Array Notes

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.

Accessing array elements

- The elements of the array are accessed by using their index. The index of an array of size N ranges from **0** to **N-1**.

For example: Accessing element at index 5: Array[5] -> this is the 6th element in the array.

- Every array is identified by its **base address** i.e the location of the first element 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 array, so address of element at index 4 will be 200 + 4 * (sizeof(int)) = 216.

Where can arrays be used?

- **Arrays** should be used where the number of elements to be stored is already known.
- **Arrays** are commonly **used** in computer programs to organize data so that a related set of values **can** be easily **sorted** or **searched**.
- Generally, when we require **very fast access times**, we usually prefer arrays since they provide O(1) access times.
- Arrays work well when we have to **organize data in multidimensional format**. We can declare arrays of as many dimensions as we want.
- If the index of the element to be modified is known beforehand, it can be efficiently modified using arrays due to **quick access time** and **mutability**.

Disadvantages of arrays

- Since **arrays** are **fixed-size** data structures you cannot dynamically alter their sizes. It creates a problem when the number of elements the array is going to store is not known beforehand.
- **Insertion** and **Deletion** in arrays are difficult and costly since the elements are stored in contiguous memory locations, hence, we need to shift the elements to create/delete space for elements.
- If more memory is allocated than required, it leads to the **wastage of memory** space and **less allocation of memory** also leads to a problem.

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.
- **Inserting elements:** Insertion of elements at the end of the array (at the index located to the right of the last element and there is still available space) takes **O(1)** time.

Insertion of elements at the beginning or at any index of the array involves moving elements to the right if there is available space.

- If we want to insert an element at index i, all the elements starting from index i need to be shifted to the right by one position. Thus, the time complexity for inserting an element at index i is **O(N - i)**.
- Inserting an element at the beginning of the array involves moving all elements by one position to their right, if there is available space, and takes **O(N)** time.
- **Finding elements:** Finding an element in an array takes **O(N)** time in the worst case, where N is the size of the array, as you may need to traverse the entire array.
- **Deleting elements:** Deletion of elements from the end of the array takes **O(1)** time.

Deleting elements from the beginning or at any index of the array involves moving elements to the left.

- If we want to delete an element at index i, all the elements starting from index (i + 1) need to be shifted to the left by one index. Thus, the time complexity for deleting an element at index i is **O(N - i)**.
- Deleting an element from the beginning involves moving all elements starting from index 1 to left by one position, and takes **O(N)** time.

Previous
Next

Mutli-Dimensional Arrays Notes

Multi-Dimensional 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.

Previous
Next

Strings Notes

Strings

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.

Previous
Next
Recursion and Backtracking Notes

Recursion and Backtracking

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:

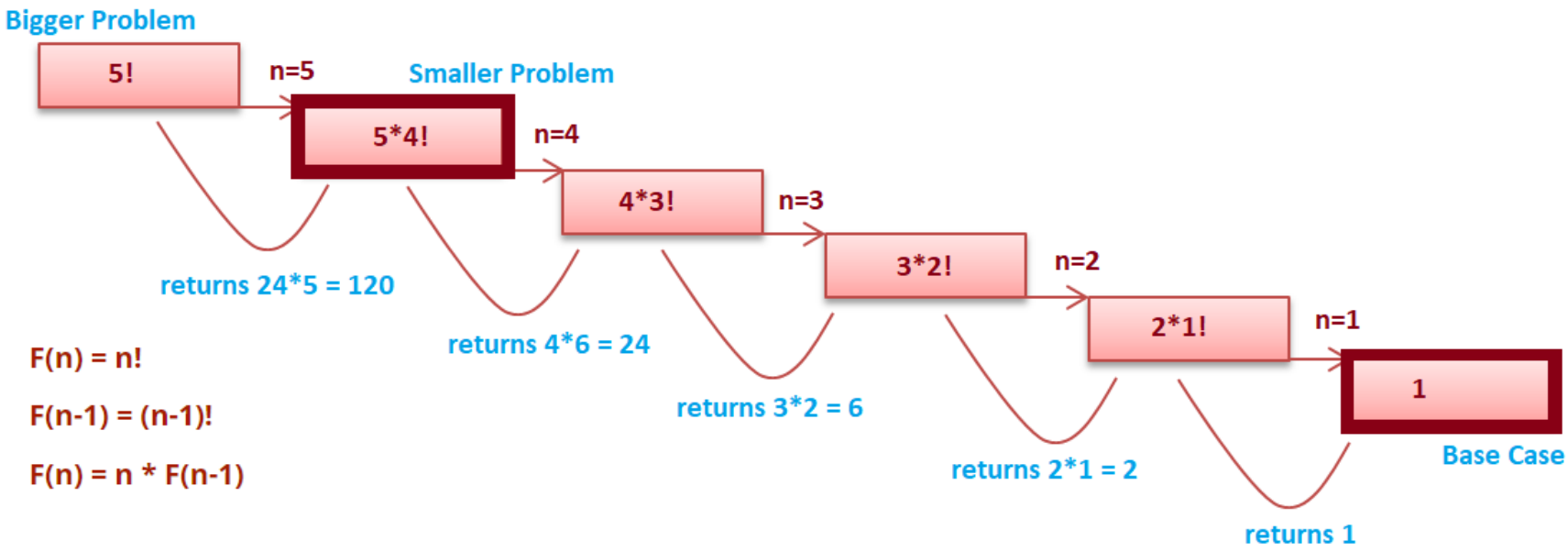
- **Base case: A recursive function must have a terminating condition at which the process will stop calling itself. Such a case is known as the base case. In the absence of a base case, it will keep calling itself and get stuck in an infinite loop. Soon, the recursion depth* will be exceeded and it 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.**
- **Small calculation (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 ;



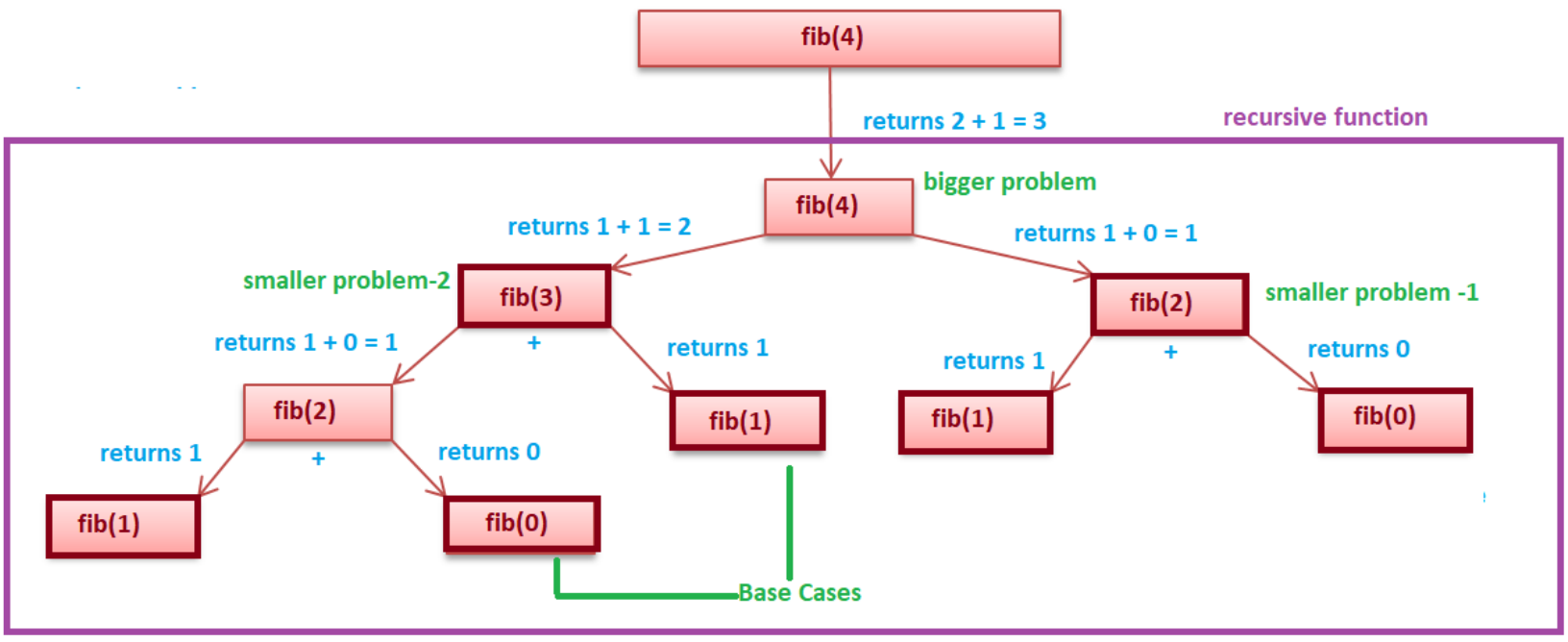
In recursion, the idea is that we represent a problem in terms of smaller problems. We know that 5! = 5 * 4!. Let's assume that recursion will give us an answer of 4!. Now to get the solution to our problem will become 4 * (the answer of the recursive call).

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.

Let n = 4



F(n) = F(n-1) + F(n-2)

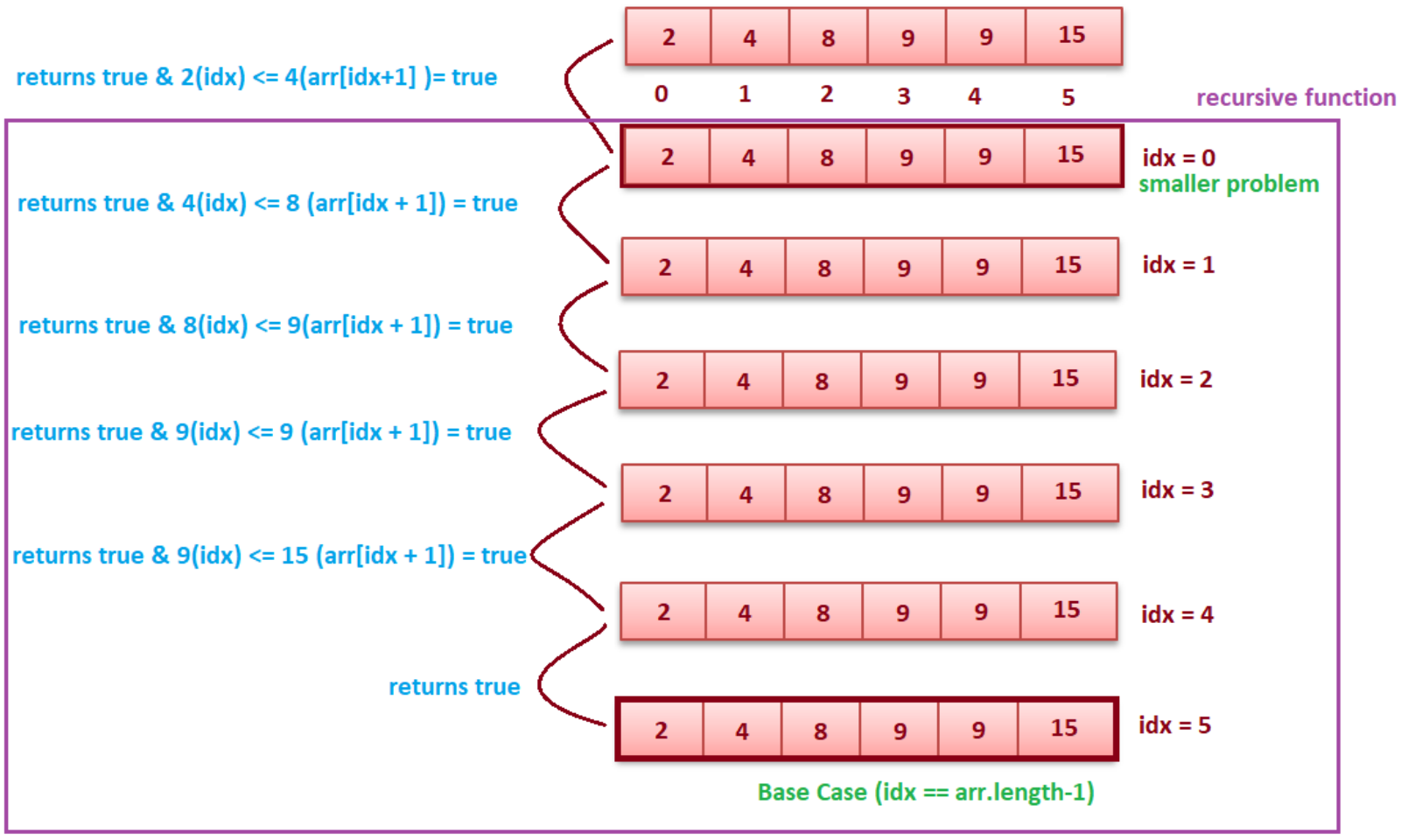
As you can see from the above fig and recursive equation that the bigger problem is dependent on 2 smaller problems. Depending upon the question, the bigger problem can depend on N number of smaller problems.

```
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
```

Binary Search with Recursion

Binary Search: Search in a sorted array by repeatedly dividing the array into half and searching in only one half.
You are given a target element X and a sorted array. You need to check if X is present in the array. Consider the algorithm given below:

- Compare X with the middle element in the array.
- **If** X is the same as the middle element, we return the index of the middle element.
- **Else if** X is greater than the mid element, then X can only lie in the right (greater) half subarray after the mid element. Thus, we apply the algorithm, recursively, for the right half.
- **Else if** X is smaller, the target X must lie in the left (lower) half. So we apply the algorithm, recursively, for the left half.

```
/*
     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)
```

Divide and Conquer

Divide and conquer is an algorithm design paradigm. A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Sorting Techniques Using Recursion/Divide and Conquer - Merge Sort

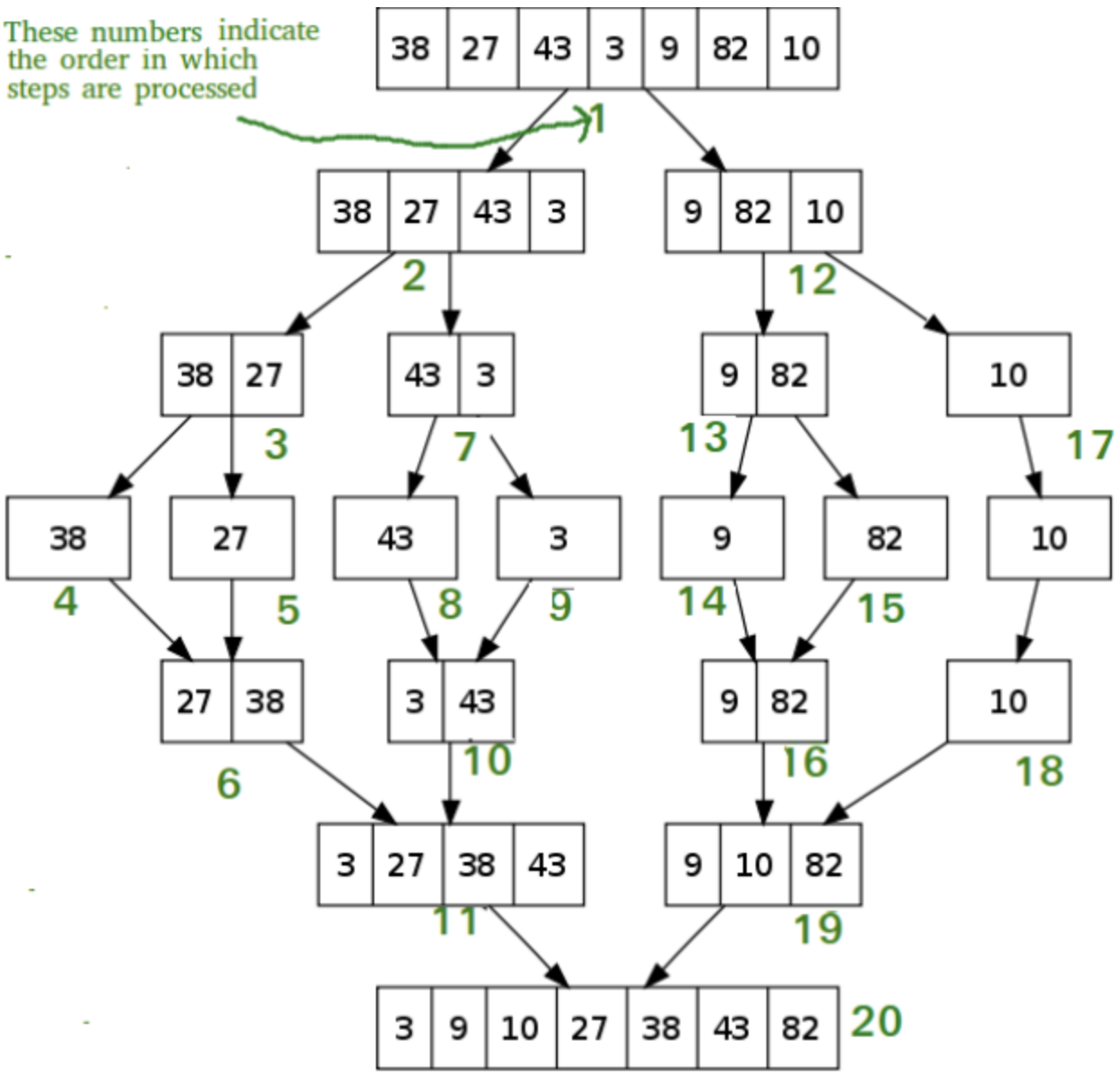
Merge sort requires dividing a given list into equal halves until it can no longer be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines/merges the smaller sorted lists keeping the new list sorted too.

- 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 and that is it's **not an in-place sorting** technique i.e. it creates a copy of the array and then works on that copy.

```
/*
          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++
```

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.



Sorting Techniques Using Recursion/Divide and Conquer - QuickSort

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

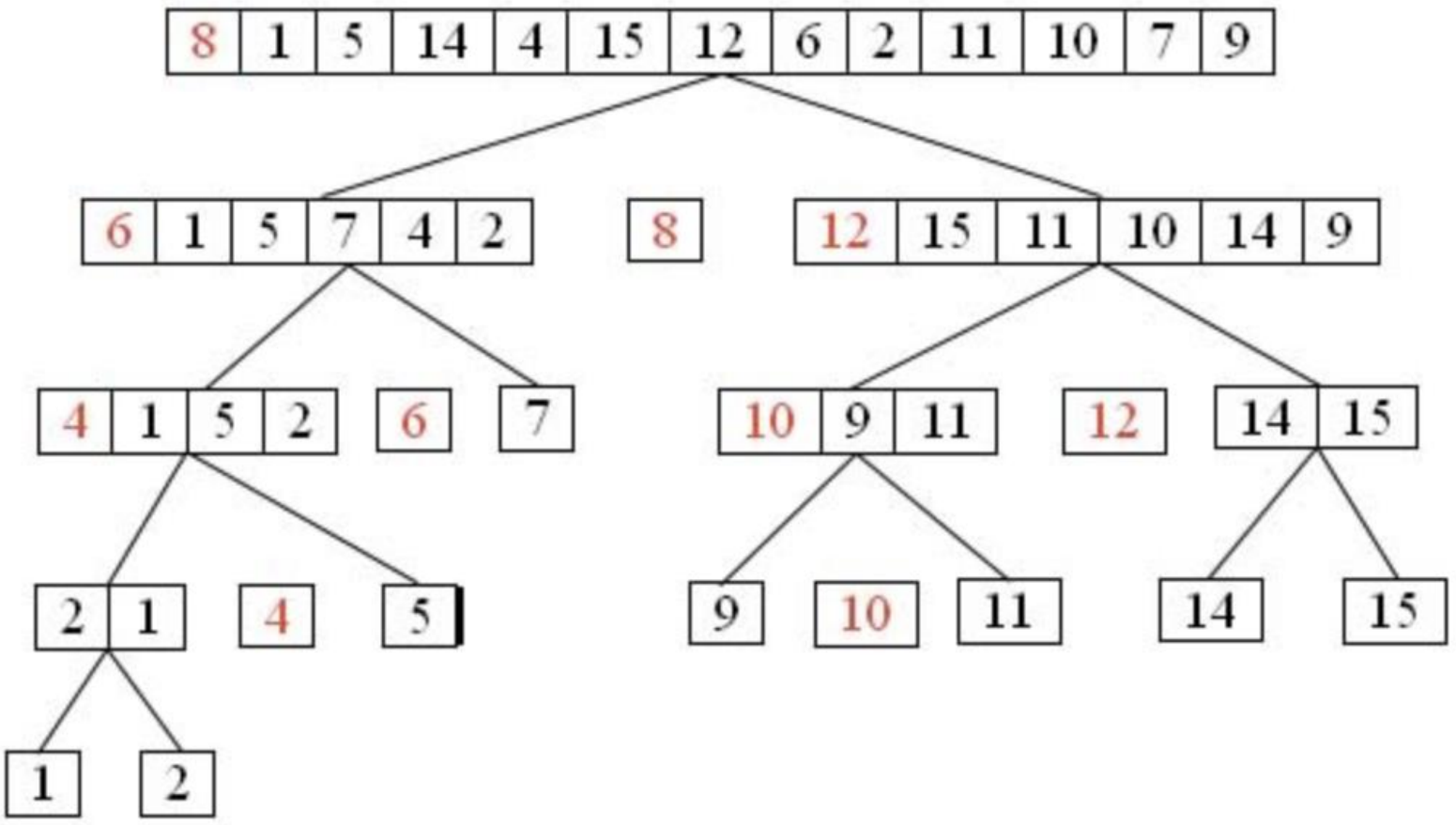
- **Divide:** The array is divided into subparts taking pivot as the partitioning point. The elements **smaller** than the pivot are placed to the **left** of the pivot and the elements **greater** than the pivot are placed to the **right** side.
- **Conquer:** The left and right sub-parts are again 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, that it is an **in-place** sorting technique.

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.

```
/*
                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)
```

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

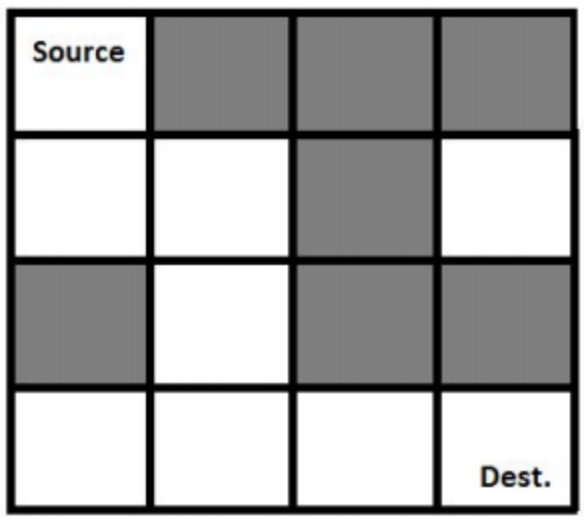
Backtracking is based on recursion, where the algorithm makes an effort to build a solution to a computational problem incrementally. Whenever the algorithm needs to choose between multiple possible alternatives, it simply tries **all possible options** for the solution recursively step-by-step. The 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.

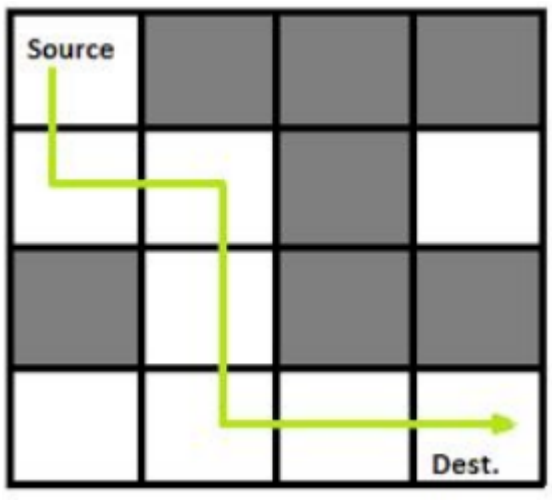
Rat Maze problem

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:

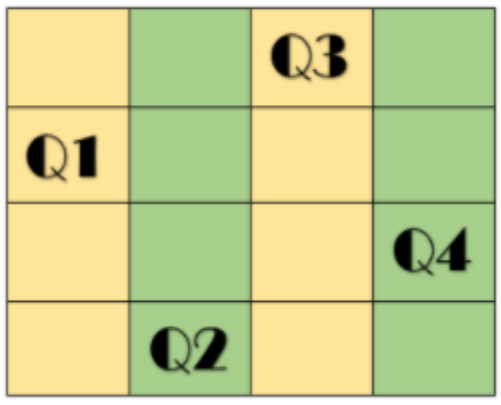
- If the destination point i.e **N** is reached, return true.
- Else
    - Check if the current position is a **valid** position i.e the position is within the **bounds** of the maze as well as it is a **free** position.
    - Mark the current position as 1, denoting that the position can lead to a possible 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.

