Tree

ตัวอย่างของ Tree

- เราได้เรียน linear data structures มาแล้วเช่น stack queue และได้เรียน recursion ต่อไปเราจะมาดูโครงสร้างข้อมูลอีกตัวที่เรียกว่า Tree
- Tree ถูกใช้ในหลายๆ ด้านของวิทยาการคอมพิวเตอร์เช่นใน operating systems, graphics, databases networking เป็นต้น
- Tree เป็นโครงสร้างข้อมูลที่คล้ายกับต้นไม้ทางชีววิทยาจริงๆ

- โครงสร้างข้อมูลแบบ Tree โดยทั่วไปแล้วจะมี root, branches และ leaves
- จุดที่แตกต่างอย่างหนึ่งระหว่างต้นไม้จริงกับ tree ในทางวิทยาการคอมพิวเตอร์คือ โครงสร้างข้อมูลแบบ tree จะมี root อยู่ด้านบนสุดและ leaves อยู่ทางด้านล่าง
- สังเกตว่าเราสามารถเริ่มจากจุดบนสุดของ tree จากนั้นจะไปตามเส้นทาง(path) ที่สร้างจาก วงกลมและลูกศร ไปด้านล่างได้

- ในแต่ละชั้นของ tree เราอาจจะถามคำถามและไปตามเส้นทางที่สอดคล้องกับคำตอบของคำถาม
- ตัวอย่างเช่น เราอาจจะถามว่า “สิ่งนี้เป็นพืชหรือสัตว์” ถ้าคำตอบเป็น “พืช” เราก็จะไปตามเส้นทางนั้น จากนั้นอาจจะถามต่อ เช่น “เป็นพืชใบเลี้ยงคู่หรือใบเลี้ยงเดี่ยว” ถ้าคำตอบเป็นไม่ใช่ เราอาจจะหยุดก็ได้
- ทั้งนี้เราอาจจะท่องไปตาม path จนกระทั่งถึงจุดล่างสุดของ tree ที่เราได้ชนิดของพืชก็ได้

- อีกคุณสมบัติหนึ่งของ tree คือ โหนดลูกๆ ของโหนดโหนดหนึ่งเป็นอิสระจากกัน นั่นคือ การเปลี่ยนแปลงของโหนดลูกไม่มีผลต่อโหนดลูกโหนดอื่น
- ตัวอย่างเช่น Genus Felis มีโหนดลูกเป็น Domestica และ Leo ส่วน Genus Musca มีโหนดลูกโหนดเดียวชื่อ Domestica แต่เป็นคนละโหนดและไม่ขึ้นกับโหนดลูกของ Felis นั่นคือหากเราเปลี่ยนโหนดลูกของ Musca จะไม่มีผลต่อลูกของ Felis

Kingdom

Phylum

Class

Order

Family

Genus

Species

Animalia

Chordate Arthropoda

Mammal Insect

Primate Carnivora Diptera

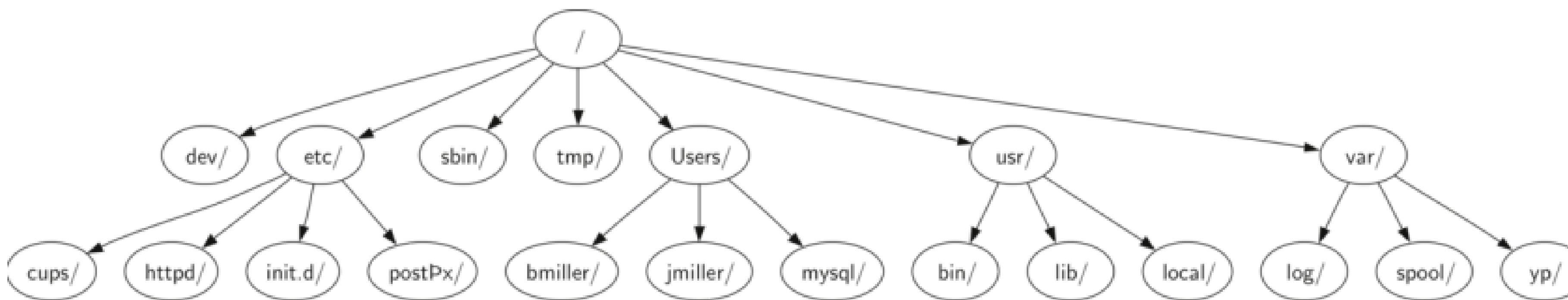
Hominidae Pongidae Felidae Muscidae

Homo Pan Felis Musca

Sapiens Troglodytes Domestica Leo Domestica

Human Chimpanzee House Cat Lion Housefly

- คุณสมบัติอีกอย่างคือโหนดใบ (leaf node) แตกต่างกัน เราสามารถระบุ path จาก root ไป leave ได้ทางเดียว ตัวอย่างจากรูปเช่น Animalia-> Chordate->Mammal->Carnivora->Felidae=>Felis->Domestica
- ตัวอย่าง tree ที่เราเจอบ่อยๆ อีกอย่างคือ file system

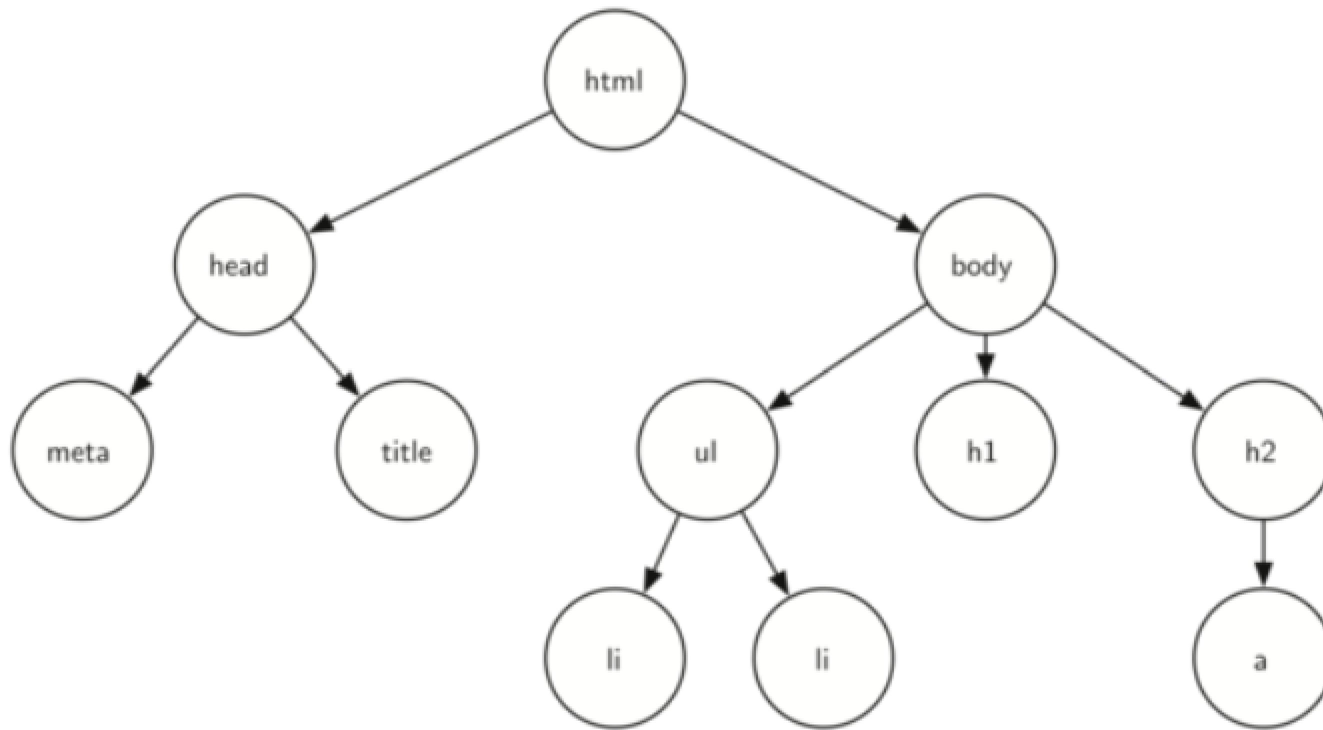


- File system นั้นมีโครงสร้างคล้ายกับต้นไม้ทางชีววิทยา เราสามารถเริ่มจาก root ไปยัง directory ใดๆ ก็ได้ path นั้นจะระบุโดย subdirectory
- คุณสมบัติที่สำคัญอีกอย่างของ tree นั้นสืบทอดมาจากลักษณะที่เป็นลำดับขั้น นั้นคือ เราสามารถย้ายทั้งส่วนของ tree (เรียกว่า subtree) ไปยังตำแหน่งใหม่ใน tree โดยที่ไม่ส่งผลต่อโครงสร้างด้านล่าง ตัวอย่างเช่น เราสามารถย้าย Folder ไปไว้ยังตำแหน่งใหม่ โดยที่ subfolder ก็ย้ายไปด้วย โครงสร้างเดิมไม่เปลี่ยนแปลง

- อีกตัวอย่างของ tree คือ web page โดย tree จะสอดคล้องกับ HTML tag ที่เราใช้ในการสร้าง page

```
<html>
<head><title> Hello</title>
</head>
<body><h1>Y First Web Page</h1>
</body>
</html>
```

