End-Term Report

Data Structures and Algorithm

PUSHPENDRA UIKEY

23B1023

Mentor: **Gaurav Koli**

July 28, 2024

Contents

Data Structures and Algorithm (DSA)

Data Structures and Algorithm (DSA) refer to the study of methods for organizing and storing data and the design of procedures (algorithms) for solving problems, which operate on these data structures.

Data Structures are essential components that help organize and store data efficiently in computer memory. They provide a way to manage and manipulate data effectively, enabling faster access, insertion, and deletion operations. Common data structures include arrays, linked lists, stacks, queues, trees, and graphs, each serving a specific purpose based on the requirements of the problem at hand.

1 Components of a C++ Code

1. **Comments:** Written after '//' for one line or '/*comment*/' for multiline.
2. **#include<bits/stdc++.h>:** Necessary to include various libraries including the input/output library in C++.
3. **using namespace std:** All the elements in the standard C++ library are declared within a namespace.
4. **int main():** The execution of any C++ program starts with the main function.
5. **{}:** Curly braces are used to specify the start and end of any function in C++.
6. **return 0:** Specifies the end of the function.

2 Data Types in C++

Data Type	Meaning	Size (in Bytes)
int	Integer	4
long long int	Long integer	8
float	Floating-point	4
double	Double Floating-point	8
char	Character	1
wchar_t	Wide Character	2
bool	Boolean	1
void	Empty	0

3 Arithmetic Operations in C++

Operator	Operation
+	Adds two operands
-	Subtracts right operand from the left operand
*	Multiplies two operands
/	Divides left operand by right operand
%	Finds the remainder after integer division
++	Increment
−−	Decrement

4 Relational Operations in C++

Operator	Operation
==	Gives true if two operands are equal
!=	Gives true if two operands are not equal
<	Gives true if the left operand is more than the right operand
>	Gives true if the left operand is less than the right operand
<=	Gives true if the left operand is more than the right operand or equal to it
>=	Gives true if the left operand is less than or equal to the right operand

5 Logical Operators in C++

Operator	Operation
&&	AND operator. Gives true if both operands are non-zero
||	OR operator. Gives true if at least one of the two operands are non-zero
!	NOT operator. Reverses the logical state of the operand

6 Bitwise Operators in C++

Operator	Operation
&	Binary AND. Copies a bit to the result if it exists in both operands.
|	Binary OR. Copies a bit if it exists in either operand.
^	Binary XOR. Copies the bit if it is set in one operand but not both.
~	Binary One's Complement. Flips the bits.
<<	Binary Left Shift. The left operand's bits are moved left by the number of places specified by the right operand.
>>	Binary Right Shift. The left operand's bits are moved right by the number of places specified by the right operand.

7 Assignment Operator in C++

Symbol	Description
$=$	Assigns the value on the right to the variable on the left
$+=$	First adds the current value of the variable on the left to the value on the right and then assigns the result to the variable on the left
$-=$	First subtracts the value on the right from the current value of the variable on the left and then assigns the result to the variable on the left
$*=$	First multiplies the current value of the variable on the left by the value on the right and then assigns the result to the variable on the left
$/=$	First divides the current value of the variable on the left by the value on the right and then assigns the result to the variable on the left

8 Time Complexity:

The time complexity of an algorithm is the amount of time it takes to complete, typically measured in terms of the number of operations performed relative to the size of the input. It provides an estimation of the worst-case scenario, best-case scenario, or average complexity of an algorithm without necessarily running the code. The notations commonly used are:

- **Big-O (O):** Describes the worst-case scenario, providing an upper time bound of the algorithm.
- **Omega (Ω):** Describes the best-case scenario, offering a lower time bound of the algorithm.
- **Theta (Θ):** Represents the average complexity of the algorithm.

