**1) What is data structure?**

Data structures refers to the way data is organized and manipulated. It seeks to find ways to make data access more efficient. When dealing with data structure, we not only focus on one piece of data, but rather different set of data and how they can relate to one another in an organized manner.

**2) Differentiate file structure from storage structure.**

Basically, the key difference is the memory area that is being accessed. When dealing with the structure that resides the main memory of the computer system, this is referred to as storage structure. When dealing with an auxiliary structure, we refer to it as file structures.

**3) When is a binary search best applied?**

A binary search is an algorithm that is best applied to search a list when the elements are already in order or sorted. The list is search starting in the middle, such that if that middle value is not the target search key, it will check to see if it will continue the search on the lower half of the list or the higher half. The split and search will then continue in the same manner.

**4) What is a linked list?**

A linked list is a sequence of nodes in which each node is connected to the node following it. This forms a chain-like link of data storage.

**5) How do you reference all the elements in a one-dimension array?**

To do this, an indexed loop is used, such that the counter runs from 0 to the array size minus one. In this manner, we are able to reference all the elements in sequence by using the loop counter as the array subscript.

**6) In what areas do data structures applied?**

Data structures is important in almost every aspect where data is involved. In general, algorithms that involve efficient data structure is applied in the following areas: numerical analysis, operating system, A.I., compiler design, database management, graphics, and statistical analysis, to name a few.

**7) What is LIFO?**

LIFO is short for Last In First Out, and refers to how data is accessed, stored and retrieved. Using this scheme, data that was stored last , should be the one to be extracted first. This also means that in order to gain access to the first data, all the other data that was stored before this first data must first be retrieved and extracted.

**8 ) What is a queue?**

A queue is a data structures that can simulates a list or stream of data. In this structure, new elements are inserted at one end and existing elements are removed from the other end.

**9) What are binary trees?**

A binary tree is one type of data structure that has two nodes, a left node and a right node. In programming, binary trees are actually an extension of the linked list structures.

**10) Which data structures is applied when dealing with a recursive function?**

Recursion, which is basically a function that calls itself based on a terminating condition, makes use of the stack. Using LIFO, a call to a recursive function saves the return address so that it knows how to return to the calling function after the call terminates.

**11) What is a stack?**

A stack is a data structure in which only the top element can be accessed. As data is stored in the stack, each data is pushed downward, leaving the most recently added data on top.

**12) Explain Binary Search Tree**

A binary search tree stores data in such a way that they can be retrieved very efficiently. The left subtree contains nodes whose keys are less than the node’s key value, while the right subtree contains nodes whose keys are greater than or equal to the node’s key value. Moreover, both subtrees are also binary search trees.

**13) What are multidimensional arrays?**

Multidimensional arrays make use of multiple indexes to store data. It is useful when storing data that cannot be represented using a single dimensional indexing, such as data representation in a board game, tables with data stored in more than one column.

**14) Are linked lists considered linear or non-linear data structures?**

It actually depends on where you intend to apply linked lists. If you based it on storage, a linked list is considered non-linear. On the other hand, if you based it on access strategies, then a linked list is considered linear.

**15) How does dynamic memory allocation help in managing data?**

Aside from being able to store simple structured data types, dynamic memory allocation can combine separately allocated structured blocks to form composite structures that expand and contract as needed.

**16) What is FIFO?**

FIFO is short for First-in, First-out, and is used to represent how data is accessed in a queue. Data has been inserted into the queue list the longest is the one that is removed first.

**17) What is an ordered list?**

An ordered list is a list in which each node’s position in the list is determined by the value of its key component, so that the key values form an increasing sequence, as the list is traversed.

**18) What is merge sort?**

Merge sort takes a divide-and-conquer approach to sorting data. In a sequence of data, adjacent ones are merged and sorted to create bigger sorted lists. These sorted lists are then merged again to form an even bigger sorted list, which continuous until you have one single sorted list.

**19) Differentiate NULL and VOID.**

Null is actually a value, whereas Void is a data type identifier. A variable that is given a Null value simply indicates an empty value. Void is used to identify pointers as having no initial size.

**20) What is the primary advantage of a linked list?**

A linked list is a very ideal data structure because it can be modified easily. This means that modifying a linked list works regardless of how many elements are in the list.

