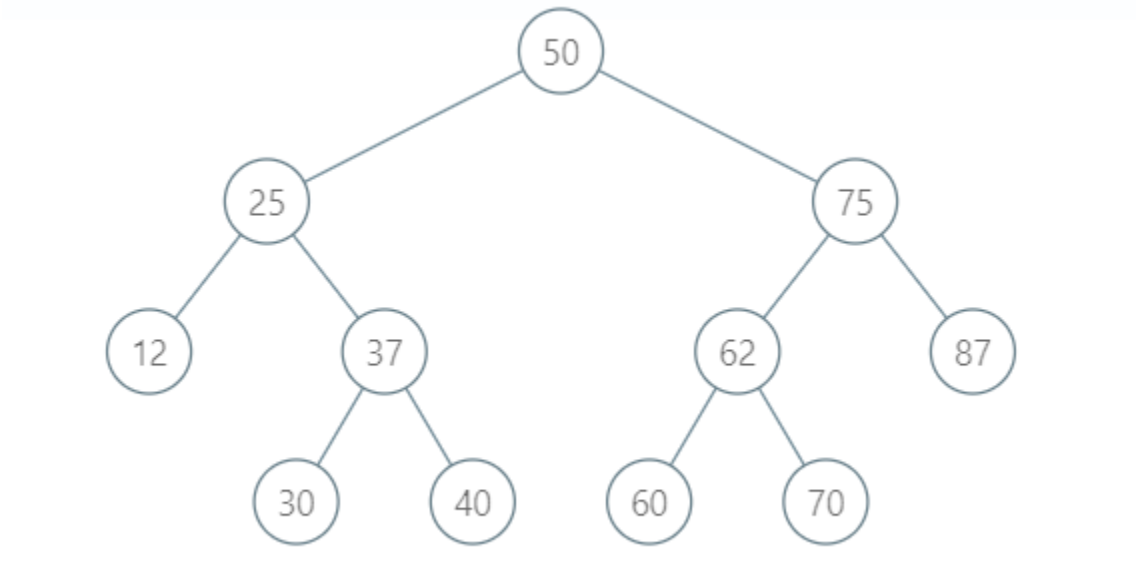
For right subtree:
Postorder array: from index psi+clc, pei - 1
Inorder array: from index isi + clc + 1, iei

Using the above arrays, all the steps are recursively repeated. The following binary tree is obtained:



Tree Views Notes

Tree Views



Vertical Level Order Traversal: -2: 12
-1: 25 30 60
0: 50 37 62
1: 75 40 70
2: 87

Horizontal Level Order Traversal: 1: 50
2: 25 75
3: 12 37 62 87
4: 30 40 60 70

Top View: 12 25 50 75 87 (first element of each vertical level)

Bottom View: 12 60 62 70 87 (last elements of each vertical level)

Left View: 50 25 12 30 (first element of each horizontal level)

Right View: 50 75 87 70 (last element of each horizontal level)

- **Top View**

In the above traversals, we did the vertical Order traversal. Now to print the top view we need to get the same treemap as in vertical order traversal. Now across every vertical level print only the first element.

- **Bottom View**

Similarly, for the bottom view just print the last element of the list across each level.

- **Left View**

The left view of the tree could be printed by printing just the first element of each level in the tree. So print the first element when moving to nextlevel i.e. when the primaryQueue becomes empty.

- **Right View**

The right view of the tree could be printed by printing just the last element of each level in the tree. So print the last element when moving to nextlevel i.e. when the primaryQueue becomes empty.

Introduction to BST

Introduction

- **These are specific types of binary trees.**

- **These are inspired by the binary search algorithm that elements on the left side of a particular element are smaller and that on the right side are greater.**
- **Time complexity on insertion, deletion, and searching reduces significantly as it works on the principle of binary search rather than linear search, as in the case of normal binary trees (will discuss it further).**

How are BSTs stored?

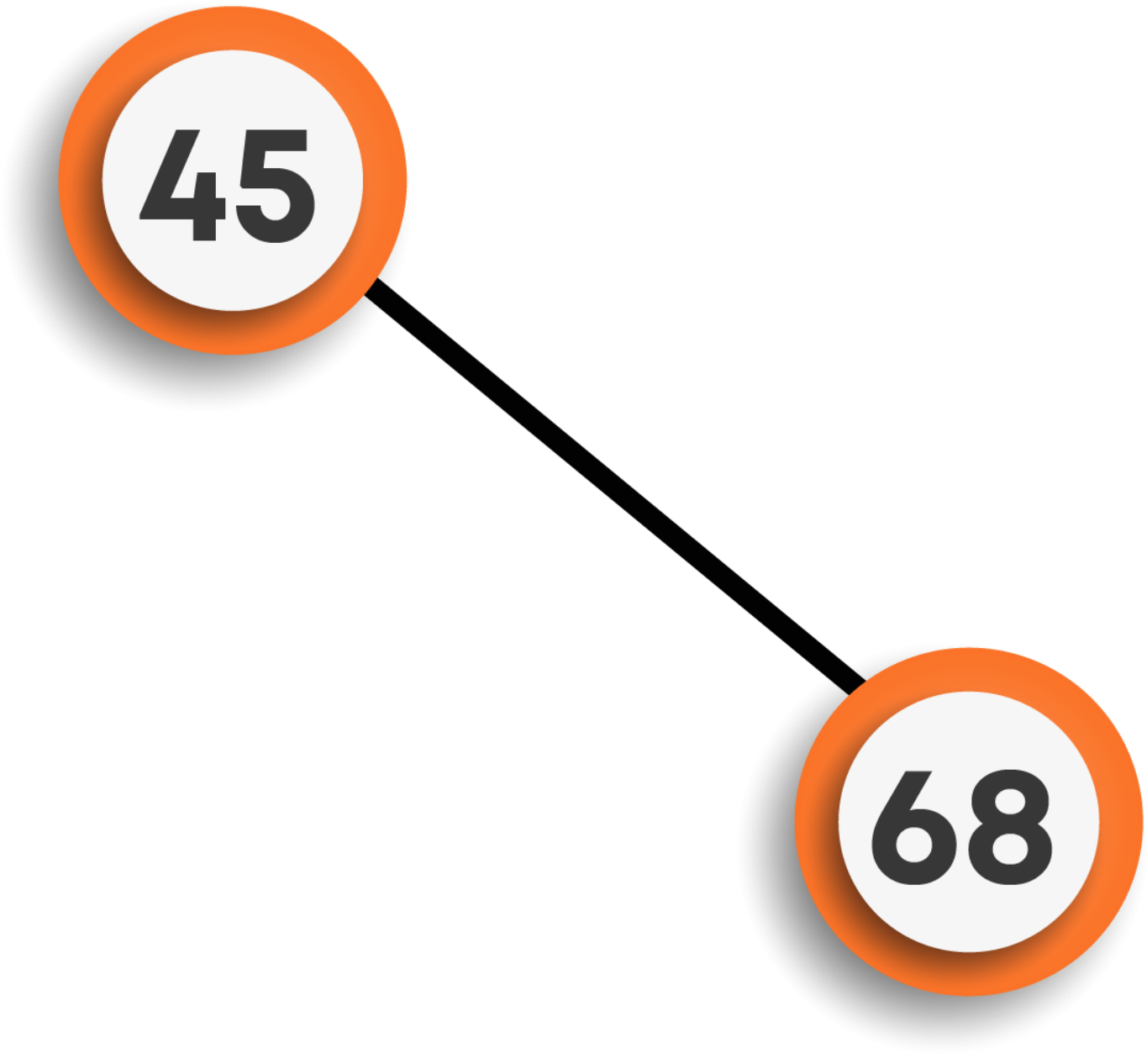
In BSTs, we always insert the node with smaller data towards the left side of the compared node and the larger data node as its right child. To be more concise, consider the root node of BST to be N, then:

- **Everything lesser than N will be placed in the left subtree.**
- **Everything greater than N will be placed in the right subtree.**

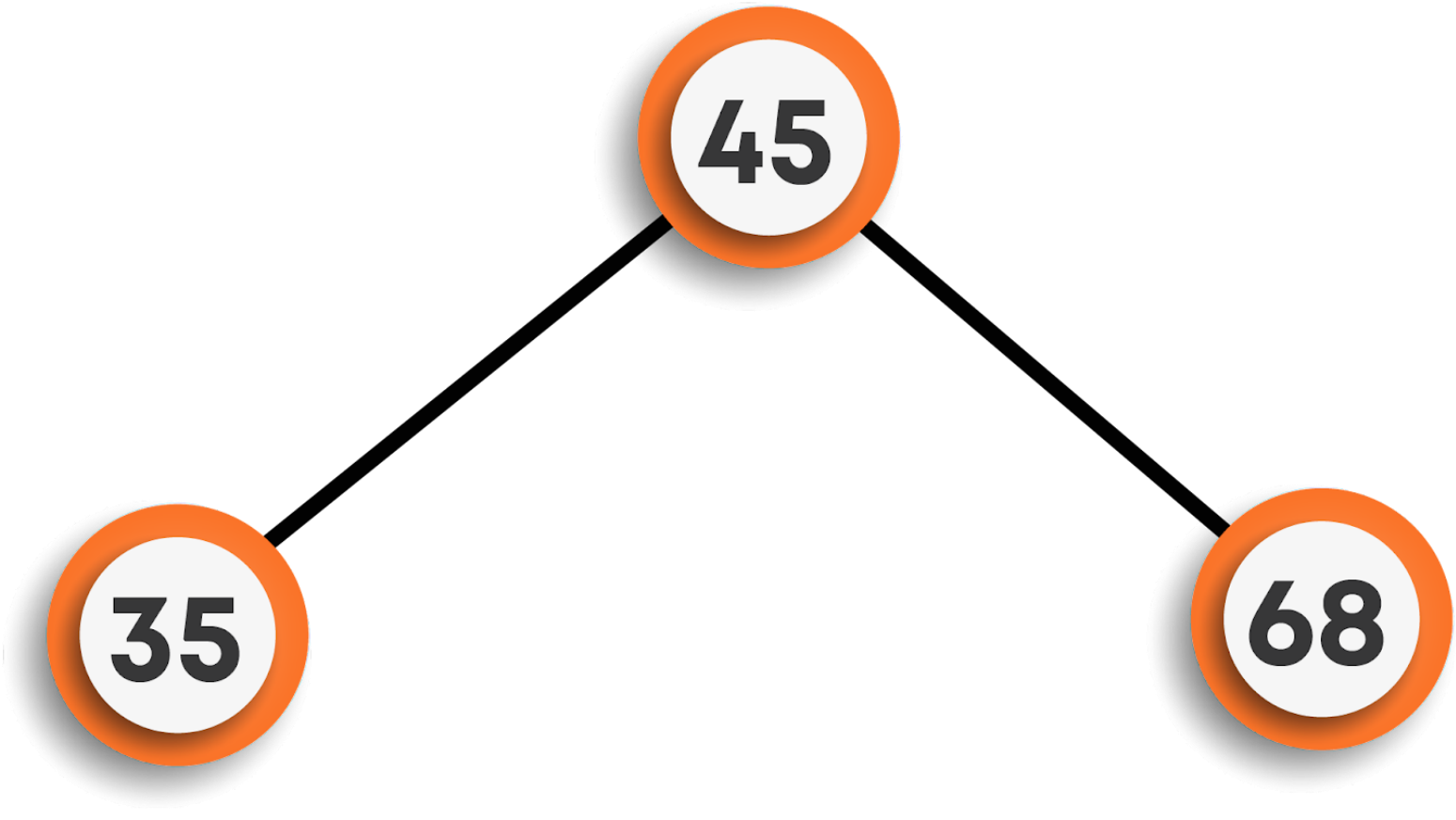
For Example: Insert {45, 68, 35, 42, 15, 64, 78} in a BST in the order they are given.
Solution: Follow the steps below:

1. **Since the tree is empty, so the first node will automatically be the root node.**

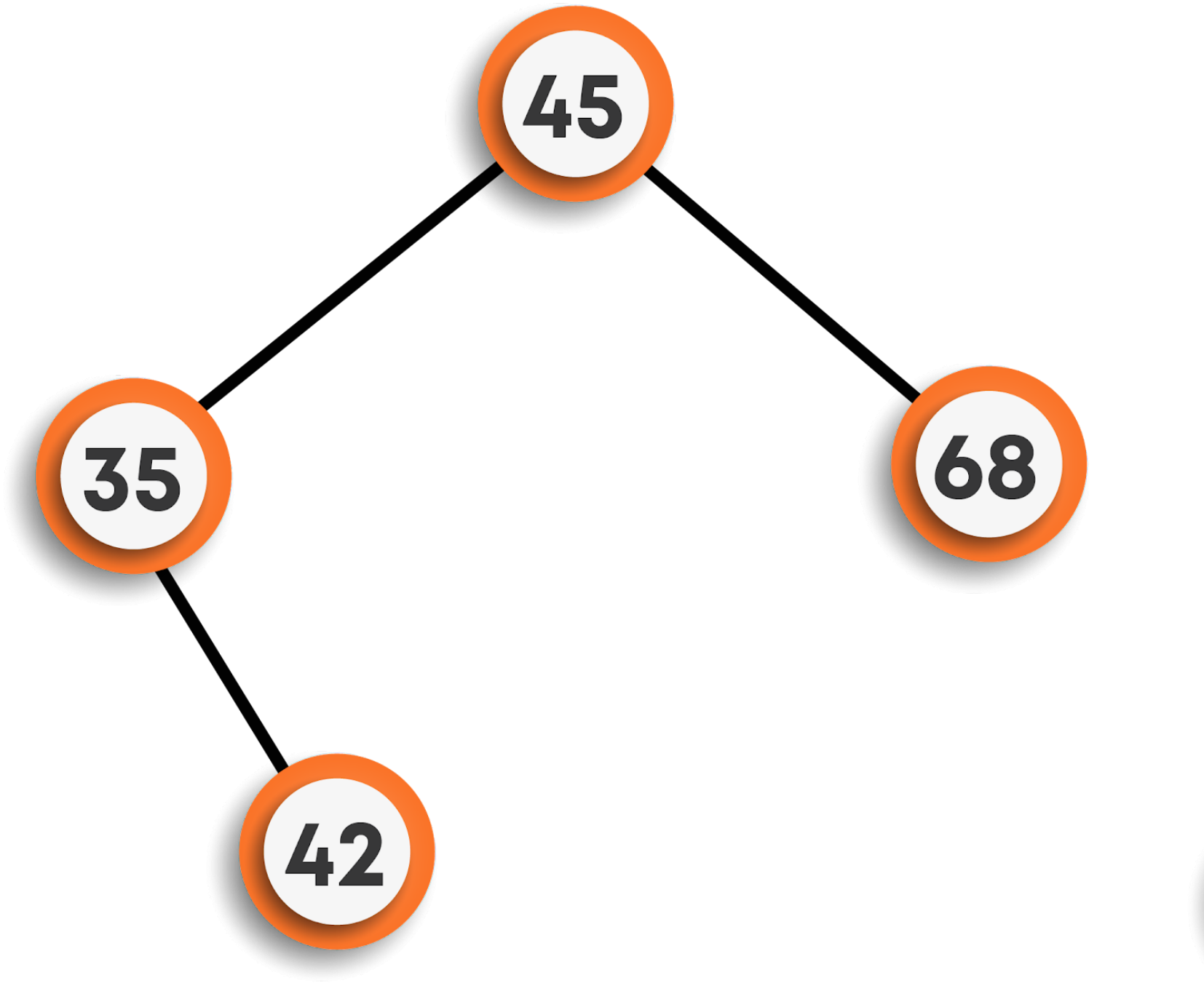
1. **Now, we have to insert 68, which is greater than 45, so it will go on the right side of the root node.**



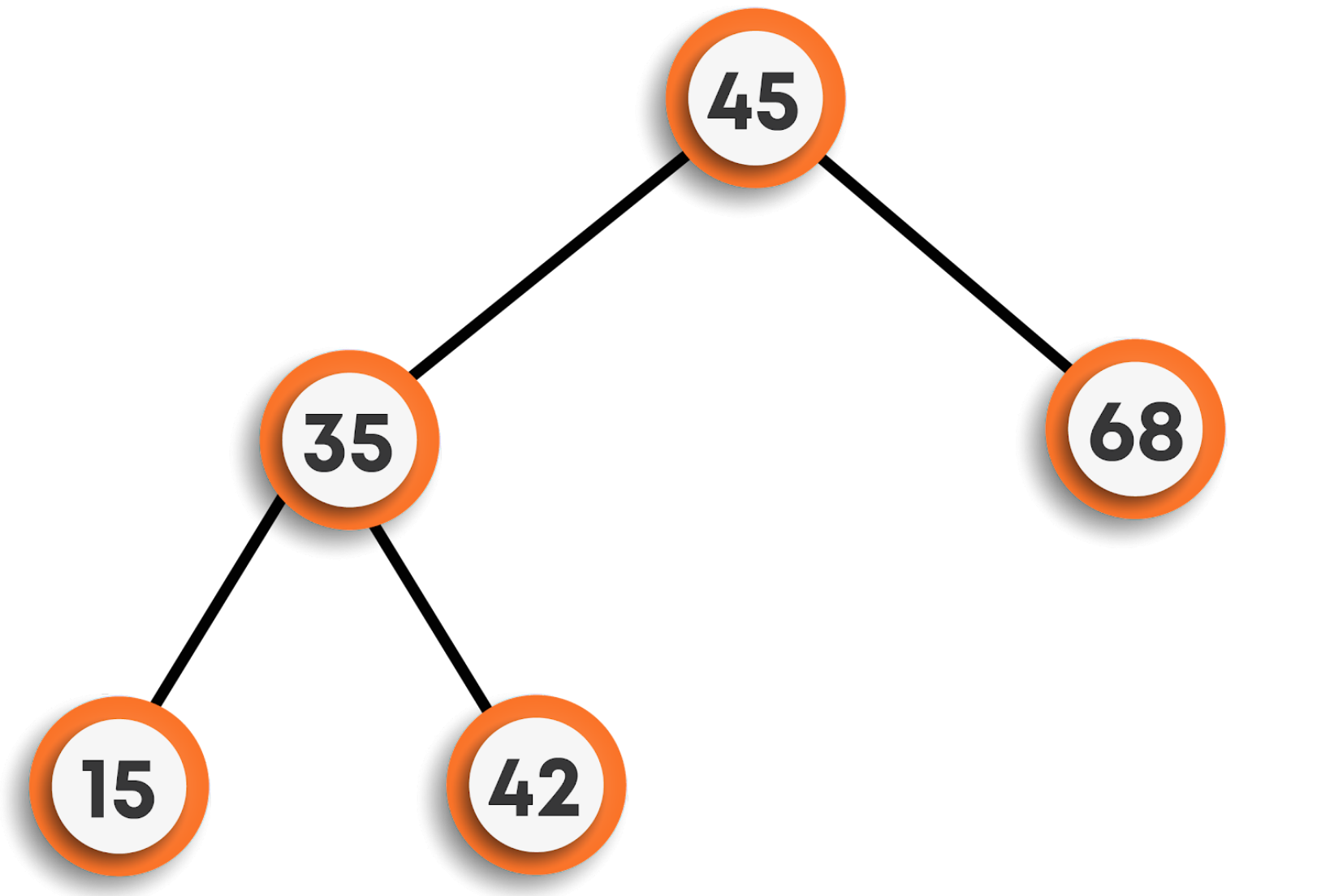
1. **To insert 35, we will start from the root node. Since 35 is smaller than 45, it will be inserted on the left side.**



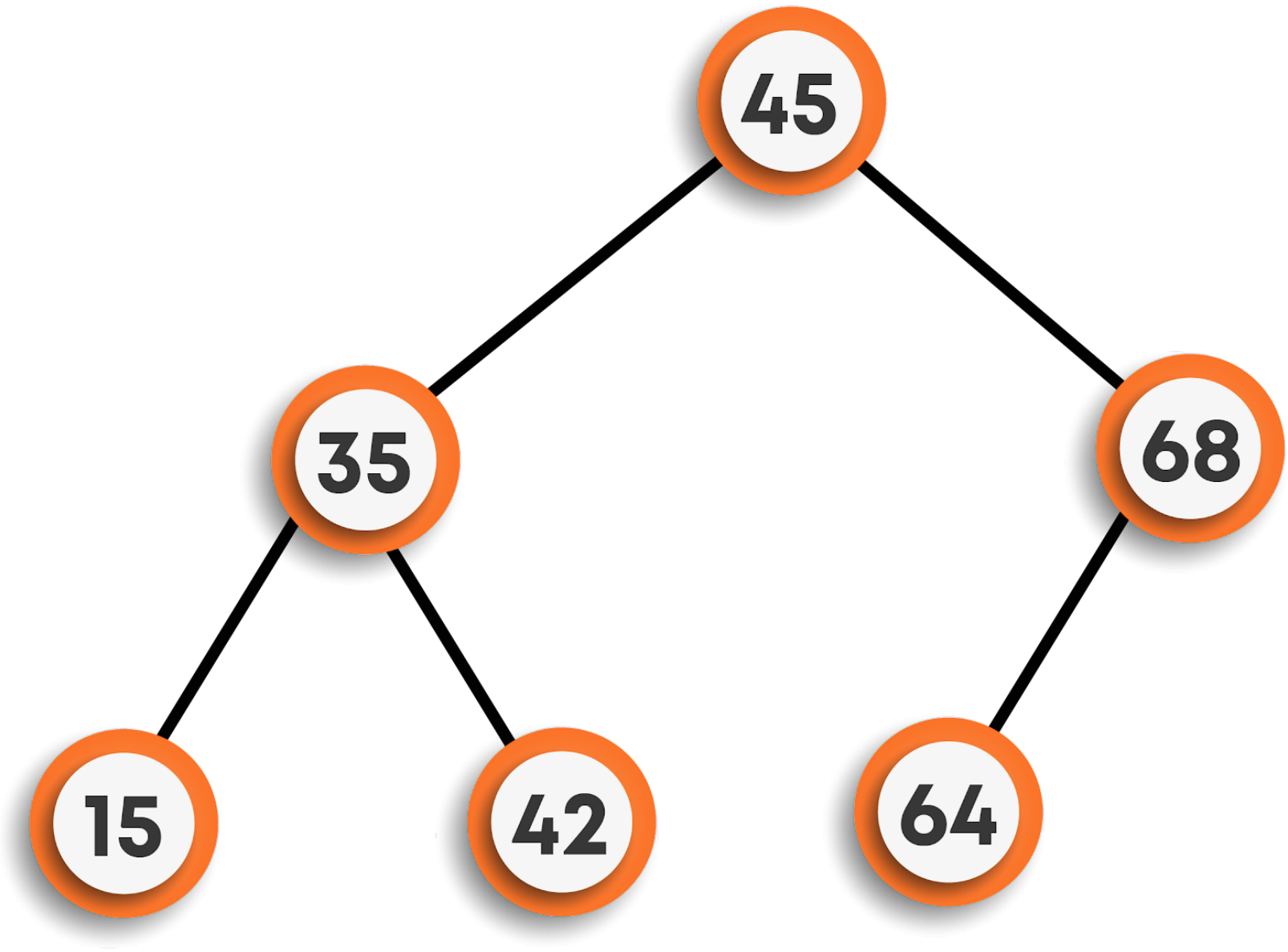
1. **Moving on to inserting 42, we can see that 42 is less than 45 so it will be placed on the left side of the root node. Now, we will check the same on the left subtree. We can see that 42 > 35, this means 42 will be the right child of 35, which is still a part of the left subtree of the root node.**



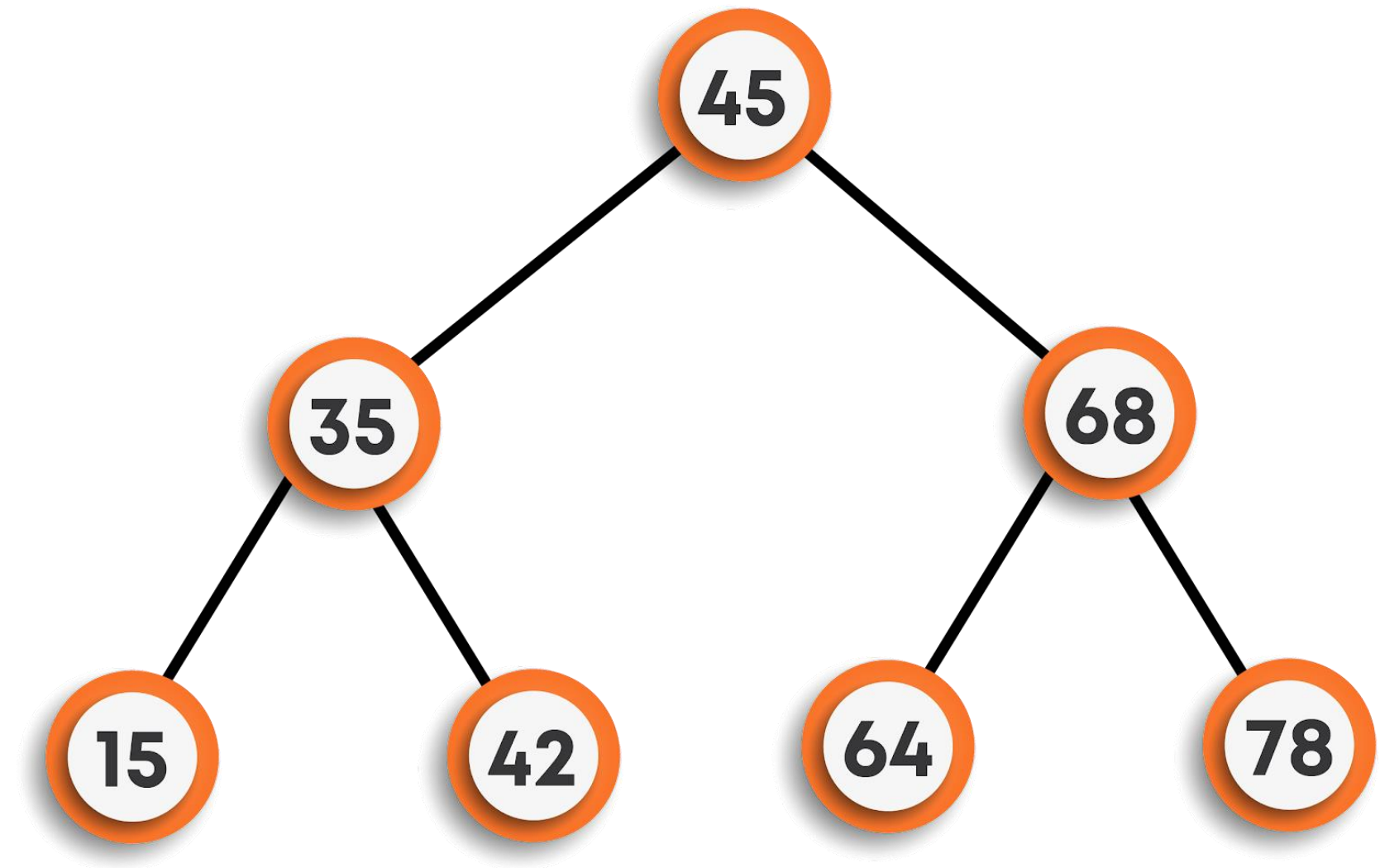
1. Now, on inserting 15, we will follow the same approach starting from the root node. Here, 15 < 45, means left subtree. Again, 15 < 35, means continue to move towards the left. As the left subtree of 35 is empty, so 15 will be the left child of 35.



1. Continuing further, to insert 64, now we found 64 > root node's data but less than 68, hence will be the left child of 68.



1. Finally, inserting 78, we can see that 78 > 45 and 78 > 68, so will be the right child of 68.



In this way, the data is stored in a BST.
If we follow the inorder traversal of the final BST, we will get all the elements in sorted order, hence to search we can now directly apply the binary search algorithm technique over it to search for any particular data.

As seen above, to insert an element, depending on the value of the element, we will either traverse the left subtree or right subtree of a node ignoring the other half straight away each time, till we reach a leaf node. Hence, the time complexity of insertion for each node is O(h) (where h is the height of the BST).

For inserting n nodes, the time complexity will be O(nh).

Note: The value of h is equal to logn on average, but can go to O(n) in the worst case (in the case of skew trees).

Applications

- **They are used to implement hashmaps and tree sets.**
- **They are used to implement dictionaries in various programming languages.**
- **They are also used to implement multi-level indexing in Databases.**

Operations in BST

Search in BST:

Given a value, we want to find out whether it is present in the BST or not.

Steps for search in the BST

- **Visit the root.**
- **If it is null return false.**
- **If it is equal to value return true.**
- **If the value is less than the root's data search in the left subtree else search in the right subtree.**

```
function search(root, val)
        if root is null
                return false
        /*
                If the current node's data is equal to val, then return true
                Else call the function for left and right subtree depending on
                the value of val
        */
        if root.data equals val
                return true

        if val < root.data
                return search(root.left, val)

        return search(root.right, val)
```

Insertion in BST:

Given a value, we want to insert it into the BST. We will assume that the value is already not present in the tree.
Steps for insert in the BST

- **Visit the root.**
- **If it is null, create a new node with this value and return it.**
- **If the value is less than the root's data insert in the left subtree else insert in the right subtree.**

```
function insert(root, val)
        if root is null
                /*
                        We have reached the correct place so we create a new
                        node with data val
                */
                temp =  newNode(val)
                return temp
        /*
                else we recursively insert to the left and the right subtree
                depending on the val
        */
        if val < root.data
                root.left = insert(root.left, val)
        else
                root.right = insert(root.right, val)
```

Deletion in BST:

Given a value, we want to delete it from the BST if present.
Steps for delete in the BST

- **Visit the root.**

- If root is null, return null
- If the value is greater than the root's data, delete in the right subtree.
- If the value is lesser than the root's data, delete in the left subtree.
- If the value is equal to the root's data, then
    - if it is a leaf, then delete it and return null.
    - if it has only one child, then return that single child.
    - else replace the root's data with the minimum in the right subtree and then delete that minimum valued node.

```
function deleteData(data, root)
        // Base case
        if root == is null
                return null

        // Finding that root by traversing the tree
        if (data > root.data)
                root.right = deleteData(data, root.right)
                return root
        else if (data < root.data)
                root.left = deleteData(data, root.left)
                return root

        // found the node with val as data
        else
                if (root.left == null and root.right == null)
                        // Leaf
                        delete root
                        return null
                else if (root.left == null)
                        // root having only right child
                        return root.right
                else if (root.right == null)
                        // root having only left child
                        return root.left
                else
                        // root having both the childs
                        minNode = root.right;
                        // finding the minimum node in the right subtree

                        while (minNode.left != null)
                                minNode= minNode.left

                        rightMin = minNode.data
                        root.data = rightMin
                        // now deleting that replaced node using recursion
                        root.right = deleteData(rightMin, root.right)
                        return root
```

The time complexity of various operations:

If n is the total number of nodes in a BST, and h is the height of the tree (which is equal to O(logn) on average and can be O(n) in worst case i.e. skew trees), then the time complexities of various operations for a single node in the worst case are as follows :

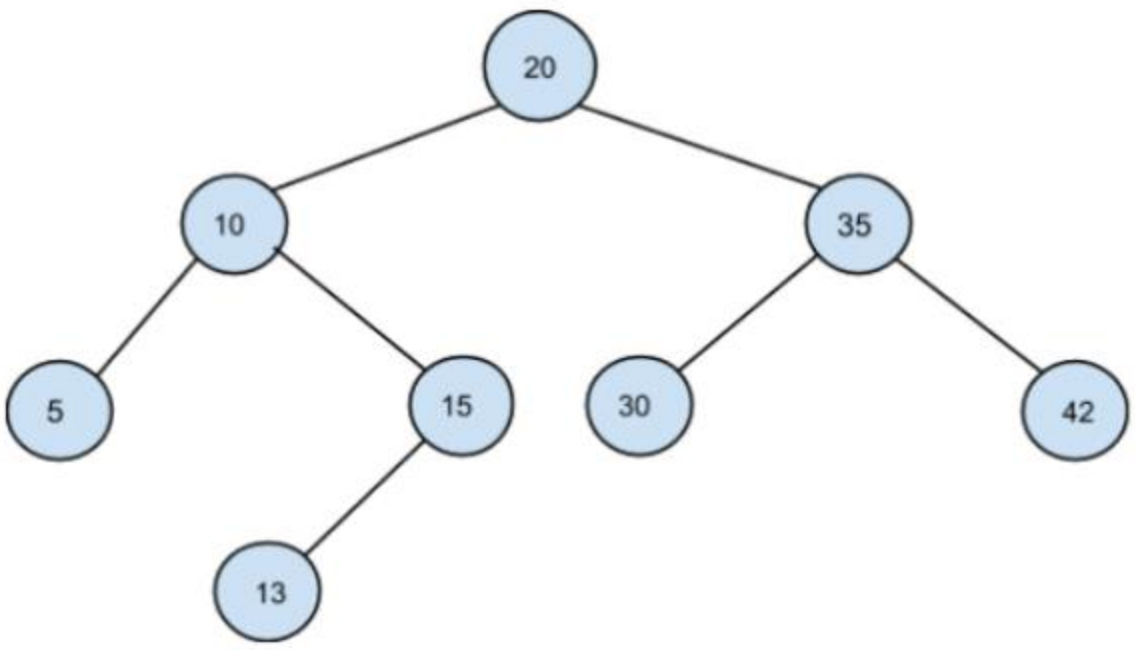
Operations	Time Complexity
Search(data)	O(h)
Insert(data)	O(h)
delete(data)	O(h)

Construction of BST notes

Construction

Construction from preorder traversal

Problem Statement: You are given the pre-order traversal of the array and you need to construct the original BST from it.
Example : For the array : [20, 10, 5, 15, 13, 35, 30, 42], the unique BST is :



The idea is based on the fact that the value of each node in a BST is greater than the value of all nodes in its left subtree and less than the value of all nodes in its right subtree.

So, instead of explicitly searching for indices that split the left and the right subtree (as in the previous approach), we will pass the information about the valid range of values for a node and its children in recursion itself.

- **We will maintain 'minVal' and 'maxVal' (initialized to a minimum and maximum possible value, respectively) to set a range for every node.**
- **We will also maintain 'preorderIndex' to keep track of the index in the PREORDER array.**
- **We initialize 'currVal' to PREORDER[preorderIndex].**
    - **If 'currVal' doesn't fall in the range [minVal: maxVal], we return NULL.**
    - **Else, we construct 'node' initialized to 'currVal' and increment 'preorderIndex'.**
- **We will then set the range for the left subtree ([minVal: currVal-1]) and the right subtree ([currVal+1: maxVal]) and recursively construct them.**

```
function preorderToBST(preorder, preorderIndex, minVal, maxVal)

        // base case to reach end of the array
        if preorderIndex == len(preorder)
                return Null
        // current value of node
        currVal = preorder[preorderIndex]
        // value outside the present range
        if currVal < minVal or currVal > maxVal
                return Null

        root = TreeNode(currVal)
        preorderIndex += 1
        // Recursively construct left subtree
        root.left = preorderToBST(preorder, preorderIndex, minVal, currVal - 1)
        // Recursively construct right subtree
        root.right = preorderToBST(preorder, preorderIndex, currVal + 1, maxVal)
        return root
```

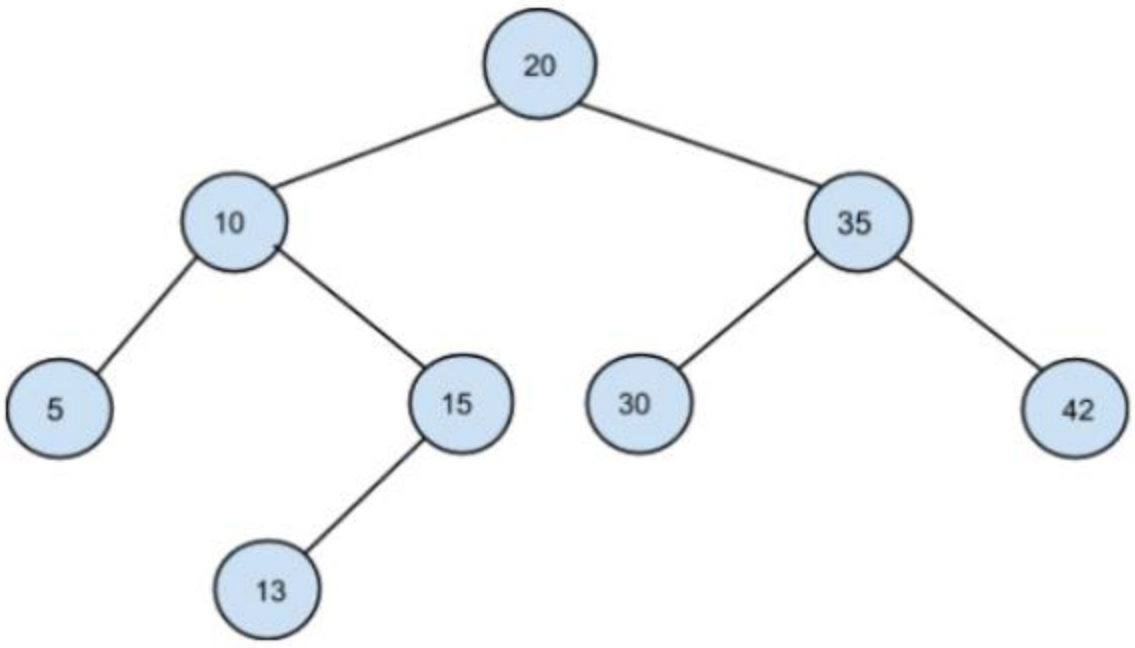
Time Complexity: O(N), where N is the number of nodes in the BST.
We are traversing through all the nodes of the tree in a preorder fashion.

