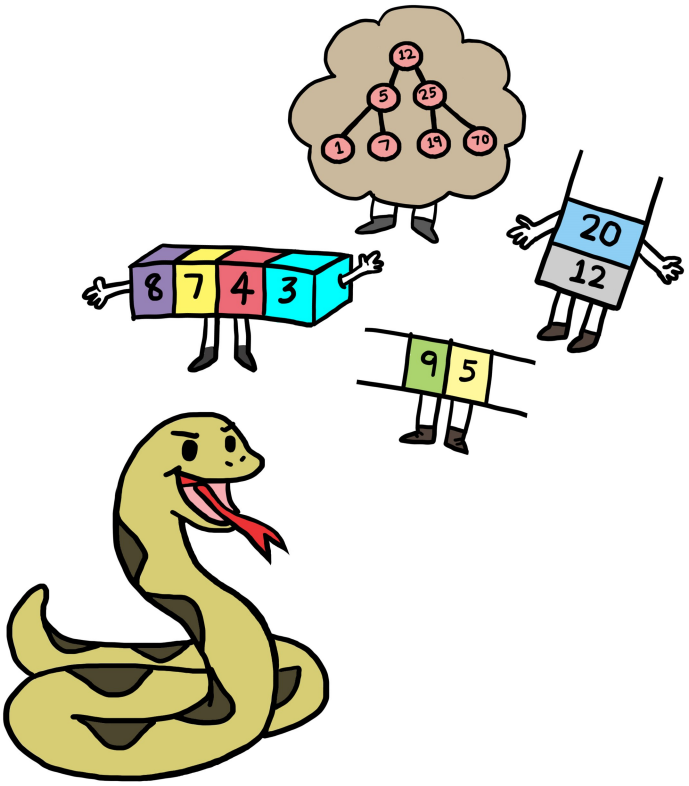
เอกสารประกอบการสอน

กระบวนวิชา

204700

โครงสร้างข้อมูลและภาษาโปรแกรม

(DATA STRUCTURES AND PROGRAMMING LANGUAGES)



จักริน ชวชาติ

ภาควิชาวิทยาการคอมพิวเตอร์

คณะวิทยาศาสตร์ มหาวิทยาลัยเชียงใหม่

2559

สาขาวิทยาการคอมพิวเตอร์ **คณะวิทยาศาสตร์**

ว.คพ. 700 (204700) : โครงสร้างข้อมูลและภาษาโปรแกรม 2(2-0-4)

เงื่อนไขที่ต้องผ่านก่อน ไม่มี

คำอธิบายลักษณะกระบวนวิชา

แบบชนิดข้อมูลนามธรรม โครงสร้างข้อมูลแบบเชิงเส้น โครงสร้างข้อมูลแบบไม่เชิงเส้น เทคนิคการค้นหาและการเรียงลำดับ รูปแบบของภาษาโปรแกรม

วัตถุประสงค์กระบวนวิชา

นักศึกษามีความรู้เกี่ยวกับแนวคิดพื้นฐานในการเขียนโปรแกรม การออกแบบข้อมูลและภาษาโปรแกรม

เนื้อหากระบวนวิชา	**จำนวนชั่วโมงบรรยาย**
แบบชนิดข้อมูลนามธรรม	2
โครงสร้างข้อมูลแบบเชิงเส้น	4
- ลิสต์	
- สแตก	
- คิว	
โครงสร้างข้อมูลแบบไม่เชิงเส้น	5
- กราฟ	
- ทรี	
เทคนิคการค้นหาและเรียงลำดับ	5
รูปแบบของภาษาโปรแกรม	14
- รูปแบบของภาษาโปรแกรมแบบลำดับ	
- รูปแบบของภาษาโปรแกรมเชิงวัตถุ	
รวม	30

คำนำ

เอกสารประกอบการสอนเล่มนี้ ผู้เขียนได้เรียบเรียงขึ้นมาเพื่อใช้ประกอบการเรียนการสอนกระบวนวิชา โครงสร้างข้อมูลและภาษาโปรแกรม (DATA STRUCTURES AND PROGRAMMING LANGUAGES) รหัสกระบวนวิชา 204700 ซึ่งเป็นวิชาในหลักสูตรวิทยาศาสตรมหาบัณฑิต ของภาควิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์ มหาวิทยาลัยเชียงใหม่

ทั้งนี้ทางผู้เขียนได้นำเอาโครงและลำดับการนำเสนอเนื้อหาบางส่วนมาจากหนังสือ "Problem Solving with Algorithms and Data Structures Using Python" ซึ่งแต่งโดย Brad Miller และ David Ranum[3] เนื่องจากอธิบายเข้าใจง่ายและทางผู้เขียนคิดว่าจะทำให้เข้าใจได้มากขึ้น

ในบทแรกจะกล่าวถึงภาษาไพธอนเบื้องต้น เนื้อหาจะเป็นการทบทวนการเขียนโปรแกรมและทำความรู้จักวิธีการรับข้อมูล การแสดงผลข้อมูล โครงสร้างการเขียนโปรแกรม ทั้งนี้ภาษาไพธอนนั้นหากผู้อ่านเคยเขียนภาษาโปรแกรมภาษาอื่นมาก่อนจะปรับตัวได้อย่างรวดเร็วเพียงแค่รู้คำสั่งที่จำเป็น ในบทที่สองจะเป็นหัวข้อโครงสร้างข้อมูลพื้นฐาน ซึ่งได้แก่ Stack, Queue, Deque และ List จากนั้นในบทที่สามจะเป็นหัวข้อการเวียนเกิด บทที่สี่จะเป็นปัญหาพื้นฐานทางวิทยาการคอมพิวเตอร์สองปัญหาคือการค้นหาและการเรียง ในบทที่สี่นี้จะได้ยกตัวอย่างการค้นหาทั้งการค้นหาตามลำดับ การค้นหาทวิภาคและการแฮช สำหรับการเรียงก็มีทั้ง Bubble sort, Selection sort, Insertion sort Merge sort และปิดท้ายหัวข้อการเรียงด้วย Quick sort เนื้อหาในบทสุดท้ายจะเป็นเรื่อง Tree ซึ่งเป็นโครงสร้างข้อมูลที่เราพบได้บ่อยในชีวิตประจำวันอีกตัวหนึ่ง

ผู้เขียนหวังว่าเอกสารประกอบการสอนฉบับนี้จะเป็นประโยชน์ต่อนักศึกษาและผู้สนใจทั่วไปไม่มากก็น้อย

จักริน ชวชาติ

สารบัญ

สารบัญรูป

สารบัญตาราง

บทที่ 1

ภาษาไพธอน

ในบทนี้จะเป็นการแนะนำเบื้องต้นเกี่ยวกับ**ภาษาไพธอน(Python)**[2] ซึ่งเป็นภาษาโปรแกรมที่เราจะใช้เรียนกันต่อไป ทำไมจึงเลือกภาษาไพธอนมาใช้ในการเรียนการสอน เนื่องจากว่าภาษาไพธอนเป็นภาษาระดับสูงสมัยใหม่ที่ง่ายต่อการเรียน โครงสร้างภาษาไม่ซับซ้อน มีส่วนเสริมให้ใช้งานมากมายและส่วนใหญ่ฟรี อีกทั้งยังเป็นภาษาเชิงวัตถุ นอกจากนี้ในปัจจุบันมีการใช้ภาษาไพธอนในการเขียนโปรแกรมที่ใช้งานจริงเป็นจำนวนมาก

1.1 เริ่มต้นการอ้างอิงและใช้ข้อมูล

ในการเขียนโปรแกรมจุดประสงค์หลักอย่างหนึ่งคือการคำนวณหาผลลัพธ์ ซึ่งการคำนวณนั้นส่วนใหญ่เป็นการดำเนินการเกี่ยวกับข้อมูลที่เรามี ดังนั้นก่อนอื่นเราจะมาดูวิธีการอ้างอิงข้อมูลและชนิดข้อมูลกันก่อน

1.1.1 ตัวแปร

การอ้างอิงค่าต่างๆ ในภาษาไพธอนนั้นจะใช้ตัวแปร(Variable) ตัวแปรเป็นสิ่งที่กำหนดมาเพื่ออ้างอิงค่าที่จะนำไปใช้ต่อไป ตัวแปรในภาษาไพธอนจะถูกสร้างเมื่อมีการอ้างอิงชื่อของตัวแปรนั้นในครั้งแรก การกำหนดการอ้างอิงค่าให้กับตัวแปรทำได้โดยตั้งชื่อตัวแปรจากนั้นตามด้วยเครื่องหมายเท่ากับ(=) และตามด้วยค่าที่ต้องการ ทั้งนี้สิ่งที่ต้องระวังคือตัวแปรจะเก็บการอ้างอิงค่า ไม่ใช่ค่านั้นโดยตรง

ตัวอย่างการประกาศตัวแปรพร้อมกำหนดการอ้างอิงค่าให้กับตัวแปร

```
a = "204700 DATA STRUCTURE AND PROGRAMMING LANGUAGES"
b = 1234567890
c = 42.195
```

ตัวแปร a อ้างอิงข้อความ ”204700 DATA STRUCTURE AND PROGRAMMING LANGUAGES” ตัวแปร b อ้างอิงเลขจำนวนเต็ม 1234567890 ส่วนตัวแปร c อ้างอิงเลขจำนวนจริง 42.195

การตั้งชื่อตัวแปรในภาษาไพธอนนั้นสามารถใช้ได้เฉพาะตัวอักษร ตัวเลขและเครื่องหมายขีดเส้นใต้(Underscore) ในการตั้งชื่อเท่านั้น โดยมีเงื่อนไขคืออักขระแรกของชื่อตัวแปรต้องขึ้นต้นด้วยตัวอักษรหรือเครื่องหมายขีดเส้นใต้เท่านั้น ส่วนอักขระตัวอื่นสามารถใช้ตัวเลขได้ด้วย ข้อควรระวังของการตั้งชื่อตัวแปรในภาษาไพธอนคือตัวอักขระพิมพ์เล็กและพิมพ์ใหญ่นั้นถือว่าไม่เหมือนกัน นั่นคือตัวแปร val กับ Val ถือว่าเป็นตัวแปรคนละตัวกัน

1.1.2 ชนิดข้อมูล

ค่าของข้อมูลที่ตัวแปรนั้นอ้างอิงในภาษาไพธอนนั้นสามารถจัดกลุ่มได้หลายชนิด แต่ละชนิดก็สามารถนำไปประมวลผลได้ต่างกัน เช่น ตัวเลขสามารถดำเนินการทางคณิตศาสตร์ได้เป็นต้น ในเบื้องต้นเราจะแนะนำชนิดข้อมูลที่ภาษาไพธอนมีให้ แบ่งได้กว้างๆ 2 ชนิดได้แก่ ชนิดข้อมูลเชิงเดี่ยว และ ชนิดข้อมูลเป็นกลุ่ม

ชนิดข้อมูลเชิงเดี่ยว

เมื่อกล่าวถึงชนิดข้อมูลเชิงเดี่ยวจะหมายถึงชนิดข้อมูลที่ไม่สามารถแบ่งย่อยได้อีก ในภาษาไพธอนนั้นมีชนิดข้อมูลเชิงเดี่ยวเกี่ยวกับจำนวนอยู่ 2 ชนิด ได้แก่ จำนวนเต็มหรือ integer และจำนวนจริงหรือ float โดยชนิดข้อมูลจำนวนเหล่านี้มีการดำเนินการพื้นฐานทางคณิตศาสตร์ได้แก่ +(บวก), -(ลบ), *(คูณ), /(หาร), //(หารไม่เก็บเศษ), %(หารเก็บเศษ) และ **(ยกกำลัง) เมื่อมีการใช้การดำเนินการมากกว่า 1 ชนิด ลำดับในการคำนวณก็จะเหมือนกันกับภาษาโปรแกรมอื่นทั่วไป นั่นคือจะทำ ** ก่อน จากนั้นทำ *, /, //, % แล้วจึงทำ +, - และหากมีลำดับในการทำงานเท่ากันจะทำจากซ้ายมือไปขวามือ ทั้งนี้เราสามารถระบุลำดับการทำก่อนหลังเองได้ด้วยการใช้วงเล็บครอบนิพจน์ที่เราต้องการให้คำนวณก่อน

ตารางที่ 1.1: ตัวอย่างการดำเนินการทางคณิตศาสตร์พื้นฐานของจำนวน

คำสั่ง	ผลลัพธ์
`print(5+6*7)`	47
`print((5+6)*7)`	77
`print(2**10)`	1024
`print(10/3)`	3.3333333333333335
`print(10//2)`	5.0
`print(10//3)`	3
`print(10%3)`	1

หมายเหตุ คำสั่ง print() เป็นคำสั่งที่ใช้แสดงผลข้อมูลที่อยู่ในวงเล็บ ส่วน # เป็นการ comment

นอกจากชนิดข้อมูลแบบจำนวน(integer และ float) แล้วยังมีชนิดข้อมูลแบบบูลีน(boolean) ซึ่งเป็นชนิดข้อมูลที่มีค่า 2 ค่าที่เป็นไปได้ได้แก่ จริง(True) และ เท็จ(False) โดยการดำเนินการพื้นฐานกับข้อมูลแบบบูลีนเป็นการดำเนินการทางตรรกศาสตร์ซึ่งได้แก่ and(และ), or(หรือ) และ not(นิเสธ)

ตัวอย่างการประกาศตัวแปรชนิดบูลีนและการดำเนินการพื้นฐาน

```
a = True
b = False
print(a or b) #True or False ==> True
print(a and b) #True and False ==> False
print(not (a or b)) #False
```

นอกจากนี้ชนิดข้อมูลแบบบูลีนนั้นยังถูกใช้เป็นผลลัพธ์จากการเปรียบเทียบอีกด้วย เช่น ใช้เป็นผลลัพธ์จากการเปรียบเทียบว่าค่าที่ตัวดำเนินการสองตัวนั้นอ้างอิงอยู่ว่าเท่ากันหรือไม่ เปรียบเทียบโดยใช้สัญลักษณ์ == เช่น a == b จะให้ผลลัพธ์เป็นจริงถ้า a และ b มีค่าเท่ากัน แล้วให้ผลลัพธ์เป็นเท็จถ้า a และ b มีค่าไม่เท่ากัน หรือ หากต้องการผลลัพธ์จากการเปรียบเทียบว่าตัวดำเนินการทางซ้ายมืออ้างอิงค่าที่มากกว่าตัวดำเนินการทางขวามือหรือไม่ เปรียบเทียบโดยใช้สัญลักษณ์ > ส่วนการใช้งาน != ใช้เมื่อต้องการเปรียบเทียบว่าตัวดำเนินการสองตัวนั้นอ้างอิงค่าไม่เท่ากันใช่หรือไม่ เป็นต้น นอกจากนี้ยังสามารถนำเอาตัวดำเนินการมาใช้ร่วมกันเพื่อเป็นการสร้างการสอบถามที่ซับซ้อนขึ้นได้ด้วย ตัวอย่างเช่น

```
val1 = 5 == 10 #False
val2 = 10 > 5 #True
print((5 >= 1) and (5 <= 10)) #True and True ==> True
```

ชนิดข้อมูลเป็นกลุ่ม

นอกจากชนิดข้อมูลเชิงเดี่ยวแล้ว ภาษาไพธอนยังมีชนิดข้อมูลที่อ้างอิงข้อมูลเป็นกลุ่มอีกด้วยโดยแบ่งออกได้เป็น 2 ประเภท ประเภทแรกคือกลุ่มของข้อมูลที่มีลำดับ ได้แก่ List, String และ Tuples ส่วนประเภทที่สองคือกลุ่มของข้อมูลที่ไม่มีลำดับได้แก่ Sets และ Dictionaries ทั้งนี้กลุ่มของข้อมูลที่มีลำดับนั้นมีการดำเนินการพื้นฐานเช่นเดียวกันดังตารางที่ 1.2

ตารางที่ 1.2: การดำเนินการกับกลุ่มข้อมูลที่มีลำดับ

ชื่อการดำเนินการ	ตัวดำเนินการ	คำอธิบาย
indexing	[ ]	การเข้าถึงสมาชิกในลำดับ
concatenation	+	การรวมลำดับเข้าด้วยกัน
repetition	*	การทำซ้ำของลำดับ
membership	in	การสอบถามว่ามี item นี้ในลำดับหรือไม่
length	len	การสอบถามจำนวน item ในลำดับ
slicing	[:]	การแบ่งส่วนของลำดับ

List

List เป็นกลุ่มข้อมูลใดๆ ที่มีลำดับ สามารถมีข้อมูลได้ตั้งแต่ 0 ตัวเป็นต้นไป List สร้างได้ด้วยการใช้ปีกกาเหลี่ยม [ ] ครอบข้อมูลที่ต้องการ ในการสร้าง List นั้นสามารถสร้างได้โดยระบุสมาชิกให้กับ List โดยคั่นสมาชิกแต่ละตัวด้วยจุลภาค ตัวอย่างเช่น [6, 'book', 4.3] หากเป็น List ว่างเขียนแทนด้วย [] ข้อสังเกตสมาชิกใน List ไม่จำเป็นต้องมีชนิดข้อมูลเดียวกัน

ในการอ้างอิงค่าของสมาชิกใน List นั้นสามารถใช้ตำแหน่งในลำดับ(index) ของข้อมูลใน List ได้โดยตำแหน่งในลำดับเริ่มต้นที่ 0 จนถึง N-1 เมื่อ N เป็นขนาดหรือจำนวนของสมาชิกใน List ซึ่งการอ้างอิงเช่นนี้มีลักษณะเดียวกับ Array ในภาษาซี หากต้องการแบ่งส่วนของ List เราจะใช้ : (Slice Operation) ในการระบุช่วงของข้อมูลใน List เช่น li[1:3] จะคืนค่าของสมาชิกของ li ที่**เริ่มต้น**ด้วยสมาชิก index ที่ 1 เป็นต้นไปจนกระทั่ง**ก่อน**สมาชิก index ที่ 3

นอกจากนี้ List ยังมีฟังก์ชัน(ทางภาษาเชิงวัตถุเรียกว่า Method) ที่สามารถเรียกใช้งานได้หลายฟังก์ชัน เพื่อช่วยให้ทำงานได้ง่ายขึ้น สมมติกำหนดให้ a เป็น List ตารางที่ 1.3 จะแสดง Method ของ List ตัวอย่างการใช้งานและคำอธิบายของ Method นั้นๆ

ตารางที่ 1.3: Method ของ List

ชื่อ Method	การใช้งาน	คำอธิบาย
append	`a.append(item)`	เพิ่ม item ใหม่ต่อท้าย List
insert	`a.insert(i,item)`	แทรก item ที่ตำแหน่ง i ใน List
pop	`a.pop()`	ลบและคืนค่า item ตัวสุดท้ายใน List
pop	`a.pop(i)`	ลบและคืนค่า item ตำแหน่งที่ i ใน List
sort	`a.sort()`	เรียงลำดับ list
sort	`a.reverse()`	เรียง list กลับด้าน
del	`del a[i]`	ลบ item ตำแหน่งที่ i ใน List
index	`a.index(item)`	คืนตำแหน่งของ item ที่เจอใน List ครั้งแรก
count	`a.count(item)`	คืนจำนวนครั้งที่พบ item ใน List
remove	`a.remove(item)`	ลบ item ใน List ที่พบตัวแรก

String

String หรือข้อความ เป็นกลุ่มลำดับของอักขระสามารถเป็นได้ทั้งตัวอักษร ตัวเลขหรือสัญลักษณ์ต่างๆ ความยาวของ String นั้นสามารถมีได้ตั้งแต่ 0 ตัวขึ้นไป การบ่งบอกว่าข้อมูลนี้เป็นข้อความหรือไม่นั้นสังเกตได้จากเครื่องหมายอัญประกาศหรือเครื่องหมายคำพูด โดยสามารถใช้ได้ทั้งอัญประกาศคู่(" ") และอัญประกาศเดี่ยว(' ') หมายเหตุ หากต้องการข้อความหลายบรรทัดสามารถใช้อัญประกาศเดี่ยวหรืออัญประกาศเดี่ยวสามอันตามด้วยข้อความ และปิดข้อความด้วยอัญประกาศชนิดก่อนหน้าสามอันได้ เนื่องจาก String นั้นเป็นลำดับของอักขระ ดังนั้นการดำเนินงานกับลำดับตามตารางที่ 1.2 สามารถใช้กับ String ได้เช่นกัน นอกจากนี้ ยังมีฟังก์ชันซึ่งช่วยให้ทำงานกับ String อีกหลายฟังก์ชัน สมมติกำหนดให้ a เป็น String ตารางที่ 1.4 จะเป็น Method ของ String ตัวอย่างการใช้งานและคำอธิบายของ Method นั้นๆ

ตารางที่ 1.4: Method ของ String

ชื่อ Method	**การใช้งาน**	**คำอธิบาย**
center	`a.center(w)`	คืนค่าข้อความความยาว w อักขระที่เอาช่องว่างไปแทรกเพื่อให้ข้อความ a อยู่ตรงกลาง
count	`a.count(item)`	คืนค่าจำนวนของ item ที่พบในข้อความ a
ljust	`a.ljust(w)`	คืนค่าข้อความความยาว w อักขระที่มีข้อความ a อยู่ชิดซ้ายจากนั้นเติมช่องว่างให้ครบ
rjust	`a.rjust(w)`	คืนค่าข้อความความยาว w ที่มีข้อความ a อยู่ชิดขวาโดยด้านซ้ายเป็นการเติมช่องว่าง
lower	`a.lower()`	คืนค่าข้อความ a ที่เปลี่ยนเป็นตัวอักขระตัวพิมพ์เล็กทั้งหมด
upper	`a.upper()`	คืนค่าข้อความ a ที่เปลี่ยนเป็นตัวอักขระตัวพิมพ์ใหญ่ทั้งหมด
find	`a.find(item)`	คืนตำแหน่งที่พบ item เป็นครั้งแรก
split	`a.split(x)`	แบ่งข้อความออกเป็น List ของข้อความย่อยโดยใช้ x เป็นตัวแบ่ง

ตัวอย่าง สมมติว่า a = 'xxx' เมื่อมีการเรียกคำสั่ง a.center(9) จะคืนค่าเป็นข้อความ ' xxx ' เมื่อมีการเรียก a.upper() จะคืนค่า 'XXX' และหาก a = 'Happy Birthday to you' แล้วเรียก a.split(' ') จะได้ List ['Happy','Birthday','to','you'] เป็นต้น

ข้อแตกต่างระหว่าง List กับ String คือ List เป็น Mutable ส่วน String เป็น Immutable นั่นคือเมื่อสร้างแล้ว List สามารถเปลี่ยนแปลงค่าของสมาชิกแต่ละตัวได้แต่ String เปลี่ยนแปลงค่าของสมาชิกไม่ได้ ตัวอย่างเช่นสามารถเปลี่ยนข้อมูลใน List ได้โดยใช้การกำหนดค่าระบุไปที่ index ที่ต้องการแต่ String ไม่อนุญาตให้ทำได้

Tuple

Tuple คล้ายกับ List ตรงที่เป็นลำดับของข้อมูลที่ชนิดข้อมูลแตกต่างกันได้ สิ่งที่ต่างกันคือ Tuple นั้นเป็น Immutable นั่นคือเปลี่ยนค่าไม่ได้เช่นเดียวกับ String การสร้าง Tuple สร้างได้ด้วยการใช้วงเล็บโดยที่ข้อมูลสมาชิกแต่ละตัวภายใน Tuple ถูกคั่นด้วยจุลภาค Tuple สามารถใช้การดำเนินงานกับลำดับตามตารางที่ 1.2 ได้เช่นกัน

```
t1 = ('Steven', 'Gerrard', 12345678, 1000)
t2 = (1, 2, 3, 4, 5, 6, 7 )

print ("t1[0]: ", t1[0]) #t1[0]:  Steven
print ("t2[2:4]: ", t2[2:4]) #t2[2:4]:  (3, 4)
```

Set

Set เป็นกลุ่มข้อมูลที่ไม่มีลำดับตั้งแต่ 0 ตัวขึ้นไป ทั้งนี้ Set ไม่อนุญาตให้มีข้อมูลซ้ำกัน Set สร้างด้วยการเขียนสมาชิกคั่นด้วยจุลภาคภายในปีกกาโค้ง $\{\}$ ไม่จำเป็นต้องมีชนิดข้อมูลประเภทเดียวกันได้ แม้ว่า Set จะไม่ใช่ข้อมูลแบบลำดับแต่ก็มีหลายการดำเนินการที่ทำได้

สมมติกำหนดให้ x และ y เป็น Set ตารางที่ 1.5 แสดงการดำเนินการกับ Set

ตารางที่ 1.5: การดำเนินการกับ Set

การดำเนินการ	การใช้งาน	คำอธิบาย
in	`a in x`	ตรวจสอบว่า a อยู่ใน Set x หรือไม่
len	`len(x)`	คืนค่าจำนวนของ item ใน Set x
|	`x | y`	คืนค่า Set ใหม่ที่มาจากทั้ง 2 Set x และ y
&	`x & y`	คืนค่า Set ใหม่ที่ต้องอยู่ในทั้ง 2 Set x และ y
-	`x - y`	คืนค่า Set ที่สมาชิกนั้นมีอยู่เฉพาะใน Set x แต่ไม่อยู่ใน Set y
<=	`x <= y`	ถามว่าทุกสมาชิกใน Set x อยู่ใน Set y

นอกจากนี้ Set ยังรองรับ Method ทางคณิตศาสตร์ดังต่อไปนี้ด้วย

ตารางที่ 1.6: Method กับ Set

ชื่อ Method	การใช้งาน	คำอธิบาย
union	`set1.union(set2)`	คืนค่า set ใหม่ที่มาจากสมาชิกทั้ง 2 set
intersection	`set1.intersection(set2)`	คืนค่า set ใหม่ที่เหมือนกันจากสมาชิกทั้ง 2 set
difference	`set1.difference(set2)`	คืนค่า set ใหม่ที่อยู่ใน set1 แต่ไม่อยู่ใน set2
issubset	`set1.issubset(set2)`	ถามว่าทุกสมาชิกของ set1 อยู่ใน set2
add	`set.add(item)`	เพิ่ม item ให้ set
remove	`set.remove(item)`	ลบ item ออกจาก set
pop	`set.pop()`	ลบ item ใดๆ ออกจาก set 1 ตัว
clear	`set.clear()`	ลบทุก item ออกจาก set

Dictionary

ชนิดข้อมูลแบบไม่มีลำดับอย่างสุดท้ายคือ Dictionary จะเป็นกลุ่มของคู่ที่สัมพันธ์กันซึ่งประกอบไปด้วย Key และ Value เปรียบได้กับ คำและความหมายในพจนานุกรมนั่นเอง โดยทั่วไปแล้ว Key และ Value จะเขียนด้วย **Key:Value** คั่นด้วยจุลภาคและอยู่ภายในปีกกาโค้ง ตัวอย่างเช่น

```
capitals = {'Thailand':'Bangkok','Japan':'Tokyo'}
```

ในการจัดการกับ Dictionary นั้นหากเราต้องการใช้ค่า Value เราจะเข้าถึงผ่านทาง Key หากต้องการเพิ่มข้อมูลก็ทำการกำหนด Value ให้กับ Key ใหม่ได้เลย ตัวอย่างเช่น

```
capitals = {'Thailand':'Bangkok','Japan':'Tokyo'}
print(capitals['Thailand'])  #Bangkok
capitals['Germany'] = 'Berlin'
```

ในบรรทัดสุดท้ายของตัวอย่างจะเป็นการกำหนดค่า Key ของตัวแปร capital มีค่าเป็น Germany และค่า Value ที่สอดคล้องคือ Berlin เนื่องจาก Germany ยังไม่มีใน capital ดังนั้นจะเป็นการเพิ่มข้อมูล

สมมติกำหนดให้ d เป็น Dictionary แล้ว Dictionary มีการดำเนินการและ Method ดังนี้

ตารางที่ 1.7: การดำเนินการกับ Dictionary

Operator	การใช้งาน	คำอธิบาย
[]	`d[k]`	จะคืนค่า Value ของ dictionary d ที่มี Key k
in	`key in d`	จะคืนค่า True ถ้ามี Key ใน d กรณีอื่นคืนค่า False
del	`del d[k]`	ลบ Key:Value ที่มี Key เป็น k ออกจาก dictionary d

ตารางที่ 1.8: Method กับ Dictionary

ชื่อ Method	การใช้งาน	คำอธิบาย
key	`d.key()`	จะคืนค่า Key ของ dictionary d
values	`d.key()`	จะคืนค่า Values ของ dictionary d
items	`d.items()`	จะคืนค่า Key-Value ของ dictionary d
get	`d.get(k)`	จะคืนค่า Value ที่สัมพันธ์กับ k ถ้าไม่เจอคืนค่า None
get	`d.get(k,alt)`	จะคืนค่า Value ที่สัมพันธ์กับ k ถ้าไม่เจอคืนค่า alt

1.2 การนำเข้าและแสดงผลข้อมูล

ในการเขียนโปรแกรมส่วนใหญ่นั้นจะมีการติดต่อกับผู้ใช้งานเพื่อรับข้อมูลเข้าที่จำเป็นในการคำนวณเข้าสู่โปรแกรม อีกทั้งเมื่อโปรแกรมคำนวณเสร็จแล้วจะมีผลลัพธ์ที่เป็นข้อมูลออกคืนกลับให้ผู้ใช้ในรูปแบบต่างๆ โปรแกรมส่วนใหญ่

ในปัจจุบันนั้นมักจะมีกล่องข้อความ(Dialog Box) ให้ผู้ใช้งานกรอกข้อมูลเพื่อความสะดวกของผู้ใช้งาน ทั้งนี้ไพธอนเองก็สามารถทำได้ แต่ในเบื้องต้นเราจะเรียกใช้ฟังก์ชัน input() ที่จะให้ผู้ใช้กรอกข้อมูล หลังจากนั้นจะคืนค่าที่รับมาในรูปของข้อความ(String)

ฟังก์ชัน input() รับ parameter 1 ตัวเป็นข้อความ โดยข้อความนี้จะแสดงเพื่อแจ้งให้แก่ผู้ใช้เพื่อทำการกรอกข้อมูล ปกติแล้วฟังก์ชัน input จะคืนค่าเป็นข้อความ หากต้องการให้ค่าที่รับมาเป็นตัวเราจะต้องทำการแปลง(Casting) ก่อนให้ไปเป็น int หรือ float เพื่อนำไปคำนวณต่อไปได้ ตัวอย่างเช่น

```
name = input('Plese input your name')
age = int(input('Plese input your age'))
height = float(input('Plese input your height'))
```

บรรทัดแรกของตัวอย่างเป็นการรับข้อมูลให้กับตัวแปร name โดยข้อมูลที่รับจะมีชนิดเป็นข้อความและจะมีข้อความ 'Plese input your name' ปรากฏให้กับผู้ใช้ก่อน บรรทัดที่สองเป็นการรับข้อมูลให้กับตัวแปร age โดยข้อมูลที่รับจะมีการแปลงให้เป็นจำนวนเต็มและบรรทัดสุดท้ายเป็นการรับข้อมูลให้กับตัวแปร height โดยข้อมูลที่รับจะมีการแปลงให้เป็นจำนวนจริง

ในการแสดงผลเราใช้ฟังก์ชัน print() เราสามารถควบคุมการแสดงผลได้โดยเพิ่ม parameter ให้กับฟังก์ชัน print()

- sep หากมีข้อมูลที่ต้องการพิมพ์มากกว่า 1 ตัวแล้ว ระหว่างข้อมูลแต่ละชุดจะให้คั่นด้วยอะไร โดยค่าเริ่มต้นเป็นช่องว่าง
- end ต้องการให้จบข้อความที่พิมพ์แล้วตามด้วยอะไร โดยค่าเริ่มต้นเป็นขึ้นบรรทัดใหม่

ตัวอย่างเช่น

```
print('hello')  #hello
print('hello','world')   #hello world
print('hello','world',sep='xx')   #helloxxworld
print('hello','world',end='yy')   #hello worldyy
```

หากเราต้องการแสดงค่าที่ตัวแปรอ้างอิงก็สามารถทำได้โดยสั่งพิมพ์แล้วใช้ชื่อตัวแปร แต่ไม่ต้องอยู่ในเครื่องหมายคำพูด ตัวอย่างเช่น

```
name = 'Lufy'
print('hello',name,'. This is Monday.')
#hello Lufy . This is Monday
```

เราสามารถใช้การจัดรูปแบบการแสดงผลได้ ซึ่งใช้งานได้คล้ายกับภาษาซี เราจะใช้ % แทรกในข้อความเพื่อที่จะนำค่าของตัวแปรมาใส่ในรูปแบบที่กำหนด โดยค่าที่จะนำมาใส่จะเรียงจากซ้ายไปขวา ตัวอย่างเช่น

```
print('Today %s is %d years old ')%(name, age)
```

