# COSC1285/2123: Algorithms & Analysis

## Decrease and Conquer

Jeffrey Chan

RMIT University
Email : jeffrey.chan@rmit.edu.au

Lecture 4

# Overview

## Levitin – The design and analysis of algorithms

This week we will be covering the material from Chapter 4.

Learning Outcomes:

- Understand the *Decrease-and-conquer* algorithmic approach.
- Understand, contrast and apply:
  - Decrease-by-a-**constant** algorithms - insertion and topological sort.
  - Decrease-by-a-**constant-factor** algorithms - binary search, fake coin problem.
  - **Variable-size** decrease algorithms - binary search tree.

# Outline

# Overview

1. Overview
2. Decrease-by-a-Constant: Insertion Sort
3. Decrease-by-a-Constant: Topological Sorting
4. Decrease-by-a-Constant-Factor Algorithms
5. Variable-Size Decrease Algorithms & Binary Search Trees
6. Case Study
7. Summary

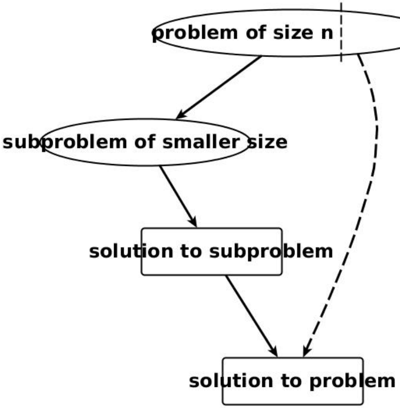
# Decrease-and-conquer Approach

Process:

1. Reduce a problem instance to a smaller instance of the same problem.
2. Solve the smaller instance.
3. Extend the solution of the smaller instance to obtain the solution to the original instance.

- Sometimes referred to as the inductive or incremental approach.

# Decrease-and-conquer Approach



# Decrease-and-Conquer – Examples

1. Decrease-by-a-constant
   - **Insertion Sorting**
   - **Topological Sorting**
   - Algorithms for generating permutations and subsets
2. Decrease-by-a-constant-factor
   - **Binary Search**
   - **Fake-coin Problem**
   - Multiplication à la russe
   - Josephus Problem
3. Variable-size-decrease
   - **Search, Insert and Delete in a Binary Search Tree**
   - Euclid's Algorithm
   - Interpolation Search
   - Selection by Partitioning
   - Game of Nim

# Overview

# Insertion Sort – Sketch

Insertion sort is the method people often use to sort a hand of playing cards. The basic idea:

# Insertion Sort – Sketch

Insertion sort is the method people often use to sort a hand of playing cards. The basic idea:

- Consider each element one at a time (left to right).
- Insert each element in its proper place among those already considered. (i.e. insert into a already sorted sub-file). This is a right to left scan.

# Insertion Sort – Example

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

# Insertion Sort – Example

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

# Insertion Sort – Example

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

# Insertion Sort – Example

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

# Insertion Sort – Example

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

# Insertion Sort – Example

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 34 | 53 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

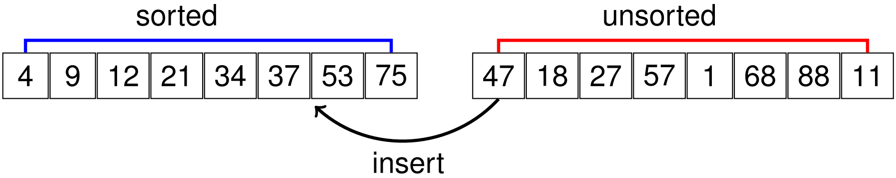
# Insertion Sort – Example

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 34 | 53 | | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

# Insertion Sort – Example

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 21 | 34 | 53 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

# Insertion Sort – Example



# Insertion Sort – Pseudocode

ALGORITHM **InsertionSort** $(A[0 \dots n-1])$
/* Sort an array using an insertion sort. */
/* INPUT : An array $A[0 \dots n-1]$ of orderable elements. */
/* OUTPUT : An array $A[0 \dots n-1]$ sorted in order. */

**for** $i = 1$ **to** $n - 1$ **do**
    $v = A[i]$
    $j = i - 1$
    **while** $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] = A[j]$
        $j = j - 1$
    **end while**
    $A[j+1] = v$
**end for**

# Insertion Sort – Analysis

- **Worst Case :** The input is in reverse order.

$$C_w(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \mathcal{O}(n^2)$$

# Insertion Sort – Analysis

- **Worst Case :** The input is in reverse order.

$$C_w(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \mathcal{O}(n^2)$$

- **Average Case :** For randomly ordered data, we expect each item to move halfway back.

$$C_a(n) \approx \frac{n^2}{4} \in \mathcal{O}(n^2)$$

- **Best Case :** Next slide

# Insertion Sort – Analysis

**Best Case :**

1. In terms of the input, what is the scenario/circumstance where we can achieve the best case?
2. What is the time complexity for the best case?

Google forms: `https://goo.gl/forms/sA69DTJHv4d3GVeE3`

# Building a better Insertion Sort

Shell's Sort is a simple but effective extension of insertion sort.

# Building a better Insertion Sort

Shell's Sort is a simple but effective extension of insertion sort.

Proposition : For partially-sorted arrays, insertion sort runs in linear time.

- What if a fast way can be designed to partially sort an input array?

# Building a better Insertion Sort

Shell's Sort is a simple but effective extension of insertion sort.

Proposition : For partially-sorted arrays, insertion sort runs in linear time.

- What if a fast way can be designed to partially sort an input array?

Approach:

- Apply insertion sort to several interleaving sublists of the array.
- The sublists are formed by stepping through the array with an increment $h_i$ which decreases by some predefined increment on each pass, where $h_1 > \ldots > h_i > \ldots > 1$.
- Choosing the optimal stepping is still an open problem, but the reverse of the sequence $1, 4, 13, 40, 121, 364, \ldots (3x + 1)$ works well in practice.

