

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. Jan 24: Python, Complexity & Big O
2. Feb 07: Array, Stack, Queue, Recursion
3. Feb 14: Linked List, Lineary Search, Binary Search
4. Feb 21: Tree
5. Feb 28: Algorithm Design & Pattern
6. Mar 07: Review

Mar 07: Resit Online Session

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

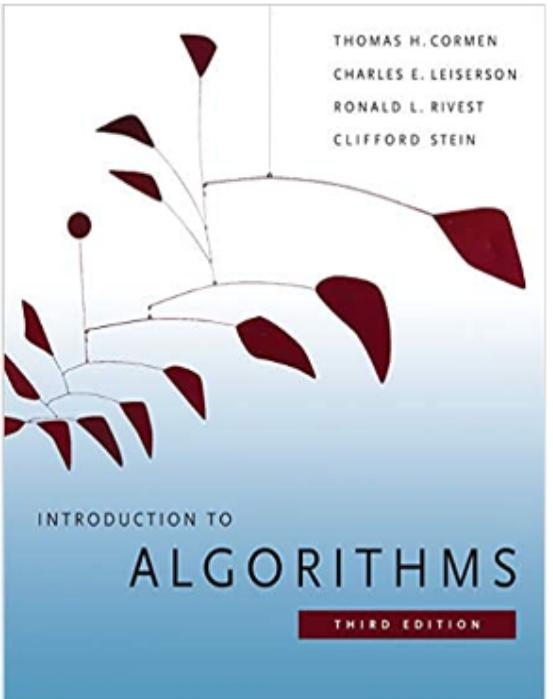
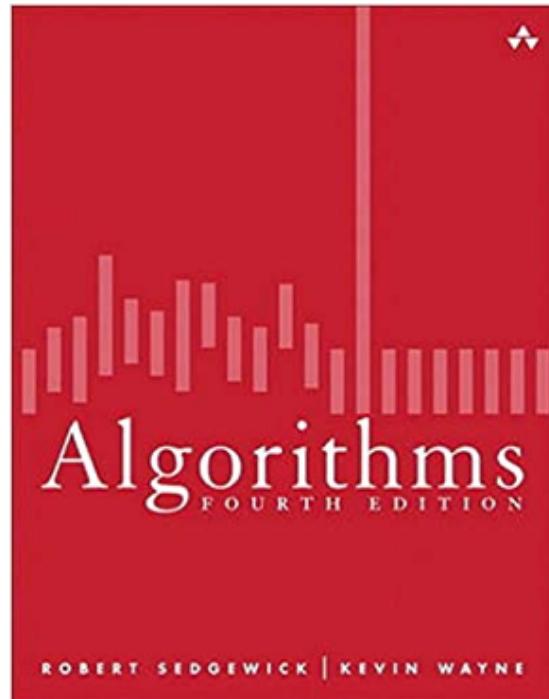
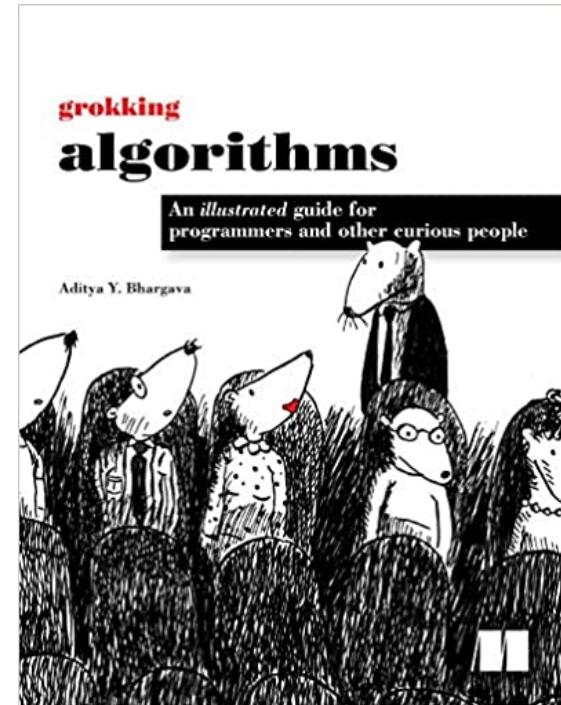
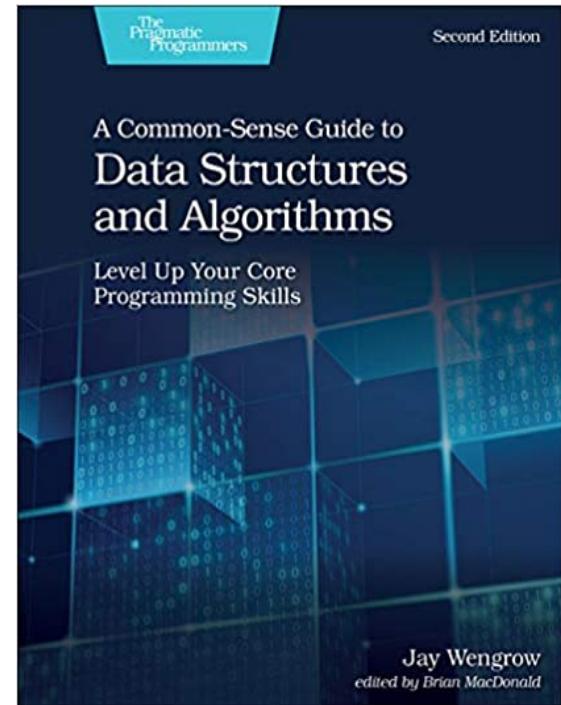
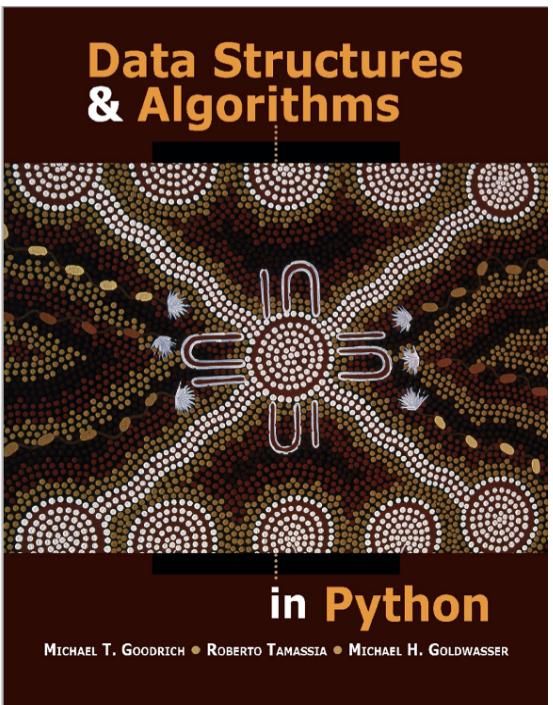
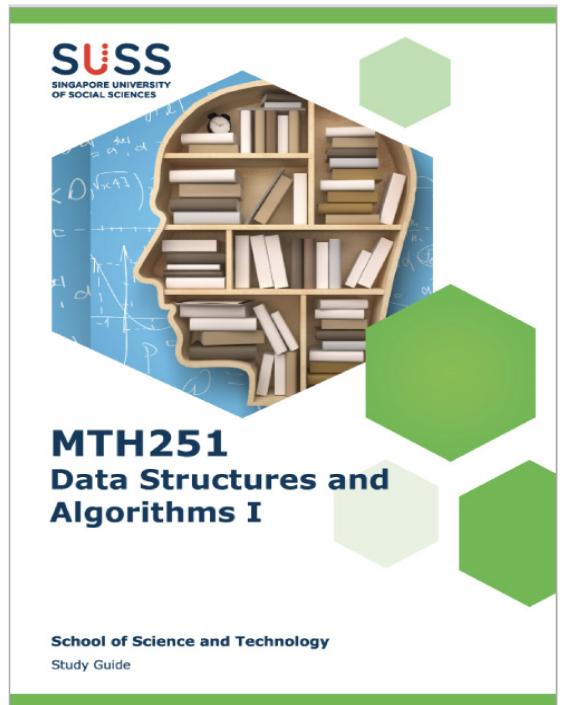
Slides & Notebooks

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

 <https://mth251.fastzhong.com/mth251.pdf>

 labs: <https://github.com/fastzhong/mth251/tree/main/public/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 under Data Structures and Algorithms. The cards include:

- Detailed Explanation of Graph (Premium)
- Introduction to Data Structure Arrays 101
- Introduction to Data Structure Linked List
- Introduction to Data Structure Binary Tree
- Introduction to Algorithms Recursion I
- Introduction to Algorithms Recursion II
- Basic Concepts in ML Machine Learning 101
- Learning about... Binary Search
- 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

Each card shows the number of chapters and items, and indicates if it is locked or premium content.

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.

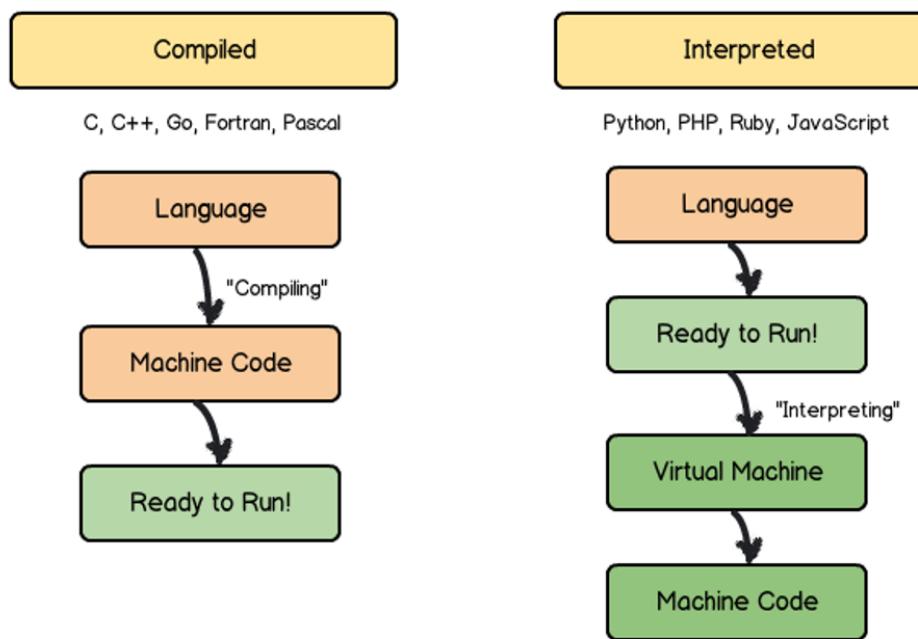
If the implementation is hard to explain, it's a bad idea.

Python Jobs

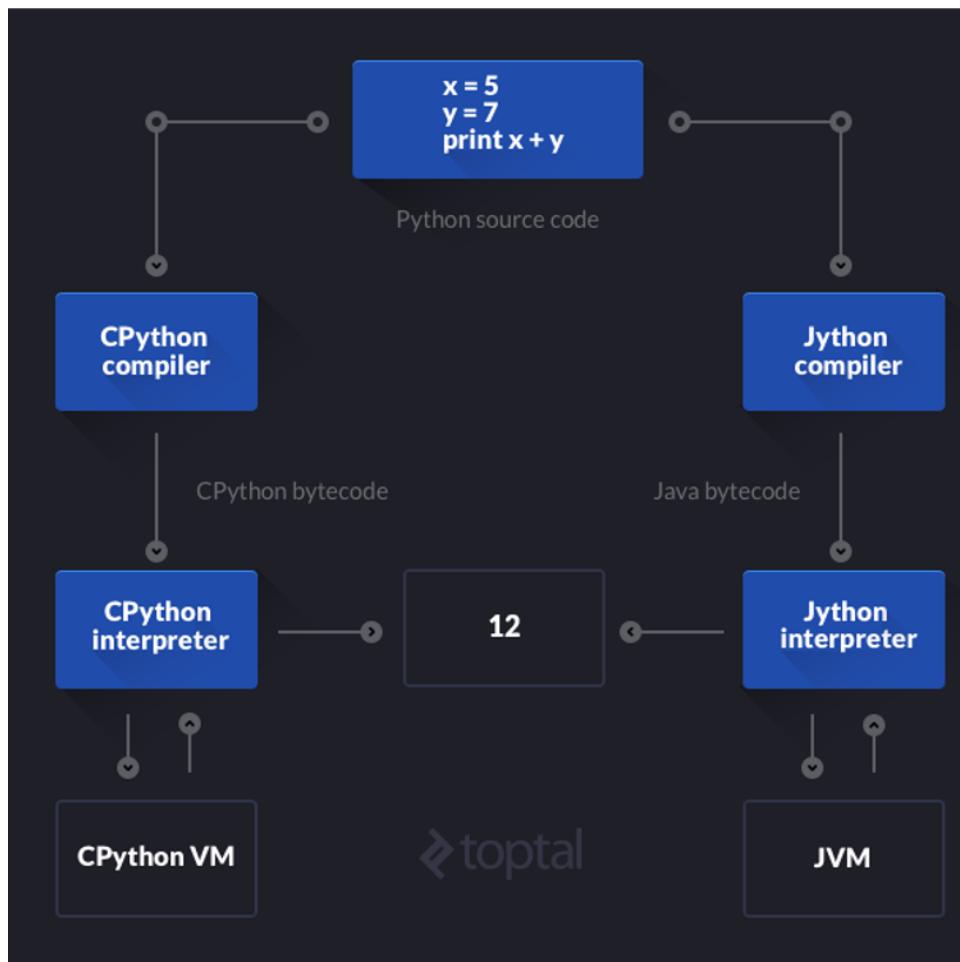
- ✓ backend: **Python** vs. Java, C++, Go, Php
- ✓ devops: **Python** vs. Go, Ruby, Shell
- ✓ test automation: **Python** vs. Groovy, shell
- ✓ data engineering: **Python** vs. Java, C++
- ✓ data analytics & visualization: **Python** vs. R, Java, C++
- ✓ data science & machine learning: **Python** vs. 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
- `dictionary`: a set of `key: value` pairs, unordered, changeable and indexed

