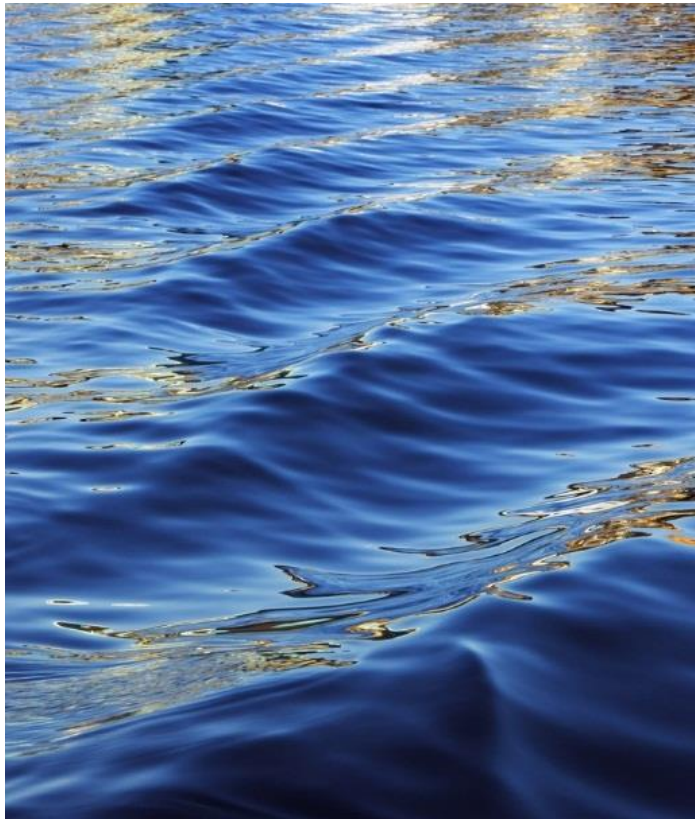
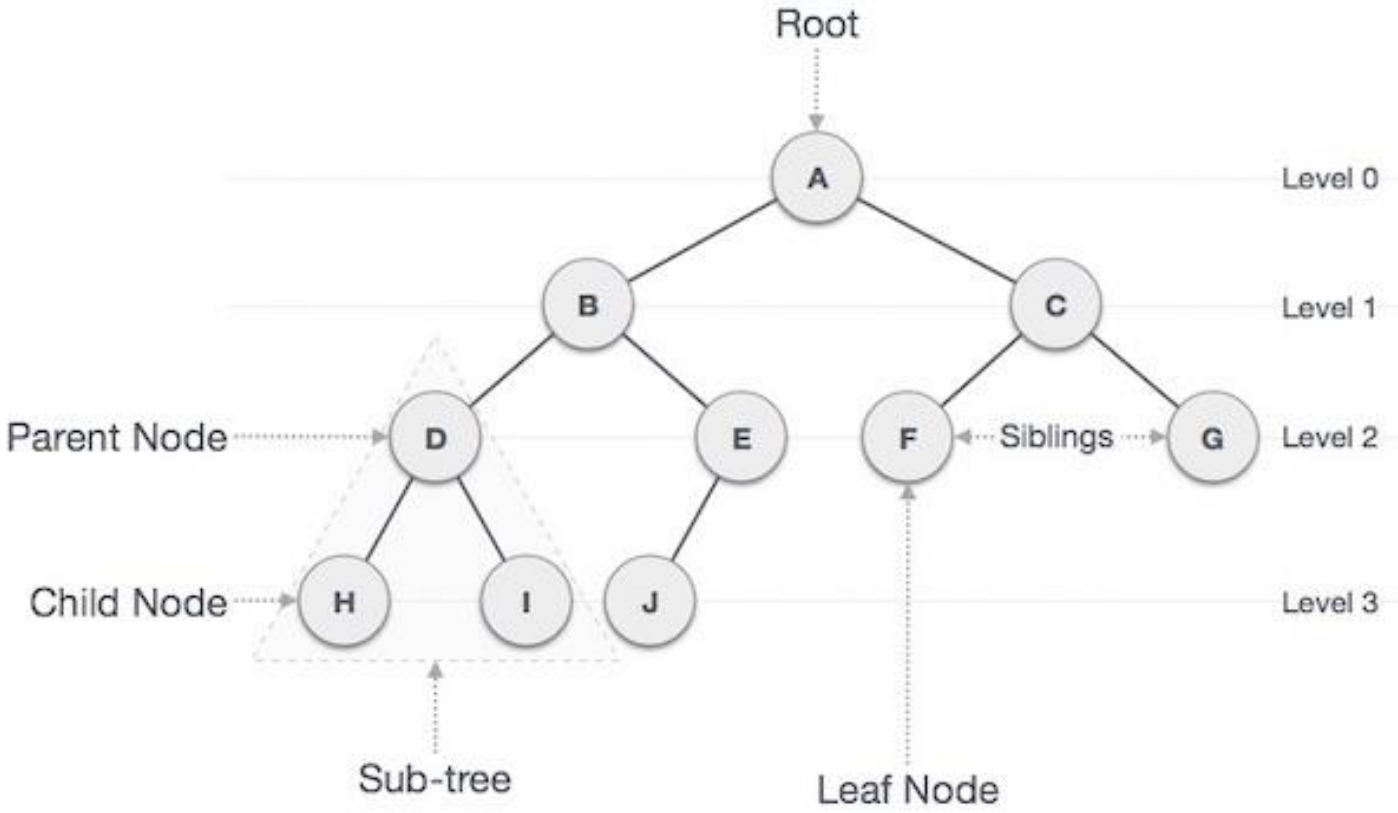
Tree & Tree Algorithms

Data structure and Algorithms (310-2101)



Tree Structure

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



Tree Structure

Terminology

Path − Path refers to the sequence of nodes along the edges of a tree

Root − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.

Parent − Any node except the root node has one edge upward to a node called parent.

Child − The node below a given node connected by its edge downward is called its child node.

Leaf − The node which does not have any child node is called the leaf node.

Subtree − Subtree represents the descendants of a node.

Visiting − Visiting refers to checking the value of a node when control is on the node.

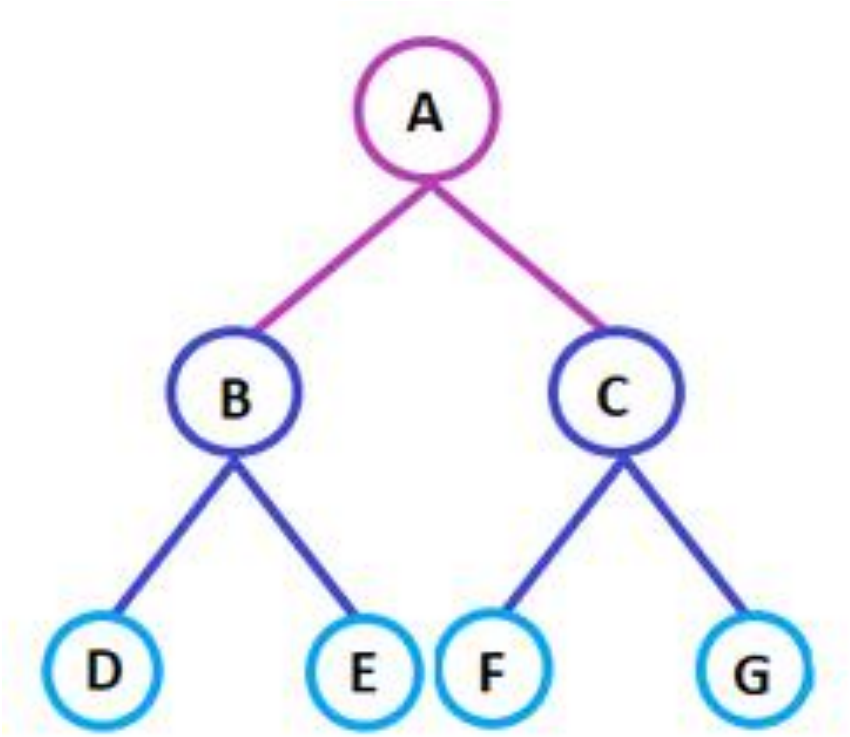
Traversing − Traversing means passing through nodes in a specific order.