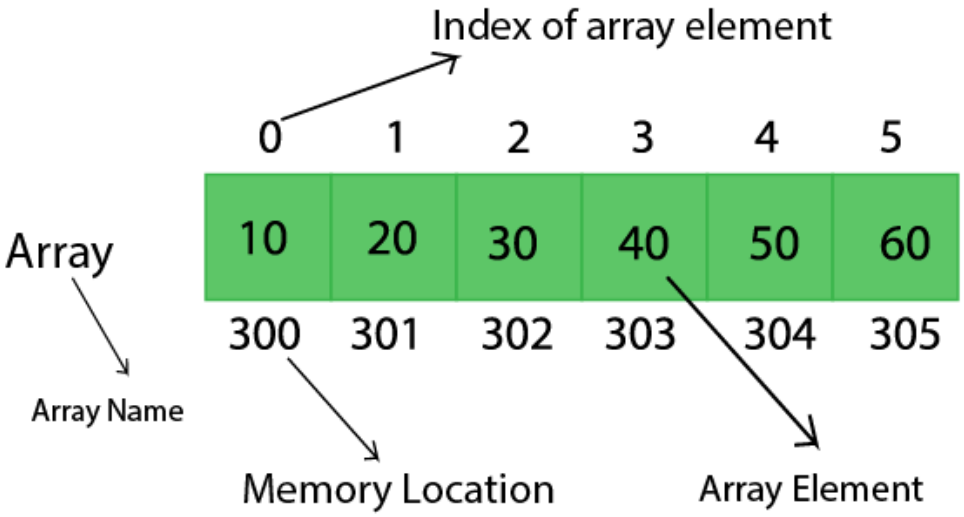
Space Complexity:

Space complexity refers to the amount of memory an algorithm needs to execute and complete its task. It measures the maximum amount of memory space required by the algorithm, typically in terms of auxiliary space (additional space other than input data).

9 Data Structures

9.1 Array

Array is a linear data structure that stores a collection of elements of the same data type. Elements are allocated contiguous memory, allowing for constant time access. Each element has a unique index number.



9.1.1 Operations on Arrays

- **Traversal:** Iterating through the elements of the array.
- **Insertion:** Adding an element to the array at a specific index.
- **Deletion:** Removing an element from a specific index.
- **Searching:** Finding an element of the array.

9.1.2 Types of Array

- **One-dimensional array:** A simple array with a single dimension.
- **Multidimensional array:** An array with multiple dimensions, such as a matrix.

9.1.3 Applications of Arrays

- Storing data in a sequential order.
- Representing matrices and tables.

9.2 String

A string is a sequence of characters, typically used to represent text. It is considered a data type that allows for manipulation and processing of textual data in computer programs.

9.2.1 Operations on Strings

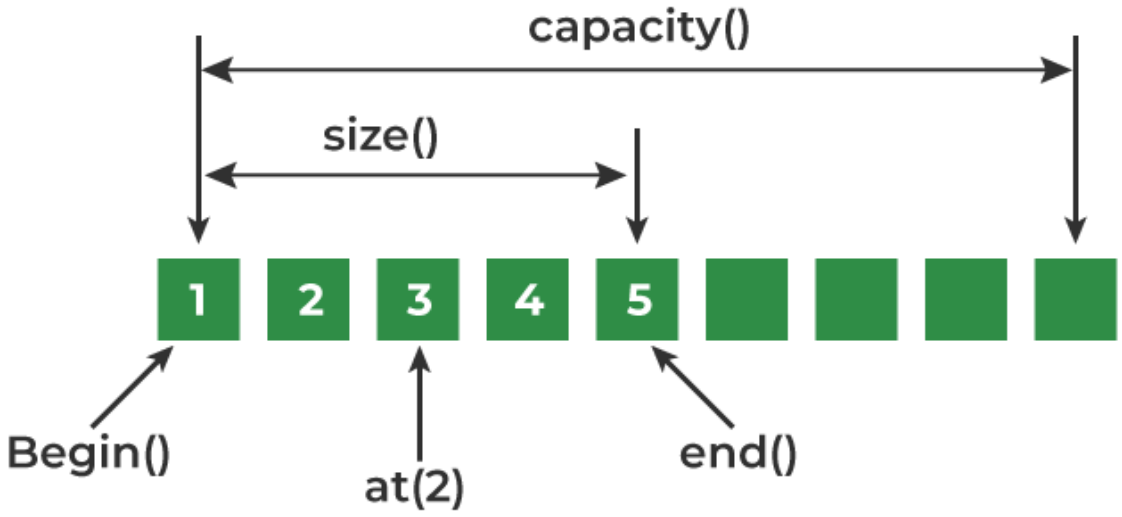
- **Concatenation:** Joining two strings together.
- **Comparison:** Comparing two strings lexicographically.
- **Substring extraction:** Extracting a substring from a string.
- **Search:** Searching for a substring within a string.
- **Modification:** Changing or replacing characters within a string.

