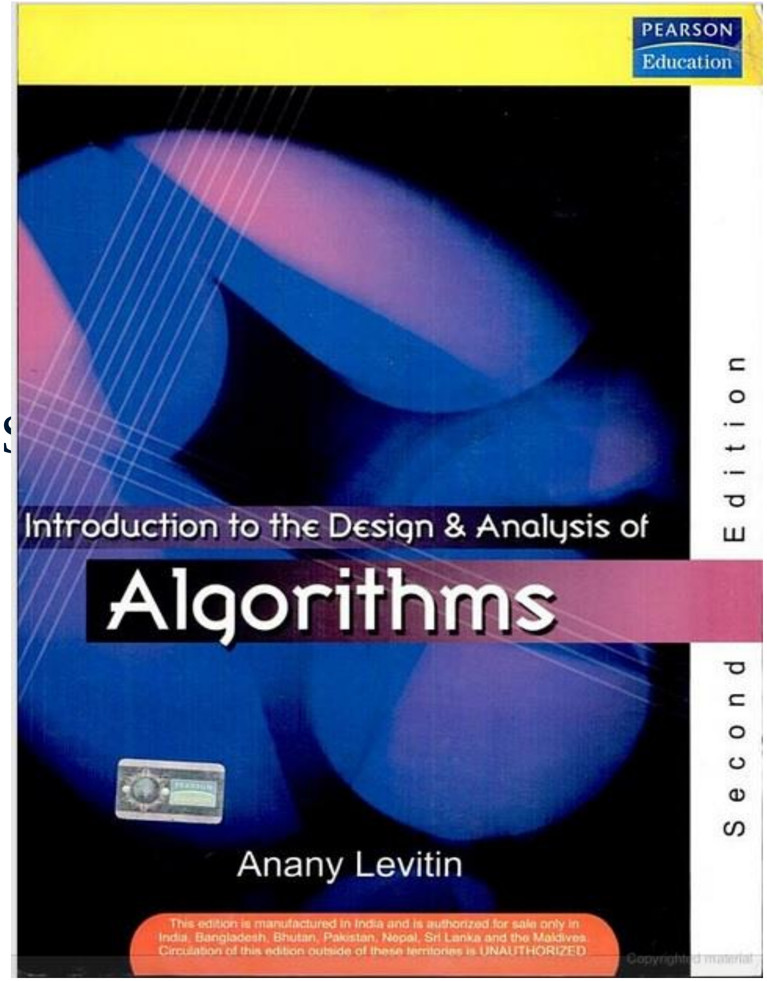
Zero Lecture
on
Design and Analysis of Algorithm
CSE408

Course Details

- LTP – 3 0 0 [Three lectures/week]

- Text Book
  - INTRODUCTION TO THE DESIGN AND ANALYSIS OF ALGORITHMS
    – ANANY LEVITIN,
    PEARSON EDUCATION



Detail of Academic Tasks

- AT1: Test1 Lecture #11(Before MTE)
- AT2: Test2 Lecture #19(Before MTE)
- AT3: Test3 Lecture #33(After MTE)

➢ Best 2 will be considered with the conditions as :

- AT1 or AT2
- AT3 will be mandatory

Weight age of AT,MTT,ETT

- Attendance: 5 Marks
- Academic Tasks(CA): 25 Marks
- MTT : 20 Marks
- ETT: 50 Marks

Need to study this course .

Why are we learning Design and analysis of algorithms?

- Algorithms are used in almost every program or software system.
- Once we design an algorithm we need to know how well it performs on any input.
- In particular we would like to know whether there are better algorithms for the problem, An answer to this first demands a way to analyze an algorithm in a machine-independent way.
- Some Specific design techniques are essential ingredients of many software applications.

The course contents CSE408

- UNIT I - Foundations of algorithm
- UNIT II - String matching algorithms and computational geometry
- UNIT III - Divide and conquer and ordered statists
- UNIT IV - Dynamic programming and greedy techniques
- UNIT V - Backtracking and approximation algorithms
- UNIT VI - Number-theoretic algorithms and complexity classes

CSE408:DESIGN AND ANALYSIS OF ALGORITHMS

Course Outcomes: Through this course students should be able to

CO1 :: explain the basic techniques of analyzing the algorithms using space and time complexity, asymptotic notations

CO2 :: analyse various string matching algorithms and understand brute force algorithm design technique

CO3 :: understand divide and conquer algorithm design technique using various searching and sorting algorithms

CO4 :: define dynamic programming and greedy algorithm design technique and solve various all pair and single source shortest path problems

CO5 :: apply the backtracking method to solve some classic problems and understand branch and bound algorithm design technique

CO6 :: define various number theory problems and understand the basics concepts of complexity classes

Unit I

Foundations of Algorithm : Algorithms, Fundamentals of Algorithmic Problem Solving:, Basic Algorithm Design Techniques, Analyzing Algorithm, Fundamental Data Structure:, Linear Data Structure, Graphs and Trees, Fundamentals of the Analysis of Algorithm Efficiency:, Measuring of Input Size, Units for Measuring Running Time, Order of Growth, Worst-Case, Best-Case, and Average-Case Efficiencies, Asymptotic Notations and Basic Efficiency Classes:, O(Big-oh)-notation, Big-omega notation, Big-theta notation, Useful Property Involving the Asymptotic Notations, Using Limits for Comparing Orders of Growth