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | | | | | 37 | | | | | 27 | | | | | 11 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | | | | | 27 | | | | | 34 | | | | | 37 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | | | | 27 | 75 | | | | 34 | 57 | | | | 37 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | | | | 27 | 57 | | | | 34 | 75 | | | | 37 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | 21 | | | 27 | 57 | 4 | | | 34 | 75 | 1 | | | 37 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | 1 | | | 27 | 57 | 4 | | | 34 | 75 | 21 | | | 37 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | 1 | 9 | | 27 | 57 | 4 | 47 | | 34 | 75 | 21 | 68 | | 37 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | 1 | 9 | 12 | 27 | 57 | 4 | 47 | 18 | 34 | 75 | 21 | 68 | 88 | 37 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | 1 | 9 | 12 | 27 | 57 | 4 | 47 | 18 | 34 | 75 | 21 | 68 | 88 | 37 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | 1 | 9 | 12 | 27 | 57 | 4 | 47 | 18 | 34 | 75 | 21 | 68 | 88 | 37 |

3-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | 1 | 9 | 12 | 27 | 57 | 4 | 47 | 18 | 34 | 75 | 21 | 68 | 88 | 37 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | 1 | 9 | 12 | 27 | 57 | 4 | 47 | 18 | 34 | 75 | 21 | 68 | 88 | 37 |

3-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | 1 | 9 | 12 | 27 | 57 | 4 | 47 | 18 | 34 | 75 | 21 | 68 | 88 | 37 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | 1 | 9 | 12 | 27 | 57 | 4 | 47 | 18 | 34 | 75 | 21 | 68 | 88 | 37 |

3-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 53 | 1 | 11 | 12 | 27 | 18 | 4 | 47 | 21 | 34 | 75 | 37 | 68 | 88 | 57 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | 1 | 9 | 12 | 27 | 57 | 4 | 47 | 18 | 34 | 75 | 21 | 68 | 88 | 37 |

3-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 53 | 1 | 11 | 12 | 27 | 18 | 4 | 47 | 21 | 34 | 75 | 37 | 68 | 88 | 57 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | 1 | 9 | 12 | 27 | 57 | 4 | 47 | 18 | 34 | 75 | 21 | 68 | 88 | 37 |

3-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 4 | 1 | 11 | 12 | 27 | 18 | 34 | 47 | 21 | 53 | 75 | 37 | 68 | 88 | 57 |

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | 1 | 9 | 12 | 27 | 57 | 4 | 47 | 18 | 34 | 75 | 21 | 68 | 88 | 37 |

3-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 4 | 1 | 11 | 12 | 27 | 18 | 34 | 47 | 21 | 53 | 75 | 37 | 68 | 88 | 57 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | 1 | 9 | 12 | 27 | 57 | 4 | 47 | 18 | 34 | 75 | 21 | 68 | 88 | 37 |

3-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 4 | 1 | 11 | 12 | 27 | 18 | 34 | 47 | 21 | 53 | 75 | 37 | 68 | 88 | 57 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | 1 | 9 | 12 | 27 | 57 | 4 | 47 | 18 | 34 | 75 | 21 | 68 | 88 | 37 |

3-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 4 | 1 | 11 | 12 | 27 | 18 | 34 | 47 | 21 | 53 | 75 | 37 | 68 | 88 | 57 |

Insertion Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 4 | 1 | 11 | 12 | 27 | 18 | 34 | 47 | 21 | 53 | 75 | 37 | 68 | 88 | 57 |

# Shell Sort

Perform Shell Sort on sequence:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 53 | 21 | 9 | 12 | 37 | 75 | 4 | 47 | 18 | 27 | 57 | 1 | 68 | 88 | 11 |

5-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 53 | 1 | 9 | 12 | 27 | 57 | 4 | 47 | 18 | 34 | 75 | 21 | 68 | 88 | 37 |

3-Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 4 | 1 | 11 | 12 | 27 | 18 | 34 | 47 | 21 | 53 | 75 | 37 | 68 | 88 | 57 |

Insertion Sort:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 4 | 9 | 11 | 12 | 18 | 21 | 27 | 34 | 37 | 47 | 53 | 57 | 68 | 75 | 88 |

# Shell Sort

- The worst case number of compares using the $3x + 1$ sequence is $\mathcal{O}(n^{3/2})$.

# Shell Sort

- The worst case number of compares using the $3x + 1$ sequence is $\mathcal{O}(n^{3/2})$.
- The algorithm is **not** stable. You should be able to give a counterexample.

# Shell Sort

- The worst case number of compares using the $3x + 1$ sequence is $\mathcal{O}(n^{3/2})$.
- The algorithm is **not** stable. You should be able to give a counterexample.
- Quite good performance in practice and is easily used in embedded and even hardware solutions.
- Simple algorithm, nontrivial performance: What is the best sequence increment? How would you even begin to attempt an average case performance analysis?
- Take home lesson: We don't know all the answers!

# Useful websites

Interesting and useful sorting visualisation websites:

`http://sorting-algorithms.com`

`http://visualgo.net/sorting.html`

# Topological Sort

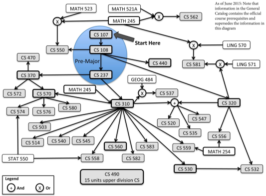
Imagine we have the following problems:

Job scheduling with order dependencies: What is the order the jobs should be processed to avoid breaking these dependencies?

# Topological Sort

Imagine we have the following problems:

Job scheduling with order dependencies: What is the order the jobs should be processed to avoid breaking these dependencies?

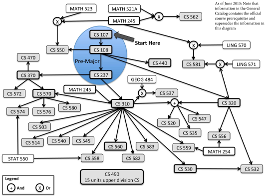



Subject selection with pre-requisites: What are the order the subjects could be taken to ensure we have all the pre-requisites?