สำหรับรูปแบบการแสดงผลดังตัวอย่าง %s คือแสดงข้อความ %d คือแสดงจำนวนเต็ม ส่วนรูปแบบอื่นที่ใช้บ่อยแสดงดังตารางที่ 1.9

ตารางที่ 1.9: การจัดรูปแบบการแสดงผลข้อความ

modifier	ตัวอย่าง	คำอธิบาย
number	%20d	ใส่ค่าในช่องที่กว้าง 20 ตัวอักษร
-	%-20d	ใส่ค่าในช่องที่กว้าง 20 ตัวอักษรโดยชิดซ้าย
+	%+20d	ใส่ค่าในช่องที่กว้าง 20 ตัวอักษรโดยชิดขวา
.	%20.2f	ใส่ค่าในช่องที่กว้าง 20 ตัวอักษรโดยมีเลขหลังจุดทศนิยม 2 ตัวแหน่ง

สำหรับข้อมูลตัวเลขยังสามารถจัดรูปแบบเพิ่มเติมได้อีกโดยระบุขอบเขตการแสดงผลดังตารางที่ 1.10

ตารางที่ 1.10: เพิ่มเติมการจัดรูปแบบการแสดงผลตัวเลข

modifier	การใช้งาน
d,i	integer
f	floating point as m.ddddd
e	floating point as m.dddddde+/-xx
c	single character
s	string

1.3 โครงสร้างควบคุม

ในการเขียนโปรแกรมเราจะใช้โครงสร้างในการควบคุมการทำงานของโปรแกรม 3 ประเภทด้วยกันได้แก่ โครงสร้างควบคุมแบบลำดับ โครงสร้างควบคุมแบบเงื่อนไข และโครงสร้างควบคุมแบบทำซ้ำ

1.3.1 โครงสร้างควบคุมแบบลำดับ

โครงสร้างควบคุมแบบลำดับเป็นลำดับการทำงานของคำสั่งจากบรรทัดบนลงมาบรรทัดล่างทีละบรรทัดต่อเนื่องกันเป็นโครงสร้างการทำงานแบบที่ง่ายที่สุดไม่ซับซ้อน เพียงแค่เราเรียงลำดับคำสั่งให้ถูกต้องว่าอะไรทำก่อนอะไรเท่านั้นเอง ตัวอย่างเช่น

```
print('Calculate circle area')
radius = float(input('Please input radius'))
area = 3.14*radius*radius
print('Area of circle is',area)
```

ในการคำนวณพื้นที่วงกลม เราต้องรู้ความยาวรัศมีก่อน ดังนั้นคำสั่งรับข้อมูลรัศมีวงกลมต้องมาก่อนคำสั่งคำนวณพื้นที่วงกลม และสุดท้ายคำสั่งแสดงพื้นที่วงกลมต้องอยู่ถัดจากคำสั่งคำนวณพื้นที่วงกลม

1.3.2 โครงสร้างควบคุมแบบเงื่อนไข

เมื่อการทำงานมีการตัดสินใจว่าหากเป็นไปตามเงื่อนไขจะทำอย่างหนึ่งแล้วถ้าไม่เป็นตามเงื่อนไขจะทำอีกอย่างหนึ่ง เราจะใช้โครงสร้างควบคุมแบบมีเงื่อนไขเข้ามาช่วยจัดการ โดยคำสั่งที่ใช้ในการตัดสินใจคือ if

โครงสร้างของคำสั่ง if จะตามด้วยเงื่อนไขที่ต้องการตรวจสอบแล้วจบด้วยโคลอน (:) จากนั้นบรรทัดต่อมาจะเป็นกลุ่มของคำสั่งที่เมื่อเงื่อนไขเป็นจริงจะให้ทำงาน โดยกลุ่มของคำสั่งนี้จะต้องมีระยะที่เยื้องหรือย่อหน้าที่เท่ากัน ตัวอย่างเช่น

```
if x > 5 :  #if x is more than 5 do
    x = x*2
    print(x)
```

เมื่อไม่ตรงเงื่อนไขแล้วต้องการให้มีการทำงานบางอย่าง สามารถทำได้โดยการเพิ่มคำสั่ง else: ให้อยู่ในระดับเดียวกับคำสั่ง if โดย if 1 คำสั่งจะมี else ได้ไม่เกิน 1 คำสั่ง ตัวอย่างเช่น

```
if x > 5 :  #if x is more than 5 do
    x = x*2
    print(x)
else:       #otherwise
    x = x*3
```

หากต้องการเพิ่มเงื่อนไขเปรียบเทียบเพิ่มสามารถทำได้ โดยใช้คำสั่ง elif แล้วตามด้วยเงื่อนไขและจบด้วยโคลอน ตัวอย่างเช่น

```
if x > 5 :  #if x is more than 5 do
    x = x*2
    print(x)
elif x==0:  #if x is equal to 0 do
    print('Zero')
```

```
else:               #otherwise
    x = x*3
```

นอกจากนี้หากต้องการการเปรียบเทียบเงื่อนไขภายใต้เงื่อนไขอีกที เราสามารถใช้คำสั่ง if ซ้อนภายในส่วนที่เป็นจริงหรือเป็นเท็จได้ ตัวอย่างเช่น

```
if x > 5 :  #if x is more than 5 do
    x = x*2
    if(x>10):
        x=x*4
    print(x)
else:       #otherwise
    x = x*3
```

1.3.3 โครงสร้างควบคุมแบบทำซ้ำ

ในการทำงานเมื่อต้องการทำอะไรบางอย่างที่ซ้ำเดิม หรือทำหลายครั้ง วิธีง่ายที่สุดก็จะเป็นการ copy code ทำซ้ำเท่ากับจำนวนครั้งที่ต้องการทำ แต่ว่าหากมีจำนวนครั้งที่มาก เช่น 10 ครั้ง หรือไม่รู้จำนวนครั้งที่แน่นอน ทำให้การ copy code นั้นไม่ยืดหยุ่น ในการทำงานจึงมีโครงสร้างควบคุมแบบทำซ้ำ ในการทำซ้ำนี้มี 2 คำสั่งคือ while และ for [1]

while

ส่วนใหญ่ใช้เมื่อไม่รู้จำนวนรอบที่แน่นอน โครงสร้างของ while คล้ายกับ if จะทำเมื่อเงื่อนไขที่กำหนดมาเป็นจริง แล้วจะทำกลุ่มคำสั่งภายใต้คำสั่ง while เมื่อทำเสร็จแล้วจะกลับมาตรวจสอบเงื่อนไขอีกครั้ง หากเป็นจริงจะทำต่อและจะเลิกทำเมื่อเงื่อนไขเป็นเท็จ ตัวอย่างเช่น

```
count=0
x=10
while x>5:
    count = count+1
    print(count, end=' ')
    x=x—1
#1 2 3 4 5
```

for

ส่วนใหญ่ใช้เมื่อรู้จำนวนรอบที่แน่นอน for มักจะทำงานคู่กับลำดับหรือฟังก์ชัน range() โดยฟังก์ชัน range จะคืนค่าลำดับที่เปลี่ยนแปลงค่าไม่ได้ของตัวเลขจากช่วงที่กำหนดให้เช่น ลำดับของเลขที่ติดกัน 0 – 3 หรือลำดับแบบช่วงห่างเท่ากันเช่น 2, 4, 6, 8 เป็นต้น range มี 3 รูปแบบการใช้งาน

- **range(stop)** เป็นการสร้างลำดับ 0, 1, 2, ..., stop-1
- **range(start, stop)** เป็นการสร้างลำดับ start, start+1, ..., stop-1
- **range(start, stop, step)** เป็นการสร้างลำดับที่เปลี่ยนแปลงจากจากค่าเดิมด้วยค่า step นั่นคือ start, start+step, start+2*step, start+3*step, ... แต่ไม่เกิน stop

ทั้งนี้ start, stop, step ต้องเป็นเลขจำนวนเต็มเท่านั้น

โครงสร้างของ for จะตามด้วยตัวแปรที่จะนำเอาค่าจากลำดับมาใช้ทีละตัวในแต่ละรอบ จากนั้นตามด้วยคำสั่ง in และ ตามด้วยลำดับที่ต้องการ ตัวอย่างเช่น

```
count = 0
for i in range(5):    # i = 0, 1, 2, 3, 4
  count = count+1
```

หรือจะสร้างลำดับเพื่อวนรอบเองก็ได้ ตัวอย่างเช่น

```
count = 0
for i in 'Thailand','Malaysia','China','Japan':
    print(i)
    #Thailand Malaysia China Japan
```

เราสามารถนำเอาโครงสร้างควบคุมแต่ละแบบมาใช้งานต่อเนื่องหรือผสมกันได้ ขึ้นอยู่กับปัญหาที่เราต้องการแก้

1.4 การนิยามฟังก์ชัน

เมื่อโปรแกรมของเราใหญ่ขึ้น การเขียนคำสั่งเพื่อทำงานอะไรบางอย่าง อาจจะมีการเรียกใช้งานกลุ่มคำสั่งซ้ำเดิม เราจึงมีการสร้างฟังก์ชันมาใช้งาน

1.4.1 ฟังก์ชันคืออะไร

ฟังก์ชันคือกลุ่มของคำสั่งที่มีการนิยามชื่อที่ใช้สำหรับอ้างอิงแทนกลุ่มคำสั่งนั้นเพื่อให้สามารถเรียกใช้งานได้ โดยกลุ่มของคำสั่งที่ประกอบกันเป็นฟังก์ชันจะทำงานอย่างใดอย่างหนึ่งโดยเฉพาะ

1.4.2 การสร้างฟังก์ชัน

มีรูปแบบดังนี้

```
def functionname():
    statements
```

ชื่อของฟังก์ชันเป็นไปตามหลักการตั้งชื่อของ Python โดยต้องมีเครื่องหมายวงเล็บเปิดและวงเล็บปิดต่อท้าย กลุ่มของคำสั่งหรือบล็อก เริ่มต้นด้วยการใช้เครื่องหมาย : ร่วมกับการเยื้องของบล็อก และสิ้นสุดบล็อกเมื่อมีการยุติการใช้ระยะเยื้องนั้น ทั้งนี้ระยะที่เยื้องเข้ามาหรือย่อหน้าอาจใช้การเคาะ spacebar อย่างน้อย 1 ครั้ง โดยบล็อกเดียวกันระยะเยื้องต้องเท่ากัน ตัวอย่างการสร้างฟังก์ชัน

```
def printHello():
    print('——Hello from python')
    print('——Nice to meet you')
```

บรรทัดแรกเป็นจุดเริ่มต้นการสร้างฟังก์ชัน(def) ฟังก์ชันชื่อ printHello สองบรรทัดถัดมาเป็นบล็อกของฟังก์ชัน ซึ่งประกอบไปด้วยฟังก์ชัน print() สองครั้งด้วยข้อความที่ต่างกัน

1.4.3 การเรียกใช้งานฟังก์ชัน

คำสั่งในฟังก์ชันจะไม่ทำงานทันทีเมื่อเรารันโปรแกรม แต่คำสั่งในฟังก์ชันจะทำงานก็ต่อเมื่อมีการเรียกใช้ฟังก์ชันนั้นจากส่วนอื่นของโปรแกรม การเรียกใช้งานฟังก์ชันทำได้โดยการอ้างชื่อฟังก์ชัน เมื่อฟังก์ชันถูกเรียกใช้มีผลให้บล็อกของฟังก์ชันทำงาน 1 ครั้ง

Parameter ของฟังก์ชัน

ฟังก์ชันสามารถรับข้อมูลเข้าไปเพื่อประมวลผลเพื่อให้ได้ผลลัพธ์ที่ต้องการได้ ในการรันโปรแกรมแต่ละครั้งเมื่อข้อมูลเข้าเปลี่ยนค่าไปจะทำให้ผลลัพธ์มีค่าเปลี่ยนแปลงไปในลักษณะที่สอดคล้องกัน ข้อมูลเข้าที่ส่งให้กับฟังก์ชันเรียกว่าพารามิเตอร์(Parameter) มีรูปแบบดังนี้

```
def functionname(parameter1, parameter2, ):
    ...
```

ฟังก์ชันแต่ละฟังก์ชัน อาจจะมีจำนวนพารามิเตอร์ที่ต่างกัน การเรียกใช้ฟังก์ชันที่มีพารามิเตอร์ จะต้องมีการส่งค่าอาร์กิวเมนต์(Argument) เพื่อไปเป็นค่าพารามิเตอร์ของฟังก์ชัน อาร์กิวเมนต์ หมายถึงค่าที่ส่งผ่านจุดที่เรียกใช้คำสั่งหรือฟังก์ชันเข้าสู่การทำงานในคำสั่งหรือฟังก์ชันนั้นๆ ตัวอย่างเช่น

```
def printHello(name):
    print('--Hello ', name)
    print('--Nice to meet you')

def bmiCal(name, weight, height):
    bmi = weight / height ** 2
    print('--Hello', name)
    print('Your BMI is',bmi)

print('BMI Calculation')
yourname = input('Enter your name')
yourweight = float(input('Enter your weight(kg)'))
yourheight = float(input('Enter your height(m)'))
bmiCal(yourname,yourweight,yourheight)
```

1.4.4 การคืนค่าจากฟังก์ชัน

ในการเรียกใช้งานฟังก์ชันบางครั้งเราต้องการให้ฟังก์ชันคืนค่าที่คำนวณได้มาให้ เพื่อเอาค่าที่คำนวณได้ไปใช้งานภายหลังสามารถทำได้โดยใช้รูปแบบดังนี้

```
return expression
```

ตัวอย่างเช่น

```
def bmi_cal(weight, height):
    bmi = weight / height ** 2
    return bmi
```

1.4.5 ขอบเขตของตัวแปร

ขอบเขตของตัวแปร(Scope of variable) หมายถึงขอบเขตในการอ้างอิงตัวแปรที่นิยามขึ้นในโปรแกรม ขอบเขตของตัวแปรมี 2 ลักษณะได้แก่ตัวแปรเฉพาะที่(Local variable) และตัวแปรส่วนกลาง(Global variable) ตัวแปรทุกตัวมีขอบเขตอยู่ในบล็อก เริ่มต้นจากตำแหน่งที่แปรนั้นถูกนิยามขึ้น ตัวแปรเฉพาะที่เป็นตัวแปรที่นิยามอยู่ในบล็อกของฟังก์ชันนับจากตำแหน่งที่เริ่มต้นนิยามและจะถูกอ้างอิงได้ภายในบล็อกของฟังก์ชันดังกล่าวเท่านั้น ไม่สามารถอ้างอิง

ในบล็อกอื่น ตัวแปรส่วนกลางเป็นตัวแปรที่นิยามอยู่ในบล็อกของโปรแกรมหลัก และสามารถอ่างอิงในบล็อกของทุกฟังก์ชัน ตัวอย่างเช่น

```
y = 5
print(y)
print(x)
x = 10
```

จากตัวอย่างข้างต้นตัวแปร y ถูกสร้างในบรรทัดแรกหลังจากนั้นมีการเรียกใช้งานในบรรทัดที่สอง ส่วนในบรรทัดที่สามไม่สามารถทำงานได้เนื่องจากมีการเรียกใช้งาน x แต่ยังไม่เคยประกาศใช้งานมาก่อน

พิจารณาตัวอย่างต่อไปนี้

```
def func():
    y = 5
    print(x,y)

x = 8
func()
```

ตัวอย่างนี้มีการประกาศฟังก์ชัน func แต่ยังไม่ได้ใช้งาน ต่อมามีการประกาศค่า x จากนั้นมีการเรียกใช้งานฟังก์ชัน func ซึ่งภายในมี y เป็น Local variable และได้มีการ สั่ง print ซึ่งจะนำค่า x ซึ่งเป็น Global variable และ y ที่เป็น Local variable มาแสดง

1.4.6 โมดูลและเนมสเปซ

โมดูล(Module) หมายถึงไฟล์โปรแกรมไพธอนที่รวบรวมฟังก์ชัน คลาส ตัวแปรที่มีความสัมพันธ์กันให้เป็นหมวดหมู่เพื่อให้สะดวกต่อการอ้างอิงด้วยโปรแกรมอื่น โมดูลอาจประกอบด้วยโปรแกรมหลักที่สามารถประมวลผลด้วยตัวเองได้ ดังนั้นโปรแกรมที่เขียนขึ้น(*.py) ทุกโปรแกรมถือเป็นโมดูล โดยมีชื่อไฟล์โปรแกรมที่ไม่รวมส่วนขยาย .py เป็นชื่อโมดูล โปรแกรมสามารถอ้างถึงโมดูลโดยใช้คำสั่ง import พร้อมระบุชื่อโมดูลที่ต้องการ

```
import module1, module2, .
```

การเรียกใช้ฟังก์ชันในโมดูลมีรูปแบบดังนี้ modulename.functionname() สมมติกำหนดให้มีไฟล์ชื่อ hello.py ที่มีคำสั่งด้านในดังนี้

```
def printHello(name):
    print('Hello',name)
```

โปรแกรมอื่นที่มีการอ้างถึงโมดูลชื่อ hello เพื่อเรียก printHello สามารถทำได้ดังนี้

```
import hello
hello.printHello('Jakarin')
```

นอกจากการเรียกใช้โมดูลที่เขียนขึ้นใหม่แล้ว เรายังสามารถเรียกใช้โมดูลในไลบรารีมาตรฐานได้ในทำนองเดียวกัน เพื่อช่วยให้เขียนโปรแกรมได้รวดเร็วขึ้น ตัวอย่างโมดูลที่มีอยู่แล้วเช่น โมดูล math ซึ่งได้นิยามฟังก์ชันทางคณิตศาสตร์ต่างๆ ให้สามารถเรียกใช้งานได้โดยไม่ต้องเขียนเอง

```
import math

print('pi=', math.pi)
print('e=', math.e)
x=3.14
print('floor of ',x, ' is ',math.floor(x))
print('ceiling of ',x, ' is ',math.ceil(x))
print('factorial of 5 is ',math.factorial(5))
content = dir(math)
print(content)
```

เราสามารถอ้างถึงโมดูลโดยกำหนดเฉพาะฟังก์ชันที่ต้องการใช้งานได้ในรูปแบบดังนี้ from ชื่อโมดูล import ชื่อฟังก์ชัน1, ชื่อฟังก์ชัน2, ...

```
from math import pi, e

print('pi=', pi)
print('e=', e)
```

1.5 การนิยามคลาส

เราได้กล่าวไปก่อนหน้านี้ว่าภาษาไพธอนเป็นภาษาเชิงวัตถุ ในหัวข้อนี้เราจะมาลองสร้าง class กัน การสร้าง class เป็นความสามารถของภาษาเชิงวัตถุในการจัดการกับข้อมูลเพื่อการแก้ปัญหา ก่อนหน้านี้เราใช้ชนิดข้อมูลมาหลายชนิดและได้ลองอ้างอิงข้อมูลและมีการดำเนินการบางอย่างกับข้อมูล จะเห็นได้ว่ามี**ข้อมูล**และ**การดำเนินการ** พอจะสรุปได้ง่ายๆว่า เราจะจัดการอะไรบางอย่างจะมีสองอย่างที่เกี่ยวข้อคือ อ้างอิงข้อมูลหรือเก็บข้อมูลอะไร และ มีการดำเนินการอะไรได้บ้าง(method นั่นเอง)

เพื่อให้มองข้อมูลว่าหน้าตาตอนเก็บเป็นอย่างไรและมันสามารถทำอะไรได้บ้าง โดยเราจะสร้าง class เราจะยกตัวอย่างด้วย class จุดในระนาบ 2 มิติโดยใช้ชื่อ class ว่า Point โดยจุดนั้นประกอบด้วยข้อมูล 2 ส่วนคือพิกัดในแกน

x และพิกัดในแกน y ทั้งนี้ class ที่สร้างเปรียบเสมือนพิมพ์เขียวยังเอาไปใช้งานไม่ได้ เราต้องสร้างวัตถุ(Object หรือ Instance) ขึ้นมาก่อนเพื่อใช้งานต่อไป

ในภาษาไพธอนนั้นการสร้าง Class ใหม่ทำได้โดยให้ชื่อและกำหนด Method ที่เกี่ยวข้อง คล้ายกับการนิยามฟังก์ชัน ตัวอย่างเช่น

```
class Point:
    #the methods
```

ในการสร้าง Method นั้น Method แรกที่ควรสร้างคือ Constructor โดย Constructor เป็น method ที่ถูกเรียกเมื่อ object ถูกสร้างจาก class ในการกำหนดค่าต่างๆ ในการสร้าง object ของ class Point นั้นเราต้องการของสองอย่างได้แก่ x และ y ในภาษาไพธอนนั้น constructor จะถูกเรียกด้วย **__init__** (มี underscore 2 อัน)

```
class Fraction:
    def __init__(self,x_input,y_input):
        self.x = x_input
        self.y = y_input
```

สังเกตว่ามี Parameter 3 ตัวได้แก่ $(self, x\_input, y\_input)$ self เป็น Parameter พิเศษที่จะถูกใช้เสมอเป็นตัวอ้างอิงกลับให้กับ Object ซึ่งมันจะต้องเป็น Parameter ตัวแรกเสมอ อย่างไรก็ตามมันไม่ต้องกำหนดค่าให้เวลาเรียก ส่วน $self.x$ ใน Constructor นั้นจะเป็นการนิยาม Point object ที่เก็บข้อมูลภายในที่ชื่อว่า x เช่นเดียวกับ $self.y$ ก็จะเป็นข้อมูลภายในเช่นกัน ค่าของ Parameter จะถูกกำหนดให้กับ x และ y เป็นค่าเริ่มต้นเมื่อสร้าง object

ในการสร้าง Instance ของ Point class(หรือสร้าง object) เราจะต้องเรียกด้วย Constructor ทำได้โดยใช้ชื่อของ Class และส่งค่าที่จำเป็นให้ (ข้อสังเกต เราจะไม่เรียก **__init__** โดยตรง) ตัวอย่างเช่น

```
class Point:
    def __init__(self,x_input,y_input):
        self.x = x_input
        self.y = y_input

myPoint = Point(3,7)
```

จากตัวอย่างเราจะได้ตัวแปร myPoint ที่เป็น object ของ Point (มองง่ายๆว่า เป็นตัวแปรที่มีชนิดข้อมูลเป็น Point) ต่อไปเราจะทำการสร้าง Method ต่างๆ ให้กับ Class เริ่มต้นเราจะสร้าง Method show สำหรับแสดงค่าของ Point object ก่อน การเรียกใช้งาน method ของ object จะเรียกใช้โดย object.method() ตัวอย่างเช่น

```
class Point:
    def __init__(self,x_input,y_input):
        self.x = x_input
```

```
        self.y = y_input
    def show(self):
        print('(',self.x,',',self.y,')')

myPoint = Point(3,7)
myPoint.show()
```

หาก method ต้องการรับค่าคืนค่าก็เขียนเหมือนตอนเขียนฟังก์ชัน สมมติว่าเราต้องการให้สามารถหาระยะห่างระหว่างจุดสองจุดได้ ก็เพิ่ม Method dist ซึ่งสามารถทำได้เช่น

```
class Point:
    def __init__(self,x_input,y_input):
        self.x = x_input
        self.y = y_input
    def show(self):
        print('(',self.x,',',self.y,')')
    def dist(self,op):
        return ((self.x-op.x)**2+(self.y-op.y)**2)**(1/2)

myPoint = Point(3,7)
myPoint2 = Point(4,2)
print(myPoint.dist(myPoint2))
```

1.6 แบบฝึกหัดท้ายบท

1. ให้เขียนโปรแกรมภาษา python เพื่อรับอุณหภูมิเป็นองศาฟาเรนไฮต์และแปลงเป็นองศาเซลเซียส
2. ให้เขียนโปรแกรมภาษา python เพื่อรับอุณหภูมิเป็นองศาเซลเซียสและแปลงเป็นองศาฟาเรนไฮต์
3. ให้เขียนโปรแกรมภาษา python เพื่อหาจุดตัดของเส้นตรงสองเส้น ที่มีสมการอยู่ในรูป
4. ให้เขียนโปรแกรมภาษา python ที่เมื่อรับเลขจำนวนเต็มมา 4 ตัวเลข ให้แสดงผลว่ามีตัวเลขที่เป็นเลขคี่กี่ตัวเลข $y = m_1x + b_1$ และ $y = m_2x + b_2$ โดยกำหนดให้ $m_1, b_1, m_2,$ และ b_2 เป็นจำนวนจริง และ $m_1 \neq m_2$
5. ให้สร้าง class เศษส่วนและสร้าง method add สำหรับบวกเศษส่วน

บทที่ 2

โครงสร้างข้อมูลพื้นฐาน

2.1 แบบชนิดข้อมูลนามธรรม

แบบชนิดข้อมูลนามธรรม(Abstract Data Type) เป็นการอธิบายเชิงตรรกะหรือการระบุส่วนประกอบต่างๆ ของ**ข้อมูล**และ**การดำเนินการ**ต่างๆ ที่ทำได้ แบบชนิดข้อมูลนามธรรมไม่ได้ระบุว่าจะต้องสร้างชนิดข้อมูลอย่างไร ทำให้สามารถสร้างได้หลากหลายแบบขึ้นกับผู้สร้าง ทั้งนี้ในการสร้างนั้นต้องเป็นไปตามที่แบบชนิดข้อมูลนามธรรมแจ้งไว้

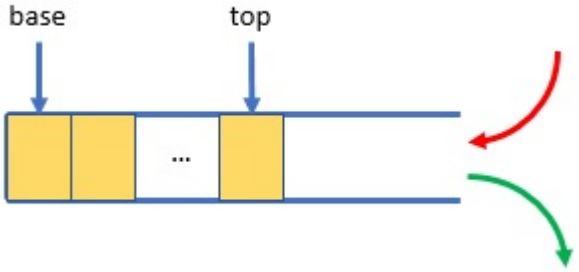
เราจะเริ่มต้นด้วยโครงสร้างข้อมูลพื้นฐาน 4 ชนิดได้แก่ Stack, Queue, Deque และ List กลุ่มของโครงสร้างข้อมูลจำพวกนี้ถูกเรียกว่าโครงสร้างข้อมูลเชิงเส้น(Linear data structures) เนื่องจากข้อมูลแต่ละตัวในโครงสร้างข้อมูลจะถูกเรียงลำดับ นั่นคือเมื่อข้อมูลถูกเพิ่มหรือลบ มันจะอยู่ในตำแหน่งที่สัมพันธ์กับข้อมูลก่อนหน้าและข้อมูลที่มาถัดจากมันอย่างไร

โครงสร้างข้อมูลเชิงเส้นนี้สามารถมองได้ง่ายๆ ว่าเป็นโครงสร้างที่มีจุดปลาย 2 จุด ในบางครั้งเราจะอ้างอิงจุดปลายเหล่านี้ด้วย left และ right, front และ rear หรือ top และ bottom อย่างไรก็ตามสิ่งที่ทำให้แต่ละโครงสร้างข้อมูลเชิงเส้นต่างจากกันคือ ทิศทางที่ข้อมูลถูกเพิ่มหรือลบออก บางโครงสร้างข้อมูลเพิ่มข้อมูลเข้าและลบข้อมูลออกทางจุดปลายเดียวกัน บางโครงสร้างเพิ่มข้อมูลเข้าและลบข้อมูลออกคนละจุดปลายกัน

ทั้งนี้ในบทนี้จะกล่าวถึง Abstract data type หลายจุดมาก เพื่อความง่ายในการเข้าใจอยากให้มองว่าเป็นการออกแบบว่าจะเก็บข้อมูลแบบใดและมีการดำเนินการอะไรที่ทำได้บ้าง ซึ่งที่เรียกว่าเป็นการออกแบบเพราะว่าไม่ได้ระบุว่าต้องสร้างอย่างไร อยู่ที่คนจะนำไปใช้จะสร้างอย่างไร เขียนโปรแกรมอย่างไร เพียงแค่เรากำหนดลักษณะให้

2.2 Stack

เราจะเริ่มต้นด้วย Abstract data type ตัวแรกที่มีคุณสมบัติคือ ข้อมูลที่ถูกเก็บไว้ล่าสุดจะถูกนำออกมาใช้ก่อน Abstract data type นี้คือ Stack ซึ่งคือกลุ่มข้อมูลที่เรียงลำดับโดยข้อมูลที่เพิ่มเข้ามาใหม่และข้อมูลที่จะถูกลบออก



รูปที่ 2.1: ตัวอย่างการทำงานของ Stack

เกิดขึ้นที่จุดปลายเดียวกัน จุดปลายนี้โดยทั่วไปเรียกว่า top ส่วนจุดปลายฝั่งตรงข้ามกับ top เรียกว่า base โดยที่ base นั้นมีความสำคัญตรงที่ข้อมูลยิ่งอยู่ใกล้ base แสดงว่ายิ่งถูกเก็บใน stack นาน ส่วนข้อมูลใหม่จะอยู่ใกล้ top การทำงานของ stack นั้นข้อมูลที่ถูกเพิ่มเข้ามาล่าสุดจะถูกลบออกก่อน การเรียงลำดับแบบนี้เรียกว่า LIFO: Last-In First-Out ดังรูปที่ 2.1 ตัวอย่างที่เห็นในชีวิตประจำวัน เช่น แก้วในในร้านสะดวกซื้อ จานสลัดในร้าน sizzler เป็นต้น

2.2.1 Stack:abstract data type

นิยามด้วยโครงสร้างและการดำเนินการต่อไปนี้ สมมติว่า s แทน Stack และ top อยู่ทางขวามือ bottom อยู่ทางซ้ายมือ การดำเนินการของ Stack มีดังตารางที่ 2.1

- `Stack()` เป็นการสร้าง Stack อันใหม่ที่ว่าง ไม่ต้องการข้อมูลเข้าและคืนค่าเป็น Stack ว่าง
- `push(item)` เพิ่มข้อมูลใหม่(item) ไปไว้ที่ top ของ Stack ต้องการข้อมูลเข้า(item) และไม่คืนค่า
- `pop()` นำข้อมูลที่ตำแหน่ง top ลบออกจาก Stack ไม่ต้องการข้อมูลเข้าและคืนค่าข้อมูล(Stack เปลี่ยนแปลง)
- `peak()` คืนค่าข้อมูลตัวที่ top แต่ไม่ลบข้อมูล ไม่ต้องการข้อมูลเข้าและคืนค่าข้อมูล(Stack ไม่เปลี่ยนแปลง)
- `is_empty()` ทดสอบว่า Stack ว่างหรือมีข้อมูลไหม ไม่ต้องการข้อมูลเข้าและคืนค่าข้อมูลเป็นจริงหรือเท็จ
- `size()` คืนค่าจำนวนของข้อมูลที่อยู่ใน Stack ไม่ต้องการข้อมูลเข้าและคืนค่าข้อมูลจำนวนเต็ม

2.2.2 Stack implementation

เราได้นิยาม Stack ไปแล้วว่ามีส่วนประกอบอะไรบ้าง ต่อไปเราจะมาลองสร้าง Stack ไว้ใช้งานกัน ทั้งนี้เมื่อเรานำเอา Abstract data type มาสร้างหรือเขียนโปรแกรม เราจะเรียกสิ่งที่สร้างออกมานั้นว่า **Data structure** เนื่องจากภาษาไพธอนเป็น Objected-oriented programming language ดังนั้นเราจะสร้าง Stack เป็น class และการดำเนินการต่างๆ จะเขียนด้วย Method(function) ในภาษาไพธอนมี List ซึ่งเป็นที่เก็บกลุ่มข้อมูลและมีการดำเนินการพื้นฐานมาให้ เราจะใช้ List ในการสร้าง Stack

ตัวอย่างถ้าเรามี List ที่ชื่อ list = [2, 5, 3, 6, 7, 4] เราเพียงแค่กำหนดว่าปลายด้านไหนของ list ที่เราจะเอามาเป็น top ของ Stack และด้านไหนเป็น base ของ Stack เมื่อเรากำหนดได้แล้ว การดำเนินการต่างๆ จะถูกสร้างโดยใช้ function ที่มีให้จาก List เช่น append(), pop() เป็นต้น ต่อไปเป็นตัวอย่างการสร้าง class Stack

```
class Stack:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def peek(self):
        return self.items[len(self.items)-1]
    def size(self):
        return len(self.items)
```

เมื่อเราสั่ง run(F5) จะยังไม่มีอะไรเกิดขึ้น เนื่องจาก sourcecode ข้างต้นเป็นการนิยาม class ซึ่งเหมือนกับพิมพ์เขียว เราจะต้องสร้าง Object มาก่อนจึงจะใช้งานได้ โดยการเรียก Constructor ด้วย Method ที่มีชื่อเดียวกับ class ดังตัวอย่าง

```
s = Stack()
```

จากนั้นลองเพิ่ม code ตามตัวอย่างการเรียกใช้งานก่อนหน้า

```
s = Stack()
print(s.is_empty())
s.push(4)
```

