



โครงสร้างซอฟต์แวร์แก้ไขปัญหา hilbert hotel

จัดทำโดย
กลุ่ม Hyper Hotel (4D)

ธีรวัฒน์ ลาภพานิช	67010448
เนติธร สุธรรมวรภาพร	67010497
ปรเมศวร์ โรจน์หล่อสกุล	67010534
ธีรภพ มหาสุข	67011464

รายงานฉบับนี้เป็นส่วนหนึ่งของการศึกษาในรายวิชา
01076109 OBJECT ORIENTED DATA STRUCTURES และ 01076110 OBJECT
ORIENTED DATA STRUCTURES PROJECT
สาขาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ ปีการศึกษา 2568

บทนำและทฤษฎีโรงแรม

รายงานนี้ใช้เพื่ออธิบายแนวคิดทฤษฎี Hilbert Hotel และพัฒนาซอฟต์แวร์เพื่อแก้ไขปัญหาแนวคิดนี้

Hilbert's Hotel เป็นแนวคิดของ David Hilbert เพื่อใช้อธิบายลักษณะของจำนวนที่มีค่าเป็นอนันต์ (infinity) โดยยกตัวอย่างโรงแรมที่มีจำนวนห้องเป็นอนันต์ และทุกห้องมีคนพักทุกห้องจนเต็ม หากมีแขกใหม่เข้ามา ก็สามารถย้ายแขกเดิมไปห้องอื่นได้เนื่องจากมีห้องเป็นอนันต์ เจ้าของโรงแรมต้องหาวิธีที่ในการจัดแขกให้แขกเดิม และแขกใหม่ทุกคนมีห้องพักโดยที่หมายเลขห้องไม่ซ้ำกัน

กรณีมีแขกใหม่จำนวน n คน จะย้ายแขกเดิมไปที่ ห้องเดิม + n เพื่อให้มีห้องว่าง n ห้องสำหรับแขกใหม่

กรณีมาเป็นคันรถจำนวน m คัน และในแต่ละคันมี n คน เจ้าของโรงแรมต้องสามารถจัดย้ายหมายเลขห้องแขกเดิมไปที่ห้องใหม่ เพื่อให้แขกที่เข้าใหม่มีห้องพัก

กรณีมีแขกจำนวนอนันต์เข้ามาพัก ก็สามารถย้ายแขกเดิมไปที่ห้องที่มีหมายเลขเป็น 2 เท่าของห้องเดิมได้ เพื่อให้ห้องเลขที่คี่ว่างและสามารถรองรับแขกจำนวนอนันต์ได้

การจัดการแขกในโรงแรม

ในโครงการนี้จะให้ AVL Tree เพื่อการจัดเก็บข้อมูลที่มีประสิทธิภาพและทำงานได้เร็วในการ Search หาข้อมูลจำนวนมาก

ในส่วนของการจัดการแขกจะมีแขก 3 แบบดังนี้

- 1.แขกที่พักอยู่ในโรงแรม
- 2.แขกที่ Check-in ปกติ (แขกที่มาจากช่องทางต่างๆ)
- 3.แขกที่ Check-in แบบ Manual

Data structure ที่ใช้

AVL Tree

โครงสร้างข้อมูลแบบ AVL Tree เป็นโครงสร้างข้อมูลที่มีการปรับสมดุลอัตโนมัติ (Self-Balancing) หลังการเพิ่มหรือลบข้อมูลทุกครั้ง โดยจะทำให้ความสูงของต้นไม้แตกต่างกันระหว่างด้านซ้ายและขวาไม่เกิน 1 ทำให้การค้นหา เพิ่ม และลบข้อมูลมีประสิทธิภาพอยู่ในระดับ $O(\log n)$ เสมอ

