

MTH251

Data Structures and Algorithms I

Zhong Lun

About Me

 <https://www.linkedin.com/in/zonglun/>

 zonglun@gmail.com

 9647 7009



Lun Zong

Lead Solutions Architect at GovTech
Singapore

Singapore · [Contact info](#)

500+ connections

Open to

Add section

More



GovTech Singapore



**National University of
Singapore**

Course Structure

6 weeks (Jan ~ Mar), 6 seminars, 6 labs:

1. Python, Complexity & Big O
2. Array, Stack, Queue
3. Linked List, Recursion, Binary Search
4. Tree
5. Algorithm Design & Pattern
6. Review

3 assignments & open book exam:

Assessment	Description	Weight Allocation
Assignment 1	Tutor-Marked Assignment 1	10%
Assignment 2	Tutor-Marked Assignment 2	10%
Assignment 3	Tutor-Marked Assignment 3	10%
Examination	Open book exam	70%
TOTAL		100%

Learning Objectives

1. python
2. algorithm time & space complexity
3. basic data structure: **array** **stacks** **queues** **list** **tree**
4. recursion algorithm

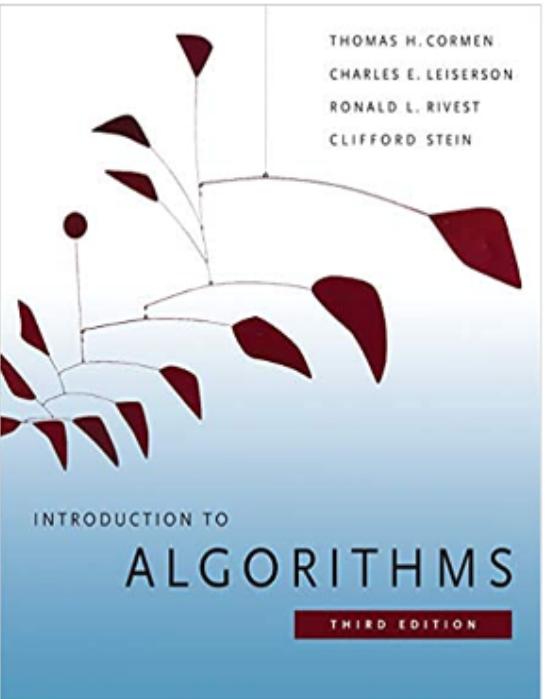
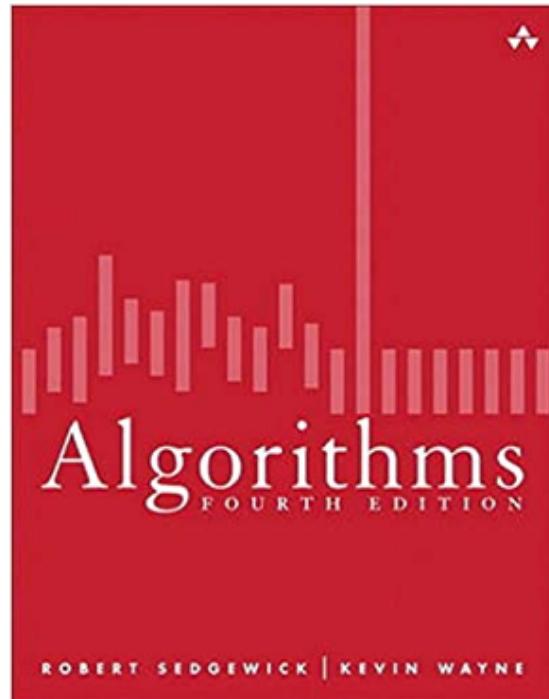
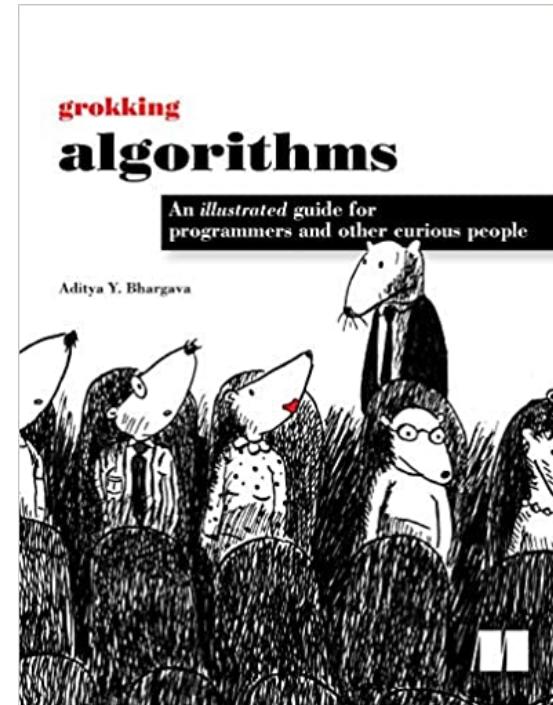
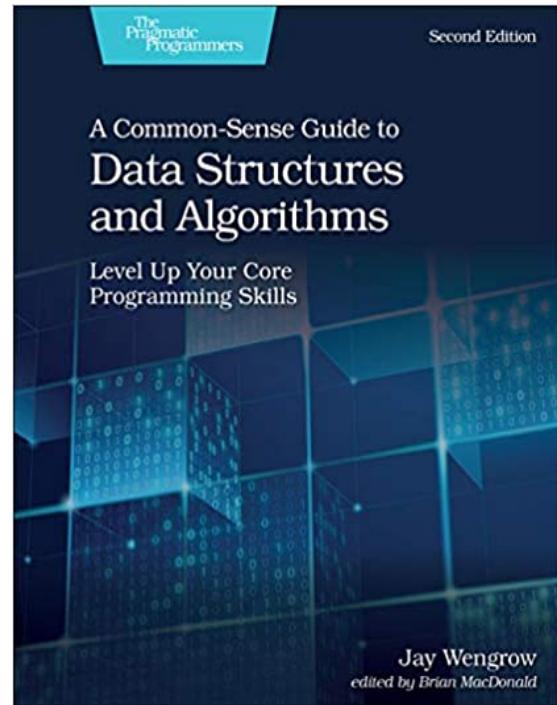
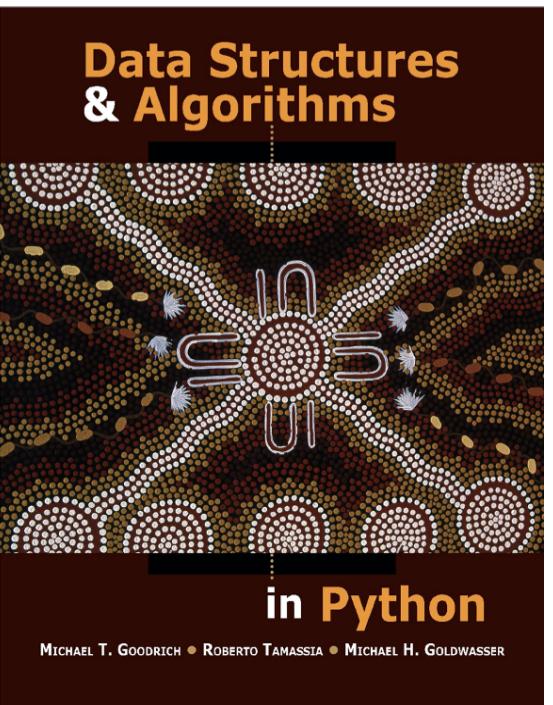
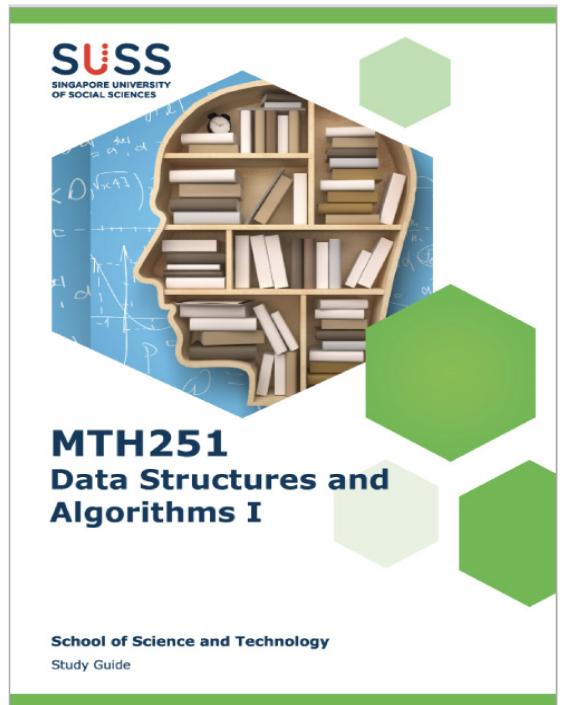
Learning Resource

slides online: <https://mth251.fastzhong.com/>

pdf:  <https://github.com/fastzhong/mth251/blob/main/mth251.pdf>

labs:  <https://github.com/fastzhong/mth251/tree/main/notebooks>

Learning Resource



Learning Resource

A screenshot of the Leetcode website's "Learn" section. The page displays a grid of 12 learning modules, each represented by a card. The modules are categorized into three main groups: Data Structures, Algorithms, and Machine Learning. Each card includes the module title, the number of chapters and items, and a progress bar indicating completion status. A "Premium" badge is visible at the top right of the grid.

Module Category	Title	Chapters	Items	Completion (%)
Data Structures	Detailed Explanation of Graph	6	56	Locked
	Introduction to Data Structure Arrays 101	6	31	0%
	Introduction to Data Structure Linked List	5	30	0%
	Introduction to Data Structure Binary Tree	3	19	0%
Algorithms	Introduction to Algorithms Recursion I	5	21	0%
	Introduction to Algorithms Recursion II	4	25	0%
	Basic Concepts in ML Machine Learning 101	3	9	0%
	Learning about... Binary Search	8	30	0%
Machine Learning	Introduction to Data Structure N-ary Tree			
	Introduction to Data Structure Binary Search Tree			
	Introduction to Data Structure Trie			
	Introduction to Data Structure Hash Table			

Why Python

The TIOBE Programming Community index is an indicator of the popularity of programming languages.

Oct 2021	Oct 2020	Change	Programming Language	Ratings	Change
1	3	▲	 Python	11.27%	-0.00%
2	1	▼	 C	11.16%	-5.79%
3	2	▼	 Java	10.46%	-2.11%
4	4		 C++	7.50%	+0.57%
5	5		 C#	5.26%	+1.10%
6	6		 Visual Basic	5.24%	+1.27%
7	7		 JavaScript	2.19%	+0.05%
8	10	▲	 SQL	2.17%	+0.61%

Why Python

- ✓ Easy To Learn
- ✓ Human Readable
- ✓ Productivity
- ✓ Cross Platform

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Readability counts.

Special cases aren't special enough to break the rules.

There should be one-- and preferably only one --obvious way to do it.

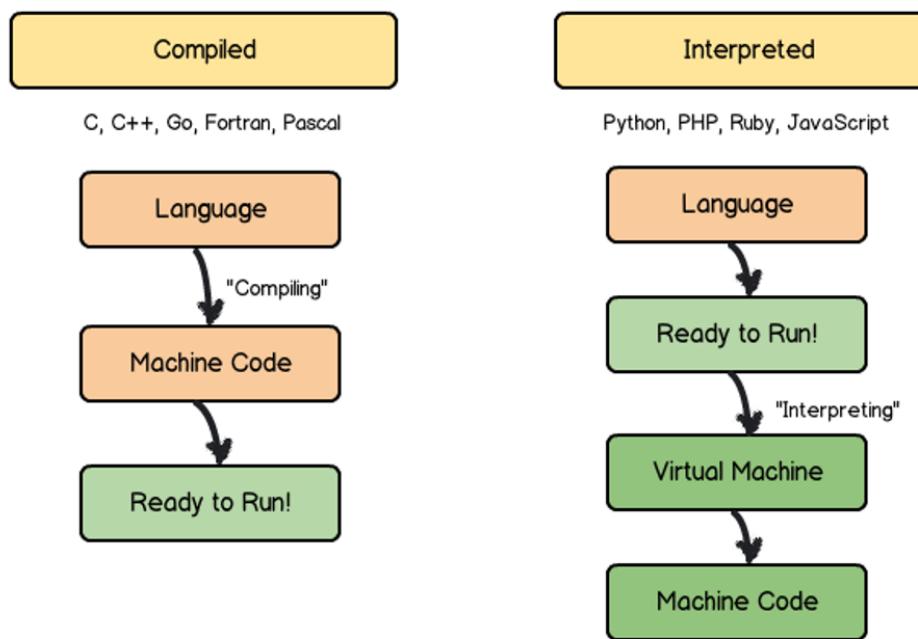
If the implementation is hard to explain, it's a bad idea.