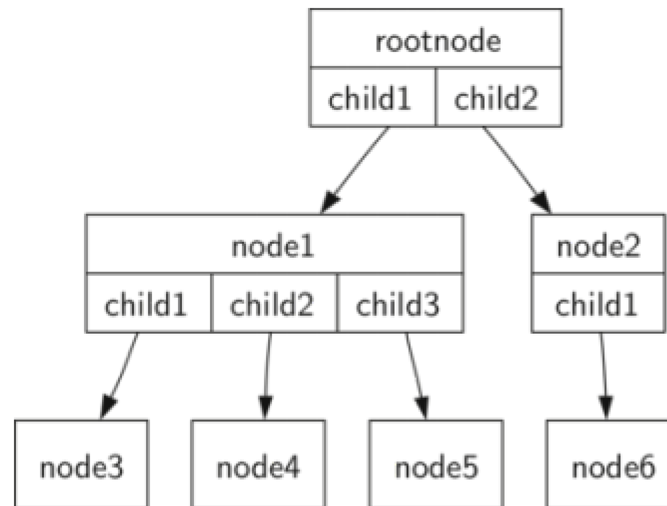
- ใน source code HTML เราสามารถใช้ tree ในการแสดงลำดับขั้นของ source ได้ แต่ละชั้นของ tree จะสอดคล้องกับ level ที่ซ้อนกันภายใน HTML tags โดย tag แรกของ source คือ <html> และ tag สุดท้ายคือ </html> tag ที่เหลือจะอยู่ภายในคู่
- ถ้าเราตรวจสอบ เราจะเห็นคุณสมบัติการซ้อนกันนี้ว่าเป็นจริงในทุก ๆ ชั้น


html
head
body
meta
title
ul
h1
h2
li
li
a

คำศัพท์และนิยามต่าง ๆ

- Node โหนด เป็นส่วนพื้นฐานของ tree เราสามารถตั้งชื่อให้กับโหนดได้ ซึ่งเราจะเรียกว่า key โหนดอาจจะเก็บข้อมูลเพิ่มได้ เราเรียกข้อมูลที่เพิ่มขึ้นมานี้ว่า payload
- Edge เส้นเชื่อมเป็นส่วนพื้นฐานอีกอันของ tree เส้นเชื่อมจะเชื่อมระหว่างโหนด 2 โหนด เพื่อแสดงความสัมพันธ์ระหว่างพวกมัน แต่ละโหนดโหนด(ยกเว้น root) จะมีเส้นเชื่อขาเข้าได้เพียงหนึ่งโหนด แต่จะมีเส้นเชื่อมขาออกได้หลายโหนด

- Root โหนดราก เป็นโหนดเพียงโหนดเดียวใน tree ที่ไม่มีเส้นเชื่อมขาเข้า
- Path เส้นทาง เป็นลำดับของโหนดใน tree ที่เชื่อมต่อกันด้วยเส้นเชื่อม ตัวอย่างเช่น Mammal->Carnivora->Fellidae->Felis
- Children โหนดลูก เป็นเซตของโหนด c ที่มีเส้นเชื่อมขาเข้าจากโหนดเดียวกันที่เราเรียกว่าเป็นโหนดลูกของโหนดนั้น
- Parent โหนดจะเป็นโหนดพ่อของทุกโหนดที่มันเชื่อมด้วยเส้นเชื่อมขาออก



- Sibling โหนดพี่น้อง โหนดใน tree ที่เป็นโหนดลูกของ parent ตัวเดียวกัน เช่น node 1 และ node 2 เป็น sibling
- Subtree เป็นเซตของโหนดและเส้นเชื่อมที่ประกอบด้วย parent และ ลูกหลานของ parent ตัวนั้น

- Leaf node เป็นโหนดที่ไม่มี children

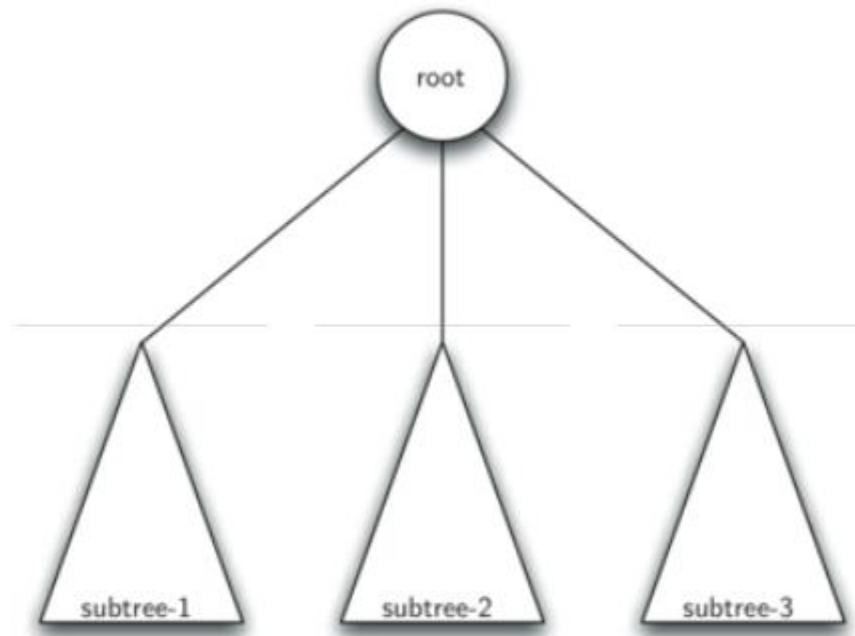
- Level level ของโหนด n คือ จำนวนของเส้นเชื่อมบน path จาก root มายังโหนด n จากนิยามนี้ทำให้ level ของ root เป็น 0

- Height ความสูงของ tree จะเท่ากับ level ที่มากที่สุดของโหนดใดๆ ใน tree

นิยาม

- นิยามอย่างเป็นทางการของ tree ในที่นี้จะมี 2 นิยาม นิยามแรกจะเกี่ยวกับโหนดและเส้นเชื่อมอีกนิยามจะเป็นแบบ recursive
- นิยาม tree ประกอบด้วยเซตของโหนดและเซตของเส้นเชื่อมที่เชื่อมระหว่างคู่ของโหนด โดยมีคุณสมบัติดังนี้
  - มีหนึ่งโหนดใน tree ที่เป็น root node
  - Path จาก root ไปยังแต่ละโหนดมีเพียง path เดียว
  - ถ้าแต่ละโหนดใน tree มีลูกไม่เกิด 2 เราจะเรียกว่า binary tree

- นิยามแบบที่สอง tree จะว่างหรือประกอบไปด้วย root และ 0 หรือมากกว่า subtree ซึ่งแต่ละอันเป็น tree ด้วย root ของแต่ละ subtree ถูกเชื่อมกับ root ของ parent tree ด้วยเส้นเชื่อม
- รูปด้านล่างเป็น tree ที่มีอย่างน้อย 4 โหนด



การสร้าง tree

- ในที่นี้เราจะสร้าง binary tree ซึ่งจะมีฟังก์ชันดังนี้
- BinaryTree() เป็นการสร้าง binary tree ใหม่
- get_left_child() คืนค่า binary tree ที่สอดคล้องกับลูกทางซ้ายของโหนดปัจจุบัน (current node)
- get_right_child() คืนค่า binary tree ที่สอดคล้องกับลูกทางขวาของโหนดปัจจุบัน (current node)
- set_root_val(val) เก็บค่าวัตถุไว้กับโหนดปัจจุบัน

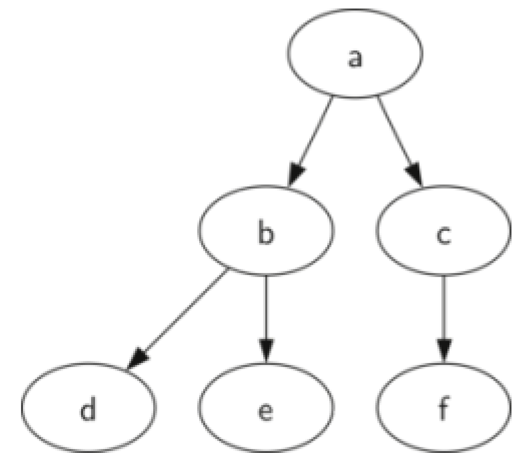
- get_root_val() คืนค่าของวัตถุที่เก็บที่โหนดปัจจุบัน
- insert_left() สร้าง binary tree ใหม่และเชื่อมันเป็นลูกทางซ้ายของโหนดปัจจุบัน
- insert_left() สร้าง binary tree ใหม่และเชื่อมันเป็นลูกทางขวาของโหนดปัจจุบัน

ในการตัดสินใจสร้าง tree เราจะเลือกวิธีการเก็บข้อมูลภายใน Python มี 2 วิธีที่น่าสนใจ เราจะทำทั้งสองอย่างก่อนที่จะเลือกใช้ ทางแรกเราใช้ list of lists ทางที่สองเราใช้ nodes and references

สร้าง tree ด้วย List of Lists

- เราจะเริ่มต้นด้วย list และเขียนฟังก์ชันบนนั้นกัน
- แม้ว่าการเขียน operation ต่างๆ บน list จะต่างจาก abstract data type แบบอื่นที่เราได้สร้างมาแล้ว มันน่าสนใจเนื่องจากมันทำให้เราเขียนแบบ recursive ได้ง่าย
- List of lists tree เราจะเก็บค่าของ root node เป็นสมาชิกตัวแรกของ list สมาชิกตัวที่สองของ list จะเป็น list ของ left subtree ตัวที่สามของ list จะเป็น list อีกตัวที่เป็น right subtree

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



- สังเกตว่าเราเข้าถึง subtree ของ list โดยใช้ index ของ list แบบธรรมดา
- root ของ tree คือ my_tree[0]
- left subtree ของ root คือ my_tree[1]
- right subtree ของ root คือ my_tree[2]
- code ต่อไปแสดงการสร้าง simple tree โดยใช้ list

- my_tree = ['a',['b',['d',[],[]],['e',[],[]]],['c',['f',[],[]],[]]]
- print(my_tree)
- print('left subtree =',my_tree[1])
- print('root =',my_tree[0])
- print('right subtree =',my_tree[2])

ต่อไปจะลองเขียนฟังก์ชัน

```
def binary_tree( r):
    return [r, [], [] ]
```

ฟังก์ชัน binary_tree เป็นการสร้าง list ที่มี root และมี sublist ว่างสองอัน

ในการเพิ่ม left subtree ให้กับ root เราจะต้อง insert list ใหม่ไปในตำแหน่งที่สองของ root list

เราจะต้องระวัง ถ้าในตำแหน่งที่สองมีของเก็บอยู่ก่อนแล้ว เราจะผลักมันลงไปเป็นลูกทางซ้ายของของที่เราจะใส่ใหม่

```
def insert_left(root, new_branch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1, [new_branch, t, []])
    else:
        root.insert(1,[new_branch, [], []])
    return root
```

- สังเกตว่าในการ insert ลูกทางซ้าย เราเริ่มจากเอา list ของลูกทางซ้ายออกมาก่อน(อาจจะไม่มีตัว)
- หลังจากนั้นเราจะเพิ่มลูกทางซ้ายใหม่ โดยให้ ลูกทางซ้ายอันเก่าเป็นลูกทางซ้ายของของที่เพิ่ม นี่ทำให้เราเชื่อมต่อโหนดใหม่เข้ากับ tree ได้ code ของ insert_right ก็คล้ายกัน

```python
def insert_right(root, new_branch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2, [new_branch, [], t])
    else:
        root.insert(2,[new_branch, [], []])
    return root
```