Space Complexity: O(N), where N is the number of nodes in the BST.
O(H) recursion stack space is used by the algorithm. In the worst case, H will become N (in the case of skew trees). Thus, space complexity is O(N).

Construction from postorder traversal

Problem Statement: You are given the post-order traversal of the array and you need to construct the original BST from it.

Example : For the array : [5, 13, 15, 10, 30, 42, 35, 20], the unique BST is :



The idea is similar to the last time, with the only difference being that here we maintain a postorderIndex that starts from the end of the array and moves towards the beginning when we call the recursive function for the right subtree and then to the left subtree.

```
function postorderToBST(postorder, postorderIndex, minVal, maxVal)

        // base case to reach beginning of the array
        if postorderIndex == -1
                return Null
        // current value of node
        currVal = postorder[postorderIndex]
        // value outside the postsent range
        if currVal < minVal or currVal > maxVal
                return Null
        root = TreeNode(currVal)
        postorderIndex -= 1
        //   Recursively construct right subtree
        root.right =postorderToBST(postorder,postorderIndex, currVal+1, maxVal)

        //   Recursively construct left subtre
        root.left = postorderToBST(postorder, postorderIndex, minVal, currVal - 1)

        return root
```

Time Complexity: O(N), where N is the number of nodes in the BST.
We are traversing through all the nodes of the tree in a preorder fashion.

Space Complexity: O(N), where N is the number of nodes in the BST.
O(H) recursion stack space is used by the algorithm. In the worst case, H will become N (in the case of skew trees). Thus, space complexity is O(N).
Priority Queues & Heaps Notes

Priority Queues & Heaps

Introduction

A priority queue ADT is a data structure that supports the operation Insert and DeleteMin (which returns and removes the minimum element) or DeleteMax (which returns and deletes the max element).
A priority queue is called an ascending - priority queue, if the item with the smallest key has the highest priority (means delete the smallest element always). Similarly, a priority queue is called descending - priority queue if the item with the largest key has a greater priority (delete the maximum priority always). Since the two operations are symmetric we will be discussing ascending priority queues.

Difference between Priority Queue and Normal Queue

In a queue, the First-In-First-Out(FIFO) rule is implemented whereas, in a priority queue, the values are removed based on priority. The element with the highest priority is removed first.

Operation on Priority Queues

Main Priority Queue Operations

- **Insert (key, data): Inserts data with a key to the priority queue. Elements are ordered based on key.**

- **DeleteMin / DeleteMax: Remove and return the element with the smallest / largest key.**
- **GetMinimum/GetMaximum: Return the element with the smallest/largest key without deleting it.**

Auxiliary Priority Queues Operations

- **kth - Smallest/kth – Largest: Returns the kth -Smallest / kth –Largest key in the priority queue.**
- **Size: Returns the number of elements in the priority queue.**
- **Heap Sort: Sorts the elements in the priority queue based on priority (key).**

Priority Queues Implementation

Before discussing the actual implementations, let us enumerate the possible options.

- **Unordered Array Implementation: Elements are inserted into the array without bothering the order. Deletions are performed by searching the minimum or maximum and deleting.**
- **Unordered List Implementation: It is similar to array implementation but instead of array linked lists are used.**
- **Ordered Array Implementation: Elements are inserted into the array in sorted order based on the key field. Deletions are performed only at one end of the array.**
- **Ordered list implementation: Elements are inserted into the list in sorted order based on the key field. Deletions are performed at only one end, hence preserving the status of the priority queue. All other functionalities associated with a linked list ADT are performed without modification.**
- **Binary Search Trees Implementation: Insertion and deletions are performed in a way such that the property of BST is reserved. Both the operations take O(logn) time on average and O(n) in the worst case.**
- **Balanced Binary Search Trees Implementation: Insertion and deletions are performed such that the property of BST is reserved and the balancing factor of each node is -1, 0, or 1. Both operations take O(logn) time.**
- **Binary Heap Implementation: We will be discussing this in detail. For now, just compare the time complexities.**

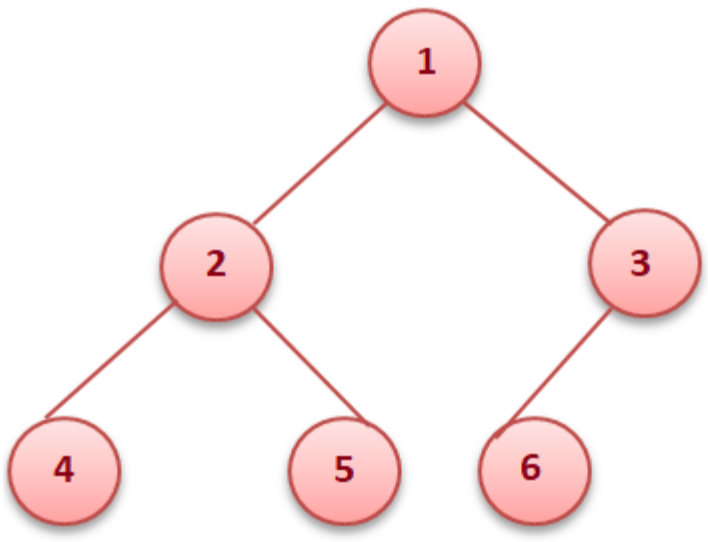
Comparing Implementation

Implementation	Insertion	Deletion(Delete max/min)	Find (Max/Min)
Unordered Array	1	n	n
Unordered List	1	n	n
Ordered Array	n	1	1
Ordered List	n	1	1
Binary Search Trees	logn(average)	logn(average)	logn(average)
Balanced Binary Search Trees	logn	logn	logn
Binary Heaps	logn	logn	1

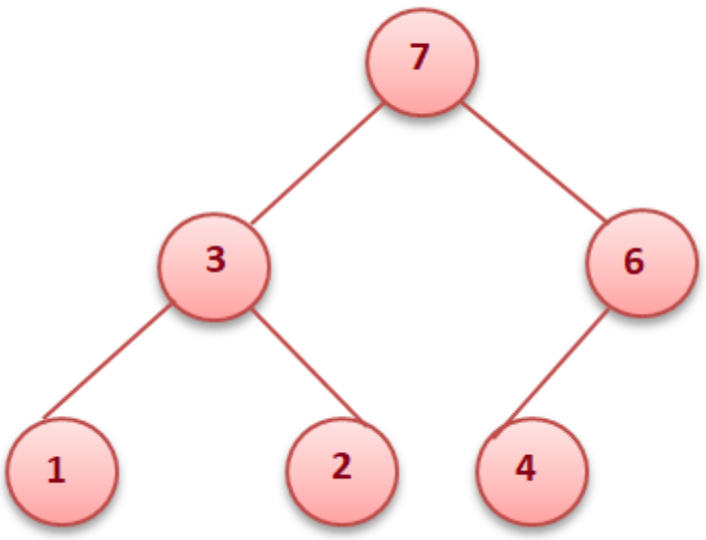
Heaps

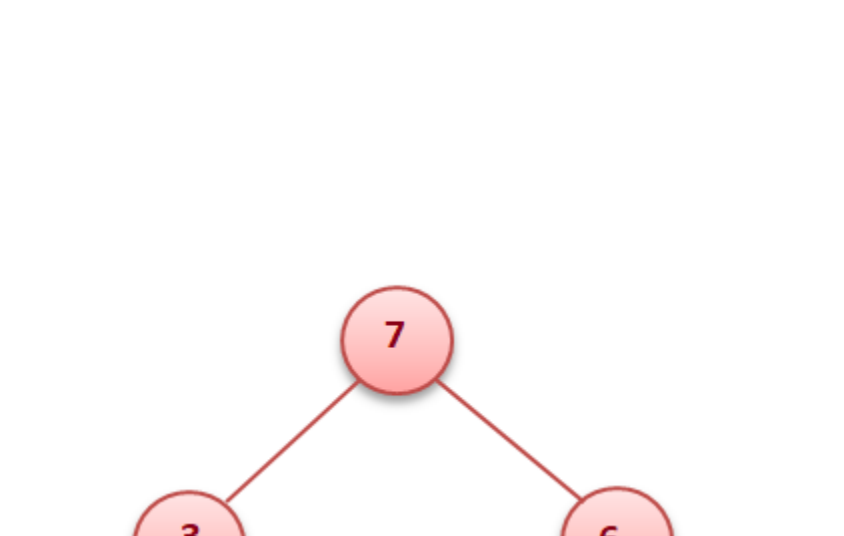
- **A heap is a binary tree with some special properties.**
- **The basic requirement of a heap is that the value of a node must be ≥ (or ≤) than the values of its children. This is called the heap property.**
- **A heap also has the additional property that all leaf nodes should be at h or h – 1 level (where h is the height of the tree) for some h > 0 (complete binary trees).**

That means the heap should form a complete binary tree (as shown below).



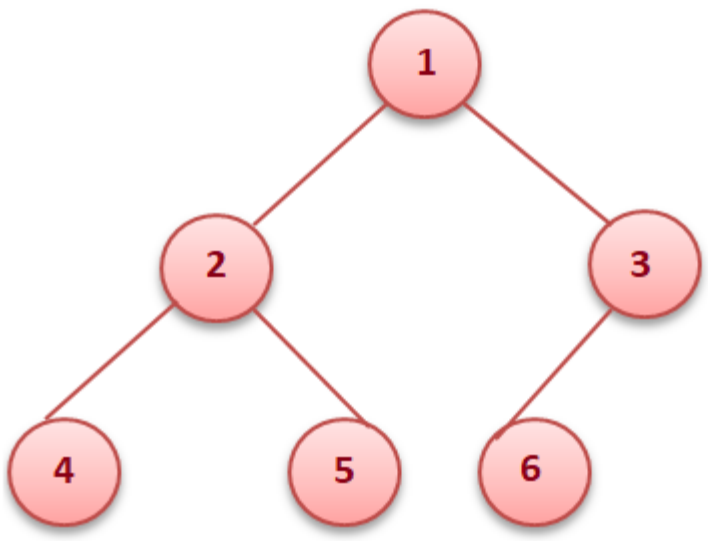
In the examples below, the left tree is a heap (each element is greater than its children) and the right is not a heap (since 11 is greater than 2 and 5 whereas the rest of the nodes are less than their children).



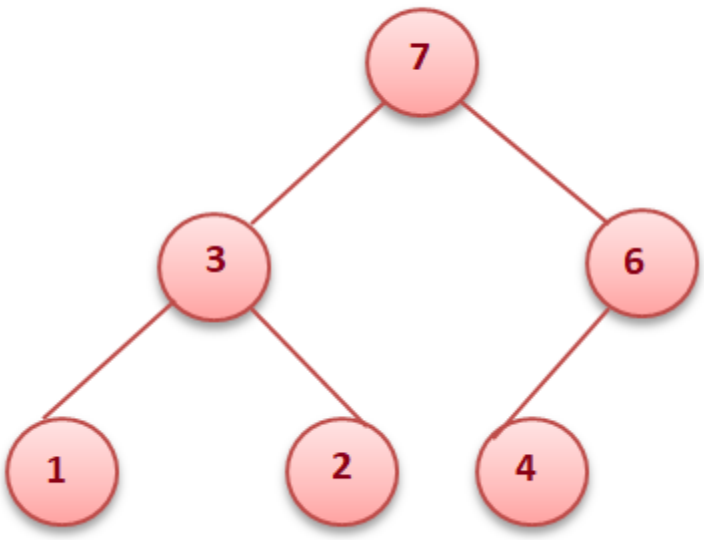


Types of heap

- **Min heap: The value of every node must be less than or equal to its children.**



- **Max heap: The value of every node must be greater than or equal to its children.**



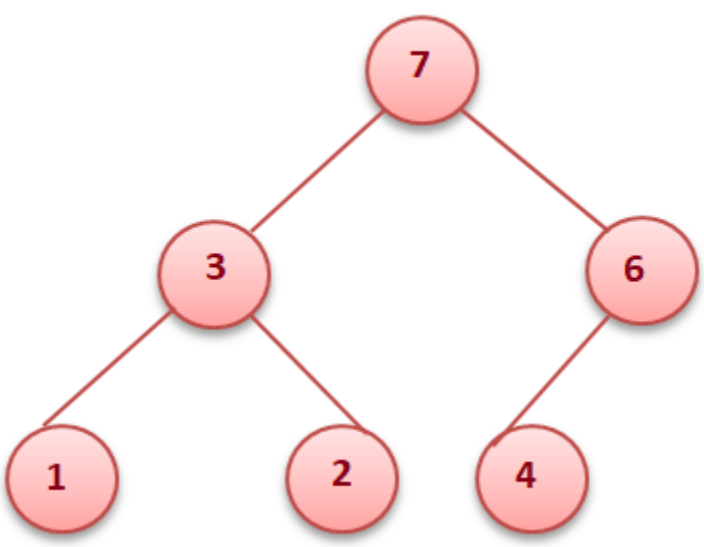
Binary Heaps

- **In a binary heap, each node may have up to two children.**
- **In practice, binary heaps are enough and we concentrate on binary min heaps and binary max heaps for the remaining discussion.**

Representing heaps

Before looking at heap operations, let us see how heaps can be represented. One possibility is using arrays. Since heaps are forming complete binary trees, there will not be any wastage of locations. For the discussion below let us assume that elements are stored in an array, which starts at index 0. The previous max heap can be represented as:

Parent of a Node: For a node at index i, its parent is at index (i - 1) / 2. In the fig below element 6 is at the second index and its parent is at the 0th index.



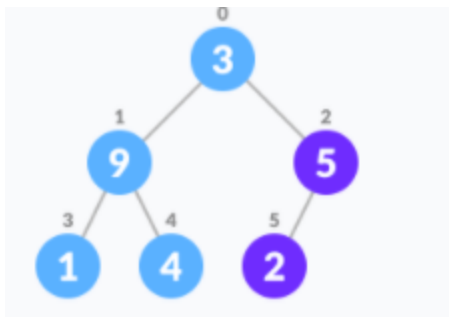
Children of a Node : For a node at index i, its children are at index (2*i + 1) and (2i + 2). Like in the fig above 3 is at index 1 and has children at index 3 (2 * 1 + 1) and at index 4 (2 * 1 + 2) .
Getting the maximum/minimum element: The maximum element in a max heap, or the minimum element in a min-heap, is always the root node. It will be stored at the 0th index.

Heapify

Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

- **Let the input array be**

- **Create a complete binary tree from the array**
- **Start from the first index of the non-leaf node whose index is given by n / 2 - 1.**



- Set current element i as largest.
- The index of the left child is given by 2i + 1 and the right child is given by 2i + 2.
- If leftChild is greater than currentElement (i.e. element at the ith index), set leftChildIndex as largest.
- If rightChild is greater than the element in largest, set rightChildIndex as largest.
- Swap largest with currentElement.

- Repeat steps 3-7 until the subtrees are also heapified.

The above steps are for Max-Heap. For Min-Heap, both leftChild and rightChild must be smaller than the parent for all nodes.

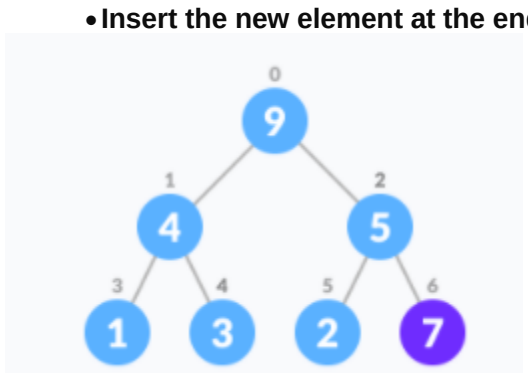
Pseudocode:

```
function  heapify(int i)

        largest = i
        l = 2 * i + 1                                   //   Index of Left Child
        r = 2 * i + 2                                   //   Index of Right Child

        if l < n && heap[i] < heap[l]
                largest = l

        if (r < n && heap[largest] < heap[r])
                largest = r

        if largest is not equal to i
                temp = heap[i]
                heap[i]  = heap[largest]
                heap[largest] = temp
                heapify(largest)
```

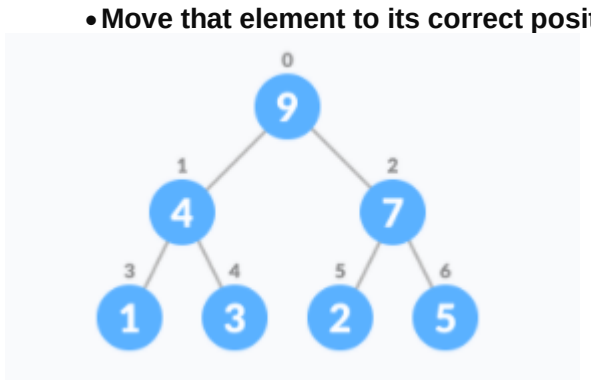
Inserting into a heap

Insertion into a heap can be done in two steps:

- **Insert the new element at the end of the tree.**



- **Move that element to its correct position in the heap.**



Pseudocode:

```
function insert(element) {

        //   add an element to the end of the heap list.
        heap.add(element)

        childIndex = heap.length - 1
        parentIndex = (childIndex - 1) / 2

        while(childIndex > 0)
```

```
                if heap[parentIndex] < heap[childIndex])

                        temp = heap[parentIndex]
                        heap[parentIndex] = heap[childIndex]
                        heap[childIndex] = temp
                        childIndex = parentIndex
                        parentIndex = (childIndex - 1) / 2

                else
                        break
```

Delete Max Element from Max Heap

There are three easy steps to remove max from the max heap, we know that the 0th element would be the max.

- **Swap the 0th element with the last element.**
- **Remove the last element from the heap.**
- **Heapify the 0th element.**

If you want to return the max element, then store it before replacing it with the last index, and return it in the end.

Pseudocode:

```
function removeMax(){

        if heap is empty
                print "heap is empty"
                    return

        retVal = heap[0]
        heap[0] = heap[heap.length - 1]
        heap.remove(heap.length - 1)

        if(heap.size() > 1)
                 heapify(0)

        return retVal
```

Implementation of Priority Queue

- **Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree.**
- **Among these data structures, heap data structure provides an efficient implementation of priority queues.**
- **Hence, we will be using the heap data structure to implement the priority queue.**

Insertion and Deletion in a Priority Queue

The first step would be to represent the priority queue in the form of a max / min-heap. Once it is heapified, the insertion and deletion operations can be performed similarly to that in a Heap.

Heap Sort

- **Heap Sort is another example of an efficient sorting algorithm. Its main advantage is that it has a worst-case runtime of O(nlog(n)) regardless of the input data.**
- **As the name suggests, Heap Sort relies heavily on the heap data structure - a common implementation of a Priority Queue.**
- **Without a doubt, Heap Sort is one of the simplest sorting algorithms to implement, and coupled with the fact that it's a fairly efficient algorithm compared to other simple implementations, it's a common one to encounter.**

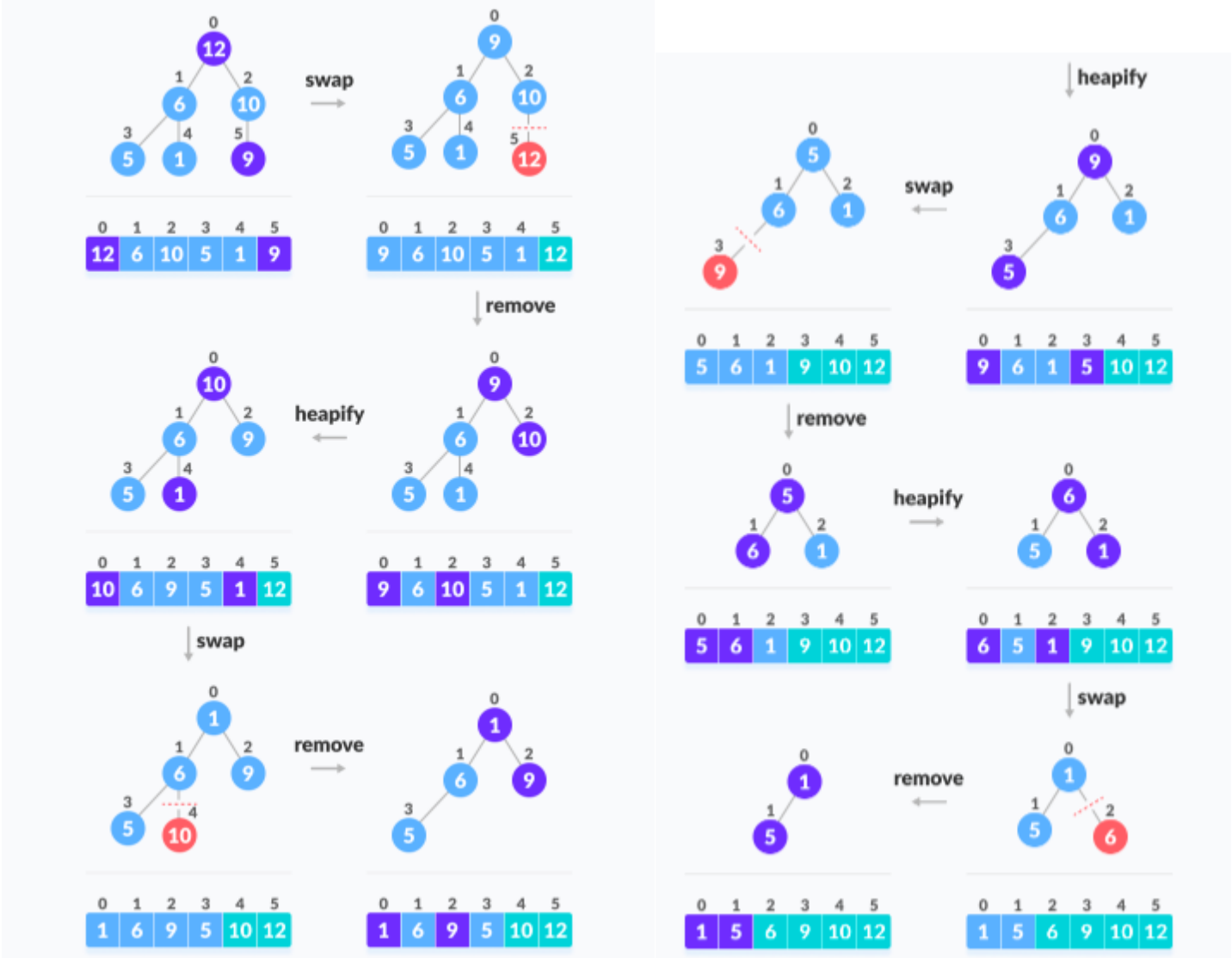
Algorithm

- **The heap-sort algorithm inserts all elements (from an unsorted array) into a maxheap.**
- **Note that heap sort can be done in-place with the array to be sorted.**
- **Since the tree satisfies the Max-Heap property, then the largest item is stored at the root node.**
- **Swap: Remove the root element and put it at the end of the array (nth position: n-1 index)**
- **Put the last item of the tree (heap) at the vacant place.**
- **Remove: Reduce the size of the heap by 1.**
- **Heapify: Heapify the root element again so that we have the highest element at root.**

- **The process is repeated until all the items in the list are sorted.**

Consider the given illustrated example:

Applying heapsort to the unsorted array [12, 6, 10, 5, 1, 9]



Pseudocode:

```
function heapSort(heap, startIndex, endIndex)

        i = heap.length / 2 - 1
        while i is greater than or equal to 0
                heapify(input, i, input.length)
                i--

        n = heap.length
        i = n-1
        while i is greater than equal to 0
                //   Move current root to end
                temp = heap[0]
                heap[0] = heap[i]
                heap[i] = temp
                i--

        //   call heapify on the reduced heap
        heapify(input, 0, i)

function heapify(heap, index, arrLength)

        largest = index
        left = 2 * index + 1
        right = 2 * index + 2

        if left < arrLength and heap[left] > heap[largest]
                largest = left

        if right < arrLength and input[right] > input[largest]
                largest = right

        if largest != index
                k = input[index]
                input[index] = input[largest]
                input[largest] = k
                heapify(input, largest, arrLength)
```

Applications Of Priority Queues

- **It is used in data Compression: Huffman Coding Algorithm**
- **Used in shortest path algorithms: Dijkstra's Algorithm**
- **Used in the minimum spanning tree algorithms: Prim's algorithm**
- **Used in Event-driven simulations: customers in a line.**
- **Used in selection problems: kth - smallest element.**

Introduction To Graph Notes

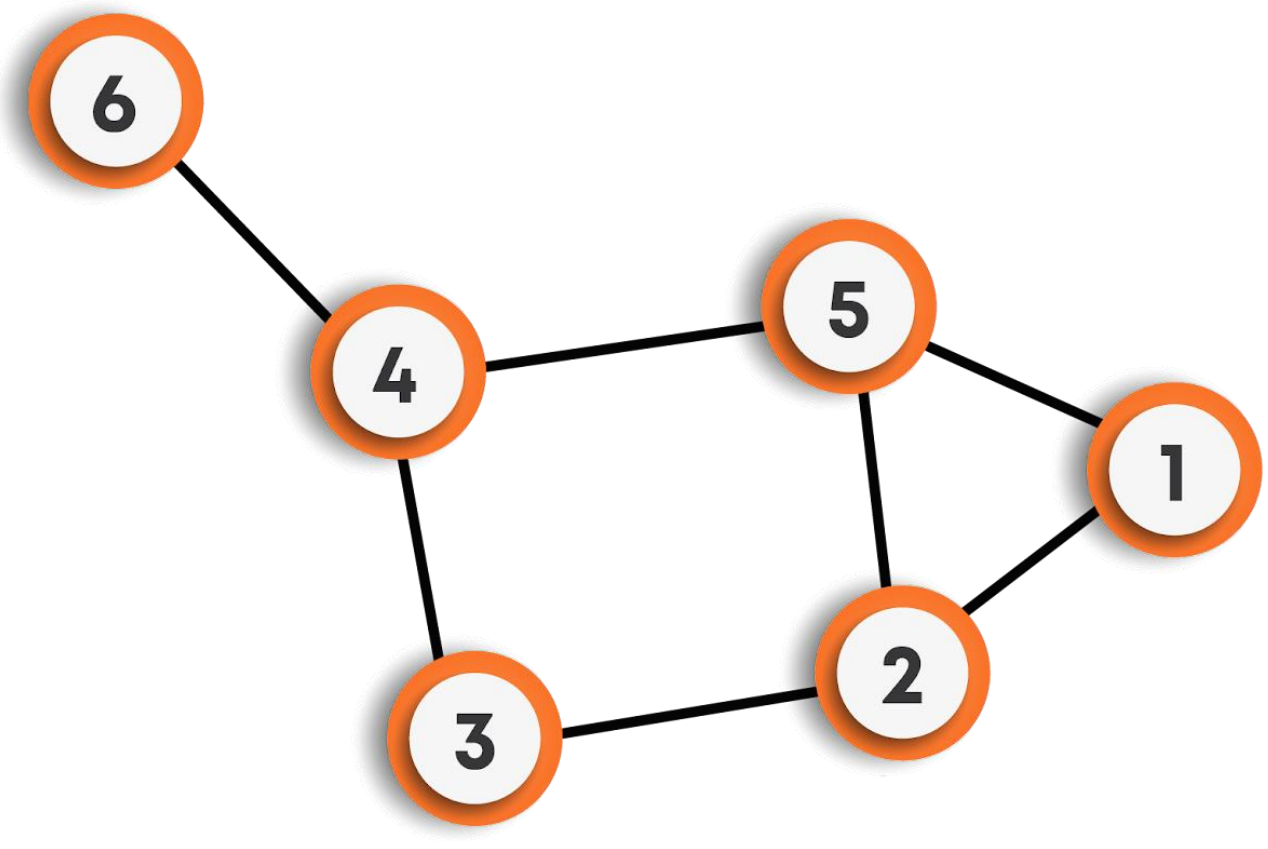
Introduction to graphs

A graph G is defined as an ordered set (*V*, *E*), where *V(G)* represents the set of *vertices*, and *E(G)* represents the *edges* that connect these vertices.

The vertices *x* and *y* of an edge {*x*, *y*} are called the *endpoints* of the edge. The edge is said to *join x* and *y* and to be *incident* on *x* and *y*. A vertex may not belong to any edge.

For example: Suppose there is a road network in a country, where we have many cities, and roads are connecting these cities. There could be some cities that are not connected by some other cities like an island. This structure seems to be non-uniform, and hence, we can't use trees to store it. In such cases, we will be using graphs. Refer to the figure below for a better understanding.

***V(G)* = {1,2,3,4,5,6}**
***E(G)* = {(1,2), (1,5), (2,5), (5,4), (2,3), (3,4), (4,6)}**



A graph can be undirected or directed.

- **Undirected Graphs: In undirected graphs, edges do not have any direction associated with them. In other words, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A.**
- **Directed Graphs: In directed graphs, edges form an ordered pair. In other words, if there is an edge from A to B, then there is a direct path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as an initial node) and terminate at node B (terminal node).**

A graph can be weighted or unweighted.

- **Unweighted Graphs: In unweighted graphs, an edge does not contain any weight.**
- **Weighted Graphs: If edges in a graph have weights associated with them, then the graph is said to be weighted.**

Relationship between graphs and trees

- **A tree is a special type of graph in which we can reach any node from any other node using some path that is unique, unlike the graphs where this condition may or may not hold.**
- **A tree is an undirected connected graph with N vertices and exactly N-1 edges.**
- **A tree does not have any cycles in it, while a graph may have cycles.**

Graph Terminology

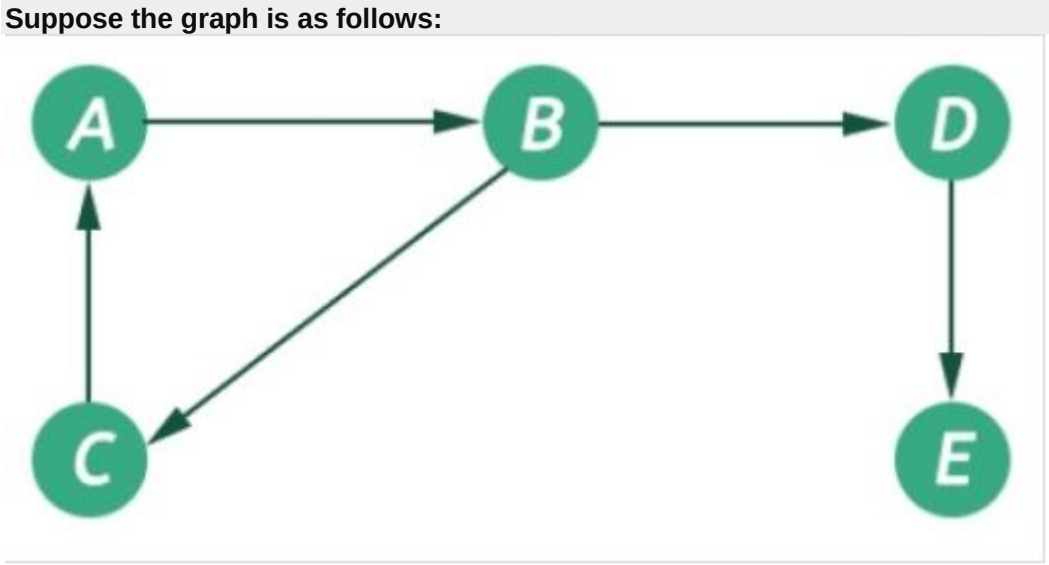
- **Nodes are called vertices, and the connections between them are called edges.**
- **Two vertices are said to be adjacent if there exists a direct edge connecting them.**
- **The degree of a node is defined as the number of edges that are incident to it. If the degree of vertex u is 0, it means that u does not belong to any edge and such a node is known as an isolated vertex.**

- A loop is an edge that connects a vertex to itself.
- Distinct edges that connect the same end-points are called multiple edges.
- A path is a collection of edges through which we can reach from one node to the other node in a graph. A path P is written as P = {v0,v1,v2,....,vn} of length n from a node u to node v, is defined as a sequence of (n+1) nodes. Here u = v0, v = vn and vi-1 is adjacent to vi for i = 1,2,3,.....,n.
- A path in which the first and the last vertices are the same forms a cycle.
- A graph is said to be simple, if it is undirected and unweighted, containing no self-loops and multiple edges.
- A graph with multiple edges and/or self-loops is called a multi-graph.
- A graph is said to be connected if there is a path between every pair of vertices.
- If the graph is not connected, then all the maximally connected subsets of the graphs are called connected components. Each component is connected within itself, but two different components of a graph are never connected.

- The minimum number of edges in a graph can be zero, which means a graph could have no edges as well.
- The minimum number of edges in a connected graph will be (N-1), where N is the number of nodes.
- In a complete graph (where each node is connected to every other node by a direct edge), there are NC2 number of edges means (N * (N-1)) / 2 edges, where N is the number of nodes, this is the maximum number of edges that a simple graph can have.
- Hence, if an algorithm works on the terms of edges, let's say O(E), where E is the number of edges, then in the worst case, the algorithm will take O(N2) time, where N is the number of nodes.
- DIRECTED GRAPHS -
    - The number of edges that originate at a node is the out-degree of that node.
    - The number of edges that terminate at a node is the in-degree of that node.
    - The degree of a node is the sum of the in-degree and out-degree of the node.
    - A digraph or directed graph is said to be strongly connected if and only if there exists a path between every pair of nodes in the graph.
    - A digraph is said to be a simple directed graph if and only if it has no parallel edges. However, a simple directed graph may contain cycles with the exception that it cannot have more than one loop at a given node.
    - A digraph is said to be a directed acyclic graph(DAG) if the directed graph has no cycles.

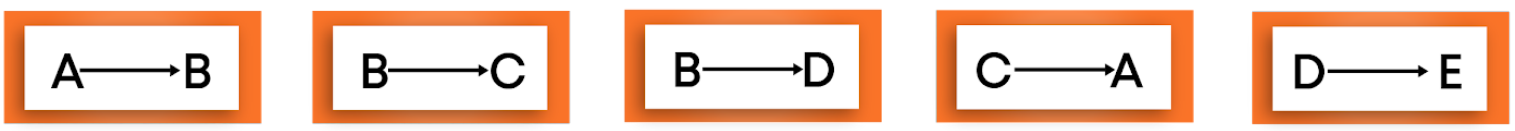
Graphs Representation

Suppose the graph is as follows:



There are the following ways to implement a graph:

- Using edge list: We can create a class that could store an array of edges. The array of edges will contain all the pairs that are connected, all put together in one place. It is not preferred to check for a particular edge connecting two nodes; we have to traverse the complete array leading to O(n2) time complexity in the worst case. Pictorial representation for the above graph using the edge list is given below:

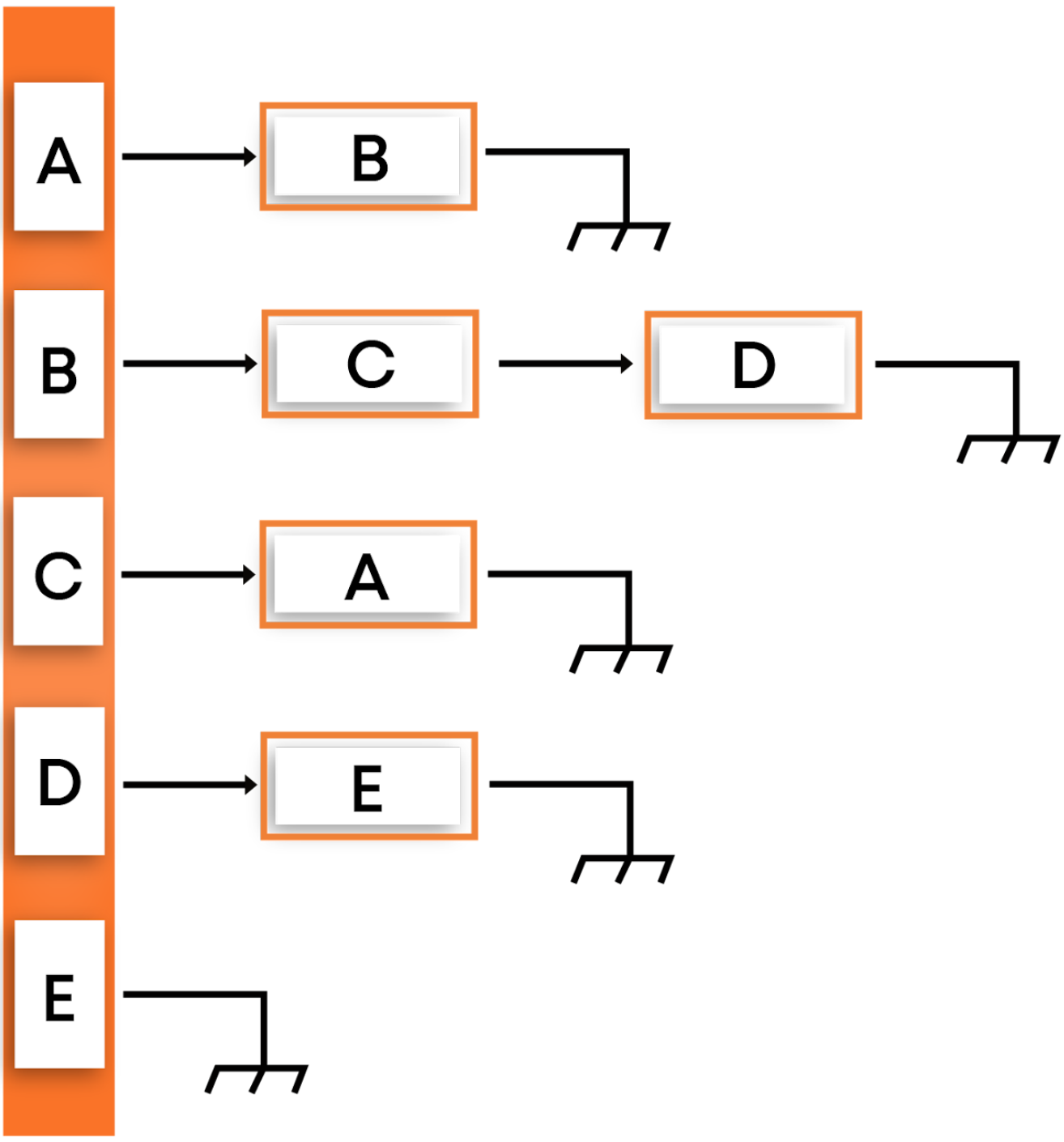


- Adjacency list: We will create an array of vertices, but this time, each vertex will have its list of edges connecting this vertex to another vertex. The key advantages of using an adjacency list are:
    - It is easy to follow and clearly shows the adjacent nodes of a particular node
    - It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
    - Adding new nodes in G is easy and straightforward when G is represented using an adjacency list.
- Adjacency matrix: Here, we will create a 2D array where the cell (i, j) will denote an edge between node i and node j. The major disadvantage of using the adjacency matrix is vast space consumption compared to the adjacency list, where each node stores only those nodes that are directly connected to them. For the above graph, the adjacency matrix looks as follows:

- For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal
- The adjacency matrix of an undirected graph is symmetric.
- The memory use of an adjacency matrix for n nodes is O(n^2), where n is the number of nodes in the graph.