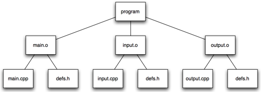
# Topological Sort

Imagine we have the following problems:

Job scheduling with order dependencies: What is the order the jobs should be processed to avoid breaking these dependencies?




Subject selection with pre-requisites: What are the order the subjects could be taken to ensure we have all the pre-requisites?




Makefile compilation: What is the order the source files should be compiled to ensure we can build a working program?

# Topological Sort

Topological sort produces a traversal ordering of the vertices/nodes for directed graphs.

# Topological Sort

Topological sort produces a traversal ordering of the vertices/nodes for directed graphs.

## Topological Sorting Problem

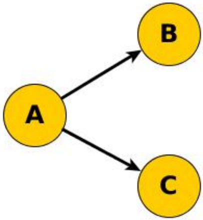
Given a digraph, list its vertices in such an order that for every edge $(a, b)$ in the graph, vertex $a$ must appear before vertex $b$ in the list.

# Topological Sort

Topological sort produces a traversal ordering of the vertices/nodes for directed graphs.

## Topological Sorting Problem

Given a digraph, list its vertices in such an order that for every edge $(a, b)$ in the graph, vertex $a$ must appear before vertex $b$ in the list.

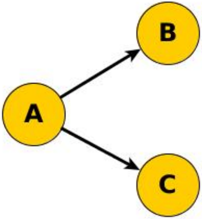


# Topological Sort

Topological sort produces a traversal ordering of the vertices/nodes for directed graphs.

## Topological Sorting Problem

Given a digraph, list its vertices in such an order that for every edge $(a, b)$ in the graph, vertex $a$ must appear before vertex $b$ in the list.

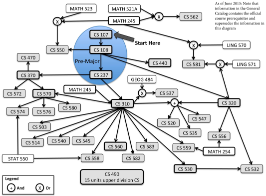



**NOTE :** If the graph is a directed acyclic graph (DAG), the topological sort has at least one solution.

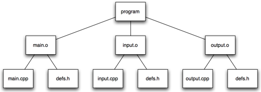
# Topological Sort - Applications

Imagine we have the following problems:

Job scheduling with order dependencies: What is the order the jobs should be processed to avoid breaking these dependencies?




Subject selection with pre-requisites: What are the order the subjects could be taken to ensure we have all the pre-requisites?




Makefile compilation: What is the order the source files should be compiled to ensure we can build a working program?

# Topological Sort – Approaches

There are two different approaches to solve this problem:

1. DFS Method
2. Source Removal Method (focus of this lecture)

# Topological Sort – Source Removal Method

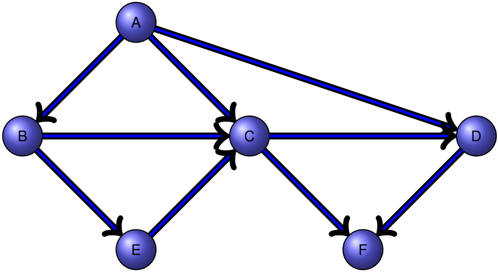
**Idea:** Select next vertex in ordering that respect the $(a, b)$ ordering. If we can find a vertex $a$ that does not have any incoming vertex, than it cannot violate the property. Such a vertex is a source vertex (one that has no incoming vertices).

**Idea:** Select next vertex in ordering that respect the $(a, b)$ ordering. If we can find a vertex $a$ that does not have any incoming vertex, than it cannot violate the property. Such a vertex is a source vertex (one that has no incoming vertices).

**Source Removal Method**:

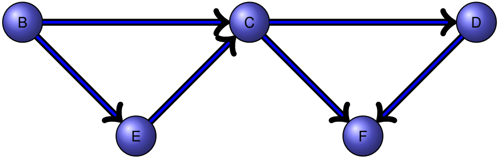
- Choose a source vertex.
- Delete the vertex and all incident edges and append the vertex to topological ordered list.
- Repeat the selection of a source vertex and deletion process for the remaining graph until no vertices are left.

# Topological Sort – Source Removal Method



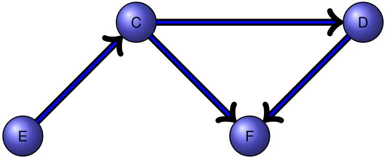

Solution :

# Topological Sorting – Source Removal Method



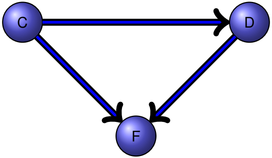

Solution : A

# Topological Sorting – Source Removal Method



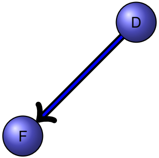

Solution : A B

# Topological Sorting – Source Removal Method




Solution : A B E

# Topological Sorting – Source Removal Method




Solution : A B E C

# Topological Sorting – Source Removal Method


Solution : A B E C D

# Topological Sorting – Source Removal Method

Solution : A B E C D F

# Topological Sorting – Summary & Questions

- Topological sorting can have more than one solution, and often does for very large DAGs.