ต่อไปเป็นฟังก์ชัน get set ของ root และ subtree

```
def get_root_val(root):
    return root[0]
def set_root_val(root,new_val):
    root[0] = new_val
def get_left_child(root):
    return root[1]
def get_right_child(root):
    return root[2]
```

จงวาดรูป tree ที่ได้

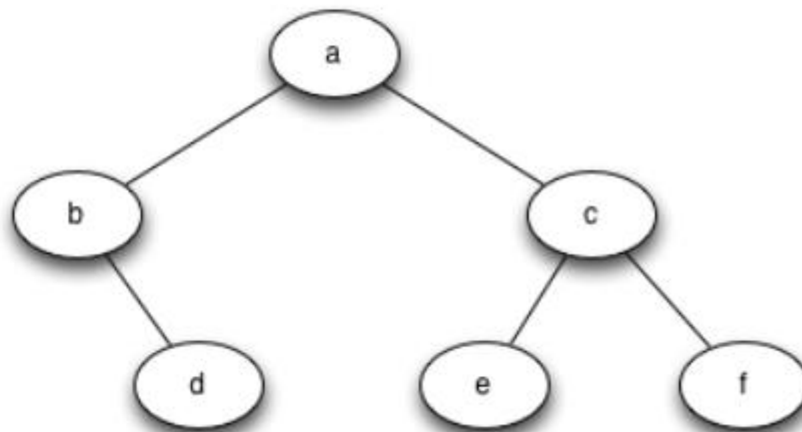
- ต่อไปนำ code มารวมกันแล้วสั่งงานด้วยคำสั่ง

```
r = binary_tree(3)
insert_left(r, 4)
insert_left(r, 5)
insert_right(r, 6)
insert_right(r, 7)
l = get_left_child(r)
print(l)
set_root_val(l, 9)
print(r)
insert_left(l, 11)
print(r)
```

จงวาดรูป tree ที่ได้

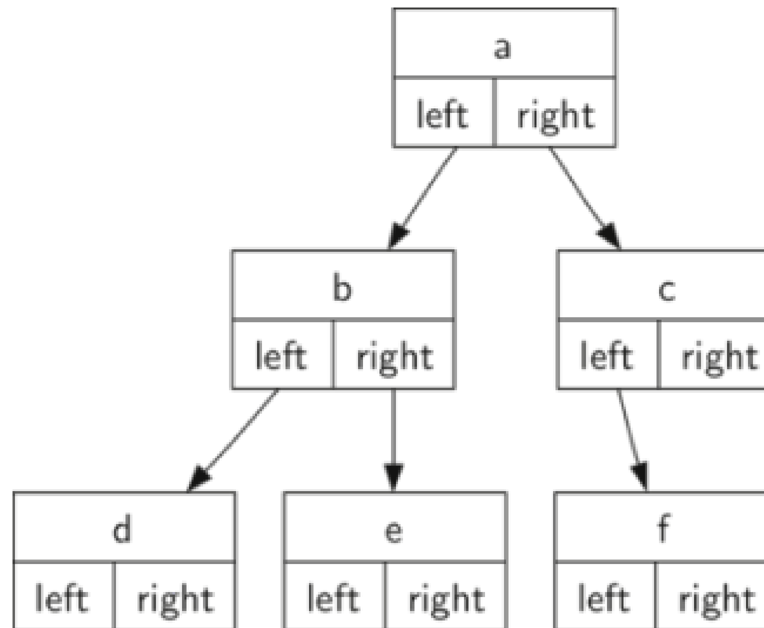
```
x = binary_tree('a')

insert_left(x,'b')

insert_right(x,'c')

insert_right(get_right_child(x), 'd')

insert_left(get_right_child(get_right_child(x)), 'e')

print(x)
```

- จงเขียนคำสั่งให้ได้ tree ดังรูป



Node and references

- วิธีการที่สองจะแสดง tree โดย nodes และ references ในกรณีนี้เราจะนิยาม class ที่มี attributes สำหรับค่า root และ left, right subtree
- การใช้ node และ reference เราจะคิดว่า tree มีโครงสร้างแบบนี้

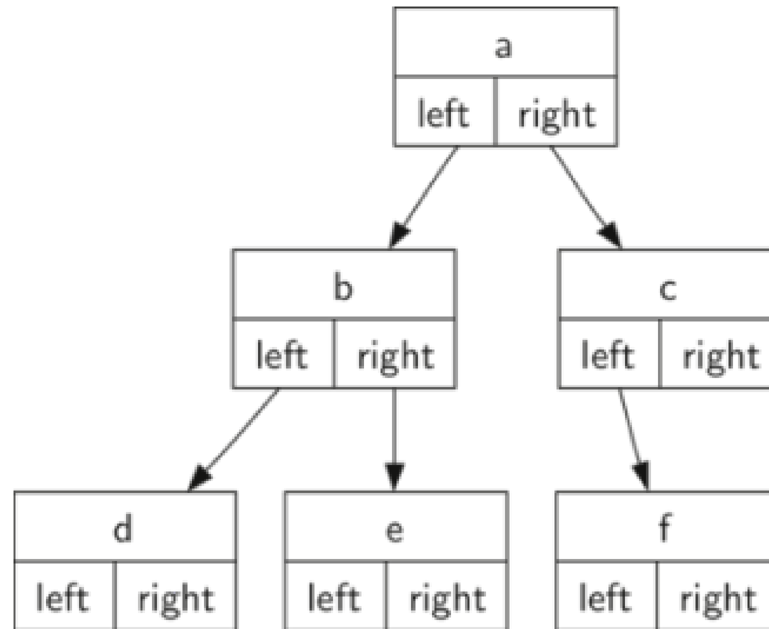


- เราจะเริ่มด้วยการนิยาม class สำหรับ node และ reference ดังหน้าต่อไป สิ่งสำคัญคือคือ attributes left และ right ซึ่งกลายเป็น reference ของตำแหน่งต่อไปใน BinaryTree class
- ตัวอย่างเช่น เมื่อเราเริ่มใส่ left node ใหม่ใน tree เราจะสร้าง instance ใหม่ของ BinaryTree และ เปลี่ยน self.left_child ที่ root มา reference node ใหม่นี้

```
class BinaryTree:
    def __init__(self, root):
        self.key = root
        self.left_child = None
        self.right_child = None
```

สังเกตได้ว่า ใน constructor นั้นต้องการของมาเก็บไว้ที่ root เช่นเดียวกับการที่เอาของมาเก็บใน list

จากตัวอย่างก่อนหน้าเราสามารถเก็บชื่อโหนดได้



- รูปนี้มี 6 instance ของ BinaryTree class

- ต่อไปเราจะมาดู function ที่เราต้องเขียนเพิ่ม ในการเพิ่ม left child ให้กับ tree เราจะสร้าง binary tree object และกำหนด left attribute ของ root มาอ้างอิง object ใหม่นี้

```
def insert_left(self, new_node):
    if self.left_child == None:
        self.left_child = BinaryTree(new_node)
    else:
        t = BinaryTree(new_node)
        t.left_child = self.left_child
        self.left_child = t
```

- ในการเพิ่มโหนดจะมีสองกรณีที่ต้องพิจารณา
- กรณีแรกคือ node ที่เราจะเพิ่มไม่มีลูกทางซ้าย เราก็เพิ่มโหนดใหม่ไปเป็นลูกทางซ้ายได้เลย
- กรณีที่สองคือ node ที่เราจะเพิ่มมีลูกทางซ้าย เราจะเพิ่มโหนดใหม่แล้วดัน node ลูกทางซ้ายตัวเก่าลงไป 1 level
- code ของการเพิ่ม node ทางขวามือก็เช่นกัน

```
def insert_right(self,new_node):
    if self.right_child == None:
        self.right_child = BinaryTree(new_node)
    else:
        t = BinaryTree(new_node)
        t.right_child = self.right_child
        self.right_child = t
```

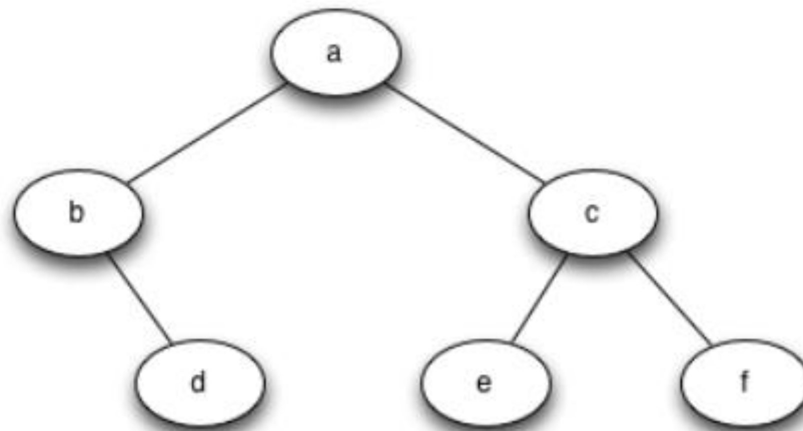
ต่อไปเป็นพวก method เข้าถึงค่าต่างๆ

```
def get_right_child(self):
    return self.right_child
def get_left_child(self):
    return self.left_child
def set_root_val(self,obj):
    self.key = obj
def get_root_val(self):
    return self.key
```

- เมื่อได้ครบทุกฟังก์ชันแล้ว เราจะมาลองตรวจสอบกัน
- เริ่มต้นจะสร้าง root node a และเพิ่ม node b และ c เป็นลูก ซึ่งจะเกี่ยวกับการเข้าถึงค่า key left right
- สังเกตว่า ลูกทางซ้ายและลูกทางขวาจะเป็นคนละ instance ของ BinaryTree กัน

```
r = BinaryTree('a')

print(r.get_root_val())

print(r.get_left_child())

r.insert_left('b')

print(r.get_left_child())

print(r.get_left_child().get_root_val())

r.insert_right('c')

print(r.get_right_child())

print(r.get_right_child().get_root_val())

r.get_right_child().set_root_val('hello')

print(r.get_right_child().get_root_val())
```

- จงเขียน function ให้ได้ tree ดังรูป



Priority Queue with Binary Heaps

- ในบทก่อนเราได้เรียนโครงสร้างข้อมูลแบบ first-in first-out นั่นคือ queue ในหัวข้อนี้เราจะสนใจ queue ชนิดหนึ่งที่เรียกว่า priority queue
- Priority queue ทำงานคล้าย queue ตรงที่จะ dequeue ข้อมูลส่วนหน้าสุดออก อย่างไรก็ตาม ใน priority queue การเรียงลำดับของข้อมูลภายในนั้นเรียงตามความสำคัญ ความสำคัญสูงสุดจะอยู่ด้านหน้า ส่วนความสำคัญต่ำจะอยู่ท้าย