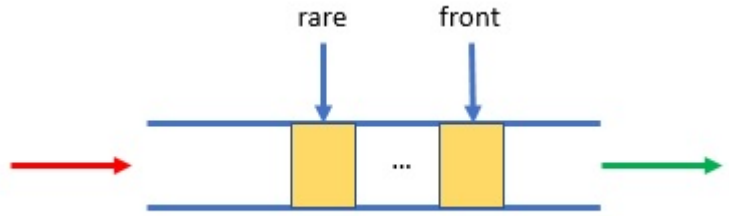
ตารางที่ 2.1: ตัวอย่างการทำงานของ Stack

การดำเนินการของ Stack	สิ่งที่อยู่ใน Stack	ค่าที่คืน
s.is_empty( )	[ ]	True
s.push(4)	[4]	
s.push('dog')	[4,'dog']	
s.peek()	[4,'dog']	'dog'
s.push(True)	[4,'dog',True]	
s.size()	[4,'dog',True]	3
s.is_empty()	[4,'dog',True]	False
s.push(8.4)	[4,'dog',True,8.4]	
s.pop()	[4,'dog',True]	8.4
s.pop()	[4,'dog']	True
s.size()	[4,'dog']	2

```
s.push('dog')
print(s.peek())
s.push(True)
print(s.size())
print(s.is_empty())
s.push(8.4)
print(s.pop())
print(s.pop())
print(s.size())
```

นอกจากนี้เราสามารถเลือกที่จะสร้าง Stack จาก List โดยที่ top เป็นส่วนเริ่มต้นแทนที่จะเป็นส่วนปลายได้ ในกรณีนี้ เราต้องแก้ไขเนื่องจาก pop และ append ใช้งานไม่ได้ เราต้องจัดการบางอย่างเพื่อที่จะใช้ index ที่ 0 ดังนั้นเราจะใช้ pop และ insert มาช่วย ตัวอย่างเช่น

```
class Stack:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def push(self, item):
        self.items.insert(0,item)
```



รูปที่ 2.2: ตัวอย่างการทำงานของ Queue

```
def pop(self):
    return self.items.pop(0)
def peek(self):
    return self.items[0]
def size(self):
    return len(self.items)
```

เพื่อความเข้าใจมากขึ้นให้ลองสร้าง method popAll() ที่ทำการ pop ของทุกตัวออกจาก Stack จนหมด

2.3 Queue

Queue คือกลุ่มข้อมูลที่เรียงลำดับโดยข้อมูลที่เพิ่มเข้ามาใหม่จะไปปรากฏอยู่ส่วนท้ายที่เรียกว่า rear และข้อมูลที่มีอยู่ที่จะถูกนำออกเกิดขึ้นที่จุดปลายด้านตรงข้ามที่เรียกว่า front ข้อมูลที่เพิ่งถูกเพิ่มเข้าไปใน Queue จะต้องรอที่ส่วนท้ายของกลุ่มข้อมูล ข้อมูลที่อยู่ในกลุ่มข้อมูลนานที่สุดจะอยู่ส่วน front ลำดับเช่นนี้เรียกว่า FIFO: first-in first-out หรือ first-come first-served ตัวอย่างที่เห็นได้ทั่วไปของ Queue คือการเข้าคิวดูภาพยนต์ เข้าคิวจ่ายเงิน ซึ่งจะไม่มีการลัดคิวเกิดขึ้น ดังรูปตัวอย่างที่ 2.2

2.3.1 Queue abstract data type

นิยามด้วยโครงสร้างและการดำเนินการต่อไปนี้ Queue เป็นโครงสร้างที่กลุ่มข้อมูลถูกเรียงลำดับโดยข้อมูลที่ถูกเพิ่มจะไปต่อส่วนท้ายที่เรียกว่า rear และข้อมูลที่จะเอาออกจะอยู่ส่วนปลายอีกด้านที่เรียกว่า front สมมติว่า q แทน Queue การดำเนินการของ Queue มีดังตารางที่ 2.2

- `Queue()` เป็นการสร้าง Queue อันใหม่ที่ว่าง ไม่ต้องการ parameter และคืนค่าเป็น Queue ว่าง

- `enqueue(item)` เพิ่มข้อมูลใหม่ให้ส่วน rear ของ Queue ต้องการข้อมูลและไม่คืนค่า
- `dequeue()` นำข้อมูลที่ front ออกจาก Queue ไม่ต้องการ parameter คืนค่าข้อมูล(Queue มีการเปลี่ยนแปลง)
- `is_empty()` ทดสอบว่า Queue ว่างไหม ไม่ต้องการ parameter คืนค่าข้อมูลเป็นจริงหรือเท็จ
- `size()` คืนค่าจำนวนข้อมูลของ Queue ไม่ต้องการ parameter คืนค่าข้อมูลจำนวนเต็ม

2.3.2 Queue implementation

เราก็จะสร้าง class ใหม่สำหรับการ implement Queue เช่นเดียวกับ Stack เราจะใช้ List ในการสร้าง Queue ทั้งนี้เราต้องตัดสินใจว่าเราจุดปลายของ List ส่วนไหนที่เราจะเอามาเป็น rear และส่วนไหนมาเป็น front สมมติว่าเราจะให้ส่วน rear เป็น index 0 ใน List ดังนั้นเราจะใช้ insert ที่ตำแหน่ง 0 เวลาที่เพิ่มข้อมูลใหม่ และใช้ pop เวลาเอาข้อมูลออก

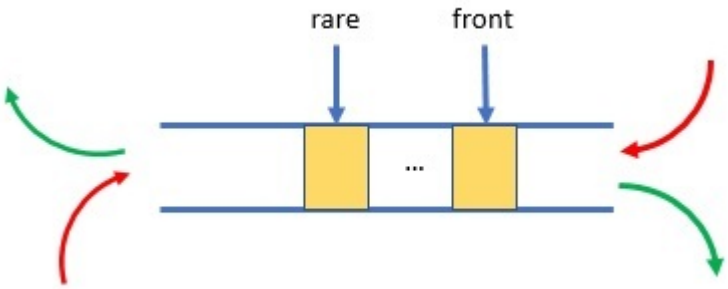
```
class Queue:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def enqueue(self, item):
        self.items.insert(0,item)
    def dequeue(self):
        return self.items.pop()
    def size(self):
        return len(self.items)
```

ลองสร้างไฟล์ใหม่ แล้วทดสอบด้วย

```
q = Queue()
q.enqueue('hello')
q.enqueue('dog')
q.enqueue(3)
q.dequeue()
```

ตารางที่ 2.2: ตัวอย่างการทำงานของ Queue

การดำเนินการของ Queue	สิ่งที่อยู่ใน Queue	ค่าที่คืน
`q.is_empty()`	[]	True
`q.enqueue(4)`	[4]	
`q.enqueue('dog')`	['dog',4]	
`q.enqueue(True)`	[True,'dog',4]	
`q.size()`	[True,'dog',4]	3
`q.is_empty()`	[True,'dog',4]	False
`q.enqueue(8.4)`	[8.4, True,'dog',4]	
`q.dequeue()`	[8.4, True,'dog',]	4
`q.dequeue()`	[8.4, True]	'dog'
`q.size()`	[8.4, True]	2



รูปที่ 2.3: ตัวอย่างการทำงานของ Deque

2.4 Deque

Deque(อ่านว่า เด็ค) เรียกอีกอย่างว่า double-end Queue คือกลุ่มข้อมูลที่เรียงลำดับคล้ายกับ Queue แต่มีจุดปลาย 2 จุดคือเป็นได้ทั้ง front และ rear สิ่งที่ทำให้ Deque ต่างออกไปคือการเพิ่มและลบข้อมูล ข้อมูลใหม่สามารถถูกเพิ่มได้ทั้งด้าน front และ rear เช่นเดียวกันข้อมูลสามารถถูกลบได้ทั้งด้าน front และ rear ข้อสังเกต Deque **ไม่มี**คุณสมบัติทั้ง LIFO และ FIFO ดังรูปตัวอย่างที่ 2.3

2.4.1 Deque:abstract data type

นิยามด้วยโครงสร้างและการดำเนินการต่อไปนี้ Deque เป็นโครงสร้างที่กลุ่มข้อมูลถูกเรียงลำดับโดยข้อมูลสามารถถูกเพิ่มและเอาออกได้ทั้งทาง front และ rear สมมติว่า d แทน Deque การดำเนินการของ Deque มีดังตารางที่ 2.3

- `Deque()` เป็นการสร้าง Deque อันใหม่ที่ว่าง ไม่ต้องการ parameter และคืนค่าเป็น Deque ว่าง
- `add_front(item)` เพิ่มข้อมูลใหม่ให้ส่วน front ของ Deque ต้องการข้อมูลและไม่คืนค่า
- `add_rear(item)` เพิ่มข้อมูลใหม่ให้ส่วน rear ของ Deque ต้องการข้อมูลและไม่คืนค่า
- `remove_front()` นำข้อมูลที่ front ออกจาก Deque ไม่ต้องการ parameter คืนค่าข้อมูล (Deque มีการเปลี่ยนแปลง)
- `remove_rear()` นำข้อมูลที่ rear ออกจาก Deque ไม่ต้องการ parameter คืนค่าข้อมูล (Deque มีการเปลี่ยนแปลง)
- `is_empty()` ทดสอบว่า Deque ว่างไหม ไม่ต้องการ parameter คืนค่าข้อมูลเป็นจริงหรือเท็จ
- `size()` คืนค่าจำนวนข้อมูลของ Deque ไม่ต้องการ parameter คืนค่าข้อมูลจำนวนเต็ม

2.4.2 Dequq implementation

เราก็จะสร้าง class ใหม่สำหรับการ implement Deque เช่นเดียวกับ Stack และ Queue เราจะใช้ List ในการสร้าง Deque สมมติว่าเราจะให้ส่วน rear เป็น index ที่ 0 ใน List ดังตัวอย่าง

```
class Deque:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def add_front(self, item):
        self.items.append(item)
    def add_rear(self, item):
        self.items.insert(0, item)
    def remove_front(self):
        return self.items.pop()
```

ตารางที่ 2.3: ตัวอย่างการทำงานของ Deque

การดำเนินการของ Deque	สิ่งที่อยู่ใน Deque	ค่าที่คืน
`d.is_empty()`	[]	True
`d.add_rear(4)`	[4]	
`d.add_rear('dog')`	['dog',4]	
`d.add_front('cat')`	['dog',4,'cat']	
`d.add_front(True)`	['dog',4,'cat',True]	
`d.size()`	['dog',4,'cat',True]	4
`d.is_empty()`	['dog',4,'cat',True]	False
`d.add_rear(8.4)`	[8.4, 'dog',4,'cat',True]	
`d.remove_rear()`	['dog',4,'cat',True]	8.4
`d.remove_front()`	['dog',4,'cat']	True

```
def remove_rear(self):
    return self.items.pop(0)
def size(self):
    return len(self.items)
```

2.5 List

จากที่เราได้เขียน Stack, Queue และ Deque มาแล้วโดยใช้ List ของ Python เราพบว่า List มีประสิทธิภาพมาก อีกทั้งยังใช้งานง่าย อย่างไรก็ตามไม่ใช่ว่าทุกภาษาโปรแกรมจะมี List ให้ใช้แบบภาษา Python ซึ่งหากเป็นเช่นนี้ก็ต้องสร้างขึ้นมาเอง ดังนั้นต่อไปเราจะมาลองสร้าง List มาใช้เองกัน

2.5.1 Unordered list:abstract data type

List เป็นกลุ่มของข้อมูลที่แต่ละสมาชิกในข้อมูลจะมีตำแหน่งสัมพันธ์กับตัวอื่นๆ เบื้องต้นเราสนใจ List แบบไม่เรียงลำดับข้อมูลก่อน เราจะพิจารณา List ตามสมาชิกตัวที่ 1 สมาชิกตัวที่ 2 ... โดยเราสามารถอ้างถึงตัวแรกและตัวสุดท้ายของ List ได้ เพื่อความง่ายเราจะสมมติว่า List ไม่เก็บข้อมูลซ้ำกัน ตัวอย่างเช่นกลุ่มของข้อมูลจำนวนเต็ม 77,23,18,42,96 จะเป็น List แบบไม่เรียงลำดับของคะแนนสอบ โครงสร้างของ List แบบไม่เรียงลำดับ เป็นกลุ่มของข้อมูลที่แต่ละข้อมูลมีความสัมพันธ์ของตำแหน่งกับข้อมูลอื่น มีการดำเนินการดังนี้

- `List()` สร้าง List เปล่า ไม่ต้องการ parameter และ return List ว่าง

- `add(item)` เพิ่มข้อมูลเข้า List ต้องส่งข้อมูลให้แต่ไม่ต้อง return สมมติว่าข้อมูลที่เพิ่มไม่มีใน List
- `remove(item)` ลบข้อมูลออกจาก List ต้องการข้อมูลที่จะลบ สมมติว่ามีข้อมูลนั้นใน List
- `search(item)` ค้นหา item ใน List ต้องการข้อมูลที่จะค้นหาและคืนค่า boolean
- `is_empty()` ตรวจสอบว่า List ว่างไหม ไม่ต้องการ parameter คืนค่า boolean
- `size()` คืนค่าจำนวนของข้อมูลใน List ไม่ต้องการ parameter คืนค่าเป็นจำนวนเต็ม
- `append(item)` เพิ่มข้อมูลลงท้าย List ต้องการข้อมูลที่จะเพิ่ม ไม่คืนค่า สมมติว่าข้อมูลที่จะเพิ่มไม่มีใน List
- `index(item)` คืนค่าตำแหน่งของข้อมูลใน List ต้องการข้อมูล คืนค่าเป็นตำแหน่ง สมมติว่าข้อมูลอยู่ใน List
- `insert(pos,item)` เพิ่มข้อมูลใหม่ลงในตำแหน่ง pos ต้องการข้อมูล ไม่คืนค่า สมมติว่าข้อมูลไม่มีใน List และมีตำแหน่งว่างพอที่จะใส่ข้อมูล `pop()` เอาข้อมูลออกและคืนค่าข้อมูลสุดท้ายใน List ไม่ต้องการ parameter และคืนค่า สมมติว่ามีข้อมูลอย่างน้อย 1 ตัว `pop(pos)` เหมือน pop แต่เอาตัวที่ตำแหน่ง pos ออก

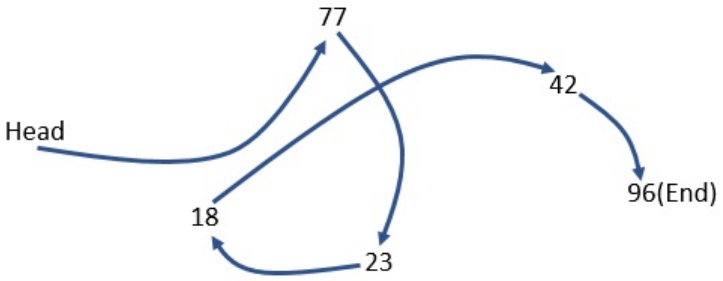
2.5.2 Unordered list implementation

ในการสร้าง List แบบไม่เรียงลำดับ โดยทั่วไปเราจะสร้างเป็น Linked List สิ่งที่เราต้องการคือ เมื่อพิจารณาข้อมูลตัวหนึ่งข้อมูลตัวถัดไปอยู่ที่ไหนในหน่วยความจำ นั่นคือเราเก็บตำแหน่งที่สัมพันธ์กันของข้อมูล แต่ไม่จำเป็นที่ต้องเก็บในตำแหน่งที่ต่อเนื่องกันในหน่วยความจำ ทำให้ค่าต่างๆ สามารถถูกเก็บได้อย่างอิสระในหน่วยความจำ ถ้าเราสามารถเก็บตำแหน่งที่เก็บของแต่ละข้อมูลได้ ในรูปเราจึงมองว่ามีลูกศรชี้ไปว่าข้อมูลต่อไปเป็นตัวไหน ดังนั้นสิ่งสำคัญคือการเก็บตำแหน่งของจุดเริ่มต้น เมื่อเรารู้ตำแหน่งของข้อมูลตัวแรกแล้วจะสามารถบอกตัวที่สองและตัวต่อๆ ไปได้ โดยทั่วไปจะเริ่มต้นที่ head ของ List และข้อมูลตัวสุดท้ายจะไม่ชี้ไปข้อมูลอื่น ตัวอย่างดังรูปที่ 2.4

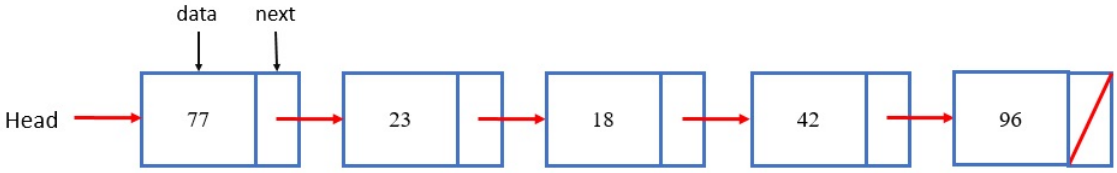
ในการสร้าง List เราจะสร้างโหนด(Node) เพื่อมาเป็นโครงสร้างพื้นฐานในการเก็บข้อมูลแต่ละตัว ตัวอย่างดังรูปที่ 2.5

แต่ละ Node จะเก็บข้อมูลหลัก 2 อย่างคือ ข้อมูลจริงๆ (data)และที่อยู่ของโหนดต่อไป(next) ดังนั้นในการสร้างโหนดจะต้องกำหนดค่าเริ่มต้นให้กับ 2 ข้อมูลข้างต้น ต่อไปเป็นตัวอย่างของ class Node

```
class Node:
    def __init__(self,initData):
        self.data = initData
```



รูปที่ 2.4: ตำแหน่งสมมติในการเก็บข้อมูล List [77,23,18,42,96] ในหน่วยความจำ



รูปที่ 2.5: ลักษณะโครงสร้างการเก็บข้อมูล List [77,23,18,42,96]

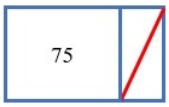
```
        self.next = None
    def get_data(self):
        return self.data
    def get_next(self):
        return self.next
    def set_data(self, new_data):
        self.data = new_date
    def set_next(self, new_next):
        self.next = new_next
```

จะเห็นได้ว่า Constructor จะกำหนดค่าข้อมูลจริงด้วย initData และที่อยู่ของโหนดถัดไปด้วย None คือไม่มีโหนดถัดไปก่อน หากเราต้องการสร้างโหนด ใหม่จะสร้างด้วยคำสั่งดังตัวอย่าง

```
temp = Node(75)
print(temp.get_data())
```

จะได้ผลลัพธ์ดังรูปที่2.6

เมื่อเรามี Node แล้วเราจะมาสร้าง Linked List กัน จากที่เราบอกว่า List แบบไม่เรียงลำดับ จะถูกสร้างจาก



รูปที่ 2.6: สร้างโหนดใหม่

กลุ่มของ Node ซึ่งแต่ละอันจะถูกเชื่อมกันด้วย next ซึ่งเป็นที่อยู่ของ Node ถัดไป ดังนั้นเริ่มต้นเราจะสร้าง head เพื่อเอาไว้เป็นต้นทางของ List และกำหนดให้ head ไม่มี Node ถัดไป หรือมีค่าเป็น Null ไว้ก่อน

```
class UnorderedList:
    def __init__(self):
        self.head = None
```

เริ่มต้นเราจะสร้าง List ที่ไม่มีข้อมูลเลย เราจะสร้างด้วย

```
mylist = UnorderedList()
```

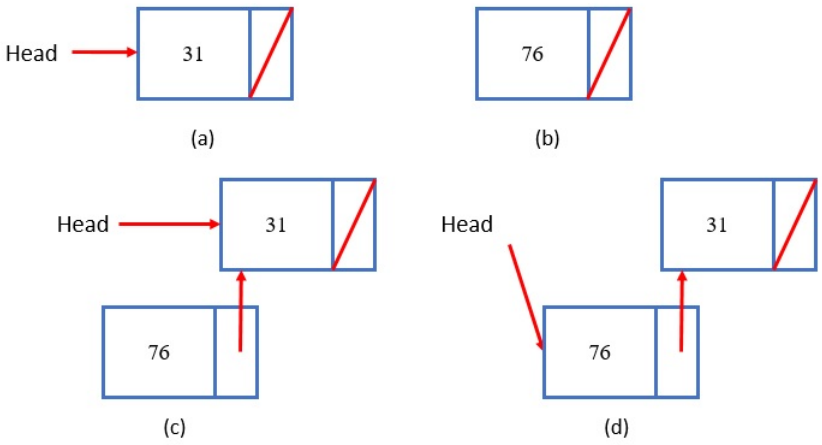
รูปที่ 2.7: List ตอนเริ่มต้นมีเพียง Head

นั่นคือเริ่มต้น head ไม่ชี้อะไรเลย ดังรูป 2.7 เราจะค่อยๆ เพิ่ม method ให้กับ class UnorderedList อย่างแรก `is_empty()` จะเป็นการตรวจสอบว่า head ของ List ว่างไหมผลลัพธ์ที่ได้จะเป็น boolean จาก `self.head == None` เป็นจริงเมื่อไม่มีโหนดใน List ซึ่งหากเป็น List ที่ถูกสร้างใหม่จะมีค่าเป็น empty

```
def is_empty(self):
    return self.head == None
```

ต่อไปเป็นการนำข้อมูลเข้าไปใน List เราต้องสร้างฟังก์ชันในการเพิ่มข้อมูล อย่างไรก็ตามก่อนที่เราจะสร้าง เราต้องตอบคำถามอย่างหนึ่งก่อน ว่าเราจะเพิ่มข้อมูลใหม่ไปใน Linked List ของเราตรงไหน เนื่องจากว่าตอนนี้เราสนใจ List แบบไม่เรียงลำดับ ดังนั้นตำแหน่งของข้อมูลใหม่จะไปอยู่ที่ใดก็ได้ ดังนั้นเราจะเพิ่มข้อมูลไปไว้ตำแหน่งที่ทำง่ายที่สุดคือด้านหน้า

จากโครงสร้างของ Linked List นั้นมีจุดทางเข้าโครงสร้าง 1 จุดตรง head ทุกโหนดจะถูกไปถึงได้ก็ต้องเริ่มที่ head จุดที่จะเพิ่มข้อมูลง่ายที่สุดคือจุดนี้โดยเราจะสร้างโหนดใหม่ แล้วให้โหนดใหม่ชี้ข้อมูลเก่า จากนั้นให้มันเป็น head ตัวอย่างการทำงานการเพิ่มข้อมูลให้กับ List แสดงดังรูปที่2.8



รูปที่ 2.8: ตัวอย่างการเพิ่มโหนดใหม่ให้กับ List

```
mylist.add(31)
mylist.add(76)
```

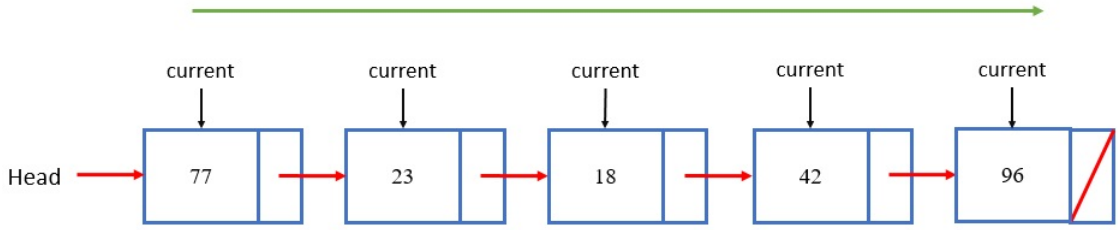
หากคำสั่งเป็นเช่นนี้โหนด 31 จะถูกเพิ่มเข้ามาดังรูปที่2.8(a) จากนั้นสร้างโหนดใหม่ที่มีข้อมูลเป็น 76 ดังรูปที่2.8(b) จากนั้นนำมาแทรกจะมาอยู่ก่อนหน้าโหนด 31 ดังนั้น method add ต้องหาทางเชื่อมโหนดใหม่เข้ากับโครงสร้างเก่า (ห้ามทำให้การเชื่อมต่อของ List ขาด) ซึ่งทำได้โดยการเชื่อม next ของโหนดใหม่เข้ากับโหนดที่ Head ชี้ก่อน ดังรูปที่2.8(c) จากนั้นจึงค่อยย้าย Head มาชี้โหนดใหม่ ดังรูปที่2.8(d)

```
def add(self,item):
    temp = Node(item)
    temp.set_next(self.head)
    self.head = temp
```

ทั้งนี้ลำดับของคำสั่งในบรรทัดที่ 3 และ 4 สำคัญมาก **คำถาม** หากสลับที่จะเกิดอะไรขึ้น ผลลัพธ์หากสลับคำสั่งเป็นอย่างไร

ต่อไปเราจะมาดู method size, search และ remove ซึ่งทั้ง 3 อันนี้จะต้องมีการท่องเข้าไปใน List เพื่อตรวจสอบ การท่องไปใน List เรียกว่า Linked list traversal โดยการ Traversal นี้จะเริ่มที่โหนดแรกใน List จากนั้นก็จะไปต่อทีละโหนดตามที่ next บอก

Method size เราก็จะท่องไปใน List ตั้งแต่ตัวแรกจนตัวสุดท้ายโดยที่นับจำนวนโหนดที่ผ่านไปด้วย เราจะมีตัวแปรที่ชื่อ current เอาไว้อ้างอิงว่าตอนนี้อยู่ที่โหนดไหน ดังนั้นเริ่มต้นให้ current เป็น head ของ List เริ่มต้นเรายังไม่ได้ท่องเข้าไปใน List จึงให้ count เป็น 0 จากนั้นเราก็จะท่องไปใน List (ตามเส้นลูกศรสีเขียวในรูปที่2.9) จนกว่าจะเจอ None คือท้าย List ดังรูปที่2.9



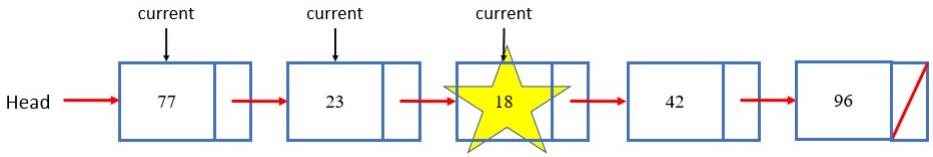
รูปที่ 2.9: ตัวอย่างการทำงานของ Method Size ใน List

```
def size(self):
    current = self.head
    count = 0
    while current != None:
        count = count +1
        current = current.get_next()
    return count
```

Method search การค้นหาค่าใน Linked list ใน List แบบไม่เรียงลำดับ ใช้หลักการการท่องไปใน List เนื่องจากว่าข้อมูลไม่ได้เรียงลำดับ ข้อมูลสามารถอยู่ตรงไหนก็ได้ใน List เราก็จะแวะไปทีละโหนดใน Linked list ถามว่ามีข้อมูลที่ต้องการเก็บอยู่ไหม ถ้าเราถามทุกโหนดแล้วแสดงว่าข้อมูลไม่มีอยู่ใน List ตัวอย่างการหาข้อมูล 18 แสดงดังรูปที่ 2.10

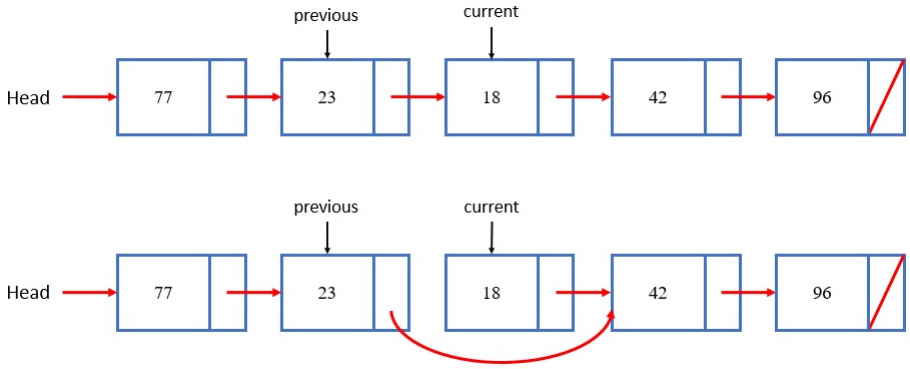
```
def search(self,item):
    current = self.head
    found = False
    while current != None and not found:
        if current.get_data() == item:
            found = True
        else:
            current = current.get_next()
    return found
```

Method remove มี 2 ขั้นตอน ขั้นตอนแรก เราต้องท่องไปใน List เพื่อหา item ที่เราต้องการลบ ขั้นที่สอง เมื่อเราพบแล้ว เราจะลบมัน ขั้นแรกเหมือนกับ traversal เริ่มต้นที่ head แล้วขยับไปเรื่อยๆ เมื่อเราพบแล้ว current จะเป็นโหนดที่เราจะลบ แล้วเราจะลบอย่างไรดี ในการลบเราจะต้องย้าย link ของโหนดก่อนหน้าข้าม current ไปชี้โหนดถัดจาก current เลย แต่เราไม่มีฟังก์ชันชี้กลับ แล้วทำอย่างไรดี เราจะสร้างตัวแปรมาอีกตัว ชื่อว่า previous ที่



รูปที่ 2.10: ตัวอย่างการค้นหาโหนด 18 จาก List

จะเป็นโหนดก่อน current นั่นคือเมื่อ current หยุดที่โหนดที่จะลบ previous จะอยู่โหนดก่อนหน้า



รูปที่ 2.11: ตัวอย่างการลบโหนด 18 จาก List

```
def remove(self,item):
    current = self.head
    previous = None
    found = False
    while not found:
        if current.get_data() == item:
            found = True
        else:
            previous = current
            current = current.get_next()
    if previous == None:
        self.head = current.get_next()
    else:
        previous.set_next(current.get_next())
```

เมื่อนำแต่ละส่วนของโหนดและ UnorderedList มารวมกันจะได้ดังต่อไปนี้

```
class Node:
    def __init__(self,init_data):
        self.data = init_data
        self.next = None
    def get_data(self):
        return self.data
    def get_next(self):
        return self.next
    def set_data(self, new_data):
        self.data = new_date
    def set_next(self, new_next):
        self.next = new_next
class UnorderedList:
    def __init__(self):
        self.head = None
    def is_empty(self):
        return self.head == None
    def add(self,item):
        temp = Node(item)
        temp.set_next(self.head)
        self.head = temp
    def size(self):
        current = self.head
        count = 0
        while current != None:
            count = count +1
            current = current.get_next()
        return count
     def search(self,item):
        current = self.head
        found = False
        while current != None and not found:
            if current.get_data() == item:
                found = True
```

```
            else:
                current = current.get_next()
                return found
    def remove(self,item):
            current = self.head
            previous = None
            found = False
            while not found:
                if current.get_data() == item:
                    found = True
                else:
                    previous = current
                    current = current.get_next()
                if previous == None:
                    self.head = current.get_next()
                else:
                    previous.set_next(current.get_next())
```

ทดลองใช้งาน

```
mylist = UnorderedList()
mylist.add(2)
print(mylist.search(2))
print(mylist.search(3))
print(mylist.size())
mylist.add(88)
print(mylist.size())
mylist.remove(2)
print(mylist.size())
```

2.5.3 Ordered list:abstract data type

ต่อไปเราจะพิจารณา List แบบเรียงลำดับ นั่นคือข้อมูลเรียงจากน้อยไปมาก เช่น 18, 23, 42, 77, 96 เป็นต้น method ส่วนใหญ่จะเหมือนกับแบบ List แบบไม่เรียงลำดับ

- `OrderedList()` สร้าง List แบบเรียงลำดับอันใหม่ขึ้นมา

- `add(item)` เพิ่ม item ใหม่เข้าไปใน List โดยเพิ่มให้ตำแหน่งถูกต้องตามลำดับ
- `remove(item)` ลบ item ใน List
- `search(item)` ค้นหา item ใน List ต้องการข้อมูลที่จะค้นหาและคืนค่า boolean
- `is_empty()` ตรวจสอบว่า List ว่างไหม ไม่ต้องการ parameter คืนค่า boolean
- `size()` คืนค่าจำนวนของข้อมูลใน List ไม่ต้องการ parameter คืนค่าเป็นจำนวนเต็ม
- `index(item)` คืนค่าตำแหน่งของ item ใน List ต้องการข้อมูลเข้าและคืนค่าเป็นจำนวนเต็ม
- `pop()` ลบข้อมูลตัวสุดท้ายของ List ไม่ต้องการ parameter คืนค่าเป็นข้อมูล
- `pop(pos)` ลบข้อมูลตำแหน่งที่ pos ของ List ต้องการ parameter คืนค่าเป็นข้อมูล