# Topological Sorting – Summary & Questions

- Topological sorting can have more than one solution, and often does for very large DAGs.
- Source removal algorithm:
    - How do you find a source (or determine that such a vertex does not exist) in a digraph represented by an adjacency matrix? What is the time efficiency? (Homework)

# Topological Sorting – Summary & Questions

- Topological sorting can have more than one solution, and often does for very large DAGs.
- Source removal algorithm:
  - How do you find a source (or determine that such a vertex does not exist) in a digraph represented by an adjacency matrix? What is the time efficiency? (Homework)
  - How do you find a source (or determine that such a vertex does not exist) in a digraph represented by an adjacency list? What is the time efficiency? (Homework)

# Topological Sorting – Summary & Questions

- Topological sorting can have more than one solution, and often does for very large DAGs.
- Source removal algorithm:
  - How do you find a source (or determine that such a vertex does not exist) in a digraph represented by an adjacency matrix? What is the time efficiency? (Homework)
  - How do you find a source (or determine that such a vertex does not exist) in a digraph represented by an adjacency list? What is the time efficiency? (Homework)
  - The source removal algorithm for a digraph represented as an adjacency list can be implemented such that the running time is $\mathcal{O}(|V| + |E|)$. How? (Homework)

# Overview

# Decrease-by-a-constant-factor Approaches

Algorithms that use this approach divide the problem into parts (half, thirds, etc), and then recursively operate on one of the halves, thirds etc.

Hence, at each iteration, we decrease the problem by a constant (a half, a third etc).

# Decrease-by-a-constant-factor Approaches

Algorithms that use this approach divide the problem into parts (half, thirds, etc), and then recursively operate on one of the halves, thirds etc.

Hence, at each iteration, we decrease the problem by a constant (a half, a third etc).

We study two examples:

- Binary search
- Fake coin problem

# Binary Search

Binary search is a worst-case optimal algorithm for searching in a sorted sequence of elements.

- Given a sorted sequence, compare the value in the array at position $n/2$ with a key $k$.
- If $A[n/2] > k$, compare $k$ with the midpoint of the lower half.
- If $A[n/2] < k$, compare $k$ with the midpoint of the upper half.
- If $A[n/2] = k$, return $n/2$ (the index of the position containing $k$).

# Binary Search - Example

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 9 | 12 | 24 | 29 | 34 | 35 | 37 | 53 | 62 | 74 | 53 | 62 | 74 | 92 |

BINARYSEARCH(5)

# Binary Search - Example

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 9 | 12 | 24 | 29 | 34 | 35 | 37 | 53 | 62 | 74 | 53 | 62 | 74 | 92 |

# Binary Search - Example

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 9 | 12 | 24 | 29 | 34 | 35 | 37 | 53 | 62 | 74 | 53 | 62 | 74 | 92 |

$5 < 35$

# Binary Search - Example

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 9 | 12 | 24 | 29 | 34 | 35 | 37 | 53 | 62 | 74 | 53 | 62 | 74 | 92 |

Continue left

# Binary Search - Example

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 9 | 12 | 24 | 29 | 34 | 35 | 37 | 53 | 62 | 74 | 53 | 62 | 74 | 92 |

$5 < 9$

# Binary Search - Example

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 9 | 12 | 24 | 29 | 34 | 35 | 37 | 53 | 62 | 74 | 53 | 62 | 74 | 92 |

$5 > 3$

# Binary Search - Example

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 9 | 12 | 24 | 29 | 34 | 35 | 37 | 53 | 62 | 74 | 53 | 62 | 74 | 92 |

FOUND!

# Binary Search – Recursive

ALGORITHM **RecursiveBinarySearch** $(A[\ell \dots r], k)$
/* A recursive binary search in an ordered array. */
/* INPUT : An array $A[\ell \dots r]$ of ordered elements, and a search key $k$. */
/* OUTPUT : an index to the position of $k$ in $A$ if $k$ is found or $-1$ otherwise. */

**if** $\ell > r$ **then**
    **return** $-1$
**else**
    $m \leftarrow \lfloor (\ell + r)/2 \rfloor$
    **if** $k = A[m]$ **then**
        **return** $m$
    **else if** $k < A[m]$ **then**
        **return RecursiveBinarySearch** $(A[\ell \dots m - 1], k)$
    **else**
        **return RecursiveBinarySearch** $(A[m + 1 \dots r], k)$
    **end if**
**end if**

# Analysis - Binary Search

1. $\mathcal{C}(n) = \mathcal{C}(\lfloor n/2 \rfloor) + 1$ for $n > 1$ and $\mathcal{C}(1) = 1$.
2. Using the smoothness rule, $n = 2^k$.
3. $\mathcal{C}(2^k) = \mathcal{C}(2^{k-1}) + 1$ for $k > 0$, $C(2^0) = C(1) = 1$.
4. Substitute $\mathcal{C}(2^{k-1}) = \mathcal{C}(2^{k-2}) + 1$.
5. $\mathcal{C}(2^k) = [\mathcal{C}(2^{k-2}) + 1] + 1 = \mathcal{C}(2^{k-2}) + 2$.
6. Substitute $\mathcal{C}(2^{k-2}) = \mathcal{C}(2^{k-3}) + 1$.
7. $\mathcal{C}(2^k) = [\mathcal{C}(2^{k-3}) + 1] + 2 = \mathcal{C}(2^{k-3}) + 3$.
8. We see the pattern $\mathcal{C}(2^k) = \mathcal{C}(2^{k-i}) + i$ emerge.
9. $\mathcal{C}(2^k) = \mathcal{C}(2^{k-i}) + i$.
10. We want $C(2^{k-i}) = C(2^0)$, or $k - i = 0$ or $i = k$.
11. $\mathcal{C}(2^k) = \mathcal{C}(2^{k-k}) + k = \mathcal{C}(2^0) + k = 1 + k$.
12. If $n = 2^k$ then $k = \log_2 n$:
13. $\mathcal{C}(n) = \log_2 n + 1 \in \mathcal{O}(\log_2 n)$.

# Binary Search Properties

Worst case of $\mathcal{O}(\log(n))$

# Binary Search Properties

Worst case of $\mathcal{O}(\log(n))$

Only achieved if:

- Array is sorted.
- The array has $\mathcal{O}(1)$ access to any position.

## Fake-Coin Problem

Given a stack of $n$ identical-looking coins which contains exactly one fake coin (which is lighter) and a scale/weigh, devise an efficient algorithm for detecting the fake coin.

