Data Structures & Algorithms

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **Cues**  NoteGem Horizontal Line    What is a data structure?    What is an interface?        What is implementation?        What are the characteristics of a data structure?                  What three problems can be solved with data structures?                                What are the execution time cases and how long is each one?                      What is data?  What is a data item?  What are group items?    What are elementary items?    What is an attribute?  What is an entity?  What is an entity set?  What is a field?  What is a record?  What is a file?        What is an algorithm?              What are the categories of algorithms?                    What are the characteristics of an algorithm?                          How do you write an algorithm?                                                                                                                                  What is a priori analysis?        What is a posterior analysis?      What is algorithm analysis?  How is running time defined?                  What are the two main factors of an algorithm?  Define each one and how it is measured.                                                                                        What is asymptotic analysis of an algorithm?    What is input bound?                        What are the three categories of time required by an algorithm?                Name the asymptotic notations.              What is big O notation?                                                What is omega notation?                                          What is theta notation?                              What is constant notation?  What is logarithmic notation?  What is linear notation?  What is n log n notation?  What is quadratic notation?  What is cubic notation?  What is polynomial notation?    What is exponential notation?          What is the greedy algorithm approach? What does it do?                                                                      What are some problems that utilize the greedy algorithm approach?                                What is the divide-and-conquer approach?                                                        What are the steps of the divide-and-conquer approach?                                    What are some algorithms based on the divide-and-conquer approach?                          What is dynamic programming?        Where is it used?                Name three features of dynamic programming and algorithms.                                Name some problems that can be solved using the dynamic programming approach.                                  What are some characteristics of a data definition?          What is a data object?    What's a data type?            What are the two data types?              What's a built-in data type?                        What's a derived data type?                              What operations can you perform on data in data structures?                What is an array?          What is an element?  What is an index in an array?                                                                  What are the basic operations you can perform on an array?                                                                                                                                                                What is a linked list?        What are the three terms to understand regarding linked lists?      How is a linked list visually represented?                                      What are the three types of linked lists?              What are the basic operations that can be performed on a linked list?                                                                                                                                                                                                                                                                                                                                            What is a doubly linked list?                                      How is a doubly linked list represented?                  What are the basic operations that can be performed on a doubly linked list?                            What is a circular linked list?        What is a singly circular linked list?                  What is a circular doubly linked list?                                      What are the basic operations supported by a circular linked list?                    What is a stack?                      How does a stack allow operations?          What is its data structure?            What are its operations?                In what means can it be implemented?  What sizes can it be?        What are its basic operations?                How do you check the status of the stack?                      What is a peek function?                What is an isfull function?                    What is an isempty function?                      What are the steps involved in a push operation?                                                What is an algorithm for a push operation?                    What is a pop operation?              What are its steps?                                                What is an algorithm for a pop operation?                What is a notation?  What is an arithmetic expression?          What are the three notations?          Why are they named so?        What is an infix notation?            What is a prefix notation?          What is a postfix notation?                                  How do you parse any arithmetic expression?              What is precedence?                          What is associativity?            what do the two determine?                                          what is the postfix evaluation algorithm?                what's a queue?  what methodology does it follow?                                How is a queue represented?                        How can it be implemented?          What are a queue's basic operations?            How do you check its status?                        What is a peek function?                What is an isfull function?                            What is an isempty function?                              What is an enqueue operation?        what are its steps?                                                      What is the enqueue operation's algorithm?                  what is a dequeue operation?        what are its steps?                                                  what is an algorithm for a dequeue operation?                    what is a linear search?                                      what is its algorithm?                      how do you write it in pseudocode?                what is a binary search?  what is its runtime complexity?                      how does a binary search work?                                                                                                                                                                    what's pseudocode for binary search?                                          what's an interpolation search?          what is its runtime complexity?                    how do you position in binary search?                                            how do you probe position in an interpolation search?                                                    what's the runtime complexity of an interpolation search algorithm?        what's the interpolation search algorithm?                    what is pseudocode for the interpolation search algorithm?                                                what's a hash table?        what operations are very fast in this data structure?  what does it use as a storage medium?    what is hashing?                                                                                            what is linear probing?                                                        what are the basic operations for a interpolation search algorithm?        what is a data item?      what is the hash method?        what are the interpolation search algorithm's operations?                                    what is sorting?          why is sorting important?  what is it used to represent?                    what is in-place sorting?  what is not-in-place sorting?                    what is stable sorting?  what is not-stable sorting?                                                                what is an adaptive sorting algorithm?        what is a non-adaptive sorting algorithm?              what is increasing order?            what is decreasing order?          what is non-increasing order?            what is non-decreasing order?              what is a bubble sort algorithm?      what is the runtime complexity of a bubble sort algorithm?          how does bubble sort work?                                                                                                                                                                              what is the algorithm for bubble  sort?                          what is the pseudocode for bubble sort?                                                      what is an insertion sort algorithm?            what is the runtime complexity of an insertion sort algorithm?            how does insertion sort work?                                                                                                                                                                                      what is the insertion sort algorithm?                      what is the pseudocode for insertion sort?                                  what is selection sort?            what is the runtime complexity of selection sort?          how does selection sort work?                                                                                                                                                                    what is the selection sort algorithm?                what is the pseudocode for selection sort?                                  what is a merge sort algorithm?                  how does merge sort work?                                                                                                                    what is the merge sort algorithm?                what is the pseudocode for a merge sort algorithm?                                                                          what is a shell sort?                      what is the runtime complexity of a shell sort?        how does shell sort work?                                                                                                                                                                                                            what is the shell sort algorithm?              what is the pseudocode for a shell sort?                                                    what is a quick sort?          what is the runtime complexity for a quick sort?      how do you partition in quick sort?                                what is the quick sort pivot algorithm?                            what is the pseudocode for quick sort pivot?                                            