2.5.4 Ordered list implementation

สำหรับ List แบบเรียงลำดับนั้นโหนดยังคงมีโครงสร้างเหมือนเดิม ส่วนที่เปลี่ยนคือ List ดังนั้นจะสร้าง class ใหม่โดยใช้ชื่อว่า OrderedList() ดังตัวอย่าง

```
class OrderedList:
        def __init__(self):
                self.head = None
```

การดำเนินการต่างๆ ของ List แบบมีลำดับนั้นคล้ายกับ List แบบไม่มีลำดับ เพียงแค่ขั้นตอนการเพิ่มจะต้องใส่ข้อมูลให้ถูกต้องตามลำดับ การลบ การค้นหา สามารถใช้ method เดิมได้ แต่หากอยากเพิ่มประสิทธิภาพ เนื่องจากข้อมูลเรียงอยู่แล้ว หาก current มีค่ามากกว่าค่าที่เราค้นหา ก็แสดงว่าไม่มีข้อมูลนั้น ทำให้หยุดการค้นหาได้ เราจะมีพิจารณา 2 method หลักได้แก่การค้นหาและการเพิ่มข้อมูล ส่วน method ที่เหลือให้ลองทำเป็นแบบฝึกหัด

การค้นหาข้อมูล เราจะใช้หลักการเดิมคือใช้ตัวแปร current ในการท่องไปใน List แต่หากข้อมูลที่ต้องการค้นหามีค่ามากกว่า ค่าที่ current ชี้แสดงว่าไม่พบข้อมูลแน่นอน ดังตัวอย่าง

```
def search(self,item):
    current = self.head
    found = False
    stop = False
    while current != None and not found and not stop:
        if current.get_data() == item:
```

```
            found = True
        else:
            if current.get_data() > item:
                stop = True
            else:
                current = current.get_next()
        return found
```

สำหรับการเพิ่มข้อมูล ขั้นแรกเราจะต้องหาตำแหน่งที่จะเพิ่มก่อน นั่นคือ ตัวแรกสุดที่มากกว่าตัวที่เราจะเพิ่ม ทั้งนี้เราต้องเก็บข้อมูลโหนดก่อนหน้าไว้ด้วย เช่นเดียวกับคราวก่อนเราจะใช้ตัวแปร previous ดังตัวอย่าง

```
def add(self,item):
    current = self.head
    previous = None
    stop = False
    while current != None and not stop:
        if current.get_data() > item:
            stop = True
        else:
            previous = current
            current = current.get_next()

    temp = Node(item)
    if previous == None:
        temp.set_next(self.head)
        self.head = temp
    else:
        temp.set_next(current)
        previus.set_next(temp)
```

2.6 แบบฝึกหัดท้ายบท

1. จง implement Stack ด้วย Queue

2. จง implement Queue ด้วย Stack

3. เมื่อกำหนด class Stack มาให้ จงสร้าง method dequeueAll() ที่ทำการ dequeue() ของใน Queue ออกจนหมด

4. เมื่อกำหนด class Deque มาให้ จงสร้าง method clearDeque() ที่ทำการลบของใน Deque ออกจนหมด

5. จง implement method index(item) ให้กับ class OrderedList

6. จง implement method pop() ให้กับ class OrderedList

7. จง implement method pop(pos) ให้กับ class OrderedList

บทที่ 3

การเวียนเกิด

3.1 Recursion คืออะไร

การเวียนเกิดหรือ Recursion คือวิธีการแก้ปัญหาแบบหนึ่งที่เกี่ยวกับการแตกปัญหาเป็นปัญหาเดิมที่มีขนาดของข้อมูลเข้าเล็กลงๆ จนกระทั่งปัญหานั้นเล็กพอที่เราจะแก้มันได้โดยง่าย ซึ่งโดยทั่วไปแล้ว Recursion จะเป็นการเขียนโปรแกรมที่มีฟังก์ชันซึ่งภายในฟังก์ชันนั้นมีการเรียกชื่อฟังก์ชันตัวเอง

3.2 การหาผลรวมด้วย Recursion

เราจะเริ่มต้นด้วยตัวอย่างของปัญหาที่ง่ายก่อน ซึ่งปัญหานี้เรารู้วิธีการแก้โดยที่ไม่ต้องใช้ Recursion อยู่แล้ว สมมติว่าเรามี List [1, 3, 5, 7, 9, 11, 13, 15] แล้วต้องการคำนวณหาผลรวมของจำนวนใน List ถ้าหากเขียนฟังก์ชันแบบวนซ้ำจะเขียนโปรแกรมได้ดังนี้

```
def list_sum(num_list):
    sum = 0
    for i in num_list:
        sum = sum + i
    return sum

print(list_sum([1,3,5,7,9,11,13,15]))
```

ในฟังก์ชัน list_sum ใช้ตัวแปร sum ในการเก็บผลรวมของจำนวนทุกจำนวนใน List โดยเริ่มต้นให้ตัวแปร sum นี้มีค่าเป็น 0 หลังจากนั้นจะเพิ่มค่าด้วยสมาชิกใน List ทีละตัว

สมมติว่าถ้าเราไม่มีคำสั่ง for หรือ while ให้ใช้ เราจะคำนวณผลรวมของจำนวนใน List ได้อย่างไร

เราอาจจะเริ่มด้วยสิ่งที่เรามี นั่นคือฟังก์ชันนั่นเอง เราจะเริ่มโดยสร้างฟังก์ชัน addition ซึ่งเป็นฟังก์ชันที่รับข้อมูลเข้าสองตัวแล้วคืนค่าเป็นผลรวมของข้อมูลเข้าสองจำนวนนั้น ดังนั้นเราสามารถเขียนผลรวมของสมาชิกใน List จากการบวกเป็นคู่ๆ โดยใช้ฟังก์ชันแทน ตัวอย่างเช่นต้องการหาค่า 1+3 เราก็เรียก addition(1,3) คืนค่า 4 แล้วหากเราต้องการหาค่า 1+3+5 เราก็เรียก addition(addition(1,3),5) เป็นต้น เดียวเราจะย่อฟังก์ชัน addition ให้เหลือแค่วงเล็บแทนเพื่อความสะดวก ดังนั้นในการบวกเลขของ List [1,3,5,7,9,11,13,15] หากเราจะเขียนด้วยวงเล็บแทนจะได้ ((((1+3)+5)+7)+9) หรืออาจจะเขียนเป็น (1+(3+(5+(7+9)))) ก็ได้ หากพิจารณาการเรียกในแบบที่สอง วงเล็บในสุด (7+9) เป็นปัญหาที่เราสามารถแก้ได้โดยไม่ต้องใช้ loop หรือคำสั่งพิเศษอื่นๆ ทั้งนี้เราใช้ลำดับที่ได้นี้เพื่อคำนวณผลรวมสุดท้ายได้

```
sum = (1+(3+(5+(7+9))))
sum = (1+(3+(5+16)))
sum = (1+(3+21))
sum = (1+24)
sum = 25
```

จากตัวอย่างสังเกตได้ว่าเป็นการแก้ปัญหาจากด้านในสุดออกมาด้านนอก เราสามารถเอาแนวคิดนี้มาเขียนเป็นภาษาไพธอนได้อย่างไร

เริ่มต้นปัญหาการหาผลรวมในรูปของ List เราอาจจะมองว่า ผลรวมของ num_list คือผลรวมของสมาชิกตัวแรก num_list[0] กับตัวที่เหลือ num_list[1:] ก็ได้ ดังนั้นหากประกาศในรูปของฟังก์ชันจะได้ว่า

`list_sum(num_list) = num_list[0]+list_sum(num_list[1:])`

ทั้งนี้เราต้องจัดการกับกรณีที่ข้อมูลมีตัวเดียวด้วย ต่อไปเป็นตัวอย่างโปรแกรม list_sum

```
def list_sum(num_list):
    if len(num_list) == 1:
        return num_list[0]
    else:
        return num_list[0] + list_sum(num_list[1:])
print(list_sum([1,3,5,7,9]))
```

จากตัวอย่างนี้เราพอจะได้แนวคิดเบื้องต้น 2 อย่าง

- แนวคิดแรกในบรรทัดที่ 2 เราได้ตรวจสอบว่าถ้า List มีความยาว 1 ตัวหรือไม่ ซึ่งการตรวจสอบนี้สำคัญมาก เพราะว่าจะเป็นจุดที่ทำให้เราออกจากฟังก์ชันได้หรือเลิกทำงานได้ ทั้งนี้ผลรวมของ List ที่มีความยาว 1 ตัวนั้นหาค่าได้ง่ายนั่นคือ **ตัวมันเอง**

- แนวคิดที่สองในบรรทัดที่ 5 ฟังก์ชันของเรามีการเรียกตัวเอง โดยฟังก์ชันที่เรียกตัวเองเราเรียกว่า Recursive function เมื่อเราทำตามไปจนถึงจุดหนึ่งที่ปัญหาแก้ง่ายแล้ว เราจะนำส่วนเล็กๆ ที่แก้แล้วมารวมกันกลับ

เป็นคำตอบของปัญหาตั้งต้น ในรูปเป็นการแสดงว่า list_sum ทำงานอย่างไร เป็นการเรียกทำงานจากหลังไปหน้า เมื่อ list_sum ไปถึงจุดบนสุด เราจะได้คำตอบ

ทุก Recursive algorithm จะมีสิ่งที่เหมือนๆ กัน 3 ข้อ

- Recursive algorithm ต้องมี Base case
- Recursive algorithm ต้องเปลี่ยน State และเปลี่ยนไปเป็น Base case
- Recursive algorithm ต้องเรียกตัวเอง

ทีนี้มาลองพิจารณาแต่ละข้อ เมื่อเปรียบเทียบกับ list_sum

- ข้อแรก Base case คือเงื่อนไขที่ทำให้หยุดการเรียกตัวเอง Base case โดยทั่วไปแล้วคือปัญหาที่มีขนาดเล็กพอที่เราจะแก้ได้โดยตรง ใน list_sum นั้น Base case คือ List ที่ความยาว 1 ตัว
- ข้อสอง เราจะต้องเรียงการเปลี่ยนแปลงของ State ที่เปลี่ยนไปเป็น Base case ทั้งนี้การเปลี่ยน State หมายถึงข้อมูลที่ใช้ถูกเปลี่ยนแปลง โดยทั่วไปแล้วข้อมูลที่แสดงในปัญหาของเรานั้น จะถูกทำให้เล็กลงได้ในบางวิธี ใน list_sum โครงสร้างข้อมูลของเราคือ List ซึ่งเราจะสนใจการเปลี่ยน State ของ List เนื่องจาก Base case คือ List ที่ยาว 1 การเปลี่ยน State เพื่อให้ไปถึง Base case ได้คือ การทำให้ List สั้นลง นั่นคือสิ่งที่เกิดขึ้นในบรรทัดที่ 5 ที่เราเรียก list_sum ด้วย List ที่สั้นลง
- กฎข้อสุดท้ายคือ ต้องเรียกตัวเอง นี่เป็นนิยามของ Recursion

3.3 การหาค่าของ x^y

ตัวอย่างนี้จะเป็นการหาค่าของ x^y ตัวอย่างเช่น $5^3 = 125$ เป็นต้น 5^3 เกิดจากอะไร แน่นอนเกิดจาก 5x5x5 เราจะพิจารณาคล้ายกับตัวอย่างก่อนหน้า $5^3 = 5 * 5^2$ แล้ว $5^2 = 5 * 5^1$ และสุดท้าย $5^1 = 5$ ดังนั้นสมมติว่าฟังก์ชันที่เราจะใช้ในการหาค่า x^y คือฟังก์ชัน power(x,y) การเรียกใช้งานของ 5^3 คือ $power(5,3) = 5 * power(5,2)$ นั่นเอง

คำถาม จุดที่ทำให้เราออกจากฟังก์ชันได้หรือเลิกทำงานได้ คือจุดใด คำตอบ $power(5,0)$ หรือ $power(5,1)$ ก็ได้คือเป็นกรณีที่ง่าย ต่อไปเป็นตัวอย่างโปรแกรม

```
def power(x,y):
    if (y==1):
        return x
    else:
```

```
        return x*power(x,y-1)
print(power(5,3))
```

ทีนี้มาลองพิจารณาแต่ละข้อ เมื่อเปรียบเทียบกับ Recursive algorithm

- ข้อแรก Base case คือเงื่อนไขที่ทำให้หยุดการเรียกตัวเอง ในตัวอย่างนี้ Base case คือ y = 1 หรือ $x^1 = x$ ให้คืนค่า x นั่นเอง
- ข้อสอง การเปลี่ยนแปลงของ State ที่เปลี่ยนไปเป็น Base case ในตัวอย่างนี้คือเราเรียก power ด้วย ค่า y ที่น้อยลง
- กฎข้อสุดท้ายคือ ต้องเรียกตัวเอง นี่เป็นนิยามของ Recursion

ให้ลองเขียน code ในกรณีที่เลือก Base case y = 0

3.4 การเปลี่ยนเลขจำนวนเต็มฐานสิบเป็นข้อความของตัวเลขนั้นในฐานอื่น

หัวข้อนี้เราจะมีดูปัญหาการเปลี่ยนเลขจำนวนเต็มฐานสิบไปเป็นข้อความของเลขตัวนั้นในฐานอื่นๆ ที่อยู่ระหว่างฐานสองกับฐานสิบหก ตัวอย่างเช่นต้องการเปลี่ยนเลข 10 ไปเป็นข้อความในฐานสิบจะได้ ”10” หรือเปลี่ยนไปเป็นข้อความในฐานสองจะได้ ”1010”

จริงๆ แล้วเราสามารถลองเขียนโปรแกรมแก้ได้โดยใช้ Stack ในบทนี้เราจะมาหาทางแก้ด้วย Recursive กันก่อนอื่นลองพิจารณาตัวอย่างของการการแปลงเลข 196 ให้เป็นข้อความในฐาน 10 ก่อน สมมติว่าเรามีลำดับของอักขระที่สอดคล้องกับฐาน 10 นั่นคือ conv_string = ”0123456789” หากเปลี่ยนตัวเลขที่น้อยกว่า 10 ไปเป็นข้อความที่เท่ากันนั้นทำได้โดยระบุ index ใน conv_string ได้เลยเช่น ถ้าจำนวนที่พิจารณาเป็น 6 แล้ว ข้อความใน `conv_string[6]` หรือ ”6” ถ้าเราสามารถแบ่งเลข 196 ออกเป็นเลขโดด 3 ตัว ได้แก่ 1, 9 และ 6 แล้วการเปลี่ยนเป็นข้อความจะง่าย ดังนั้นจำนวนที่น้อยกว่า 10 น่าจะเป็น base case ภาพรวมของโปรแกรมของเราจะเกี่ยวข้องกับ 3 ส่วน

- ลดเลขตั้งต้นให้เป็นลำดับของเลขโดด
- เปลี่ยนเลขโดดให้เป็นข้อความโดยใช้การเทียบ
- ต่อข้อความที่เป็นเลขโดดเข้าด้วยกันเป็นคำตอบ

ขั้นต่อไปเป็นการหาว่าจะเปลี่ยน State อย่างไรเพื่อให้มีความก้าวหน้าจนไปถึง Base case ได้ เนื่องจากเราทำงานกับจำนวนเต็ม พิจารณาว่าการดำเนินการทางคณิตศาสตร์อะไรที่ทำให้จำนวนลดลง ตัวเลือกที่น่าจะเป็นไปได้คือ การหาร และ การลบ

การลบน่าจะใช้ได้แต่ว่ามันไม่ชัดเจนว่าเราจะลบอะไร การหารกับส่วนที่เหลือให้สิ่งที่เราต้องการ ดังนั้นเราจะมาดูว่าอะไรเกิดขึ้นถ้าเราหารด้วยฐานที่เราต้องการเปลี่ยน เมื่อเราหาร 196 ด้วย 10 เราจะได้ 19 และได้เศษ 6 อย่างแรกเราได้เศษ ซึ่งคือจำนวนที่น้อยกว่าฐานที่เราต้องการเปลี่ยน ที่สำคัญเราเปลี่ยนเป็นข้อความได้เลย อย่างที่สองเราได้จำนวนที่น้อยกว่าค่าเริ่มต้นและลดลงมุ่งไปสู่ Base case ดังนั้นงานต่อไปก็คือ เราจะเปลี่ยน 19 ให้ไปเป็นข้อความซึ่งเราจะได้ 1 เศษ 9 สุดท้ายเราลดรูปปัญหาได้เหลือเพียงการเปลี่ยน 1 จากเงื่อนไขที่ว่า n <= base ซึ่ง base = 10 หากเขียนเป็น Python ระหว่างฐาน 2 ถึงฐาน 16 จะได้

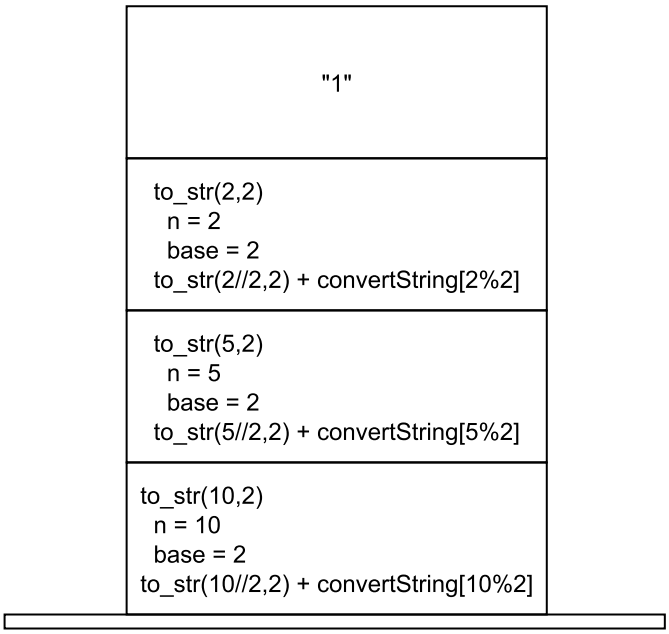
```
def to_str(n , base):
    convert_string = "012345679ABCDEF"
    if n < base:
        return convert_string[int(n)]
    else:
        return to_str(int(n/base),base)+convert_string[int(n % base)]

print(to_str(1453,16))
```

สังเกตว่าบรรทัดที่ 3 เราจะตรวจสอบ Base case ว่า n น้อยกว่า base ที่เราต้องการเปลี่ยนหรือไม่ ถ้าตรวจสอบพบ เราจะหยุดเรียกตัวเอง และคืนค่าข้อความจาก index ใน convert_string ในบรรทัดที่ 6 สอดคล้องกับกฎข้อ 2 และ 3 โดยเรียกตัวเองและลดขนาดปัญหาด้วยการหาร

3.5 การ Implement recursion ด้วย Stack

เราสามารถมองการทำงานของ Recursion โดยใช้ Stack ได้ สมมติว่าแทนที่เราจะเชื่อมผลลัพธ์ของการเรียกตัวเองของ to_str กับข้อความจาก convert_string โดยปรับอัลกอริทึมของเราให้ push ข้อความลง Stack แทนการเรียก Recursive สามารถปรับ code ได้ดังนี้

```
from mystack import Stack # As previously defined
r_stack = Stack()
def to_str(n, base):
    convert_string = "0123456789ABCDEF"
    while n > 0:
        if n < base:
            r_stack.push(convert_string[n])
        else:
            r_stack.push(convert_string[n % base])
        n = n // base
```

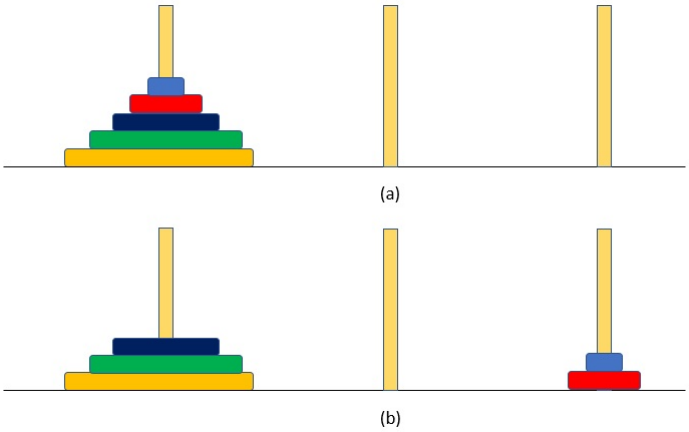


รูปที่ 3.1: ตัวอย่าง Stack frame ของการเรียก to_str(10, 2)

```
    res = ""
    while not r_stack.is_empty():
        res = res + str(r_stack.pop())
    return res

print(to_str(10, 2))
```

ในแต่ละครั้งที่มีการเรียก to_str เราจะ push อักขระไว้บน stack จากตัวอย่างก่อนหน้า เราจะเห็นได้ว่าหลังจากมีการเรียก to_str ในครั้งที่ 4 ครั้ง Stack จะมีหน้าตาแบบนี้

สังเกตว่าเราสามารถ pop อักขระออกจาก Stack แล้วเชื่อมกันเป็นผลลัพธ์สุดท้าย 1010 ตัวอย่างก่อนหน้าทำให้เห็นว่าภาษาไพธอนนั้นสร้างการเรียก Recursive function นั้นทำอย่างไร เมื่อฟังก์ชันถูกเรียกในภาษาไพธอนแล้ว Stack frame จะถูกจองเพื่อจัดการเก็บ Local variable ของฟังก์ชันไว้ เหมือนบันทึกสถานะไว้นั่นเอง เมื่อฟังก์ชันคืนค่า ค่าที่คืนคือ top ของ Stack สำหรับการเรียกฟังก์ชัน ตัวอย่างของการเรียก Stack ที่ถูกสร้างจาก to_str(10,2) นั้น Stack frame ให้ขอบเขตของตัวแปรที่ใช้ได้ของฟังก์ชัน แม้ว่าเราเรียกฟังก์ชันเดียวกันซ้ำแล้วซ้ำอีก การเรียกแต่ละครั้งจะสร้าง Scope ใหม่ของตัวแปรซึ่งเป็น Local variable ของฟังก์ชันขึ้นมา ดังนั้นถ้าหากเข้าใจเรื่อง Stack จะทำให้เข้าใจ Recursive function ง่ายขึ้น



รูปที่ 3.2: หอคอยแห่งฮานอย (a)เริ่มต้น (b)มีการย้ายจาน

3.6 หอคอยแห่งฮานอย

หอคอยแห่งฮานอย(Tower of Hanoi)[4] เป็นเกมคณิตศาสตร์ ประกอบด้วยหมุด 3 แท่ง และ จานกลมแบนขนาดต่างๆ ซึ่งมีรูตรงกลางสำหรับให้หมุดลอด เกมเริ่มจากจานทั้งหมดวางอยู่ที่หมุดเดียวกัน โดยเรียงตามขนาดจากใหญ่ที่สุดอยู่ทางด้านล่าง จนถึงจานขนาดเล็กที่สุดอยู่ด้านบนสุด เป็นลักษณะกรวยคว่ำตามรูปที่3.2

เป้าหมายของเกมคือ พยายามย้ายกองจานทั้งหมดไปไว้ที่อีกหมุดหนึ่ง โดยการเคลื่อนย้ายจานจะต้องเป็นไปตามกฎคือ

- สามารถย้ายจานได้เพียงครั้งละ 1 ใบ
- ไม่สามารถวางจาน ไว้บนจานที่มีขนาดเล็กกว่าได้

หอคอยฮานอยเป็นเกมปริศนาที่สร้างขึ้นมาโดย นักคณิตศาสตร์ชื่อ Edouard Lucas ในปี 1883 มีตำนานเกี่ยวกับวัดฮินดู ซึ่งมีห้องที่ภายใน มีเสา 3 หลัก และ จานทองอยู่ 64 ใบ คล้องอยู่กับเสา โดยที่พราหมณ์ในโบสถ์นั้นจะทำการเคลื่อนย้ายจานทองตามคำสั่งที่ระบุไว้ในคำพยากรณ์ โดยการเคลื่อนย้ายนั้นจะต้องเป็นไปตามเงื่อนไขของเกมปัญหา

เราจะแก้ปัญหานี้แบบ recursive ได้อย่างไร ลองคิดแก้ปัญหานี้จากล่างขึ้นบน สมมติว่าเรามีเสาที่มี 5 จาน(เสาแรกหรือเสาซ้ายมือสุด) ถ้าเรารู้วิธีย้ายเสาเมื่อมี 4 จาน เราจะทำอย่างไร เราก็ย้าย 4 จานมาไว้เสากลาง ย้ายจานสุดท้ายไปไว้เสาที่สาม(ขวามือสุด) แล้วย้าย 4 จานไปไว้เสาที่สาม ถ้าเราไม่รู้วิธีการย้ายเสาเมื่อมี 4 จาน สมมติว่าเรารู้วิธีการย้าย เสาที่มี 3 จาน เราก็จะย้ายเสา 3 จานไปไว้เสาต้นกลาง แล้วย้ายจานที่ 4 ไปไว้เสาที่ 3 จากนั้นย้ายจานจากเสาต้นกลางมาต้นขวาสุด ถ้าไม่รู้ก็ดูว่า 2 จานย้ายอย่างไร ... ก็ทำแบบเดิม

แนวทางคือสมมติว่าเรามีจานจำนวน height ใบ เราจะย้ายจากหมุดเริ่มต้นไปยังหมุดปลายทางโดยใช้หมุดช่วยเหลือ

1. ย้ายหมุดที่มีจาน height-1 ใบไปหมุดช่วยเหลือ โดยใช้หมุดปลายทางเป็นตัวฝากของ

2. ย้ายจานที่ค้างอยู่ไปยังหมุดปลายทาง

3. ย้ายหมุดที่มีจาน height-1 ใบจากหมุดช่วยเหลือไปหมุดปลายทางโดยใช้หมุดเริ่มต้นเป็นตัวฝากของ

ตราบเท่าที่เรายังใช้กฎที่ว่า ไม่สามารถวางจาน ไว้บนจานที่มีขนาดเล็กกว่าได้ เราจะสามารถใช้ 3 ขั้นตอนที่ว่าแบบ Recursive ได้ โดยทำเหมือนกับว่าจานขนาดใหญ่ไม่ได้อยู่ตรงนั้น

สิ่งที่ขาดไปคือ การจัดการกับ Base case ดังนั้นเราต้องพิจารณาก่อนว่าปัญหาง่ายสุดของหอคอยแห่งฮานอยคือมีจานกี่ใบ คำตอบคือ ใบเดียว นั่นคือคือเราย้ายไปยังเสาปลายทางได้เลย แล้ว State จะเปลี่ยนไปยัง Base case คือ การลดความสูงของหมุดในขั้นที่ 1 และ 3

```
def move_tower(height, fp, tp, wp):
    if height >= 1:
        move_tower(height - 1, fp, wp, tp)
        move_disk(fp, tp)
        move_tower(height - 1, wp, tp, fp)
```

หัวใจของอัลกอริทึมนี้คือการเรียก Recursive 2 อันที่ต่างกัน บรรทัดที่ 3 และ 5 บรรทัดที่ 3 ย้ายทุกจานยกเว้นจานสุดท้ายจากเสาเริ่มต้นไปยังเสาที่พัก บรรทัดต่อมาย้ายจานสุดท้ายไปยังเสาปลายทาง บรรทัดที่ 5 ย้ายทุกจานจากเสาที่พักไปยังเสาปลายทางวางบนจานสุดท้ายนั่นเอง ส่วน Base case จะตรวจพบเมื่อความสูงเป็น 0 ในกรณีนี้จะไม่ทำอะไร

ต่อไปเราเพิ่มฟังก์ชัน moveDisk ซึ่งเป็นการ print ว่าเราย้ายจานจากหมุดไหนไปหมุดไหน

```
def move_tower(height, fp, tp, wp):
    if height >= 1:
        move_tower(height - 1, fp, wp, tp)
        move_disk(fp, tp)
        move_tower(height - 1, wp, tp, fp)

def move_disk(fp,tp):
    print("moving disk from",fp,"to",tp)
move_tower(3, "A", "B", "C")
```

3.7 แบบฝึกหัดท้ายบท

1. จงเขียนฟังก์ชันแบบ Recursive เพื่อ คำนวณค่า n! (factorial) เมื่อให้ค่า n

2. จงเขียนฟังก์ชันที่รับ String แล้วตรวจสอบว่า String ที่รับมาเป็น palindrome หรือไม่ ให้คืนค่า True / False String เป็น Palindrome ถ้าสะกดจากด้านหน้าไปหลัง และหลังไปหน้าได้ตัวเดียวกัน ตัวอย่างเช่น kayak

3. จงเขียนฟังก์ชัน คำนวณค่าลำดับ Fibonacci ลำดับเริ่มที่ 1 เช่น print(fibo(5)) ได้ค่า 5 `print(fibo(6))` ได้ค่า 8

บทที่ 4

การค้นหาและการเรียงลำดับ

ในบทนี้จะกล่าวถึงปัญหาพื้นฐานที่พบเสมอทั้งในชีวิตประจำวันและในการคำนวณต่างๆ นั่นคือการค้นหาและการเรียงลำดับ โดยเราจะนำเสนอการค้นหาแบบที่เราไม่รู้ว่าข้อมูลของเราเป็นอย่างไร ซึ่งไม่มีทางเลือกเราก็ต้องค่อยๆ ไล่ค้นหาไปทีละตัวๆ หรือการค้นหาตามลำดับ เราจะค้นหาจนพบข้อมูลหรือจนข้อมูลหมดแล้วไม่พบ จากนั้นหากเรารู้ว่าข้อมูลของเรามีการเรียงลำดับ เราจะมีวิธีการค้นหาที่เร็วกว่าเดิมโดยใช้การค้นหาแบบทวิภาค และส่วนสุดท้ายของการค้นหาเราจะค้นหาให้เร็วๆ ทำได้อย่างไรในการแฮช สำหรับการเรียงลำดับเราจะเริ่มจากการเรียงที่มีประสิทธิภาพใกล้เคียงกันคือ Bubble sort, Selection sort, Insertion sort ก่อน จากนั้นจะนำเสนอ Merge sort ที่มีความเร็วในการทำงานสูงขึ้นและสุดท้าย Quick sort ซึ่งเป็นการเรียงที่โดยเฉลี่ยแล้วมีความเร็วในการทำงานดีและถูก builtin ในหลายภาษา

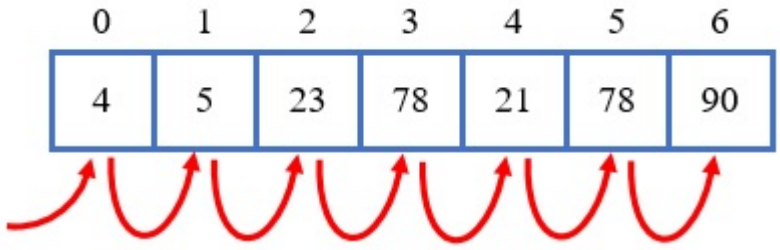
4.1 การค้นหา

การค้นหาคือกระบวนการที่เป็นขั้นตอนในการหาวัตถุจากกลุ่มของวัตถุ โดยทั่วไปแล้วผลลัพธ์ของการค้นหาคือจริง(True) เมื่อมีวัตถุที่ต้องการหาอยู่ในกลุ่มของวัตถุ และเท็จ(False) ในกรณีที่ไม่มีวัตถุที่ต้องการหาอยู่ในกลุ่มของวัตถุ อย่างไรก็ตามเมื่อเขียนโปรแกรมอาจจะคืนค่าต่างออกไปจากนี้ก็ได้เช่นคืนค่าว่าอยู่ตำแหน่งใดในกลุ่มวัตถุ

ก่อนอื่นในภาษาไพธอนนั้นหากกลุ่มของข้อมูลถูกเก็บด้วย List จะมีฟังก์ชันที่ช่วยในการค้นหาว่ามีข้อมูลที่ต้องการอยู่ในกลุ่มของข้อมูลหรือไม่คือ ฟังก์ชัน in ดังตัวอย่าง

```
25 in [4,5,23,78,21,78,90] #False
21 in [4,5,23,78,21,78,90] #True
```

แม้ว่าจะมีฟังก์ชัน in ให้ใช้ แต่เราก็ควรเข้าใจการทำงานภายในว่าทำงานได้อย่างไร ซึ่งในหัวข้อย่อยต่อไปจะเป็นการค้นหาตามลำดับ การค้นหาทวิภาคและการแฮช



รูปที่ 4.1: ตัวอย่างการทำงานของการค้นหาตามลำดับใน List

4.1.1 การค้นหาตามลำดับ

เมื่อข้อมูลถูกเก็บเป็นกลุ่มเช่นถูกเก็บใน List เราจะบอกว่าข้อมูลเหล่านี้มีความสัมพันธ์กันเชิงเส้นหรือมีลำดับต่อกัน เนื่องจากข้อมูลแต่ละตัวนั้นถูกเก็บในตำแหน่งที่สัมพันธ์กับข้อมูลตัวอื่น ซึ่ง List ใน python นั้นตำแหน่งที่สัมพันธ์กันคือค่า index ของแต่ละข้อมูล เนื่องจากว่า index นั้นเป็นค่าที่มีลำดับจากน้อยไปมาก ทำให้เราสามารถค้นหาตามลำดับนี้ได้ ซึ่งการค้นหาตามลำดับ หรือตาม index นี้คือการค้นหาแบบลำดับหรือ Sequential search นั่นเอง