Levels − Level of a node represents the generation of a node.

If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.

keys − Key represents a value of a node based on which a search operation is to be carried out for a node.

Tree Structure

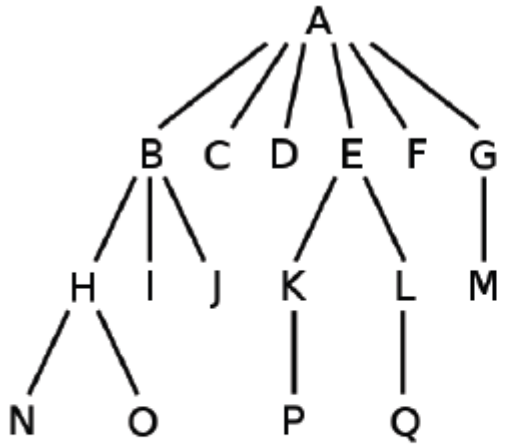


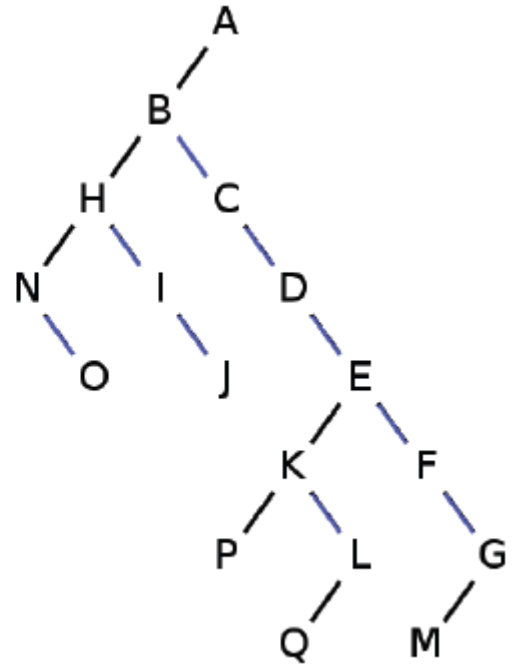
Example

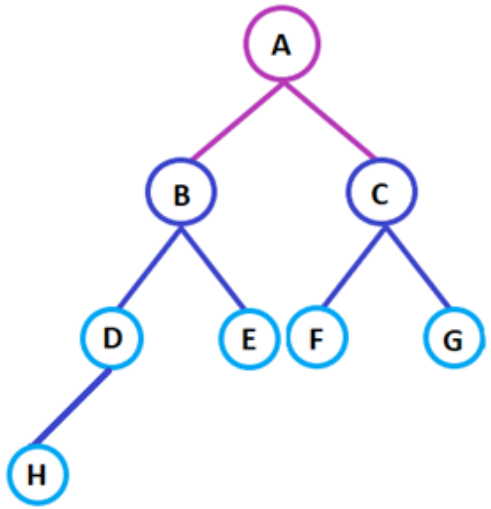
- Node A is the root node
- B is the parent of D and E
- D and E are the siblings
- D, E, and F are the leaf nodes
- A and B are the ancestors of E

Type of Tree Structure

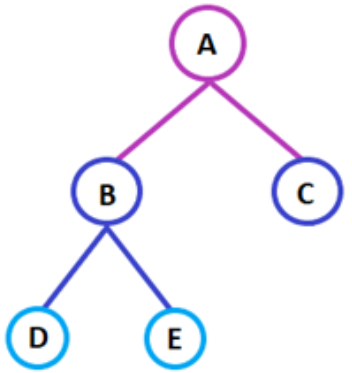
1. **General Tree**
2. **Binary Tree**
3. Balanced Tree
4. **Binary Search Tree**







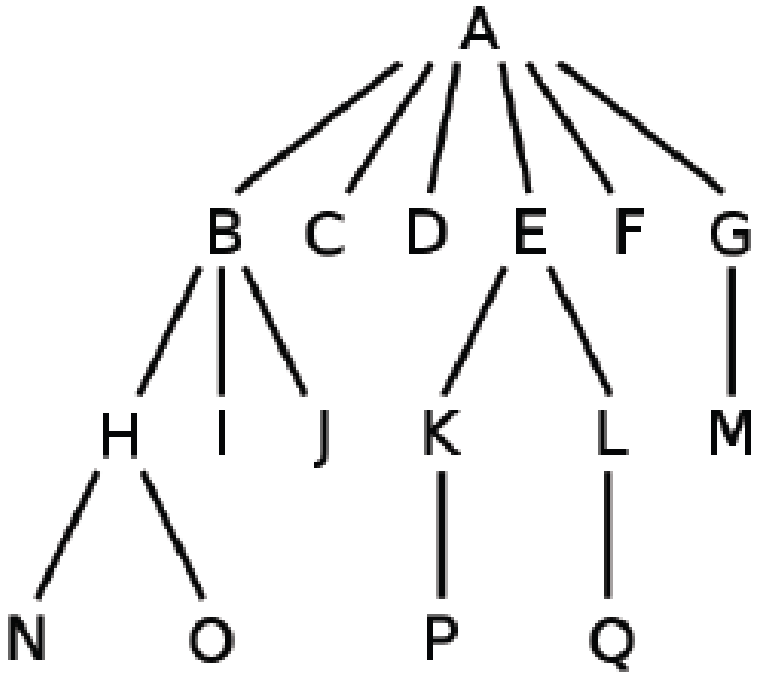
Balanced tree



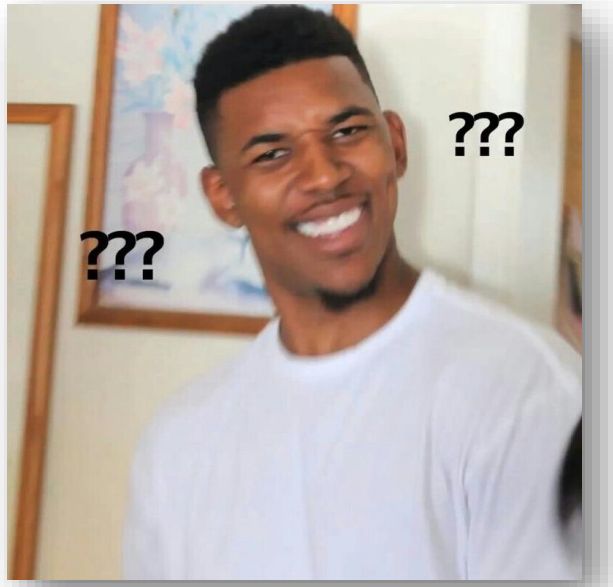
Unbalanced tree

Tree: Implementation

1.) General Tree **List of Lists Representation**

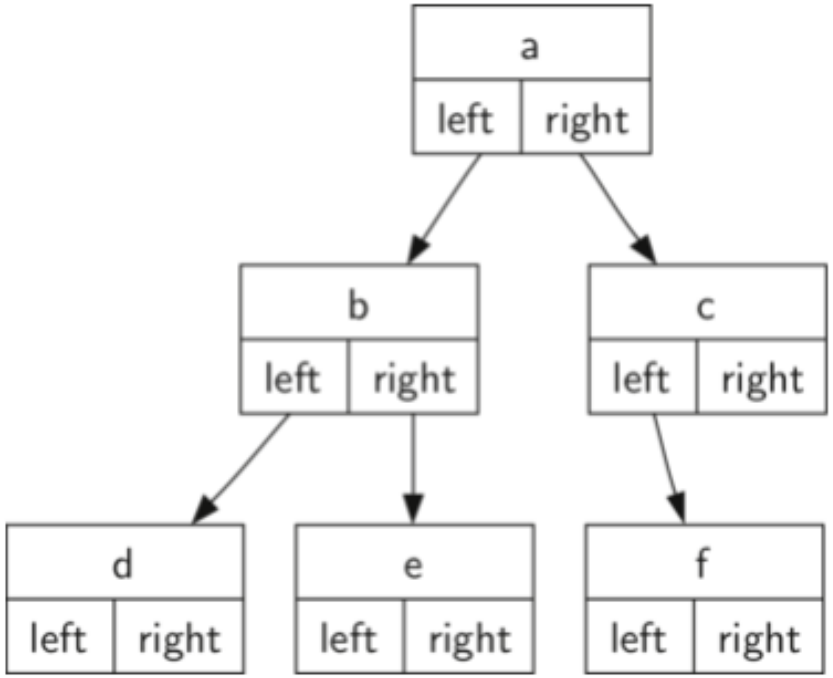



```
my_tree = ['a', #root
     ['b', #left subtree
      ['d' [], []],
      ['e' [], [] ],
     ['c', #right subtree
      ['f' [], []],
      [] ]
    ]
```