Python Jobs

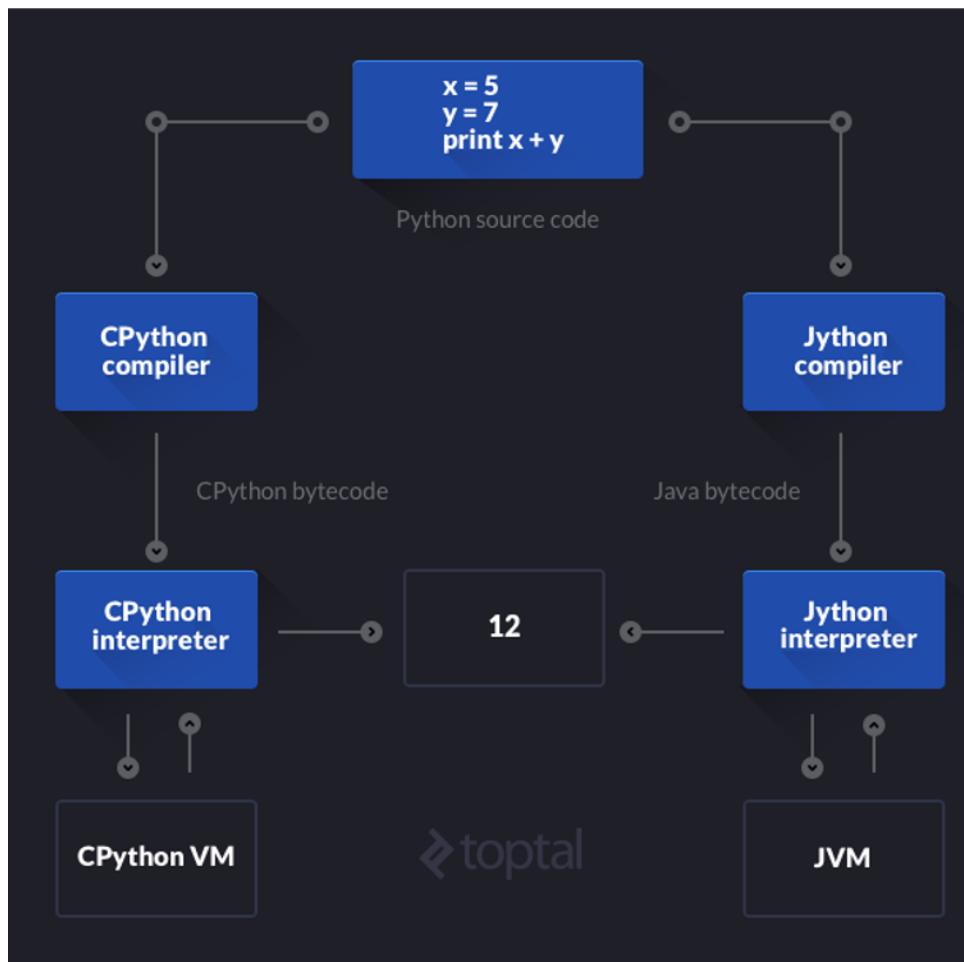
- ✓ backend: **Python** vs. Java, C++, Go, Php
- ✓ devops: **Python** vs. Go, Ruby, Shell
- ✓ test automation: **Python** vs. Groovy, shell
- ✓ data engineering: **Python** vs. Java, C++
- ✓ data analytics & visualization: **Python** vs. R, Java, C++
- ✓ data science & machine learning: **Python** vs. R, Julia, C++

Python 101

- Compiled vs. Interpreted



- CPython bytecode



- Python implementations

Implementation	Virtual Machine	Ex) Compatible Language
CPython	CPython VM	C
Jython	JVM	Java
IronPython	CLR	C#
Brython	JavaScript engine (e.g. V8)	JavaScript
RubyPython	Ruby VM	Ruby

Python Data Type & Operators



- numbers: `int` `float` `complex`
 - arithmetic operator: `+` `-` `*` `/` `//` `%` `**`
 - bitwise operator: `&` `|` `^` `>>` `<<` `~`
 - `range()`: a list of integers
- strings: `''` `'''` `''` `"` `\t` `\n` `\r` `\\"` etc.
 - `join()` `split()` `ljust()` `rjust()` `lower()` `upper()` `lstrip()` `rstrip()` `strip()` etc.
- boolean: `True` `False`
 - `True`: non-zero number, non-empty string, non-empty list
 - `False`: `0`, `0.0`, `""`, `[]`, `None`

Python Data Type & Operators



- boolean: `True` `False`
 - logic operator: `and` `or` `not`
 - comparison operator: `>` `<` `>=` `<=` `==` `!=`
 - identity operator: `is` `is not`
- None
- type conversion/casting: `int()` `float()` `str()` `bool()` `hex()` `ord()`

Python Collections



- collections: `list` `tuple` `set` `dictionary`
 - membership operator: `in` `not in`
- `list []`: a collection of items, usually the items all have the same type
 - sequence type
 - sortable
 - grow and shrink as needed
 - most widely used
- `tuple ()`: a collection which is ordered and unchangeable
- `set {}`: a collection which is unordered and unindexed

Python Program Structure



- variable
- statement & comments
 - Python uses new lines to complete a command, as opposed to other programming languages often use ; or () ; relies on indentation (whitespace sensitive), to define scope, such as the scope of loops, functions and classes, as opossed to other programming languages often use {}
- control flow
 - if ... elif ... else
 - while for break continue

Python Program Structure



- function

- `def` `return`
- `main`
- advanced:
 - `lambda`
 - `decorator`
 - `closure`

- error/exception

- handling exception: `try ... except ... else ... finally`
- raise exception: `raise`

Python OO & Class

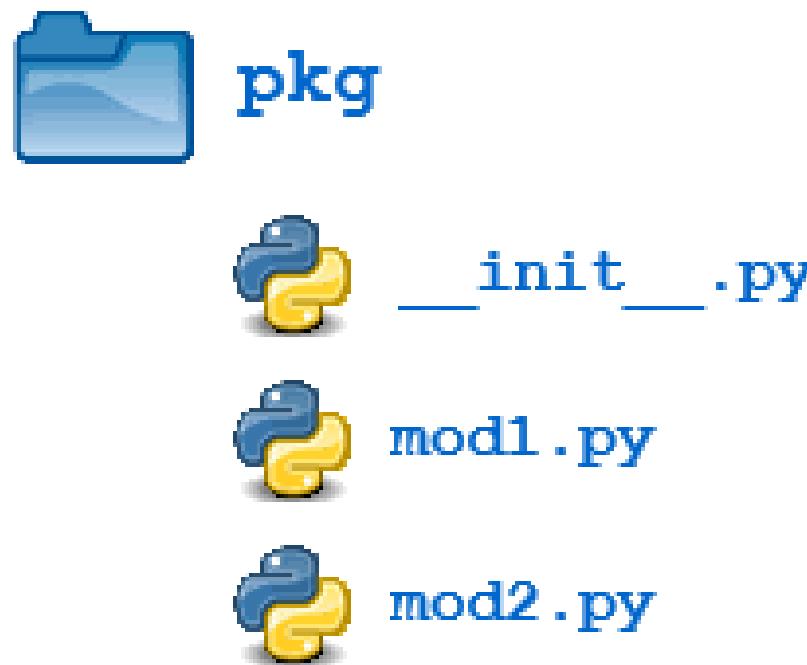


- Procedural vs. OOP vs. FP
- OO Principal
 - Inheritance
 - Encapsulation
 - Polymorphism
- class, instance, attributes, properties, method
- override vs. overload vs. overwrite

Misc.



- modular programming: function → class → module → package
- Modules: Python module (default main module), C module, Build-in module
- Packages:



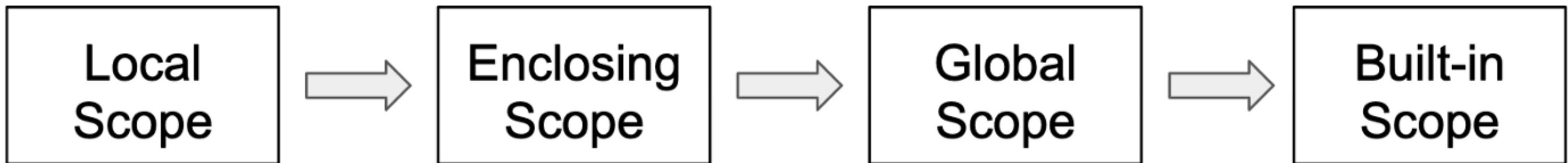
- Standard Lib: math random re os itertools collections

Misc.



- Namespaces & Scopes: `LEGB` rule

The LEGB Rule



- help

Python Tutorials

Programming with Mosh

- ▶ [Python Tutorial - Python for Beginners 2020](#)

freeCodeCamp

- ▶ [Learn Python - Full Course for Beginners Tutorial](#)
- ▶ [Python for Everybody - Full University Python Course](#)
- ▶ [Intermediate Python Programming Course](#)

Tech With Tim

- ▶ [Learn Python - Full Course for Beginners Tutorial](#)

CheatSheet:

-  [Python Crash Course - Cheat Sheets](#)
-  [Comprehensive Python Cheatsheet](#)

Data Structure & Algorithms

A data structure is a way of organizing information so that it can be used effectively by computer

Algorithms provides computer step by step instructions to process the information and solve a problem

Program = Data Structure + Algorithm

- Finiteness
- Definiteness
- Effectiveness
- Input
- Output

Data Structure & Algorithms

Example1:

Data Structure & Algorithms

Example2:

Data Structure & Algorithms

Data Structure

- Linear
 - Array, String, Linked List
 - Stack, Queue, Deque, Set, Map/Hash, etc.
- Non-Linear
 - Tree, Graph
 - Binary Search Tree, Red-Black Tree, AVL, Heap, Disjoin Set, Trie, etc.
- Others
 - Bitwise, BloomFilter, LRU Cache

Algorithms & Data Structure

Algorithms

- branching: if-else, switch
- iteration: for, while loop
- recursion: divide & conquer, backtrace
- searching: binary search, depth first, breath first, A*, etc.
- sorting: quick sort, bubble sort, merge sort, etc.
- dynamic programming
- greedy
- ...

Algorithms & Data Structure

Why

- ✓ deeper understanding of computer system
- ✓ improve coding skill
- ✓ coding interview
- ✓ building framework and library

How

 learning by doing, implementing from scratch

 problem solving

Algorithm Complexity Analysis

Performance

- cpu, memory, io, networking, etc.
- worst case, avg case, best case
- `data.size()`
- no. of lines
- ...

Algorithm Complexity Analysis

- **Time Complexity** : by giving the size of the data set as integer N, consider the number of operations that need to be conducted by computer before the algorithm can finish
- **Space Complexity** : by giving the size of the data set as integer N, consider the size of extra space that need to be allocated by computer before the algorithm can finish
- When: Accessing, Searching, Inserting, Deleting

Big-O

Big-O describes the trend of algorithm performance when the data size increases

$O(1)$: constant complexity

$O(\log_* n)$: logarithmic complexity

$O(n)$: constant complexity

$O(n^2)$: N square complexity

$O(2^n)$: exponential complexity

$O(n!)$: factorial

Big-O

👉 Master theorem (analysis of algorithms)

$$O(f) = f$$

$$O(c \cdot f) = O(f)$$

$$O(f + g) = O(\max(f, g))$$

$$O(f) \cdot O(g) = O(f \cdot g)$$

$$O(f \cdot g) \leq O(f \cdot h) \text{ if & only if } O(g) \leq O(h)$$

$$O(x^a) \leq O(x^b) \text{ if & only if } a \leq b$$

$$O(a^x) < O(b^x) \text{ if & only if if } a < b$$

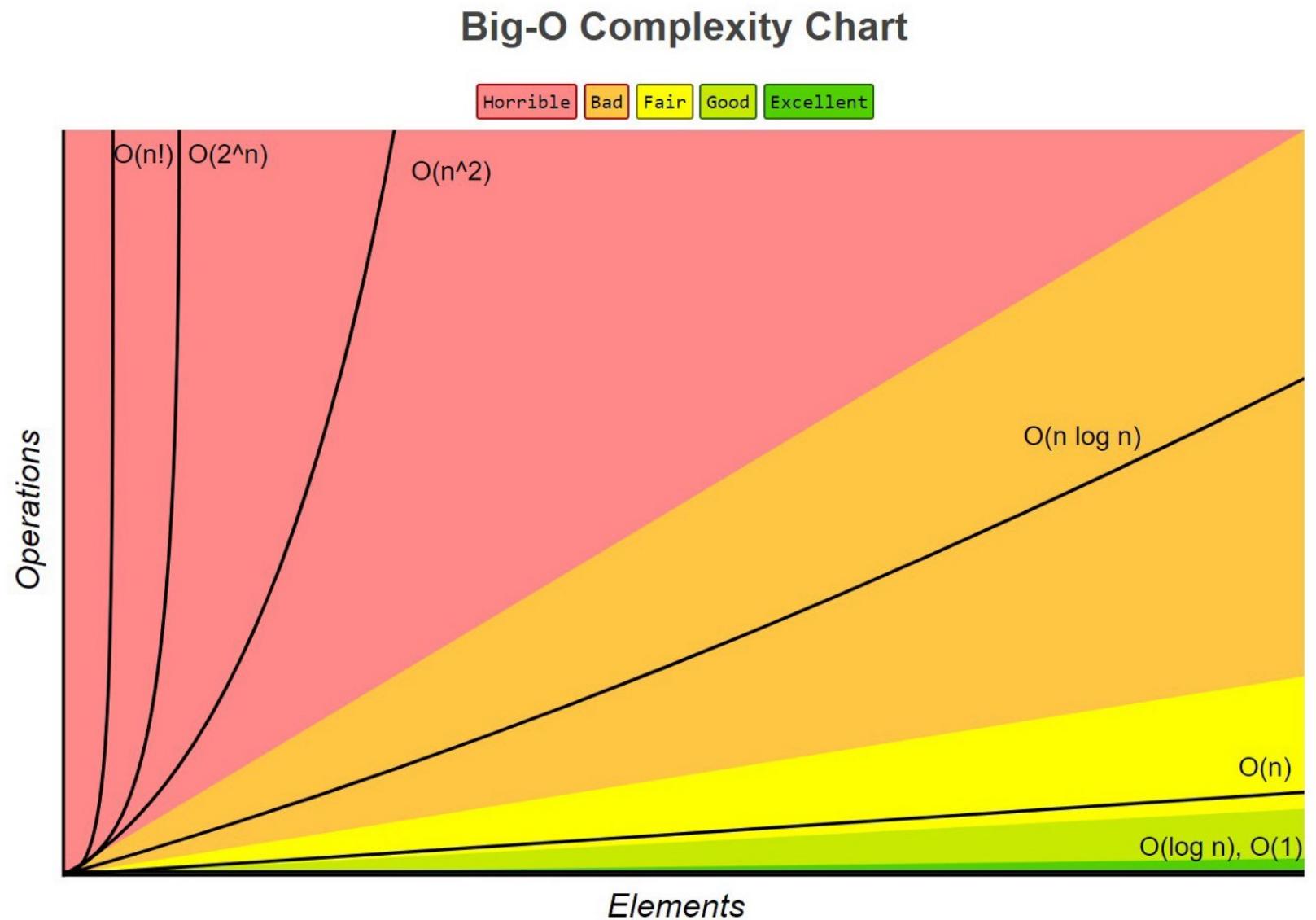
$$O(x^a) < O(b^x) \text{ if & only if if } d > 1 \text{ (assuming } c \geq 1 \text{ and } d \geq 1\text{)}$$

$$O(\log_* x) < O(x^c) \text{ if & only if if } c > 0$$

Big-O

↳ <https://www.bigocheatsheet.com/>

$$O(1) < O(\log_* n) < O(n) < O(n \log_* n) < O(n^2) < O(2^n) < O(n!)$$



Array

To store a list of similar things, example:

A list of names: ["Alex", "Bob", "Charles", "David"]

A list of numbers: [1, 2, 3, 4]

Each item in the array referred as “**element**”

Array

- Element Type: same type (array is structured data)
- Element Size: fixed

```
1 # java
2 String[] cars = {"BMW", "Toyota", "Tesla"} // declare & init
3
4 Integer[] scores = new Integer[10] // declare
5 // init
6 scores[0] = 90
7 scores[1] = 80
```

- Element Index: 0, 1, ..., length - 1

Array 2-D

```
students = [  
    ["Alex", "M", "S111111A"],  
    ["Bob", "M", "S222222B"],  
    ["James", "M", "S333333C"],  
]
```

students[2] → ["James", "M", "S333333C"]
Students[1][2] → "S222222B"

Index	0	1	2
0	Alex	M	S111111A
1	Bob	M	S222222B
2	James	M	S333333C

Array Address

str = "HELLO" = ['H', 'E', 'L', 'L', 'O']

Memory Address	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	...
Computer's memory	H	E	L	L	L	O				...	
Index Number	0	1	2	3	4						

data type: char

data type size: 2 byte (1 byte = 8 bits, 0000 0000 ~ 1111 1111)

total_size = array_size * data_type_size

array[i].address = base_address + i * data_type_size

👉 O(1)

Array Operations

Operation	Array	Dynamic Array
Accessing	O(1)	O(1)
Searching	O(n)	O(1)
Inserting	-	O(n)
Deleting	-	O(n)

ADT vs. Data Structure

An **abstract data type** (ADT) is an abstraction of a **data structure** which provides only the interface to which a data structure must adhere to. The interface does not give any specific details about how something should be implemented.

Programming language provides different **data types** to implement/represent different data structure.

i.e. Array

- a linear abstract data type
- a java data type

Dynamic Array

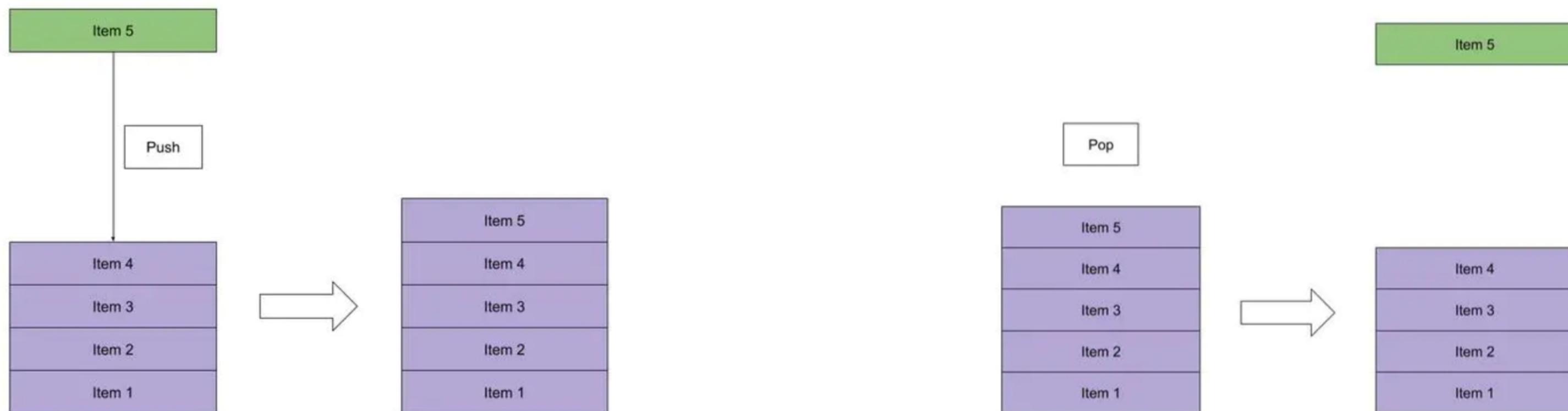
Stack

- Sequential Access vs Random Access (such as Array)
- **LIFO** (Last In First Out) sequential collection



Stack: Operations

- **push()** – pushing (storing) an element on the stack
- **pop()** – removing (accessing) an element from the stack
- **top()/peek()** – get the top data element of the stack, without removing it
- **size(), isEmpty(), isFull()**



Stack Operations

Operation	Stack
Accessing	$O(n)$
Searching	$O(n)$
Inserting	$O(1)$ (push)
Deleting	$O(1)$ (pop)

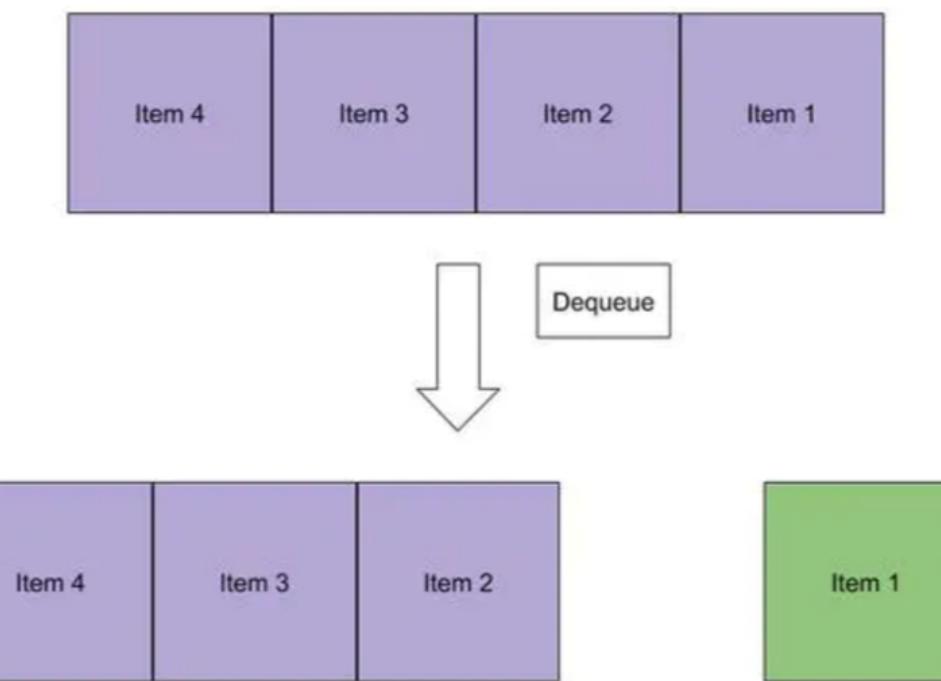
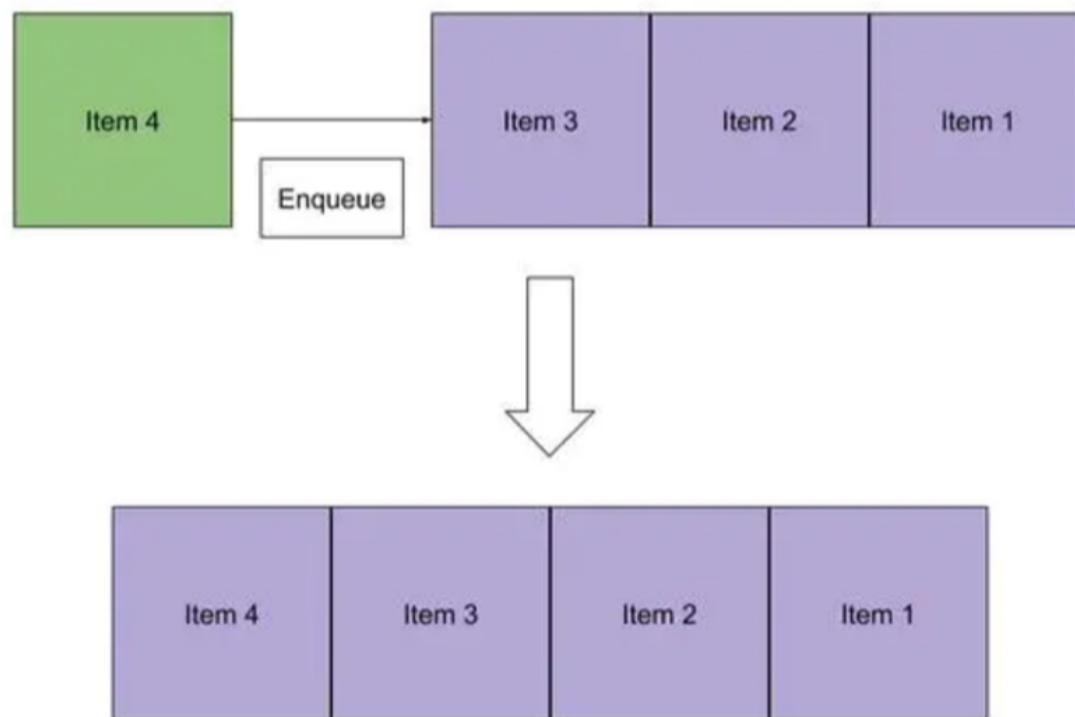
Queue

- **FIFO** (First In First Out) sequential collection



Queue: Operations

- **enqueue()** – adding (storing) an element to the queue
- **dequeue()** – removing (accessing) an element from the queue
- `fist()/peek()` – get the first element of the queue, without removing it
- `size()`, `isEmpty()`, `isFull()`

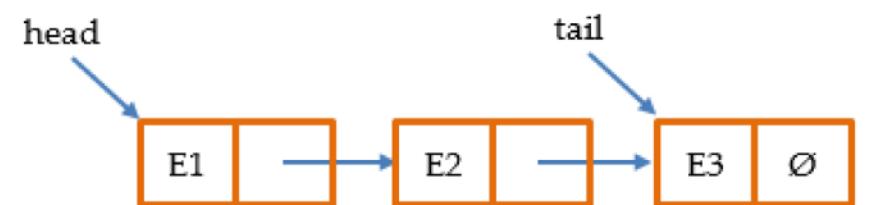


Queue Operations

Operation	Queue
Accessing	$O(n)$
Searching	$O(n)$
Inserting	$O(1)$ (enqueue)
Deleting	$O(1)$ (dequeue)

Linked List

- dynamic linear data structure
- data stored in a “Node” class
- data & pointer



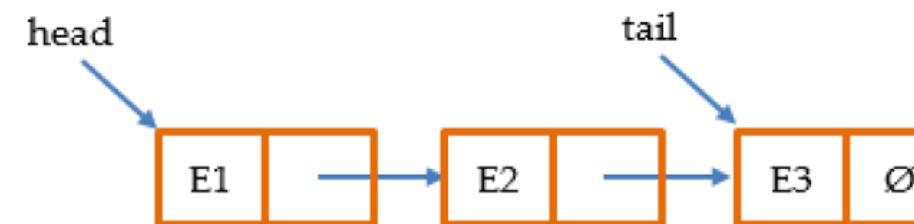
code here

Linked List Operations

Operation	Linked List	Dynamic Array
Accessing	O(n)	O(1)
Searching	O(n)	O(n)
Inserting	O(1)	O(n)
Deleting	O(1)	O(n)