# Fake-Coin Problem

## Fake-Coin Problem

Given a stack of $n$ identical-looking coins which contains exactly one fake coin (which is lighter) and a scale/weigh, devise an efficient algorithm for detecting the fake coin.

# Fake-Coin Problem

## Fake-Coin Problem

Given a stack of $n$ identical-looking coins which contains exactly one fake coin (which is lighter) and a scale/weigh, devise an efficient algorithm for detecting the fake coin.

The solution is to use a decrease by half algorithm:

- Divide into two sub-stacks of $n/2$ coins and weigh on scale.
- The lighter sub-stack contains the fake coin.
- Repeat process with the lighter sub-stack until we have two coins remaining. The fake coin must be one of the two.

# Fake-Coin Problem

## Fake-Coin Problem

Given a stack of $n$ identical-looking coins which contains exactly one fake coin (which is lighter) and a scale/weigh, devise an efficient algorithm for detecting the fake coin.

The solution is to use a decrease by half algorithm:

- Divide into two sub-stacks of $n/2$ coins and weigh on scale.
- The lighter sub-stack contains the fake coin.
- Repeat process with the lighter sub-stack until we have two coins remaining. The fake coin must be one of the two.

The recurrence relation for number of weighings: $C(n) = C(\lfloor n/2 \rfloor) + 1$ for $n > 1$, $C(1) = 0$. Gives worst case of $\mathcal{O} log_2(n)$.

# Fake-Coin Problem – Example

```
http://www.youtube.com/watch?v=wVPCT1VjySA
```

# Trees

In its most general form, a tree is a connected acyclic graph.

There are many different types of trees used in computer science:

- binary trees
- *m*-ary search trees
- balanced trees (AVL, Red-Black)
- forests
- Kd-tree

Here we focus on binary search trees.

# Binary Tree

A binary tree is:

- a tree (hence has a root node)
- every vertex has no more than two children

# Binary Tree

A binary tree is:

- a tree (hence has a root node)
- every vertex has no more than two children

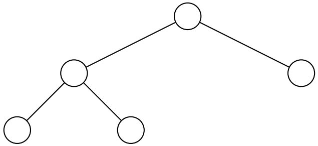
Each child is designated as either a left child or a right child of its parent

# Binary Tree

A binary tree is:

- a tree (hence has a root node)
- every vertex has no more than two children

Each child is designated as either a left child or a right child of its parent

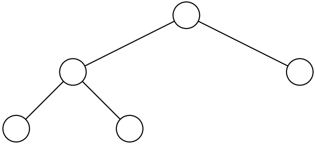


# Binary Tree

A binary tree is:

- a tree (hence has a root node)
- every vertex has no more than two children

Each child is designated as either a left child or a right child of its parent



Technical point: If a node does not have a child, the pointer/reference to the child is set to null.

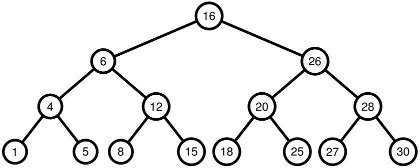
# Binary Search Tree

A binary search tree is a binary tree that additionally:

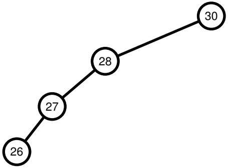
- have values associated with each node
- ordered such that for each parent vertex:
    - all values in its left subtree are smaller than the parent's value; and
    - all values in its right subtree are larger than the parent's value.

A binary search tree is a binary tree that additionally:

- have values associated with each node
- ordered such that for each parent vertex:
  - all values in its left subtree are smaller than the parent's value; and
  - all values in its right subtree are larger than the parent's value.




Example of a balanced, full binary search tree.

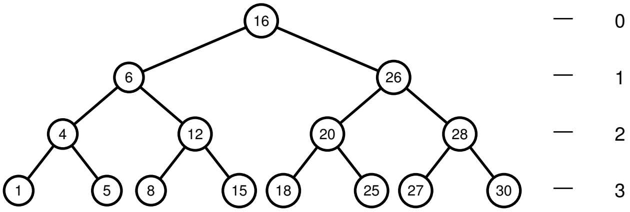



Example of a unbalanced, "stick" binary search tree.

# BST Properties

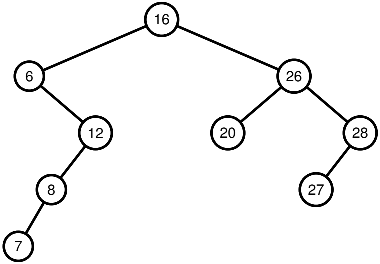
- The height of a tree is the length of the path from the root to the deepest node in a tree. A tree with only a root node has a height of 0.
- The depth of a node is the length of the path from the root to the node. The root node has a depth 0.
- The level of a tree is the set of all nodes at a given depth.

# BST Properties - Example



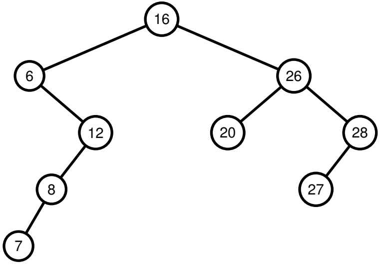

- Height = 3
- Depth of node 12 = 2
- Depth of node 18 = 3
- Nodes in level 1 = {6, 26}

# BST Properties - Example



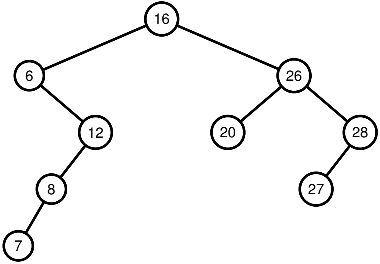

- Root is node 16.
- Height?