Tree: Implementation

2.) Binary Tree




```python

class BinaryTree():
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data

if __name__ == '__main__':

    r = BinaryTree('A')
    r.left = BinaryTree('B')
    r.right = BinaryTree('C')
    r.left.left = BinaryTree('D')
    r.left.right = BinaryTree('E')
    r.right.left = BinaryTree('F')

```

Tree: Implementation

2.) Binary Tree

```
1
2  class BinaryTree:
3      def __init__(self, root):
4          self.key = root
5          self.left_child = None
6          self.right_child = None
7
8      def insert_left(self, new_node):
9          if self.left_child == None:
10             self.left_child = BinaryTree(new_node)
11         else:
12             t = BinaryTree(new_node)
13             t.left_child = self.left_child
14             self.left_child = t
15
16     def insert_right(self, new_node):
17         if self.right_child == None:
18             self.right_child = BinaryTree(new_node)
19         else:
20             t = BinaryTree(new_node)
21             t.right_child = self.right_child
22             self.right_child = t
23
```

```
24     def get_right_child(self):
25         return self.right_child
26
27     def get_left_child(self):
28         return self.left_child
29
30     def set_root_val(self, obj):
31         self.key = obj
32
33     def get_root_val(self):
34         return self.key
35
```

Tree: Implementation

2.) Binary Tree

```
35
36  if __name__ == '__main__':
37
38      r = BinaryTree('a')
39      print(r.get_root_val())
40      print(r.get_left_child())
41
42      r.insert_left('b')
43      print(r.get_left_child().get_root_val())
44
45      r.insert_right('c')
46      print(r.get_right_child().get_root_val())
47
48      r.get_right_child().set_root_val('hello')
49      print(r.get_right_child().get_root_val())
50
```

Can you sketch the result?

Tree: Binary Tree Application

Heap : กองสะสม การทับถม

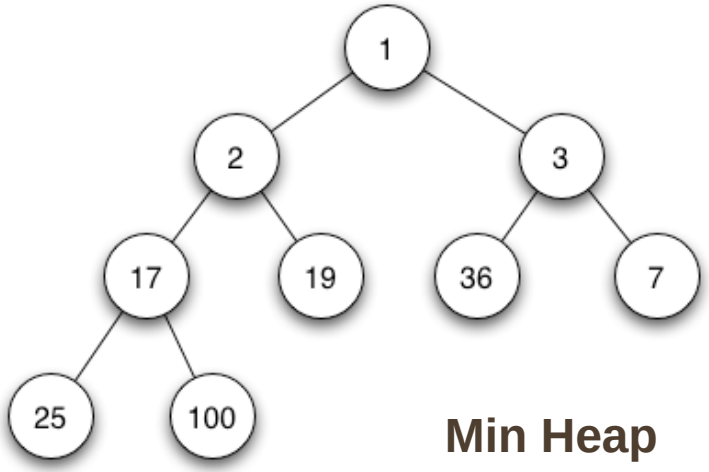
2.) Binary Tree: **Binary Heap**

A binary heap is a heap data structure that takes the form of a binary tree.

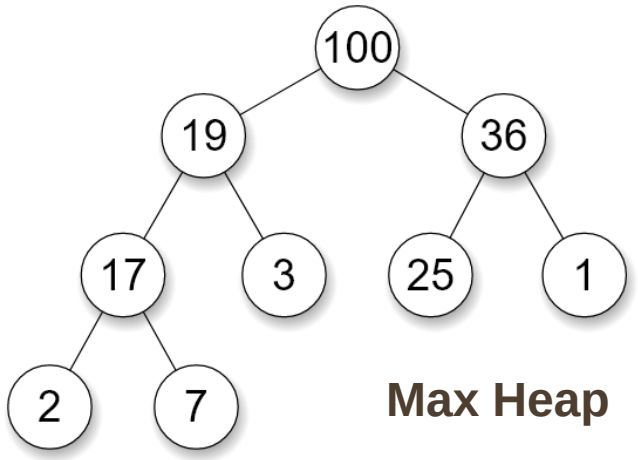
Binary heaps are a common way of implementing priority queues.

A binary heap is defined as a binary tree with two additional constraints:

- Shape property: a binary heap is a complete binary tree
- Heap property: the key stored in each node is either greater than or equal to (≥) or less than or equal to (≤) the keys in the node's children, according to some total order.