Unit II

String Matching Algorithms and Computational Geometry : Sequential Search and Brute-Force String Matching, Closest-Pair and Convex-Hull Problem, Exhaustive Search, Voronoi Diagrams, Naiva String-Matching Algorithm, Rabin-Karp Algorithm, Knuth-Morris-Pratt Algorithm

Unit III

Divide and Conquer and Order Statistics : Merge Sort and Quick Sort, Binary Search, Multiplication of Large Integers, Strassen's Matrix Multiplication, Substitution Method for Solving Recurrences, Recursion-Tree Method for Solving Recurrences, Master Method for Solving Recurrence, Closest-Pair and Convex-Hull Problems by Divide and Conquer, Decrease and Conquer: Insertion Sort, Depth-First Search and Breadth-First Search, Connected Components, Topological Sort, Transform and Conquer: Presorting, Balanced Search Trees, Minimum and Maximum, Counting Sort, Radix Sort, Bucket Sort, Heaps and Heapsort, Hashing, Selection Sort and Bubble Sort

Unit IV

Dynamic Programming and Greedy Techniques : Dynamic Programming: Computing a Binomial Coefficient, Warshall's and Floyd's Algorithm, Optimal Binary Search Trees, Knapsack Problem and Memory Functions, Matrix-Chain Multiplication, Longest Common Subsequence, Greedy Technique and Graph Algorithm: Minimum Spanning Trees, Prim's Algorithm, Kruskal's Algorithm, Dijkstra's Algorithm, Huffman Code, Single-Source Shortest Paths, All-Pairs Shortest Paths, Iterative Improvement: The Maximum-Flow Problem, Limitations of Algorithm Power: Lower-Bound Theory

Unit V

Backtracking and Approximation Algorithms : Backtracking: n-Queens Problem, Hamiltonian Circuit Problem, Subset-Sum Problem, Branch-and-Bound: Assignment Problem, Knapsack Problem, Traveling Salesman Problem, Vertex-Cover Problem and Set-Covering Problem, Bin Packing Problems

Unit VI

Number-Theoretic Algorithms and Complexity Classes : Number Theory Problems: Modular Arithmetic, Chinese Remainder Theorem, Greatest Common Divisor, Optimization Problems, Basic Concepts of Complexity Classes- P, NP, NP-hard, NP-complete Problems

Course Outcomes

CO1 ::Explain the basic techniques of analyzing the algorithms using space and time complexity, asymptotic notations

CO2 ::Analyze various string matching algorithms and understand brute force algorithm design technique

CO3 :: Understand divide and conquer algorithm design technique using various searching and sorting algorithms

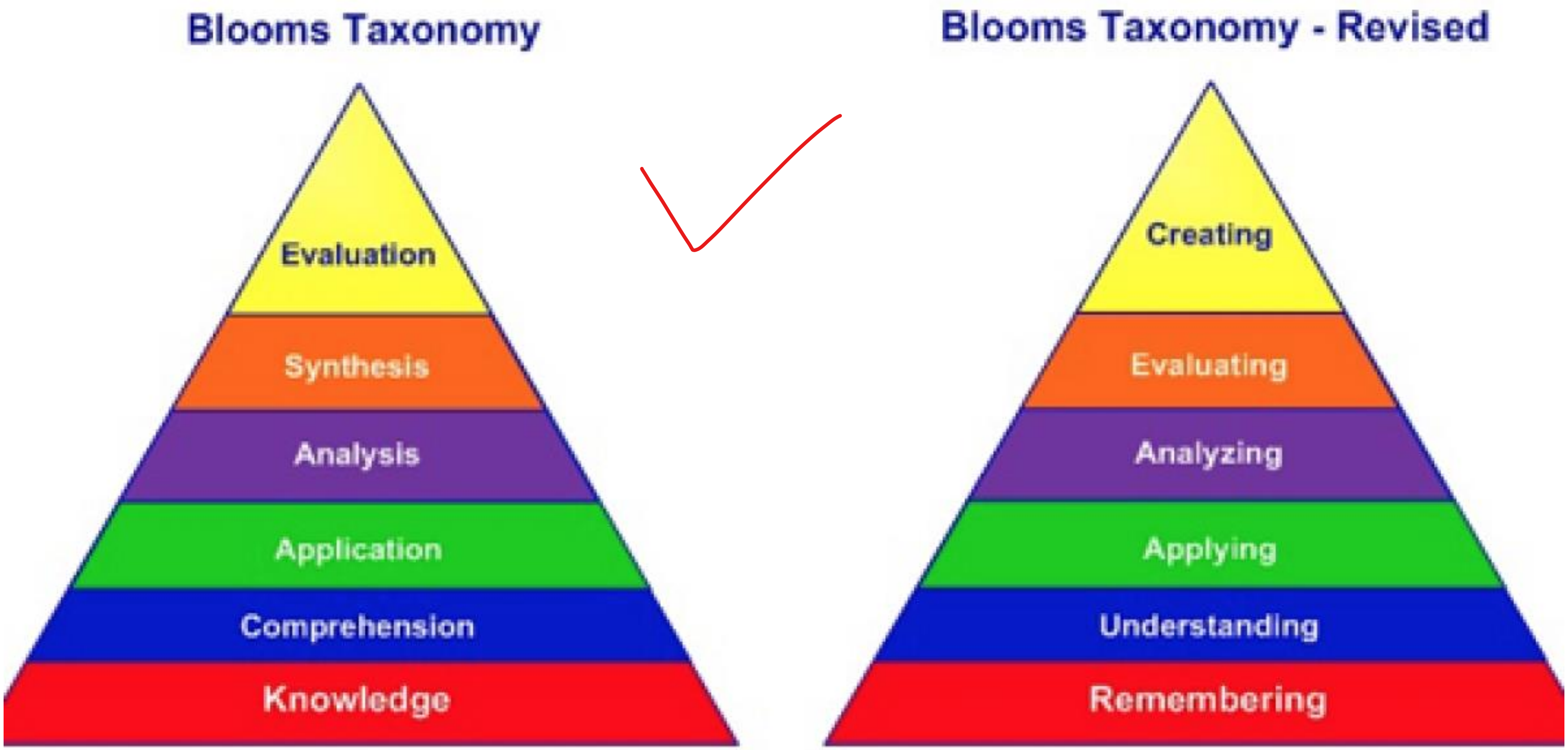
..contd..