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

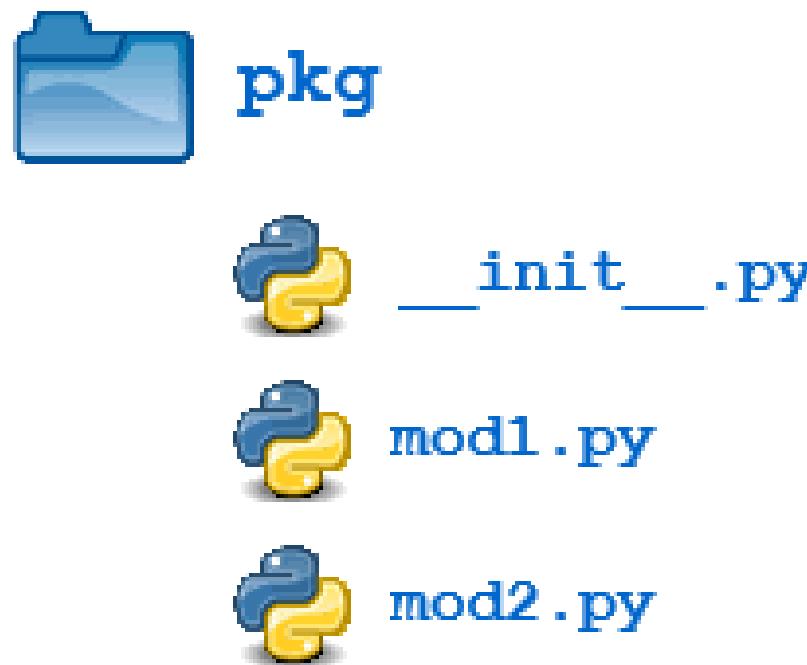


- Procedural vs. OOP vs. FP
- OO Principal
 - Inheritance
 - Encapsulation
 - Polymorphism
- class, instance, attributes, properties, method
- override vs. overload vs. overwrite

Misc.



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



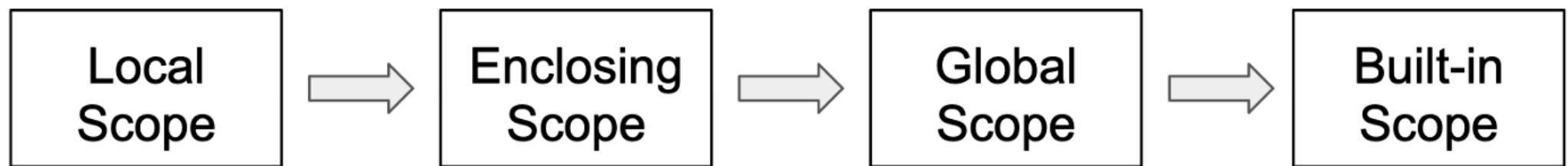
- Standard Lib: math random re os itertools collections

Misc.



- Namespaces & Scopes: `LEGB` rule

The LEGB Rule



- memory, copy vs. deepcopy
- help

Cheatsheet:



Python Crash Course - Cheat Sheets



Comprehensive Python Cheatsheet

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](#)

The Hitchhiker's Guide to Python!

Data Structure & Algorithms (DSA)

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

- Input
- Output
- Definiteness
- Effectiveness
- Finiteness

Data Structure & Algorithms

Example: keyword searching

Data Structure: index

Algorithm: looking up the page no.

738

Index

abc module, 60, 93, 306
Abelson, Hal, 182
abs function, 29, 75
abstract base class, 60, **93–95**, 306, 317, 406
abstract data type, v, 59
 deque, 247–248
 graph, 620–626
 map, 402–408
 partition, 681–684
 positional list, 279–281
 priority queue, 364
 queue, 240
 sorted map, 427
 stack, 230–231
 tree, 305–306
abstraction, 58–60
(*a,b*) tree, 712–714
access frequency, 286
accessors, 6
activation record, 23, 151, 703
actual parameter, 24
acyclic graph, 623
adaptability, 57, 58
adaptable priority queue, 390–395, 666, 667
AdaptableHeapPriorityQueue class, 392–394, 667
adapter design pattern, 231
Adel'son-Vel'skii, Georgii, 481, 535
adjacency list, 627, 630–631
adjacency map, 627, 632, 634
adjacency matrix, 627, 633
ADT, *see* abstract data type
Aggarwal, Alok, 719
Aho, Alfred, 254, 298, 535, 618
Ahuja, Ravindra, 696
algorithm, 110
algorithm analysis, 123–136
 average-case, 114
 worst-case, 114
alias, 5, 12, 101, 189
all function, 29
alphabet, 583
amortization, 164, **197–200**, 234, 237, 246, 376, 681–684
ancestor, 302
and operator, 12
any function, 29
arc, 620
arithmetic operators, 13
arithmetic progression, 89, 199–200
ArithmetError, 83, 303
array, 9, **183–222**, 223, 227
 compact, 190, 711
 dynamic, 192–201, 246
array module, 191
ArrayQueue class, **242–246**, 248, 292, 306
ASCII, 721
assignment statement, 4, 24
 chained, 17
 extended, 16
 simultaneous, 45, 91
asymptotic notation, 123–127, 136
 big-Oh, 123–127
 big-Omega, 127, 197
 big-Theta, 127
AttributeError, 33, 100
AVL tree, 481–488
 balance factor, 531
 height-balance property, 481
back edge, 647, 689
backslash character, 3
Baeza-Yates, Ricardo, 535, 580, 618, 719
Barůvka, Otakar, 693, 696
base class, 82
BaseException, 83, 303
Bayer, Rudolf, 535, 719
Beazley, David, 55
Bellman, Richard, 618
Bentley, Jon, 182, 400, 580
best-fit algorithm, 699
BFS, *see* breadth-first search
biconnected graph, 690
big-Oh notation, 123–127
big-Omega notation, 127, 197
big-Theta notation, 127
binary heap, 370–384
binary recursion, 174
binary search, 155–156 162–163, 428–433, 571
binary search tree, 332, 460–479

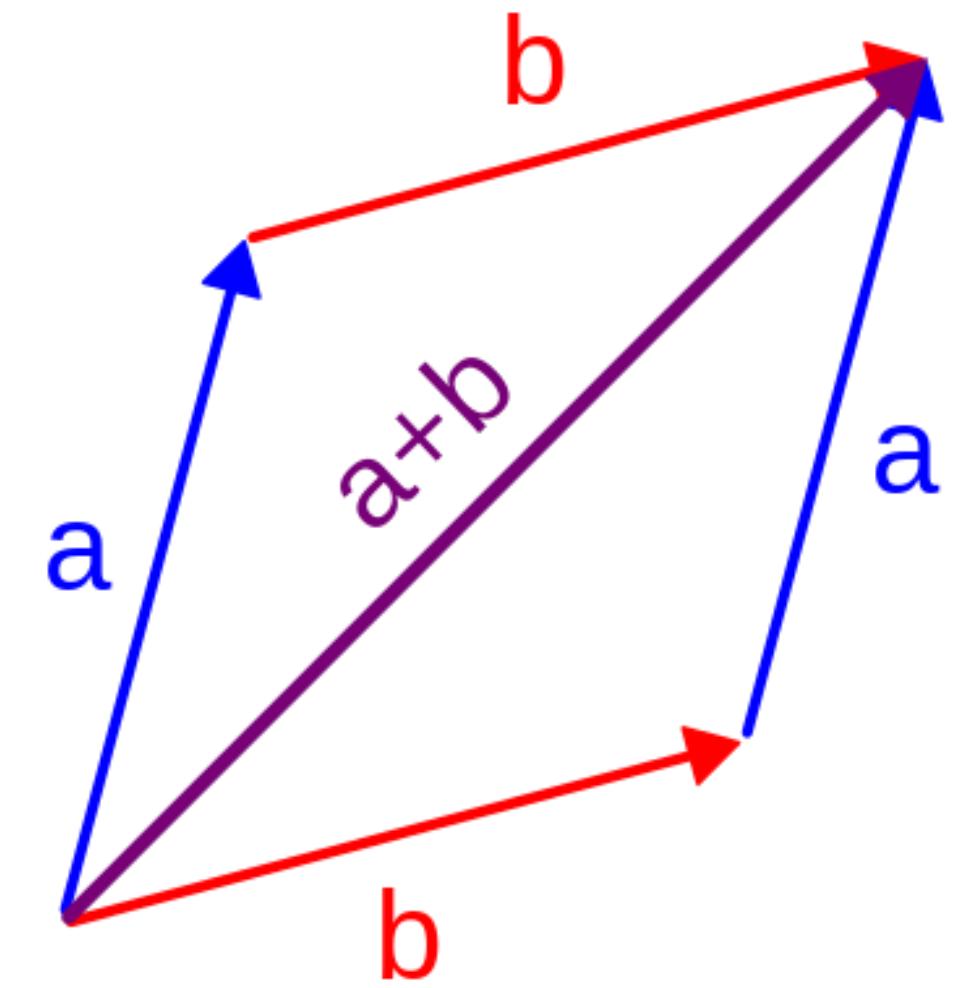
Data Structure & Algorithms

Example: adding two vectors

Data Structure: complex number

Algorithm: adding two complex numbers

$$a + b = (x + yi) + (u + vi) = (x + u) + (y + v)i$$



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 powerful framework and library

How

 learning by doing, implementing from scratch

 problem solving

Algorithm Complexity Analysis



How to measure Performance/Efficiency ?

- cpu, memory, io, networking, etc.
- no. of lines
- worst case vs. best case
- code slows as data grows
- ...

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

Good code:

- ✓ readability
- ✓ speed
- ✓ memory

👉 When: Accessing, Searching, Inserting, Deleting, ...