โครงสร้างของ AVL Tree

- Root : โหนดรากของต้นไม้ เป็นจุดเริ่มต้นของการดำเนินการ ค้นหา แทรก ลบ
- getHeight() : ความสูงของโหนด ใช้คำนวณสมดุลของต้นไม้
- getBalance() : ค่าผลต่างระหว่างความสูงของ subtree ซ้ายและขวา
ถ้าค่า Balance Factor อยู่นอกช่วง -1 ถึง 1 จะต้องมีการหมุนเพื่อปรับสมดุล

หลักการทำงานของ AVL Tree

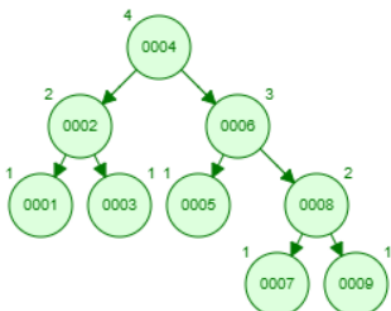
เมื่อมีการเพิ่มหรือลบโหนด จะมีการคำนวณค่า Balance Factor ย้อนกลับขึ้นไปจากโหนดที่เปลี่ยนแปลงจนถึงราก และทำการหมุนหากพบว่าไม่สมดุล โดยมีรูปแบบการหมุน 4 แบบ

1. LL Rotation
2. RR Rotation
3. LR Rotation
4. RL Rotation

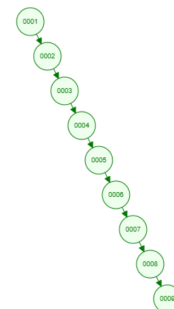
เหตุผลที่เลือกใช้ AVL Tree

ตัว AVL Tree จะทำ balance โหนดหากไม่สมดุล ทำให้ความลึกของ Leaf Node มีค่าใกล้เคียงกันเสมอ ทำให้ความเร็วในการค้นหาโหนดดีกว่า Tree แบบปกติที่ไม่มีการ balance ดังตัวอย่างได้ทำการเพิ่มเลข 1 - 9 เข้าไปใน AVL และ binary tree

ภาพ AVL tree



ภาพ binary tree



การออกแบบโปรแกรม

หน้าต่างเริ่มต้น

```
===== welcome to =====  
MAAN MAI RODO  
  
-----  
TYPE 1 - 7  
1 : Check-in guests by channel  
2 : Manually check-in guests  
3 : Manually check-out guests  
4 : Search room  
5 : Export to CSV file  
6 : Print Available Room  
7 : Quit App  
-----  
Select your option : s  
Invalid Input  
Press enter to continue|
```

หลังจากที่เปิดโปรแกรมมา ผู้ใช้จะเจอกับตัวเลือก 8 ตัวเลือก ได้แก่

- 1 : Check-in guests by channel
- 2 : Manually check-in guests
- 3 : Manually check-out guests
- 4 : Search room
- 5 : Export to CSV file
- 6 : Print Available Room
- 7 : Get Memory Usage
- 8 : Quit App

โดยให้ผู้ใช้ใส่ตัวเลือกเป็นตัวเลข 1 - 8 หากกรอกอย่างอื่นที่ไม่ใช่ตัวเลข จะแสดงคำว่า
Invalid Input

ผู้ใช้สามารถกด enter เพื่อกลับไปหน้าจอหลัก

โดยตัวเลือกที่แสดงผลในแต่ละตัวเลือก จะอธิบายในส่วนถัดๆไป

1.รับแขกที่เข้ามาจากแต่ละช่องทาง (Check-in guests by channel)

เมื่อเลือกเมนูนี้ผู้ใช้จะต้องกรอกแขกซึ่งเป็นตัวรขนาดใหญ่ ที่เข้ามาในแต่ละช่องทาง โดยจะต้องใส่ input เป็นชื่อช่องทาง (ห้ามมีเว้นวรรคระหว่างคำ) เว้นวรรคด้วยจำนวน หากจะใส่เพิ่ม ต้องใส่ (,) แล้วใส่ input เหมือนเดิมทำได้เรื่อยๆจนพอใจ