**21) What is the difference between a PUSH and a POP?**

Pushing and popping applies to the way data is stored and retrieved in a stack. A push denotes data being added to it, meaning data is being “pushed” into the stack. On the other hand, a pop denotes data retrieval, and in particular refers to the topmost data being accessed.

**22) What is a linear search?**

A linear search refers to the way a target key is being searched in a sequential data structure. Using this method, each element in the list is checked and compared against the target key, and is repeated until found or if the end of the list has been reached.

**23) How does variable declaration affect memory allocation?**

The amount of memory to be allocated or reserved would depend on the data type of the variable being declared. For example, if a variable is declared to be of integer type, then 32 bits of memory storage will be reserved for that variable.

**24) What is the advantage of the heap over a stack?**

Basically, the heap is more flexible than the stack. That’s because memory space for the heap can be dynamically allocated and de-allocated as needed. However, memory of the heap can at times be slower when compared to that stack.

**25) What is a postfix expression?**

A postfix expression is an expression in which each operator follows its operands. The advantage of this form is that there is no need to group sub-expressions in parentheses or to consider operator precedence.

[](http://ebook.guru99.com/you-are-hired/)

**26) What is Data abstraction?**

Data abstraction is a powerful tool for breaking down complex data problems into manageable chunks. This is applied by initially specifying the data objects involved and the operations to be performed on these data objects without being overly concerned with how the data objects will be represented and stored in memory.

**27) How do you insert a new item in a binary search tree?**

Assuming that the data to be inserted is a unique value (that is, not an existing entry in the tree), check first if the tree is empty. If it’s empty, just insert the new item in the root node. If it’s not empty, refer to the new item’s key. If it’s smaller than the root’s key, insert it into the root’s left subtree, otherwise, insert it into the root’s right subtree.

**28) How does a selection sort work for an array?**

The selection sort is a fairly intuitive sorting algorithm,, though not necessarily efficient. To perform this, the smallest element is first located and switched with the element at subscript zero, thereby placing the smallest element in the first position. The smallest element remaining in the subarray is then located next with subscripts 1 through n-1 and switched with the element at subscript 1, thereby placing the second smallest element in the second position. The steps are repeated in the same manner till the last element.

**29) How do signed and unsigned numbers affect memory?**

In the case of signed numbers, the first bit is used to indicate whether positive or negative, which leaves you with one bit short. With unsigned numbers, you have all bits available for that number. The effect is best seen in the number range (unsigned 8 bit number has a range 0-255, while 8-bit signed number has a range -128 to +127.

**30) What is the minimum number of nodes that a binary tree can have?**

A binary tree can have a minimum of zero nodes, which occurs when the nodes have NULL values. Furthermore, a binary tree can also have 1 or 2 nodes.

**31) What are dynamic data structures?**

Dynamic data structures are structures that expand and contract as a program runs. It provides a flexible means of manipulating data because it can adjust according to the size of the data.

**32) In what data structures are pointers applied?**

Pointers that are used in linked list have various applications in data structure. Data structures that make use of this concept include the Stack, Queue, Linked List and Binary Tree.

**33) Do all declaration statements result in a fixed reservation in memory?**

Most declarations do, with the exemption of pointers. Pointer declaration does not allocate memory for data, but for the address of the pointer variable. Actual memory allocation for the data comes during run-time.

**34) What are ARRAYs?**

When dealing with arrays, data is stored and retrieved using an index that actually refers to the element number in the data sequence. This means that data can be accessed in any order. In programming, an array is declared as a variable having a number of indexed elements.

**35) What is the minimum number of queues needed when implementing a priority queue?**

The minimum number of queues needed in this case is two. One queue is intended for sorting priorities while the other queue is intended for actual storage of data.

**36) Which sorting algorithm is considered the fastest?**

There are many types of sorting algorithms: quick sort, bubble sort, balloon sort, radix sort, merge sort, etc. Not one can be considered the fastest because each algorithm is designed for a particular data structure and data set. It would depend on the data set that you would want to sort.

**37) Differentiate STACK from ARRAY.**

Data that is stored in a stack follows a LIFO pattern. This means that data access follows a sequence wherein the last data to be stored will the first one to be extracted. Arrays, on the other hand, does not follow a particular order and instead can be accessed by referring to the indexed element within the array.