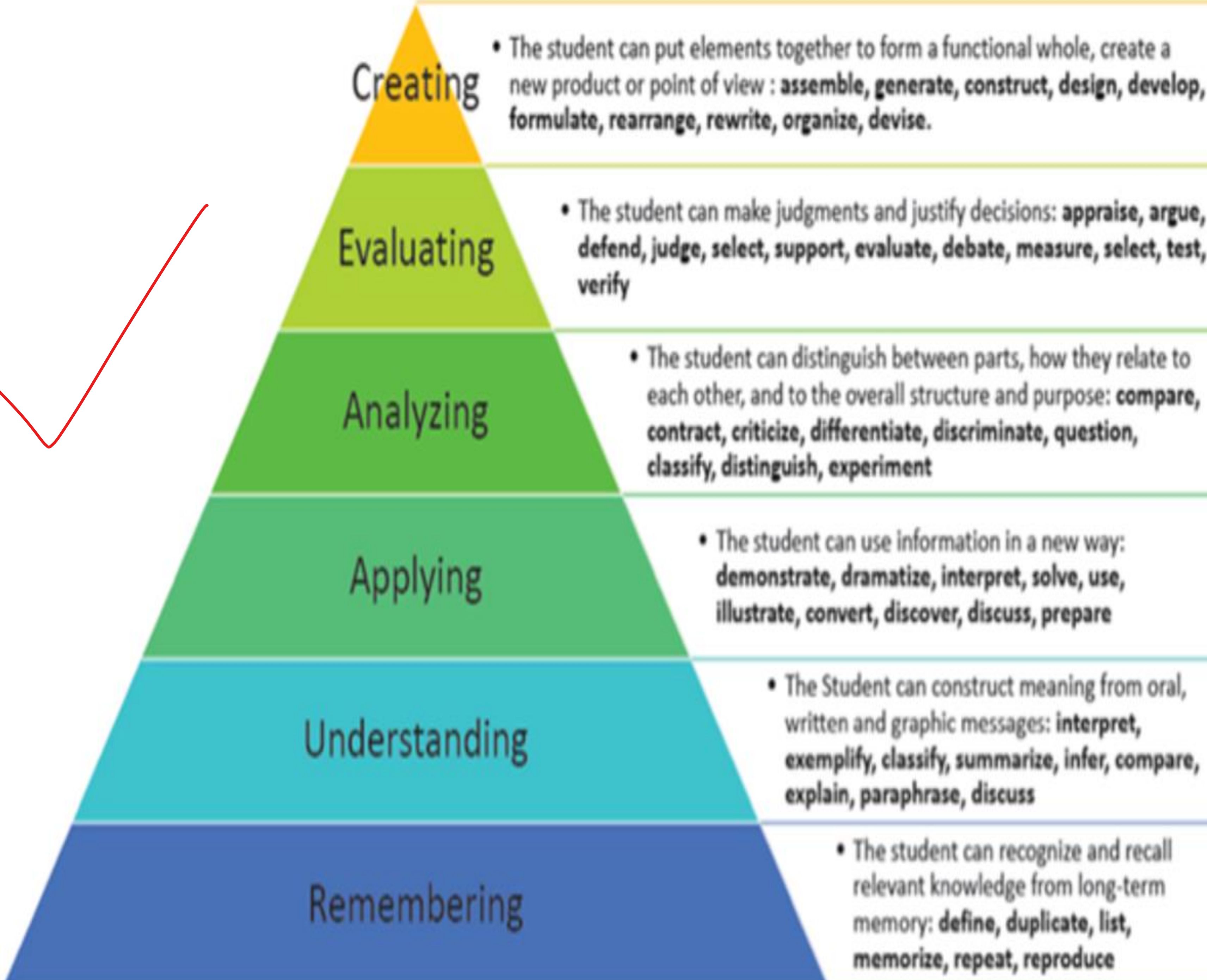
CO4 :: define dynamic programming and greedy algorithm design technique and solve various all pair and single source shortest path problems

CO5 :: apply the backtracking method to solve some classic problems and understand branch and bound algorithm design technique

CO6 :: define various number theory problems and understand the basics concepts of complexity classes

RBT– Revised Blooms Tax anomy




Creating
• The student can put elements together to form a functional whole, create a new product or point of view : **assemble, generate, construct, design, develop, formulate, rearrange, rewrite, organize, devise.**
Evaluating
• The student can make judgments and justify decisions: **appraise, argue, defend, judge, select, support, evaluate, debate, measure, select, test, verify**
Analyzing
• The student can distinguish between parts, how they relate to each other, and to the overall structure and purpose: **compare, contract, criticize, differentiate, discriminate, question, classify, distinguish, experiment**
Applying
• The student can use information in a new way: **demonstrate, dramatize, interpret, solve, use, illustrate, convert, discover, discuss, prepare**
Understanding
• The Student can construct meaning from oral, written and graphic messages: **interpret, exemplify, classify, summarize, infer, compare, explain, paraphrase, discuss**
Remembering
• The student can recognize and recall relevant knowledge from long-term memory: **define, duplicate, list, memorize, repeat, reproduce**