เมื่อกรอกจนพอใจแล้วจะแสดงหลอดความคืบหน้า (progress bar) แสดงการความคืบหน้าย้ายแขกเข้าโรงแรม (AVL Tree)

```
Select your option : 1

Input Format
Channel Name : amount , Channel Name : amount , ...
Example : Lopburi 10000, Home 1

Input : Home 2,FriedHome 1
Adding rooms: 100%| 3/3 [00:00<00:00, 43842.90room/s]
```

เมื่อหลอดความคืบหน้า (progress bar) เต็มแล้วจะแสดงผลการทำงานของ function ทั้ง memory ที่ใช้ และเวลาที่ใช้ในการทำงาน โดยจำนวนคนที่เข้ามาจะแปรผันตรงกับเวลาที่ใช้ในการทำงานของฟังก์ชันนี้

```
=====memory used stat=====
Function : insert
memory consumed: 1,323,008 bytes
function execution time : 0.015866756439208984 seconds
=====

Press enter to continue|
```

2. เพิ่มหมายเลขห้องแบบ manual (Manually check-in guests)

เมื่อเลือกเมนูนี้ ผู้ใช้จะต้องกรอกหมายเลขห้องที่ต้องการจะเพิ่ม จากนั้นตัวโปรแกรมจะเพิ่มห้องใหม่เข้าไปในโรงแรม (AVL Tree) พร้อมทั้งแสดงผลการทำงานของ function ทั้ง memory ที่ใช้ และเวลาที่ใช้ในการทำงาน โดยค่าของเลขที่รับเข้ามาจะแปรผันตรงกับเวลาที่ใช้ในการทำงานของฟังก์ชันนี้

```
Select your option : 2
Enter Room Number : 12

=====memory used stat=====
Function : manual_add
memory consumed: 4,096 bytes
function execution time : 0.00013875961303710938 seconds
=====

Press enter to continue|
```

ข้อมูลของแขก

โปรแกรมที่พวกเราออกแบบจะจำลองการแก้ปัญหา โดยแขกที่ได้รับมาจะมีอยู่ 2 แบบ คือแขกที่เข้ามาในแต่ละช่องทาง ดังตัวอย่าง

```
Room 5 : Home_1_2_0
Room 12 : manual
```

ในห้องหมายเลข 12 เกิดจากการเพิ่มคนโดยใช้ตัวเลือกที่ 2 เพิ่มหมายเลขห้องแบบ manual (Manually check-in guests)

ในห้องหมายเลข 5 เกิดจากการเพิ่มคนโดยใช้ตัวเลือกที่หนึ่งรับแขกเข้ามาจากแต่ละช่องทาง (Check-in guests by channel) โดยตัวเลขจะถูกคั่นด้วย _ ซึ่งแต่ละตัวมีความหมายดังนี้

คำที่ 1 (Home) : ชื่อช่องทาง

เลขตัวที่ 2 : ลำดับของช่องทางที่มา

เลขตัวที่ 3 : ลำดับที่นั่งของแขกในช่องทาง

เลขตัวที่ 4 : รอบที่แขกกลุ่มนี้เข้ามา (จำนวนครั้งที่ใช้ตัวเลือกที่หนึ่ง (Check-in guests by channel))

3. ลบหมายเลขห้องแบบ manual (Manually check-out guests)

เมื่อเลือกเมนูนี้ ผู้ใช้จะต้องกรอกหมายเลขห้องที่ต้องการจะลบ จากนั้นตัวโปรแกรมจะลบห้องในโรงแรม (AVL Tree) หากไม่เจอห้องนั้น จะแสดงผล Room not found พร้อมทั้งแสดงผลการทำงานของ function ทั้ง memory ที่ใช้ และเวลาที่ใช้ในการทำงาน

```
Select your option : 3
Enter room number : 5
Remove Room 5

=====memory used stat=====
Function : manual_remove
memory consumed: 0 bytes
function execution time : 0.0005366802215576172 seconds
=====

Press enter to continue|
```