**38) Give a basic algorithm for searching a binary search tree.**

1. if the tree is empty, then the target is not in the tree, end search  
   2. if the tree is not empty, the target is in the tree  
   3. check if the target is in the root item  
   4. if target is not in the root item, check if target is smaller than the root’s value  
   5. if target is smaller than the root’s value, search the left subtree  
   6. else, search the right subtree

**39) What is a dequeue?**

A dequeue is a double-ended queue. This is a structure wherein elements can be inserted or removed from either end.

**40) What is a bubble sort and how do you perform it?**

A bubble sort is one sorting technique that can be applied to data structures such as an array. It works by comparing adjacent elements and exchanges their values if they are out of order. This method lets the smaller values “bubble” to the top of the list, while the larger value sinks to the bottom.

**41) What are the parts of a linked list?**

A linked list typically has two parts: the head and the tail. Between the head and tail lie the actual nodes, with each node being linked in a sequential manner.

**42) How does selection sort work?**

Selection sort works by picking the smallest number from the list and placing it at the front. This process is repeated for the second position towards the end of the list. It is the simplest sort algorithm.

**43) What is a graph?**

A graph is one type of data structure that contains a set of ordered pairs. These ordered pairs are also referred to as edges or arcs, and are used to connect nodes where data can be stored and retrieved.

**44) Differentiate linear from non linear data structure.**

Linear data structure is a structure wherein data elements are adjacent to each other. Examples of linear data structure include arrays, linked lists, stacks and queues. On the other hand, non-linear data structure is a structure wherein each data element can connect to more than two adjacent data elements. Examples of non linear data structure include trees and graphs.

**45) What is an AVL tree?**

An AVL tree is a type of binary search tree that is always in a state of partially balanced. The balance is measured as a difference between the heights of the subtrees from the root. This self-balancing tree was known to be the first data structure to be designed as such.

**46) What are doubly linked lists?**

Doubly linked lists are a special type of linked list wherein traversal across the data elements can be done in both directions. This is made possible by having two links in every node, one that links to the next node and other one that links to the previous node.

**47) What is Huffman’s algorithm?**

Huffman’s algorithm is associated in creating extended binary trees that has minimum weighted path lengths from the given weights. It makes use of a table that contains frequency of occurrence for each data element.

**48) What is Fibonacci search?**

Fibonacci search is a search algorithm that applies to a sorted array. It makes use of a divide-and-conquer approach that can greatly reduce the time needed in order to reach the target element.

**49) Briefly explain recursive algorithm.**

Recursive algorithm targets a problem by dividing it into smaller, manageable sub-problems. The output of one recursion after processing one sub-problem becomes the input to the next recursive process.

**50) How do you search for a target key in a linked list?**

To find the target key in a linked list, you have to apply sequential search. Each node is traversed and compared with the target key, and if it is different, then it follows the link to the next node. This traversal continues until either the target key is found or if the last node is reached.

From data structure point of view, following are some important categories of algorithms −

* **Search** − Algorithm to search an item in a datastructure.
* **Sort** − Algorithm to sort items in certain order
* **Insert** − Algorithm to insert item in a datastructure
* **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 below mentioned characteristics −

* **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their input/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.

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** − The time is measured by counting the number of key operations such as comparisons in sorting algorithm
* **Space Factor** − The 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 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. 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 & constant 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 addition of two bits. Here, we observe that **T(n)** grows linearly as input size increases.

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 are 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, running time of one operation is computed as *f*(n) and may be for another operation it is computed as *g*(n2). Which means first operation running time will increase linearly with the increase in n and running time of second operation will increase exponentially when n increases. Similarly the running time of both operations will be nearly same if n is significantly small.

Usually, 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.

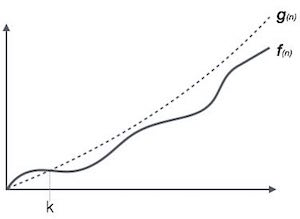
## Asymptotic Notations

Following are commonly used asymptotic notations used in calculating running time complexity of an algorithm.

* Ο Notation
* Ω Notation
* θ Notation

### **Big Oh Notation, Ο**

The 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 longest amount of time an algorithm can possibly take to complete.