- เมื่อเรา enqueue ข้อมูลลงใน priority queue ข้อมูลใหม่นี้อาจจะถูกย้ายตำแหน่งไปยังด้านหน้าสุด
- เราสามารถสร้าง priority queue ได้อย่างง่ายๆ โดยใช้การ sort ข้อมูลแล้วแทรกลง list ซึ่งทำงานช้า
- เราจะสร้าง priority queue โดยใช้โครงสร้างข้อมูลที่เรียกว่า binary heap ซึ่งทำงานได้เร็ว

- Binary heap มีหน้าตาคล้าย tree แต่เราจะสร้างมันโดยใช้ list เพียงอันเดียว

Binary heap มี 2 ชนิด

Min heap มีลักษณะคือ key ที่มีค่าน้อยสุดจะอยู่ด้านหน้า

Max heap มีลักษณะคือ key ที่มีค่ามากสุดจะอยู่ด้านหน้า

ในที่นี้เราจะสร้าง min heap

Binary heap operations

- การดำเนินการของ binary heap

BinaryHeap() สร้าง binary heap อันใหม่

insert(k) เพิ่มข้อมูลใหม่ลงใน heap

find_min() คืนค่าข้อมูลที่มี key ที่น้อยที่สุด ข้อมูลยังอยู่ใน heap

del_min() คืนค่าข้อมูลที่มี key ที่น้อยที่สุด ลบข้อมูลนั้นด้วย

is_empty() คืนค่า True ถ้า heap ว่าง กรณีอื่น False

size() คืนค่าจำนวนข้อมูลใน heap

build_heap(list) สร้าง heap ใหม่จาก list ของข้อมูล

```
import BinHeap

bh = BinHeap()

bh.insert(5)

bh.insert(7)

bh.insert(3)

bh.insert(11)

>>> print(bh.del_min())
3
>>> print(bh.del_min())
5
>>> print(bh.del_min())
7
>>> print(bh.del_min())
11
```

จะได้ค่า min เสมอ

จะได้ค่า min เสมอ

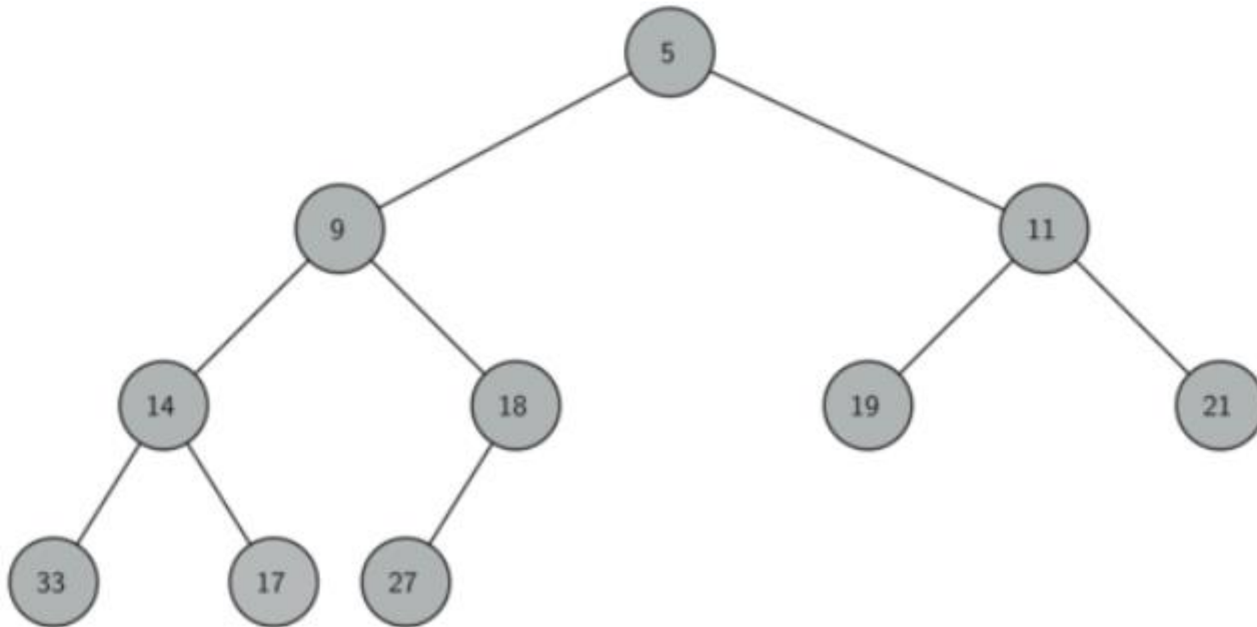
จะได้ค่า min เสมอ

จะได้ค่า min เสมอ

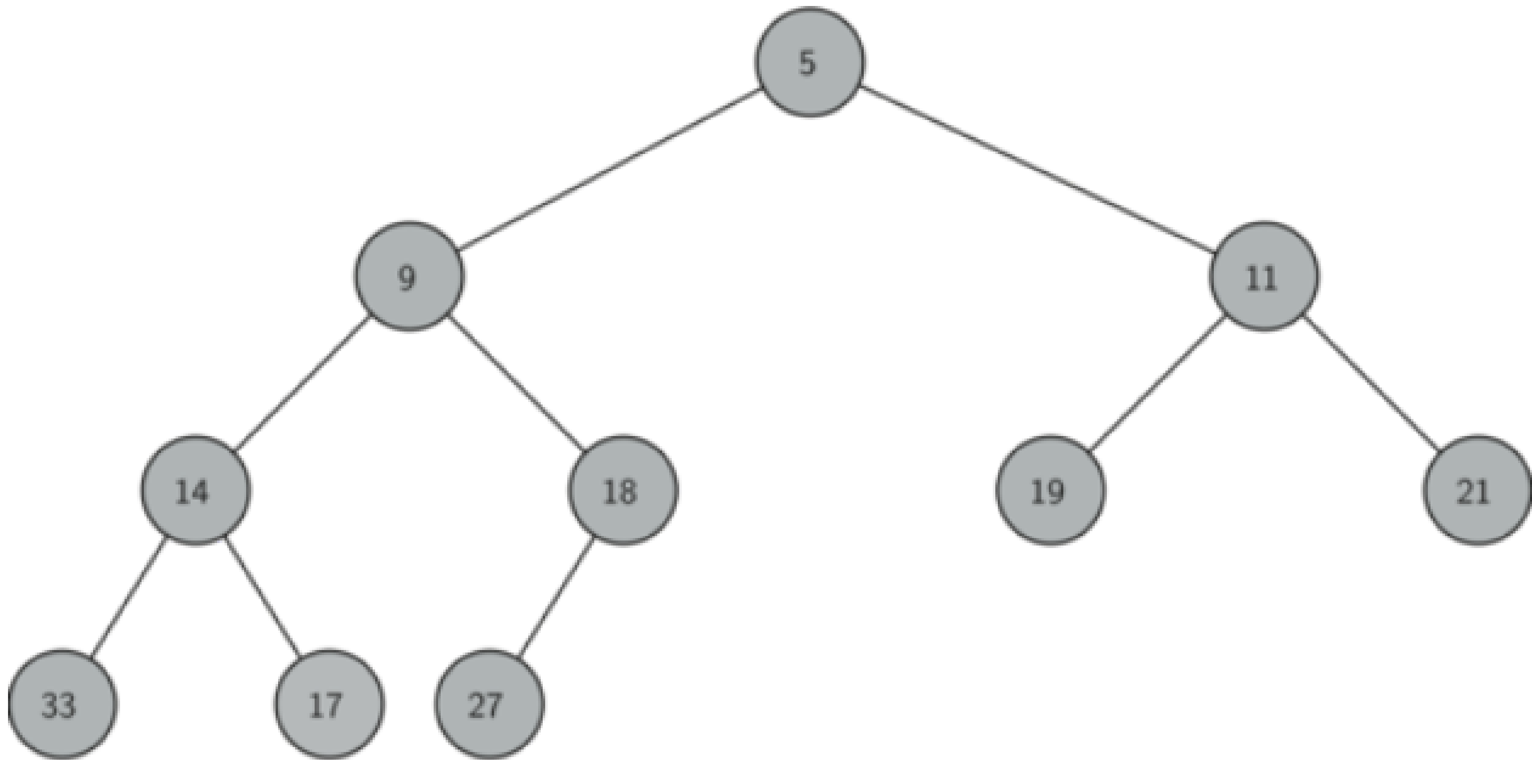
Binary heap implementation

- ในการสร้าง binary heap เราต้องการให้การดำเนินการต่างๆ รวดเร็ว ดังนั้นเราจะต้องทำให้ข้อมูลมี level ที่ต่ำ นั่นคือ การทำให้ tree balance
- Balanced binary tree นั้นโหนดจะมีจำนวนโหนดใน subtree ทางซ้ายและ subtree ทางขวาแทบจะเท่ากัน
- heap ที่เราจะสร้างนั้นเราจะทำให้ tree balance โดยการสร้าง complete binary tree

- Complete binary tree คือ tree ที่ในแต่ละ level มีจำนวนโหนดเต็มยกเว้น level ล่างสุด โดยเราจะเติมโหนดจากซ้ายไปขวา



- คุณสมบัติที่สำคัญอีกอย่างจอง complete tree คือ เราสามารถใช้ list แทนได้ เราไม่จำเป็นต้องสร้าง node และ reference หรือ list of lists
- เนื่องจากว่า tree นั้น complete โหนดลูกทางซ้ายของ parent (ตำแหน่ง p) คือโหนดที่อยู่ที่ตำแหน่ง 2p ใน list ส่วนโหนดลูกทางขวาคือโหนดที่อยู่ที่ตำแหน่ง 2p+1 ใน list
- การหาว่า parent คือโหนดใดเราก็ใช้การหาร สมมติว่าโหนดในตำแหน่งที่ n โหนดพ่อของโหนดนี้คือโหนดตำแหน่ง n/2


5
9
11
14
18
19
21
33
17
27
0 5 9 11 14 18 19 21 33 17 27
0 1 2 3 4 5 6 7 8 9 10 11