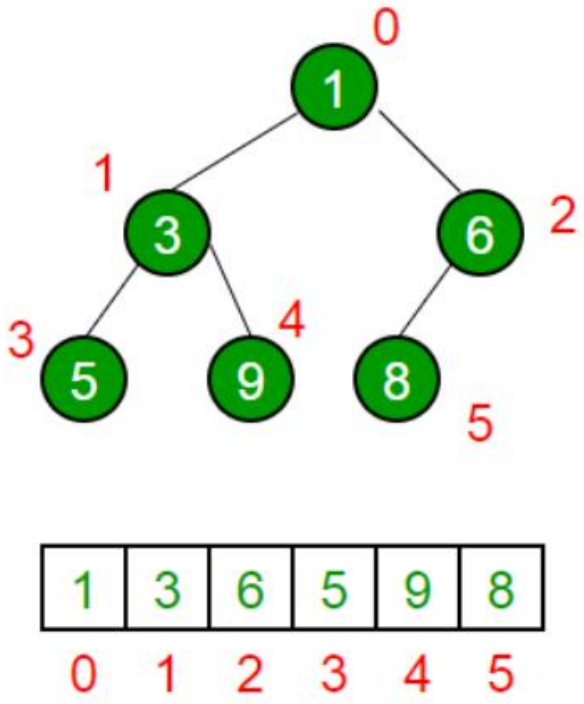
Min Heap



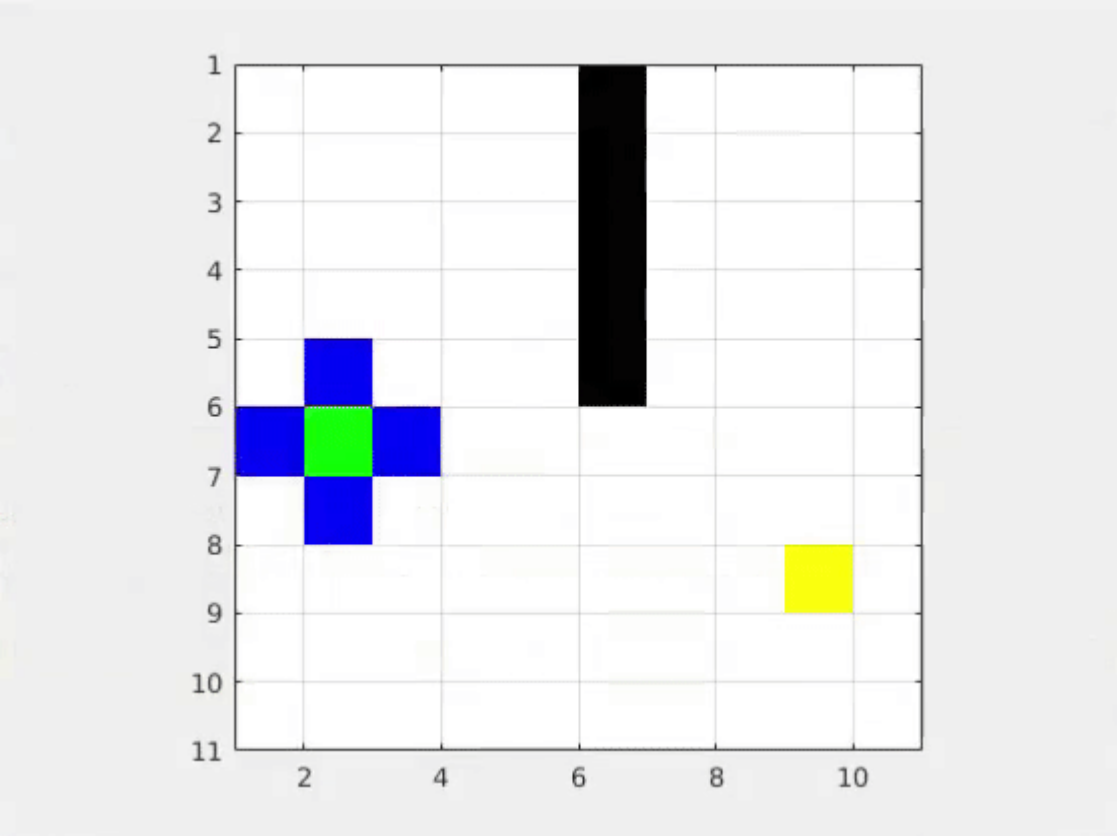
Max Heap

Tree: Binary Tree Application

2.) Binary Tree: **Binary Heap**



Array Representation Of Binary Heap



Dijkstra's algorithm

Shortest Path Finding

Tree: Binary Tree Application

2.) Binary Tree: **Binary Heap**

Do this…

[1] Create Binary Heap…

[2] Insert: 36, 5, 3, 10, 1 then print them and see the result…

```
54  if __name__ == '__main__':
55
56      h = BinHeap()
57      h.build_heap([5, 9, 11, 14, 18, 19, 21, 33, 17, 27])
58      for idx in range(0, h.current_size+1):
59          print(h.heap_list[idx])
60
```

Tree: Binary Tree Application

2.) Binary Tree: **Parse Tree**

1. If the current token is a '(', add a new node as the left child of the current node, and descend to the left child.
2. If the current token is in the list ['+','−','/','∗'], set the root value of the current node to the operator represented by the current token. Add a new node as the right child of the current node and descend to the right child.
3. If the current token is a number, set the root value of the current node to the number and return to the parent.
4. If the current token is a ')', go to the parent of the current node.

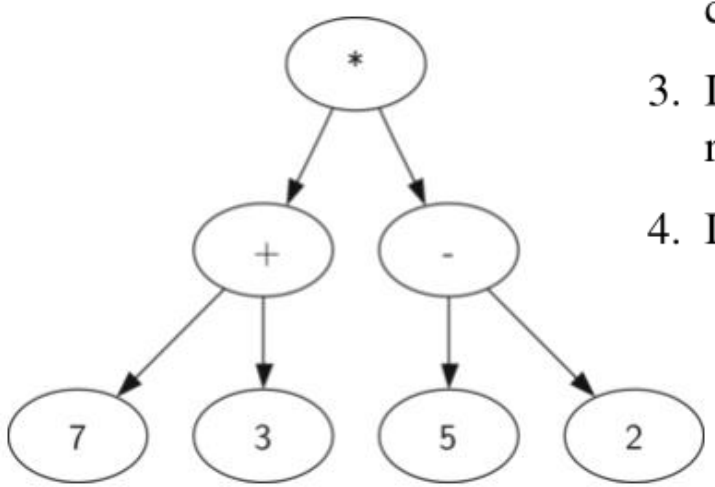


Figure 6.14: Parse Tree for $((7+3)*(5-2))$

Tre e Application

2.) B

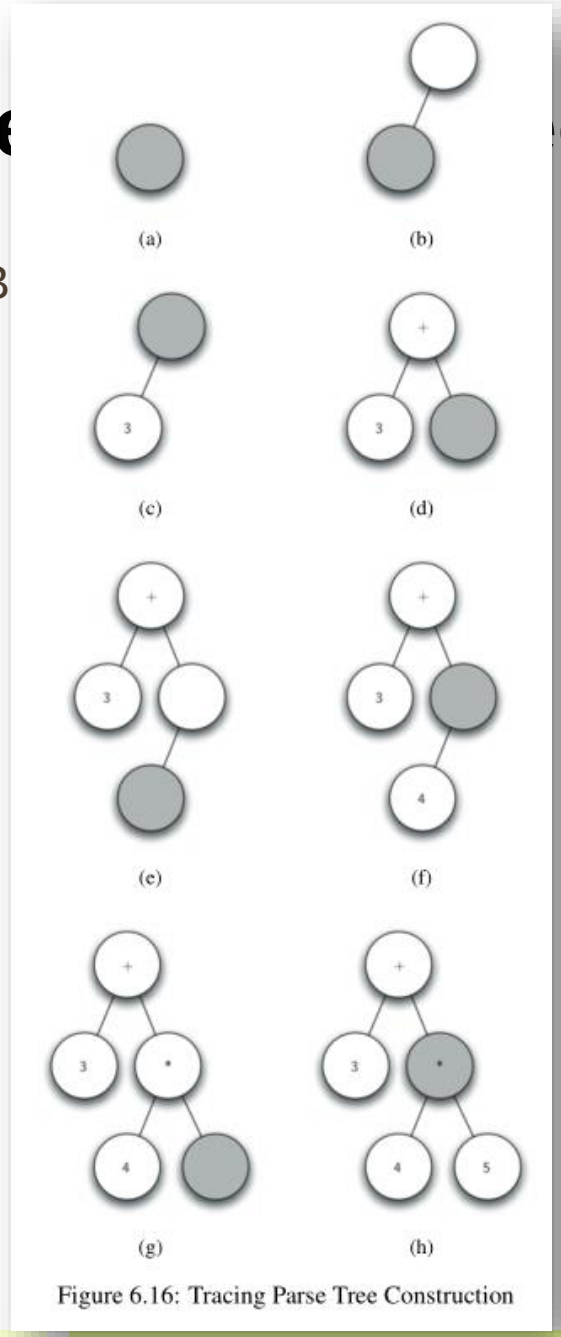


Figure 6.16: Tracing Parse Tree Construction

$$(3 + (4 * 5))$$

['(', '3', '+', '(', '4', '*', '5' ,')', ')']

1. Create an empty tree.
2. Read (as the first token. By rule 1, create a new node as the left child of the root. Make the current node this new child.
3. Read 3 as the next token. By rule 3, set the root value of the current node to 3 and go back up the tree to the parent.
4. Read + as the next token. By rule 2, set the root value of the current node to + and add a new node as the right child. The new right child becomes the current node.
5. Read a (as the next token. By rule 1, create a new node as the left child of the current node. The new left child becomes the current node.
6. Read a 4 as the next token. By rule 3, set the value of the current node to 4. Make the parent of 4 the current node.
7. Read * as the next token. By rule 2, set the root value of the current node to * and create a new right child. The new right child becomes the current node.
8. Read 5 as the next token. By rule 3, set the root value of the current node to 5. Make the parent of 5 the current node.
9. Read) as the next token. By rule 4 we make the parent of * the current node.
10. Read) as the next token. By rule 4 we make the parent of + the current node. At this point there is no parent for + so we are done.