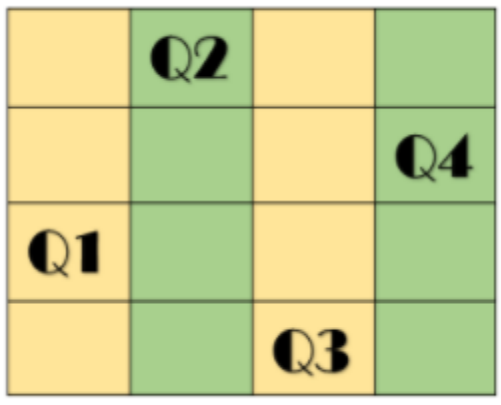
Pseudocode:

```
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 if
                                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.

Previous

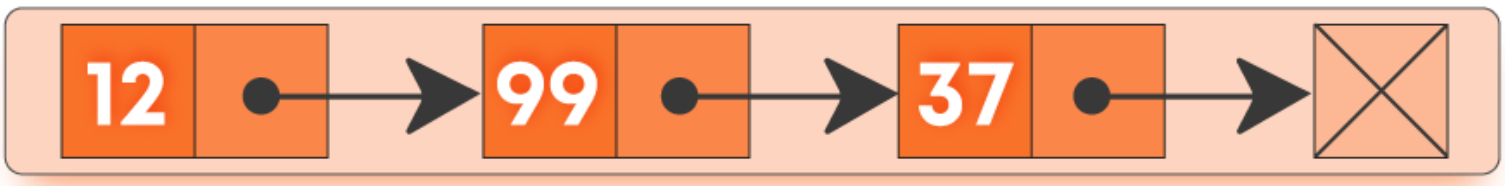
Next

LinkedLists Notes

Linked Lists

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

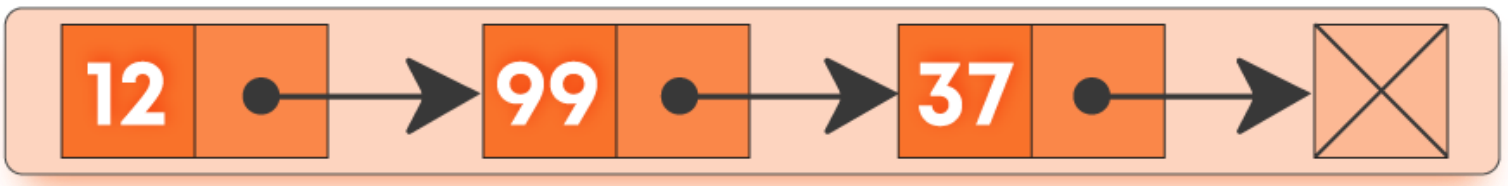
- A linked list is a **dynamic** data structure, which means the list can grow or shrink easily as the nodes are stored in memory in a non-contiguous fashion.

- 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.

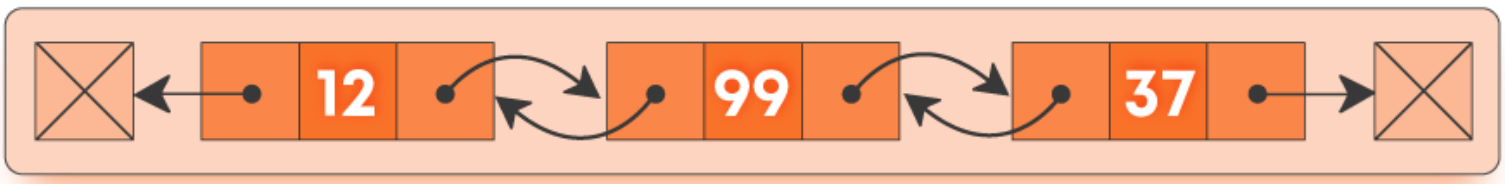
Types of Linked Lists

There are generally three types of linked list:

- **Singly:** Each node contains only one link which points to the subsequent node in the list.



- **Doubly:** It’s a two-way linked list as each node points not only to the next pointer but also to the previous pointer.



- **Circular:** There is no tail node i.e., the next field is never NULL in any node, and the next field for the last node points to the head node.

Singly Linked Lists

Operations on Singly Linked Lists

INSERTION OPERATION

- **Insertion at beginning**

```
function insertAtBeginning(data)
          /*
               create a new node : newNode
               set newNode's data to data
          */
          newNode.data =  data

          //   If the list is empty, set head as newNode
          if head is null
                head = newNode
                return head
          /*
               Otherwise set newNode as new head after setting its next
               pointer to current head.
          */

          newNode.next = head
          head = newNode
          return head
```

- **Insertion at end**

```
function insertAtEnd(data)
        /*
           create a new node : newNode
           set newNode's data to data
        */
        newNode.data =  data

        //   If the 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
        */

        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
        return head
```

- **Insertion at a given index (idx will be 0 indexed)**

```
function insertAtIdx(idx, data)
        /*
           create a new node : newNode
           set newNode's data to data
        */
        newNode.data =  data

        if (idx == 0)
              //   Case of insertion at beginning
              insertAtBeginning(data)
        else
               /*
                  Moving the cur node pointer till the given index and
                  maintaining a prev node where the node is to be
                  inserted
               */
               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
                    */
                    newNode.next = cur.next
                    /*
                       Updating the cur node next field as the address of
                       the newNode
                    */
                    cur.next = newNode
```

```
        return head
```

DELETION OPERATION

- **Deletion from beginning**

```
function deleteFromBeginning()

        //   If the list is empty, return null
        if head is null
            print “Linked List is empty”
            return null
        /*
            Otherwise set the new head to the node whose
            address is stored in the current head.
        */
        temp = head
        head = head.next
        delete temp
        return head
```

- **Deletion from end**

```
function deleteFromEnd()
        //   If the list is empty, return null
        if head is null
                print “Linked List is empty”
                return null

        //   Otherwise, check if the list contains a single node
        if head.next is null
                temp = head
                head = head.next
                delete head
                return head
        /*
            Otherwise, create a cur and prev node pointer and keep moving cur until it reaches the last
node
            of the list
        */
        cur = head.next
        prev = head
        while cur.next is not null
                prev = cur
                cur = cur.next


        /*
            Now cur points to the last node of the linked list and prev to the node behind it; set the
next
            pointer of prev node null.
        */
        prev.next = null
        delete cur
        return head
```

- **Deletion from the given index (idx will be 0 indexed)**

```
function deleteFromIdx(idx)

            if (idx == 0)
                //   Case of deletion from beginning
                deleteFromBeginning()
            else
                count = 0
                cur = head
                /*
                   Moving the cur node pointer until count is less than the index-1 from where the node
is
                   to be deleted
                */
                while count < idx - 1 and cur is not null
                count += 1
                cur = cur.next
```

```
            /*
               If the cur node reaches null or the tail of the list, then the given index is greater than
the
               size of the list
            */
            if cur is null or cur.next is null
                 print “invalid index”
                 return null
            /*
               Otherwise updating the next field of cur node to be the address of the node present in the
               next field of the node to be deleted
            */
            nextNode = cur.next
            cur.next = nextNode.next
            delete nextNode

            return head
```

SEARCH OPERATION

```
function search(data)
            //   Create a cur node pointer initialized to head of the linked list
            cur = head
            /*
               Traverse the linked list, and compare the data of cur and the data to be searched
            */

            while cur is not null
                 if cur.data equals data
                      return true
                 cur = cur.next

            //   If data is not found, return false
            return false
```

DISPLAY OPERATION

```
function display()
            //   Create a cur node pointer initialized to head of the linked list
            cur = head
            /*
               Traverse the linked list and print the data of the element represented by the cur pointer
            */
            while cur is not null
                 print cur.data
                 cur = cur.next
```

Time Complexity of various operations

Let ‘n’ be the number of nodes in the linked list. The time complexities of linked list operations in the worst case can be given as:

Operations	Time Complexity
Insertion at beginning(data)	O(1)
Insertion at end(data)	O(n)
Insertion at given index(idx,data)	O(n)
Deletion from beginning()	O(1)
Deletion from end()	O(n)
Deletion from given index(idx)	O(n)
Search(data)	O(n)
Display	O(n)

Applications of Linked Lists

- Linked Lists can be used to implement useful data structures like **stacks and queues**.
- Linked Lists can be used to implement **hash tables**, each bucket of the hash table can be a linked list.
- Linked Lists can be used to implement graphs (**Adjacency List representation** of graph).
- Linked Lists can be used in a refined way in implementing different **file systems** in one form or another.

Advantages of Linked Lists

- It is a dynamic data structure, which can grow or shrink according to the requirements.
- Insertion and deletion of elements can be done easily and does not require movement of all elements when compared to arrays.

- Allocation and deallocation of memory can be done easily during execution.
- Insertion at the beginning is a constant time operation and takes O(1) time, as compared to arrays where inserting an element at the beginning takes O(n) time, where n is the number of elements in the array.

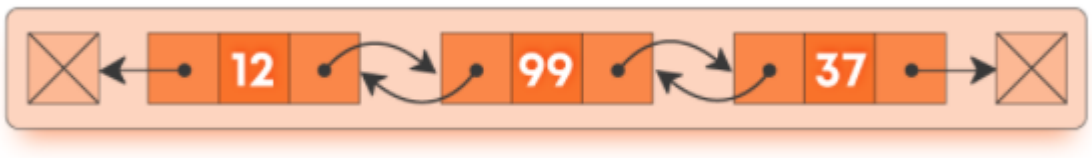
Disadvantages of Linked Lists

- Linked lists consume more memory as compared to arrays.
- As the elements in a linked list are not stored in a contiguous fashion in memory, so require more time to access the elements as compared to arrays.
- Appending an element to a linked list is a costly operation, and takes O(n) time, where n is the number of elements in the linked list, as compared to arrays that take O(1) time.

Doubly Linked Lists

Why doubly linked lists?

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)

Advantages of Doubly Linked Lists

- Doubly Linked Lists can be traversed in both directions.
- Deletion is easy in doubly linked lists if we know the address of the node to be deleted whereas in singly-linked lists we need to traverse the list to get the previous node.

- Reversing a doubly linked list is easier.

Disadvantages of Doubly Linked Lists

- We need to maintain an extra pointer for each node.
- Every operation in doubly linked lists requires updating of an extra pointer.

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.

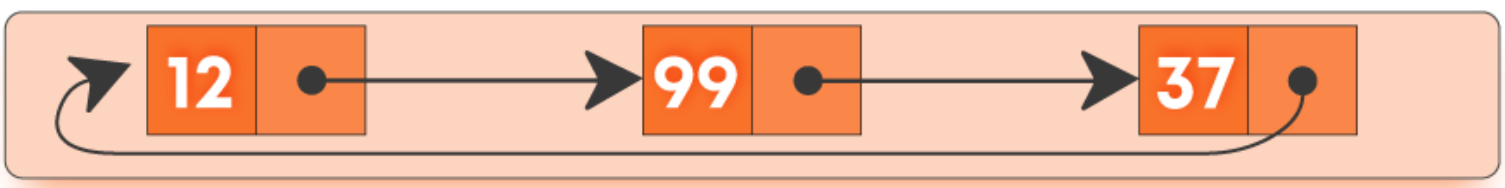
Circular Linked Lists

What are circular linked lists?

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)

Advantages of Circular Linked Lists

- We can start traversing the list from any node. We just have to keep track of the first visited node.
- They make the implementation of data structures like queues a lot easier and more space-efficient as compared to singly-linked lists.
- They can perform all the functionalities supported by singly-linked lists in addition to their own advantages.

Disadvantages of Circular Linked Lists

- Operations like insertion and deletion from the beginning become very expensive as compared to singly-linked lists.
- We need to maintain an extra pointer that marks the beginning of the list to prevent getting into an infinite loop.

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.

Previous

Next

Stacks & Queues Notes

Stacks & Queues

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**.

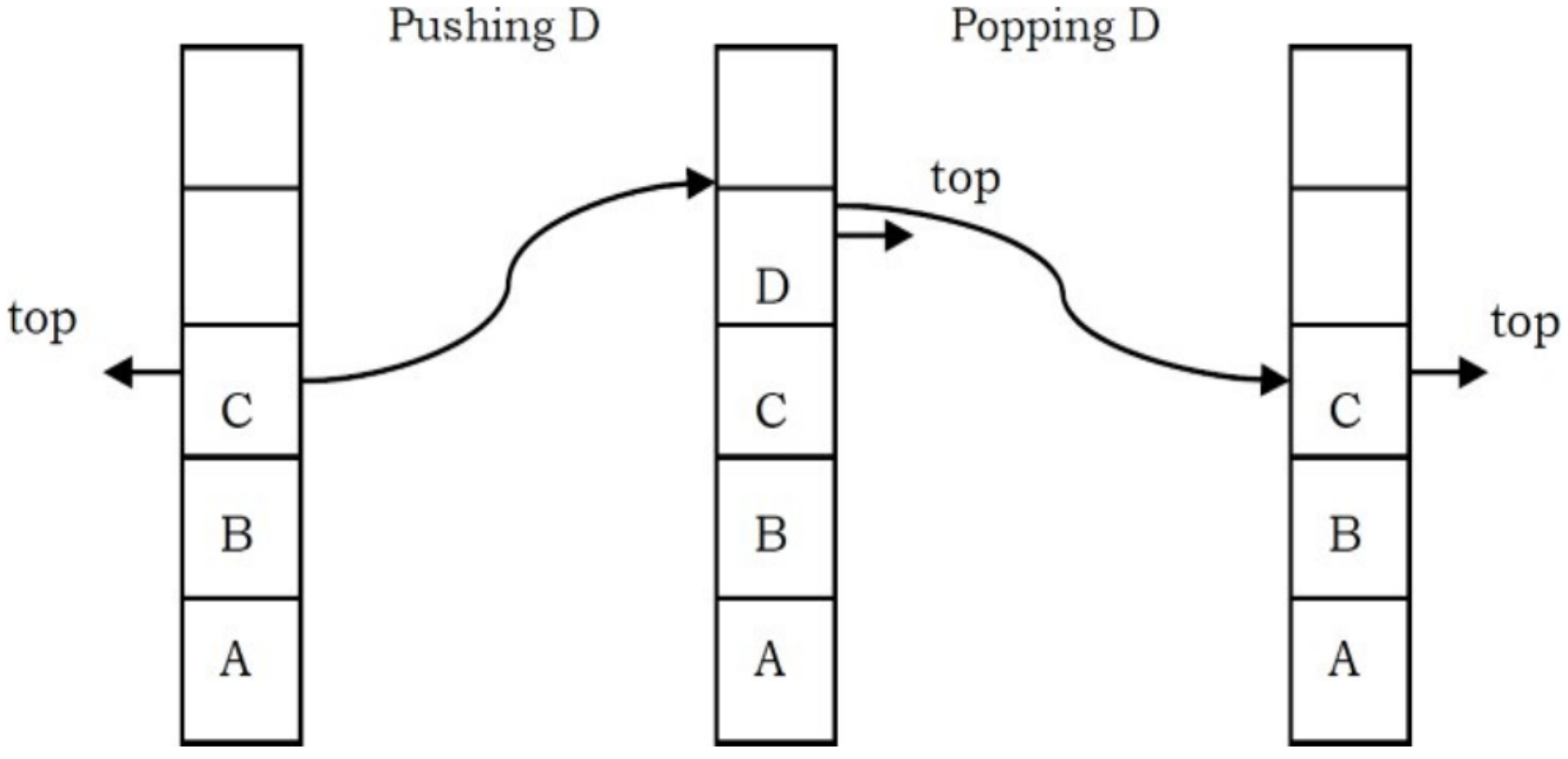
- **Insertion**: This is known as a **push** operation.
- **Deletion**: This is known as a **pop** operation.

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 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

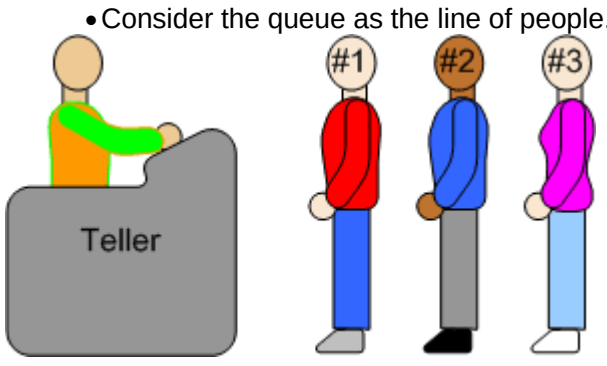
- Attempting the execution of an operation may sometimes cause an error condition, called an exception.
- Exceptions are said to be “thrown” by an operation that cannot be executed.
- Attempting the execution of pop() on an empty stack throws an exception called **Stack Underflow**.

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.**

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.

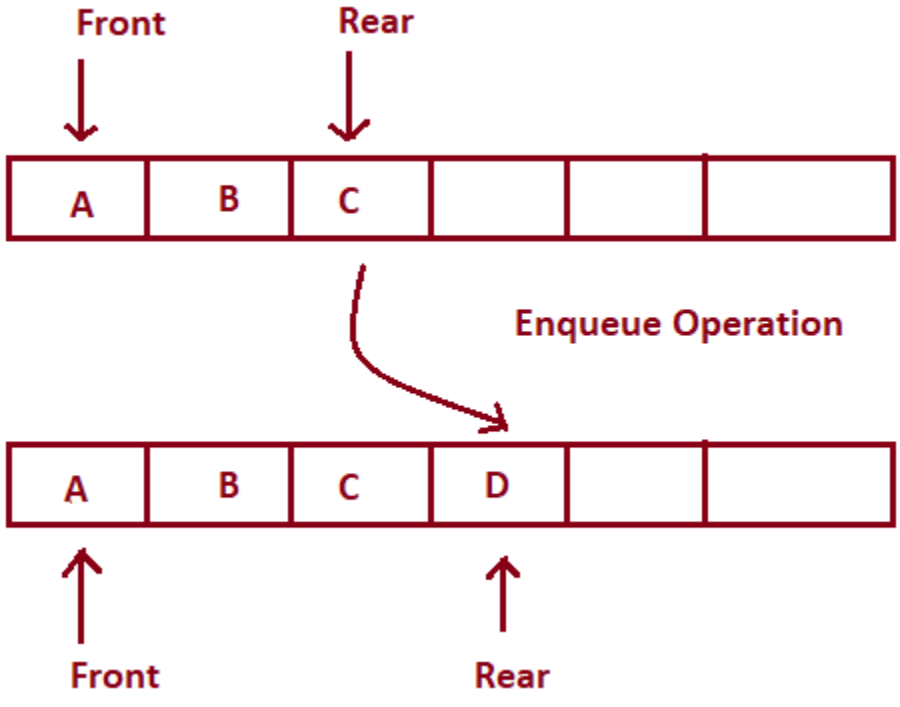


Here when we enter the queue / line, we stand at the end of line and the person who is at the front of the line is the one who will be served first.

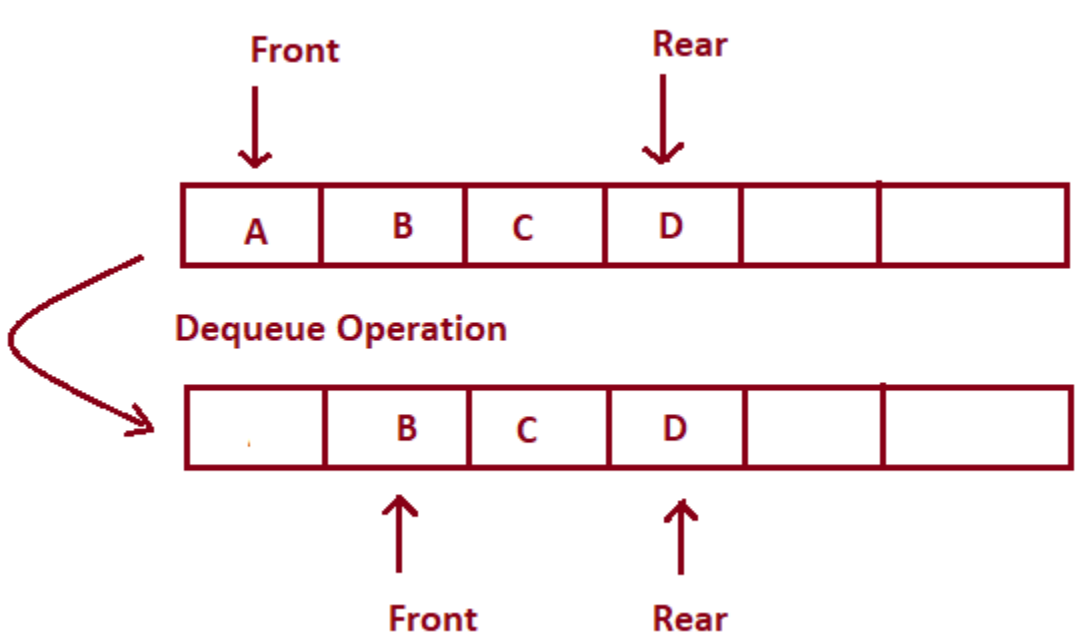
After the first person is served, he will exit and as a result the next person will come at the head of the line. Similarly, the head of the queue will keep exiting the line as we move towards the front / head of the line. Finally, we reach the front. we are served and we exit. This behaviour is useful 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. Concept of inserting an element into the queue is called Enqueue. Enqueueing an element when the queue is full is called **overflow**.
- **Deletion**: Adding elements at the front side. 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.

Implementation of Queues

Queues can be implemented using arrays, linked lists or stacks. The basic algorithm for implementing a queue remains the same. 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 centres
- 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.

Previous

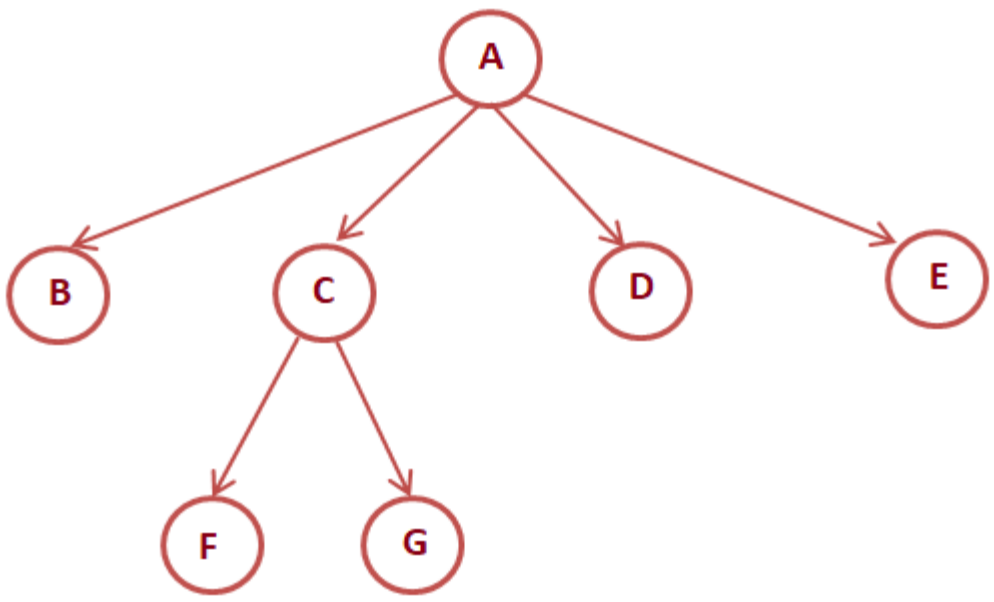
Next

Trees Notes

Trees

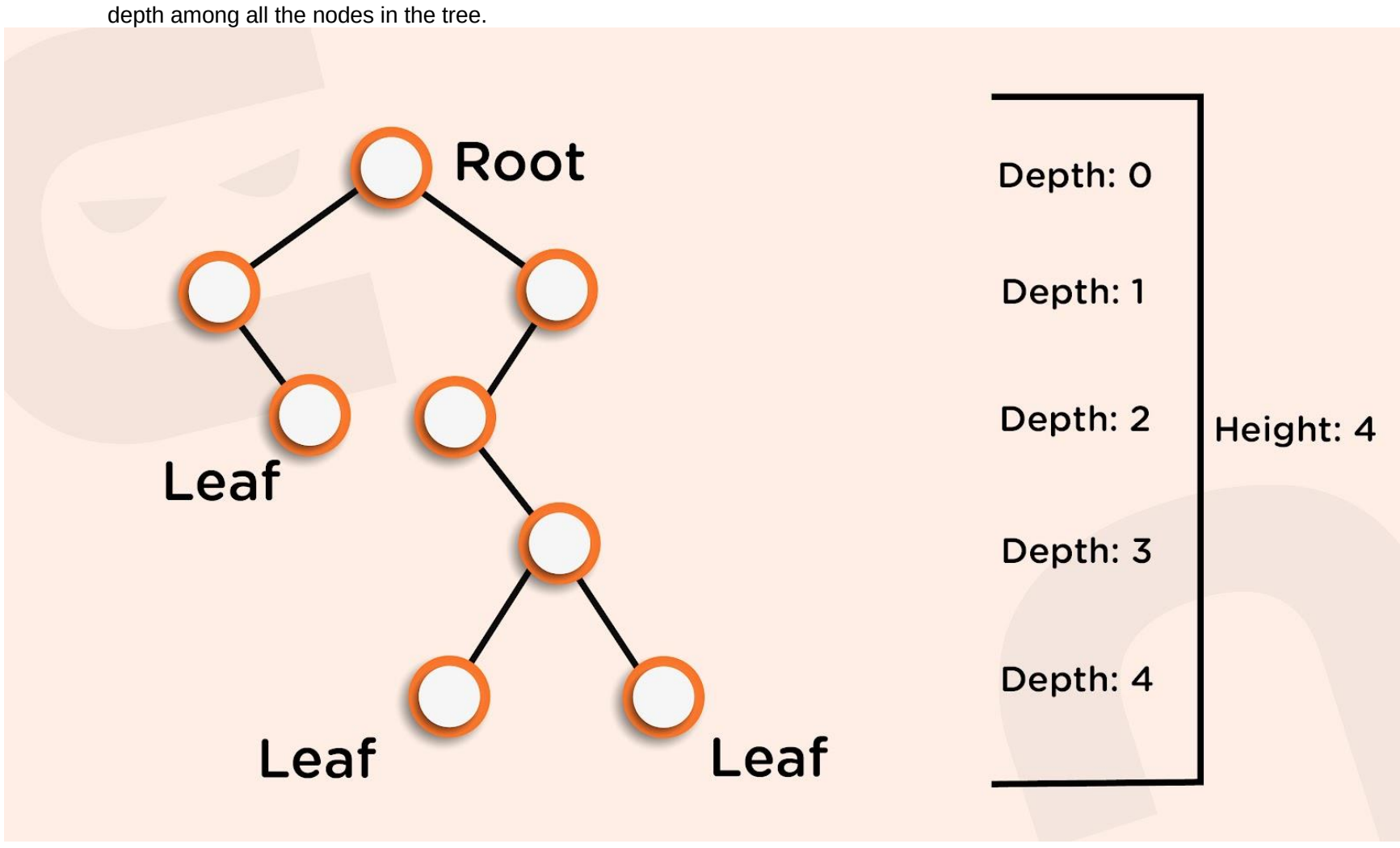
Introduction to Trees

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.

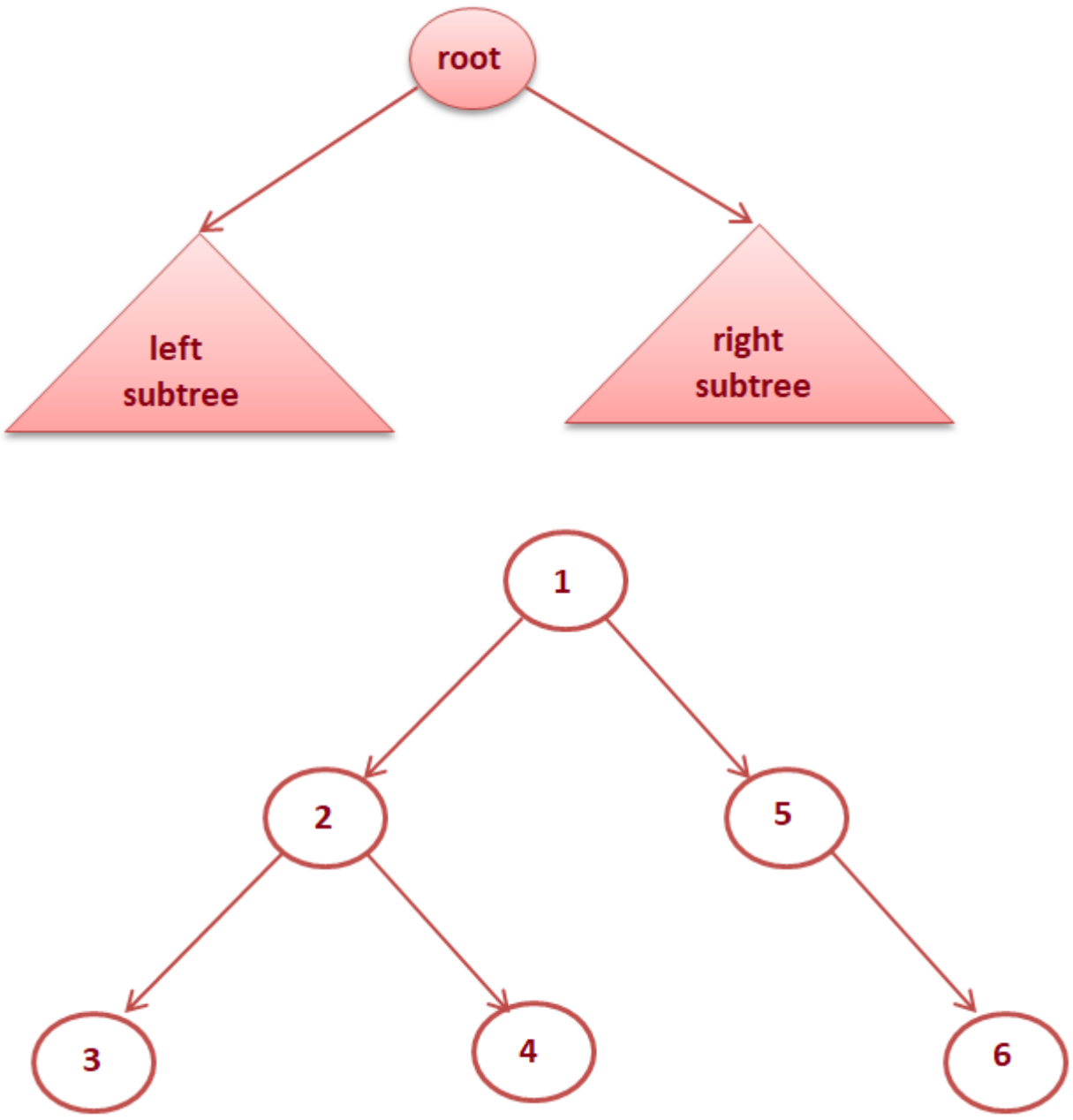


- **Height**: The height of the node is the length of the path from that node to the deepest node in the tree. The height of the tree is the length of the path from the root node to the deepest node in the tree. (Eg: the height of the tree in the above example is four (count the edges, not the nodes)).
- 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.

Binary Trees

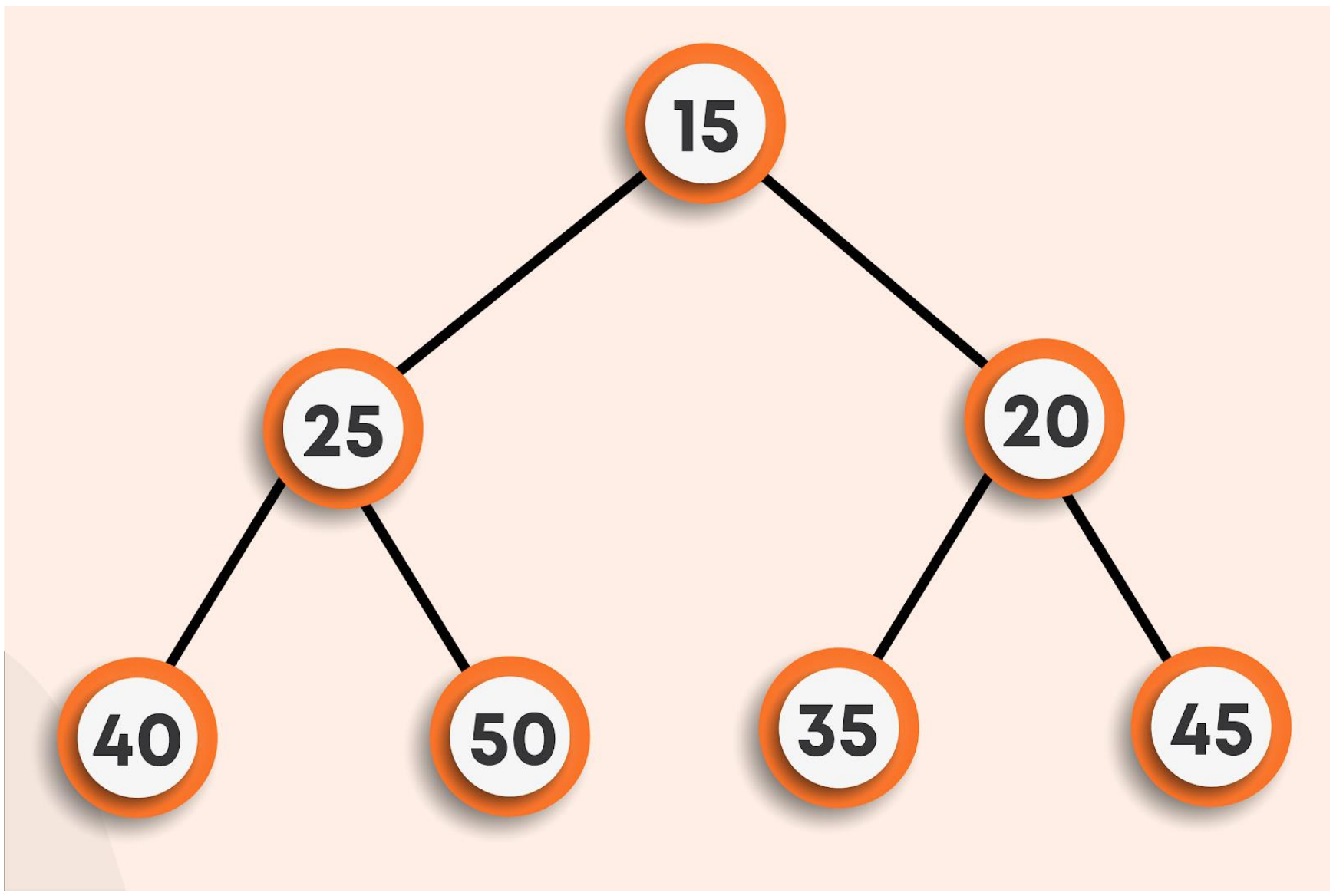
A tree is called a binary tree if every node in the tree has either zero, one, or two children. An empty binary tree is also a valid binary tree. A binary tree is made of nodes that constitute a left pointer, a right pointer, and a data element. The root pointer is the topmost node in the tree.

We can visualize a binary tree as the root node and two disjoint binary trees, called the left subtree and the right subtree.

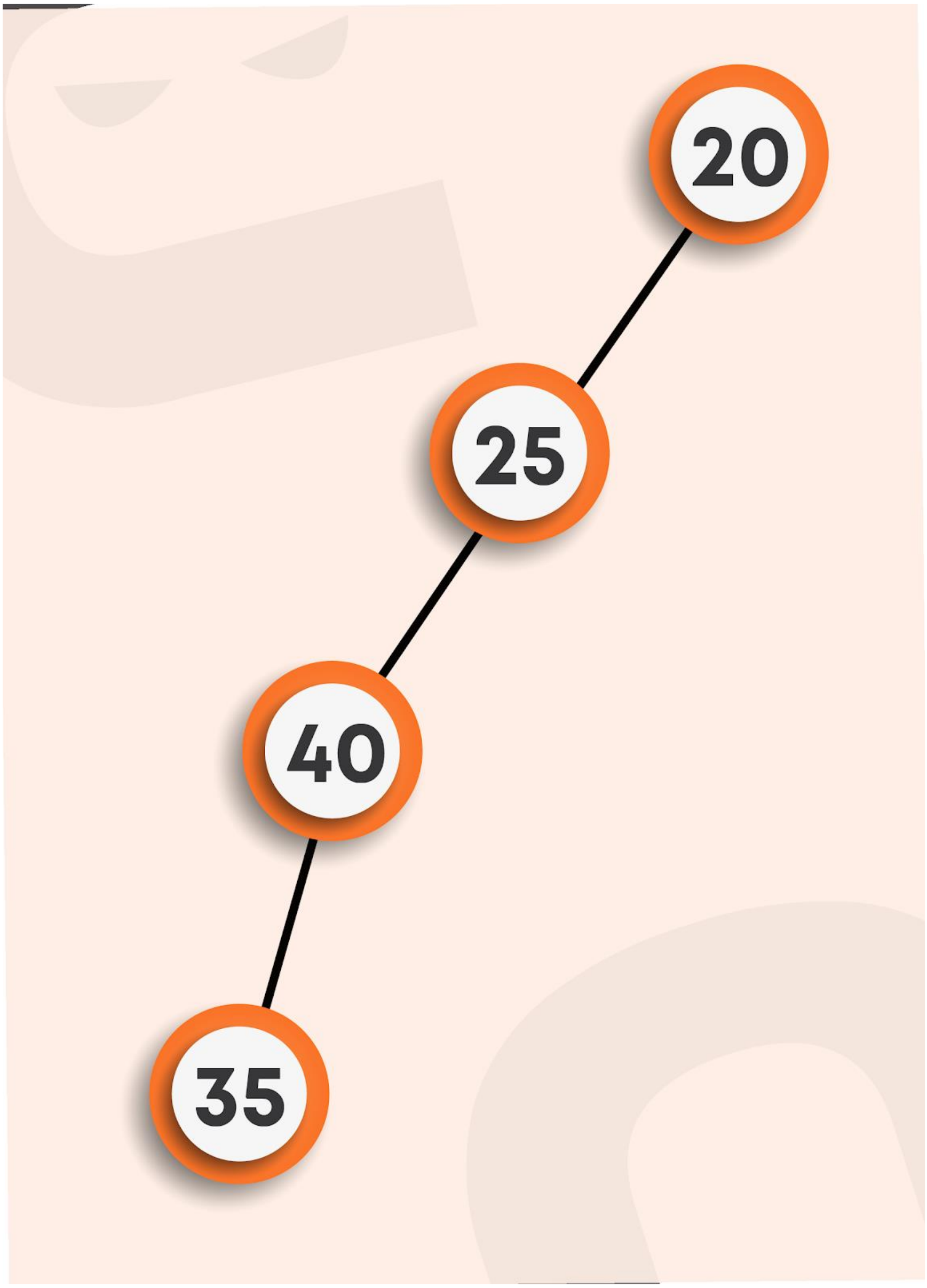


Types of Binary Trees

- **Strict Binary Tree:** A binary tree in which each node has exactly zero or two children.
- **Full Binary Tree**: A binary tree in which each node has exactly two children and all the leaf nodes are at the same level.



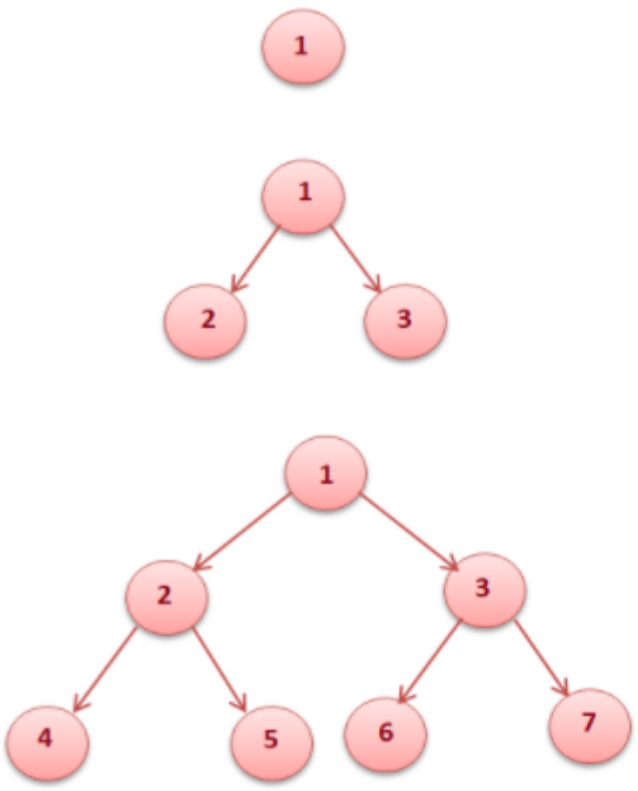
- **Complete Binary Tree**: A complete binary tree has all the levels filled except for the last level, which has all its nodes as much as to the left.
- **A degenerate tree:** In a degenerate tree, each internal node has only one child.



Properties of a Binary Tree

For the given properties, let’s assume that the tree has a height of h, also the root node is at a height of 0.

Height	Number of nodes at level h
h = 0	1
h = 1	2
h = 2	4



From the diagram, we can say that

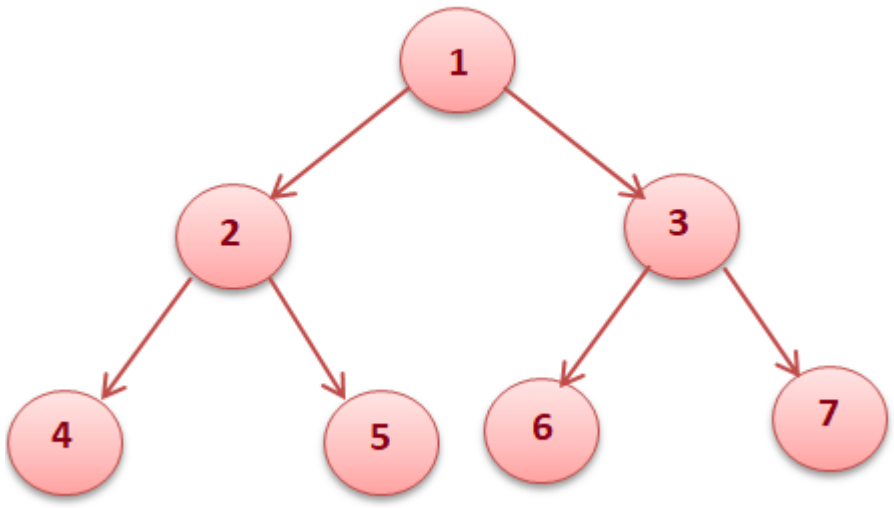
- The number of nodes in a full binary tree is 2^**(h+1**) -1
- The number of leaf nodes in a full binary tree is 2^**h**
- The number of node links in a complete binary tree of n nodes is n + 1 .

Structure of Binary trees

Each node in a binary tree contains **data**, a **left** pointer, and a **right** pointer.

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

Operations on a binary tree

- Count nodes in a binary tree.
- Find the maximum value in a binary tree.
- Search a value in a binary tree.
- Height of a binary tree.

Count nodes in a binary tree.