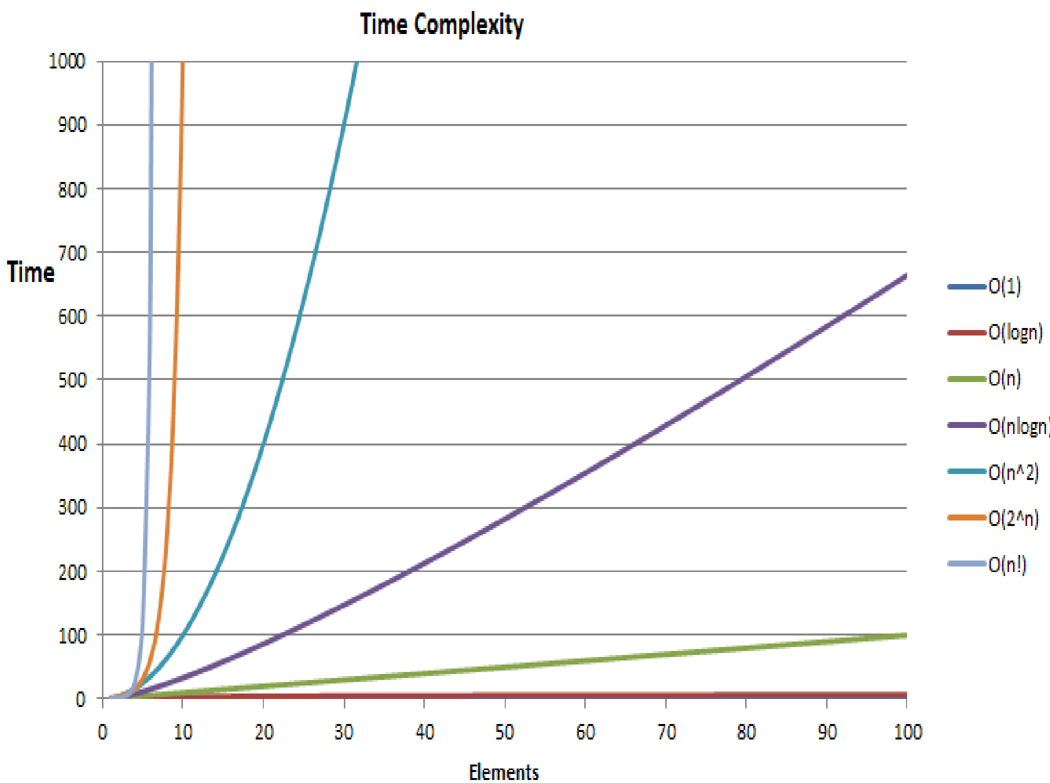
MOOC-Massive Open Online Course Approved (MOOC Course)