- **The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.**

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	1	0	0	0	0
D	0	0	0	0	1
E	0	0	0	0	0



Graph Traversal Notes

Graph traversal algorithms

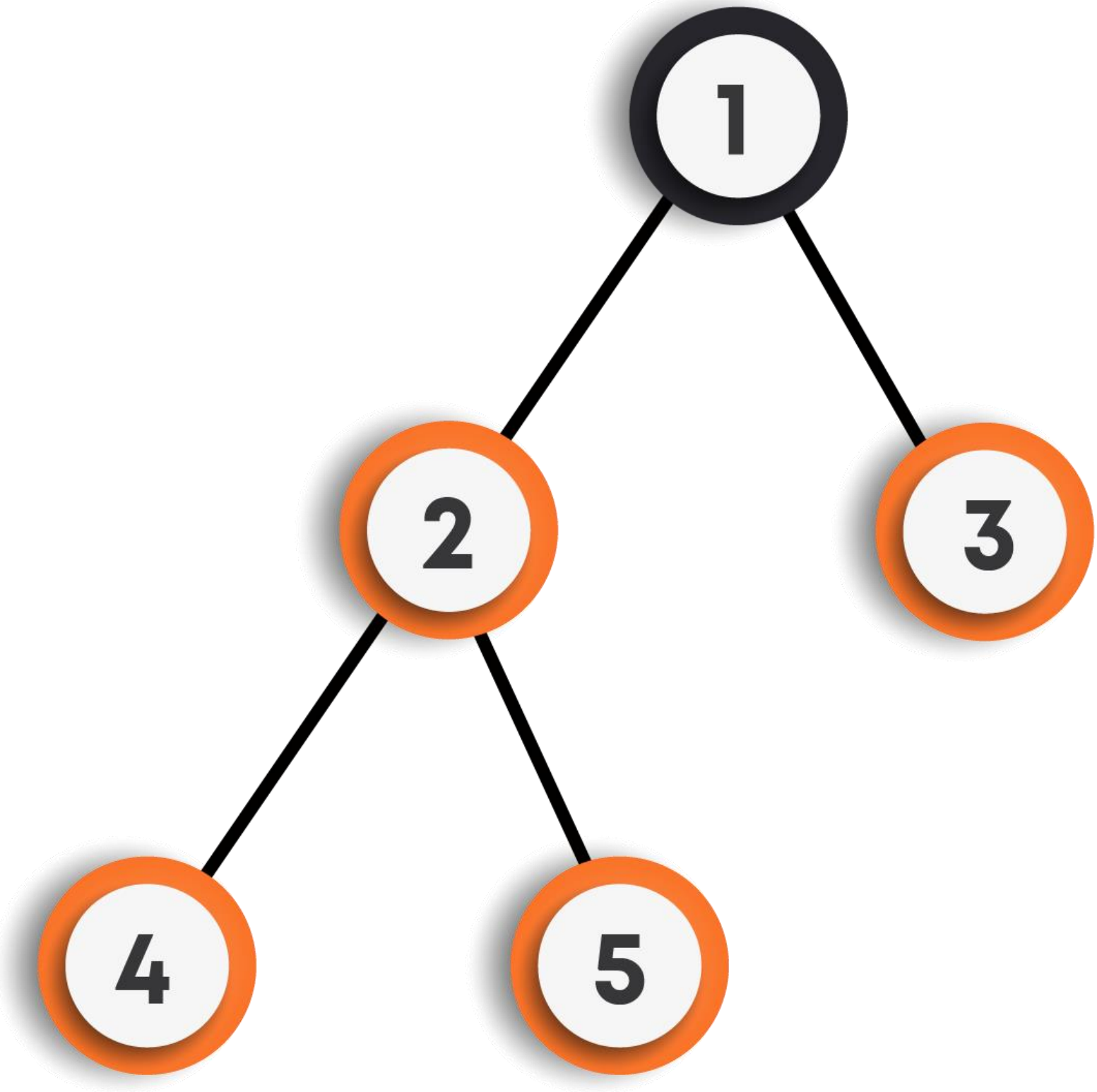
Traversing a graph means examining the nodes and edges of the graph. There are two standard methods of graph traversal which we will discuss in this section. These two methods are -

a) Depth-first search
b) Breadth-first search.

While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a stack. But both these algorithms make use of a bool variable VISITED. During the execution of the algorithm, every node in the graph will have the variable VISITED set to false or true, depending on its current state, whether the node has been processed/visited or not.

Depth-first search (DFS)

The depth-first search(DFS) algorithm, as the name suggests, first goes into the depth and then recursively does the same in other directions, it progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
In other words, the depth-first search begins at a starting node A which becomes the current node. Then, it examines each node along with a path P which begins at A. That is, we process a neighbor of A, then a neighbor of the processed node, and so on. During the execution of the algorithm, if we reach a path that has a node that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.
For example: DFS for the below graph is:



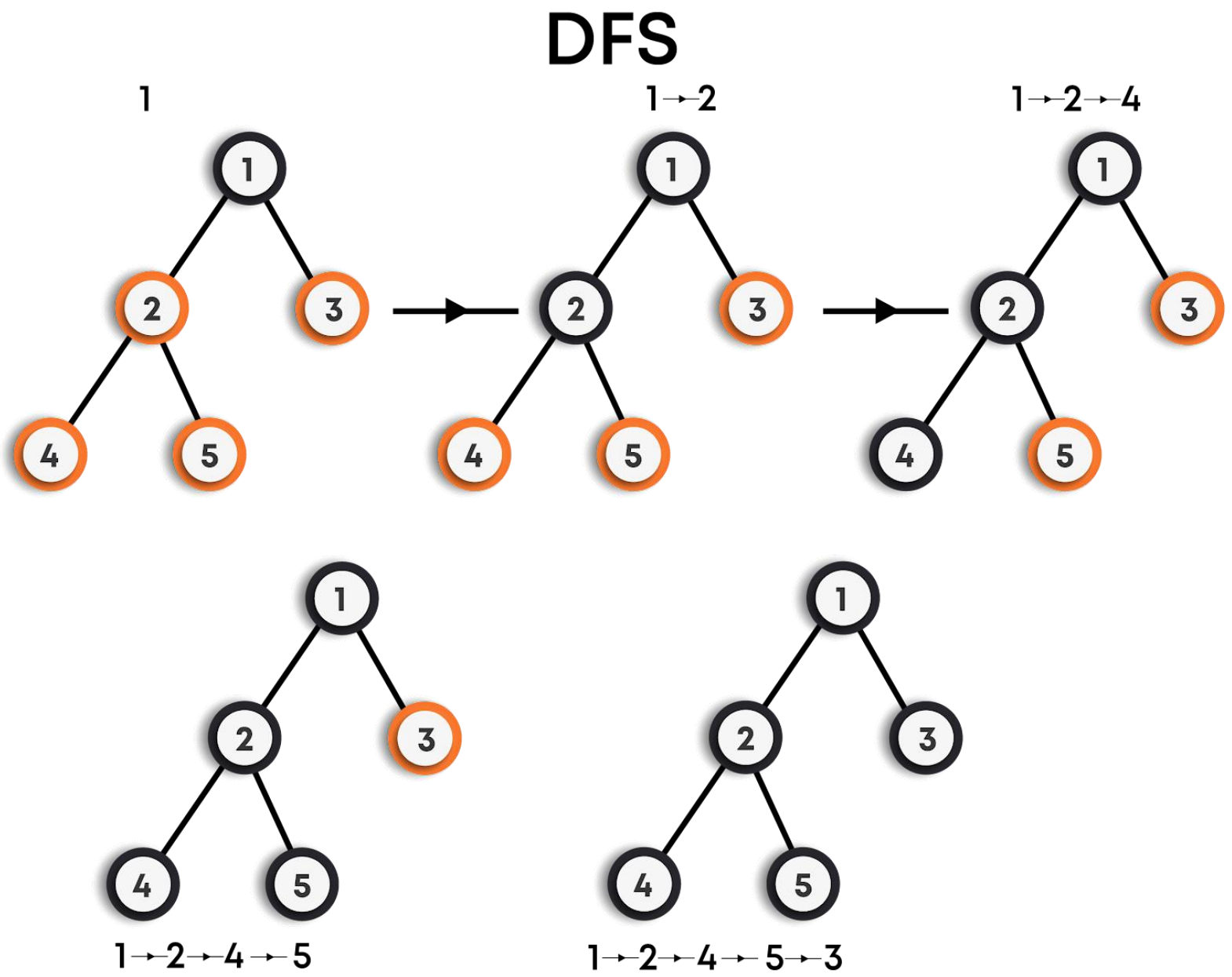
DFS Traversal : 1->2->4->5->3

Other possible DFS traversals for the above graph can be:

1. **1->3->2->4->5**
2. **1->2->5->4->3**
3. **1->3->2->5->4**

We can clearly observe that there can be more than one DFS traversals for the same graph.

Implementation of DFS (Iterative) :




```
function DFS_iterative(graph,source)

        /*
        Let St be a stack, pushing source vertex in the stack.
        St represents the vertices that have been processed/visited
        so far.
        */
        St.push(source)

        //   Mark source vertex as visited.
        visited[source] = true
        //   Iterate through the vertices present in the stack
        while St is not empty
                //   Pop a vertex from the stack to visit its neighbors
                cur = St.top()
                St.pop()
                /*
                Push all the neighbors of the cur vertex that have not been visited yet, push them into the
stack and mark them as visited.
                */
                for all neighbors v of cur in graph:
                        if visited[v] is false
                                St.push(v)
                                visited[v] = true

        return
```

Implementation of DFS (Recursive)

```
function DFS_recursive(graph,cur)
        //   Mark the cur vertex as visited.
        visited[cur] = true
        /*
        Recur for all the neighbors of the cur vertex that have not been visited yet.
        */
        for all neighbors v of cur in graph:
                if visited[v] is false
```

```
                        DFS_recursive(graph,v)
        return
```

Features of Depth-First Search Algorithm

Time Complexity: The time complexity of a depth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as (O(|V| + |E|)), where |V| is the number of vertices and |E| is the number of edges in the graph considering the graph is represented by adjacency list.

Completeness: Depth-first search is said to be a complete algorithm in the case of a finite graph. If there is a solution, a depth-first search will find it regardless of the kind of graph. But in the case of an infinite graph, where there is no possible solution, it will diverge.

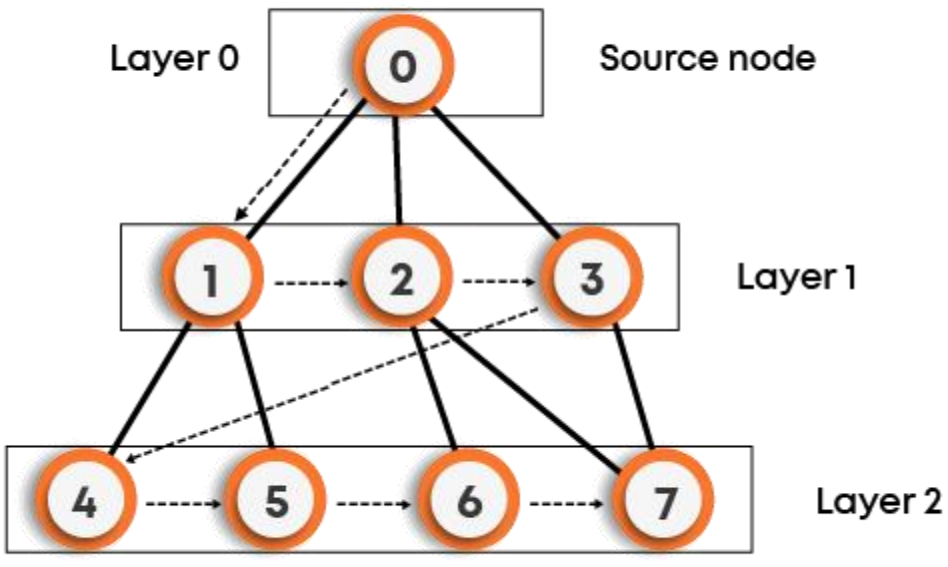
Applications of Depth-First Search Algorithm

- **Finding a path between two specified nodes, u and v, of an unweighted graph.**
- **Finding a path between two specified nodes, u and v, of a weighted graph.**
- **Finding whether a graph is connected or not.**
- **Computing the spanning tree of a connected graph**

Breadth-first search (BFS)

Breadth-first search(BFS) is a graph search algorithm where we start from the selected node and traverse the graph level-wise or layer-wise, thus exploring the neighbor nodes (which are directly connected to the starting node), and then moving on to the next level neighbor nodes.
As the name suggests:

- **We first move horizontally and visit all the nodes of the current layer.**
- **Then move to the next layer.**



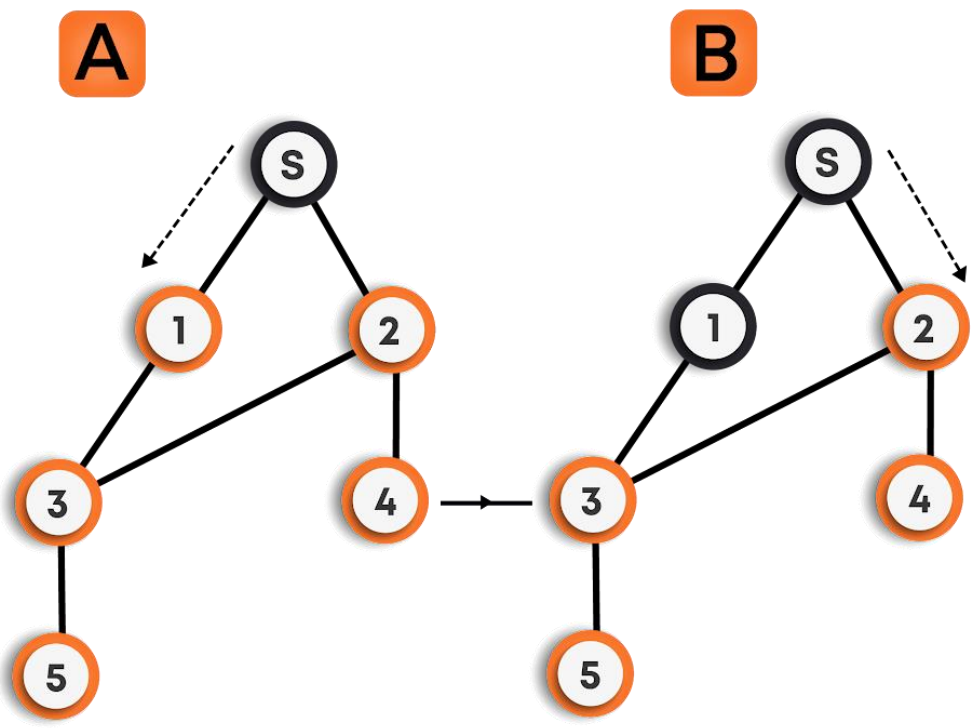
BFS Traversal: 0->1->2->3->4->5->6->7
Other possible BFS traversals for the above graph can be:

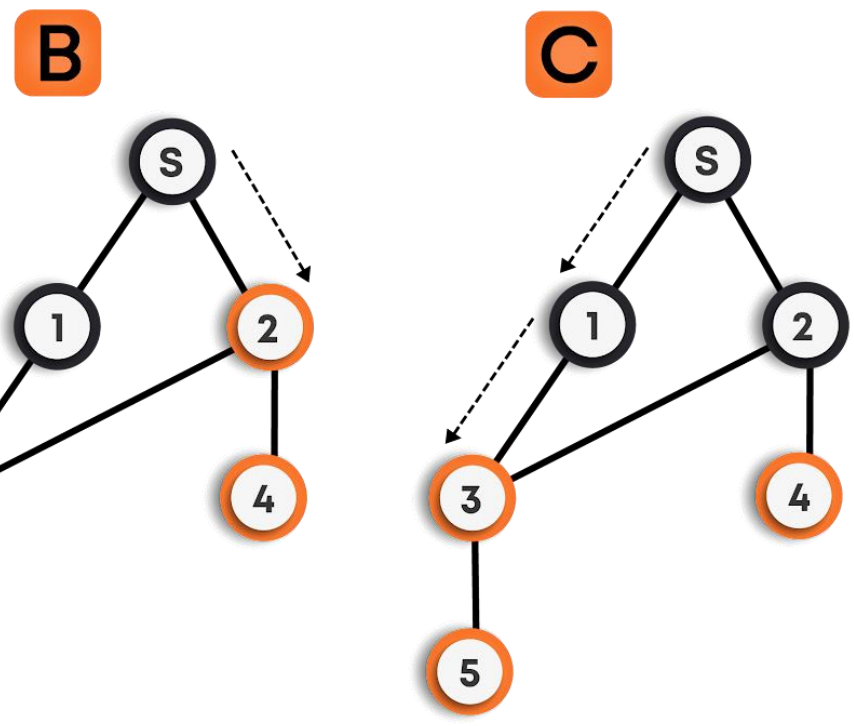
1. **0->3->2->1->7->6->5->4**
2. **0->1->2->3->5->4->7->6**

We can clearly observe that there can be more than one BFS traversals for the same graph, as shown for the above graph.

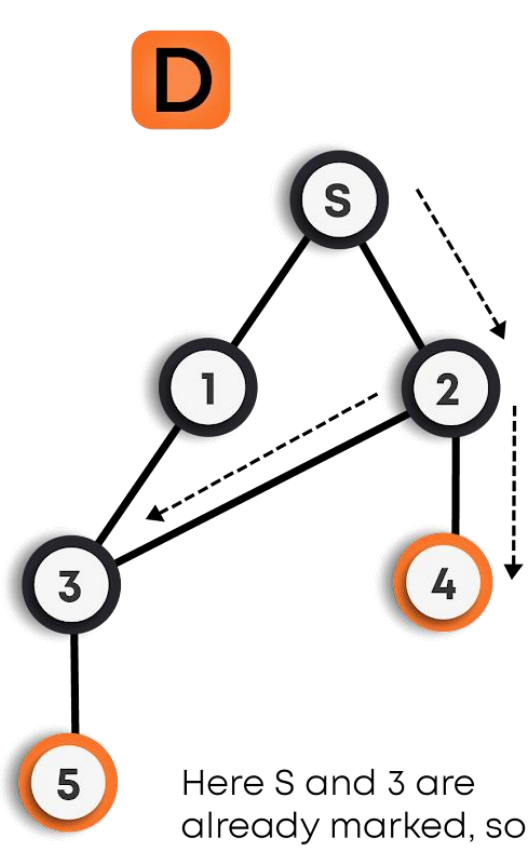
That is, we start examining say node A and then all the neighbors of A are examined. In the next step, we examine the neighbors of A, so on, and so forth. This means that we need to track the neighbors of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable VISITED to represent the current state of the node.

For example: BFS for the below graph is:

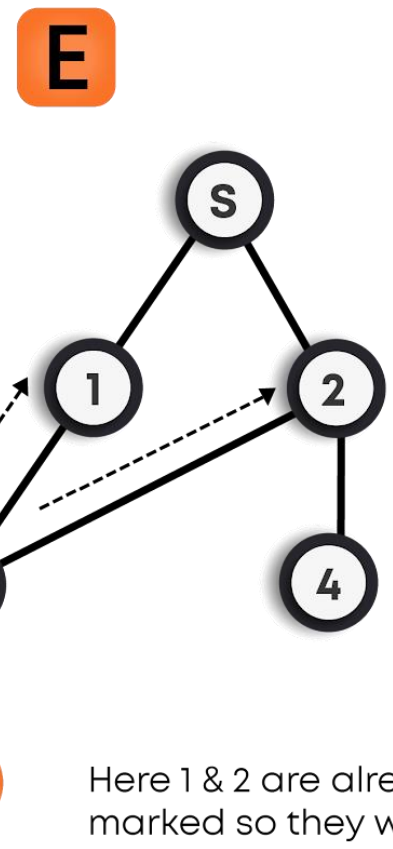




Here S is already marked, so it will be ignored



Here S and 3 are already marked, so they will be ignored



Here 1 & 2 are already marked so they will be ignored

Implementation of BFS

```
function BFS(graph,source)
        /*
        Let Q be a queue, pushing source vertex in the queue.
        Q represents the vertices that have not been processed
        /visited so far, and their neighbors have not been processed.
        */
        Q.enqueue(source)
        visited[source] = true
        //   Iterate through the vertices in the queue.
        while Q is not empty
                //   Pop a vertex from the queue to visit its neighbors
                cur = Q.front()
                Q.dequeue()
                /*
```

```
                Push all the neighbors of the cur vertex that have not been visited yet, push them into the
queue and mark them as visited.
                */
                        for all neighbors v of cur in graph:
                                if visited[v] is false
                                        Q.enqueue(v)
                                        visited[v] = true

        return
```

Features of Breadth-First Search Algorithm

Time Complexity: The time complexity of a breadth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as (O(|V| + |E|)), where |V| is the number of vertices in the graph and |E| is the number of edges in the graph, considering the graph is represented by adjacency list.

Completeness: Breadth-first search is said to be a complete algorithm in the case of a finite graph because if there is a solution, the breadth-first search will find it regardless of the kind of graph. But in the case of an infinite graph where there is no possible solution, it will diverge.

Applications of Breadth-First Search algorithm

- **Finding all connected components in a graph G.**
- **Finding all nodes within an individual connected component**
- **Finding the shortest path between two nodes, u and v, of an unweighted graph. (The shortest path is in terms of the minimum number of moves required to visit v from u or vice-versa).**
- **NOTE**
    - **The DFS and BFS algorithms work for both directed and undirected graphs, weighted and unweighted graphs, connected and disconnected graphs.**
    - **For disconnected graphs, for traversing the whole graph, we need to call DFS/BFS for each unvisited vertex.**
    - **For getting the number of connected components in a graph, the number of times we need to call DFS/BFS traversal for the graph on an unvisited vertex gives the number of connected components in the graph.**

MST Notes

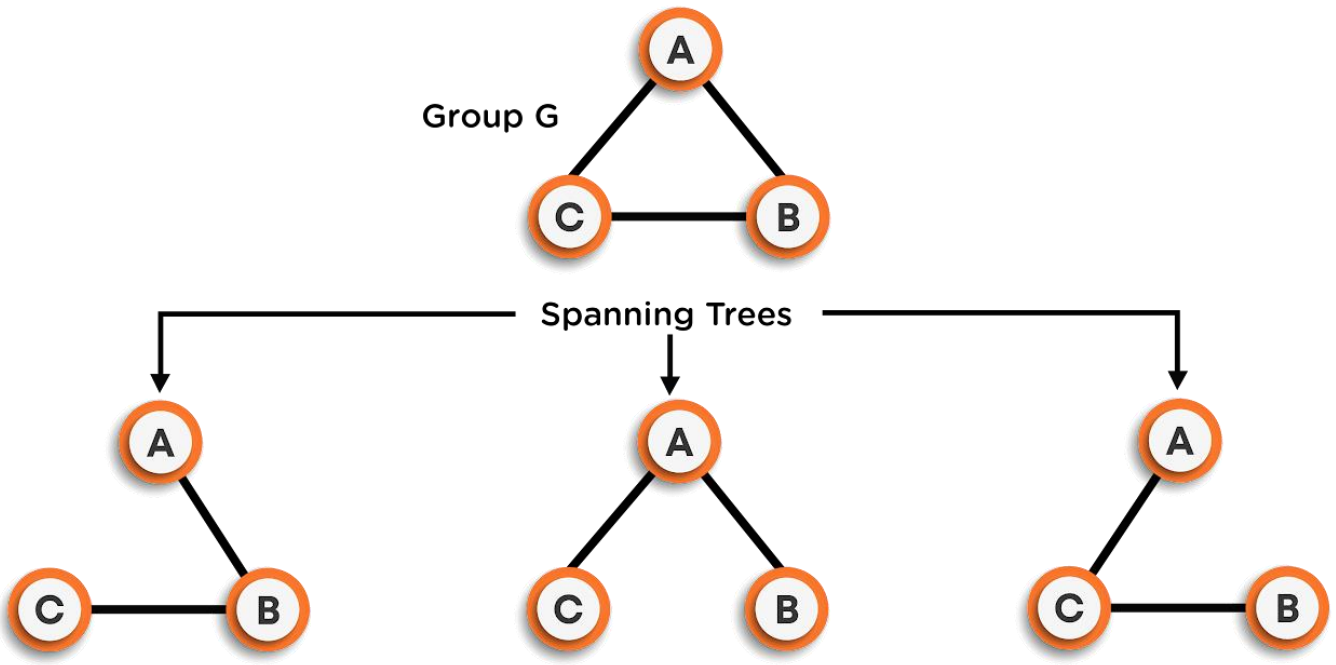
Minimum Spanning Tree(MST)

A tree is a graph, which:

- **Is always connected.**
- **Contains no cycle.**

If we are given an undirected and connected graph, a spanning tree is a tree that contains all the vertices(V) of the graph and |V|-1 edges. For a given graph, we can have multiple spanning trees.

Refer to the example below for a better understanding of spanning trees.



If there are *n* vertices and *e* edges in the graph, then any spanning tree corresponding to that graph contains *n* vertices and *n-1* edges.

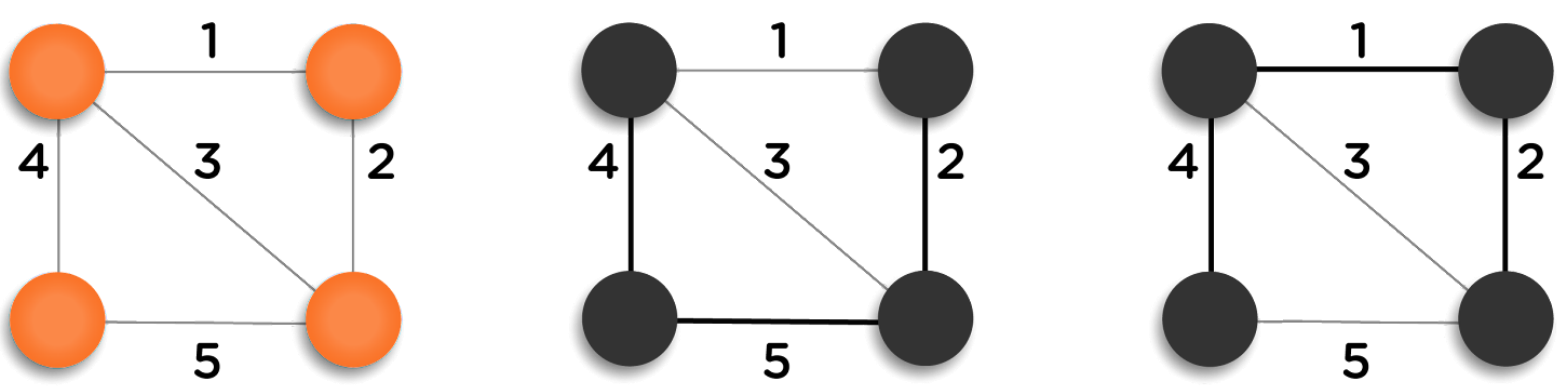
Properties of spanning trees:

- **A connected and undirected graph can have more than one spanning tree.**

- **The spanning tree is free of loops, i.e., it is acyclic.**
- **Removing any one of the edges will make the graph disconnected.**
- **Adding an extra edge to the spanning tree will create a loop in the graph.**

Minimum Spanning Tree(MST)

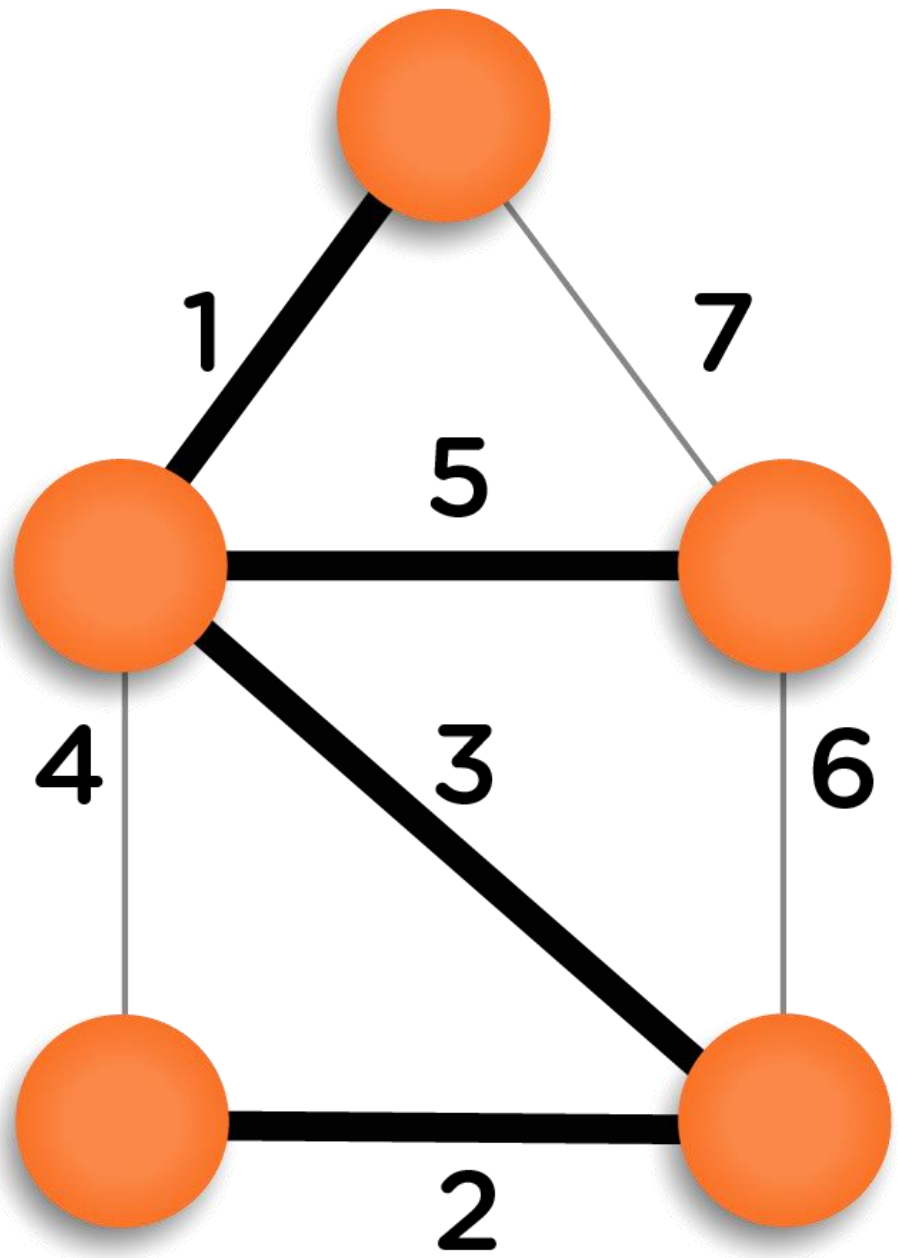
In a weighted graph, the MST is a spanning tree with minimum weight than all other spanning trees of that graph. Refer to the image below for better understanding, where the edges marked in bold represent the edges of the spanning tree

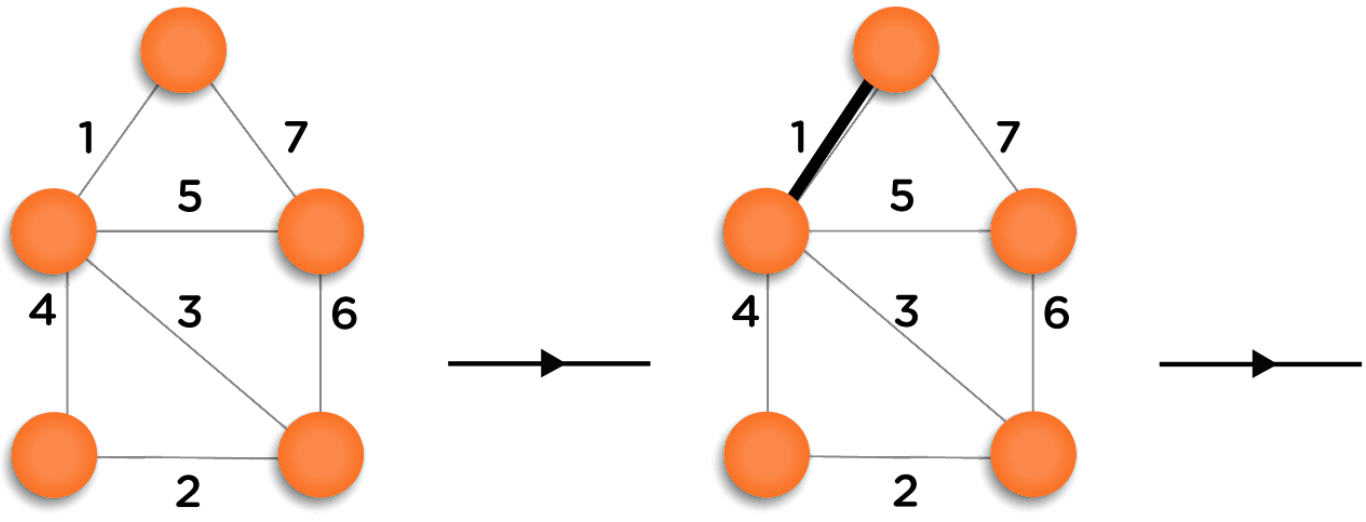


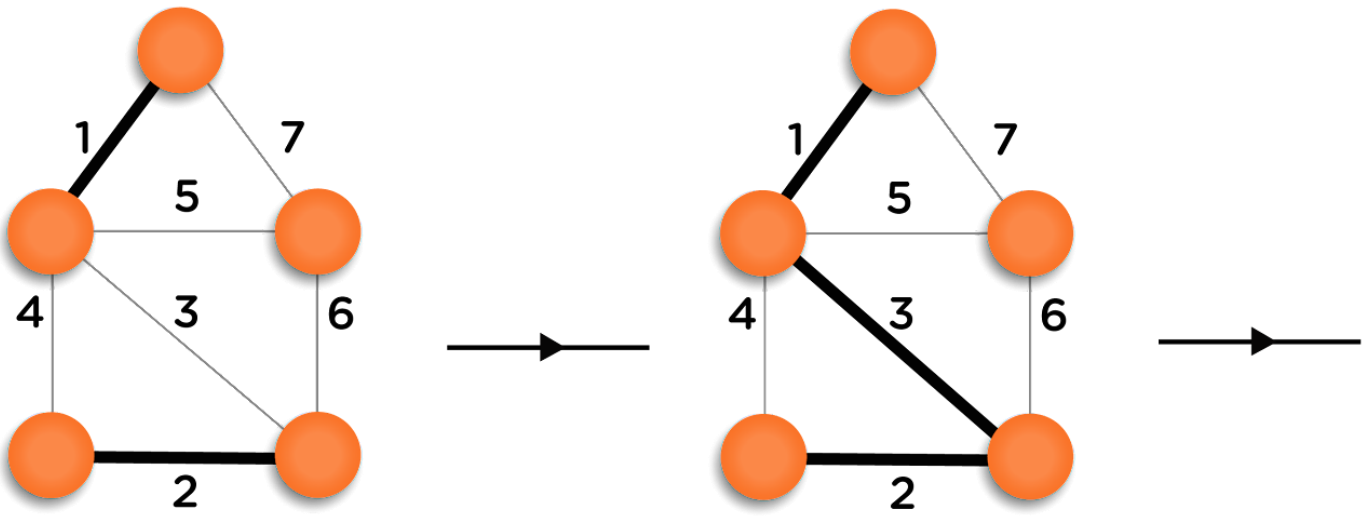
Kruskal's Algorithm:

This algorithm is used to find MST for the given graph. It builds the spanning tree by adding edges one at a time. We start by picking the edge with minimum weight, adding that edge into the MST, and increasing the count of edges in the spanning tree by one. Now, we will be picking the minimum weighted edge by excluding the already chosen ones and correspondingly increasing the count. While choosing the edge, we will also make sure that the graph remains acyclic after including the same. This process will continue until the count of edges in the MST reaches *n-1*. Ultimately, the graph obtained will be MST.