Heap order property

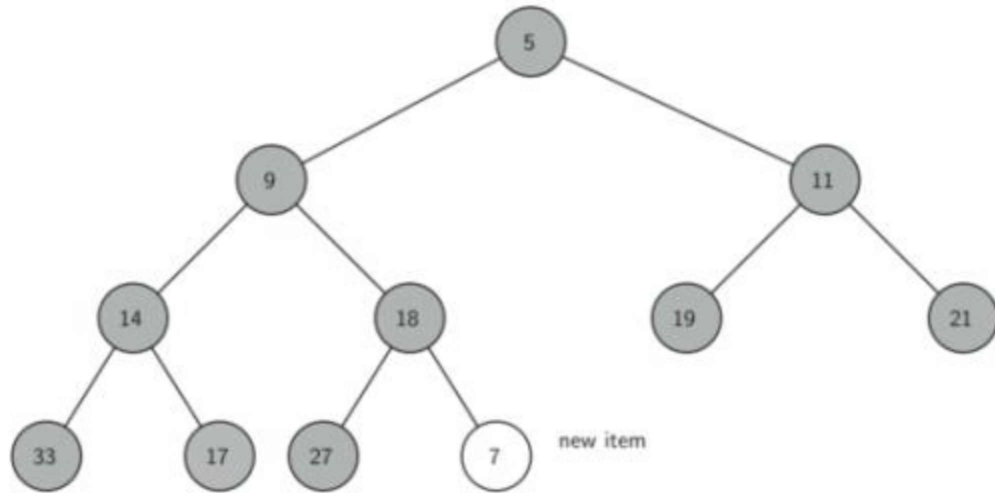
- วิธีที่เราใช้ในการเก็บข้อมูลลง heap นั้นขึ้นกับการรักษาลำดับใน heap
- heap order property เป็นดังนี้
- ใน heap แต่ละโหนด x ที่มี parent p key ใน p จะน้อยกว่าหรือเท่ากับ key ใน x รูปก่อนหน้าเป็น complete binary tree ที่มี heap order property

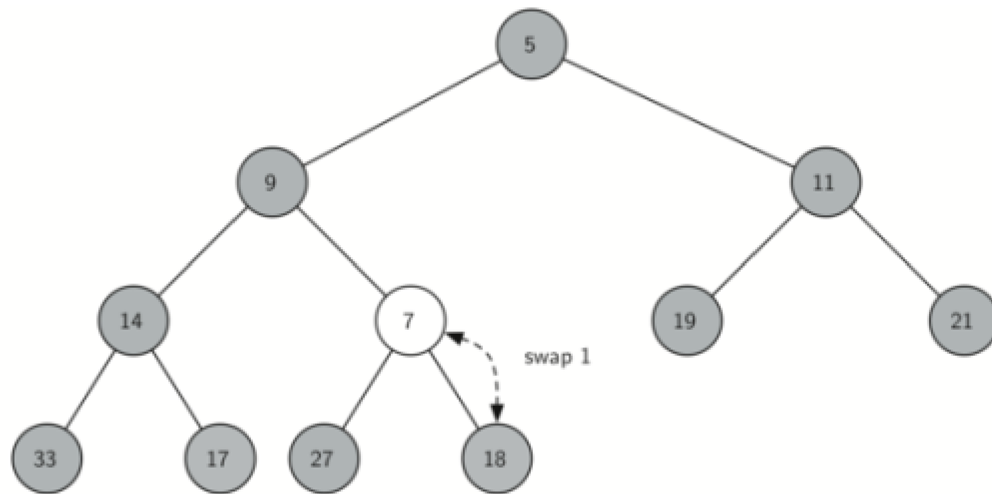
Heap operations

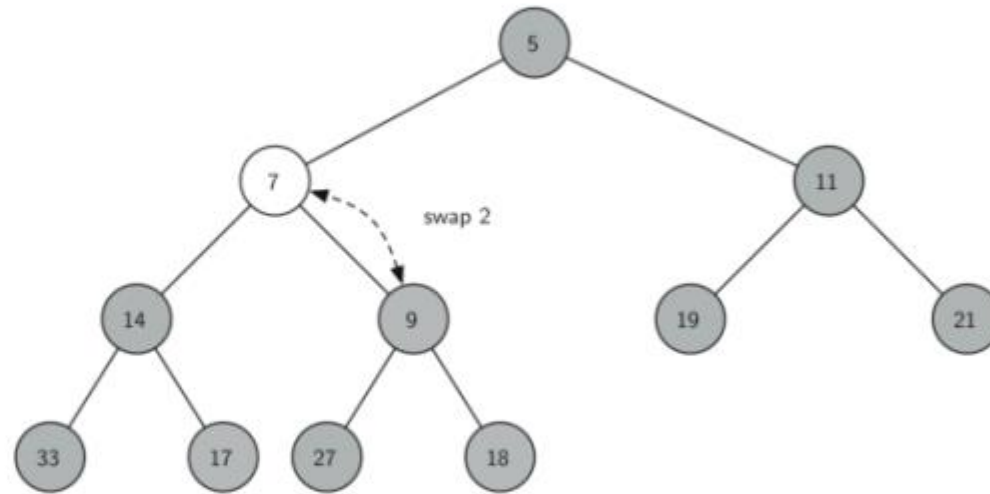
- เริ่มต้นด้วย constructor เนื่องจาก binary heap สามารถใช้ list เพียงอันเดียวในการสร้างได้ เริ่มต้นเราต้องสร้าง list และสร้าง attribute current_size ในการเก็บว่าตอนนี้มีขนาดเท่าไร
- ในที่นี้ binary heap เปล่าเราจะให้มีสมาชิก 0 เก็บไว้เป็นสมาชิกตัวแรกของ heap_list ทั้งนี้ 0 ไม่ถูกใช้งาน แต่เพื่อให้เราอ้างอิง index ได้ง่าย เราจะได้เริ่มตัวแรกที่ index เป็น 1

```
class BinHeap:
    def __init__(self):
        self.heap_list = [0]
        self.current_size = 0
```

- ต่อไปเป็น method insert วิธีที่ง่ายและมีประสิทธิภาพที่สุดคือเพิ่มเข้าไปท้าย list
- การเพิ่มไปท้าย list นี้ยังทำให้คุณสมบัติ complete tree เป็นจริงอยู่ แต่ปัญหาคือการนำเอาข้อมูลใหม่ไปต่อท้ายมันจะขัดกับคุณสมบัติของ heap
- แต่เราจะใช้วิธีเปรียบเทียบข้อมูลใหม่กับ parent ของมัน ถ้าข้อมูลที่เพิ่มเข้าไปใหม่น้อยกว่า parent เราก็จะสลับที่กัน จากนั้นตรวจสอบกับ parent ตัวต่อไป จนทำไม่ได้หรือถึง root


5
9
11
14
18
19
21
33
17
27
7
new item


5
9
11
14
7
19
21
swap 1
33
17
27
18



- สังเกตว่าการปรับข้อมูลขึ้นไป ทำให้เรารักษาคุณสมบัติของ heap ระหว่างตัวใหม่กับตัว parent ของมันได้ แล้วยังรักษาคุณสมบัตินี้กับตัวที่อยู่ด้านล่างอีกด้วย ต่อไปเป็น code ของการปรับขึ้น (perculate up)

```python
def perc_up(self, i):
    while i // 2 > 0:
        if self.heap_list[i] < self.heap_list[i//2]:
            tmp = self.heap_list[i//2]
            self.heap_list[i//2] = self.heap_list[i]
            self.heap_list[i] = tmp
        i = i // 2
```

- ต่อไปมาดู code ของการ insert เมื่อเราเพิ่มของเข้าไปต่อท้าย tree จากนั้นเราจะ perculate up เพื่อปรับตำแหน่งให้ถูก

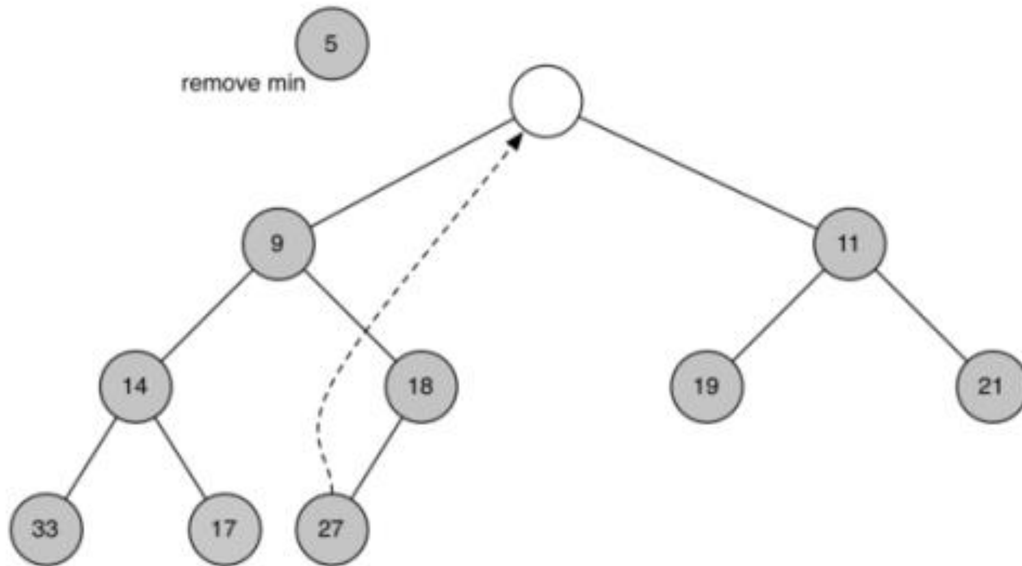
```
def insert (self, k):
    self.heap_list.append(k)
    self.current_size = self.current_size+1
    self.perc_up(self.current_size)
```

