

ALGORITHMS & DS

Jayaram P
CDAC

Computational Thinking

My Grand Vision for the Field

- Computational thinking will be a fundamental skill used by everyone in the world by the middle of the 21st Century.
 - Just like reading, writing, and arithmetic.
 - Imagine every child knowing how to think like a computer scientist!
 - Incestuous: Computing and computers will enable the spread of computational thinking.

Computational Thinking

- C.T. enables what one human being cannot do alone
 - For solving problems
 - For designing systems
 - For understanding the power and limits of human and machine intelligence

The Two A's of Computational Thinking

- Abstraction
 - C.T. is operating in terms of multiple layers of abstraction simultaneously
 - C.T. is defining the relationships between layers
- Automation
 - C.T. is thinking in terms of mechanizing the abstraction layers and their relationships
 - Mechanization is possible due to precise and exacting notations and models
 - There is some “machine” below (human or computer, virtual or physical)
- They give us the ability and audacity to scale.

Examples of Computational Thinking

- How difficult is this problem and how best can I solve it?
 - Theoretical computer science gives precise meaning to these and related questions and their answers.
- C.T. is thinking recursively.
- C.T. is reformulating a seemingly difficult problem into one which we know how to solve.
 - Reduction, embedding, transformation, simulation
- C.T. is choosing an appropriate representation or modeling the relevant aspects of a problem to make it tractable.
- C.T. is interpreting code as data and data as code.
- C.T. is using abstraction and decomposition in tackling a large complex task.
- C.T. is judging a system's design for its simplicity and elegance.
- C.T. is type checking, as a generalization of dimensional analysis.
- C.T. is prevention, detection, and recovery from worst-case scenarios through redundancy, damage containment, and error correction.
- C.T. is modularizing something in anticipation of multiple users and prefetching and caching in anticipation of future use.
- C.T. is calling gridlock deadlock and avoiding race conditions when synchronizing meetings.
- C.T. is using the difficulty of solving hard AI problems to foil computing agents.
- C.T. is taking an approach to solving problems, designing systems, and understanding human behavior that draws on concepts fundamental to computer science.

Please tell me your favorite examples of computational thinking!

Evidence of Computational Thinking's Influence

- Computational thinking, in particular, machine learning has revolutionized **Statistics**
 - Statistics departments in the US are hiring computer scientists
 - Schools of computer science in the US are starting or embracing existing Statistics departments
- Computational thinking is our current big bet in **Biology**
 - Algorithms and data structures, computational abstractions and methods will inform biology.
- Computational thinking in other disciplines (more examples later)
 - **Game Theory**
 - CT is influencing **Economics**
 - **Nanocomputing**
 - CT is influencing **Chemistry**
 - **Quantum computing**
 - CT is influencing **Physics**

Analogy

The **boldness** of my vision: Computational thinking is not just for other scientists, it's for *everyone*.

- Ubiquitous computing was yesterday's dream, today's reality
- Computational thinking is today's dream, tomorrow's reality

Computational Thinking: What It Is and Is Not

- Conceptualizing, not programming
 - Computer science is not just computer programming
- Fundamental, not rote skill
 - A skill every human being needs to know to function in modern society
 - Rote: mechanical. Need to solve the AI Grand Challenge of making computers “think” like humans. Save that for the second half of this century!
- A way that humans, not computers think
 - Humans are clever and creative
 - Computers are dull and boring

Computational Thinking: What It Is and Is Not

- Complements and combines mathematical and engineering thinking
 - C.T. draws on math as its foundations
 - But we are constrained by the physics of the underlying machine
 - C.T. draws on engineering since our systems interact with the real world
 - But we can build virtual worlds unconstrained by physical reality
- Ideas, not artifacts
 - It's not just the software and hardware that touch our daily lives, it will be the computational concepts we use to approach living.
- It's for everyone, everywhere
 - C.T. will be a reality when it is so integral to human endeavors that it disappears as an explicit philosophy.

Two Messages for the General Public

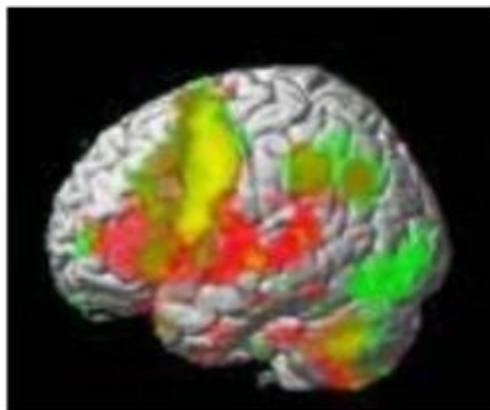
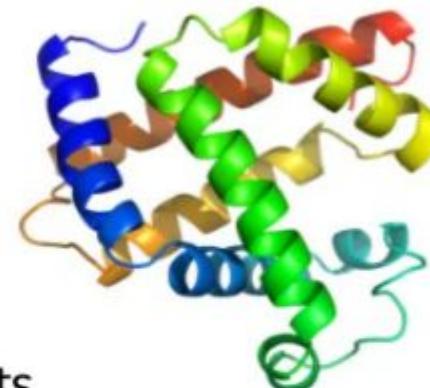
- Intellectually challenging and engaging scientific problems in computer science remain to be understood and solved.
 - Limited only by our curiosity and creativity
- One can major in computer science and do anything.
 - Just like English, political science, or mathematics

Research Implications

CT in Other Sciences, Math, and Engineering

Biology

- Shotgun algorithm expedites sequencing of human genome
- DNA sequences are strings in a language
- Protein structures can be modeled as knots
- Protein kinetics can be modeled as computational processes
- Cells as a self-regulatory system are like electronic circuits



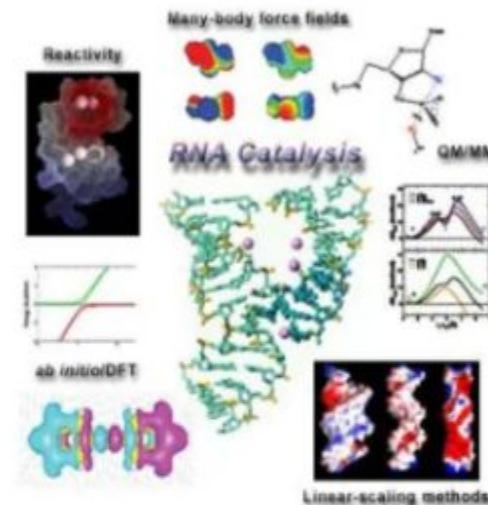
Brain Science

- Modeling the brain as a computer
- Vision as a feedback loop
- Analyzing fMRI data with machine learning

CT in Other Sciences, Math, and Engineering

Chemistry [Madden, Fellow of Royal Society of Edinburgh]

- Atomistic calculations are used to explore chemical phenomena
- Optimization and searching algorithms identify best chemicals for improving reaction conditions to improve yields



[York, Minnesota]



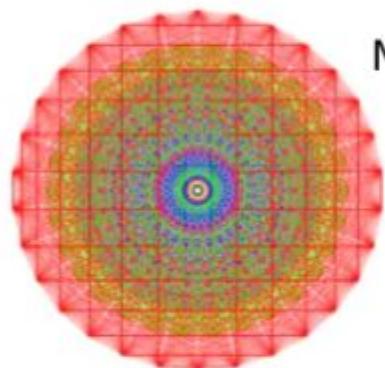
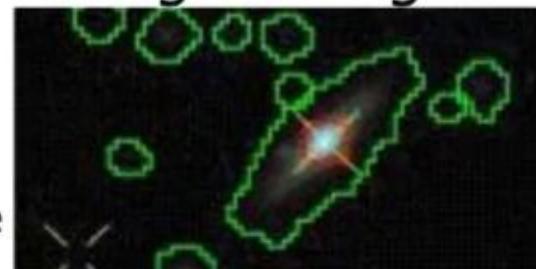
Geology

- "The Earth is an analogue computer."
[Boulton, Edinburgh]
- Abstraction boundaries and hierarchies of complexity model the earth and our atmosphere

CT in Other Sciences, Math, and Engineering

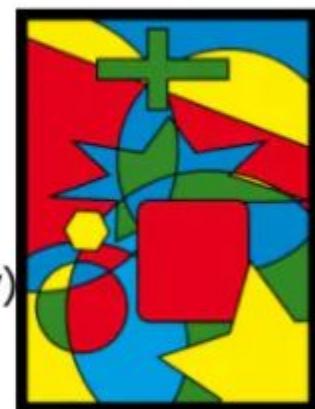
Astronomy

- Sloan Digital Sky Server brings a telescope to every child
- KD-trees help astronomers analyze very large multi-dimensional datasets



Mathematics

- Discovering E8 Lie Group:
18 mathematicians, 4 years and 77 hours of supercomputer time (200 billion numbers).
Profound implications for physics (string theory)
- Four-color theorem proof



Engineering (electrical, civil, mechanical, aero&astro, ...)

- Calculating higher order terms implies more precision, which implies reducing weight, waste, costs in fabrication
- Boeing 777 tested via computer simulation alone, not in a wind tunnel



CT for Society

Economics

- Automated mechanism design underlies electronic commerce, e.g., ad placement, on-line auctions, kidney exchange
- MIT PhDs in CS are quants on Wall Street

Microsoft Digital Advertising Solutions



Social Sciences

- Social networks explain phenomena such as MySpace, YouTube
- Statistical machine learning is used for recommendation and reputation services, e.g., Netflix, affinity card

CT for Society

Medicine

- Robotic surgery
- Electronic health records require privacy technologies
- Scientific visualization enables virtual colonoscopy



Law

- Stanford CL approaches include AI, temporal logic, state machines, process algebras, petri nets
- POIROT Project on fraud investigation is creating a detailed ontology of European law
- Sherlock Project on crime scene investigation

CT for Society

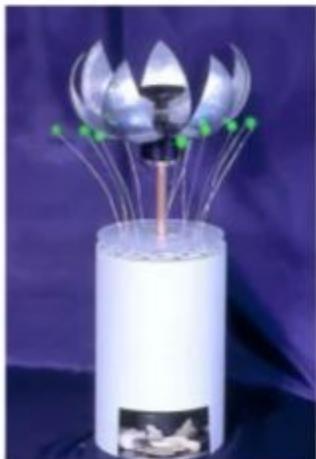
Entertainment



- Games
- Movies
 - Dreamworks uses HP data center to render *Shrek* and *Madagascar*
 - Lucas Films uses 2000-node data center to produce *Pirates of the Caribbean*.



Arts



- Art (e.g., Robotticelli)
- Drama
- Music
- Photography



Sports

- Lance Armstrong's cycling computer tracks man and machine statistics
- Synergy Sports analyzes digital videos NBA games



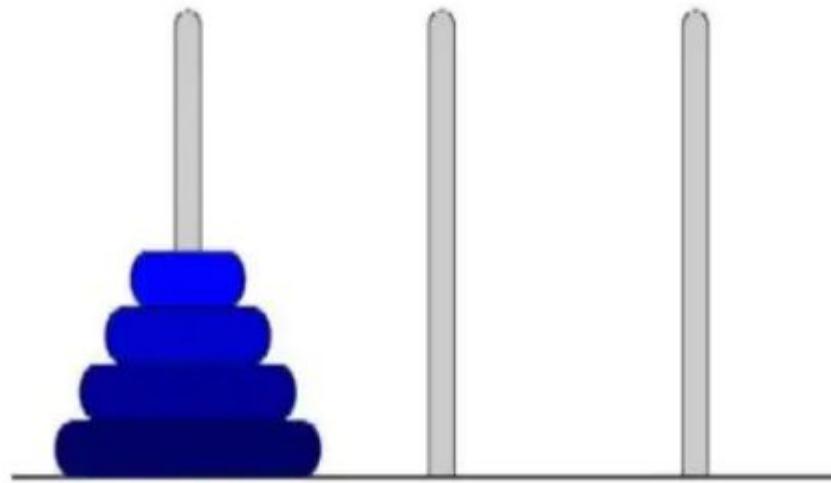
Jeannette M. Wing

Ways To Think Like A Computer Scientist

- Freshmen year course
- Suitable for non-majors, but inspirational for majors
- Lessons
 - Thinking Recursively
 - Thinking Abstractly
 - Thinking Ahead (caching, pre-fetching...)
 - Thinking Procedurally
 - Thinking Logically
 - Thinking Concurrently
 - ...
- Problem sets: pencil-and-paper thought exercises, programming exercises

Recursion: Towers of Hanoi

Goal: Transfer the entire tower to one of the other pegs, moving only one disk at a time and never a larger one onto a smaller.



Data Abstraction and Representation



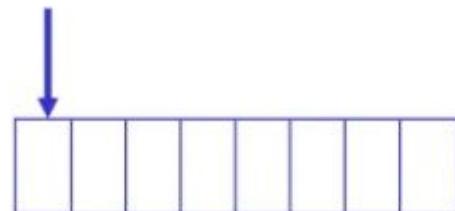
stack



queue



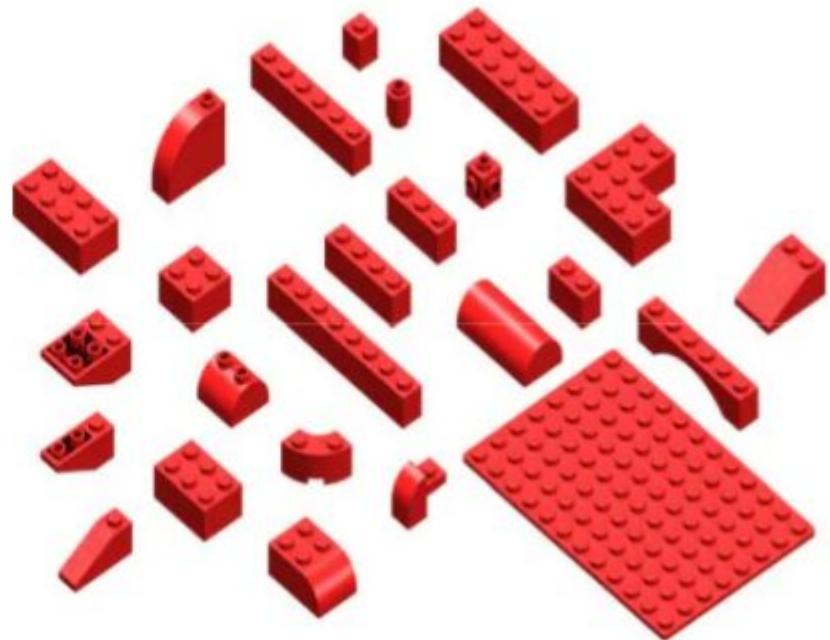
tree
(upside down)



array and pointer

representation invariant

Composition and Decomposition



Sorting and Search



Web Images Video News Maps more »

Advanced Search
Preferences
Language Tools

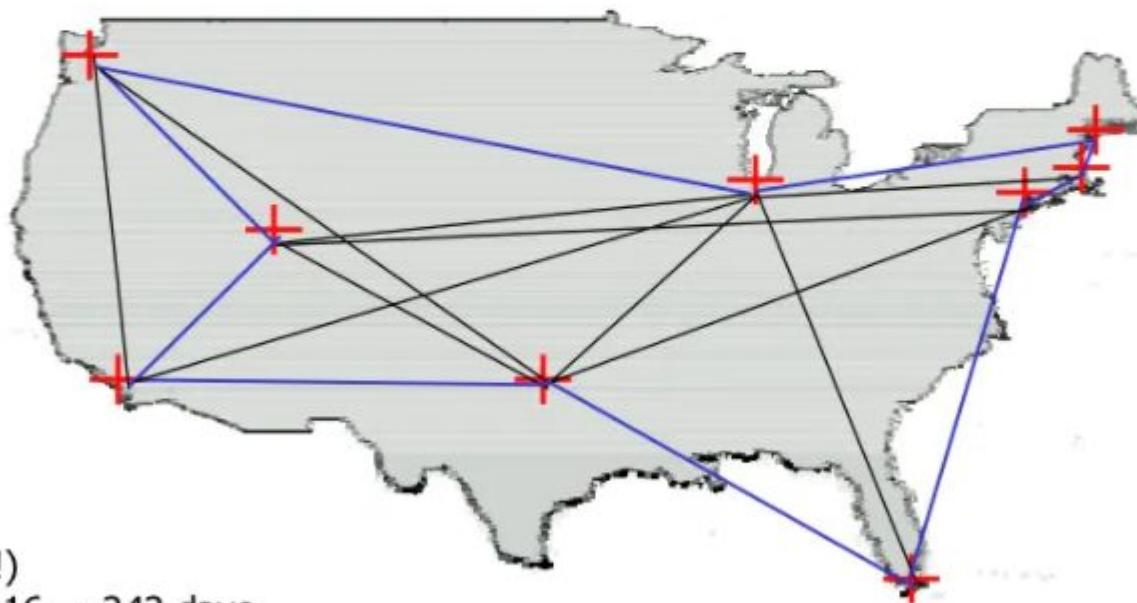
Organize and share holiday pictures with [Google's photo software](#).

[Advertising Programs](#) - [Business Solutions](#) - [About Google](#)

©2006 Google

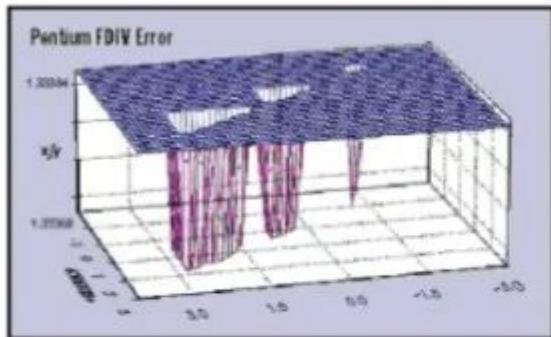
Intractability: Traveling Salesman

Problem: A traveling salesperson needs to visit n cities.
Is there a route of at most d in length?



$O(n!)$
 $n = 16 \rightarrow 242$ days
 $n = 25 \rightarrow 5 \times 10^{15}$ centuries

Correctness: Avoiding Bugs to Save Money and Lives



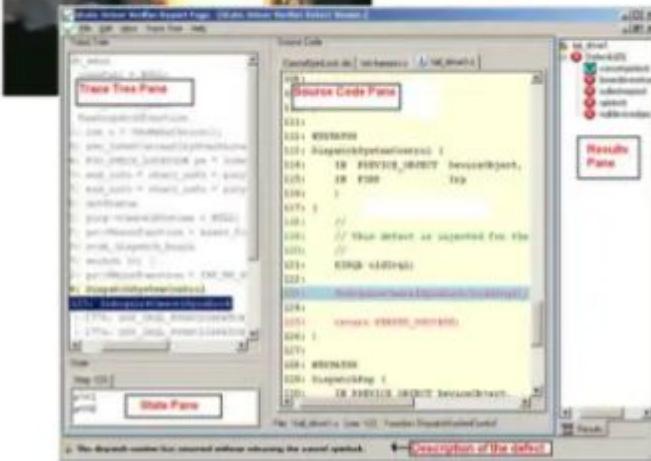
Intel Pentium FPU error



Now Intel uses formal verification.



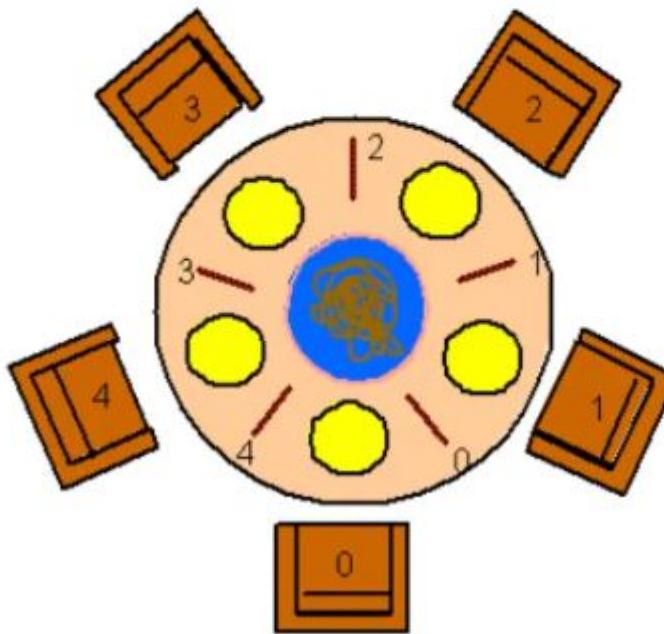
Ariane 5 failure



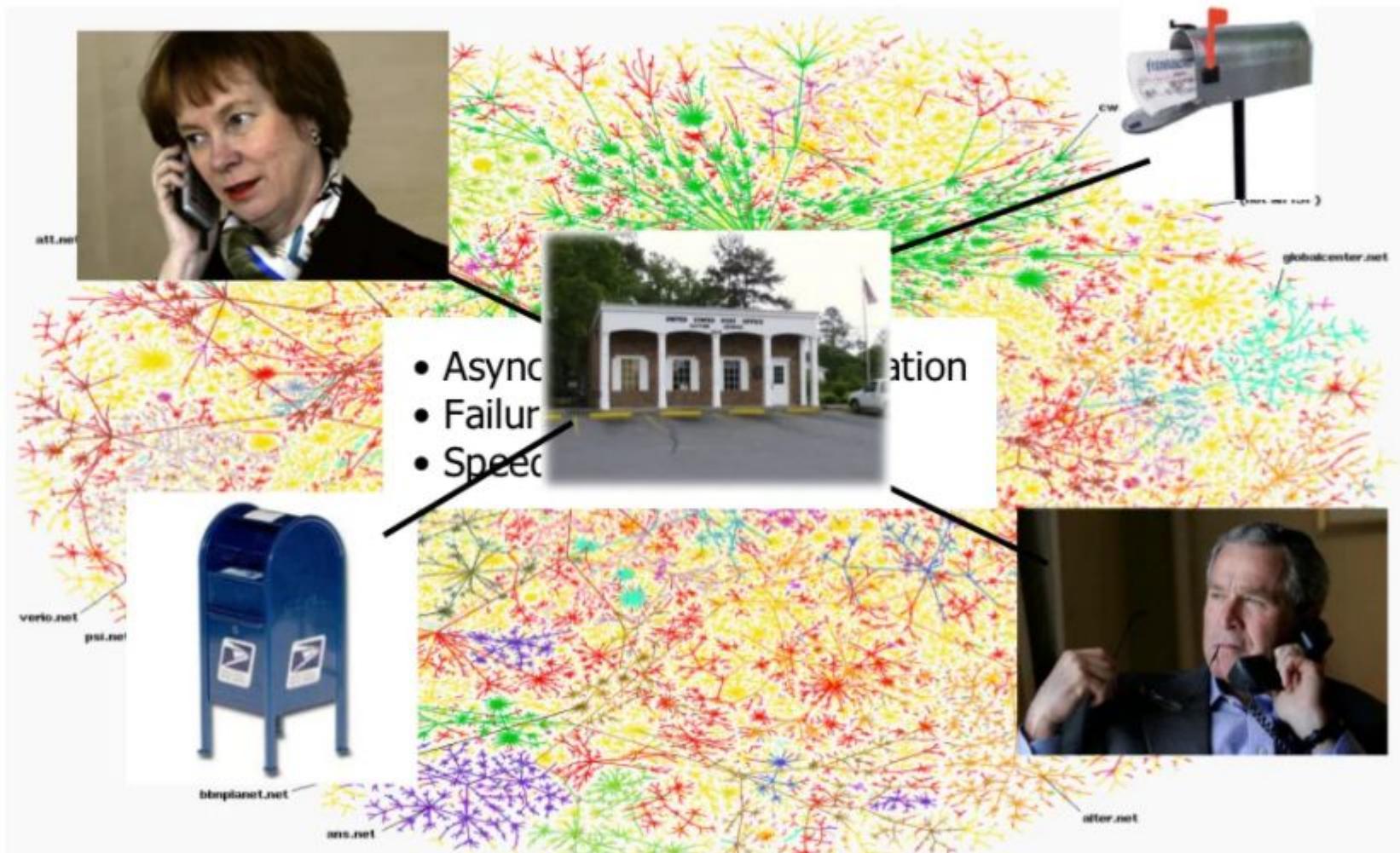
Now Microsoft uses formal verification.

Concurrency: Dining Philosophers

Five philosophers sit around a circular table. Each philosopher spends his life alternately **thinking** and **eating**. In the centre of the table is a large bowl of spaghetti. A philosopher needs two forks to eat a helping of spaghetti.



Distributed Computing: The Internet



Deep Questions in Computer Science

- Does P = NP ?
- What is computable?
- What is intelligence?
- What is system complexity?

What is Computable?

- What are the power and limits of computation?
- What is computable when one considers The Computer as the combination of Human and Machine?

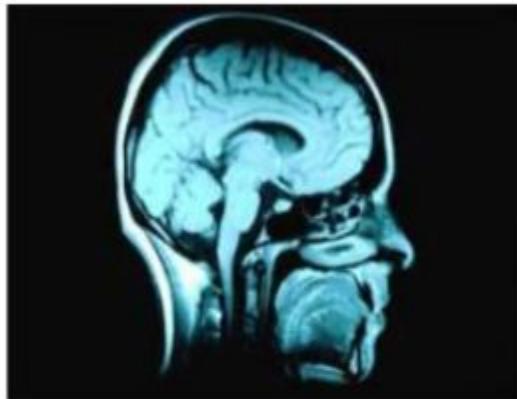


CAPTCHAs

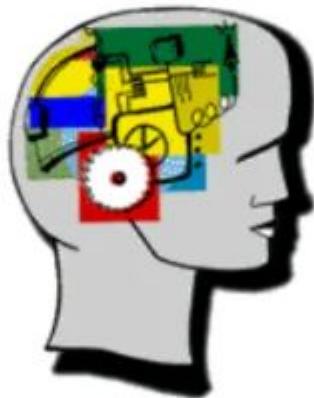
The ESP Game
As seen on CNN and
newspapers around the world!
beta

Labeling Images on the Web

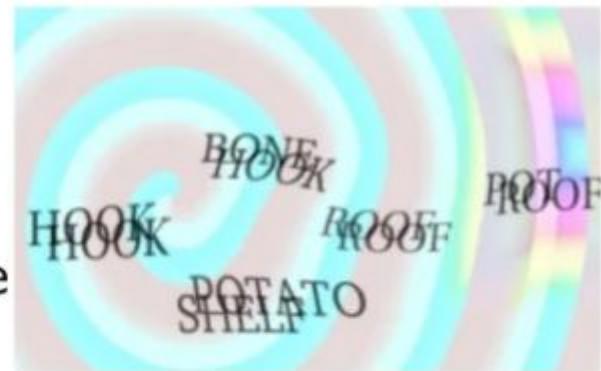
What is Intelligence?



Human and Machine



"Computing Versus Human Thinking," **Peter Naur**,
Turing Award 2005 Lecture,
CACM, January 2007.



invariant representations.
On Intelligence,
by Jeff Hawkins, creator
of PalmPilot and Treo



Human vs. Machine

What is System Complexity?

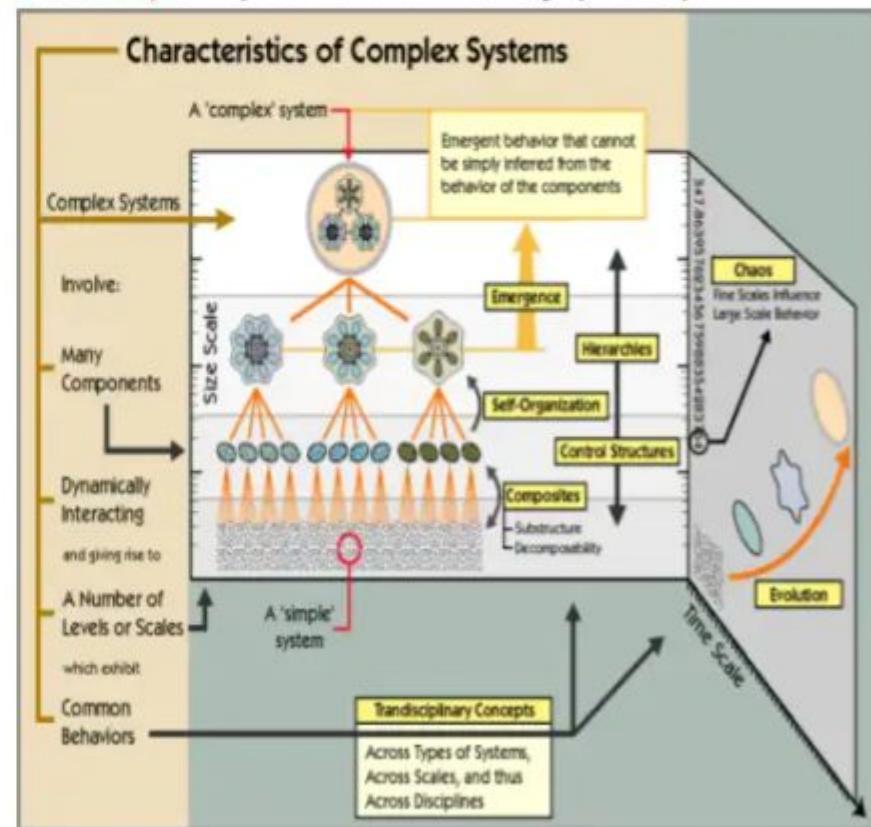
Question 1: Do our systems have to be so complex?

- Can we build **systems with simple designs**, that are easy to understand, modify, and maintain, yet provide the **rich complexity in functionality** of systems that we enjoy today?

Further, observe:

- We have complexity classes from theory.
- We build complex systems that do amazing, but often unpredictable, things.

Question 2: Is there a meaning of **system complexity** that spans the theory and practice of computing?



Algorithms & Data Structures

Datastructures

Data structures is concerned with the representation and manipulation of data.

All programs manipulate data.

So, all programs represent data in some way.

Each of them allows us to perform different operations on data

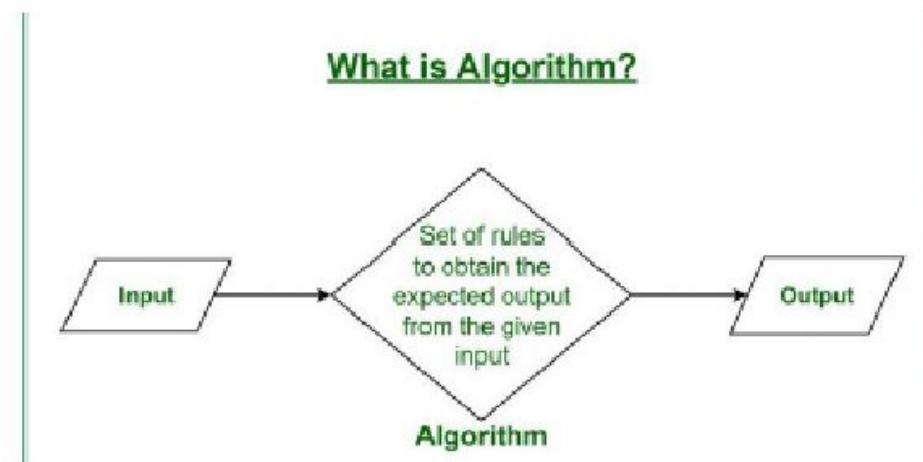
Data manipulation requires an algorithm.

Algorithm design methods needed ,to develop programs that do the data manipulation

Arrays, linked lists, stacks, graphs, trees etc. are all data structures

What is an Algorithm?

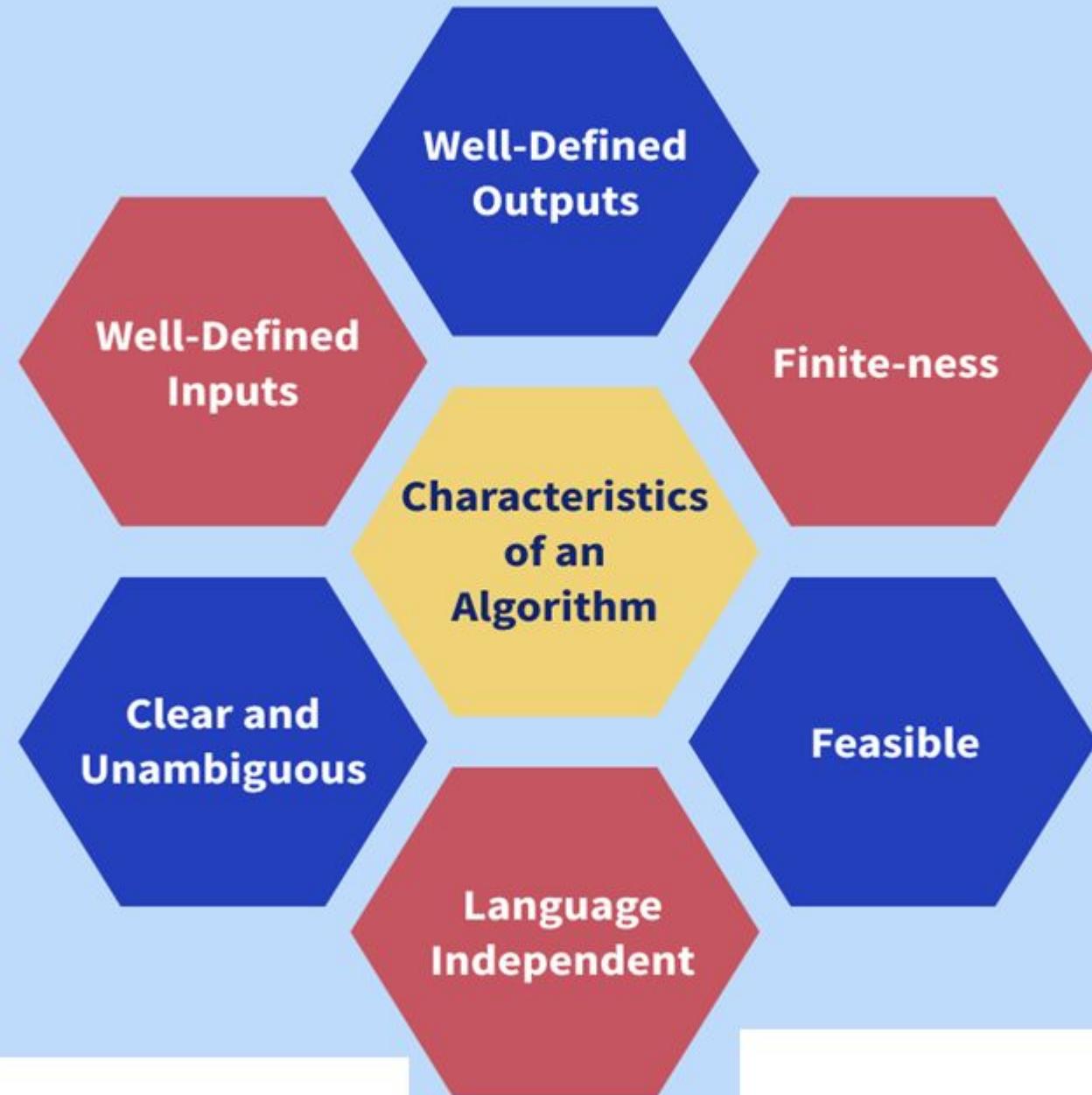
- An algorithm is the step-by-step instructions to a given problem.
- Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.
- The Algorithm designed are language-independent, i.e. they are just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.



The Need For Algorithms (Advantages of Algorithms)

The following are some of the benefits of using algorithms in real-world problems.

- Algorithms boost the effectiveness of an existing method.
- It is easy to compare an algorithm's performance to those of other approaches using various methods (Time Complexity, Space Complexity, etc.).
- Algorithms provide the designers with a detailed description of the criteria and goals of the problems.
- They also enable a reasonable comprehension of the program's flow.
- Algorithms evaluate how well the approaches work in various scenarios (Best cases, worst cases, average cases).
- An algorithm also determines which resources (input/output, memory) cycles are necessary.
- We can quantify and assess the problem's complexity in terms of time and space using an algorithm.
- The cost of design is also reduced if proper algorithms are used.



How to Design an Algorithm?

Inorder to write an algorithm, **following things are needed as a pre-requisite:**

- The **problem** that is to be solved by this algorithm.
- The **constraints** of the problem that must be considered while solving the problem.
- The **input** to be taken to solve the problem.
- The **output** to be expected when the problem is solved.
- The **solution** to this problem, in the given constraints.

Consider the Example :finding length of string

Step 1: Fulfilling the pre-requisites

The problem that is to be solved by this algorithm:

Find the length of a string.

The constraints of the problem that must be considered while solving the problem:

Data should be a string

The input to be taken to solve the problem:

The string or variable storing the length of a string.

The output to be expected when the problem the is solved:

The length of a string,i.E the number of characters in string.

The solution to this problem, in the given constraints

- The solution consists of counting the number of character in the string,or use built in function strlen.
- The data can be stored in array . result can be stored in an integer variable

Step 2: Designing the algorithm

Design the algorithm with the help of previous pre-requisites:

Algorithm to find length of string

- START
- Declare array .to store the string.
- Read string to array.
- Declare an integer variable length to store the resultant length of string
- To find length of string :Use loop variable i to count from str to end of string,
- Store i to length variable
- Print the string length.
- END

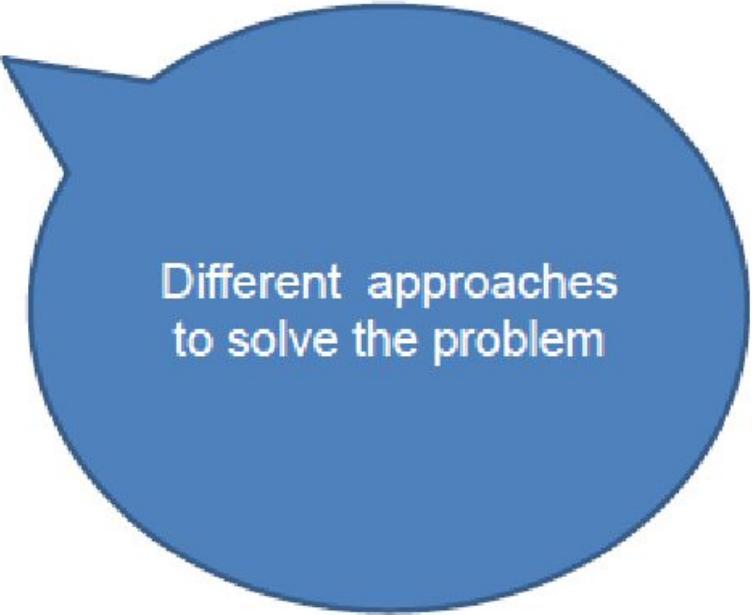
Step 3: Testing the algorithm by implementing it

- Inorder to test the algorithm, we can implement it in any programming language.
- C,C++,Java etc,

Algorithm Design Technique

Algorithm Design Technique

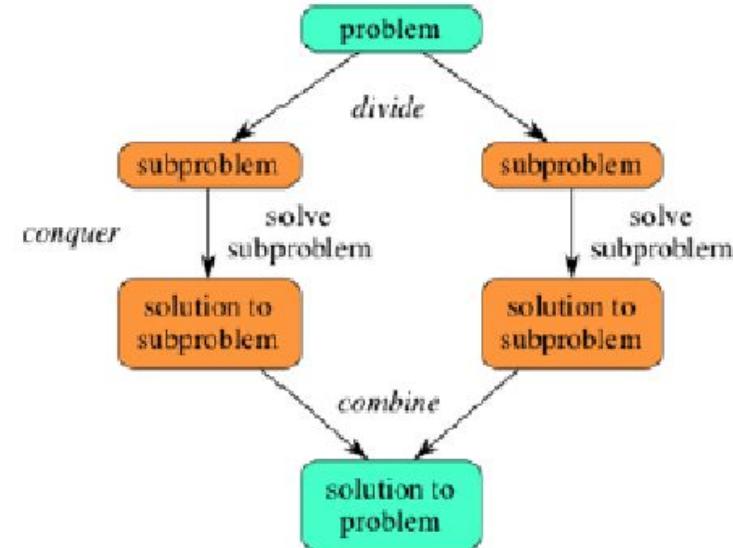
- Divide and conquer
- Brute force
- Backtracking



Different approaches
to solve the problem

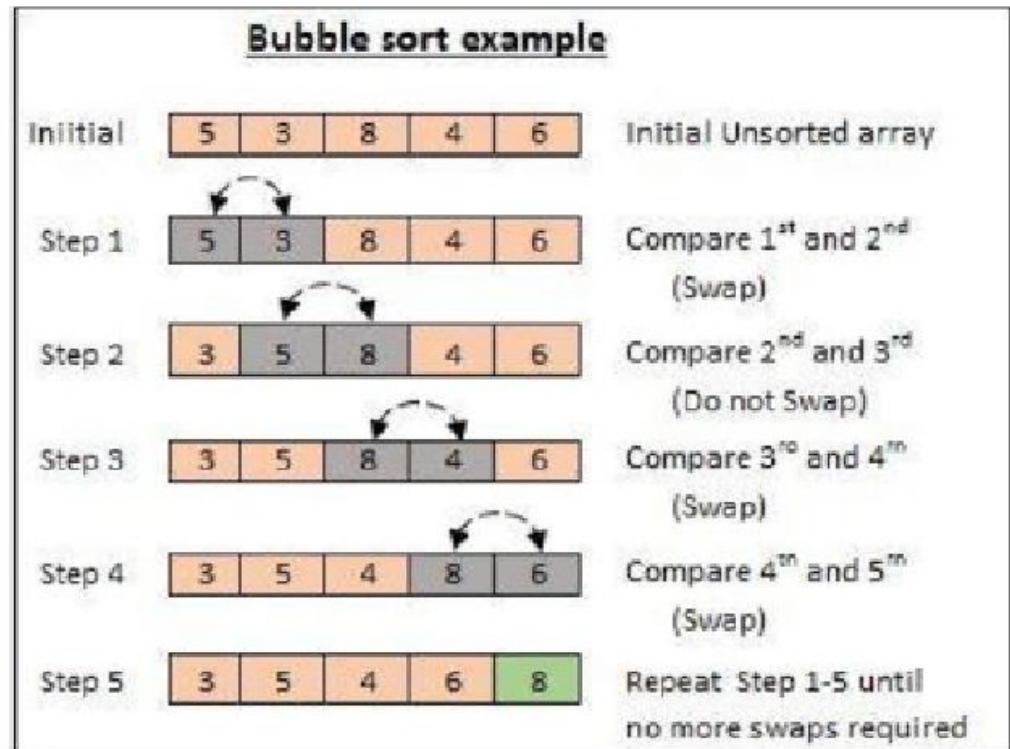
Divide and conquer

- Divide and conquer algorithms decomposes a problem into several smaller independent parts
- Similar to modularization approach to software designs
- solves a problem by
 - Divide :breaking problem into sub problems
 - Recursion:recursively solving these subproblems
 - Conquer combining there answers.
 - Binary search,merge and quick sort



Brute Force

- Brute force is a straightforward approach to solve a problem based on the problem's statement and definitions of the concepts involved. It is considered as one of the easiest approach to apply and is useful for solving small – size instances of a problem
- Eg:bubble sort



Back tracking algorithms

Backtracking is a methodical way of trying out various sequences of decisions, until you find one that “works”

For many real-world problems, the solution process consists of working your way through a sequence of decision points in which each choice leads you further along some path. If you make the correct set of choices, you end up at the solution.

On the other hand, if you reach a dead end or otherwise discover that you have made an incorrect choice somewhere along the way, you have to backtrack to a previous decision point and try a different path. Algorithms that use this approach are called backtracking algorithms.

DFS

- the basic question here is: How to choose the approach? First, by understanding the problem, and second, by knowing various problems and how they are solved using different approaches

Analysis of Algorithms

Analysis of Algorithms

Why Analysis of Algorithms?

- Algorithm analysis helps us determining which of them is efficient in terms of time and space consumed.

If two algorithms, for a task, how do we find out which one is better?

One method:

Implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time.