Linked List

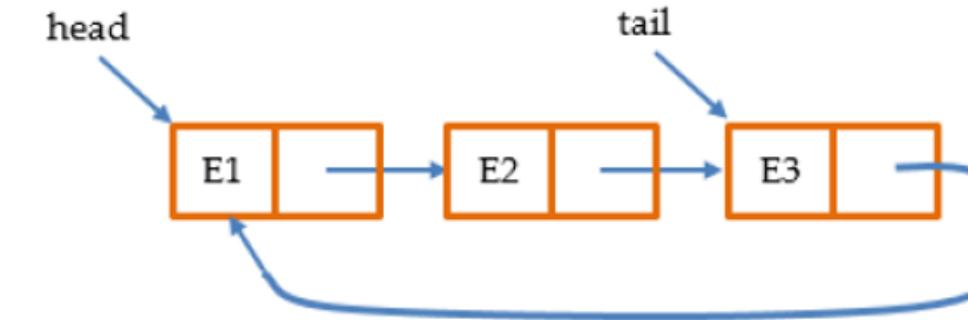
Singly Linked List



Doubly Linked List



Circular Linked List



Positional Linked List



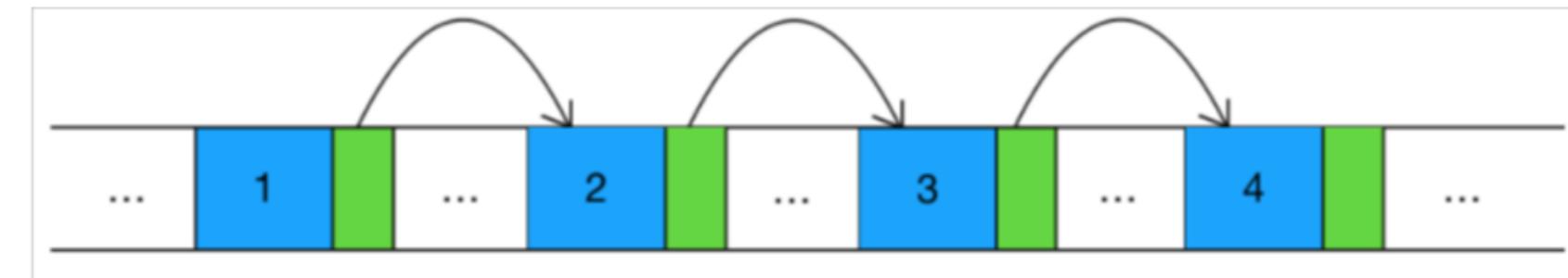
Linked List vs. Array

- ✓ dynamic, no need to deal with fixed memory size
- ✗ accessing speed

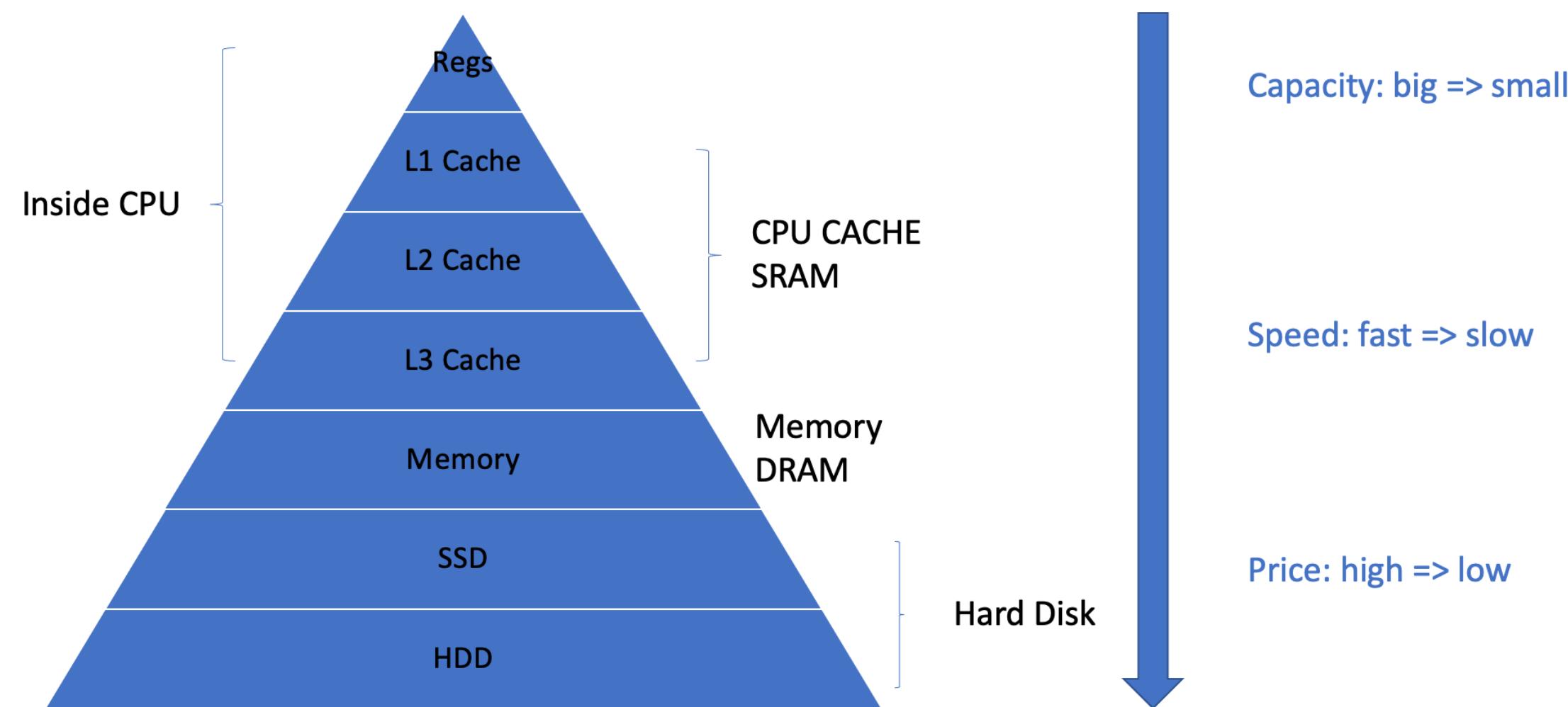
Array:

Memory Address	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	...
Computer's memory	H	E	L	L	O					...	
Index Number	0	1	2	3	4					...	

Linked List:



Linked List vs. Array



Recursion

Recursion by definition is a function that calls itself.

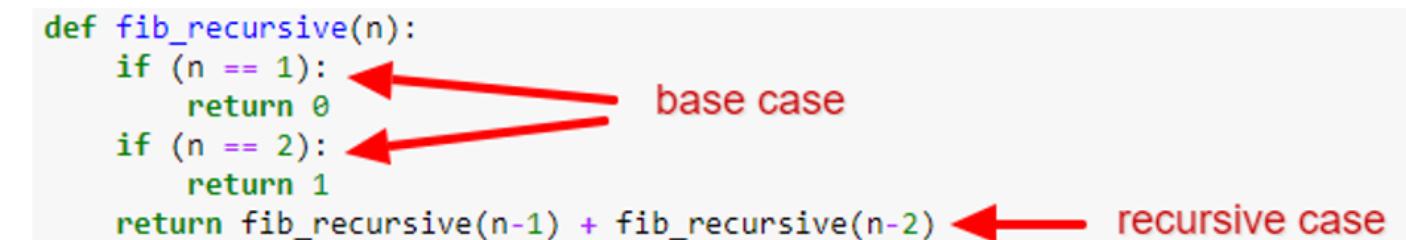
- base case
- recursive case

Example:

Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, ...

- when $n = 1$, $\text{fib}(1) = 0$
- when $n = 2$, $\text{fib}(2) = 1$
- when $n > 2$, $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$

```
def fib_recursive(n):
    if (n == 1):
        return 0
    if (n == 2):
        return 1
    return fib_recursive(n-1) + fib_recursive(n-2)
```



The code snippet shows a recursive function for calculating the nth Fibonacci number. It uses two conditional statements to handle the base cases for n=1 and n=2, returning 0 and 1 respectively. For n > 2, it calls itself twice with arguments n-1 and n-2, and adds the results. Red arrows point from the text labels 'base case' and 'recursive case' to the corresponding parts of the code: the base cases are indicated by the arrows pointing to the first two 'return' statements, and the recursive case is indicated by the arrow pointing to the final line where the function calls itself.

Recursion vs. Iterative

Anything with a recursion can be done iteratively (loop)

😊 Intuitive/DRY, code readability

```
def fib_recursive(n):
    if (n == 1):
        return 0
    if (n == 2):
        return 1
    return fib_recursive(n-1) + fib_recursive(n-2)
```

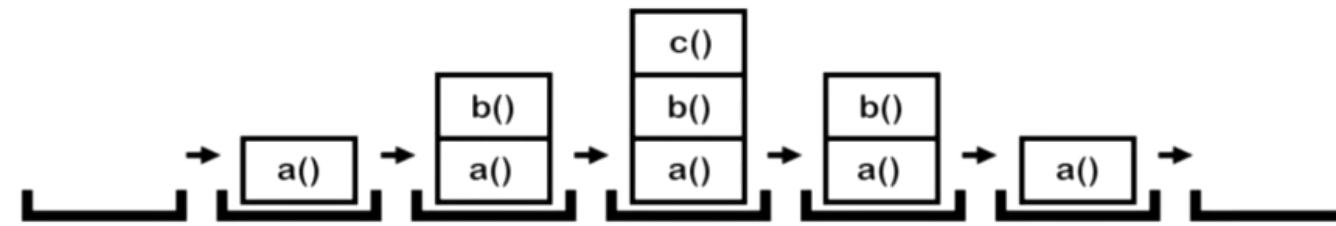
base case
recursive case

🤔 Optimization, call stack

```
def fib_iterative(n):
    fib = [0,1]
    i = 2
    while (i < n): # index: 2, 3, ..., n-1
        fib.append(fib[i-1] + fib[i-2]) # fib[i]
        i += 1
    return fib[n-1] # the nth number
```

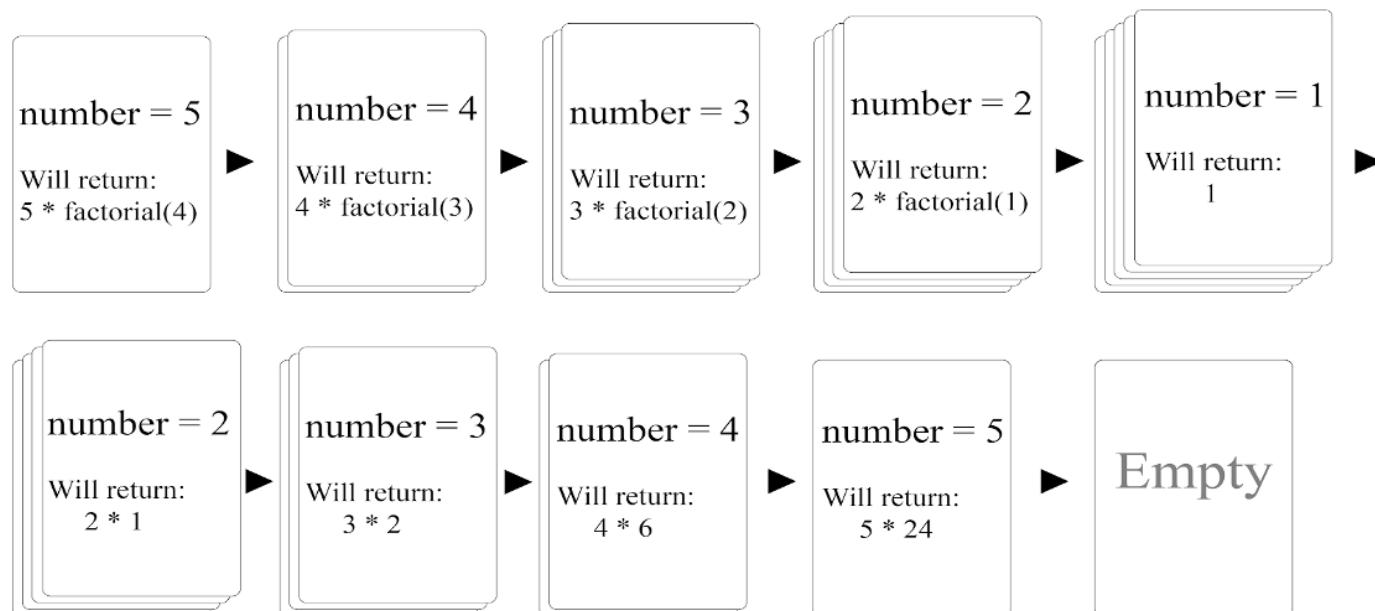
Recursive Call

Call Stack:



The “call stack” is a stack of “frame objects”.
(frame object == a function call)

fib_recursive(5):



Recursive Call

- Max call stack size (stack overflow error)
- Tail Call Optimization
- Memorization

Recursive Call

Fundamental technique to solve problem:

- Identifying the base case
- Identifying the recursion formula/equation to transform the problem to smaller version
 - Problem requires back-tracking
 - Problem has tree structure