Tree: Binary Tree Application

2.) Binary Tree: **Parse Tree**

Do this…

[1] Create Parse Tree

[2] Print the result with postorder() and preorder() and see the difference

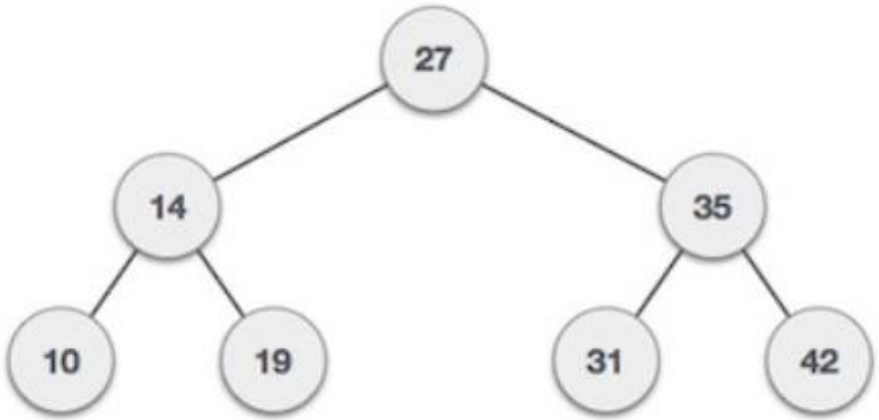
```
101
102  if __name__ == '__main__':
103      pt = build_parse_tree("( ( 10 + 5 ) * ( 3 - 2 ) )")
104      pt.postorder()
105
```

Tree: Binary Tree - Traversals

preorder In a preorder traversal, we visit the root node first, then recursively do a preorder traversal of the left subtree, followed by a recursive preorder traversal of the right subtree.

inorder In an inorder traversal, we recursively do an inorder traversal on the left subtree, visit the root node, and finally do a recursive inorder traversal of the right subtree.

postorder In a postorder traversal, we recursively do a postorder traversal of the left subtree and the right subtree followed by a visit to the root node.



Preorder: [27] [14] [10] [19] [35] [31] [42]

Inorder: [10] [14] [19] [27] [31] [35] [42]

Postorder: [10] [19] [14] [31] [42] [35] [27]

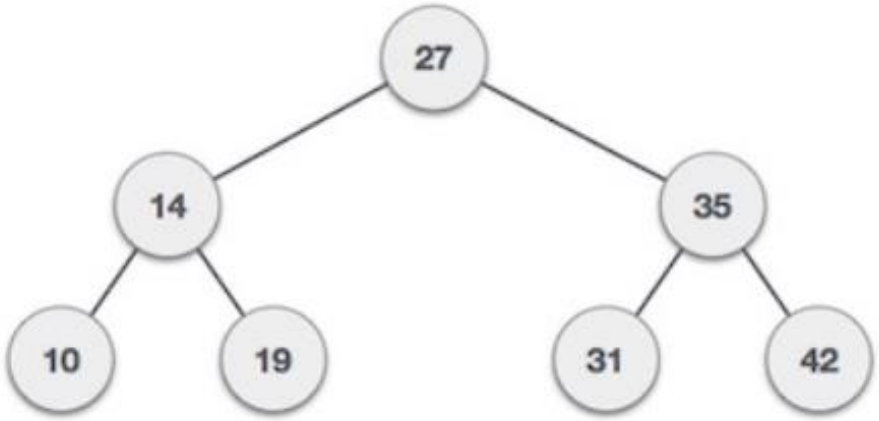
Tree: Binary Tree - Traversals

Preorder: [27] [14] [10] [19] [35] [31] [42]

Inorder: [10] [14] [19] [27] [31] [35] [42]

Postorder: [10] [19] [14] [31] [42] [35] [27]

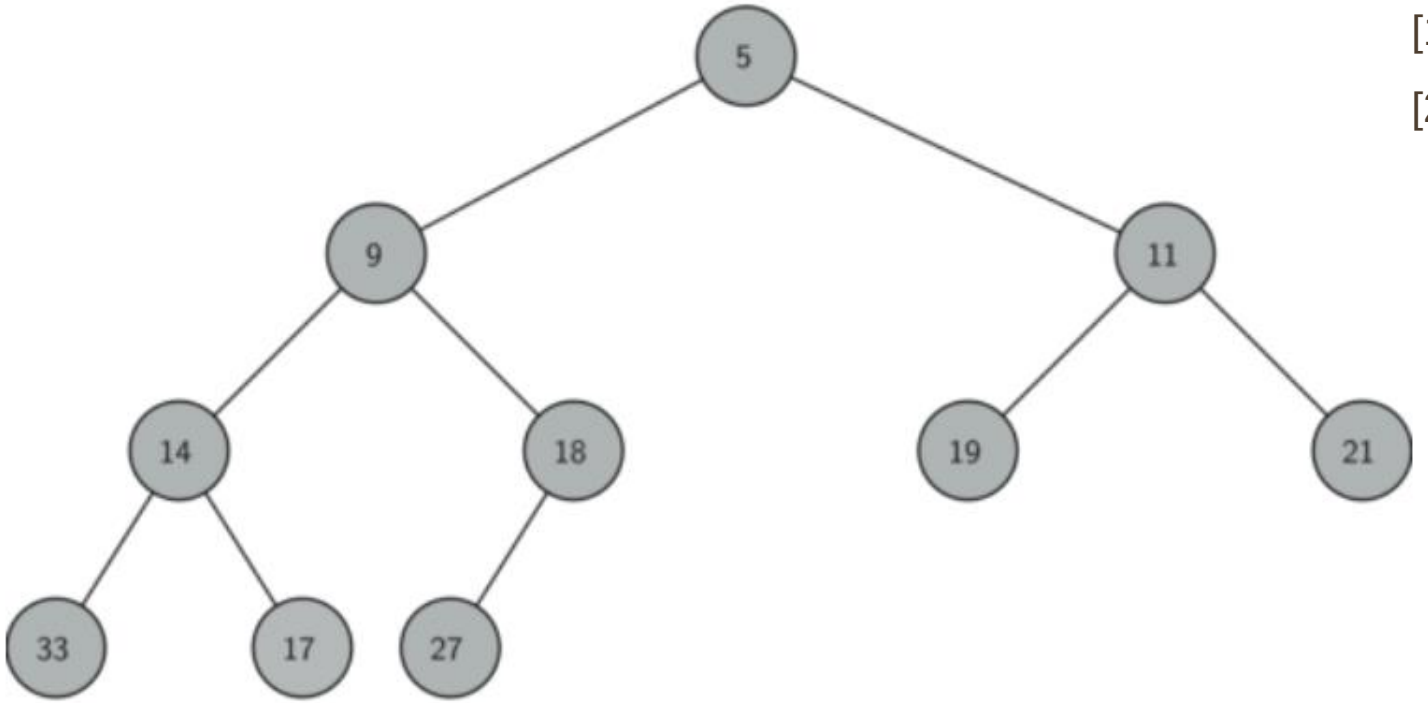
2.) Binary Tree: **Traversals**




```
8   # In-order Traversal
9   def inorder(node):
10      if node:
11          inorder(node.left)
12          print(node.data)
13          inorder(node.right)
14
15  # Pre-order Traversal (Breath-First Search:BFS)
16  def preorder(node):
17      if node:
18          print(node.data)
19          preorder(node.left)
20          preorder(node.right)
21
22  # Post-order Traversal (Depth-First Search:DFS)
23  def postorder(node):
24      if node:
25          postorder(node.left)
26          postorder(node.right)
27          print(node.data)
28
```

Tree: Binary Tree - Traversals

2.) Binary Tree: **Traversals**



Do this…

[1] Build this tree

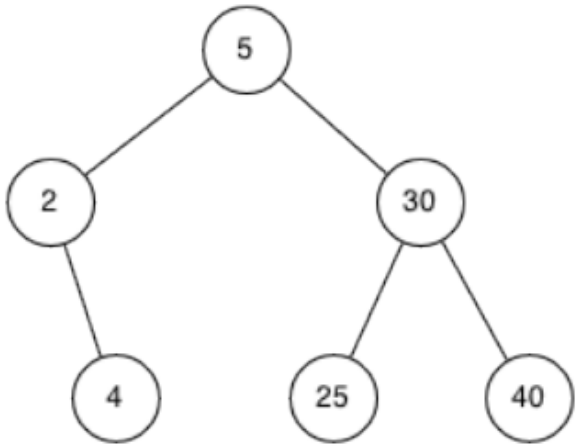
[2] Print tree with 3 types of traversals

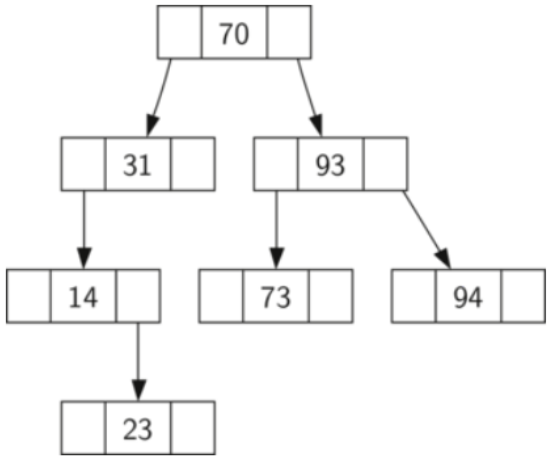
Tree: Binary Search Tree (BST)