# BST Properties - Example




- Root is node 16.
- Height? 4

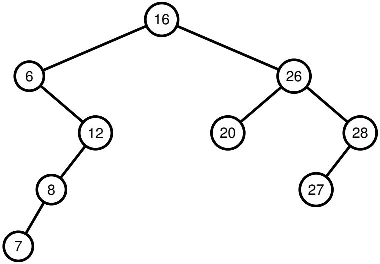
# BST Properties - Example




- Root is node 16.
- Height? 4

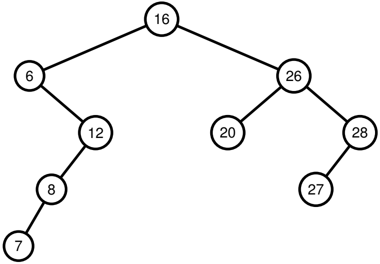
- Nodes on Level 2?

# BST Properties - Example



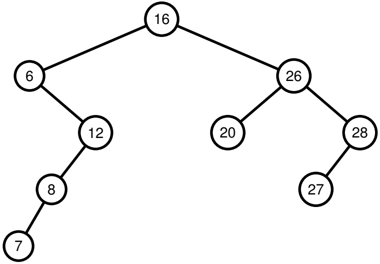

- Root is node 16.
- Height? 4
- Nodes on Level 2? {12, 20, 28}

# BST Properties - Example




- Root is node 16.
- Height? 4
- Nodes on Level 2? {12, 20, 28}
- Depth of node 27?

# BST Properties - Example




- Root is node 16.
- Height? 4
- Nodes on Level 2? {12, 20, 28}
- Depth of node 27? 3

# BST algorithms

- Searching for a value in a BST (variable-size-decrease)
- Inserting a new value into a BST (variable-size-decrease)
- Deleting a value and associated node from a BST

# BST Search

**Aim:** Search for a key $k$ in tree $T$ (represented by its root node).

# BST Search

**Aim:** Search for a key $k$ in tree $T$ (represented by its root node).

**Idea:** Recursively search for the key $k$ in tree, taking advantage of its structure.

# BST Search

**Aim:** Search for a key $k$ in tree $T$ (represented by its root node).

**Idea:** Recursively search for the key $k$ in tree, taking advantage of its structure.

Search($T$, $k$):

1. If $T$ is empty, return **null**.
2. If value of $k = val(T)$, return **T**.
3. If value of $k < val(T)$, search the left subtree (return Search($T_L$, $k$))
4. If value of $k > val(T)$, search the right subtree (return Search($T_R$, $k$))

# BST Search

**Aim:** Search for a key $k$ in tree $T$ (represented by its root node).

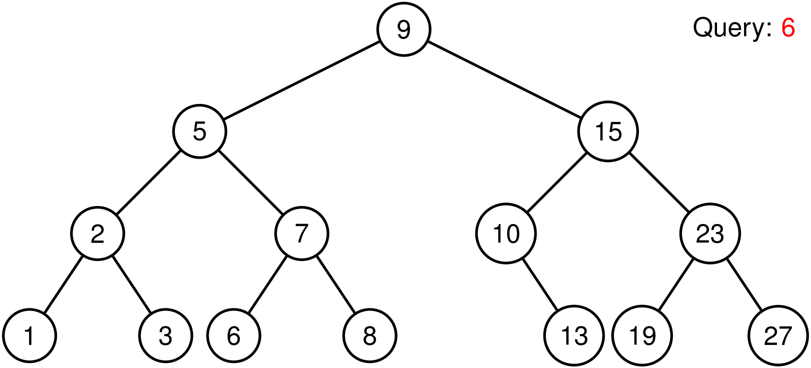
**Idea:** Recursively search for the key $k$ in tree, taking advantage of its structure.

Search($T$, $k$):

1. If $T$ is empty, return **null**.
2. If value of $k = val(T)$, return **T**.
3. If value of $k < val(T)$, search the left subtree (return Search($T_L$, $k$))
4. If value of $k > val(T)$, search the right subtree (return Search($T_R$, $k$))

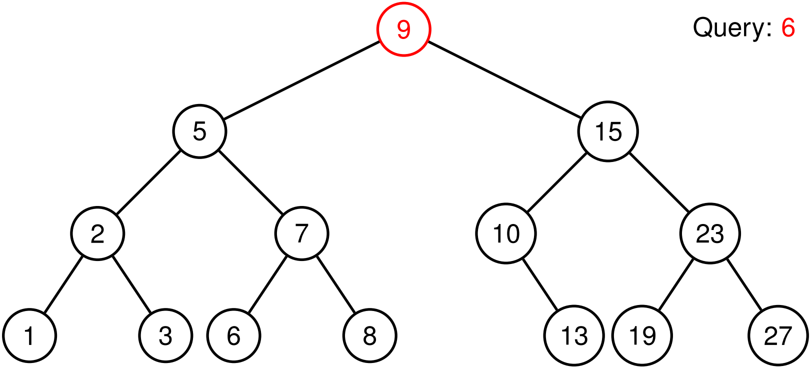
**NOTE :** This is a variable-size-decrease algorithm, as each iteration we decrease the remaining problem size by a variable amount.