- There are many problems with this approach for analysis of algorithms.
 - 1) It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
 - 2) It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

```
import java.util.concurrent.TimeUnit;

// Program to measure elapsed time in Java
class Main
{
    public static void main(String[] args) throws InterruptedException
    {
        long startTime = System.currentTimeMillis();

        /* ... The code being measured starts ... */

        ......

        ......

        /* ... The code being measured ends ... */

        long endTime = System.currentTimeMillis();

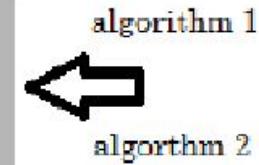
        long timeElapsed = endTime - startTime;

        System.out.println("Execution time in milliseconds: " + timeElapsed);
    }
}
```

Analysis of Algorithms



find length of string



For different inputs and see
which one takes less time

The result

It might be possible that for some inputs, first algorithm performs better than the second.

And for some inputs second performs better.

If two algorithms, for a task, how do we find out which one is better?

One method:

Implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time.

- There are many problems with this approach for analysis of algorithms.
 - 1) It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
 - 2) It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs

Asymptotic Analysis is can be used to overcome these issues in analysing algorithms

Asymptotic Analysis

- Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time).
- We calculate, how does the time (or space) taken by an algorithm increases with the input size.

How asymptotic analysis is done.?

let us consider the search problem (searching a given item) in a sorted array.

Two ways :

Linear search

Binary search

Linear Search (order of growth is linear)

Search (order of growth is logarithmic)

If we run we run the Linear Search



Fast computer

Binary Search



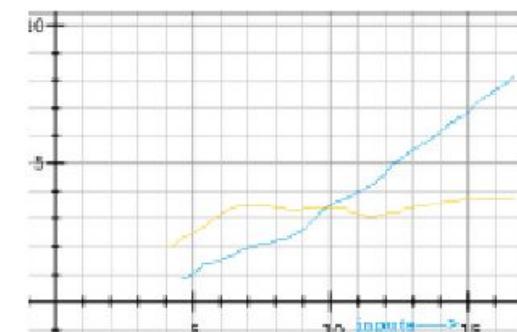
Slow computer

For small values of input array size n

Less time

after certain value of input array size

Less time

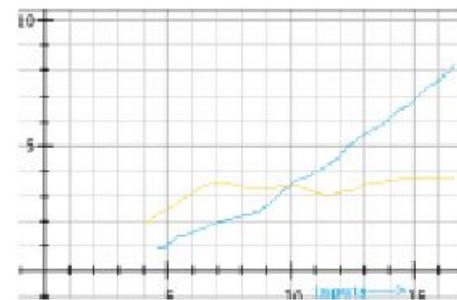


How asymptotic analysis is done.? Cont..

- But, after certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine.
- The reason is the order of growth of Binary Search with respect to input size is logarithmic while the order of growth of Linear Search is linear.
- So the machine dependent constants can always be ignored after certain values of input size

Rate of Growth

The rate at which the running time increases as a function of input is called rate of growth.



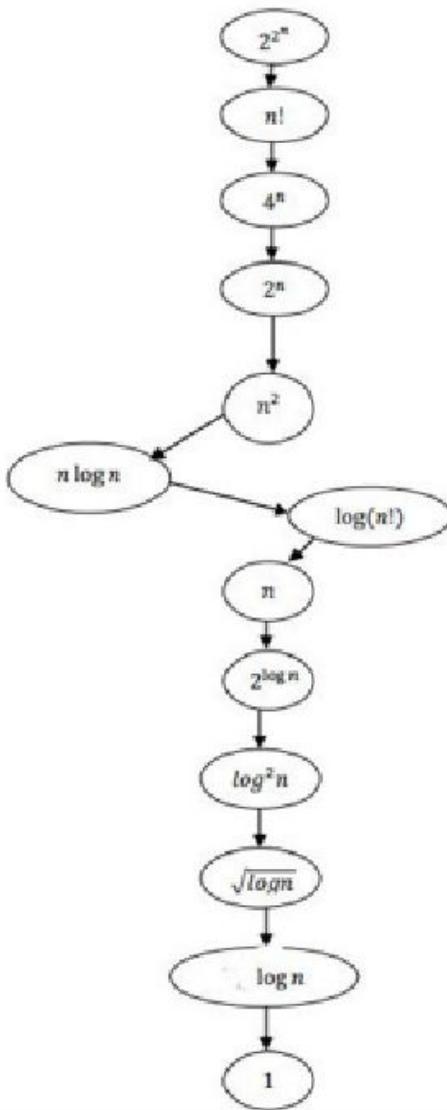
if the run time for an algorithm is expressed as:

$$T(n) = 10n^2 + 2n + 5$$

Then the higher order term n^2 is of significance



Rate of Growth



Decreasing Rates Of Growth

Asymptotic analysis. Types of Analysis

If we have an algorithm for a problem and want to know on what inputs the algorithm is taking less time (performing well) and on what inputs the algorithm is taking huge time.

Worst case

- o Defines the input for which the algorithm takes huge time.
- o Input is the one for which the algorithm runs the slower.

Best case

- o Defines the input for which the algorithm takes lowest time.
- o Input is the one for which the algorithm runs the fastest.

Average case

- o Provides a prediction about the running time of the algorithm
- o Assumes that the input is random

Lower Bound <= Average Time <= Upper Bound

let $f(n)$ be the function which represents the given algorithm then

$f(n)=n^2+500$ **for worst case**

$f(n)=n+100n+500$ **for best case**

Linear search

The worst case : the element is not found or is found at the end of the search

The best case is that the element is in first slot

- Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.
- BIG-O
- Θ Notation
- Ω notation

Asymptotic Notations

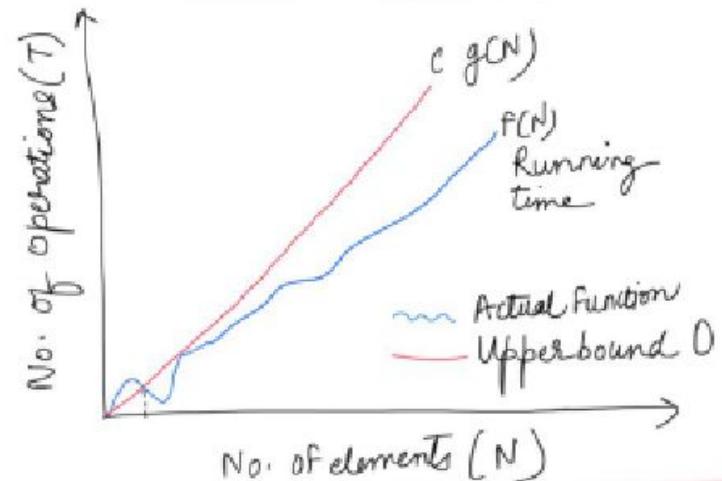
- Considering the performance of algorithm when applied to very large inputs.
- A way of analysing how fast a programs runtime grows as the size of input increases towards infinity.
- $f(x) \in g(x)$ if there exists some value x, x_0 and constant c for which $f(x)$ is less than or equal to that constant times of $g(x)$ for all $x > x_0$

BIG-O

- It's the upper bound on the complexity of an algorithm.
- It is used to denote the worst behavior of an algorithm.
- It's a measure of the longest amount of time it could possibly take for the algorithm to complete.
- $f(n)=O(g(n))$ for larger values of n , the upper bound of $f(n)$ will be $g(n)$.

c is a constant... $f(n)$ is the runtime function for which the upper bound is $g(n)$.

$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 <= f(n) <= c \cdot g(n) \text{ for all } n >= n_0\}$



NOTE: $g(N)$ bounds actual function of running time from above.

EXAMPLE

Binary search algorithm for finding element in already sorted list of elements.

- **Approach** : looks the middle element if value to be searched is $>$ than middle element the right half is searched else the left part of array is searched.
- $O(\log n)$

if input size is 8 : 3 operations : $\log_2 8$

if input size is 16 : 4 operations : $\log_2 16$

- But in the above situation if the value to be found is the one at the middle ?
 - It takes a constant time only

Eg:for binary search

- Best case it takes $\Omega(1)$
- Worst case it takes $O(\log n)$

Ω

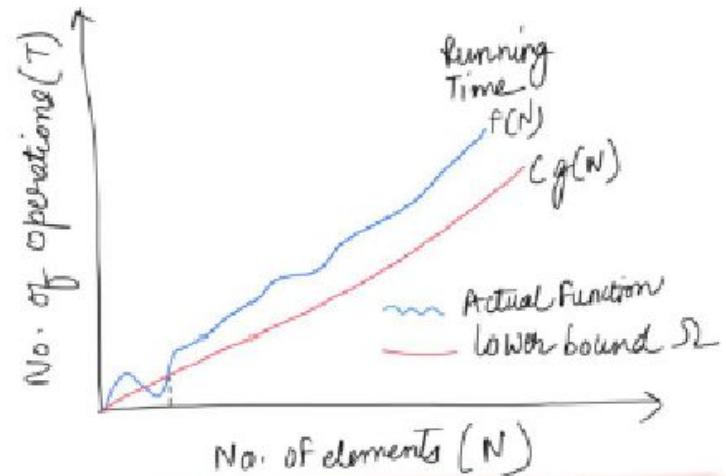
used to represent lower bounds or the best case

For larger values of n , the tight lower bound of $f(n)$ is $g(n)$.

For Linear search

- Best case $\Omega(1)$
- Worst case $O(n)$

$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 <= c*g(n) <= f(n) \text{ for all } n >= n_0\}$.



NOTE: $g(n)$ bounds actual function of running time from below.

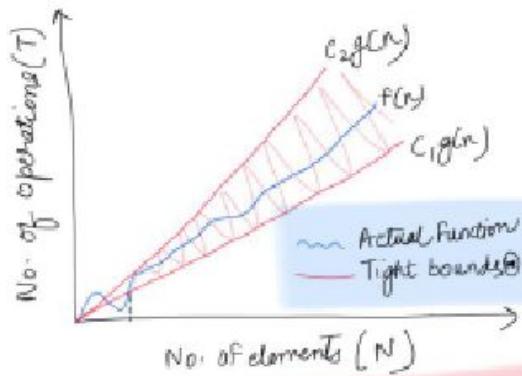
c is a constant. $f(n)$ is the runtime function for which the lower bound is $g(n)$.

Θ

algorithms where the best and worst cases are same. Or average cases

eg: Problem to find String length. By looking at len variable

Best case : $\Omega(1)$ worst case : $O(1)$



NOTE: Actual function is always within the bounds of tight bound function, for large values of N .

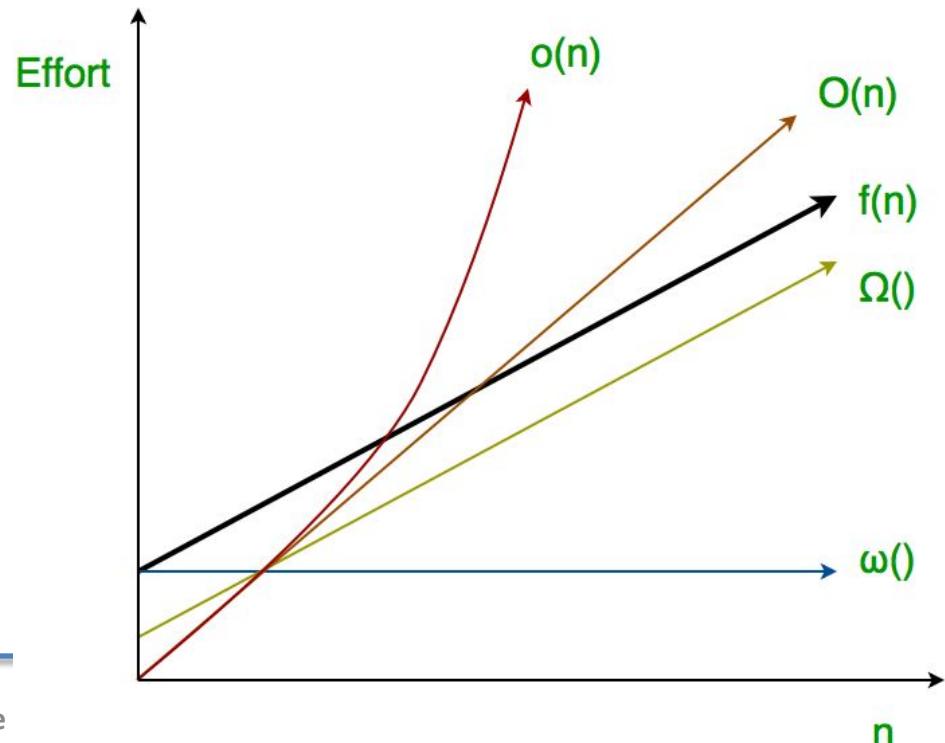
$\Theta(g(n)) = \{f(n): \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 <= c_1*g(n) <= f(n) <= c_2*g(n) \text{ for all } n >= n_0\}$

Little o

- Big-O is used as a tight upper-bound on the growth of an algorithm's effort.
- “Little-o” ($o()$) notation is used to describe an upper-bound that cannot be tight.

Definition : Let $f(n)$ and $g(n)$ be functions that map positive integers to positive real numbers. We say that $f(n)$ is $o(g(n))$ if for **any real** constant $c > 0$, there exists an integer constant $n_0 \geq 1$ such that $0 \leq f(n) < c*g(n)$.

Thus, little $o()$ means **loose upper-bound** of $f(n)$. Little o is a rough estimate of the maximum order of growth whereas Big-O may be the actual order of growth.



Little ω

Definition : $f(n)$ is $\omega(g(n))$ (or $f(n) \in \omega(g(n))$) if for any real constant $c > 0$, there exists an integer constant $n_0 \geq 1$ such that $f(n) > c * g(n) \geq 0$ for every integer $n \geq n_0$.

$f(n)$ has a higher growth rate than $g(n)$ so main difference between Big Omega (Ω) and little omega (ω) lies in their definitions.

In the case of Big Omega $f(n)=\Omega(g(n))$ and the bound is $0 \leq cg(n) \leq f(n)$, but in case of little omega, it is true for $0 \leq c*g(n) < f(n)$.

Little Omega (ω) is a rough estimate of the order of the growth whereas Big Omega (Ω) may represent exact order of growth.

Analysis of Algorithms - Loops

O(1)

- Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion, and call to any other non-constant time function.

// set of non-recursive and non-loop statements

For example,

swap() function has O(1) time complexity.

- A loop or recursion that runs a constant number of times is also considered as O(1).

For example, the following loop is O(1).

// Here c is a constant

```
for (int i = 1; i <= c; i++)  
{  
// some O(1) expressions  
}
```

O(n):

Time Complexity of a loop is considered as O(n) if the loop variables are incremented/decremented by a constant amount.

For example following functions have O(n) time complexity.

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
    // some O(1) expressions
}
```

```
for (int i = n; i > 0; i -= c) {
    // some O(1) expressions
}
```

O(n^c):

Time complexity of nested loops is equal to the number of times the innermost statement is executed.

For example, the following sample loops have **O(n^2)** time complexity.

```
for (int i = 1; i <=n; i += c) {  
    for (int j = 1; j <=n; j += c) {  
        // some O(1) expressions  
    }  
}
```

```
for (int i = n; i > 0; i -= c) {  
    for (int j = i+1; j <=n; j += c) {  
        // some O(1) expressions  
    }
```

O(Logn):

Time Complexity of a loop is considered as O(Logn) if the loop variables are divided/multiplied by a constant amount.

```
for (int i = 1; i <=n; i *= c) {  
    // some O(1) expressions  
}  
for (int i = n; i > 0; i /= c) {  
    // some O(1) expressions  
}
```

O(LogLogn):

Time Complexity of a loop is considered as O(LogLogn) if the loop variables are reduced/increased exponentially by a constant amount.

```
// Here c is a constant greater than 1
for (int i = 2; i <=n; i = pow(i, c)) {
    // some O(1) expressions
}
```

```
//Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 1; i = fun(i)) {
    // some O(1) expressions
}
```

□ How to combine the time complexities of consecutive loops?

When there are consecutive loops, we calculate time complexity as a sum of time complexities of individual loops.

```
for (int i = 1; i <=m; i += c) {  
    // some O(1) expressions  
}  
for (int i = 1; i <=n; i += c) {  
    // some O(1) expressions  
}
```

Time complexity of above code is $O(m) + O(n)$ which is $O(m+n)$
If $m == n$, the time complexity becomes $O(2n)$ which is $O(n)$.

□ How to calculate time complexity when there are many if, else statements inside loops?

As discussed here, worst-case time complexity is the most useful among best, average and worst.

Therefore we need to consider the worst case.

We evaluate the situation when values in if-else conditions cause a maximum number of statements to be executed.

For example,

consider the linear search function where we consider the case when an element is present at the end or not present at all.

When the code is too complex to consider all if-else cases, we can get an upper bound by ignoring if-else and other complex control statements.

Commonly used rates of growth

Time complexity	Name
1	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	Linear Logarithmic
n^2	Quadratic
n^3	Cubic
2^n	Exponential

Finding Time Complexity

Analysis of Algorithms : solving loops

Program statements analysis of iterative programs	Time complexity	
function or set of statements if it doesn't contain loop, recursion and call to any other non-constant time function	$O(1)$	
Loops , if the loop variables is incremented / decremented by a constant amount.	$O(n)$	<pre>for (int i = 1; i <= n; i += c) { // some O(1) expressions }</pre>
Time complexity of nested loops is equal to the number of times the innermost statement is executed	$O(n^2)$	<pre>for (int i = 1; i <=n; i += c) { for (int j = 1; j <=n; j += c) { // some O(1) expressions } }</pre>
if the loop variables is divided / multiplied by a constant amount.	$O(\log n)$	<pre>for (int i = 1; i <=n; i *= c) { // some O(1) expressions } for (int i = n; i > 0; i /= c) { // some O(1) expressions }</pre>

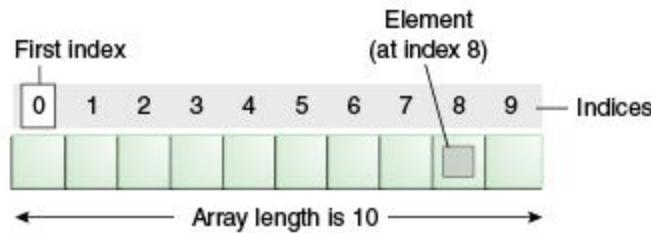
Space Complexity

- How much space or memory will the algorithm take in terms of N , where N is very large.
- Space complexity is a measure of the amount of working storage an algorithm needs.
- That means how much memory, in the worst case, is needed at any point in the algorithm.

Basic Data Structures

□ Arrays :

- Normally, an array is a collection of similar type of elements which has contiguous memory location.
- **Java array** is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location.
- It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.
- Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.
- Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.



Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

□ Single Dimensional Array in Java

Syntax to Declare an Array in Java

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

Instantiation of an Array in Java

1. arrayRefVar=**new** datatype[size];

□ Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

1.int a[]={33,3,4,5};//declaration, instantiation and initialization

Let's see the simple example to print this array.

1.//Java Program to illustrate the use of declaration, instantiation

2.//and initialization of Java array in a single line

3.class Testarray1 {

4.public static void main(String args[]){

5.int a[]={33,3,4,5};//declaration, instantiation and initialization

6.//printing array

7.for(int i=0;i<a.length;i++)//length is the property of array

8.System.out.println(a[i]);

9.}}

□ Passing Array to a Method in Java

```
1.//Java Program to demonstrate the way of passing an array
2.//to method.
3.class Testarray2{
4.//creating a method which receives an array as a parameter
5.static void min(int arr[]){
6.int min=arr[0];
7.for(int i=1;i<arr.length;i++)
8. if(min>arr[i])
9. min=arr[i];
10.
11.System.out.println(min);
12.}
13.
14.public static void main(String args[]){
15.int a[]={33,3,4,5};//declaring and initializing an array
16.min(a);//passing array to method
17.}}
```

□ Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

```
1.//Java Program to demonstrate the way of passing an anonymous array  
2.//to method.  
3.public class TestAnonymousArray{  
4.//creating a method which receives an array as a parameter  
5.static void printArray(int arr[]){  
6.for(int i=0;i<arr.length;i++)  
7.System.out.println(arr[i]);  
8.}  
9.  
10.public static void main(String args[]){  
11.printArray(new int[]{10,22,44,66});//passing anonymous array to method  
12.}}
```

□ **ArrayIndexOutOfBoundsException**

The Java Virtual Machine (JVM) throws an `ArrayIndexOutOfBoundsException` if length of the array is negative, equal to the array size or greater than the array size while traversing the array.

1.//Java Program to demonstrate the case of
2.//`ArrayIndexOutOfBoundsException` in a Java Array.
3.public class TestArrayException{
4.public static void main(String args[]){
5.int arr[]={**50,60,70,80**};
6.for(int i=0;i<=arr.length;i++){
7.System.out.println(arr[i]);
8.**}**
9.**}****}**

□ Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. **dataType []arrayRefVar[];**

Example to instantiate Multidimensional Array in Java

1. **int[][] arr=new int[3][3];//3 row and 3 column**

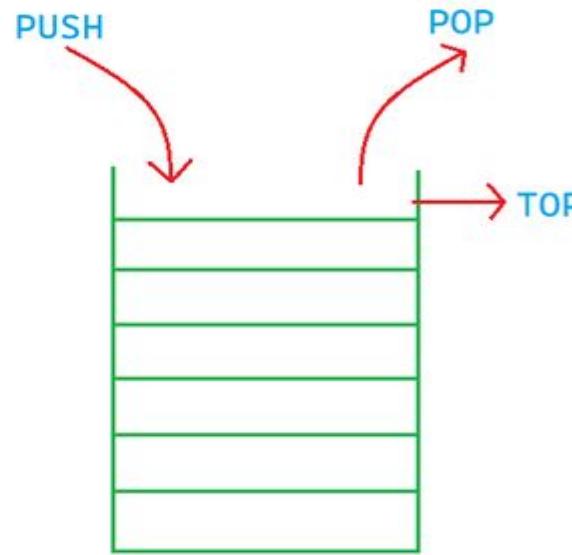
Example to initialize Multidimensional Array in Java

1. arr[0][0]=1;
 2. arr[0][1]=2;
 3. arr[0][2]=3;
 4. arr[1][0]=4;
-

Stack Data Structure

- Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last O

Stack
Insertion & Deletion
happens on the same end.



There are many real-life examples of a stack. Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time.

□ Operations:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **POP:** items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **p:** Removes an item from the stack.
- **Peek or Top:** Returns the top element of the stack.
- **isEmpty:** Returns true if the stack is empty, else false.

□ Time Complexities of operations on stack:

push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

□ Applications of stack:

- Balancing of symbols
- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.
- String reversal is also another application of stack.

□ Implementation:

There are two ways to implement a stack:

- Implementing Stack using Arrays**

```
/* Java program to implement basic stack
operations */
class Stack {
    static final int MAX = 1000;
    int top;
    int a[] = new int[MAX]; // Maximum size
of Stack

    boolean isEmpty()
    {
        return (top < 0);
    }
    Stack()
    {
        top = -1;
    }
```

```
boolean push(int x)
{
    if (top >= (MAX - 1)) {
        System.out.println("Stack
Overflow");
        return false;
    }
    else {
        a[++top] = x;
        System.out.println(x + " pushed into
stack");
        return true;
    }
}
```

```
int pop()
{
    if (top < 0) {
        System.out.println("Stack
Underflow");
        return 0;
    }
    else {
        int x = a[top--];
        return x;
    }
}
```

```
int peek()
{
    if (top < 0) {
        System.out.println("Stack
Underflow");
        return 0;
    }
    else {
        int x = a[top];
        return x;
    }
}

void print(){
    for(int i = top;i>-1;i--){
        System.out.print(" "+ a[i]);
    }
}
```

```
// Driver code
class Main {
    public static void main(String args[])
    {
        Stack s = new Stack();
        s.push(10);
        s.push(20);
        s.push(30);
        System.out.println(s.pop() + " Popped from stack");
        System.out.println("Top element is :" + s.peek());
        System.out.print("Elements present in stack :");
        s.print();
    }
}
```

Stack Class in Java

The class supports one *default constructor* **Stack()** which is used to *create an empty stack*.

Declaration:

```
public class Stack<E> extends Vector<E>
```

How to Create a Stack?

In order to create a stack, we must import **java.util.stack** package and use the **Stack()** constructor of this class. The below example creates an empty Stack.

```
Stack<E> stack = new Stack<E>();
```

Evaluation of Postfix Expression using Stack

- Scan the expression from left to right.
- If we encounter any operand in the expression, then we push the operand in the stack.
- When we encounter any operator in the expression, then we pop the corresponding operands from the stack.
- When we finish with the scanning of the expression, the final value remains in the stack.

Evaluation of Postfix Expression using Stack

Example 1: Postfix expression: 2 3 4 * +

Input	Stack	
2 3 4 * +	empty	Push 2
3 4 * +	2	Push 3
4 * +	3 2	Push 4
* +	4 3 2	Pop 4 and 3, and perform $4*3 = 12$. Push 12 into the stack.
+	12 2	Pop 12 and 2 from the stack, and perform $12+2 = 14$. Push 14 into the stack.

Evaluation of Postfix Expression using Stack

Example 2: Postfix expression: 3 4 * 2 5 * +

Input	Stack	
3 4 * 2 5 * +	empty	Push 3
4 * 2 5 * +	3	Push 4
* 2 5 * +	4 3	Pop 3 and 4 from the stack and perform $3*4 = 12$. Push 12 into the stack.
2 5 * +	12	Push 2
5 * +	2 12	Push 5
* +	5 2 12	Pop 5 and 2 from the stack and perform $5*2 = 10$. Push 10 into the stack.
+	10 12	Pop 10 and 12 from the stack and perform $10+12 = 22$. Push 22 into the stack.



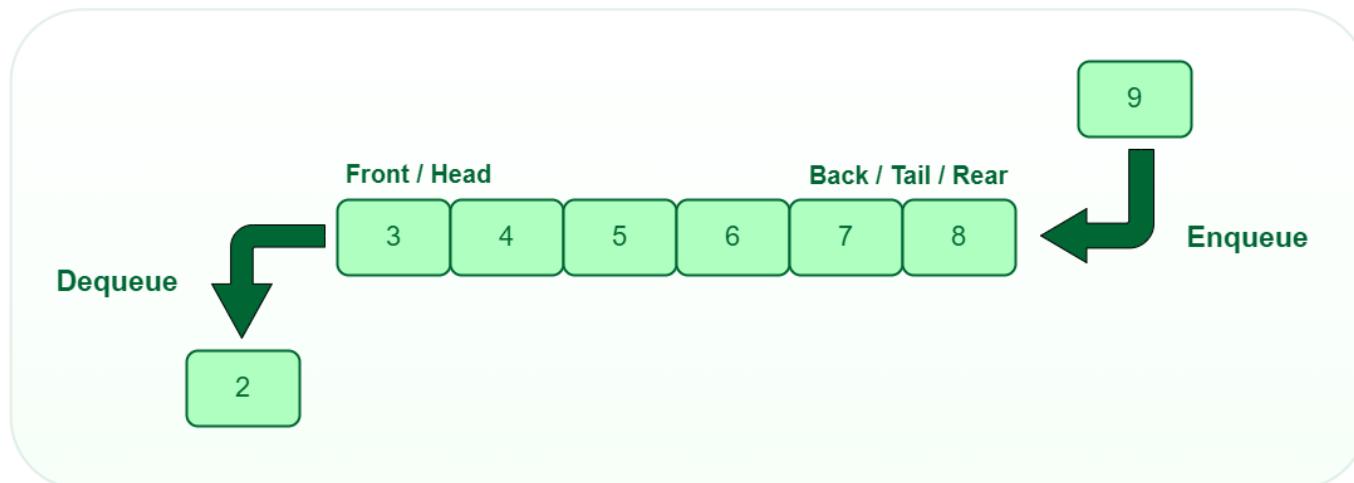
THANK YOU

[jayaram@cda
c.in](mailto:jayaram@cda.in)

Queue Data Structure

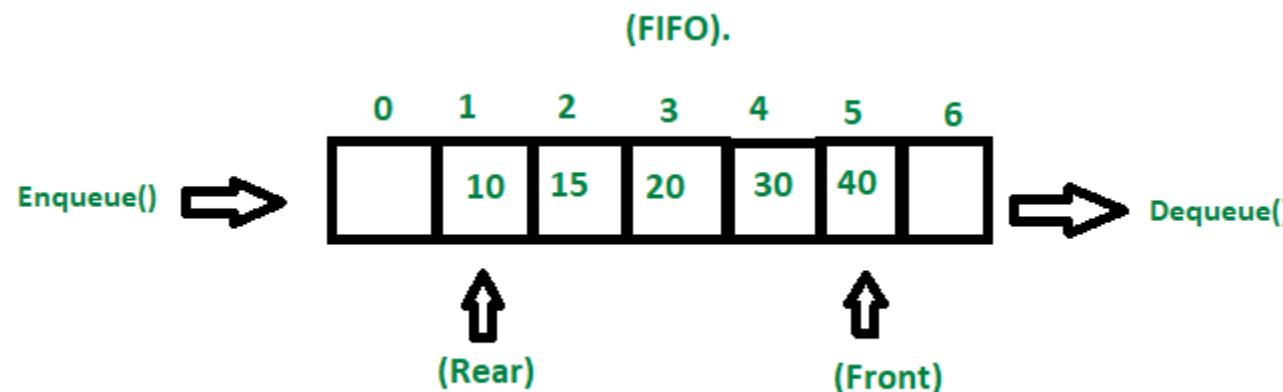
A queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.

We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. The element which is first pushed into the order, the operation is first performed on that.



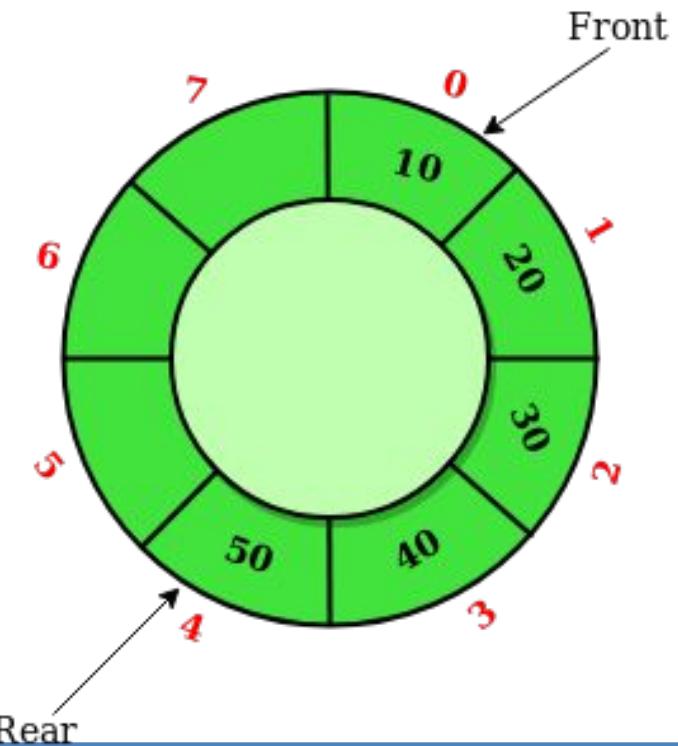
FIFO Principle of Queue:

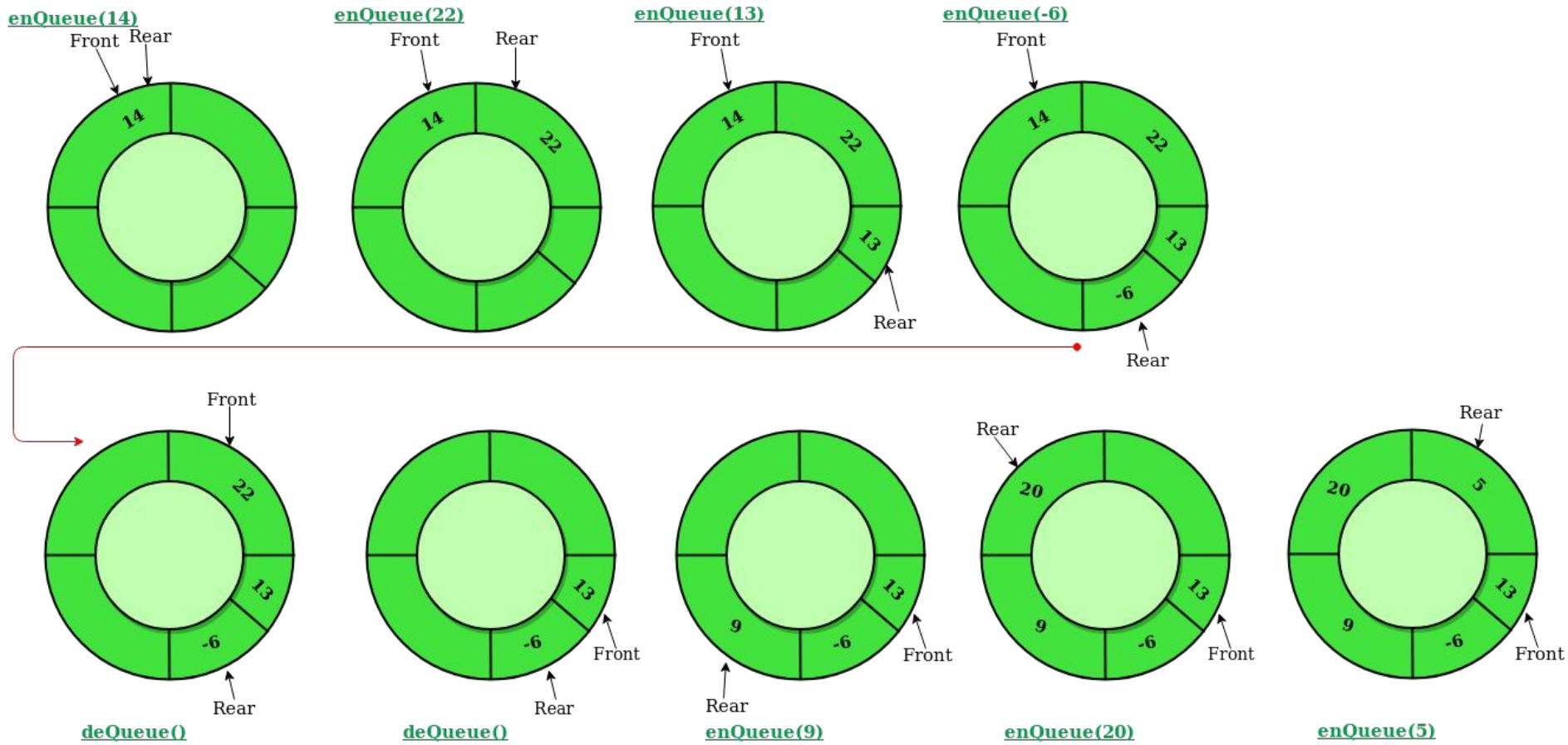
- A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First come first serve).
- Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the **front** of the queue(sometimes, **head** of the queue), similarly, the position of the last entry in the queue, that is, the one most recently added, is called the **rear** (or the **tail**) of the queue. See the below figure.



Circular Queue

- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.
- It is also called '**Ring Buffer**'.





□ Operations on Circular Queue:

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.
 - Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
 - If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.

□ Operations on Circular Queue:

- **deQueue()**

This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.

- Check whether queue is Empty means check ($\text{front} == -1$).
- If it is empty then display Queue is empty.
- Check if ($\text{front} == \text{rear}$) if it is true then set $\text{front} = \text{rear} = -1$ else check if ($\text{front} == \text{size} - 1$), if it is true then set $\text{front} = 0$ and return the element.

```
// Java program for insertion and
// deletion in Circular Queue
import java.util.ArrayList;

class CircularQueue{

    // Declaring the class variables.
    private int size, front, rear;

    // Declaring array list of integer type.
    private ArrayList<Integer> queue = new
    ArrayList<Integer>();

    // Constructor
    CircularQueue(int size)
    {
        this.size = size;
        this.front = this.rear = -1;
    }

    // Function to insert element
    void insert(int value)
    {
        if ((front == -1) && (rear == -1))
            queue.add(value);
        else if ((front + 1) % size == rear)
            System.out.println("Circular Queue is full");
        else
            queue.add(front + 1, value);
    }

    // Function to delete element
    void delete()
    {
        if ((front == -1) && (rear == -1))
            System.out.println("Circular Queue is empty");
        else if (front == rear)
            queue.remove(front);
        else
            queue.remove(front + 1);
    }

    // Function to display elements
    void display()
    {
        if ((front == -1) && (rear == -1))
            System.out.println("Circular Queue is empty");
        else
            for (int i = front + 1; i <= rear; i++)
                System.out.print(queue.get(i) + " ");
    }
}
```

```
// Method to insert a new element in the
queue.
public void enQueue(int data)
{
    // Condition if queue is full.
    if((front == 0 && rear == size - 1) ||
       (rear == (front - 1) % (size - 1)))
    {
        System.out.print("Queue is Full");
    }

    // condition for empty queue.
    else if(front == -1)
    {
        front = 0;
        rear = 0;
        queue.add(rear, data);
    }
}
```

```
else if(rear == size - 1 && front != 0)
{
    rear = 0;
    queue.set(rear, data);
}

else
{
    rear = (rear + 1);

    // Adding a new element if
    if(front <= rear)
    {
        queue.set(rear, data);
    }

    // Else updating old value
    else
    {
        queue.set(rear, data);
    }
}
```

```
// Function to dequeue an element  
// from the queue.  
public int deQueue()  
{  
    int temp;  
  
    // Condition for empty queue.  
    if(front == -1)  
    {  
        System.out.print("Queue is Empty");  
  
        // Return -1 in case of empty queue  
        return -1;  
    }  
  
    temp = queue.get(front);
```

```
// Condition for only one element  
if(front == rear)  
{  
    front = -1;  
    rear = -1;  
}  
  
else if(front == size - 1)  
{  
    front = 0;  
}  
else  
{  
    front = front + 1;  
}  
  
// Returns the dequeued element  
return temp;
```

```
// Method to display the elements of queue  
public void displayQueue()  
{
```

```
// Condition for empty queue.  
if(front == -1)  
{  
    System.out.print("Queue is Empty");  
    return;  
}
```

```
// If rear has not crossed the max size  
// or queue rear is still greater than  
// front.  
System.out.print("Elements in the " +  
    "circular queue are: ");
```

```
if(rear >= front)  
{  
  
    // Loop to print elements from  
    // front to rear.  
    for(int i = front; i <= rear; i++)  
    {  
        System.out.print(queue.get(i));  
        System.out.print(" ");  
    }  
    System.out.println();
```

```
// If rear crossed the max index and  
// indexing has started in loop  
else  
{  
  
    // Loop for printing elements from  
    // front to max size or last index  
    for(int i = front; i < size; i++)  
    {  
        System.out.print(queue.get(i));  
        System.out.print(" ");  
    }  
  
    // Loop for printing elements from  
    // 0th index till rear position  
    for(int i = 0; i <= rear; i++)  
    {  
        System.out.print(queue.get(i));  
        System.out.print(" ");  
    }  
    System.out.println();  
}
```

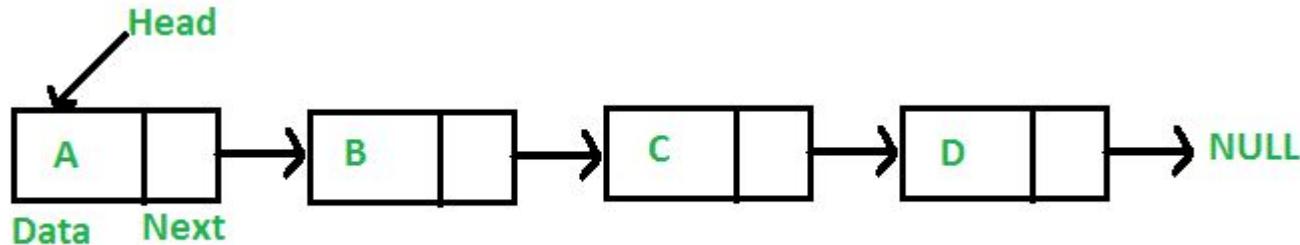
Time Complexity: Time complexity of enQueue(), deQueue() operation is $O(1)$ as there is no loop in any of the operation.

Applications:

- 1. Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
- 2. CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

Linked List

- Like arrays, Linked List is a linear data structure.
- Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.



Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have the following limitations.

- 1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
- 2) Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

For example, in a system, if we maintain a sorted list of IDs in an array id[].
id[] = [1000, 1010, 1050, 2000, 2040].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

Advantages over arrays

- 1) Dynamic size
- 2) Ease of insertion/deletion

Drawbacks:

- 1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.
- 2) Extra memory space for a pointer is required with each element of the list.
- 3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation:

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. **If the linked list is empty, then the value of the head is NULL.**

Each node in a list consists of at least two parts:

- 1) data
- 2) Pointer (Or Reference) to the next node

```
class LinkedList {  
    Node head;  
  
    class Node {  
        int data;  
        Node next;  
  
        Node(int d) { data = d; }  
    }  
}
```

```

class LinkedList {
    Node head; // head of list

    /* Linked list Node. This inner class is made
    static so that
    main() can access it */

    static class Node {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        } // Constructor
    }
}

```

/* method to create a simple linked list with 3 nodes*/

```

public static void main(String[] args)
{
    /* Start with the empty list. */
    LinkedList llist = new LinkedList();

    llist.head = new Node(1);
    Node second = new Node(2);
    Node third = new Node(3);

    /* Three nodes have been allocated
    dynamically.