9.2.2 Applications of Strings

- Text processing
- Pattern matching
- Data validation
- Database management

9.3 Vector

Vectors are same as dynamic arrays with the ability to resize themselves automatically when an element is inserted or deleted. Their storage is handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.



9.3.1 Operation on Vector

- **push_back**: Push an element into vector from back.
- **pop_back**: Used to remove element from the back.
- **erase**: used to remove elements from the specified position or range.
- **empty**: Checks if vector is empty.
- **size**: returns the number of elements in the array.
- **front**: returns a reference to the first element in the vector.
- **back**: returns a reference to last element in the vector.

9.3.2 Applications of Vector

- Dynamic Arrays
- Data Storage
- Function Parameters
- Sorting and Searching
- Efficient insertions and Deletions

9.4 Pair

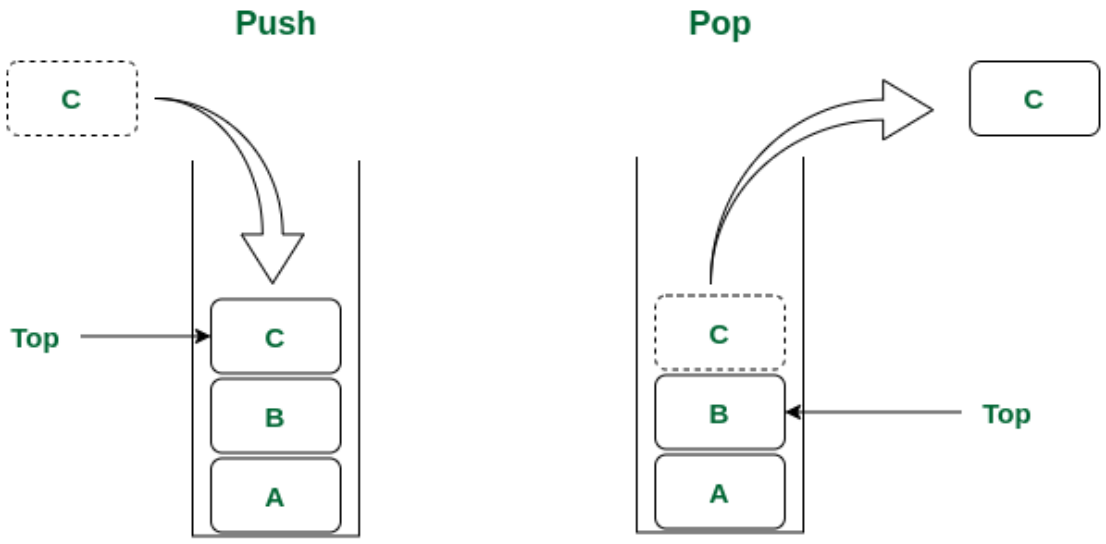
Pair is used to combine together two values that may be of different data types. It is basically used if we want to store tuples.
The first element is referenced as 'first' and the second element as 'second' and the order is fixed (first, second)

9.4.1 Operation on Pair

- **make_pair**: To create a value pair without writing the types explicitly.
- **swap**: The function swaps the content of one pair with another pair object.
- All the assignment operations can be done to this.