```
function countNodesInABinaryTree(root)

        if the root is null
                return 0
        /*
            Count nodes in the left subtree and right subtree recursively
            and add 1 for self node
        */
        leftCount = countNodesInABinaryTree(root.left)
        rightCount = countNodesInABinaryTree(root.right)

        return leftCount + rightCount + 1
```

Find maximum value in a binary tree.

```
function findMax(root)
        if root is null
                return Integer.MIN_VALUE
        /*
          Find Maximum in the left subtree and right subtree recursively
          and compare self data, max of left subtree, max of right
          subtree and return max of these three
        */

        max = root.data
        leftMax = findMax(root.left)
        rightMax = findMax(root.right)

        if leftMax > max
                max = leftMax
        if rightMax > max
                max = rightMax
        return max
```

Search a value in a binary tree.

```
function search(root, val)
        if the 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.
        */
        if root.data equals val
                    return true

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

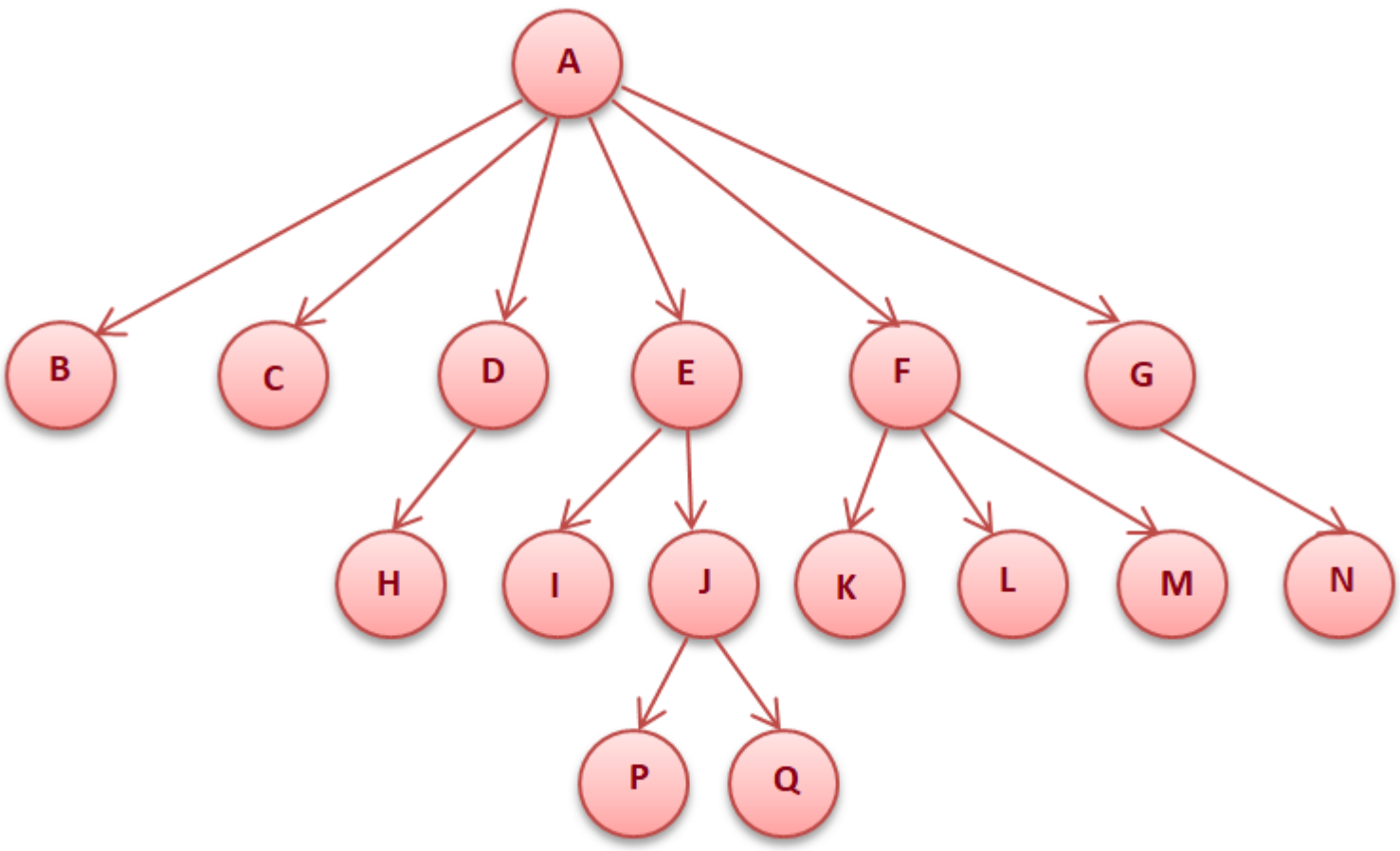
Height of a binary tree.

```
function findHeight(root)
        if root is null
                    return 0
        /*
          Find the height of the left subtree and right subtree.
          Then add 1 (for current node) to max value from leftHeight
          and rightHeight
        */
        leftHeight = findHeight(root.left)
        rightHeight = findHeight(root.right)
        return 1 + max(leftHeight, rightHeight)
```

Time Complexity

Operations	Time Complexity
preOrder Traversal()	O(n)
inOrder Traversal()	O(n)
postOrder Traversal()	O(n)
levelOrder Traversal()	O(n)
findMax()	O(n)
search(val)	O(n)
countNodesInABinaryTree()	O(n)
findHeight()	O(n)

Generic Trees (N-ary Trees)



The generic tree is a collection of nodes in which node has at most N children. A generic tree consists of a **list of pointers to the child nodes** and the **data** element. The **root** pointer is the topmost node in the tree.