4. ค้นหาห้อง (Search room)

เมื่อเลือกเมนูนี้ ผู้ใช้จะต้องกรอกหมายเลขห้องของแขกที่ต้องการจะค้นหา จากนั้นตัวโปรแกรมจะแสดงผลแขกพร้อมช่องทางที่แขกคนนั้นเข้ามา (AVL Tree) หากไม่เจอห้องนั้น จะแสดงผล Room not found พร้อมทั้งแสดงผลการทำงานของ function ทั้ง memory ที่ใช้ และเวลาที่ใช้ในการทำงาน

```
Select your option : 4
Enter room number : 4

Room 4 : FriendHome_2_1_0

=====memory used stat=====
Function : search
memory consumed: 0 bytes
function execution time : 0.0004572868347167969 seconds
=====

Press enter to continue|
```

5. บันทึกข้อมูลเป็นไฟล์ (Export to CSV file)

เมื่อเลือกเมนูนี้ โปรแกรมจะไปดึงข้อมูลของห้องที่มีแขกทั้งหมดในโรงแรม (AVL Tree) มาแปลงเป็น List แล้วเขียนแต่ละห้องเป็นบรรทัดในไฟล์ hotel.csv

```
Select your option : 5
```

```
Export Success
```

```
=====memory used stat=====
```

```
Function : export_to_file
```


```
memory consumed: 4,096 bytes
```

```
function execution time : 0.02538466453552246 seconds
```

```
=====
```

```
Press enter to continue|
```

ตัวอย่างข้อมูลในไฟล์

 hotel.csv

1 Room 2 : Home_1_1_0

2 Room 4 : FriendHome_2_1_0

3 Room 12 : manual

6. แสดงแขกทั้งหมดในโรงแรม

เมื่อเลือกเมนูนี้ โปรแกรมจะไปดึงข้อมูลของห้องที่มีแขกทั้งหมดในโรงแรม (AVL Tree) มาแปลงเป็น List แล้วแสดงผลบน terminal

```
Select your option : 6
Room 2 : Home_1_1_0
Room 4 : FriendHome_2_1_0
Room 12 : manual
Press enter to continue|
```

7. Get Memory Usage

เมื่อเลือกเมนูนี้ โปรแกรมจะแสดงหน่วยความจำที่ถูกใช้ในโปรแกรมนี้อย่างไร

```
Select your option : 7

=====memory used stat=====
Function : Program Memory Used
memory consumed: 24,793,088 bytes
=====

Press enter to continue|
```

8. Quit App

เมื่อเลือกเมนูนี้ ผู้ใช้จะสามารถออกจากโปรแกรมนี้อย่างไรได้ เมื่อกด enter ตัวโปรแกรมจะปิดทันที ผู้ใช้สามารถสามารถออกจากโปรแกรมได้อีกวิธีด้วยการกดคีย์ลัดปุ่ม Ctrl+C

การวิเคราะห์ Big O ของแต่ละฟังก์ชัน

1. การเพิ่มหมายเลขห้องแบบ manual

การเพิ่มห้องในโรงแรมนั้นจะเรียกใช้ผ่านฟังก์ชัน `manual_add()` ตามรูปด้านล่าง

```
#For requirement 4
@track
def manual_add(self, count):
    if self.tree.search(count) is None:
        self.tree.add(Room(f"manual",count))

    else:
        print("Room is not Empty")
    return
```

Function `manual_add()` มี parameter รับค่า `count` เป็นหมายเลขห้องที่ต้องการจะเพิ่ม ในส่วนของบรรทัดต่อมาจะทำการค้นหาห้องด้วยคำสั่ง `search()` ซึ่งในส่วนของ function `search()` ห้องจะถูกนำไปขยายความในหัวข้อ 4. การค้นหาหมายเลขห้อง ซึ่ง Big O ของ function นี้จะมีค่าเป็น $O(\log n)$ โดยที่ n คือจำนวน node ทั้งหมดใน tree