    We have references to these three
    blocks as head,
    second and third

```

llist.head	second	third
-----+-----+-----+	-----+-----+-----+	-----+-----+-----+
1 null	2 null	3 null
-----+-----+-----+	-----+-----+-----+	-----+-----+-----+

*/

```
llist.head.next = second; // Link first node  
with the second node
```

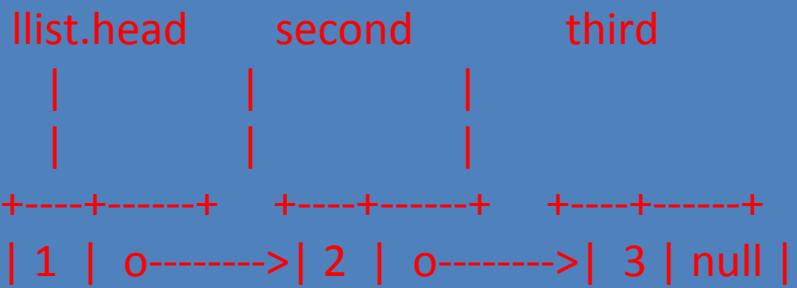
/* Now next of the first Node refers
to the second. So they
both are linked.

llist.head	second	third
+-----+ +-----+ +-----+	+-----+ +-----+ +-----+	+-----+ +-----+ +-----+
1 o-----> 2 null 3 null		
+-----+ +-----+ +-----+	+-----+ +-----+ +-----+	+-----+ +-----+ +-----+

*/

```
second.next = third; // Link second node  
with the third node
```

/* Now next of the second Node refers
to third. So all three
nodes are linked.



*/
}

Linked List Traversal

```
/* This function prints contents of linked list starting
from head */
public void printList()
{
    Node n = head;
    while (n != null) {
        System.out.print(n.data + " ");
        n = n.next;
    }
}
```

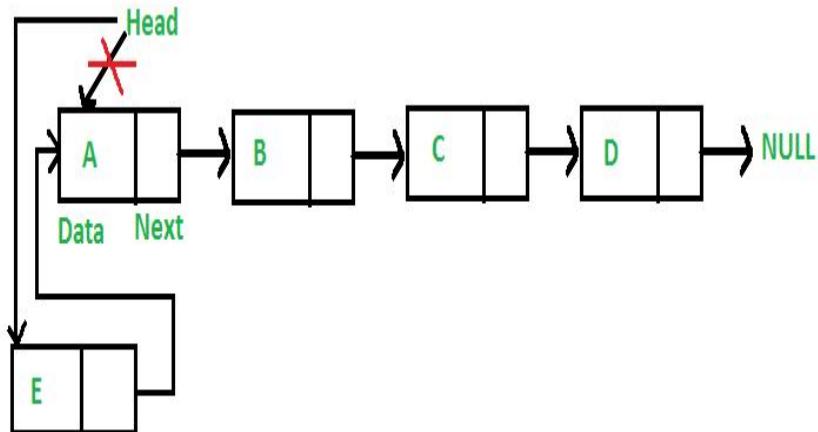
Linked List (Inserting a node)

A node can be added in three ways □

- 1) At the front of the linked list
- 2) After a given node.
- 3) At the end of the linked list.

Add a node at the front: (4 steps process)

- The new node is always added before the head of the given Linked List.
- And newly added node becomes the new head of the Linked List.
- For example, if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25.
- Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node



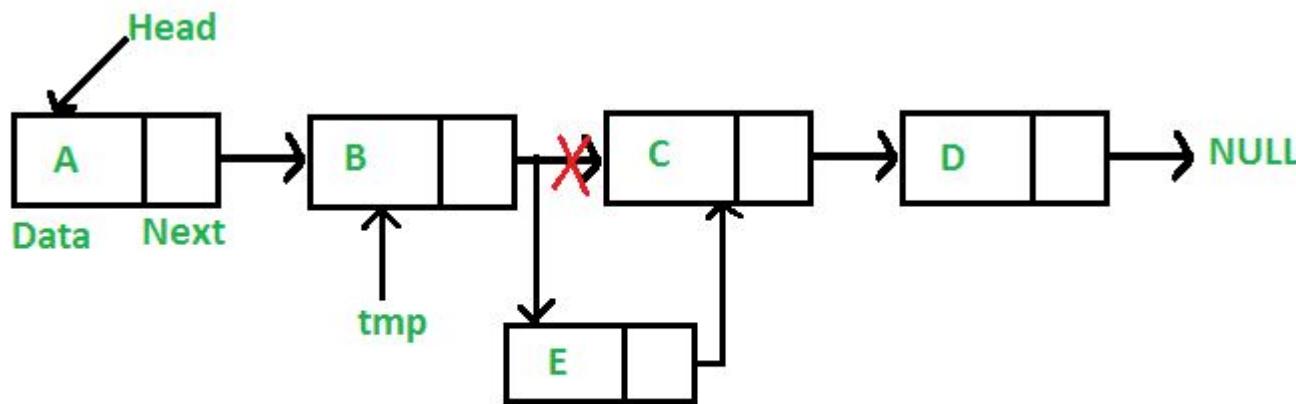
```
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}
```

Time complexity of push() is O(1) as it does a constant amount of work.

□ Add a node after a given node: (5 steps process)



```
public void insertAfter(Node prev_node, int new_data)
{
    /* 1. Check if the given Node is null */
    if (prev_node == null)
    {
        System.out.println("The given previous node cannot be null");
        return;
    }

    /* 2. Allocate the Node &
       3. Put in the data*/
    Node new_node = new Node(new_data);

    /* 4. Make next of new Node as next of prev_node */
    new_node.next = prev_node.next;

    /* 5. make next of prev_node as new_node */
    prev_node.next = new_node;
}
```

```
public void insertAfter(Node prev_node, int new_data)
{
    /* 1. Check if the given Node is null */
    if (prev_node == null)
    {
        System.out.println("The given previous node cannot be null");
        return;
    }

    /* 2. Allocate the Node &
       3. Put in the data*/
    Node new_node = new Node(new_data);

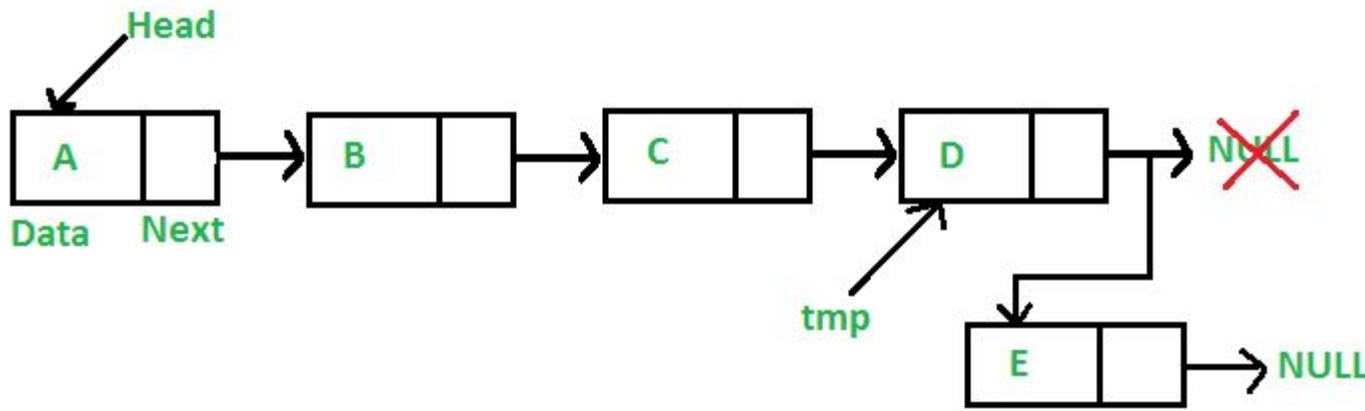
    /* 4. Make next of new Node as next of prev_node */
    new_node.next = prev_node.next;

    /* 5. make next of prev_node as new_node */
    prev_node.next = new_node;
}
```

Time complexity of insertAfter() is O(1) as it does a constant amount of work.

□ Add a node at the end: (6 steps process)

- The new node is always added after the last node of the given Linked List.
- For example if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30.
- Since a Linked List is typically represented by the head of it, we have to traverse the list till the end and then change the next to last node to a new node



```
public void append(int new_data)
{
    /* 1. Allocate the Node &
       2. Put in the data
       3. Set next as null */
    Node new_node = new Node(new_data);

    /* 4. If the Linked List is empty, then make the
       new node as head */
    if (head == null)
    {
        head = new Node(new_data);
        return;
    }

    /* 4. This new node is going to be the last node, so
       make next of it as null */
    new_node.next = null;

    /* 5. Else traverse till the last node */
    Node last = head;
    while (last.next != null)
        last = last.next;

    /* 6. Change the next of last node */
    last.next = new_node;
    return;
}
```

- Time complexity of append is $O(n)$ where n is the number of nodes in the linked list.
- Since there is a loop from head to end, the function does $O(n)$ work.
-

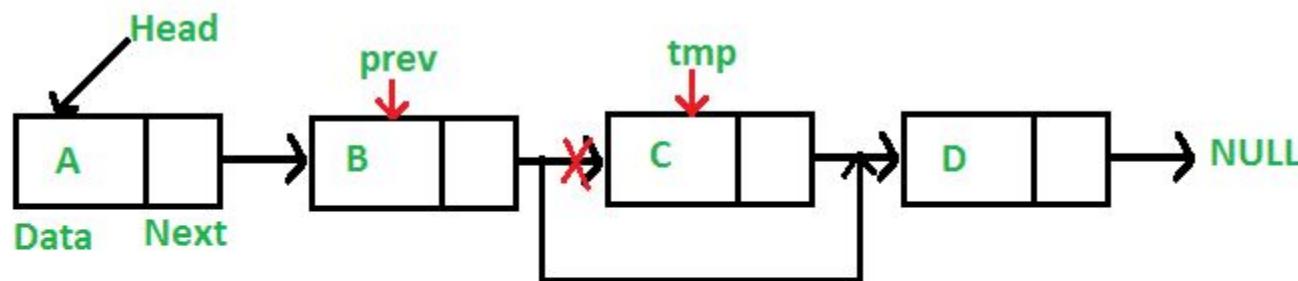
Linked List (Deleting a node)

Given a 'key', delete the first occurrence of this key in the linked list.

Iterative Method:

To delete a node from the linked list, we need to do the following steps.

- 1) Find the previous node of the node to be deleted.
- 2) Change the next of the previous node.
- 3) Free memory for the node to be deleted.



```
/* Given a key, deletes the first
   occurrence of key in
   * linked list */
void deleteNode(int key)
{
    // Store head node
    Node temp = head, prev = null;

    // If head node itself holds the key to be deleted
    if (temp != null && temp.data == key) {
        head = temp.next; // Changed head
        return;
    }

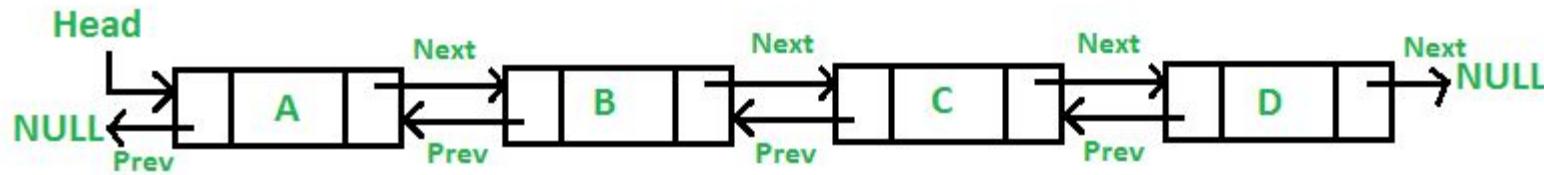
    // Search for the key to be deleted, keep track of
    // the previous node as we need to change temp.next
    while (temp != null && temp.data != key) {
        prev = temp;
        temp = temp.next;
    }

    // If key was not present in linked list
    if (temp == null)
        return;

    // Unlink the node from linked list
    prev.next = temp.next;
}
```

Doubly Linked List (Introduction and Insertion)

A Doubly Linked List (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.



```
// Class for Doubly Linked List
public class DLL {
    Node head; // head of list

    /* Doubly Linked list Node*/
    class Node {
        int data;
        Node prev;
        Node next;

        // Constructor to create a new node
        // next and prev is by default initialized as null
        Node(int d) { data = d; }
    }
}
```

5 May 2022

Advantages over singly linked list

- 1)** A DLL can be traversed in both forward and backward direction.
- 2)** The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
- 3)** We can quickly insert a new node before a given node.

In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Disadvantages over singly linked list

- 1)** Every node of DLL Require extra space for an previous pointer.
- 2)** All operations require an extra pointer previous to be maintained.

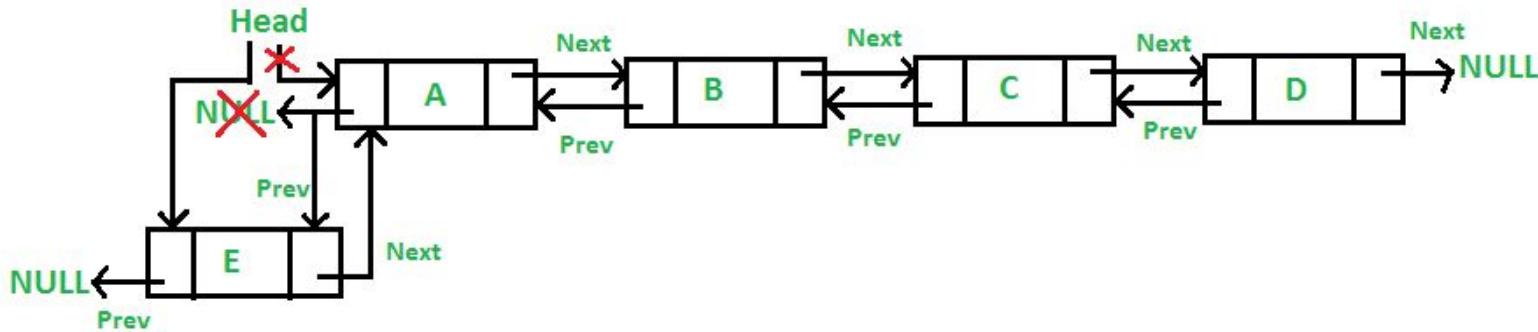
Insertion

A node can be added in four ways

- 1) At the front of the DLL
- 2) After a given node.
- 3) At the end of the DLL
- 4) Before a given node.

1) Add a node at the front: (A 5 steps process)

- The new node is always added before the head of the given Linked List.
- And newly added node becomes the new head of DLL.
- Let us call the function that adds at the front of the list is push().
- The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node



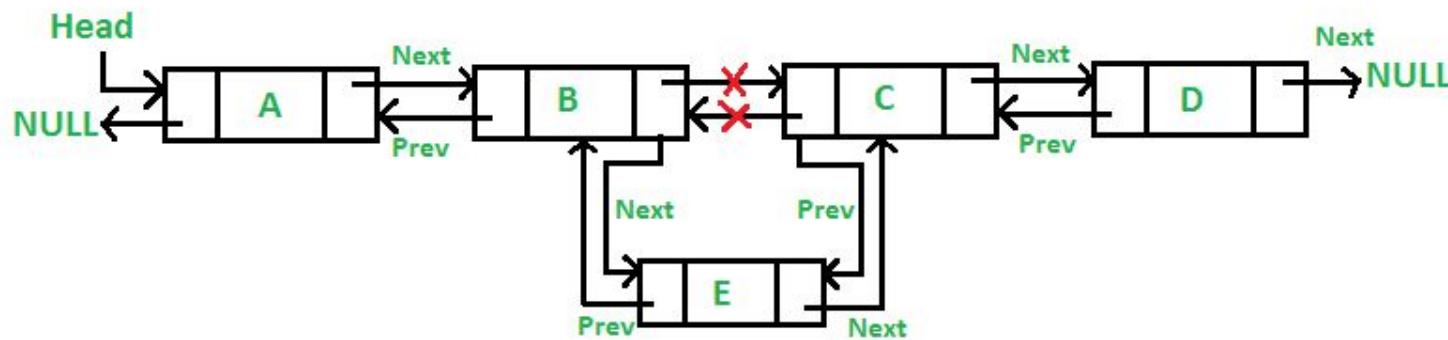
```
// Adding a node at the front of the list
public void push(int new_data)
{
    /* 1. allocate node
     * 2. put in the data */
    Node new_Node = new Node(new_data);

    /* 3. Make next of new node as head and previous as NULL */
    new_Node.next = head;
    new_Node.prev = null;

    /* 4. change prev of head node to new node */
    if (head != null)
        head.prev = new_Node;

    /* 5. move the head to point to the new node */
    head = new_Node;
}
```

2) Add a node after a given node.: (A 7 steps process)



```
/* Given a node as prev_node, insert a new node after the given node */
public void InsertAfter(Node prev_Node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_Node == null) {
        System.out.println("The given previous node cannot be NULL ");
        return;
    }

    /* 2. allocate node
     * 3. put in the data */
    Node new_node = new Node(new_data);

    /* 4. Make next of new node as next of prev_node */
    new_node.next = prev_Node.next;

    /* 5. Make the next of prev_node as new_node */
    prev_Node.next = new_node;

    /* 6. Make prev_node as previous of new_node */
    new_node.prev = prev_Node;

    /* 7. Change previous of new_node's next node */
    if (new_node.next != null)
        new_node.next.prev = new_node;
}
```

Circular Linked List (Introduction and Applications)

- ***Circular linked list*** is a *linked list* where all nodes are connected to form a circle.
- There is no *NULL* at the end.
- A *circular linked list* can be a *singly circular linked list* or *doubly circular linked list*.



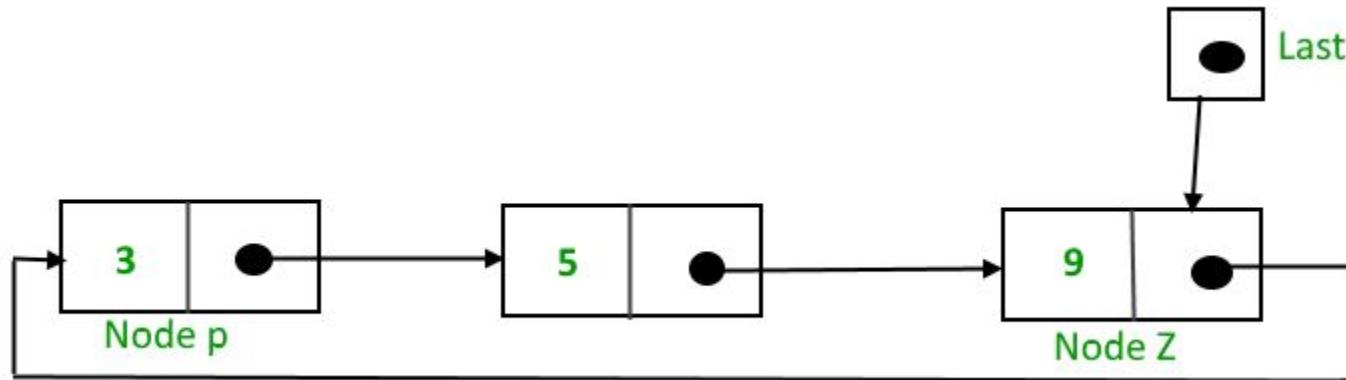
Circular Linked List (Introduction and Applications)

Why Circular?

- In a singly linked list, for accessing any node of the linked list, we start traversing from the first node.
- If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node.
- This problem can be solved by slightly altering the structure of a singly linked list.
- In a singly linked list, the next part (pointer to next node) of last node is NULL.
- If we utilize this link to point to the first node, then we can reach the preceding nodes.

Implementation

- To implement a circular singly linked list, we take an external pointer that points to the last node of the list.
- If we have a pointer last pointing to the last node, then last -> next will point to the first node.



Advantages of Circular Linked Lists:

- 1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- 2) Useful for implementation of queue.
- 3) Circular lists are useful in applications to repeatedly go around the list.
For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
- 4) Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

□ Circular Singly Linked List | Insertion

Insertion

A node can be added in three ways:

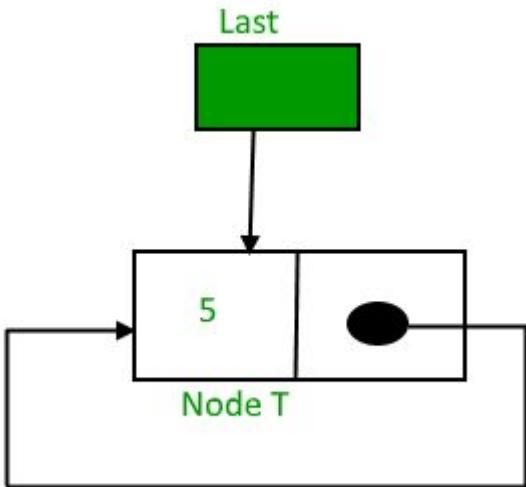
- Insertion in an empty list
- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in between the nodes

Insertion in an empty List

Initially, when the list is empty,
the *last* pointer will be NULL.



After inserting node T,



After insertion, T is the last node, so the pointer *last* points to node T.

And Node T is the first and the last node, so T points to itself.

```

static Node addToEmpty(Node last, int data)
{
    // This function is only for empty list
    if (last != null)
        return last;

    // Creating a node dynamically.
    Node temp = new Node();

    // Assigning the data.
    temp.data = data;
    last = temp;
    // Note : list was empty. We link single node
    // to itself.
    temp.next = last;

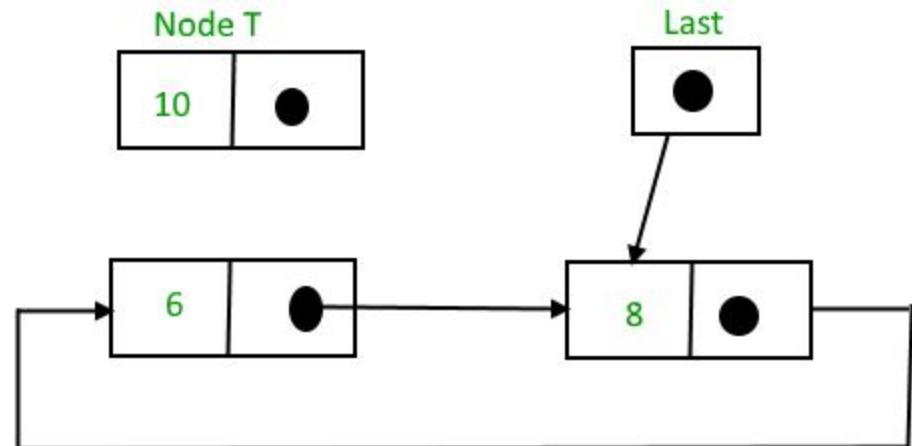
    return last;
}

```

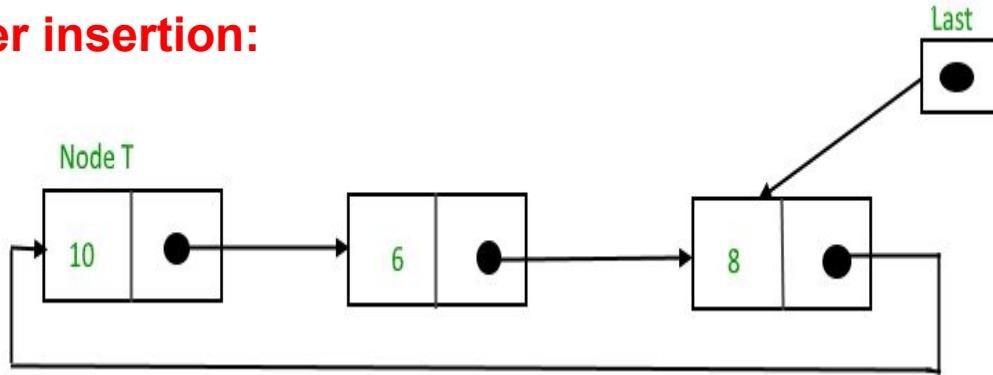
Insertion at the beginning of the list

To insert a node at the beginning of the list follow these steps:

1. Create a node, say T.
2. Make $T \rightarrow \text{next} = \text{last} \rightarrow \text{next}$.
3. $\text{last} \rightarrow \text{next} = T$.



After insertion:



```
static Node addBegin(Node last, int data)
{
    if (last == null)
        return addToEmpty(last, data);

        // Creating a node dynamically
    Node temp = new Node();

        // Assigning the data
    temp.data = data;

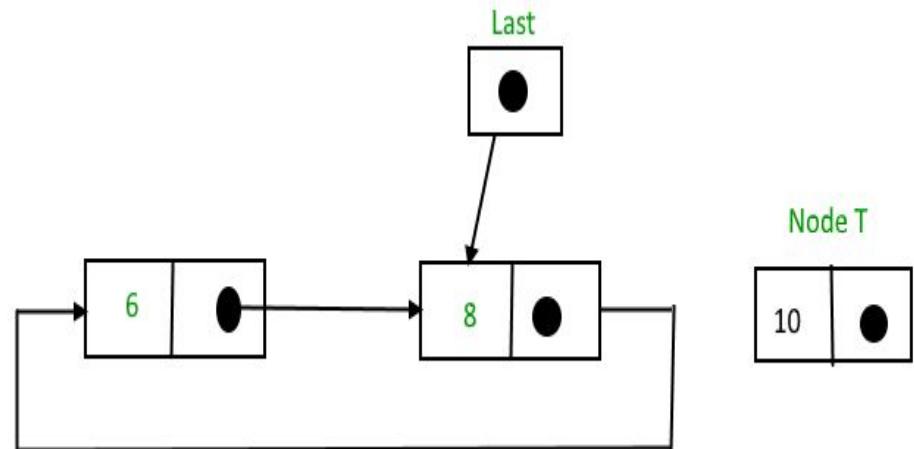
        // Adjusting the links
    temp.next = last.next;
    last.next = temp;

    return last;
}
```

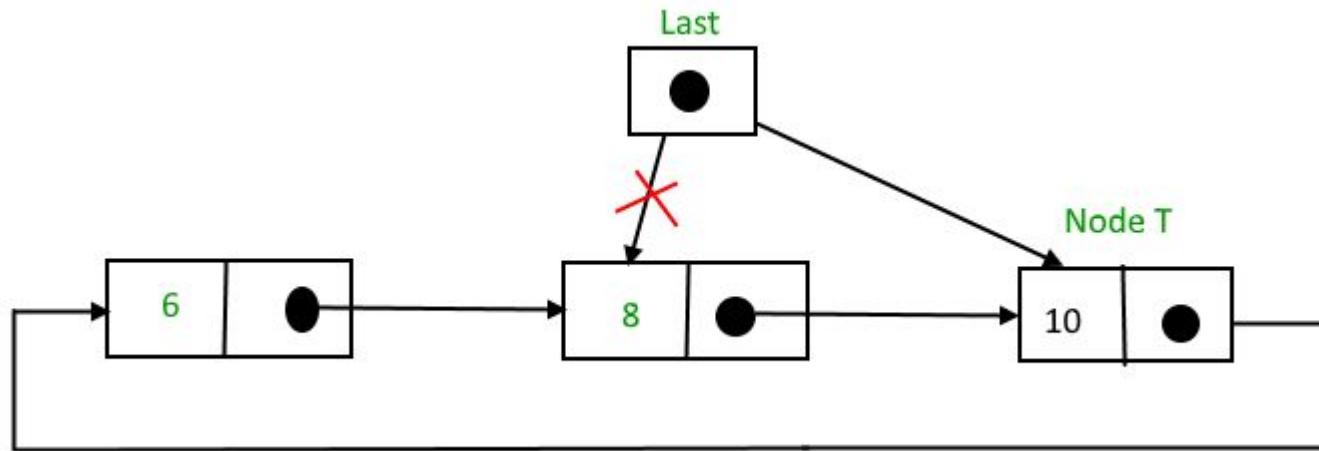
Insertion at the end of the list

To insert a node at the end of the list, follow these steps:

1. Create a node, say T.
2. Make $T \rightarrow \text{next} = \text{last} \rightarrow \text{next}$;
3. $\text{last} \rightarrow \text{next} = T$.
4. $\text{last} = T$.



After insertion:



```
static Node addEnd(Node last, int data)
{
    if (last == null)
        return addToEmpty(last, data);

    // Creating a node dynamically.
    Node temp = new Node();

    // Assigning the data.
    temp.data = data;

    // Adjusting the links.
    temp.next = last.next;
    last.next = temp;
    last = temp;

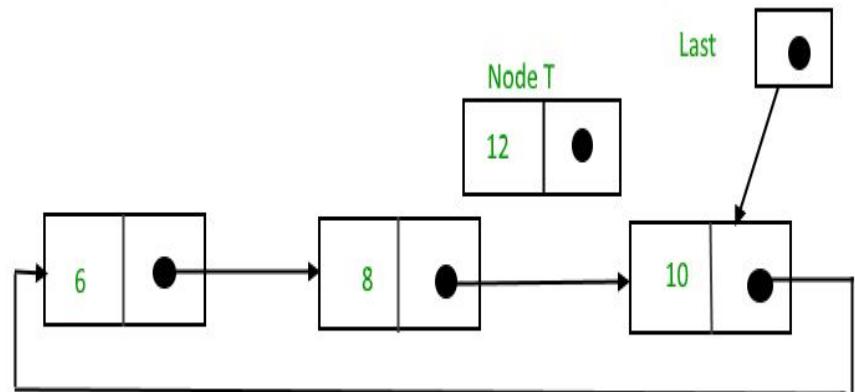
    return last;
}
```

Insertion in between the nodes:

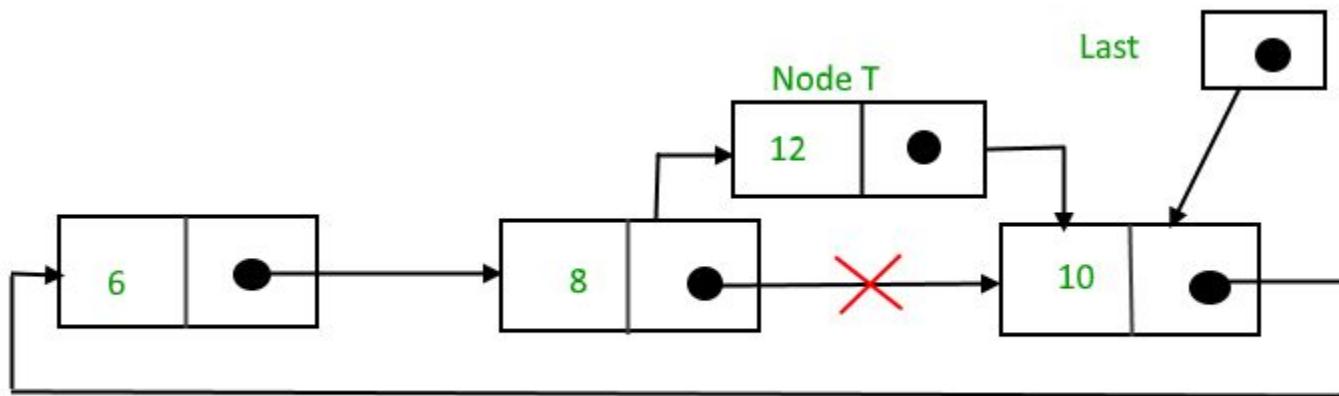
To insert a node in between the two nodes, follow these steps:

1. Create a node, say T.
2. Search for the node after which T needs to be inserted, say that node is P.
3. Make $T \rightarrow \text{next} = P \rightarrow \text{next}$;
4. $P \rightarrow \text{next} = T$.

Suppose 12 needs to be inserted after the node has the value 10,



After searching and insertion:



```
static Node addAfter(Node last, int data, int item)
{
    if (last == null)
        return null;

    Node temp, p;
    p = last.next;
    do
    {
        if (p.data == item)
        {
            temp = new Node();
            temp.data = data;
            temp.next = p.next;
            p.next = temp;

            if (p == last)
                last = temp;
            return last;
        }
        p = p.next;
    } while(p != last.next);

    System.out.println(item + " not present in the list.");
    return last;
}
```

Recursion

- Recursion means "defining a problem in terms of itself".
- This can be a very powerful tool in writing algorithms.
- Recursion comes directly from Mathematics, where there are many examples of expressions written in terms of themselves.
- For example, the Fibonacci sequence is defined as:
$$F(i) = F(i-1) + F(i-2)$$

- Recursion is the process of defining a problem (or the solution to a problem) in terms of (a simpler version of) itself.

For example, we can define the operation "find your way home" as:

1. If you are at home, stop moving.
2. Take one step toward home.
3. "find your way home"

Here the solution to finding your way home is two steps (three steps). First, we don't go home if we are already home. Secondly, we do a very simple action that makes our situation simpler to solve. Finally, we redo the entire algorithm.

The above example is called tail recursion. This is where the very last statement is calling the recursive algorithm. Tail recursion can directly be translated into loops.

Another example of recursion would be finding the maximum value in a list of numbers. The maximum value in a list is either the first number or the biggest of the remaining numbers. Here is how we would write the pseudocode of the algorithm:

Function find_max(list)

 possible_max_1 = first value in list

 possible_max_2 = find_max (rest of the list);

if (possible_max_1 > possible_max_2)

 answer is possible_max_1

else

 answer is possible_max_2

end

end

□ Parts of a Recursive Algorithm

All recursive algorithms must have the following:

1. Base Case (i.e., when to stop)
2. Work toward Base Case
3. Recursive Call (i.e., call ourselves)

The "work toward base case" is where we make the problem simpler (e.g., divide list into two parts, each smaller than the original).

The recursive call, is where we use the same algorithm to solve a simpler version of the problem.

The base case is the solution to the "simplest" possible problem (For example, the base case in the problem 'find the largest number in a list' would be if the list had only one number... and by definition if there is only one number, it is the largest).

□ Simple Example: Sum of digits

```
class sum_of_digits
{
    // Function to check sum
    // of digit using recursion
    static int sum_of_digit(int n)
    {
        if (n == 0)
            return 0;
        return (n % 10 + sum_of_digit(n / 10));
    }

    // Driven Program to check above
    public static void main(String args[])
    {
        int num = 12345;
        int result = sum_of_digit(num);
        System.out.println("Sum of digits in " +
                           num + " is " + result);
    }
}
```

□ Simple Example: Fibonacci Series

```
public class fib {  
    int fibser(int n)  
    {  
        if(n==0)  
        {  
            return 0;  
        }  
        else if(n==1 || n==2)  
        {  
            return 1;  
        }  
        else  
        {  
            return ( fibser(n-1)+fibser(n-2));  
        }  
    }  
}
```

```
fib fibobj = new fib();
```

```
int maxNumber = 10;
```

```
System.out.print("Fibonacci Series of "+maxNumber+" numbers: ");
```

```
for(int i = 0; i < maxNumber; i++)  
{  
    System.out.println(fibobj.fibser(i) + " ");  
}
```

□ How a particular problem is solved using recursion?

- The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion.
- For example, we compute factorial n if we know factorial of $(n-1)$.
- The base case for factorial would be $n = 0$. We return 1 when $n = 0$.

□ Why Stack Overflow error occurs in recursion?

- If the base case is not reached or not defined, then the stack overflow problem may arise.
- Let us take an example to understand this.

```
int fact(int n)
{ // wrong base case (it may cause
// stack overflow).
if (n == 100)
    return 1;
else
    return n*fact(n-1);
}
```

If `fact(10)` is called, it will call `fact(9)`, `fact(8)`, `fact(7)` and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

□ What is the difference between direct and indirect recursion?

- A function fun is called direct recursive if it calls the same function fun.
- A function fun is called indirect recursive if it calls another function say fun_new and fun_new calls fun directly or indirectly.

```
// An example of direct recursion
void directRecFun()
{
    // Some code.....

    directRecFun();

    // Some code...
}
```

```
void indirectRecFun2()
{
    // Some code...

    indirectRecFun1();

    // Some code...
}
```

```
// An example of indirect recursion
void indirectRecFun1()
{
    // Some code...

    indirectRecFun2();

    // Some code...
}
```

□ Tail Recursion :

What is tail recursion?

A recursive function is tail recursive when a recursive call is the last thing executed by the function.

```
// An example of tail recursive function
static void print(int n)
{
    if (n < 0)
        return;

    System.out.print(" " + n);

    // The last executed statement
    // is recursive call
    print(n - 1);
}
```

Why do we care?

- The tail recursive functions considered better than non tail recursive functions as tail-recursion can be optimized by the compiler.
- Compilers usually execute recursive procedures by using a **stack**.
- This stack consists of all the pertinent information, including the parameter values, for each recursive call.
- When a procedure is called, its information is **pushed** onto a stack, and when the function terminates the information is **popped** out of the stack.
-
-
- The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use

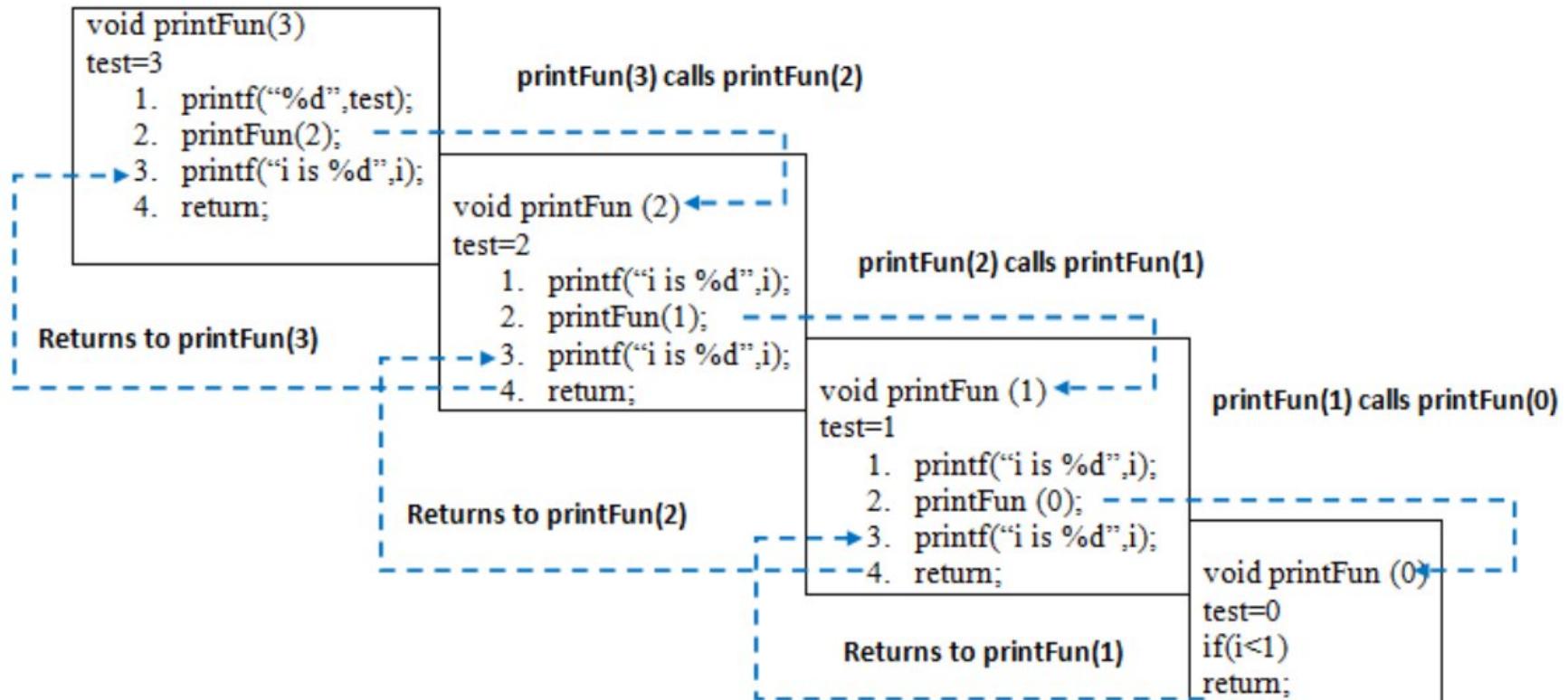
□ How memory is allocated to different function calls in recursion?

- [REDACTED]
- [REDACTED]
- A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to calling function and different copy of local variables is created for each function call.
- When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

```
// A Java program to demonstrate working of
// recursion
class GFG {
    static void printFun(int test)
    {
        if (test < 1)
            return;
        else {
            System.out.printf("%d ", test);
            printFun(test - 1); // statement 2
            System.out.printf("%d ", test);
            return;
        }
    }

    // Driver Code
    public static void main(String[] args)
    {
        int test = 3;
        printFun(test);
    }
}
```

- When **printFun(3)** is called from **main()**, memory is allocated to **printFun(3)** and a local variable test is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram.
It first prints ‘3’.
- In statement 2, **printFun(2)** is called and memory is allocated to **printFun(2)** and a local variable test is initialized to 2 and statement 1 to 4 are pushed in the stack.
- Similarly, **printFun(2)** calls **printFun(1)** and **printFun(1)** calls **printFun(0)**.
printFun(0) goes to if statement and it return to **printFun(1)**.
- Remaining statements of **printFun(1)** are executed and it returns to **printFun(2)** and so on.
- In the output, value from 3 to 1 are printed and then 1 to 3 are printed.



□ What are the disadvantages of recursive programming over iterative programming?

- Note that both recursive and iterative programs have the same problem-solving powers, i.e., every recursive program can be written iteratively and vice versa is also true.
- The recursive program has greater space requirements than iterative program as all functions will remain in the stack until the base case is reached.
- It also has greater time requirements because of function calls and returns overhead.

□ What are the advantages of recursive programming over iterative programming?

- Recursion provides a clean and simple way to write code.
- Some problems are inherently recursive like tree traversals, [Tower of Hanoi](#), etc.
- For such problems, it is preferred to write recursive code.
- We can write such codes also iteratively with the help of a stack data structure.



सी.डैक
CDAC

प्रगत संगणन विकास केंद्र
CENTRE FOR DEVELOPMENT OF ADVANCED COMPUTING

Data Structures

Dr. Priya P Sajan

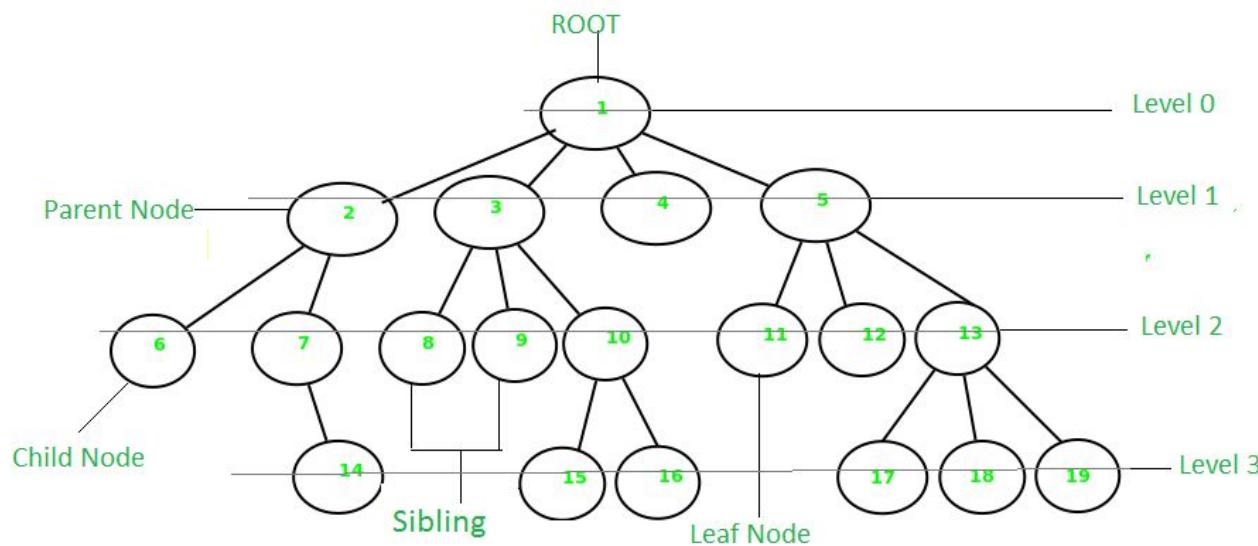
C-DAC

Thiruvananthapuram



Trees

- A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value, a list of references to nodes (the “children”).



Why Tree Data Structure?

- Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.
- Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure

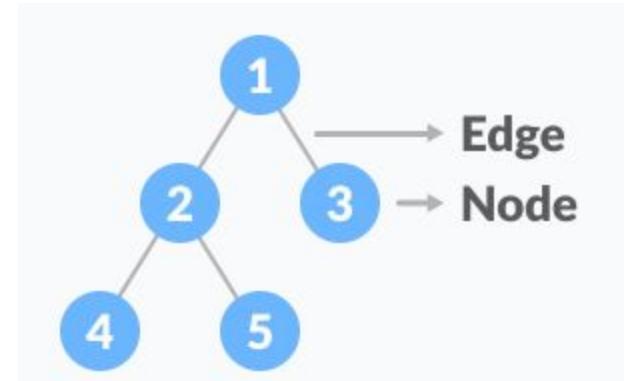
Tree Terminologies

Node

- A node is an entity that contains a key or data value.
- It is a data structure that do not contain a link/pointer to child nodes.
- It is a data structure.

Edge

- It is the link between any two nodes.



Tree Terminologies

Root

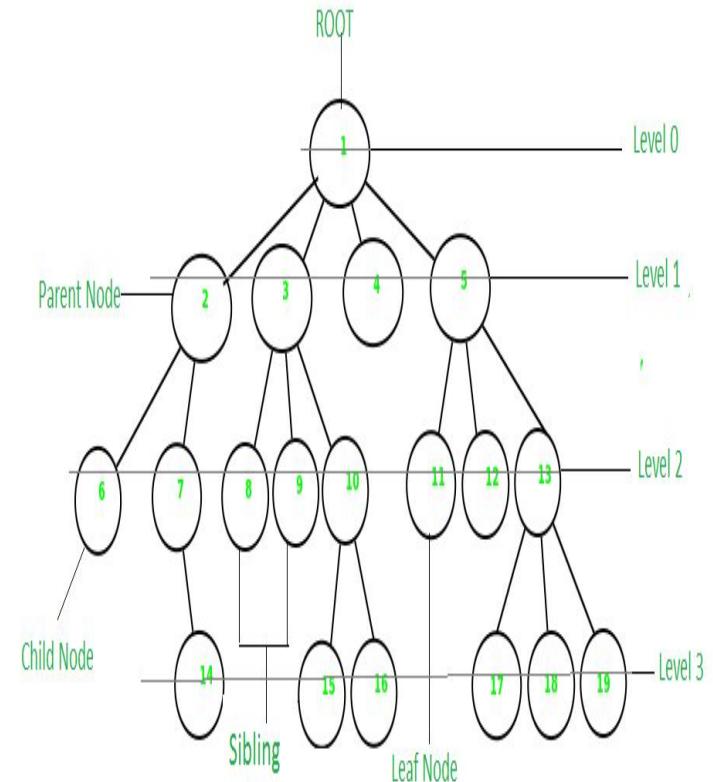
- It is the topmost node of a tree.

Height of a Node

- The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

Depth of a Node

- The depth of a node is the number of edges from the root to the node.



Tree Terminologies

Height of a Tree

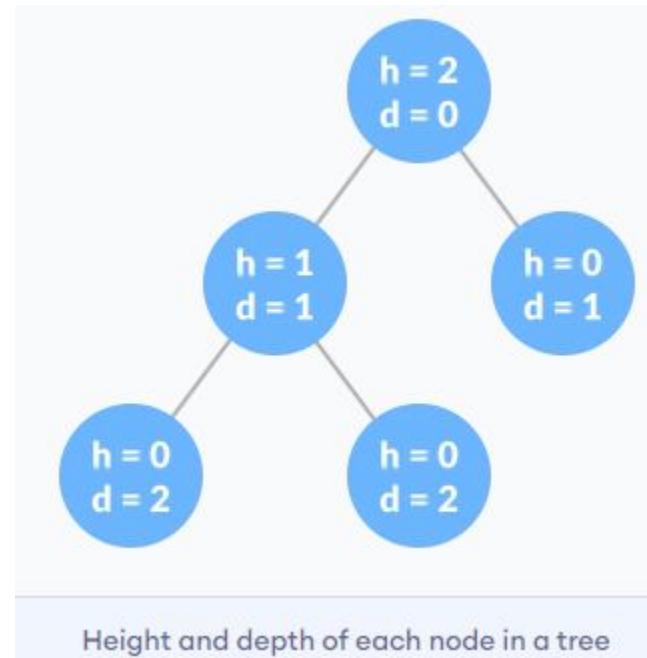
- The height of a Tree is the height of the root node or the depth of the deepest node.

Degree of a Node

- The degree of a node is the total number of branches of that node.

Forest

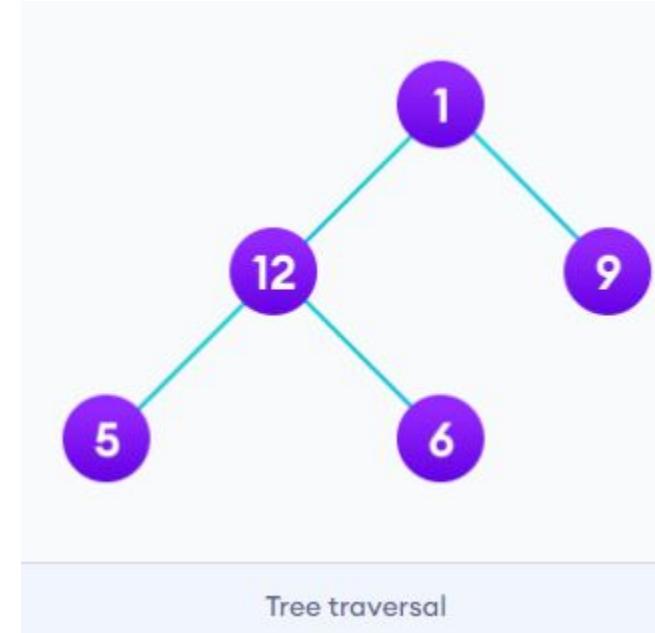
-
-



Implementation of Tree

- The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:
- According to this structure, every tree is a combination of
 - A node carrying data
 - Two subtrees