# BST Search



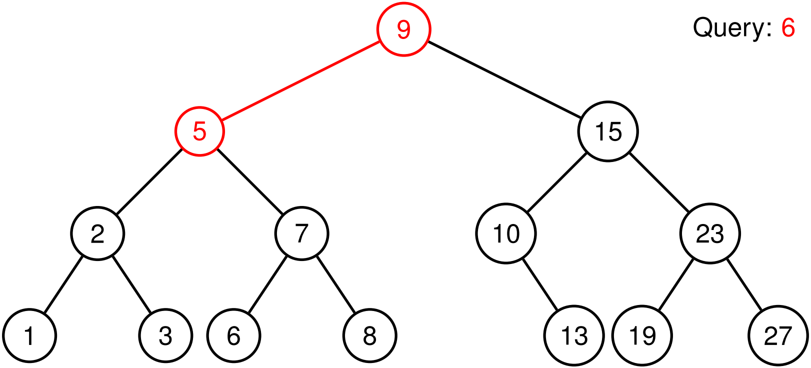

Query: 6

# BST Search

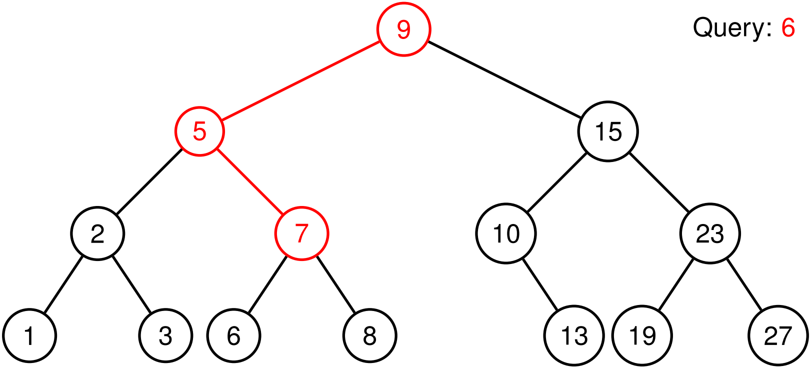



Query: 6

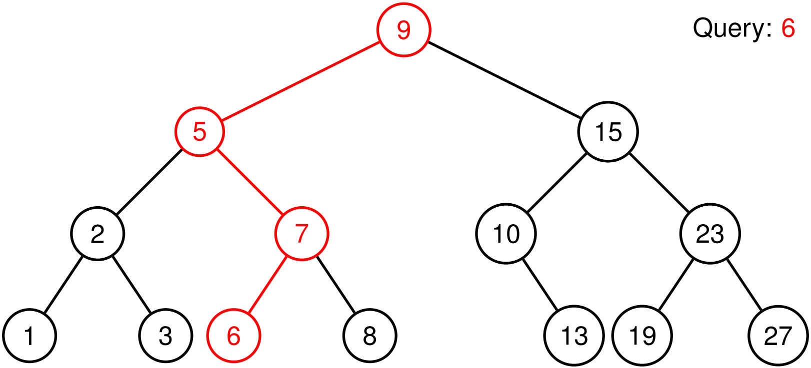
# BST Search



# BST Search



# BST Search



# Recursive BST Search

ALGORITHM **BSTSearch**$(T, k)$
// Recursive search in a binary tree.
// INPUT : Root node $T$ of a BST and a search key $k$.
// OUTPUT : A reference to the node containing $k$ or null.

**if** $T$ = null **or** $k = val(T)$ **then**
    **return** $T$
**end if**
**if** $k < val(T)$ **then**
    **return BSTSearch**$(T_L, k)$
**else**
    **return BSTSearch**$(T_R, k)$
**end if**

# BST Search - Complexity

Worst case complexity?

# BST Search - Complexity

Worst case complexity? $\mathcal{O}(n)$

# BST Search - Complexity

Worst case complexity? $\mathcal{O}(n)$

Can we do better?

# BST Search - Complexity

Worst case complexity? $\mathcal{O}(n)$

Can we do better? Yes if tree is balanced!

# BST Search - Complexity

Worst case complexity? $\mathcal{O}(n)$

Can we do better? Yes if tree is balanced!

Worst case complexity is dependent on the height of the tree.

# BST Search - Complexity

Worst case complexity? $\mathcal{O}(n)$

Can we do better? Yes if tree is balanced!

Worst case complexity is dependent on the height of the tree.

If tree is a “stick”, height = $n - 1$

# BST Search - Complexity

Worst case complexity? $\mathcal{O}(n)$

Can we do better? Yes if tree is balanced!

Worst case complexity is dependent on the height of the tree.

If tree is a "stick", height = $n - 1$

If tree is balanced, height = $\log_2 n$

# BST Insert

**Aim:** Insert key/value $k$ into tree $T$ (represented by its root node).

# BST Insert

**Aim:** Insert key/value $k$ into tree $T$ (represented by its root node).

**Idea:** Recursively traverse the tree to find a position for the key $k$ in $T$.

# BST Insert

**Aim:** Insert key/value $k$ into tree $T$ (represented by its root node).

**Idea:** Recursively traverse the tree to find a position for the key $k$ in $T$.

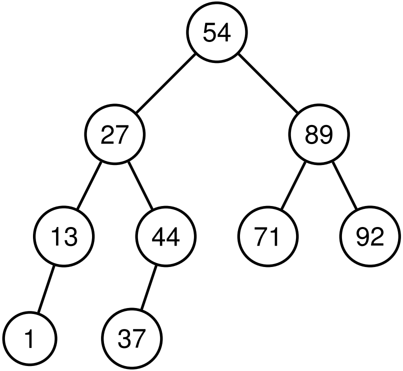
Insert($T$, $k$):

1. If $T$ is empty, place $k$ at this position.
2. If $k < val(T)$, traverse into the left subtree (Insert($T_L$, $k$)).
3. If $k > val(T)$, traverse into the right subtree (Insert($T_R$, $k$)).

**NOTE :** This is a variable-size-decrease algorithm, as each iteration we decrease the remaining problem size by a variable amount.