เมื่อตรวจสอบว่าห้องถูกสร้างแล้ว หากพบห้อง จะถือว่าห้องเต็มและแจ้งว่า "ห้องไม่ว่าง" แต่หากไม่พบ จะเรียกใช้คำสั่ง `add()` เพื่อเพิ่มห้องตามรูปในหน้าต่อไป

```

def add(self, data):
    self.root = self._add(self.root, data)

def _add(self, root, data):
    if root is None:
        return AVLNode(data)
    if int(data) < int(root.data):
        root.left = self._add(root.left, data)
    else:
        root.right = self._add(root.right, data)
    root = self.rebalance(root)
    return root

```

Function add() จะรับ parameter ในชื่อ data ซึ่งเป็น class Room และส่งต่อไปให้ function _add() ที่รับ parameter เป็น root ซึ่งเป็น node และ data ที่เป็น Room โดยเริ่มจากการเช็คว่างที่ root ว่างหรือไม่ ถ้าใช่จะทำการสร้างและสร้าง node ใหม่ แล้วต่อมาจะทำ search หา node ที่ data นั้นควรอยู่ด้วยการท่อง tree ไปตาม node ต่างๆโดยใช้วิธีเปรียบเทียบหมายเลขห้องมากกว่าหรือน้อยกว่า ซึ่งจะได้ค่า Big O จากการค้นหาคือเป็น $O(\log n)$

ในขั้นตอนสุดท้าย tree จะทำการ rebalance ตัวเองผ่าน function rebalance() หลังจากการ add node เสร็จสิ้น โดยปกติการหมุน tree จะมีค่าเป็น $O(1)$ แต่เนื่องจากการ rebalance จะถูกเรียกเป็น stack เริ่มจากล่างสุด (ที่ไม่ใช่ node ที่เพิ่ง add) ไปจนถึงบนสุด (root / top node) ซึ่งจากการเรียกใช้นี้จะได้ค่า Big O จากการ rebalance เป็น $O(\log n)$

จากทั้งหมดใน function นี้จะสามารถวิเคราะห์ค่า Big O ของ function manual_add() เป็น $O(\text{count}) * (O(\log n) + O(\log n)) + O(\log n) = O(\text{count} * \log n)$

2.การลบหมายเลขห้องแบบ manual

การลบห้องในโรงแรมนั้นจะเรียกใช้ผ่านฟังก์ชัน manual_remove() ตามรูปด้านล่าง

```
#For requirement 5
@track
def manual_remove(self, room_number):
    result = self.tree.search(room_number)
    if not result:
        print("Room not found")
        return
    self.tree.remove(int(room_number))
    print(f"Remove Room {room_number}")
    return
```

Function manual_remove() มี parameter รับค่า room_number เป็นหมายเลขห้องที่ต้องการจะ remove ห้องออก ซึ่งในส่วนของ function search() ห้องจะถูกนำไปขยายความในหัวข้อ 4.การค้นหาหมายเลขห้อง ซึ่ง Big O ของ function นี้จะมีค่าเป็น $O(\log n)$ โดยที่ n คือจำนวน node ทั้งหมดใน tree