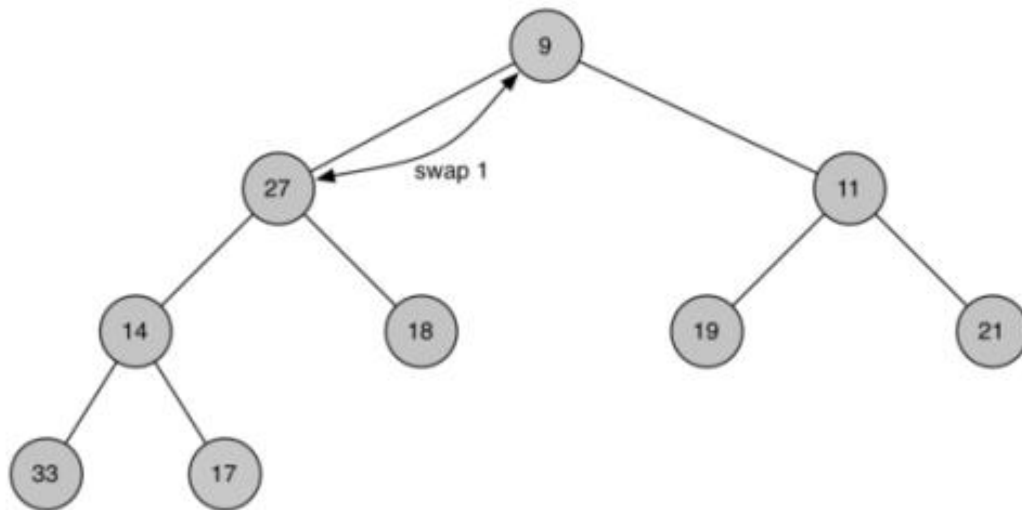
- ต่อไปเราจะสร้าง function del_min เนื่องจาก คุณสมบัติของ heap นั้น root มีค่าน้อยที่สุด เพื่อให้หาค่าน้อยที่สุดได้อย่างรวดเร็ว ส่วนที่ยากของ del_min คือ พอเอาตัวแรกออกแล้วต้องปรับโครงสร้างของ heap อย่างไรเพื่อให้ยังคงคุณสมบัติของ heap
- เริ่มต้นเราจะแทนที่ root ช่องที่ว่างด้วยข้อมูลตัวสุดท้ายใน list การย้ายตัวสุดท้ายมาทำให้ยังคงรักษาโครงสร้างของ heap ไว้ได้ แต่มันทำลายคุณสมบัติของ heap ดังนั้นเราจะทำให้รักษาคุณสมบัติของ heap โดยปรับตัว root ใหม่ลงไปให้อยู่ถูกที่

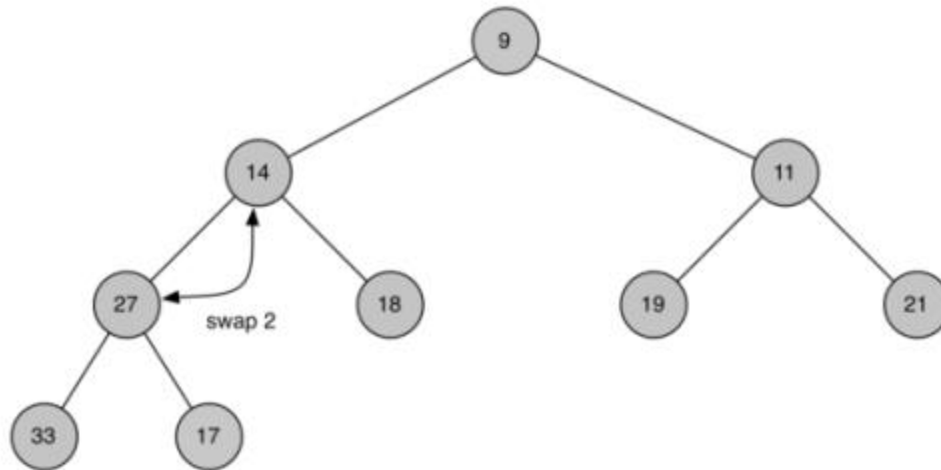
- ในการปรับเพื่อให้รักษาคุณสมบัติของ heap เราจะสลับ root กับลูกของมันที่น้อยกว่า จากนั้นอาจจะต้องสลับอีกพอไปอยู่ตำแหน่งใหม่แล้ว ทำจนกระทั่งไม่มีลูกตัวไหนน้อยกว่ามัน เราจะสร้างฟังก์ชันที่เรียกว่า perc_down โดยมีฟังก์ชันสำหรับหาลูกตัวที่น้อยสุดคือ min_child

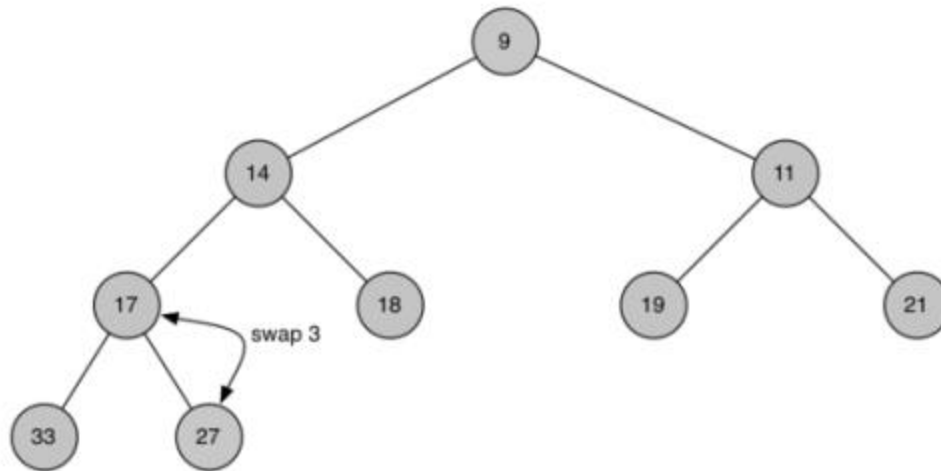
```
def min_child(self, i):
    if i*2+1  self.current_size:
        return i+2
    else:
        if self.heap_list[i*2] < self.heap_list[i*2+1]:
            return i*2
        else:
            return i*2+1
```

```
def perc_down(self, i):
    while (i*2) <=self.current_size:
        mc = self.min_child(i)
        if self.heap_list[i] > self.heap_list[mc]:
            tmp = self.heap_list[i]
            self.heap_list[i] = self.heap_list[mc]
            self.heap_list[mc] = tmp
        i = mc
```


5
remove min
9
11
14
18
19
21
33
17
27


9
27
swap 1
11
14
18
19
21
33
17


9
14
11
27
18
19
21
swap 2
33
17


9
14
11
17
18
19
21
swap 3
33
27

- ต่อไปเป็น code ของ del_min มีการเรียกใช้ perc_down

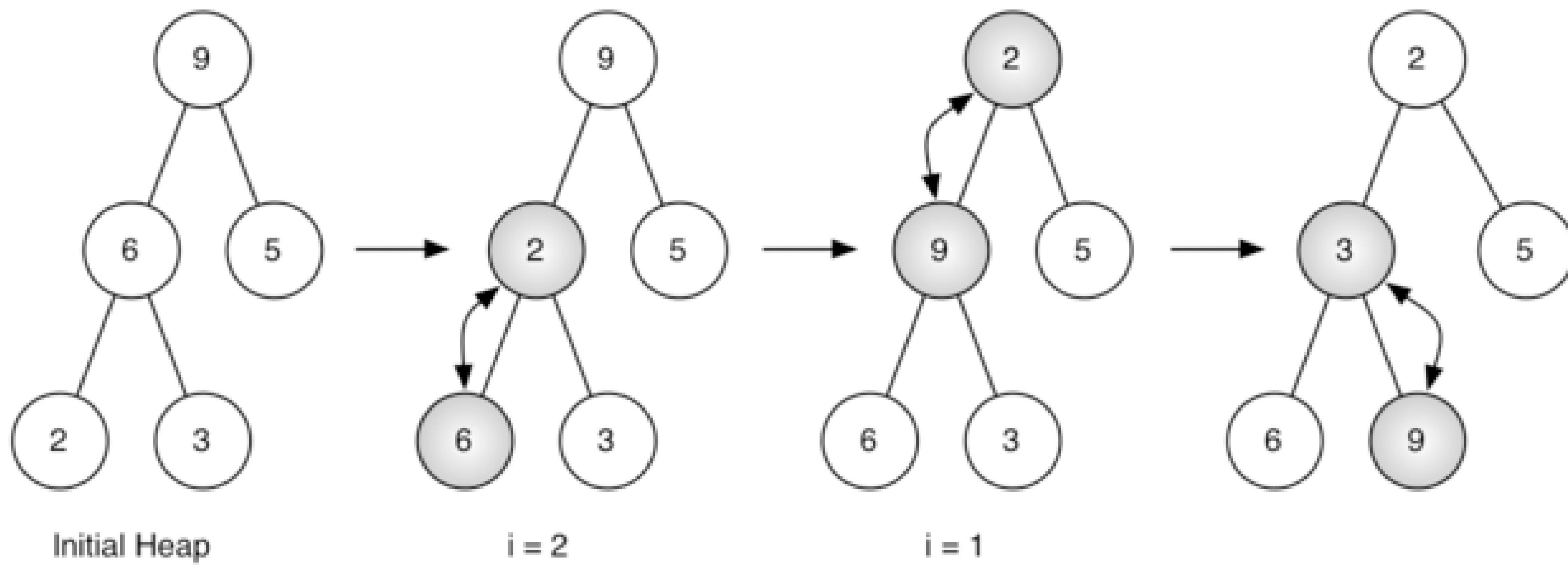
```
def del_min(self):
    ret_val = self.heap_list[1]
    self.heap_list[1] = self.heap_list[self.current_size]
    self.current_size = self.current_size – 1
    self.heap_list.pop()
    self.perc_down(1)
    return ret_val
```

- หากเรามี list แล้วเราต้องการสร้าง heap เราจะทำอย่างไร

เราอาจจะเริ่มต้นด้วย insert key ทีละตัวจากนั้นใช้ เมื่อ list มีตัวเดียวมันก็ sort แล้ว พอ insert ตัวใหม่มาก็ใช้ binary search หาตำแหน่งที่เหมาสมสำหรับตัวใหม่ ทำไปเรื่อยๆ

หรือ เราจะใช้ฟังก์ชันที่เราเขียนไปสร้างใหม่

```
def build_heap(self, a_list):
    i = len(a_list)//2
    self.current_size = len(a_list)
    self.heap_list = [0] + a_list[:]
    while(i>0):
        self.perc_down(i)
        i = i-1
```