จากรูปที่ 4.1 แสดงให้เห็นถึงการทำงานของการค้นหาตามลำดับ เริ่มต้นจะทำการค้นหาที่ตัวแรกของ List หากไม่ใช่ตัวที่ค้นหาจะทำการค้นหาต่อไปยังตัวถัดไปใน List ทำเช่นนี้ต่อไปเรื่อยๆ จนกระทั่งพบข้อมูลที่ต้องการหรือค้นหาจนครบทุกตัวแล้วไม่เจอ

จากวิธีการข้างต้นหากเราสามารถเขียนฟังก์ชันเองได้ดัง code ต่อไปนี้ สมมติให้ฟังก์ชันชื่อ sequentialSearch รับข้อมูลเข้า 2 ตัวคือ List ที่ต้องการค้นหาและ item ที่ต้องการหาใน List ดังกล่าว โดยฟังก์ชัน sequentialSearch จะคืนค่าเป็นบูลีนที่บ่งบอกถึงการพบ item ใน List โดยจะคืนค่าเป็นจริงหรือไม่พบ item คืนค่าเป็นเท็จ

```
def sequentialSearch(mylist, item):
    pos = 0
    found = False

    while pos < len(mylist) and not found:
        if mylist[pos] == item:
            found = True
        else:
            pos = pos+1
    return found
mylist = [4,5,23,78,21,78,90]
print(sequentialSearch(mylist, 25))
```

หากข้อมูลใน List เรียงจากน้อยไปมากแล้วเราใช้การค้นหาตามลำดับ เราจะเขียน code ให้ทำงานได้ดีขึ้นอย่างไร คำตอบถ้าเกิด item ที่เราต้องการหามีค่าน้อยกว่า mylist[pos] จะยังหาเจออยู่ไหม ทำให้เราสามารถ

ปรับปรุง code ได้ดังตัวอย่างต่อไป คือพอเรารู้ว่าตำแหน่งจากนี้ไปไม่มีข้อมูลแน่ๆ ก็หยุดการทำงาน

```
def sequentialSearch(mylist, item):
    pos = 0
    found = False
    stop = False
    while pos < len(mylist) and not found and not stop:
        if mylist[pos] == item:
            found = True
        else:
            if mylist[pos] > item:
                stop = True
            else:
                pos = pos+1
    return found
mylist = [4,5,23,78,21,78,90]
print(sequentialSearch(mylist, 25))
```

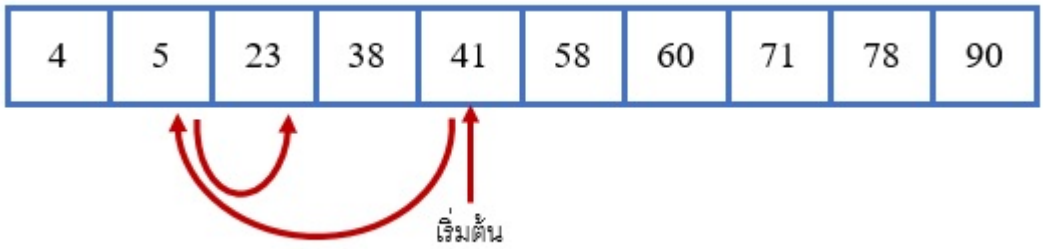
อย่างไรก็ตามกรณีที่แย่ที่สุดคือ หาไม่พบ ซึ่งทั้งสองฟังก์ชันข้างต้นก็ต้องทำงานทั้งหมด n รอบ นั่นคือต้องตรวจสอบทุกตัวนั่นเอง

4.1.2 การค้นหาแบบทวิภาค

หากเรายังจำเกมทายเลข ที่กำหนดช่วง 1 ถึง 100 มาให้แล้วเราทายเลขให้ถูก โดยใช้จำนวนครั้งน้อยที่สุดได้ ในหัวข้อย่อยนี้เราจะใช้หลักการเดียวกับเกมทายเลขนั้น หากเรารู้ว่าข้อมูลของเราเรียงลำดับอยู่แล้ว เราสามารถใช้วิธี**การค้นหาแบบทวิภาค**มาช่วยให้การค้นหาเร็วขึ้นได้ เริ่มต้นพิจารณาตำแหน่งตรงกลางก่อน ถ้าตำแหน่งตรงกลางไม่ใช่ข้อมูลที่เราต้องการหา เราจะได้ข้อมูลเพิ่มคือ ข้อมูลที่เราต้องการหาอยู่ในกลุ่มทางซ้ายมือของข้อมูลตัวตรงกลาง หรือข้อมูลที่เราต้องการหาอยู่ในกลุ่มทางขวามือของข้อมูลตัวตรงกลาง จากนั้นเราก็ไปตรวจสอบตรงกลางของกลุ่มนั้น ทำเช่นนี้ต่อไปจนพบข้อมูล การทำเช่นนี้ทำให้เราสามารถตัดข้อมูลทิ้งไปได้ครึ่งหนึ่งที่ไม่ใช่คำตอบของเราแน่ๆ สิ่งที่เราต้องทำก็เพียงแค่พิจารณาข้อมูลที่เหลือซึ่งเราก็พิจารณาเช่นนี้ไปเรื่อยๆ จนพบข้อมูลหรือไม่มีข้อมูลให้ค้นหาต่อซึ่งคือไม่พบข้อมูลนั่นเอง

ตัวอย่างฟังก์ชัน binarySearch

```
def binarySearch(mylist, item):
    first = 0
    last = len(mylist)-1
```

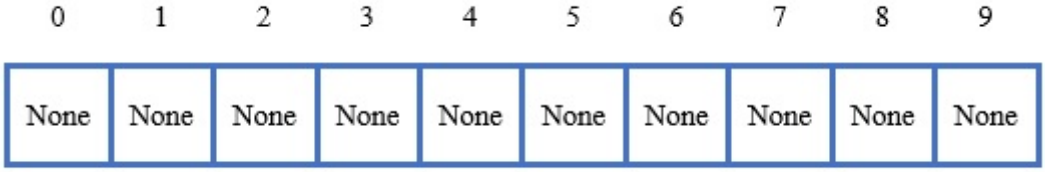


รูปที่ 4.2: ตัวอย่างการทำงานของการค้นหา 23 แบบทวิภาคใน List

```
    found = False

    while first<=last and not found:
        mid = (first + last)//2
        if mylist[mid] == item:
            found = True
        else:
            if mylist[mid] > item:
                last = mid—1
            else:
                first = mid+1
    return found
mylist = [4,5,23,38,41,58,60,71,78,90]
print(binarySearch(mylist, 23))
```

การแบ่งข้อมูลออกเป็นสองส่วนแล้วเลือกทำส่วนนึงที่เราคิดว่ามีคำตอบ เมื่อเราพบคำตอบของส่วนเล็กเราจะนำมาเป็นคำตอบของข้อมูลตั้งต้น นี่เป็นตัวอย่างนึงของการแก้ปัญหาแบบแบ่งแยกและเอาชนะ(Divide and conquer) โดยการแบ่งแยกและเอาชนะนี้ หมายถึงแบ่งปัญหาออกเป็นปัญหาย่อยๆ ที่มีขนาดของปัญหาเล็กลง จากนั้นแก้ปัญหาที่มีขนาดเล็กลงนี้ แล้วนำกลับมาสร้างเป็นคำตอบของปัญหาตั้งต้น เราสามารถเขียนด้วย Recursive ได้ดังนี้

```
def binarySearch(mylist, item):
    if len(mylist) == 0:
        return False
    else:
        mid = len(mylist)//2
        if mylist[mid] == item:
```



รูปที่ 4.3: ตัวอย่างตารางแฮชที่มี 10 ช่อง

```
            return True
        else:
            if mylist[mid] > item:
                return binarySearch(mylist[:mid], item)
            else:
                return binarySearch(mylist[mid+1:], item)

mylist = [4,5,23,38,41,58,60,71,78,90]
print(binarySearch(mylist, 23))
```

เวลามากที่สุดในการค้นหาข้อมูลคือ ค้นหาแล้วไม่พบข้อมูล คำถามเราต้องค้นหากี่รอบ นั่นคือ เราแบ่งขอ้มูลออกเป็นสองส่วน จนไม่เหลือตัวเลยกี่ครั้งนั่นเอง จาก n -> $\frac{n}{2}$ -> $\frac{n}{4}$ -> ... -> $\frac{n}{n}$ นั่นคือ ไม่เกิน $\log(n) + 1$ รอบนั่นเอง

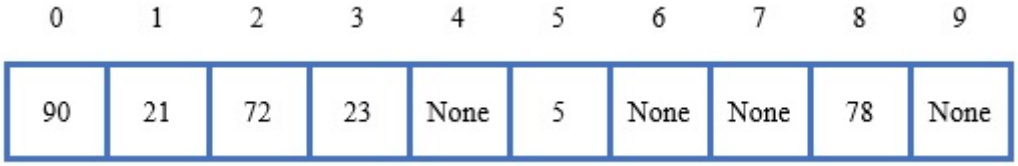
4.1.3 การแฮช

ก่อนหน้านี้เราได้พิจารณาการค้นหา ซึ่งหากข้อมูลของเราเรียงลำดับเราจะมีขั้นตอนวิธีที่ช่วยให้การค้นหาเร็วขึ้น ในหัวข้อย่อยนี้เราจะมาพิจารณาโครงสร้างข้อมูลที่ทำให้การค้นหาเร็วขึ้นไปอีกเป็นค่าคงที่ นั่นคือการแฮช (Hashing)

ในการทำให้ได้ความเร็วในการค้นหาเช่นนี้ เราต้องรู้ว่าข้อมูลเก็บที่ไหนแล้วเราจะไปค้นหาตรงนั้นได้อย่างไร ถ้าทุกๆ ข้อมูลอยู่ในที่ที่ควรอยู่เราจะใช้การเปรียบเทียบเพียงครั้งเดียวก็ทำให้รู้ว่าข้อมูลอยู่หรือไม่อยู่

ตารางแฮช(Hash table) เป็นกลุ่มของข้อมูลที่ถูกเก็บในวิธีที่ง่ายต่อการค้นหาภายหลัง แต่ละตำแหน่งของตารางแฮชเราจะเรียกว่า slot ซึ่งจะเก็บข้อมูลและตั้งชื่อด้วยเลขจำนวนเต็มเริ่มต้นที่ 0 ตัวอย่างเช่น เรามี slot หมายเลข 0, slot หมาบเลข 1, slot หมาบเลข 2 ไปเรื่อยๆ เริ่มต้นตารางแฮชไม่มีข้อมูลเลขดังนั้นทุก slot จะว่าง เราสามารถสร้างตารางแฮชได้โดยการใช้ list ที่เริ่มต้นข้อมูลใน list เก็บค่า None ไว้ก่อน รูปที่ เป็นตารางแฮชที่มีขนาด m = 10 หรือมองง่ายๆ คือ เรามีตาราง m ช่องที่ช่องมีหมายเลขขกำกับ 0 ถึง 9

อีกส่วนหนึ่งเป็นการจับคู่ระหว่างข้อมูลที่จะเก็บกับ slot ในตารางแฮช เราจะใช้ฟังก์ชันแฮช(Hash function) ฟังก์ชันแฮชจะรับข้อมูลใดๆ มาจากนั้นจะแปลงเป็นเลขจำนวนเต็มในช่วง 0 ถึง m-1



รูปที่ 4.4: ตัวอย่างตารางแฮชที่มี 10 ช่อง

สมมติว่าเรามีเซตของจำนวนเต็ม 5,23,78,21,72 และ 90 วิธีการจับคู่ข้อมูลกับหมายเลข slot ในตารางแฮชแบบแรกที่จะยกตัวอย่างคือ วิธีการเศษเหลือ(Remainder method) วิธีนี้จะเอาข้อมูลหารด้วยขนาดของตารางแฮชเลย(h(item) = item%10) เศษที่เหลือจากการหารก็จะเป็นหมายเลขช่องของข้อมูลนั้น ตารางที่4.1 เป็นตัวอย่างของค่าแฮชที่ได้จากเซตของจำนวนเต็มก่อนหน้า

ตารางที่ 4.1: ตัวอย่างวิธีการเศษเหลือ (h(item) = item%10)

ข้อมูล	**ค่าแฮช(Hash value)**
5	5
23	3
78	8
21	1
72	2
90	0

จากตารางที่4.1 ตัวอย่างที่สอง 23 %10 เหลือเศษ 3 นั่นเอง เมื่อคำนวณค่าแฮชของข้อมูลแล้วแล้วเราจะใส่ข้อมูลลงในตารางแฮชในช่องของค่าแฮช ดังรูปที่4.4

จากตัวอย่างเราพบว่ามีช่องที่ถูกใช้งาน 6 ช่องจาก 11 ช่อง อัตราส่วนระหว่างสองค่านี้เรียกว่า Load factor แทนด้วย $\lambda = \frac{numberofitem}{tablesize}$ ซึ่งสำหรับตัวอย่างข้างต้นนี้คือ $\lambda = 0.6$

เมื่อเราเก็บของได้แล้ว ต่อไปเวลาที่เราจะค้นหาของ วิธีการง่ายมากเราก็เพียงแค่ใช้ฟังก์ชันแฮชในการคำนวณหาช่องที่ต้องการในตารางแฮชนั่นเอง จากนั้นก็เพียงแค่ตรวจสอบว่าในช่องนี้มีค่าที่เราต้องการ หรือไม่มีของเก็บไว้เลย ซึ่งในการหาค่านี้ใช้เวลาเป็นค่าคงที่เพราะว่าเป็นการคำนวณ

วิธีการนี้จะทำงานได้อย่างไม่มีปัญหาเมื่อข้อมูลแต่ละตัวได้ช่องที่แตกต่างกันหมด พิจารณาตัวอย่างต่อไปนี้ ถ้ามีข้อมูลใหม่ที่ต้องการเก็บไว้ในตารางแฮช เป็นค่า 33 ค่าแฮชที่ได้คือ 3(33%10==3) เนื่องจากว่าก่อนหน้านี้มี 23 อยู่ก่อนแล้ว ดังนั้นเกิดปัญหาขึ้นแล้วจะทำอย่างไรดี เนื่องจากว่ามีข้อมูลที่มีค่าแฮชเดียวกันมากกว่าหนึ่งตัว เหตุการณ์เช่นนี้เรียกว่า การชน(Collision) ซึ่งเราจะมาแก้ปัญหานี้กันภายหลัง

ฟังก์ชันแฮช

เมื่อกำหนดกลุ่มของข้อมูลมาให้ ฟังก์ชันแฮชที่จับคู่ข้อมูลแต่ละตัวลงให้ช่องว่างของตารางแฮชได้จะถูกเรียกว่าฟังก์ชันแฮชสมบูรณ์(Perfect hash function) แต่ที่แย่คือเมื่อกำหนดกลุ่มของข้อมูลใดๆ มาการสร้าง Perfect hash function นั้นทำได้ยาก แต่มันก็ไม่ได้แย่มากเพราะว่าเราไม่จำเป็นต้องสร้างฟังก์ชันแฮชสมบูรณ์ให้ได้แล้วเรายังได้ประสิทธิภาพในการทำงานที่ดีอยู่

วิธีหนึ่งที่ถูกใช้บ่อยในการสร้าง Perfect hash function คือการเพิ่มขนาดของตารางแฮชเพื่อที่ว่าข้อมูลแต่ละจะได้ค่าที่ไม่ซ้ำกัน แต่การเพิ่มขนาดตารางนี้ทำได้กับข้อมูลจำนวนไม่มาก แต่ไม่เหมาะกับข้อมูลขนาดใหญ่เพราะว่าต้องจองเนื้องที่ตารางใหญ่ตามไปด้วยนั่นเอง อย่างเช่นเราจะเก็บข้อมูลนักเรียนในห้องเรียน เราใช้เลขบัตรประชาชน 13 หลัก เราก็จองตารางแฮชที่รองรับเลข 13 หลักเลยซึ่งไม่ชนแน่นอน แต่ถ้านักเรียนในห้องมีเพียง 30 คน เป็นการจองเนื้อที่ที่สิ้นเปลืองมาก

จุดมุ่งหมายของเราคือการสร้างฟังกันแฮชที่มีการชนกันน้อย ง่ายต่อการคำนวณและกระจายข้อมูลในตารางแฮช ต่อไปจะเสนอวิธีซึ่งทำต่อจากวิธีการเศษเหลือ

วิธีแรก Folding method เริ่มต้นด้วยแบ่งข้อมูลเป็นกลุ่มให้แต่ละกลุ่มมีขนาดเท่าๆ กัน(กลุ่มสุดท้ายอาจจะไม่เท่ากันได้) แต่ละกลุ่มจะถูกบวกรวมกันเพื่อให้ได้ค่าแฮช ตัวอย่างเช่น ถ้าเรามีหมายเลขโทรศัพท์ 053943411 เรานำเบอร์โทรนี้มาแบ่งกลุ่ม สมมติว่ากลุ่มละ 2 ตัว จะได้ 05, 39, 43, 41, 1 หลังจากนั้นเอามาบวกกันจะได้ 05 + 39 + 43 + 41 + 1 ซึ่งจะได้ 129 ถ้าสมมติว่าตารางแฮชมี 13 ช่อง เราก็จะเอามาหารเก็บเศษด้วย 13 ซึ่งจะได้ 129%13= 12 ดังนั้นในกรณีนี้หมายเลขโทรศัพท์นี้จะถูกเก็บในช่อง 12

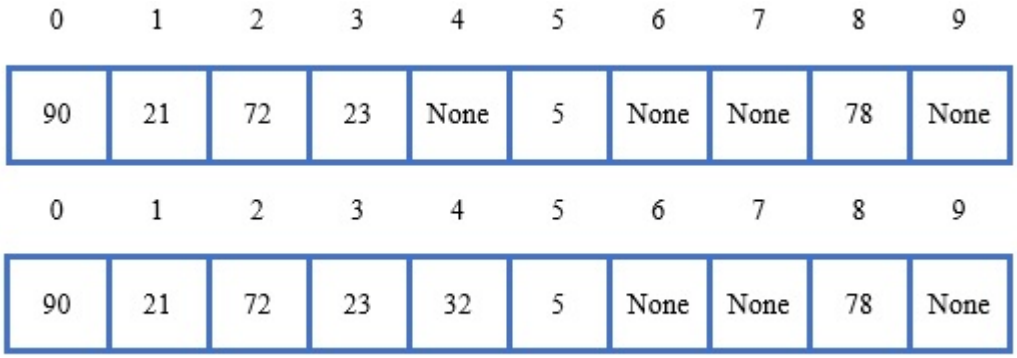
วิธีที่สอง Mid-square method เริ่มต้นด้วยนำข้อมูลยกกำลังสองจากนั้นสกัดเอาบางส่วนมาใช้ ตัวอย่างเช่น สมมติเรามีข้อมูล 39 เริ่มต้นยกกำลังสองจะได้ $39^2 = 1521$ สมมติว่าสกัดเอาข้อมูลสองตัวตรงกลางจะได้ 52 จากนั้นนำเอาค่านี้ไปใช้ต่อ ถ้าสมมติว่าตารางแฮชมี 13 ช่อง ก็จะได้ 0(52%13==0)

ตัวอย่างก่อนหน้านี้ ข้อมูลของเราเป็นจำนวน หากข้อมูลเข้าของเราเป็นตัวหนังสือ สมมุติเป็นคำว่า "god" เราจะทำอย่างไร วิธีการหนึ่งคือเราทำการแปลงแต่ละอักขระให้เป็นตัวเลข จากนั้นนำมาดำเนินการต่อด้วยวิธีการบางอย่าง เช่นบวกรวมกัน ทั้งนี้ใน ภาษาไพธอนมีคำสั่ง ord ที่ทำการแปลงอักขระเป็นตัวเลข ตัวอย่างเช่น

```
ord('g')        #103
ord('o')        #111
ord('d')        #100
```

เราสามารถเอาค่า Ordinal ของอักขระนั้นๆ มารวมกันแล้วใช้ Remainder method เพื่อให้ได้ค่าแฮชตัวอย่างเช่น 'god' ทำการแปลงแต่ละอักขระด้วยคำสั่ง ord แล้วนำมารวมกันจะได้ค่า 103+111+100= 314 จากนั้นนำเอาค่านี้ไปใช้ต่อ ถ้าสมมติว่าตารางแฮชมี 13 ช่อง ก็จะได้ 2(314%13==2) ต่อไปเป็นตัวอย่างฟังก์ชันชื่อ hash ที่รับข้อความและขนาดของตารางแฮชและคืนค่าเป็นเลขในช่วง 0 ถึง(ขนาดตาราง -1)

```
def hash(my_string,table_size):
    sum = 0
    for pos in range(len(my_string)):
```



รูปที่ 4.5: เทคนิค open addressing ด้วยวิธี linear probing เมื่อเพิ่มข้อมูล 32

```
        sum = sum + ord(my_string[pos])
    return sum% table_size
```

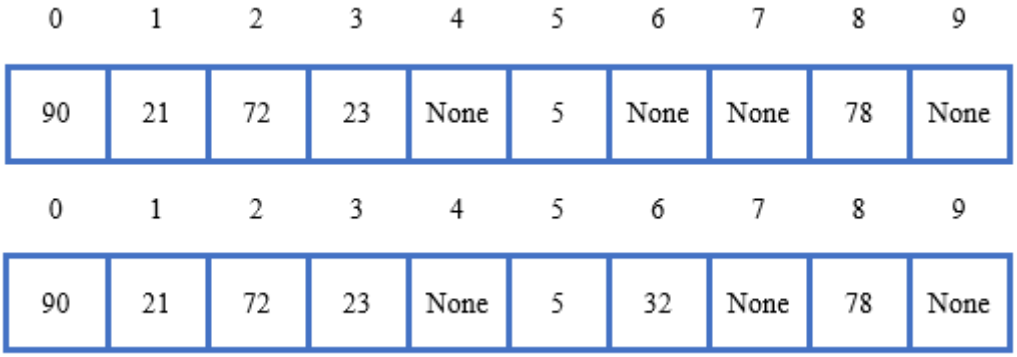
แต่การทำเช่นนี้ สิ่งที่เราพบคือ 'god' และ 'dog' ได้ค่าเท่ากันเนื่องจากว่าใช้อักขระชุดเดียวกัน วิธีการแก้ก็เพิ่มค่าน้ำหนักให้กับตำแหน่งของอักขระ ตัวอย่างเช่น ให้ค่าน้ำหนักมีค่าเท่ากับลำดับที่ของอักขระ ดังนั้น 'god' จะได้ 103*1+111*2+100*3 = 625 ในขณะที่ 'dog' จะได้ 100*1+111*2+103*3 = 631 ทั้งนี้เราอาจจะสร้างวิธีการขึ้นมาเองก็ได้ โดยสิ่งสำคัญคือฟังก์ชันแฮชจะต้องมีประสิทธิภาพทั้งในแง่เนื้อที่ และความเร็วในการค้นหา

การแก้ปัญหาการชนกัน

เมื่อมีข้อมูลสองตัวที่ผ่านฟังก์ชันแฮชแล้วได้ค่าช่องเดียวกันในตารางแฮช เราต้องมีวิธีการที่เป็นระบบชัดเจน ในการจัดการข้อมูลตัวที่สองนี้ว่าจะเห็บไว้ที่ใดในตารางแฮชเพื่อที่เราจะได้ค้นหาได้พบด้วย วิธีการนี้เรียกว่า การแก้ปัญหาการชนกัน หรือ Collision resolution

วิธีการหนึ่งในการแก้ปัญหาการชนกันคือการดูในตารางแฮชและพยายามหาช่องว่างช่องอื่นเพื่อที่จะเก็บของที่ชนกัน วิธีการที่ง่ายวิธีหนึ่งคือเริ่มต้นที่ช่องที่แฮชมาได้จากนั้นขยับไปช่องถัดไปเพื่อหาช่องว่าง ทำเช่นนี้ไปเรื่อยๆ ซึ่งพอเราไปถึงช่องสุดท้ายของตารางแฮชแล้วการขยับครั้งแต่ไปก็ต้องไปที่ช่องแรก เหมือนขยับวนเป็นวงกลมนั่นเอง วิธีการทำเช่นนี้เรียกว่า Open addressing คือการหาช่องว่างถัดไป และวิธีการที่เข้าไปสำรวจทีละช่องว่าว่างไหม เป็นเทคนิค Open addressing ที่เรียกว่า Linear probing ตัวอย่างเช่น หาข้อมูลของเรามีค่า 5,23,78,21,72 และ 90 แล้วใช้วิธี Remainder method ด้วยขนาดตาราง 10 ช่องซึ่งหากมีข้อมูล 32 ต้องการเก็บในตารางแฮช วิธีการ Linear probing เราจะดูช่องว่างช่องถัดไป จะได้ว่าเราจะเก็บค่า 32 ที่ช่อง 4

เมื่อเราสร้างตารางแฮชโดยใช้ Open addressing และ Linear probing เราก็ต้องมาจัดการวิธีการค้นหาของด้วย สมมติว่าเราจะหาข้อมูล 16 เราก็เพียงคำนวณว่า 16 ควรอยู่ช่องไหนจะได้ว่าควรอยู่ช่อง 6 แต่ว่าช่อง 6 ว่างแสดงว่าไม่มีข้อมูลก็คืนค่า False เช่นกันถ้าหากจะหาข้อมูล 21 ก็เพียงคำนวณว่า 21 ควรอยู่ช่องไหน เราพบว่าอยู่ช่อง 1 และช่อง 1 มีค่าเป็น 21 แสดงว่าคืนค่า True หากเราต้องการหาค่า 32 เราพบว่าค่าแฮชของ 32 คือ 2 แต่ในช่อง 2



รูปที่ 4.6: linear probing ที่ขยับไป 4 ช่อง เมื่อเพิ่มข้อมูล 32

มีค่า 72 อยู่ เราควรจะคืนค่า False เลยหรือไม่ คำตอบคือไม่ เพราะว่ามีค่าอื่นอยู่แสดงว่าเกิดการชนกัน ดังนั้นเราต้องกาต่อเราก็ทำการหาต่อตามลำดับทีละช่องเราก็พบ 32 ในช่องที่ 4 คำถามเราจะรู้ได้อย่างไรว่าต้องหาต่อไปจนถึงเมื่อไร คำตอบเราจะหาต่อไปจนช่องที่หาเป็น None เนื่องจากว่าการที่เป็น None ได้แสดงว่าไม่มีค่าที่ซ้ำมาใส่ต่อได้นั่นเอง

ข้อเสียของ Linear probing คือการเกาะกลุ่มกันของข้อมูล(Clustering) นั่นคือถ้ามีการชนกันเกิดขึ้นด้วยค่าเดียวกันมากๆ ผลที่ได้คือข้อมูลจะติดกันในตารางแฮช วิธีจัดการวิธีหนึ่งคือปรับ linear probing โดยแทนที่จะขยับทีละช่องต่อเนื่องกัน เราจะมีการข้ามช่องบ้าง เช่นการขยับดูช่องต่อไปให้ขยับไปทีละ 4 ช่อง การทำเช่นนี้จะลดการเกิด clustering ได้

ชื่อทั่วไปของกระบวนการหาช่องว่างช่องอื่นหลังจากการชนคือ Rehashing ดังนั้น Linear probing แบบธรรมดา Rehash function คือ

new_hash_value = rehash(old_hash_value)

เมื่อ

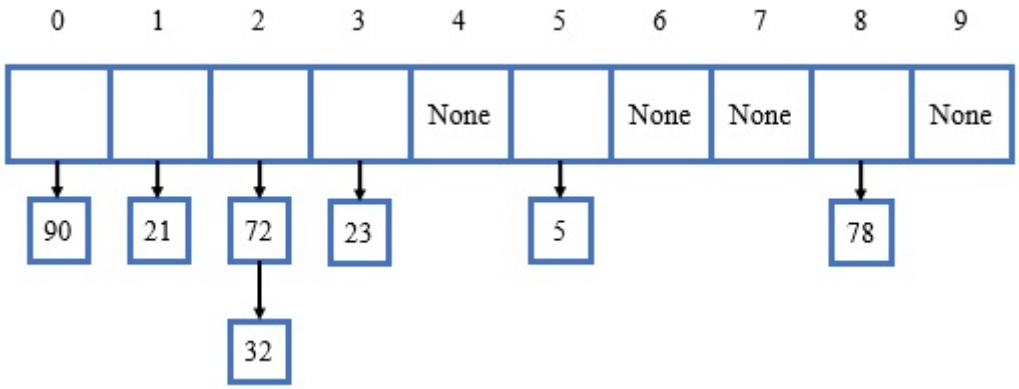
rehash(pos) = (pos+1)%size_of_table

ส่วนการที่เรา ’+4’ สามารถเขียนได้ดังนี้

rehash(pos) = (pos+4)%size_of_table

หรือโดยทั่วไปเขียนได้ดังนี้

rehash(pos) = (pos+skip)%size_of_table



รูปที่ 4.7: เทคนิค chaining

โดยที่ขนาดของ skip จะต้องทำให้แต่ละช่องนั้นในตารางแฮชถูกเข้าถึงได้ถ้าไม่เช่นนั้นบางช่องอาจจะไม่ถูกเข้าถึงได้ ก่อนหน้านี้ได้ยกตัวอย่างขนาดตารางเป็นเลขที่ทำให้คำนวณง่ายๆ แต่จริงๆ แล้วเพื่อให้ทุกช่องถูกเข้าถึงได้จะแนะนำให้ใช้เลขจำนวนเฉพาะ

นอกจาก Linear probing แล้วยังมี Quadratic probing โดยแทนที่จะใช้ค่าคงที่ในการ skip เราจะใช้ Rehash function ที่เพิ่มค่าแฮชด้วย 1,4,9,16, ... นั่นคือ ถ้าค่าแฮชตัวแรกเป็น h ค่าถัดไปจะเป็น h+1, h+4,h+9,h+16 ไปเรื่อยๆ

อีกวิธีที่จัดการการชนกันคือ หากชนกันแล้วสร้างสายข้อมูลของ slot นั้นต่อกันไป วิธีนี้เรียกว่า Chaining ตัวอย่างเช่น 32 ชนกัน 72 จะสร้าง Linked list ของข้อมูล 32 ต่อ 72 ดังนั้นเมื่อต้องการค้นหาข้อมูลก็จะต้องไปยังช่องที่ถูกต้องจากนั้นทำการค้นหาในสายข้อมูลของช่องนั้นว่ามีข้อมูลที่ต้องการหรือไม่

การสร้าง Map Abstract Data Type

หนึ่งในชนิดข้อมูลแบบกลุ่มของภาษาไพธอน คือ Dictionary ซึ่งเก็บ Key และ Data โดย Key จะเป็นส่วนที่เอาไว้ค้นหาค่าที่สอดคล้องกัน โดยทั่วไปแนวคิดเช่นนี้เรียกว่า Map

Map:abstract data type

นิยามได้ดังนี้ Map เป็นกลุ่มของข้อมูลที่ไม่มีลำดับที่มีความสัมพันธ์กันระกว่าง Key และ Data โดย Key ใน Map นั้นไม่ซ้ำกันโดยจะมีความสัมพันธ์แบบ หนึ่งต่อหนึ่ง ระหว่าง Key กับ Data value การดำเนินการมีดังนี้

- `Map()` สร้าง Map ว่างอันใหม่คืนค่าเป็น Map ว่าง
- `put(key,value)` เพิ่ม Key-Value อันใหม่เข้าไปใน Map ถ้ามี Key นี้แล้วให้แทนที่ค่าเก่าของ Value ด้วยค่าใหม่
- `get(key)` คืนค่า Value ของ Key นั้นที่เก็บใน Map หรือคืน None ในกรณีอื่น

- **del** ลบ Key-Value จาก Map โดยใช้คำสั่ง del map[key]
- **len()** คืนค่าจำนวน Key-Value ที่เก็บใน Map
- **in** คืนค่า True จากคำสั่ง key in Map ถ้ามี key ดังกล่าวเก็บไว้ใน Map และคืนค่า False ในกรณีอื่น

หนึ่งในข้อดีของ Dictionary คือเมื่อให้ Key มาเราสามารถค้นหาข้อมูลได้อย่างรวดเร็ว ซึ่งในการที่จะทำให้ค้นหาได้เร็วนั้น เราต้องสร้างการค้นหาที่มีประสิทธิภาพ เราสามารถใช้ list ที่ใช้การค้นหาตามลำดับหรือการค้นหาหวิภาคได้ แต่การใช้ตารางแฮชที่อธิบายไปในหัวข้อก่อนหน้านั้นมีประสิทธิภาพดีกว่า