For example, for a function *f*(n)

Ο(*f*(n)) = { *g*(n) : there exists c > 0 and n0 such that *g*(n) ≤ c.*f*(n) for all n > n0. }

### **Omega Notation, Ω**

The 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 best amount of time an algorithm can possibly take to complete.

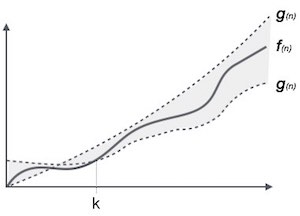


For example, for a function *f*(n)

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 θ(n) is the formal way to express both the lower bound and upper bound of an algorithm's running time. It is represented as following −



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

|  |  |  |
| --- | --- | --- |
| 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 Definition

Data Definition defines a particular data with following characteristics.

* **Atomic** − Definition should define a single concept
* **Traceable** − Definition should be be able to be mapped to some data element.
* **Accurate** − Definition should be unambiguous.
* **Clear and Concise** − Definition should be understandable.

## Data Object

Data Object represents an object having a data.

## Data Type

Data type is 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. Data type of two 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 provides 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 other way are known as derived data types. These data types are normally built by combination of primary or built-in data types and associated operations on them. For example −

* List
* Array
* Stack
* Queue

## Basic Operations

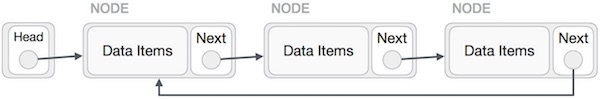
The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

* Traversing
* Searching
* Insertion
* Deletion
* Sorting
* Merging

Circular Linked List is a variation of Linked list in which first element points to last element and last element points to first element. Both Singly Linked List and Doubly Linked List can be made into as circular linked list.

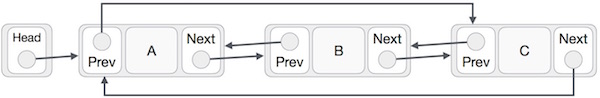
Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



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.



As per above shown illustrations, following are the important points to be considered.

* Last Link's next points to first link of the list in both cases of singly as well as doubly linked list.
* First Link's prev 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** − insert an element in the start of the list.
* **delete** − insert an element from the start of the list.
* **display** − display the list.

The way to write arithmetic expression is known as **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of 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 **prefix**ed 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, operator is **postfix**ed to the operands i.e., operator is written after the operands. For example **ab+**. This is equivalent to its infix notation **a+b**.

The below table briefly tries to show difference in all three notations −

|  |  |  |  |
| --- | --- | --- | --- |
| **S.n.** | **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 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 computated.

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 operator, 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 firs. A table of operator precedence is provided later.

### **Associativity**

Associativity describes the rule where operators with same precedence appear in an expression. For example, in expression **a+b−c**, both + and − has same precedence, then which part of 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. An operator precedence and associativity table is given below (highest to lowest) −

|  |  |  |  |
| --- | --- | --- | --- |
| **S.n.** | **Operator** | **Precedence** | **Associativity** |
| 1 | Esponentiation **^** | 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, as multiplication as precedence over addition. We here use parenthesis to make **a+b** be evaluated first, like **(a+b)\*c**.

## Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation −

Step 1 − scan the expression from left to right

Step 2 − if it is an operand push it to stack

Step 3 − if it is an operator pull operand from stack and perform operation

Step 4 − store the output of step 3, back to stack

Step 5 − scan the expression until all operands are consumed

Step 6 − pop the stack and perform operation

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 −



In the above graph,

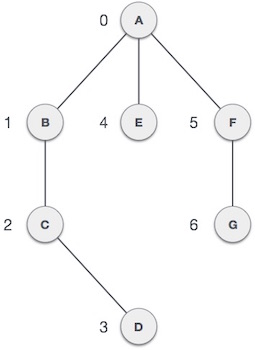
V = {a, b, c, d, e}

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 example given below, labeled circle represents vertices. So A to G are vertices. We can represent them using an array as shown in image below. 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 example given below, lines from A to B, B to C and so on represents edges. We can use a two dimensional array to represent array as shown in image below. 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 example given below, B is adjacent to A, C is adjacent to B and so on.
* **Path** − Path represents a sequence of edges between two vertices. In example given below, ABCD represents a path from A to D.