Big-O



**Lef $f(n)$ and $g(n)$ be functions from positive integers to positive integers to positive reals
 $f = O(g)$ if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ for large n**

⌚ $f(n)$ grows no faster than $\underline{g(n)}$

e.g.

$$f(n) = O(4n^2 + 8n + 16) \rightarrow O(n^2) \rightarrow g(n) = n^2$$

$$O(4n^2 + 8n + 16) = O(n^2)$$

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

Big-O

$O(1)$: constant

$O(\log_* n)$: logarithmic

$O(n)$: linear

$O(n \log_* n)$: linearithmic

$O(n^2)$: polynomial

$O(2^n)$: exponential

$O(n!)$: factorial

Big-O

👉 Master theorem (analysis of algorithms)

$$O(f) = f$$

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

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

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

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

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

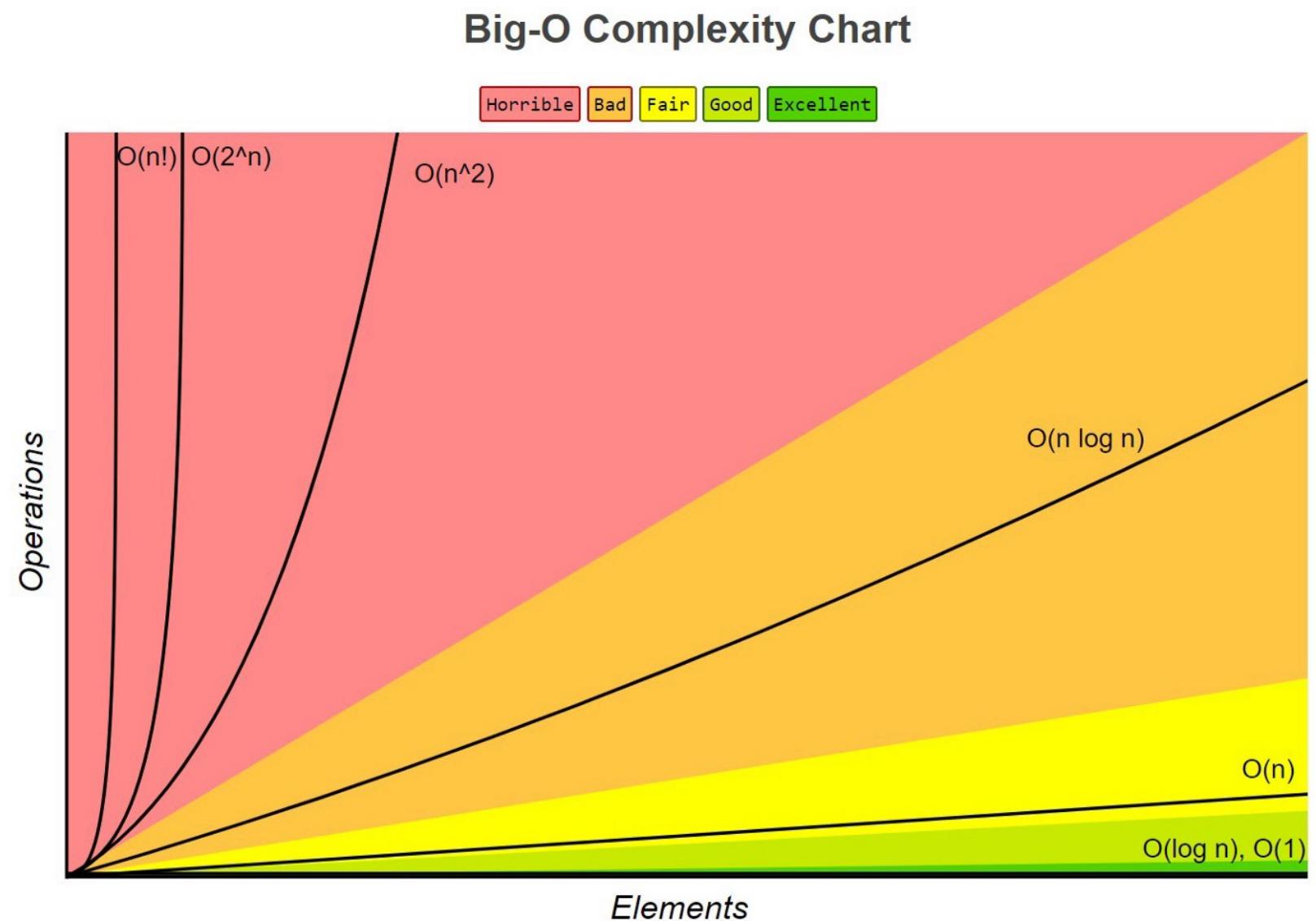
$$O(a^x) < O(b^x) \text{ if & only if } a < b$$

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

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

Big-O

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



↗ <https://www.bigocheatsheet.com/>

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

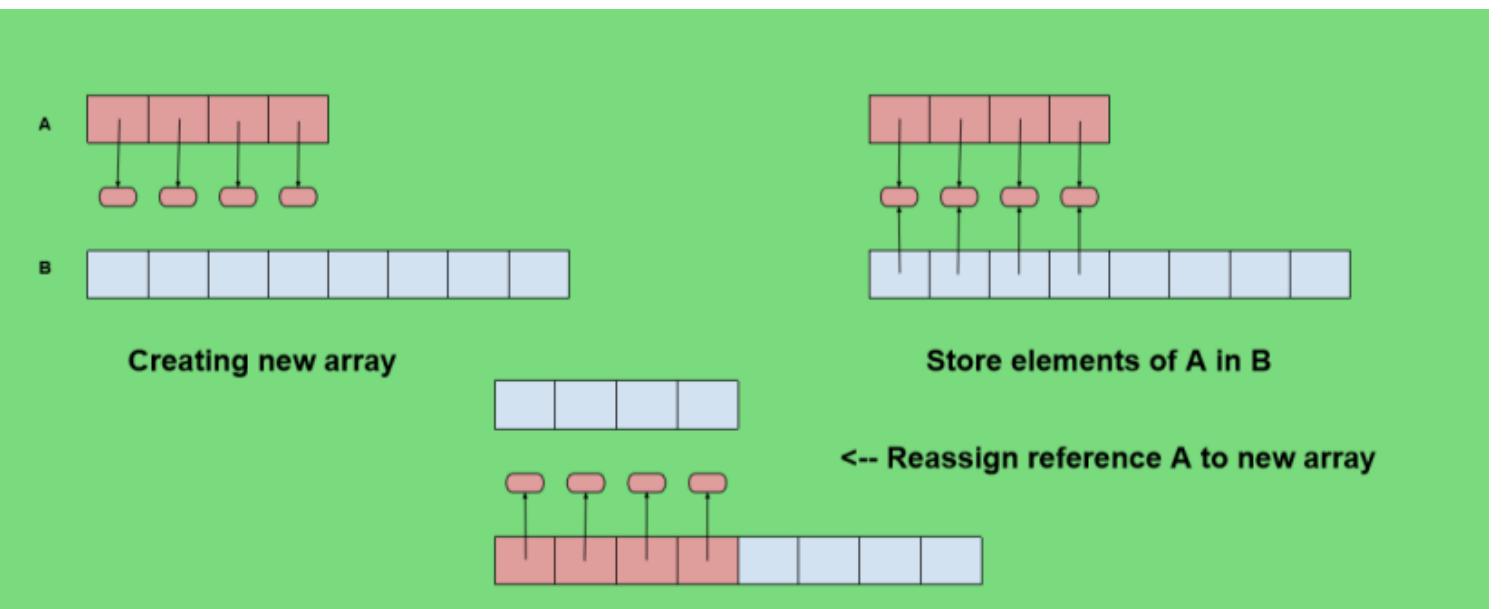
array[i].address = base_address + i * data_type_size

👉 O(1)

Dynamic Array



- what is the good space/slot size for an array?
- when is the good time to expand/shrink the array?



Growth Pattern:

- Python $0, 4, 8, 16, 25, 35, 46, 58, 72, 88, \dots$
- Java $((size * 3)/2) + 1$
- C# $size * 2$

👉 memory management is one of the biggest programming challenges.

Array Complexity

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

Python Collection Complexity

| List [a, b, c, ...] | Dicts {k:v, ...} | Set {a, b, c,} |
|---------------------|------------------|--------------------|
| mylist.append(val) | O(1) | mydict[key] = val |
| mylist[i] | O(1) | mydict[key] |
| val in mylist | O(N) | key in mydict |
| for val in mylist: | O(N) | for key in mydict: |
| mylist.sort() | O(NlogN) | |

Trade-offs

```
1  # .. make a list ..
2  if thing in my_list: # O(N)
3
4  # ✅ Good
5  # .. make a set ..
6  if thing in my_set: # O(1)
7
8  # ❌ Bad
9  # .. make a list ..
10 my_set = set(my_list) # O(N)
11 if thing in my_set: # O(1)
12
13 # ✅ Good
14 # .. make a list ..
15 my_set = set(my_list) # O(N)
16 for many_times:
17     if thing in my_set: # O(1)
18
```

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

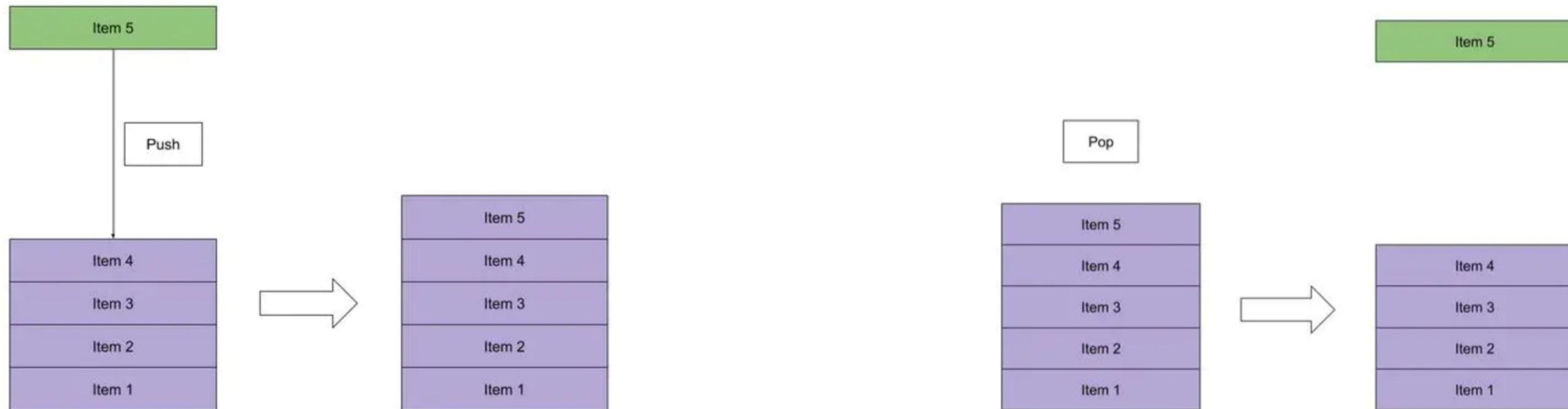
Stack

- 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 Complexity

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

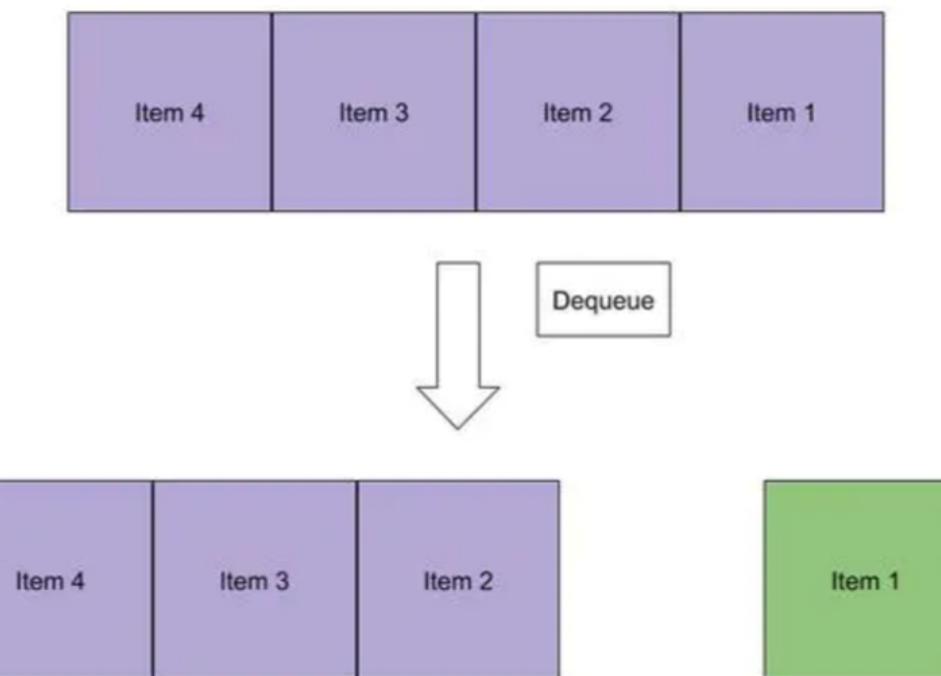
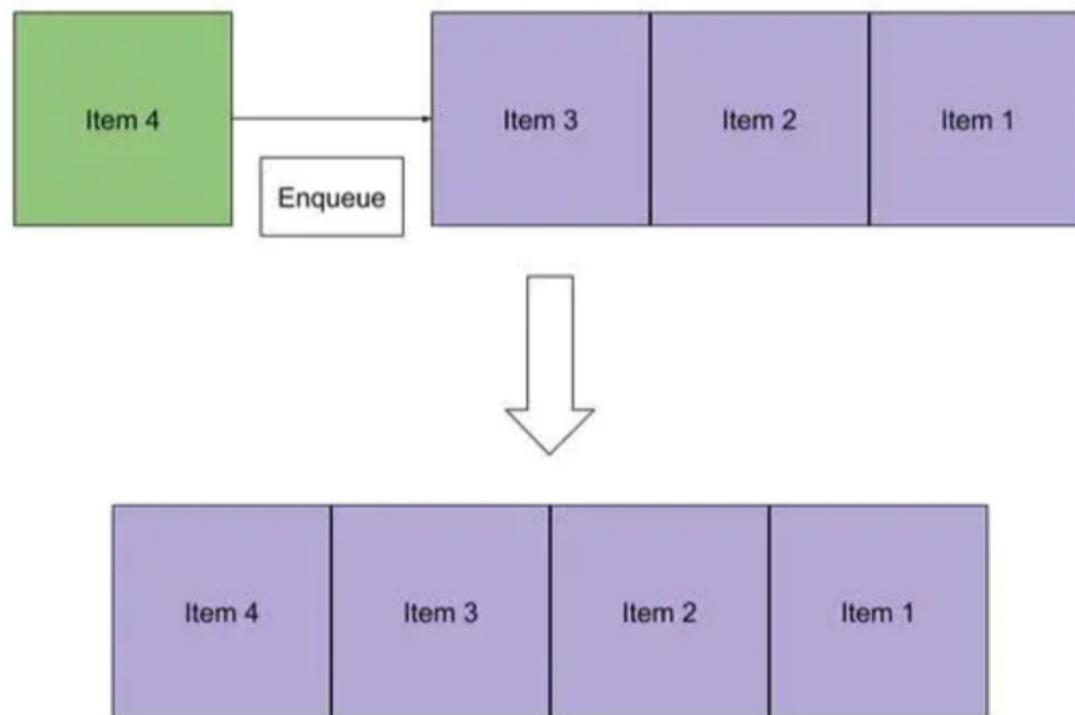
Queue