ส่วนต่อไปเป็นการสร้าง class HashTable ซึ่งสร้างโดย Map abstract data type โดยจะมี List 1 ตัวที่เราเรียกว่า slots ซึ่งจะเป็นที่เก็บ Key และมี List แบบขนานที่เรียกว่า Data เอาไว้เก็บข้อมูล เมื่อเราทำการค้นหา Key ตำแหน่งที่สอดคล้องกันของ Data list จะเก็บค่าที่สอดคล้องกัน เราจะทำ Key list เป็น Hash table โดนใช้แนวคิดก่อนหน้า ขนาดของ Hash table จะเป็นเลขจำนวนเฉพาะเพื่อที่จะได้แก้ปัญหาเรื่องการชนไปด้วย

```
class HashTable:
    def __init__(self):
        self.size = 11
        self.slots = [None] * self.size
        self.data = [None] * self.size
```

เราจะสร้าง Hash function โดยใช้ Remainder method และการชนกันจะใช้วิธี Linear probing ด้วย '+1' เป็น Rehash function ฟังก์ชัน put จะสมมติว่าเป็นช่องว่างถ้าไม่เช่นนั้นจะต้องมีค่าใน self.slots มันจะคำนวนด้วยค่าแฮชเริ่มต้นและถ้าไม่ว่างจะมีการเรียก Rehash

```
def put(self, key, data):
    hash_value = self.hash_function(key,len(self.slots))
    if self.slots[hash_value] == None:
        self.slots[hash_value] = key
        self.data[hash_value] = data
    else:
        if self.slots[hash_value] == key:
            self.data[hash_value] = data # replace
        else:
            next_slot = self.rehash(hash_value,len(self.slots))
            while self.slots[next_slot] != None and \
                        self.slots[next_slot] != key:
                next_slot = self.rehash(next_slot,len(self.slots))
            if self.slots[next_slot] == None:
```

```
                self.slots[next_slot] = key
                self.data[next_slot] = data
            else:
                self.data[next_slot] = data #replace

def hash_function(self, key, size):
    return key % size

def rehash(self, old_hash, size):
    return (old_hash + 1) % size
```

ฟังก์ชัน get เริ่มต้นจะคำนวณค่าแฮชเริ่มต้นให้ ถ้าค่าที่ต้องการนั้นไม่ใช่ค่าแฮชเริ่มต้นจะเรียก Rehash เพื่อหาตำแหน่งต่อไป ซึ่งการค้นหาจะถูกยกเลิกโดยการตรวจสอบว่าเราจะไม่วนกลับมาที่ค่าแฮชเริ่มต้น ถ้าเกิดขึ้นเช่นนั้นแสดงว่าเราได้ค้นหาทุกช่องแล้วและข้อมูลที่ต้องการไม่อยู่ในตารางแฮช

อีกสอง Method ใน HashTable ที่มีเพิ่มให้คือ **__getitem__** และ **__setitem__** เป็น Method ที่อนุญาตให้เข้าถึงข้อมูลได้โดยใช้เครื่องหมาย '[]' ส่วน Method อื่นให้ทำเป็นแบบฝึกหัด

```
def get(self, key):
    start_slot = self.hash_function(key, len(self.slots))
    data = None
    stop = False
    found = False
    position = start_slot
    while self.slots[position] != None and \
                    not found and not stop:
        if self.slots[position] == key:
            found = True
            data = self.data[position]
        else:
            position=self.rehash(position, len(self.slots))
            if position == start_slot:
                stop = True
    return data

def __getitem__(self, key):
    return self.get(key)
```

```
def __setitem__(self, key, data):
    self.put(key, data)
```

ต่อไปเป็นตัวอย่างการเรียกใช้

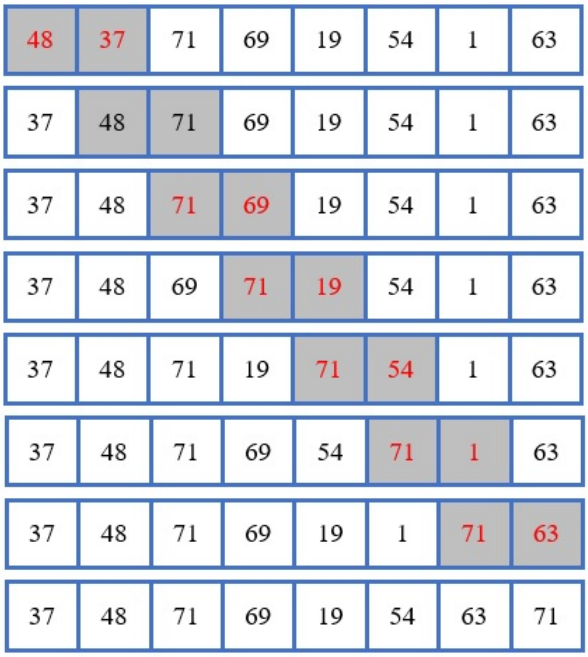
```
h=HashTable()
h[5]="Chiang Mai"
h[23]="Nan"
h[78]="Bangkok"
h[21]="Phuket"
h[72]="Saraburi"
h[90]="Lampang"
h[32]="Tak"
print(h.slots)
#[32, 23, 78, 90, None, 5, 72, None, None, None, 21]
print(h.data)
#['Tak', 'Nan', 'Bangkok', 'Lampang', None, 'Chiang Mai',
#'Saraburi', None, None, None, 'Phuket']
```

4.2 การเรียงลำดับ

การเรียงลำดับเป็นกระบวนการของการวางตำแหน่งสมาชิกจากกลุ่มของมันในลำดับบางอย่าง ตัวอย่างเช่นเราเรียงกลุ่มของคำตามตัวอักษร หรือตามความยาว ชื่อจังหวัดถูกเรียงตามจำนวนประชากรของจังหวัดนั้น หรือตามภูมิภาค เป็นต้น ในปัจจุบันมีขั้นตอนวิธีหลายอย่างในการเรียงลำดับนั่นแสดงให้เห็นว่าปัญหาการเรียงเป็นหนึ่งในปัญหาสำคัญทางด้านวิทยาการคอมพิวเตอร์ ซึ่งในบทนี้เราจะได้แนะนำขั้นตอนวิธีในการเรียงข้อมูลหลายตัวด้วยกัน

4.2.1 Bubble sort

Bubble sort หลักการคือ สมมติเราจะเรียงข้อมูลจากน้อยไปมาก ในรอบแรก ข้อมูลตัวแรกจะถูกเปรียบเทียบข้อมูลตัวที่ติดกัน ว่าลำดับตัวแรกกับตัวที่ติดกันถูกแล้วหรือไม่ ถ้าไม่ให้สลับกัน จากนั้นจึงไปตรวจสอบตัวที่สองในลักษณะนี้ไปเรื่อยๆ สังเกตว่าเมื่อเปรียบเทียบ 1 คู่ข้อมูลมากสุดจะอยู่ทางขวา เมื่อทำจนถึงตัวสุดท้ายข้อมูลตัวสุดท้ายจะมีค่ามากที่สุด นั่นหมายถึงข้อมูลอยู่ถูกตำแหน่งแล้วที่ตำแหน่งสุดท้าย จากนั้นเราก็ทำการสลับในรอบที่สองจะได้ข้อมูลตัวรองสุดท้ายเป็นค่ามากสุดจากข้อมูลที่เหลือ(ตัวสุดท้ายมากสุดแล้ว) ทำเช่นนี้จนครบ n รอบก็จะได้ข้อมูลที่เรียงกันหมด



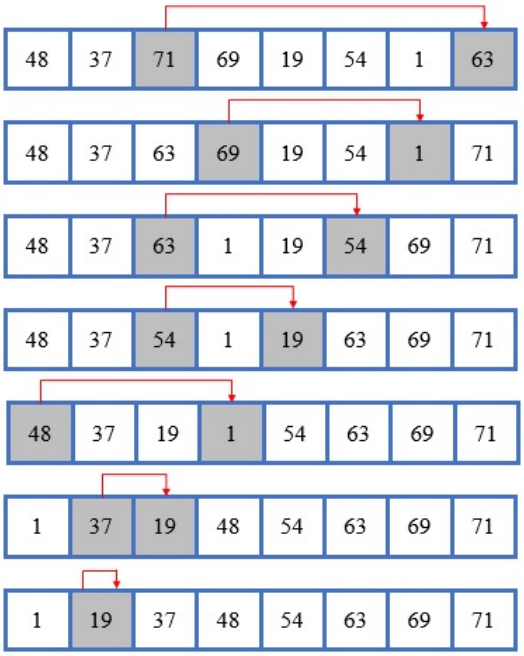
รูปที่ 4.8: ตัวอย่าง Bubble sort ในรอบแรก

เมื่อมีเลข 48, 37, 71, 69, 19, 54, 1, 63 รูปที่4.8 เป็นตัวอย่างการทำงานในรอบแรก เริ่มต้น 48 และ 37 เปรียบเทียบกันในรูปคู่ที่เทียบกันแสดงด้วยสีเทา พบว่าผิดลำดับแสดงด้วยสีแดงในแถวที่ 1 ดังนั้นจะสลับ 48 และ 37 จากนั้นในการเทียบครั้งที่สอง 48 และ 71 เทียบกันพบว่าถูกตำแหน่งแล้ว ดังนั้นไม่มีการสลับ ในการเทียบครั้งที่สาม 71 และ 69 พบว่าผิดลำดับ จึงต้องสลับตำแหน่งกัน ทำเช่นนี้จนเทียบตัวสุดท้าย หลังการทำพบว่า 71 ซึ่งเป็นตัวเลขที่มากที่สุดจะถูกย้ายไปขวามือสุด

ในรอบที่สอง ค่ามากสุดอยู่ถูกที่แล้วคือตำแหน่งขวาสุด ดังนั้นจะมีข้อมูลเพียง n-1 ตัวที่เราต้องการเรียง แสดงว่าเราต้องเปรียบเทียบเพียง n-2 ครั้งก็พอ เพื่อที่จะทำให้ตัวที่มากที่สุดจาก n-1 ตัวนั้นถูกย้ายไปอยู่ตำแหน่งตัวที่สองจากทางขวามือ ซึ่งกลักการนี้ก้จะถูกใช้กับรอบต่อๆ ไป

ต่อไปเป็นตัวอย่าง code ของ Bubble sort

```
def bubbleSort(mylist):
    for num_remain in range(len(mylist)-1,0,-1):
        for i in range(num_remain):
            if mylist[i]>mylist[i+1]:
                temp = mylist[i]
                mylist[i] = mylist[i+1]
                mylist[i+1] = temp
mylist=[48, 37, 71, 69, 19, 54, 1, 63]
```



รูปที่ 4.9: ตัวอย่าง Seletion sort

```
bubbleSort(mylist)
print(mylist)
```

สังเกตุว่า code ของ Bubble sort นั้นง่ายมาก แต่ Bubble sort นี้ยังสามารถเพิ่มประสิทธิภาพได้อีก หากข้อมูลเรียงแล้ว ถ้าหากใช้ Bubble sort ข้างต้นก็ยังมีการเปรียบเทียบอยู่ ให้ทำเป็นแบบงึกหัดว่าเราจะทำการลดการเปรียบเทียบได้อย่างไร

4.2.2 Selection sort

สังเกตได้อย่างหนึ่งว่า Bubble sort นั้นมีการสลับตำแหน่งบ่อยมาก Selection sort พัฒนาจาก Bubble sort โดยที่สลับตำแหน่งเพียงครั้งเดียวในแต่ละรอบ หลักการของ Selection sort นั้นไม่ยากเช่นกันคือ ในแต่ละรอบจะหาค่าที่มากที่สุด พอครบรอบแล้วก็ย้ายตัวที่มากที่สุดนี้ไปยังตำแหน่งที่เหมาะสม ซึ่งคล้ายกับ Bubble sort คือพอครบรอบตัวมากสุดย้ายไปอยู่ตำแหน่งที่เหมาะสม เราจะทำทั้งหมด n-1 รอบในการเรียงเลข n ตัว เนื่องจากรอบสุดท้ายเหลือเลข 1 ตัวและอยู่ถูกตำแหน่งแล้วนั่นเอง

จากรูปตัวอย่างที่ 4.9 แสดงให้เห็นถึงการทำงานของ Selection sort โดยในแต่ละรอบจะหาค่ามากที่สุดจากนั้นจะสลับกับตำแหน่งสุดท้าย ในรอบแรก 71 ถูกสลับที่กับตำแหน่งสุดท้าย พอเสร็จรอบแรกในรอบที่สองเราจะหาตัวที่มากที่สุดของ n-1 ตัวมาแทนในช่องรองสุดท้าย พบเสร็จรอบที่สองจะพบว่าตัวรองสุดท้ายและตัวสุดท้ายอยู่ถูก

ตำแหน่งแล้ว ทำเช่นนี้จนครบ n-1 รอบ

สำหรับ code ของ Selection sort เขียนได้ดังนี้

```
def selectionSort(mylist):
    for num_remain in range(len(mylist)-1,0,-1):
        maxpos=0
        for location in range(1,num_remain+1):
            if mylist[location]>mylist[maxpos]:
                maxpos = location
        temp = mylist[num_remain]
        mylist[num_remain] = mylist[maxpos]
        mylist[maxpos] = temp
mylist=[48, 37, 71, 69, 19, 54, 1, 63]
selectionSort(mylist)
print(mylist)
```

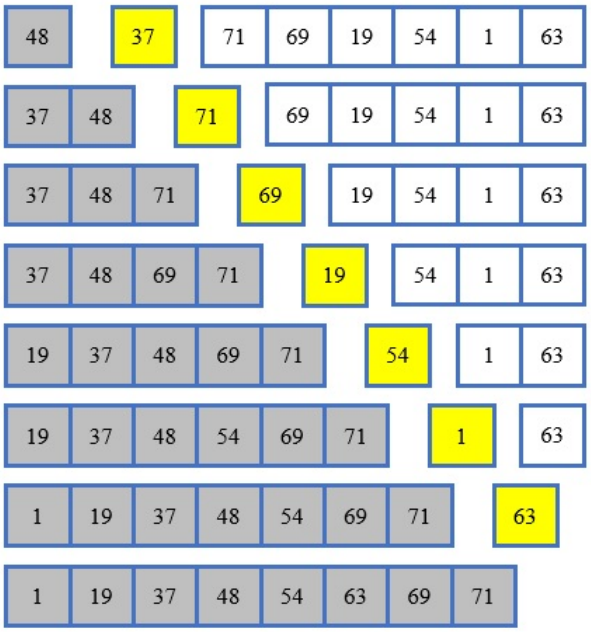
4.2.3 Insertion sort

เป็นการเรียงข้อมูลที่ถูกใช้มากตอนเล่นไพ่ พอเราได้รับไพ่ใบใหม่มาเราจะนำเอาไพ่ใบใหม่นี้มาหาตำแหน่งที่เหมาะสมจากไพ่ในมือก่อนหน้า หลักการคือเราจะเก็บกลุ่มของข้อมูลที่เรียงแล้วไว้ จากนั้นเมื่อมีข้อมูลใหม่จะนำไปแทรกกับข้อมูลที่เรียงแล้ว ทำให้ได้ข้อมูลที่เรียงแล้วที่มีขนาดใหญ่ขึ้น

เราจะเริ่มต้นด้วยข้อมูล 1 ตัว ซึ่งข้อมูลหากมีหนึ่งตัวก็ถือว่าเรียงแล้ว จากนั้นในแต่ละรอบเราจะนำเอาข้อมูลใหม่ไปต่อท้ายจากนั้นหาตำแหน่งที่เหมาะสม แล้วเลื่อนส่วนที่มากกว่าไปทางขวามือ จากนั้นแทรกข้อมูลใหม่ ทำให้ได้ข้อมูลใหม่ที่มีขนาดใหญ่ขึ้น

จากรูปที่ 4.10 สีเทาคือข้อมูลที่เรียงแล้ว ส่วนสีเหลืองคือข้อมูลใหม่ที่ต้องการนำมาเรียง ในรอบแรกเรามีข้อมูลที่เรียงอยู่แล้ว 1 ตัว คือ 48 แล้วเราต้องการเพิ่มข้อมูล 37 37 ต้องไปหาตำแหน่งที่ถูกต้องซึ่งคืออยู่ก่อนหน้า 48 พอหาตำแหน่งที่ถูกต้องและแทรกลงไปได้แล้ว ข้อมูลที่เรียงแล้วจะมีขนาดใหญ่ขึ้น ในรอบต่อมาข้อมูลเป็น 71 71 ต้องไปหาตำแหน่งที่ถูกต้องซึ่งคืออยู่ต่อท้าย พอแทรกได้แล้ว ข้อมูลที่เรียงแล้วจะมีขนาดใหญ่ขึ้น ทำเช่นนี้ต่อไปเรื่อยๆ จนแทรกข้อมูลครบทุกตัว การเลื่อนแทรกทำได้โดยเปรียบเทียบข้อมูลที่เรียงแล้วจากด้านหลังไปด้านหน้า หากข้อมูลใหม่มีค่าน้อยกว่าข้อมูลเก่าจะขยับข้อมูลเก่าไปทางขวามือ จากนั้นพิจารณาตัวก่อนหน้าไปอีก 1 ตัว ทำเช่นนี้จนพบกรณีที่ข้อมูลใหม่น้อยกว่าข้อมูลเก่า แสดงว่าพบจำแหน่งที่เหมาะสมในการแทรกแล้ว

```
def insertionSort(mylist):
    for index in range(1,len(mylist)):
        newitem = mylist[index]
```

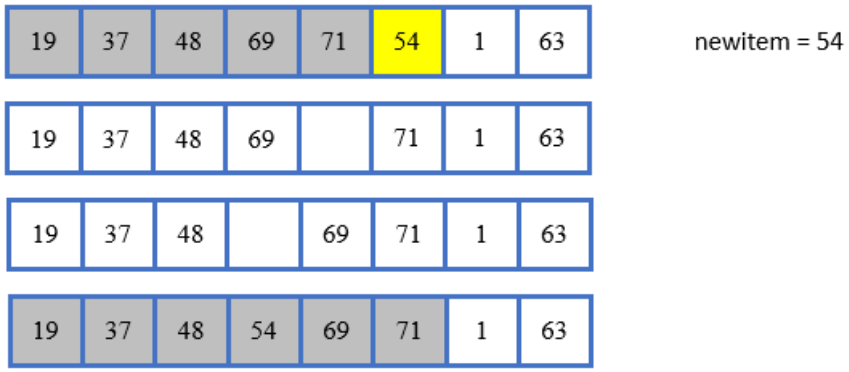


รูปที่ 4.10: ตัวอย่าง Insertion sort

```
        position = index
        while position > 0 and mylist[position-1]>newitem:
            mylist[position] = mylist[position-1]
            position = position-1
        mylist[position] = newitem

mylist=[48, 37, 71, 69, 19, 54, 1, 63]
insertionSort(mylist)
print(mylist)
```

4.2.4 Merge sort

ในหัวข้อย่อยนี้จะกล่าวถึงการเรียงอีกประเภทนึงที่มีความเร็วในการทำงาน เร็วกว่า 3 วิธีข้างต้นโดยใช้วิธีการแบบแบ่งแยกและเอาชนะ (Divide and conquer) Merge sort จะแบ่งข้อมูลออกเป็น 2 ส่วนย่อยที่เท่ากัน โดยแต่ละส่วนย่อยนี้จะแบ่งครึ่งต่อไปเรื่อยๆ จนกลายเป็นข้อมูลที่เรียงแล้วนั่นคือ ถ้าแบ่งแล้วไม่มีตัวเลย หรือมี 1 ตัวจะเป็นกรณีที่ข้อมูลเรียงแล้ว จากนั้นนำคำตอบย่อยที่เรียงแล้วเหล่านี้มารวมกันกลับเป็นคำตอบที่ใหญ่ขึ้น โดยเรามีข้อมูลที่เรียงแล้ว 2 ชุด เราก็จะนำมาสร้างเป็นข้อมูลที่เรียงแล้ว 1 ชุดที่มีขนาดเป็นผลรวมของสองชุดนั้นได้ โดยค่อยๆ หยิบตัว

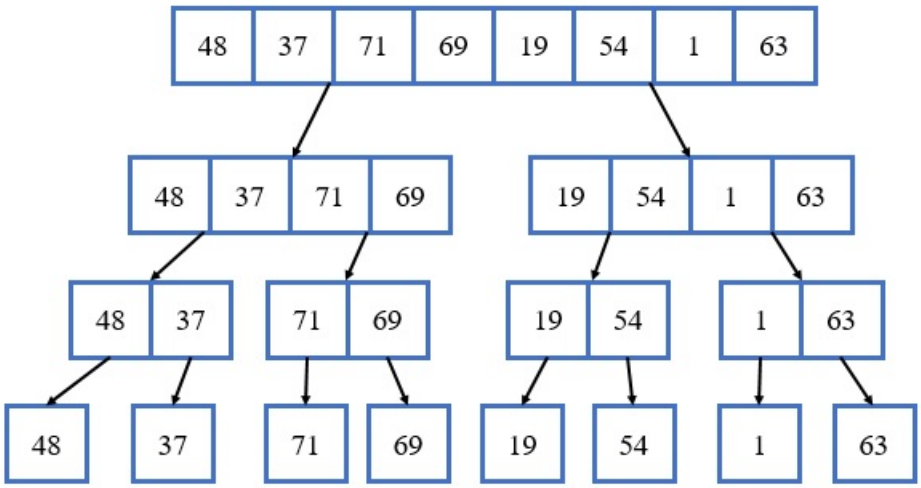


รูปที่ 4.11: ตัวอย่าง Insertion sort ในรอบที่เพิ่ม 54

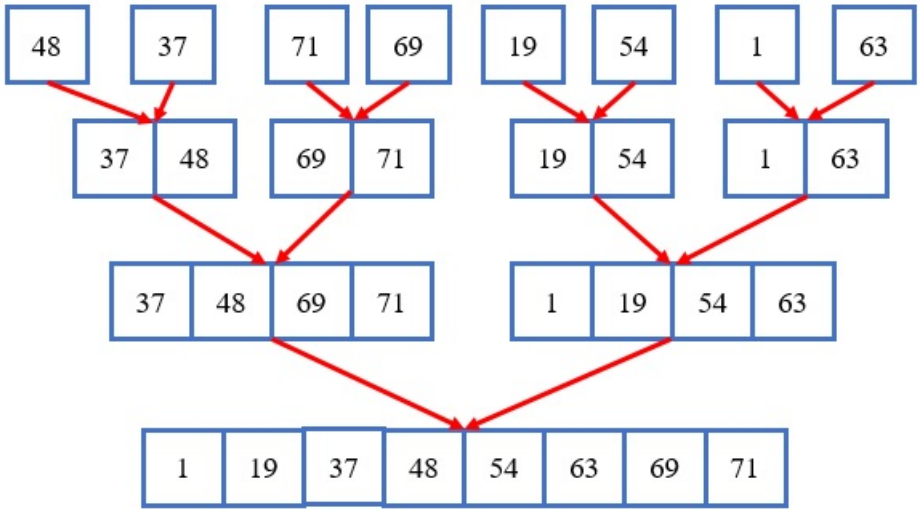
น้อยสุดจากสองชุดมาสร้างเป็นชุดใหม่ จากนั้นรวมต่ออีก คนคำตอบนั้นเป็นคำตอบของปัญหาตั้งต้น รูปที่4.12เป็นตัวอย่างการแบ่งข้อมูลใน list ออกเป็นส่วนย่อยๆ และรูปที่4.13 เป็นการรวมข้อมูลกลับมาเป็นคำตอบ

ตัวอย่าง code เป็นดังนี้

```
def mergeSort(mylist):
    #split phase
    print(mylist)
    if len(mylist)>1:
        mid = len(mylist)//2
        leftPart = mylist[:mid]
        rightPart = mylist[mid:]

        mergeSort(leftPart)
        mergeSort(rightPart)
        #merge
        i = 0
        j = 0
        k = 0
        while i<len(leftPart) and j<len(rightPart):
            if leftPart[i] < rightPart[j]:
                mylist[k] = leftPart[i]
                i = i+1
            else:
```



รูปที่ 4.12: การแบ่งข้อมูลออกเป็นส่วนย่อยๆ ของ list



รูปที่ 4.13: การรวมผลลัพธ์ย่อยกลับมาเป็นคำตอบที่ปัญหาที่ใหญ่ขึ้นจนได้คำตอบปัญหาหลัก

```
                mylist[k] = rightPart[j]
                j = j+1
            k = k+1
        while i<len(leftPart):
            mylist[k] = leftPart[i]
            i = i+1
            k = k+1
        while j<len(rightPart):
            mylist[k] = rightPart[j]
            j = j+1
            k = k+1
mylist=[48, 37, 71, 69, 19, 54, 1, 63]
mergeSort(mylist)
print(mylist)
```

สังเกตุได้ว่า mergeSort นั้นเป็นฟังก์ชัน Recursive และกรณี Base case คือ กรณีที่ความยาวของ List มีค่าน้อยกว่าหรือเท่ากับ 1 ซึ่งถ้า List เรายาวน้อยกว่าหรือเท่ากับ 1 แล้วแสดงว่า List ของเราเรียงแล้ว แต่ถ้าไม่ใช่เราจะแบ่งข้อมูลออกเป็นสองส่วน leftPart และ rightPart ขณะแบ่งกรณีที่เกิดขึ้นได้คือ ได้สองส่วนเท่ากัน หรือต่างกัน 1 หลังจากนั้นเมื่อเรียก mergeSort(leftPart) และ mergeSort(rightPart) แล้วเราสมมติได้เว่าข้อมูลถูกเรียงแล้ว ดังนั้นในส่วนที่เหลือขงฟังก์ชัน จะเป็นการรวม List 2 อันที่เรียงแล้วให้กลายเป็น List ที่ยาวขึ้น โดยเราจะหยิบเอาตัวน้อยสุดจากทั้งสอง List มาใส่ mylist ทำจน List ย่อยอันใดอันหนึ่งหมดจากนั้นที่เหลือจะถูกนำเอามาต่อท้าย

4.2.5 Quick sort

ต่อไปเป็นเป็นวิธีการเรียงข้อมูลอย่างสุดท้าย สังเกตอย่างหนึ่งว่า mergeSort นั้นต้องการใช้เนื้อที่ในหน่วยความจำเพิ่มอีกอย่างน้อย 1 เท่าเอาไว้เก็บข้อมูลแต่ quick sort นี้ใช้หลักการคล้ายกับ Merge sort แต่ไม่ต้องใช้เนื้อที่ในหน่วยความจำเพิ่ม

หลักการของ Quick sort คือจะเลือกสมาชิกใน List มาตัวหนึ่ง อาจจะเป็นตัวหน้าสุดมาให้ชื่อว่าเป็น pivot หลังจากนั้นเราจะแบ่งข้อมูลใน List โดยใช้ pivot นี้โดยจะได้ ส่วนที่น้อยว่า pivot ให้อยู่ใน List ทางซ้ายมือของ pivot ส่วนที่มากกว่า pivot ให้อยู่ใน List ทางขวามือของ pivot การทำเช่นนี้หลังทำแล้วสิ่งที่เกิดขึ้นคือ pivot อยู่ถูกตำแหน่ง หลังจากนั้นเราจะ Recursive ไปทำส่วนที่น้อยกว่าทางซ้าย และส่วนที่มากกว่าทางขวา ทำเช่นนี้ไปเรื่อยๆ จนแบ่งได้ข้อมูลตัวเดียวหรือไม่มี ซึ่งคือเรียงแล้วนั้นเอง

ในการแบ่งข้อมูลเราจะมีตัวแปรที่ค่อยช่วยเรา 2 ตัวคือ leftmark กับ rightmark ซึ่งแทนจุดเริ่มต้นและจุดปลายของข้อมูลที่เหลือ จุดมุ่งหมายของการแบ่งคือ ย้ายข้อมูลที่อยู่ผิดฝั่งเมื่อเทียบกับ pivot ไปไว้ให้ถูกฝั่ง โดย leftmark จะเคลื่อนที่ไปทางขวาจนกว่าจะเจอตัวที่มากกว่า pivot หลังจากนั้น rightmark จะเคลื่อนที่ไปทางซ้าย

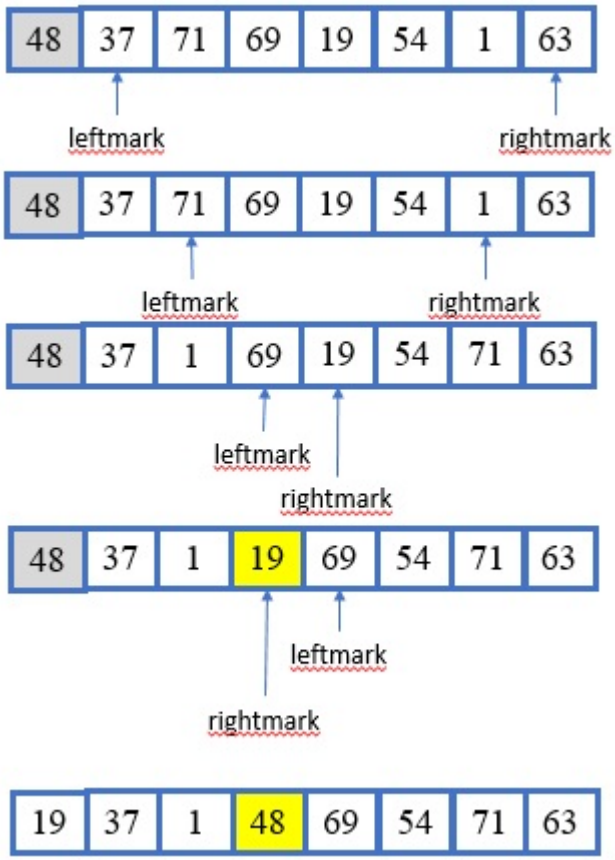
รูปที่ 4.14: เลือก pivot โดยให้ตัวแรกของ list เป็น pivot

จนกว่าจะเจอตัวที่น้อยกว่า pivot หากเจอแล้วและ leftmark น้อยกว่า rightmark(ยังไม่ซ้อนกัน) จะสลับข้อมูลของสองตำแหน่งนั้นกัน ทำเช่นนี้ไปเรื่อยๆ จน leftmark มากกว่า rightmark แล้วจะได้ว่าตำแหน่ง rightmark คือตำแหน่งที่เราจะแบ่ง โดยจะสลับ pivot กับค่าในตำแหน่ง rightmark ดังตัวอย่างรูปที่4.15 ซึ่งเราจะนำมาเขียนเป็นฟังก์ชัน partition

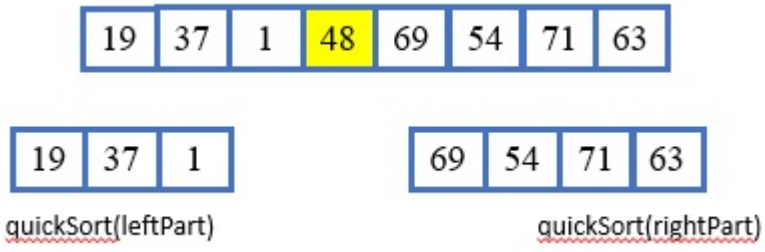
จากตัวอย่างเราจะได้ว่า 48 อยู่ถูกตำแหน่งแล้ว หลังจากนั้นข้อมูลจะถูกแบ่งเป็นสองส่วนคือส่วนที่น้อยว่า 48 และส่วนที่มากกว่า 48 ซึ่งเราก็จะแก้ปัญหาด้วยวิธี Quick sort เช่นเดิม ซึ่งเขียนได้ด้วยฟังก์ชัน quiclSort

ต่อไปเป็น code ตัวอย่างของ Quick sort

```
def quickSort(mylist,first,last):
    if first<last:
        splitPoint = partition(mylist,first,last)
        quickSort(mylist,first,splitPoint-1)
        quickSort(mylist,splitPoint+1,last)

def partition(mylist,first,last):
    pivot = mylist[first]
    leftmark = first+1
    rightmark = last

    done = False
    while not done:
        while leftmark <=rightmark and mylist[leftmark]<=pivot:
            leftmark = leftmark+1
        while leftmark <=rightmark and mylist[rightmark]>=pivot:
            rightmark = rightmark-1
        if  rightmark< leftmark:
            done = True
        else:
            temp = mylist[leftmark]
            mylist[leftmark] = mylist[rightmark]
```



รูปที่ 4.15: การหาตำแหน่งในการแบ่ง



รูปที่ 4.16: แต่ละส่วนไปเรียก quickSort ทำงานต่อ

```
            mylist[rightmark] = temp

    temp = mylist[first]
    mylist[first] = mylist[rightmark]
    mylist[rightmark] = temp
    return rightmark

mylist=[48, 37, 71, 69, 19, 54, 1, 63]
quickSort(mylist, 0, len(mylist)-1)
print(mylist)
```