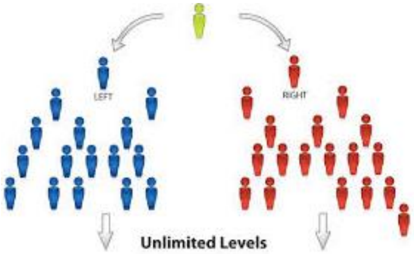
1. NPTEL

   https://nptel.ac.in/courses/106106131 /

Benefit to register the Approved MOOC is ***<u>All CAs</u>***

Foundations of algorithm

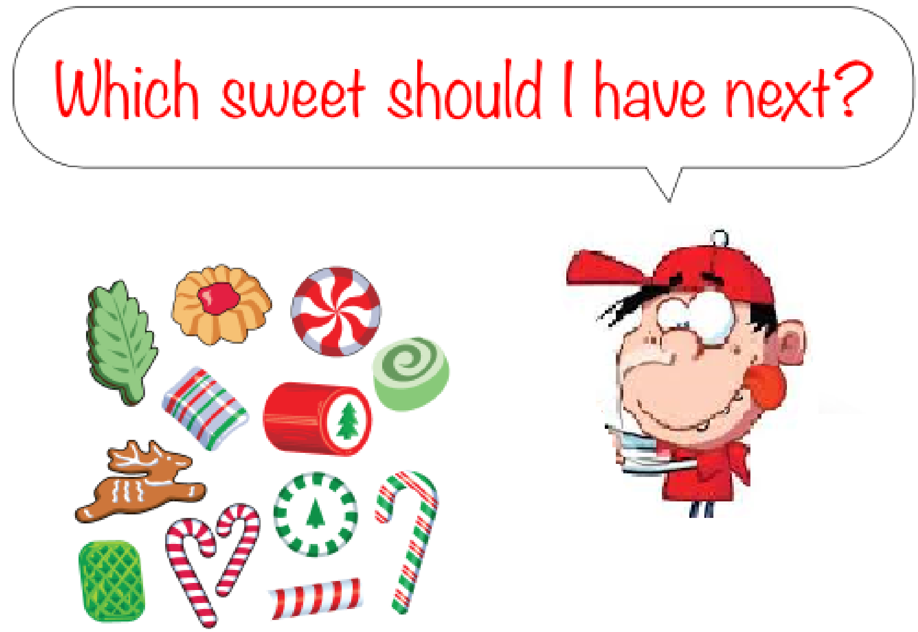




String matching algorithms and computational geometry

Divide and conquer and ordered statists

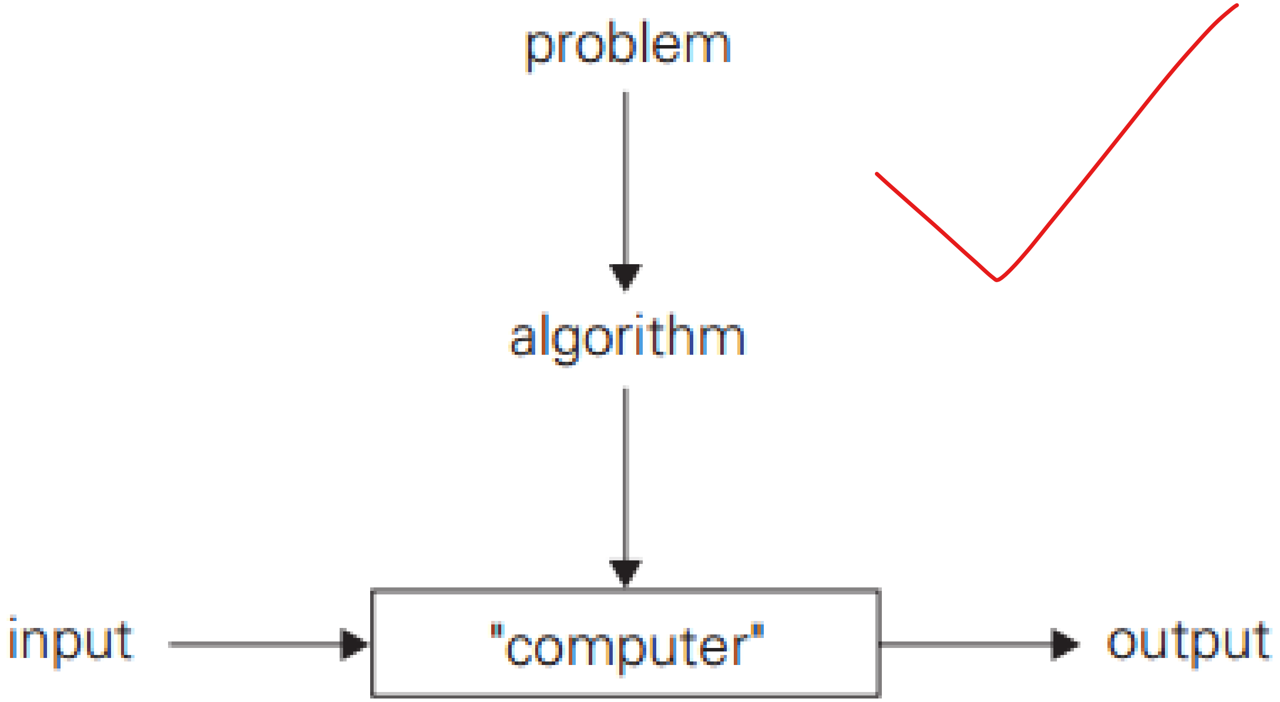
Dynamic programming and greedy technique



Algorithm

An ***algorithm*** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

- The nonambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.


problem
algorithm
input
"computer"
output

Problem for Swapping 2 Numbers

Using Temporary Variable	Without using temporary Variable

void swap(int a, int b)
{
int temp;
temp = a;
a = b;
b = temp;
}
	void swap(int a, int b)
	{
	a = a+b;
	b = a-b;
	a = a-b;
	}

Fundamentals of algorithmic problem solving

- Understanding the problem
- Ascertaining the capabilities of a computational device.
- Choosing between exact and approximate problem solving.
- Deciding an appropriate Data Structure
- Algorithm design techniques.
- Methods of specifying an algorithm.
- Proving an algorithms correctness
- Analyzing an algorithm.
- Coding an algorithm