9.5 Stack

Stack is a linear data structure that follows a particular order in which operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out). LIFO implies that the element that is inserted last comes out first, and FILO implies that the element that is inserted first comes out last.



Stack Data Structure

9.5.1 Operations on Stack

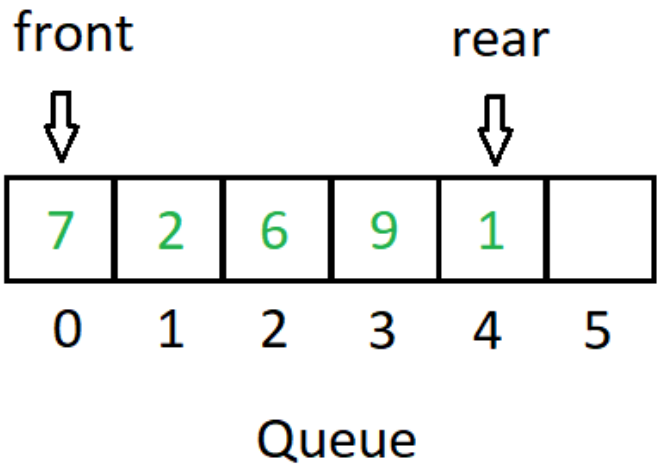
- **Push:** Adds an element to the top of the stack.
- **Pop:** Removes and returns the element at the top of the stack.
- **Peek:** Returns the element at the top of the stack without removing it.
- **Size:** Returns the number of elements in the stack.
- **IsEmpty:** Checks if the stack is empty.

9.5.2 Applications of Stack

- **Function calls.**
- **Expression evaluation.**
- **Backtracking.**
- **Undo/redo operations.**

9.6 Queue

Queue Data Structure is a linear data structure that follows FIFO (First In First Out) Principle, so the first element inserted is the first to be popped out.
Queue is a kind of list in which all the insertions are made from one end(back of queue) and all the deletion are at the other end(front of queue).



9.6.1 Types of Queue

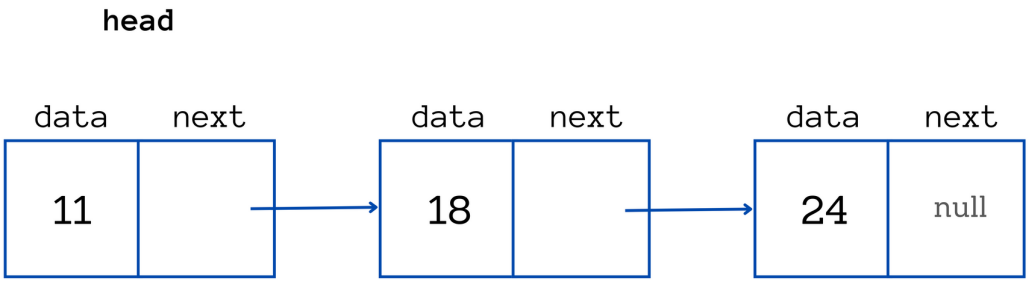
- **Simple queue**: Simply follows FIFO structures.
- **Double-Ended queue**: Insertion and deletion both can be followed from both the end.
- **Circular queue**: Special queue in which last position is connected back to the first position.
- **Priority queue**: A priority queue is a special queue where the elements are accessed on the basis of priority assigned to them.

9.6.2 Operations on Queue

- **Enqueue**: Adds an element to the end of queue.
- **Dequeue**: Remove an element from the front of queue.
- **Peek or front**: Acquires the data from the front node of queue without deleting it.
- **isEmpty**: checks if the queue is empty.