Refer to the example below for a better understanding of the same.


1
7
5
4
3
6
2





This is the final MST obtained using Kruskal's algorithm. It can be checked manually that the final graph is the MST for the given graph.

Cycle Detection

While inserting a new edge in the MST, we have to check if introducing that edge makes the MST cyclic or not. If not, then we can include that edge, otherwise not.

Now, let's figure out a way to detect the cycle in a graph. The following are the possible cases:

- **By including an edge between nodes A and B, if both nodes A and B are not present in the graph, then it is safe to include that edge as including it, will not bring a cycle to the graph.**
- **Out of two vertices, if any one of them has not been visited (or not present in the MST), then that vertex can also be included in the MST.**
- **If both the vertices are already present in the graph, they can introduce a cycle in the MST. It means we can't use this method to detect the presence of the cycle.**

Union-Find Algorithm:

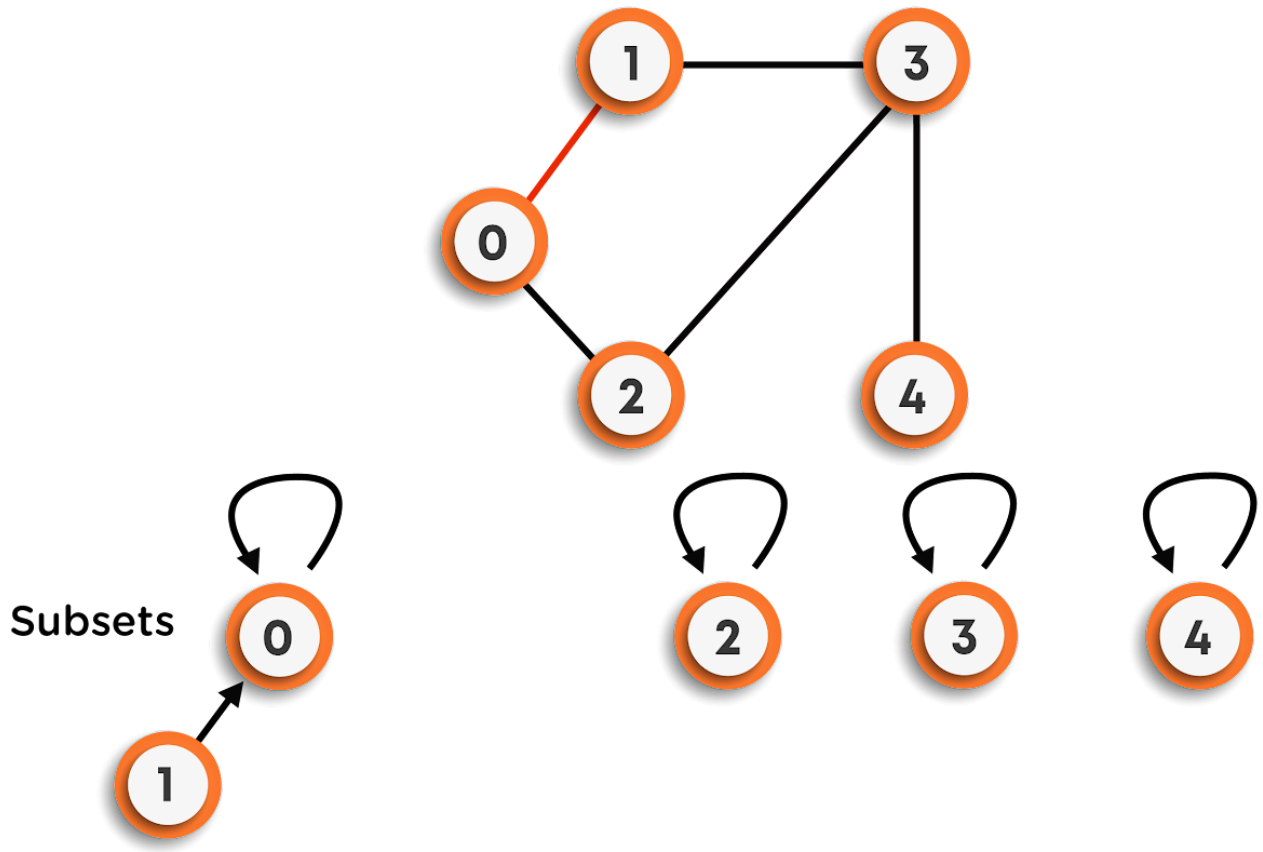
Before adding any edge to the graph, we will check if the two vertices of the edge lie in the same component of MST or not, in other words, we are checking whether the addition of an edge will lead to a cycle or not. If not, then it is safe to add that edge to the MST.

Following the steps of the algorithm:

- **We will assume that initially, the total number of disjoint sets is equal to the number of vertices in the graph starting from 0 to n-1.**
- **We will maintain a parent array specifying the parent vertex of each of the vertex of the graph. Initially, as each vertex belongs to a different disjoint set (connected component), hence each vertex will be its parent.**
- **Now, before inserting any edge into the MST, we will check the parent of the vertices. If their parent vertices are equal, they belong to the same connected component; hence it is unsafe to add that edge.**

- **Otherwise, we can add that edge into the MST, and simultaneously update the parent array so that they belong to the same component(Refer to the code on how to do so).**

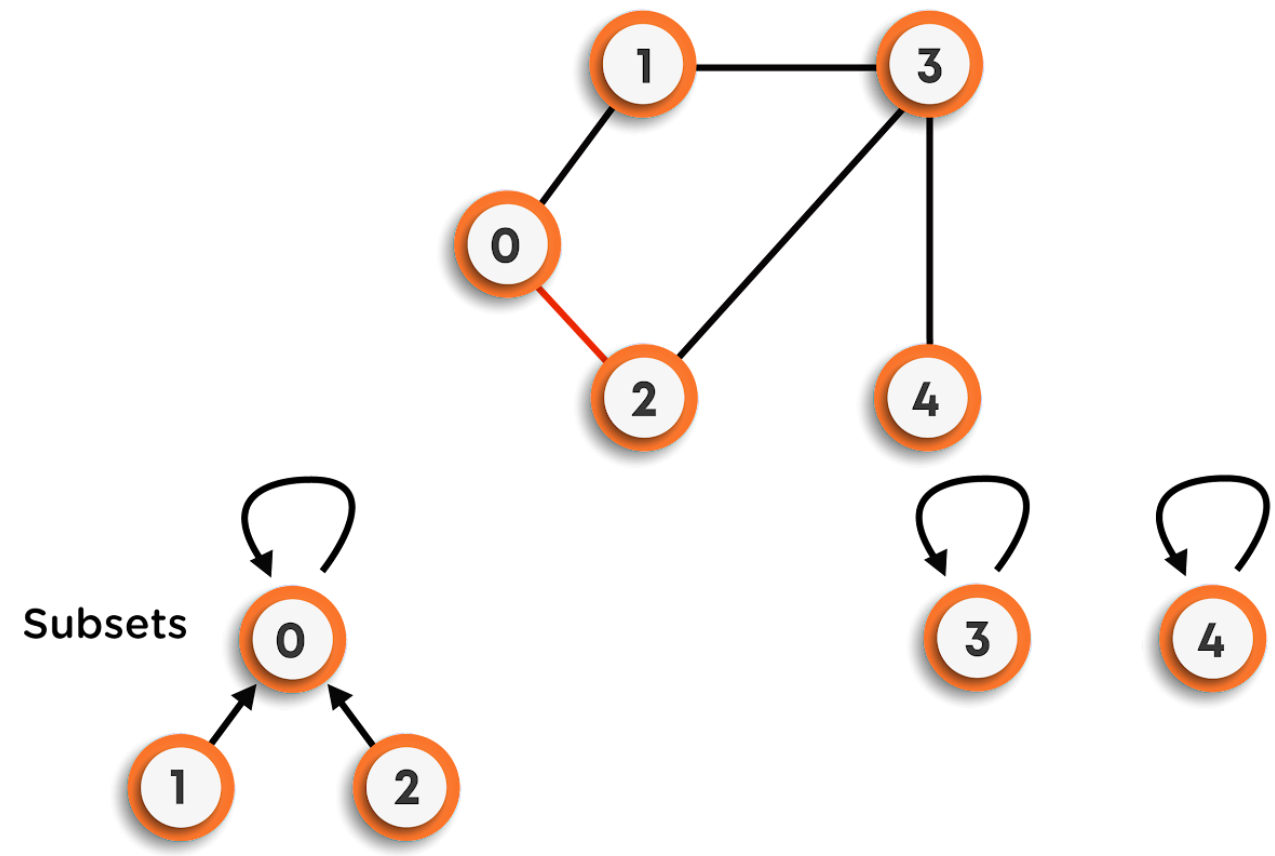
Look at the following example, for better understanding:



Edge 0-1

Find: 0 belongs to subset 0 and 1 belongs to subset 1 so they are in different subsets.

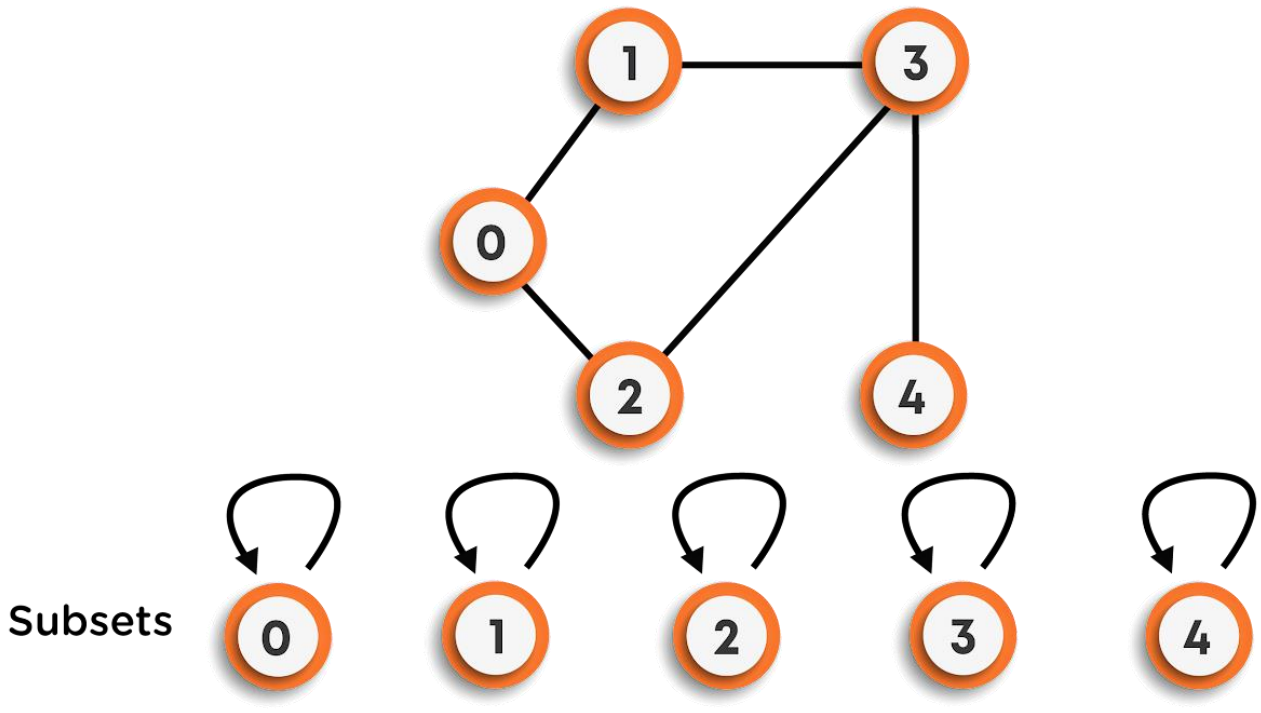
Union: Make 0 as the parent of 1, Updated set is {0,1}. 0 is the set representative since 0 is parent for itself.



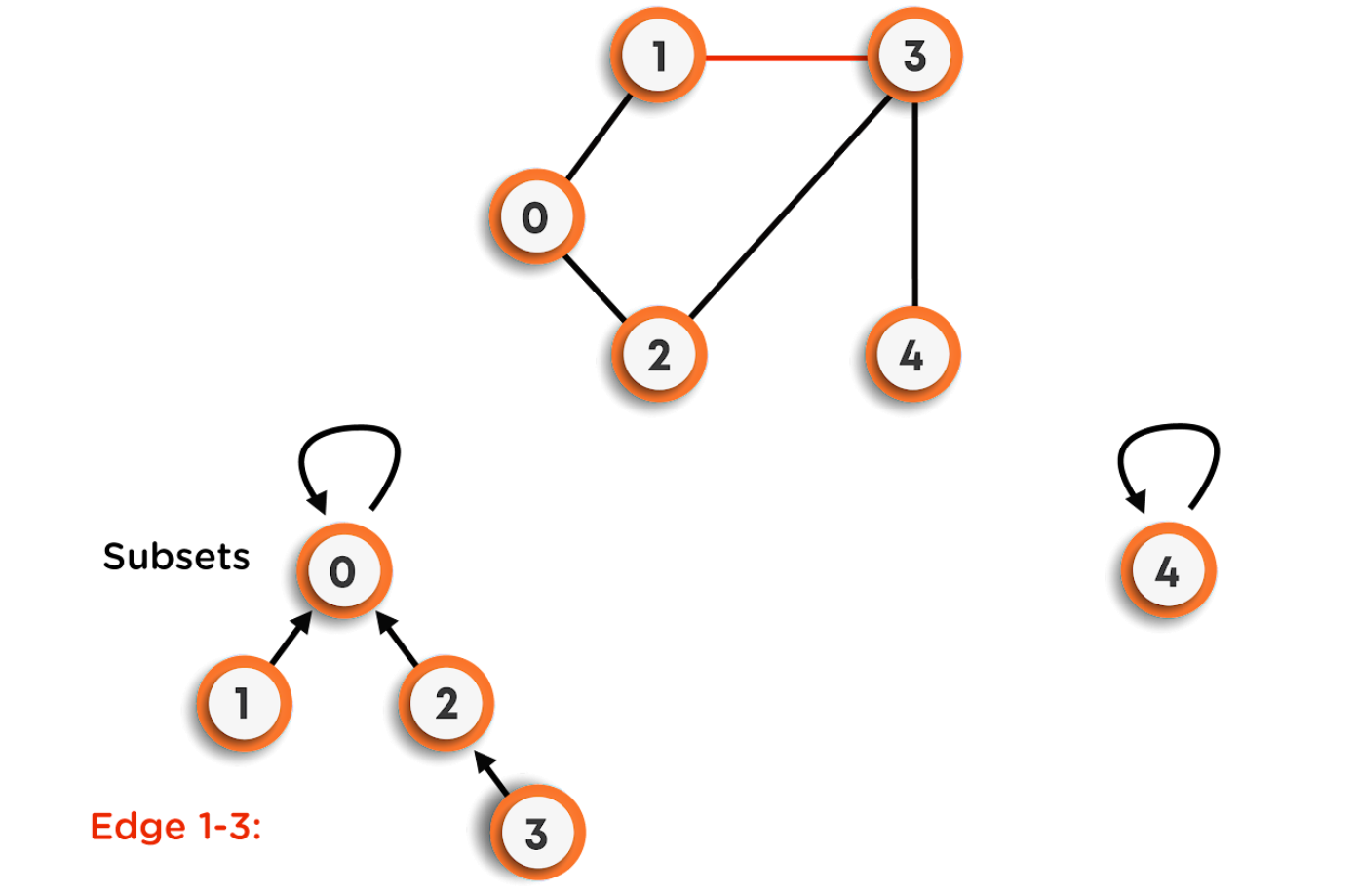
Edge 0-2:

Find: 0 belongs to subset 0 and 2 belongs to subset 2 so they are in different subsets.

Union: Make 0 as the parent of 2, Updated set is {0,1,2}. 0 is the set representative since 0 is parent for itself.

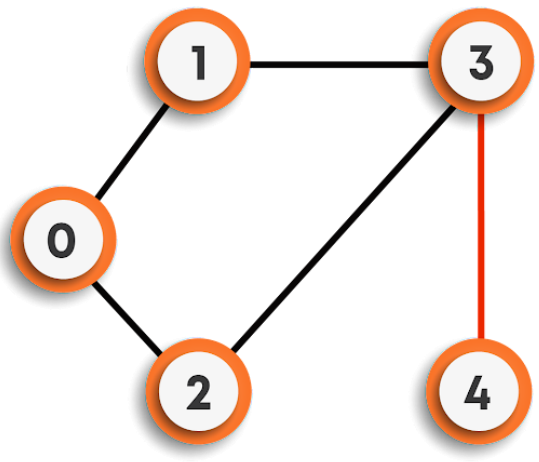


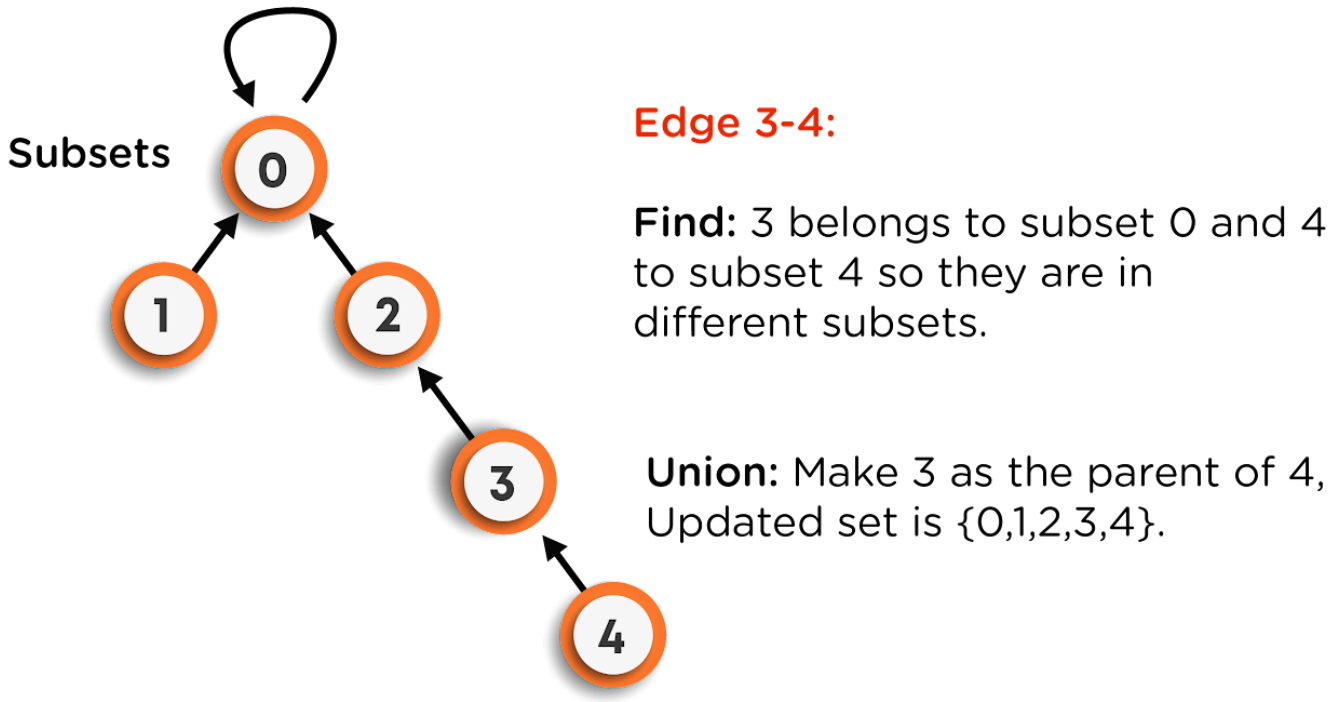
Initially all parent pointers are pointing to self
means only one element in each subset



Find: 1 belongs to subset 0 and 3 belongs to subset 3 so they are in different subsets.

Union: Make 1 as the parent of 3, Updated set is {0,1,2,3}. 0 is the set representative since 0 is parent for itself.





Edge 3-4:

Find: 3 belongs to subset 0 and 4 belongs to subset 4 so they are in different subsets.

Union: Make 3 as the parent of 4, Updated set is {0,1,2,3,4}.

Note: While finding the parent of the vertex, we will be finding the topmost parent (Oldest Ancestor). For example: suppose, the vertex 0 and the vertex 1 were connected, where the parent of 0 is 1, and the parent of 1 is 1. Now, while determining the parent of the vertex 0, we will visit the parent array and check the vertex at index 0. In our case, it is 1. Now we will go to index 1 and check the parent of index 1, which is also 1. Hence, we can't go any further as the index is the parent of itself. This way, we will be determining the parent of any vertex.

The time complexity of the union-find algorithm becomes O(V) for each vertex in the worst case due to skewed-tree formation, where V is the number of vertices in the graph. Here, we can see that time complexity for cycle detection has significantly improved compared to the previous approach.

Time Complexity of Kruskal's Algorithm:

In our code, we have the following three steps: (Here, the total number of vertices is n, and the total number of edges is E)

- **Take input in the array of size E.**
- **Sort the input array based on edge-weight. This step has the time complexity of O(E log(E)).**
- **Pick (n-1) edges and put them in MST one-by-one. Also, before adding the edge to the MST, we checked for cycle detection for each edge. For cycle detection, in the worst-case time complexity of E edges will be O(E.n), as discussed earlier.**

Hence, the total time complexity of Kruskal's algorithm becomes O(E log(E) + n.E). This time complexity is bad and needs to be improved.

We can't reduce the time taken for sorting, but the time taken for cycle detection can be improved using another algorithm named Union by Rank and Size, Path Compression. The basic idea in these algorithms is that we will be avoiding the formation of skewed-tree structure, which reduces the time complexity for each vertex to O(log(E)).

Prim's Algorithm

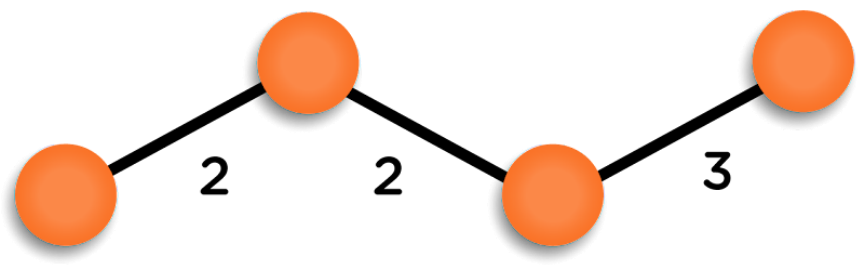
This algorithm is used to find MST for a given undirected-weighted graph (which can also be achieved using Kruskal's Algorithm).

In this algorithm, the MST is built by adding one edge at a time. In the beginning, the spanning tree consists of only one vertex, which is chosen arbitrarily from the set of all vertices of the graph. Then the minimum weighted edge, outgoing from this vertex, is selected and simultaneously inserted into the MST. Now, the tree contains two edges. Further, we will be selecting the edge with the minimum weight such that one end is already present there in the MST and the other one from the unselected set of vertices. This process is repeated until we have inserted a total of (n-1) edges in the MST.

Consider the following example for a better understanding.

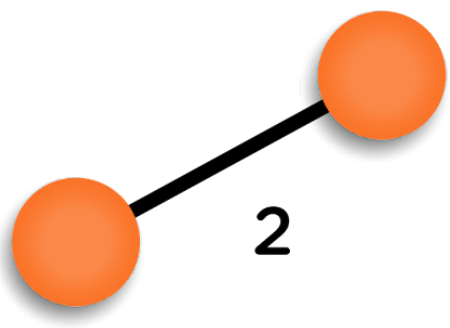
Step: 2

Choose a vertex



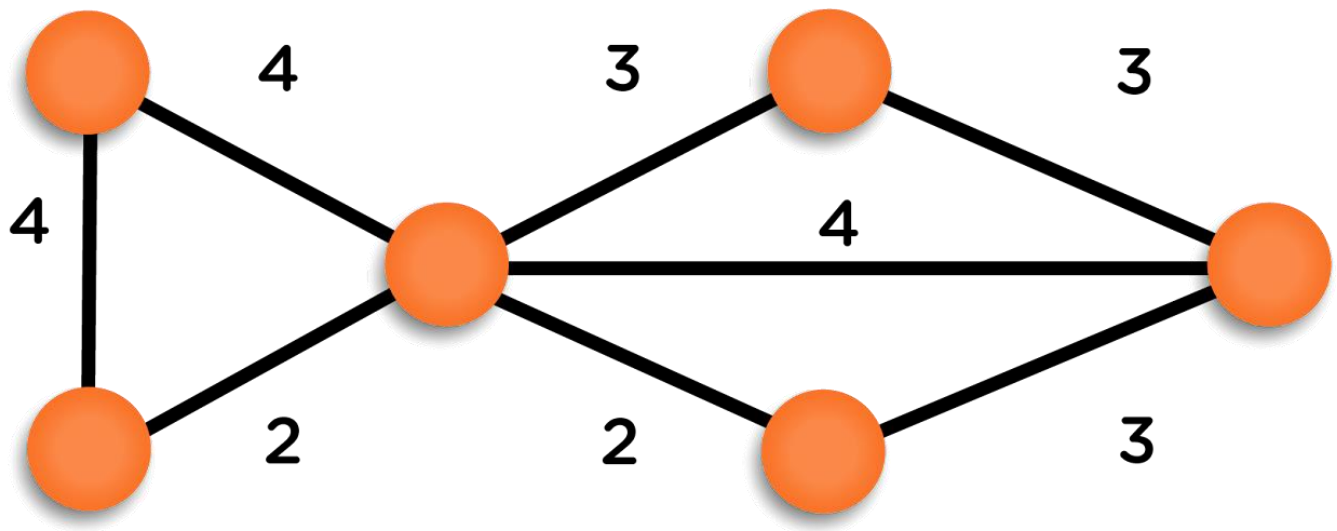
Step: 5

Choose the nearest edge not yet in the solution,
if there are multiple choices, choose one at random



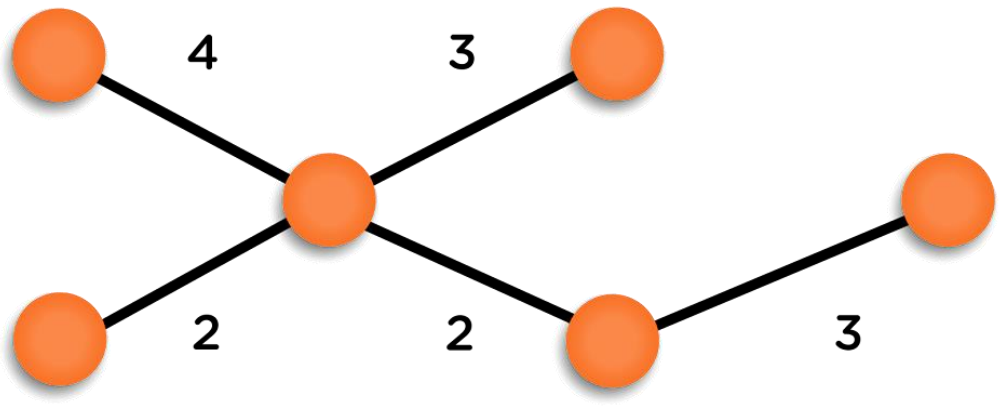
Step: 3

Choose the shortest edge from this vertex add it



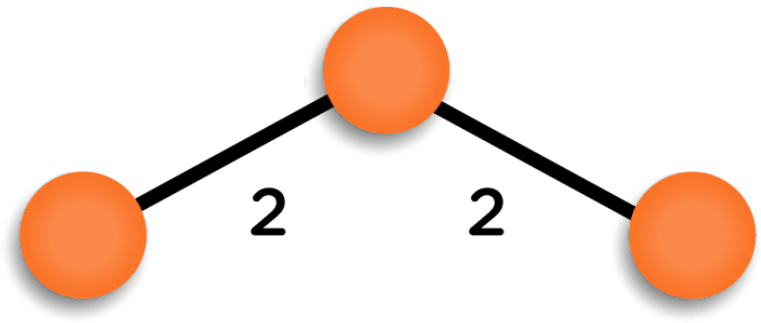
Step: 1

Start with a weighted graph



Step: 6

Repeat until you have a spanning tree



Step: 4

Choose the nearest vertex not yet in the solution

Implementation:

- **We are considering the starting vertex to be 0 with a parent equal to -1, and weight is equal to 0 (The weight of the edge from vertex 0 to vertex 0 itself).**
- **The parent of all other vertices is assumed to be NIL, and the weight will be equal to infinity, which means that the vertex has not been visited yet.**
- **We will mark the vertex 0 as visited and the rest as unvisited. If we add any vertex to the MST, then that vertex will be shifted from the unvisited section to the visited section.**
- **Now, we will update the weights of direct neighbors of vertex 0 with the edge weights as these are smaller than infinity. We will also update the parent of these vertices and assign them 0 as we reached these vertices from vertex 0.**
- **This way, we will keep updating the weights and parents, according to the edge, which has the minimum weight connected to the respective vertex.**

Time Complexity of Prim's Algorithm:

Here, n is the number of vertices, and E is the number of edges.

- **The time complexity for finding the minimum weighted vertex is O(n) for each iteration. So for (n-1) edges, it becomes O(n2).**
- **Similarly, for exploring the neighbor vertices, the time taken is O(n2).**

It means the time complexity of Prim's algorithm is O(n2). We can improve this in the following ways:

- **For exploring neighbors, we are required to visit every vertex because of the adjacency matrix. We can improve this by using an adjacency list instead of a matrix.**
- **Now, the second important thing is the time taken to find the minimum weight vertex, which is also taking a time of O(n2). Here, out of the available list, we are trying to figure out the one with minimum weight. This can be optimally achieved using a priority queue where the priority will be taken as weights of the vertices. This will take O(log(n)) time complexity to remove a vertex from the priority queue.**

These optimizations can lead us to the time complexity of O((n+E)log(n)), which is much better than the earlier one. Try to write the optimized code by yourself.

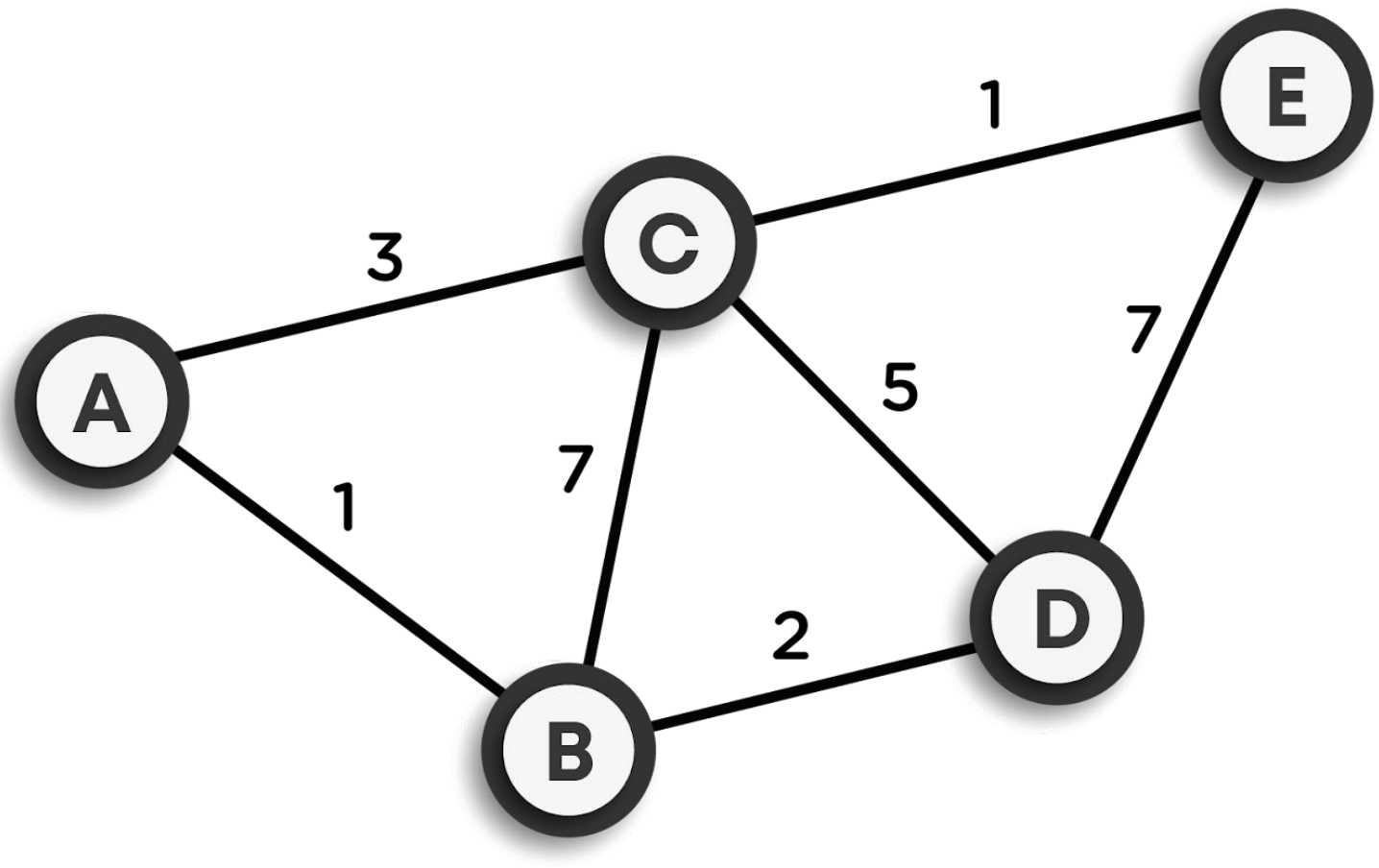
Shortest Path Algorithms Notes

Dijkstra’s Algorithm: Minimum cost path

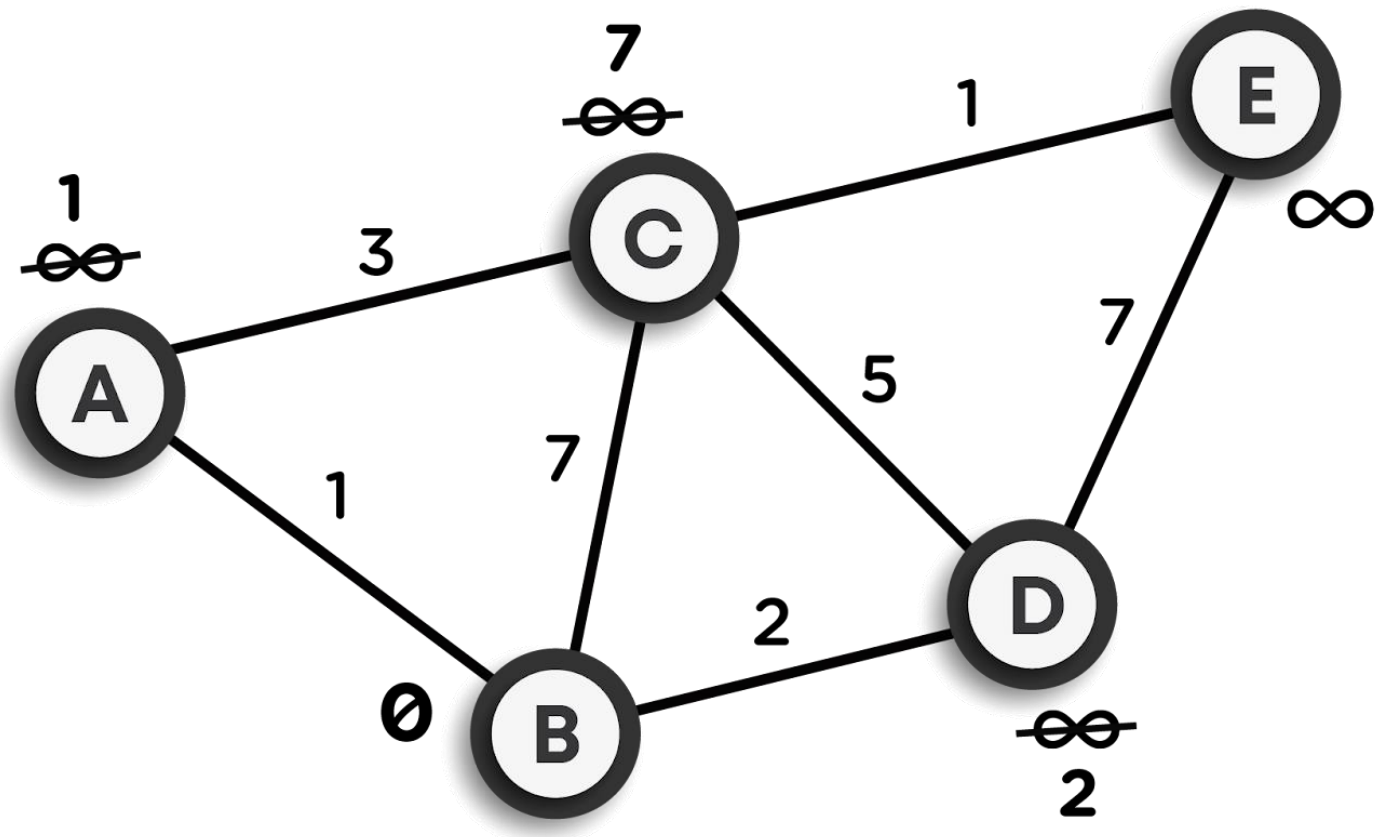
This algorithm is used to find the shortest distance between any two vertices in a weighted non-cyclic graph.
Here, we will be using a slight modification of the algorithm according to which we will be figuring out the minimum distance of all the vertices from the particular source vertex.