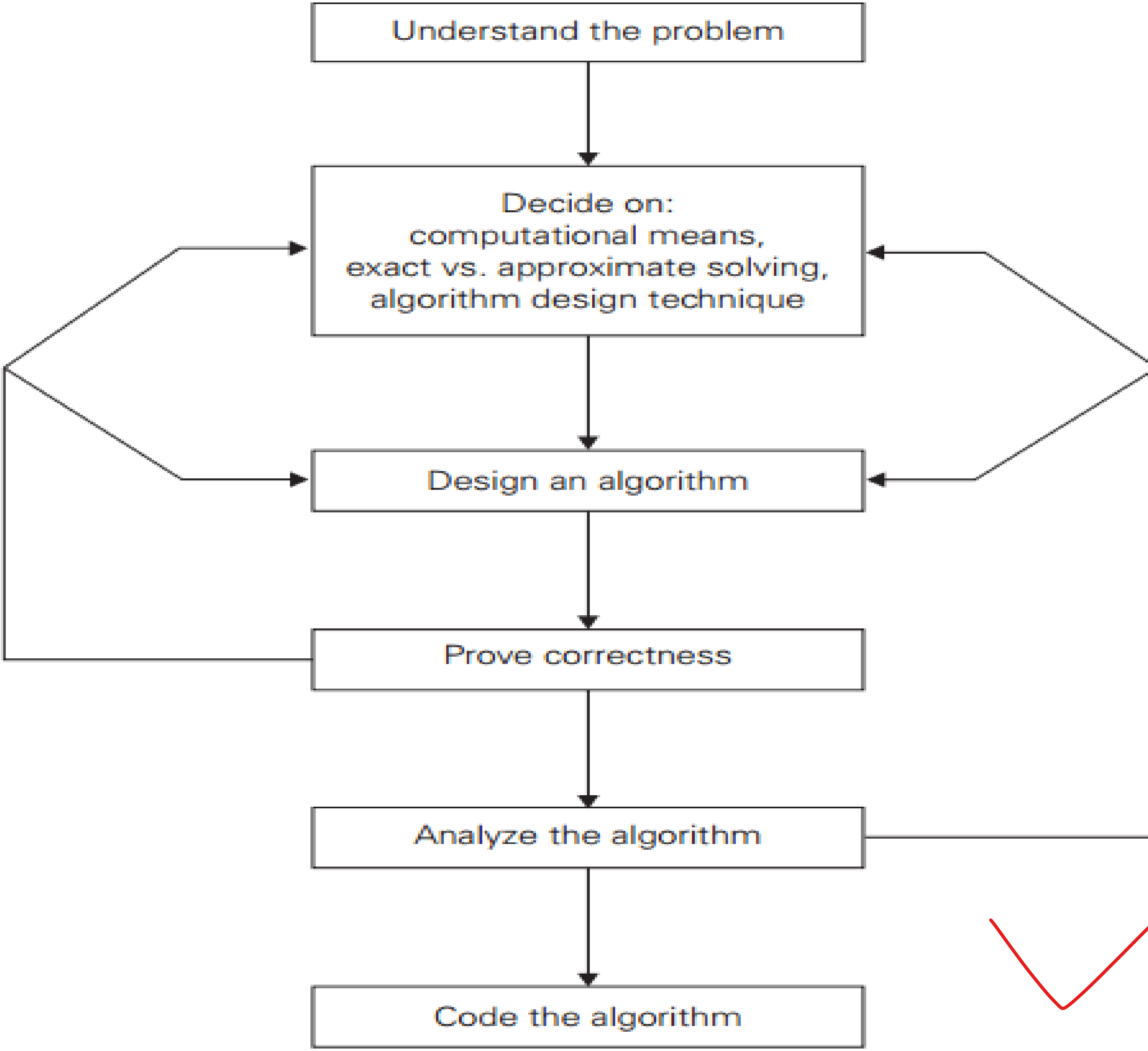


FIGURE 1.2 Algorithm design and analysis process.

Step 1- Understanding the problem

- Understand the given problem completely.
- Read the problem description carefully
- Ask doubts about the problem
- Do few examples
- Think about special cases like boundary value
- Know about the already known algorithms for solving that problem
- Clear about the input to an algorithm.

a computational device

- Sequential Algorithms: Instructions are executed one after another and one operation at a time. Algorithms designed to be executed on such machine are called sequential algorithms
- Parallel Algorithms: Modern computers that can execute instructions in parallel. Algorithms designed to be executed on such machine are called parallel algorithms

approximate problem solving

- Exact Algorithm: Solving problem exactly
- Approx. Algm. : Solving problem approximately
- Why Approx. Algm.?
- Some problems cannot be solved exactly
- Solving problem exactly can be unacceptably slow because of the problem complexity.

Step 4- Deciding on appropriate Data Structures

- Decide on the suitable data structure
- A Data structure is defined as a particular scheme of organizing related data items.

Step 5- Algorithm Design techniques

- An algorithm design technique is a general approach to solve problems algorithmically.
- The various design techniques are:
- Brute force
- Divide and conquer
  - Greedy approach
  - Dynamic programming
  - Backtracking

Step 6- Method of specifying an algorithm

- Natural language
- There are 3 methods:
  - Easy but ambiguous
- Pseudo code
  - Mixture of programming language and natural language
- Flow chart
  - It is pictorial representation of the algorithm.
  - Here symbols are used.

Step 7- Proving an algorithm's correctness

- Correctness: Prove that the algo. Yields the required result for every legitimate input in a finite amount of time.
- A common technique for proving correctness is to use mathematical induction
- In order to show the alg. Is incorrect, we have to take just one instance of its input for which the algm. Fails.
- If a algo. is found to be incorrect, then reconsider one or more those decisions
- For approx. algo. We should show that the error produced by the algo does not exceed the predefined limit.

Step 8- Analyzing an algorithm

- Analyze the following qualities of the algorithm
- Efficiency:
  - Time efficiency and Space efficiency
- Simplicity
  - Simple algorithms mean easy to understand and easy to program.
- Generality
  - Design an algm. for a problem in more general terms. In some cases, designing more general algm. Is unnecessary or difficult.
  - Optimality
    - The algorithm should produce optimal results.

Step 9- Coding an algorithm

- Algorithms are designed to be implemented as computer programs.
- Use code tuning technique. For eg., replacing costlier multiplication operation with cheaper addition operation.