9.7 Linked List

A linked list is a linear data structure that consists of a series of nodes connected by pointers. Each node contains data and a pointer to the next node in the list. Nodes are not stored contiguously in memory.



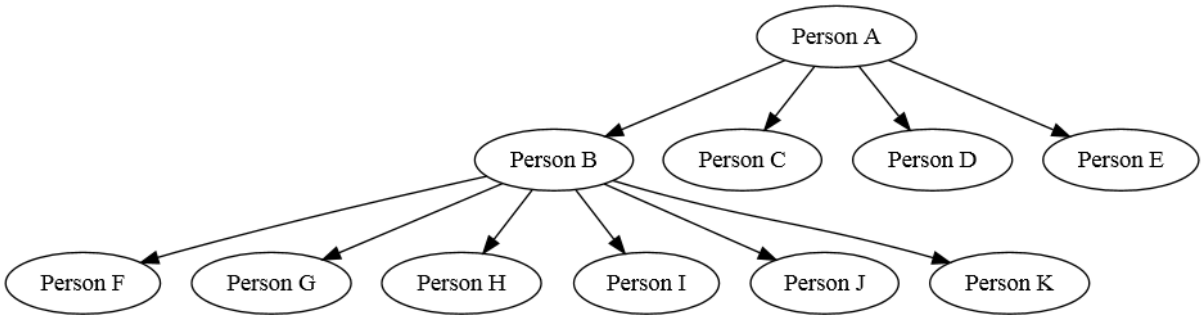
9.7.1 Types of Linked List

- **Singly Linked List**
- **Doubly Linked List**
- **Circular Linked List**
- **Circular doubly linked list**

9.8 Tree

A tree data structure is a hierarchical framework used to represent and organize data for efficient navigation and searching. It is composed of nodes connected by edges, establishing a hierarchical relationship among the nodes.

The topmost node is known as the root, and the nodes directly connected to the root are called child nodes. Each node can have multiple child nodes, which can further have their own child nodes, creating a recursive structure.



9.8.1 Terminology in Tree

- **Parent Node**: Node which is a predecessor of a node is called the parent node of that node.
- **Child Node**: The node which is immediate successor of a node is called child node of that node.
- **Root Node**: The topmost node of a tree or the node which does not have any parent node.

- **Leaf Node**: The nodes which do not have any child node are called leaf node.

9.9 Binary Tree

A Binary Tree Data Structure is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child. It is commonly used in computer science for efficient storage and retrieval of data, with various operations such as insertion, deletion, and traversal.
Each node has of Binary Tree has three parts:

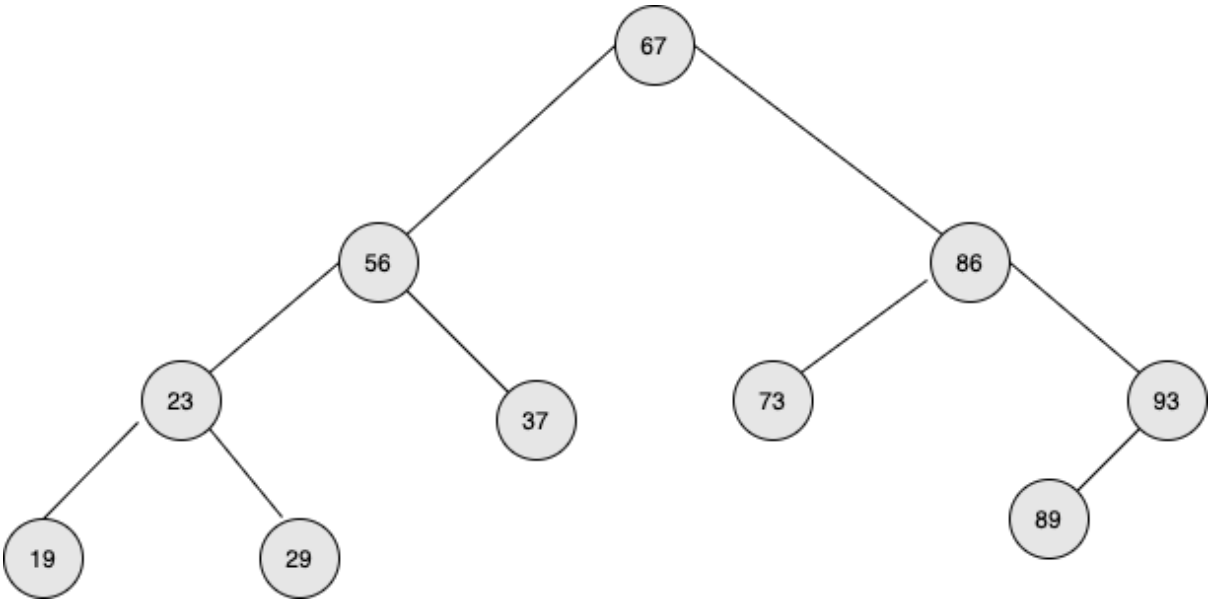
- Data
- Pointer to left child
- Pointer to right child