Printing a generic tree (recursively)


```
function printTree(root)

        //   process current node
        print “root.data”

        //   calling function recursively for childNodes
        for every childNode in the root.children list
                printTree(childNode)
```

The output of the above tree: A B C D H E I J P Q F K L M G N

LevelOrder Printing of Generic Tree

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 the 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()
                print “removedNode.data”

                /*
                        Add the childNodes of the removedNode to the queue
                */
                for every childNode in removedNode.children list
                        queue.add(childNode)
         end
```

LevelOrder output of above tree: A B C D E F G H I J K L M N P Q

Application of trees

- Expression trees are used in compilers

- Huffman coding trees are used in data compression algorithms.
- Used to implement indexing in databases through B- Tree and B+ Tree.
- Used to implement dictionaries(tries) for lookup.

Previous

Next

BST Notes

Binary Search Trees(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?

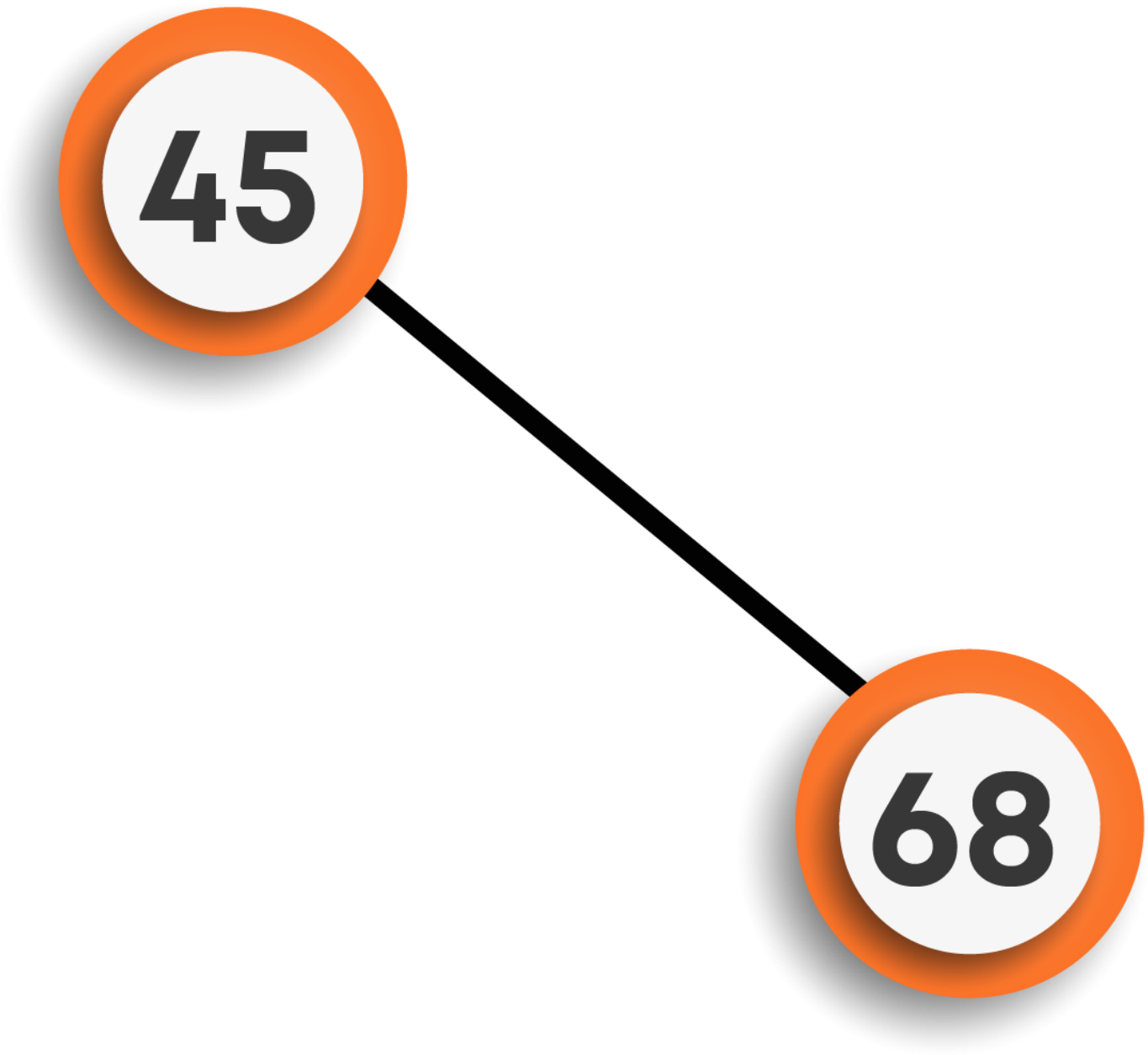
In BSTs, we always insert the node with smaller data towards the left side of the compared node and the larger data node as its right child. To be more concise, consider the root node of BST to be N, then:

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

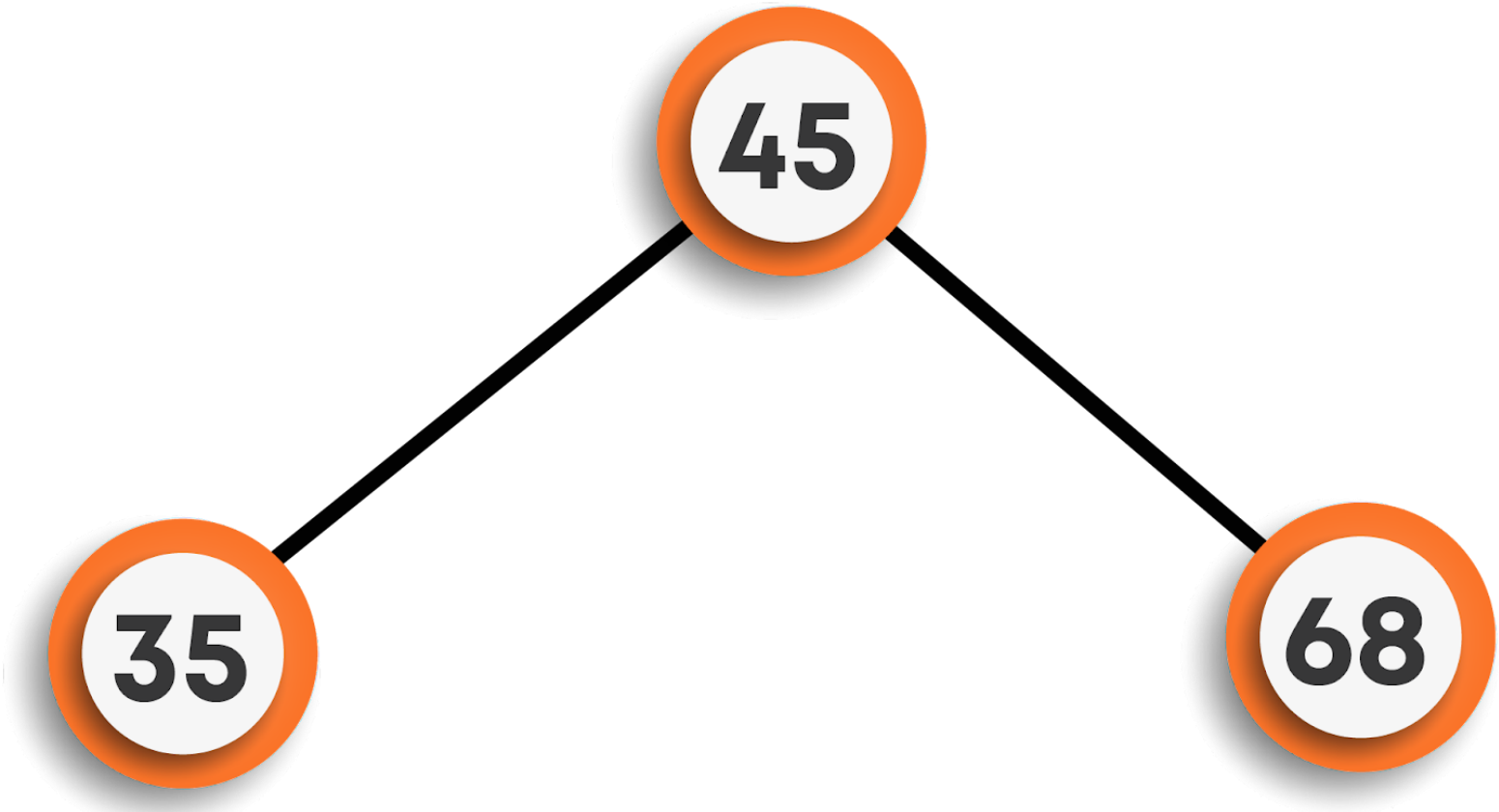
For Example: Insert {45, 68, 35, 42, 15, 64, 78} in a BST in the order they are given.
Solution: Follow the steps below:

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

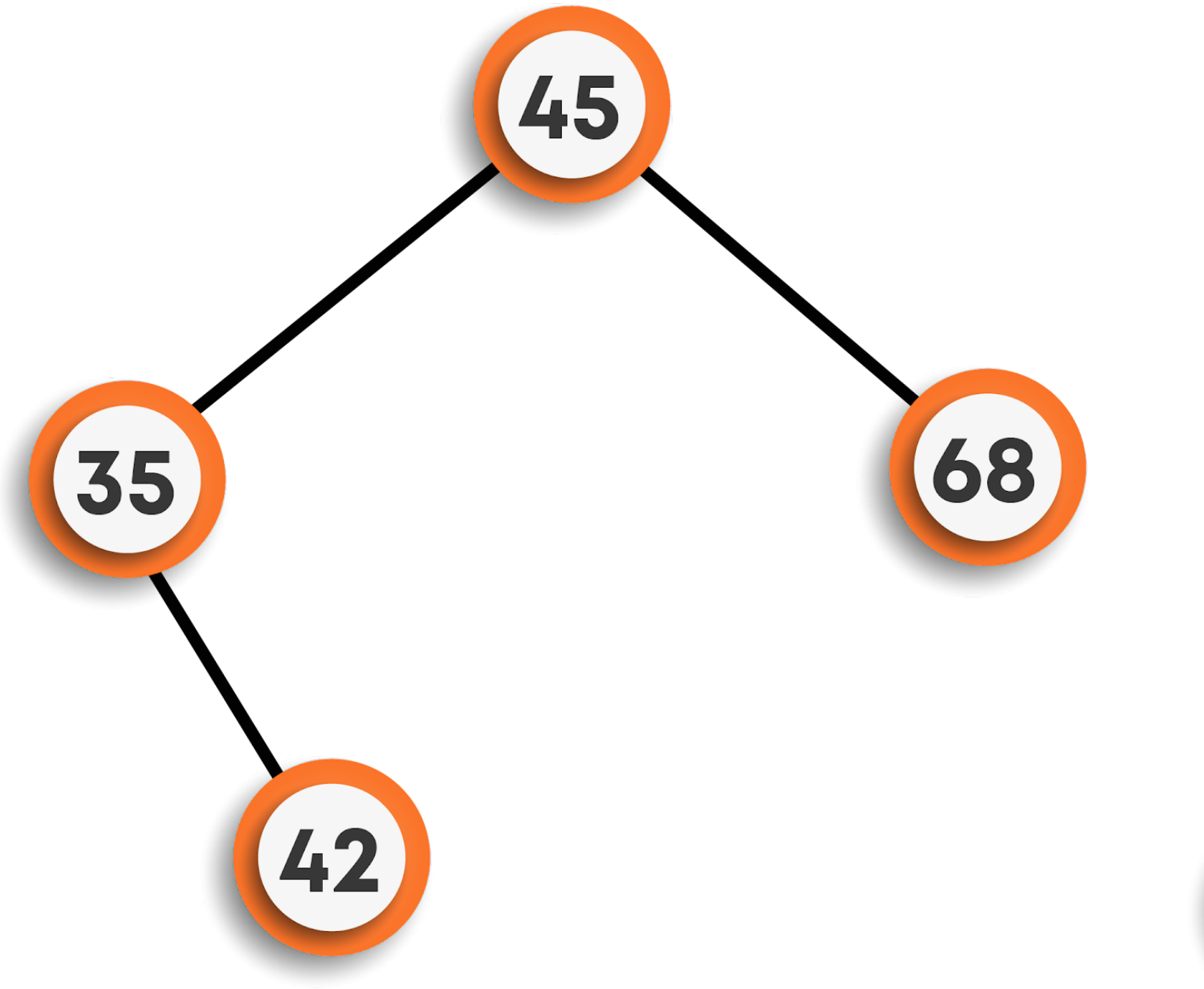
- Now, we have to insert 68, which is greater than 45, so it will go on the right side of the root node.



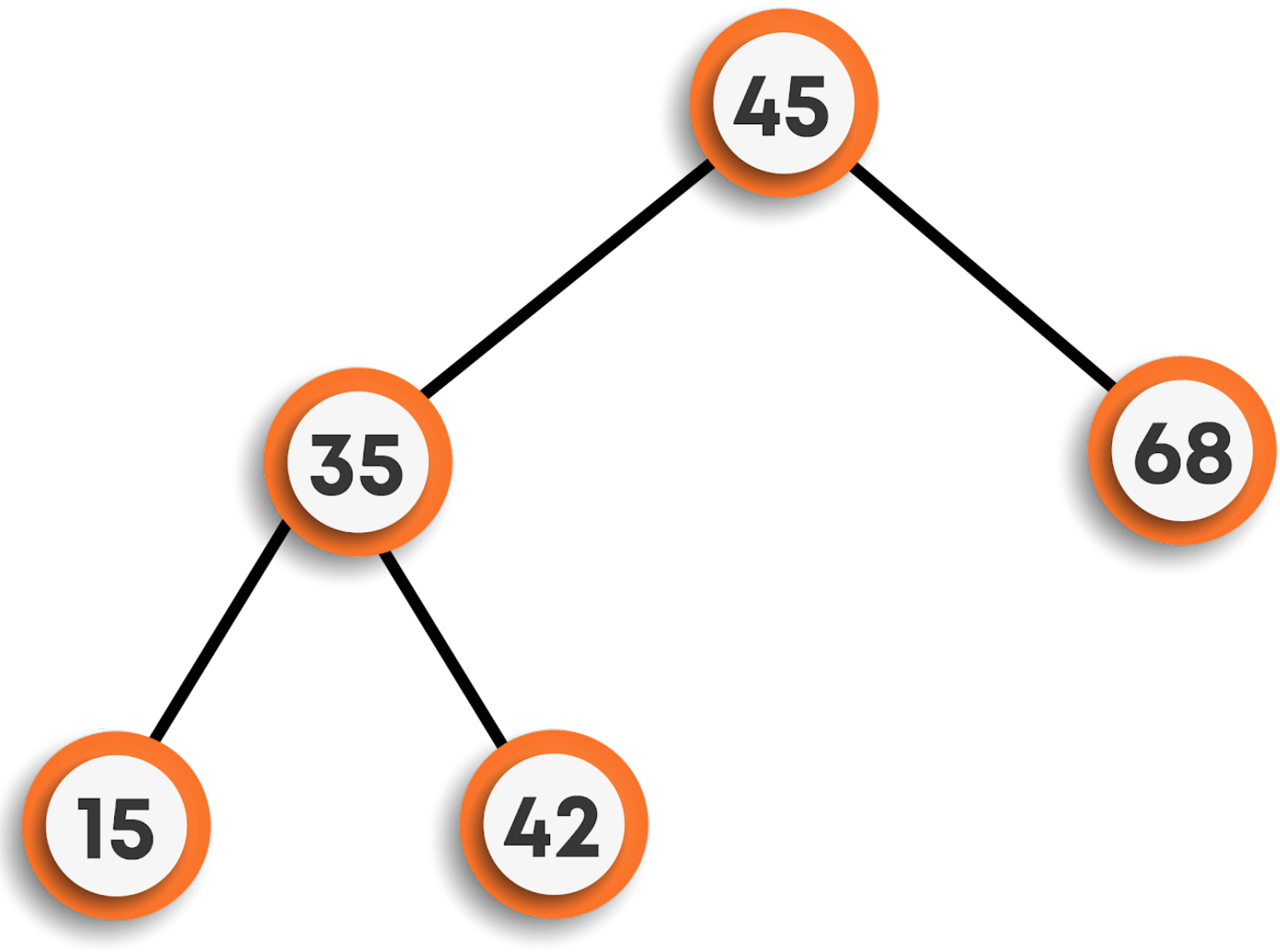
- To insert 35, we will start from the root node. Since 35 is smaller than 45, it will be inserted on the left side.



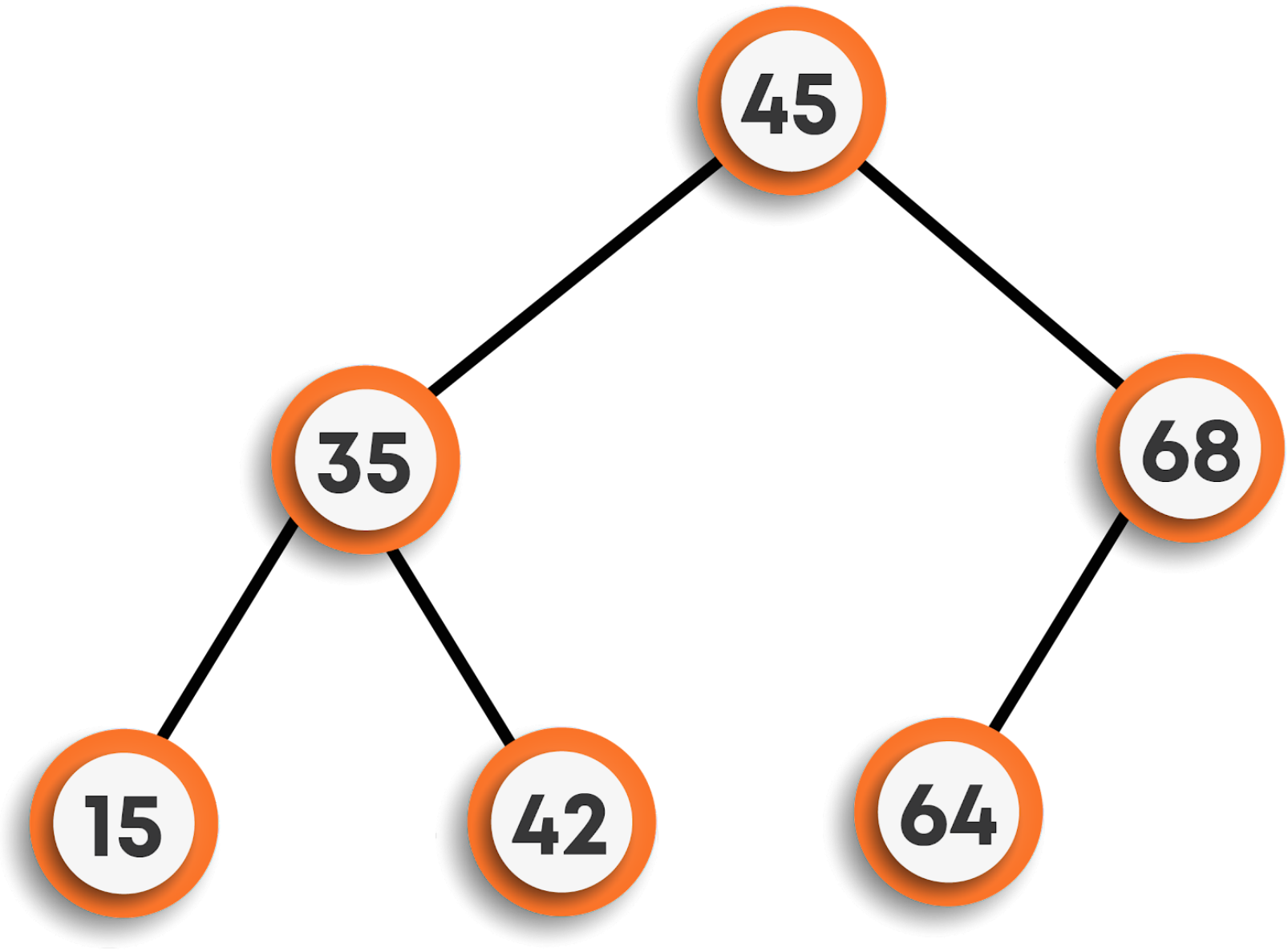
- Moving on to inserting 42, we can see that 42 is less than 45 so it will be placed on the left side of the root node. Now, we will check the same on the left subtree. We can see that 42 > 35, this means 42 will be the right child of 35, which is still a part of the left subtree of the root node.



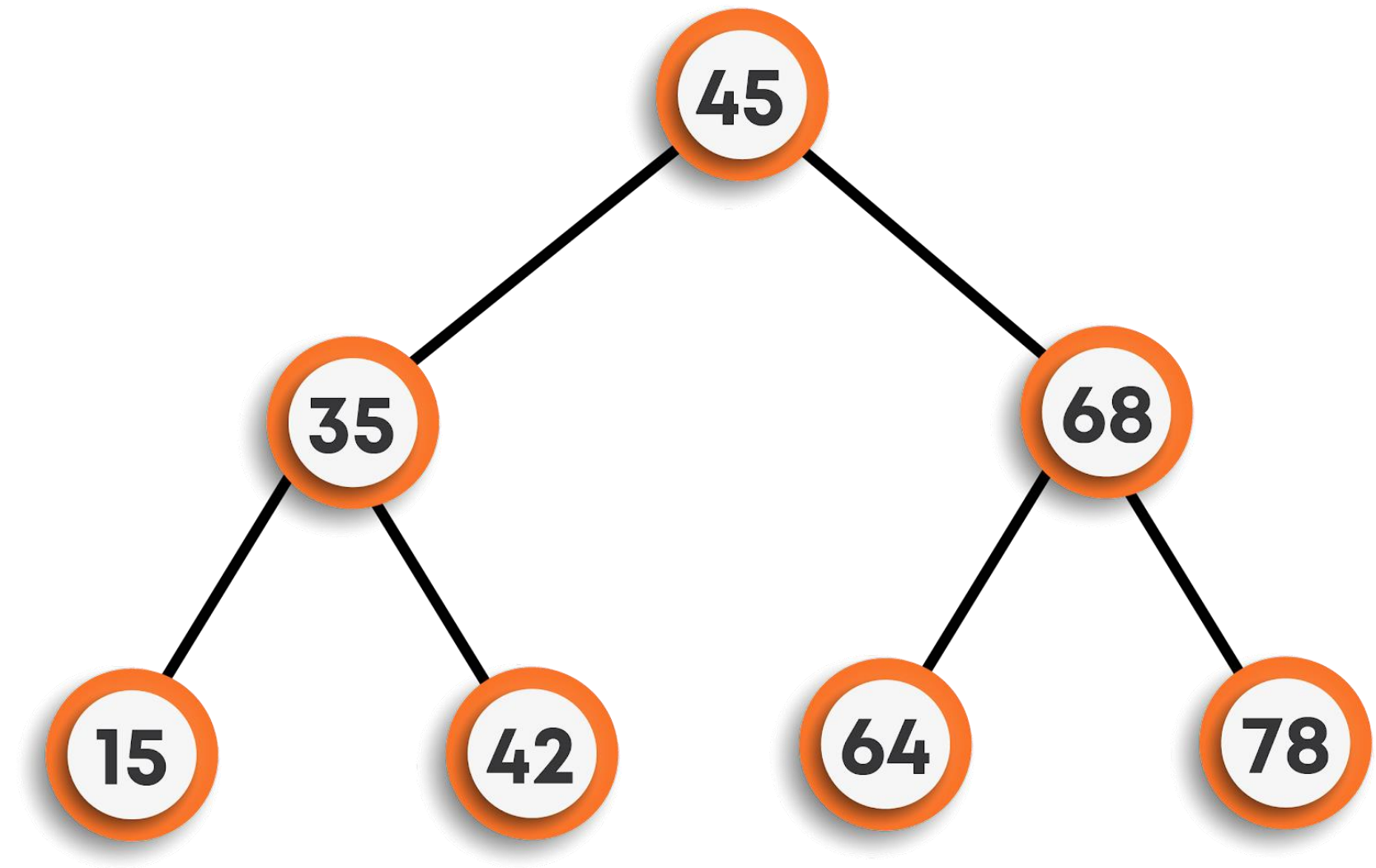
- Now, on inserting 15, we will follow the same approach starting from the root node. Here, 15 < 45, means left subtree. Again, 15 < 35, means continue to move towards the left. As the left subtree of 35 is empty, so 15 will be the left child of 35.



- 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.



- 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).

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 (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
```

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)

Variants of Balanced BST

For a balanced BST:
|Height_of_left_subtree - Height_of_right_subtree| <= 1

This equation must be valid for each and every node present in the BST.
It can be proved that the height of a Balanced BST is O(logn), where n is the number of nodes in the tree.
This can be summarised as the time complexity of operations like searching, insertion, and deletion can be performed in O(logn).
There are many Binary search trees that maintain balance.

- AVL Trees (also known as self-balancing BST, uses rotation to balance)
- Red-Black Trees
- 2 - 4 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.

Previous

Next

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.