Basic Operations

Following are basic primary operations of a Graph which are following.

* **Add Vertex** − add a vertex to a graph.
* **Add Edge** − add an edge between two vertices of a graph.
* **Display Vertex** − display a vertex of a graph.

Depth First Search algorithm(DFS) 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.



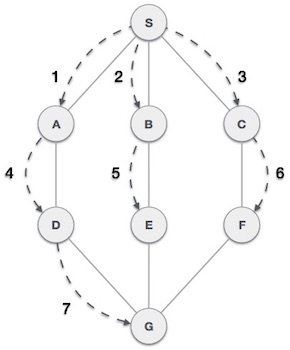
As in example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs following rules.

* **Rule 1** − Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack.
* **Rule 2** − If no adjacent vertex found, pop up a vertex from 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 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 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 visited and put onto the stack. Here we have **B** and **C** nodes which are adjacent to **D** and both are unvisited. But we shall again choose in alphabetical order. |
| 5. | Depth First Search Step Five | We choose **B**, mark it visited and put onto 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 stack top for return to previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of stack. |
| 7. | Depth First Search Step Seven | Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it 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 which has unvisited adjacent node. In this case, there's none and we keep popping until stack is empty.

Breadth First Search algorithm(BFS) traverses a graph in a breadthwards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.



As in 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 following rules.

* **Rule 1** − Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue.
* **Rule 2** − If no adjacent vertex found, remove the first vertex from queue.
* **Rule 3** − Repeat Rule 1 and Rule 2 until 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 visited. |
| 3. | Breadth First Search Step Three | We then see unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A** mark it visited and enqueue it. |
| 4. | Breadth First Search Step Four | Next unvisited adjacent node from **S** is **B**. We mark it visited and enqueue it. |
| 5. | Breadth First Search Step Five | Next unvisited adjacent node from **S** is **C**. We mark it 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 visited and enqueue it. |

At this stage we are left with no unmarked (unvisited) nodes. But as per algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied the program is over.

# **Data Structure - Tree**

Tree represents nodes connected by edges. We'll going to 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 two children at maximum. A binary tree have benefits of both an ordered array and a linked list as search is as quick as in sorted array and insertion or deletion operation are as fast as in linked list.



Terms

Following are important terms with respect to tree.

* **Path** − Path refers to sequence of nodes along the edges of a tree.
* **Root** − Node at the top of the tree is called root. There is only one root per tree and one path from root node to any node.
* **Parent** − Any node except root node has one edge upward to a node called parent.
* **Child** − Node below a given node connected by its edge downward is called its child node.
* **Leaf** − Node which does not have any child node is called leaf node.
* **Subtree** − Subtree represents descendents of a node.
* **Visiting** − Visiting refers to checking 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 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 behaviour. A node's left child must have value less than its parent's value and node's right child must have value greater than it's parent value.



We're going to implement tree using node object and connecting them through references.

Node

A tree node should look like the below structure. It has data part and references to its left and right child nodes.

struct node {

int data;

struct node \*leftChild;

struct node \*rightChild;

};

In a tree, all nodes share common construct.

BST Basic Operations

The basic operations that can be performed on binary search tree data structure, are following −

* **Insert** − insert an element in a tree / create a tree.
* **Search** − search an element in a tree.
* **Preorder Traversal** − traverse a tree in a preorder manner.
* **Inorder Traversal** − traverse a tree in an inorder manner.
* **Postorder Traversal** − traverse a tree in a postorder manner.

We shall learn creating (inserting into) tree structure and searching a data-item in a tree in this chapter. We shall learn about tree traversing methods in the coming one.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted. First locate its proper location. Start search from root node then if data is less than key value, search empty location in left subtree and insert the data. Otherwise search empty location in 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

Implementation

The implementation of insert function should look like this −

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, create root node

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;

}

}

}

}

}

Search Operation

Whenever an element is to be search. Start search from root node then if data is less than key value, search element in left subtree otherwise search element in 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

The implementation of this algorithm should look like this.

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;

}

}

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 random access a node in 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 given item or key in the tree or to print all the values it contains.