what is the quick sort algorithm?                  what is the quick sort pseudocode?                        what is a graph data structure?                                        how is a graph represented?        what is a vertex?            what is an edge?              what is adjacency?      what is a path?                                                    what are the basic operations of a graph?                    what is depth first traversal?                                                      what rules does depth first traversal employ?                  what are the steps for a depth first traversal?                                                                              what is a breadth first traversal?                                                          what rules does breadth first traversal employ?                  what are the steps for a breadth first traversal?                                                                        what is a tree?      what is a binary tree?  what is the special condition of a binary tree?  what are its benefits?                                          what is a path?  what is a root?    what is a parent?    what is a child?    what is a leaf?  what is a subtree?  what is visiting?    what is traversing?    what is a level?    what is a key?        how is a binary search tree represented?                                      what do all nodes share in a tree?              what are the basic operations of a binary search tree?                        what is an insert operation of a binary search tree?            what's the algorithm for an insert operation of a binary search tree?                          what is a search operation of a binary search tree?            what's the algorithm for a search operation of a binary search tree?                                what is tree traversal?          what are the ways you can traverse a tree?                what is in-order traversal?                                                                  what's the algorithm for an in-order traversal?          what's a pre-order traversal?                                                          what's the algorithm for a pre-order traversal?            what's a post-order traversal?                                                          what's the algorithm for a post-order traversal?            what's a binary search tree?                          how is a binary search tree represented?                                              what are the basic operations of a binary search tree?                                  what is a search operation of a binary search tree?            what's the algorithm for a search operation of a binary search tree?                                          what is an insert operation of a binary search tree?            what's the algorithm for an insert operation of a binary search tree?                                                                    what is an AVL tree?                                            what's the runtime complexity of an AVL tree?        what is the balance factor?                                                            what rotations can an AVL tree perform?                          what is a left rotation?                                        what is a right rotation?                                      what is a left-right rotation?                                                                                    what is a right-left rotation?                                                                                            what is a spanning tree?        what is a disconnected graph?                                                              what are the general properties of a spanning tree?                        what are the mathematical properties of a spanning tree?                        what are some applications of a spanning tree?                  what is a minimum spanning tree?  how can its weight be measured?          what are the two algorithms for a minimum spanning tree?    what type of algorithm are these?      what is a heap data structure?                  what is min-heap?                                      what is max-heap?                                          how do you construct a max-heap algorithm?                                                                            how do you delete from max heap?                                                what is recursion?  what are some examples of recursion              what are some properties of a recursive function?                  how is recursion implemented?          what is an activation record?                                                        why use recursion over iteration?            what's the time complexity of recursion?            what's the space complexity of recursion?                what is the Tower of Hanoi?                                  what are the rules of the Tower of Hanoi?                                          What is the minimum number of steps in which the Tower of Hanoi can be solved?        what's the algorithm for the Tower of Hanoi?                                                                                what's the pseudocode for the Tower of Hanoi?                          what is the Fibonacci series?                                                                        what is the Fibonacci iterative algorithm?                              what is the Fibonacci recursive algorithm? | **Notes**  NoteGem Horizontal Line    **Data Structure**: systematic way to organize data in order to use it efficiently.    **Interface**: Each data structure has an interface. Represents the set of operations that a data structure supports. Only provides the list of supported operations, type of parameters they can accept, and return type of these operations.    **Implementation**: internal representation of a data structure. Provides the definition of the algorithms used in the operations of the data structure.    Characteristics of a Data Structure     * **Correctness**: Data structure implementation should implement its interface correctly. * **Time Complexity**: Running time or the execution time of operations of data structure must be as small as possible. * **Space Complexity**: Memory usage of a data structure operation should be as little as possible.     Need for Data Structure  Three common problems that applications face now:     * **Data Search**: Consider an inventory of 1 million(106) items of a store. If the application is to search an item, it has to search an item in 1 million(106) items every time slowing down the search. As data grows, search will become slower. * **Processor speed**: Processor speed although being very high, falls limited if the data grows to billion records. * **Multiple requests**: As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.     To solve the above-mentioned problems, use data structures. Data can be organized in a data structure in such a way that all items may not be required to be searched and the required data can be searched almost instantly.    Execution Time Cases  There are three cases which are usually used to compare various data structure's execution time in a relative manner.     * **Worst Case**: This is the scenario where a particular data structure operation takes maximum time it can take. If an operation's worst case time is ƒ(n) then this operation will not take more than ƒ(n) time where ƒ(n) represents function of n. * **Average Case**: This is the scenario depicting the average execution time of an operation of a data structure. If an operation takes ƒ(n) time in execution, then m operations will take mƒ(n) time. * **Best Case**: This is the scenario depicting the least possible execution time of an operation of a data structure. If an operation takes ƒ(n) time in execution, then the actual operation may take time as the random number which would be maximum as ƒ(n).     Basic Terminology   * **Data** − Data are values or set of values. * **Data Item** − Data item refers to single unit of values. * **Group Items** − Data items that are divided into sub items are called as Group Items. * **Elementary Items** − Data items that cannot be divided are called as Elementary Items. * **Attribute and Entity** − An entity is that which contains certain attributes or properties, which may be assigned values. * **Entity Set** − Entities of similar attributes form an entity set. * **Field** − Field is a single elementary unit of information representing an attribute of an entity. * **Record** − Record is a collection of field values of a given entity. * **File** − File is a collection of records of the entities in a given entity set.     Data Structures - Algorithms Basics  **Algorithm**: a step-by-step procedure which defines a set of instructions to be executed in a certain order to get the desired output.   * Generally created independent of underlying languages. * Can be implemented in more than one programming language.     From the data structure point of view, following are some important categories of algorithms:     * **Search** − Algorithm to search an item in a data structure. * **Sort** − Algorithm to sort items in a certain order. * **Insert** − Algorithm to insert item in a data structure. * **Update** − Algorithm to update an existing item in a data structure. * **Delete**− Algorithm to delete an existing item from a data structure.     Characteristics of an Algorithm  Not all procedures can be called an algorithm. An algorithm should have the following characteristics:     * **Unambiguous**− Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning. * **Input**− An algorithm should have 0 or more well-defined inputs. * **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output. * **Finiteness** − Algorithms must terminate after a finite number of steps. * **Feasibility** − Should be feasible with the available resources. * **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.     How to Write an Algorithm?  There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.    As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.    We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.    Example  Let's try to learn algorithm-writing by using an example.    Problem − Design an algorithm to add two numbers and display the result.     1. START 2. declare three integers **a**, **b** & **c** 3. define values of **a** & **b** 4. add values of **a** & **b** 5. store output of step 4 to **c** 6. print **c** 7. STOP     Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as:     1. START ADD 2. get values of **a** & **b** 3. c ← a + b 4. display c 5. STOP   In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.    Writing step numbers, is optional.    We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.    one problem many solutions    Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.    Algorithm Analysis  Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following:     * ***A Priori* Analysis** − Theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation. * ***A Posterior* Analysis** − Empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.     