- **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()`

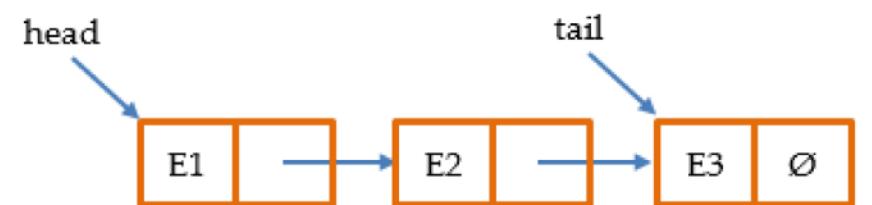


Queue Complexity

| 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



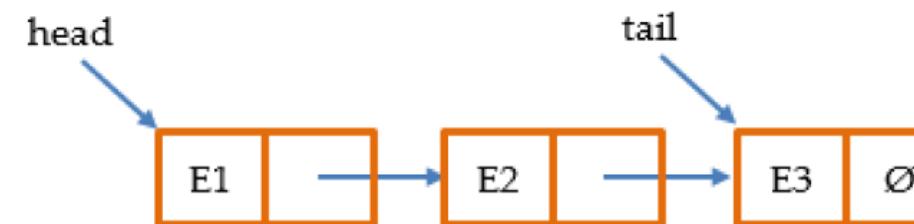
code here

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

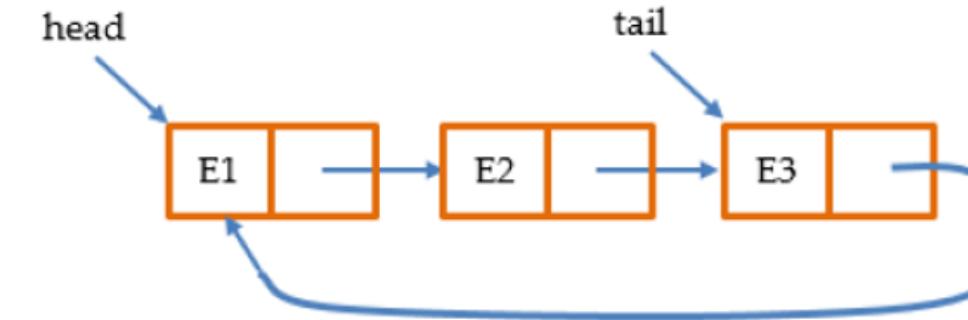
Singly Linked List



Doubly Linked List



Circular Linked List



Positional Linked List



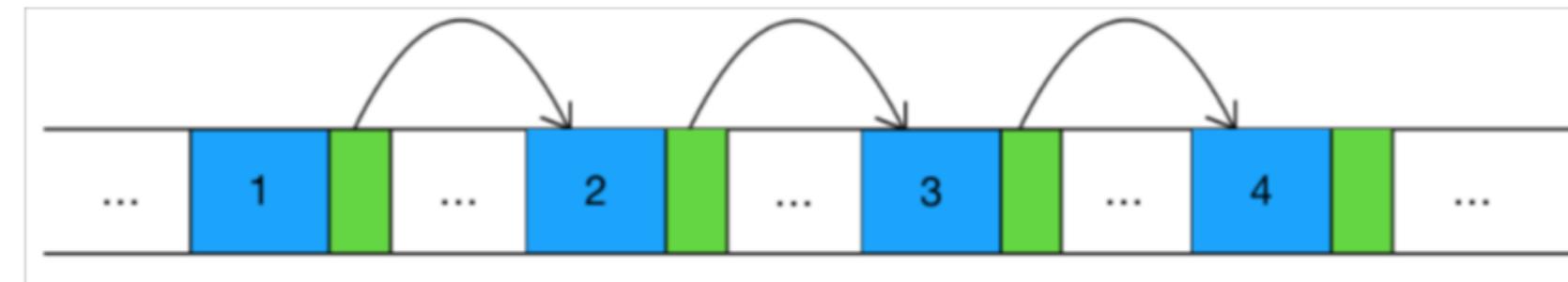
Linked List vs. Array

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

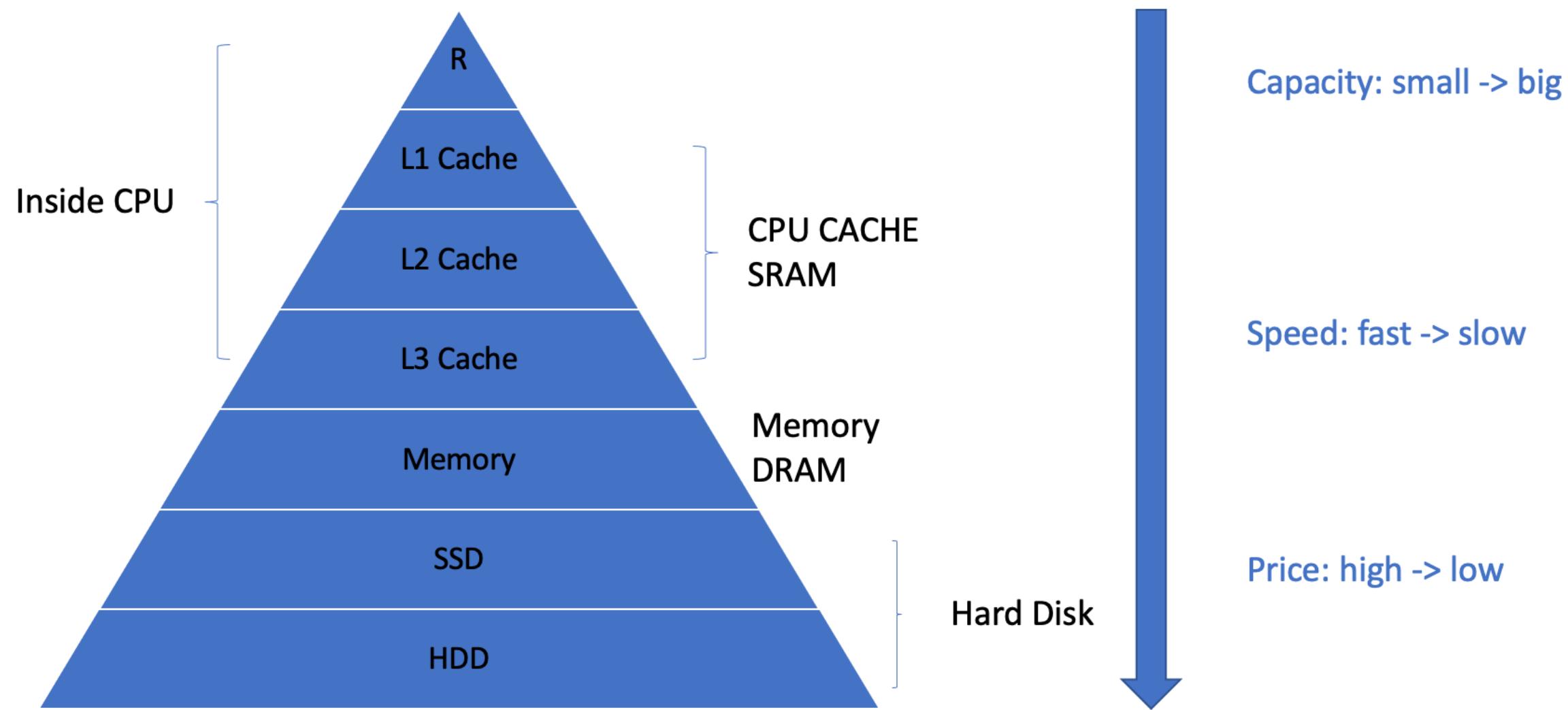
Array:

| 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

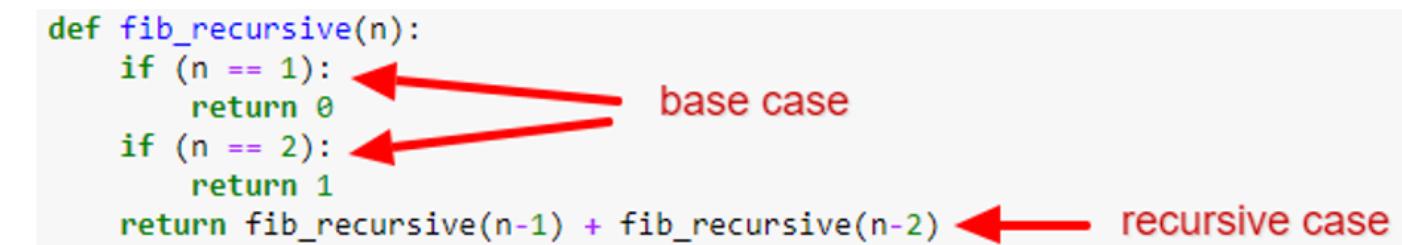
Recursion by definition is a function that calls itself.

- base case
- recursive case

Example:

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

```
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 defines a function `fib_recursive` that takes an argument `n`. It uses two `if` statements to handle the base cases where `n` is 1 or 2, returning 0 and 1 respectively. For any `n` greater than 2, it returns the sum of the previous two Fibonacci numbers, obtained by recursive calls to `fib_recursive(n-1)` and `fib_recursive(n-2)`.

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

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)
```