```
struct node
{
    int data;
    struct node *left;
    struct node *right;
}
```



Tree Traversal

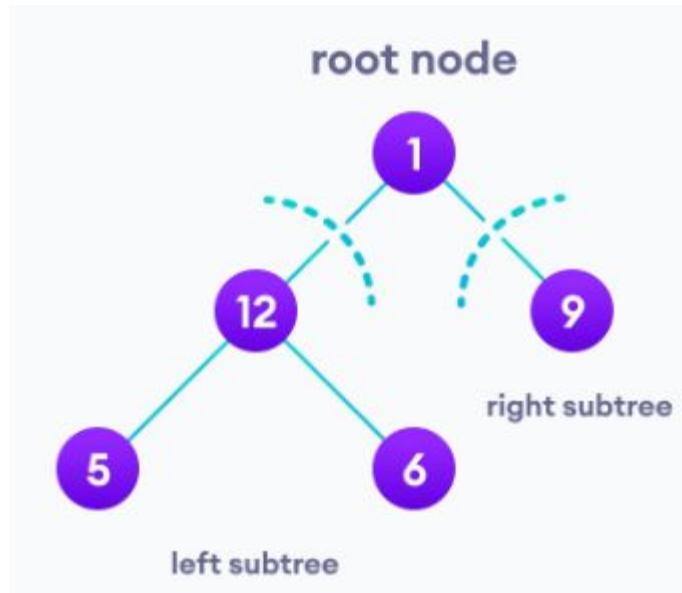
- Traversing a tree means visiting every node in the tree. Inorder to add all the values in the tree or find the largest one, it's required to visit each node of the tree.
- Linear data structures like arrays, stacks, queues, and linked list have only one way to read the data. But a hierarchical data structure like a tree can be traversed in different ways.
 - Inorder
 - Preorder
 - Postorder

Tree Traversal

- Inorder
- Preorder
- Postorder

Inorder Tree Traversal

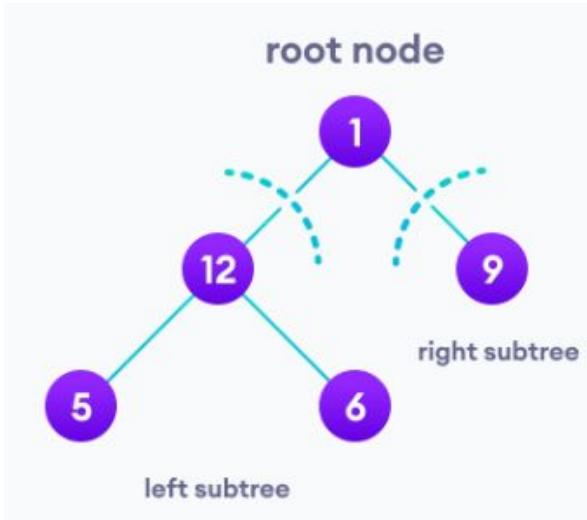
- First, visit all the nodes in the left subtree
- Then the root node
- Visit all the nodes in the right subtree



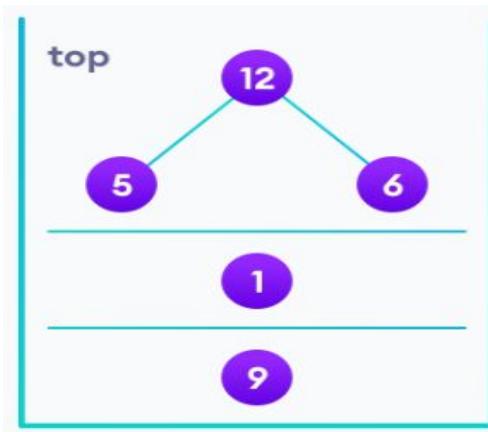
```
inorder(root->left)
display(root->data)
inorder(root->right)
```

Inorder Tree Traversal

- Traverse the left subtree first. Then visit the root node and right subtree .



- Let's put all this in a stack

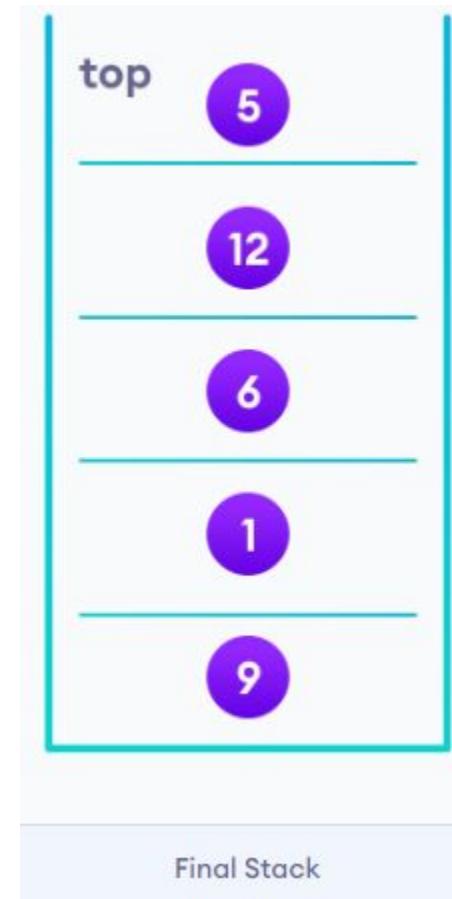


Inorder Tree Traversal

- Now traverse to the subtree pointed on the TOP of the stack.
- Again, follow inorder rule

Left subtree -> root -> right subtree

- After traversing the left subtree,



Preorder Tree Traversal

- Visit root node
- Visit all the nodes in the left subtree
- Visit all the nodes in the right subtree

```
display(root->data)
preorder(root->left)
preorder(root->right)
```

Postorder Tree Traversal

- Visit all the nodes in the left subtree
- Visit all the nodes in the right subtree
- Visit the root node

```
postorder(root->left)
postorder(root->right)
display(root->data)
```

InOrder(root) visits nodes in the following order:

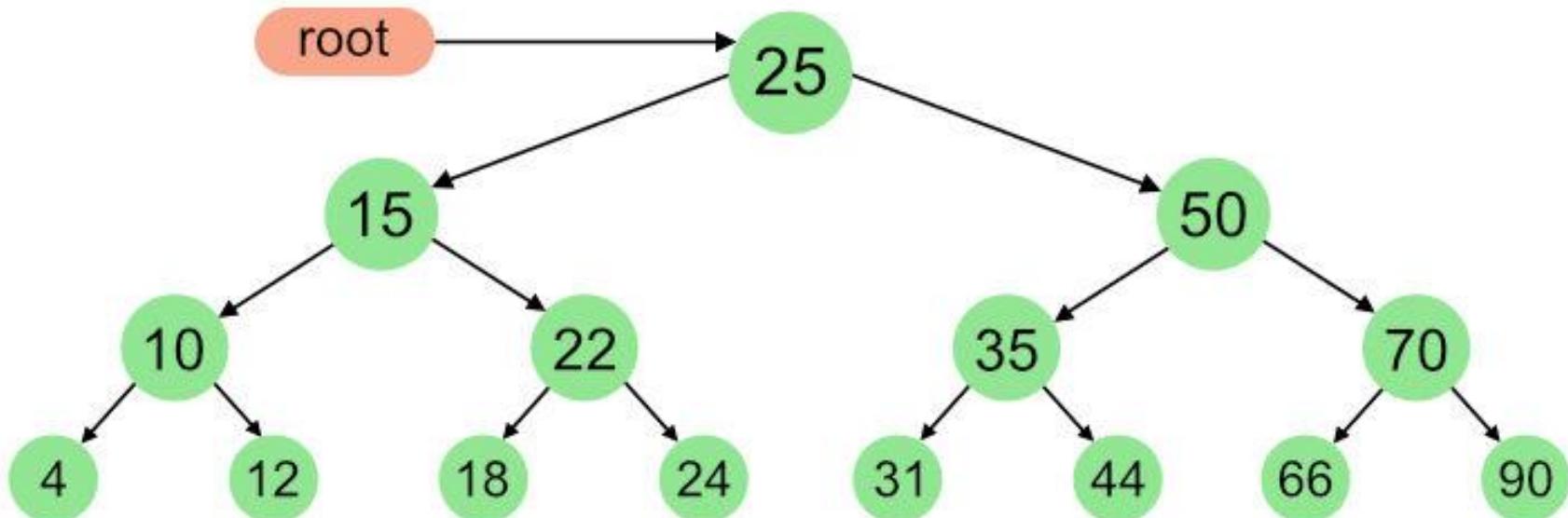
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

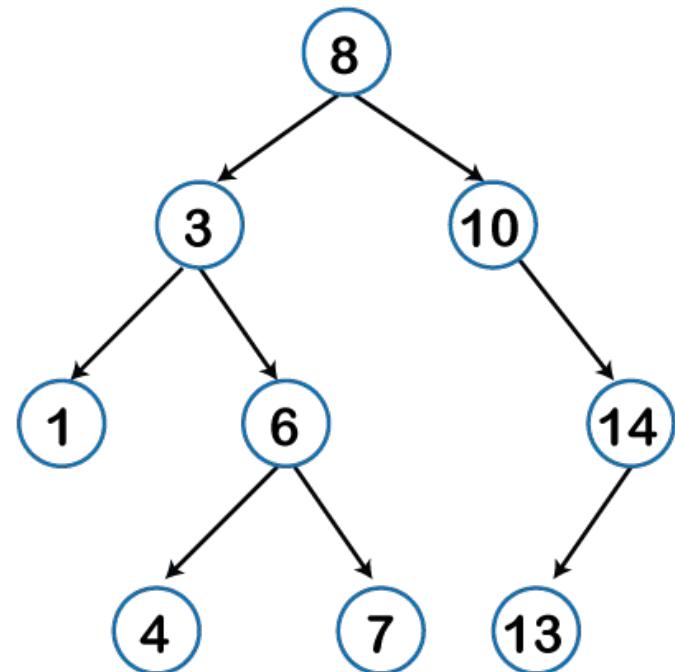
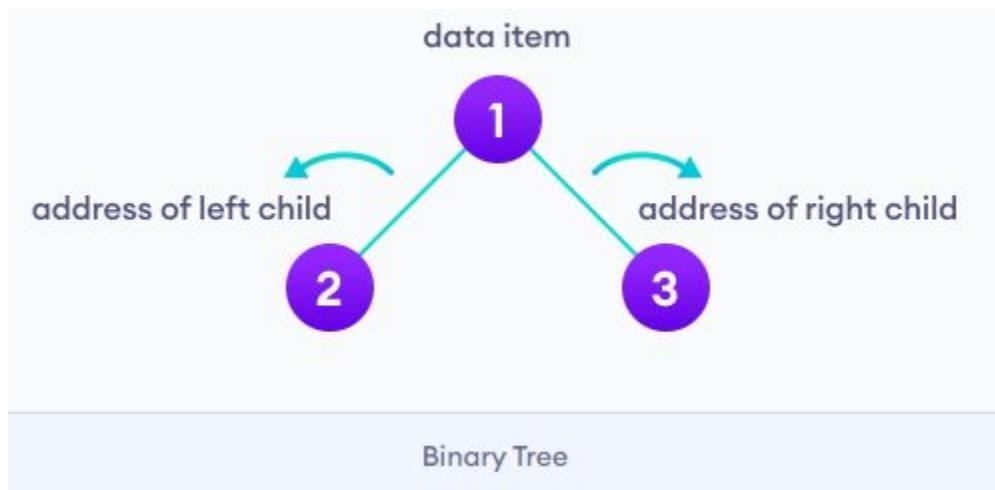
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Binary Trees

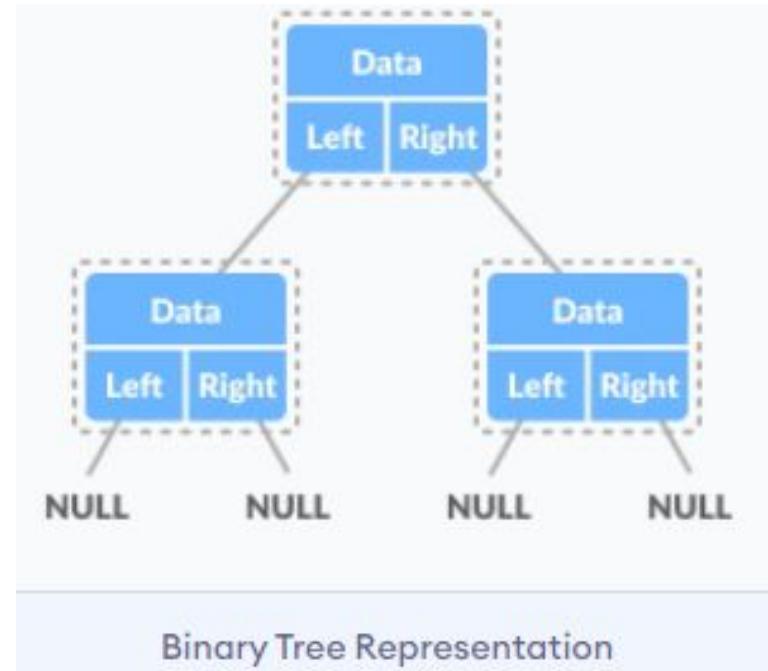
A binary tree is a tree data structure in which each parent node can have at most two children. Each node of a binary tree consists of three items:

- data item
- address of left child
- address of right child



Binary Tree Representation

```
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
```



Binary Tree Representation

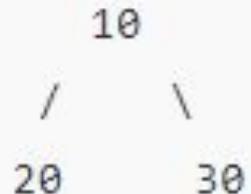
```
/* Class containing left and
right child of current
node and key value*/
class Node
{
    int key;
    Node left, right;

    public Node(int item)
    {
        key = item;
        left = right = null;
    }
}
```

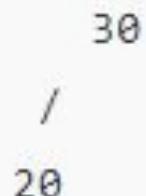
Binary Tree Deletion

1. Starting at the root, find the deepest and rightmost node in binary tree and node to delete.
2. Replace the deepest rightmost node's data with the node to be deleted.
3. Then delete the deepest rightmost node.

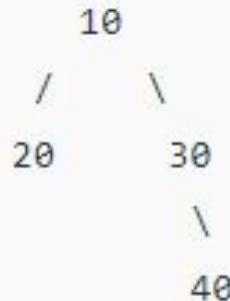
Delete 10 in below tree



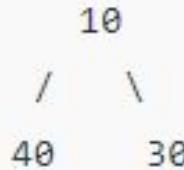
Output :

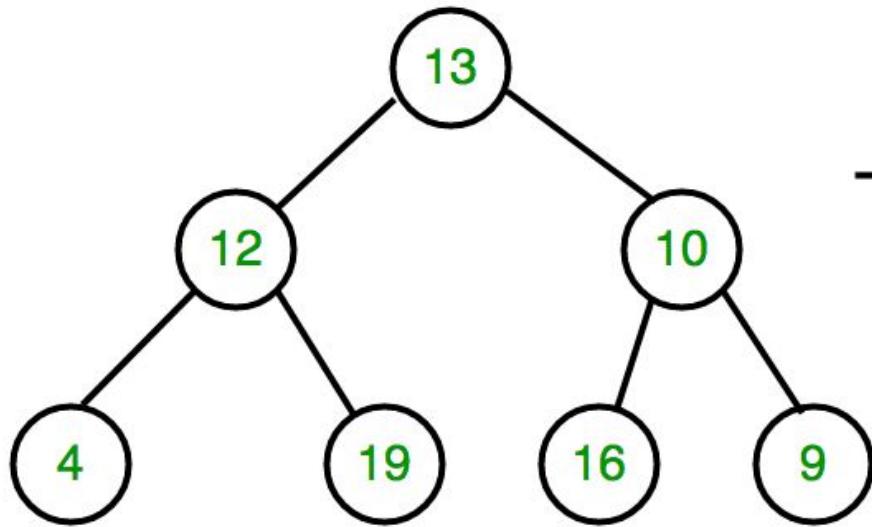


Delete 20 in below tree

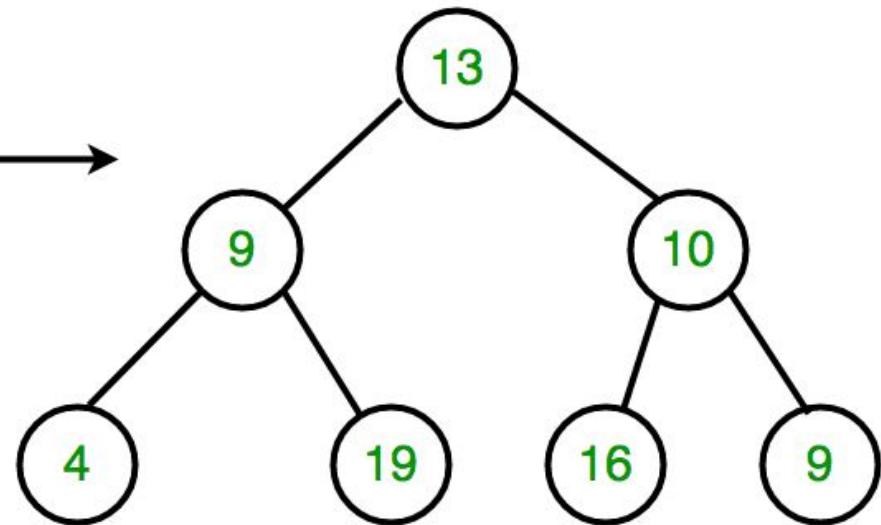


Output :

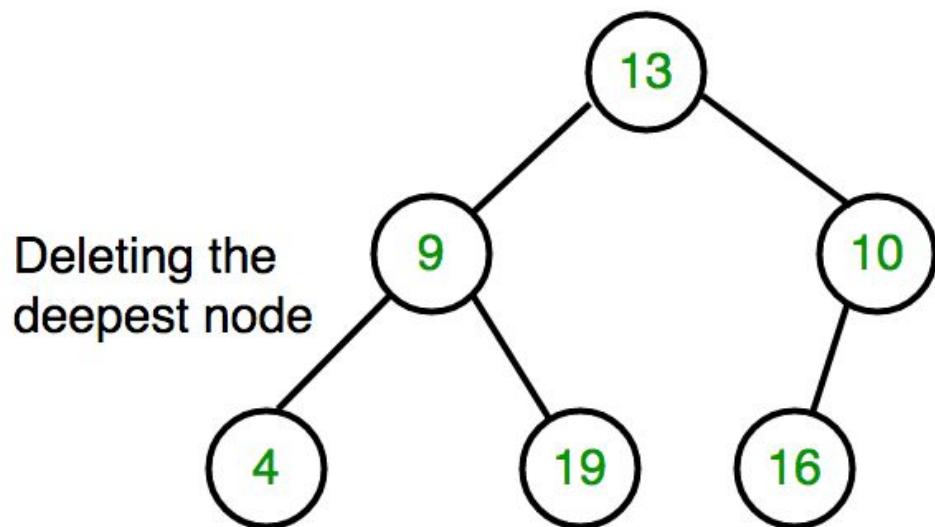




Node to be deleted is 12



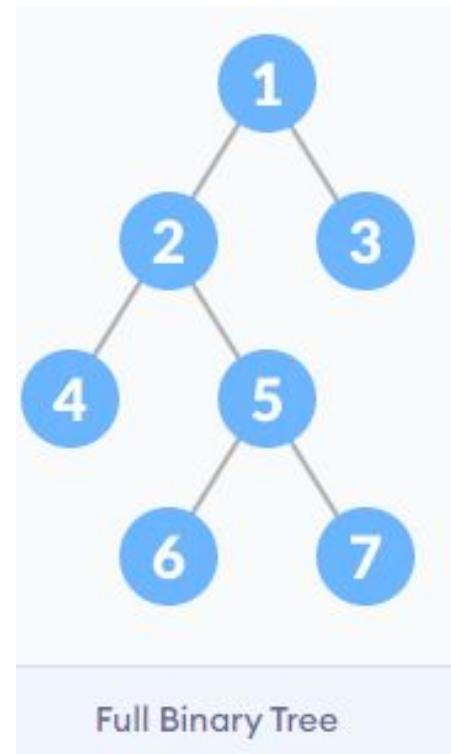
Replacing 12 with
deepest node



Deleting the
deepest node

Binary Tree Properties

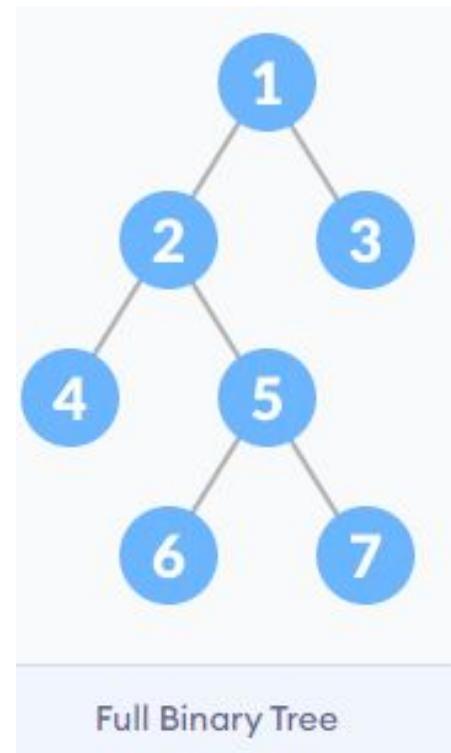
- The maximum number of nodes at level ‘l’ of a binary tree is 2^l .
- The Maximum number of nodes in a binary tree of height ‘h’ is $2^h - 1$.



Types of Binary Tree

Full Binary Tree

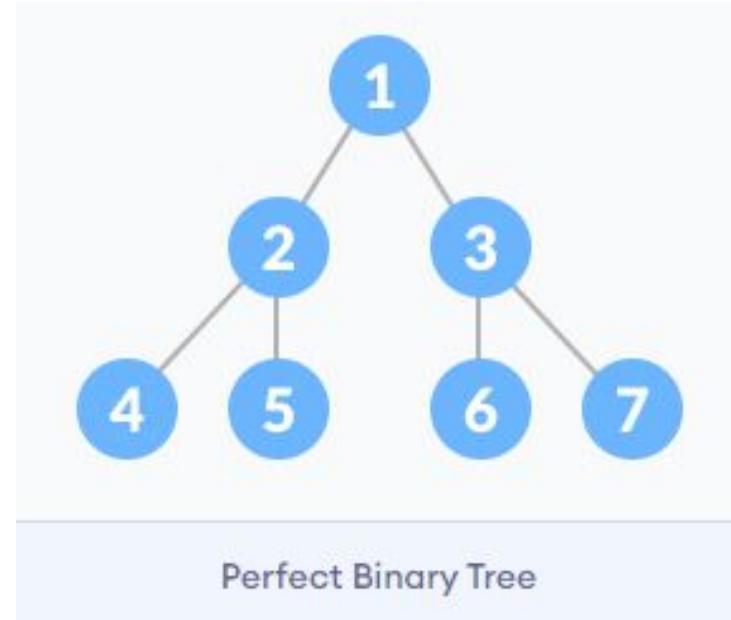
- A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.



Types of Binary Tree

Perfect Binary Tree

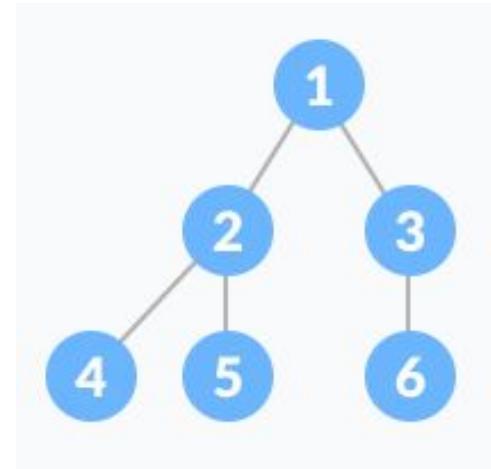
- A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



Types of Binary Tree

Complete Binary Tree

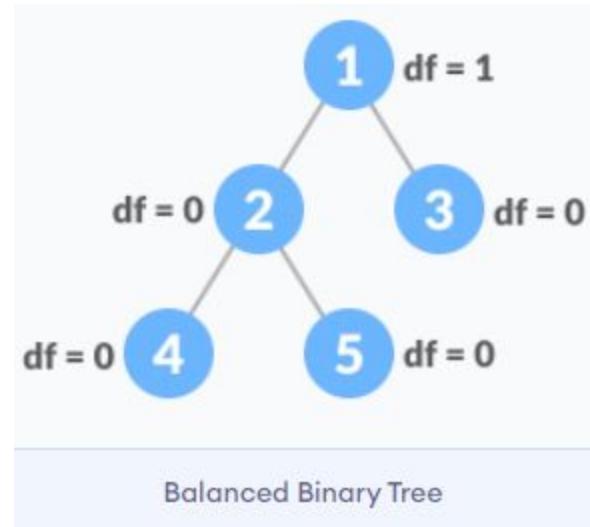
- A complete binary tree is just like a full binary tree, but with two major differences
 - Every level must be completely filled
 - All the leaf elements must lean towards the left.
 - The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



Types of Binary Tree

Balanced Binary Tree

- It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.



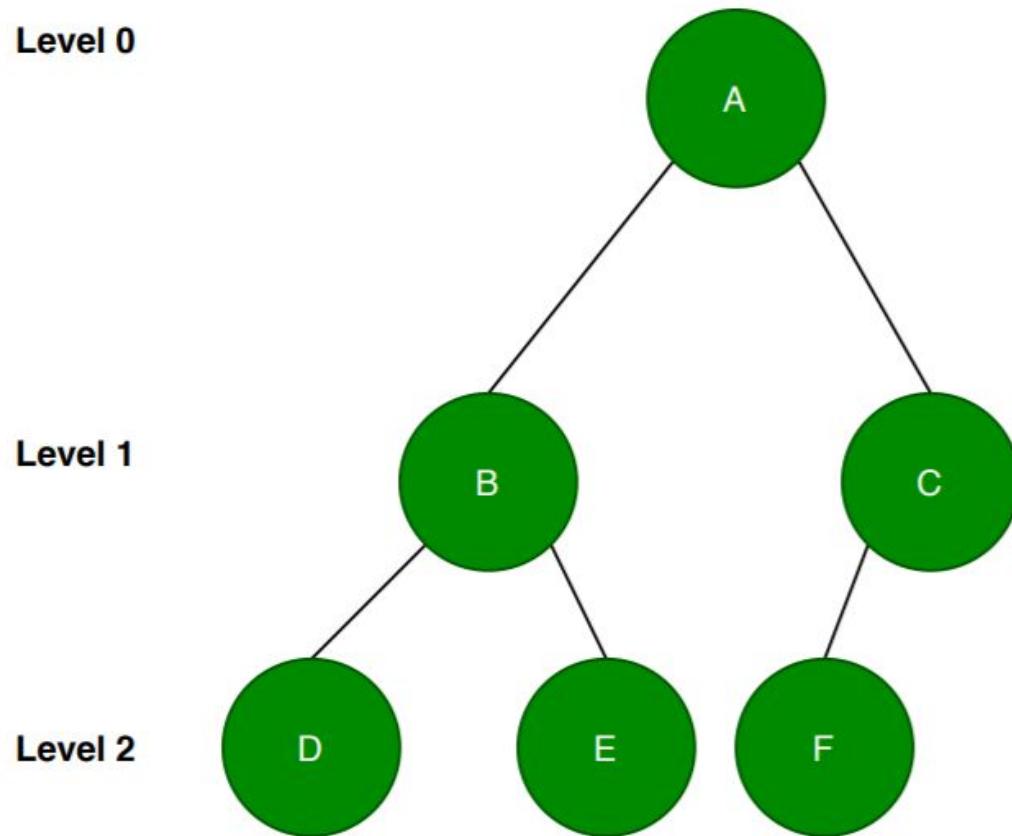
Types of Binary Tree

Almost Complete Binary Tree

- An almost complete binary tree is a special kind of binary tree where insertion takes place level by level and from left to right order at each level and the last level is not filled fully always.
-
-
-
-

Types of Binary Tree

Almost Complete Binary Tree



Types of Binary Tree

Almost Complete Binary Tree

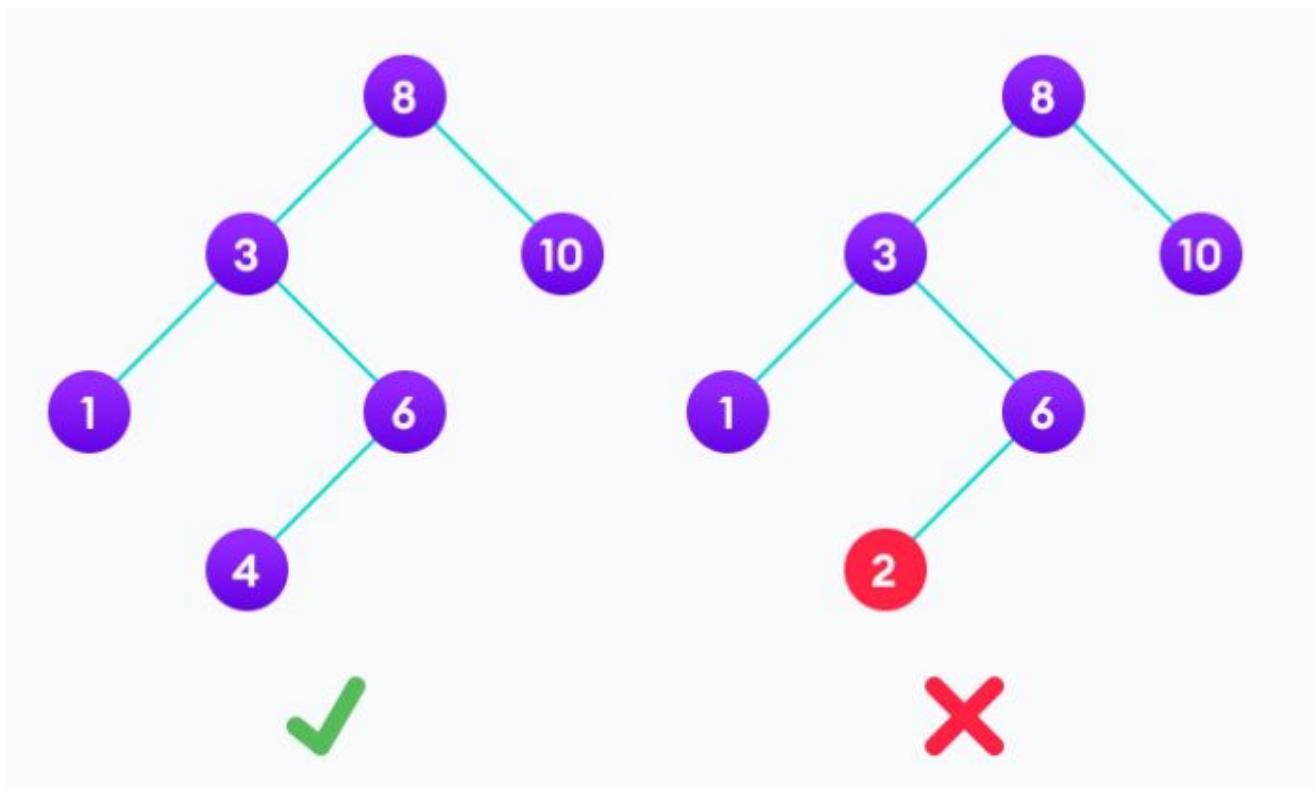
Parameter	Complete Binary Tree	Almost Complete Binary Tree
Definition	When the tree is complete then the last level might or might not be full.	In an almost complete binary tree the last level is not full for sure.
Property	A complete binary tree may or may not be an almost complete binary tree.	An almost complete binary tree will always be a complete binary tree.
Application	Heap data structure.	It doesn't have any applications. If needed, a complete binary tree is used

Binary Search Tree

- Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.
 - It is called a binary tree because each tree node has a maximum of two children.
 - It is called a search tree because it can be used to search for the presence of a number in $O(\log(n))$ time.
- The properties that separate a binary search tree from a regular binary tree is
 - All nodes of left subtree are less than the root node
 - All nodes of right subtree are more than the root node

Binary Search Tree

- The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.



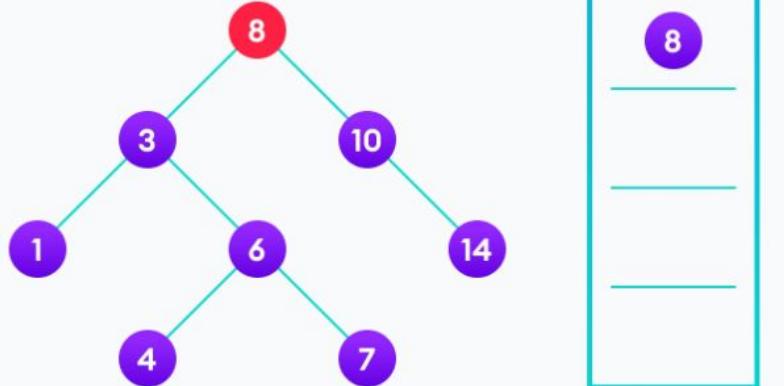
A tree having a right subtree with one value smaller than the root is shown to demonstrate that it is not a valid binary search tree

Binary Search Tree -Search Operation

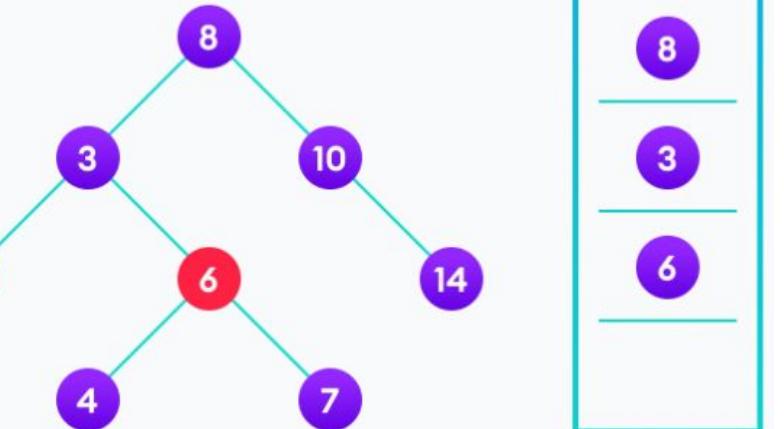
- The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.
- If the value is below the root, then value is not in the right subtree; need to only search in the left subtree and if the value is above the root, then the value is not in the left subtree; and to only search in the right subtree.