We shall learn about *a priori* algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.    Algorithm Complexity  Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.     * **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm. * **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.     The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.    Space Complexity  Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components:     * A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc. * A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.     Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept:    Algorithm: SUM(A, B) Step 1 - START Step 2 - C ← A + B + 10 Step 3 - Stop  Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.    Time Complexity  Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.    For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.    Data Structures - Asymptotic Analysis  Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.    Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.    Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as *f*(n) and may be for another operation it is computed as *g*(n2). This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is significantly small.    Usually, the time required by an algorithm falls under three types:     * **Best Case** − Minimum time required for program execution. * **Average Case** − Average time required for program execution. * **Worst Case** − Maximum time required for program execution.     Asymptotic Notations  Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.     * Ο Notation * Ω Notation * θ Notation     Big Oh Notation, Ο  The notation Ο(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.    Big O Notation    For example, for a function *f*(n)  Ο(*f*(n)) = { *g*(n) : there exists c > 0 and n0 such that *f*(n) ≤ c.*g*(n) for all n > n0. }    Omega Notation, Ω  The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.    Omega Notation    For example, for a function *f*(n)  Ω(*f*(n)) ≥ { *g*(n) : there exists c > 0 and n0 such that *g*(n) ≤ c.*f*(n) for all n > n0. }    Theta Notation, θ  The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows:    Theta Notation    θ(*f*(n)) = { *g*(n) if and only if *g*(n) = Ο(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > n0. }    Common Asymptotic Notations  Following is a list of some common asymptotic notations:     |  |  |  | | --- | --- | --- | | constant | − | Ο(1) | | logarithmic | − | Ο(log n) | | linear | − | Ο(n) | | n log n | − | Ο(n log n) | | quadratic | − | Ο(n2) | | cubic | − | Ο(n3) | | polynomial | − | nΟ(1) | | exponential | − | 2Ο(n) |     Data Structures - Greedy Algorithms  An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.    Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.    Counting Coins  This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of ₹ 1, 2, 5 and 10 and we are asked to count ₹ 18 then the greedy procedure will be:     * 1 − Select one ₹ 10 coin, the remaining count is 8 * 2 − Then select one ₹ 5 coin, the remaining count is 3 * 3 − Then select one ₹ 2 coin, the remaining count is 1 * 4 − And finally, the selection of one ₹ 1 coins solves the problem     Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.    For the currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example, the greedy approach will use 10 + 1 + 1 + 1 + 1 + 1, total 6 coins. Whereas the same problem could be solved by using only 3 coins (7 + 7 + 1).    Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.  Examples  Most networking algorithms use the greedy approach. Here is a list of few of them:     * Travelling Salesman Problem * Prim's Minimal Spanning Tree Algorithm * Kruskal's Minimal Spanning Tree Algorithm * Dijkstra's Minimal Spanning Tree Algorithm * Graph - Map Coloring * Graph - Vertex Cover * Knapsack Problem * Job Scheduling Problem     There are lots of similar problems that uses the greedy approach to find an optimum solution.    Data Structures - Divide and Conquer  In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.    Divide and Conquer    Broadly, we can understand divide-and-conquer approach in a three-step process.    Divide/Break  This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.    Conquer/Solve  This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.    Merge/Combine  When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.    Examples  The following computer algorithms are based on divide-and-conquer programming approach:     * Merge Sort * Quick Sort * Binary Search * Strassen's Matrix Multiplication * Closest pair (points)     There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.    Data Structures - Dynamic Programming  Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.    Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.    So we can say that:     * The problem should be able to be divided into smaller overlapping sub-problem. * An optimum solution can be achieved by using an optimum solution of smaller sub-problems. * Dynamic algorithms use memoization.     Comparison  In contrast to greedy algorithms, where local optimization is addressed, dynamic algorithms are motivated for an overall optimization of the problem.  In contrast to divide and conquer algorithms, where solutions are combined to achieve an overall solution, dynamic algorithms use the output of a smaller sub-problem and then try to optimize a bigger sub-problem. Dynamic algorithms use memoization to remember the output of already solved sub-problems.    Example  The following computer problems can be solved using dynamic programming approach:     * Fibonacci number series * Knapsack problem * Tower of Hanoi * All pair shortest path by Floyd-Warshall * Shortest path by Dijkstra * Project scheduling     Dynamic programming can be used in both top-down and bottom-up manner. And of course, most of the times, referring to the previous solution output is cheaper than recomputing in terms of CPU cycles.    Data Structures & Algorithm Basic Concepts  This chapter explains the basic terms related to data structure.    **Data Definition** = defines a particular data with the following characteristics.     * **Atomic** − Definition should define a single concept. * **Traceable**− Definition should be able to be mapped to some data element. * **Accurate**− Definition should be unambiguous. * **Clear and Concise** − Definition should be understandable.     **Data Object** = Represents an object having a data.    **Data Type** = a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data.    There are two data types:     * Built-in Data Type * Derived Data Type     Built-in Data Type  Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.     * Integers * Boolean (true, false) * Floating (Decimal numbers) * Character and Strings     Derived Data Type  Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them.    For example:     * List * Array * Stack * Queue     Basic Operations  The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.     * Traversing * Searching * Insertion * Deletion * Sorting * Merging     Data Structures and Algorithms - Arrays  Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.     * **Element** − Each item stored in an array is called an element. * **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.     Array Representation  Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.    Array Declaration      Array Representation    As per the above illustration, following are the important points to be considered.     * Index starts with 0. * Array length is 10 which means it can store 10 elements. * Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.     Basic Operations  Following are the basic operations supported by an array.     * **Traverse** − print all the array elements one by one. * **Insertion** − Adds an element at the given index. * **Deletion** − Deletes an element at the given index. * **Search** − Searches an element using the given index or by the value. * **Update**− Updates an element at the given index.     In C, when an array is initialized with size, then it assigns defaults values to its elements in following order.     |  |  | | --- | --- | | **Data Type** | **Default Value** | | bool | false | | char | 0 | | int | 0 | | float | 0.0 | | double | 0.0f | | void |  | | wchar\_t | 0 |     Traverse Operation  This operation is to traverse through the elements of an array.    Insertion Operation  Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array. For variations of array insertion operation [click here](https://www.tutorialspoint.com/data_structures_algorithms/array_insertion_algorithm.htm)    Deletion Operation  Deletion refers to removing an existing element from the array and re-organizing all elements of an array.    Algorithm  Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to delete an element available at the Kth position of LA.    1. Start 2. Set J = K 3. Repeat steps 4 and 5 while J < N 4. Set LA[J] = LA[J + 1] 5. Set J = J+1 6. Set N = N-1 7. Stop  Search Operation  You can perform a search for an array element based on its value or its index.    Algorithm  Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to find an element with a value of ITEM using sequential search.  1. Start 2. Set J = 0 3. Repeat steps 4 and 5 while J < N 4. IF LA[J] is equal ITEM THEN GOTO STEP 6 5. Set J = J +1 6. PRINT J, ITEM 7. Stop    Update Operation  Update operation refers to updating an existing element from the array at a given index.    Algorithm  Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to update an element available at the Kth position of LA.    1. Start 2. Set LA[K-1] = ITEM 3. Stop    Data Structure and Algorithms - Linked List  A linked list is a sequence of data structures, which are connected together via links.    **Linked List** is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.     * **Link** − Each link of a linked list can store a data called an element. * **Next** − Each link of a linked list contains a link to the next link called Next. * **LinkedList** − A Linked List contains the connection link to the first link called First.     Linked List Representation  Linked list can be visualized as a chain of nodes, where every node points to the next node.    Linked List    As per the above illustration, following are the important points to be considered.     * Linked List contains a link element called first. * Each link carries a data field(s) and a link field called next. * Each link is linked with its next link using its next link. * Last link carries a link as null to mark the end of the list.     Types of Linked List  Following are the various types of linked list.     * Simple Linked List − Item navigation is forward only. * Doubly Linked List − Items can be navigated forward and backward. * Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.     Basic Operations  Following are the basic operations supported by a list.     * Insertion − Adds an element at the beginning of the list. * Deletion − Deletes an element at the beginning of the list. * Display − Displays the complete list. * Search − Searches an element using the given key. * Delete − Deletes an element using the given key.     