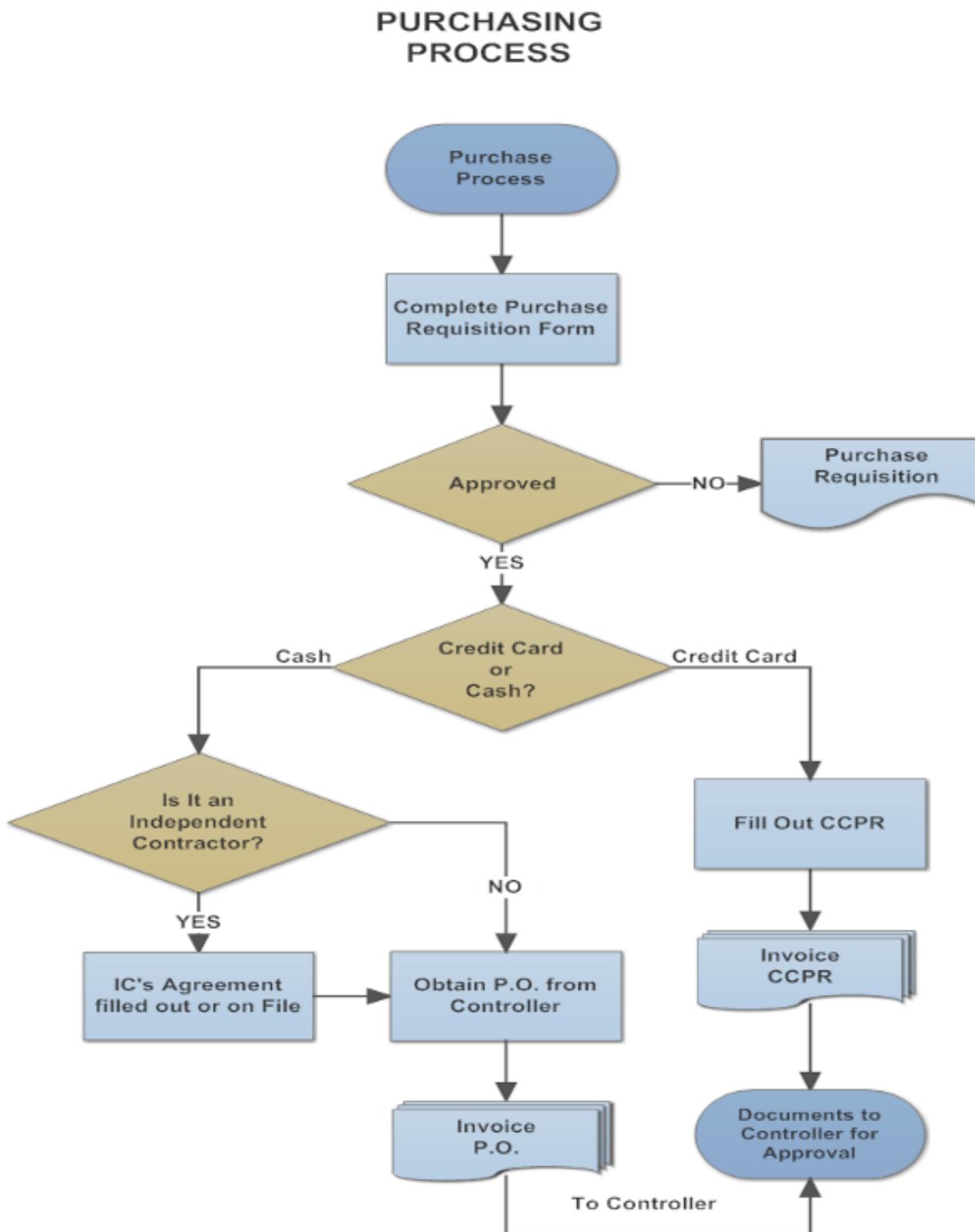
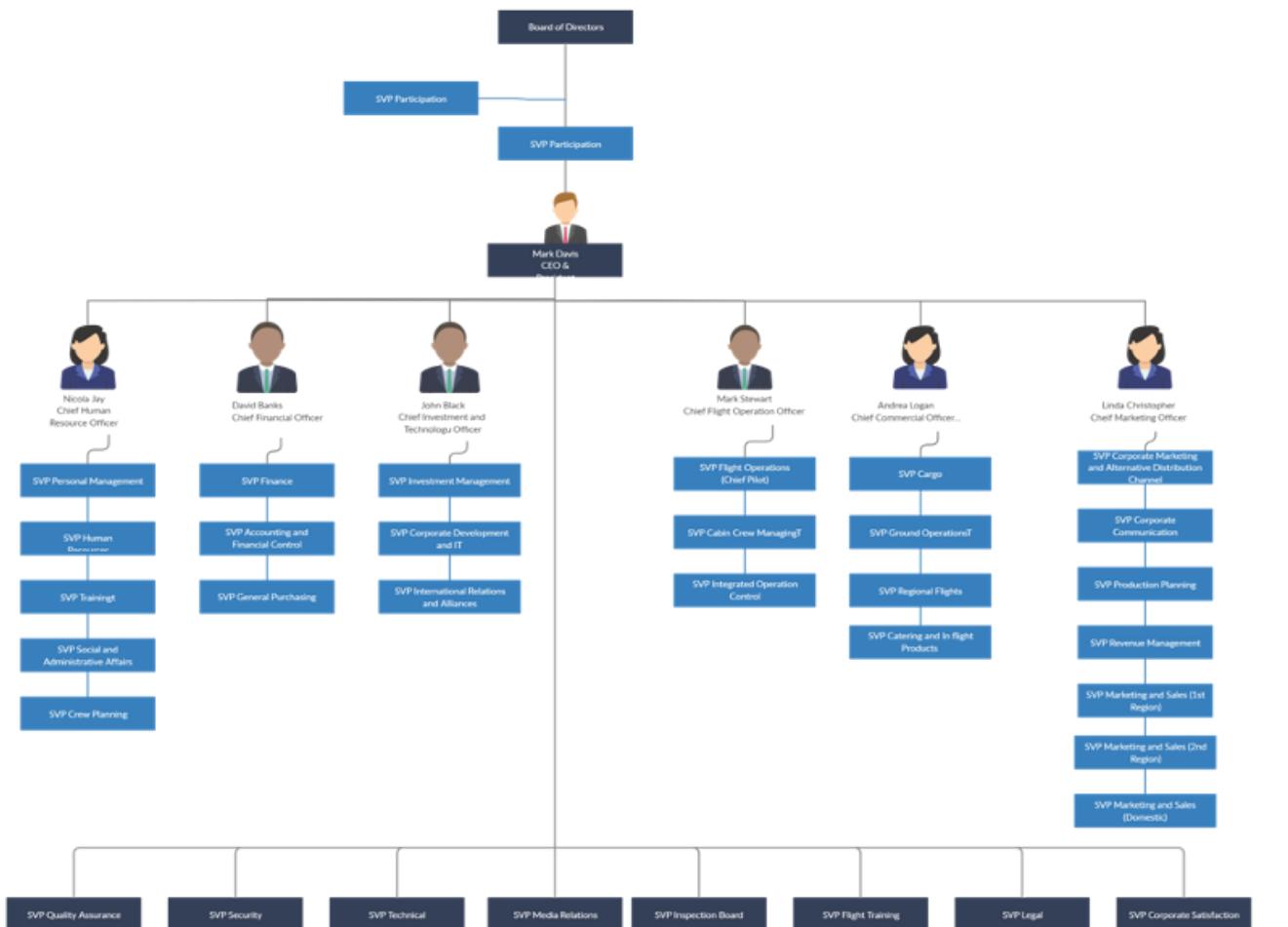
Linear Search

- Input: array, target element
- Output: position (-1 if not existing)

Binary Search

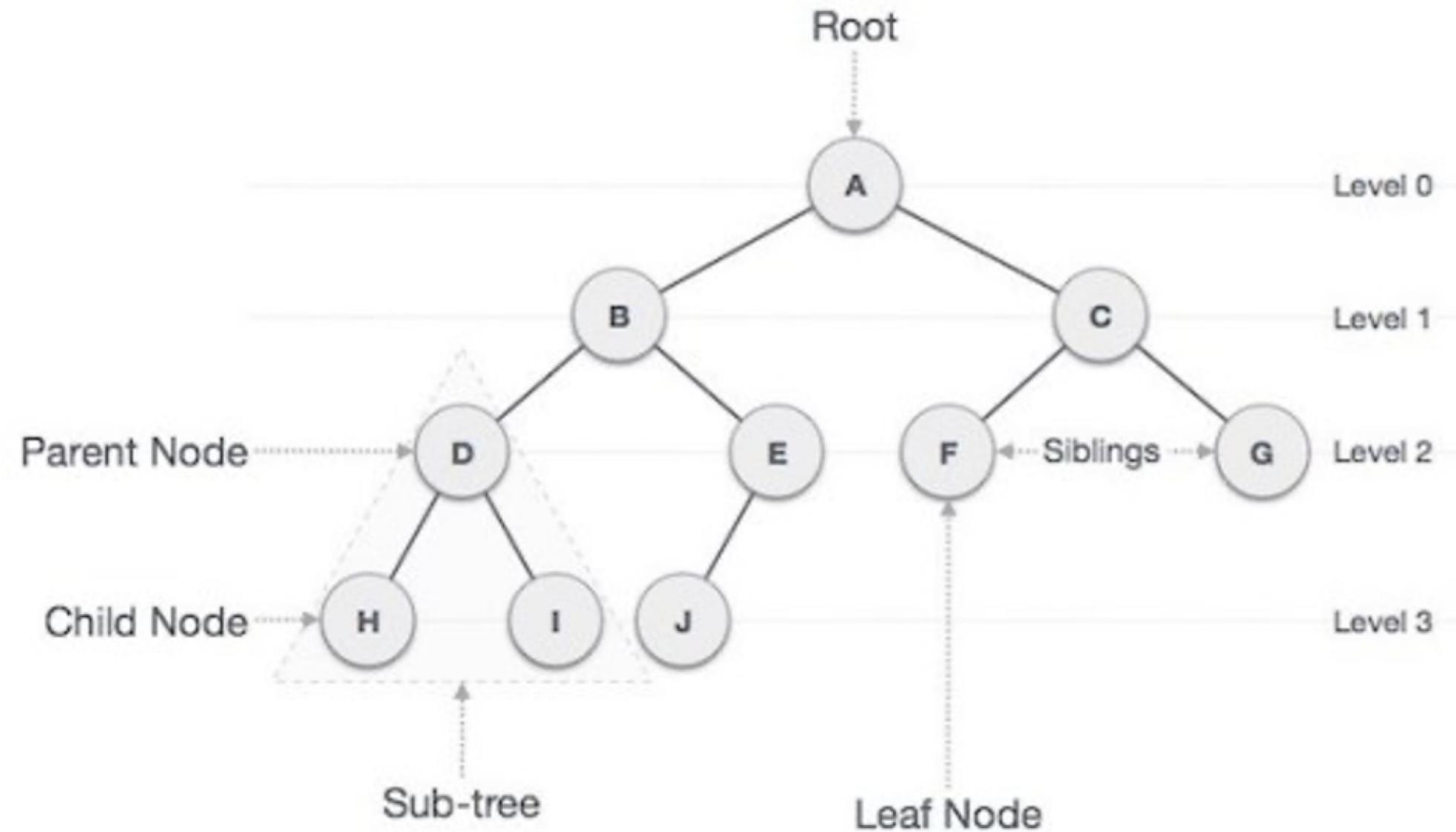
- Input: array, target element
- Output: position (-1 if not existing)

Tree



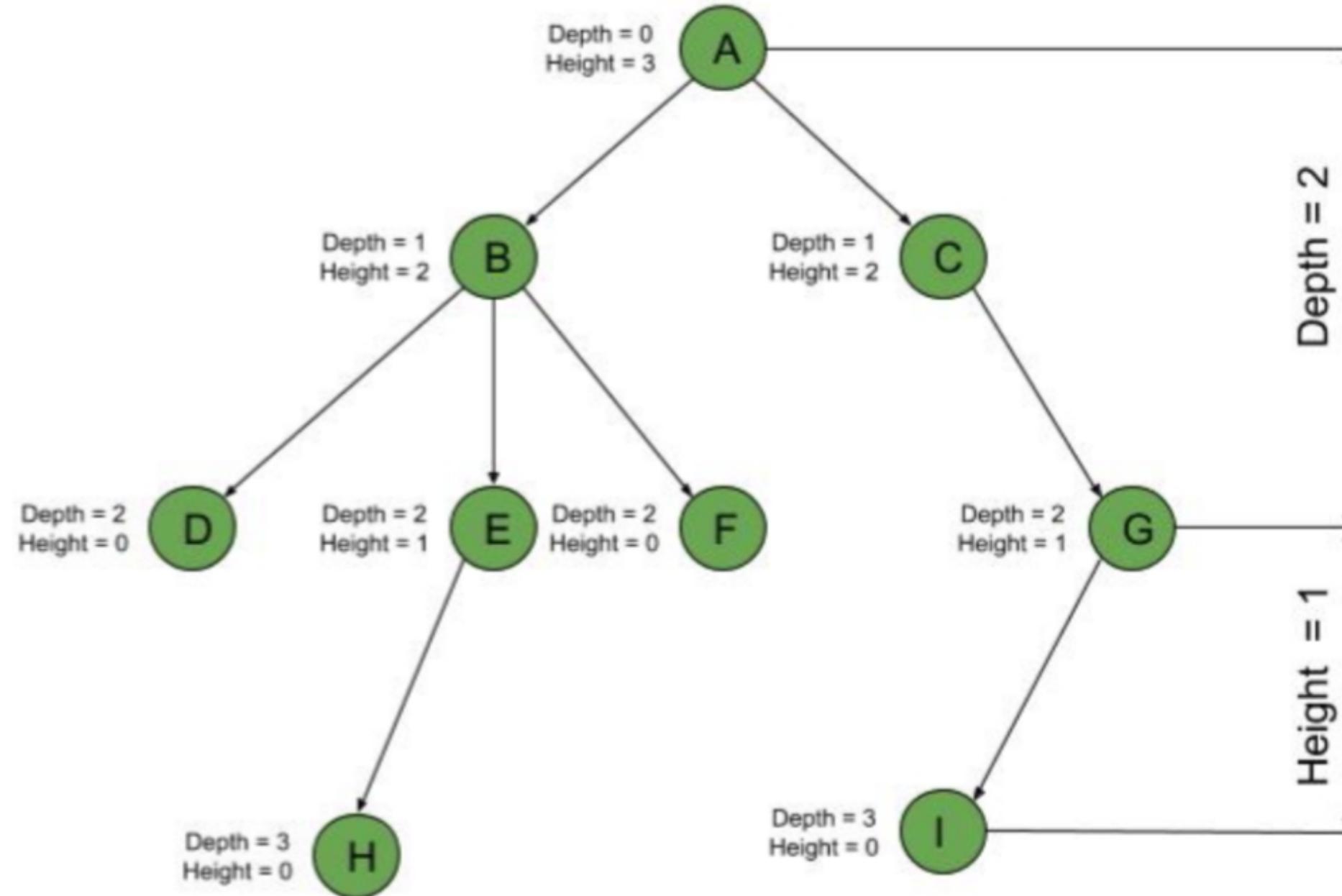
Tree Terminology

- Node: Root, leaf, Internal Node
- Parent, Children, Sibling
- Edge, Degree
- SubTree
- Path
- Level