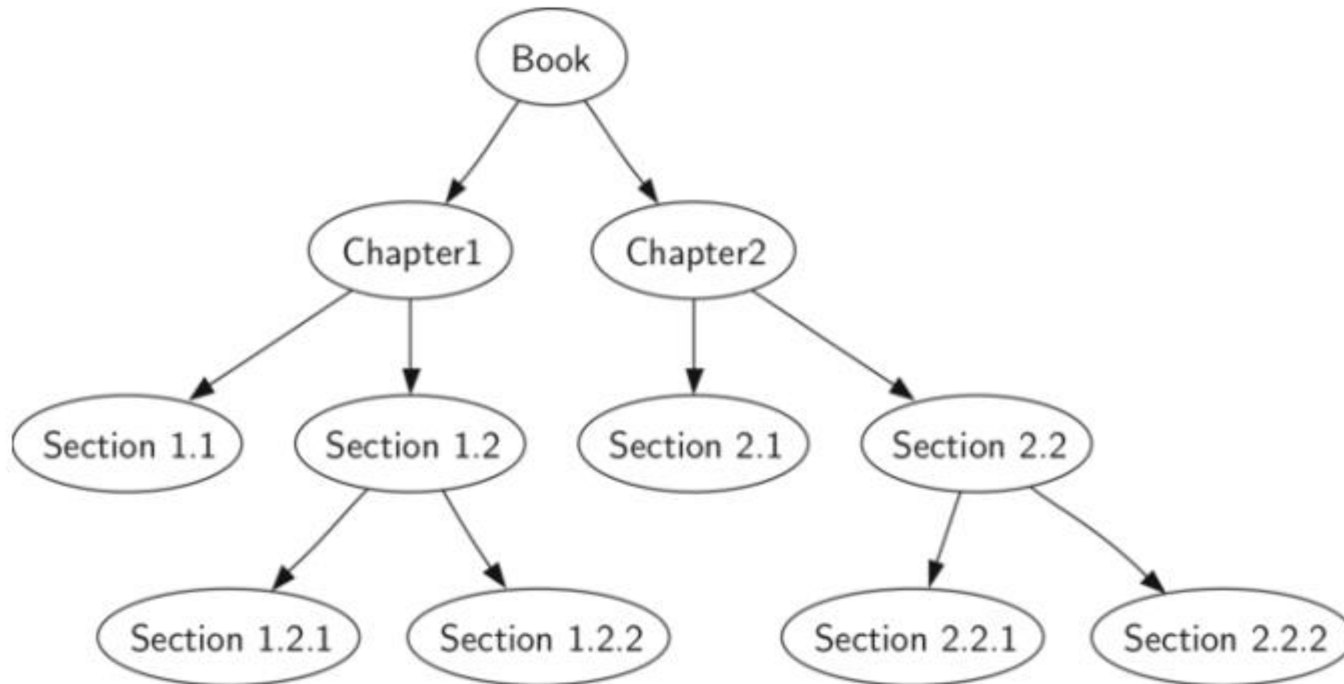

9
6
5
2
3
Initial Heap
9
2
5
6
3
i = 2
2
9
5
6
3
i = 1
2
3
5
6
9

- จากรูป เริ่มต้น list เป็น [9,6,5,2,3] แล้ว build_heap จะย้ายตำแหน่งให้เหมาะสม แม้ว่าจะเริ่มต้นที่ตรงกลาง tree และทำงานย้อนไปทาง root ฟังก์ชัน perc_down ทำให้แน่ใจว่าตัวมากสุดจะถูกดันลง
- เนื่องจากว่า heap เป็น complete binary tree โหนดใดๆ ที่ index เกินครึ่งจะมีลูกเป็นโหนดใบ หรือไม่มีลูกเลย สังเกตว่าเมื่อ i=1 เราจะ perc_down จาก root ลงมาซึ่งอาจใช้หลายครั้ง

Tree traversal

- ต่อไปจะมาดูการดำเนินการบน tree ที่สำคัญอย่างหนึ่งคือการตรวจสอบ tree เราจะตรวจสอบ tree ไปยังทุกโหนดแบบมีรูปแบบ
- รูปแบบในการตรวจสอบมีหลักๆ 3 แบบขึ้นกับลำดับในการตรวจสอบ preorder, inorder และ postorder

- preorder ในการ traversal แบบ preorder เราจะตรวจสอบ root ก่อนจากนั้นจะไปตรวจสอบแบบ recursive กับ left subtree แล้วตามด้วย recursive กับ right subtree
- inorder ในการ traversal แบบ inorder เราจะตรวจสอบแบบ recursive กับ left subtree ก่อนจากนั้นจะไปตรวจสอบ root แล้วตามด้วย recursive กับ right subtree
- postorder ในการ traversal แบบ postorder เราจะตรวจสอบแบบ recursive กับ right subtree ก่อนจากนั้นจะไปตรวจสอบ root แล้วตามด้วย recursive กับ left subtree


Book
Chapter1
Chapter2
Section 1.1
Section 1.2
Section 2.1
Section 2.2
Section 1.2.1
Section 1.2.2
Section 2.2.1
Section 2.2.2

- ถ้าเราต้องการอ่านหนังสือจากหน้าไปหลัง preorder จะทำให้ได้ลำดับที่ถูกต้อง เริ่มต้นที่ book จากนั้นจะเรียก preorder กับลูกทางซ้ายแบบ recursive ซึ่งคือ Chapter 1 จากนั้นเรียก preorder กับลูกทางซ้ายอีก จะได้ Section 1.1 เนื่องจาก Section 1.1 ไม่มีลูกแล้ว ก็ไม่เรียก recursive ก็จะย้อนกลับมาที่ chapter 1 แล้วก็จะไปทางขวานั่นคือ section 1.2 เช่นเดิมก็จะไปตรวจสอบทางซ้ายก่อนจะได้ section 1.2.1 แล้วเป็น section 1.2.2 พอทำเสร็จ Section 1.2 ก็เสร็จแล้วย้อนกลับไป Chapter 1 แล้วค่อยไปทำ chapter 2

- หากเป็นฟังก์ชันภายนอก

```
def preorder(tree):
    if tree:
        print(tree.get_root_val())
        preorder(tree.get_left_child())
        preorder(tree.get_right_child())
```

- หากเขียนเพิ่มใน BinaryTree class

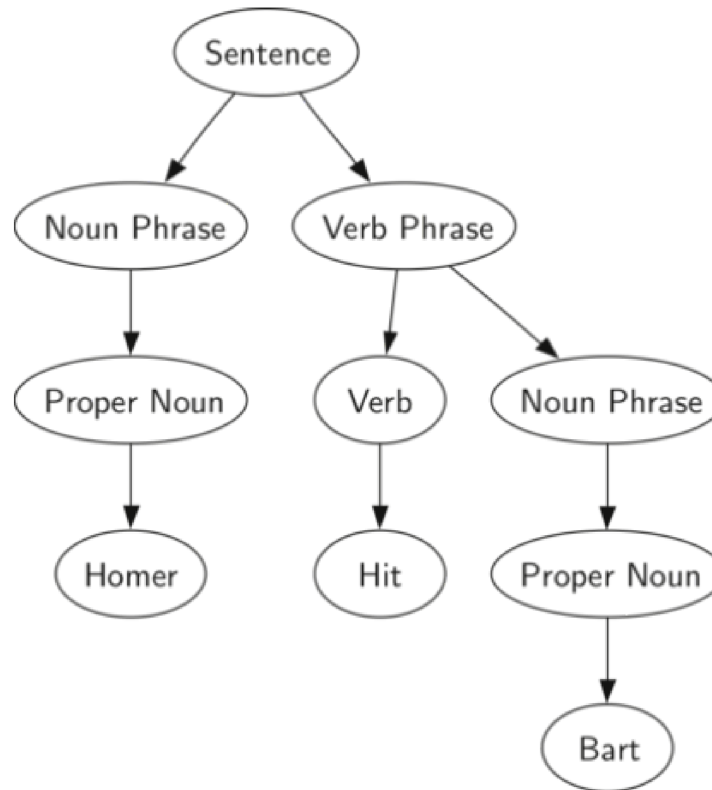
```
def preorder(self):
    print(self.key)
    if self.left_child:
        self.left_child.preorder()
    if self.right_child:
        self.right_child.preorder()
```

```
def postorder(tree):
    if tree != None:
        postorder(tree.get_left_child())
        postorder(tree.get_right_child())
        print(tree.get_root_val())
```

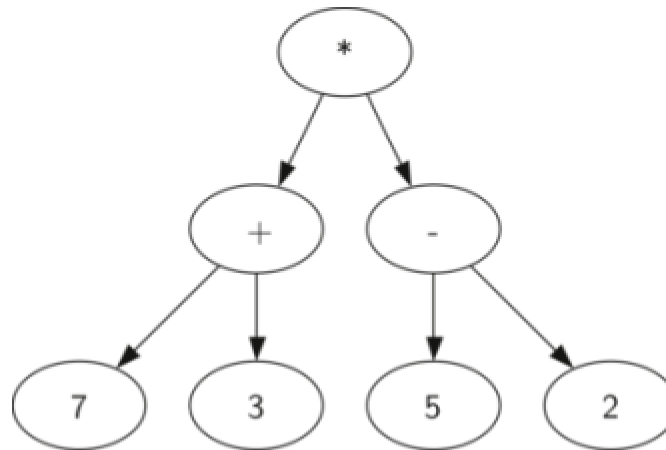
ให้ลองทำ inorder

Binary tree application

การนำเอา tree ไปใช้กับปัญหาจริง โดยตัวอย่างที่จะยกมาให้ดูคือ parse tree ซึ่งถูกใช้งานกับประโยคต่าง ๆ และในทางคณิตศาสตร์

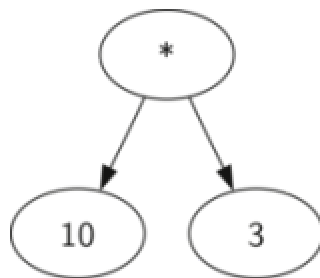


- อีกตัวอย่างเป็นการแสดง mathematical expression เช่น ((7+3)*(5-2))



- เรารู้อะไรกับ expression นี้บ้าง
- เรารู้ว่าการคูณสำคัญกว่าการบวกและการลบ แต่เนื่องจากมีวงเล็บเรารู้ว่าก่อนที่จะคูณได้ ต้องคำนวณในวงเล็บให้เสร็จก่อน
- ลำดับชั้นของ tree ทำให้เราเข้าใจลำดับการทำงานของการประมวลผล expression

- ก่อนที่จะทำ level บน เราต้องทำ level ล่างให้เสร็จก่อนในรูป คือ เราต้องบวกลบให้เสร็จก่อนคูณ การบวกซึ่งเป็น left sub tree ได้ผลลัพธ์ 10 การลบได้ผลลัพธ์ 3



- จากนั้นเราจะเปลี่ยน subtree เป็นโหนดได้แล้วค่อยไปคูณต่อ

ต่อไปเราจะมาดู

- วิธีการสร้าง parse tree จาก mathematical expression
- วิธีคำนวณ expression ที่เก็บใน parse tree
- วิธีเปลี่ยนกลับเป็น mathematical expression