9.9.1 Operation on Binary Tree

- Insertion in Binary Tree
- Traversal of Binary Tree
- Deletion of node
- Searching

9.10 Binary Search Tree

Binary Search Tree is a data structure used in computer science for organizing and storing data in a sorted manner. Binary search tree follows all properties of binary tree and its left child contains values less than the parent node and the right child contains values greater than the parent node.



Handling duplicate values in the Binary Search Tree:

- We must follow a consistent process throughout i.e. either store duplicate value at the left or store the duplicate value at the right of the root, but be consistent with your approach.

9.10.1 Operations on Binary Search Tree

- Searching a Node in BST
- Insert a node into a BST
- Delete a node of BST
- Inorder Traversal of BST

9.11 Heaps

A heap is a binary tree-based data structure that satisfies the heap property: the value of each node is greater than or equal to the value of its children. This property makes sure that the root node contains the maximum or minimum value (depending on the type of heap), and the values decrease or increase as you move down the tree.

9.11.1 Types of heaps

- **Max Heap**: The root node contains the maximum value, and the values decrease as you move down the tree.
- **Min Heap**: The root node contains the minimum value, and the values increase as you move down the tree.

9.11.2 Heaps Operations

- Insert: Adds a new element to the heap while maintaining the heap property.
- Extract Max/Min: Removes the maximum or minimum element from the heap and returns it.
- Heapify: Converts an arbitrary binary tree into a heap.

9.12 Graph

Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. A graph is composed of a set of vertices(V) and a set of edges(E).

9.12.1 Components of a Graph

- **Vertices**: Vertices are the fundamental units of graph. Sometimes vertices are also known as vertex or nodes.
- **Edges**: Edges are used to connect two nodes of the graph. Edges can connect any two nodes in any possible ways.

9.12.2 Operations on Graphs

- Insertion of nodes/edges in the graph
- Deletion of nodes/edges in the graph
- Searching on graphs
- Traversal of graphs

9.13 Maps

Maps are associative containers that stores elements in a mapped fashion. Each value has a key-value and a mapped value. No two mapped values can have the same key values.

9.13.1 map member function

- **begin()**: Returns an iterator to the first element in the map.
- **end()**: Returns an iterator of the position after the of the last element in the map.
- **size()**: Returns the number of elements in the map.
- **empty()**: tells whether the map is empty.
- **clear()**: Removes all the elements from the map.

10 Algorithms

10.1 Searching Algorithms

Searching algorithms are used to locate specific data within a larger set of data. They help find the presence of a target value within the data. There are various types of searching algorithms, each with its own approach and efficiency.