```
If root == NULL
    return NULL;
If number == root->data
    return root->data;
If number < root->data
    return search(root->left)
If number > root->data
    return search(root->right)
```

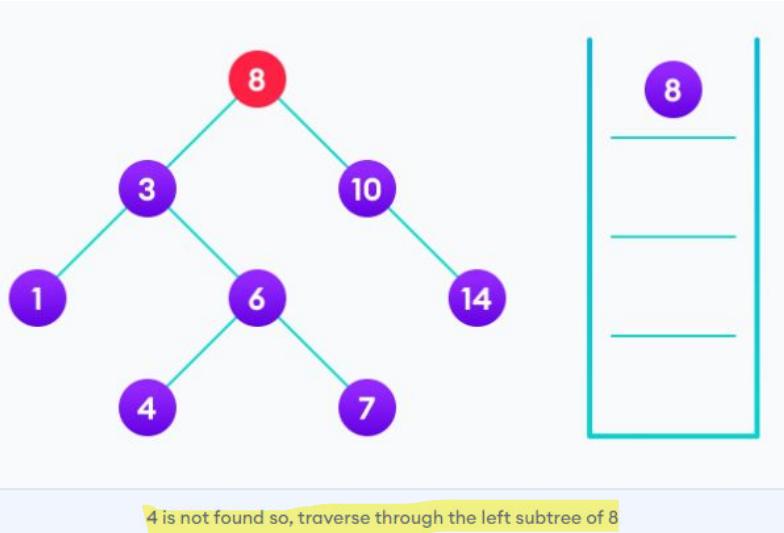
Binary Search Tree -Search Operation



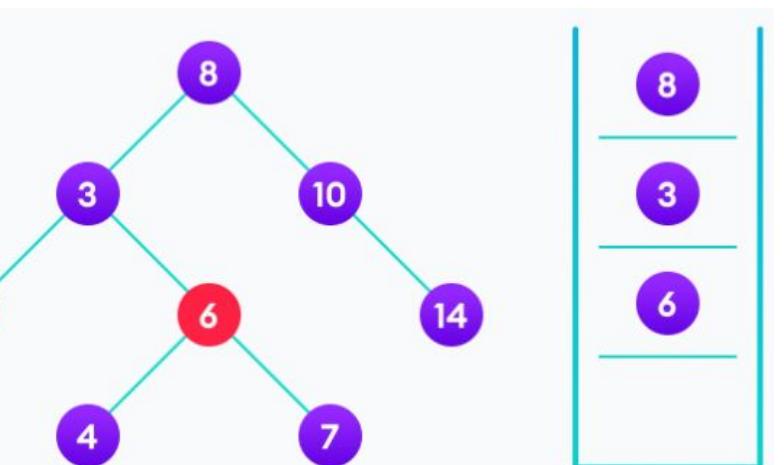
4 is not found so, traverse through the left subtree of 8



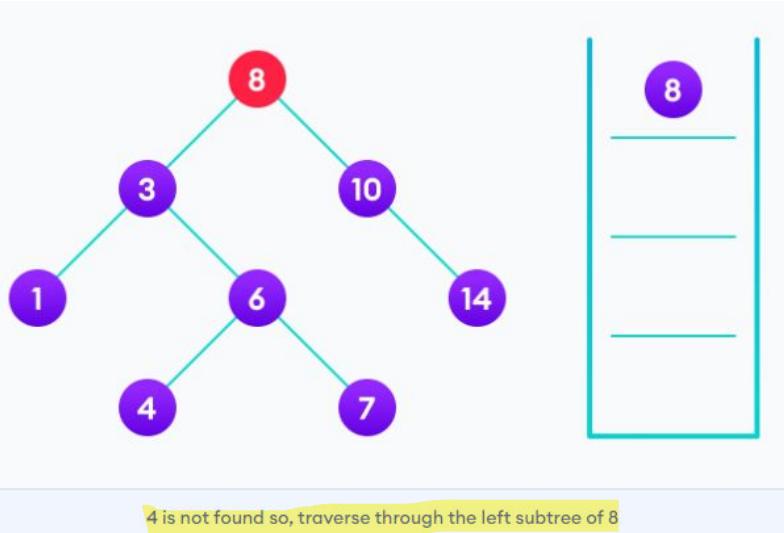
4 is not found so, traverse through the left subtree of 6



4 is not found so, traverse through the right subtree of 3



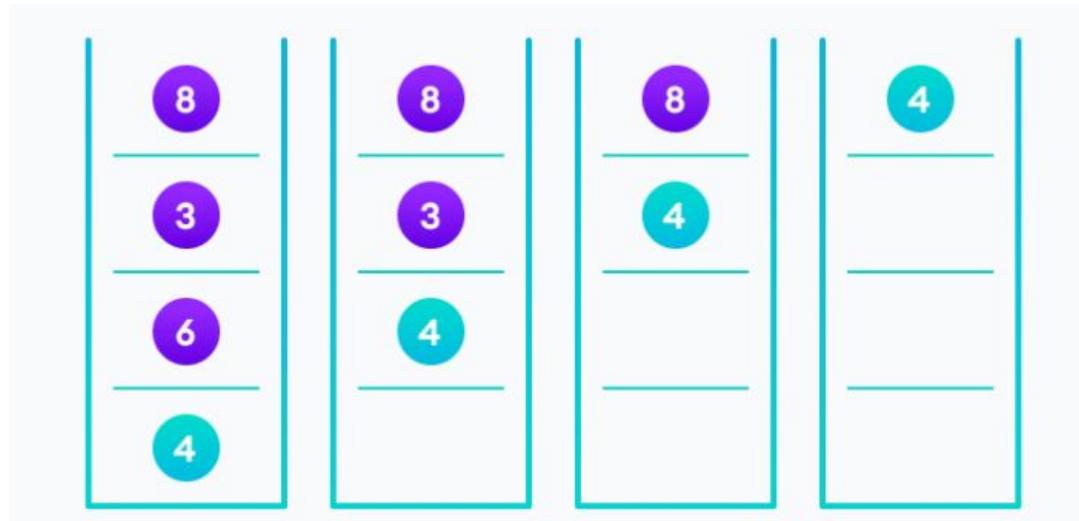
4 is not found so, traverse through the left subtree of 6



4 is found

Binary Search Tree -Search Operation

- If the value is found, return the value so that it gets propagated in each recursion step as shown in the image.
- The value gets returned again and again until search(root) returns the final result
- If the value is not found, we eventually reach the left or right child of a leaf node which is NULL and it gets propagated and returned.



If the value is found in any of the subtrees, it is propagated up so that in the end it is returned, otherwise null is returned

AVL tree

Adelson Velsky and Landis

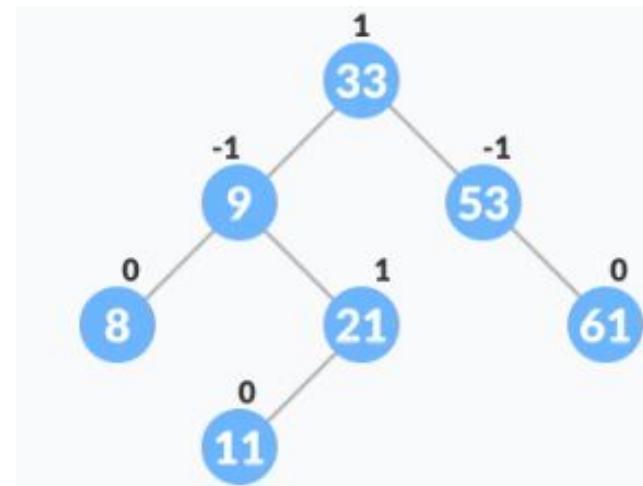
- AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.
- AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.

Balance Factor

- $(\text{Height of Right Subtree} - \text{Height of Left Subtree})$ or (Height

AVL tree

- An example of a balanced avl tree is
- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.



AVL tree

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

AVL tree

Complexity

Algorithm	Average case	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

Why AVL tree ?

- Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take $O(h)$ time where h is the height of the BST.
- The cost of these operations may become $O(n)$ for a skewed Binary tree.
- If it is sure that height of the tree remains $O(\log n)$ after every insertion and deletion, then it is guaranteed an upper bound of $O(\log n)$ for all these operations.
- The height of an AVL tree is always $O(\log n)$ where n is the number of nodes in the tree (See this video lecture for proof).

AVL Rotations

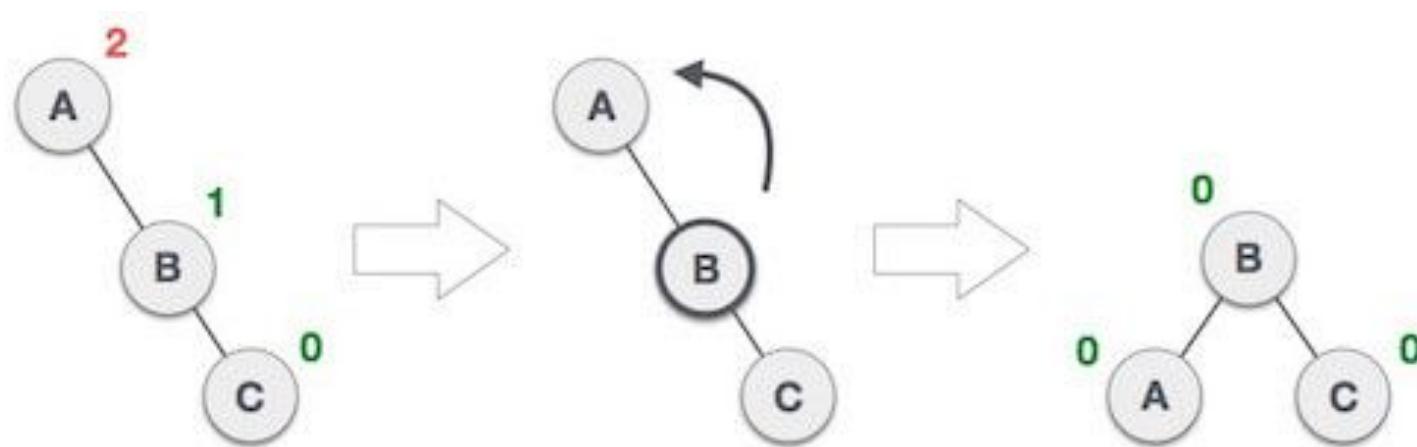
We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations.

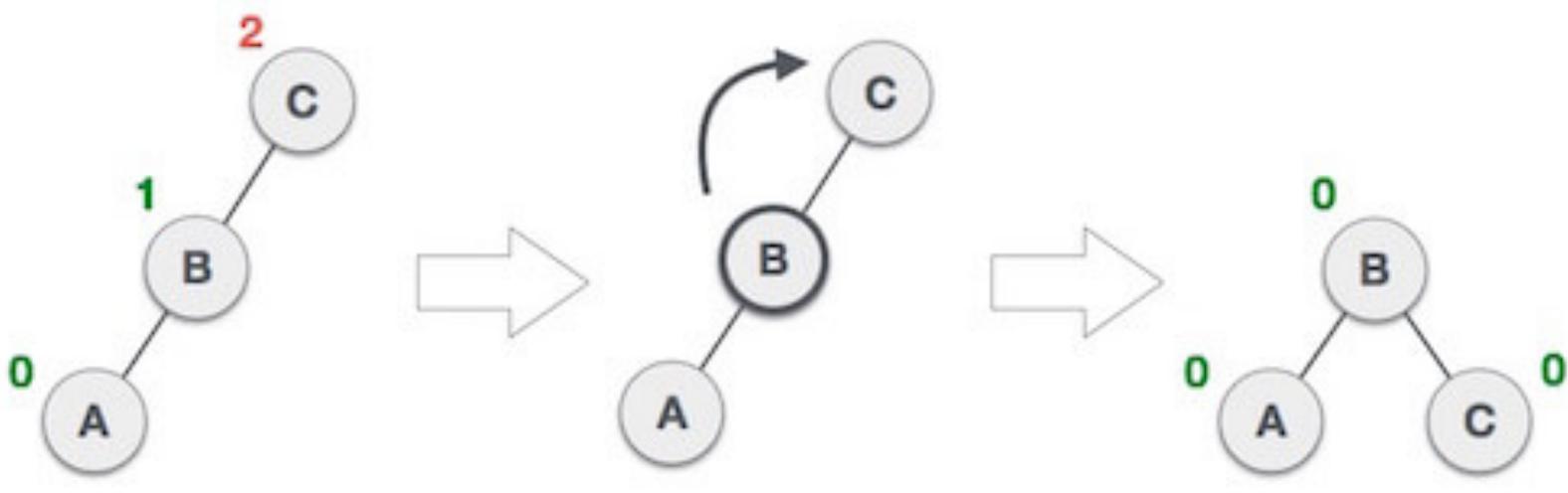
AVL Tree Operations - RR Rotation

- When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2
- In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

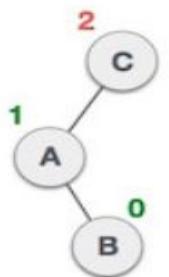


AVL Tree Operations – LL Rotation

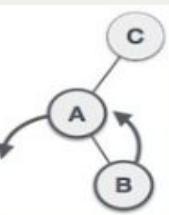
- When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, **LL rotation is clockwise rotation**, which is applied on the edge below a node having balance factor 2.
- In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.



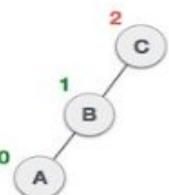
AVL Tree Operations – LR Rotation



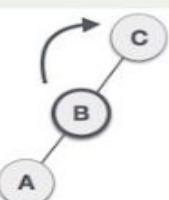
A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C



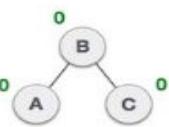
As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**.



After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C**

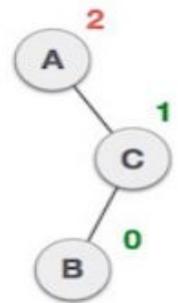


Now we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has now become the right subtree of node B, A is left subtree of B

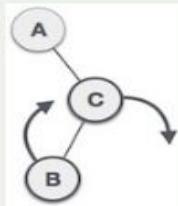


Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

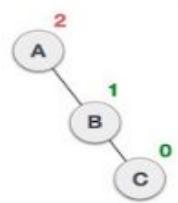
AVL Tree Operations –RL Rotation



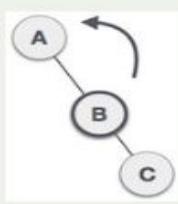
A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A



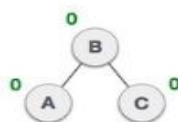
As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**.



After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.



Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B.

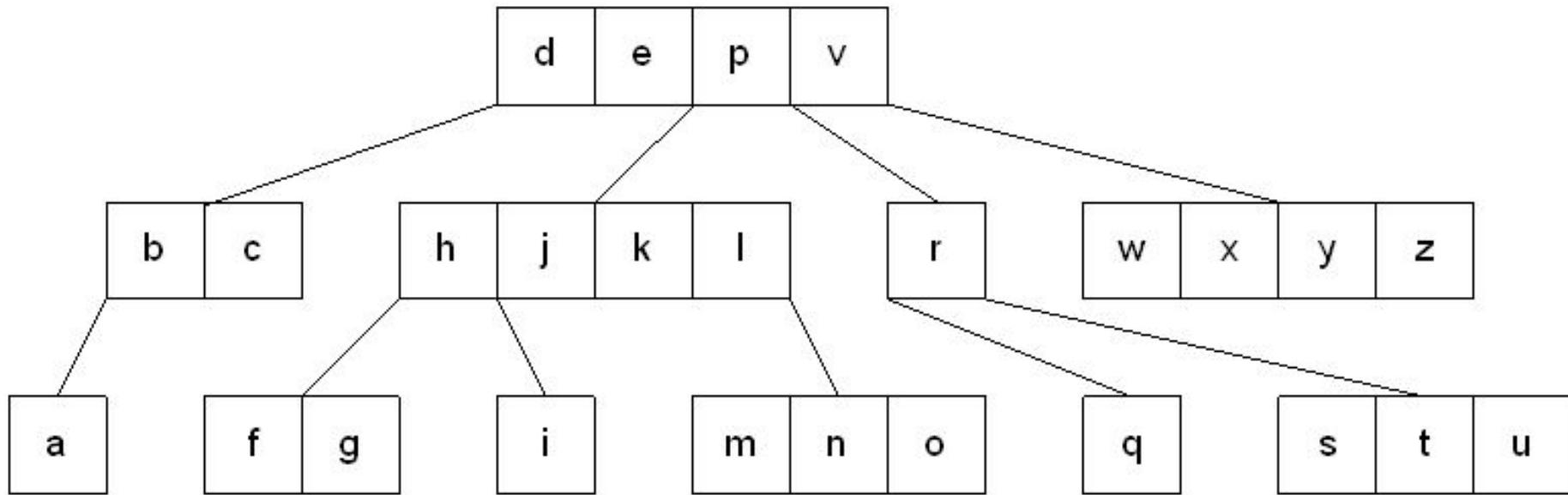


Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

Multiway Trees

- A multiway tree is a tree that can have more than two children.
- A multiway tree of order m (or an m -way tree) is one in which a tree can have m children.
- As with the other trees that have been studied,

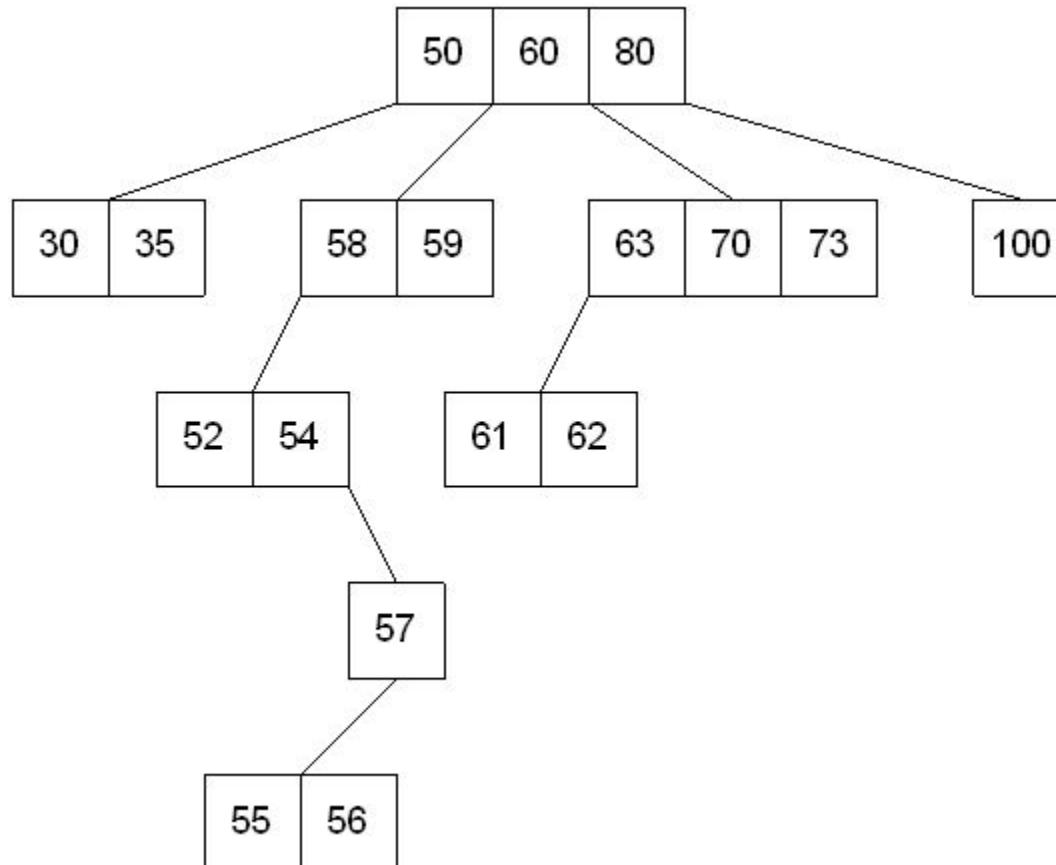
Multiway Trees



Multiway Tree

- To make the processing of m-way trees easier some type of order will be imposed on the keys within each node, resulting in a multiway search tree of order m (or an m-way search tree).
- By definition an m-way search tree is a m-way tree in which:
 - Each node has m children and m-1 key fields
 - The keys in the first i children are smaller than the ith key
 - The keys in the last m-i children are larger than the ith key

Multiway Tree



B-Tree

- An extension of a multiway search tree of order m is a B-tree of order m .
-
-
- In addition, it contains the following properties.
- Every node in a B-Tree contains at most m children.
-
-
-
- These restrictions make B-trees always at least half full, have few levels, and remain perfectly balanced

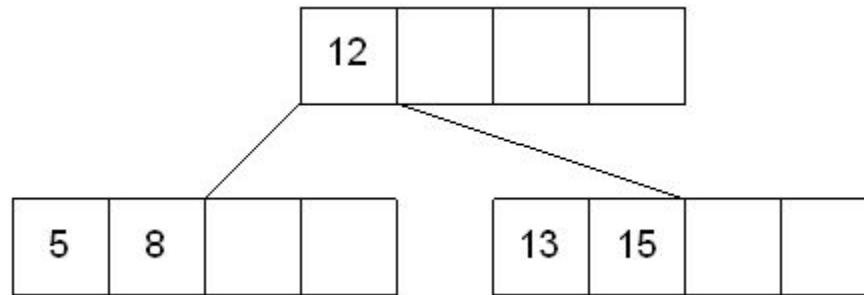
Searching a B-Tree

- An algorithm for finding a key in B-tree is simple.
- Start at the root and determine which pointer to follow based on a comparison between the search value and key fields in the root node.
- Follow the appropriate pointer to a child node.
- Examine the key fields in the child node and continue to follow the appropriate pointers until the search value is found or a leaf node is reached that doesn't contain the desired search value.

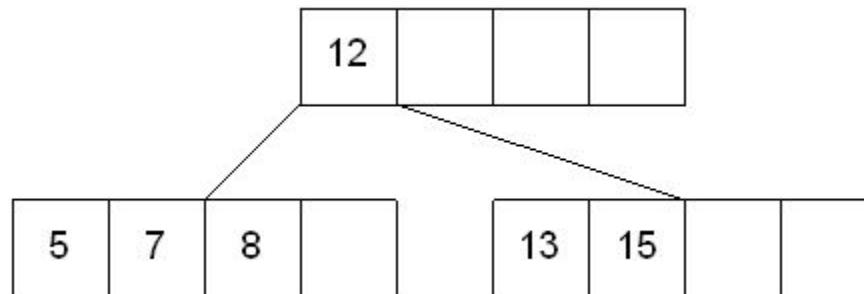
Insertion on a B-Tree

- When inserting into a B-tree, a value is inserted directly into a leaf. This leads to three common situations that can occur:
- A key is placed into a leaf that still has room.
- The leaf in which a key is to be placed is full.
- The root of the B-tree is full.

Insertion on a B-Tree



Inserting the number 7 results in:

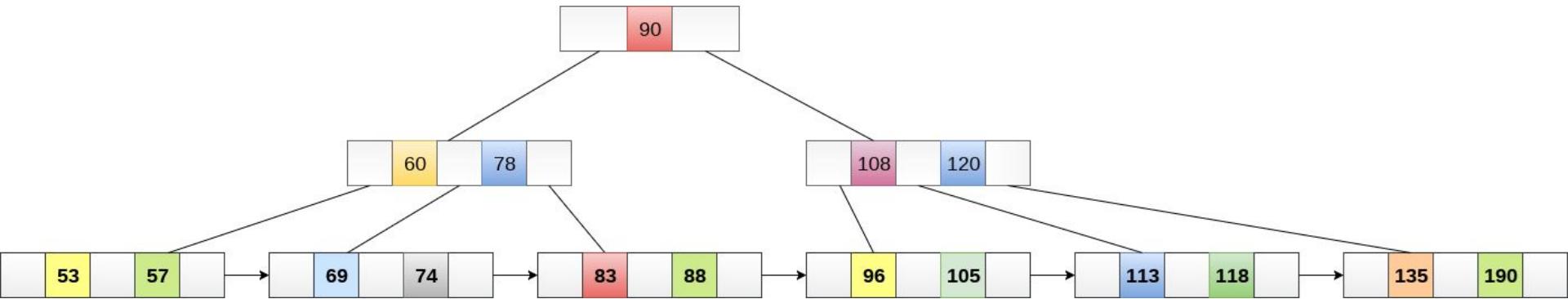


B-Tree - Application

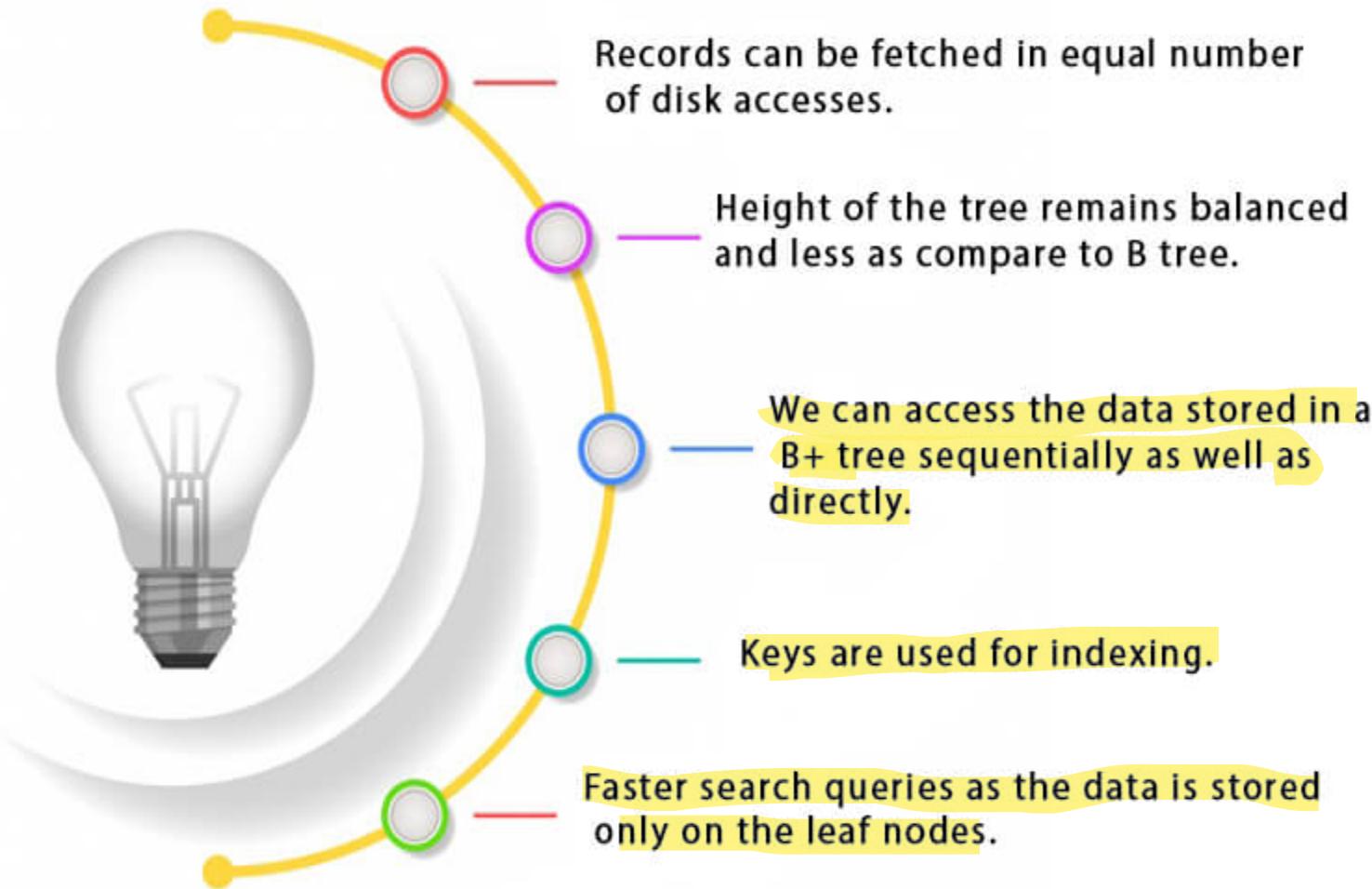
- B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.
- Searching an un-indexed and unsorted database containing n key values needs $O(n)$ running time in worst case.
-
-

- extension of B Tree which allows efficient insertion, deletion and search operations.
 -
 -
 -
 -
 -
 -
 -
 -
 -

B+Tree



Advantages of B+ Tree



B Tree VS B+ Tree

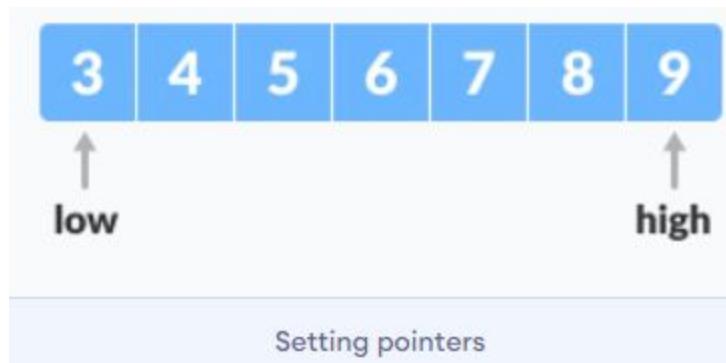
SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

Searching and Sorting

Binary Search

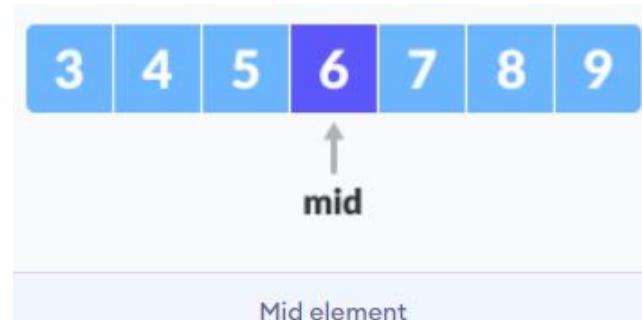


- Let $x = 4$ be the element to be searched.
- Set two pointers low and high at the lowest and the highest positions respectively.



Binary Search

- Find the middle element mid of the array ie. $\text{arr}[(\text{low} + \text{high})/2] = 6$.

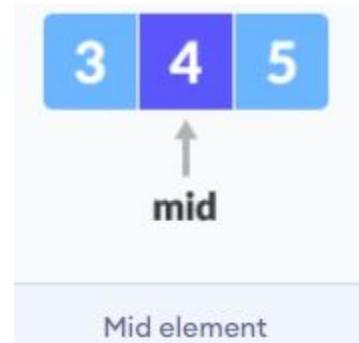


- If $x == \text{mid}$, then return mid. Else, compare the element to be searched with m.
- If $x > \text{mid}$, compare x with the middle element of the elements on the right side of mid. This is done by setting low to $\text{low} = \text{mid} + 1$.
- Else, compare x with the middle element of the elements on the left side of mid. This is done by setting high to $\text{high} = \text{mid} - 1$.



Binary Search

- Repeat steps 3 to 6 until low meets high.



- $x = 4$ is found



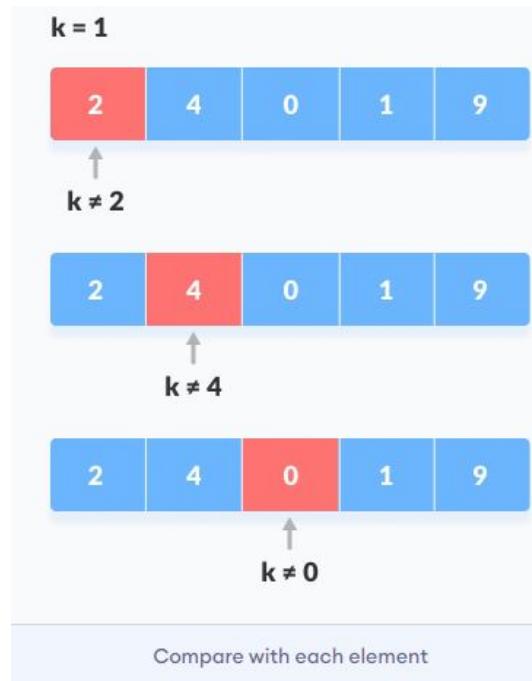
Linear Search

- Linear search is a sequential searching algorithm where to start from one end and check every element of the list until the desired element is found. It is the simplest searching algorithm.
 - `LinearSearch(array, key)`
 - for each item in the array
 - if item == value
 - return its index
- The following steps are followed to search for an element k = 1 in the list below.

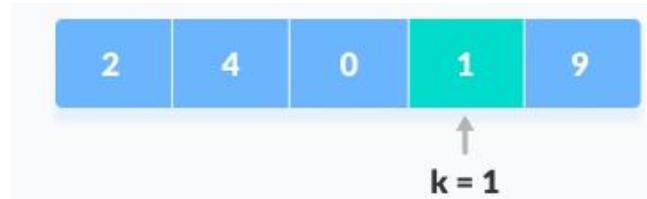
2	4	0	1	9
Array to be searched for				

Linear Search

- Start from the first element, compare k with each element x



- If $x == k$, return the index.
- Else, return not found.



Sorting

Sorting

- Rearranges a given array or list elements according to a comparison operator on the elements.
- The comparison operator is used to decide the new order of element in the respective data structure.
 - Selection sort
 - Insertion sort
 - Bubble sort
 - Heap sort
 - Merge sort
 - Quick sort

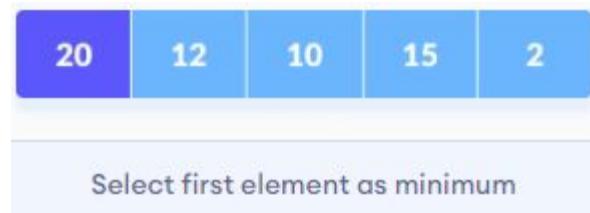
Selection Sort

- Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

```
selection Sort(array, size)
repeat (size - 1) times
    set the first unsorted element as the minimum
    for each of the unsorted elements
        if element < currentMinimum
            set element as new minimum
        swap minimum with first unsorted position
end selectionSort
```

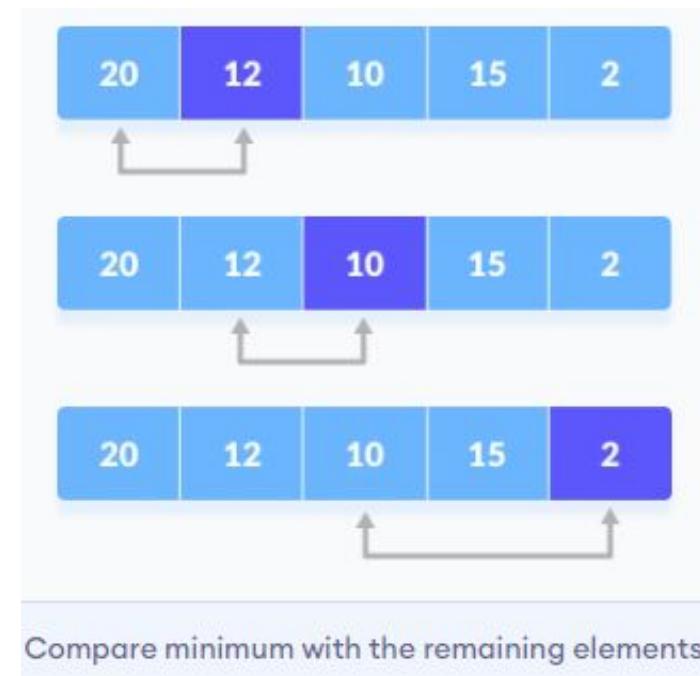
Working of Selection Sort

- Set the first element as minimum



Working of Selection Sort

- Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.
- Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.

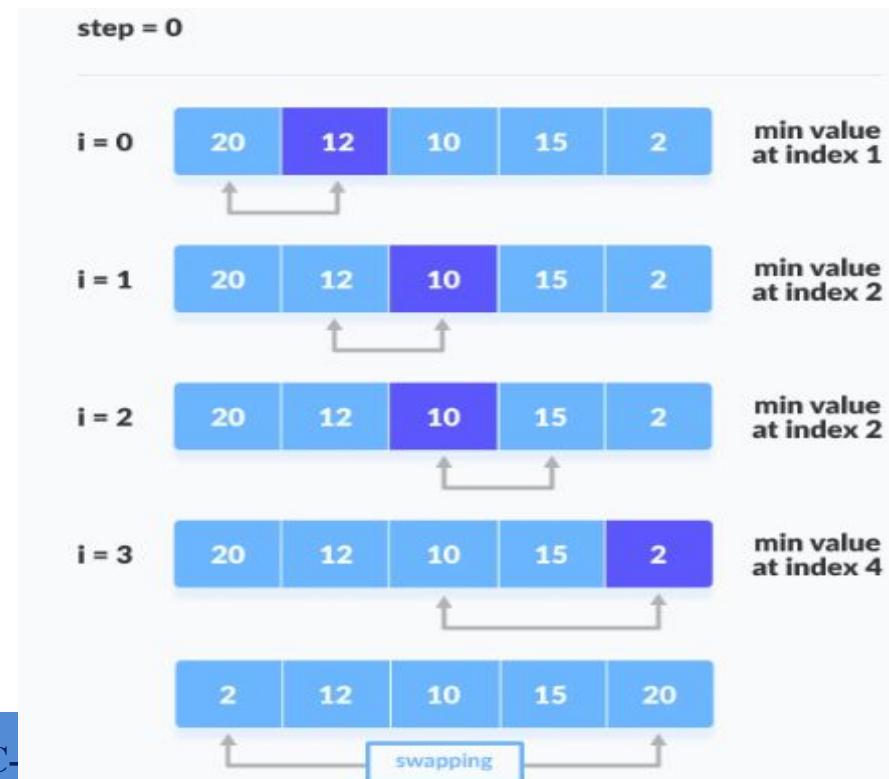


Working of Selection Sort

- After each iteration, minimum is placed in the front of the unsorted list

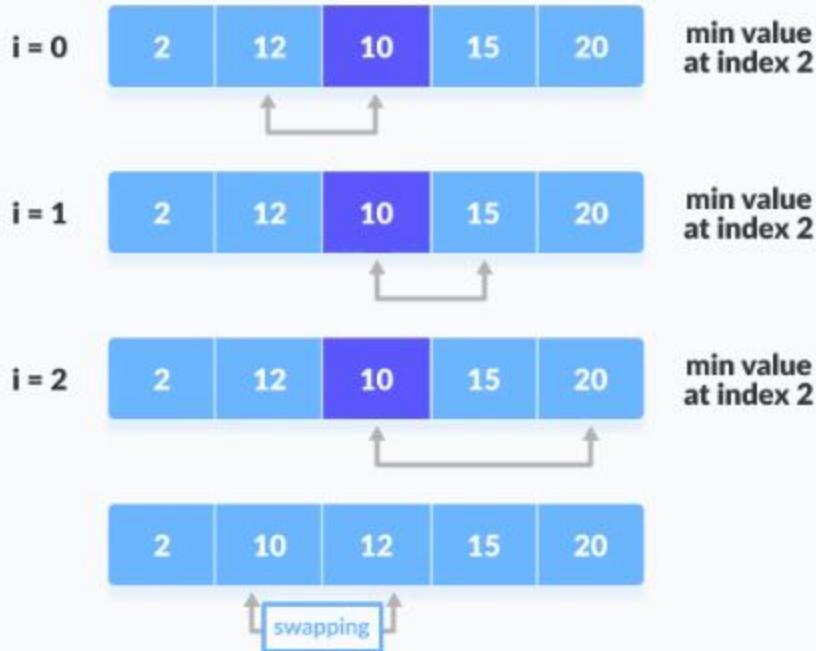


- For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.



Working of Selection Sort

step = 1



step = 2



step = 3



Insertion Sort

- Insertion sort places an unsorted element at its suitable place in each iteration.
- Insertion sort works similarly as sort cards in our hand in a card game. Assume that the first card is already sorted then, select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

insertion Sort(array)

 mark first element as sorted

 for each unsorted element X

 'extract' the element X

 for j <- lastSortedIndex down to 0

 if current element j > X

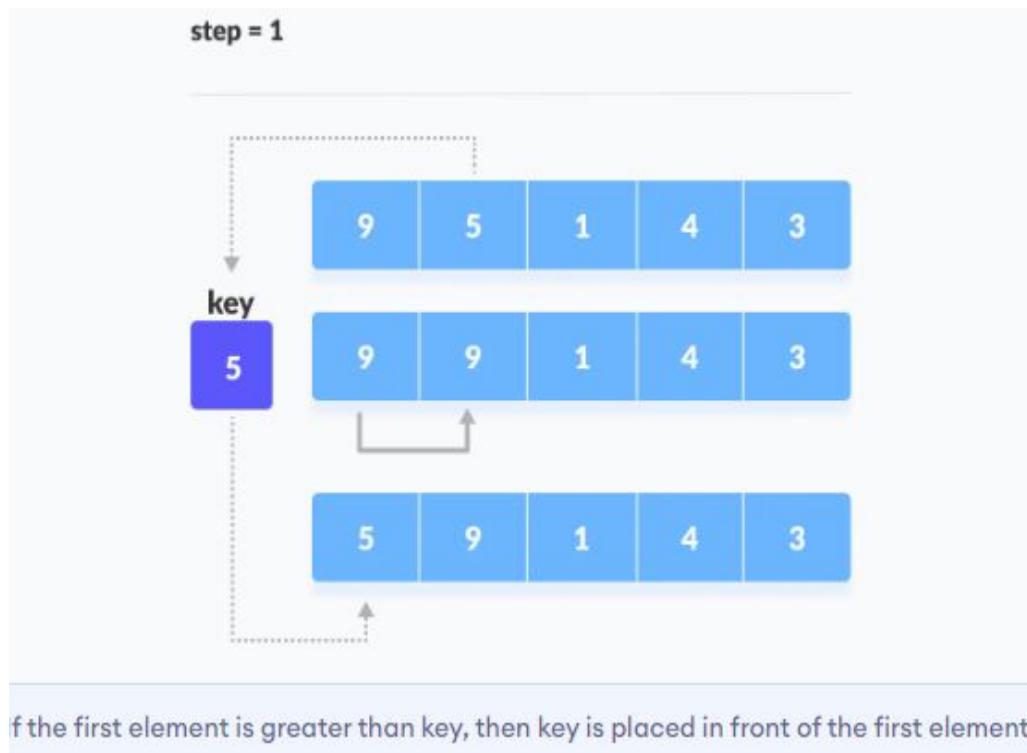
 move sorted element to the right by 1

 break loop and insert X here

 end insertionSort

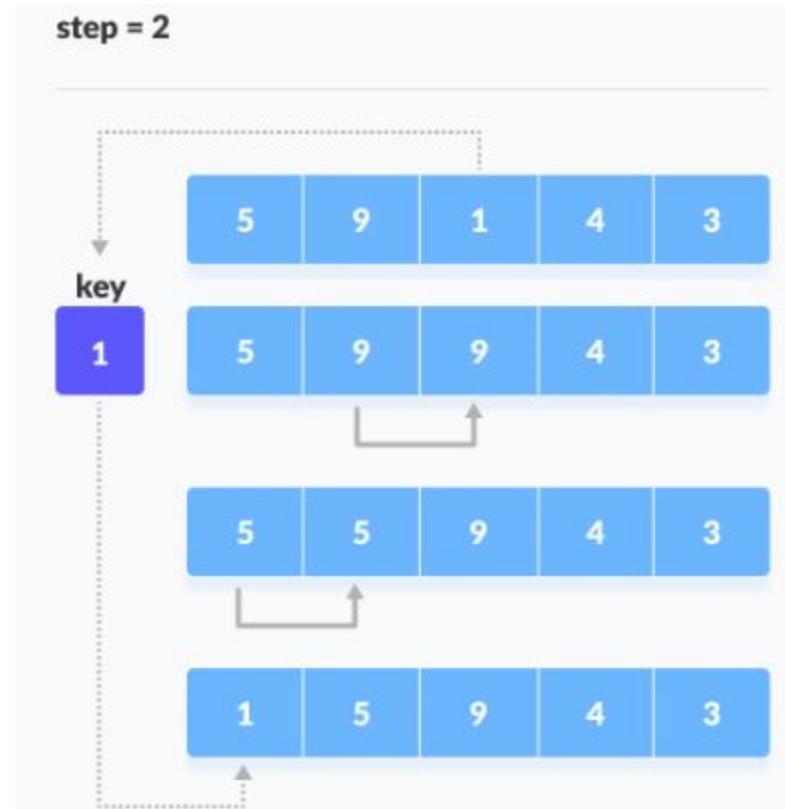
Insertion Sort

- The first element in the array is assumed to be sorted. Take the second element and store it separately in key.
- Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.



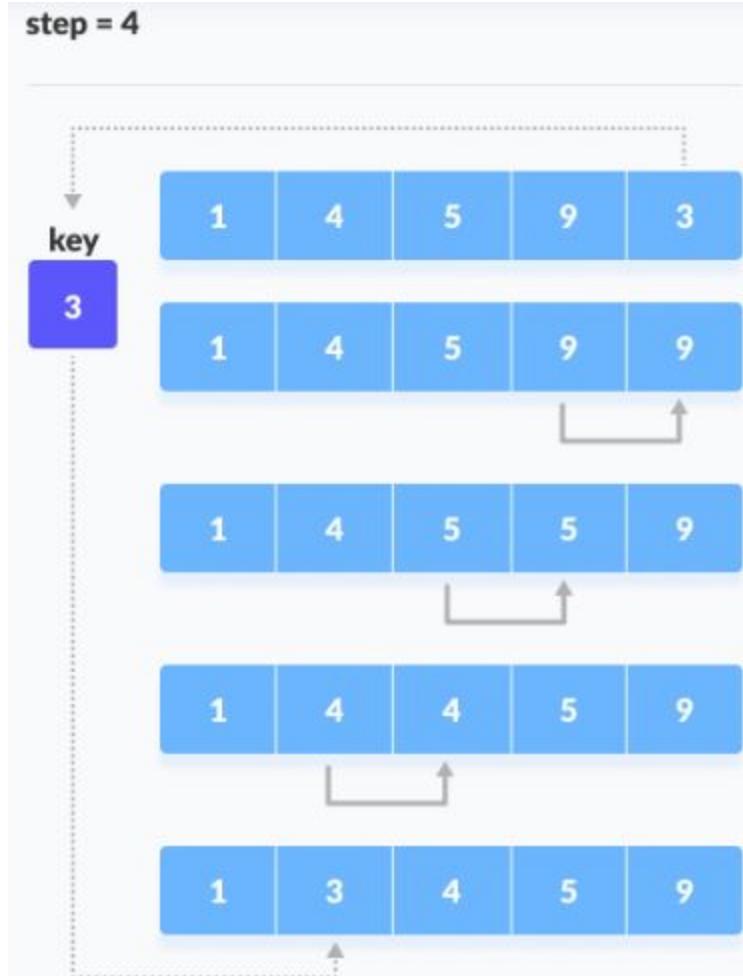
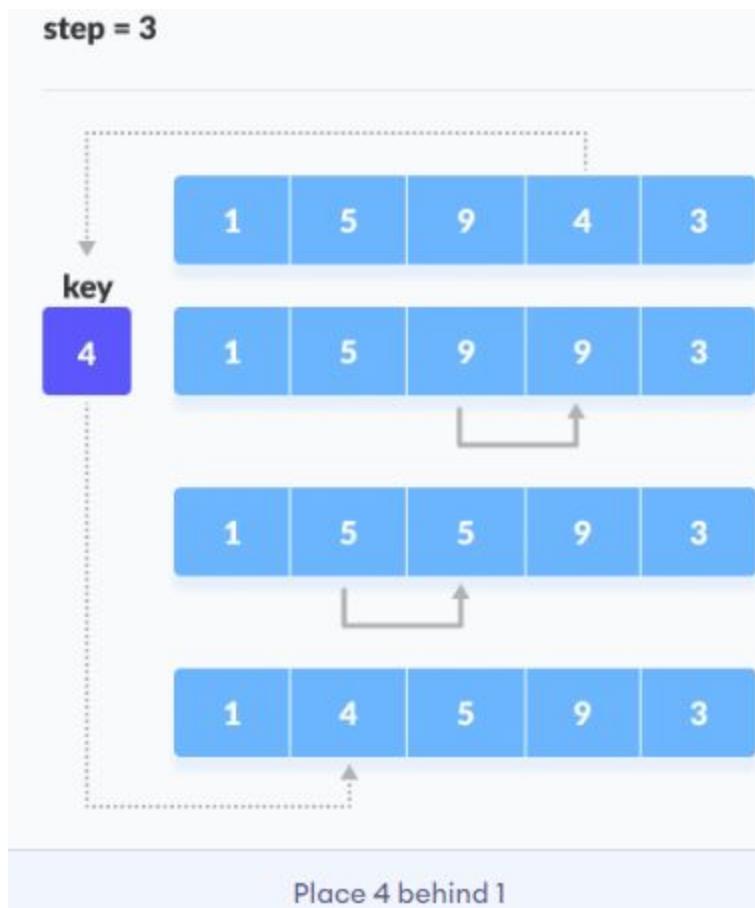
Insertion Sort

- Now, the first two elements are sorted.
- Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.



Insertion Sort

- Similarly, place every unsorted element at its correct position



Bubble Sort

- Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are not in the intended order