Tree Terminology

- Level vs. Depth vs. Height



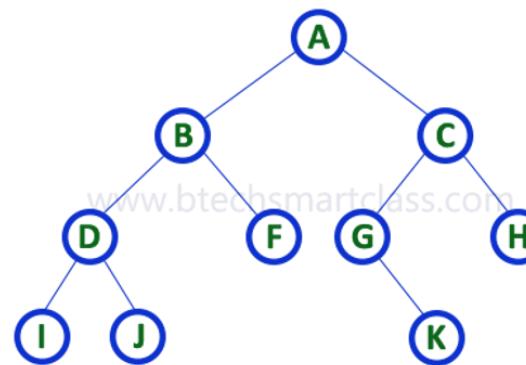
Binary Tree

- One one root
- Max 2 child nodes
- One and only one path from root to each node
- Max nodes on level: 2^l
- Max nodes total: $2^{h+1} - 1$

Binary Tree

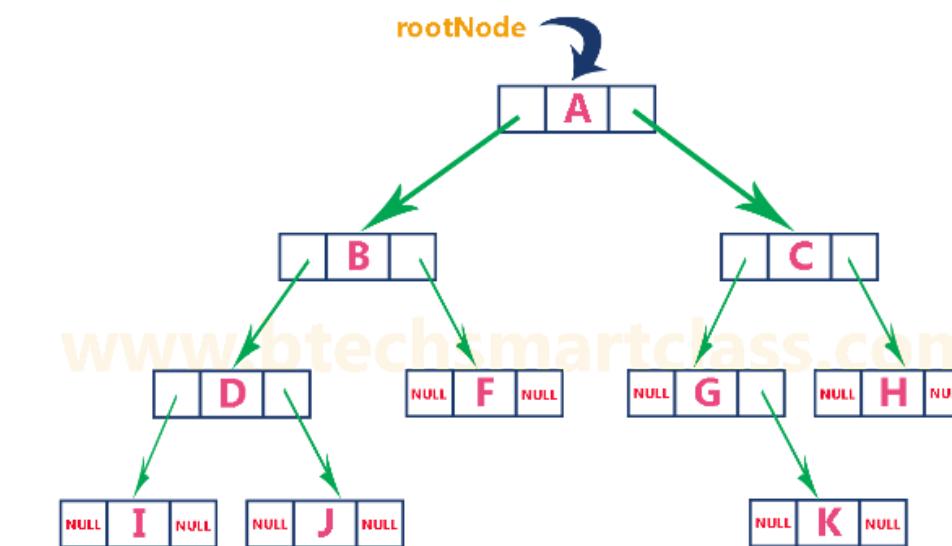
- Array

A B C D F G H I J - - - K - - - - - - - -



- Left/Right Linked List

Left Child Address	Data	Right Child Address
--------------------	------	---------------------

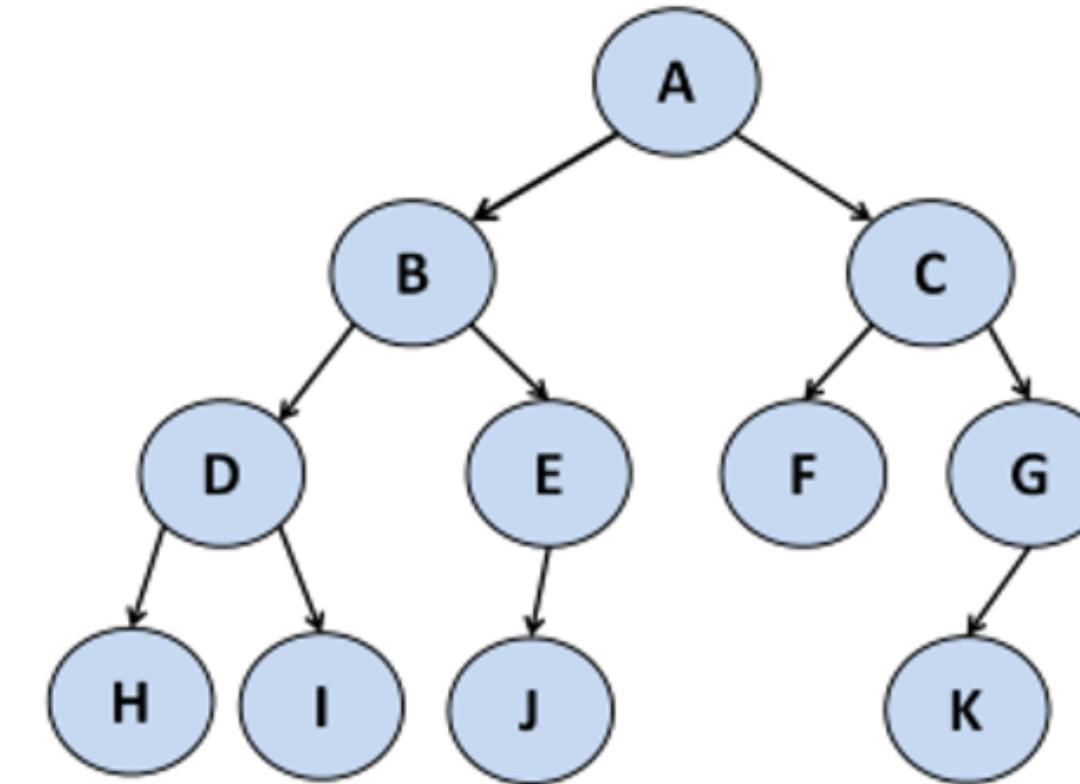


Binary Tree Traverse (DFS): pre-order

ROOT → Left → Right:

1. Visit the root
2. Traverse the left subtree
3. Traverse the right subtree

A B D H I E J C F G K

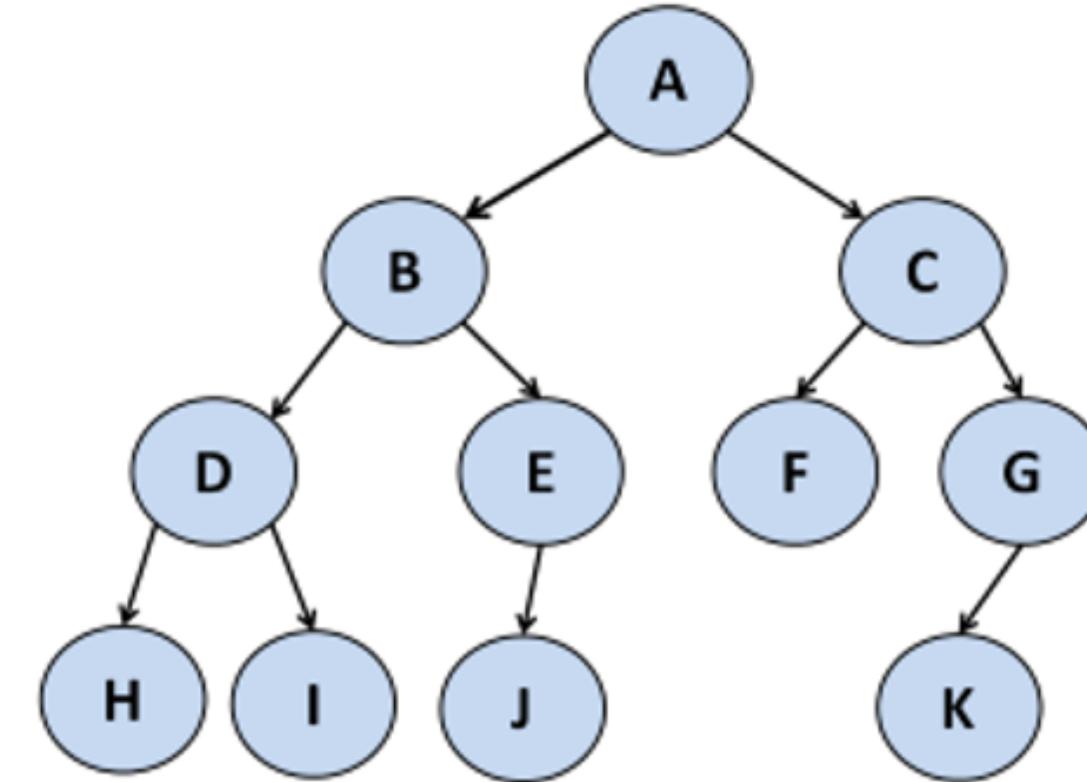


Binary Tree Traverse (DFS): in-order

Left → Root → Right:

1. Traverse the left subtree
2. Visit the root
3. Traverse the right subtree

H D I B J E A F C K G

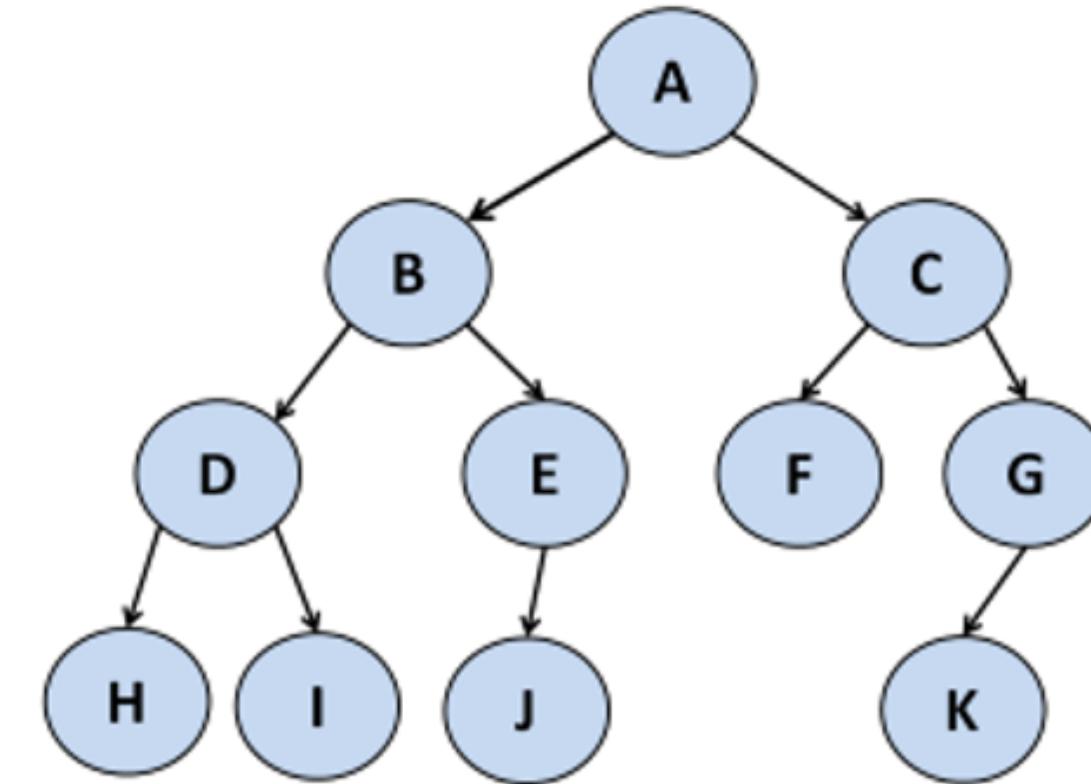


Binary Tree Traverse (DFS): post-order

Left → Right → Root:

1. Traverse the left subtree
2. Traverse the right subtree
3. Visit the root

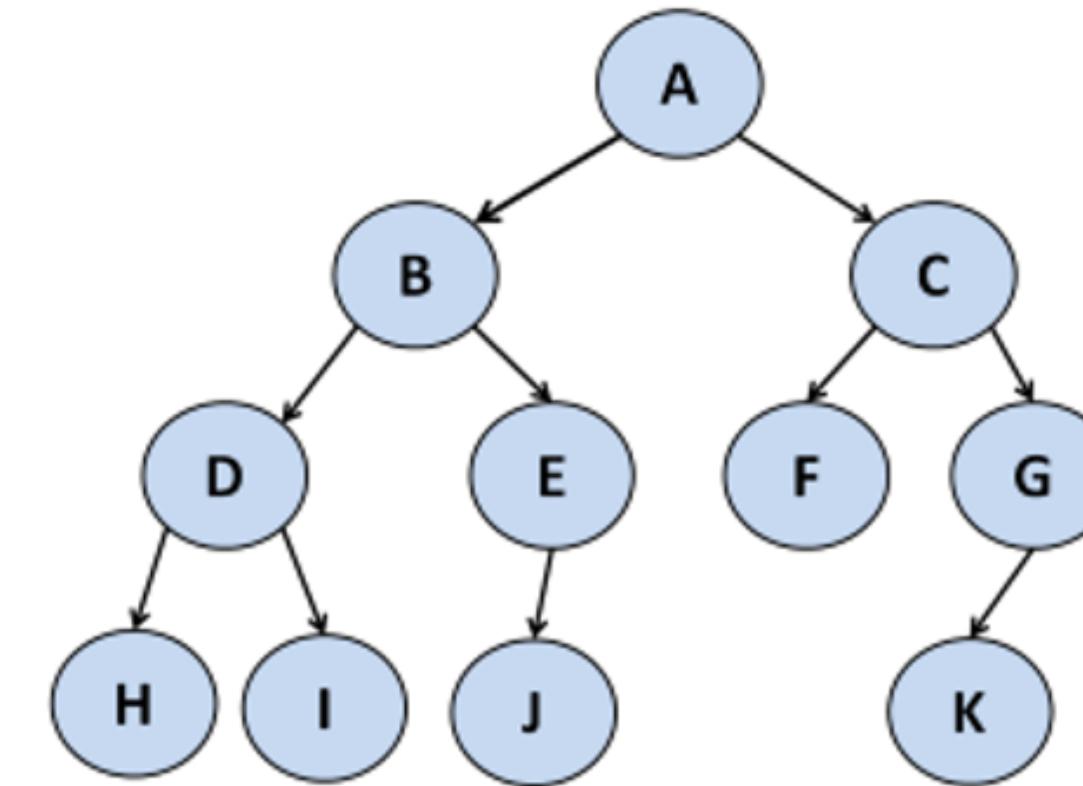
H I D J E B F K G C A



Binary Tree Traverse (BFS): level order

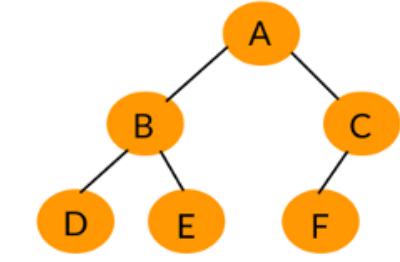
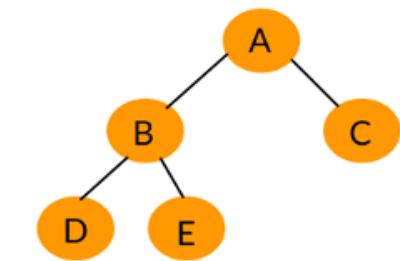
1. Visit the root
2. Visit the left node
3. Visit the right node
4. Go to next level

A B C D E F G H I J K

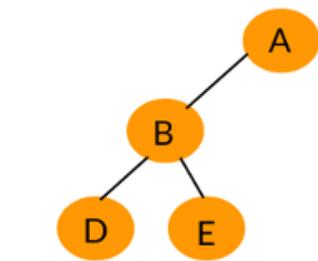


Binary Tree

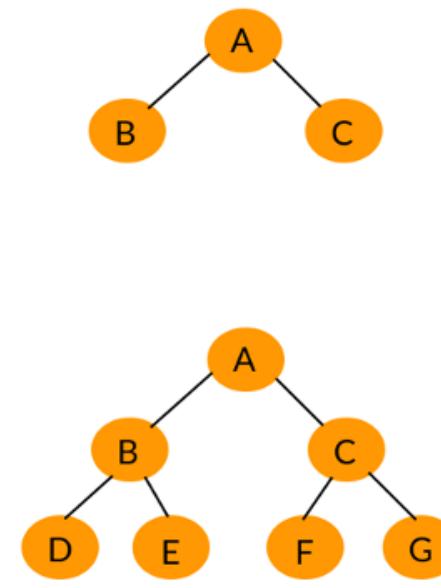
- Complete Tree: every level is completely filled except the last (leaf) and all nodes are as far left as possible
- Full Binary Tree: every node has two child nodes except leaf
- Perfect Binary Tree: every node has two child nodes except leaf and all leaves on same level



Complete Binary Tree

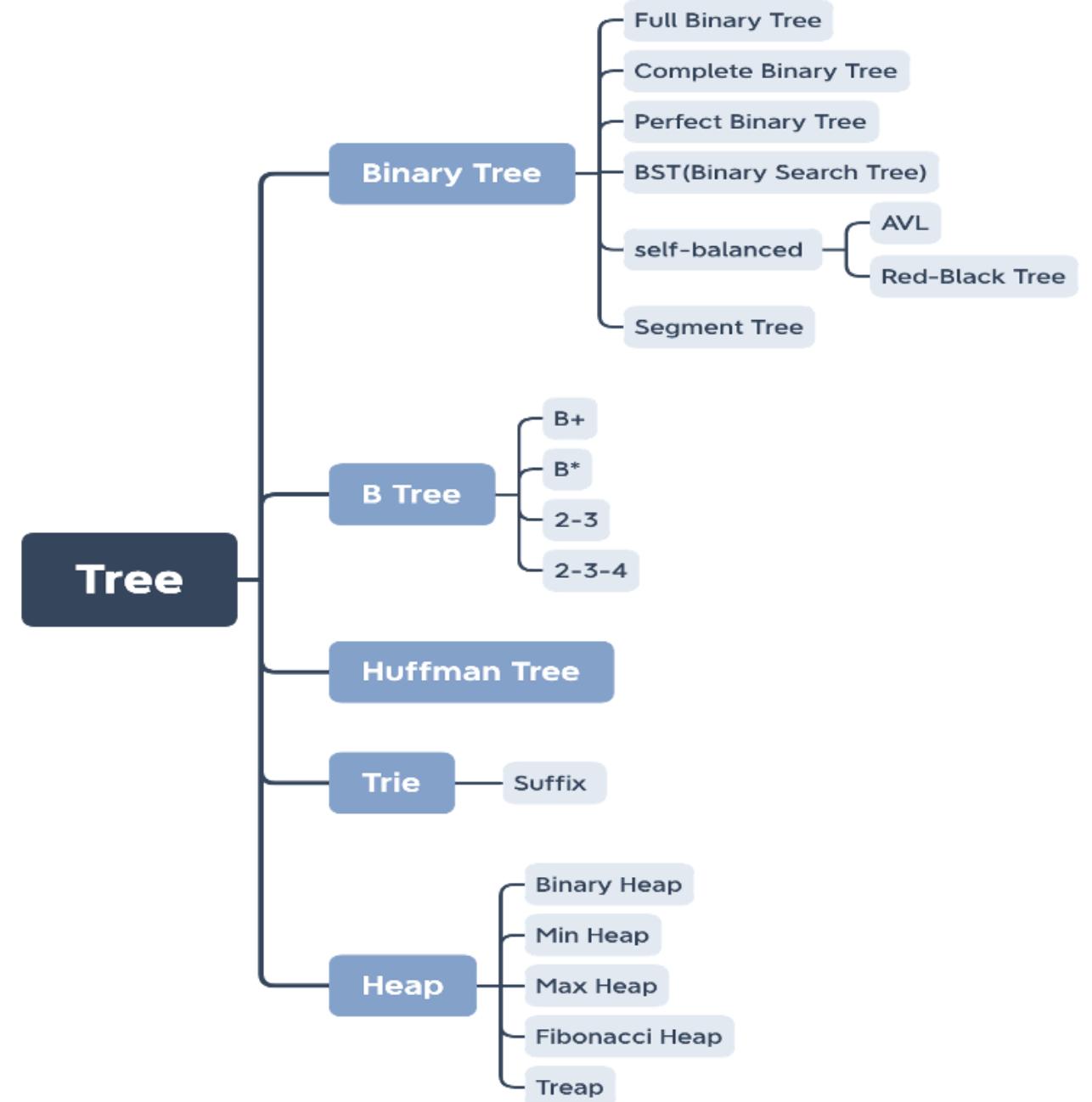


Full Binary Tree



Perfect Binary Tree

Trees



layout: center

Summary

Array

Concept	<ul style="list-style-type: none">consecutive memory space: $\text{arr}[i].\text{address} = \text{base_address} + i * \text{data_type_size}$same data type → same size for each elementfixed length
Operations	Accessing O(1) , Searching O(n)
Notes	<ul style="list-style-type: none">not memory friendlycpu cacheableindex from 0fundamental data structure to implement others such as stack, queue, heapdata type (programming language) vs. data structure
Hands-on	dynamic array, stack/queue, binary search, etc.

Stack

Concept	LIFO/FILO
Operations	Accessing O(n), Searching O(n), Inserting/push O(1), Deleting/pop O(1)
Notes	Stack implementation by dynamic array or linked list
Hands-on	function call stack, expression matching, etc.

Queue

Concept	FIFO/LIFO
Operations	Accessing O(n), Searching O(n), Inserting/enqueue O(1), Deleting/dequeue O(1)
Notes	Queue implementation by dynamic array or linked list
Hands-on	priority queue, circular queue, job queue, resource pool, etc.

Linked List

Concept	<ul style="list-style-type: none">nonconsecutive memory spacenode: data + pointerSingle Linked List, Doubly Linked List, Circular Linked List, Positional Linked List
Operations	Accessing O(n), Searching O(n), Inserting O(1), Deleting O(1)
Notes	<ul style="list-style-type: none">accessing slower than arraywith/without head/tail node (which don't store any data)fundamental data structure to implement others such as skip list, hash table, etc.
Hands-on	stack, queue, traverse/reverse/update/merge, etc.

Binary Tree

Concept	<ul style="list-style-type: none">■ one root■ max 2 child nodes■ height & depth■ 4 traversal (DFS/BFS): in-order(left-root-right), pre-order(root-left-right), post-order(left-right-root), level-order■ proper, perfect, full, complete binary tree
Operations	<ul style="list-style-type: none">■ DFS: time $O(n)$, space $O(h)$■ BFS: time $O(n)$, space $O(n)$
Notes	stored in array or linked nodes
Hands-on	4 traversal

SUSS

SINGAPORE UNIVERSITY
OF SOCIAL SCIENCES

LAB



Lab1



- download and install Anaconda
- create and Activate your Anaconda Python env
- (Optional) install and setup VS Code
- familiar yourself with Python and do exercise [lab1.ipynb](#)

Lab1

Download and install Anaconda

<https://www.anaconda.com/products/individual>

The screenshot shows the Anaconda Individual Edition landing page. At the top, there's a navigation bar with links for Products, Pricing, Solutions, Resources, Blog, Company, and a prominent 'Get Started' button. Below the navigation, the page features a large green 'Q' icon and the text 'Individual Edition'. The main headline reads 'Your data science toolkit'. A descriptive paragraph explains that the Individual Edition is the easiest way to perform Python/R data science and machine learning on a single machine, developed for solo practitioners. It highlights over 20 million users worldwide and thousands of open-source packages and libraries. A 'Download' button is located below this text. At the bottom, there are three links: 'Open Source' (with a green snake icon), 'Conda Packages' (with a brown box icon), and 'Manage Environments' (with a server icon).

Lab1

Create and Activate your Anaconda Python env

Lab1

Create and Activate your Anaconda Python env

💬 Also possible to perform via command line:

```
1 > # create the env
2 > conda create -n mth251 python=3.8
3 > # activate the env
4 > conda activate mth251
5 > # install jupyter
6 > conda install -c conda-forge notebook
7 > # multipledispatch for lab1
8 > conda install -c anaconda multipledispatch
9 > # start jupyter notebook
10 > jupyter notebook
```

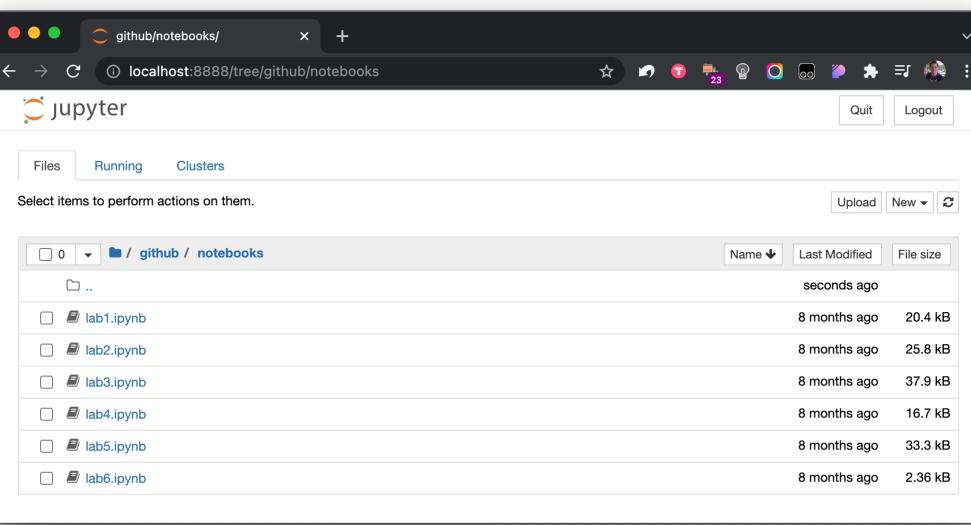
- copy & paste the Jupyter link in the prompt to your browser
- Control-C to stop Jupyter from the command line

Lab1

Create and Activate your Anaconda Python env

6. Now you are ready to create, edit and run

Jupyter notebooks (lab1.ipynb):



Lab1

Install and setup VS Code

VS Code now fully integrated with Jupyter notebook, refer to this link:

 [Jupyter Notebooks in VS Code](#)

Lab1

lab1.ipynb

- Python data type
- Python program structure
- OO

Lab2



- review Array, Stack, Queue
- exercise [lab2.ipynb](#)
- priority queue
- circular queue

Lab2

Exercise 1: two sum

Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

Example 1

Input: `nums = [2,7,11,15]`, `target = 9`

Output: `[0,1]`

Because `nums[0] + nums[1] == 9`, we return `[0, 1]`

Example 2

Input: `nums = [3,2,4]`, `target = 6`

Output: `[1,2]`

Example 3

Input: `nums = [3,3]`, `target = 6`

Output: `[0,1]`

Lab2

Exercise 2: remove duplicate numbers

Given a sorted array `nums`, remove the duplicates in-place such that each element appears only once and returns the new length.