4.3 แบบฝึกหัดท้ายบท

1. จงปรับปรุง bubbleSort เพื่อลดการเปรียบเทียบหากพบว่าข้อมูลเรียงแล้ว

2. จงปรับปรุง อัลกอริทึมการเรียงข้างต้นทุกอันให้เรียงลำดับจากมากไปน้อย

บทที่ 5

ต้นไม้

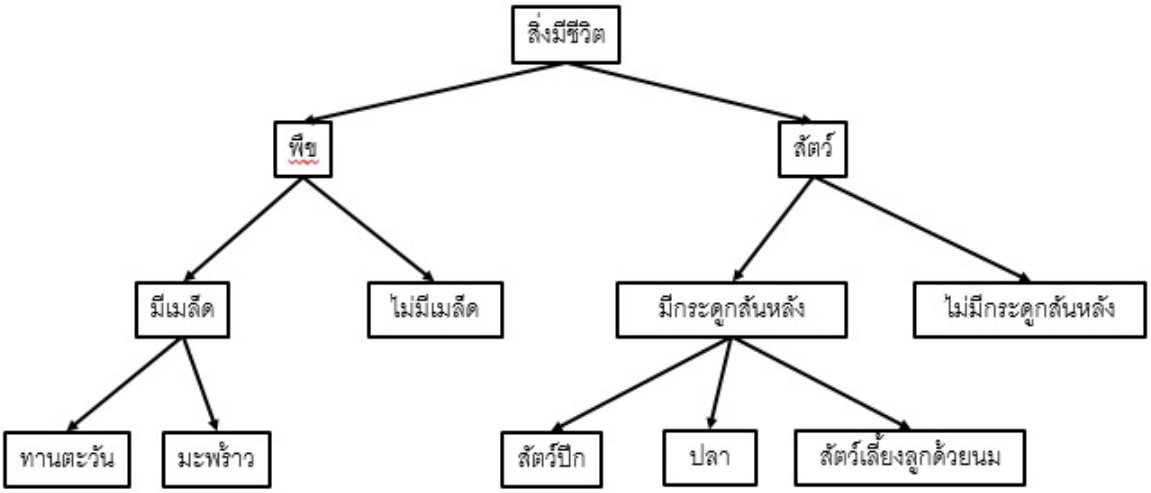
ในหัวข้อก่อนหน้าเราได้เรียน Linear data structures มาแล้วเช่น Stack Queue Deque และได้เรียน Recursion ต่อไปเราจะมาดูโครงสร้างข้อมูลอีกชนิดที่เรียกว่าต้นไม้(Tree) Tree ถูกใช้ในหลายๆ ด้านของวิทยาการคอมพิวเตอร์ เช่นใน ระบบปฏิบัติการ(Operating systems), คอมพิวเตอร์กราฟฟิค(Compuer graphics), ฐานข้อมูล(Databases), เครือข่าย(Networking) เป็นต้น ในระบบปฏิบัติการนั้น Tree ที่เราเจอบ่อยได้แก่ระบบไฟล์ในเครื่อง(File system) File system นั้นมีโครงสร้างคล้ายกับต้นไม้ทางชีววิทยา เราสามารถเริ่มจาก Root ไปยัง Directory ใดๆ ก็ได้ path นั้นจะระบุโดย Subdirectory

ทั้งนี้หากมองดูให้ดี Tree เป็นโครงสร้างข้อมูลที่เลียนแบบมาจากต้นไม้ในชีวิตจริงๆ ซึ่งต้นไม้จริงๆ ประกอบด้วยหลายส่วนเช่น ราก กิ่ง ก้าน ใบ เป็นต้น ส่วนโครงสร้างข้อมูลแบบ Tree โดยทั่วไปแล้วจะมี Root, Branches และ Leaves สิ่งที่แตกต่างกันหลักๆ คือต้นไม้จริงรากจะอยู่ด้านล่าง ใบจะอยู่ด้านบน แต่ส่วนใหญ่ Tree ที่เราจะสร้างนั้นรากจะอยู่ด้านบน ใบจะอยู่ด้านล่าง เหมือนมองต้นไม้กลับหัว

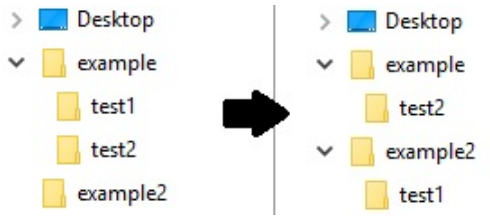
ในบางครั้งเราใช้ Tree ในการตอบคำถามโดยจะไปตามเส้นทางที่สอดคล้องจนถึงปลายทางซึ่งเป็นคำตอบดังตัวอย่างในรูปที่5.1

ในตัวอย่างนี้เราอาจจะถามว่า “สิ่งมีชีวิตนี้เป็นพืชหรือสัตว์” ถ้าคำตอบเป็น “พืช” เราก็จะไปถามต่อที่โหนดพืช ถ้าเป็นสัตวีก็ไปถามต่อยังโหนดสัตว์ จากนั้นอาจจะถามต่อ เช่น “เป็นสัตว์มีกระดูสันหลังหรือไม่มีกระดูกสันหลัง” ถ้าคำตอบเป็นไม่มีกระดูสันหลังก็จะหยุด แต่ถ้ามีกระดูกสันหลังก็จะแยได้เป็น สัตว์ปีก ปลา สัตว์เลี้ยงลูกด้วยนม เป็นต้น

คุณสมบัติของ Tree นั้นมีหลายอย่าง ตัวอย่างเช่น เมื่อพิจารณาโหนดใบ (Leaf node) ที่แตกต่างกัน เราสามารถระบุเส้นทาง(Path) จากโหนดราก root ไปโหนดใบใดๆ ได้เพียงทางเดียว ตัวอย่างจากรูปที่5.1 เช่น สิ่งมีชีวิต-> สัตว์-> สัตว์มีกระดูสันหลัง->สัตว์เลี้ยงลูกด้วยนม คุณสมบัติที่สำคัญอีกอย่างของ Tree นั้นคือการสืบทอดคุณลักษณะที่เป็นลำดับขั้น นั่นคือเราสามารถย้ายทั้งส่วนของ Tree (เรียกว่า Subtree) ไปยังตำแหน่งใหม่ใน Tree โดยที่ไม่ส่งผลต่อโครงสร้างด้านล่าง ตัวอย่างเช่นในรูปที่5.2 เราสามารถย้าย Folder test1 ซึ่งอยู่ภายใน Folder example ไปไว้ยังตำแหน่งใหม่ใน Folder example2 โดยที่ ข้อมูลต่างๆ ใน Folder test1 ก็ย้ายไปด้วย นั่นคือโครงสร้างเดิมภายใน Folder test1 ไม่เปลี่ยนแปลง



รูปที่ 5.1: ตัวอย่าง Tree ที่ใช้ในการตอบคำถามสิ่งมีชีวิต

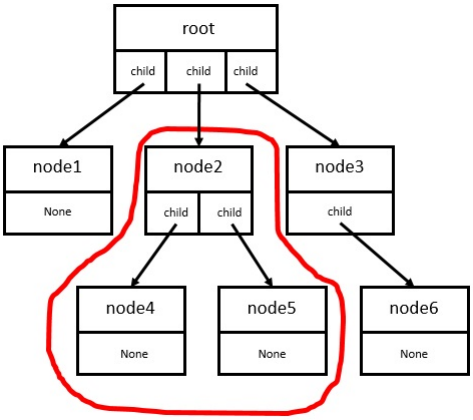


รูปที่ 5.2: ตัวอย่างการย้าย Folder

5.1 คำศัพท์และนิยามต่างๆ

ในหัวข้อย่อยนี้จะเป็นการนิยามคำศัพท์ที่เกี่ยวข้อง

- **Node** โหนด เป็นส่วนพื้นฐานของ Tree เป็นวัตถุชนิดหนึ่งที่เราสามารถตั้งชื่อได้ เก็บข้อมูลเพิ่มได้
- **Edge** เส้นเชื่อม เป็นส่วนพื้นฐานอีกอย่างของ Tree เส้นเชื่อมจะเชื่อมระหว่างโหนด 2 โหนด เพื่อแสดงความสัมพันธ์ระหว่างพวกมัน จริงๆ แล้วมีทั้งแบบมีทิศทาง(มีหัวลูกศรกำกับทิศ)และไม่มีทิศทาง(เป็นเส้นตรงเชื่อมระหว่างโหนดเฉยๆ) ทั้งนี้ในที่นี้เพื่อความเข้าใจจะให้มีทิศทางก่อน โดยแต่ละโหนด จะมีเส้นเชื่อมขาเข้าได้เพียงหนึ่งโหนด แต่จะมีเส้นเชื่อมขาออกได้หลายโหนด ยกเว้นโหนดราก(Root)
- **Root** โหนดราก เป็นโหนดเพียงโหนดเดียวใน Tree ที่ไม่มีเส้นเชื่อมขาเข้า
- **Path** เส้นทาง เป็นลำดับของโหนดใน Tree ที่เชื่อมต่อกันด้วยเส้นเชื่อม ตัวอย่างเช่น Root-> node3->node6
- **Children** โหนดลูก เป็นเซตของโหนด c ที่มีเส้นเชื่อมขาเข้าจากโหนดเดียวกันที่เราเรียกว่าเป็นโหนดลูกของโหนดนั้น เช่น node1, node2 และ node3 เป็นเซตของโหนดลูกของ Root
- **Parent** โหนด x จะเป็นโหนดพ่อของโหนด y ถ้ามีเส้นเชื่อมระหว่าง x และ y โดยที่ x เป็นต้นทางและ y เป็นปลายทางเช่น node3 เป็น Parent ของ node6
- **Sibling** โหนดพี่น้อง โหนดใน Tree ที่เป็นโหนดลูกของ parent ตัวเดียวกัน เช่น node4 และ node5 เป็นพี่น้องกัน
- **Subtree** เป็นเซตของโหนดและเส้นเชื่อมที่ประกอบด้วย Parent และ ลูกหลานของ parent ตัวนั้น เช่นในวงสีแดงเป็น Subtree ที่มี node2 เป็น Root
- **Leaf** เป็นโหนดที่ไม่มี Children เช่น node1, node4, node5 และ node6 เป็นโหนดใบ
- **Level** ระดับของโหนด n คือ จำนวนของเส้นเชื่อมบน Path จาก Root มายังโหนด n จากนิยามนี้ทำให้ Level ของ Root เป็น 0
- **Height** ความสูงของ Tree จะเท่ากับ Level ที่มากที่สุดของโหนดใดๆ ใน Tree



รูปที่ 5.3: ตัวอย่าง Tree ที่สร้างจากเซตของโหนดและเส้นเชื่อม

5.1.1 นิยาม

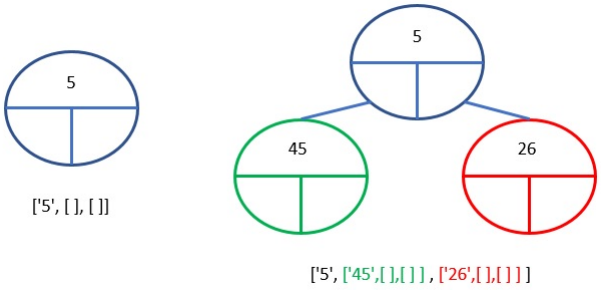
นิยามอย่างเป็นทางการของ Tree ในที่นี้จะมี 2 นิยาม นิยามแรกจะเกี่ยวกับโหนดและเส้นเชื่อมอีกนิยามจะเป็นแบบ recursive •

- Tree ประกอบด้วยเซตของโหนดและเซตของเส้นเชื่อมที่เชื่อมระหว่างคู่ของโหนด โดยมีคุณสมบัติดังนี้ มีหนึ่งโหนดใน Tree ที่เป็น Root node, Path จาก Root ไปยังแต่ละโหนดมีเพียง Path เดียวและถ้าแต่ละโหนดใน Tree มีลูกไม่เกิน 2 โหนดเราจะเรียกว่า Binary tree
- นิยามแบบที่สอง Tree จะว่างหรือประกอบไปด้วย Root และมี Subtree จำนวน 0 หรือมากกว่านั้นซึ่งแต่ละอันเป็น Tree ด้วยโดย Root ของแต่ละ Subtree ถูกเชื่อมกับ Root ของ Parent tree ด้วยเส้นเชื่อม

5.2 การสร้าง Tree

หลังจากเรานิยามสิ่งต่างๆ เพื่อให้เข้าใจตรงกันแล้ว ต่อไปเราจะสร้าง Binary tree ที่มีฟังก์ชันดังนี้กัน

- `BinaryTree()` เป็นการสร้าง Binary tree ใหม่
- `get_left_child()` คืนค่า Binary tree ที่สอดคล้องกับลูกทางซ้ายของโหนดปัจจุบัน(Current node)
- `get_right_child()` คืนค่า Binary tree ที่สอดคล้องกับลูกทางขวาของโหนดปัจจุบัน(Current node)



รูปที่ 5.4: ตัวอย่างการสร้าง Tree ด้วย List of Lists

- `set_root_val(val)` เก็บค่าวัตถุไว้กับโหนดปัจจุบัน
- `get_root_val()` คืนค่าของวัตถุที่เก็บที่โหนดปัจจุบัน
- `insert_left()` สร้าง Binary tree ใหม่และเชื่อมมันเป็นลูกทางซ้ายของโหนดปัจจุบัน
- `insert_right()` สร้าง Binary tree ใหม่และเชื่อมมันเป็นลูกทางขวาของโหนดปัจจุบัน

ในการตัดสินใจสร้าง Tree เราจะเลือกวิธีว่าจะเก็บข้อมูลภายในกันอย่างไรดี ในภาษาไพธอนมี 2 วิธีที่น่าสนใจ ซึ่งเราจะลองทำทั้งสองวิธี วิธีแรกเราใช้ List ซ้อนไปใน List (List of Lists) และวิธีที่สองเราใช้การสร้างโหนดและเชื่อมต่อกัน (Nodes and References)

5.2.1 การสร้าง Tree ด้วย List of Lists

เราจะสร้าง Tree ด้วย List of Lists หรือสร้าง List แล้วใส่ List เข้าไปข้างใน หลักการง่ายๆ คือ แต่ละโหนดคือ List ที่ประกอบด้วย 3 ส่วนได้แก่ ข้อมูลตัวเอง, ลูกทางซ้ายและลูกทางขวาซึ่งเริ่มต้นเป็น List ว่างเพราะว่ายังไม่มีข้อมูล ทีนี้หากเราจะเพิ่มโหนดลูกทางใด เราก็สร้างโครงสร้างที่มีสามส่วนนี้แทนที่ลูกทางนั้นซ้อนไปเท่านั้นเอง ดังรูปที่5.4

หากพูดให้ชัดเจน Tree เราจะเก็บค่าของ Root node เป็นสมาชิกตัวแรกของ List สมาชิกตัวที่สองของ List จะเป็น List ของ Left subtree ตัวที่สามของ List จะเป็น List ของ Right subtree ดังตัวอย่างต่อไปนี้

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

```
]
```

จากคุณสมบัติของ List ทำให้เราเข้าถึง Subtree ของ List โดยใช้ index นั่นคือ Root ของ Tree คือ myTree[0] Left subtree ของ Root คือ myTree[1] Right subtree ของ Root คือ myTree[2] ต่อไปเป็นตัวอย่างการสร้าง Tree โดยใช้ List of Lists

```
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

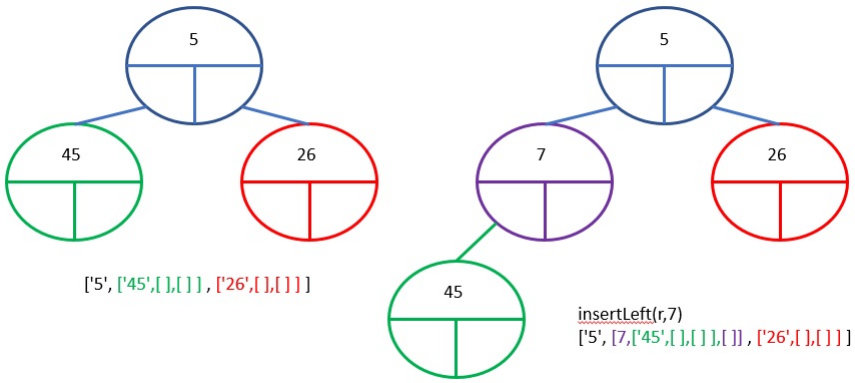
```
def binaryTree( r):
    return [r, [], [] ]
```

ฟังก์ชัน binaryTree เป็นการสร้าง List ที่มีสมาชิกเป็น Root และมี List ว่างสองอัน ในการเพิ่ม Left subtree ให้กับ Root เราจะต้องแทรก List ใหม่ไปในตำแหน่งที่สองของ List ทั้งนี้สิ่งที่ต้องระวังคือถ้ามีของอยู่แล้วเราจะทำอย่างไร ทับเลยหรือว่าทำอย่างไร ในที่นี้เราจะพยายามแทรกข้อมูลใหม่เข้าไปโดยเราจะผลักข้อมูลเก่าไปเป็นลูกทางซ้ายของข้อมูลที่เราจะใส่ใหม่

```
def insertLeft(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
```

สังเกตว่าในฟังก์ชัน insertLeft เริ่มต้นจะจากเอา List ของลูกทางซ้ายทั้งหมดออกมาก่อน(อาจจะไม่มีตัว) หลังจากนั้นเราจะเพิ่มลูกทางซ้ายใหม่ โดยให้ ข้อมูลลูกทางซ้ายอันเก่าเป็นลูกทางซ้ายของข้อมูลที่เพิ่ม นี่ทำให้เราเชื่อมต่อโหนดใหม่เข้ากับ Tree ได้ code ของ insertRight ก็คล้ายกัน

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



รูปที่ 5.5: ตัวอย่าง insertLeft

```
        root.insert(2,[new_branch, [], []])
    return root
```

ต่อไปเป็นฟังก์ชัน get และ set ค่าของ 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]
```

ต่อไปลองนำเอา code มารวมกันแล้วสั่งงานด้วยคำสั่ง

```
r = binaryTree(3)
insertLeft(r, 4)
insertLeft(r, 5)
insertRight(r, 6)
insertRight(r, 7)
l = get_left_child(r)
print(l)
set_root_val(l, 9)
print(r)
insertLeft(l, 11)
```

```
print(r)
```

5.2.2 การสร้าง Tree ด้วย Node and References

การสร้าง Tree ในวิธีนี้จะทำโดยสร้างโหนดและกำหนดความสัมพันธ์ให้กับโหนด(Node and References) ในการสร้างโหนดใน Binary tree เราจะสร้าง class BinaryTreeNode ขึ้นมาโดยที่เราจะให้โหนดเก็บค่าของตัวมันเอง, โหนดลูกทางซ้ายเป็นใครและโหนดลูกทางขวาเป็นใคร เมื่อมองเป็น Attributes จะได้ว่าเก็บค่า key, left_child และ right_child

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

ขั้นแรกเราจะสร้าง Instance หรือ Object ใหม่ของ BinaryTreeNode เพื่อมาเป็น Root ของ Binary tree ก่อน ที่เริ่มต้นค่าของลูกทางซ้ายและลูกทางขวาจะยังไม่มีนั่นคือเป็นค่า None ก่อน เมื่อเราต้องการเพิ่มโหนดลูกทางซ้ายไปใน Tree เราจะสร้าง Instance หรือ Object ใหม่ของ BinaryTreeNode เพื่อมาเป็นโหนดใหม่ก่อน จากนั้นจึงเปลี่ยน `self.left_child` ของ Root ให้มีค่าเป็นโหนดใหม่ซึ่งก็เหมือนการสร้างเส้นเชื่อมระหว่าง Root กับโหนดลูกโหนดใหม่นั่นเอง ทั้งนี้วิธีนี้สามารถใช้ได้กับโหนดอื่นๆ ได้ด้วยเพียงแค่เราหาโหนดต้นทางที่จะเชื่อมกับระบุโหนดใหม่ให้ได้ สังเกตได้ว่า ใน Constructor นั้นต้องการข้อมูลมาเก็บไว้ที่ key เช่นเดียวกับการที่เอาข้อมูลมาเก็บในไว้ที่ตำแหน่งแรกของการสร้าง Tree แบบ List of Lists

ต่อไปเราจะพิจารณาฟังก์ชัน ที่เราต้องเขียนเพิ่ม ในการเพิ่มโหนดลูกทางซ้ายให้กับ Tree

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

ในการเพิ่มโหนดให้เป็นโหนดลูกทางซ้ายจะมีสองกรณีที่ต้องพิจารณา กรณีแรกคือ โหนดที่เราจะเพิ่มไม่มีโหนดลูกทางซ้าย เราก็เพิ่มโหนดใหม่ไปเป็นลูกทางซ้ายได้เลย กรณีที่สองคือ โหนดที่เราจะเพิ่มมีโหนดลูกทางซ้าย เราจะเพิ่มโหนดใหม่แล้วย้ายโหนดลูกทางซ้ายเดิมไปเป็นโหนดลูกทางซ้ายของโหนดใหม่หรือย้ายลงไปอีก 1 Level นั่นเอง

code ของการเพิ่มโหนดทางขวามือก็เช่นกัน

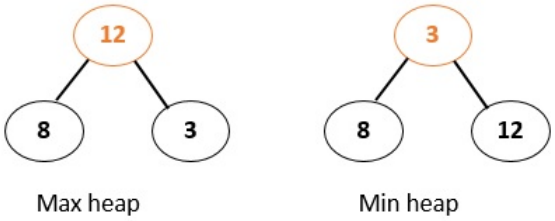
```
def insert_right(self,new_node):
    if self.right_child == None:
        self.right_child = BinaryTreeNode(new_node)
    else:
        t = BinaryTreeNode(new_node)
        t.right_child = self.right_child
        self.right_child = t
```

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 ให้มีค่าเป็น a และเพิ่มโหนด b และ c เป็นลูกทางซ้ายและทางขวาตามลำดับ ซึ่งจะเกี่ยวกับการเข้าถึงค่า key left_child right_child สังเกตว่าลูกทางซ้ายและลูกทางขวาจะเป็นคนละ Instance ของ BinaryTreeNode กัน

```
r = BinaryTreeNode('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())
```



รูปที่ 5.6: ตัวอย่าง Max heap และ Min heap

5.3 Priority Queue with Binary Heaps

ในบทก่อนเราได้เรียนโครงสร้างข้อมูลแบบ first-in first-out นั่นคือ Queue ในหัวข้อนี้เราจะสนใจ Queue ชนิดพิเศษชนิดหนึ่งที่เรียกว่า Priority queue ซึ่งทำงานคล้ายกับ Queue เพียงแต่ตอนนำข้อมูลออก ข้อมูลจะถูกนำออกตามความสำคัญ ในการสร้าง Priority queue เพื่อให้ทำงานคล้ายกับ Queue เราจะใช้ List มาเก็บข้อมูล และ dequeue จะเป็นการนำเอาข้อมูลตัวหน้าสุดออก แต่เราจะจัดการอย่างไรเพื่อให้ข้อมูลภายในมีการเรียงลำดับตามความสำคัญ โดย ข้อมูลที่มีความสำคัญสูงสุดจะอยู่ด้านหน้า

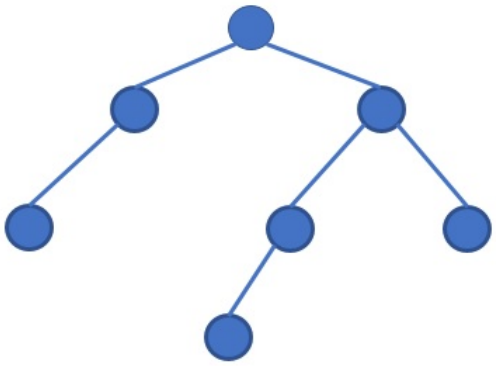
เราจะจัดการกับปัญหานี้เมื่อเรียก enqueue เมื่อเรา enqueue ข้อมูลลงใน Priority queue แล้ว ข้อมูลใหม่นี้อาจจะถูกย้ายตำแหน่งไปยังด้านหน้าสุด วิธีการง่ายที่สุดคือใช้การเรียงข้อมูล เมื่อเราเพิ่มข้อมูลใหม่ลงไปใน List แล้วให้เรียงข้อมูล แต่วิธีนี้ถ้าเพิ่ม 10 ครั้งก็ต้องเรียงข้อมูล 10 ครั้งซึ่งทำงานช้า

เราจะสร้าง Priority queue โดยใช้โครงสร้างข้อมูลที่เรียกว่า Binary heap ซึ่งทำงานได้เร็วกว่าการเรียงข้อมูลใหม่ทุกครั้ง Binary heap มีหน้าตาคล้าย Tree แต่เราจะสร้างมันโดยใช้ List เพียงอันเดียว ทั้งนี้ Binary heap มี 2 ชนิด

- Min heap มีลักษณะคือ key ที่มีค่าน้อยสุดจะอยู่ด้านหน้า
- Max heap มีลักษณะคือ key ที่มีค่ามากสุดจะอยู่ด้านหน้า

ในที่นี้เราจะสร้าง Min heap โดยที่การดำเนินการของ Binary heap มีดังนี้ การดำเนินการของ Binary heap

- `BinaryHeap()` สร้าง Binary heap อันใหม่
- `insert(k)` เพิ่มข้อมูลใหม่ลงใน Binary heap
- `find_min()` คืนค่าข้อมูลที่มี key ที่น้อยที่สุด ข้อมูลยังอยู่ใน Binary heap
- `del_min()` คืนค่าข้อมูลที่มี key ที่น้อยที่สุดพร้อมทั้งลบข้อมูลนั้นด้วย



รูปที่ 5.7: ตัวอย่าง Balance binary tree

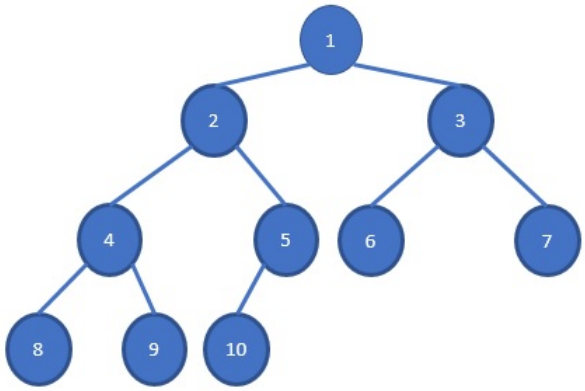
- `is_empty()` คืนค่า True ถ้าBinary heap ว่าง กรณีอื่น False
- `size()` คืนค่าจำนวนข้อมูลใน Binary heap
- `build_heap(list)` สร้าง Binary heap ใหม่จาก list ของข้อมูล

ในการสร้าง Binary heap เราต้องการให้การดำเนินการต่างๆ รวดเร็ว ดังนั้นเราจะต้องทำให้จำนวนครั้งในการเปรียบเทียบข้อมูลเพื่อให้ได้คำตอบไม่มาก หากเป็น Tree ก็ต้องมี Level ที่ต่ำ การที่จำทำให้ Tree โดยรวมมี Level ที่ต่ำนั่นคือ การทำให้ที่ลูกส่วนใหญ่ไม่สูงมาก หรือเรียกว่าการทำให้ Tree balance

Balanced binary tree นั้นแต่ละโหนดจะมีจำนวนโหนดใน Subtree ทางซ้ายและ Subtree ทางขวาต่างกันไม่เกิน 1 ชั้นซึ่งแทบจะเท่ากัน สำหรับ Binary heap ที่เราจะสร้างนั้นเราจะทำให้ Tree balance โดยการสร้าง Complete binary tree ซึ่งคือ Tree ที่ในแต่ละ Level มีจำนวนโหนดเต็มยกเว้น Level ล่างสุด โดยเราจะเติมโหนดจากซ้ายไปขวา

คุณสมบัติที่สำคัญอีกอย่างของ Complete binary tree คือ เราสามารถใช้ List แทนได้ เราไม่จำเป็นต้องสร้าง Tree แบบ Node and Reference หรือ List of Lists เนื่องจากว่า Tree นั้น Complete เราจึงสามารถเก็บโหนดแบบเรียงตามลำดับโดยให้ Root เป็น List ช่องที่ 1 ถัดมาพิจารณา Level 1 ลูกทางซ้ายของ Root จะเก็บใน List ช่อง 2 ลูกทางขวาช่อง 3 แล้วไป Level ถัดไปจากซ้ายไปขวาจนครบ ซึ่งการทำเช่นนี้สามารถพบความสัมพันธ์ได้คือ โหนดลูกทางซ้ายของ Parent (ตำแหน่ง p) คือโหนดที่อยู่ที่ตำแหน่ง 2p ใน list ส่วนโหนดลูกทางขวาคือโหนดที่อยู่ที่ตำแหน่ง 2p+1 ใน List การหาว่า Parent คือโหนดใดเราก็ใช้การหาร สมมติว่าโหนดในตำแหน่งที่ n โหนดพ่อของโหนดนี้คือโหนดตำแหน่ง n/2

ก่อนหน้านนี้เราพูดถึงความสัมพันธ์กันระหว่างตำแหน่ง ต่อไปเราจะพูดถึงความสัมพันธ์ของค่าของข้อมูล วิธีที่เราใช้ในการเก็บข้อมูลลง Binary heap จะต้องรักษาคุณสมบัติของ Heap หรือ Heap order property ไว้ด้วย โดยใน Binary heap แต่ละโหนด x ที่มี parent p ค่า key ที่ p เก็บจะน้อยกว่าหรือเท่ากับ key ใน x รูปก่อนหน้าเป็น Complete binary tree ที่มี Heap order property นั่นคือ โหนดพ่อจะมีค่าไม่มากไปกว่าโหนดลูก ทุกโหนดต้องรักษาคุณสมบัตินี้



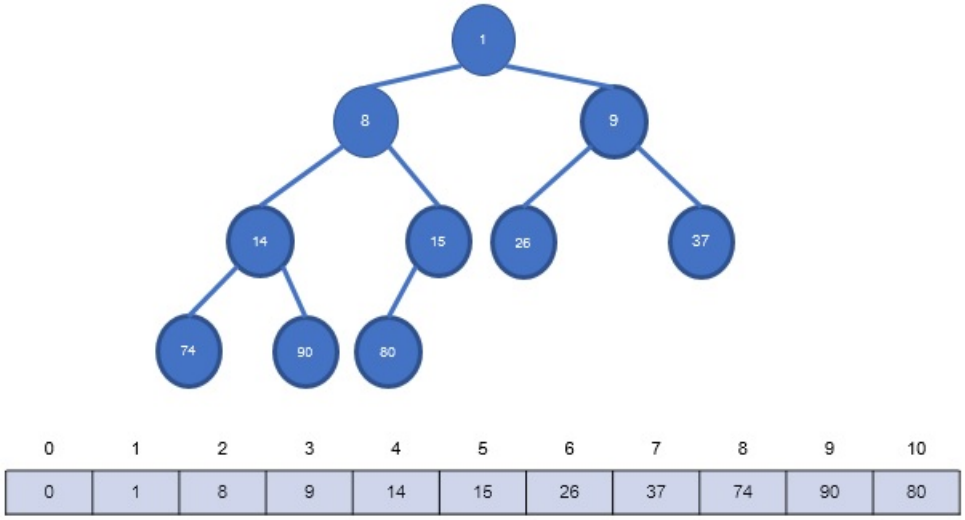
รูปที่ 5.8: ตัวอย่าง Complete binary tree ที่มี Heap order property

เริ่มต้นการสร้างด้วย Constructor เนื่องจาก Binary heap สามารถใช้ List เพียงอันเดียวในการสร้างได้ เริ่มต้นเราต้องสร้าง List และสร้าง Attribute current_size ในการเก็บว่าตอนนี้มีขนาดเท่าไร ในที่นี้ Binary heap เปล่า เราจะให้มีสมาชิกช่อง 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
```

ต่อไปเป็นฟังก์ชัน insert โดยวิธีที่ง่ายและมีประสิทธิภาพที่สุดคือเพิ่มเข้าไปท้าย List ซึ่งการเพิ่มไปท้าย List นี้ยังทำให้คุณสมบัติ Complete binary tree เป็นจริงอยู่ แต่ปัญหาคือการนำเอาข้อมูลใหม่ไปต่อท้าย ค่าของข้อมูลใหม่อาจจะขัดกับ คุณสมบัติของ Heap เราจะปรับปรุงโครงสร้างให้รักษาคุณสมบัติโดยเราจะใช้วิธีเปรียบเทียบข้อมูลใหม่กับ Parent ปัจจุบันของมันว่าถ้าข้อมูลที่เพิ่มเข้าไปใหม่มีค่าน้อยกว่า Parent เราก็จะสลับที่กัน จากนั้นตรวจสอบกับ Parent ตัวต่อไป จนทำไม่ได้หรือถึง Root สังเกตว่าการปรับข้อมูลขึ้นไป ทำให้เรารักษาคุณสมบัติของ Heap ระหว่างตัวใหม่กับตัว parent ของมันได้ แล้วยังรักษาคุณสมบัตินี้กับตัวที่อยู่ด้านล่างอีกด้วย ต่อไปเป็น code ของการปรับขึ้น(Perculate up)