## Inorder Traversal

In this traversal method, the left left-subtree is visited first, then root and then the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **inorder**, the output will produce sorted key values in ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**.**B** is also traversed in-ordered. And the process goes on until all the nodes are visited. The output of in-order traversal of this tree will be −

***D → B → E → A → F → C → G***

### **Algorithm**

Until all nodes are traversed −

**Step 1** − Recursively traverse left subtree.

**Step 2** − Visit root node.

**Step 3** − Recursively traverse right subtree.

## Preorder Traversal

In this traversal method, the root node is visited first, then left subtree and finally right sub-tree.



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-ordered. And 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 −

**Step 1** − Visit root node.

**Step 2** − Recursively traverse left subtree.

**Step 3** − Recursively traverse right subtree.

## Postorder Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse left subtree, then right subtree and finally root.



We start from **A**, and following pre-order traversal, we first visit left subtree **B**.**B** is also traversed post-ordered. And 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 −

**Step 1** − Recursively traverse left subtree.

**Step 2** − Recursively traverse right subtree.

**Step 3** − Visit root node.

A binary search tree (BST) is a tree in which all nodes follows the below mentioned properties −

* The left sub-tree of a node has key less than or equal to its parent node's key.
* The right sub-tree of a node has key greater than or equal to its parent node's key.

Thus, a binary search tree (BST) divides all its sub-trees into two segments;*left* sub-tree and *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 key and associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

An example of BST −



We observe that the root node key (27) has all less-valued keys on the left sub-tree and higher valued keys on the right sub-tree.

Basic Operations

Following are basic primary operations of a tree which are following.

* **Search** − search an element in a tree.
* **Insert** − insert an element in a tree.
* **Preorder Traversal** − traverse a tree in a preorder manner.
* **Inorder Traversal** − traverse a tree in an inorder manner.
* **Postorder Traversal** − traverse a tree in a postorder 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 search. Start search from root node then if data is less than key value, search element in left subtree otherwise search element in right subtree. Follow the same algorithm for each node.

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 search from root node then if data is less than key value, search empty location in left subtree and insert the data. Otherwise search empty location in right subtree and insert the data.

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;

}

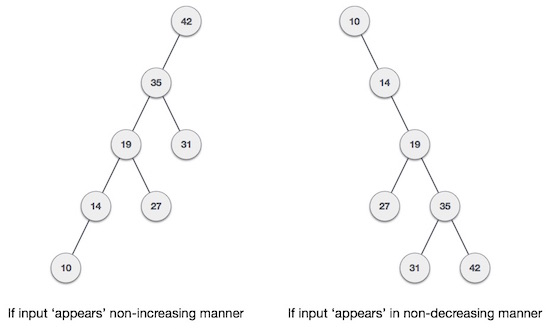
}

}

}

}

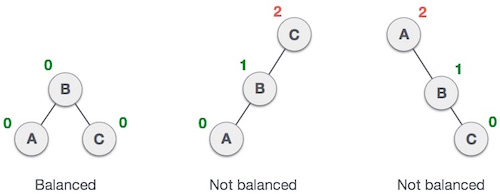
What if the input to binary search tree comes in sorted (ascending or descending) manner? It will then look like this −



It is observed that BST's worst-case performance closes 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 existing BST.

Named after their inventor ***Adelson***, ***Velski*** & ***Landis***, **AVL** trees are height balancing binary search tree. AVL tree checks the height of left and right sub-trees and assures that the difference is not more than 1. This difference is called ***Balance Factor***.

Here we see that the first tree is balanced and next two trees are not balanced −



In second tree, the left subtree of **C** has height 2 and right subtree has height 0, so the difference is 2. In third tree, the right subtree of **A** has height 2 and 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 make itself balanced, an AVL tree may perform four kinds of rotations −

* Left rotation
* Right rotation
* Left-Right rotation
* Right-Left rotation

First two rotations are single rotations and next two rotations are double rotations. Two 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 become unbalanced, when a node is inserted into the right subtree of right subtree, then we perform single left rotation −



In our example, node **A** has become unbalanced as a node is inserted in right subtree of A's right subtree. We perform left rotation by making **A** left-subtree of B.