Do not allocate extra space for another array, you must do this by modifying the input array in-place with O(1) extra memory.

Example 1

Input: `nums = [1,1,2]`

Output: `2, nums = [1,2]`

Explanation: Your function should return length = 2, with the first two elements of `nums` being 1 and 2 respectively. It doesn't matter what you leave beyond the returned length.

Example 2

Input: `nums = [0,0,1,1,1,2,2,3,3,4]`

Output: `5, nums = [0,1,2,3,4]`

Explanation: Your function should return length = 5, with the first five elements of `nums` being modified to 0, 1, 2, 3, and 4 respectively. It doesn't matter what values are set beyond the returned length.

Lab2

Priority Queue is similar to queue but the element with higher priority can be moved forward to the front.
Use existing Queue class to implement a priority queue (element with lower value has higher priority).

- Priority Queue can be used in Printer Jobs or Schedule Tasks.

Lab2

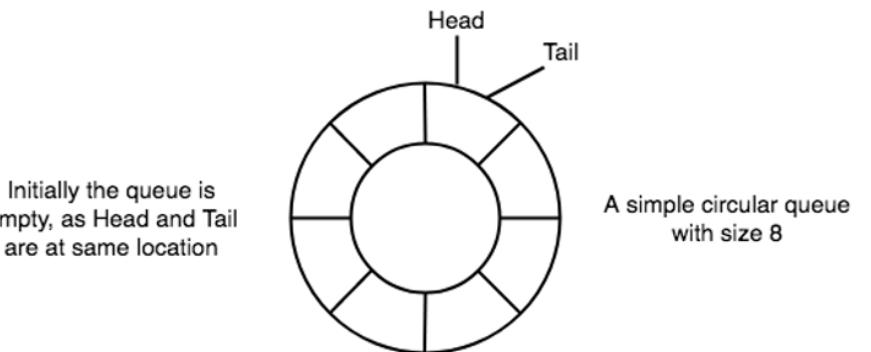
Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

Design your implementation of circular queue.

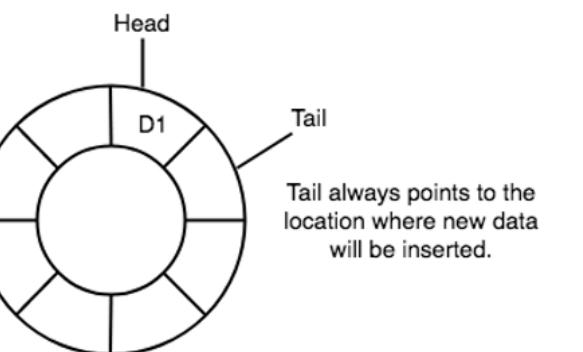
Lab2

Circular Queue

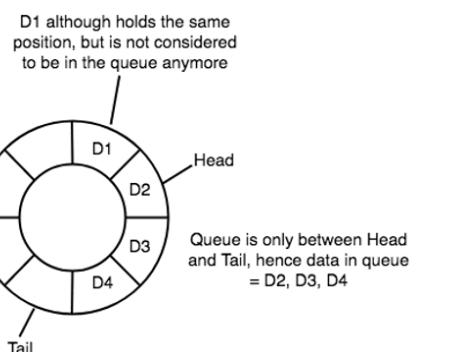
1. init



2. enqueue D1



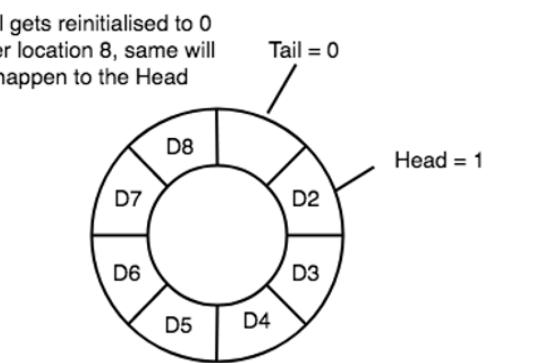
3. enqueue D2, D3, D4 and dequeue D1



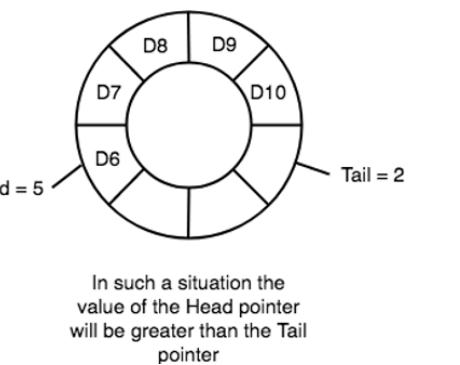
Lab2

Circular Queue

4. enqueue D5, D6, D7, D8



5. dequeue D2, D3, D4, D5 and enqueue D9, D10



Lab2

Circular Queue

- ❑ One of the benefits of the circular queue is that we can make use of the spaces in front of the queue. In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue (and it does not prevent the program accidentally creates a large queue or stack and use up the memory).

Implementation of CircularQueue class:

- **enqueue()**: insert the element
- **dequeue()**: delete the element
- **front()**: return the first element in the queue, if queue is empty, return None
- **rear()**: return the last element in the queue, if queue is empty, return None
- **is_empty()**: return true if queue is empty
- **is_full()**: return true if queue is full

Lab3



- linear search, binary search
- review Singly Linked List, Doubly Linked List, Recursion
- exercise [lab3.ipynb](#)

Lab3

binary search

1. Go to <https://www.cs.usfca.edu/~galles/visualization/Search.html> to understand how Linear Search & Binary Search is working
2. Implement Linear Search & Binary Search in Python by yourself:
 - familiar with Python coding style
 - understand the input, output, steps and ending condition
 - learn and compare different approaches (time & space complexity)
 - test code reliability with different cases

Lab3

- implement Stack by linked list
- implement Queue by linked list
- reverse a linked list
 - recursive implementation
 - iterative implementation

Lab4



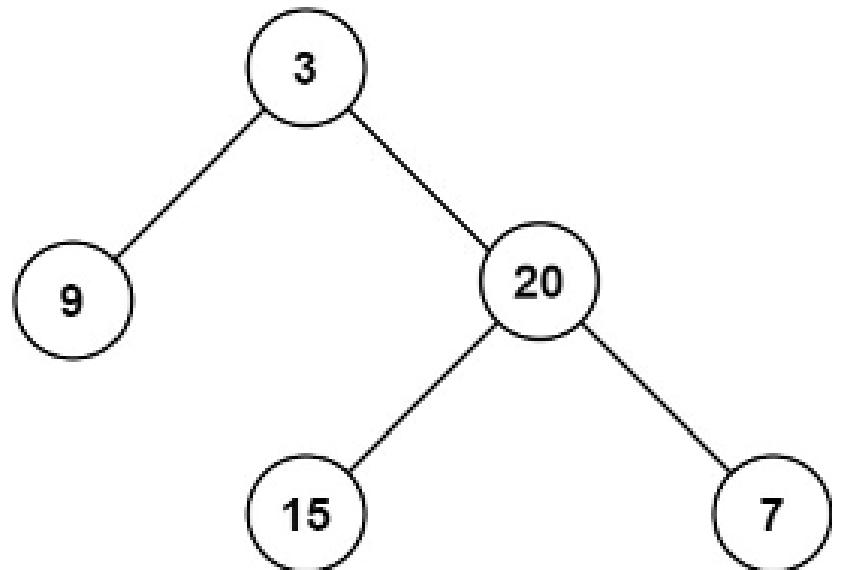
- review Binary Tree and 4 traverse methods
- exercise [lab4.ipynb](#)

Lab4

Exercise 1: get maximum depth of binary tree

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

For example: the maximum depth is 3

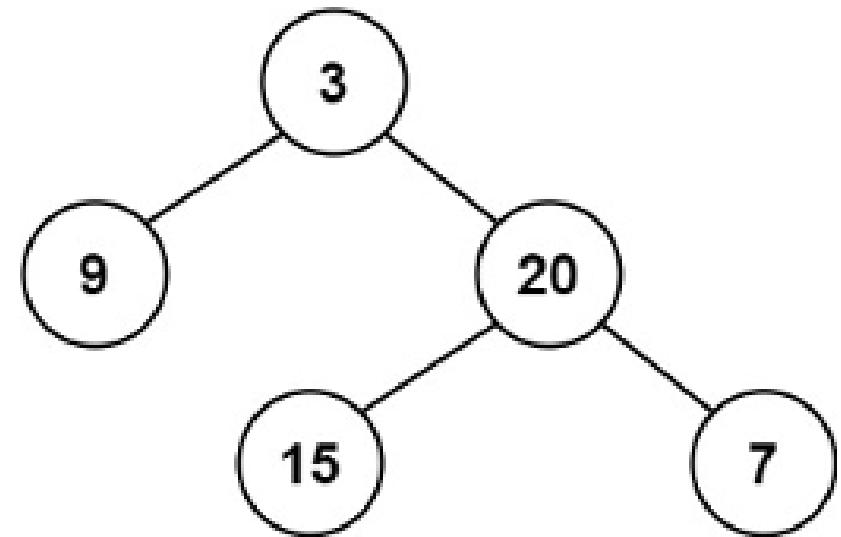


Lab4

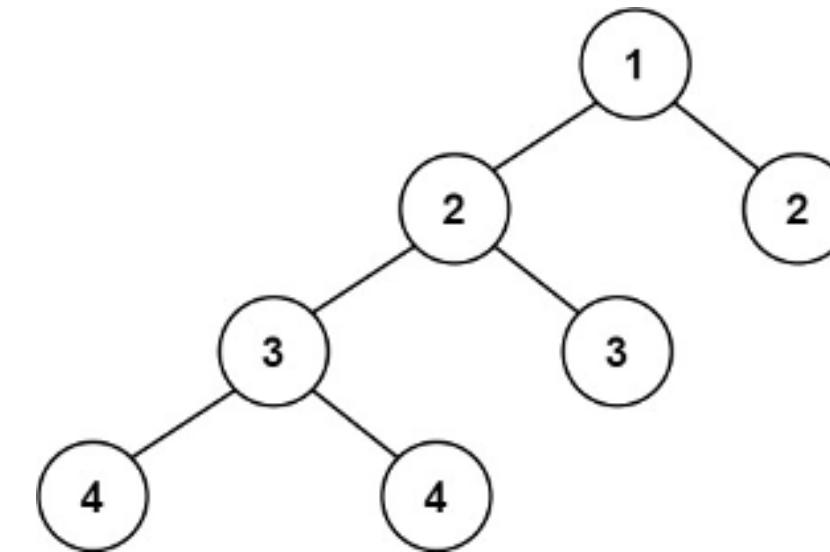
Exercise 2: check a balanced binary tree

A binary tree in which the left and right subtrees of every node differ in height by no more than 1.

balanced = true:



balanced = false:

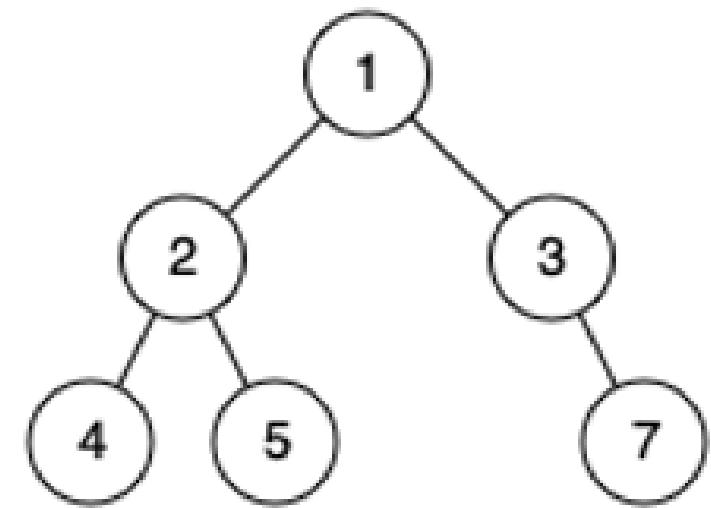


Lab4

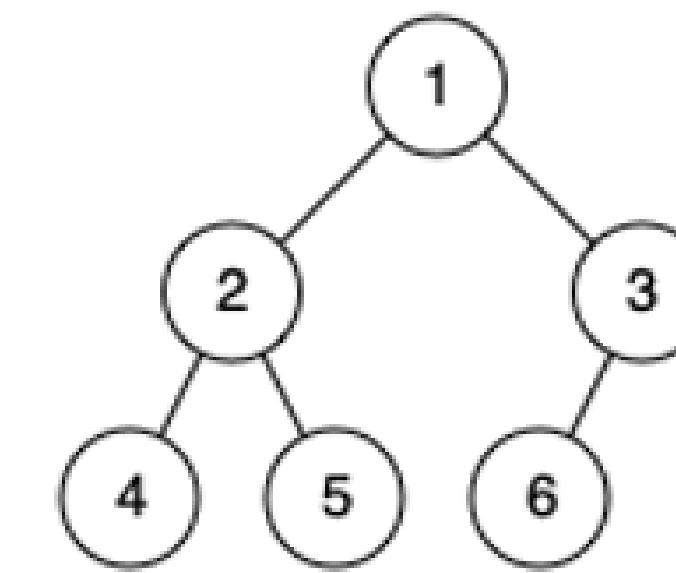
Exercise 3: check a complete binary tree

In a complete binary tree, every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes inclusive at the last level h .

complete = true:



complete = false:



Lab5



- exercise lab5.ipynb

Lab6



- exercise lab6.ipynb