Insertion Operation  Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.    Linked List Insertion    Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode). Then point B.next to C −  NewNode.next −> RightNode;  It should look like this:    Linked List Insertion    Now, the next node at the left should point to the new node.    LeftNode.next −> NewNode;  Linked List Insertion    This will put the new node in the middle of the two. The new list should look like this:    Linked List Insertion    Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.    Deletion Operation  Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.    Linked List Deletion    The left (previous) node of the target node now should point to the next node of the target node:    LeftNode.next −> TargetNode.next;  Linked List Deletion    This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.    TargetNode.next −> NULL;  Linked List Deletion    We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.    Linked List Deletion    Reverse Operation  This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.    Linked List Reverse Operation    First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node:    Linked List Reverse Operation    We have to make sure that the last node is not the last node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.    Linked List Reverse Operation    Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.    Linked List Reverse Operation    We'll make the head node point to the new first node by using the temp node.    Linked List Reverse Operation    The linked list is now reversed.    Data Structure - Doubly Linked List  Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.     * **Link**− Each link of a linked list can store a data called an element. * **Next** − Each link of a linked list contains a link to the next link called Next. * **Prev**− Each link of a linked list contains a link to the previous link called Prev. * **LinkedList** − A Linked List contains the connection link to the first link called First and to the last link called Last.     Doubly Linked List Representation    Doubly Linked List    As per the above illustration, following are the important points to be considered.     * Doubly Linked List contains a link element called first and last. * Each link carries a data field(s) and two link fields called next and prev. * Each link is linked with its next link using its next link. * Each link is linked with its previous link using its previous link. * The last link carries a link as null to mark the end of the list.     Basic Operations  Following are the basic operations supported by a list.     * **Insertion** − Adds an element at the beginning of the list. * **Deletion** − Deletes an element at the beginning of the list. * **Insert Last** − Adds an element at the end of the list. * **Delete Last** − Deletes an element from the end of the list. * **Insert After** − Adds an element after an item of the list. * **Delete**− Deletes an element from the list using the key. * **Display forward** − Displays the complete list in a forward manner. * **Display backward** − Displays the complete list in a backward manner.     Insertion Operation  Deletion Operation  Insertion at the End of an Operation  Data Structure - Circular Linked List  Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.    Singly Linked List as Circular  In singly linked list, the next pointer of the last node points to the first node.    Singly Linked List as Circular Linked List    Doubly Linked List as Circular  In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.    Doubly Linked List as Circular Linked List    As per the above illustration, following are the important points to be considered.     * The last link's next points to the first link of the list in both cases of singly as well as doubly linked list. * The first link's previous points to the last of the list in case of doubly linked list.     Basic Operations  Following are the important operations supported by a circular list.     * insert − Inserts an element at the start of the list. * delete − Deletes an element from the start of the list. * display − Displays the list.     Insertion Operation  Deletion Operation  Display List Operation  Data Structure and Algorithms - Stack  A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.    Stack Example    A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.    This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.    Stack Representation  The following diagram depicts a stack and its operations:  Stack Representation    A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.    Basic Operations  Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −   * **push()** − Pushing (storing) an element on the stack. * **pop()** − Removing (accessing) an element from the stack.     When data is PUSHed onto stack.    To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks:     * **peek()** − get the top data element of the stack, without removing it. * **isFull()** − check if stack is full. * **isEmpty()** − check if stack is empty.     At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named top. The top pointer provides top value of the stack without actually removing it.    First we should learn about procedures to support stack functions:  peek()  Algorithm of peek() function −  begin procedure peek  return stack[top] end procedure    isfull()  Algorithm of isfull() function −  begin procedure isfull  if top equals to MAXSIZE  return true  else  return false  endif end procedure  isempty()  Algorithm of isempty() function −  begin procedure isempty  if top less than 1  return true  else  return false  endif  end procedure    Push Operation  The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps:     1. Checks if the stack is full. 2. If the stack is full, produces an error and exit. 3. If the stack is not full, increments top to point next empty space. 4. Adds data element to the stack location, where top is pointing. 5. Returns success.     Stack Push Operation    If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.    Algorithm for PUSH Operation  A simple algorithm for Push operation can be derived as follows −  begin procedure push: stack, data  if stack is full  return null  endif    top ← top + 1  stack[top] ← data  end procedure  Pop Operation  Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.    A Pop operation may involve the following steps:     1. Checks if the stack is empty. 2. If the stack is empty, produces an error and exit. 3. If the stack is not empty, accesses the data element at which top is pointing. 4. Decreases the value of top by 1. 5. Returns success.     Stack Pop Operation  Algorithm for Pop Operation  A simple algorithm for Pop operation can be derived as follows −  begin procedure pop: stack  if stack is empty  return null  endif  data ← stack[top]  top ← top - 1  return data  end procedure    Data Structure - Expression Parsing  The way to write arithmetic expression is known as a notation. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression.    These notations are:     * Infix Notation * Prefix (Polish) Notation * Postfix (Reverse-Polish) Notation     These notations are named as how they use operator in expression. We shall learn the same here in this chapter.    Infix Notation  We write expression in infix notation, e.g. a - b + c, where operators are used in-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.    Prefix Notation  In this notation, operator is prefixed to operands, i.e. operator is written ahead of operands. For example, +ab. This is equivalent to its infix notation a + b. Prefix notation is also known as Polish Notation.    Postfix Notation  This notation style is known as Reversed Polish Notation. In this notation style, the operator is postfixed to the operands i.e., the operator is written after the operands. For example, ab+. This is equivalent to its infix notation a + b.    The following table briefly tries to show the difference in all three notations:     |  |  |  |  | | --- | --- | --- | --- | | **Sr.No.** | **Infix Notation** | **Prefix Notation** | **Postfix Notation** | | 1 | a + b | + a b | a b + | | 2 | (a + b) ∗ c | ∗ + a b c | a b + c ∗ | | 3 | a ∗ (b + c) | ∗ a + b c | a b c + ∗ | | 4 | a / b + c / d | + / a b / c d | a b / c d / + | | 5 | (a + b) ∗ (c + d) | ∗ + a b + c d | a b + c d + ∗ | | 6 | ((a + b) ∗ c) - d | - ∗ + a b c d | a b + c ∗ d - |     Parsing Expressions  As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.    To parse any arithmetic expression, we need to take care of operator precedence and associativity also.    Precedence  When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others.    For example:    Operator Precendence    As multiplication operation has precedence over addition, b \* c will be evaluated first. A table of operator precedence is provided later.    Associativity  Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as (a + b) − c.    Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest):     |  |  |  |  | | --- | --- | --- | --- | | **Sr.No.** | **Operator** | **Precedence** | **Associativity** | | 1 | Exponentiation ^ | Highest | Right Associative | | 2 | Multiplication ( ∗ ) & Division ( / ) | Second Highest | Left Associative | | 3 | Addition ( + ) & Subtraction ( − ) | Lowest | Left Associative |     The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis.    For example:    In a + b\*c, the expression part b\*c will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for a + b to be evaluated first, like (a + b)\*c.    Postfix Evaluation Algorithm  We shall now look at the algorithm on how to evaluate postfix notation −   1. scan the expression from left to right 2. if it is an operand push it to stack 3. if it is an operator pull operand from stack and perform operation 4. store the output of step 3, back to stack 5. scan the expression until all operands are consumed 6. pop the stack and perform operation     Data Structure and Algorithms - Queue  Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.    Queue Example    A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.    Queue Representation  As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure:    Queue Example    As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.    Basic Operations  Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues:     * enqueue() − add (store) an item to the queue. * dequeue() − remove (access) an item from the queue.     Few more functions are required to make the above-mentioned queue operation efficient. These are:     * peek() − Gets the element at the front of the queue without removing it. * isfull() − Checks if the queue is full. * isempty() − Checks if the queue is empty.     In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear pointer.  