Right Rotation

AVL tree may become unbalanced if a node is inserted in the left subtree of left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes 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 combination of left rotation followed by right rotation.

|  |  |
| --- | --- |
| **State** | **Action** |
| Right Rotation | A node has been inserted into right subtree of left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation. |
|  |  |
| Left Rotation | We first perform left rotation on left subtree of **C**. This makes **A**, left subtree of **B**. |
|  |  |
| Left Rotation | Node **C** is still unbalanced but now, it is because of left-subtree of left-subtree. |
|  |  |
| Right Rotation | We shall now right-rotate the tree making **B** new root node of this subtree. **C** now becomes right subtree of its own left subtree. |
|  |  |
| Balanced Avl Tree | The tree is now balanced. |

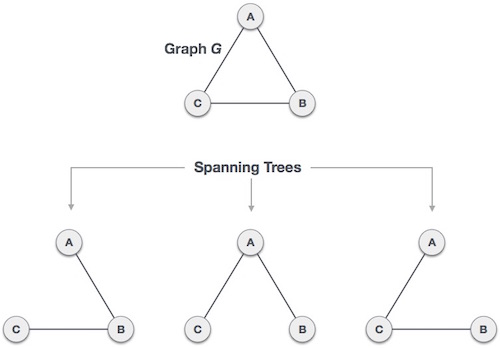
Right-Left Rotation

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 left subtree of right subtree. This makes **A** an unbalanced node, with balance factor 2. |
|  |  |
| Subtree Right Rotation | First, we perform right rotation along **C** node, making **C**the right subtree of its own left subtree **B**. Now, **B**becomes right subtree of **A**. |
|  |  |
| Right Unbalanced Tree | Node **A** is still unbalanced because of 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 left subtree of its right subtree **B**. |
|  |  |
| Balanced AVL Tree | The tree is now balanced. |

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 can not be disconnected.

By this definition we can draw a conclusion that every connected & undirected Graph G has at least one spanning tree. A disconnected graph do not have any spanning tree, as it can not spanned to all its vertices.



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 number of nodes. In 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 trees. Below are few properties is spanning tree of given connected graph G −

* A connected graph G can have more than one spanning tree.
* All possible spanning trees of graph G, have same number of edges and vertices.
* Spanning tree does not have any cycle (loops)
* Removing one edge from spanning tree will make the graph disconnected i.e. spanning tree is **minimally connected**.
* Adding one edge to a spanning tree will create a circuit or loop i.e. spanning tree is **maximally acyclic**.

Mathematical properties of spanning tree

* Spanning tree has n-1 edges, where n is 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.

So we can conclude here that spanning trees are subset of a connected Graph G and disconnected Graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find minimum paths to connect all nodes in a graph. Common application of spanning trees are −

* **Civil Network Planning**
* **Computer Network Routing Protocol**
* **Cluster Analysis**

Lets understand this by a small example. Consider city network as a huge graph and now plan to deploy telephone lines such a way that in minimum lines we can connect to all city nodes. This is where spanning tree comes in the picture.

Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight that 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 −

Heap is a special case of balanced binary tree data structure where 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 root node is less than or equal to either of its children.



* **Max-Heap** − where the value of root node is greater than or equal to either of its children.



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 ones.

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 already heapified tree.

**Step 1 −** Create a new node at the end of heap.

**Step 2 −** Assign new value to the node.

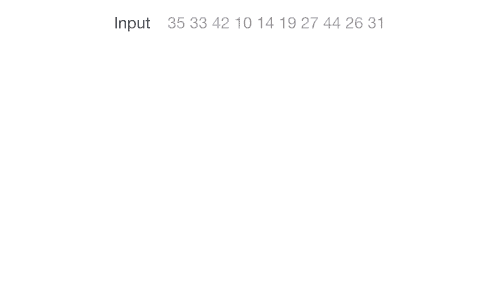
**Step 3 −** Compare the value of this child node with its parent.

**Step 4 −** If value of parent is less than child, then swap them.

**Step 5 −** Repeat step 3 & 4 until Heap property holds.

**Note** − In Min Heap construction algorithm we expect the value of parent node to be less than that of child node.

Let's understand Max Heap construction by an animated illustration. We take the same input sample that we use earlier.



Max Heap Deletion Algorithm

Lets derive an algorithm to delete from max-heap. Deletion in Max (or Min) Heap is always happen at the root to remove the Maximum (or minimum) value.