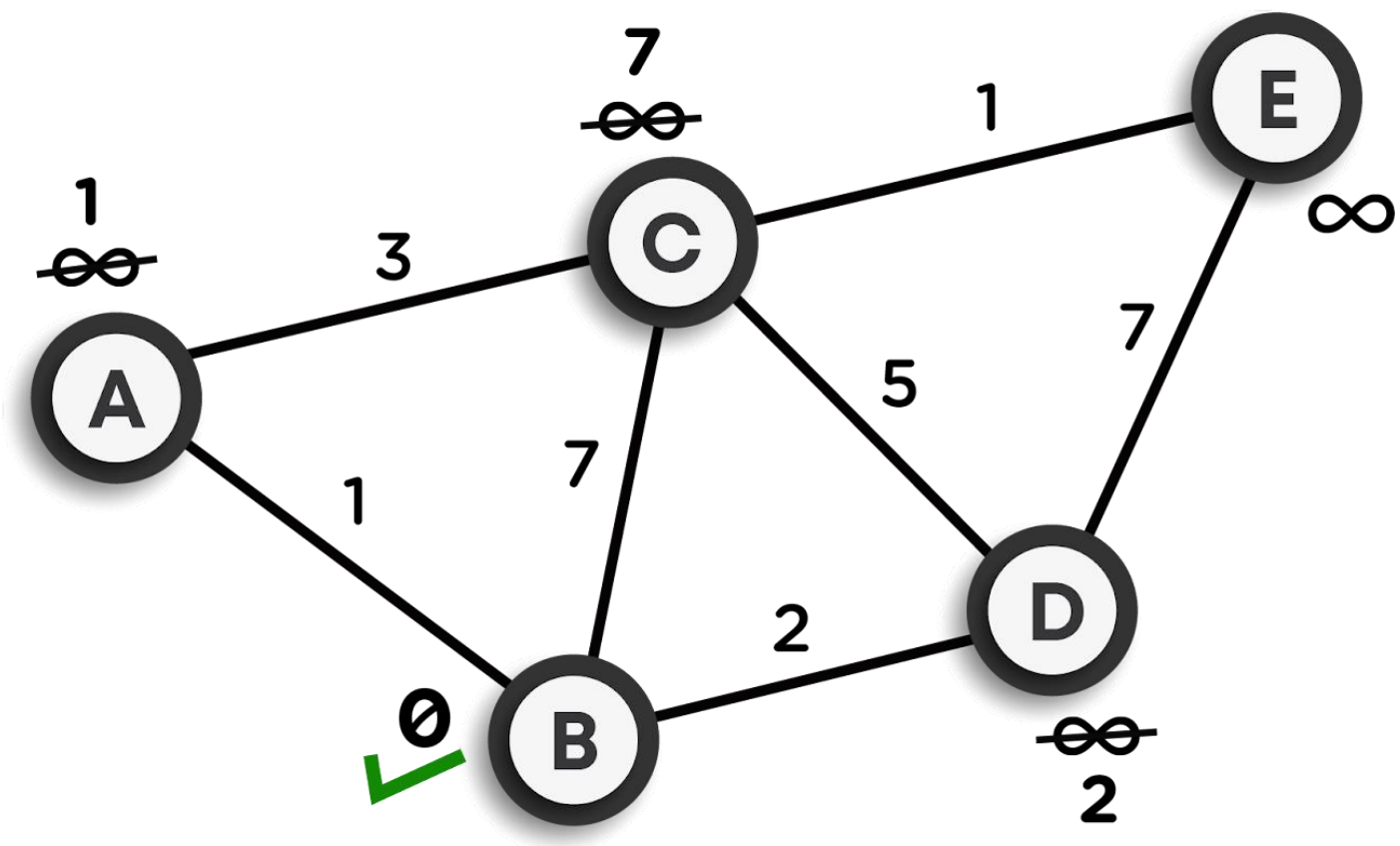
Let’s consider the algorithm with an example:

- **We want to calculate the shortest path between the source vertex C and all other vertices in the following graph.**

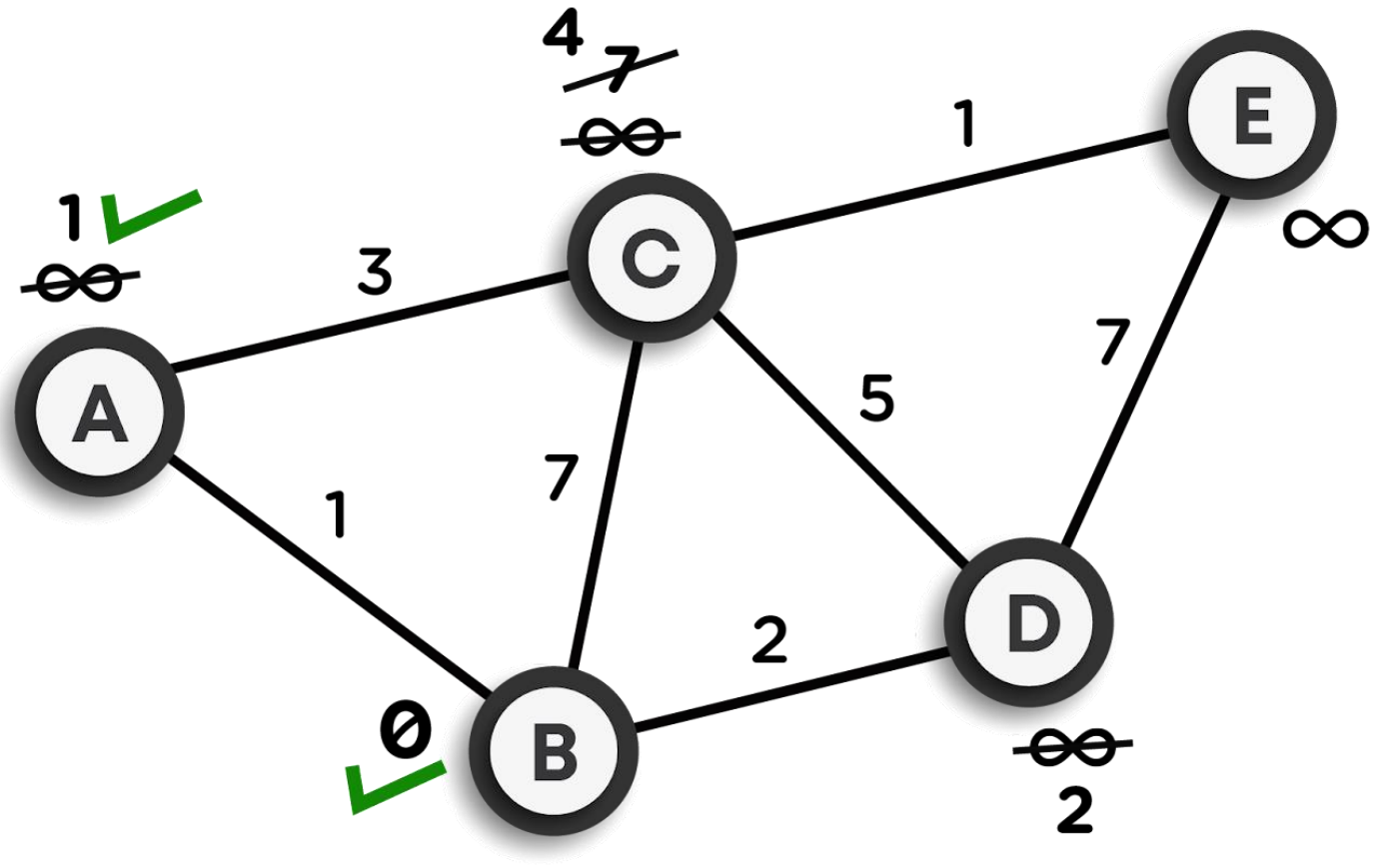


- **While executing the algorithm, we will mark every node with its minimum distance to the selected node, which is C in our case. Obviously, for node C itself, this distance will be 0, and for the rest of the nodes, we will assume that the distance is infinity, which also denotes that these vertices have not been visited till now.**
- **Now, we will check for the neighbors of the current node, which are A, B, and D. Now, we will add the minimum cost of the current node to the weight of the edge connecting the current node and the particular neighbor node. For example, for node B, its weight will become minimum(infinity, 0+7) = 7. This same process is repeated for other neighbor nodes.**

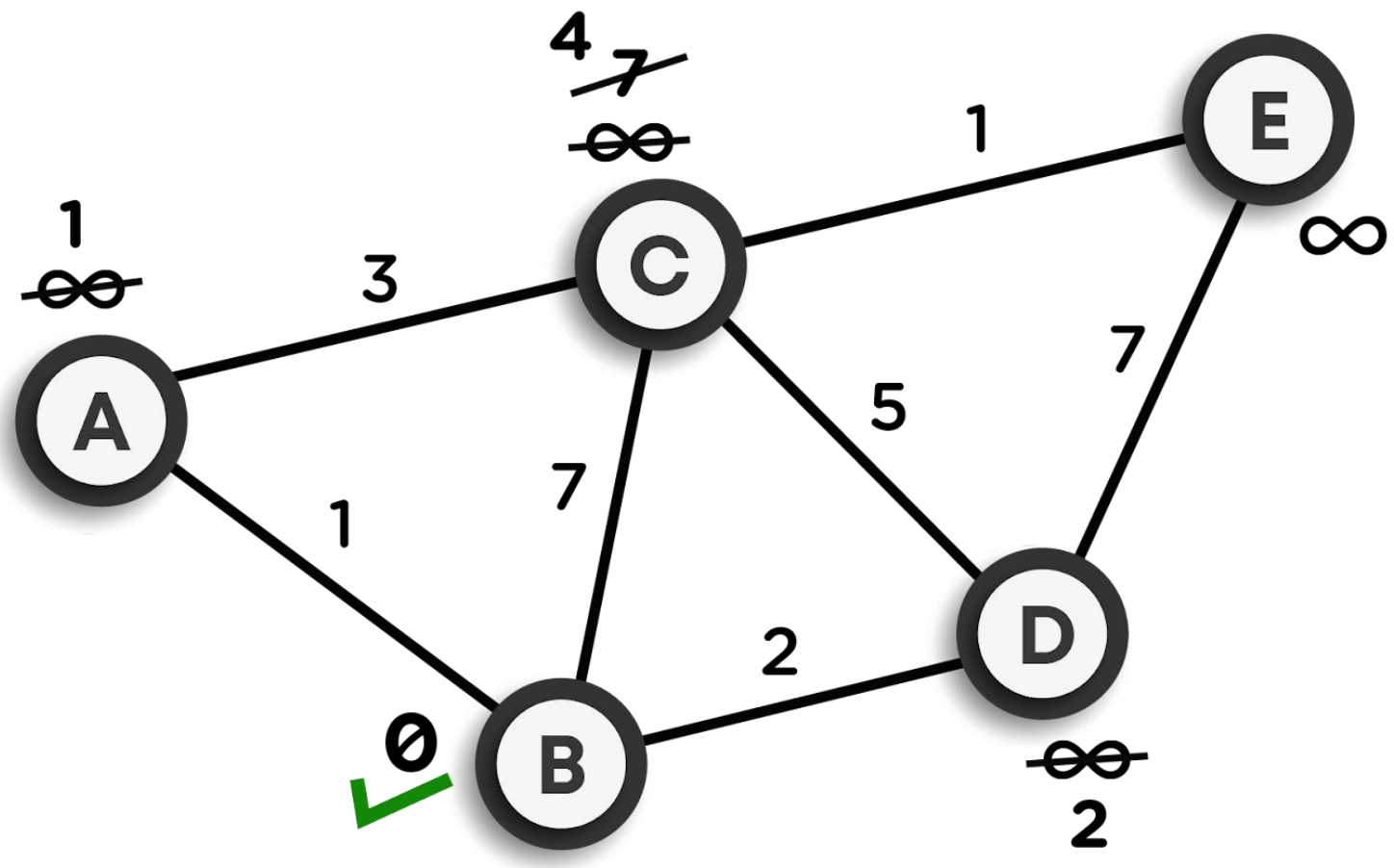


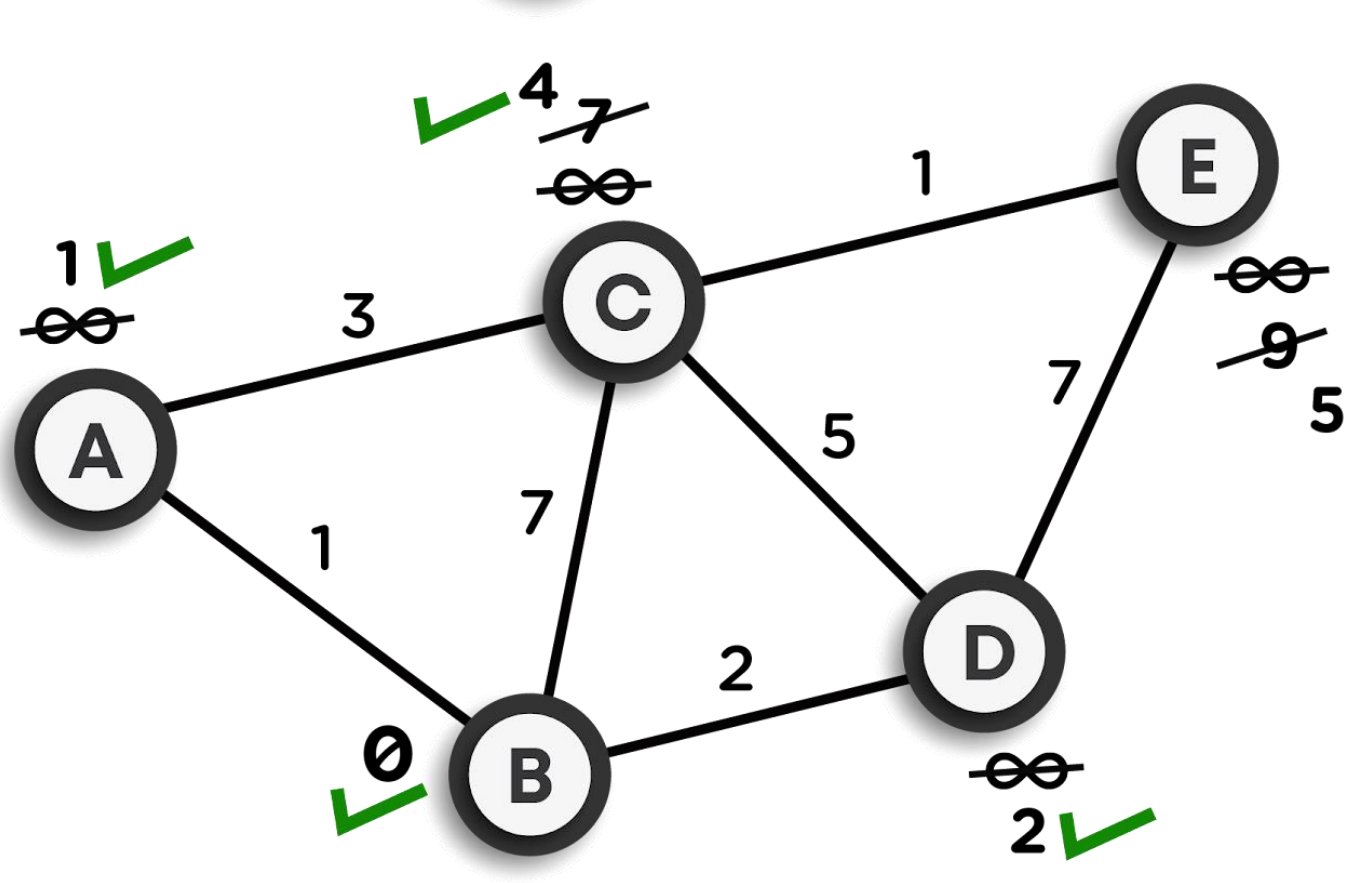


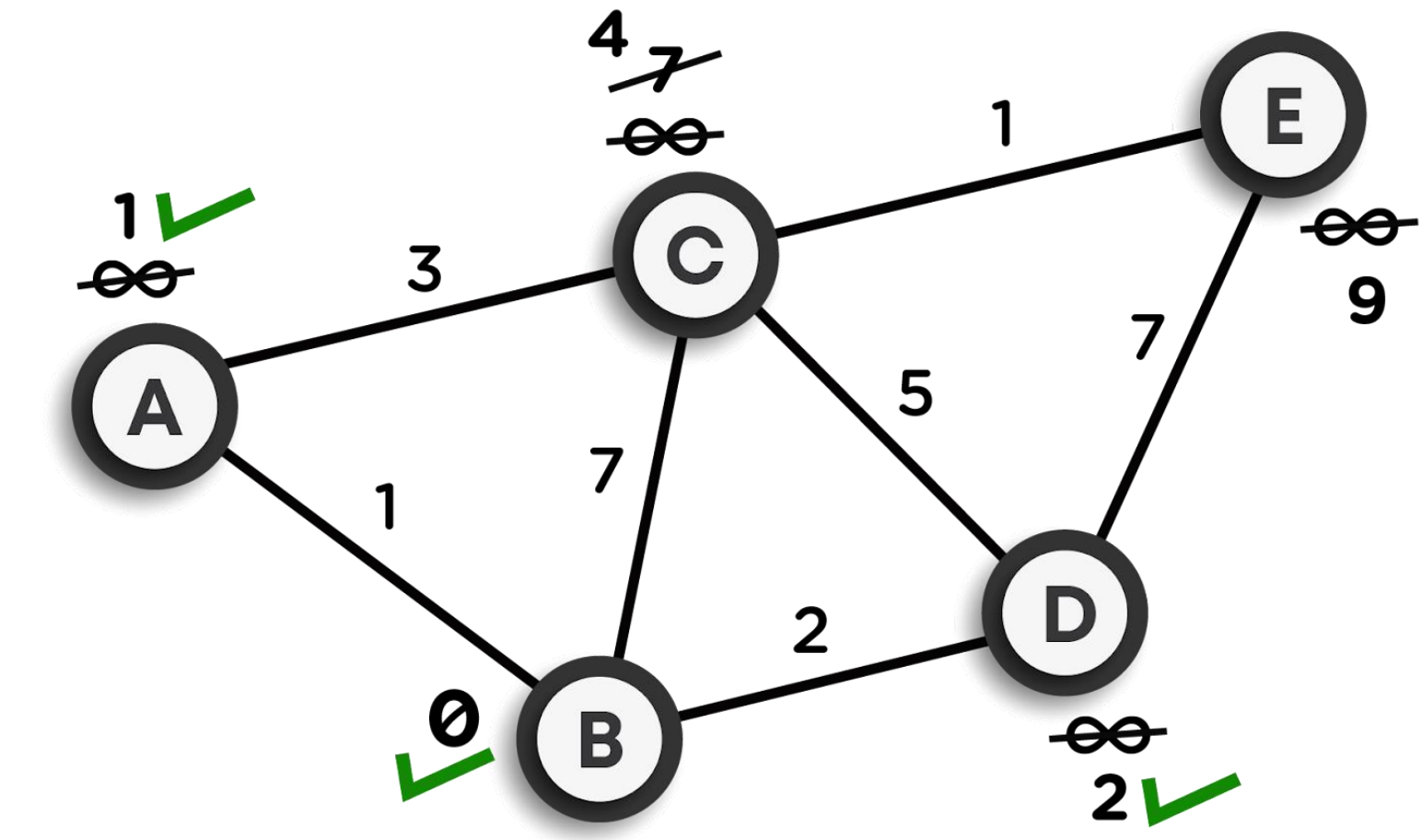
- **Now, as we have updated the distance of all the neighbor nodes of the current node, we will mark the current node as visited.**

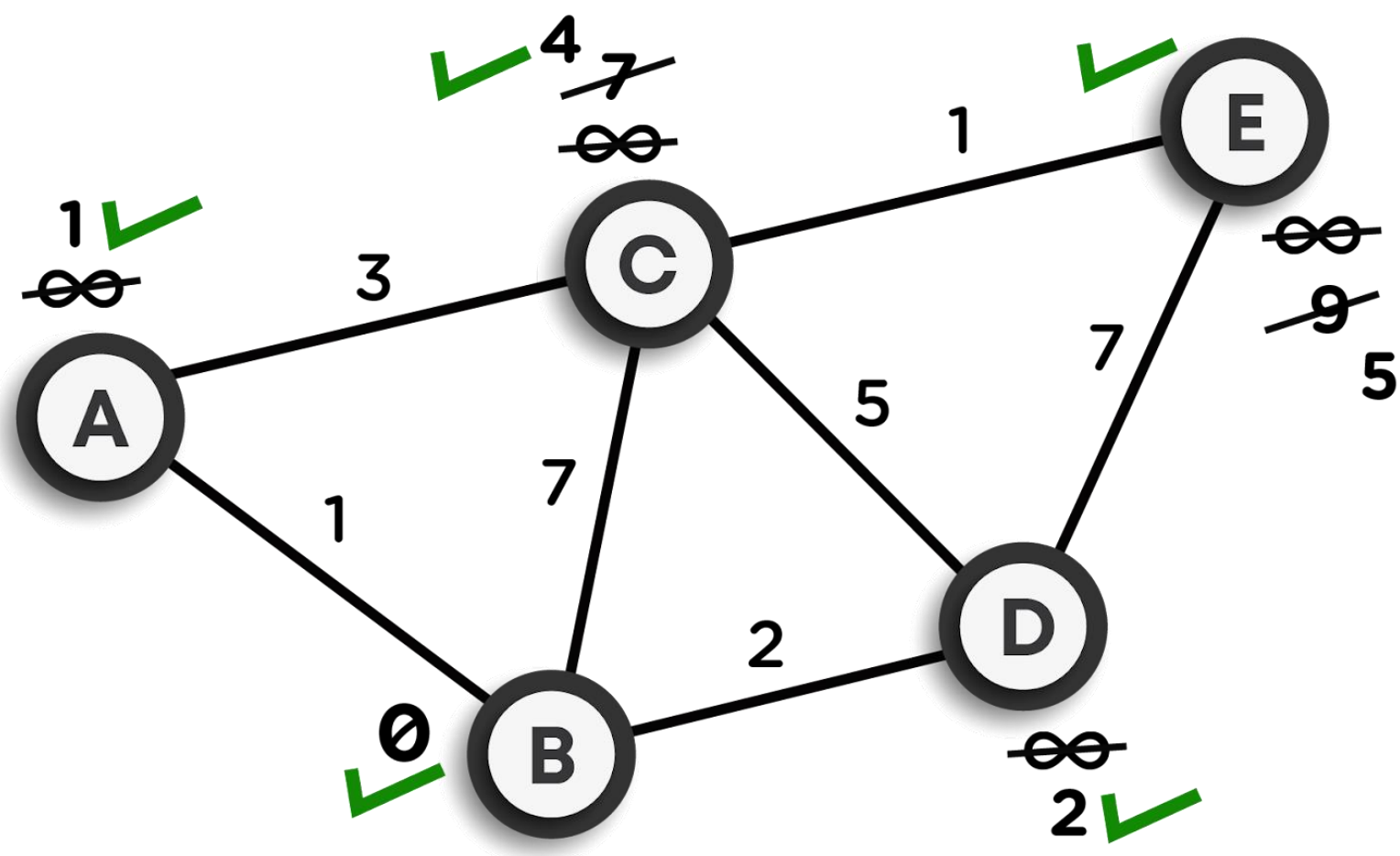


- **After this, we will be selecting the minimum weighted node among the remaining vertices. In this case, it is node A. Take this node as the current node.**
- **Now, we will repeat the above steps for the rest of the vertices. The pictorial representation of the same is shown below:**

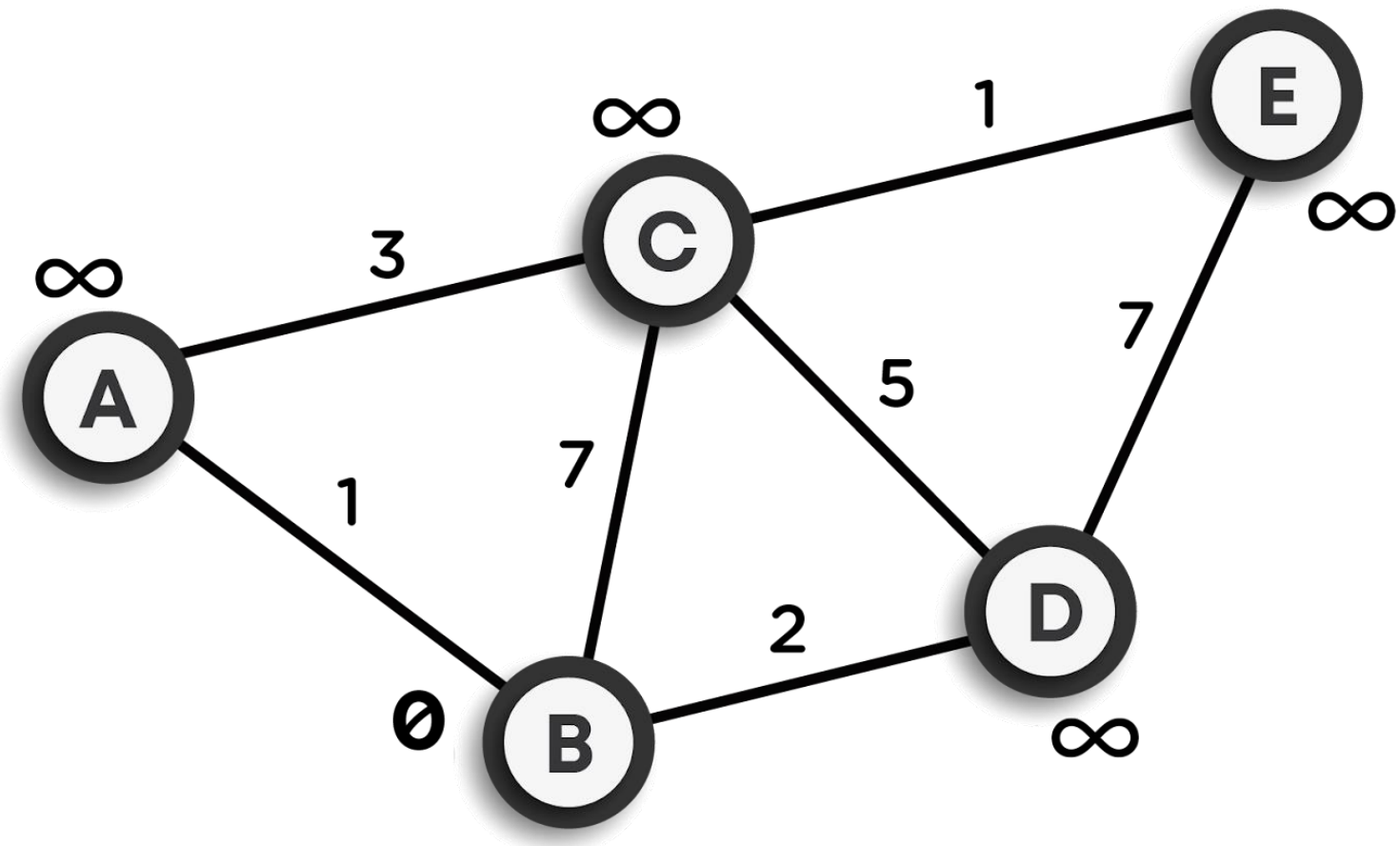

4
7
1
E
1
∞
C
3
A
7
5
7
1
B
2
D
0
2


4
7
1
E
1
C
3
9
5
A
7
5
1
B
2
D
0
2





- **Finally, we will get the graph as follows:**
- **The distances finally marked at each node are minimum from node C.**



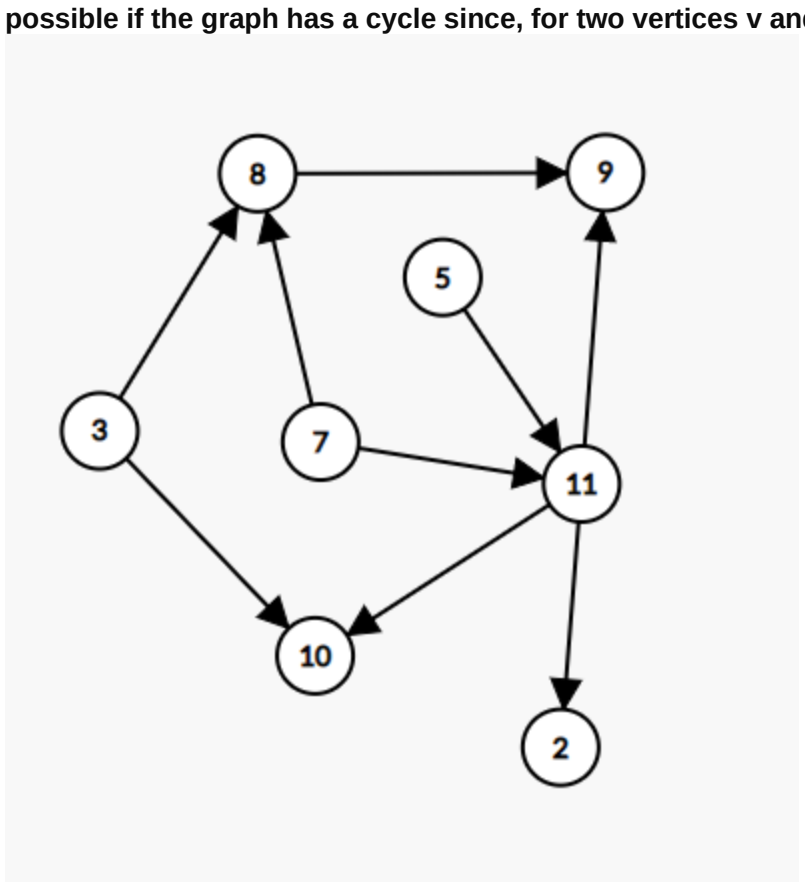
Time Complexity of Dijkstra’s algorithm:

The time complexity is also the same as that of Prim’s algorithm, i.e., O(n2). This can be reduced by using the same approaches as discussed in Prim’s algorithm’s content.

Topological Sort

Topological Sort

Topological sort is an ordering of vertices in a directed acyclic graph in which each node comes before all the nodes to which it has outgoing edges. As an example consider the course prerequisites structure at universities. A directed edge(v, w) indicates that course v must be completed before course w. Topological ordering for this example is the sequence that does not violate the prerequisite requirement. Every directed acyclic graph must have one or more topological ordering. Topological Sort is not possible if the graph has a cycle since, for two vertices v and w in the cycle, v precedes w and w precedes v.



The topological sorts for above graph are: 7, 5, 3, 11, 8, 2, 9, 10

3, 5, 7, 8, 11, 2, 9, 10

Algorithm (Kahn’s algorithm):

1. **Indegree is computed for all vertices, starting with the vertex having indegree 0.**
2. **To keep a track of vertices with in-degree 0 we can use a queue.**

3. All vertices of in-degree 0 are placed in a queue.
4. While the queue is not empty, a vertex v is removed and all vertices adjacent to v have their indegrees decremented.
5. A vertex is put on the queue as soon as its indegree falls to 0.
6. The topological ordering is the order in which the vertices dequeue from the queue.

Time Complexity: O(|E| + |V|)

Application of Topological Sort:

- Representing course prerequisites
- In detecting deadlocks
- The pipeline of computing jobs.
- Checking for symbolics link loop
- Evaluating formulae in a spreadsheet

Dynamic Programming Notes

Dynamic Programming and Memoization

What is Dynamic Programming?

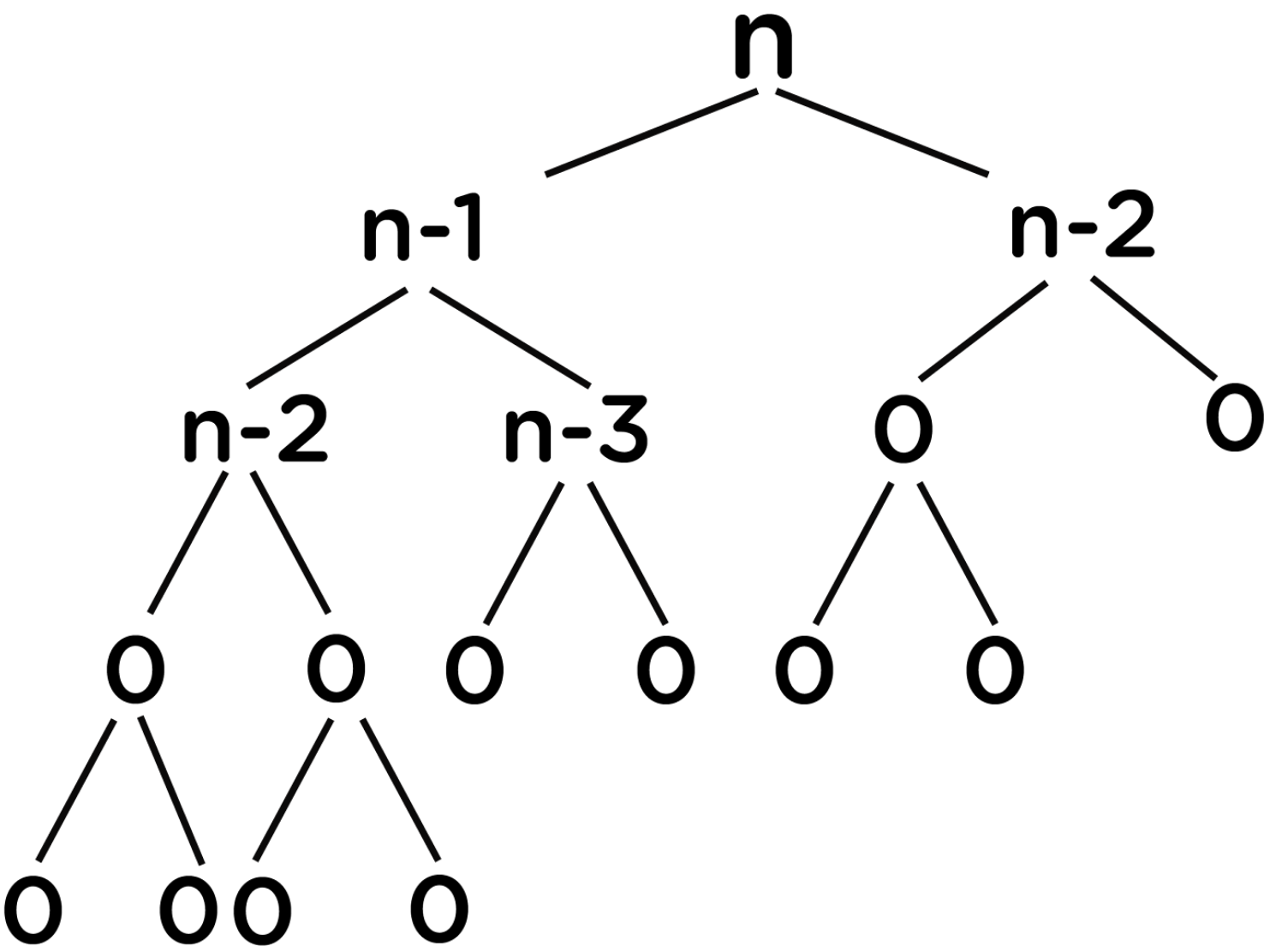
Dynamic Programming is a programming paradigm, where the idea is to memoize the already computed values instead of calculating them again and again in the recursive calls. This optimization can reduce our running time significantly and sometimes reduce it from exponential-time complexities to polynomial time.