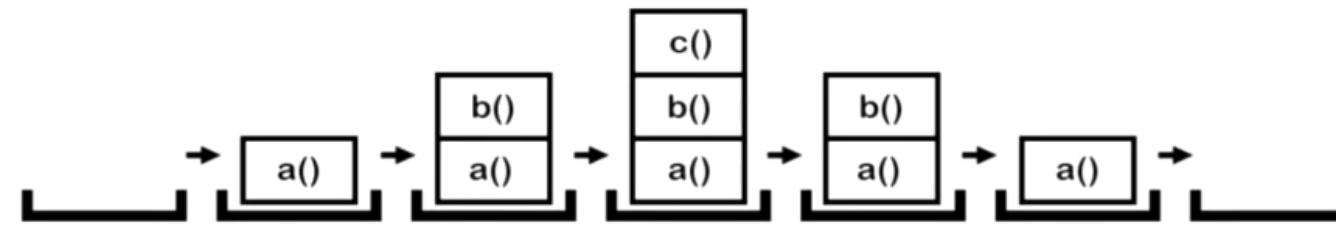
base case
recursive case

🤔 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
```

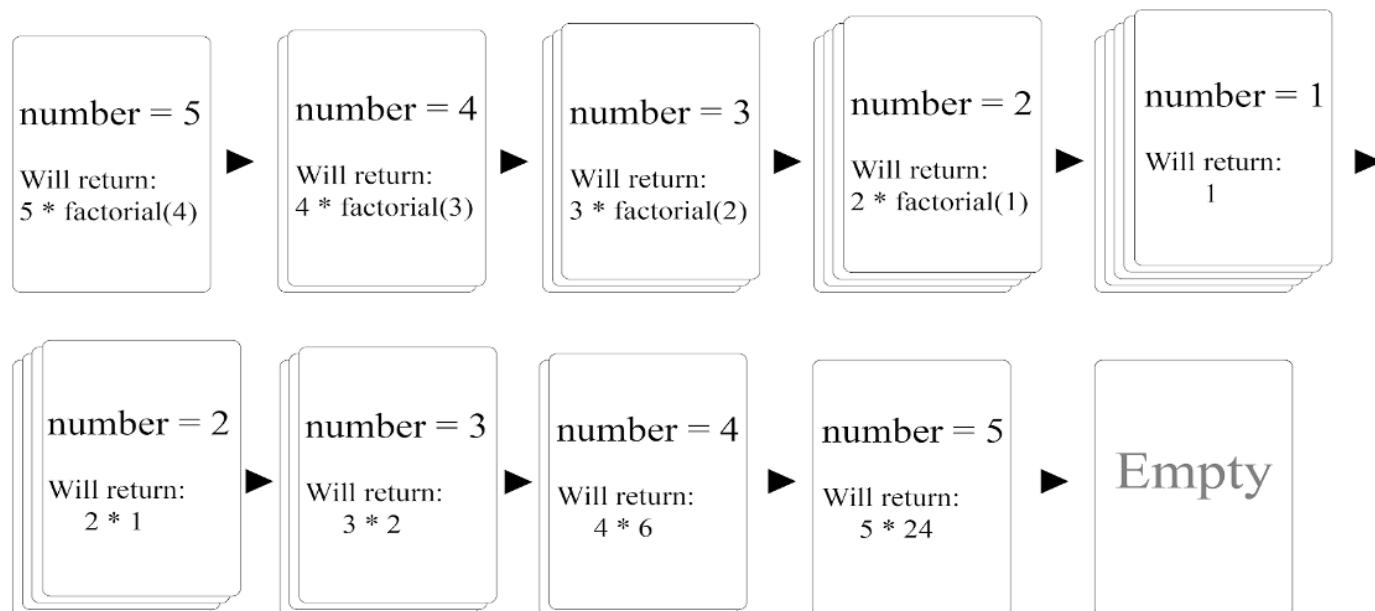
Recursive Call

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



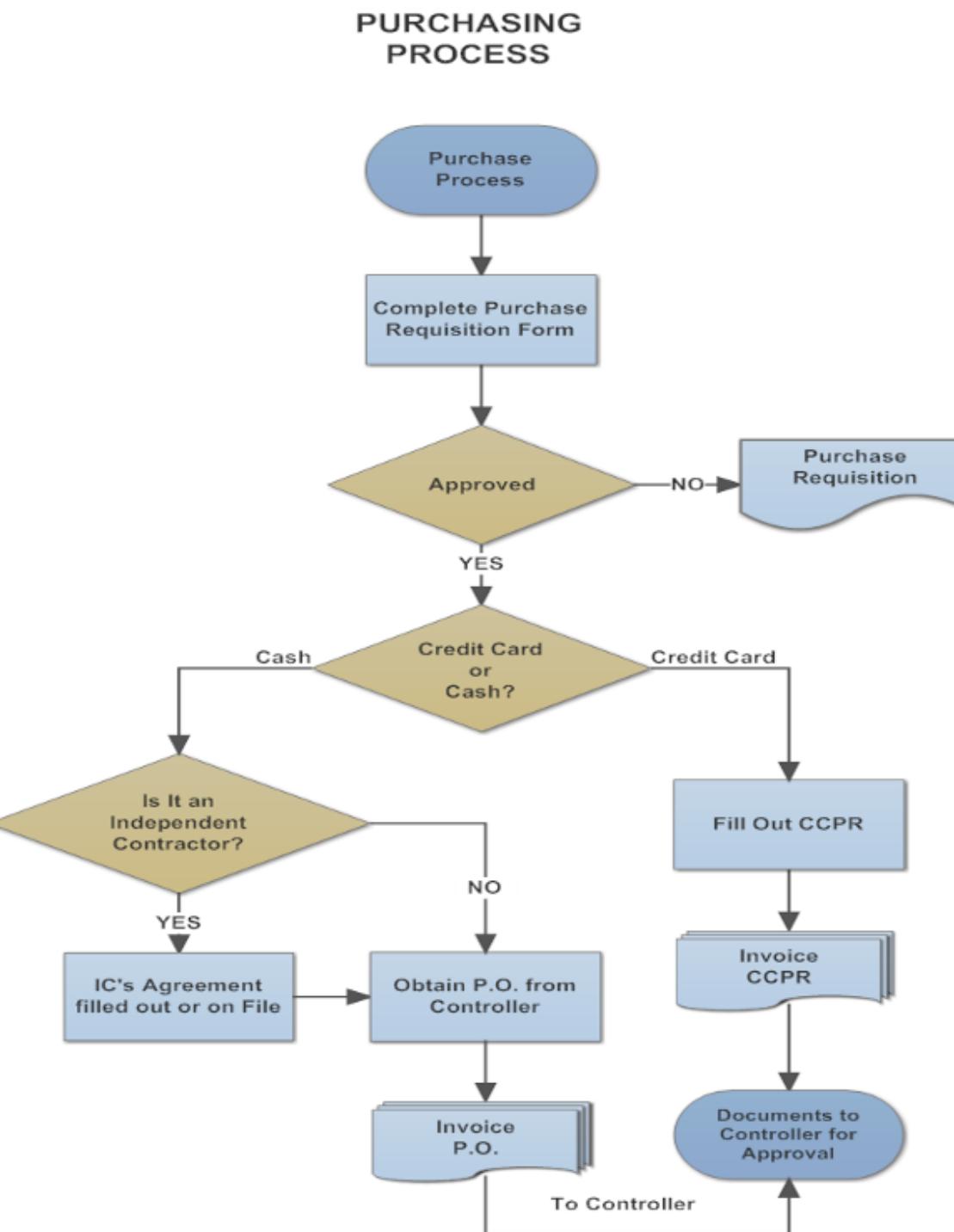
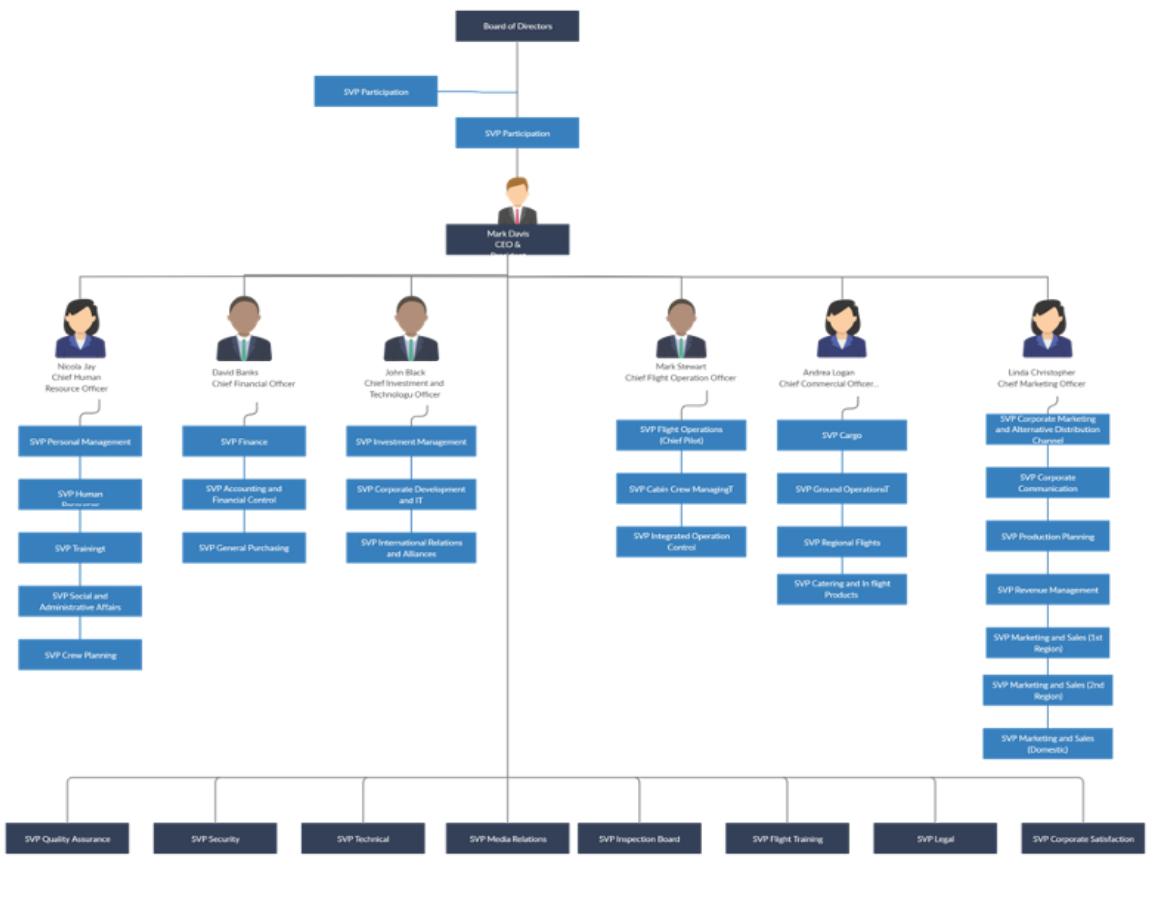
- Input: array, target element
- Output: position (-1 if not existing)

Binary Search



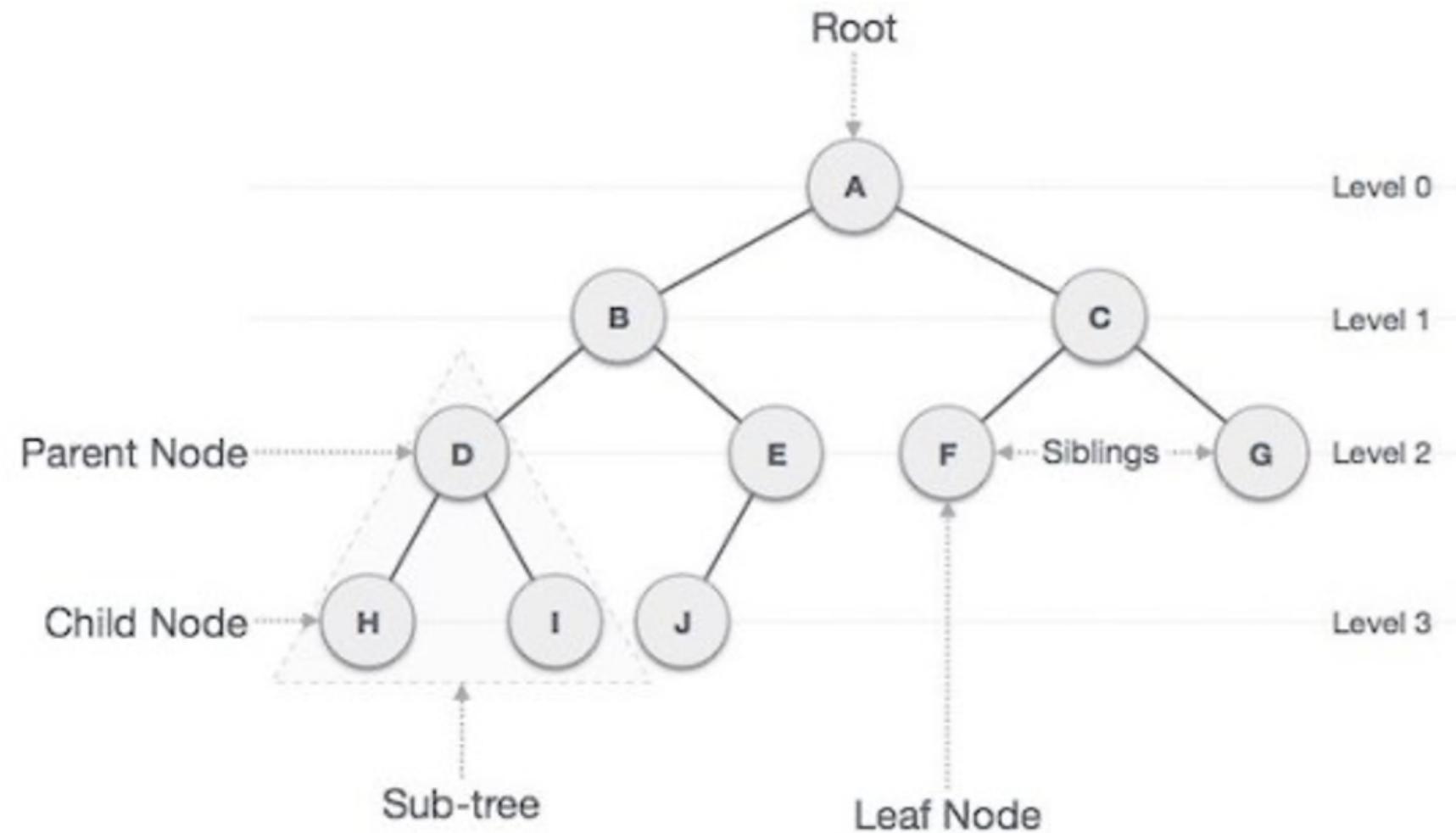
- Input: array, target element
- Output: position (-1 if not existing)

Tree



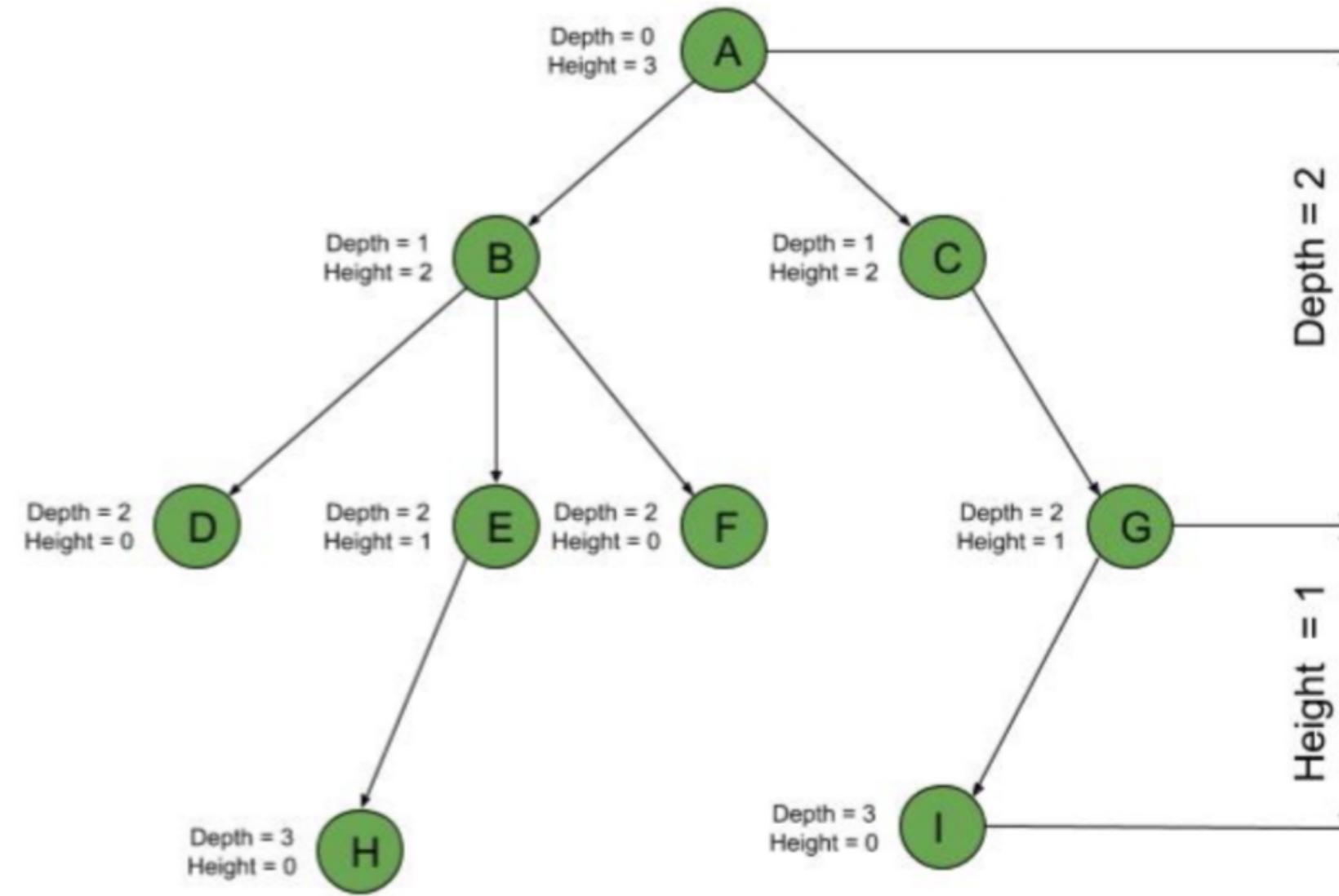
Tree Terminology

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



Tree Terminology

- Level vs. Depth vs. Height



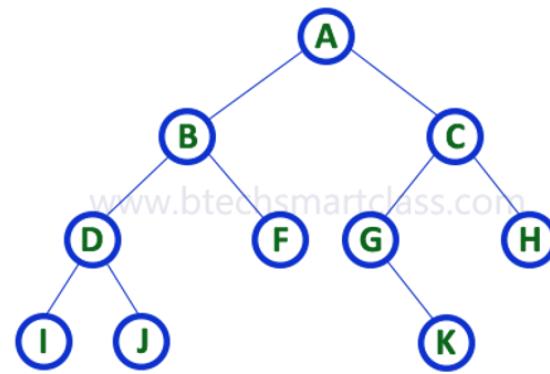
Binary Tree

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

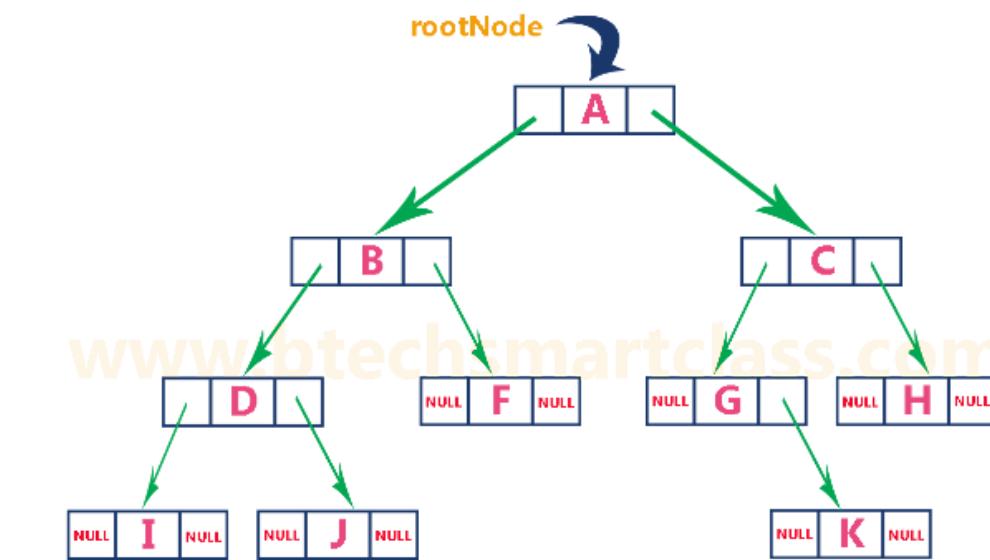
Binary Tree

- Array

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



- Left/Right Linked List



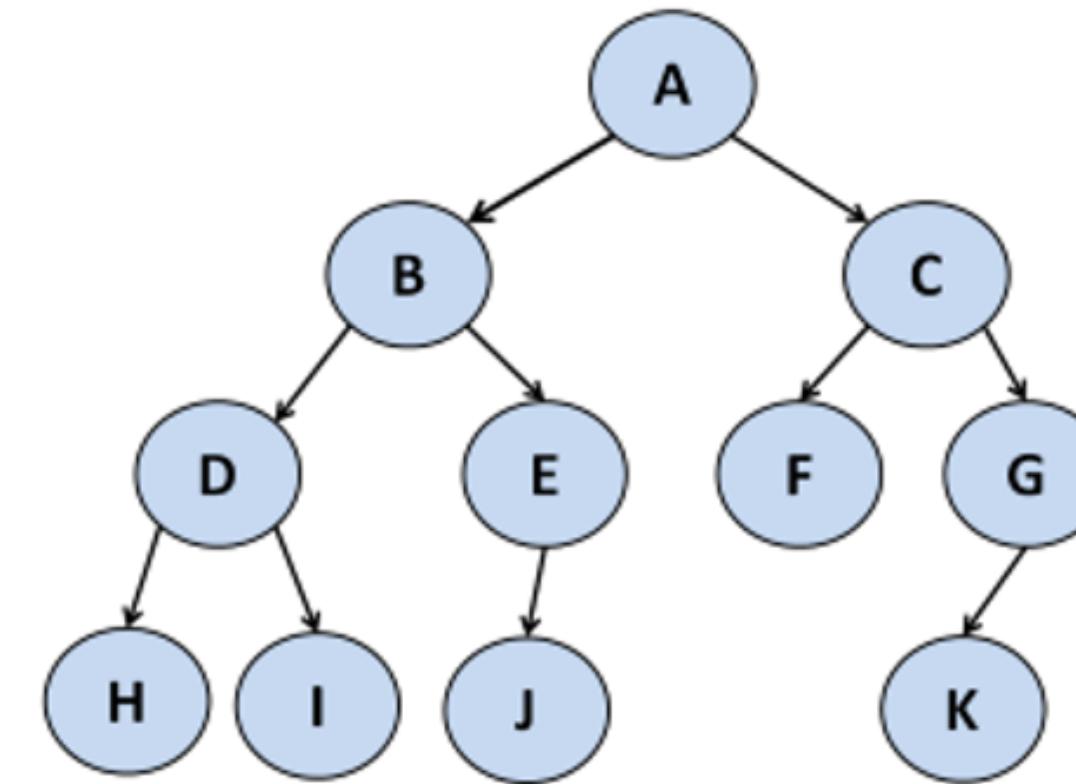
Binary Tree Traverse (DFS): pre-order



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



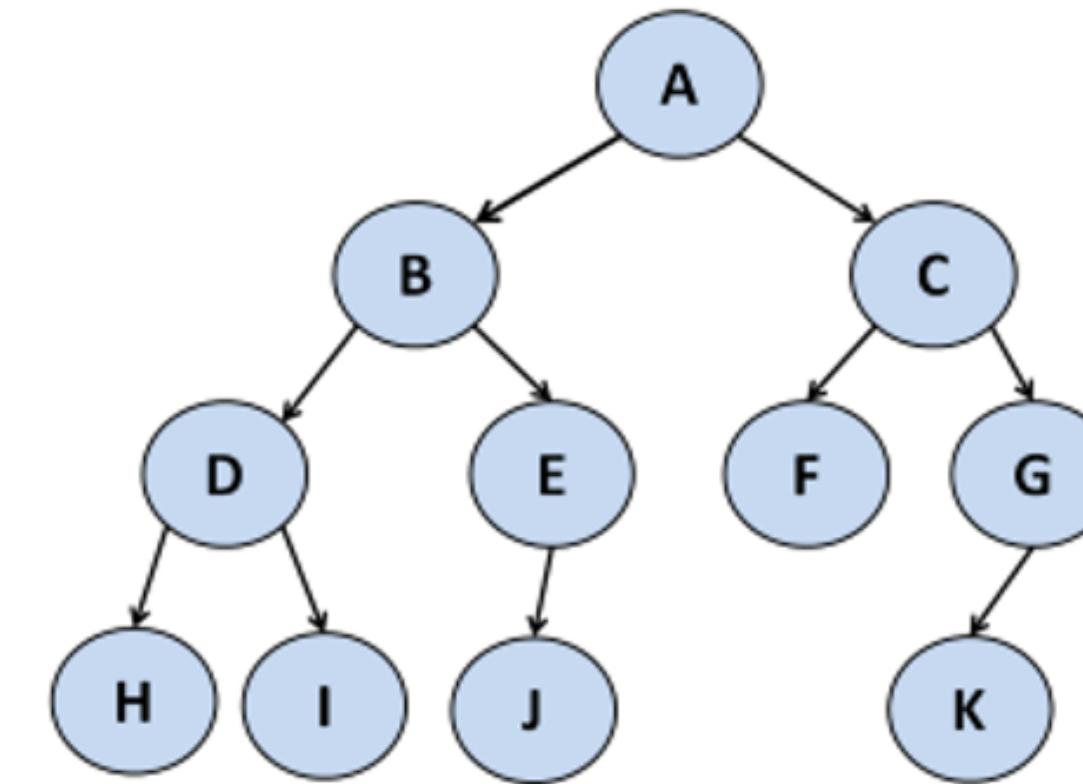
Binary Tree Traverse (DFS): in-order



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



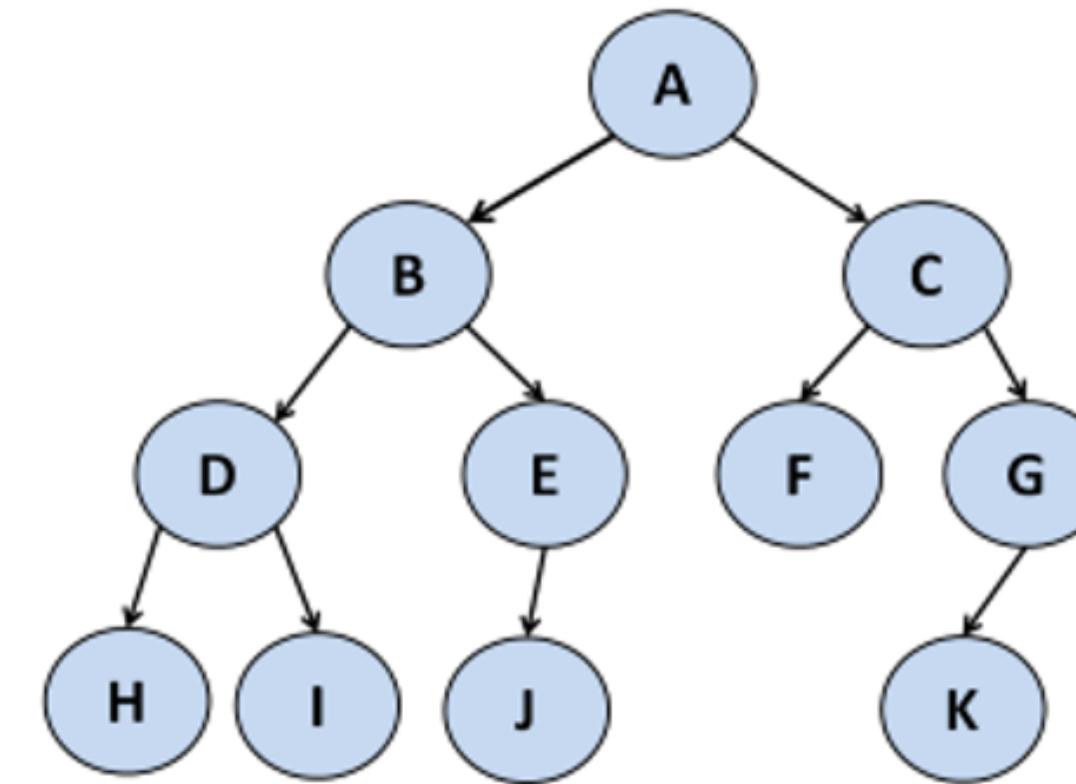
Binary Tree Traverse (DFS): post-order



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

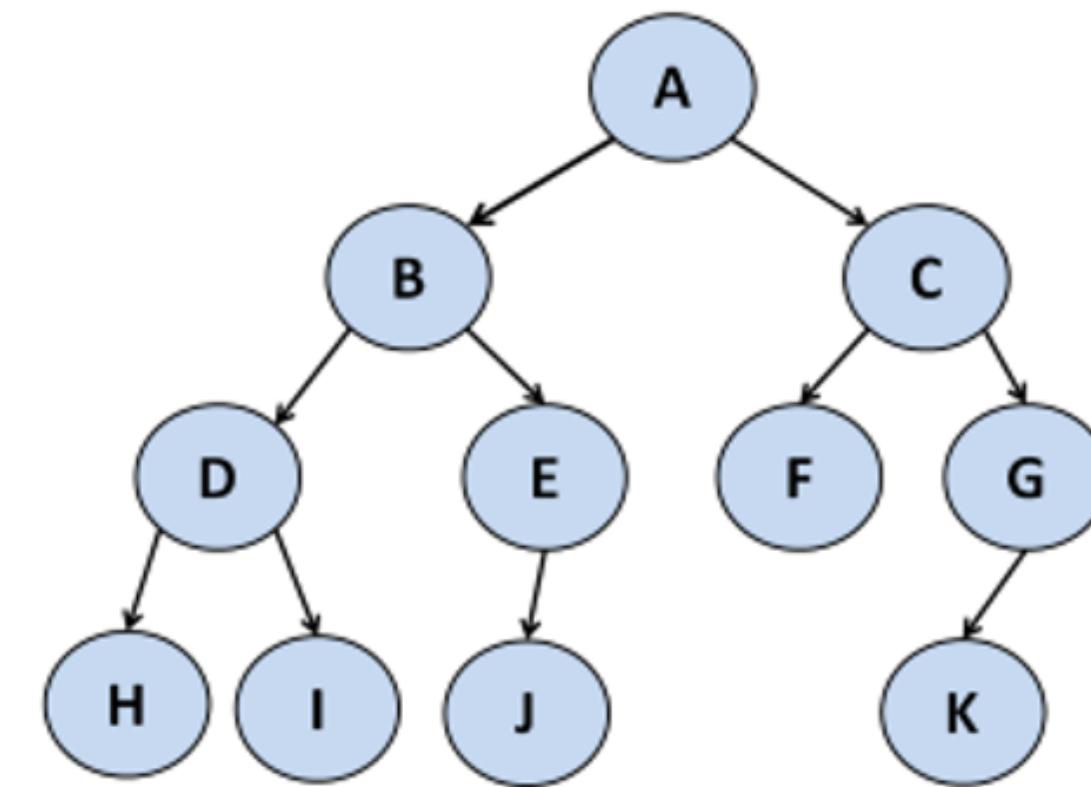


Binary Tree Traverse (BFS): level order



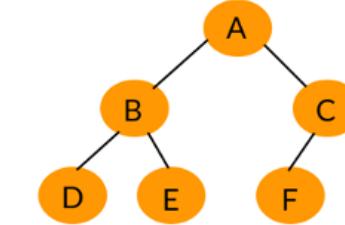
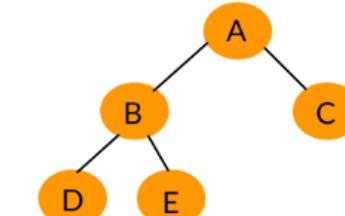
1. Visit the root
2. Visit the left node
3. Visit the right node
4. Go to next level

A B C D E F G H I J K