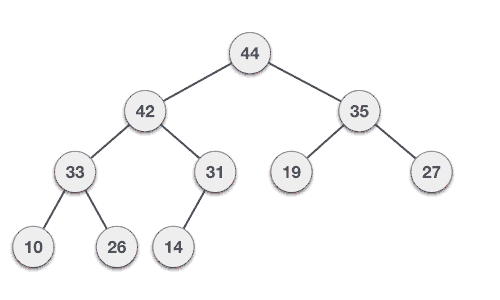
**Step 1 −** Remove root node.

**Step 2 −** Move the last element of last level to root.

**Step 3 −** Compare the value of this child node with its parent.

**Step 4 −** If value of parent is less than child, then swap them.

**Step 5 −** Repeat step 3 & 4 until Heap property holds.



# **Data Structure - Recursion Basics**

Some computer programming languages allows a module or function to call itself. This technique is known as recursion. In recursion, a fuction **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.**

int function(int value) {

if(value < 1)

return;

function(value - 1);

printf("%d ",value);

}

**Example − a function that calls another function which in turn calls it again.**

int function(int value) {

if(value < 1)

return;

function(value - 1);

printf("%d ",value);

}

## Properties

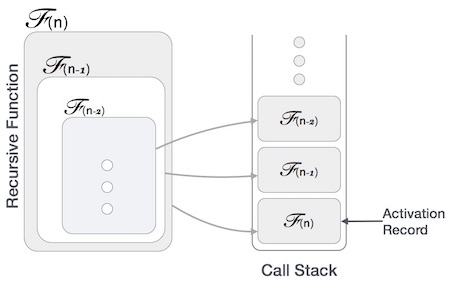
A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have −

* **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 callee. This transfer process may also involve some data to be passed from caller to callee.

This implies, the caller function has to suspend its execution temporarily and resume later when the execution control returns from callee function. Here, caller function needs to start exactly from the point of execution where it put 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 caller function.



This activation record keeps the information about local variables, formal parameters, return address and all informations passed to called function.

## Analysis of recursion

One may argue that 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 today's enhance CPU systems, recursion is more efficient than iterations.

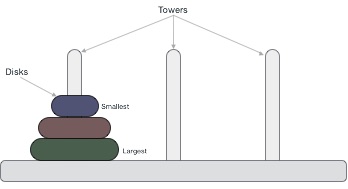
### **Time complexity**

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 time recursive call is being made. A call made to a function is Ο(1), hence the (n) number of time 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. 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. So it is considered that space complexity of recursive function may go higher than that of a function with iteration.

Tower of Hanoi, is a mathematical puzzle which consists of three tower (pegs) and more than one rings; as depicted below −



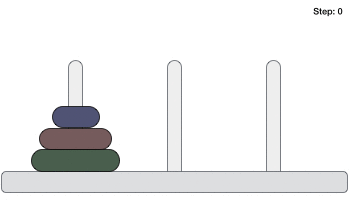
These rings are of different sizes and stacked upon in ascending order i.e. the smaller one sits over the larger one. There are other variations of puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. The below mentioned are few rules which are to be followed for tower of hanoi −

* 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.

Here is an animated representation of solving a tower of hanoi puzzle with three disks −



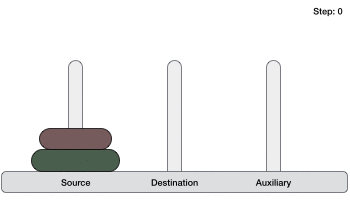
Tower of hanoi puzzle with **n** disks can be solved in minimum **2n−1** steps. This presentation shows that a puzzle with 3 disks has taken **23−1** = 7 steps.

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 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 one (top) disk to aux peg
* Then we move the larger one (bottom) disk to destination peg
* And finally, we move the smaller one from aux to destination peg.



So now we are in a position to design algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (nthdisk) is in one part and all other (n-1) disks are in second part.

Our ultimate aim is to move disk n from source to destination and then put all other (n-1) disks onto it. Now we can imagine to apply the same in recursive way for all given set of disks.

So steps to follow are −

**Step 1 −** Move n-1 disks from **source** to **aux**

**Step 2 −** Move nth disk from **source** to **dest**

**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 == 0, 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