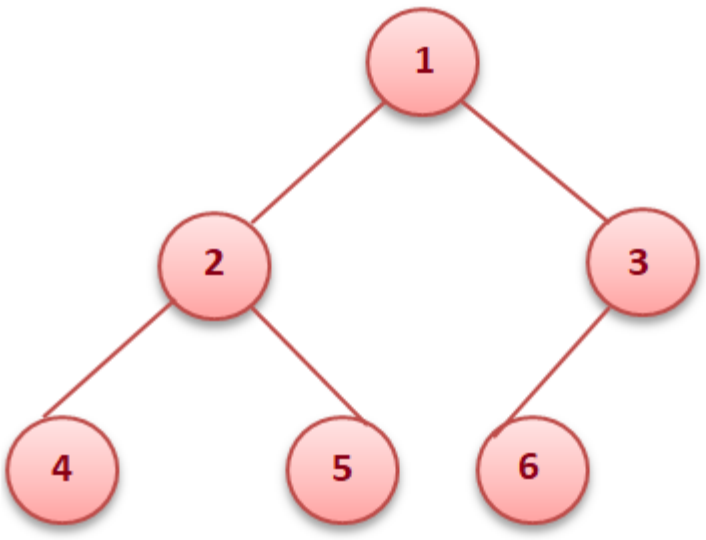
Comparing Implementation

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

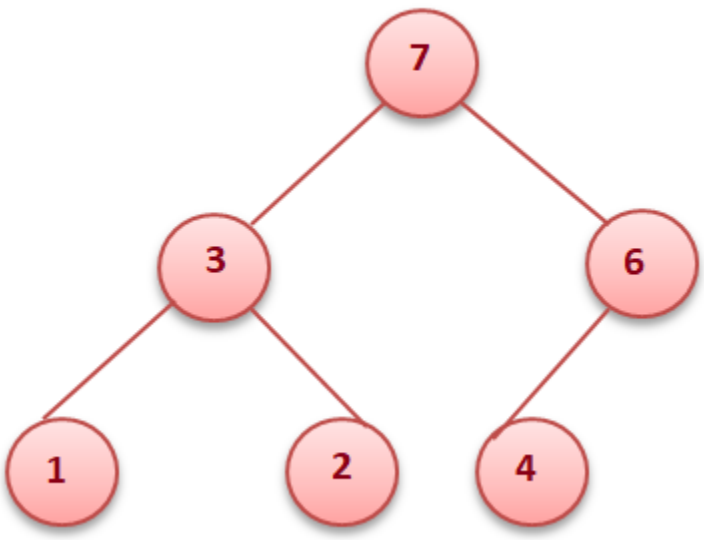
Heaps

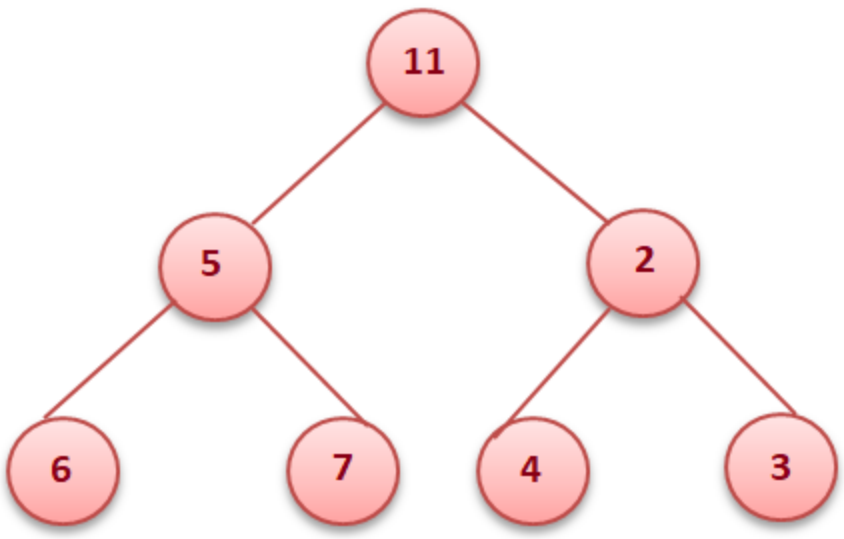
- A heap is a binary tree with some special properties.
- The basic requirement of a heap is that the value of a node must be ≥ (or ≤) than the values of its children. This is called the **heap property**.
- A heap also has the additional property that all leaf nodes should be at **h** or **h – 1** level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



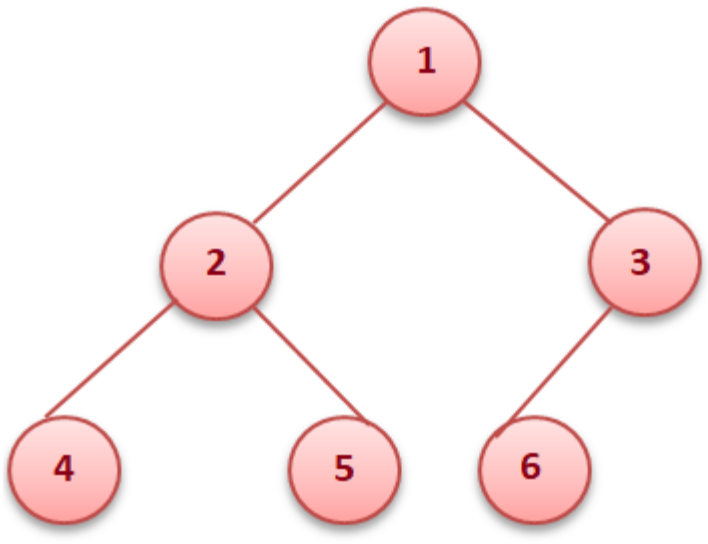
In the examples below, the left tree is a heap (each element is greater than its children) and the right is not a heap (since 11 is greater than 2 and 5 whereas the rest of the nodes are less than their children).



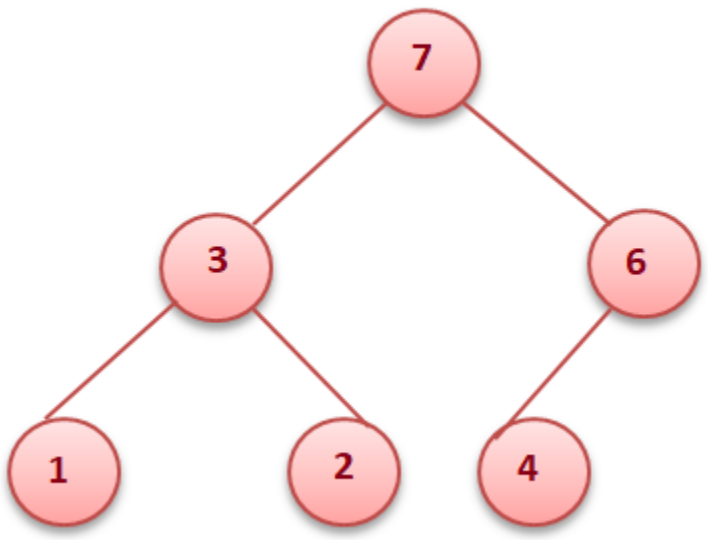


Types of heap

- **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.



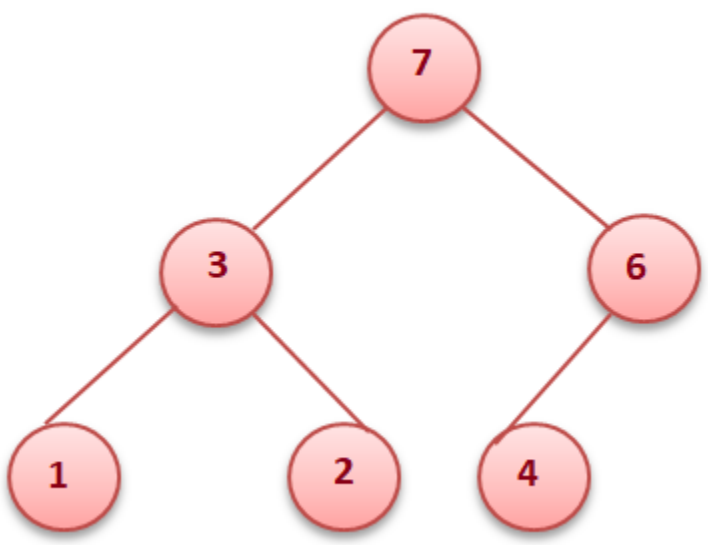
Binary Heaps

- 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.

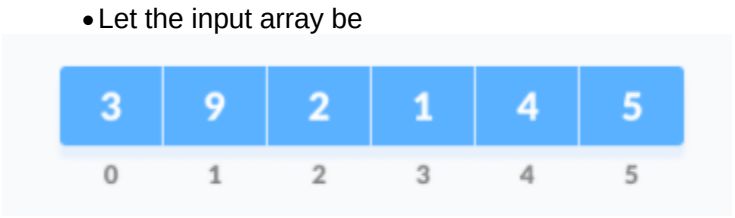


Children of a Node : For a node at index i, its children are at index **(2*i + 1)** and (**2i + 2**). Like in the fig above **3 is at index 1** and has children at index 3 **(2 * 1 + 1)** and at index 4 **(2 * 1 + 2) .**
Getting the maximum/minimum element: The maximum element in a max heap, or the minimum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

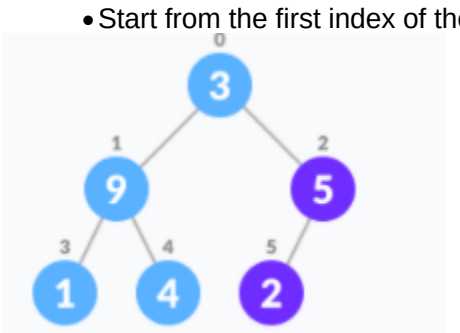
Heapify

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.

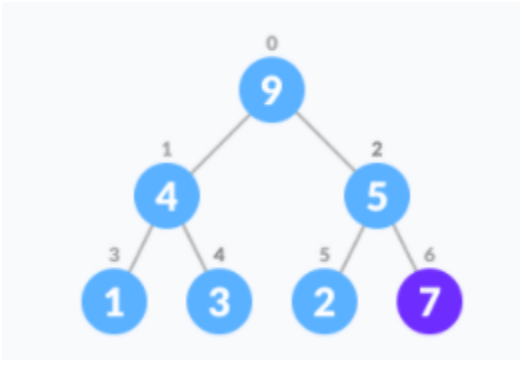
Pseudocode:

```
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)
```

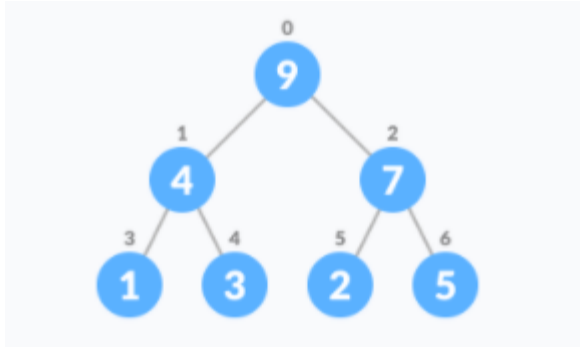
Inserting into a heap

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

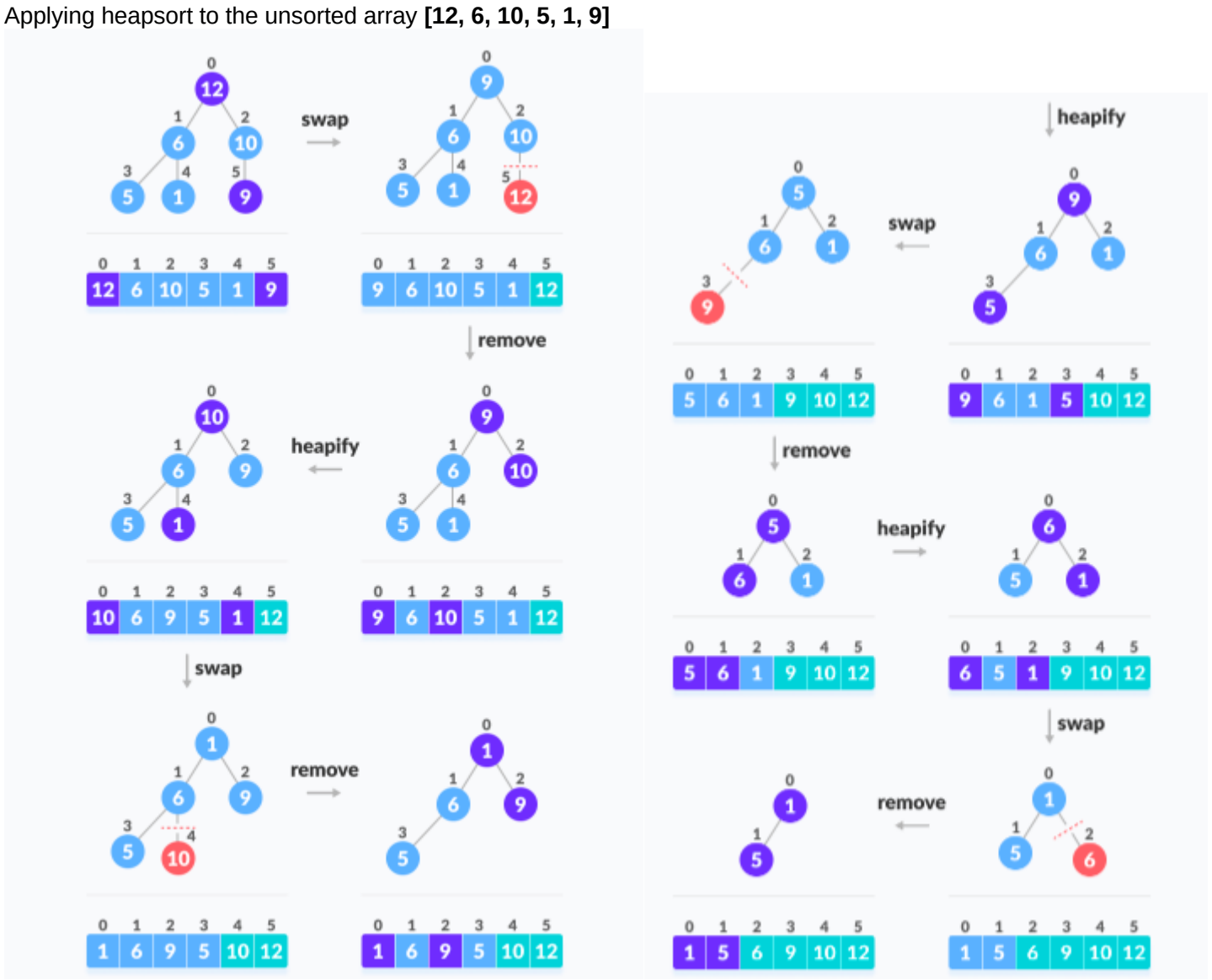
- Heap Sort is another example of an efficient sorting algorithm. Its main advantage is that it has a worst-case runtime of **O(nlog(n))** regardless of the input data.
- As the name suggests, Heap Sort relies heavily on the heap data structure - a common implementation of a Priority Queue.
- Without 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.

Previous

Next

Graphs Notes

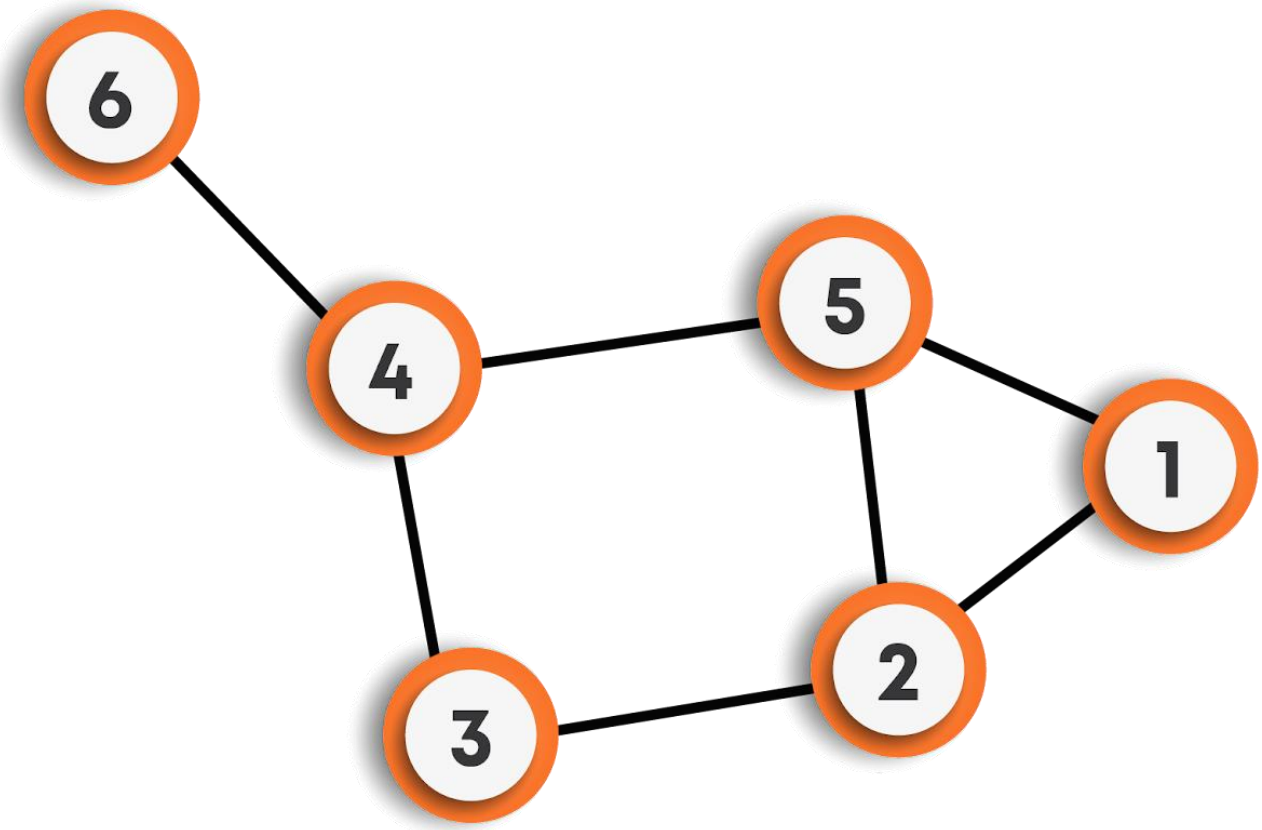
Graphs

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 for 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.

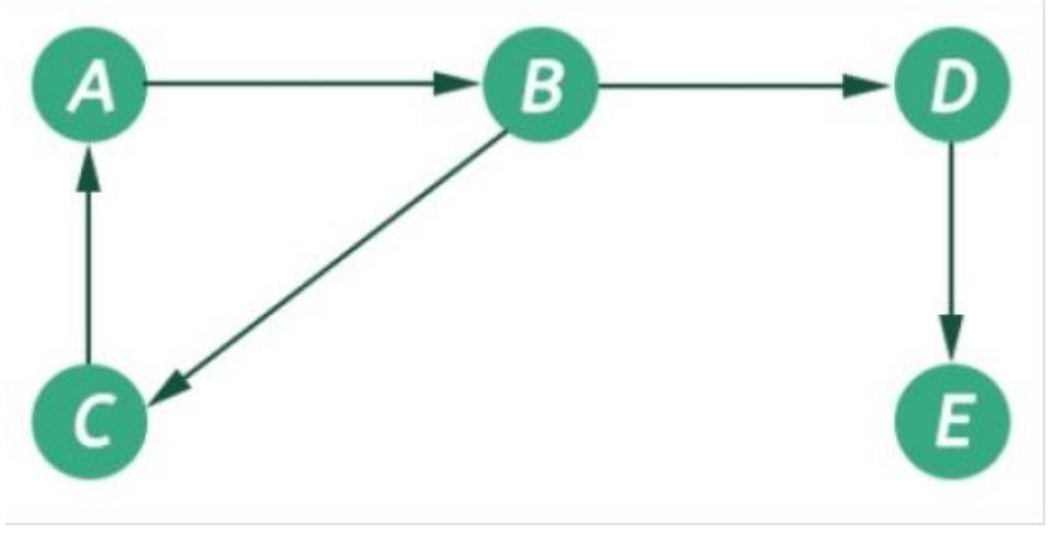
DIRECTED GRAPHS -

- 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.

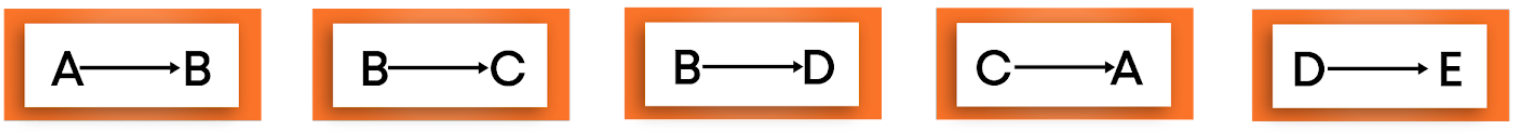
Graphs Representation

Suppose the graph is as follows:

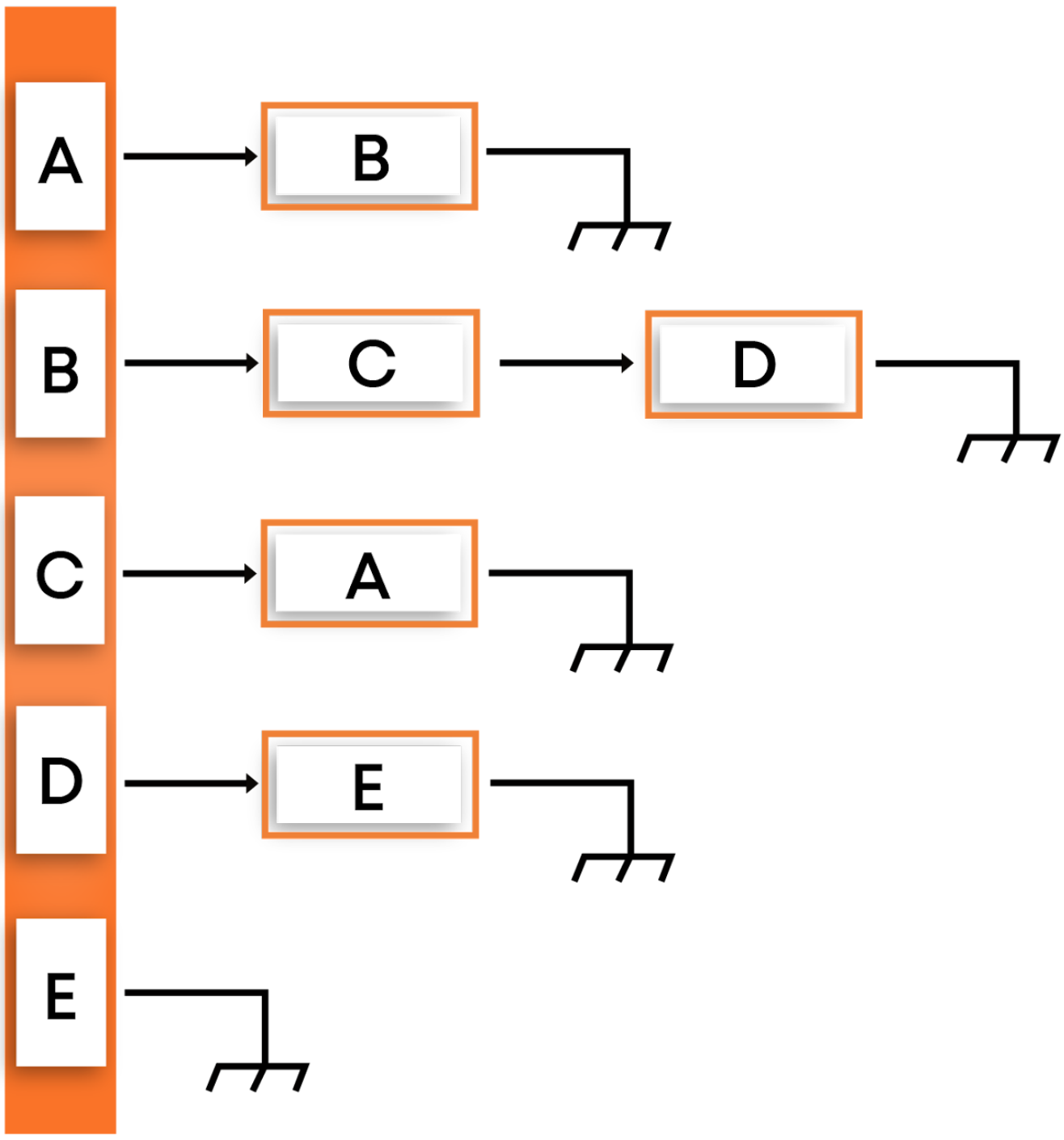


There are the following ways to implement a graph:

- **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:



1. It is easy to follow and clearly shows the adjacent nodes of a particular node
2. 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.
3. 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:

	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

1. For a **simple graph** (that has no loops), the adjacency matrix has 0s on the diagonal
2. The adjacency matrix of an **undirected graph** is symmetric.
3. The memory use of an adjacency matrix for **n** nodes is **O(n^2)**, where **n** is the number of nodes in the graph.
4. The adjacency matrix for a **weighted graph** contains the weights of the edges connecting the nodes.

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.