3.) Binary Search Tree (BST)

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties.The left sub-tree of a node has a key less than or equal to its parent node's key.The right sub-tree of a node has a key greater than to its parent node's key.Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree

```
left_subtree (keys)  ≤  node (key)  ≤  right_subtree (keys)
```





Tree: Binary Search Tree (BST)

3.) Binary Search Tree (BST) : Implementation

Do this…

[1] Download Binary Search Tree (BST) from Classroom

[2] Enter the following code

[3] Are there any error when we try to access [0], [1] ?

```
199  if __name__ == '__main__':
200
201      my_tree = BinarySearchTree()
202      my_tree[3] = 16
203      my_tree[4] = 6
204      my_tree[6] = 1
205      my_tree[2] = 468
206
207      print(my_tree[0])
208      print(my_tree[1])
209      print(my_tree[2])
210      print(my_tree[3])
211      print(my_tree[4])
212      print(my_tree[5])
213      print(my_tree[6])
214
```

Take Home Message (Tree Structure)

[1] Tree Data Structure

- Binary Heap
- Binary Tree
- Binary Search Tree (BST)

[2] Binary Tree: Parse Tree

[3] Binary Tree: Traversals

Bit of Knowledge

Overload Operator in Python

Operator	Magic Method
+	__add__(self, other)
-	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)
>>	__rshift__(self, other)
<<	__lshift__(self, other)
&	__and__(self, other)
|	__or__(self, other)
^	__xor__(self, other)

```python
# ============================================================ #
#                                                              #
# Operator Overloading in Python                               #
# https://www.geeksforgeeks.org/operator-overloading-in-python/ #
#                                                              #
# ============================================================ #

class complex:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    # adding two objects
    def __add__(self, other):
        return self.a + other.a, self.b + other.b

if __name__ == '__main__':
    Ob1 = complex(1, 2)
    Ob2 = complex(2, 3)
    Ob3 = Ob1 + Ob2
    print(Ob3)

```

Bit of Knowledge

Overload Operator in Python

Comparison Operators :

OperatorMagic Method	Method
<	__LT__(SELF, OTHER)
>	__GT__(SELF, OTHER)
<=	__LE__(SELF, OTHER)
>=	__GE__(SELF, OTHER)
==	__EQ__(SELF, OTHER)
!=	__NE__(SELF, OTHER)

Assignment Operators :

OperatorMagic Method	Method
-=	__ISUB__(SELF, OTHER)
+=	__IADD__(SELF, OTHER)
*=	__IMUL__(SELF, OTHER)
/=	__IDIV__(SELF, OTHER)
//=	__IFLOORDIV__(SELF, OTHER)
%=	__IMOD__(SELF, OTHER)
**=	__IPOW__(SELF, OTHER)
>>=	__IRSHIFT__(SELF, OTHER)
<<=	__ILSHIFT__(SELF, OTHER)
&=	__IAND__(SELF, OTHER)
|=	__IOR__(SELF, OTHER)
^=	__IXOR__(SELF, OTHER)

Unary Operators :

OperatorMagic Method	Method
-	__NEG__(SELF, OTHER)
+	__POS__(SELF, OTHER)
~	__INVERT__(SELF, OTHER)

Bit of Knowledge

Overload Function in Python

```
1
2   class Purchase:
3       def __init__(self, basket, consumer):
4           self.basket = list(basket)
5           self.consumer = consumer
6
7       def __len__(self):
8           print('Call from overload function (>_<) this is fake length')
9           return 10
10
11  if __name__ == '__main__':
12      purchase = Purchase(['pencil ', 'book '], 'python ')
13      print(len(purchase))
```

Bit of Knowledge

Overload Function in Python

```
11  if __name__ == '__main__':
12
13      print(dir(list))
14
```

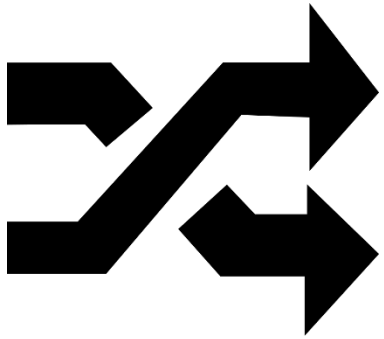
Python - Magic or Dunder Methods

```
>>> dir(int)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
'__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__',
'__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__',
'__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__',
'__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__',
'__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__',
'__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__',
'__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__',
'__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag',
'numerator', 'real', 'to_bytes']
```

Random Number Generator

คอมพิวเตอร์สามารถสร้างการสุ่มตัวเลขขึ้นมาได้อย่างไร
(How Do Computers Generate Random Numbers?)

ตัวเลขสุ่ม (Random number) นั้นเป็นข้อมูลที่สามารถนำมาใช้ประโยชน์ได้หลากหลายงาน เช่น การเข้ารหัส การทำให้เกิดผลลัพธ์ที่มีความหลากหลายไม่ซ้ำซ้อน การสุ่มตัวอย่างเพื่อทดสอบความผิดพลาดของกระบวนการทำงาน



การสุ่มนั้นเป็นสิ่งที่ทำให้เกิดขึ้นในระบบคอมพิวเตอร์ได้ยาก เนื่องจากในระบบคอมพิวเตอร์ การสร้างผลลัพธ์ต่างๆ ให้เกิดขึ้น เป็นสิ่งที่มีสาเหตุและที่มาเสมอ ทำให้เราสามารถคำนวณหาผลลัพธ์ได้แน่นอน

Method to generate random number

[1] True Random Numbers (TRNGs)

กระบวนการสุ่มซึ่งทำให้เกิดผลลัพธ์ทางกายภาพ (Physical Process) โดยการเก็บผลลัพธ์ที่ได้จากปรากฏการณ์ทางฟิสิกส์ที่คาดหวังว่าผลลัพธ์มีลักษณะสุ่ม (physical phenomenon that expected to be random)

photoelectric effect,

cosmic background radiation,

atmospheric noise

Method to generate random number

[2] Pseudorandom Numbers (PRNGs)

ตัวเลขสุ่มถึงถูกสร้างขึ้นจาก Algorithm ที่ทำให้เกิดผลลัพธ์ในลักษณะสุ่ม ตัวเลขสุ่มที่ได้จากกรณีนี้เป็นตัวเลขที่ไม่ได้เกิดการ Random จริง เพราะสุดท้ายแล้ว Algorithm ต่างๆ นั้นมี สมการ อยู่เบื้องหลังทำให้เราสามารถคาดการณ์ผลลัพธ์ล่วงหน้าได้เสมอ

การสร้างตัวเลขสุ่มในรูปแบบนี้จะเกิดจากค่าตั้งต้น (Initial value)

ซึ่งมักจะถูกเรียกว่า Seed หรือ Key

ในทางปฏิบัติงานทางคอมพิวเตอร์ ตัวเลขสุ่ม หรือ ตัวแปรสุ่มที่ได้จาก PRNGs นั้นมีความสะดวกมากกว่า เพราะสามารถทำการทดสอบตัวเองซ้ำ (Repeating itself) ในเหตุการณ์เดิมได้ (Reproduce same result later)

Pseudorandom Numbers Generator

กระบวนการสุ่มในรูปแบบนี้จะประกอบไปด้วยกระบวนการพื้นฐาน 4 ขั้นตอน

- Accept some initial input number, that is a seed or key
- Apply that seed in a sequence of mathematical operations to generate the result.