Let's first learn about supportive functions of a queue:    peek()  This function helps to see the data at the front of the queue. The algorithm of peek() function is as follows:    begin procedure peek  return queue[front] end procedure    isfull()  As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function:    begin procedure isfull  if rear equals to MAXSIZE  return true  else  return false  endif  end procedure    isempty()  Algorithm of isempty() function:    begin procedure isempty  if front is less than MIN OR front is greater than rear  return true  else  return false  endif  end procedure    If the value of front is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.    Enqueue Operation  Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.    The following steps should be taken to enqueue (insert) data into a queue:     1. Check if the queue is full. 2. If the queue is full, produce overflow error and exit. 3. If the queue is not full, increment rear pointer to point the next empty space. 4. Add data element to the queue location, where the rear is pointing. 5. return success.     Insert Operation  Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.    Algorithm for enqueue operation  procedure enqueue(data)  if queue is full  return overflow  endif   rear ← rear + 1  queue[rear] ← data  return true  end procedure    Dequeue Operation  Accessing data from the queue is a process of two tasks − access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation:     1. Check if the queue is empty. 2. If the queue is empty, produce underflow error and exit. 3. If the queue is not empty, access the data where front is pointing. 4. Increment front pointer to point to the next available data element. 5. Return success.     Remove Operation  Algorithm for dequeue operation    procedure dequeue   if queue is empty  return underflow  end if  data = queue[front]  front ← front + 1  return true  end procedure      Data Structure and Algorithms Linear Search  Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.    Linear Search Animation    Algorithm  Linear Search ( Array A, Value x)   1. Set i to 1 2. if i > n then go to step 7 3. if A[i] = x then go to step 6 4. Set i to i + 1 5. Go to Step 2 6. Print Element x Found at index i and go to step 8 7. Print element not found 8. Exit     Pseudocode  procedure linear\_search (list, value)  for each item in the list  if match item == value  return the item's location  end if  end for  end procedure    Data Structure and Algorithms Binary Search  Binary search is a fast search algorithm with run-time complexity of Ο(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.    Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.    How Binary Search Works?  For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.    Binary search    First, we shall determine half of the array by using this formula:    mid = low + (high - low) / 2    Here it is, 0 + (9 - 0 ) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.    Binary search    Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.    Binary search    We change our low to mid + 1 and find the new mid value again.    low = mid + 1 mid = low + (high - low) / 2    Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.    Binary search    The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.    Binary search    Hence, we calculate the mid again. This time it is 5.    Binary search    We compare the value stored at location 5 with our target value. We find that it is a match.    Binary search    We conclude that the target value 31 is stored at location 5.    Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.    Pseudocode  The pseudocode of binary search algorithms should look like this:    Procedure binary\_search  A ← sorted array  n ← size of array  x ← value to be searched  Set lowerBound = 1  Set upperBound = n  while x not found  if upperBound < lowerBound   EXIT: x does not exists.  set midPoint = lowerBound + ( upperBound - lowerBound ) / 2   if A[midPoint] < x  set lowerBound = midPoint + 1   if A[midPoint] > x  set upperBound = midPoint - 1  if A[midPoint] = x   EXIT: x found at location midPoint  end while  end procedure    Data Structure - Interpolation Search  Interpolation search is an improved variant of binary search. This search algorithm works on the probing position of the required value. For this algorithm to work properly, the data collection should be in a sorted form and equally distributed.    Binary search has a huge advantage of time complexity over linear search. Linear search has worst-case complexity of Ο(n) whereas binary search has Ο(log n).    There are cases where the location of target data may be known in advance. For example, in case of a telephone directory, if we want to search the telephone number of Morphius. Here, linear search and even binary search will seem slow as we can directly jump to memory space where the names start from 'M' are stored.    Positioning in Binary Search  In binary search, if the desired data is not found then the rest of the list is divided in two parts, lower and higher. The search is carried out in either of them.    BST Step One    BST Step Two    BST Step Three    BST Step Four    Even when the data is sorted, binary search does not take advantage to probe the position of the desired data.    Position Probing in Interpolation Search  Interpolation search finds a particular item by computing the probe position. Initially, the probe position is the position of the middle most item of the collection.    Interpolation Step One    Interpolation Step Two    If a match occurs, then the index of the item is returned. To split the list into two parts, we use the following method:    mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) \* (X - A[Lo])    where −  A = list  Lo = Lowest index of the list  Hi = Highest index of the list  A[n] = Value stored at index n in the list  If the middle item is greater than the item, then the probe position is again calculated in the sub-array to the right of the middle item. Otherwise, the item is searched in the subarray to the left of the middle item. This process continues on the sub-array as well until the size of subarray reduces to zero.    Runtime complexity of interpolation search algorithm is Ο(log (log n)) as compared to Ο(log n) of BST in favorable situations.    Algorithm  As it is an improvisation of the existing BST algorithm, we are mentioning the steps to search the 'target' data value index, using position probing:     1. Start searching **data** from middle of the list. 2. If it is a match, return the index of the item, and exit. 3. If it is not a match, probe position. 4. Divide the list using probing formula and find the new middle. 5. If data is greater than middle, search in higher sub-list. 6. If data is smaller than middle, search in lower sub-list. 7. Repeat until match.     Pseudocode  A → Array list N → Size of A X → Target Value  Procedure Interpolation\_Search()  Set Lo → 0  Set Mid → -1  Set Hi → N-1  While X does not match   if Lo equals to Hi OR A[Lo] equals to A[Hi]  EXIT: Failure, Target not found  end if   Set Mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) \* (X - A[Lo])  if A[Mid] = X  EXIT: Success, Target found at Mid  else   if A[Mid] < X  Set Lo to Mid+1  else if A[Mid] > X  Set Hi to Mid-1  end if  end if  End While  End Procedure    Data Structure and Algorithms - Hash Table  Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.    Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.    Hashing  Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.    Hash Function     * (1,20) * (2,70) * (42,80) * (4,25) * (12,44) * (14,32) * (17,11) * (13,78) * (37,98)      |  |  |  |  | | --- | --- | --- | --- | | **Sr.No.** | **Key** | **Hash** | **Array Index** | | 1 | 1 | 1 % 20 = 1 | 1 | | 2 | 2 | 2 % 20 = 2 | 2 | | 3 | 42 | 42 % 20 = 2 | 2 | | 4 | 4 | 4 % 20 = 4 | 4 | | 5 | 12 | 12 % 20 = 12 | 12 | | 6 | 14 | 14 % 20 = 14 | 14 | | 7 | 17 | 17 % 20 = 17 | 17 | | 8 | 13 | 13 % 20 = 13 | 13 | | 9 | 37 | 37 % 20 = 17 | 17 |     Linear Probing  As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.     |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Sr.No.** | **Key** | **Hash** | **Array Index** | **After Linear Probing, Array Index** | | 1 | 1 | 1 % 20 = 1 | 1 | 1 | | 2 | 2 | 2 % 20 = 2 | 2 | 2 | | 3 | 42 | 42 % 20 = 2 | 2 | 3 | | 4 | 4 | 4 % 20 = 4 | 4 | 4 | | 5 | 12 | 12 % 20 = 12 | 12 | 12 | | 6 | 14 | 14 % 20 = 14 | 14 | 14 | | 7 | 17 | 17 % 20 = 17 | 17 | 17 | | 8 | 13 | 13 % 20 = 13 | 13 | 13 | | 9 | 37 | 37 % 20 = 17 | 17 | 18 |     Basic Operations  Following are the basic primary operations of a hash table.     * Search − Searches an element in a hash table. * Insert − inserts an element in a hash table. * delete − Deletes an element from a hash table.     DataItem  Define a data item having some data and key, based on which the search is to be conducted in a hash table.    Hash Method  Define a hashing method to compute the hash code of the key of the data item.    Search Operation  Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.    Insert Operation  Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.    Delete Operation  Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.    Data Structure - Sorting Techniques  Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.    The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios:     * Telephone Directory − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily. * Dictionary − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.     In-place Sorting and Not-in-place Sorting  Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called in-place sorting. Bubble sort is an example of in-place sorting.    However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called not-in-place sorting. Merge-sort is an example of not-in-place sorting.    Stable and Not Stable Sorting  If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called stable sorting.    Stable Sorting    If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called unstable sorting.    Unstable Sorting    Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.    Adaptive and Non-Adaptive Sorting Algorithm  A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.    A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness.    Important Terms  Some terms are generally coined while discussing sorting techniques, here is a brief introduction to them:    Increasing Order  A sequence of values is said to be in increasing order, if the successive element is greater than the previous one. For example, 1, 3, 4, 6, 8, 9 are in increasing order, as every next element is greater than the previous element.    Decreasing Order  A sequence of values is said to be in decreasing order, if the successive element is less than the current one. For example, 9, 8, 6, 4, 3, 1 are in decreasing order, as every next element is less than the previous element.    Non-Increasing Order  A sequence of values is said to be in non-increasing order, if the successive element is less than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 9, 8, 6, 3, 3, 1 are in non-increasing order, as every next element is less than or equal to (in case of 3) but not greater than any previous element.    