```
bubbleSort(array)
```

```
    for i <- 1 to indexOfLastUnsortedElement-1
```

```
        if leftElement > rightElement
```

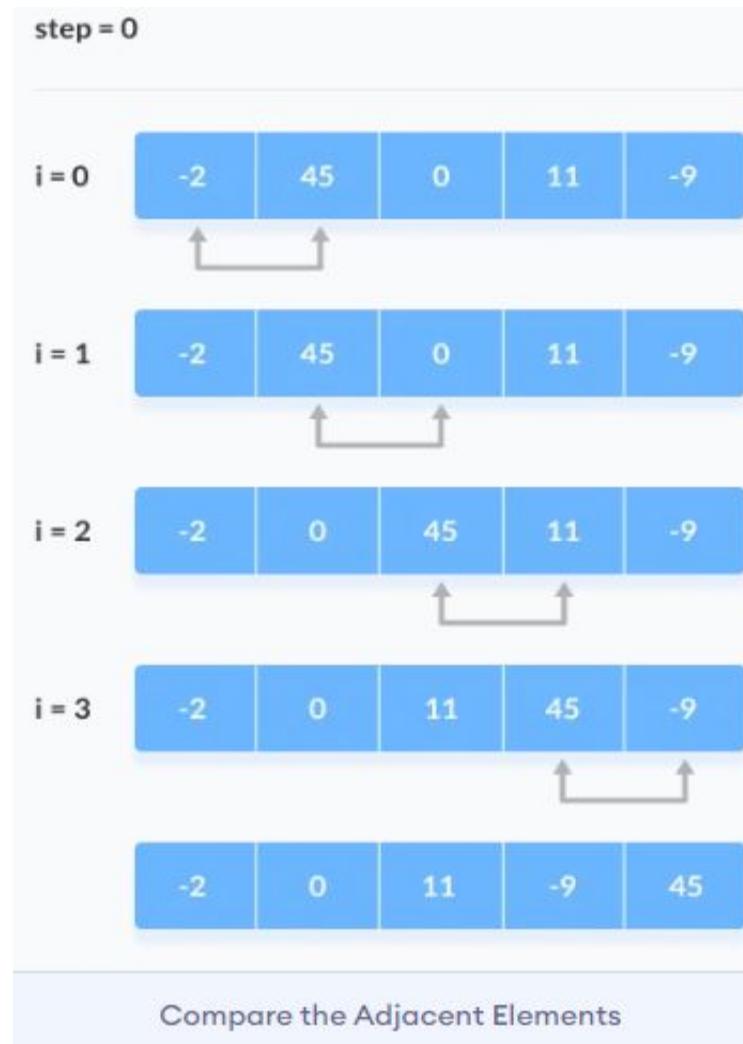
```
            swap leftElement and rightElement
```

```
    end bubbleSort
```

Bubble Sort

1. First Iteration (Compare and Swap)

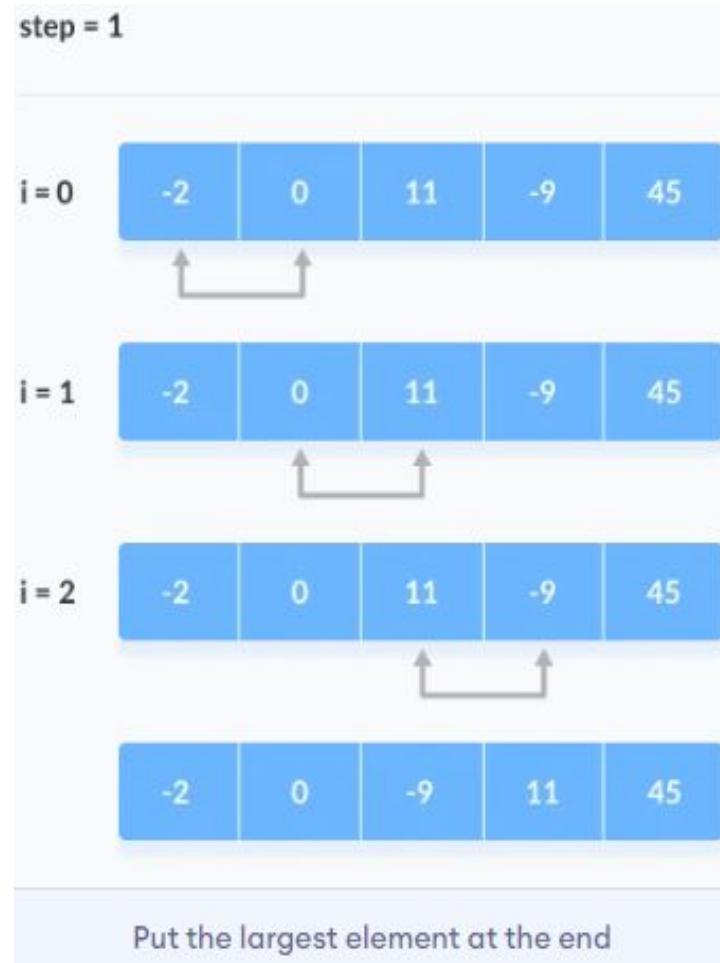
- Starting from the first index, compare the first and the second elements.
- If the first element is greater than the second element, they are swapped.
- Now, compare the second and the third elements. Swap them if they are not in order.
- The above process goes on until the last element



Bubble Sort

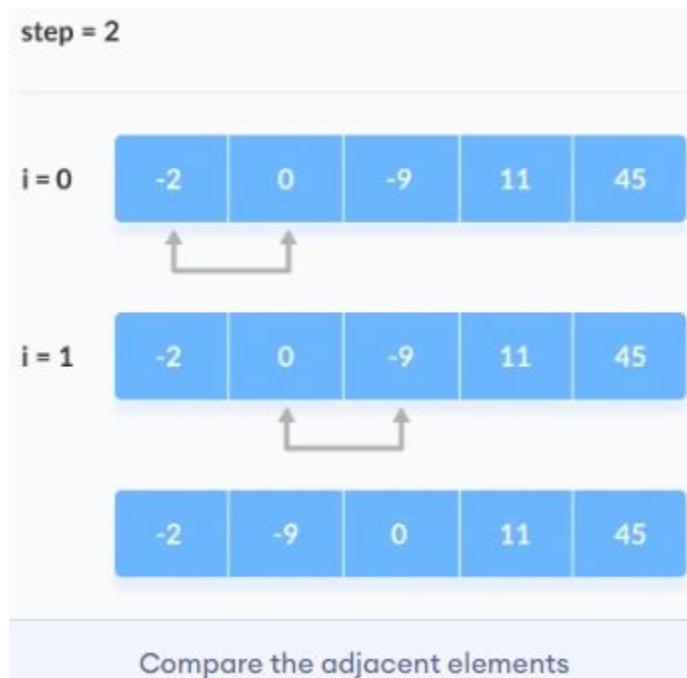
2. Remaining Iteration

- The same process goes on for the remaining iterations.
- After each iteration, the largest element among the unsorted elements is placed at the end.



Bubble Sort

- In each iteration, the comparison takes place up to the last unsorted element
- The array is sorted when all the unsorted elements are placed at their correct positions.



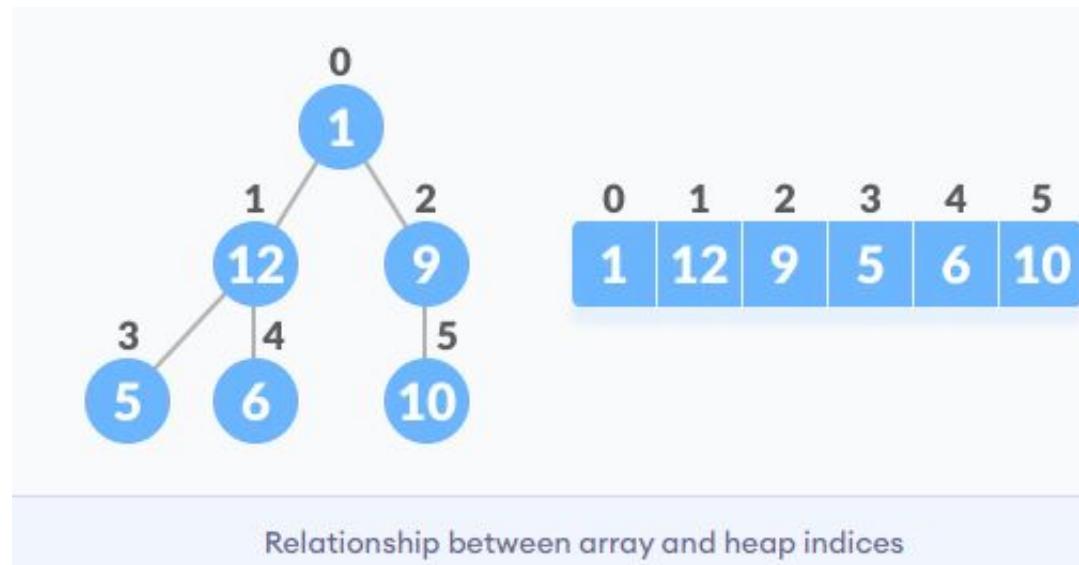
The array is sorted if all elements are kept in the right order

Heap Sort

- Heap Sort is a popular and efficient sorting algorithm in computer programming. Learning how to write the heap sort algorithm requires knowledge of two types of data structures - arrays and trees.
- The initial set of numbers that we want to sort is stored in an array e.g. [10, 3, 76, 34, 23, 32] and after sorting, we get a sorted array [3,10,23,32,34,76].
- Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called a heap.

Heap Sort

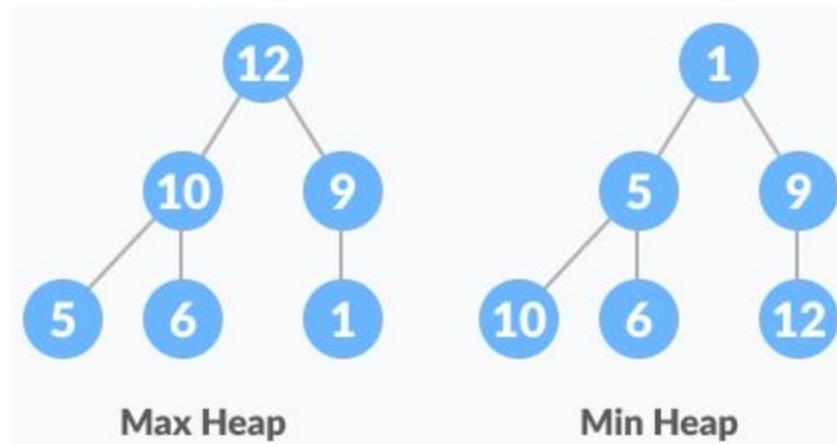
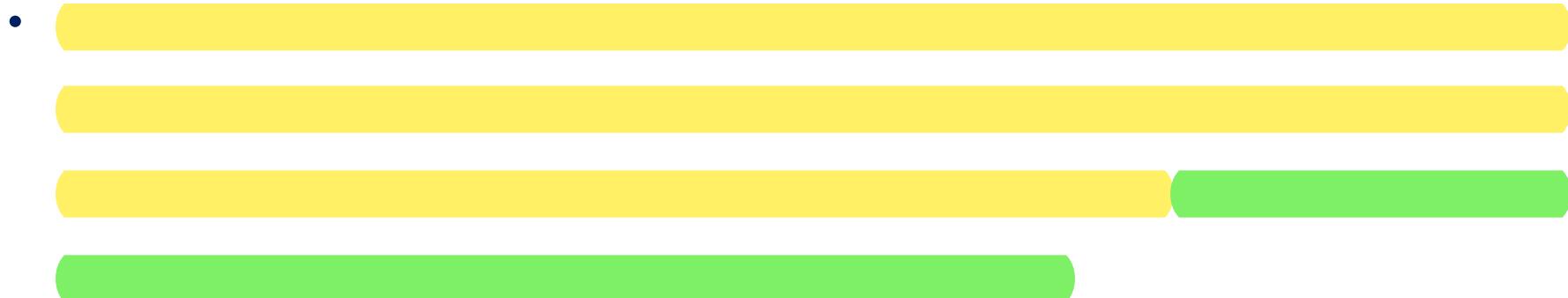
- A complete binary tree has an interesting property that can find the children and parents of any node.
- If the index of any element in the array is i, the element in the index $2i+1$ will become the left child and element in $2i+2$ index will become the right child. Also, the parent of any element at index i is given by the lower bound of $(i-1)/2$.



Heap Sort

Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if

- it is a complete binary tree



Heap Sort

heapify(array)

Root = array[0]

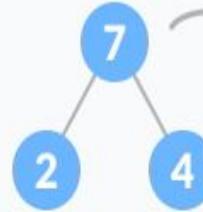
Largest = largest(array[0] , array [2*0 + 1]. array[2*0+2])

if(Root != Largest)

Swap(Root, Largest)

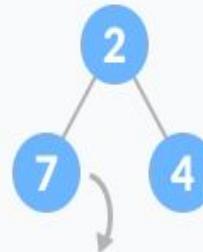
- Two scenarios - one in which the root is the largest element and don't need to do anything. And another in which the root had a larger element as a child and needed to swap to maintain max-heap property.

Scenario-1

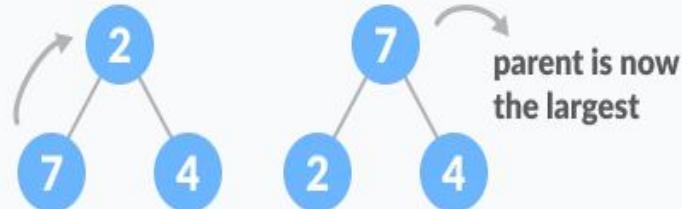


parent is already
the largest

Scenario-2



child is greater
than the parent

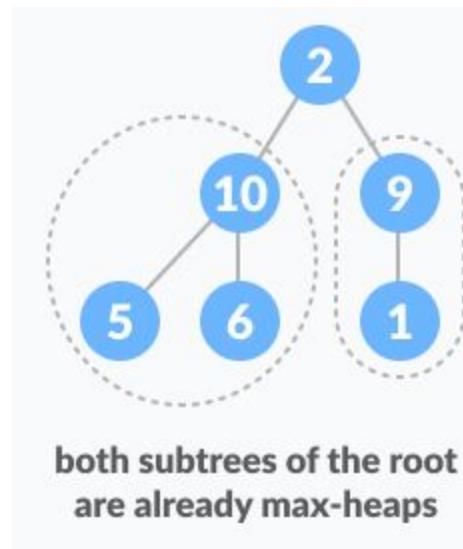


parent is now
the largest

Heapify base cases

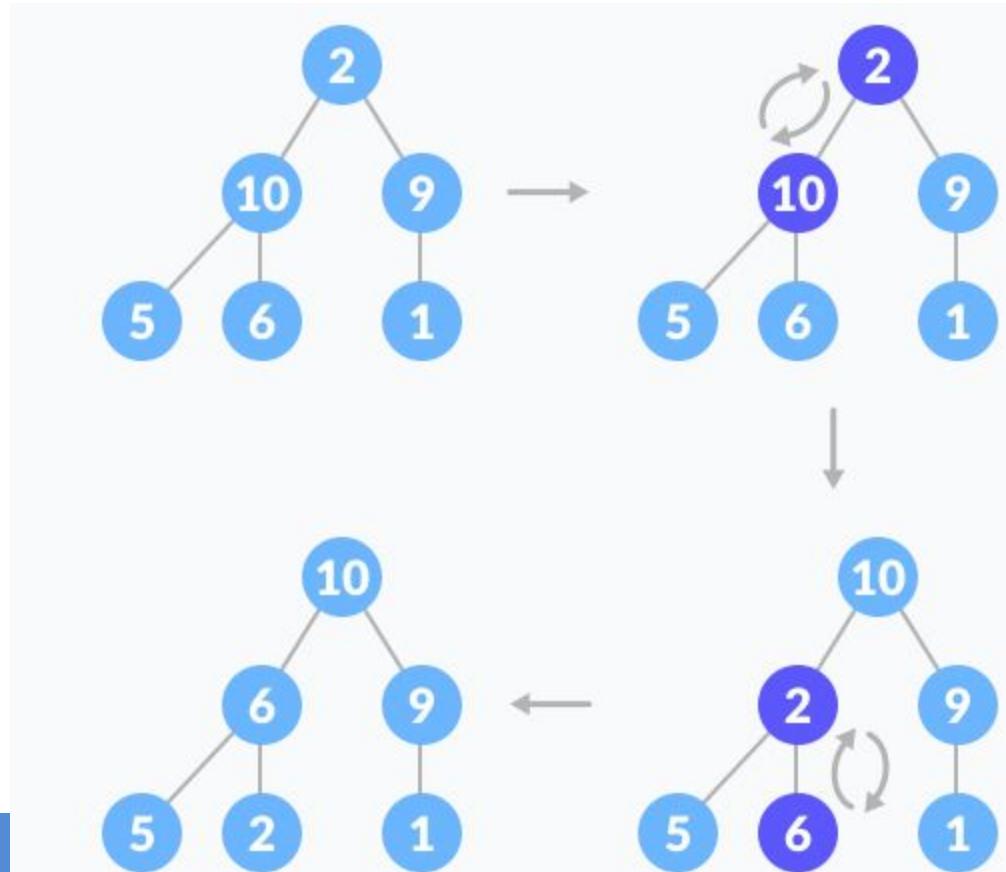
Heap Sort

- Now let's think of another scenario in which there is more than one level.



Heap Sort

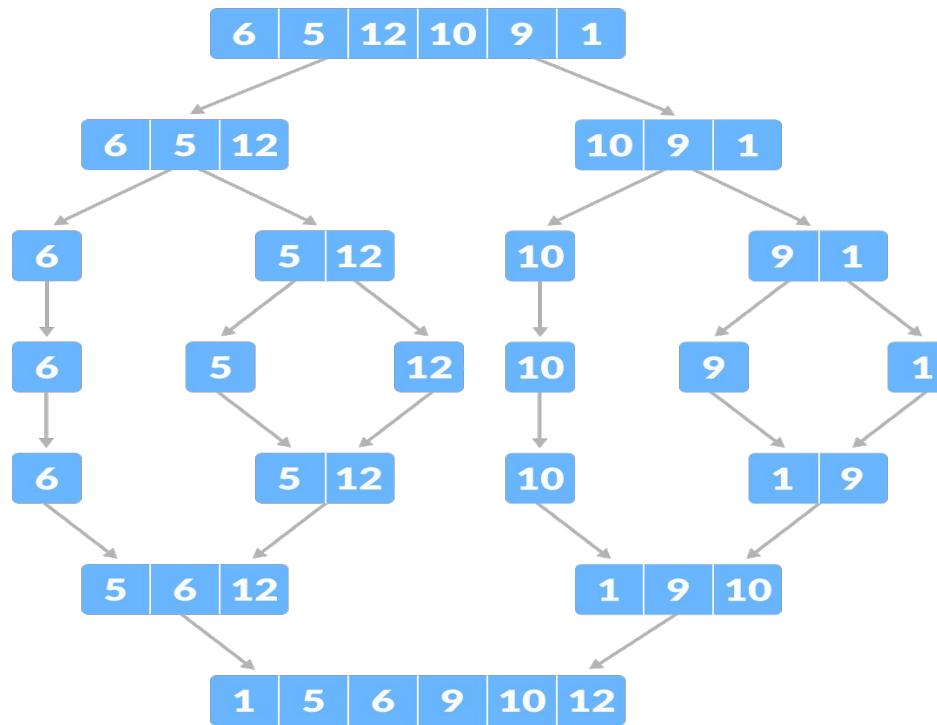
- The top element isn't a max-heap but all the sub-trees are max-heaps.
- To maintain the max-heap property for the entire tree, we will have to keep pushing 2 downwards until it reaches its correct position.



for detail explanation look into note book

Merge Sort

- Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.
- Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.



Merge Sort

- Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.
- Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.

Have we reached the end of any of the arrays?

No:

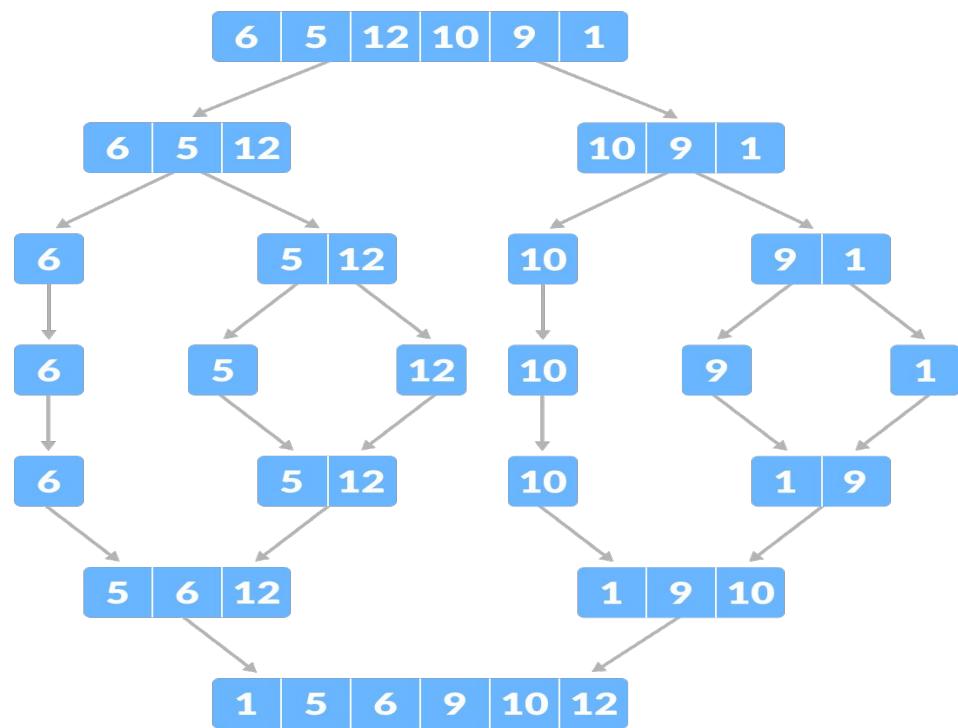
Compare current elements of both arrays

Copy smaller element into sorted array

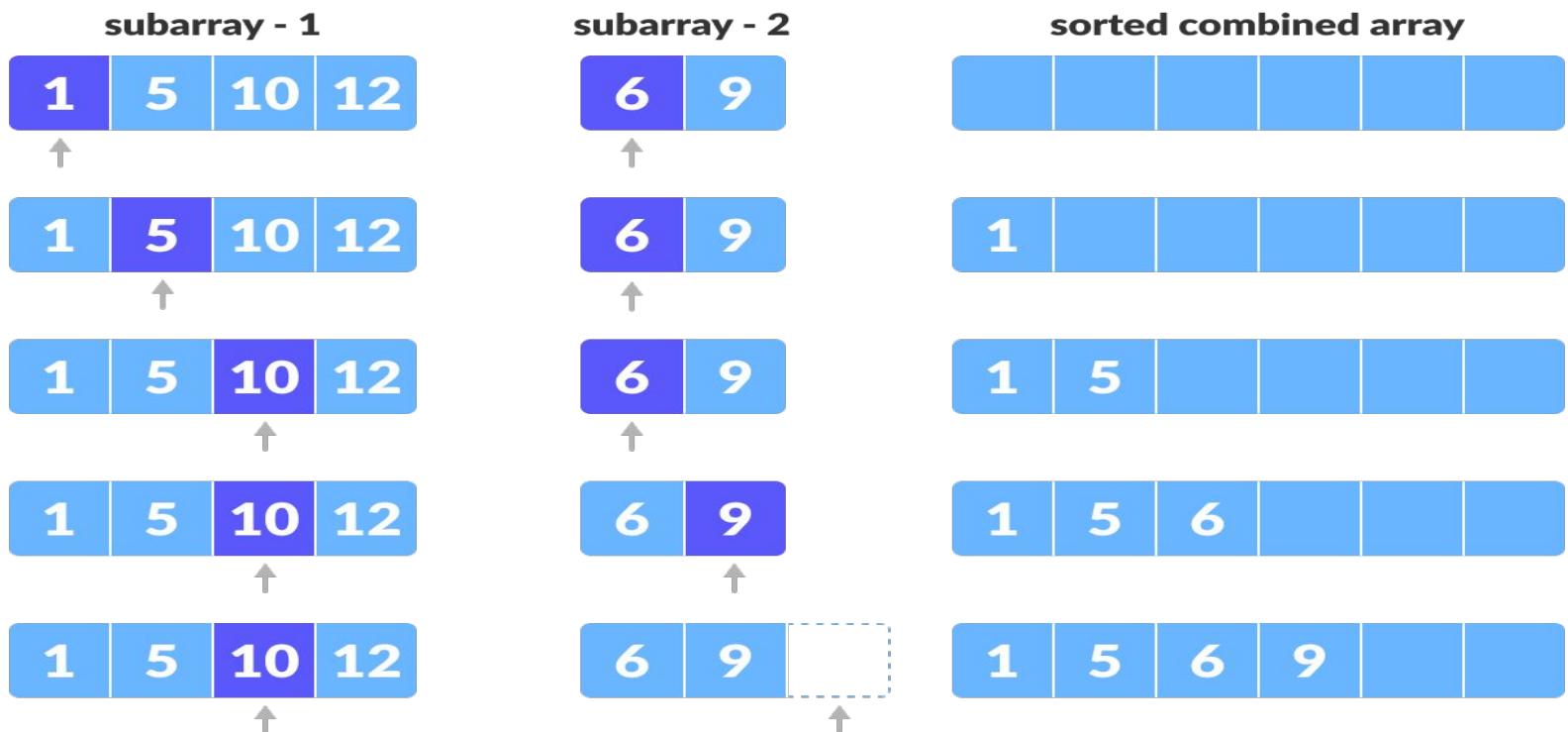
Move pointer of element containing smaller element

Yes:

Copy all remaining elements of non-empty array



Merge Sort



Since there are no more elements remaining in the second array, and we know that both the arrays were sorted when we started, we can copy the remaining elements from the first array directly.



Quick Sort

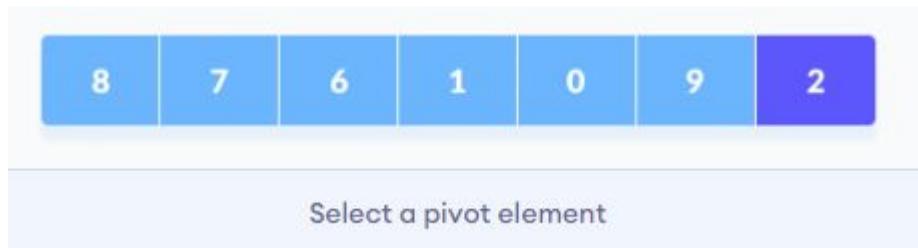
Quicksort is based on the divide and conquer approach where

- An array is divided into subarrays by selecting a pivot element (element selected from the array).
-
-
-
-
-
-

Quick Sort

1. Select the Pivot Element

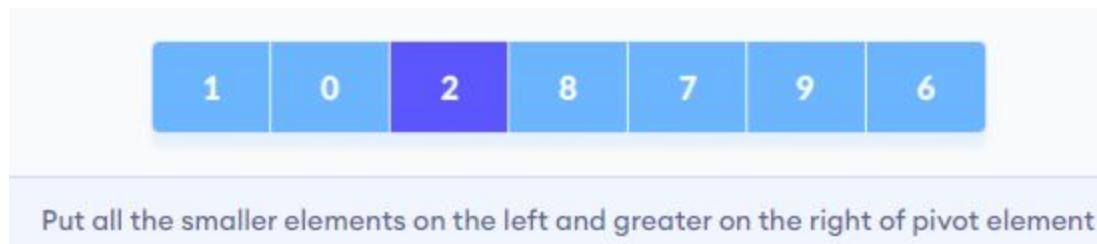
There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.



Quick Sort

2. Rearrange the Array

Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.



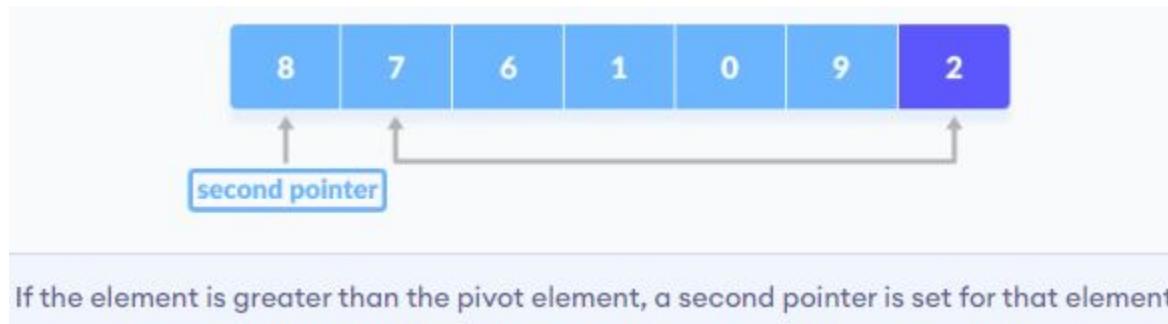
Here's how we rearrange the array:

A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.

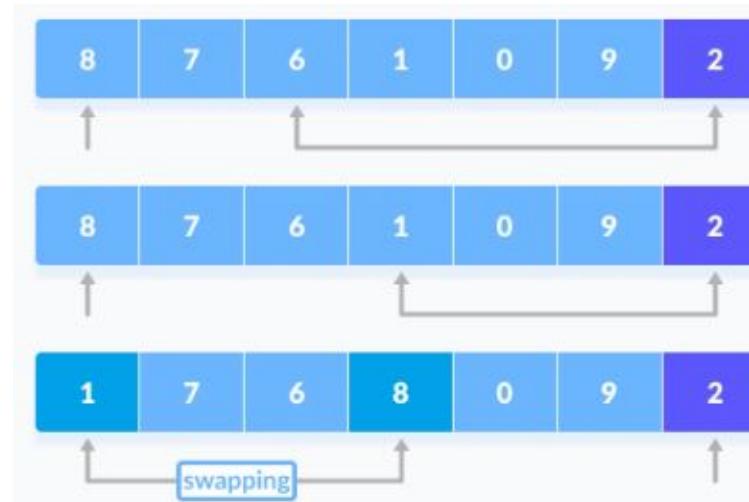


Quick Sort

If the element is greater than the pivot element, a second pointer is set for that element.



Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.



Pivot is compared with other elements.

Quick Sort

Again, the process is repeated to set the next greater element as the second pointer.
And, swap it with another smaller element.

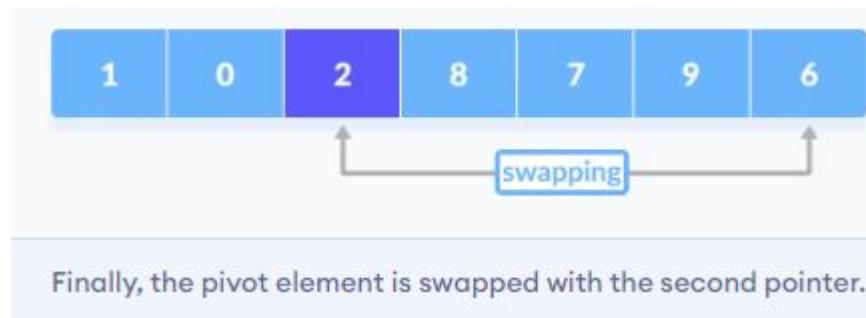


Quick Sort

The process goes on until the second last element is reached..



Finally, the pivot element is swapped with the second pointer.



Quick Sort

quicksort(arr, pi, high)

The positioning of elements after each call of partition algo





Thank you . . .

priyasajan@cdac.in



Hash Functions & Hash Tables

Hashing

- ▶ Hashing is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing $O(1)$ search time on average (under reasonable assumptions) and $O(n)$ in worst case.
- ▶ Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table.

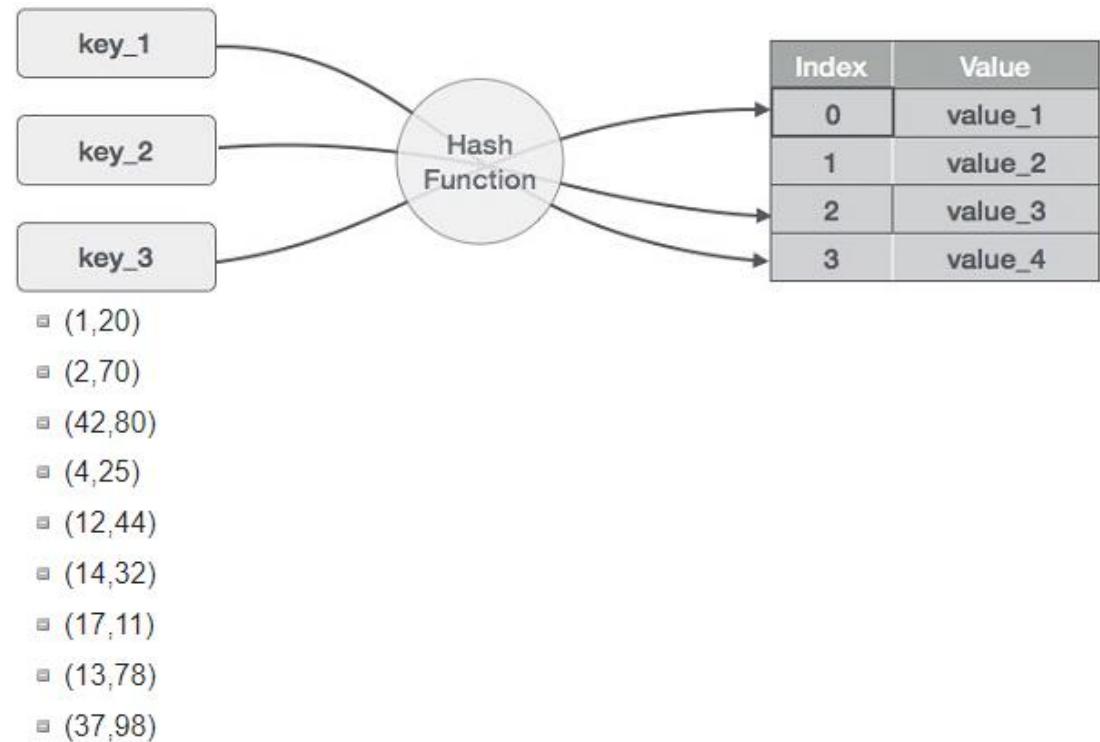
Hash Function

- ▶ A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.
- ▶ A good hash function should have following properties
 - 1) Efficiently computable.
 - 2) Should uniformly distribute the keys (Each table position equally likely for each key)
- ▶ For example for phone numbers a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

Hash Table

Hash Table:

- ▶ An array that stores pointers to records corresponding to a given phone number.
- ▶ An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.



Types of Hash functions

- ▶ Division Method.
- ▶ Mid Square Method.
- ▶ Folding Method.
- ▶ Multiplication Method.

Division Method

- ▶ This is the most simple and easiest method to generate a hash value. The hash function divides the value k by M and then uses the remainder obtained.
- ▶ Formula: $h(K) = k \bmod M$
Here, k is the key value, and M is the size of the hash table.
- ▶ It is best suited that M is a prime number as that can make sure the keys are more uniformly distributed. The hash function is dependent upon the remainder of a division.
- ▶ Example:

$$k = 12345$$

$$M = 95$$

$$\begin{aligned}h(12345) &= 12345 \bmod 95 \\&= 90\end{aligned}$$

Division Method

Pros:

- ▶ This method is quite good for any value of M.
- ▶ The division method is very fast since it requires only a single division operation.

Cons:

- ▶ This method leads to poor performance since consecutive keys map to consecutive hash values in the hash table.
- ▶ Sometimes extra care should be taken to chose value of M.

Mid Square Method

- It involves two steps to compute the hash value-
 - 1) Square the value of the key k i.e. k^2
 - 2) Extract the middle r digits as the hash value.
- Formula: $h(K) = h(k \times k)$

Here, k is the key value. The value of k is based on the size of the table.

- Example:

Suppose the hash table has 100 memory locations. So $r = 2$ because two digits are required to map the key to the memory location.

$$k = 60$$

$$\begin{aligned}k \times k &= 60 \times 60 \\&= 3600\end{aligned}$$

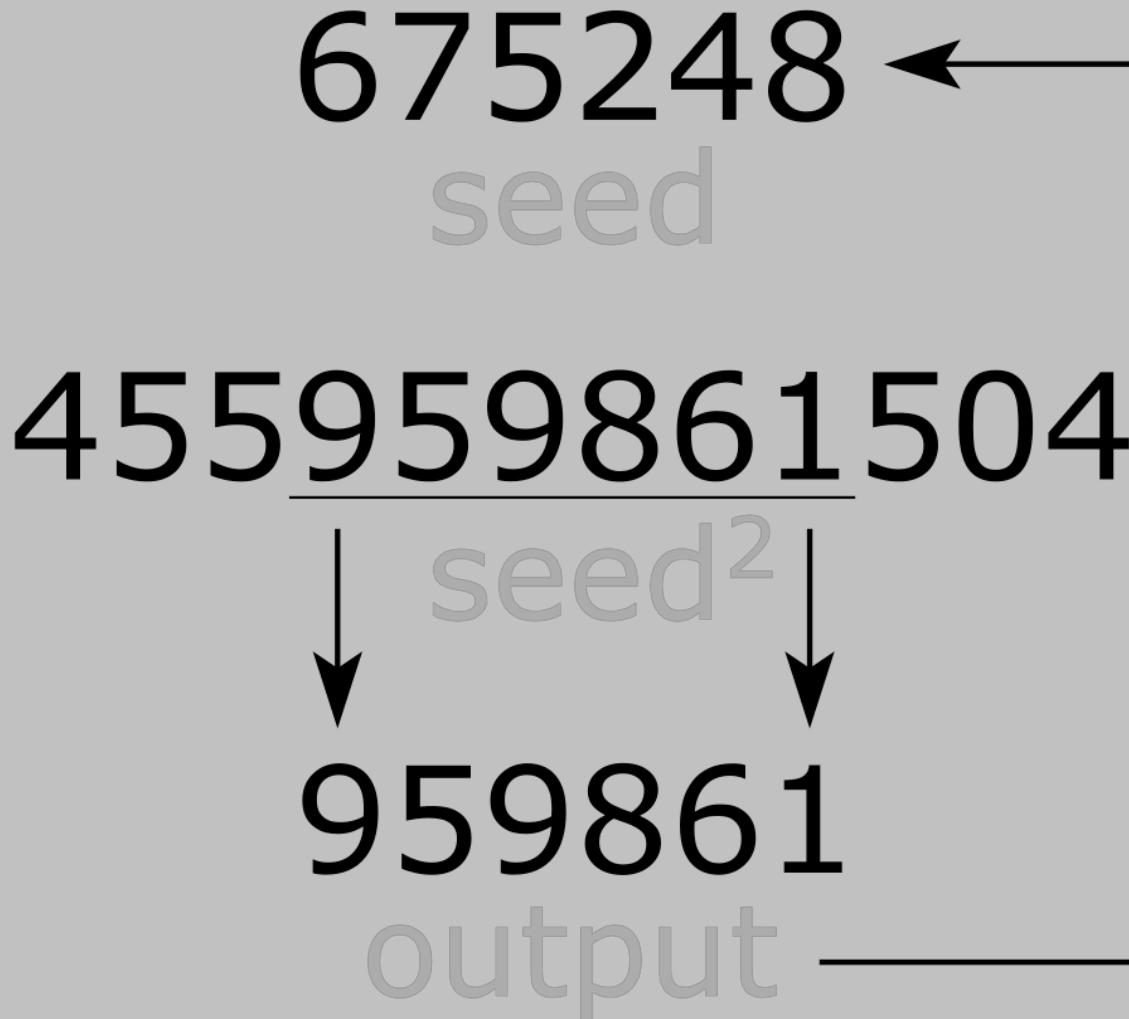
$$h(60) = 60$$

- The hash value obtained is 60

Mid Square Method - Example

- ▶ Suppose a 4-digit seed is taken. seed = 4765
- ▶ Hence, square of seed is = $4765 * 4765 = 22705225$
- ▶ Now, from this 8-digit number, any four digits are extracted (Say, the middle four).
- ▶ So, the new seed value becomes seed = 7052
- ▶ Now, square of this new seed is = $7052 * 7052 = 49730704$
- ▶ Again, the same set of 4-digits is extracted.
- ▶ So, the new seed value becomes seed = 7307
- ▶ This process is repeated as many times as a key is required.

Mid Square Method - Example



Mid Square Method

Pros:

- ▶ The performance of this method is good as most or all digits of the key value contribute to the result. This is because all digits in the key contribute to generating the middle digits of the squared result.
- ▶ The result is not dominated by the distribution of the top digit or bottom digit of the original key value.

Cons:

- ▶ The size of the key is one of the limitations of this method, as the key is of big size then its square will double the number of digits.
- ▶ Another disadvantage is that there will be collisions but we can try to reduce collisions.

Digit Folding Method

► This method involves two steps:

1. Divide the key-value k into a number of parts i.e. $k_1, k_2, k_3, \dots, k_n$, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.
2. Add the individual parts. Hash value is obtained by ignoring last carry if any.

► Formula:

$$k = k_1, k_2, k_3, k_4, \dots, k_n$$

$$s = k_1 + k_2 + k_3 + k_4 + \dots + k_n$$

$$h(K) = s$$

► Here, s is obtained by adding the parts of the key k

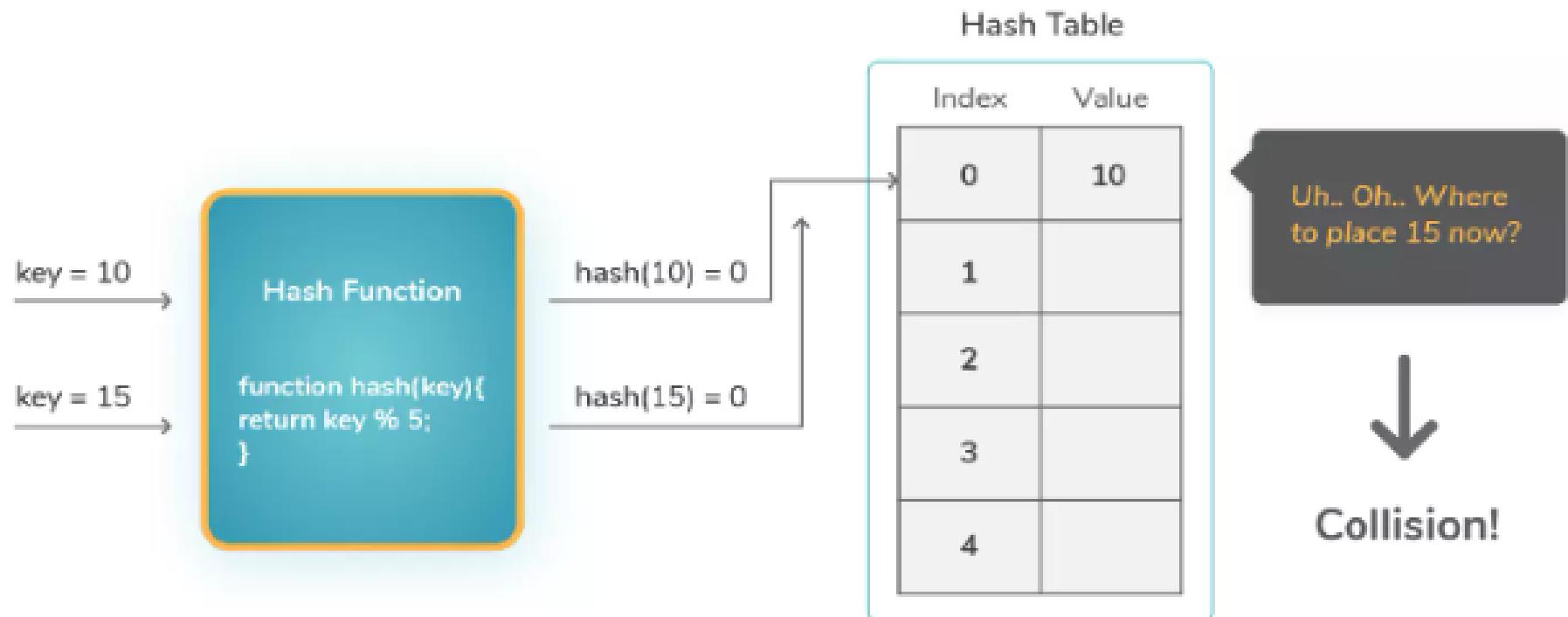
► Example:

$$k = 12345; k_1 = 12, k_2 = 34, k_3 = 5$$

$$s = k_1 + k_2 + k_3$$

$$= 12 + 34 + 5 = 51 \text{ ie } h(K) = 51$$

Collision Resolution



Collision Resolution

1) Separate chaining:-

- ▶ This technique creates a linked list to the slot for which collision occurs.
- ▶ The new key is then inserted in the linked list.
- ▶
- ▶ That is why, this technique is called as separate chaining.
- ▶
- ▶ Data type for storage :- `LinkedList[] Table; Table = new LinkedList(N), where N is the table size`

Collision Resolution

2) Open chaining (Probing)

- (i) **Linear probing**:- : When collision occurs, scan down the array one cell at a time looking for an empty cell.
- (ii) **Quadratic probing**:- when an incoming data's hash value indicates it should be stored in an already-occupied slot or bucket

Double Hashing

1. Take the key to store on the hash-table.
2. Apply the first hash function $h_1h1(key)$ over key to get the location to store the key.
3. If the location is empty, place the key on that location.
4. If the location is already filled, apply the second hash function $h_2h2(key)$ in combination with the first hash function $h_1h1(key)$ to get the new location for the key.

Double Hashing

Lets say, **Hash1 (key) = key % 13**

Hash2 (key) = 7 – (key % 7)

$$\text{Hash1}(19) = 19 \% 13 = 6$$

$$\text{Hash1}(27) = 27 \% 13 = 1$$

$$\text{Hash1}(36) = 36 \% 13 = 10$$

$$\text{Hash1}(10) = 10 \% 13 = 10$$

$$\text{Hash2}(10) = 7 - (10 \% 7) = 4$$

$$(\text{Hash1}(10) + 1 * \text{Hash2}(10)) \% 13 = 1$$

$$(\text{Hash1}(10) + 2 * \text{Hash2}(10)) \% 13 = 5$$

Collision

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

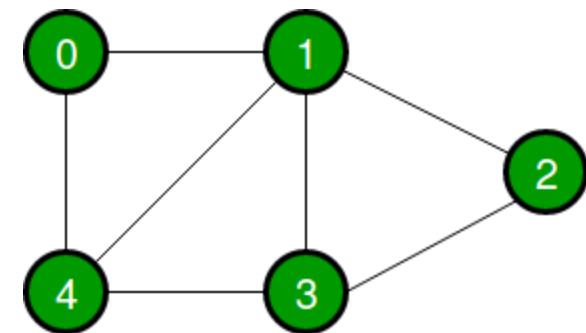
- Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array.
- Use linear probing to get the element ahead if an element is not found at the computed hash code.
- When found, store a dummy item there to keep the performance of the hash table intact.

Graphs

Graph

- ▶ A graph is a data structure that consists of the following two components:
 1. A finite set of vertices also called as nodes.
 2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v . The edges may contain weight/value/cost.

Following is an example of an undirected graph with 5 vertices.



Graph - terminology

Point

- ▶ A point is a particular position in a one-dimensional, two-dimensional, or three-dimensional space. It can be represented with a dot.
- ▶ Example



Line

- ▶ A Line is a connection between two points. It can be represented with a solid line.
- ▶ Example



Graph - terminology

Vertex

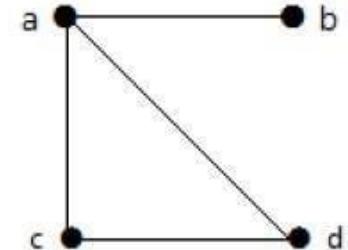
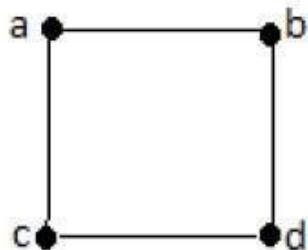
- A vertex is a point where multiple lines meet. It is also called a node. Similar to points, a vertex is also denoted by an alphabet.

Edge

- An edge is the mathematical term for a line that connects two vertices. Many edges can be formed from a single vertex. Without a vertex, an edge cannot be formed. There must be a starting vertex and an ending vertex for an edge

Graph

- A graph 'G' is defined as $G = (V, E)$ Where V is a set of all vertices and E is a set of all edges in the graph.



Graph - terminology

Loop

- In a graph, if an edge is drawn from vertex to itself, it is called a loop.

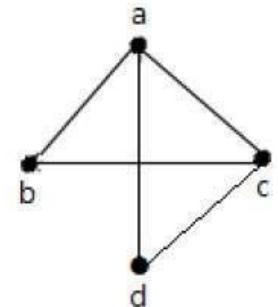
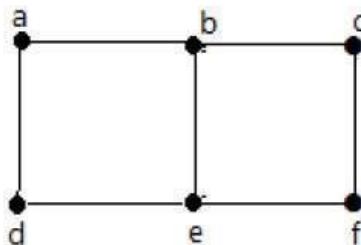


Degree of Vertex

- It is the number of vertices adjacent to a vertex V.

Adjacency

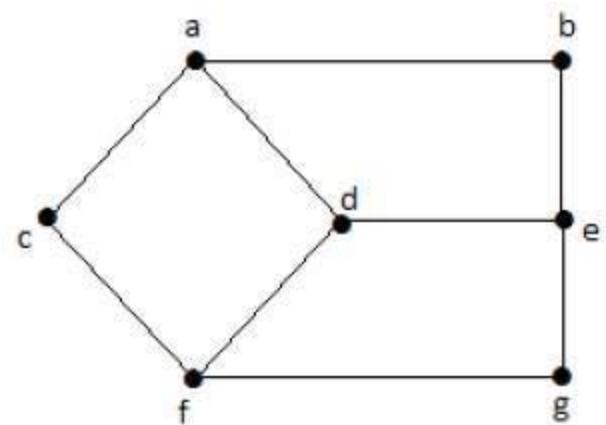
- In a graph, two edges are said to be adjacent, if there is a common vertex between the two edges. Here, the adjacency of edges is maintained by the single vertex that is connecting two edges.



Graph - terminology

Distance between Two Vertices

- It is number of edges in a shortest path between Vertex U and Vertex V.
- If there are multiple paths connecting two vertices, then the shortest path is considered as the distance between the two vertices.
- Notation – $d(U,V)$
- There can be any number of paths present from one vertex to other.
- Among those, choose only the shortest one.



Graph - terminology

Eccentricity of a Vertex

- ▶ The maximum distance between a vertex to all other vertices is considered as the eccentricity of vertex.

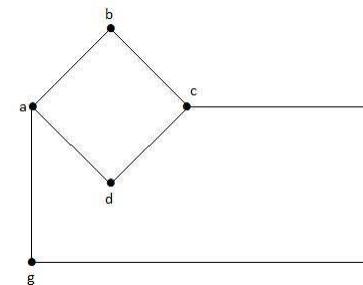
Notation – $e(V)$

- ▶ The distance from a particular vertex to all other vertices in the graph is taken and among those distances .

Graph - Types

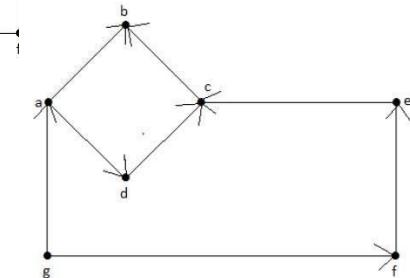
Null Graph

- A graph having no edges



Trivial Graph

- A graph with only one vertex is called a Trivial Graph.

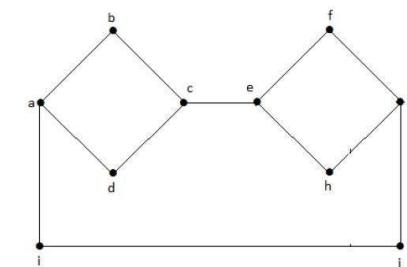


Non-Directed Graph

- A non-directed graph contains edges but the edges are not directed ones.

Directed Graph

- In a directed graph, each edge has a direction.



Connected Graph

- A graph G is said to be connected if there exists a path between every pair of vertices. There should be at least one edge for every vertex in the graph.

Simple Graph

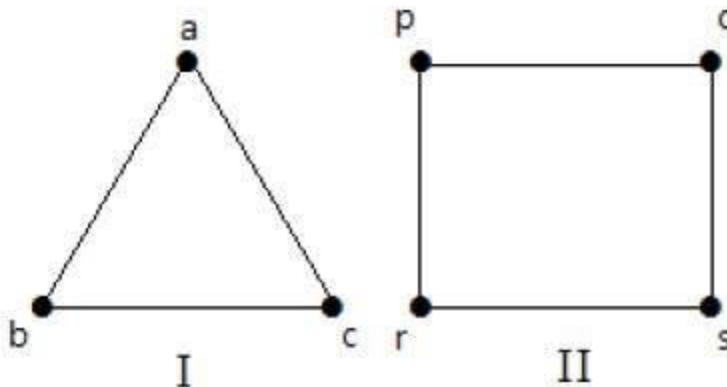
- A graph having no loops and no parallel edges

Disconnected Graph

- A graph G is disconnected, if it does not contain at least two connected vertices.

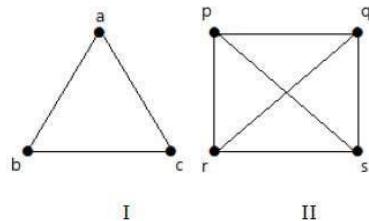
Regular Graph

- A graph G is said to be regular, if all its vertices have the same degree..



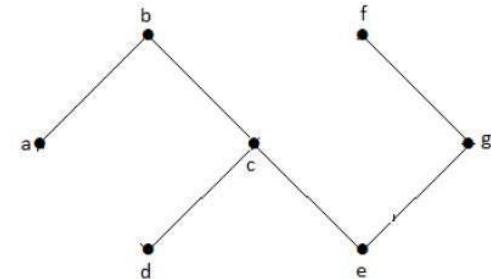
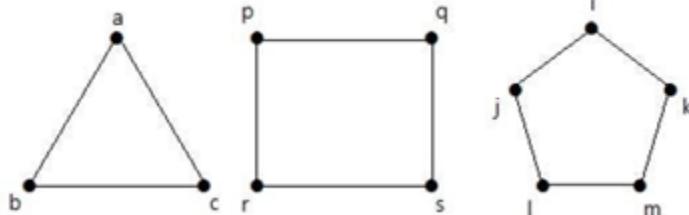
Complete Graph

- if a vertex is connected to all other vertices in a graph, then it is called a complete graph.



Cycle and Acycle Graph

- If the degree of each vertex in the graph is two, then it is called a Cycle Graph
-



Graph - Types

Tree

- A connected acyclic graph is called a tree

Forest

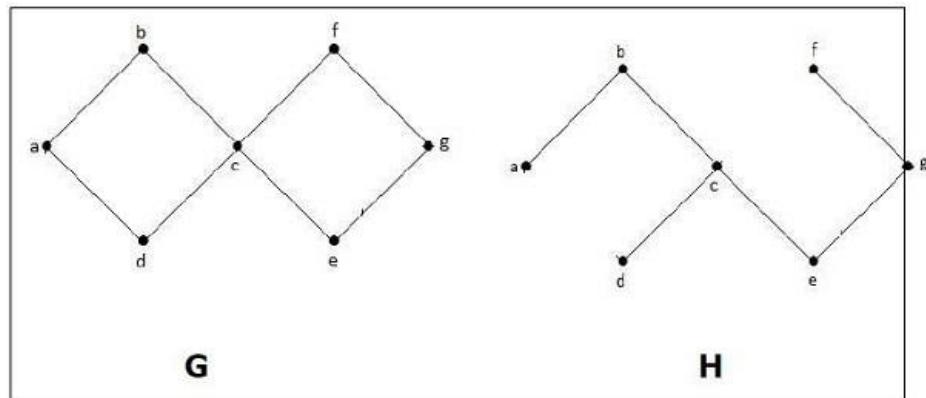
- A forest is a disjoint union of trees.

Spanning Trees

- Let G be a connected graph, then the sub-graph H of G is called a spanning tree of G if –

H is a tree

H contains all vertices of G.



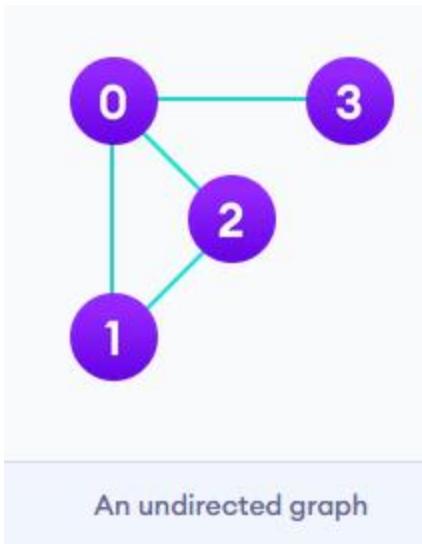
Graph Representation

- ▶ Adjacency Matrix
- ▶ Adjacency List

Graph Representation

Adjacency Matrix

- Each cell in the above table/matrix is represented as A_{ij} , where i and j are vertices. The value of A_{ij} is either 1 or 0 depending on whether there is an edge from vertex i to vertex j .



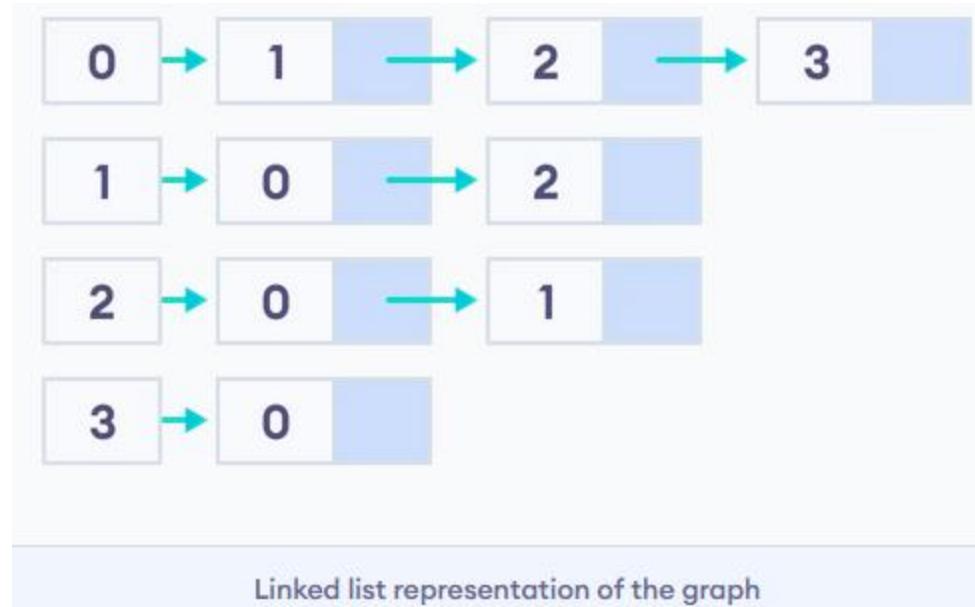
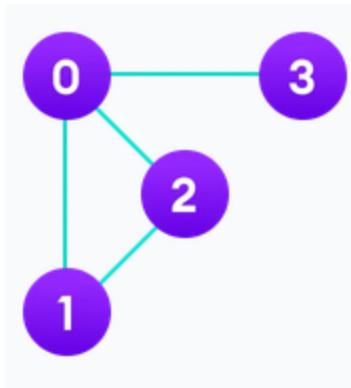
j \ i	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Matrix representation of the graph

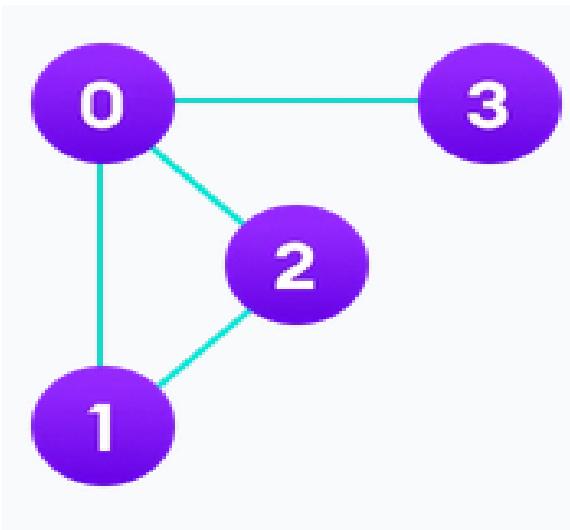
Graph Representation

Adjacency List

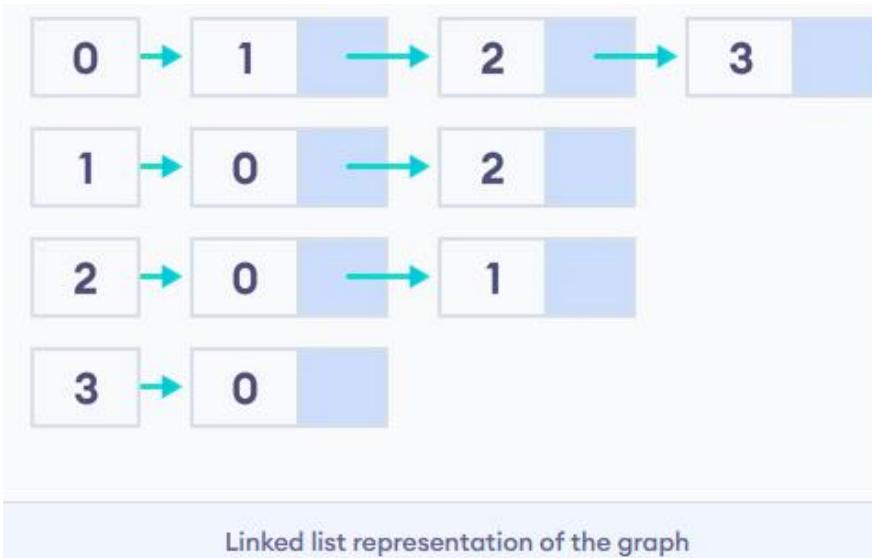
- ▶ An adjacency list represents a graph as an array of linked lists. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.



Graph Representation



i \ j	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0



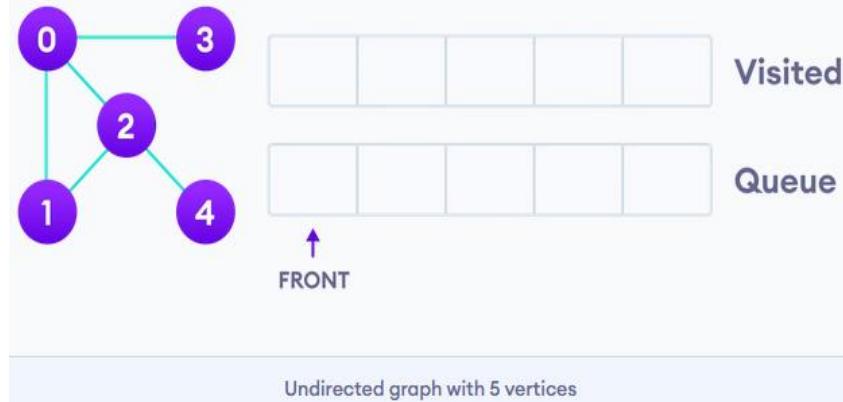
Matrix representation of the graph

Breadth First Search

- ▶ A standard BFS implementation puts each vertex of the graph into one of two categories: Visited and Not Visited
- ▶ The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
- ▶ Work flow
 - Start by putting any one of the graph's vertices at the back of a queue.
 - Take the front item of the queue and add it to the visited list.
 - Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
 - keep repeating steps 2 and 3 until the queue is empty.

Graph Traversal - Breadth First Search

- start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



- Next, visit element at the front of queue i.e. 1 and go to its adjacent nodes of 0. Since 0 has already been visited, visit 2 instead.

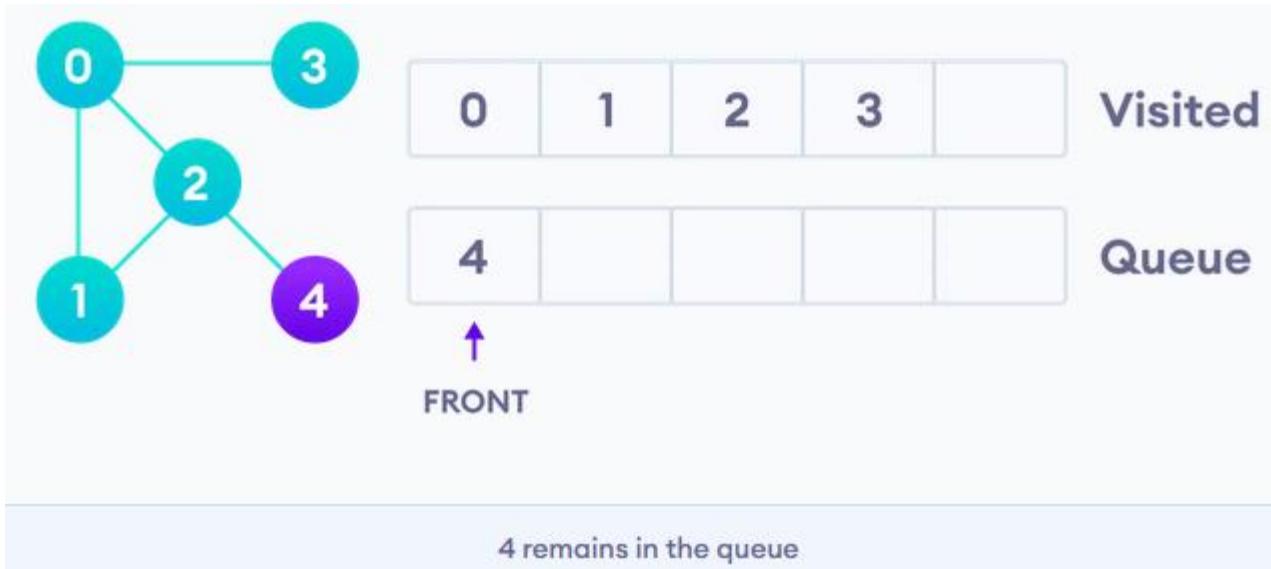


Graph Traversal - Breadth First Search



- Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.

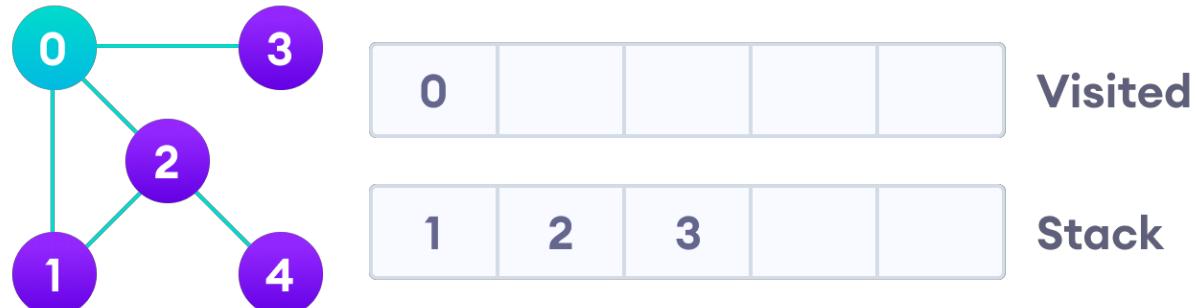
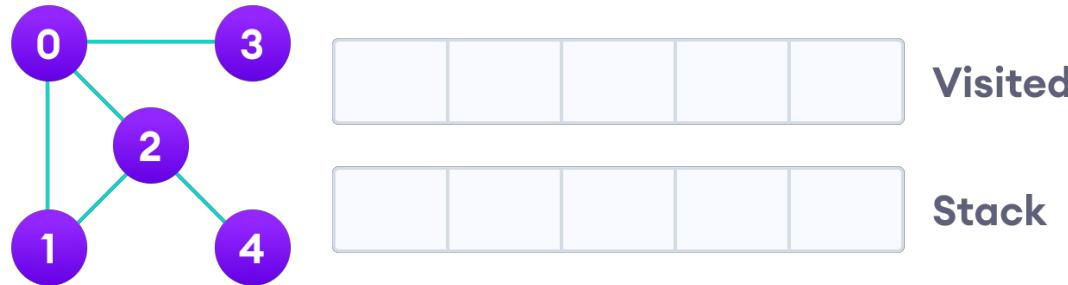
Graph Traversal - Breadth First Search



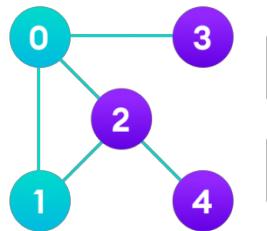
Graph Traversal - Breadth First Search



Graph Traversal – Depth First Search



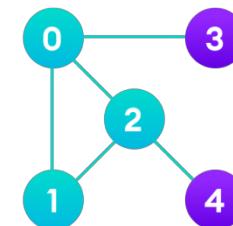
Graph Traversal – Depth First Search



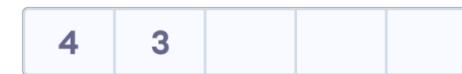
Visited



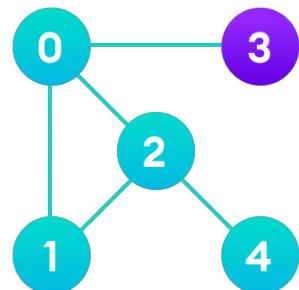
Stack



Visited



Stack

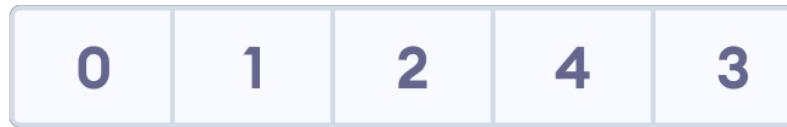
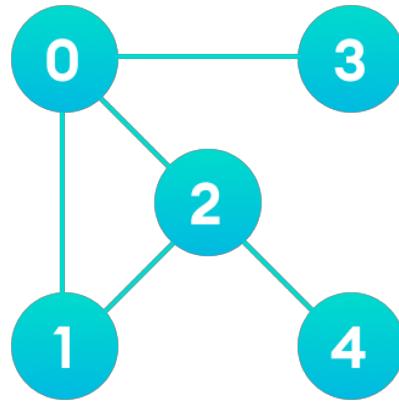


Visited



Stack

Graph Traversal – Depth First Search



Visited



Stack

Graph Traversal – Depth First Search

- Convert the following into Java