  That result is the random number
- Use that resulting random number as the seed for the next iteration
- Repeat the process to emulate randomness

Pseudorandom Numbers Generator

Linear Congruential Generator

most common and oldest algorithm for generating pseudo randomized numbers

$$X_{n+1} = (aX_n + c) \bmod m$$

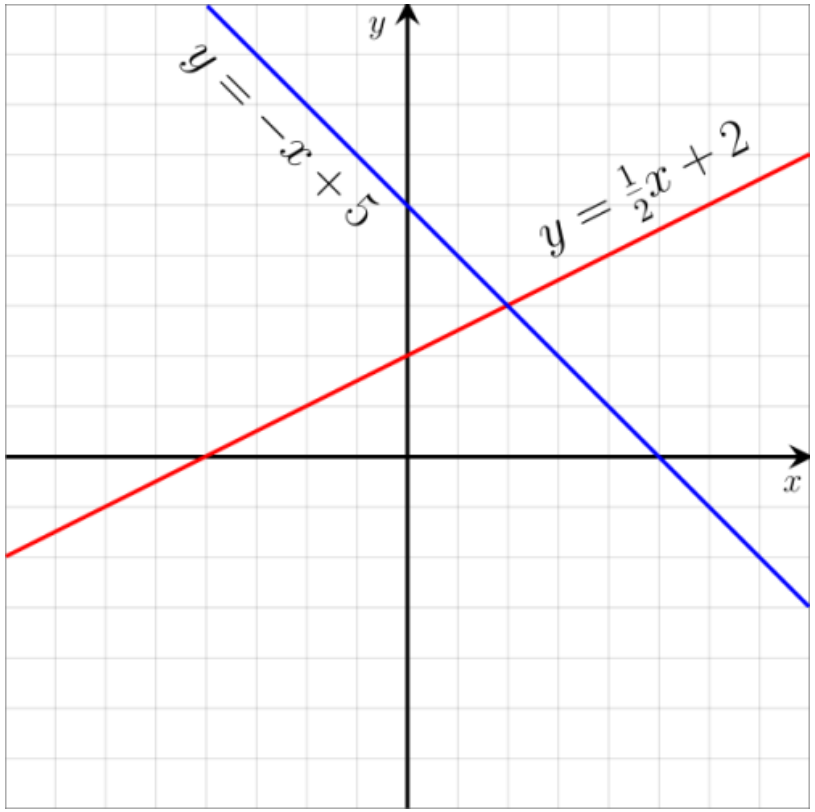
ตัวแปร *X* เป็น Sequence ของค่า Pseudorandom number

m, $0 < m$ — the "modulus"

a, $0 < a < m$ — the "multiplier"

c, $0 \leq c < m$ — the "increment"

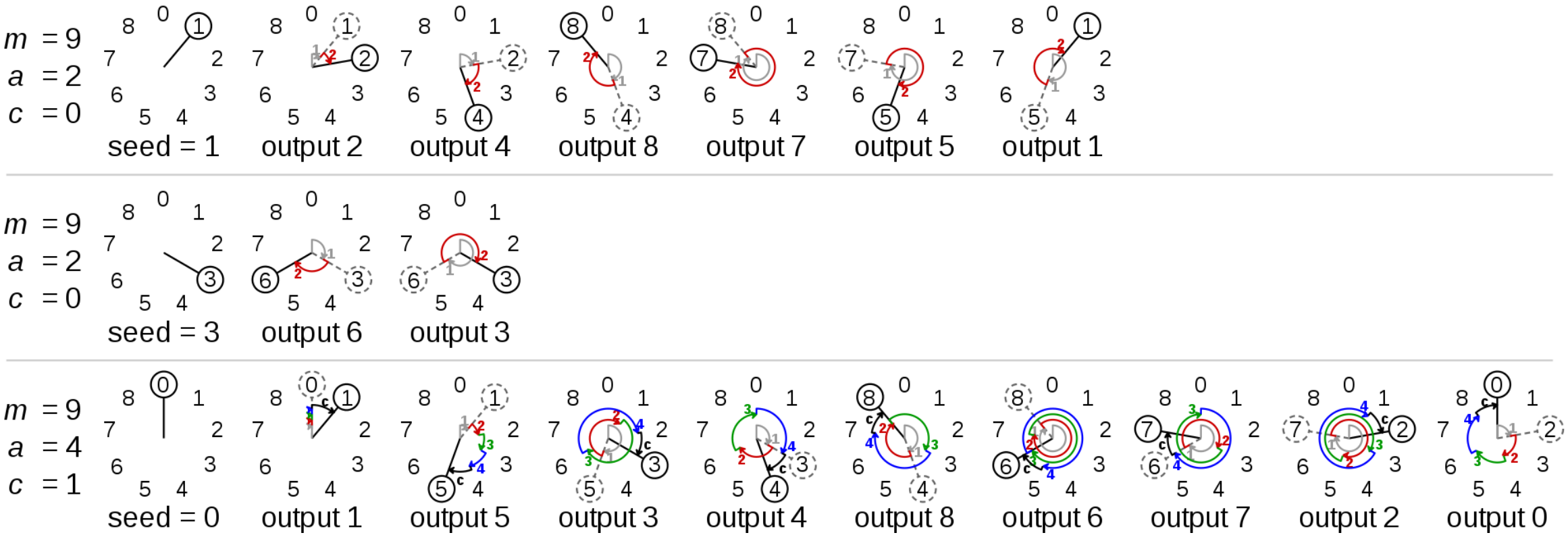
X_0, $0 \leq X_0 < m$ — the "seed" or "start value"



Pseudorandom Numbers Generator

Linear Congruential Generator

most common and oldest algorithm for generating pseudo randomized numbers



Pseudorandom Numbers Generator

Linear Congruential Generator

most common and oldest algorithm for generating pseudo randomized numbers

Source	modulus *m*	multiplier *a*	increment *c*	output bits of seed in *rand()* or *Random(L)*
Borland C/C++	2^{32}	22695477	1	bits 30..16 in *rand()*, 30..0 in *lrand()*
glibc (used by GCC)[17]	2^{31}	1103515245	12345	bits 30..0
C++11's `minstd_rand` [22]	$2^{31} - 1$	48271	0	see MINSTD
MMIX by Donald Knuth	2^{64}	6364136223846793005	1442695040888963407	

Pseudorandom Numbers Generator

Linear Congruential Generator

```python
from collections.abc import Generator

def lcg(modulus: int, a: int, c: int, seed: int) -> Generator[int, None, None]:
    """Linear congruential generator."""
    while True:
        seed = (a * seed + c) % modulus
        yield seed

if __name__ == "__main__":
    m = 9
    a = 2
    c = 0
    seed = 1

    print("Random number")
    for x in lcg(m,a,c,seed):
        print(x, end=" ")

```

What does the "yield" keyword do?

Pseudorandom Numbers Generator

What does the “yield” keyword do?

yield นั้นมีความคล้ายกับ return สิ่งที่ทำให้ 2 keyword นี้มีความแตกต่างกัน คือ

return statement จะสิ้นสุด (terminate) เมื่อมีการเรียกฟังก์ชัน (execution of the function)

yield statement จะหยุดชั่วคราว (pause) เมื่อมีการเรียกฟังก์ชัน (execution of the function)

```
1   def fun_generator():
2       yield "Hello world!!"
3       yield "Geeksforgeeks"
4
5   if __name__ == "__main__":
6       obj = fun_generator()
7
8       print(type(obj))
9
10      print(next(obj))
11      print(next(obj))
12
```

```
<class 'generator'>
Hello world!!
Geeksforgeeks
```

ใช้ Yield เมื่อทำ Generator function

จะให้ผลลัพธ์ที่รวดเร็วกว่า Return

Pseudorandom Numbers Generator

What does the “yield” keyword do?

🔷 ข้อดีของ `yield`

- ✅ **ประหยัดหน่วยความจำ** – คืนค่าเฉพาะที่ต้องการ ไม่ต้องโหลดข้อมูลทั้งหมด
- ✅ **ทำให้โค้ดอ่านง่าย** – เหมาะกับการวนลูปข้อมูลขนาดใหญ่
- ✅ **รองรับการทำงานแบบขนาน** – สามารถสร้าง pipeline ของข้อมูลแบบ streaming ได้

🔷 ใช้เมื่อไหร่?