Non-Decreasing Order  A sequence of values is said to be in non-decreasing order, if the successive element is greater than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 1, 3, 3, 6, 8, 9 are in non-decreasing order, as every next element is greater than or equal to (in case of 3) but not less than the previous one.    Data Structure - Bubble Sort Algorithm  Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2) where n is the number of items.    How Bubble Sort Works?  We take an unsorted array for our example. Bubble sort takes Ο(n2) time so we're keeping it short and precise.    Bubble Sort    Bubble sort starts with very first two elements, comparing them to check which one is greater.    Bubble Sort    In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.    Bubble Sort    We find that 27 is smaller than 33 and these two values must be swapped.    Bubble Sort    The new array should look like this:    Bubble Sort    Next we compare 33 and 35. We find that both are in already sorted positions.    Bubble Sort    Then we move to the next two values, 35 and 10.    Bubble Sort    We know then that 10 is smaller 35. Hence they are not sorted.    Bubble Sort    We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this:    Bubble Sort    To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this:    Bubble Sort    Notice that after each iteration, at least one value moves at the end.    Bubble Sort    And when there's no swap required, bubble sorts learns that an array is completely sorted.    Bubble Sort    Now we should look into some practical aspects of bubble sort.    Algorithm  We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.    begin BubbleSort(list)  for all elements of list  if list[i] > list[i+1]  swap(list[i], list[i+1])  end if  end for   return list  end BubbleSort    Pseudocode  We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.    To ease-out the issue, we use one flag variable swapped which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.    Pseudocode of BubbleSort algorithm can be written as follows:  procedure bubbleSort( list : array of items )  loop = list.count;   for i = 0 to loop-1 do:  swapped = false                  for j = 0 to loop-1 do:   /\* compare the adjacent elements \*/   if list[j] > list[j+1] then  /\* swap them \*/  swap( list[j], list[j+1] )                   swapped = true  end if   end for   /\*if no number was swapped that means   array is sorted now, break the loop.\*/   if(not swapped) then  break  end if   end for  end procedure return list    Data Structure and Algorithms Insertion Sort  This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.    The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2), where n is the number of items.    How Insertion Sort Works?  We take an unsorted array for our example.    Unsorted Array    Insertion sort compares the first two elements.    Insertion Sort    It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.    Insertion Sort    Insertion sort moves ahead and compares 33 with 27.    Insertion Sort    And finds that 33 is not in the correct position.    Insertion Sort    It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.    Insertion Sort    By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.    Insertion Sort    These values are not in a sorted order.    Insertion Sort    So we swap them.    Insertion Sort    However, swapping makes 27 and 10 unsorted.    Insertion Sort    Hence, we swap them too.    Insertion Sort    Again we find 14 and 10 in an unsorted order.    Insertion Sort    We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.    Insertion Sort    This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.    Algorithm  Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.     1. If it is the first element, it is already sorted. return 1; 2. Pick next element 3. Compare with all elements in the sorted sub-list 4. Shift all the elements in the sorted sub-list that is greater than the value to be sorted 5. Insert the value 6. Repeat until list is sorted   Pseudocode  procedure insertionSort( A : array of items )  int holePosition  int valueToInsert  for i = 1 to length(A) inclusive do:  /\* select value to be inserted \*/  valueToInsert = A[i]  holePosition = i   /\*locate hole position for the element to be inserted \*/  while holePosition > 0 and A[holePosition-1] > valueToInsert do:  A[holePosition] = A[holePosition-1]  holePosition = holePosition -1  end while                  /\* insert the number at hole position \*/  A[holePosition] = valueToInsert   end for         end procedure    Data Structure and Algorithms Selection Sort  Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.    The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.  This algorithm is not suitable for large data sets as its average and worst case complexities are of Ο(n2), where n is the number of items.    How Selection Sort Works?  Consider the following depicted array as an example.    Unsorted Array    For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.    Selection Sort    So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.    Selection Sort    For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.    Selection Sort    find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.    Selection Sort    After two iterations, two least values are positioned at the beginning in a sorted manner.    Selection Sort    The same process is applied to the rest of the items in the array.  Following is a pictorial depiction of the entire sorting process:    Selection Sort    Now, let us learn some programming aspects of selection sort.    Algorithm   1. Set MIN to location 0 2. Search the minimum element in the list 3. Swap with value at location MIN 4. Increment MIN to point to next element 5. Repeat until list is sorted   Pseudocode  procedure selection sort   list : array of items  n : size of list  for i = 1 to n - 1  /\* set current element as minimum\*/  min = i   /\* check the element to be minimum \*/  for j = i+1 to n   if list[j] < list[min] then  min = j;  end if  end for  /\* swap the minimum element with the current element\*/  if indexMin != i then  swap list[min] and list[i]  end if  end for         end procedure    Data Structures - Merge Sort Algorithm  Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being Ο(n log n), it is one of the most respected algorithms.    Merge sort first divides the array into equal halves and then combines them in a sorted manner.    How Merge Sort Works?  To understand merge sort, we take an unsorted array as the following:    Unsorted Array    We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.    Merge Sort Division    This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.    Merge Sort Division    We further divide these arrays and we achieve atomic value which can no more be divided.    Merge Sort Division  Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.    We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.    Merge Sort Combine    In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.    Merge Sort Combine    After the final merging, the list should look like this:    Merge Sort    Now we should learn some programming aspects of merge sorting.    Algorithm  Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.     1. if it is only one element in the list it is already sorted, return. 2. divide the list recursively into two halves until it can no more be divided. 3. merge the smaller lists into new list in sorted order.   Pseudocode  We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions − divide & merge.    Merge sort works with recursion and we shall see our implementation in the same way.    procedure mergesort( var a as array )  if ( n == 1 ) return a  var l1 as array = a[0] ... a[n/2]  var l2 as array = a[n/2+1] ... a[n]  l1 = mergesort( l1 )  l2 = mergesort( l2 )  return merge( l1, l2 ) end procedure  procedure merge( var a as array, var b as array )  var c as array  while ( a and b have elements )  if ( a[0] > b[0] )  add b[0] to the end of c  remove b[0] from b  else  add a[0] to the end of c  remove a[0] from a  end if  end while  while ( a has elements )  add a[0] to the end of c  remove a[0] from a  end while   while ( b has elements )  add b[0] to the end of c  remove b[0] from b  end while   return c         end procedure    Data Structure and Algorithms - Shell Sort  Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.    This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval. This interval is calculated based on Knuth's formula as:    Knuth's Formula  h = h \* 3 + 1 where −  h is interval with initial value 1  This algorithm is quite efficient for medium-sized data sets as its average and worst case complexity are of Ο(n), where n is the number of items.    How Shell Sort Works?  Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}.    Shell Sort    We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this:    Shell Sort    Then, we take interval of 2 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}.    Shell Sort    We compare and swap the values, if required, in the original array. After this step, the array should look like this:    Shell Sort    Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.    Following is the step-by-step depiction:    Shell Sort    We see that it required only four swaps to sort the rest of the array.    Algorithm  Following is the algorithm for shell sort.     1. Initialize the value of *h* 2. Divide the list into smaller sub-list of equal interval *h* 3. Sort these sub-lists using **insertion sort** 4. Repeat until complete list is sorted   Pseudocode  Following is the pseudocode for shell sort.    procedure shellSort()  A : array of items           /\* calculate interval\*/  while interval < A.length /3 do:  interval = interval \* 3 + 1           end while   while interval > 0 do:  for outer = interval; outer < A.length; outer ++ do:  /\* select value to be inserted \*/  valueToInsert = A[outer]  inner = outer;  /\*shift element towards right\*/  while inner > interval -1 && A[inner - interval] >= valueToInsert do:  A[inner] = A[inner - interval]  inner = inner - interval  end while  /\* insert the number at hole position \*/  A[inner] = valueToInsert  end for  /\* calculate interval\*/  interval = (interval -1) /3;  end while  end procedure    Data Structure and Algorithms - Quick Sort  Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.  Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(nLogn) and image.png(n2), respectively.  Partition in Quick Sort  Following animated representation explains how to find the pivot value in an array.  Quick Sort Partition Animation  The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.  Quick Sort Pivot Algorithm  Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.     