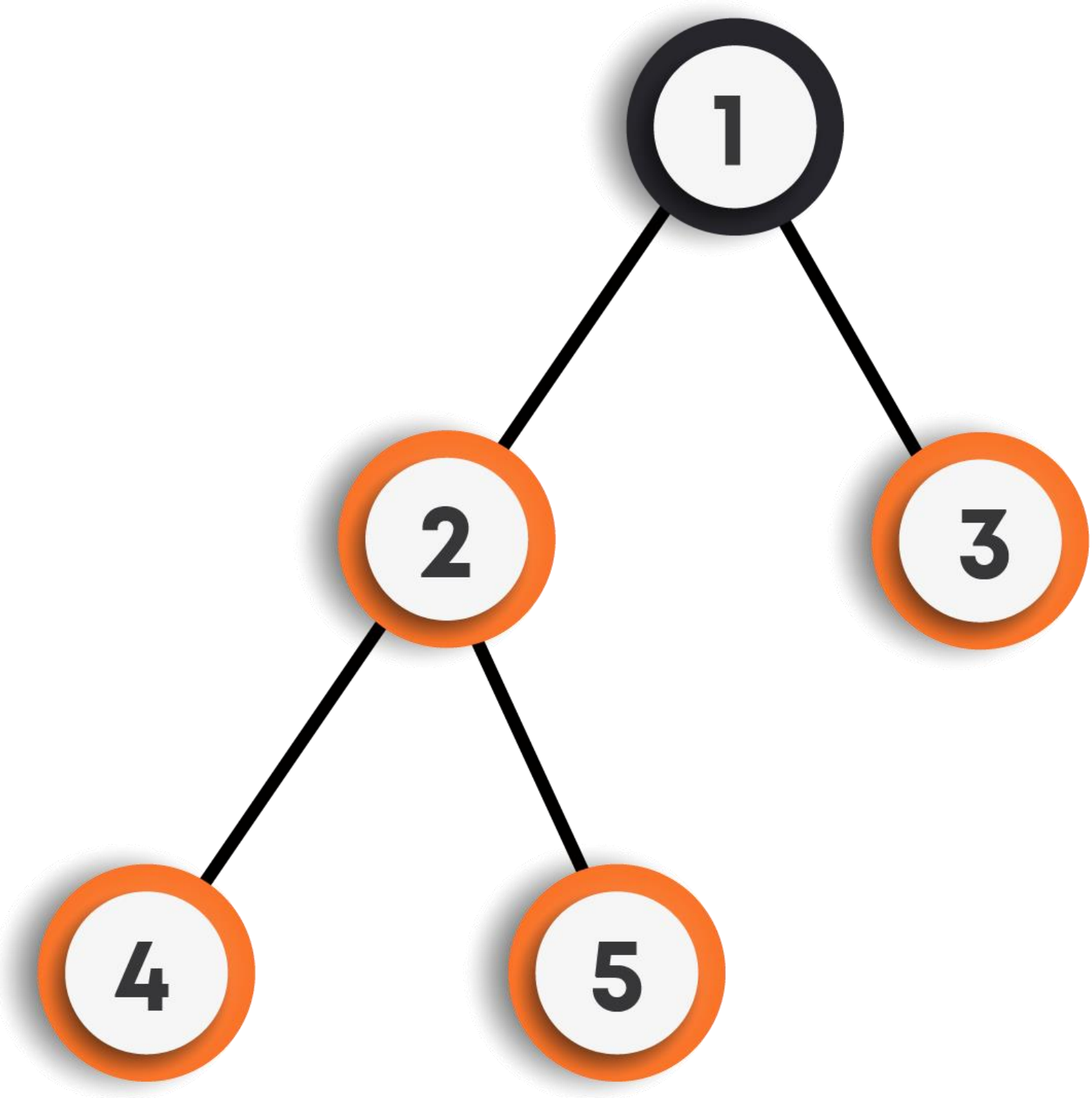
While breadth-first search uses a **queue** as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a **stack**. But both these algorithms make use of a bool variable **VISITED**. During the execution of the algorithm, every node in the graph will have 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:

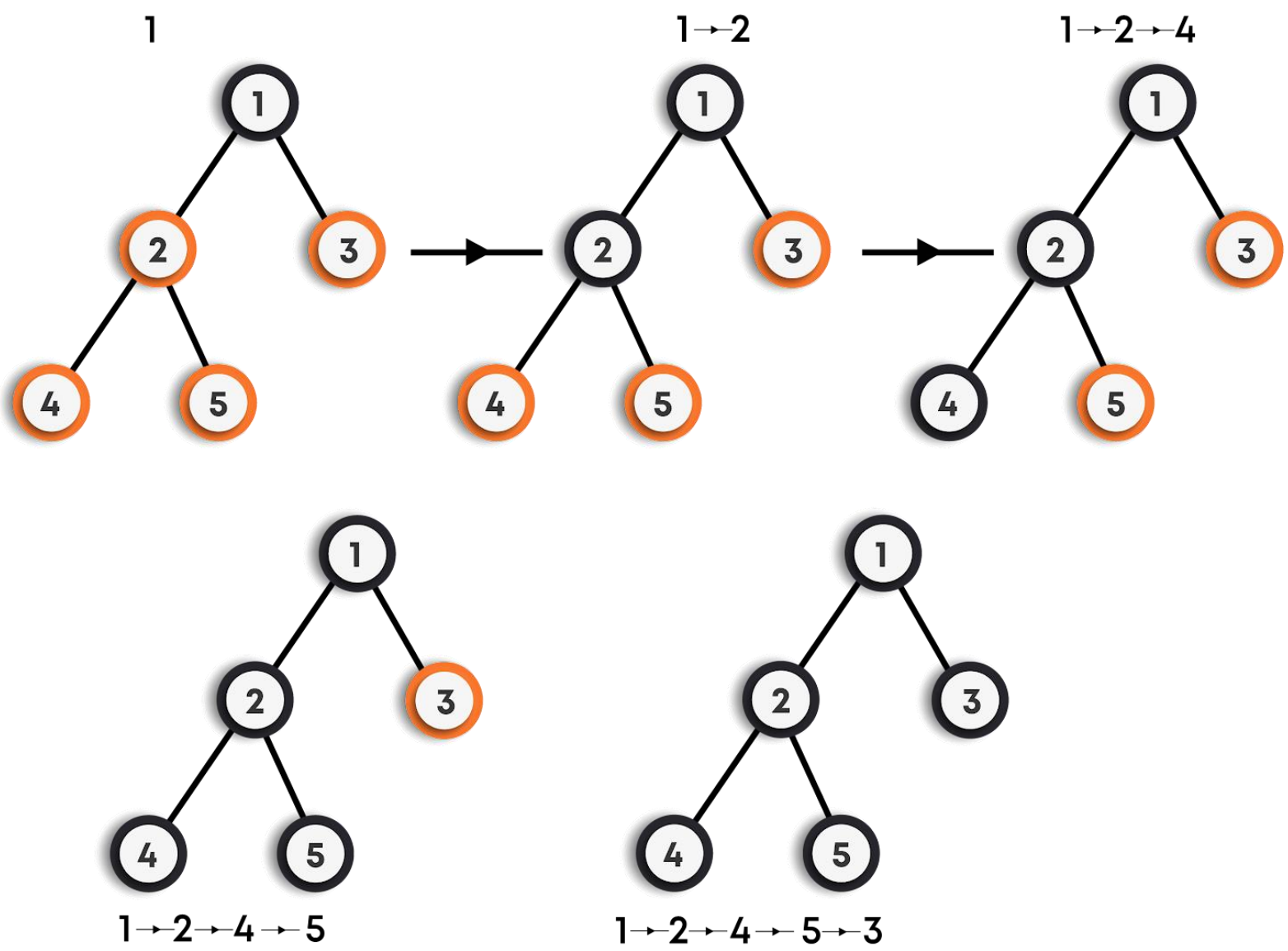


DFS Traversal : 1->2->4->5->3

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.



Pseudocode(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
```

Pseudocode(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.

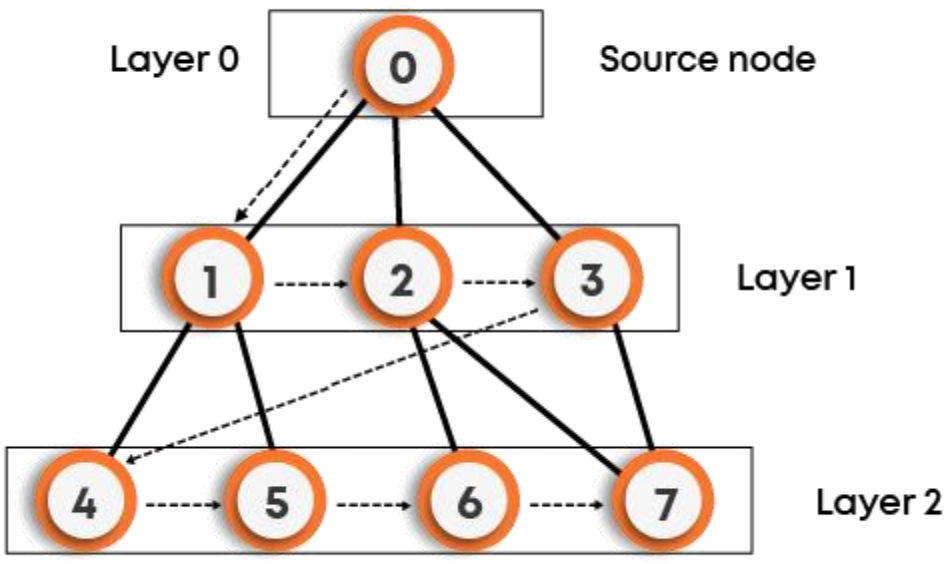
Applications of Depth-First Search Algorithm

- 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.

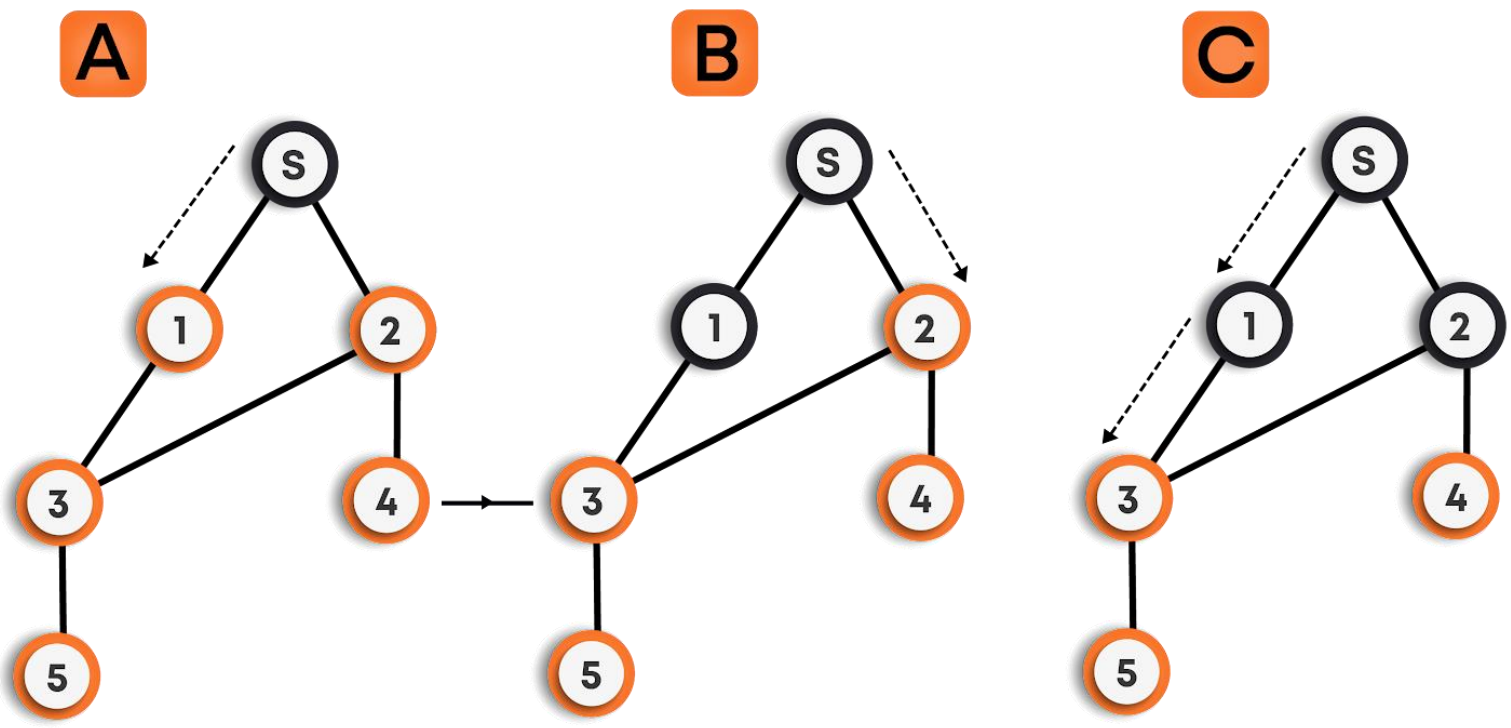


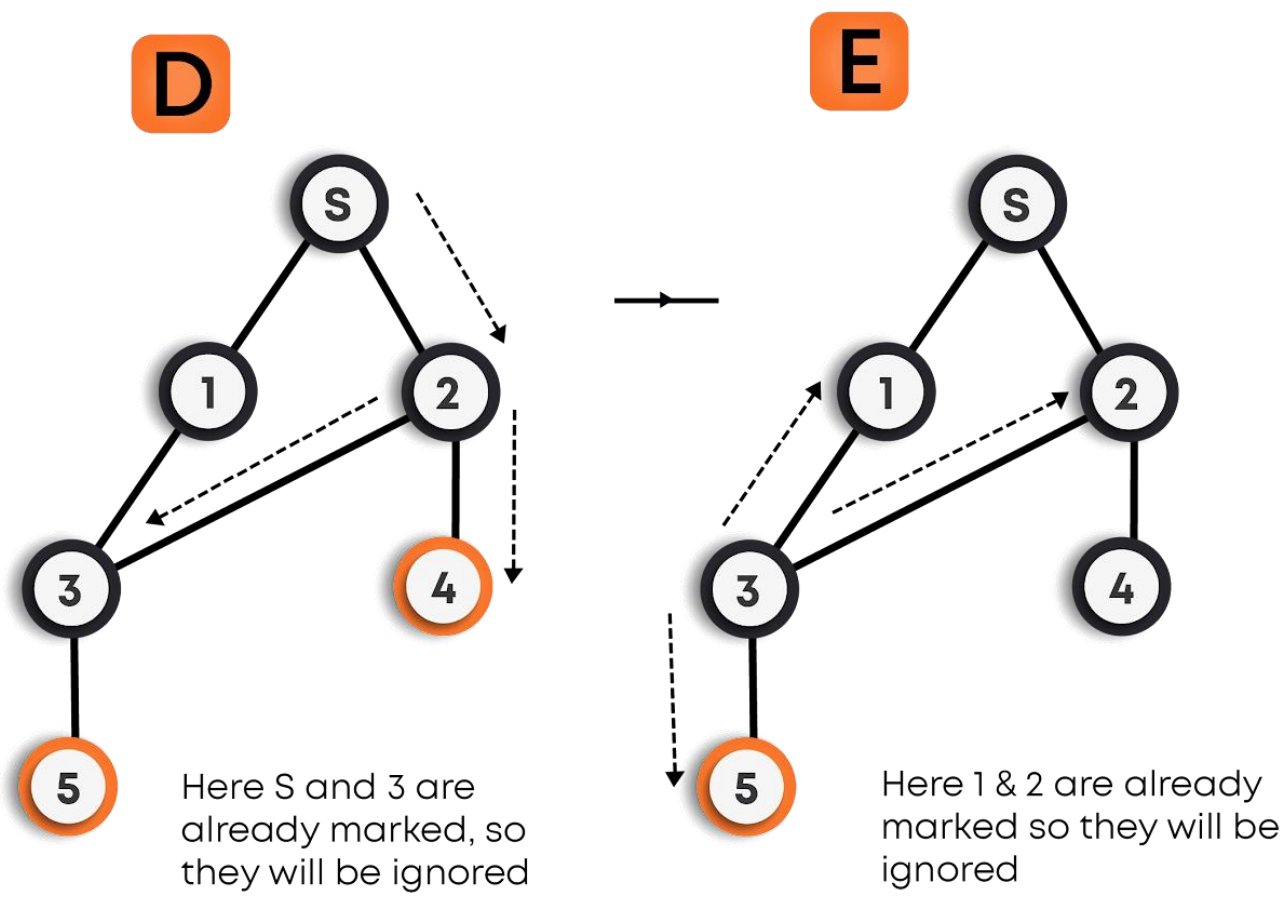
BFS Traversal: 0->1->2->3->4->5->6->7
Other possible BFS traversals for the above graph can be:

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:





Pseudocode(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 or
                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.

Applications of graphs

Graphs are very powerful data structures, they are very important in modeling data, many problems can be reduced to known graph problems, hence find many applications, some of them are -

- **Social network graphs:** Graphs can be modeled to represent social networks to represent who knows whom, who communicates with whom, who influences whom, or other relationships in social structures
- **Transportation networks:** In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.
- **Document link graphs:** The best-known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze the relevance of web pages, the best sources of information, and good link sites.
- **Neural Networks:** Vertices represent neurons and edges represent the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^11 neurons and close to 10^15 synapses.
- **Epidemiology:** Vertices represent individuals and directed edges represent the transfer of an infectious disease from one individual to another. Analyzing such graphs has become an important component in understanding and controlling the spread of diseases.
- **Scene graphs:** In graphics and computer games scene graphs represent the logical or spatial relationships between objects in a scene. Such graphs are very important in the computer games industry.
- **Compilers:** Graphs are used extensively in compilers. They can be used for type inference, for so-called data flow analysis, register allocation, and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.
- **Dependence graphs:** Graphs can be used to represent dependencies or precedences among items. Such graphs are often used in large projects in laying out what components rely on other components and used to minimize the total time or cost to completion while abiding by the dependencies.
- **Robot planning:** Vertices represent states the robot can be in and the edges the possible transitions between the states. Such graph plans are used, for example, in planning paths for autonomous vehicles.

Previous

Next

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.

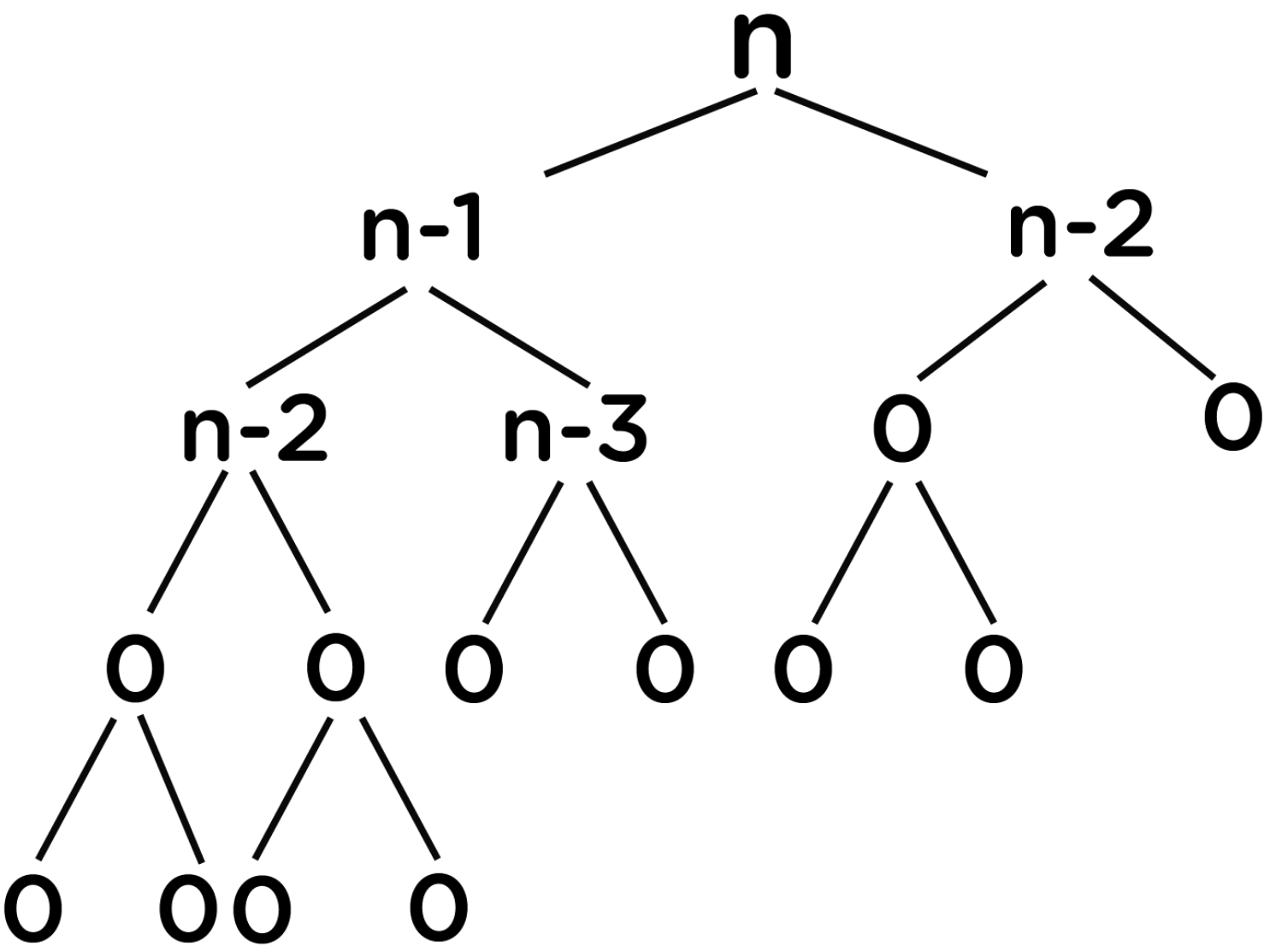
Introduction

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)
```

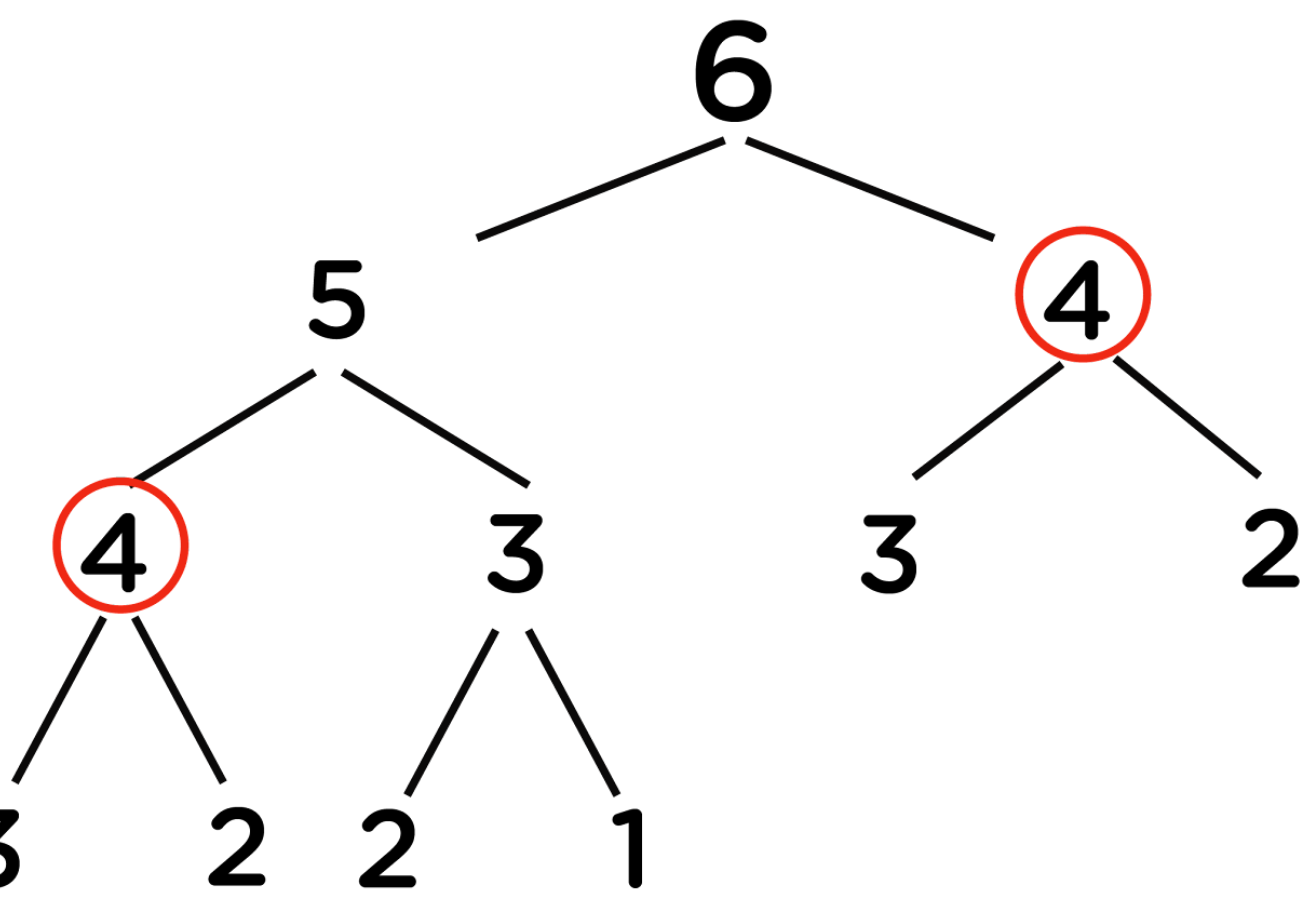
- Here, for every n, we need to make a recursive call to **f(n-1)** and **f(n-2)**.
- For **f(n-1)**, we will again make the recursive call to **f(n-2)** and **f(n-3).**
- Similarly, for **f(n-2)**, recursive calls are made on **f(n-3)** and **f(n-4)** until we reach the base case.

- 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 6**th** 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.