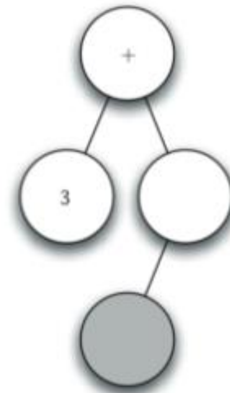
- ขั้นแรกในการสร้าง parse tree คือการแตก expression ออกเป็น list ของ token (สัญลักษณ์)
- token มี 4 กลุ่มคือ วงเล็บเปิด วงเล็บปิด operator operand
- เรารู้ว่าเมื่อไรก็ตามที่อ่านแล้วเจอวงเล็บเปิด นั่นคือเรากำลังจะเริ่ม expression ใหม่ ดังนั้นเราควรจะสร้าง tree ใหม่ ในทางกลับกันเมื่อเราเจอวงเล็บปิดแสดงว่าจบ expression
- เรารู้อีกว่า operand จะไปเป็น leaf node และเป็นลูกของ operator สุดท้ายเรารู้ว่าทุก operator จะมีลูกทางซ้ายและขวา

- ทำให้เราสร้างกฎ 4 ข้อได้ดังนี้
- ถ้า token ปัจจุบันเป็น '(' จะเพิ่ม node ใหม่เป็นลูกทางซ้ายของ node ปัจจุบันแล้วย้ายไปลูกทางซ้าย
- ถ้า token ปัจจุบันเป็น['+','-','*','/'] กำหนดค่าให้โหนดปัจจุบันเป็น operator ตัวนั้น เพิ่มโหนดใหม่เป็นลูกทางขวาแล้วย้ายไปลูกทางขวา
- ถ้า token ปัจจุบันเป็นตัวเลข กำหนดค่าให้โหนดนั้นเป็นนตัวเลขแล้วกลับมาที่ parent
- ถ้า token ปัจจุบันเป็น ')' กลับไปที่ parent ของโหนดปัจจุบัน

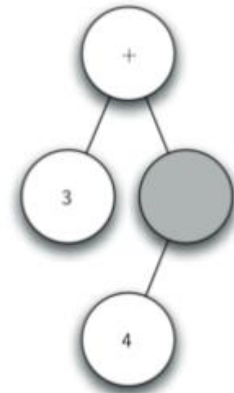
- ตัวอย่าง สมมติว่ามี expression (3+(4*5))

(a)

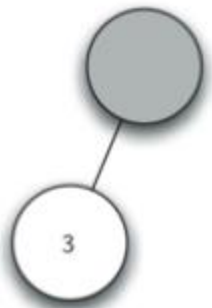
(b)



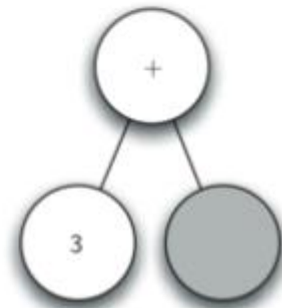
(e)



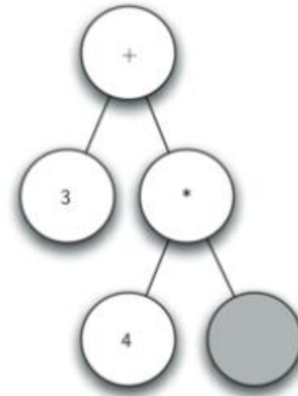
(f)



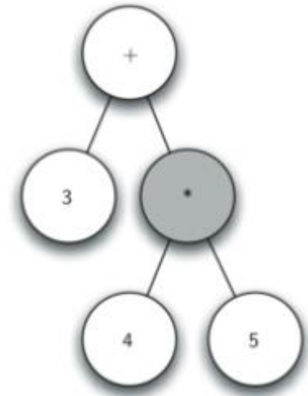
(c)



(d)



(g)



(h)

- เราจะนำเอา class tree ที่เขียนไว้มาประยุกต์ใช้ จากตัวอย่าง จะเห็นได้ว่าเราสนใจ โหนดปัจจุบัน parent ของมัน ลูกทางซ้าย ลูกทางขวา ใน code เรามี get_left_child และ get_right_child
- แล้วเราจะจัดการอย่างไรกับ parent
- วิธีง่ายสุดคือใช้ stack เพื่อจะได้ย้อนกลับได้
- เมื่อไรก็ตามที่เราจะต้องลงไปที่โหนดลูก เราจะ push โหนดปัจจุบันลง stack ไว้ เมื่อเราจะกลับมาเราก็ pop เอา

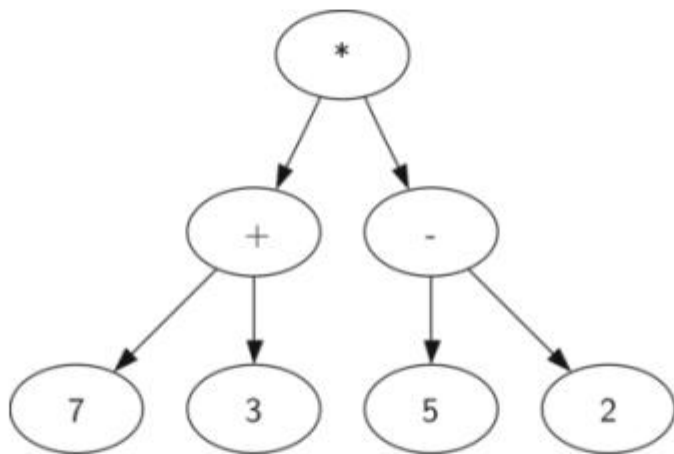
- เราจะใช้ทั้ง class Stack และ BinaryTree

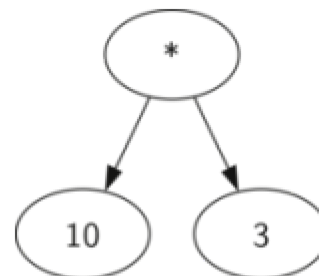
```
def build_parse_tree(fp_exp):
    fp_list = fp_exp.split()
    p_stack = Stack()
    e_tree = BinaryTree('')
    p_stack.push(e_tree)
    current_tree = e_tree
    for i in fp_list:
        if i == '(':
            current_tree.insert_left('')
            p_stack.push(current_tree)
            current_tree = current_tree.get_left_child()
        elif i not in ['+','-','*','/',')']:
            current_tree.set_root_val(int(i))
            parent = p_stack.pop()
            current_tree = parent
        elif i in ['+','-','*','/']:
            current_tree.set_root_val(int(i))
            current_tree.insert_right('')
            p_stack.push(current_tree)
            current_tree = current_tree.get_right_child()
```

```
        elif i == ')':
            current_tree = p_stack.pop()
        else:
            raise ValueError
    return e_tree
pt = build_parse_tree("( ( 10 + 5 ) * 3")
pt.postorder()
```

- จากกฎทั้ง 4 ข้อเราก็มาเขียนเป็น if ในบรรทัดที่ 11, 15,19 และ 24 ในแต่ละกรณีเราก็ code ตามกฎโดยมีการเรียกใช้ BinaryTree และ Stack
- ส่วนที่ตรวจสอบ error คือ else อันสุดท้ายที่เราเรียก ValueError ซึ่งเกิดเมื่อ token ที่รับเข้ามาเราไม่รู้จัก
- ตอนนี้เราได้ parse tree แล้วต่อไปเราจะเขียนฟังก์ชัน ประมวลผล โดยคืนผลลัพธ์เป็นตัวเลข

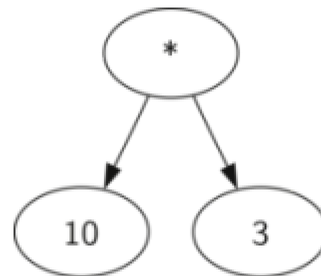
- เราจะเปลี่ยนจาก tree ต้นฉบับเป็นด้านขวา แล้วเปลี่ยนเป็นผลลัพธ์





- เราจะเขียนเป็น recursive เราจะเริ่มต้นการออกแบบ recursive algorithm ด้วย base case
- Base case คือตรวจสอบ leaf node ซึ่งใน parse tree นั้น leaf node จะเป็น operand เสมอ
- เนื่องจากวัตถุที่เป็นเลขจำนวนเต็มหรือจำนวนจริง ไม่ต้องการแปลความ ใช้ได้เลยในฟังก์ชัน evaluate เราก็จะคืนค่าที่เก็บไว้ที่ leaf node ได้เลย

- ในการเก็บผลลัพธ์ของการเรียก recursive 2 ครั้ง เราจะนำเอา operand ที่เก็บใน โหนด parent มาประมวลผลคำตอบของการคำนวณค่าของ node ลูก
- ตัวอย่างเช่น โหนดลูกสองตัวประมวลผลได้ 10 และ 3 หลังจากนั้นประมวลผลต่อได้ผลลัพธ์สุดท้ายคือ 30



- Code ของ evaluate เริ่มต้นจะได้ค่าของลูกทางซ้ายและขวาของโหนดปัจจุบัน ถ้าลูกทางซ้ายและขวา ประมวลได้ None แสดงว่า โหนดปัจจุบันเป็น leaf node

ถ้าโหนดปัจจุบันไม่ใช่ leaf node จะมองไปที่ operator ของ current node แล้วใช้มันหาผลลัพธ์จากการประมวลของลูกทางซ้ายและลูกทางขวา

- ในการเขียน code เราจะใช้ dictionary กับ key '+','-','*','/' ค่าที่เก็บใน dictionary เป็นฟังก์ชันจาก Python operator module
- Operator module จะฟังก์ชันหลายรูปแบบที่ใช้กันบ่อยๆ
- เมื่อเราค้นหาใน dictionary เราก็จะได้ฟังก์ชันที่สอดคล้องกัน
- ตัวอย่างเช่นหากค้นหาได้ opers['+'](2,2) เทียบได้กับการเรียกฟังก์ชัน operator.add(2,2)

```
import operator
def  evaluate(parse_tree):
    opers = {'+':operator.add,'-':operator.sub,'*':operator.mul,'/':operator.truediv}
    left = parse_tree.get_left_child()
    right = parse_tree.get_right_child()
    if left and right:
        fn = opers[parse_tree.get_root_val()]
        return fn(evaluate(left), evaluate(right))
    else:
        return parse_tree.get_root_val()
```