1. Choose the highest index value has pivot 2. Take two variables to point left and right of the list excluding pivot 3. left points to the low index 4. right points to the high 5. while value at left is less than pivot move right 6. while value at right is greater than pivot move left 7. if both step 5 and step 6 does not match swap left and right 8. if left ≥ right, the point where they met is new pivot   Quick Sort Pivot Pseudocode  The pseudocode for the above algorithm can be derived as:    function partitionFunc(left, right, pivot)  leftPointer = left  rightPointer = right - 1  while True do  while A[++leftPointer] < pivot do  //do-nothing   end while                  while rightPointer > 0 && A[--rightPointer] > pivot do  //do-nothing   end while                  if leftPointer >= rightPointer  break  else   swap leftPointer,rightPointer  end if                  end while           swap leftPointer,right  return leftPointer         end function    Quick Sort Algorithm  Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows:     1. Make the right-most index value pivot 2. partition the array using pivot value 3. quicksort left partition recursively 4. quicksort right partition recursively   Quick Sort Pseudocode  To get more into it, let see the pseudocode for quick sort algorithm:    procedure quickSort(left, right)  if right-left <= 0  return  else   pivot = A[right]  partition = partitionFunc(left, right, pivot)  quickSort(left,partition-1)  quickSort(partition+1,right)   end if  end procedure    Data Structure - Graph Data Structure  A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.    Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph:    Graph Basics    In the above graph,  V = {a, b, c, d, e}  E = {ab, ac, bd, cd, de}    Graph Data Structure  Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms:     * Vertex − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.      * Edge − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.      * Adjacency − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.      * Path − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.     graph    Basic Operations  Following are basic primary operations of a Graph:     * Add Vertex − Adds a vertex to the graph. * Add Edge − Adds an edge between the two vertices of the graph. * Display Vertex − Displays a vertex of the graph.     To know more about Graph, please read [Graph Theory Tutorial](https://www.tutorialspoint.com/graph_theory/index.htm). We shall learn about traversing a graph in the coming chapters.    Data Structure - Depth First Traversal  Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.    Depth First Travesal    As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.     * Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack. * Rule 2 − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.) * Rule 3 − Repeat Rule 1 and Rule 2 until the stack is empty.      |  |  |  | | --- | --- | --- | | **Step** | **Traversal** | **Description** | | 1 | Depth First Search Step One | Initialize the stack. | | 2 | Depth First Search Step Two | Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order. | | 3 | Depth First Search Step Three | Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only. | | 4 | Depth First Search Step Four | Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order. | | 5 | Depth First Search Step Five | We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack. | | 6 | Depth First Search Step Six | We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack. | | 7 | Depth First Search Step Seven | Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack. |     As C does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.    Data Structure - Breadth First Traversal  Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.    Breadth First Traversal    As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.     * Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue. * Rule 2 − If no adjacent vertex is found, remove the first vertex from the queue. * Rule 3 − Repeat Rule 1 and Rule 2 until the queue is empty.      |  |  |  | | --- | --- | --- | | **Step** | **Traversal** | **Description** | | 1 | Breadth First Search Step One | Initialize the queue. | | 2 | Breadth First Search Step Two | We start from visiting **S** (starting node), and mark it as visited. | | 3 | Breadth First Search Step Three | We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it. | | 4 | Breadth First Search Step Four | Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it. | | 5 | Breadth First Search Step Five | Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it. | | 6 | Breadth First Search Step Six | Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**. | | 7 | Breadth First Search Step Seven | From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it. |     At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.    Data Structure and Algorithms - Tree  Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.    Binary Tree is a special datastructure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.    Binary Tree    Important Terms  Following are the important terms with respect to tree.     * **Path**: Path refers to the sequence of nodes along the edges of a tree. * **Root**: The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node. * **Parent**: Any node except the root node has one edge upward to a node called parent. * **Child**: The node below a given node connected by its edge downward is called its child node. * **Leaf**: The node which does not have any child node is called the leaf node. * **Subtree**: Subtree represents the descendants of a node. * **Visiting**: Visiting refers to checking the value of a node when control is on the node. * **Traversing**: Traversing means passing through nodes in a specific order. * **Levels**: Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on. * **keys**: Key represents a value of a node based on which a search operation is to be carried out for a node.     Binary Search Tree Representation  Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.    Binary Search Tree    We're going to implement tree using node object and connecting them through references.    Tree Node  The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.    In a tree, all nodes share common construct.    BST Basic Operations  The basic operations that can be performed on a binary search tree data structure, are the following:     * **Insert** − Inserts an element in a tree/create a tree. * **Search** − Searches an element in a tree. * **Preorder Traversal** − Traverses a tree in a pre-order manner. * **Inorder Traversal** − Traverses a tree in an in-order manner. * **Postorder Traversal** − Traverses a tree in a post-order manner.     We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.    Insert Operation  The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.    Algorithm  If root is NULL   then create root node return  If root exists then  compare the data with node.data   while until insertion position is located  If data is greater than node.data  goto right subtree  else  goto left subtree  endwhile   insert data         end If    Search Operation  Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.    Algorithm  If root.data is equal to search.data  return root else  while data not found  If data is greater than node.data  goto right subtree  else  goto left subtree    If data found  return node  endwhile     return data not found   end if    Data Structure & Algorithms - Tree Traversal  Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree:     * In-order Traversal * Pre-order Traversal * Post-order Traversal     Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.    In-order Traversal  In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.    If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.    In Order Traversal    We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be:    *D → B → E → A → F → C → G*    Algorithm  Until all nodes are traversed:     1. Recursively traverse left subtree. 2. Visit root node. 3. Recursively traverse right subtree.   Pre-order Traversal  In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.    Pre Order Traversal    We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be:    *A → B → D → E → C → F → G*    Algorithm  Until all nodes are traversed:     1. Visit root node. 2. Recursively traverse left subtree. 3. Recursively traverse right subtree.   Post-order Traversal  In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.    Post Order Traversal    We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be:    *D → E → B → F → G → C → A*    Algorithm  Until all nodes are traversed:     1. Recursively traverse left subtree. 2. Recursively traverse right subtree. 3. Visit root node.     Data Structure - Binary Search Tree  A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties:     * The left sub-tree of a node has a key less than or equal to its parent node's key. * The right sub-tree of a node has a key greater than to its parent node's key.     Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as:    left\_subtree (keys) ≤ node (key) ≤ right\_subtree (keys)  Representation  BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.    Following is a pictorial representation of BST:    Binary Search Tree    We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.    Basic Operations  Following are the basic operations of a tree:     * **Search** − Searches an element in a tree. * **Insert** − Inserts an element in a tree. * **Pre-order Traversa**l − Traverses a tree in a pre-order manner. * **In-order Traversal** − Traverses a tree in an in-order manner. * **Post-order Traversal** − Traverses a tree in a post-order manner.     Node  Define a node having some data, references to its left and right child nodes.    struct node {  int data;   struct node \*leftChild;  struct node \*rightChild; };    Search Operation  Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.    Algorithm  struct node\* search(int data){  struct node \*current = root;  printf("Visiting elements: ");          while(current->data != data){          if(current != NULL) {  printf("%d ",current->data);                          //go to left tree  if(current->data > data){  current = current->leftChild;  } //else go to right tree  else {   current = current->rightChild;  }                          //not found  if(current == NULL){  return NULL;  }  }                          }   return current; }    Insert Operation  Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.    Algorithm  void insert(int data) {  struct node \*tempNode = (struct node\*) malloc(sizeof(struct node));  struct node \*current;  struct node \*parent;  tempNode->data = data;  tempNode->leftChild = NULL;  tempNode->rightChild = NULL;  //if tree is empty  if(root == NULL) {  root = tempNode;  } else {  current = root;  parent = NULL;  while(1) {   parent = current;                          //go to left of the tree  if(data < parent->data) {  current = current->leftChild;   //insert to the left                                  if(current == NULL) {  parent->leftChild = tempNode;  return;  }  } //go to right of the tree  else {  current = current->rightChild;   //insert to the right  if(current == NULL) {  parent->rightChild = tempNode;  return;  }  }  }   } }    Data Structure and Algorithms - AVL Trees  What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this:    Unbalanced BST    It is observed that BST's worst-case performance is closest to linear search algorithms, that is Ο(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.    