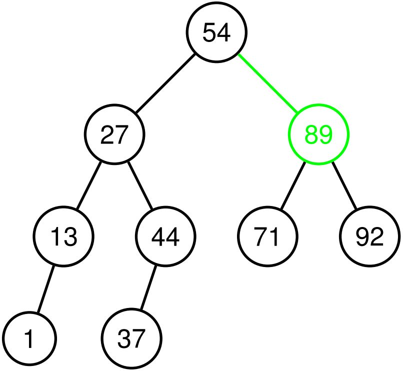
# BST Insert

Insert: **64**



# BST Insert

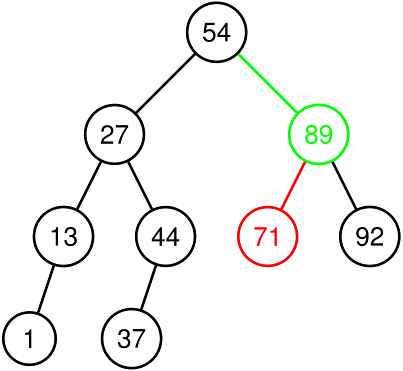
Insert: **64**




$64 > 54 \rightarrow$ right

# BST Insert

Insert: **64**

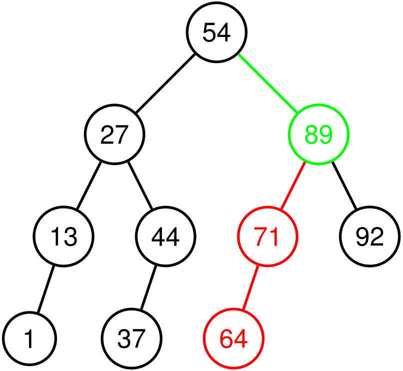



$64 > 54 \rightarrow$ right

$64 < 89 \rightarrow$ left

# BST Insert

Insert: **64**




$64 > 54 \rightarrow$ right

$64 < 89 \rightarrow$ left

$64 < 71 \rightarrow$ insert left

# BST Delete

**Aim:** Remove or delete a node from a BST.

# BST Delete

**Aim:** Remove or delete a node from a BST.

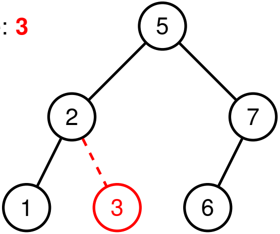
**Idea:** It can be solved using one of three cases:

- **Case 1** : The deleted node has no children.
- **Case 2** : The deleted node has one child.
- **Case 3** : The deleted node has two children.

**NOTE** : If need to, use BST search to find node to be deleted. This operation does not fall nicely into any of the algorithmic frameworks we study.

# BST Delete

Delete: **3**

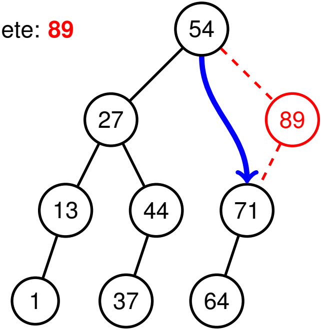



**Case 1** : Deleted node has no children

1. If the deleted node is the root of a single-node tree, make the tree empty.
2. If the deleted node is a leaf node, set the reference from its parent to the itself to null.

# BST Delete

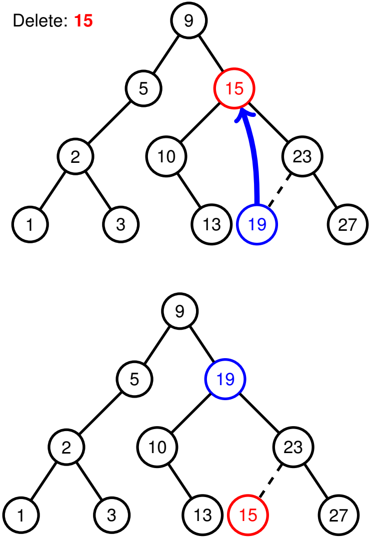
Delete: **89**




**Case 2** : Deleted node has one child

1. If the deleted node is the root node with a single child, make the child the new root.
2. If the deleted node is not the root, make the reference from its parent to point to its single child.

Delete: **15**




**Case 3**: Deleted node has two children

Use the following three-stage procedure:

1. First, find the node $k'$ which contains the smallest key in the right subtree.
2. Second, exchange the deleted node and node $k'$.
3. Third, remove the deleted node from its new position by using either **Case 1** or **Case 2**, depending on whether that node is a leaf (no children) or has a single child.

# Overview

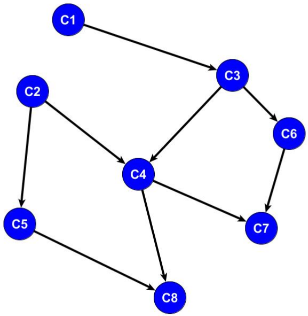
# Case Study - Problem

## Case Study Problem

Easy-as-123 University has used experienced course advisors to build valid study plans for their students. But they want to explore more automated approaches to help their advisors to quickly devise valid plans. Given courses and their pre-requisites, they want a tool that can produce valid study plans that a student can only study a course if they have satisfy the pre-requisites already.
They asked you to help them. How would you approach this problem?

# Case Study - Mapping the Problem to a Known Problem

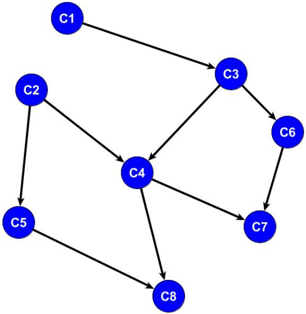
This can be mapped into a graph problem. Each course is a vertex in the graph. Each pre-requisite is a directed edge, from the pre-requisite to the course requiring it.



# Case Study - Solving the Problem

A valid plan is a vertex traversal and one that satisfies and respects all the edge directions. This is exactly the properties that a topological sort has.

We can use DFS or source removal algorithm to solve this. Is it possible to have more than one valid plan?



# Overview

# Summary

- Introduced the *Decrease-and-conquer* algorithmic approach.
- Decrease-by-a-constant algorithms (Insertion sorting and Topological sort).
- Decrease-by-a-constant-factor algorithms (Binary search, Fake coin).
- Variable-size decrease algorithms (Binary search tree).