Fundamental data structures

- list
  - array
  - linked list
  - string
- stack
- queue
- priority queue/heap
- **graph**
- **tree and binary tree**
- **set and dictionary**

TYPES OF DATA STRUCTURES

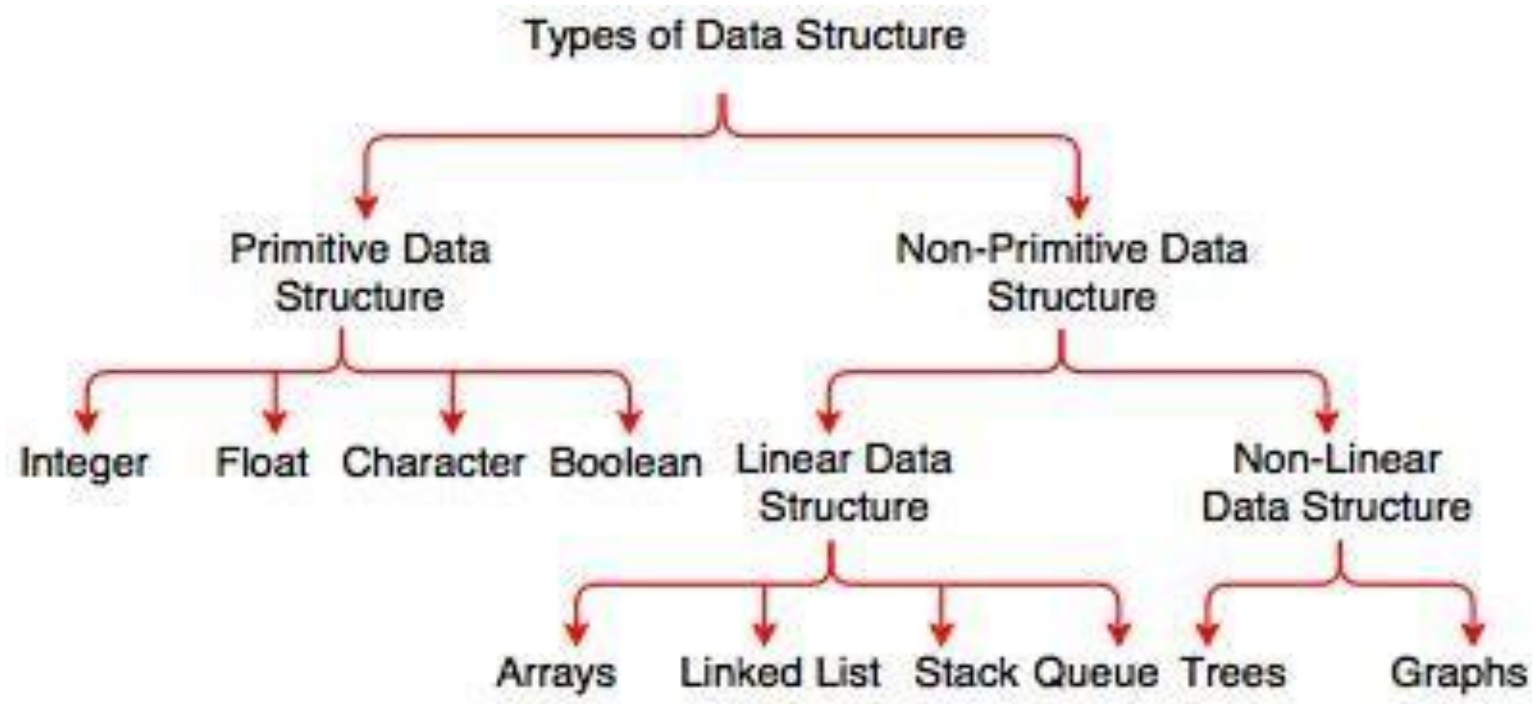


Fig. Types of Data Structure

Linear Data Structures

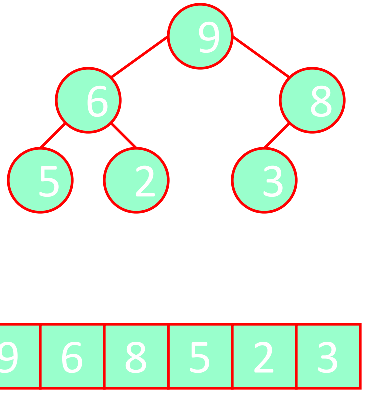
- **Arrays**
  - A sequence of n items of the same data type that are stored contiguously in computer memory and made accessible by specifying a value of the array's index.
- **Linked List**
  - A sequence of zero or more nodes each containing two kinds of information: some data and one or more links called pointers to other nodes of the linked list.
  - Singly linked list (next pointer)
  - Doubly linked list (next + previous pointers)

Stacks and Queues

- Stacks
  - A stack of plates
    - insertion/deletion can be done only at the top.
    - LIFO
  - Two operations (push and pop)
- Queues
  - A queue of customers waiting for services
    - Insertion/enqueue from the rear and deletion/dequeue from the front.
    - FIFO
  - Two operations (enqueue and dequeue)

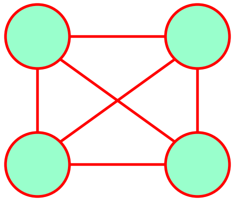
Priority Queue and Heap

- Priority queues (implemented using heaps)
  - A data structure for maintaining a set of elements, each associated with a key/priority, with the following operations
    - Finding the element with the highest priority
    - Deleting the element with the highest priority
    - Inserting a new element
  - Scheduling jobs on a shared computer



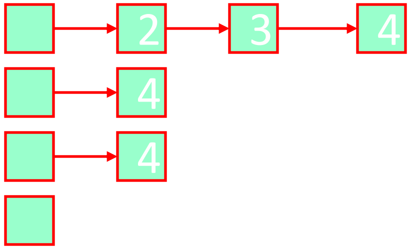
Graphs

- Formal definition
  - A graph $G = \langle V, E\rangle$ is defined by a pair of two sets: a finite set V of items called vertices and a set E of vertex pairs called edges.
- Undirected and directed graphs (digraphs).
- maximum number of edges in an undirected graph with |V| vertices
- Complete graphs
  - A graph with every pair of its vertices connected by an edge is called complete, $K_{|V|}$



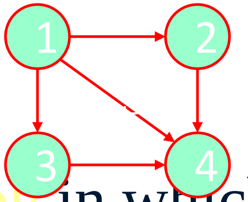
Graph Representation

- Adjacency matrix
  - n x n boolean matrix if |V| is n.
  - The element on the ith row and jth column is 1 if there’s an edge from ith vertex to the jth vertex; otherwise 0.
  - The adjacency matrix of an undirected graph is symmetric.
- Adjacency linked lists
  - A collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list’s vertex.
- Which data structure would you use if the graph is a 100-node star shape?