Named after their inventor Adelson, Velski & Landis, AVL trees are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the Balance Factor.    Here we see that the first tree is balanced and the next two trees are not balanced:    Unbalanced AVL Trees    In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of A has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.    ***BalanceFactor*** = height(left-sutree) − height(right-sutree)  If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.    AVL Rotations  To balance itself, an AVL tree may perform the following four kinds of rotations:     * Left rotation * Right rotation * Left-Right rotation * Right-Left rotation     The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.    Left Rotation  If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation:    Left Rotation    In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left-subtree of B.    Right Rotation  AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.    Right Rotation    As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.    Left-Right Rotation  Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.     |  |  | | --- | --- | | **State** | **Action** | | Right Rotation | A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation. | | Left Rotation | We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**. | | Left Rotation | Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree. | | Right Rotation | We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree. | | Balanced Avl Tree | The tree is now balanced. |     Right-Left Rotation  The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.     |  |  | | --- | --- | | **State** | **Action** | | Left Subtree of Right Subtree | A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2. | | Subtree Right Rotation | First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**. | | Right Unbalanced Tree | Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation. | | Left Rotation | A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**. | | Balanced AVL Tree | The tree is now balanced. |     Data Structure & Algorithms - Spanning Tree  A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.    By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.    Spanning Trees    We found three spanning trees off one complete graph. A complete undirected graph can have maximum nn-2 number of spanning trees, where n is the number of nodes. In the above addressed example, n is 3, hence 33−2 = 3 spanning trees are possible.    General Properties of Spanning Tree  We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G:     * A connected graph G can have more than one spanning tree. * All possible spanning trees of graph G, have the same number of edges and vertices. * The spanning tree does not have any cycle (loops). * Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected. * Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.     Mathematical Properties of Spanning Tree     * Spanning tree has n-1 edges, where n is the number of nodes (vertices). * From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree. * A complete graph can have maximum nn-2 number of spanning trees.     Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.    Application of Spanning Tree  Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are:     * Civil Network Planning * Computer Network Routing Protocol * Cluster Analysis     Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.    Minimum Spanning Tree (MST)  In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.    Minimum Spanning-Tree Algorithm  We shall learn about two most important spanning tree algorithms here:     * [Kruskal's Algorithm](https://www.tutorialspoint.com/data_structures_algorithms/kruskals_spanning_tree_algorithm.htm) * [Prim's Algorithm](https://www.tutorialspoint.com/data_structures_algorithms/prims_spanning_tree_algorithm.htm)     Both are greedy algorithms.    Heap Data Structures  Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If α has child node β then:    key(α) ≥ key(β)    As the value of parent is greater than that of child, this property generates Max Heap. Based on this criteria, a heap can be of two types:    For Input → 35 33 42 10 14 19 27 44 26 31  **Min-Heap** − Where the value of the root node is less than or equal to either of its children.    Max Heap Example    **Max-Heap** − Where the value of the root node is greater than or equal to either of its children.    Max Heap Example    Both trees are constructed using the same input and order of arrival.    Max Heap Construction Algorithm  We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.    We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.     1. Create a new node at the end of heap. 2. Assign new value to the node. 3. Compare the value of this child node with its parent. 4. If value of parent is less than child, then swap them. 5. Repeat step 3 & 4 until Heap property holds.   Note − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.    Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.    Max Heap Animated Example    Max Heap Deletion Algorithm  Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.     1. Remove root node. 2. Move the last element of last level to root. 3. Compare the value of this child node with its parent. 4. If value of parent is less than child, then swap them. 5. Repeat step 3 & 4 until Heap property holds.   Max Heap Deletion Animated Example    Data Structure - Recursion Basics  Some computer programming languages allow a module or function to call itself. This technique is known as recursion. In recursion, a function α either calls itself directly or calls a function β that in turn calls the original function α. The function α is called recursive function.    Example − a function calling itself.  Example − a function that calls another function which in turn calls it again.    Properties  A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have:     * Base criteria − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively. * Progressive approach − The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.     Implementation  Many programming languages implement recursion by means of stacks. Generally, whenever a function (caller) calls another function (callee) or itself as callee, the caller function transfers execution control to the callee. This transfer process may also involve some data to be passed from the caller to the callee.    This implies, the caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function. Here, the caller function needs to start exactly from the point of execution where it puts itself on hold. It also needs the exact same data values it was working on. For this purpose, an activation record (or stack frame) is created for the caller function.    Activation Records    This activation record keeps the information about local variables, formal parameters, return address and all information passed to the caller function.    Analysis of Recursion  One may argue why to use recursion, as the same task can be done with iteration. The first reason is, recursion makes a program more readable and because of latest enhanced CPU systems, recursion is more efficient than iterations.    Time Complexity  In case of iterations, we take number of iterations to count the time complexity. Likewise, in case of recursion, assuming everything is constant, we try to figure out the number of times a recursive call is being made. A call made to a function is Ο(1), hence the (n) number of times a recursive call is made makes the recursive function Ο(n).    Space Complexity  Space complexity is counted as what amount of extra space is required for a module to execute. In case of iterations, the compiler hardly requires any extra space. The compiler keeps updating the values of variables used in the iterations. But in case of recursion, the system needs to store activation record each time a recursive call is made. Hence, it is considered that space complexity of recursive function may go higher than that of a function with iteration.    Data Structure & Algorithms - Tower of Hanoi  Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted −  Tower Of Hanoi    These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.    Rules  The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are:     * Only one disk can be moved among the towers at any given time. * Only the "top" disk can be removed. * No large disk can sit over a small disk.     Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.    Tower Of Hanoi    Tower of Hanoi puzzle with n disks can be solved in minimum 2n−1 steps. This presentation shows that a puzzle with 3 disks has taken 23 - 1 = 7 steps.    Algorithm  To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, source, destination and aux (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.  If we have 2 disks:     * First, we move the smaller (top) disk to aux peg. * Then, we move the larger (bottom) disk to destination peg. * And finally, we move the smaller disk from aux to destination peg.     Tower Of Hanoi with Two Disks    So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (nth disk) is in one part and all other (n-1) disks are in the second part.    Our ultimate aim is to move disk n from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.    The steps to follow are:     1. **Step 1** − Move n-1 disks from **source** to **aux** 2. **Step 2** − Move nth disk from **source** to **dest** 3. **Step 3** − Move n-1 disks from **aux** to **dest**     A recursive algorithm for Tower of Hanoi can be driven as follows:    START Procedure Hanoi(disk, source, dest, aux)  IF disk == 1, THEN  move disk from source to dest   ELSE  Hanoi(disk - 1, source, aux, dest) // Step 1  move disk from source to dest // Step 2  Hanoi(disk - 1, aux, dest, source) // Step 3  END IF   END Procedure STOP    Data Structure & Algorithms Fibonacci Series  Fibonacci series generates the subsequent number by adding two previous numbers. Fibonacci series starts from two numbers − F0 & F1. The initial values of F0 & F1 can be taken 0, 1 or 1, 1 respectively.    Fibonacci series satisfies the following conditions:    Fn = Fn-1 + Fn-2    Hence, a Fibonacci series can look like this:    F8 = 0 1 1 2 3 5 8 13    or, this:    F8 = 1 1 2 3 5 8 13 21    For illustration purpose, Fibonacci of F8 is displayed as:    Fibonacci Animation    Fibonacci Iterative Algorithm  First we try to draft the iterative algorithm for Fibonacci series.    Procedure Fibonacci(n)  declare f0, f1, fib, loop   set f0 to 0  set f1 to 1   **display f0, f1**   for loop ← 1 to n   fib ← f0 + f1   f0 ← f1  f1 ← fib  **display fib**  end for         end procedure    Fibonacci Recursive Algorithm  Let us learn how to create a recursive algorithm Fibonacci series. The base criteria of recursion.    START Procedure Fibonacci(n)  declare f0, f1, fib, loop     set f0 to 0  set f1 to 1    **display f0, f1**    for loop ← 1 to n    fib ← f0 + f1   f0 ← f1  f1 ← fib  **display fib**  end for  END | |
| |  | | --- | | **Summary** | |