```
void DFS(vector <int >adj[],int s, bool visited[])
```

```
{   visited[s] = true;
```

```
cout<<" "<< s;
```

```
vector <int >::iterator it;
```

```
for(it = adj[s].begin(); it!= adj[s].end(); it++)
```

```
if(!visited[*it])
```

```
    DFS(adj,*it,visited);
```

```
}
```

```
void BFS(vector <int >adj[],int s,int n)
```

```
{   bool visited[n];
```

```
memset(visited,0, sizeof(visited));
```

```
visited[s] = 1;
```

```
queue <int >Q;
```

```
Q.push(s);
```

```
while(!Q.empty())
```

```
{
```

```
    int v = Q.front();
```

```
    Q.pop();
```

```
    cout<<" "<<v;
```

```
    vector <int >::iterator it;
```

```
    for(it = adj[v].begin(); it != adj[v].end(); it++)
```

```
        if(!visited[*it])
```

```
{
```

```
    visited[*it] = 1;
```

```
    Q.push(*it);
```

```
}
```

```
}
```

Graph Traversal – Depth First Search

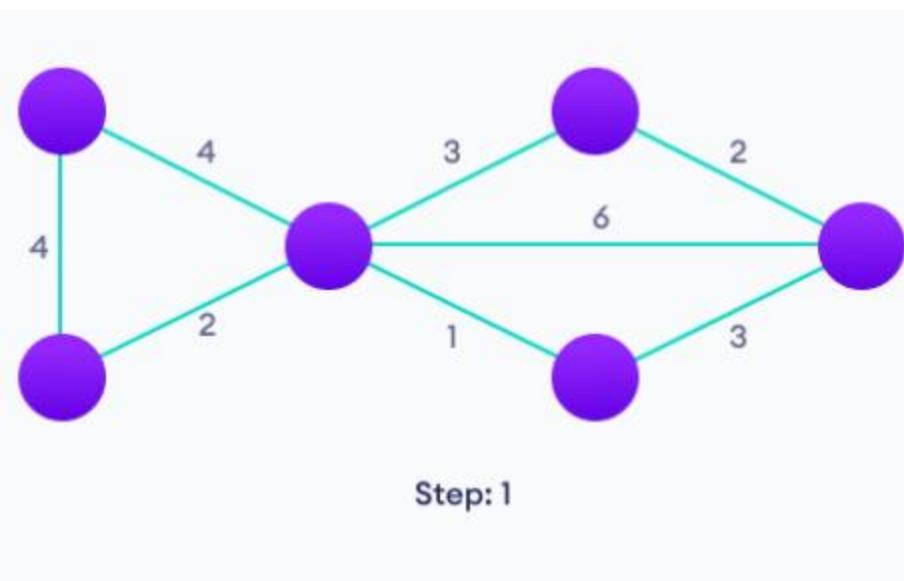
```
void addEdge(vector <int> adj[], int src, int dest)
{
    adj[src].push_back(dest);
}