10.1.1 Common Searching Algorithms

- **Linear Search**: Iteratively searches from one end of the list to the other, checking each element to see if it matches the target value.
- **Binary Search**: A divide-and-conquer search method applicable to sorted arrays.

  Binary search is a search algorithm used to find the position of a target value within a sorted array. It works by repeatedly dividing the search interval in half until the target value is found or the interval is empty. The search interval is halved by comparing the target element with the middle value of the search space. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to $O(\log N)$.

  In any search problem, the basic motive is to reduce the decision space progressively. The more aggressively the search space is reduced, the more efficient the algorithm. To reduce decision space means to eliminate certain portions completely from the search in the future.

  To apply the Binary Search algorithm:

  - The data structure must be sorted.
  - Access to any element of the data structure takes constant time.

10.2 Sorting Algorithms

Sorting algorithms are used to arrange the elements of a list in a specific order, such as numerical or alphabetical. They organize the items systematically, making it easier to search for and access specific elements.

10.2.1 Common Sorting Algorithms

Bubble Sort Bubble Sort iteratively compares adjacent elements in the list and swaps them if they are out of order. With each pass through the list, the largest element ”bubbles” to the end. This process repeats until the entire list is sorted.

- Time Complexity: $O(n^2)$
- Space Complexity: $O(1)$

Selection Sort Selection Sort repeatedly finds the minimum element from the unsorted portion of the list and swaps it with the first element. It continues this process by progressively moving the boundary between the sorted and unsorted sections of the list, sorting the entire list.

- Time Complexity: $O(n^2)$
- Space Complexity: $O(1)$

Insertion Sort Insertion Sort builds the sorted list one element at a time by taking each unsorted element and inserting it into its correct position within the sorted portion. This algorithm is efficient for small datasets and nearly sorted data.

- Time Complexity: $O(n^2)$
- Space Complexity: $O(1)$

Merge Sort Merge Sort follows a divide-and-conquer strategy. It recursively divides the list into smaller sublists until each sublist contains a single element. Then, it merges these sublists in a way that results in a sorted list. This method ensures a consistently efficient sorting process.

- Time Complexity: $O(n \log n)$
- Space Complexity: $O(n)$

Quick Sort QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array. The key process in quickSort is a partition. The target of partitions is to place the pivot at its correct position in the sorted array and put all smaller elements to the left of pivot, and all greater elements to the right of pivot.

- Time Complexity: $O(n \log n)$
- Space Complexity: $O(n)$

10.3 Recursion Algorithms

Recursion is technique used in computer science to solve big problems by breaking them into smaller, similar problems. The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.
Recursion works by creating a stack of function calls. When a function calls itself, a new instance of the function is created and pushed onto the stack. This process continues until a base case is reached, which is a condition that stops the recursion. Once the base case is reached, the function calls start popping off the stack and returning their results.
Recursive algorithms have two parts:

- **Base Case**: Condition that stops the function.
- **Recursive Case**: Call to itself with a smaller version of itself.

10.4 Backtracking

Backtracking algorithms are like problem-solving strategies that help explore different options to find the best solution.
A backtracking algorithm works by recursively exploring all possible solutions to a problem. It starts by choosing an initial solution, and then it explores all possible extensions of that solution. If an extension leads to a solution, the algorithm returns that solution. If an extension does not lead to a solution, the algorithm backtracks to the previous solution and tries a different extension.

10.4.1 How it work?

- Choose an initial solution.
- Explore all possible extensions of current solutions.
- If an extension leads to solution, return that solution.
- If solution doesn't lead to the solution then backtrack to the previous solution and try a different extension.
- Repeat above steps until all the possibilities have been explored.

10.5 Divide and Conquer

Divide and Conquer algorithm is a problem-solving strategy in which we break down a complex problem into smaller, more manageable parts, solving each part individually, and then combining the solutions to solve the original problem.

1) Divide

- Break the original problem into smaller subproblems, which are the part of the original problem.
- Divide the problem until no further division possible.