Introduction

We know that Fibonacci numbers are defined as follows
fibo(n) = n for n <= 1
fibo(n) = fibo (n - 1) + fibo (n - 2) otherwise
Suppose we need to find the nth Fibonacci number using recursion that we have already found out in our previous sections.

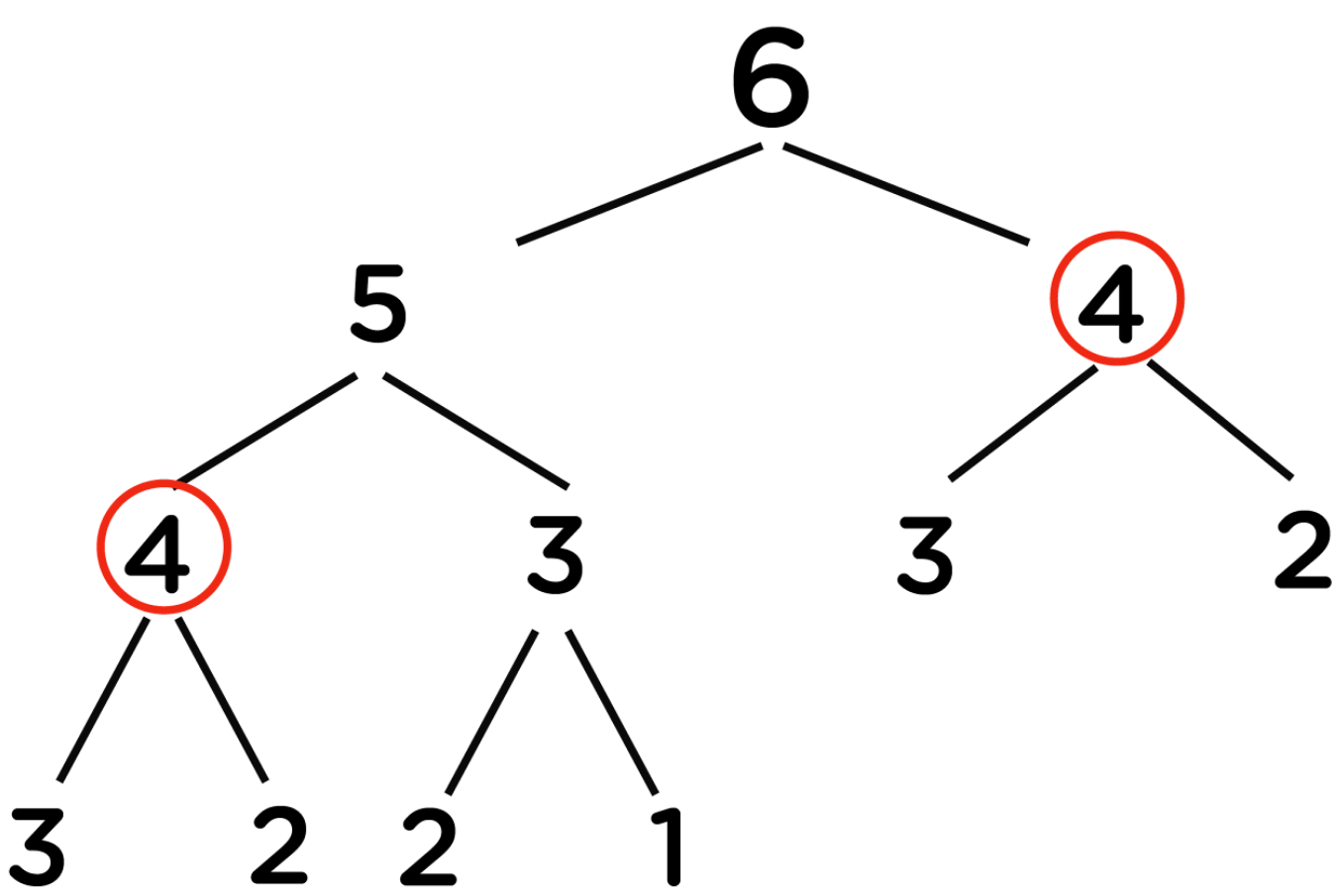
```
function fibo(n):
        if(n <= 1)
                return n

        return fibo(n-1) + fibo(n-2)
```

- Here, for every n, we need to make a recursive call to f(n-1) and f(n-2).
- For f(n-1), we will again make the recursive call to f(n-2) and f(n-3).
- Similarly, for f(n-2), recursive calls are made on f(n-3) and f(n-4) until we reach the base case.
- The recursive call diagram will look something like shown below:



- **At every recursive call, we are doing constant work(k) (addition of previous outputs to obtain the current one).**
- **At every level, we are doing (2^n) * K work (where n = 0, 1, 2, …).**
- **Since reaching 1 from n will take n calls, therefore, at the last level, we are doing (2^(n-1)) * k work.**

- **Total work can be calculated as:(2^0 + 2^1 + 2^2 + ... + 2^(n-1)) * k ≃ (2^n) * k**
- **Hence, it means time complexity will be O(2^n).**
- **We need to improve this complexity. Let's look at the example below for finding the 6th Fibonacci number.**



Important Observations

- **We can observe that there are repeating recursive calls made over the entire program.**

- **As in the above figure, for calculating f(5), we need the value of f(4) (first recursive call over f(4)), and for calculating f(6), we again need the value of f(4) (second similar recursive call over f(4)).**
- **Generally, while recursing, there are repeated recursion calls, which increases the time complexity of the program.**

To overcome this problem, we will store the output of previously encountered values (preferably in arrays as these are most efficient to traverse and extract data). Next time whenever we will be making the recursive calls over these values, we will directly consider their already stored outputs and then use these in our calculations instead of calculating them over again.
This way, we can improve the running time of our code.

Memoization

This process of storing each recursive call's output and then using them for further calculations preventing the code from calculating these again is called Memoization.

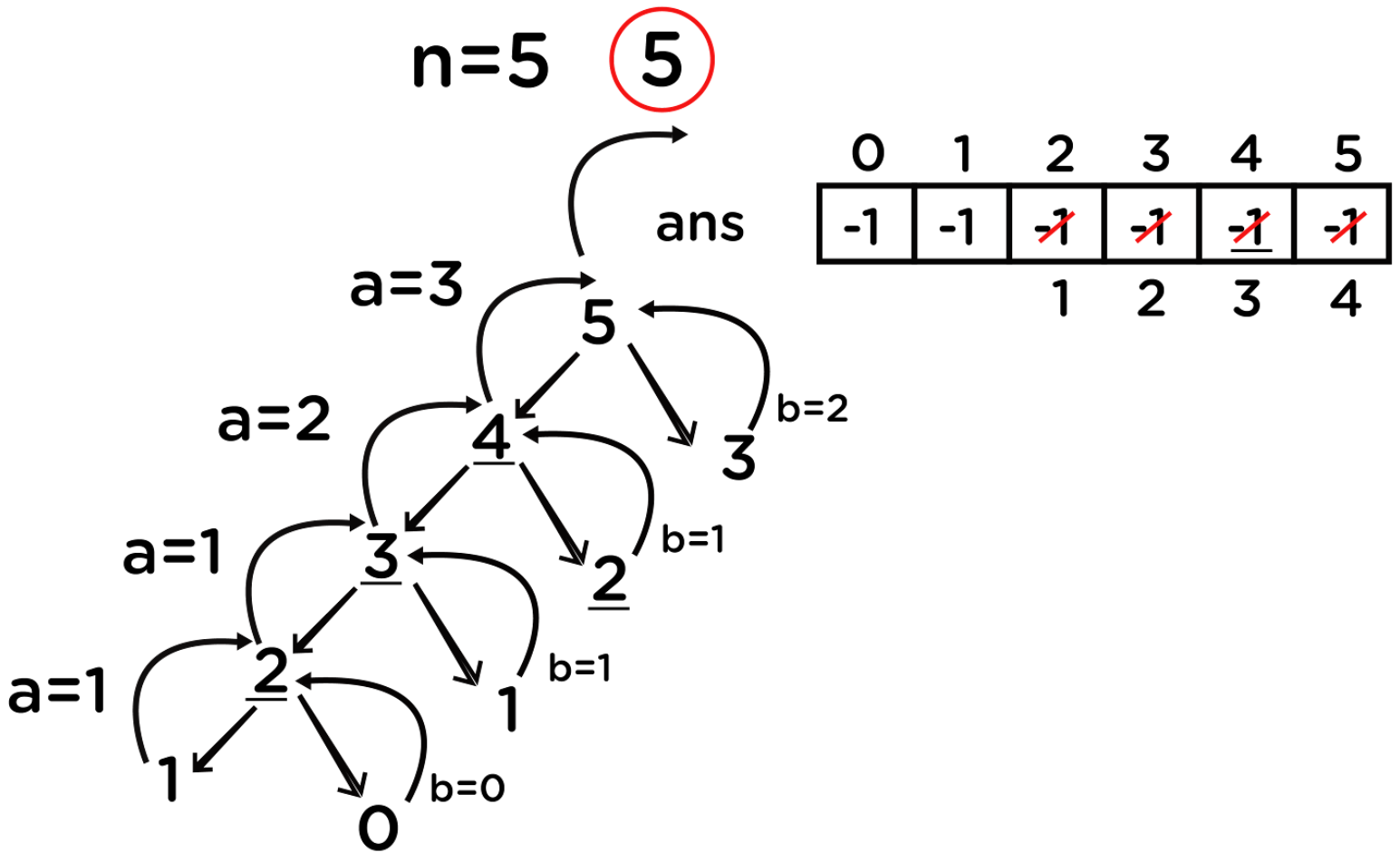
- **Notice that or answer cannot be -1. Hence if we encounter any value equal to it, we can know that this value is yet to be completed. We could have used any other value as well that cannot be a possible answer. Let us take -1 as of now.**
- **To achieve this in our example we will simply take an answer array, initialized to -1.**
- **Now while making a recursive call, we will first check if the value stored in this answer array corresponding to that position is -1 or not.**
- **If it is -1, it means we haven't calculated the value yet and need to proceed further by making recursive calls for the respective value.**
- **After obtaining the output, we need to store this in the answer array so that next time, if the same value is encountered, it can be directly used from this answer array.**

Now in this process of memoization, considering the above Fibonacci numbers example, it can be observed that the total number of unique calls will be at most (n+1) only.
Let's look at the memoization code for Fibonacci numbers below:

Pseudocode:

```
function fibo(n, dp)
        //   base case
        if n equals 0 or n equals 1
                return n

        //   checking if has been already calculated
        if dp[n] not  equals -1
                return dp[n]

        //   final ans
        myAns = fibo(n - 1) + fibo(n - 2)
        dp[n] = myAns

        return myAns
```

Let's dry run for n = 5, to get a better understanding:



Again, if we observe carefully, we can see that for any number, we are not able to make a recursive call on the right side of it. This means that we can make at most 5+1 = 6 (n+1) unique recursive calls which reduce the time complexity to O(n) which is highly optimized as compared to simple recursion.

Top-down approach

- **Memoization is a top-down approach, where we save the answers to recursive calls so that they can be used to calculate future answers when the same recursive calls are to be evaluated and improve the time complexity to a greater extent.**
- **Finally, what we are doing is making a recursive call to every index of the answer array and calculating the value for it using previous outputs stored.**
- **Recursive calls terminate over the base case, which means we are already aware of the answers that should be stored in the base case's indexes.**
- **In cases of Fibonacci numbers, these indexes are 0 and 1 as f(0) = 0 and f(1) = 1. So we can directly allot these two values to our answer array and then use these to calculate f(2), which is f(1) + f(0), and so on for every other index.**
- **This can be simply done iteratively by running a loop from i = (2 to n).**
- **Finally, we will get our answer at the 5th index of the answer array as we already know that the i-th index contains the answer to the i-th value.**

Bottom-up approach

We are first trying to figure out the dependency of the current value on the previous values and then using them to calculate our new value. Now, we are looking for those values which do not depend on other values, which means they are independent (the base case's values as these are the smallest problems about which we are already aware of). Finally, we will follow a bottom-up approach to reach the desired index.
Let us now look at the DP code for calculating the nth Fibonacci number:

Pseudocode:

```
function fibonacci(n):

        f = array[n+1]
        //   base case
        f[0] = 0
        f[1] = 1

        for i from 2 to n:
                //   calculating the f[i] based on the last two values
                f[i] = f[i-1] + f[i-2]

        return f[n]
```

General Steps

- **Figure out the most straightforward approach for solving a problem using recursion.**

- **Now, try to optimize the recursive approach by storing the previous answers using memoization.**
- **Finally, replace recursion by iteration using dynamic programming. (It is preferred to be done in this manner because recursion generally has an increased space complexity as compared to iteration methods.)**

Min Cost Path

Problem Statement: Given an integer matrix of size m*n, you need to find out the value of minimum cost to reach from the cell (0, 0) to (m-1, n-1). From a cell (i, j), you can move in three directions : (i+1, j), (i, j+1) and (i+1, j+1). The cost of a path is defined as the sum of values of each cell through which the path passes.
For example, The given input is as follows-

3 4
3 4 1 2
2 1 8 9
4 7 8 1

The path that should be followed is 3 -> 1 -> 8 -> 1. Hence the output is 13.

Approach:

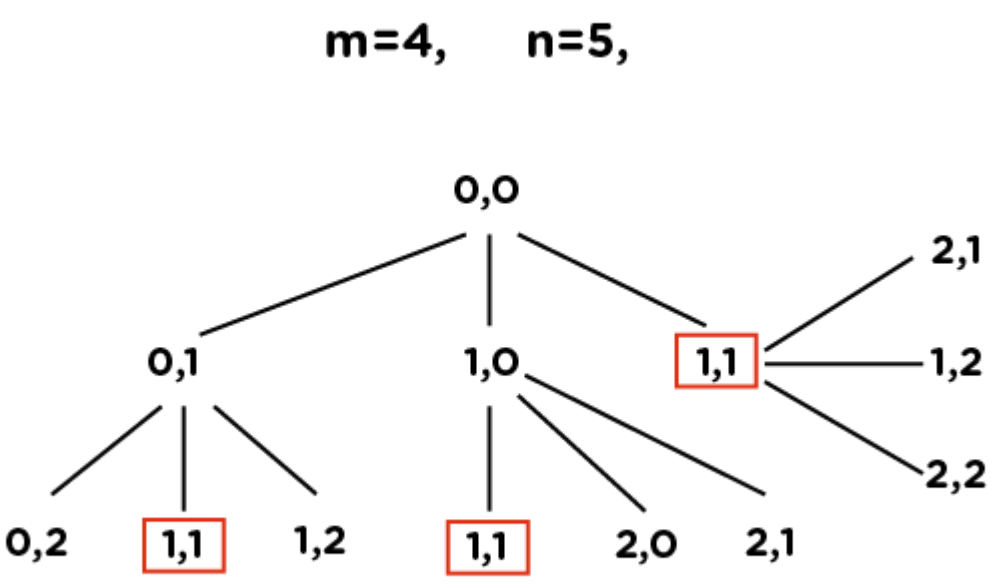
- **Thinking about the recursive approach to reach from the cell (0, 0) to (m-1, n-1), we need to decide for every cell about the direction to proceed out of three.**
- **We will simply call recursion over all the three choices available to us, and finally, we will be considering the one with minimum cost and add the current cell's value to it.**

Let's now look at the recursive code for this problem:

Pseudocode:

```
function minCost(cost,m , n, dp)
        //   base case
        If m == 0 AND n == 0
                return cost[m][n]

        //   outside the grid
        if m < 0 or n < 0
                return infinity

        recursionResult1 = minCost(cost, m-1, n , dp)
        recursionResult1 = minCost(cost, m, n-1 , dp)
        recursionResult1 = minCost(cost, m, n-1 , dp)
        myResult = min(recursionResult1, recursionResult2, recursionResult3) + cost[m][n]

        return myResult
```

Let's dry run the approach to see the code flow. Suppose, m = 4 and n = 5; then the recursive call flow looks something like below:



Here, we can see that there are many repeated/overlapping calls (for example: (1,1) is one of them), leading to exponential time complexity, i.e., O(3^n). If we store the output for each recursive call after their first occurrence, we can easily avoid the repetition. It means that we can improve this using memoization.

Now, let's move on to the Memoization approach.

In memoization, we avoid repeated overlapping calls by storing the output of each recursive call in an array. In this case, we will be using a 2D array instead of 1D, as the storage used for the memoization depends on the states, which are basically the necessary variables whose value at a particular instant is required to calculate the optimal result.

Refer to the memoization code (along with the comments) below for better understanding:

Pseudocode:

```
function minCost(cost,m , n, dp)
        //   base case
        If m == 0 AND n == 0
                return cost[m][n]

        //   outside the grid
        if m < 0 or n < 0
                return infinity

        if dp[m][n] != -1
                return dp[m][n]

        recursionResult1 = minCost(cost, m-1, n , dp)
        recursionResult1 = minCost(cost, m, n-1 , dp)
        myResult = min(recursionResult1, recursionResult2, recursionResult3) + cost[m][n]

        //   store in dp
        dp[m][n] = myresult

        return myResult
```

To get rid of the recursion, we will now proceed towards the DP approach.

The DP approach is simple. We just need to create a solution array (lets name that as ans), where:

ans[i][j] = minimum cost to reach from (i, j) to (m-1, n-1)

Now, initialize the last row and last column of the matrix with the sum of their values and the value, just after it. This is because, in the last row or column, we can reach there from their forward cell only (You can manually check it), except the cell (m-1, n-1), which is the value itself.

ans[m-1][n-1] = cost[m-1][n-1]
ans[m-1][j] = ans[m-1][j+1] + cost[m-1][j] (for 0 < j < n)
ans[i][n-1] = ans[i+1][n-1] + cost[i][m-1] (for 0 < i < m)

Next, we will simply fill the rest of our answer matrix by checking out the minimum among values from where we could reach them. For this, we will use the same formula as used in the recursive approach:

ans[i][j] = min(ans[i+1][j], ans[i+1][j+1], ans[i][j+1]) + cost[i][j]

Finally, we will get our answer at the cell (0, 0), which we will return.
The code looks as follows:

Pseudocode:

```
function minCost(cost, m, n)
        ans = array[m+1][n+1]
        ans[0][0] = cost[0][0]

        //   Initialize first column of ans array
        for i from 1 to m
                ans[i][0] = ans[i-1][0] + cost[i][0]

        //   Initialize first row of ans array
        for j from 1 to n
                ans[0][j] = ans[0][j-1] + cost[0][j]

        //   Construct rest of the ans array
        for i from 1 to m
                for j from 1 to
```

```
                        min_temp = min(ans[i-1][j-1], ans[i-1][j], ans[i][j-1])
                        ans[i][j] = min_temp + cost[i][j]

        return ans[m][n]
```

Note: This is the bottom-up approach to solve the question using DP.

Time Complexity: Here, we can observe that as we move from the cell (0,0) to (m-1, n-1), in general, the i-th row varies from 0 to m-1, and the j-th column runs from 0 to n-1. Hence, the unique recursive calls will be a maximum of (m-1) * (n-1), which leads to the time complexity of O(m*n).

Space Complexity: Since we are using an array of size (m*n) the space complexity turns out to be O(m*n).

LCS (Longest Common Subsequence)

Problem statement: The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

Note: Subsequence is a part of the string which can be made by omitting none or some of the characters from that string while maintaining the order of the characters.

If s1 and s2 are two given strings then z is the common subsequence of s1 and s2, if z is a subsequence of both of them.

Example 1:
s1 = "abcdef"
s2 = "xyczef"

Here, the longest common subsequence is cef; hence the answer is 3 (the length of LCS).

Approach: Let's first think of a brute-force approach using recursion. For LCS, we have to match the starting characters of both strings. If they match, then simply we can break the problem as shown below:
s1 = "x|yzar"
s2 = "x|qwea"

The rest of the LCS will be handled by recursion. But, if the first characters do not match, then we have to figure out that by traversing which of the following strings, we will get our answer. This can't be directly predicted by just looking at them, so we will be traversing over both of them one-by-one and check for the maximum value of LCS obtained among them to be considered for our answer.

For example:
Suppose, string s = "xyz" and string t = "zxay" .
We can see that their first characters do not match so that we can call recursion over it in either of the following ways:

A =

B =

C =

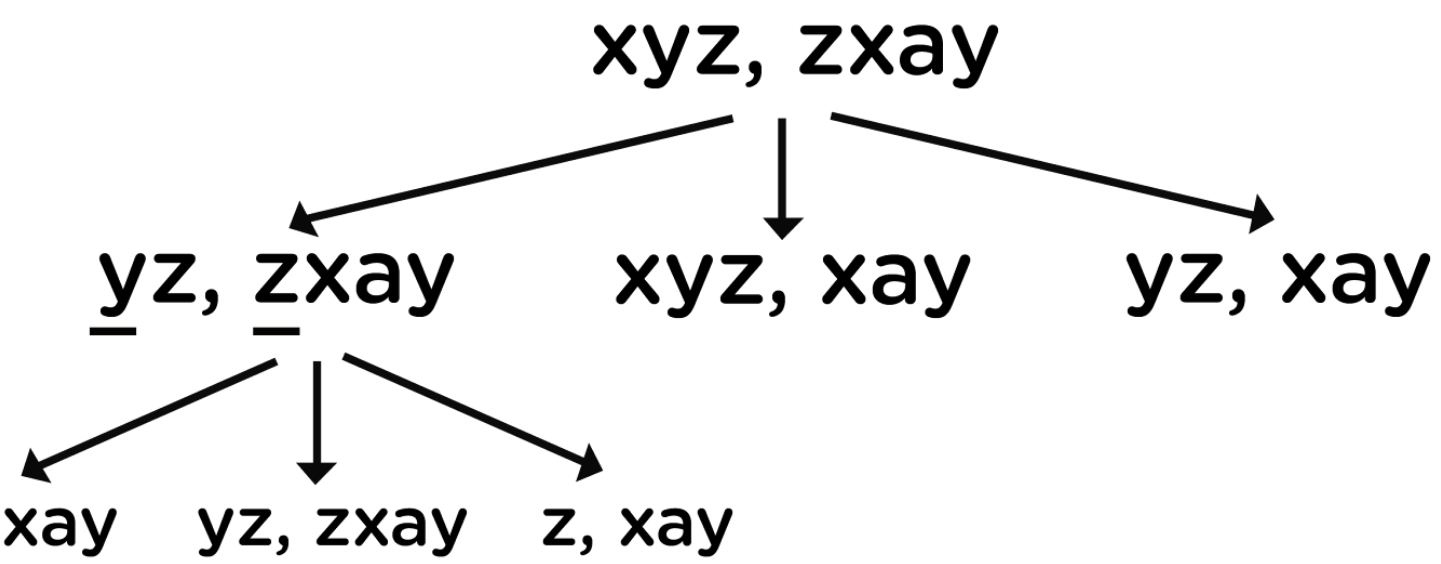
Finally, our answer will be: LCS = max(A, B, C)
Check the code below and follow the comments for a better understanding.

Pseudocode:

```
function lcs(s, t, m, n):
        //   Base Case
        if m equals 0 or n equals 0
```

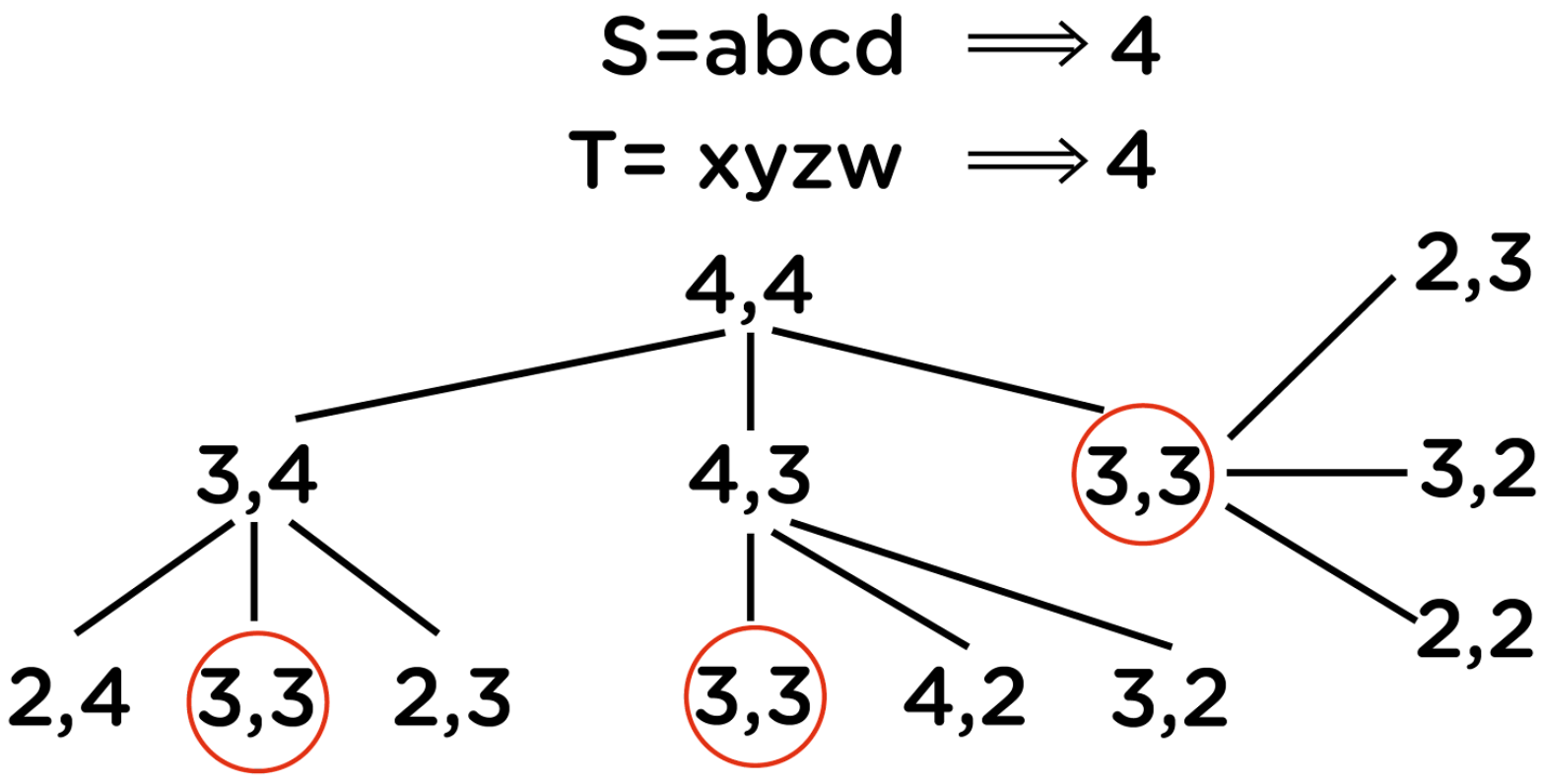
```
                return 0

        //   match m -1th character of s with n -1 th character of t
        else if s[m-1] equals t[n-1]
                return 1 + lcs(s, t, m-1, n-1)
        else:
                return max(lcs(s, t, m, n-1), lcs(s, t, m-1, n))
```

If we dry run this over the example: s = "xyz" and t = "zxay", it will look something like below:



Here, as for each node, we will be making three recursive calls, so the time complexity will be exponential and is represented as O(2^(m+n)), where m and n are the lengths of both strings. This is because, if we carefully observe the above code, then we can skip the third recursive call as it will be covered by the two others.
Now, thinking, over improving this time complexity...

Consider the diagram below, where we are representing the dry run in terms of its length taken at each recursive call:



As we can see there are multiple overlapping recursive calls, the solution can be optimized using memoization followed by DP. So, beginning with the memoization approach, as we want to match all the subsequences of the given two strings, we have to figure out the number of unique recursive calls. For string s, we can make at most length(s) recursive calls, and similarly, for string t, we can make at most length(t) recursive calls, which are also dependent on each other's solution. Hence, our result can be directly stored in the form of a 2-dimensional array of size (length(s)+1) * (length(t) + 1) as for string s, we have 0 to length(s) possible combinations, and the same goes for string t.

So for every index 'i' in string *s* and 'j' in string *t*, we will choose one of the following two options:

1. **If the character s[i] matches t[j], the length of the common subsequence would be one plus the length of the common subsequence till the i-1 and j-1 indexes in the two respective strings.**
2. **If the character s[i] does not match t[j], we will take the longest subsequence by either skipping i-th or j-th character from the respective strings.**

Hence, the answer stored in the matrix will be the LCS of both strings when the length of string s will be ‘i’ and the length of string t will be ‘j’. Hence, we will get the final answer at the position matrix[length(s)][length(t)]. Moving to the code:

Pseudocode:

```
function LCS(s, t, i, j, memo)
        //   one or both of the strings are fully traversed
        if i equals len(s) or j equals len(t)
                return 0

        //   if result for the current pair is already present in the table
        if memo[i][j] not equals -1
                return memo[i][j]

        if s[i] equals t[j]
                //   check for the next characters in both the strings
                memo[i][j] = lcs(s, t, i + 1, j + 1, memo) + 1
        else
                //   check if the current characters in both the strings are equal
                memo[i][j] = max(lcs(s, t, i, j+1, memo), lcs(s, t, i+1, j, memo))

        return memo[i][j]
```

Now, converting this approach into the DP code:

Pseudocode:

```
function LCS(s , t)

        //   find the length of the strings
        m = len(s)
        n = len(t)

        //   declaring the array for storing the dp values
        L = array[m + 1][n + 1]

        for i from 0 to m
                for j from 0 to n
                        if i equals 0 or j equals 0
                                L[i][j] = 0
                        else if s[i-1] equals t[j-1]
                                L[i][j] = L[i-1][j-1]+1
                        else:
                                L[i][j] = max(L[i-1][j] , L[i][j-1])

        //   L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]
        return L[m][n]
```

Time Complexity: We can see that the time complexity of the DP and memoization approach is reduced to O(m*n) where m and n are the lengths of the given strings.

Space Complexity: Since we are using an array of size (m*n) the space complexity turns out to be O(m*n).

Applications of Dynamic programming

- **They are often used in machine learning algorithms, for eg Markov decision process in reinforcement learning.**
- **They are used for applications in interval scheduling.**
- **They are also used in various algorithmic problems and graph algorithms like Floyd warshall’s algorithms for the shortest path, the sum of nodes in subtree, etc.**

Trie Notes

What are Tries?

Tries are a type of search tree, a tree data structure that is used to look for specific keys in a set of keys. In order to insert or access a key, we traverse in the depth-first order in the tree.

Introduction to tries

Suppose we want to implement a word-dictionary using a C++ program and perform the following functions:

- **Addition of the word(s)**
- **Searching a word**
- **Removal of the word(s)**

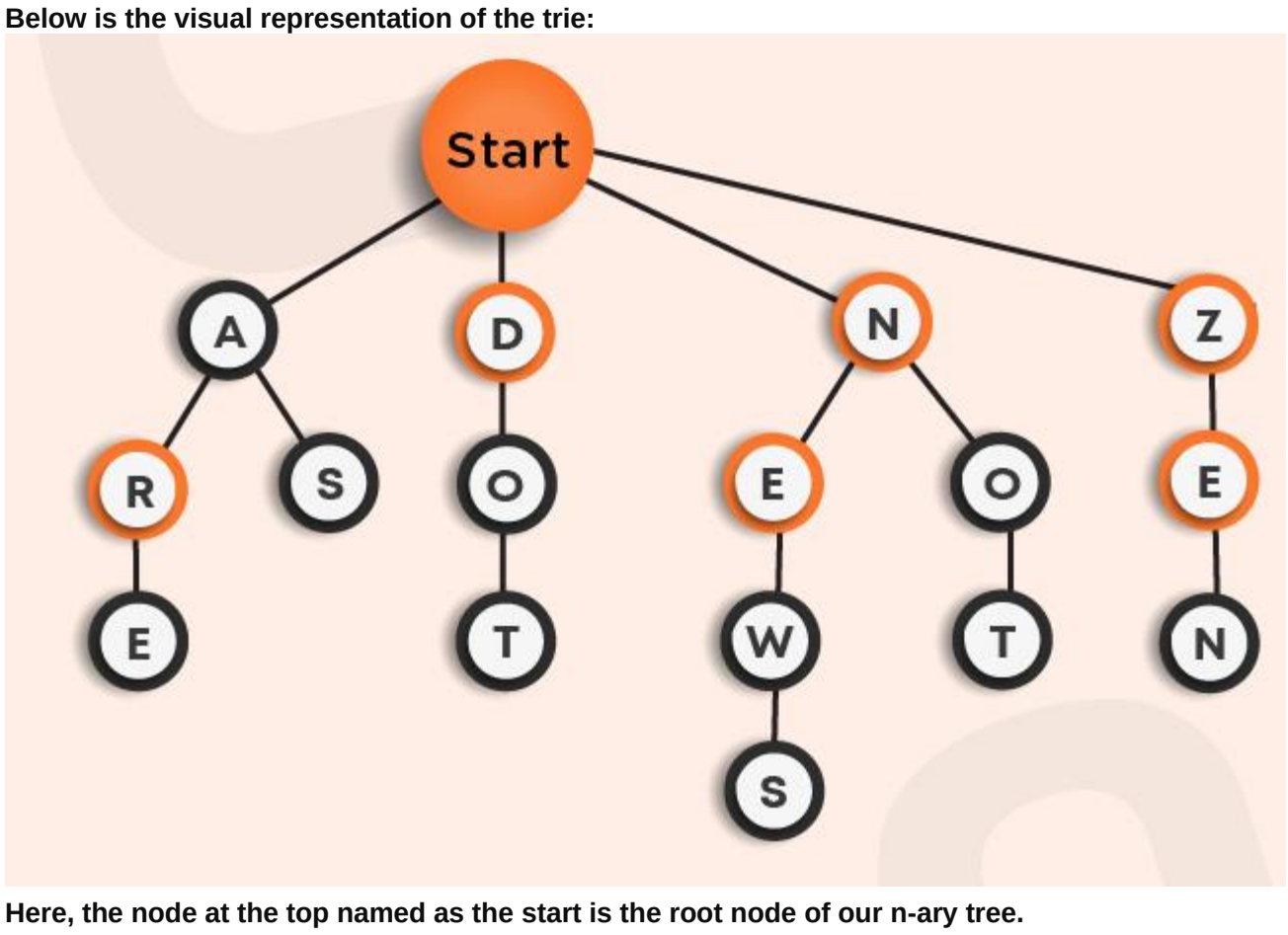
To do the same, hashmaps can be thought of as an efficient data structure as the average case time complexity of insertion, deletion, and retrieval is O(1) .

Let us discuss the time complexity of the same in the case of strings.

Suppose we want to insert string *abc* in our hashmap. To do so, first, we would need to calculate the hashcode for it, which would require the traversal of the whole string *abc*. Hence, the time taken will be the length of the entire string irrespective of the time taken to perform any of the above operations. Thus, we can conclude that the insertion of a string will be O(string_length).

To search a word in the hashmap, we again have to calculate the hashcode of the string to be searched, and for that also, it would require O(string_length) time.
Similarly, for removing a word from the hashmap, we would have to calculate the hashcode for that string. It would again require O(string_length) time.

For the same purpose, we can use another data structure known as tries.
Below is the visual representation of the trie:



Here, the node at the top named as the start is the root node of our n-ary tree.

Suppose we want to insert the word *ARE* in the trie. We will begin from the root, search for the first letter *A* and then insert *R* as its child and similarly insert the letter *E*. You can see it as the left-most path in the above trie.

Now, we want to insert the word *AS* in the trie. We will follow the same procedure as above. First-of-all, we find the letter *A* as the child of the root node, and then we will search for *S*. If *S* was already present as the child of *A*, then we will do nothing as the given word is already present, otherwise, we will insert *S*. You can see this in the above trie. Similarly, we added other words in the trie.

This approach also takes O(word_length) time for insertion as everytime we explore through one of the branches of the Trie to check for the prefix of the word already present.

Moving on to searching a word in the trie, for that also, we would have to traverse the complete length of the string, which would take O(word_length) time.

Let us take an example. Suppose we want to search for the word *DOT* in the above trie, we will first-of-all search for the letter *D* as the direct child of the start node and then search for *O* and *T* and, then we will return true as the word is present in the trie.