int main(int argc, char const *argv[])
{
    int v = 5;
    vector<int> adj[v];
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 4);
    addEdge(adj, 1, 2);
    addEdge(adj, 4, 2);
    addEdge(adj, 2, 3);
    cout<<"DFS traversal = "<<endl;
    bool visited[v];
    memset(visited, 0, sizeof(visited));
    DFS(adj, 0, visited);
    cout<<endl<<endl;
    cout<<"BFS traversal = "<<endl;
    BFS(adj, 0, v);
    cout<<endl<<endl;
    return 0;
}
```

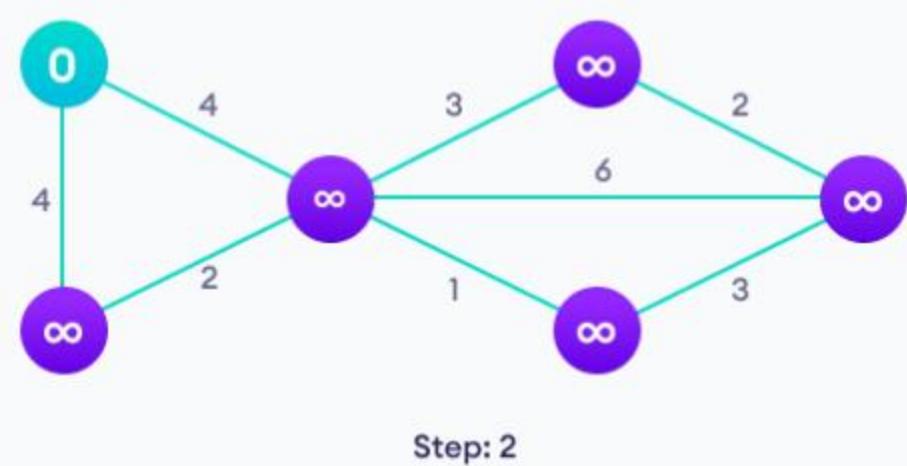
Shortest Path Problem - Dijkstra's algorithm

- ▶ Shortest path problem is a problem of finding the shortest path(s) between vertices of a given graph.
- ▶ Shortest path between two vertices is a path that has the least cost as compared to all other existing paths.
- ▶ Dijkstra's algorithm allows to find the shortest path between any two vertices of a graph.

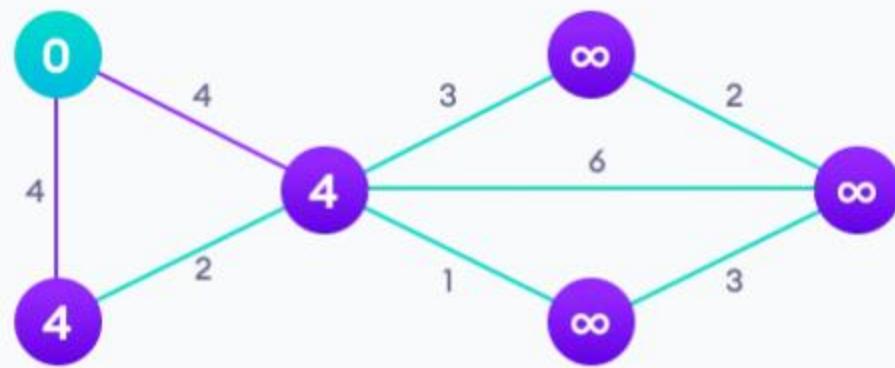
Dijkstra's algorithm



Start with a weighted graph

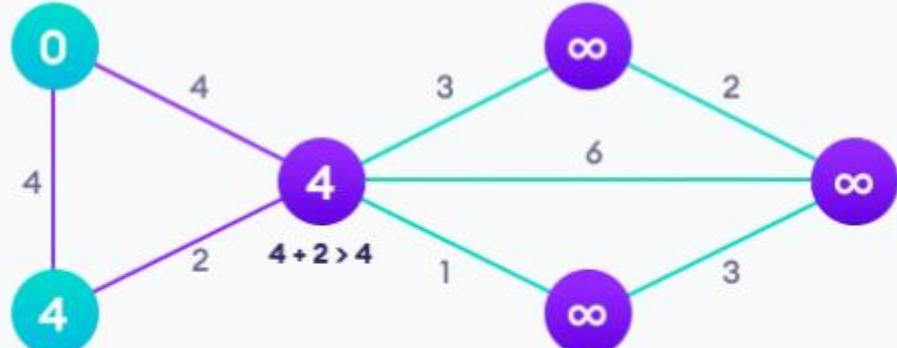


Dijkstra's algorithm



Step: 3

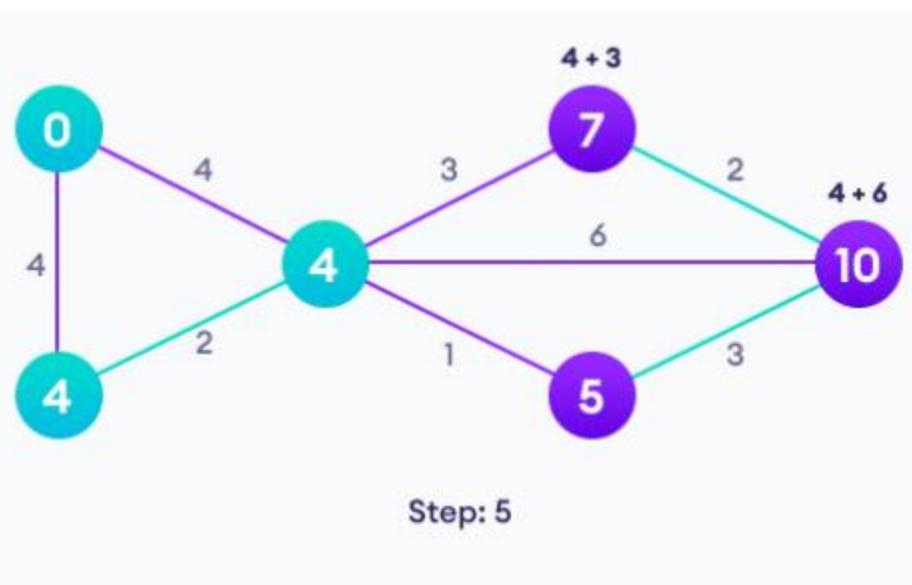
Go to each vertex and update its path length



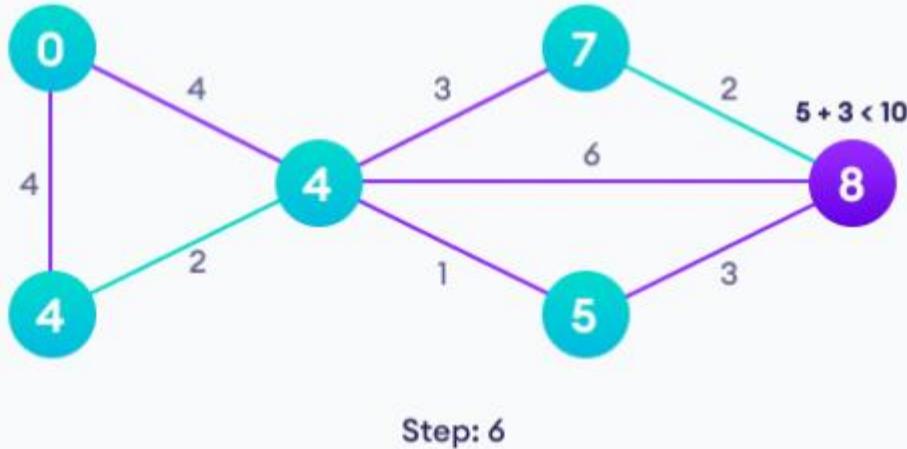
Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update

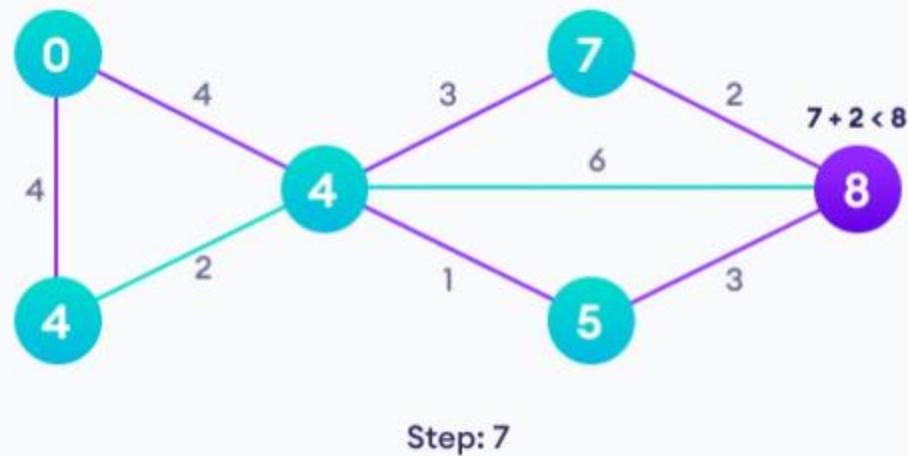
Dijkstra's algorithm



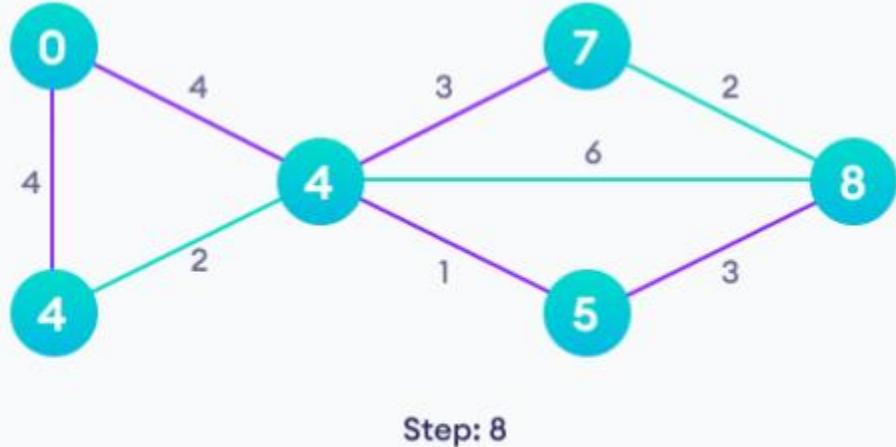
Avoid updating path lengths of c



Dijkstra's algorithm



Notice how the rightmost vertex has its path length updated twice



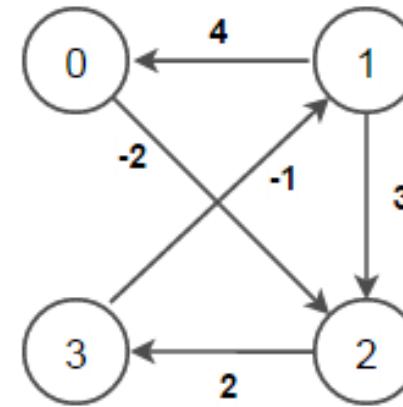
All-Pairs Shortest Paths – Floyd Warshall Algorithm

- ▶ Floyd–Warshall algorithm is an algorithm for finding the shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles).
- ▶ It does so by comparing all possible paths through the graph between each pair of vertices and that too with $O(V^3)$ comparisons in a graph.

All-Pairs Shortest Paths – Floyd Warshall Algorithm

Given a set of vertices V in a weighted graph where its edge weights $w(u, v)$ can be negative, find the shortest path weights $d(s, v)$ from every source s for all vertices v present in the graph. If the graph contains a negative-weight cycle, report it.

For example, consider the following graph:



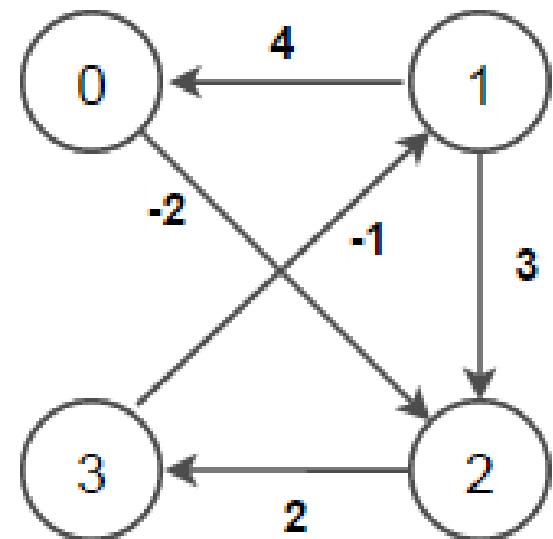
Adjacency matrix containing shortest distance is:

0	-1	-2	0
4	0	2	4
5	1	0	2
3	-1	1	0

All-Pairs Shortest Paths – Floyd Warshall Algorithm

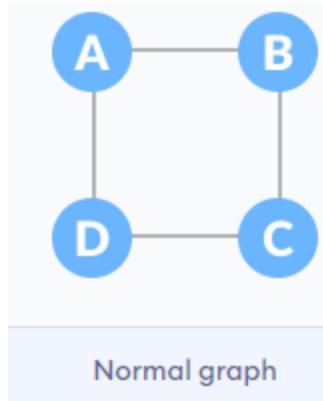
► The shortest path from:

- vertex 0 to vertex 1 is $[0 \rightarrow 2 \rightarrow 3 \rightarrow 1]$
- vertex 0 to vertex 2 is $[0 \rightarrow 2]$
- vertex 0 to vertex 3 is $[0 \rightarrow 2 \rightarrow 3]$
- vertex 1 to vertex 0 is $[1 \rightarrow 0]$
- vertex 1 to vertex 2 is $[1 \rightarrow 0 \rightarrow 2]$
- vertex 1 to vertex 3 is $[1 \rightarrow 0 \rightarrow 2 \rightarrow 3]$
- vertex 2 to vertex 0 is $[2 \rightarrow 3 \rightarrow 1 \rightarrow 0]$
- vertex 2 to vertex 1 is $[2 \rightarrow 3 \rightarrow 1]$
- vertex 2 to vertex 3 is $[2 \rightarrow 3]$
- vertex 3 to vertex 0 is $[3 \rightarrow 1 \rightarrow 0]$
- vertex 3 to vertex 1 is $[3 \rightarrow 1]$
- vertex 3 to vertex 2 is $[3 \rightarrow 1 \rightarrow 0 \rightarrow 2]$



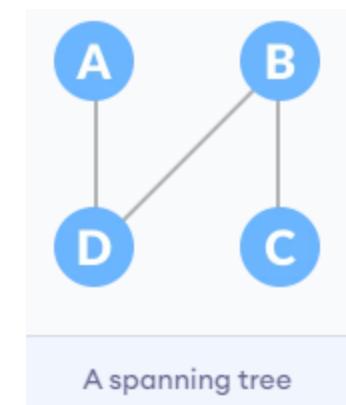
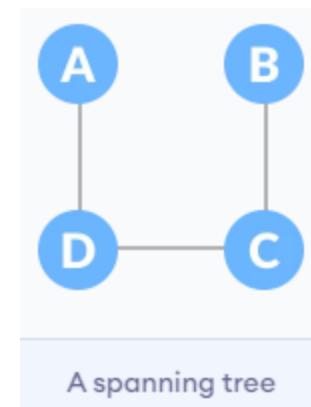
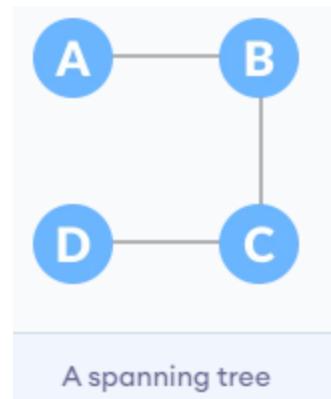
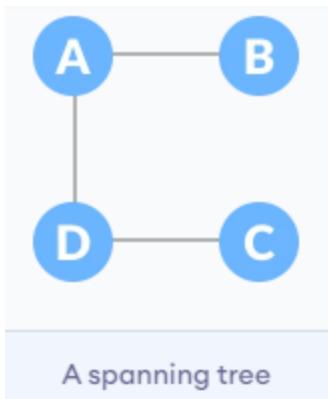
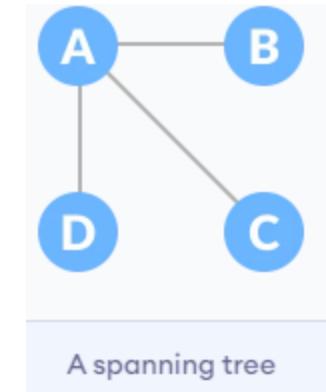
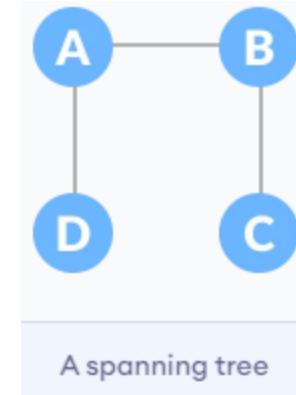
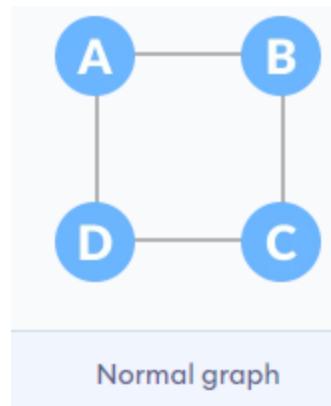
Spanning Trees

- A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.
- The edges may or may not have weights assigned to them.
- Example



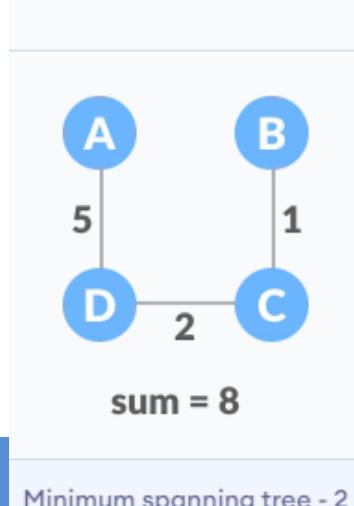
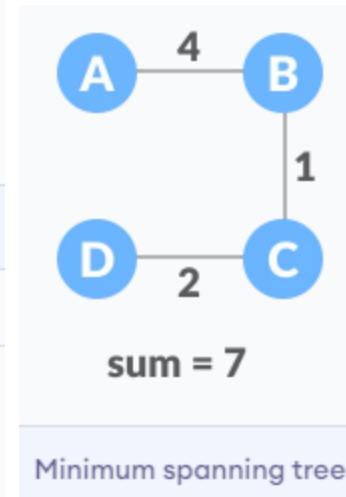
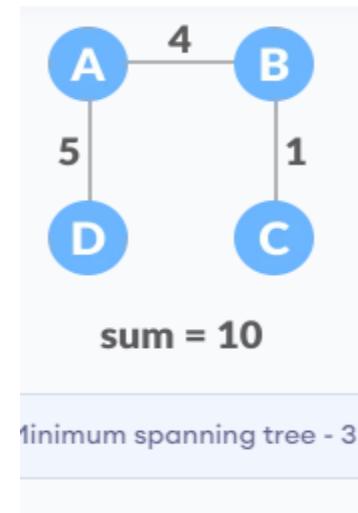
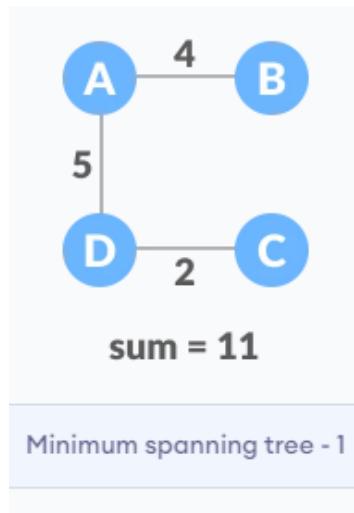
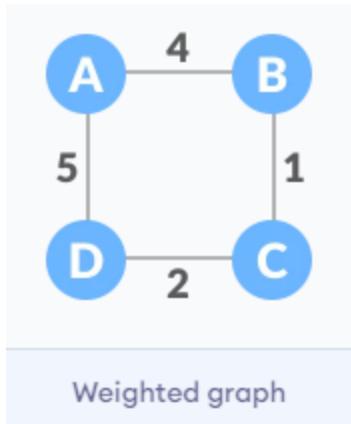
Spanning Trees

- Some of the possible spanning trees that can be created are:



Minimum Spanning Tree

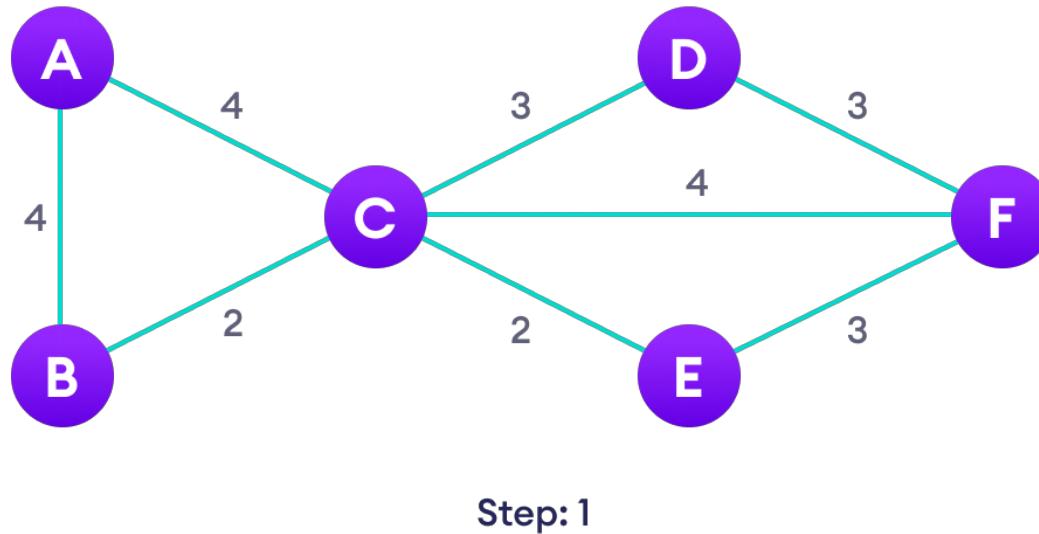
- ▶ A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.



Prim's Algorithm

- ▶ Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which
 - form a tree that includes every vertex
 - has the minimum sum of weights among all the trees that can be formed from the graph
1. Initialize the minimum spanning tree with a vertex chosen at random.
 2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
 3. Keep repeating step 2 until we get a minimum spanning tree

Prim's Algorithm

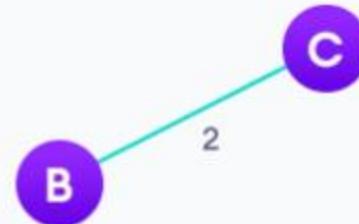


Prim's Algorithm

C

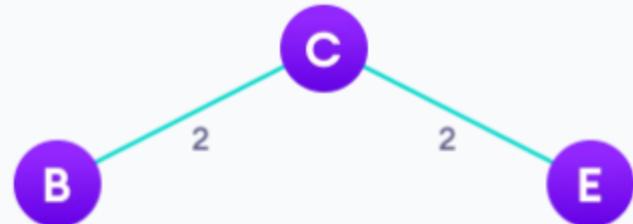
Step: 2

Choose a vertex



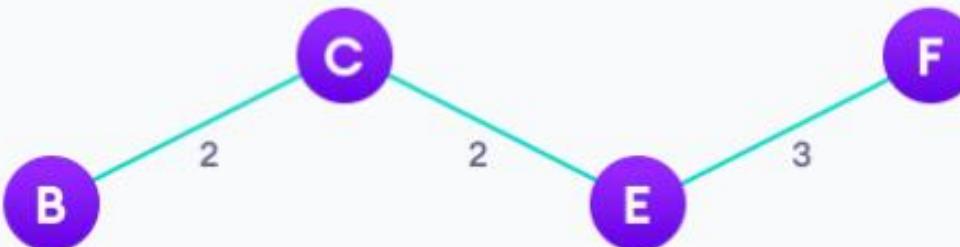
Step: 3

Choose the shortest edge from this vertex and add it



Step: 4

Choose the nearest vertex not yet in the solution



Step: 5

Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random

Prim's Algorithm

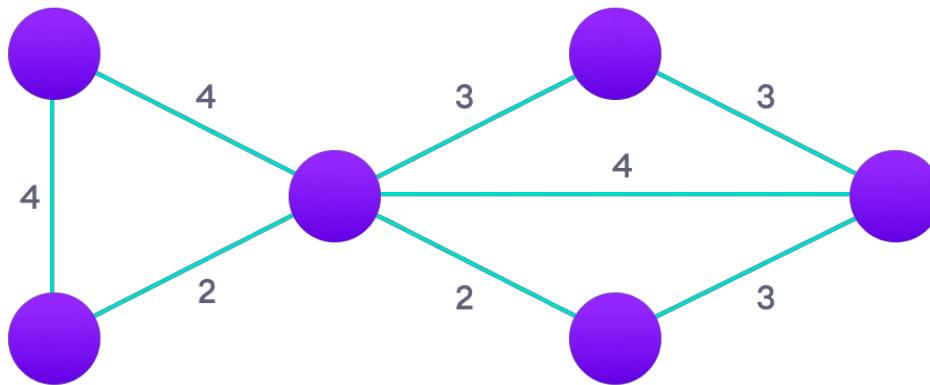


Kruskal's Algorithm

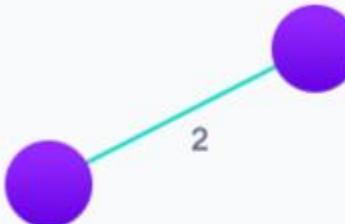
- ▶ Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which
 - form a tree that includes every vertex
 - has the minimum sum of weights among all the trees that can be formed from the graph

- ▶ The steps for implementing Kruskal's algorithm are as follows:
 - Sort all the edges from low weight to high
 - Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
 - Keep adding edges until we reach all vertices.

Kruskal's Algorithm



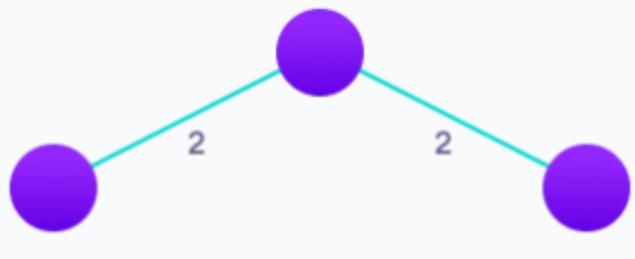
Step: 1



Step: 2

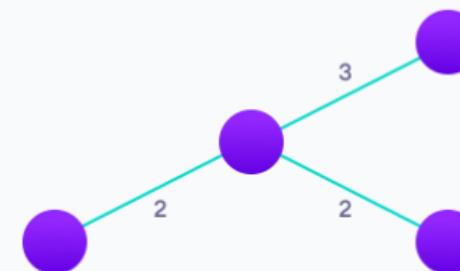
Choose the edge with the least weight, if there are more than 1, choose anyone

Kruskal's Algorithm



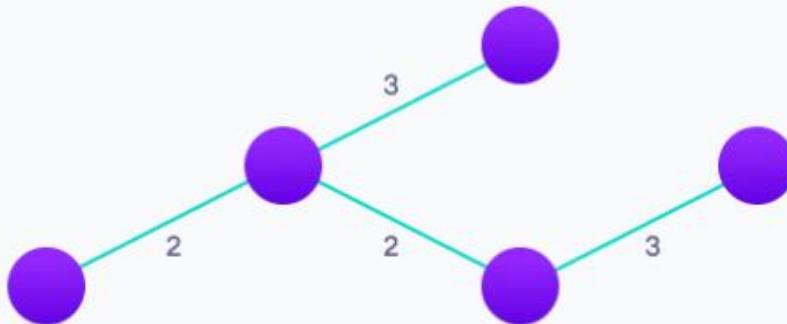
Step: 3

Choose the next shortest edge and add it



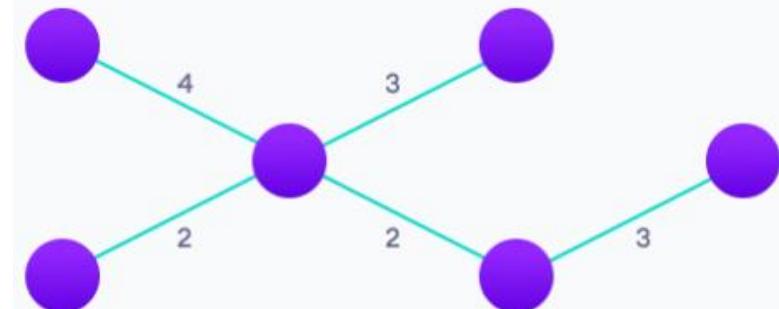
Step: 4

Choose the next shortest edge that doesn't create a cycle and add it



Step: 5

Choose the next shortest edge that doesn't create a cycle and add it



Step: 6

Repeat until you have a spanning tree

Algorithm Designs

What are the different class of algorithms

- ▶ An algorithm refers to the sequential steps and processes that should be followed to solve a problem.
- ▶ There can be various kinds of algorithms devised to solve different problems although in programming consider the following important Algorithms to solve a problem.
 - Brute Force algorithm
 - Greedy algorithm
 - Recursive algorithm
 - Backtracking algorithm
 - Divide & Conquer algorithm
 - Dynamic programming algorithm
 - Randomised algorithm

Brute Force Algorithm

- ▶ The simplest possible algorithm that can be devised to solve a problem is called the brute force algorithm.
- ▶ To device an optimal solution first we need to get a solution at least and then try to optimise it.
- ▶ Every problem can be solved by brute force approach although generally not with appreciable space and time complexity
- ▶ Example – searching for an element in a sorted array of elements

Greedy Algorithm

- ▶ Decision is made that is good at that point without considering the future. This means that some local best is chosen and considers it as the global optimal. There are two properties in this algorithm.
 - Greedily choosing the best option
 - If an optimal solution can be found by retrieving the optimal solution to its subproblems.
- ▶ This algorithm is easy to device and most of the time the simplest one. **But making locally best decisions does not always work as it sounds.** So,
- ▶ Applications - Sorting: Selection Sort, Topological sort, Prim's & Kruskal's algorithms, Coin Change problem, Fractional Knapsack Problem, Job Scheduling algorithm

Recursive Algorithm

- ▶ It does not require specifically think about every sub-problem.
- ▶ just need to think about the existing cases and the solution of the simplest subproblem, all other complexity will be handled by it automatically.
- ▶ Recursion is a very powerful tool that take care of memory management
- ▶ Recursion simply means calling itself to solve its subproblems.

- ▶ For Example: Factorial of a number

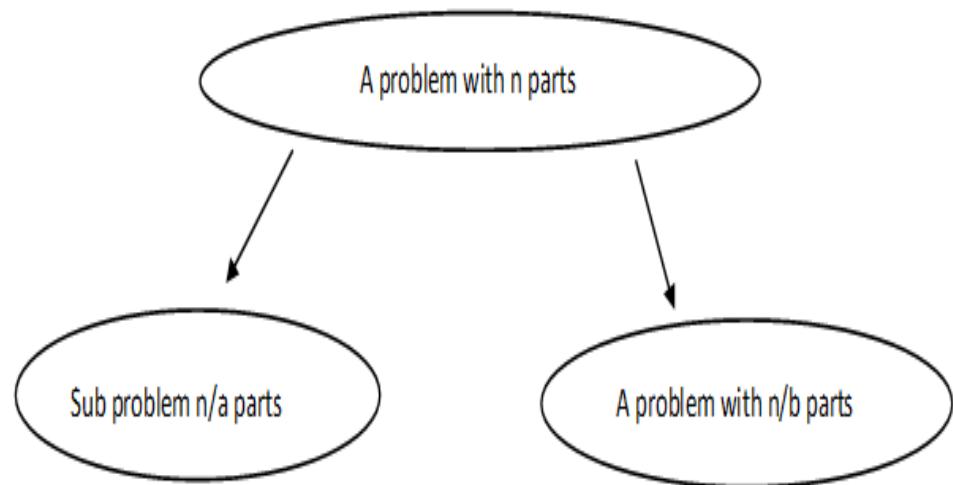
Backtracking Algorithm

- ▶ It is an improvement to the brute force approach.
- ▶ Start with one possible option out of many available and try to solve the problem with the selected move then print the solution, else backtrack and select some other and try to solve it.
- ▶ It is a form of recursion, it's just that when a given option cannot give a solution, backtrack to the previous option which can give a solution, and proceed with other options
- ▶ Applications
 - Generating all Binary strings
 - N-Queens Problem
 - Knapsack Problem
 - Graph colouring Problem

Divide & Conquer Algorithm

- ▶ As the name suggests it has two parts: Divide the problem into subproblems and solve them.
- ▶ Combine the solution of each above problems.
- ▶ This algorithm is extensively used in various problems as it is quite stable and optimal for most of the problems asked.
- ▶ Applications:

- Binary Search
- Merge Sort & Quick Sort
- Median Finding
- Matrix Multiplication



Dynamic Algorithm

- ▶ Its simply means remembering the past and apply it to future corresponding results and hence this algorithm is quite efficient in terms of time complexity.
- ▶ This algorithm has two version:
 - **Bottom-Up Approach:** Starts solving from the bottom of the problems i.e. solving the last possible sub-problems first and using the result of those solving the above sub-problems.
 -
 -
 -
- ▶ Applications – Fibonacci sequence, Bellman-Ford algorithm, Chain Matrix multiplication, Subset Sum, Knapsack Problem

Effectively using loops- Initialization

- ▶ Choose the loop counter such that it can address the smallest possible instance of the problem. Consider the problem of summing the first n integers. The smallest possible instance is to add a zero number. In that case, the minimum value of the sum variable and the loop counter i will be zero.
 - $i = 0$
 - $s = 0$

Find iterative construct

- ▶ Find the problem with next smallest instance. Careful inspection of loop and logic tells that in each iteration variable i needs to be incremented and A[i] should be added to the partially computed sum.

i = i + 1

sum = sum + A[i]

- ▶ The complete code segment would be :

i ← 0

sum ← 0

while i < n do

 i ← i + 1

 sum ← sum + a[i] // Assume that array index starts from 1.

end

Termination

- A number of iterations may be known in advance or it may depend on certain conditions to be satisfied. To find $5!$ loop should be iterated 5 times

- The loop count is known in advance:

```
for (i = 1; i <= 5; i++)  
{ ...  
}
```

- The loop count is not predefined: Sometimes loop may be forced to terminate from in-between

```
for (i = 0; i < n; i++)  
{  
    if A[i] == x  
        break;  
}
```

Remove redundant operations

► Redundant code that is repeated within loops has a negative impact on execution time.

Algorithm DRAW_ELLIPSE(a, b)

```
for x ← 0 to a do
    y ← (b/a)*sqrt(a*a - x*x)
    PlotPixel(x, round(y))
end
```

Algorithm EFFICIENT_DRAW_ELLIPSE(a, b)

```
t ← b/a;
m ← a * a;
for x ← 0 to a do
    y ← t*sqrt(m - x*x)
    PlotPixel(x, round(y))
end
```

Referencing Array

- One memory access corresponds to a reference to one array element. If the array is large, this may result in a significant number of memory accesses, slowing down the process.
- Arrays are typically used to retain a huge amount of data and are accessible via the loop. Consider the following two scenarios for locating the array's smallest element.

Algorithm POOR_FIND_MIN

$k \leftarrow 0$

for $i \leftarrow 1$ to n do

 if $A[i] < A[k]$ then

$k \leftarrow i$

 end

$Min \leftarrow A[k]$

Referencing Array

- Efficient version of this algorithm would be:

Algorithm EFFICIENT_FIND_MIN

Min \leftarrow A[0]

for i \leftarrow 1 to n do

 if A[i] < Min then

 Min \leftarrow A[i]

 k \leftarrow i

 end

Min \leftarrow A[k]

- The POOR_FIND_MIN algorithm requires two array accesses for each comparison, whereas EFFICIENT_FIND_MIN needs only one. Arithmetic on the array needs to address calculation on each array access.
- EFFICIENT_FIND_MIN compares A[i] with constant Min, which reduces the array access by a factor of two

Late termination

- Sometimes, inefficiency in the program is also introduced due to unnecessary more tests. Program keep iterating after producing useful results. Consider the problem of linear search.

Algorithm POOR_LINEAR_SEARCH(A, Key)

Index \leftarrow -1

for i \leftarrow 1 to n do

 if A[i] == key then

 Index \leftarrow i;

 end

end

if Index == -1 then

 print “Key not found”

else

 print “Key found on index i”

end

Late termination

- Efficient version of this algorithm would be:

```
Algorithm EFFICIENT_LINEAR_SEARCH(A, Key)
```

```
flag ← 0
```

```
for i ← 1 to n do
```

```
    if A[i] == key then
```

```
        flag ← 1
```

```
        break
```

```
    end
```

```
if flag == 0 then
```

```
    print “Key not found”
```

```
Else print “Key found on index i”
```

```
end
```

- POOR_LINEAR_SEARCH goes on to iterate n times Key is found at the first location. Whereas, EFFICIENT_LINEAR_SEARCH breaks the loop once Key is found, which makes it more efficient

Late termination

- Efficient version of this algorithm would be:

```
Algorithm EFFICIENT_LINEAR_SEARCH(A, Key)
```

```
flag ← 0
```

```
for i ← 1 to n do
```

```
    if A[i] == key then
```

```
        flag ← 1
```

```
        break
```

```
    end
```

```
if flag == 0 then
```

```
    print “Key not found”
```

```
Else print “Key found on index i”
```

```
end
```

- POOR_LINEAR_SEARCH goes on to iterate n times Key is found at the first location. Whereas, EFFICIENT_LINEAR_SEARCH breaks the loop once Key is found, which makes it more efficient

Analysis of an Algorithm

- ▶ The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.
- ▶ An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.
- ▶ The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

Asymptotic Notations

- ▶ These are mathematical notations used to describe running time of an algorithm when the input tends towards a particular value or a limiting value.
- ▶ For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case. But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.
- ▶ When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.
- ▶ There are mainly three asymptotic notations:
 - Big-O notation
 - Omega notation
 - Theta notation

Big-O Notation (O-notation)

- ▶ Big-O notation represents the upper bound of the running time of an algorithm.
Thus, it gives the worst-case complexity of an algorithm.
- ▶ Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

Omega Notation (Ω -notation)

- ▶ Omega notation represents the lower bound of the running time of an algorithm.
Thus, it provides the best case complexity of an algorithm.
- ▶ For any value of n , the minimum time required by the algorithm is given by
 $\text{Omega } \Omega(g(n))$

Theta Notation (Θ -notation)

- ▶ Theta notation encloses the function from above and below.
- ▶ Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

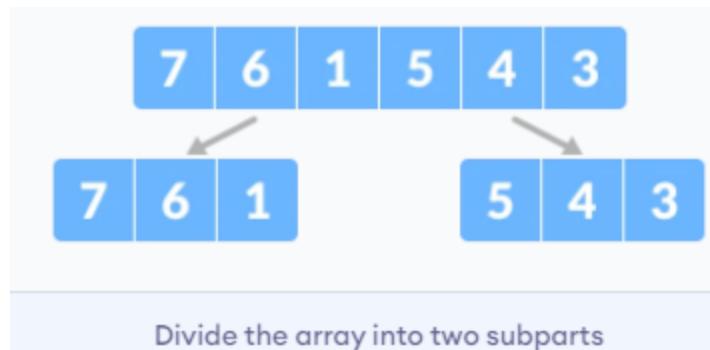
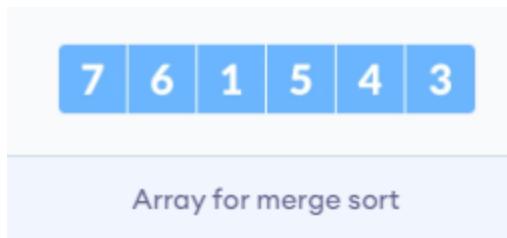
Divide and Conquer Algorithm

- ▶ A divide and conquer algorithm is a strategy of solving a large problem by
 - breaking the problem into smaller sub-problems
 - solving the sub-problems, and
 - combining them to get the desired output.

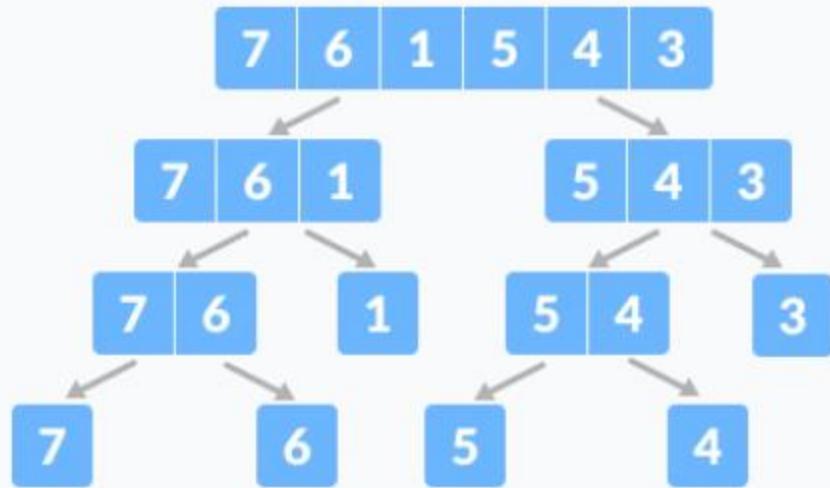
- ▶ To use the divide and conquer algorithm, recursion is used..

How Divide and Conquer Algorithms Work?

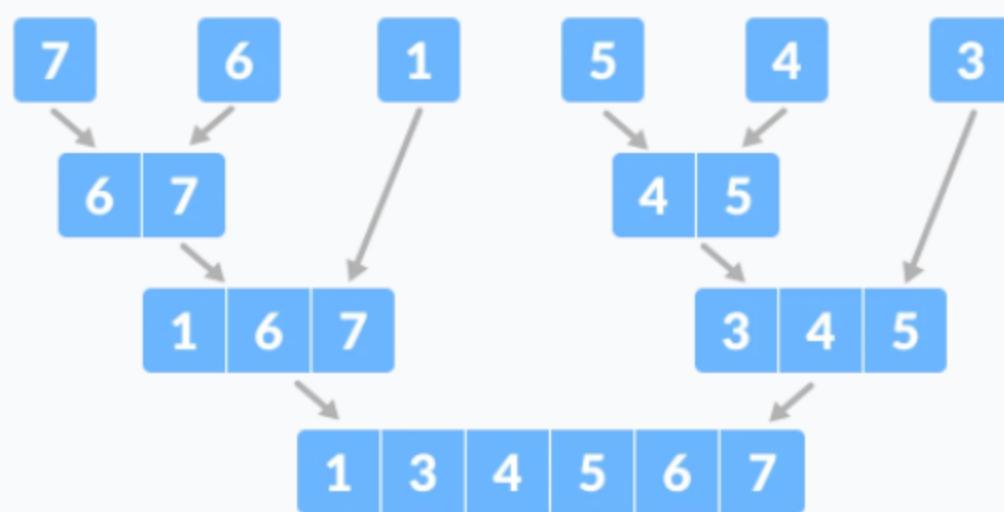
- ▶ **Divide:** Divide the given problem into sub-problems using recursion.
- ▶ **Conquer:** Solve the smaller sub-problems recursively. If the sub-problem is small enough, then solve it directly.
- ▶ **Combine:** Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.



How Divide and Conquer Algorithms Work?



Divide the array into smaller subparts



Combine the subparts

► Divide and Conquer approach:

fib(n)

If $n < 2$, return 1

Else , return $f(n - 1) + f(n -2)$

► Dynamic approach:

mem = []

fib(n)

If n in mem: return mem[n]

else,

If $n < 2$, $f = 1$

else , $f = f(n - 1) + f(n -2)$

mem[n] = f

return f

► In a dynamic approach, mem stores the result of each sub-problem.

Greedy algorithm

- A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.
- The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.
- This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result..

Greedy algorithm - Properties

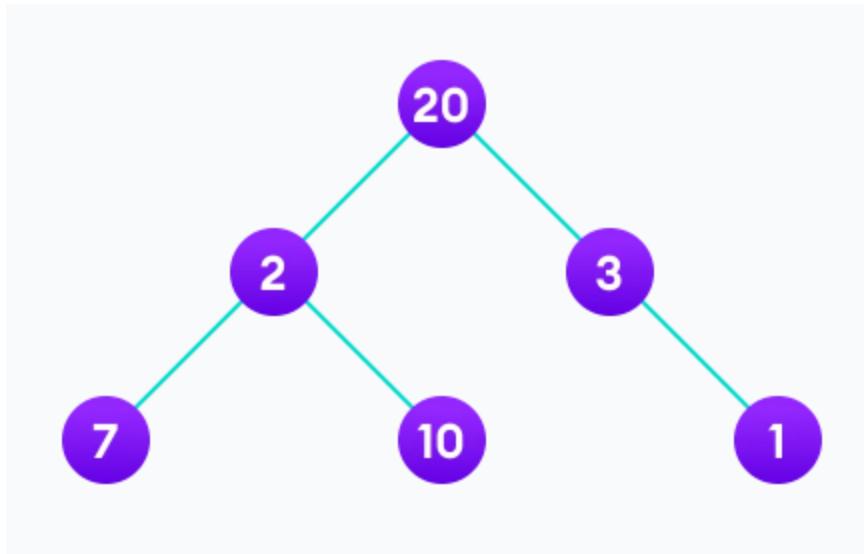
1. Greedy Choice Property

- ▶ If an optimal solution to the problem can be found by choosing the best choice at each step without reconsidering the previous steps once chosen, the problem can be solved using a greedy approach. This property is called greedy choice property.

2. Optimal Substructure

- ▶ If the optimal overall solution to the problem corresponds to the optimal solution to its subproblems, then the problem can be solved using a greedy approach. This property is called optimal substructure.

Greedy algorithm

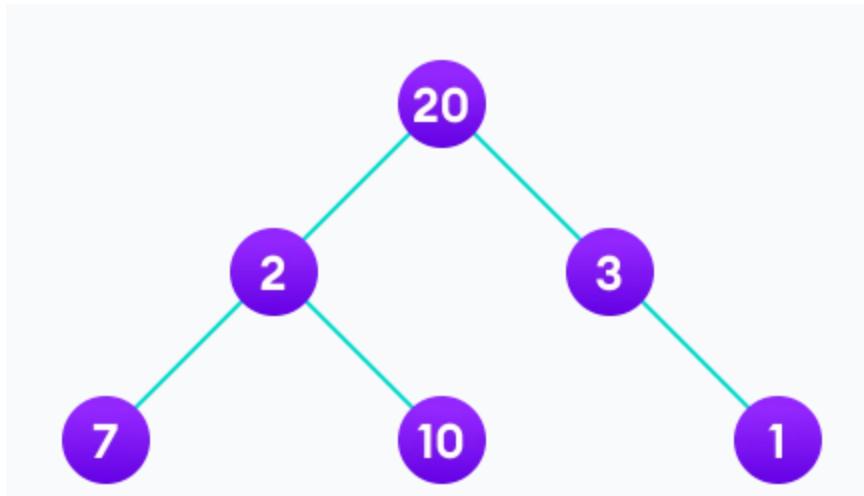


Apply greedy approach to this tree to find the longest route

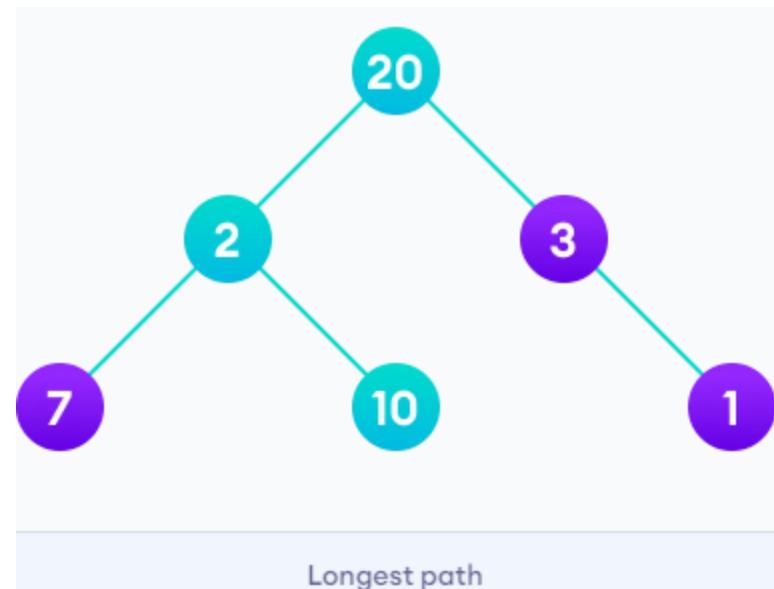
Greedy algorithm

- ▶ 1. Let's start with the root node 20. The weight of the right child is 3 and the weight of the left child is 2.
- ▶ 2. Problem is to find the largest path. And, the optimal solution at the moment is 3. So, the greedy algorithm will choose 3.
- ▶ 3. Finally the weight of an only child of 3 is 1. This gives us our final result $20 + 3 + 1 = 24$.
- ▶ However, it is not the optimal solution. There is another path that carries more weight ($20 + 2 + 10 = 32$) as shown in the image below.

Greedy algorithm



Apply greedy approach to this tree to find the longest route



Longest path

Dynamic Programming algorithm

- ▶ Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler sub-problems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its sub-problems.
- ▶ From a dynamic programming point of view, Dijkstra's algorithm for the shortest path problem is a successive approximation scheme that solves the dynamic programming functional equation for the shortest path problem by the Reaching method.

Dynamic Programming algorithm

- ▶ Let's find the fibonacci sequence upto 5th term. A fibonacci series is the sequence of numbers in which each number is the sum of the two preceding ones. For example, 0,1,1, 2, 3. Here, each number is the sum of the two preceding numbers.

- ▶ Algorithm
 - Let n be the number of terms.
 - 1. If $n \leq 1$, return 1.
 - 2. Else, return the sum of two preceding numbers.

Dynamic Programming algorithm

Calculating the fibonacci sequence up to the 5th term.

- ▶ The first term is 0. The second term is 1.
- ▶ The third term is sum of 0 (from step 1) and 1(from step 2), which is 1.
- ▶ The fourth term is the sum of the third term (from step 3) and second term (from step 2) i.e. $1 + 1 = 2$.
- ▶ The fifth term is the sum of the fourth term (from step 4) and third term (from step 3) i.e. $2 + 1 = 3$.
- ▶ Hence, sequence is 0,1,1, 2, 3. Here, results of the previous steps are used. This is called a dynamic programming approach.

$$F(0) = 0$$

$$F(1) = 1$$

$$F(2) = F(1) + F(0)$$

$$F(3) = F(2) + F(1)$$

$$F(4) = F(3) + F(2)$$

Brute force algorithm

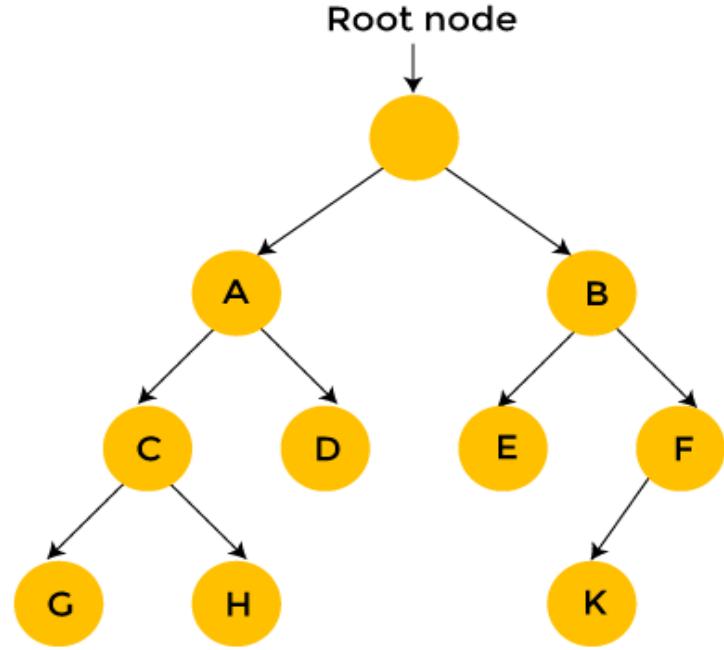
- ▶ This approach finds all the possible solutions to find a satisfactory solution to a given problem.
- ▶ It can be of following types:
- ▶ **Optimizing:** In this case, the best solution is found. To find the best solution, it may either find all the possible solutions to find the best solution or if the value of the best solution is known, it stops finding when the best solution is found. For example: Finding the best path for the travelling salesman problem. Here best path means that travelling all the cities and the cost of travelling should be minimum.

Brute force algorithm

- ▶ **Satisficing:** It stops finding the solution as soon as the satisfactory solution is found. Or example, finding the travelling salesman path which is within 10% of optimal.
- ▶ Often Brute force algorithms require exponential time. Various heuristics and optimization can be used:
- ▶ **Heuristic:** A rule of thumb that helps to decide which possibilities should look at first.
- ▶ **Optimization:** A certain possibilities are eliminated without exploring all of them.

Brute force algorithm

- ▶ Brute force search considers each and every state of a tree, and the state is represented in the form of a node. As far as the starting position is concerned, have two choices, i.e., A state and B state. In the case of B state, we have two states, i.e., state E and F.
- ▶ In the case of brute force search, each state is considered one by one. As we can observe in the above tree that the brute force search takes 12 steps to find the solution.



Brute force algorithm

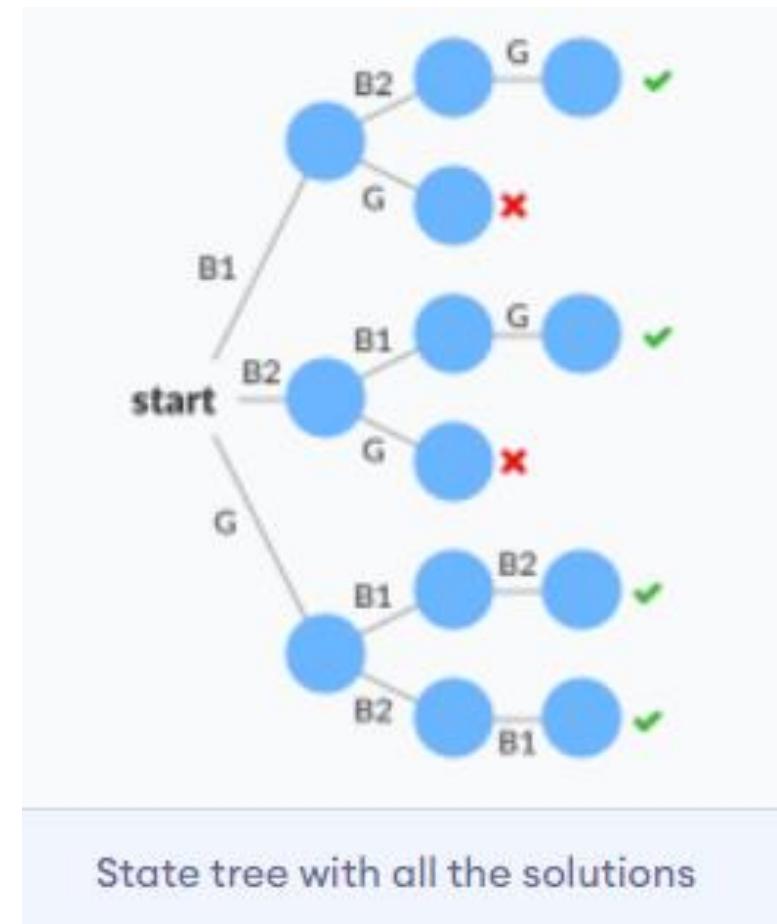
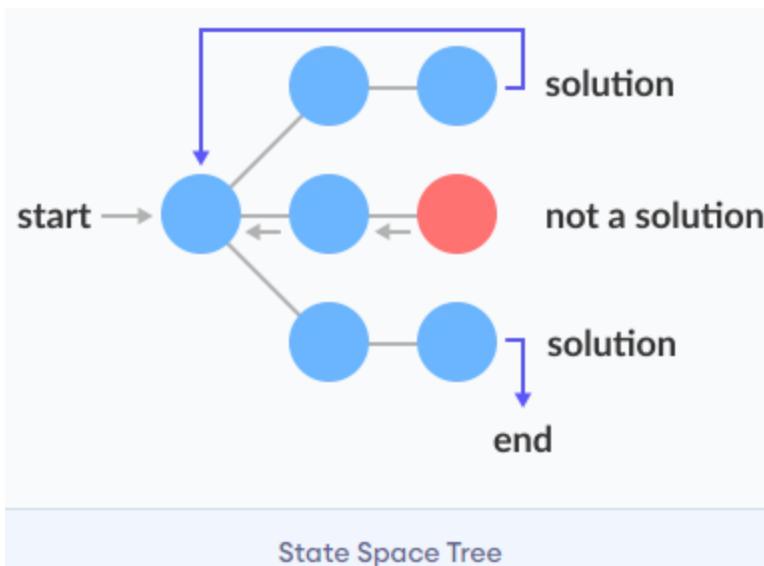
- ▶ On the other hand, backtracking, which uses Depth-First search, considers the below states only when the state provides a feasible solution. Consider the above tree, start from the root node, then move to node A and then node C. If node C does not provide the feasible solution, then there is no point in considering the states G and H. We backtrack from node C to node A. Then, we move from node A to node D. Since node D does not provide the feasible solution, we discard this state and backtrack from node D to node A.
- ▶ We move to node B, then we move from node B to node E. We move from node E to node K; Since k is a solution, so it takes 10 steps to find the solution. In this way, we eliminate a greater number of states in a single iteration. Therefore, we can say that backtracking is faster and more efficient than the brute force approach.

Backtracking algorithms

- ▶ A backtracking algorithm is a problem-solving algorithm that uses a brute force approach for finding the desired output.
- ▶ The Brute force approach tries out all the possible solutions and chooses the desired/best solutions.
- ▶ The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach.
- ▶ This approach is used to solve problems that have multiple solutions. To have an optimal solution, one must go for dynamic programming.

Backtracking algorithms

- ▶ A space state tree is a tree representing all the possible states (solution or non-solution) of the problem from the root as an initial state to the leaf as a terminal state.



- ▶ Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly.

Branch-and-bound algorithms

- ▶ The above are jobs, problems and problems given. $\text{Jobs} = \{j_1, j_2, j_3, j_4\}$, $P = \{10, 5, 8, 3\}$, $d = \{1, 2, 1, 2\}$. Solutions in two ways are:
 - 1) first way of representing the solutions is the subsets of jobs. $S_1 = \{j_1, j_4\}$
 - 2) second way of representing the solution is that first job is done, second and third jobs are not done, and fourth job is done. $S_2 = \{1, 0, 0, 1\}$
- ▶ The solution s_1 is the variable-size solution while the solution s_2 is the fixed-size solution.

Branch-and-bound algorithms

- ▶ The above are jobs, problems and problems given. $\text{Jobs} = \{j_1, j_2, j_3, j_4\}$, $P = \{10, 5, 8, 3\}$, $d = \{1, 2, 1, 2\}$. Solutions in two ways are:
 - 1) first way of representing the solutions is the subsets of jobs. $S_1 = \{j_1, j_4\}$
 - 2) second way of representing the solution is that first job is done, second and third jobs are not done, and fourth job is done. $S_2 = \{1, 0, 0, 1\}$
- ▶ The solution s_1 is the variable-size solution while the solution s_2 is the fixed-size solution.

Branch-and-bound algorithms