2) Conquer

- Solve each of smaller subproblems individually.
- if we reach the base case it is solved directly without using recursion.

3) Merge

- Combine the subproblems to get the final solution of the whole problem.
- Once the smaller subproblems are sovled, we recursively combine their solutions to get the solutions of larger problem.

10.6 Common Mathematical Algorithms

GCD and LCM Find the greatest common divisor (GCD) and least common multiple (LCM) of two numbers.

Prime Factorization Decompose a number into its prime factors.

Fibonacci Numbers Generate the Fibonacci sequence, where each number is the sum of the two preceding ones.

Catalan Numbers Count the number of valid expressions with a given number of pairs of parantheses.

Modular Arithmetic Perform arithmetic operations on numbers modulo a given value.

nCr Computations Calculate the binomial coefficient, which represents the number of ways to choose r elements from the set of n elements.

Prime Numbers and Primality Tests Determine whether a given number is prime and find prime numbers efficiently.

Sieve Algorithms Find all Prime numbers up to a given limit using coefficient techniques like the Sieve of Eratosthenes.

Binary Exponentiation Calculates x raise to power n efficiently especially where the exponent n is very large.

10.7 Traversal Algorithms

10.7.1 Binary Tree

- **Breadth-First Search**: Level order traversal: idea is to go through each level beginning from root till leaf.
- Level order traversal of Binary Tree in spiral form using recursion: The idea is to first calculate the height of the tree, then recursively traverse each level and print the level order traversal according to the current level.
- Reverse Level Order Traversal: Same as level order traversal but leaf level will be traversed first and root level will be the last.
- **Depth first traversal**:
  1) Inorder Traversal:
    - Traverse the left subtree
    - visit the root
    - Traverse the right subtree

  1) Preorder Traversal:
    - visit the root
    - Traverse the left subtree
    - Traverse the right subtree

  1) Postorder Traversal:
    - Traverse the left subtree
    - Traverse the right subtree
    - visit the root
- **Morris Traversal**: Using Morris Traversal we can traverse through tree without using stack or recursion which gives us the benefit of O(1) space complexity.

- Diagonal Traversal: idea is to traverse through all the element in the same diagonal and then go to the next diagonal.
- Boundary Traversal: Traverse through left boundary(left most nodes in tree) then leaf boundary(all the leaf nodes) then right boundary(right most nodes of tree).

10.8 STL Algorithms

1. **sort(first_iterator, last_iterator)** Sorts the given vector.
2. **sort(first_iterator, last_iterator, greater<int>())** To sort the given container/vector in descending order.
3. **reverse(first_iterator, last_iterator)** To reverse a vector. (if ascending $\rightarrow$ descending OR if descending $\rightarrow$ ascending)
4. **max_element(first_iterator, last_iterator)** Finds the maximum element of a vector.
5. **min_element(first_iterator, last_iterator)** Finds the minimum element of a vector.
6. **accumulate(first_iterator, last_iterator, initial value of sum)** Does the summation of vector elements.
7. **count(first_iterator, last_iterator, x)** To count the occurrences of x in vector.
8. **find(first_iterator, last_iterator, x)** Returns an iterator to the first occurrence of x in vector, or points to `(name_of_vector).end()` if element is not present in vector.
9. **binary_search(first_iterator, last_iterator, x)** Returns true if x exists in sorted vector.
10. **lower_bound(first_iterator, last_iterator, x)** Returns an iterator pointing to the first element in the range $[first, last)$ which has a value not less than x.
11. **upper_bound(first_iterator, last_iterator, x)** Returns an iterator pointing to the first element in the range $[first, last)$ which has a value greater than x.
12. **arr.erase(position to be deleted)** This erases selected element in vector and shifts and resizes the vector elements accordingly.
13. **arr.erase(unique(arr.begin(),arr.end()),arr.end())** This erases the duplicate occurrences in sorted vector in a single line.