Consider another example *AR*, which we want to search for in the given trie. Following the above approach, we would return true as the given word is present in it. However, ideally, we should return false as the actual word was *ARE* and not *AR*. To overcome this, it can be observed that in the above trie, some of the letters are marked with a bolder circle, and others are not. This boldness represents the termination of a word starting from the root node. Hence, while searching for a word, we will always be careful about the last letter that it should be bolded to represent the termination.

Note: While insertion in a trie, the last letter of the word should be bolded as a mark of termination of a word.

In the above trie, the following are all possibilities that can occur:

- **A, ARE, AS**
- **DO, DOT**
- **NEW, NEWS, NO, NOT**
- **ZEN**

Here, to remove the word, we will simply unbold the terminating letter as it will make that word invalid. For example: If you want to remove the string *DO* from the above trie by preserving the occurrence of string DOT, then we will reach *O* and then unbold it. This way the word *DO* is removed but at the same time, another word *DOT* which was on the same path as that of word *DO* was still preserved in the trie structure.

Similarly, if we want to remove *DOT* while keeping *DO* in the trie we traverse the trie and when we reach T we delete it. Now the child of *O* is deleted but *O* is still bolded because it is the end of the word of another key, so we don't remove it from the trie and simply return from the function call.

For the removal of a word from trie, the time complexity is still O(word_length) as we are traversing the complete length of the word to reach the last letter to unbold it.

Operations on Trie

Insertion in Trie:

Given a word, we want to insert it into the Trie. We will assume that the word is already not present in the trie.
Steps for insert in the Trie

- **Start from the root.**
- **For each character of the string, from first to the last, check if it has a child corresponding to itself in the root and if not exists create a new one.**
- **Descend from the root to the child corresponding to the current character.**
- **set the last character's terminal property to be true.**

```
function insert(root, word)

    for each character c in word
                /*
                        check if it already present and
                        if not then create a new node
                */
                index = c - 'a'
                if root.child[index] equals null
                        root.child[index] = new node

                root = root.child[index]

        //  mark the last character to be the end of the word
        root.isTerminal = true
        return root
```

Search in Trie:

Given a word, we want to find out if it is present in the Trie or not.
Steps for search in the Trie

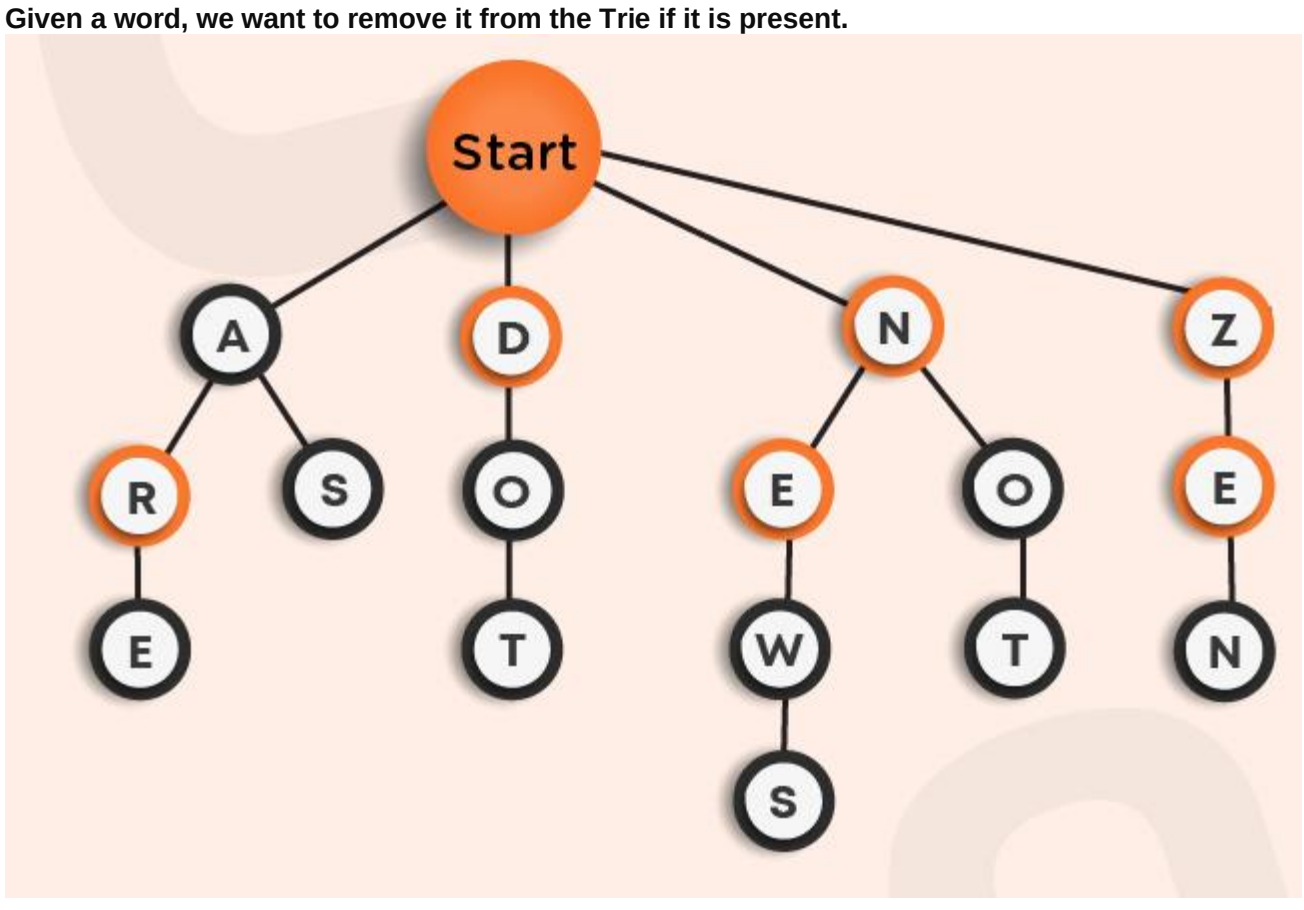
- **Start from the root.**
- **For each character of the string, from first to the last check. if it has a child corresponding to itself in the root, and if not then return false.**
- **Descend from the root to the child corresponding to the current character.**
- **Return true on successful completion of the above loop if the last node is not null and its terminal property is true.**

```
function search(root, word)

    for each character c in word
                /*
                        check if it already present and
                        if not then return false
                */
```

```
                index = c - 'a'
                if root.child[index] equals null
                        return false

                root = root.child[index]

    if root != null and root.isTerminal equals True
        return true
    else
        return false
```

Delete in Trie:

Given a word, we want to remove it from the Trie if it is present.



Consider the above example. Suppose we want to remove *NEWS* from the trie. We recursively call the delete function and as soon as we reach *S* and mark its isTerminal property as false. Now we see that it has no children remaining so we delete it. After returning we get to *W*. We see that the condition: isTerminal is false and no children remaining, is false for this node hence we just simply return again. Proceeding the same way back up to *E* and *N* we delete the word *NEWS* from the trie.

Steps for delete in the Trie

- **Call the function delete for the root of the trie**
- **Update the child corresponding to the current character by calling delete for the child.**
- **If we have reached the final character, mark the isTerminal property of the node as False and delete this node if all the children are NULL for this node.**
- **If all the children of this node have been deleted and this character is not the prefix of any other word we also delete the current node.**

```
function delete(root, depth, word)
        if root is null
                /*
                        The word does not exist
                        hence return null
                */
                return null
        if depth == word.size
                /*
                        mark the isTerminal as false and delete
                        if no child is present
                */
                root.isTerminal = false
                if root.children are null
                        delete(root)
                        root = null
                return root
        index = word[depth] - 'a'
        /*
                update the child of the root corresponding
                to the current character
```

```
        */
        root.children[index] = delete(root.children[index], depth + 1 , word)

        if(root.children are null and root.isTerminal == false)
                delete(root)
                root = null
        return root
```

Time Complexity

if L is the length of the string we want to insert, search or delete from the trie, the time complexities of various operations are as follows: -

Operations	Time Complexity
Insert(word)	O(L)
Search(word)	O(L)
delete(word)	O(L)

Applications

- **Tries are used to implement data structures and algorithms like dictionaries, lookup tables, matching algorithms, etc.**
- **They are also used for many practical applications like auto-complete in editors and mobile applications.**
- **They are used in phone book search applications where efficient searching of a large number of records is required.**

Bit Manipulation

Bit Manipulation

What is Bit Manipulation?

Bit manipulation is basically the act of algorithmically manipulating bits (smallest unit of data in a computer).
A bit represents a logical state with one of two possible values ‘0’ or ‘1’, which in other representations can be referred to as ‘true’ or ‘false’ / ‘on’ or ‘off’.Since bits represent logical states efficiently, this makes the binary system suitable for electronic circuits that use logic gates and this is why binary is used in all modern computers.

Decimal and Binary number systems

- **Decimal number system:**

We usually represent numbers in decimal notation(base 10) that provides ten unique digits - 0,1,2,3,4,5,6,7,8,9. To form any number, we can combine these digits to form a sequence such that each decimal digit represents a value multiplied by a power of 10 in this sequence.

For example:
244 = 200 + 40 + 4 = 2*10^2 + 4*10^1 + 4*10^0.

- **Binary number system:**

The binary system works in a similar manner to that of the decimal number system, the only difference lies in the fact that binary notation(base 2) provides two digits - 0 and 1. A binary number is defined as a sequence of 1s and 0s like ‘010100’, ‘101’, ‘0’,’1111’,’000111’, etc. In a binary number, each digit is referred to as a bit, where each such bit represents a power of decimal 2.

Similar to decimal notation we can convert a given binary sequence to its decimal equivalent.

For example:
(base 2)1100: (Base 10) 1*2^3 + 1*2^2 + 0*2^1 + 0*2^0 = 8 + 4 + 0 + 0 = 12.
A bit is said to be set if it is ‘1’ and unset if the bit is ‘0’.

Bitwise Operators

Bitwise operators are used to performing bitwise operations. A bitwise operation operates on a bit string(string consisting of 0s and 1s only) or a bit array(array of 0s and 1s only) at the level of its individual bits. Bit operations are usually fast and are used for calculations and comparisons at the processor level.

- **Bitwise AND(&)**

A binary operator that takes bit representation of its operands and the resulting bit is 1 if both the bits in bits patterns are 1 otherwise, the resulting bit is 0.

For example:
X = 5 (101)2 and Y = 3 (011)2
X & Y = (101)2 & (011)2 = (001)2 = 1

- **Bitwise OR(|)**

A binary operator that takes bit representation of its operands and the resulting bit is 0 if both the bits in bits patterns are 0 otherwise, the resulting bit is 1.

For example:
X = 5 (101)2 and Y = 3 (011)2
X | Y = (101)2 | (011)2 = (111)2 = 7

- **Bitwise NOT(~)**

A unary operator that takes bit representation of its operand and just flips the bits of the bit pattern, i.e if the ith bit is 0 then it will be changed to 1 and vice versa.
Bitwise NOT is the 1's complement of a number.

For example:
X = 5 (101)2
~X = (010)2 = 2

- **Bitwise XOR(^)**

A binary operator that takes bit representations of its operands and the resulting bit is 0 if the bits in bit patterns are the same i.e the bits in both the patterns at a given position is either 1 or 0 otherwise, the resulting bit is 1 if the bits in bit patterns are different.

For example:
X = 5 (101)2 and Y = 3 (011)2
X ^ Y = (101)2 ^ (011)2 = (110)2 = 6

- **Left shift operator(<<)**

A binary operator that left shifts the bits of the first operand, with the second operand deciding the number of places to shift left, and append that many 0s at the end. We discard the k left most bits.

For example:
X = 5 (101)2
X << 2 = (00101)2 << 4 = (10100)2 = 20
In other words, left shift by k places is equivalent to multiplying the bit pattern by 2^k, as shown in the above example - 5 << 2 = 5 * 2^2 = 20.
So, A << x = A * 2^x that is why (1<< n) is equal to power(2,n) or 2^n.

- **Right shift operator(>>)**

A binary operator that right shifts the bits of the first operand, with the second operand deciding the number of places to shift right. We discard the k rightmost bits.

For example:
X = 5 (101)2
X >> 2 = (101)2 = (001)2 = 1

In other words, the right shift by k places is equivalent to dividing the bit pattern by 2^k (integer division), as shown in the above example - 5 >> 2 = 5 / (2^2) = 1.

So, A >> x = A / (2^x) (integer division).

X	Y	X & Y	X | Y	X ^ Y	~X
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Some useful bitwise algorithms/bit hacks

- **Checking if a number is a power of 2**

Algorithm:

The numbers which are powers of 2 have only one set bit in their binary representation. If the number is 0 or is not a power of 2 then it will have more than one bit set in binary representation.
Let us say, we need to check if a number x is a power of 2 or not.

Now, think about the binary representation of (x-1), the binary representation of (x-1) will have all the bits the same as x except for the rightmost set bit in x and all the bits to the right of the rightmost set bit.

For example:
x = 4 (00100)2
x-1 = 3 (00011)2
x = 10 (1010)2
x-1 = 9 (1001)2

Now, how (x-1) can help determine whether x is a power of 2?
The binary representation of (x-1) can be simply obtained from the binary representation of x by flipping all the bits to the right of the rightmost set bit including the rightmost set bit itself, so if x has only one-bit set, then x & (x-1) must be equal to 0.

For example:
x = 4 (00100)2
x-1 = 3 (00011)2
x & (x-1) = (00000)2
In the above example, since the rightmost set bit and all the bits to the right of it were flipped, so the bitwise and will make all those bits 0 and we have no set bits left, which tells us that there was only one set bit in x.

Pseudocode:

```
function isPowerOf2(num)
        return (num && ! (num & (num - 1)))
```

A similar approach can be used for checking if a given number is even or odd.

One of the obvious approaches would be to check the number's divisibility by 2, but the other approach is to check if the last(rightmost) bit i.e if the last bit is 1 or not if the last bit is 1 then the number is odd, otherwise even, as the rightmost bit i.e 2^0 only contributes odd value - 1 to the number, all other bits represent the value of 2^i where i > 0, so all values remain even.

- **Checking if the kth bit is set/unset**

Algorithm:

We need to check if the kth bit in the binary representation of the given number is set or unset. We can use bitwise AND operations to check this efficiently.
We can take another number n = (1<<k), whose all bits are 0 except the kth bit.
Now doing AND operation with the given number say x, must return a positive number if the kth bit is set, otherwise the result of bitwise AND will be 0.

For example:
x = 5 (101)2 , k = 2
So, let no = (1 << k) = 2^k = 4 = (100)2
x & no = (101)2 & (100)2 = (100)2 = 4.
So the 2nd bit of x is set.

Pseudocode:

```
function checkBit(N, k)
        if (N & (1<<k))
                return true
        else
                return false
```

- **Counting number of 1s in the binary representation of a number**

The naive approach is to simply convert the given number into its binary representation and simply count the number of 1s in the binary representation, which takes O(log2N) time, where N is the number.

However, we can efficiently count the number of 1s in the binary representation of a number whose time complexity depends on the number of 1s in its binary representation.

Algorithm:

Let us say we want to find the number of 1s in the binary representation of x, we can perform a similar operation used to check if a given number is a power of 2.

We can apply bitwise and between x and (x-1) and then update the value of x to x & (x-1), every time this operation flips the rightmost set bit and the bits after the rightmost set bit, so every time the rightmost set bit is unset, which helps us count the number of 1s in the binary representation of the number.

For example:
x = 5 (101)2

Step 1: x = 5 (101)2
x-1 = 4 (100)2
x & (x-1) = (100)2 = 4, update x to 4.

Step 2: x = 4 (100)2
x-1 = 3 (011)2,
x & (x-1) = (000)2 = 0
So, the number of set bits are 2 in x.

The above algorithm described is known as Brian Kernighan's Algorithm. The time complexity of the algorithm is dependent on the number of 1s in the binary representation of the number, however in the worst case still the time complexity will be O(log2N) as all bits of the number may be set, where N is the number.

Pseudocode:

```
function count1s(num)
        count = 0
        while(num > 0)
                num = num & (num-1)
                count = count+1
        return count
```

- **Generating all possible subsets of a set**

Bit manipulation can be useful for the generation of all the possible subsets of a set. We can represent the elements of the set in the form of a binary sequence having a length equal to the number of elements in the set. Since the number of subsets of a set having N elements is 2^N, so a binary sequence of length N will represent values from range 0 to 2^N-1 which is equal to the number of subsets of the set.

All set bits in the binary sequence denote the elements present from the set, thus representing a subset.

For example:
Set S = {10,20,30}
We know that the number of subsets of a set having N elements is 2^N.In this case N = 3, so 2^N = 8 subsets.
So taking a binary sequence of length 3, where each bit value represents whether the element of the set is present(bit value is 1) or not(bit value is 0) in the corresponding subset.
0 = 000 => {} - empty subset
1 = 001 => {30}
2 = 010 => {20}
3 = 011 => {20,30}
4 = 100 => {10}
5 = 101 => {10,30}
6 = 110 => {10,20}
7 = 111 => {10,20,30}

Hence, the binary sequence is useful to represent all the subsets of a set.

Pseudocode:

```
function generateSubsets(arr, N)
        /*
                arr represents the set of size N
                Iterating from 0 to 2^N-1 to get all subsets
        */
        for i = 0 to 2^N-1
                /*
                        For each i representing a subset checking which
                        bits are set in binary representation of i
                */
                for j = 0 to N-1
                        if (i & (1 << j))
                                print arr[j]
                print newline
        return
```

Some other bitwise tricks

- **res = x | (1 << k): Set the bit at kth position in the binary representation of x.**

For example:
x = 5 (101)2 , k = 1
Let no = (1 << k) = 2^1 = 2 = (010)2
res = x | no = (101)2 | (010)2 = (111)2 = 7

- **res = x & (~(1 << k)): Unset the bit at kth position in binary representation of x.**

For example:

x = 5 (101)2 , k = 2
Let no = (1 << k) = 2^2 = 4 = (100)2
no = ~(no) = (011)2
res = x & no = (101)2 & (011)2 = (001)2 = 1

- **res = x ^ (1 << k): Toggle the bit at kth position in the binary representation of x i.e if the bit at kth position is 0 then change it to 1 and vice versa.**

For example:
x = 5 (101)2 , k = 0
Let no = (1 << k) = 2^0 = 1 = (001)2
res = x ^ no = (101)2 ^ (001)2 = (100)2 = 4

- **res = x & (x - 1): Unsetting the rightmost set bit of x.**

For example:
x = 5 (101)2
x-1 = 4 (100)2
res = x & (x-1) = (101)2 & (100)2 = (100)2 = 4

- **res = ~x + 1 (2's complement): 2's complement of a number is its 1's complement +1. 2's complement of a number is the same as negative of the number i.e -x.**
- **res = x & (-x): Getting the rightmost set bit of x.**

For example:
x = 5 (101)2
1's complement: ~x = (010)2
2's complement: ~x + 1 = (011)2
res = x & (-x) = (101)2 & (011)2 = (001)2 = 1

Applications of bit manipulation

- **Bitwise operations are prominent in embedded systems, control systems, etc where memory(data transmission/data points) is still an issue.**
- **They are also useful in networking where it is important to reduce the amount of data, so booleans are packed together. Packing them together and taking them apart use bitwise operations and shift instructions.**
- **Bitwise operations are also heavily used in the compression and encryption of data.**
- **Useful in graphics programming, older GUIs are heavily dependent on bitwise operations like XOR(^) for selection highlighting and other overlays.**

Greedy Notes

Introduction

A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. Greedy algorithms work in stages. In each stage, a decision is made that is good at that point, without bothering about the future. This means that some local best is chosen. It assumes that a local good selection makes for a globally optimal solution.

Elements of the greedy algorithm

The basic properties of greedy algorithms are

- **Greedy choice property: This property says that the globally optimal solution can be obtained by making a locally optimal solution. The choice made in a greedy algorithm may depend on earlier choices but never on future choices. It iteratively makes one greedy choice after another and reduces the given problem to a smaller one.**
- **Optimal substructure: A problem exhibits optimal substructure if the optimal solution to the problem contains optimal solutions to the subproblems. That means we can solve subproblems and build up the solution to solve the larger problems.**

Example: Fractional Knapsack Problem: Given item t1, t2, …, tn (items we might want to carry in a backpack) with associated weights s1, s2, …, sn and benefit values v1, v2, …, vn, how can we maximize the total benefit considering we are subject to an absolute weight limit C?

Algorithm:

1. **Compute value per size density for each item di = vi/si.**
2. **Sort each item by its value density.**
3. **Take as much as possible of the density item not already in the bag.**

Time Complexity: O(nlogn) for sorting and O(n) for greedy selection.

Advantages of Greedy Algorithm

The main advantage of the greedy method is that it is straightforward, easy to understand, and easy to code. In greedy algorithms, once we make a decision, we do not have to spend time re-examining the already computed values.

Disadvantages of Greedy Algorithm

The main disadvantage is that for many problems there is no greedy algorithm. This means that in many cases there is no guarantee that making locally optimal improvements gives the optimal global solution. We also need proof of local optimality as to why a particular greedy solution will work.

Applications of Greedy Algorithm

- **Sorting: Selection sort, Topological sort**
- **Priority Queues: Heap Sort**
- **Huffman coding Compression algorithm**
- **Prim's and Kruskal's algorithm**
- **Shortest path in the weighted graph(Dijkstra's)**
- **Coin change problem (for Indian currency)**
- **Fractional knapsack problem**
- **Disjoint sets: union by size or union by rank**
- **Job scheduling algorithm**
- **The greedy technique can be used as an approximation algorithm for complex problems.**

Hashmap Notes

Introduction

Suppose we are given a string or a character array and we are asked to find the maximum occurring character. It could be quickly done using arrays. We can simply create an array of size 256, initialize this array to zero, and then, simply traverse the array to increase the count of each character against its ASCII value in the frequency array. In this way, we will be able to figure out the maximum occurring character in the given string.

The above method will work fine for all the 256 characters whose ASCII values are known. But what if we want to store the maximum frequency string out of a given array of strings? It can't be done using a simple frequency array, as the strings do not possess any specific identification value like the ASCII values. For this, we will be using a different data structure called hashmaps.

In hashmaps, the data is stored in the form of keys against which some value is assigned. Keys and values don't need to be of the same data type.

If we consider the above example in which we were trying to find the maximum occurring string, using hashmaps, the individual strings will be regarded as keys, and the value stored against them will be considered as their respective frequency.

For example, The given string array is:
str[] = {"abc", "def", "ab", "abc", "def", "abc"}

Hashmap will look as follows with strings as keys and frequencies as values :

Key (datatype = string)	Value (datatype = int)
"abc"	3
"def"	2
"ab"	1

From here, we can directly check for the frequency of each string and hence figure out the most frequent string among them all.

Note: One more limitation of arrays is that the indices could only be whole numbers but this limitation does not hold for hashmaps.

The values are stored corresponding to their respective keys and can be invoked using these keys. To insert, we can do the following:

- **hashmap[key] = value**
- **hashmap.insert(key, value)**

The functions that are required for the hashmaps are (using templates):

- **insert(k key, v value): To insert the value of type v against the key of type k.**
- **getValue(k key): To get the value stored against the key of type k.**
- **deleteKey(k key): To delete the key of type k, and hence the value corresponding to it.**

To implement the hashmaps, we can use the following data structures:

1. **Linked Lists: To perform insertion, deletion, and search operations in the linked list, the time complexity will be O(n) for each as:**

1. **For insertion, we will first have to check if the key already exists or not, if it exists, then we just have to update the value stored corresponding to that key.**
2. **For search and deletion, we will be traversing the length of the linked list.**

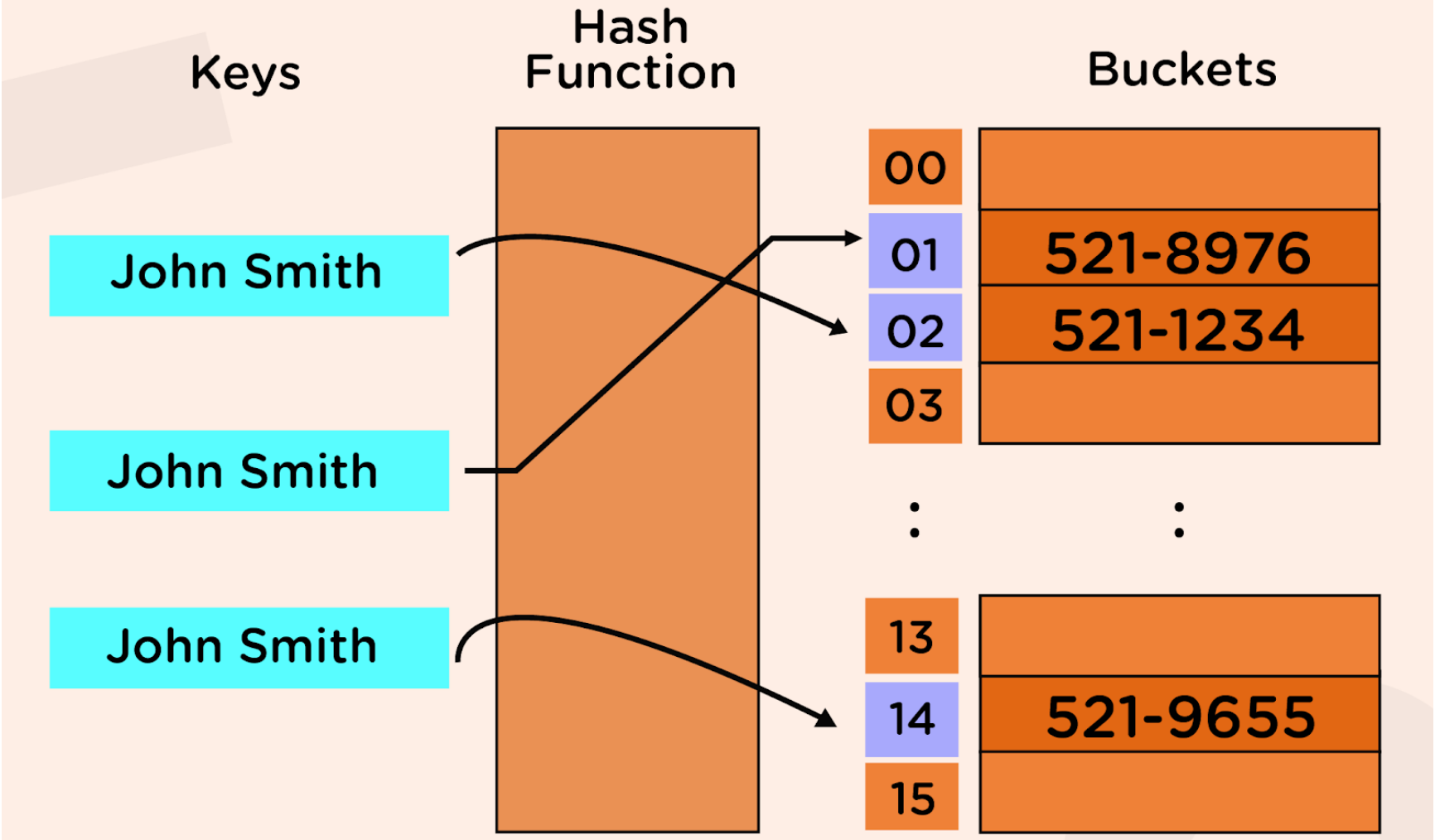
2. **BST: We can use some kind of a balanced BST so that the height remains of the order O(logN). For using a BST, we will need some sort of comparison between the keys. In the case of strings, we can do the same. Hence, insertion, search, and deletion operations are directly proportional to the height of the BST. Thus the time complexity reduces to O(logN) for each.**
3. **Hash table: Using a hash table, the time complexity of insertion, deletion, and search operations, could be improved to O(1) (same as that of arrays). We will study this in further sections.**

Bucket array and Hash function

Now, let's see how to perform insertion, deletion, and search operations using hash tables. Arrays are one of the fastest ways to extract data as compared to other data structures as the time complexity of accessing the data in the array is O(1), so we will try to use them in implementing the hashmaps.

Now, we want to store the key-value pairs in an array, named bucket array. We need an integer corresponding to the key so that we can keep it in the bucket array. To do so, we use a hash function. A hash function converts the key into an integer, which acts as the index for storing the key in the array.

For example: Suppose, we want to store some names from the contact list in the hash table, check out the following image:



Suppose we want to store a string in a hash table, and after passing the string through the hash function, the integer we obtain is equal to 10593, but the bucket array's size is only 20. So, we can't store that string in the array as 10593, as this index does not exist in the array of size 20.

To overcome this problem, we will divide the hashmap into two parts:

- **Hash code**
- **Compression function**

The first step to store a value into the bucket array is to convert the key into an integer (this could be any integer irrespective of the size of the bucket array). This part is achieved by using a hashcode. For different types of keys, we will be having different kinds of hash codes. Now we will pass this value through the compression function, which will convert that value within the range of our bucket array's size. Now, we can directly store that key against the index obtained after passing through the compression function.

The compression function can be used as (%bucket_size).

One example of a hash code could be: (Example input: "abcd")

"abcd" = ('a' * p^3) + ('b' * p^2) + ('c' * p^1) + ('d' * p^0)

Where p is generally taken as a prime number so that they are well distributed.

But, there is still a possibility that after passing the key through from hash code, when we give the same through the compression function, we can get the same values of indices. For example, let s1 = “ab” and s2 = “cd”. Now using the above hash function for p = 2, h1 = 292 and h2 = 298. Let the bucket size be equal to 2. Now, if we pass the hash codes through the compression function, we will get:

Compression_function1 = 292 % 2 = 0
Compression_function2 = 298 % 2 = 0

This means they both lead to the same index 0.
This is known as a collision.

Collision Handling

We can handle collisions in two ways:

- **Closed hashing (or closed addressing)**
- **Open addressing**

In closed hashing, each entry of the array will be a linked list. This means it should be able to store every value that corresponds to this index. The array position holds the address to the head of the linked list, and we can traverse the linked list by using the head pointer for the same and add the new element at the end of that linked list. This is also known as separate chaining.

On the other hand, in open addressing, we will check for the index in the bucket array if it is empty or not. If it is empty, then we will directly insert the key-value pair over that index. If not, then will we find an alternate position for the same. To find the alternate position, we can use the following:

hi(a) = hf(a) + f(i)

Where hf(a) is the original hash function, and f(i) is the ith try over the hash function to obtain the final position hi(a).

To figure out this f(i), the following are some of the techniques:

1. **Linear probing: In this method, we will linearly probe to the next slot until we find the empty index. Here, f(i) = i.**
2. **Quadratic probing: As the name suggests, we will look for alternate i^2 positions ahead of the filled ones, i.e., f(i) = i^2.**
3. **Double hashing: According to this method, f(i) = i * H(a), where H(a) is some other hash function.**

In practice, we generally prefer to use separate chaining over open addressing, as it is easier to implement and is also more efficient.

Advantages of HashMaps

- **Fast random memory access through hash functions**
- **Can use negative and non-integral values to access the values.**
- **Keys can be stored in sorted order hence can iterate over the maps easily.**

Disadvantages of HashMaps

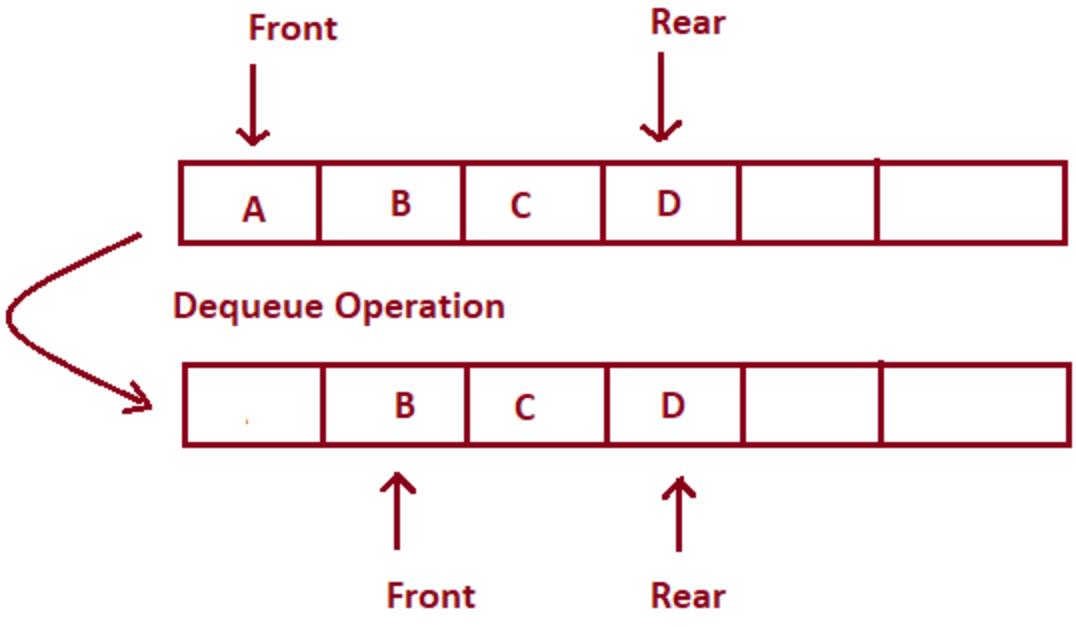
- **Collisions can cause large penalties and can blow up the time complexity to linear.**
- **When the number of keys is large, a single hash function often causes collisions.**

Applications of HashMaps

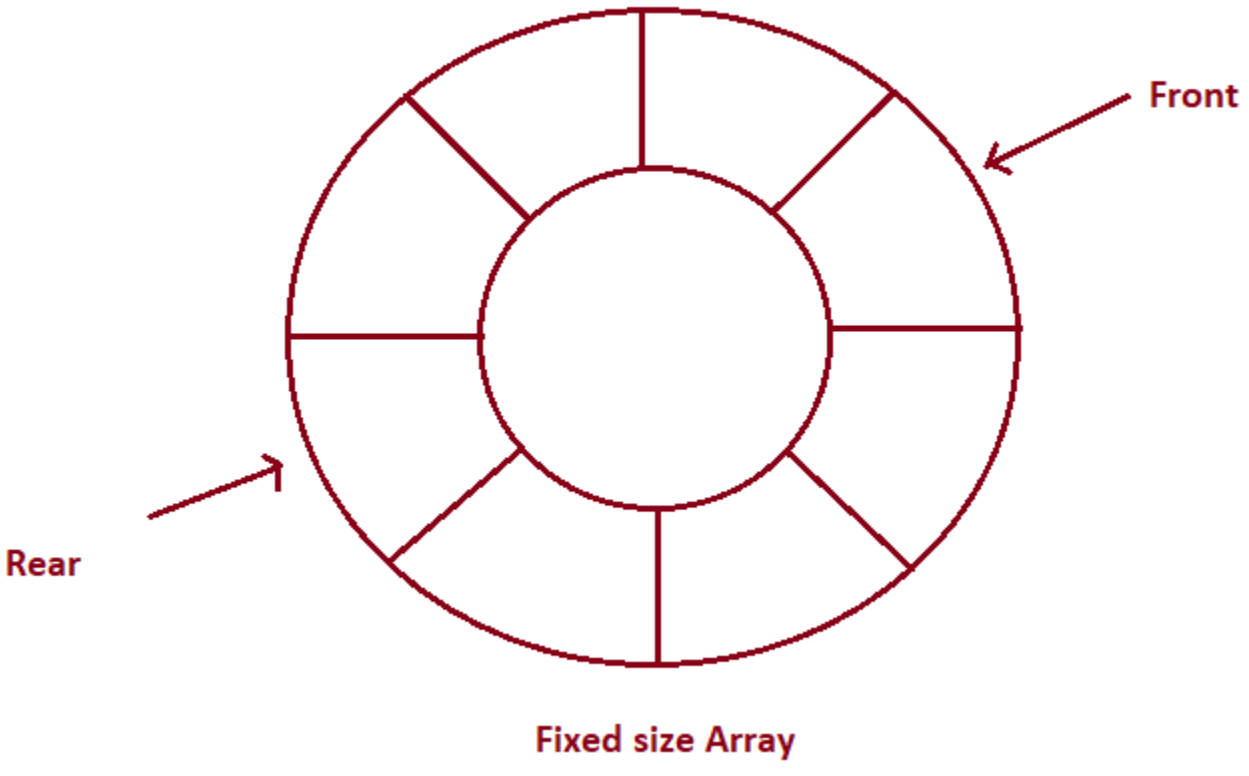
- **These have applications in implementations of Cache where memory locations are mapped to small sets.**
- **They are used to index tuples in Database management systems.**
- **They are also used in algorithms like the Rabin Karp pattern matching algorithm.**

Circular Queues Notes

Why circular Queues?