```
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
```



รูปที่ 5.9: ตัวอย่าง Complete binary tree เมื่อเก็บใน List

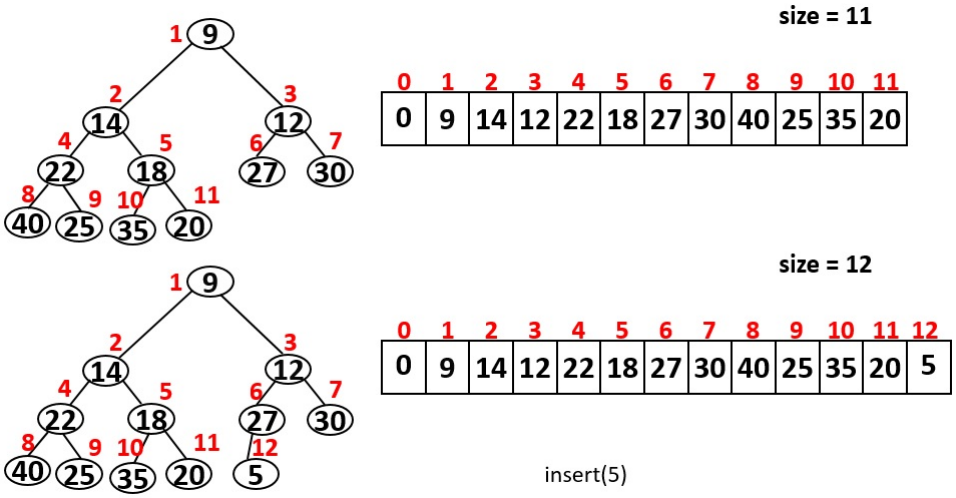
ต่อไปมาดู code ของการ insert เมื่อเราเพิ่มของเข้าไปต่อท้าย Tree จากนั้นเราจะ Perculate up เพื่อปรับตำแหน่งให้ถูกต้อง

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

ต่อไปเราจะสร้างฟังก์ชัน del_min เนื่องจาก คุณสมบัติของ Heap นั้น Root มีค่าน้อยที่สุด เพื่อให้หาค่าน้อยที่สุดได้อย่างรวดเร็ว ส่วนที่ยากของ del_min คือ พอเอาตัวแรกออกแล้วต้องปรับโครงสร้างของ Heap อย่างไรเพื่อให้ยังคงคุณสมบัติของ Heap เริ่มต้นเราจะแทนที่ Root ด้วยข้อมูลตัวสุดท้ายใน List โดยการย้ายตัวสุดท้ายมา ซึ่งรักษาคุณสมบัติของ Complete binary tree ไว้ได้ แต่มันทำลายคุณสมบัติของ Heap ดังนั้นเราจะทำให้รักษาคุณสมบัติของ Heap โดยปรับตัว Root ใหม่ลงไปให้อยู่ถูกที่ ในการปรับเพื่อให้รักษาคุณสมบัติของ Heap เราจะสลับ Root กับลูกของมันที่น้อยกว่า จากนั้นอาจจะต้องสลับอีกจนกระทั่งไม่มีลูกตัวไหนน้อยกว่ามัน เราจะสร้างฟังก์ชันที่เรียกว่า perc_down โดยมีฟังก์ชันสำหรับหาลูกตัวที่น้อยสุดคือ min_child

---

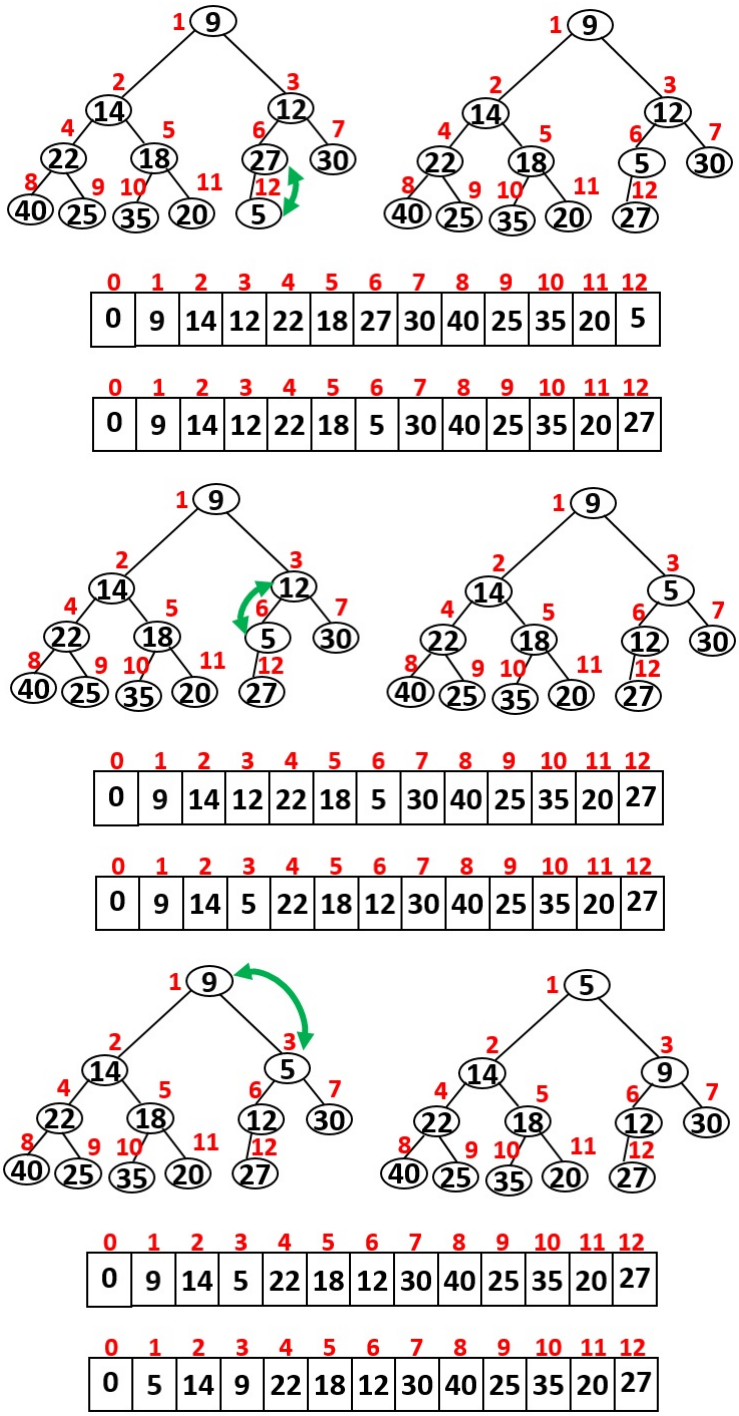
```
def min_child(self, i):
    if i*2+1  self.current_size:
        return i*2
    else:
        if self.heap_list[i*2] < self.heap_list[i*2+1]:
```



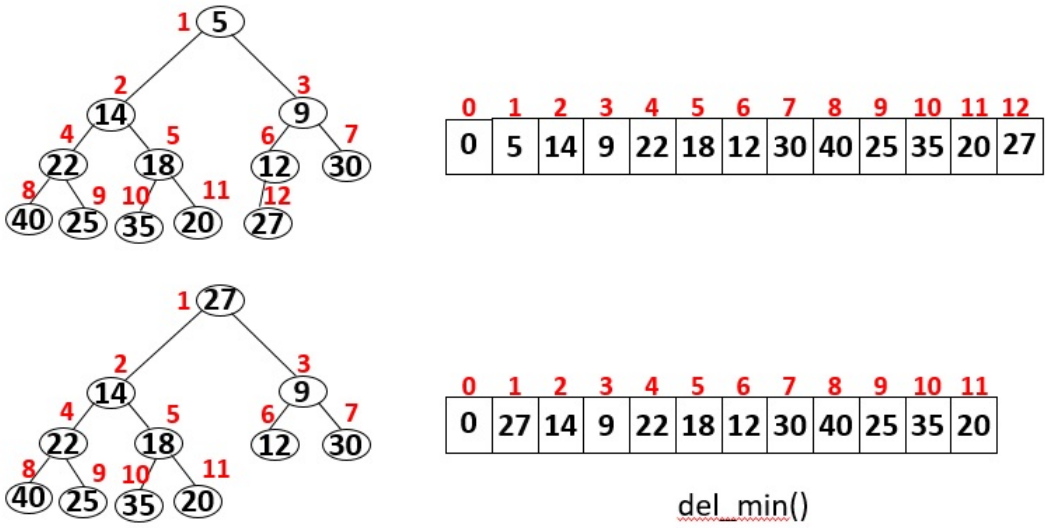
รูปที่ 5.10: ตัวอย่าง insert(5)

```
            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

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
```

สำหรับในกรณีที่เรามี List แล้วเราต้องการสร้าง Heap เราจะทำอย่างไร เราอาจจะเริ่มต้นด้วย insert Key ที



รูปที่ 5.11: ตัวอย่างการปรับขึ้นของ 5 (Perculate up)



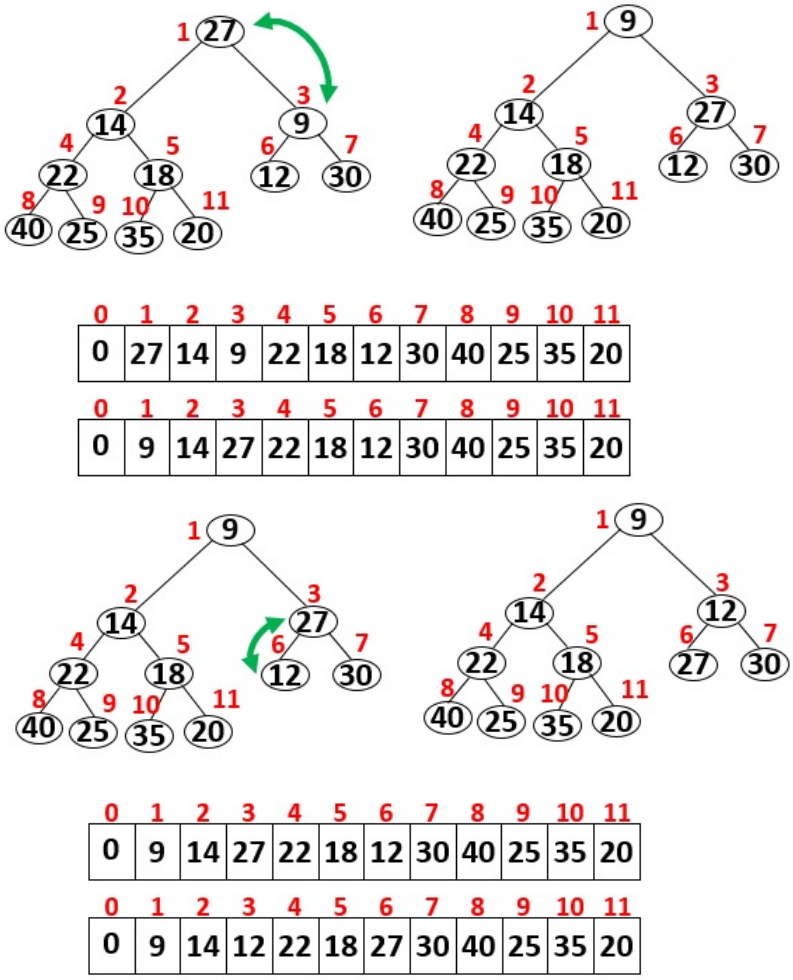
รูปที่ 5.12: ตัวอย่าง del_min(5)

ละตัวก็ได้ จากนั้นใช้ เมื่อ List มีตัวเดียวมันก็เรียงแล้ว พอ insert ตัวใหม่มาก็ใช้ Binary search หาตำแหน่งที่เหมาสมสำหรับตัวใหม่ ทำไปเรื่อยๆ หรือ เราจะใช้ฟังก์ชันที่เราเขียนไปสร้างใหม่

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

ให้ลองวาดรูป Tree ในแต่ละขั้นตอนที่ได้จากการ build_heap โดยให้ mylist เป็น [89,16,55,32,43] ต่อไปลองเรียกใช้งาน

```
import BinHeap
bh = BinHeap()
bh.insert(5)
bh.insert(7)
bh.insert(3)
bh.insert(11)
>>> print(bh.del_min())                    # min
3
```



รูปที่ 5.13: ตัวอย่างการปรับลง (Perculate down)

```
>>> print(bh.del_min())                    # min
5
>>> print(bh.del_min())                    # min
7
>>> print(bh.del_min())                    # min
11
>>>
```

5.4 Tree traversal

เมื่อเรามีเก็บข้อมูลใน Tree แล้ว คำถามต่อมาเราจะดูว่า Tree ของเรามีข้อมูลอะไรบ้างเราจะทำอย่างไร? ในหัวข้อย่อยต่อไปนี้เราจะมาพิตารณาวิธีการท่องไปใน Tree ว่ามีข้อมูลอะไรบ้าง ซึ่งไม่ว่างอย่างไรก็ตามวิธีการท่องไปในทรีนั้นต้องมีแบบแผนไม่ใช่ว่าจะเข้าไปดูข้อมูลอย่างไรก็ได้ ซึ่งอาจจะทำให้ได้ข้อมูลไม่ครบได้ ในการท่องไปใน Tree เรียกอีกอย่างว่า Traversal ไปยังทุกโหนดแบบมีหลายรูปแบบ แต่ส่วนใหญ่ที่เหมือนกันคือจะเริ่งต้นการค้นหาที่ Root

รูปแบบในการท่องไปใน Tree ในที่นี้จะยกตัวอย่าง 3 แบบขึ้นกับลำดับในการตรวจสอบว่าลำดับในการตรวจสอบโหนดตัวเองและโหนดลูกเป็นลำดับอย่างไร ได้แก่ preorder, inorder และ postorder

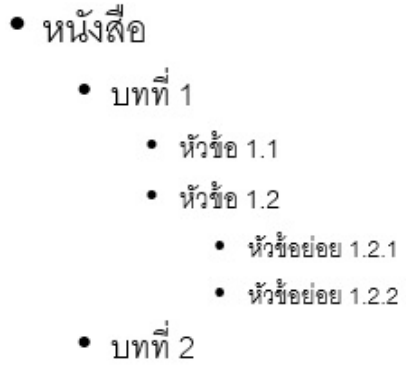
- **preorder** ในการ Traversal แบบ preorder เราจะตรวจสอบ Root ก่อนจากนั้นจะไปตรวจสอบแบบ Recursive กับ Left subtree แล้วตามด้วย Recursive กับ Right subtree (pre ก่อน นั่นคือตรวจสอบตัวเองก่อนตรวจสอบลูก)

- **inorder** ในการ Traversal แบบ inorder เราจะตรวจสอบแบบ Recursive กับ Left subtree ก่อนจากนั้นจะไปตรวจสอบ Root แล้วตามด้วย Recursive กับ Right subtree (in ใน นั่นคือตรวจสอบตัวเองตรงกลาง ระหว่าง Left subtree และ Right subtree)

- **postorder** ในการ Traversal แบบ postorder เราจะตรวจสอบแบบ Recursive กับ Right subtree ก่อนจากนั้นจะไปตรวจสอบ Root แล้วตามด้วย Recursive กับ Left subtree (post หลัง นั่นคือตรวจสอบตัวเองทีหลังตรวจสอบลูก)

ลองพิจารณารูปที่5.14ซึ่งเป็นตัวอย่างหัวข้อในหนังสือ หากเราต้องการเขียนโปรแกรมเข้าไปอ่านหัวข้อย่อยๆ ตามลำดับของหนังสือเล่มหนึ่งเราจะเลือกใช้การ Traversal แบบใด

หากเราต้องการอ่านหนังสือจากหน้าไปหลังตามหัวข้อ Traversal แบบ preorder เป็นคำตอบ เพื่อที่จะทำให้ได้ลำดับที่ถูกต้อง จะเริ่มต้นที่ หนังสือ จากนั้นจะเรียก preorder กับลูกทางซ้ายแบบ Recursive ซึ่งคือ บทที่ 1 จากนั้นเรียก preorder กับลูกทางซ้ายอีก จะได้ หัวข้อ 1.1 เนื่องจาก หัวข้อ 1.1 ไม่มีลูกแล้วก็จะไม่เรียก Recursive ก็จะย้อนกลับมาที่ บทที่ 1 แล้วจะไปทำงานต่อทางขวานั่นคือ หัวข้อ 1.2 เช่นเดิมก็จะไปตรวจสอบทางซ้ายก่อนจะได้



รูปที่ 5.14: ตัวอย่างหัวข้อในหนังสือ

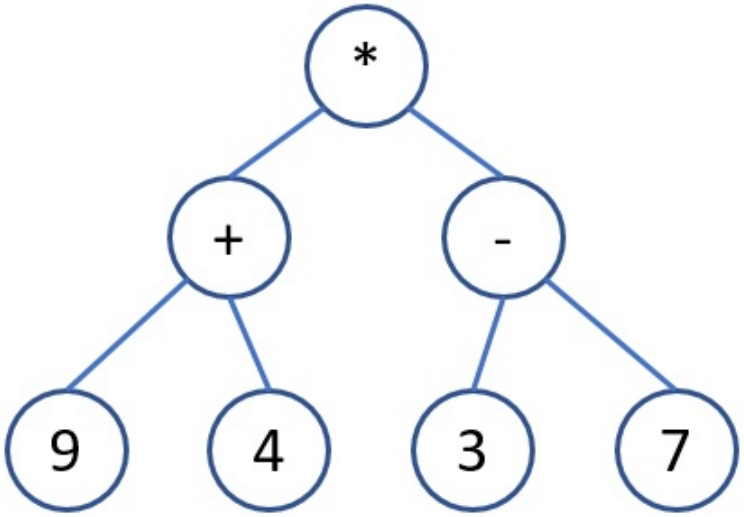
หัวข้อย่อย 1.2.1 จากนั้นเป็น หัวข้อย่อย 1.2.2 เมื่อเสร็จแล้ว จะย้อนกลับไปที่ หัวข้อ 1.2 เมื่อเสร็จแล้วย้อนกลับไปบทที่ 1 ซึ่งครับบทที่ 1 แล้วค่อยไปทำ บทที่ 2 ต่อไป ต่อไปเป็น code ของฟังก์ชัน preorder ที่อยู่ภายนอก

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

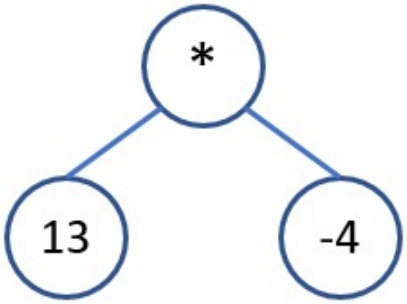
ต่อไปเป็นตัวอย่าง หากเขียนเพิ่มใน BinaryTree class

```
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())
```

แบบฝึกหัด ให้ลองทำ inorder



รูปที่ 5.15: ตัวอย่าง Parse tree ของ ((9+4)*(3-7))



รูปที่ 5.16: ตัวอย่าง Parse tree ของ ((13)*(-4))

5.5 Binary tree application

ในหัวข้อย่อยนี้จะนำเอา Binary tree มาใช้กับปัญหาจริง โดยตัวอย่างที่จะยกมาคือ การสร้าง Parse tree ซึ่ง Parse tree ที่ถูกใช้ในงานด้านภาษาเกี่ยวกับประโยคต่างๆ และในทางคณิตศาสตร์เพื่อหาคำตอบของ Expression เช่น ((9+4)*(3-7)) ดังรูปที่ 5.15

เรารู้อะไรกับ Expression นี้บ้าง เรารู้ว่าการคูณสำคัญกว่าการบวกและการลบ แต่เนื่องจากมีวงเล็บทำให้เรารู้ว่าก่อนที่จะคูณได้ ต้องคำนวณในวงเล็บให้เสร็จก่อน ลำดับชั้นของ Tree ทำให้เราเข้าใจลำดับการทำงานของการประมวลผล Expression ก่อนที่จะทำ Level บน เราต้องทำ Level ล่างให้เสร็จก่อนในรูปคือ เราต้องบวกลบให้เสร็จก่อนคูณ การบวกซึ่งเป็น Left subtree ได้ผลลัพธ์ 13 การลบได้ผลลัพธ์ -4 จากนั้นเราจะเปลี่ยน subtree เป็นโหนดได้แล้วค่อยไปคูณต่อดังรูปที่ 5.16

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 และ Operand เรารู้ว่าเมื่อไรก็ตามที่อ่านแล้วเจอวงเล็บเปิด นั่นคือเรากำลังจะเริ่ม Expression ใหม่ ดังนั้นเราควรจะสร้าง Tree ใหม่ ในทางกลับกันเมื่อเราเจอวงเล็บปิดแสดงว่าจบ Expression เรารู้อีกว่า Operand จะไปเป็น Leaf node และเป็นลูกของ Operator สุดท้ายเรารู้ว่าทุก Operator จะมีลูกทางซ้ายและขวา

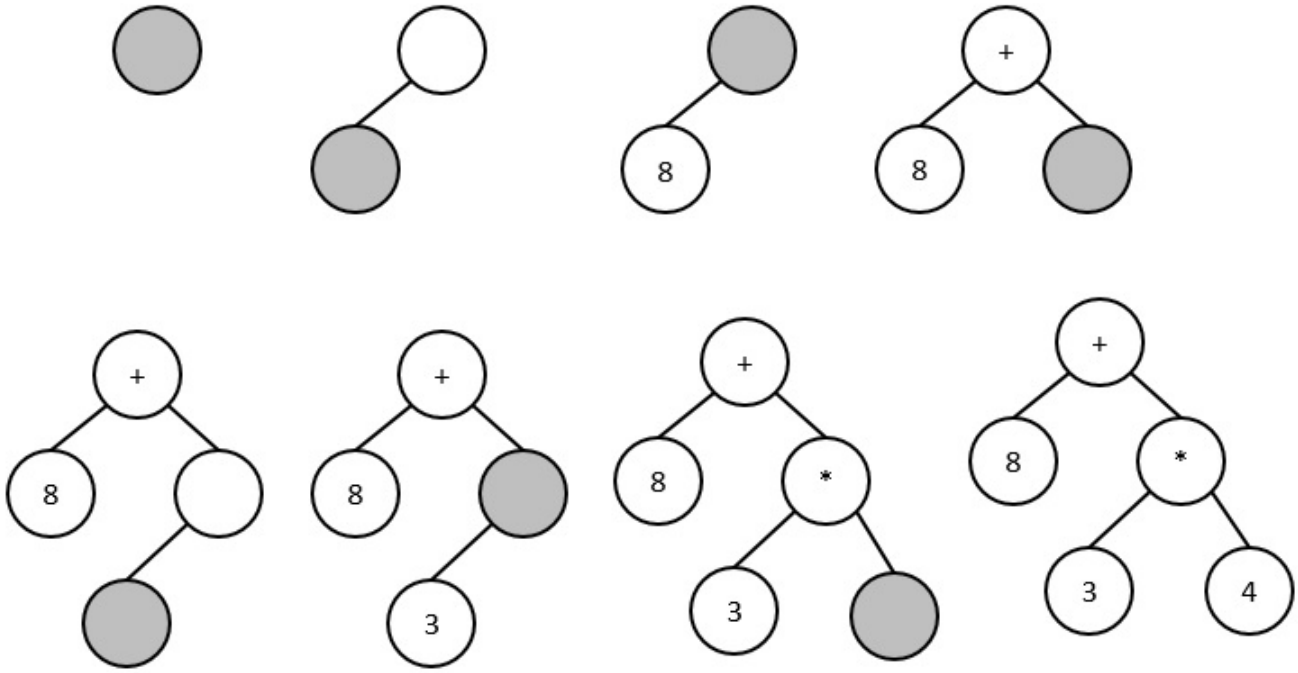
ทำให้เราสร้างกฎ 4 ข้อได้ดังนี้

1. ถ้า Token ปัจจุบันเป็น '(' จะเพิ่ม node ใหม่เป็นลูกทางซ้ายของ node ปัจจุบันแล้วย้ายไปลูกทางซ้าย
2. ถ้า Token ปัจจุบันเป็น ['+','-','*','/'] กำหนดค่าให้โหนดปัจจุบันเป็น operator ตัวนั้น เพิ่มโหนดใหม่เป็นลูกทางขวาแล้วย้ายไปลูกทางขวา
3. ถ้า Token ปัจจุบันเป็นตัวเลข กำหนดค่าให้โหนดนั้นเป็นนตัวเลขแล้วกลับมาที่ Parent
4. ถ้า Token ปัจจุบันเป็น ')' กลับไปที่ Parent ของโหนดปัจจุบัน

ตัวอย่าง สมมติว่ามี Expression (8+(3*4)) รูปที่5.17 แสดงขั้นตอนในการสร้าง Parse tree ทีละขั้นตามกฎที่นิยามไว้ก่อนหน้า

เราจะนำเอา class tree ที่เขียนไว้มาประยุกต์ใช้ จากตัวอย่างจะเห็นได้ว่าเราสนใจ โหนดปัจจุบัน Parent ของมัน ลูกทางซ้ายและลูกทางขวา ใน code เรามี get_left_child และ get_right_child แล้ว ยังขาดวิธีถามว่า parent คือใคร แล้วเราจะจัดการอย่างไรกับ parent เราจะใช้ Stack เพื่อจะได้ย้อนกลับได้ เมื่อไรก็ตามที่เราจะต้องลงไปที่โหนดลูก เราจะ push โหนดปัจจุบันลง Stack ไว้ เมื่อเราจะกลับมาเราก็ pop เอา ดังนั้นเราจะใช้ทั้ง class Stack และ BinaryTree

```
def build_parse_tree(fp_exp):
    fp_list = fp_exp.split()
    p_stack = Stack()
    e_tree = BinaryTreeNode('')
    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 ['+','—','*','/',')']:
```



รูปที่ 5.17: ตัวอย่างการสร้าง Parse tree ของ (8+(3*4))

```
            current_tree.set_root_val(int(i))
            parent = p_stack.pop()
            current_tree = parent
        elif i in ['+','—','*','/']:
            current_tree.set_root_val(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
```

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 แล้วต่อไปเราจะเขียนฟังก์ชันประมวลผล โดยคืนผลลัพธ์เป็น

ตัวเลข

เราจะเขียนเป็น Recursive เราจะเริ่มต้นการออกแบบ Recursive algorithm ด้วย 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 มาประมวลผลคำตอบของการคำนวณค่าของโหนดลูก

Code ของ evaluate เริ่มต้นจะได้ค่าของลูกทางซ้ายและขวาของโหนดปัจจุบัน ถ้าลูกทางซ้ายและขวา ประมวลได้ None แสดงว่า โหนดปัจจุบันเป็น Leaf node ถ้าโหนดปัจจุบันไม่ใช่ ศeaf node จะมองไปที่ Operator ของ Current node แล้วใช้มันหาผลลัพธ์จากการประมวลของลูกทางซ้ายและลูกทางขวา ในการเขียน code เราจะใช้ Dictionary กับ key '+', '-', '*', '/' ค่าที่เก็บใน Dictionary เป็นฟังก์ชันจาก Python operator module ซึ่ง 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()
```

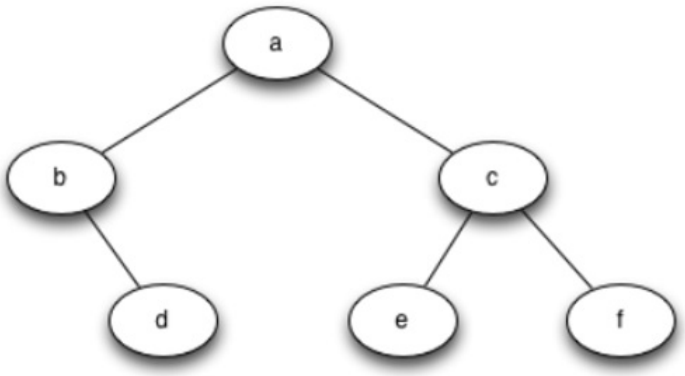
จากนั้นลองทดสอบด้วย

```
pt = build_parse_tree("( 10 + 5 ) * 3")
print(evaluate(pt))
```

5.6 แบบฝึกหัดท้ายบท

1. กำหนด code มาให้ต่อไปนี้จงวาดรูป Tree ผลลัพธ์ที่ได้

```
x = BinaryTreeNode('a')
insertLeft(x,'b')
```



รูปที่ 5.18: ตัวอย่าง Tree

```
insertRight(x,'c')
insertRight(get_right_child(x), 'd')
insertLeft(get_right_child(get_right_child(x)), 'e')
print(x)
```

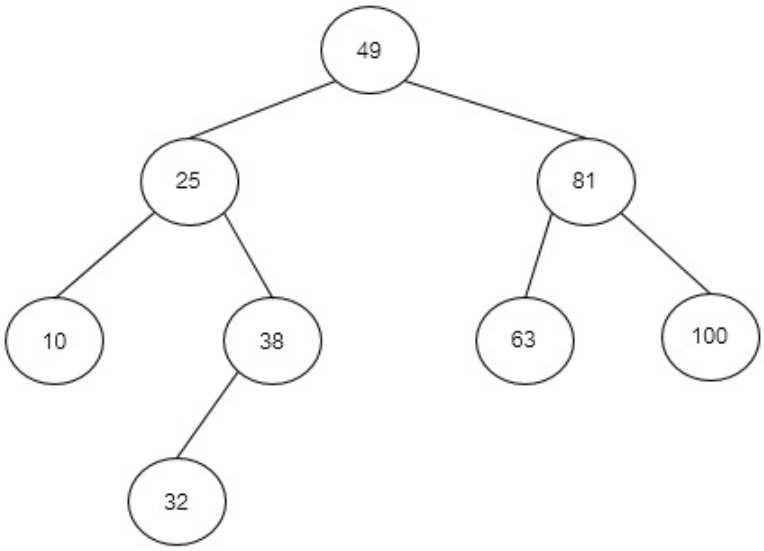
2. จงเขียนคำสั่งให้ได้ Tree ดังรูปที่5.18

3. จงเขียน function traversal แบบ inorder

4. ให้นำเอาความรู้ในบทนี้มาสร้าง Binary search tree ใน Binary search tree นั้นแต่ละโหนดมีลูกได้ไม่เกิน 2 โหนด และโหนดลูกทางซ้ายจะมีค่าน้อยกว่าโหนดพ่อ ส่วนโหนดลูกทางขวาจะมีค่ามากกว่าหรือเท่ากับโหนดพ่อ ดังตัวอย่างในรูป5.19

   แต่ละโหนดเก็บข้อมูล ค่าของโหนดตัวเอง ลูกทางซ้าย ลูกทางขวา โหนดพ่อ

```
class BinarySearchTree:
    def __init__(self):
        self.root = None
        self.size=0
    def sizeofTree(self):
        return self.size

class TreeNode:
    def __init__(self,val,left=None,right=None,parent=None):
```



รูปที่ 5.19: ตัวอย่าง Binary Search Tree

```
        self.val = val
        self.leftChild = left
        self.rightChild = right
        self.parent=parent
    def has_left_child(self):
        return self.leftChild
    def has_right_child(self):
        return self.rightChild
    def is_left_child(self):
        return self.parent and self.parent.leftChild==self
    def is_right_child(self):
        return self.parent and self.parent.rightChild==self
    def is_root(self):
        return not self.parent
    def is_leaf(self):
        return not(self.rightChild or self.leftChild)
t = BinarySearchTree()
t.put(45)
```

```
t.put(30)
t.put(70)
t.put(35)
t.put(10)
print(t.find_min())
```

กำหนด code เบื้องต้น(ยังไม่สมบูรณ์)มาให้จงเขียนฟังก์ชัน insert ข้อมูล ที่สอดคล้องกับคุณสมบัติของ Binary search tree และให้เขียนฟังก์ชัน findmin ซึ่งเป็นการหาข้อมูลตัวที่น้อยที่สุดใน Binary search tree

เอกสารอ้างอิง

[1] J. Chawachat. Selection and loop. http://www.cs.science.cmu.ac.th/course/204101/. [Online; accessed February 25, 2016].

[2] P. S. Foundation. Python 3.6.2 documentation. https://docs.python.org/3/. [Online; accessed February 25, 2016].

[3] B. Miller and D. Ranum. *Problem Solving with Algorithms and Data Structures Using Python*. Franklin Beedle Series. Franklin, Beedle & Associates, 2011.

[4] Wikipedia. Tower of Hanoi. https://en.wikipedia.org/wiki/Tower_of_Hanoi. [Online; accessed February 25, 2016].