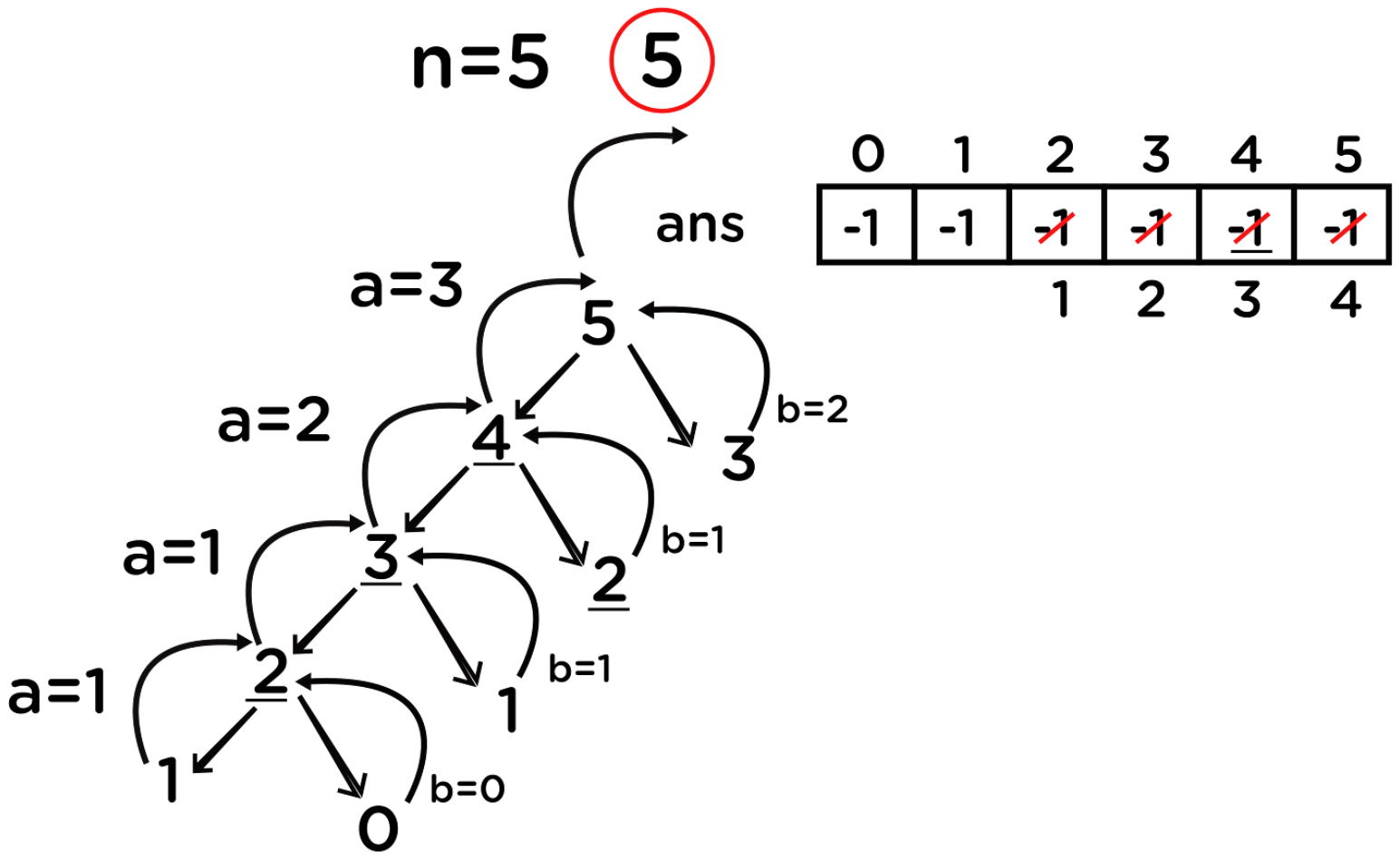
Now in this process of memoization, considering the above Fibonacci numbers example, it can be observed that the total number of unique calls will be at most **(n+1)** only.
Let's look at the memoization code for Fibonacci numbers below:

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 5**th** 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 n**th** 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:

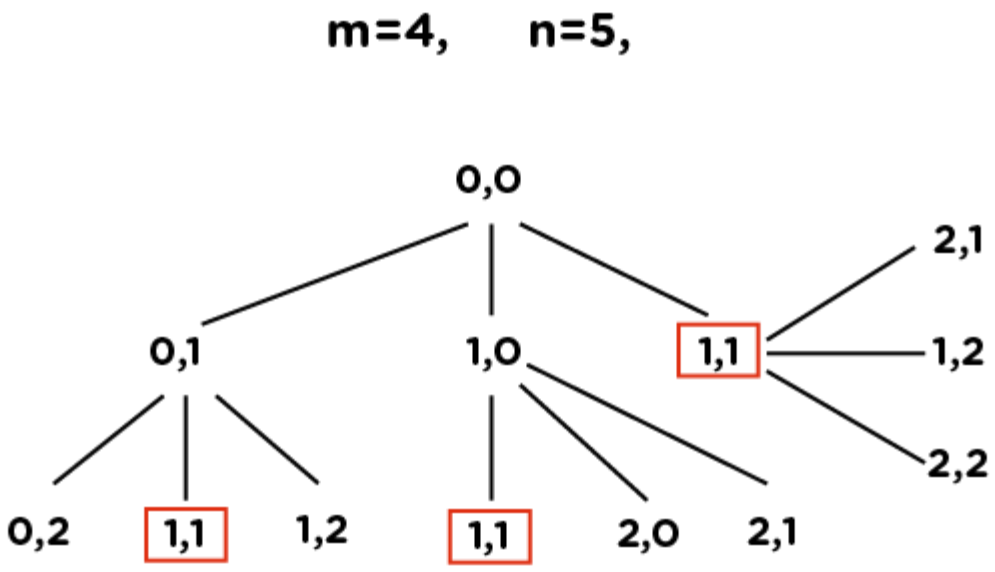
- Thinking about the **recursive approach** to reach from the cell **(0, 0)** to **(m-1, n-1)**, we need to decide for every cell about the direction to proceed out of three.
- We will simply call recursion over all the three choices available to us, and finally, we will be considering the one with minimum 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)
        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**"

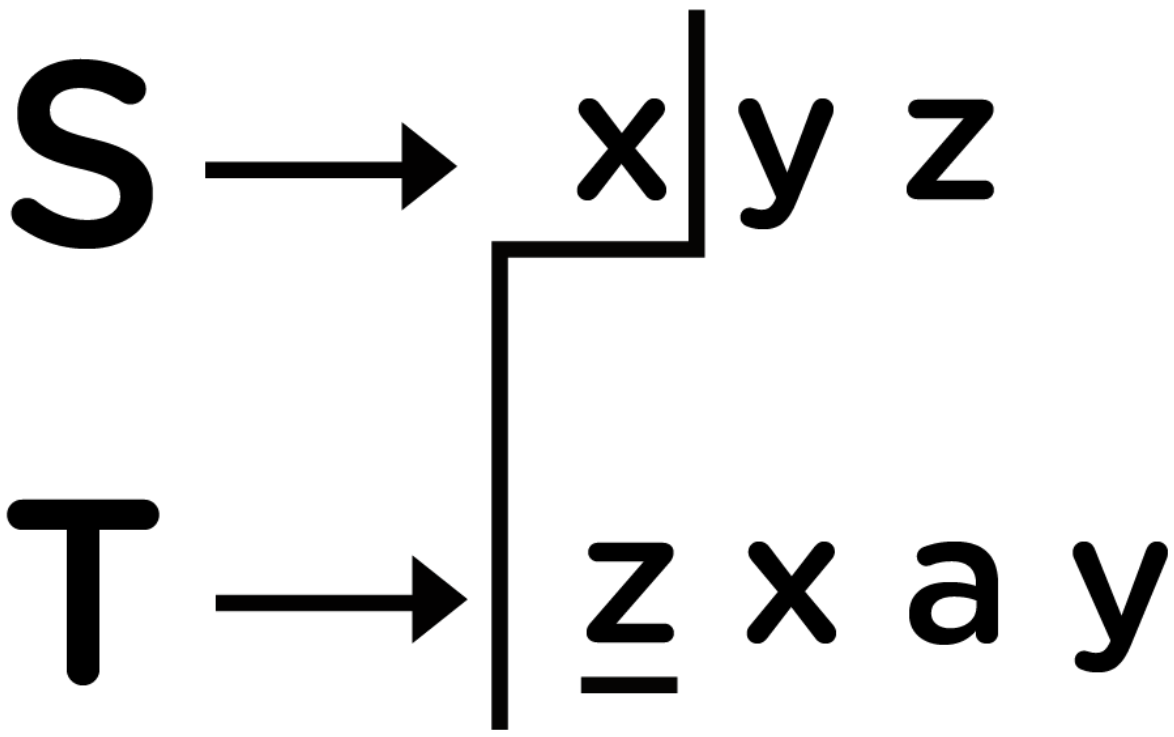
Here, the longest common subsequence is **cef**; hence the answer is 3 (the length of LCS).

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**"

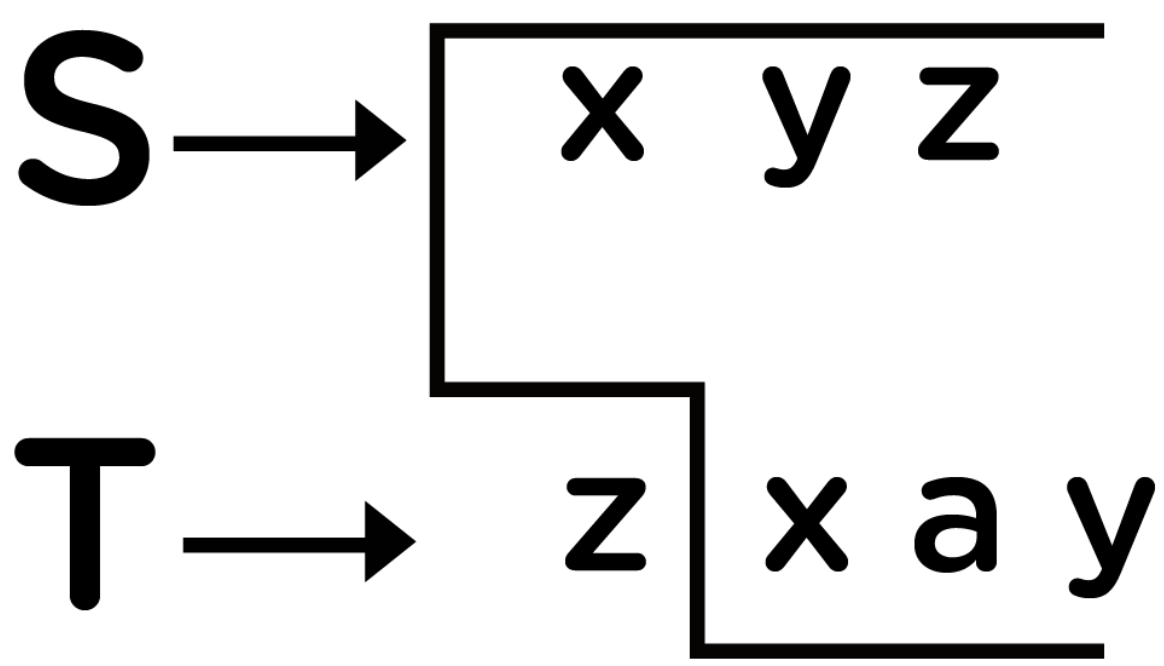
The rest of the LCS will be handled by recursion. But, if the first characters do not match, then we have to figure out that by traversing which of the following strings, we will get our answer. This can’t be directly predicted by just looking at them, so we will be traversing over both of them one-by-one 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:

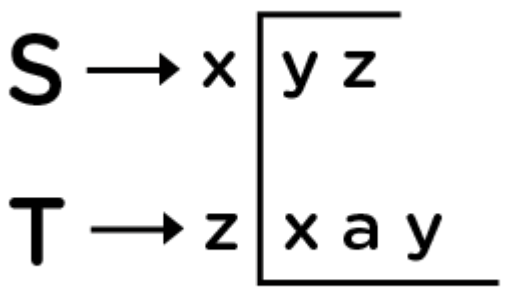
A =



B =



C =



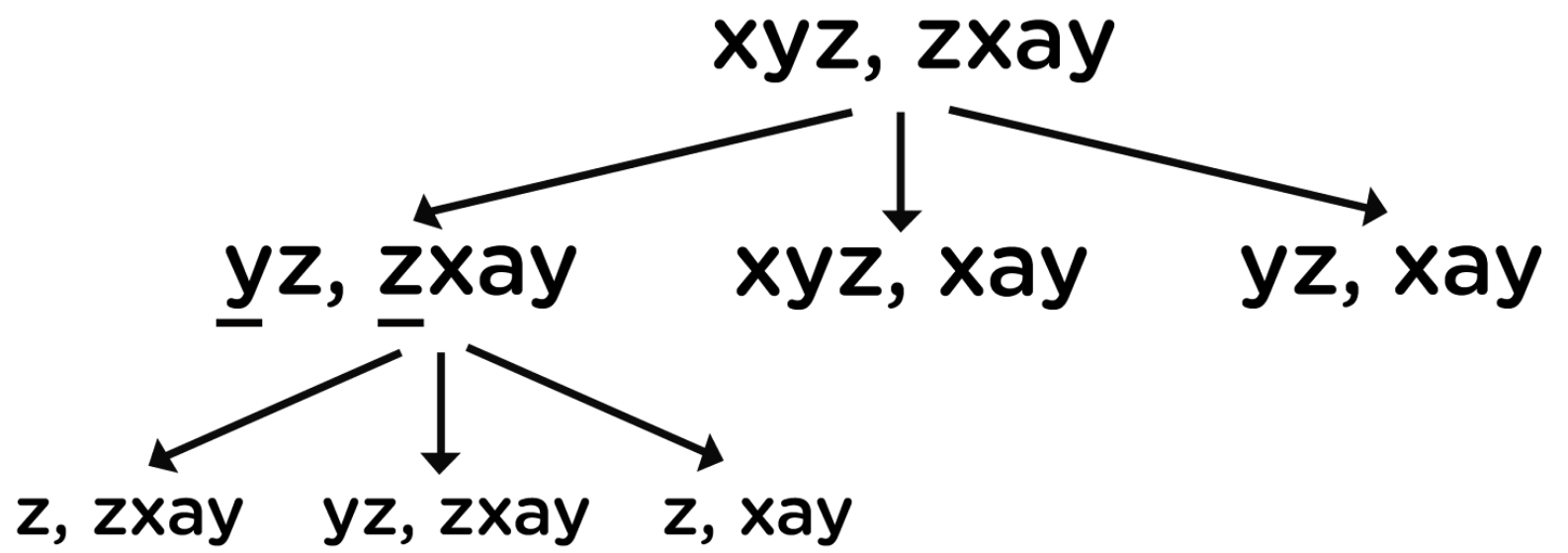
Finally, our answer will be: **LCS = max(A, B, C)**
Check the code below and follow the comments for a better understanding.

Pseudocode:

```
function lcs(s, t, m, n):
        //   Base Case
```

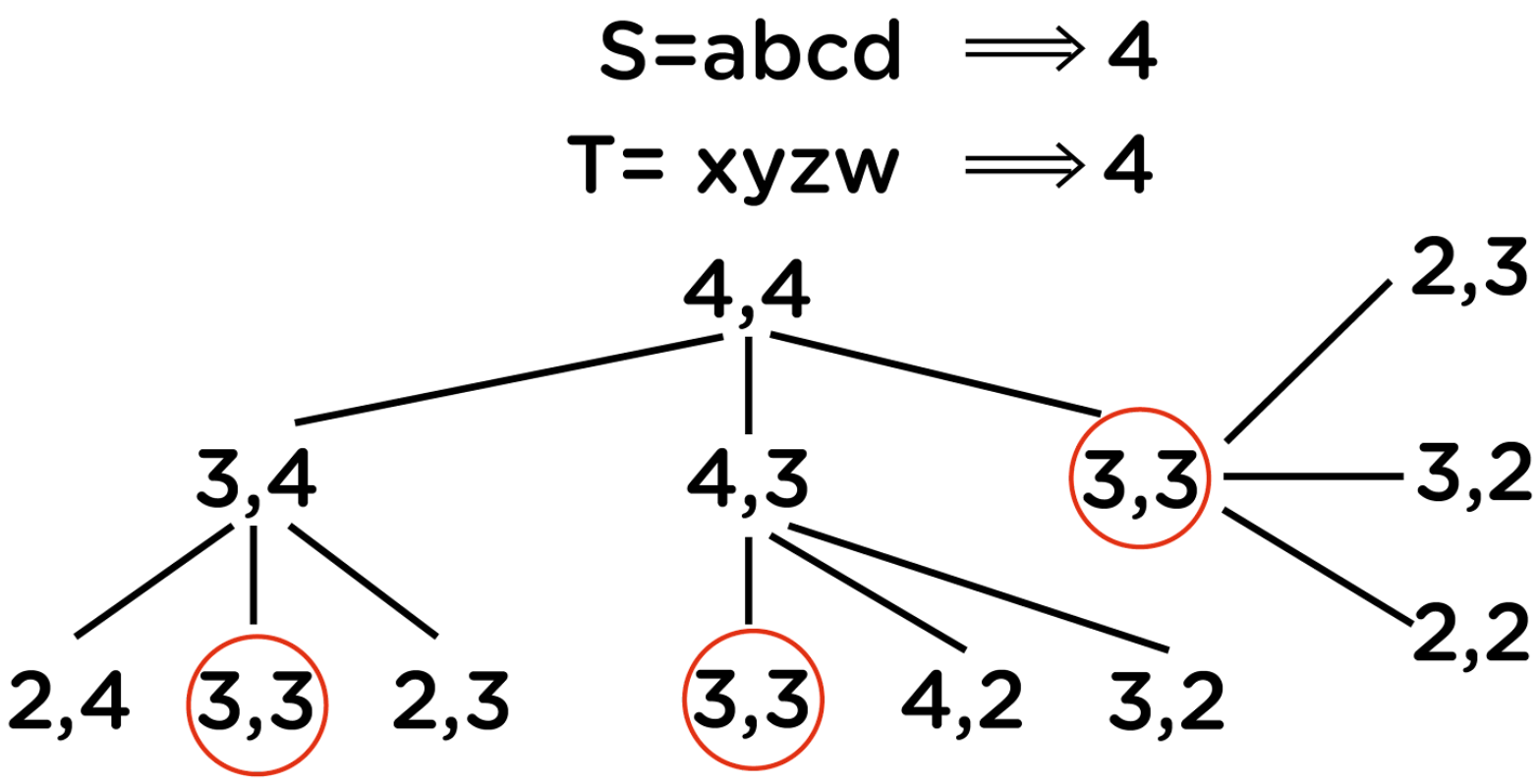
```
        if m equals 0 or n equals 0
                return 0

        //   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.

Previous
Next
HashMap Notes

HashMaps

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:

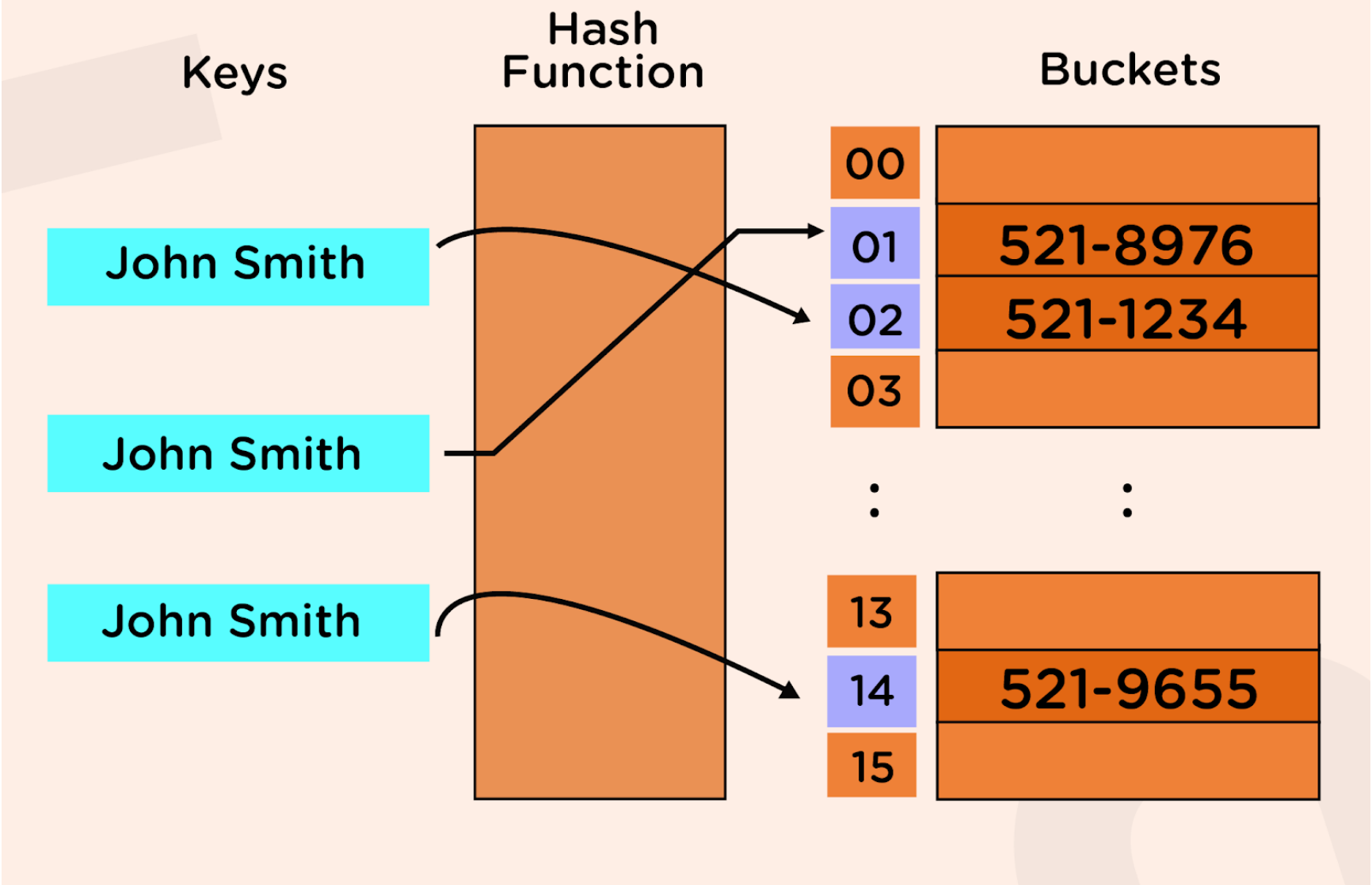
1. **Linked Lists:** To perform insertion, deletion, and search operations in the linked list, the time complexity will be O(n) for each as:
    1. For insertion, we will first have to check if the key already exists or not, if it exists, then we just have to update the value stored corresponding to 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:

$$\text{Compression\_function1} = 292 \% 2 = 0$$
$$\text{Compression\_function2} = 298 \% 2 = 0$$

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.

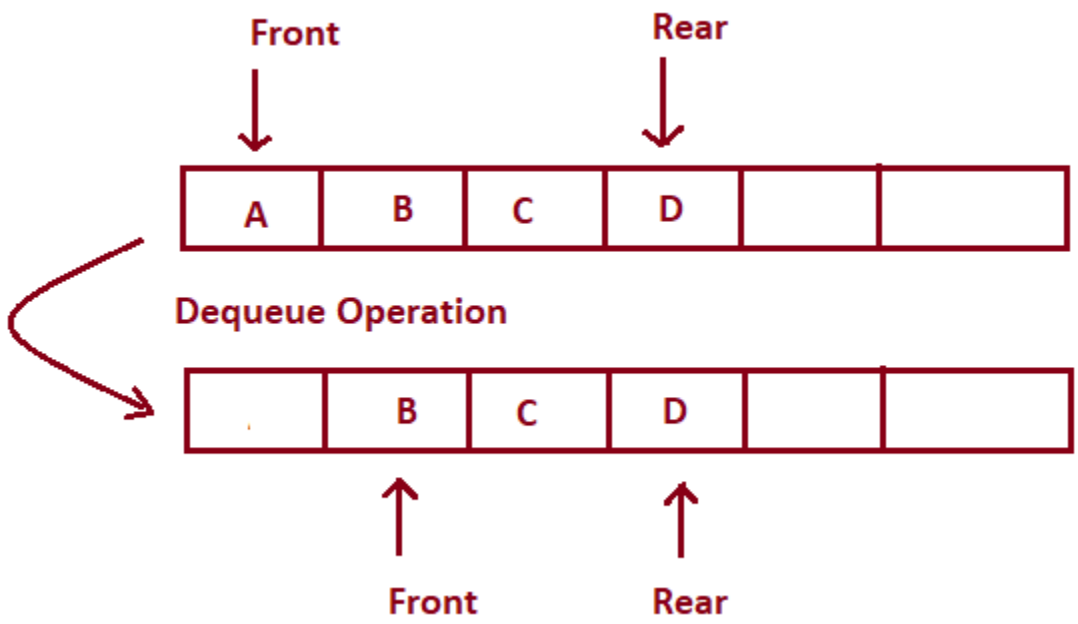
Applications of HashMaps

- 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.

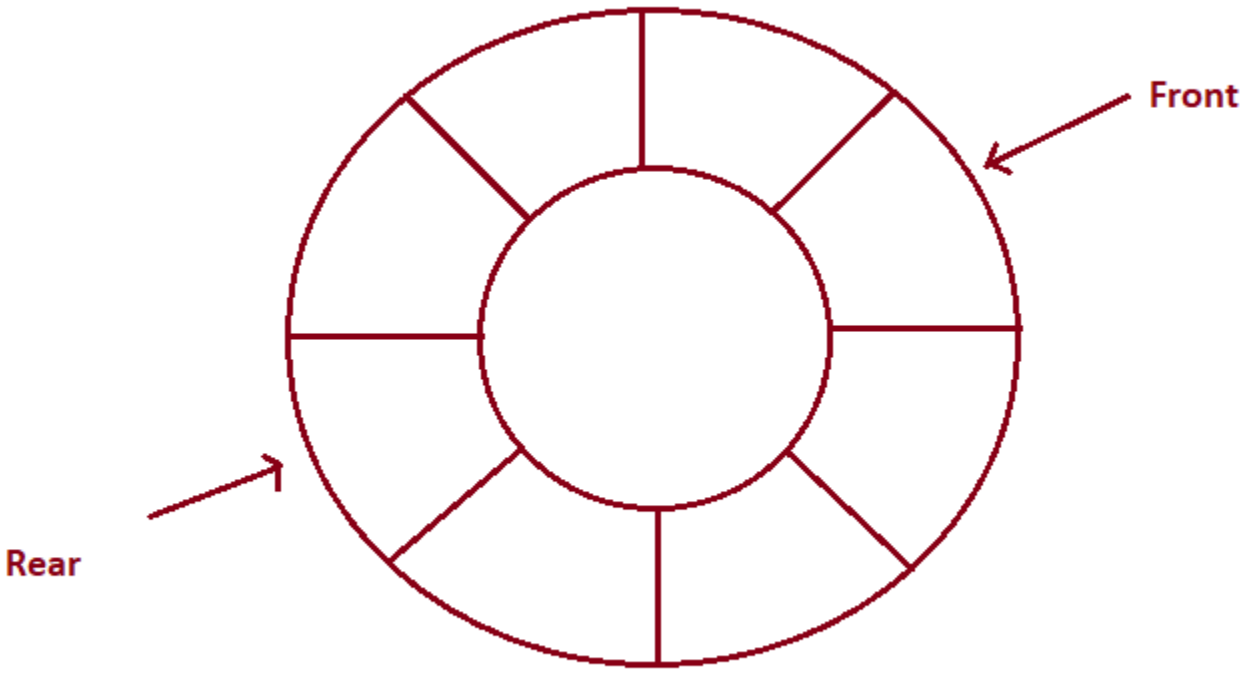
Previous
Next
Circular Queues Notes

Circular Queues

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.

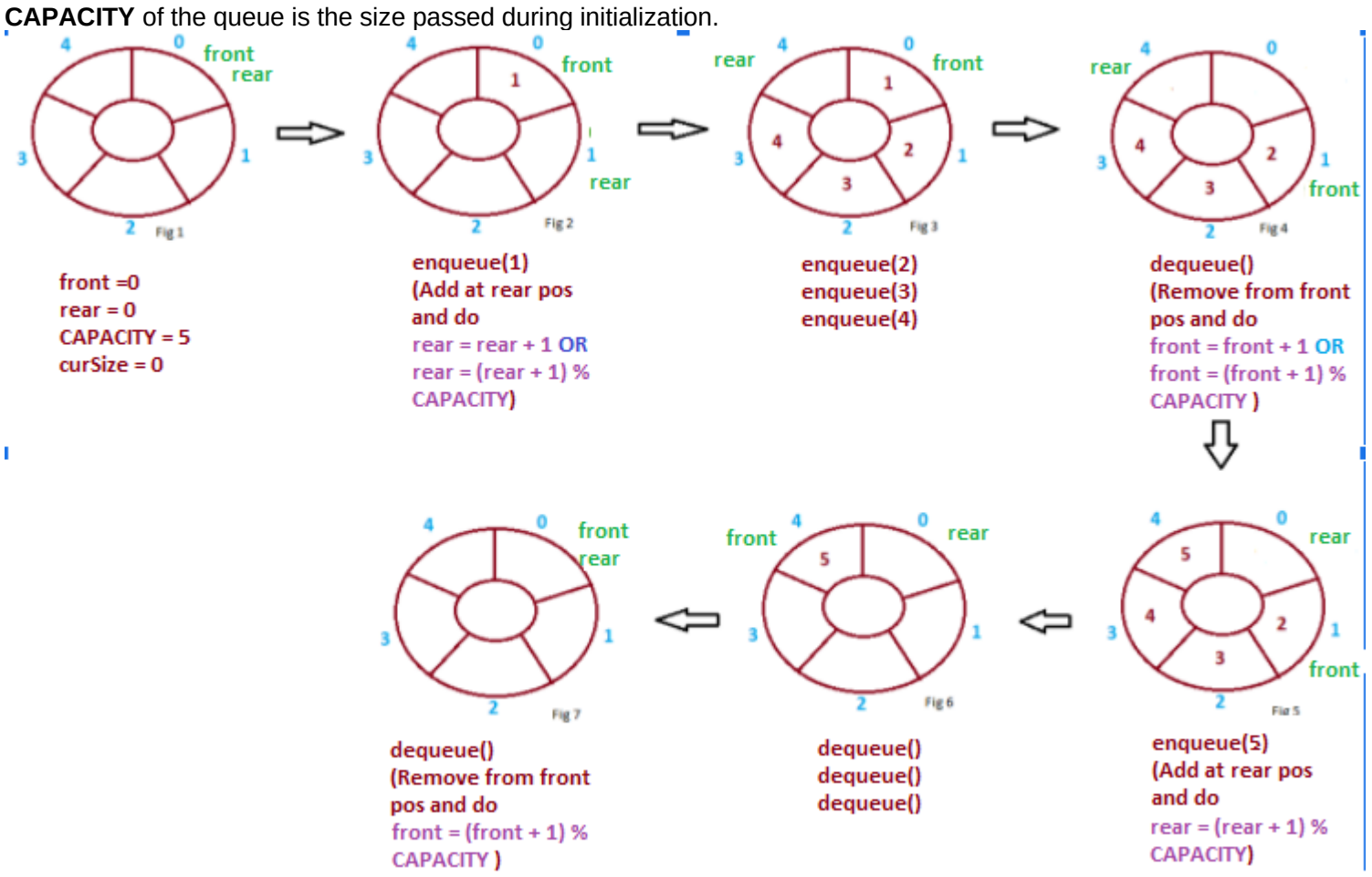


Fixed size Array

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)

Advantages of circular queues

- All operations occur in **constant time**.
- **Never** has to reorganize/copy data
- Circular queues are **memory efficient**.

Application of circular queues

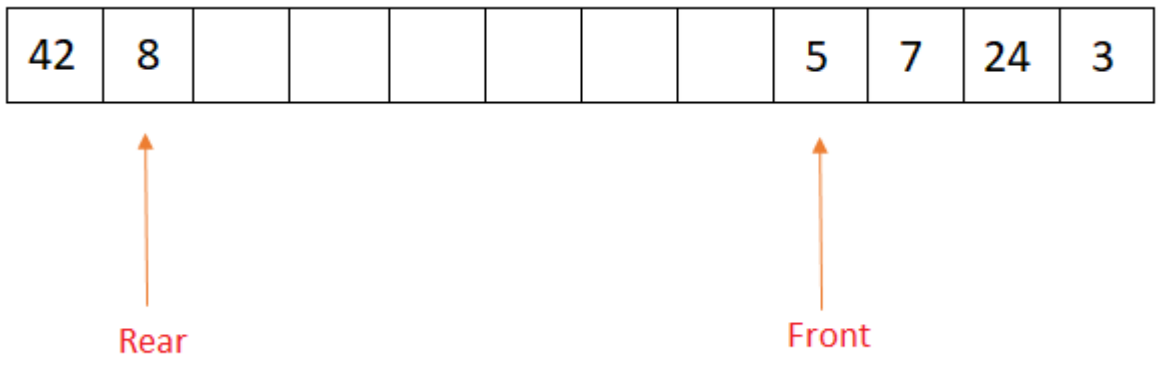
- It is used in the **looped execution** of slides of a presentation.
- It is used in browsing through the open windows applications using **alt + tab** (Microsoft Windows)
- It is used for **round-robin execution** of jobs in multiprogramming OS.
- Used in the functioning of **traffic lights**.
- Used 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.

Previous
Next
Deques Notes

Deques

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.

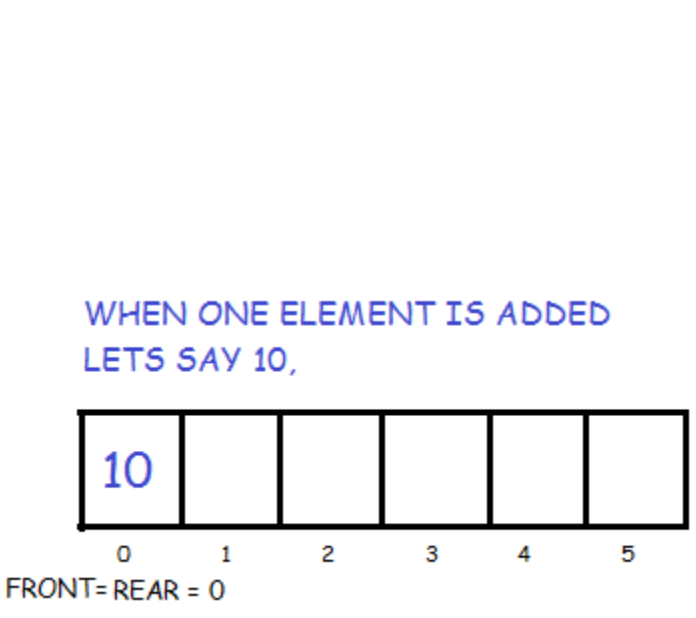
Implementation of Deques

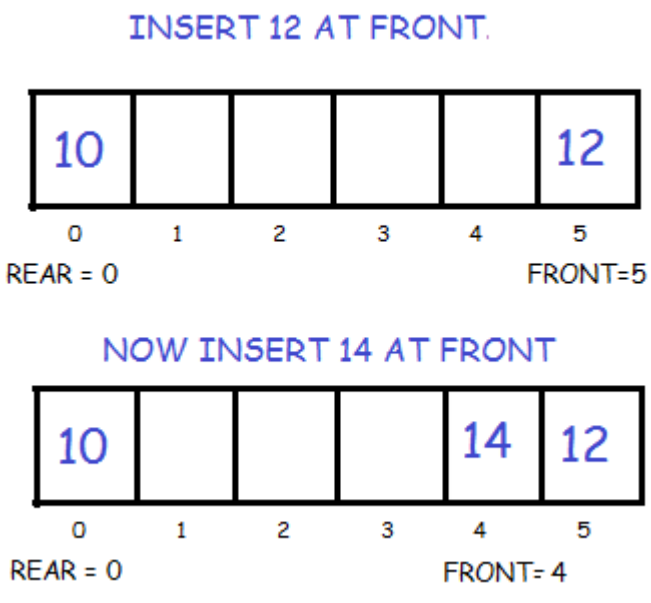
Deques can be implemented using data structures like **circular arrays** or **doubly-linked lists**. Below is the circular array implementation for deques, the same approach can be used to implement deque using doubly-linked lists.
We maintain two variables: **front** and **rear**, front represents the **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:

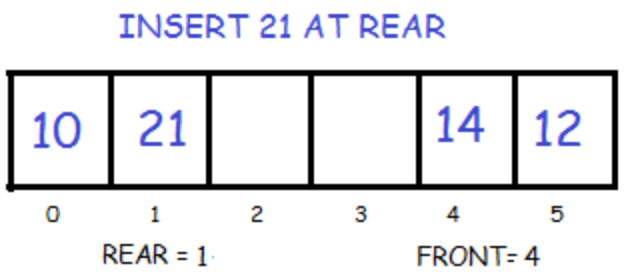
```
function enqueue_front(data)