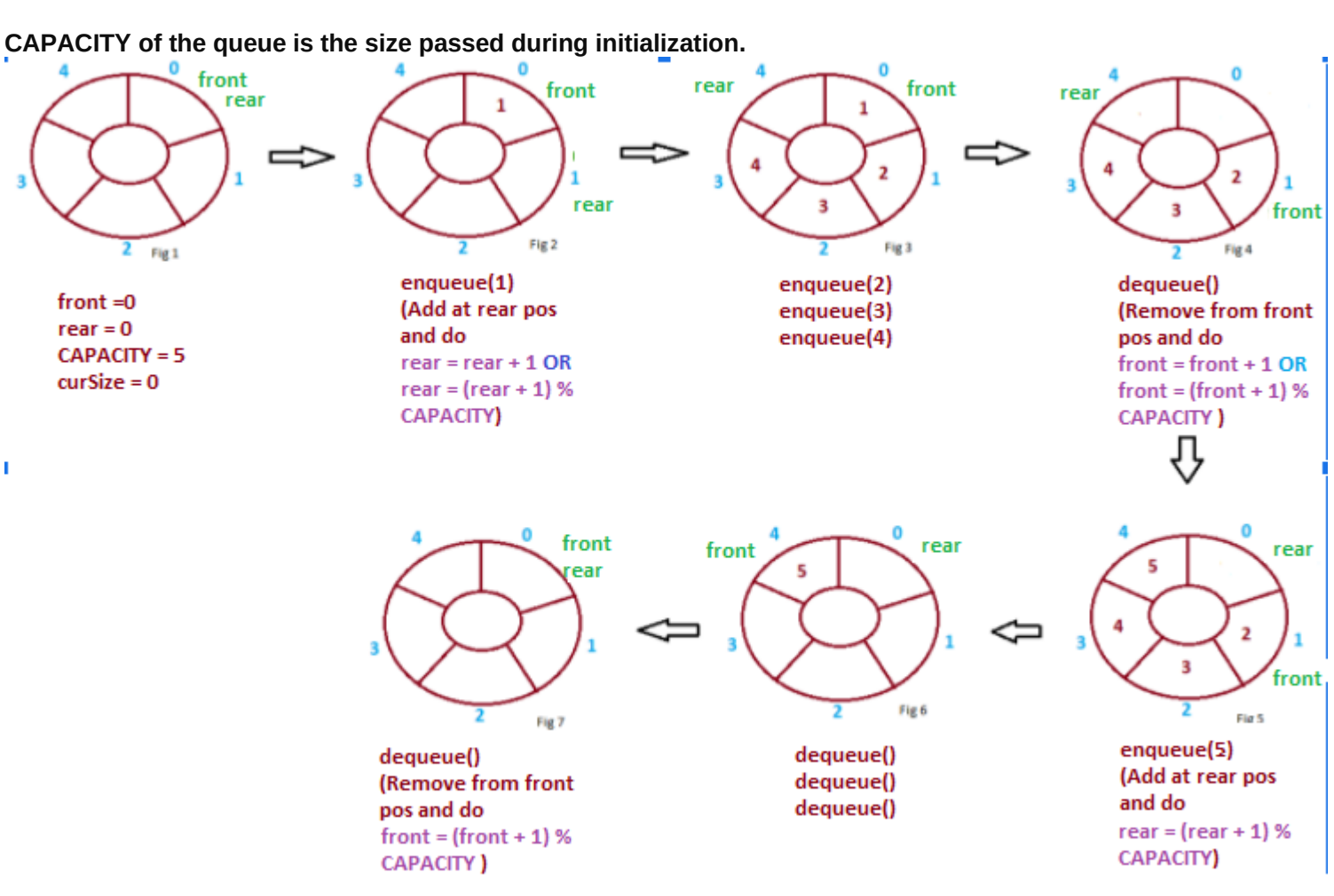
In the figure above, it can be seen that starting slots of the array are getting wasted. Therefore, simple array implementation of a queue is not memory efficient. To solve this problem, we assume arrays as circular arrays. With this representation, if we have any free slots at the beginning, the rear pointer can go to its next free slot.



How can circular queues be implemented?

Queues can be implemented using arrays and linked lists. The basic algorithm for implementing a queue remains the same. We maintain three variables for all the operations: front, rear, and curSize.

CAPACITY of the queue is the size passed during initialization.



For the front and rear to always remain within the valid bounds of indexing, we update front as (front + 1) % CAPACITY and rear as (rear + 1) % CAPACITY. This allows front and rear to never result in an index out of bounds exception.

Notice in fig 5 we cannot do rear = rear + 1 as it will result in index out of bound exception, but there is an empty position at index 0, so we do rear = (rear + 1) % CAPACITY. Similarly, in fig 7, we cannot do front = front + 1 as it will also give an exception therefore we do front = (front + 1) % CAPACITY.

- **Enqueue Operation**

Pseudocode:

```
function enqueue(data)

          //   curSize = CAPACITY when queue is full
          if queue is full
                    return "Full Queue Exception"

          curSize++
          queue[rear] = data

          //   updating rear to the next position in the circular queue
          rear = (rear+1) % CAPACITY
```

- **Dequeue Operation**

Pseudocode:

```
function  dequeue()

          //   front = rear = 0 OR curSize = 0 when queue is empty
          if queue is empty
                    return "Empty Queue Exception"

          curSize--
          temp = queue[front % CAPACITY]
          front = (front + 1 ) % CAPACITY
          return temp
```

- **getFront Operation**

Pseudocode:

```
function  getFront()

          if queue is empty
                    return "Empty Queue Exception"

          temp = queue[front]
          return temp
```

Time Complexity of various operations

Let 'n' be the number of elements in the queue. The complexities of queue operations with this representation can be given as:

Operations	Time Complexity
enqueue(data)	O(1)
dequeue()	O(1)
getFront()	O(1)
boolean isEmpty()	O(1)
int size()	O(1)

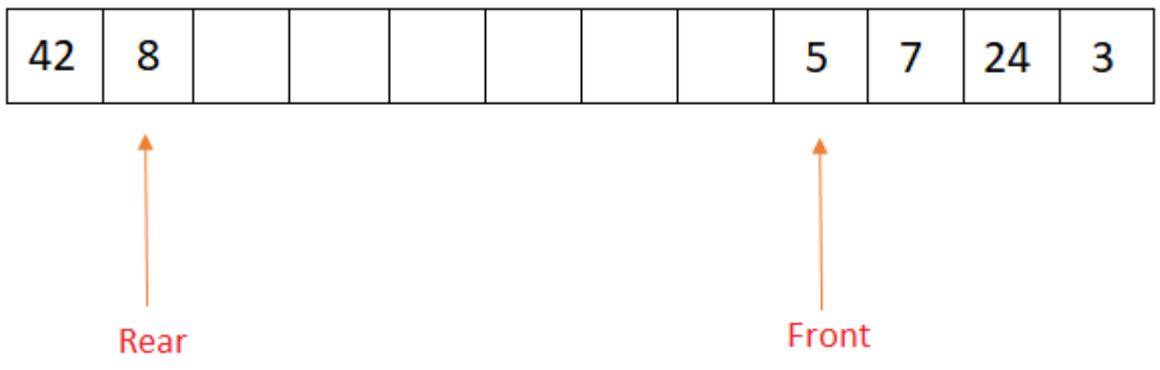
Application of circular queues

- **It is used in the looped execution of slides of a presentation.**
- **It is used in browsing through the open windows applications using alt + tab (Microsoft Windows)**
- **It is used for round-robin execution of jobs in multiprogramming OS.**
- **Used in the functioning of traffic lights.**
- **Used in page replacement algorithms: a circular list of pages is maintained and when a page needs to be replaced, the page in the front of the queue will be chosen.**

Deques Notes

Introduction to Deques

A deque, also known as the double-ended queue is an ordered list in which elements can be inserted or deleted at either end. It is also known as a head-tail linked list because elements can be added to or removed from either the front (head) or the back (tail).



Properties of Deques

- **Deque can be used both as stack and queue as it allows insertion and deletion of elements from both ends.**
- **Deque does not require LIFO and FIFO orderings enforced by data structures like stacks and queues.**
- **There are two variants of double-ended queues -**

1. **Input restricted deque: In this deque, insertions can only be done at one of the ends, while deletions can be done from both ends.**
2. **Output restricted deque: In this deque, deletions can only be done at one of the ends, while insertions can be done on both ends.**

Operations on Deques

- **enqueue_front(data): Insert an element at the front end.**
- **enqueue_rear(data): Insert an element at the rear end.**
- **dequeue_front(): Delete an element from the front end.**
- **dequeue_rear(): Delete an element from the rear end.**
- **front(): Return the front element of the dequeue.**
- **rear(): Return the rear element of the dequeue.**
- **isEmpty(): Returns true if the deque is empty.**
- **isFull(): Returns true if the deque is full.**

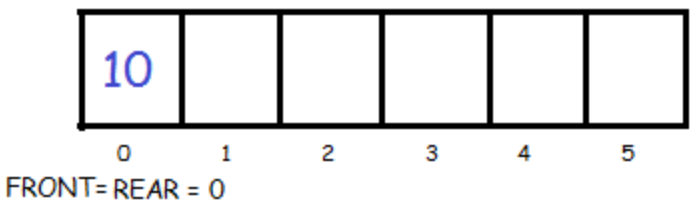
Implementation of Deques

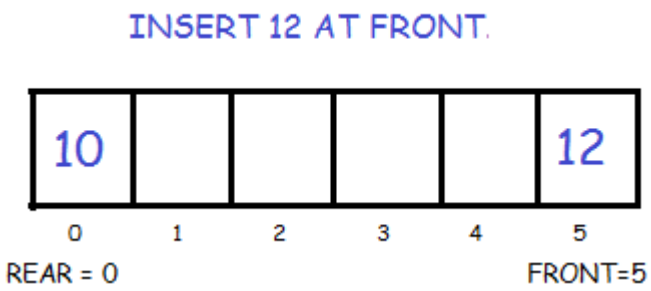
Deques can be implemented using data structures like circular arrays or doubly-linked lists. Below is the circular array implementation for deques, the same approach can be used to implement deque using doubly-linked lists.
We maintain two variables: front and rear, front represents the front end of the deque and rear represents the rear end of the deque.
The circular array is represented as "carr", and size represented by size, having elements indexed from 0 to size - 1.

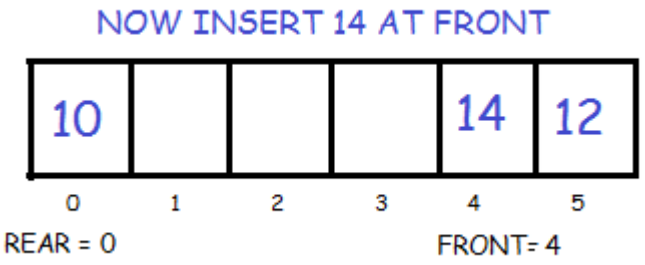
- **Inserting an element at the front end involves decrementing the front pointer.**
- **Deleting an element from the front end involves incrementing the front pointer.**
- **Inserting an element at the rear end involves incrementing the rear pointer.**
- **Deleting an element from the rear end involves decrementing the rear pointer.**
- **NOTE: However, front and rear pointers need to be maintained, such that they remain within the bounds of indexing of 0 to size - 1 of the circular array.**
- **Initially, the deque is empty, so front and rear pointers are initialized to -1, denoting that the deque contains no element.**

Enqueue_front operation
Steps:

- **If the array is full, the data can't be inserted.**
- **If there are not any components within the Deque(or array) it means the front is equal to -1, increment front and rear, and set carr[front] as data.**
- **Else decrement front and set carr[front] as data.**







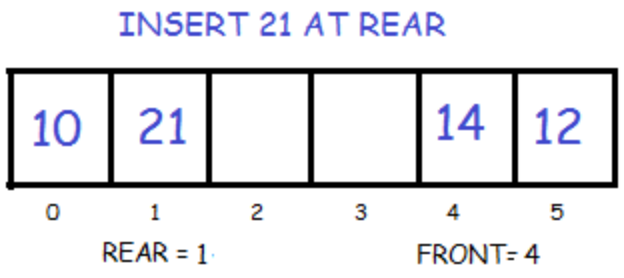
Pseudocode:

```
function enqueue_front(data)

        //      Check if deque is full
        if (front equals 0 and rear equals size-1) or (front equals rear+1)
                print ("Overflow")
                return

        /*
                Check if the deque is empty, insertion of an element from the front or rear will be
equivalent.
                So update both front and rear to 0.
        */
        if front equals -1
                front = 0
                rear = 0
                carr[front] = data
                return

        //      Otherwise check if the front is 0
        if front equals 0
                /*
                        Updating front to size-1, so that front remains within the bounds of the circular
array.
                */

                front = size-1
        else
                front = front-1

        carr[front] = data
        return
```

Enqueue_rear operation

Steps:

- **If the array is already full then it is not possible to insert more elements.**
- **If there are not any elements within the Deque i.e. rear is equal to -1, increase front and rear and set carr[rear] as data.**
- **Else increment rear and set carr[rear] as data.**



Pseudocode:

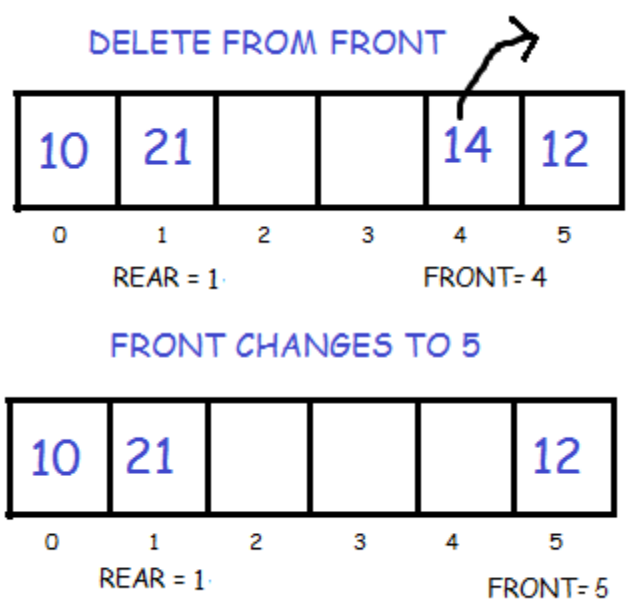
```
function enqueue_rear(data)

        //   Check if deque is full
        if (front equals 0 and rear equals size-1) or (front equals rear+1)
```

```
                print ("Overflow")
                return


        /*
                Check if the deque is empty, insertion of an element from the front or rear will be
equivalent.
                So update both front and rear to 0.
        */

        if front equals -1
                front = 0
                rear = 0
                carr[front] = data
                return

        //   Otherwise check if rear equals size-1

        if rear equals size-1
                /*
                        Updating rear to 0, so that rear remains within the bounds of the circular array.
                */
                rear = 0
        else
                rear = rear+1

        carr[rear] = data
        return
```

Dequeue_front operation

Steps:

- **If the Deque is empty, return.**
- **If there is only one element in the Deque, that is, front equals rear, set front and rear as -1.**
- **Else increment front by 1.**



Pseudocode:

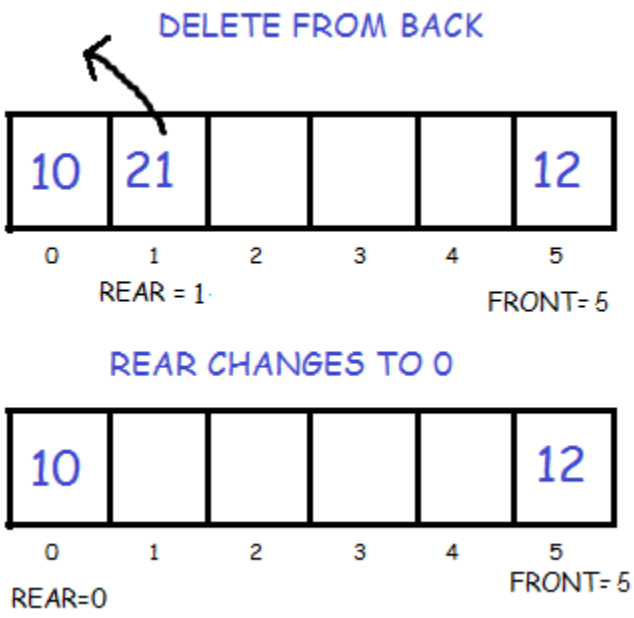
```
function dequeue_front()

        //   Check if deque is empty

        if front equals -1
                print ("Underflow")
                return

        /*
                Otherwise, check if the deque has a single element i.e front and rear are equal and will be
non-negative as
                the deque is non-empty.
        */
        if front equals the rear
                /*
                        Update front and rear back to -1, as the deque becomes empty.
```

```
                */
                front = -1
                rear = -1
                return

        //   If the deque contains at least 2 elements.

        if front equals size-1
                //   Bring front back to the start of the circular array.
                front = 0
        else
                front = front+1
        return
```

Dequeue_rear operation

Steps:

- **If the Deque is empty, return.**
- **If there's just one component within the Deque, that is, rear equals front, set front and rear as -1.**
- **Else decrement rear by one.**



Pseudocode:

```
function dequeue_rear()

        //   Check if deque is empty
        if front equals -1
                print ("Underflow")
                return

        /*
                Otherwise, check if the deque has a single element i.e front and rear are equal and will be
non-negative as
                the deque is non-empty.
        */
        if front equals the rear
                /*
                        Update front and rear back to -1, as the deque becomes empty.
                */
                front = -1
                rear = -1
                return

        //   If the deque contains at least 2 elements.

        if rear equals 0
                //   Bring rear back to the last index i.e size-1 of the circular array.
                rear = size-1
        else
                rear = rear-1
        return
```

Front operation

Steps:

- **If the Deque is empty, return.**
- **Else return carr[front].**

Pseudocode:

```
function front()

        //  Check if deque is empty
        if front equals -1
                print ("Deque is empty")
                return

        //  Otherwise return the element present at the front end
        return carr[front]
```

Rear operation

Steps:

- **If the Deque is empty, return.**
- **Else return carr[rear].**

Pseudocode:

```
function rear()

        //   Check if deque is empty
        if front equals -1
                print ("Deque is empty")
                return

        //   Otherwise return the element present at the rear end
        return carr[rear]
```

Is Empty operation

Steps:

- **If front equals -1, the Deque is empty, else it's not.**

Pseudocode:

```
function isEmpty()

        //   Check if front is -1 i.e no elements are present in deque.
        if front equals -1
                return true
        else
                return false
```

Is Full operation

Steps:

- **If front equals 0 and rear equals size - 1, or front equals rear + 1, then the Deque is full, else it's not. Here "size" is the size of the circular array.**

Pseudocode:

```
function isFull()

        /*
                Check if the front is 0 and rear is size-1 or front == rear+1, in both cases, we cannot move
front and rear
                to perform any insertions
        */

        if (front equals 0 and rear equals size-1) or (front equals rear+1)
                return true
        else
                return false
```

Time complexity of various operations

Let 'n' be the number of elements in the deque. The time complexities of deque operations in the worst case can be given as:

Operations	Time Complexity
Enqueue_front(data)	O(1)
Enqueue_rear(data)	O(1)
Dequeue_front()	O(1)
Dequeue_rear()	O(1)
Front()	O(1)
Rear()	O(1)
isEmpty()	O(1)
isFull()	O(1)

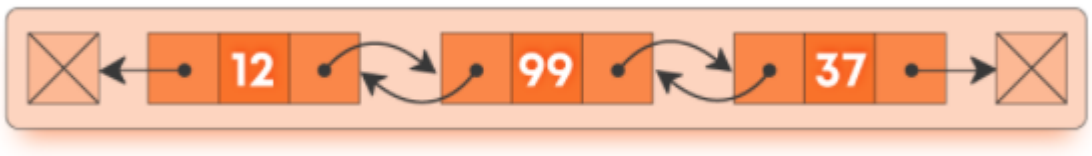
Applications of Deques

- **Since deques can be used as stack and queue, they can be used to perform undo-redo operations in software applications.**
- **Deques are used in the A-steal job scheduling algorithm that implements task scheduling for multiple processors (multiprocessor scheduling).**
- **They are also helpful in finding max/min values of all subarrays of size k in the array in O(n) time, where n is the size of the array.**

Doubly Linked Lists Notes

Introduction

Doubly Linked Lists contain an extra pointer pointing towards the previous node (called the previous pointer) in addition to the pointer pointing to the next node (called the next pointer).
The advantage of using doubly linked lists is that we can navigate in both directions.
A node in a singly linked list can not be removed unless we have the predecessor node. But in a doubly-linked list, we don't need access to the predecessor node.



Operations on doubly linked lists

Insertion Operations

- **insertAtBeginning(data): Inserting a node in front of the head of a linked list.**
- **insertAtEnd(data): Inserting a node at the tail of a linked list.**
- **insertAtIdx(idx, data): Inserting a node at a given index.**

Deletion Operations

- **deleteFromBeginning: Deleting a node from the front of a linked list.**
- **deleteFromEnd: Deleting a node from the end of a linked list.**
- **deleteFromIdx(idx): Deleting a node at a given index.**

Implementation of doubly LinkedList

Doubly Linked Lists contain a head pointer that points to the first node in the list (head is null of the list is empty).
Each node in a doubly-linked list has three properties: data, previous(pointer to the previous node), next(pointer to the next node).

- **Insert at beginning**

```
function insertAtBeginning(data)
            /*
                create a new node : newNode
                set newNode's data to data
            */
            newNode.data =  data

            //   If list is empty, set head as newNode
            if head is null
                  head = newNode
                  return head

            newNode.next = head
            head.previous = newNode
            head = newNode
            return head
```

- **Insert at end**

```
function insertAtEnd(data)
            /*
               create a new node : newNode
               set newNode's data to data
            */
            newNode.data =  data

            //   If list is empty, set head as newNode
            if head is null
                  head = newNode
                  return head

            /*
               Otherwise, create a cur node pointer and keep moving it
               until it reaches the last node of the list
            */

            cur = head
            while cur.next is not null
                  cur = cur.next

            /*
                Now cur points to the last node of the linked list; set the next
                pointer of this node to the newNode
            */
            cur.next = newNode
            newNode.previous = cur
            return head
```

- **Insert at given index (idx will be 0 indexed)**

```
function insertAtGivenIdx(idx, data)
            /*
               create a new node : newNode
               set newNode's data to data
            */

            newNode.data =  data
            //   call insertAtBeginning if idx = 0
            if idx == 0
                insertAtBeginning(data)
                return head
            count = 0
            cur = head

            while count < idx - 1 and cur.next is not null
                 count += 1
                 cur = cur.next
                 /*
                    If count does not reach (idx - 1), then the given index
                    is greater than the size of the list
                 */
                 if count < idx - 1
                      print "invalid index"
                      return head

                 /*
                   Otherwise setting the newNode next field as the address of the node present at position
idx
                 */

            nextNode = cur.next
            cur.next = newNode
            newNode.prev = cur
            if nextNode is not null
                nextNode.prev = newNode
                newNode.next = nextNode

            return head
```

- **Delete from beginning**

```
function deleteFromBeginning()
            //   if the head is null, return
            if the head is null
                        print "Linked List is Empty"
                        return head

            temp  = head
            head = head.next
            head.prev = null
            delete temp
            return head
```

- **Delete from end.**

```
function deleteFromEnd()

           //   list is empty if the head is null
           if the head is null
                      print  "list is Empty"
                      return head

           /*
                      Keep a cur pointer and let it point to the head; move the cur
                      pointer till cur.next is not equal to  null
           */

           cur = head
           while cur.next is not equal to null
                      cur = cur.next

           prevNode = cur.prev
           prevNode.next = null
           delete cur
           return head
```

- **Delete from given index (idx will be 0 indexed)**

```
function deleteFromGivenIdx(idx)

           //   call deleteFromBeginning if idx = 0
           if idx == 0
                       deleteFromBeginning()
                       return head

           count = 0
           temp = head

          while count < idx - 1 and temp is not equal to null
                      count++
                      temp = temp.next

          if temp = null or temp.next is equal to null
                      print "Invalid Index"
                      return head

            nextNode = temp.next
            prevNode = temp.prev
            prevNode.next = nextNode
            nextNode.previous = prevNode
            delete temp
            return head
```

Time Complexity of various operations

Let ‘n’ be the number of elements in the linked lists. The complexities of linked list operations with this representation can be given as:

Operations	Time Complexity
insertAtBeginning(data)	O(1)
insertAtEnd(data)	O(n)
insertAtGivenIdx(idx, data)	O(n)
deleteFromBeginning()	O(1)
deleteFromEnd()	O(n)
deleteFromGivenIdx(idx)	O(n)

Applications of Doubly Linked Lists

- **It is used by web browsers for backward and forward navigation of web pages**
- **LRU (Least Recently Used) / MRU (Most Recently Used) Cache is constructed using Doubly Linked Lists.**
- **Used by various applications to maintain undo and redo functionalities.**
- **In Operating Systems, a doubly-linked list is maintained by the thread scheduler to keep track of processes that are being executed at that time.**

Circular Linked Lists Notes

Introduction

Circular linked lists are just linked lists where the next pointer of the last node is pointing to the first node of the list. There is no head node.

Why circular linked lists?

The advantage of using a circular linked list is that when we want to traverse in only one direction and move to the first node cyclically, we don’t need to store additional pointers to mark the first and the last node. Typical usage of this concept is to implement queues using one pointer.

Operations on circular linked lists

- **insertAtBeginning(data): Inserting a node in front of the head of a linked list.**
- **deleteFromBeginning: Deleting a node from the front of a linked list.**
- **display: Display the contents of the linked list**

Implementation of circular linked list

Circular Linked Lists will have one pointer: head

Each node in a circular-linked list has two properties: data, next(pointer to the next node).

- **Insert at beginning**

```
function insertAtBeginning(data)
            /*
               create a new node : newNode
               set newNode's data to data
            */
            newNode.data =  data

            //   If list is empty, set head as newNode
            if head is null
                        head = newNode
                        head.next = head
                        return head
            // traverse to the end of the circular list

            temp = head
            while temp.next != head
                         temp = temp.next
```

```
            // set the next of temp as newNode and return head
            newNode.next = head
            temp.next = newNode
            head = newNode
            return head
```

- **Delete from beginning**

```
function deleteFromBeginning()
            //   if the head is null, return
            if head is null
                        print "Linked List is Empty"
                        return head
            if head.next == head
                        head = NULL
                        return head
            //   traverse to the end of the circular list

            temp = head
            while temp.next != head
                        temp = temp.next
            //   set the next of temp as next of head
            temp.next = head.next
            delete head
            head = temp.next
            return head
```

- **Display**

```
function display()
            /*
               create a new node : newNode
               set newNode's data to data
            */

            //   If the list is empty, print nothing
            if the head is null
                        print "list empty"
                        return
            // traverse to the end of the circular list until the head is encountered

            temp = head
            while temp.next != head
                        print (temp.data)
                        temp = temp.next
            print(temp.data)

            return
```

Time Complexity of various operations

Let 'n' be the number of elements in the linked lists. The complexities of linked list operations with this representation can be given as:

Operations	Time Complexity
insertAtBeginning(data)	O(n)
deleteFromBeginning()	O(n)
display()	O(n)

Applications of Circular Linked Lists

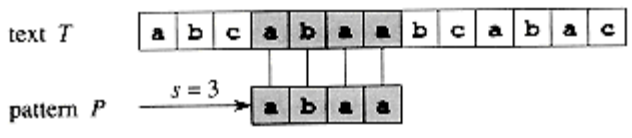
- **They are used for implementations of data structures like Fibonacci heap where a circular doubly linked list is used.**
- **They can be used to implement circular queues that have applications in CPU scheduling algorithms like round-robin scheduling.**
- **They are used to switch between players in multiplayer games.**

String Algorithms Notes

In this section we will primarily talk about two string searching algorithms, Knuth Morris Pratt algorithm and Z - Algorithm.

Introduction

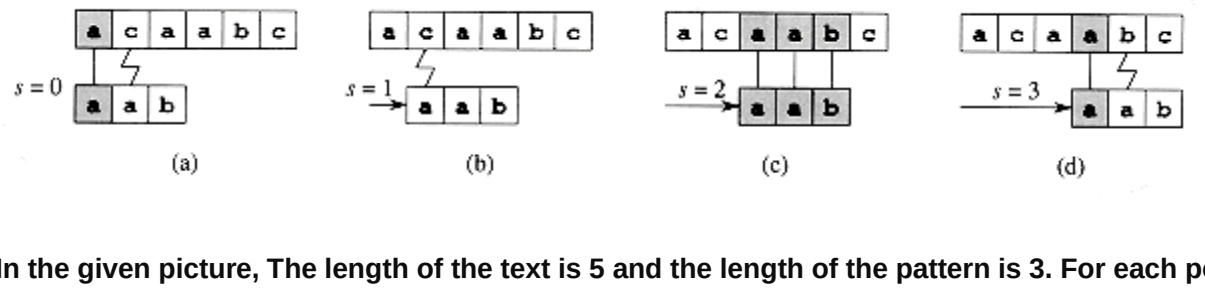
Suppose you are given a string text of length n and a string pattern of length m. You need to find all the occurrences of the pattern in the text or report that no such instance exists.



In the above example, the pattern “abaa” appears at position 3 (0 indexed) in the text “abcabaabcabac”.

Naive Algorithm

A naive way to implement this pattern searching is to move from each position in the text and start matching the pattern from that position until we encounter a mismatch between the characters or say that the current position is a valid one.



In the given picture, The length of the text is 5 and the length of the pattern is 3. For each position from 0 to 3, we choose it as the starting position and then try to match the next 3 positions with the pattern.

Naive pattern matching

- **For each i from 0 to N - M**
- **For each j from 0 to M - 1, try to match the jth character of the pattern with (i + j)th character of the string text.**
- **If a mismatch occurs, skip this instance and continue to the next iteration.**
- **Else output this position as a matching position.**

Pseudocode:

```
function NaivePatternSearch(text, pattern)
        //   iterate for each candidate position
        for i from 0 to text.length - pattern.length

                // boolean variable to check if any mismatch occurs
                match  = True

                for j from 0 to pattern.length - 1
                        //   if mismatch make match = False
                        if text[i  + j] not equals pattern[j]
                                match = False

                //   if no mismatch print this position
                if match == True
                        print the occurrence  i
        return
```

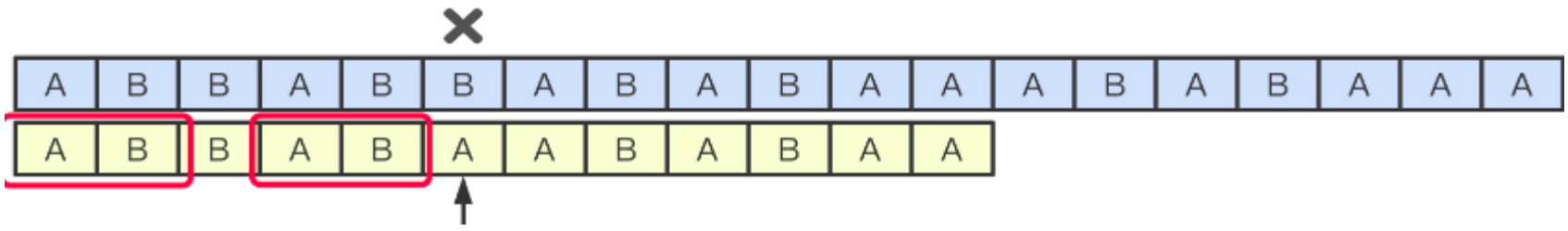
Knuth Morris Pratt Algorithm

We first define the prefix function of the string - The prefix function of a string s is an array lps of length same as that of s such that lps[i] stores the information about the string s[0.. i]. It stores the length of the maximum prefix that is also the suffix of the substring s[0..i].

For example :

For the pattern “AAAABAA”,
lps[] is [0, 1, 2, 0, 1, 2, 3]

lps[0] is 0 by definition. The longest prefix that is also the suffix of string s[0..1] which is AA is 1. (Note that we are only considering the proper prefix and suffix). Similarly, For the whole string AAABAAA it is 3, hence the lps[6] is 3.



Algorithm for Computing the LPS array.

- **We compute the prefix values lps[i] in a loop by iterating from 1 to n - 1.**
- **To calculate the current value lps[i] we set the variable j denoting the length of best suffix for i - 1. So j = lps[i - 1].**
- **Test if the suffix of length j + 1 is also a prefix by comparing s[j] with s[i]. If they are equal then we assign lps[i] = j + 1 else reduce j = lps[j - 1].**
- **If we have reached j = 0 we assign lps[i] = 0 and continue to the next iteration .**

Pseudocode:

```
function PrefixArray(s)
        n = s.length;
        //   initialize to all zeroes
        lps = array[n];

        for i from 1 to n - 1
                j = lps[i-1];
                //   update j untill s[i] becomes equal to s[j]
                while j greater than 0 && s[i] no equal to s[j]
                        j = lps[j-1];

                //   if extra character matches increase j
                if s[i] equal to s[j]
                        j += 1;

                //   update lps[i]
                lps[i] = j;

        //   return the array
        return lps
```

Algorithm for searching the pattern.

Now consider a new string S' = pattern + '#' + 'text' where + denotes the concatenation operator.

Now, what is the condition that pattern appears at position [i - M + 1... i] in the string text. The lps[i] should be equal to M for the corresponding position of i in S'. Note that lps[i] cannot be larger than M because of the '#' character.

- **Create S' = pattern + '#' + 'text'**
- **Compute the lps array of S'**
- **For each i from 2*M to M + N check the value of lps[i].**
- **If it is equal to M then we have found an occurrence at the position i - 2*M in the string text.**

Pseudocode:

```
function StringSearchKMP(text, pattern)
        //   construct the new string
        S' = pattern + '#' + text

        //   compute its prefix array
        lps = PrefixArray(S')
        N = text.length
        M = pattern.length

    for i from 2*M to M + N
                //   longest prefix match is equal to the length of pattern
                if lps[i] == M
                        //   print the corresponding position
                        print the occurrence i - 2*M
        return
```

Z - Algorithm

We first define the Z function of the string - The Z function of a string S is an array Z of length same as that of S such that Z[i] denotes the length of the largest prefix that matches from the substring starting at position I.

For example :

For the pattern "AAAABAA",
Z[] is [0, 3, 2, 1, 0, 2, 1]

Z[0] is 0 by definition. The longest prefix that is also the prefix of string s[1..6] which is "AAABAA" is 3(This is equal to "AAA"). Similarly, For the whole string AAABAAA it is 1, hence the Z[6] is 1 since s[6.. 6] is 'A' and that is the longest possible prefix we can match.

Algorithm for Computing the Z array.

The idea is to maintain an interval [L, R] which is the interval with max R such that [L, R] is a prefix substring (substring which is also prefix).

- **if i > R, no larger prefix-substring is possible.**
- **Compute the new interval by comparing S[0] to S[i] i.e. string starting from index 0 i.e. from start with substring starting from index i and find z[i] using z[i] = R - L + 1.**
- **Else if, i ≤ R, [L, R] can be extended to i.**
- **For k = i - L, Z[i] ≥ min(Z[k] , R - i + 1).**
- **If Z[k] < R - i + 1, no longer prefix substring s[i] exist.**
- **Else Z[k] ≥ R - i + 1, then there can be a longer substring.**
- **update [L, R] by changing L = i and changing R by matching from S[R+1]**

Pseudocode:

```
function ZArray(s)
        //   initialize to all zeroes
        Z = array[n];
        //   set the current window to the first character
        l = 0
        r = 0

    for i from 1 to n - 1
                //   first case i <= r
                if i <= r
                        z[i] = min (r - i + 1, z[i - l]);

                //   increase prefix length while they are matching
                while i + z[i] < n and s[z[i]] == s[i + z[i]]
                        z[i] += 1;

                //   update the window if i + z[i] crosses the window
                if i + z[i] - 1 > r
                        l = i
                        r = i + z[i] - 1;
        //   return the array
        return z
```

Algorithm for searching the pattern.

Now consider a new string S' = pattern + '#' + 'text' where + denotes the concatenation operator.

Now, what is the condition that pattern appears at position [i. ...i + M - 1] in the string text. The Z[i] should be equal to M for the corresponding position of i in S'. Note that Z[i] cannot be larger than M because of the '#' character.

- **Create S' = pattern + '#' + 'text'**
- **Compute the lps array of S'**
- **For each i from M + 1 to N + 1 check the value of lps[i].**
- **If it is equal to M then we have found an occurrence at the position i - M - 1 in the string text.**

Pseudocode:

```
function StringSearchZ_Algo(text, pattern)
        //   construct the new string
```

```
        S' = pattern + '#' + text

        //   compute its prefix array
        Z = ZArray(S')
        N = text.length
        M = pattern.length

    for i from M + 1 to  N + 1
                //   longest prefix match is equal to the length of pattern
                if Z[i] == M
                        //   print the corresponding position
                        print the occurrence i - M - 1
        return
```

Time Complexities of string algorithms

Here ‘N’ is the total length of the pattern and ‘M’ is the length of the pattern we need to search.

Algorithm	Time Complexity
Naive Pattern Matching	O(N * M)
KMP Algorithm (including calculation of lps array)	O(N + M)
Z – Algorithm (including calculation of Z array)	O(N + M)

Applications

- **Used in plagiarism detection between documents and spam filters.**
- **Used in bioinformatics and DNA sequencing to match DNA and RNA patterns**
- **Used in various editors and spell checkers to correct the spellings**