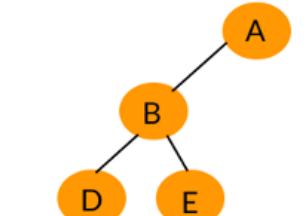


Binary Tree

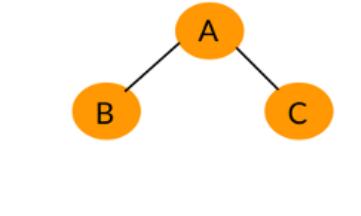
- Complete Tree: every level is completely filled except the last (leaf) and all nodes are as far left as possible
- Full Binary Tree: every node has two child nodes except leaf
- Perfect Binary Tree: every node has two child nodes except leaf and all leaves on same level



Complete Binary Tree

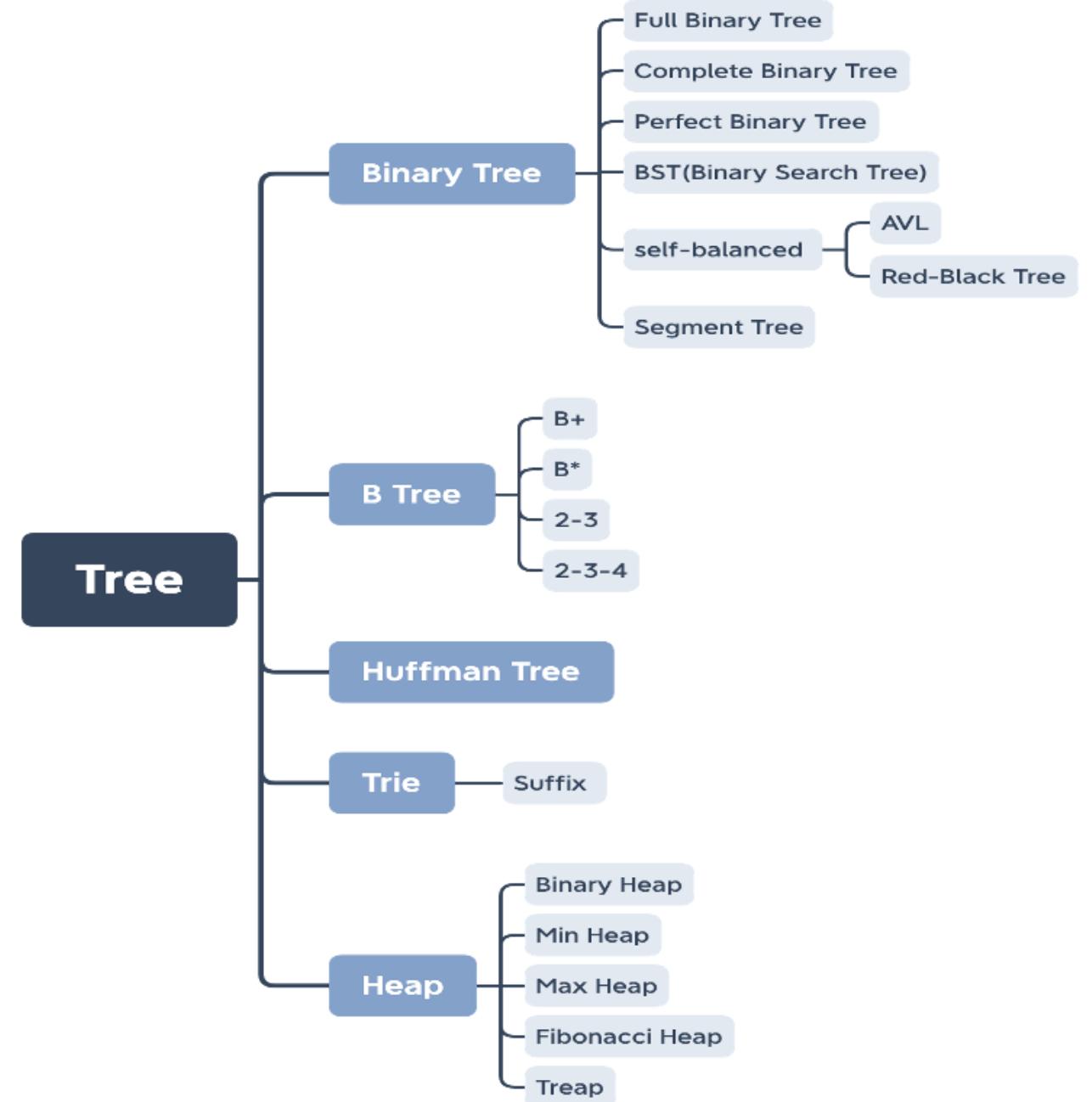


Full Binary Tree



Perfect Binary Tree

Trees



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 |
|------------|--|
| Complexity | 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 |
|------------|--|
| Complexity | 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 |
|------------|---|
| Complexity | 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 |
|------------|---|
| Complexity | 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 |
|------------|--|
| Complexity | <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



Lab 1



- 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](#)

Lab 1

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, and Company. A prominent 'Get Started' button is located in the top right. Below the navigation, the 'Individual Edition' section features a large green 'Q' icon and the text '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 visible. At the bottom, there are three links: 'Open Source' (with a Python logo), 'Conda Packages' (with a package icon), and 'Manage Environments' (with a server icon).

Lab 1

Create and Activate your Anaconda Python env

Lab 1

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
```