Weighted Graphs

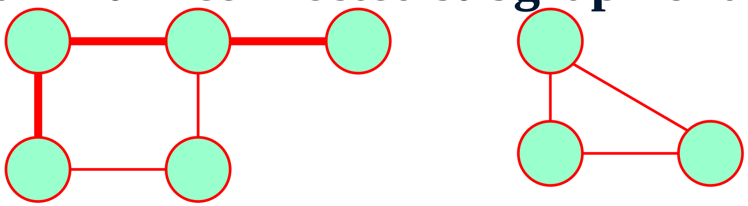
- Weighted graphs
  - Graphs or digraphs with numbers assigned to the edges.



- A dense graph is a graph in which the number of edges is close to the maximal number of edges. The opposite, a graph with only a few edges, is a sparse graph.

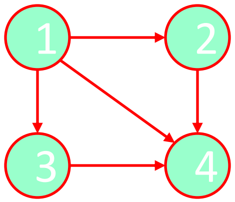
Graph Properties -- Paths and Connectivity

- **Paths**
  - **A path from vertex u to v of a graph G is defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v.**
  - **Simple paths: All edges of a path are distinct.**
  - **Path lengths: the number of edges, or the number of vertices – 1.**
- **Connected graphs**
  - **A graph is said to be connected if for every pair of its vertices u and v there is a path from u to v.**
- **Connected component**
  - **The maximum connected subgraph of a given graph.**



Graph Properties -- Acyclicity

- **Cycle**
  - A simple path of a positive length that starts and ends a the same vertex.
- **Acyclic graph**
  - A graph without cycles
  - DAG (Directed Acyclic Graph)

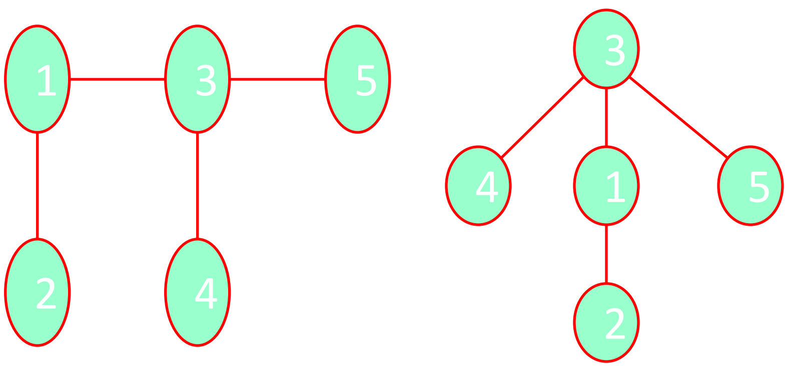


Trees

- Trees
  - A tree (or free tree) is a connected acyclic graph.
  - Forest: a graph that has no cycles but is not necessarily connected.
- Properties of trees

  - For every two vertices in a tree there always exists exactly one simple path from one of these vertices to the other.

    - Rooted trees: The above property makes it possible to select an arbitrary vertex in a free tree and consider it as the root of the so called rooted tree.
    - Levels in a rooted tree.

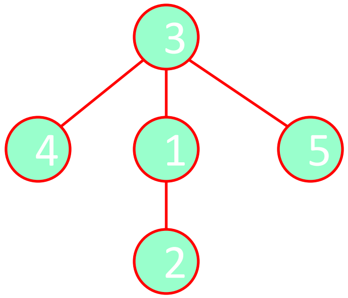


Rooted Trees (I)

- Ancestors
  - For any vertex *v* in a tree *T*, all the vertices on the simple path from the root to that vertex are called ancestors.
- Descendants
  - All the vertices for which a vertex *v* is an ancestor are said to be descendants of *v*.
- Parent, child and siblings
  - If *(u, v)* is the last edge of the simple path from the root to vertex *v*, *u* is said to be the parent of *v* and *v* is called a child of *u*.
  - Vertices that have the same parent are called siblings.
- Leaves
  - A vertex without children is called a leaf.
- Subtree
  - A vertex *v* with all its descendants is called the subtree of *T* rooted at *v*.

Rooted Trees (II)

- Depth of a vertex
  - The length of the simple path from the root to the vertex.
- Height of a tree
  - The length of the longest simple path from the root to a leaf.



Ordered Trees

- Ordered trees
  - An ordered tree is a rooted tree in which all the children of each vertex are ordered.
- Binary trees
  - A binary tree is an ordered tree in which every vertex has no more than two children and each children is designated s either a left child or a right child of its parent.
- Binary search trees
  - Each vertex is assigned a number.
  - A number assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree.
- $\lfloor \log_2 n \rfloor \le h \le n - 1$, where h is the height of a binary tree and n the size.