- เมื่อทำงานกับ **ข้อมูลขนาดใหญ่** เช่น อ่านไฟล์ทีละบรรทัด (`yield` จะช่วยให้ไม่ต้องโหลดไฟล์ทั้งหมดลงใน RAM)
- เมื่อสร้าง **stream ของข้อมูล** เช่น คำนวณตัวเลขที่ต้องการเรียกใช้แบบต่อเนื่อง
- เมื่อสร้าง **infinite sequence** เช่น ตัวเลขฟีโบนักชีที่สามารถดึงค่าได้เรื่อย ๆ

```python
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib = fibonacci()
for _ in range(5):
    print(next(fib))  # 0, 1, 1, 2, 3
```

Pseudorandom Numbers Generator

```python
# generator to print even numbers
def print_even(test_list):
    for i in test_list:
        if i % 2 == 0:
            yield i

# initializing list
test_list = [1, 4, 5, 6, 7]

# printing initial list
print("The original list is : " + str(test_list))

# printing even numbers
print("The even numbers in list are : ", end=" ")
for j in print_even(test_list):
    print(j, end=" ")
```

Pseudorandom Numbers Generator

Characteristics of PRNG

- **Efficient:** PRNG **can produce many numbers in a short time** and is advantageous for applications that need many numbers
- **Deterministic:** A given sequence of numbers can be reproduced at a later date if the starting point in the sequence is known. Determinism is handy **if you need to replay** the same sequence of numbers again at a later stage.
- **Periodic:** PRNGs are periodic, which means that the **sequence will eventually repeat itself**. While periodicity is hardly ever a desirable characteristic, modern PRNGs have a period that is so long that it can be ignored for most practical purposes

Pseudorandom Numbers Generator

Python code to generate PRNG

Example2.py > ...

```python
import random
from datetime import datetime

# Passing the current time as the seed value
random.seed(datetime.now())

for i in range(5):
    print(random.randint(0, 10), end="\t")

```

From statistics to probability

From a simple coin...

... we can generate a series of flips as our **data**

T, T, H, T, H, H...

And then calculate the results of what we observed

H: 43%, T: 57%

If we keep generating more data sets and calculations...

H, H, H, T, T, H...
H: 53%, T: 47%

Repeat many many times....

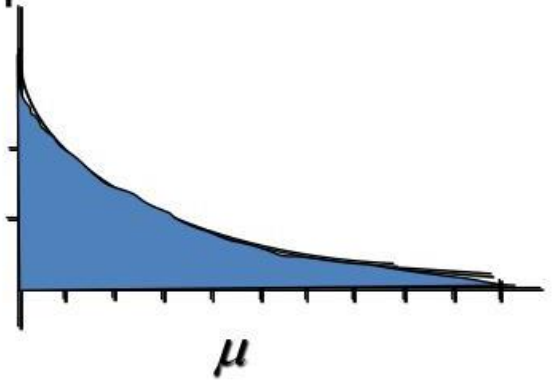
We can take the average of these results.

H: ~50%, T: ~50%

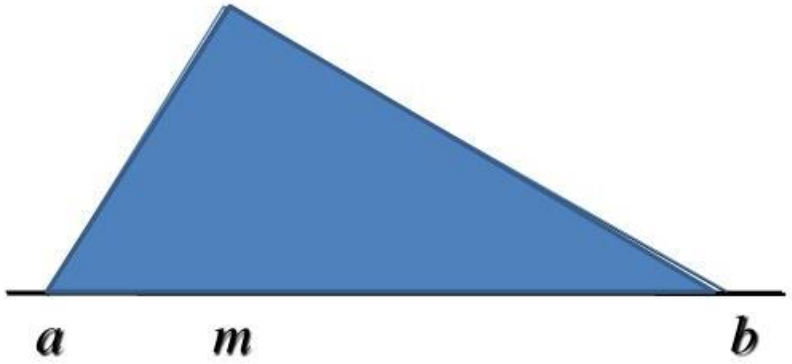
Transforming uniform random variables to other distributions

Uniform Distribution

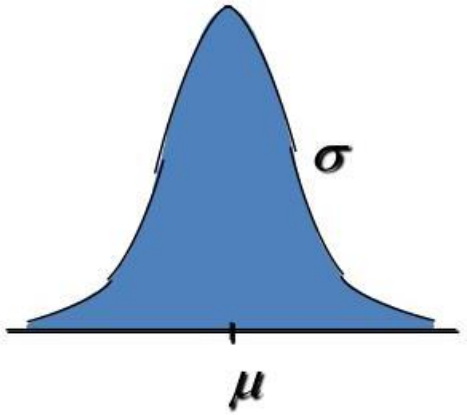
Exponential Distribution



Triangular Distribution



Normal Distribution



Just some idea to control rate of random value (result)

Legendary	Rare	Common
5%	20%	75%

How to generate this possibility from typical rand function?

This is just for fun, don’t need to take it serious

https://www.ascii-art-generator.org

โปรแกรมจัดการ Card ใน Deck พื้นฐาน

พัฒนาโปรแกรม จัดการสำรับไพ่ (Deck) ให้มีความสามารถต่อไปนี้

1.	Put()	วาง Card เข้าไปในสำรับไพ่ (ด้านบน)
2.	Draw()	หยิบ Card ใบบนสุดออกมาจากสำรับ
3.	Show()	แสดง Card ทั้งหมดในสำรับจาก บน -> ล่าง
4.	Sort()	เรียงตามตัวเลข 3 < 4 < 5 < ... < 10 < 11 < 12 < 13 < 1 < 2 ค่าของหน้าไพ่ club < diamonds < heart < spade ถ้าไพ่สองใบมีตัวเลขเท่ากัน ให้เปรียบเทียบจากหน้าไพ่
5.	Shuffle()	สลับตำแหน่งของหน้าไพ่ใน Deck
6.	Sum()	รวมตัวเลขของไพ่ในแต่ละหน้าแล้วทำการแสดงผล

Card.py > ...

```python
class Card:
    def __init__(self, value, suit):
        self.value = None
        self.suit = None
        self.set(value, suit)

    def set(self, value, suit):...

    def print(self):...

if __name__ == "__main__":

    import os
    os.system('cls')

    a = Card("5", "s")
    a.print()
```

Card
value suit
set(value, suit) print()

นักศึกษาสามารถดัดแปลง
แก้ไข Source Code ที่ให้ได้

Deck.py > ...

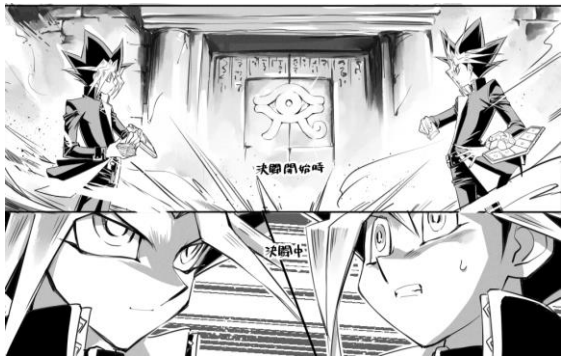
```python
from Card import *

class Deck:
    def __init__(self):
        cards = []

    def put():
        pass

    def draw():
        pass

    def show():
        pass

    def sort():
        pass

    def shuffle():
        pass

    def sum():
        pass

```

Homework ???: Card Battle Game

1) แต่ละฝ่ายสุ่มหยิบไพ่ขึ้นมาใส่สำหรับของตัวเอง 5 ใบ
2) ใช้หน้าไพ่ (suit) เป็นตัวกำหนดการกระทำ

   [C] Draw เพิ่ม, [S] สุ่มทิ้งการ์ดอีกฝ่ายหนึ่งใบ,

   [D] ลด Health Point ฝ่ายตรงข้าม, [H] ขโมยการ์ดฝ่ายตรงข้ามหนึ่งใบ
3) ใช้ตัวเลข (value) เป็นตัวกำหนดโอกาส ที่จะทำสำเร็จ
4) แต่ละรอบ (Turn) ให้แต่ละฝ่ายหยิบการ์ดเพิ่ม 1 ใบ
5) ฝ่ายที่ Health Point เป็น 0 ก่อนถือว่า “แพ้” => จบเกม
6) สำรับไพ่กองกลางมีจำนวนจำกัด (13 x 4) ใบ หมดสำหรับ => “เสมอ”



Health Point

เริ่มต้นที่ 7