        //   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:

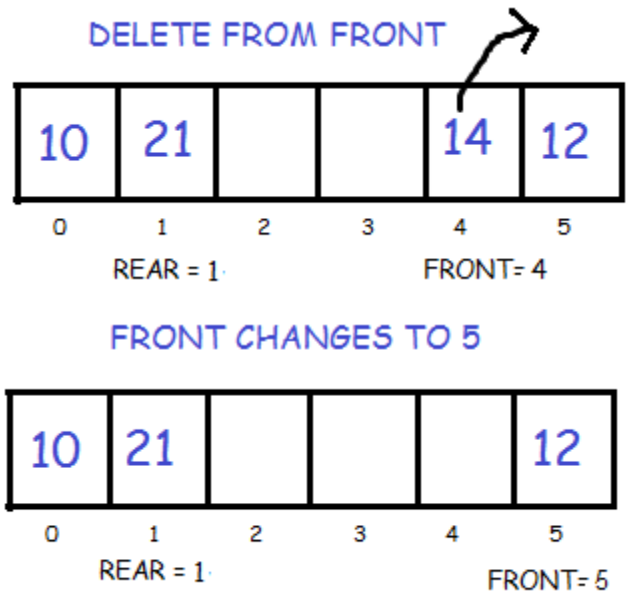
```
function enqueue_rear(data)

        //   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.



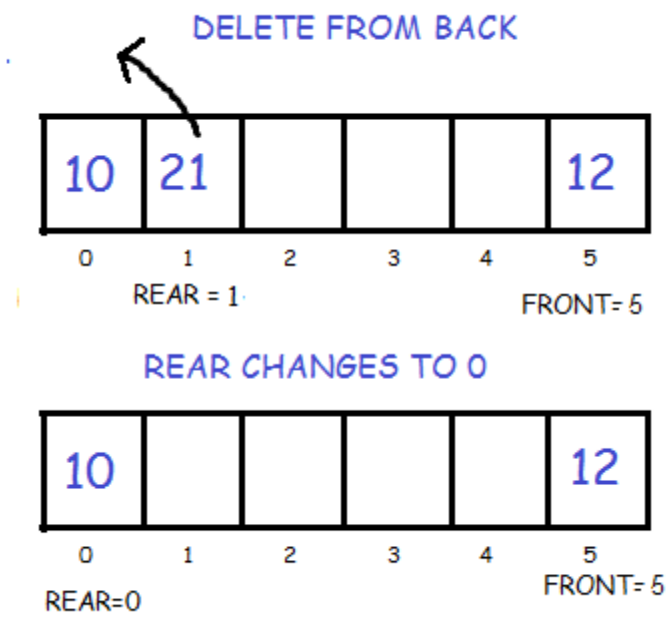
Pseudocode:

```
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)

Advantages of Deques

- Deques are powerful data structures that offer the functionalities of both stacks and queues.
- Highly useful for operations that involve addition/deletion of elements from either end, as all operations can be performed in constant O(1) time.

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.

Previous
Next
Tries Notes

Tries

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.

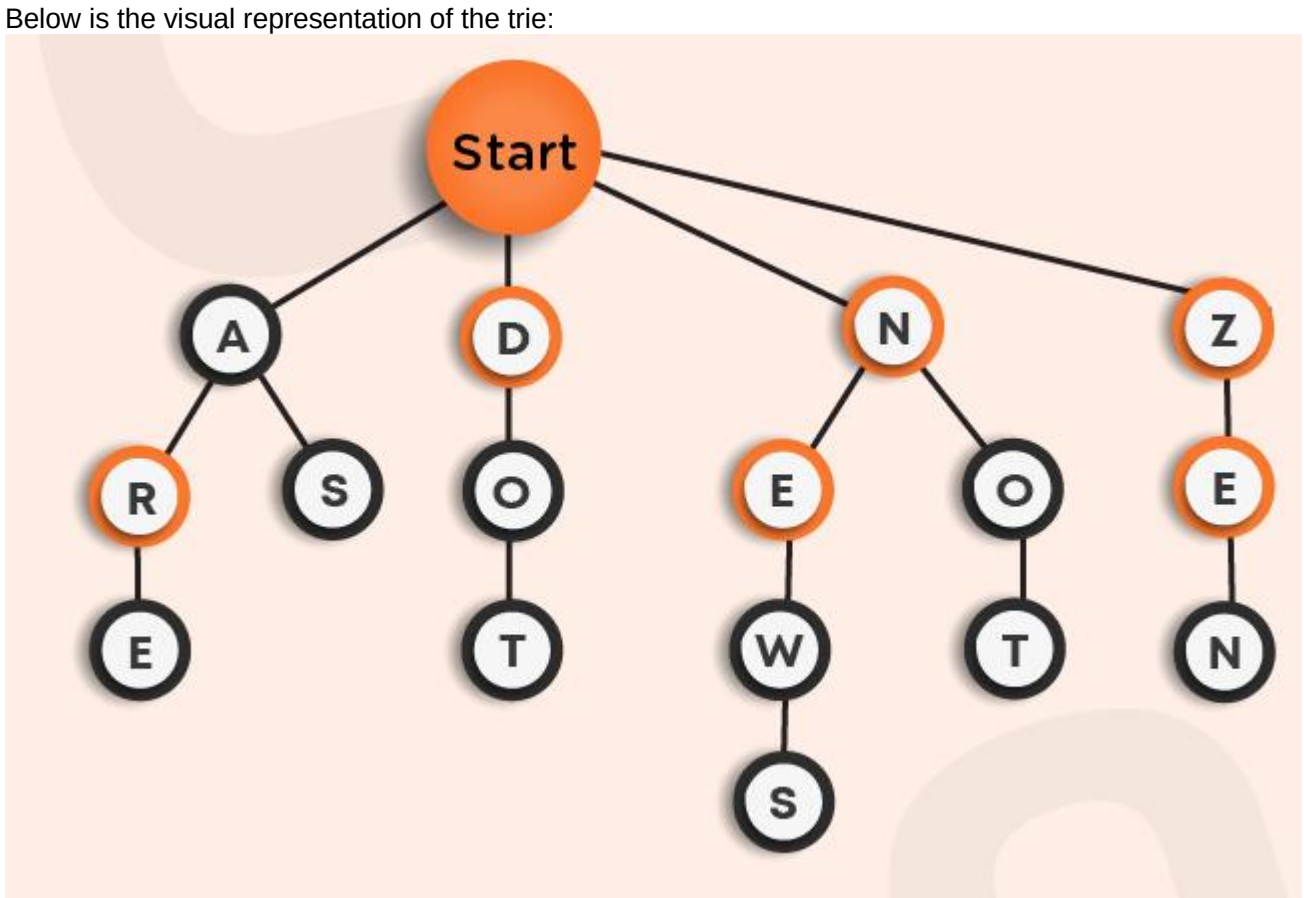
Suppose we want to insert string ***abc*** in our hashmap. To do so, first, we would need to calculate the hashcode for it, which would require the traversal of the whole string ***abc*****.** Hence, the time taken will be the length of the entire string irrespective of the time taken to perform any 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 every time 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 inserting 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.

When to use tries?

It can be observed that by using tries, we can improve the space complexity.

For example, We have 1000 words starting from character ***A*** that we want to store. Now, if you try to hold these words using a hashmap, then for each word, we have to store character ***A*** differently corresponding to different strings. But in case of tries, we only need to store the character ***A*** once. In this way, we can save a lot of space, and hence space optimization leads us to prefer tries over hashmaps in such scenarios.

In the beginning, we thought of implementing a dictionary. Let's recall a feature of a dictionary, namely **Auto-Search.** While browsing the dictionary, we start by typing a character. All the words beginning from that character appear in the search list. But this functionality can't be achieved using hashmaps as in the hashmap, the data stored is independent of each other, whereas, in the case of tries, the data is stored in the form of a tree-like structure. Hence, here also, tries prove to be efficient over hashmaps.

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.

Pseudocode:

```
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

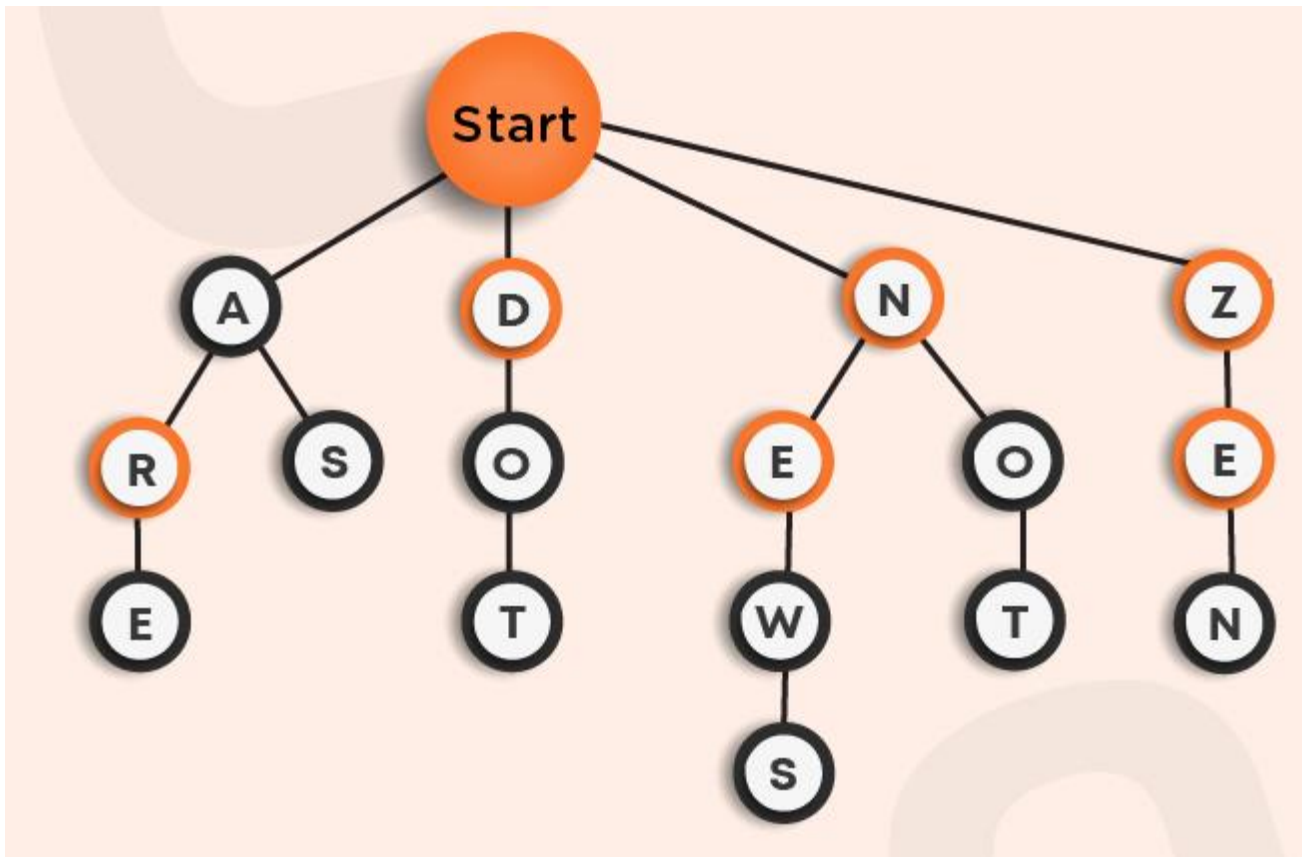
- Start from the root.
- For each character of the string, from first to the last check. if it has a child corresponding to itself in the root, and if not then return false.
- Descend from the root to the child corresponding to the current character.
- Return true on successful completion of the above loop if the last node is not null and its terminal property is true.

Pseudocode:

```
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.

Pseudocode:

```
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)

Types of tries

There are two types of tries:

Compressed tries:

- Majorly, used for space optimization.
- We generally club the characters if they have at most one child.
- **General rule:** Every node has at least two child nodes.

Pattern matching Tries:

- Used to match patterns in the trie.Example: In the **figure-1** (shown above), if we want to search for pattern **ben** in the trie, but the word **bend** was present instead, using the normal search function, we would return false, as the last character **n**'s **isTerminal** property was false, but in this trie, we would return true.
- To overcome this problem of the last character's identification, just remove the **isTerminal** property of the node.
- In the **figure-1**, instead of searching for the pattern **ben**, we now want to search for the pattern **en**. Our trie would return false if **en** is not directly connected to the root. But as the pattern is present in the word **ben**, it should return true. To overcome this problem, we need to attach each of the prefix strings to the root node so that every pattern is encountered. **For example:** for the string **ben,** we would store **ben**, **en**, **n** in the trie as the direct children of the root node.

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.

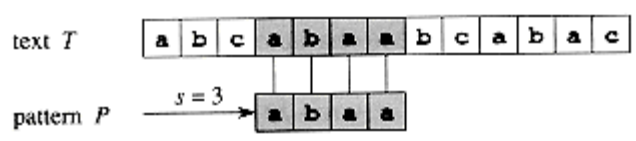
Previous
Next
String Algorithms Notes

String Algorithms

In this section we will primarily talk about two string searching algorithms, **Knuth Morris Pratt a**lgorithm and **Z - Algorithm**.

Introduction

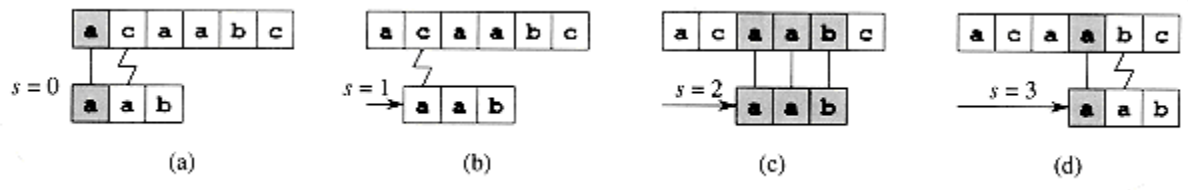
Suppose you are given a string text of length n and a string pattern of length m. You need to find all the occurrences of the pattern in the text or report that no such instance exists.



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
```

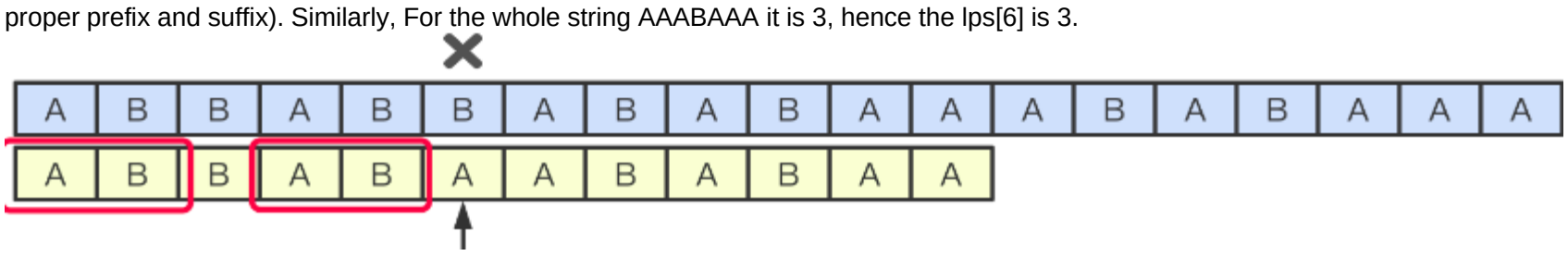
Knuth Morris Pratt Algorithm

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

Previous
Next