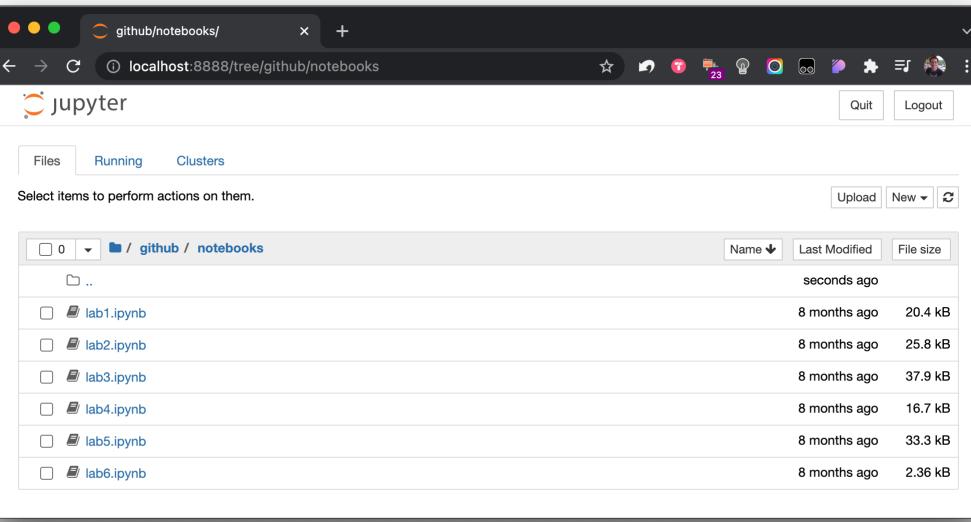
- copy & paste the Jupyter link in the prompt to your browser
- save notebooks to your local
- Control-C to stop Jupyter from the command line

Lab 1

Create and Activate your Anaconda Python env

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

Jupyter notebooks (lab1.ipynb):



Lab 1



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

[Jupyter Notebooks in VS Code](#)



Google provides online Jupyter env:

<https://colab.research.google.com/>

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

colab: <https://colab.research.google.com/github/fastzhong/mth251/blob/main/public/notebooks/lab1.ipynb>

Lab 1

lab1.ipynb

- Python
 - 1. Data Type & Operators
 - 2. Collections
 - 3. Program Structure
 - 4. OO & Class
- Big O

Lab 2



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

Lab 2

Exercise: 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]`

Lab 2

Exercise: valid parentheses

Given a string s containing just the characters $'($, $)$, $'\{$, $\}$, $'[$ and $']$, determine if the input string is valid.

An input string is valid if:

- Open brackets must be closed by the same type of brackets.
- Open brackets must be closed in the correct order.

Example 1

Input: $s = "()\[\{\}]"$ Output: true

Example 2

Input: $s = "(\[)"$ Output: false

Example 3

Input: $s = "(\[\])"$ Output: false

Example 4

Input: $s = "\{\[\]\}"$ Output: true

Lab 2

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.

Lab 2

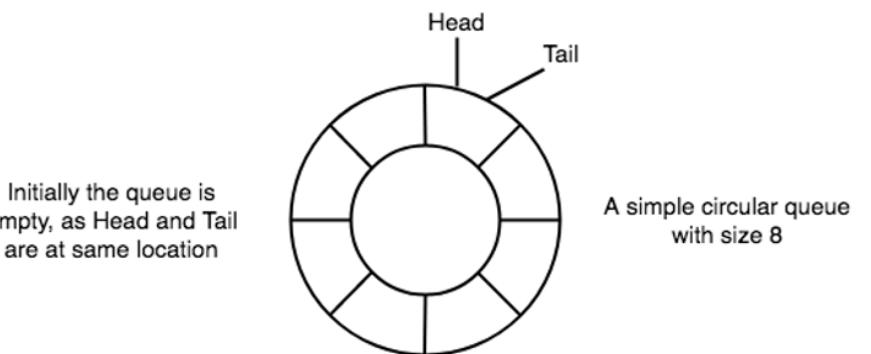
Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

Design your implementation of circular queue.

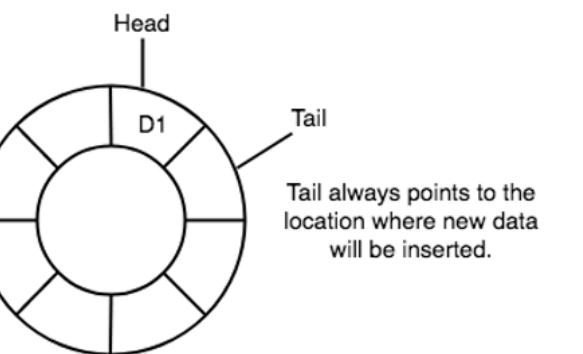
Lab 2

Circular Queue

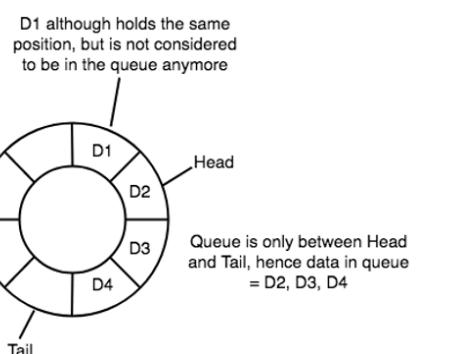
1. init



2. enqueue D1



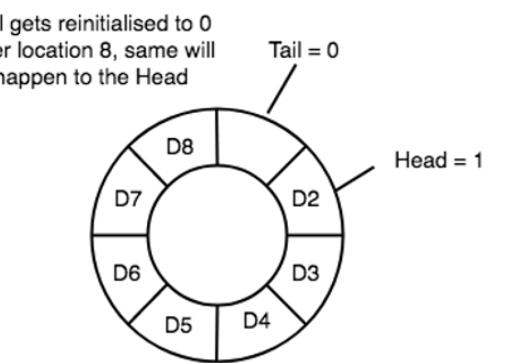
3. enqueue D2, D3, D4 and dequeue D1



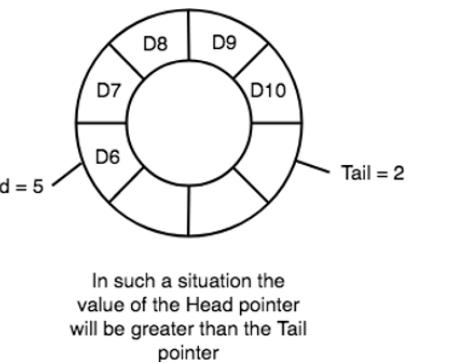
Lab 2

Circular Queue

4. enqueue D5, D6, D7, D8



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



Lab 2

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

Lab 3



- Python (lab1)
 - 5. misc.
 - 6. PEP8
- review Singly Linked List, Doubly Linked List
- review linear search, binary search
- exercise lab3.ipynb

Lab 3

lineary search & 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

Lab 3

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

Lab 3

Exercise: palindrome

Implement a Python function to determine if a string is a palindrome, for example, 'racecar' and 'level' are palindromes

Lab 4



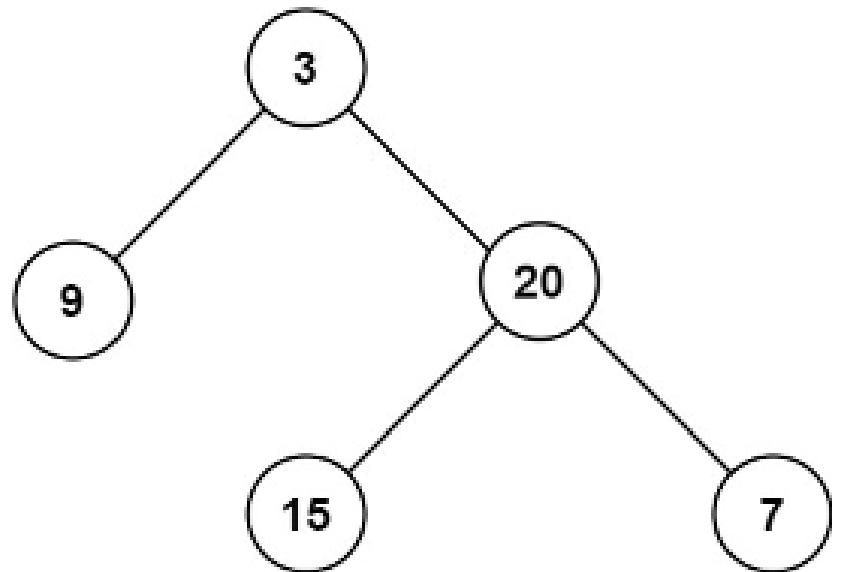
- review Binary Tree and 4 traverse methods
- exercise [lab4.ipynb](#)

Lab 4

Exercise: 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

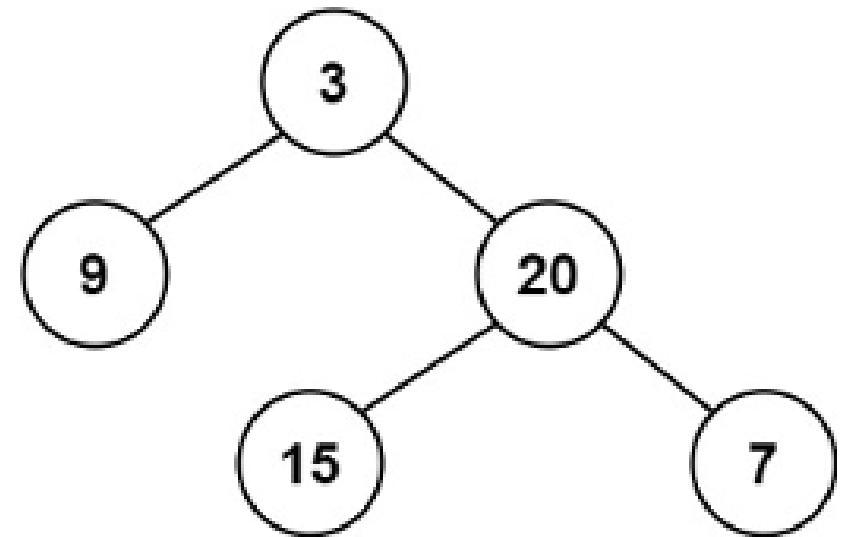


Lab 4

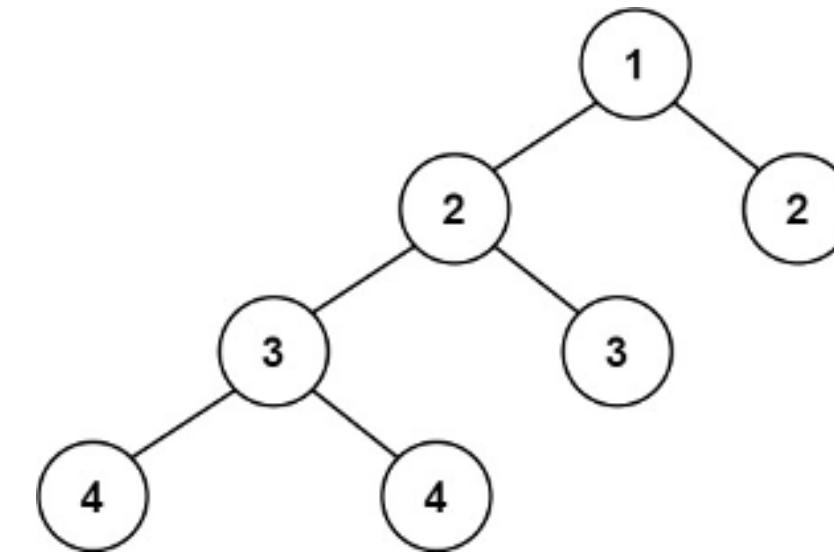
Exercise: check a balanced binary tree

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:

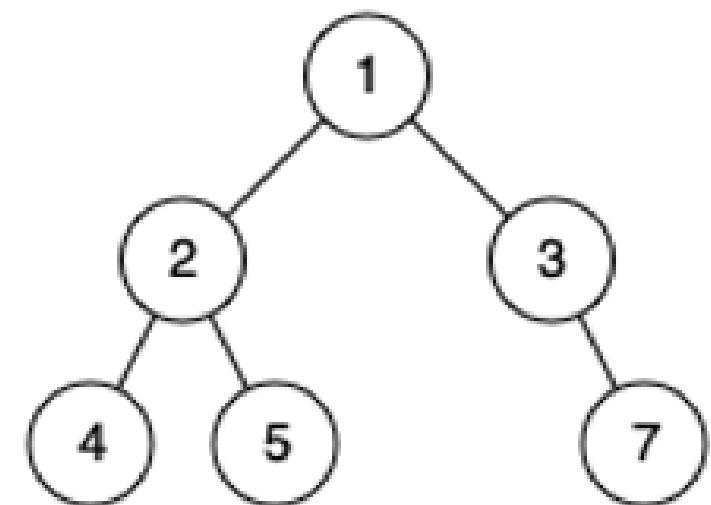


Lab 4

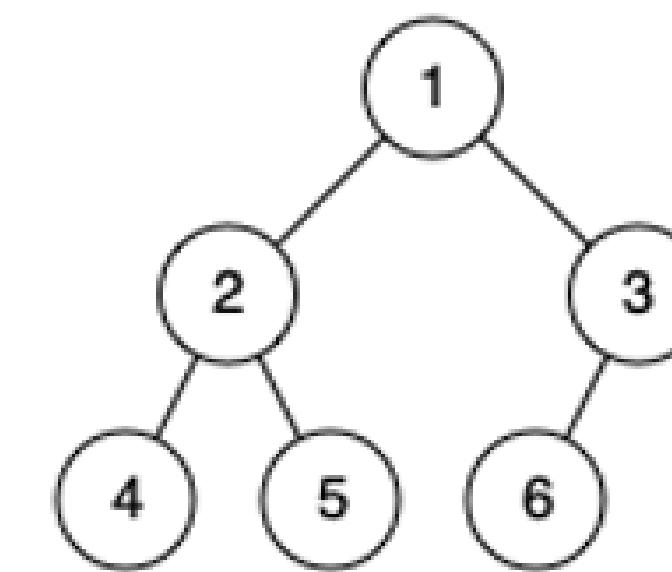
Exercise: check a complete binary tree

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:



Lab 5



- let us do some leetcode exercises  [lab5.ipynb](#)

Lab 5

Exercise: 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.

Lab 6



-  lab6.ipynb
-