หลังจากการค้นหาห้องใน Hotel สำเร็จจะทำการเช็คว่างห้องที่หาไม่ ถ้าไม่เจอก็ print ข้อความไม่เจอ พร้อมกับ return เพื่อจบการทำงานของโปรแกรม ในกรณีที่เจอจะทำการเรียกใช้ function remove() โดยการส่ง parameter เป็นเลขห้องไป แล้ว print แสดงผลว่าทำการ remove room เรียบร้อยแล้ว พร้อมกับ return เพื่อจบการทำงานของ function

```
def remove(self, data):
    self.root = self._remove(self.root, data)

def _remove(self, node, data):
    if node is None:
        return None
    if data < node.data:
        node.left = self._remove(node.left, data)
    elif data > node.data:
        node.right = self._remove(node.right, data)
    else:
        if node.left is None:
            return node.right
        elif node.right is None:
            return node.left
        temp = self.get_successer(node.right)
        node.data = temp.data
        node.right = self._remove(node.right, temp.data)
    return self.rebalance(node)
```

Function `remove()` จะเรียกใช้ function `_remove()` ที่รับ parameter เป็น node และ data เป็นหมายเลขห้องที่ต้องการจะลบ ในส่วนแรกจะทำการค้นหา node ใน tree ที่ต้องการจะลบโดยใช้การเปรียบเทียบหมายเลขห้องเพื่อที่จะท่อง tree ไปตาม node ต่างๆจนเจอ node ตัวที่ต้องการจะลบ ซึ่งสามารถคำนวณค่า Big O ออกมาได้เป็น **$O(\log n)$**

ส่วนถัดมาจะทำการเช็คค่า node ตัวนั้น มีลูกไหม ถ้าไม่มีลูกหรือมีลูกหนึ่ง จะให้ node ลูกขึ้นมาแทนตัวมัน หรือ remove เลย (กลายเป็น None) ในกรณีที่ไม่มีลูก แต่ถ้ามีลูกถึงสองฝั่งจะทำการหา successor คือ หา leaf node ที่มีค่าน้อยที่สุดของลูกฝั่งที่มากกว่า node นั้น เพื่อนำ data ของ leaf node นั้นมาแทนที่ node ที่เราต้องการจะลบ แล้วทำการเปลี่ยนไปลบ successor หรือ leaf node นั้นทิ้ง ซึ่งสามารถคำนวณ Big O การหา successor ได้เป็น **$O(\log n)$** ในสุดท้ายจะทำการ rebalance tree เพื่อให้ tree ยังคง balance ซึ่งมีค่า Big O เป็น **$O(\log n)$**

จากทั้งหมดใน function นี้จะสามารถวิเคราะห์ค่า Big O ของ function `manual_remove()` เป็น $O(\log n) + O(\log n) + O(\log n) + O(\log n) = \mathbf{O(\log n)}$

3.การจัดเรียงลำดับหมายเลขห้อง

การจัดเรียงเลขห้องในโรงแรมนั้นจะเรียกใช้ผ่านฟังก์ชัน update() ตามรูปด้านล่าง

```
def get_all_available_room(self):
    data = self.tree.inorder()
    if data is None:
        print("\nNo Room\n")
        return
    for item in data:
        print(item)
```

```
def inorder(self):
    if self.root == None:
        return None
    op = []
    return self._inorder(self.root, op)

def _inorder(self, focus, List):
    # Left → Root → Right
    if focus.left is not None:
        self._inorder(focus.left, List)
    List.append(focus.data)
    if focus.right is not None:
        self._inorder(focus.right, List)
    return List
```

Function `get_all_available_room` มี เป็น value ซึ่งเป็น Function ที่จะดึงข้อมูลของห้องทุกห้องที่มีคนเข้าพักในโรงแรมนี้เป็น List มา โดยจะเรียกใช้ function `inorder` ของ AVLTree เพื่อเรียกข้อมูลที่มีการจัดเรียงแบบ inorder การทำงานใน function เริ่มจากการเช็คค่าที่ root ของ tree ว่าว่างหรือไม่ จากนั้นจะสร้าง List เพื่อนำไปเก็บค่าใน Tree และจะ return ค่าเป็น function `_inorder()` ที่รับ parameter เป็น focus ซึ่งเป็น node และ List ที่เป็น List

ใน function `_inorder` จะทำการเช็คค่า node นั้นมีลูกทางไหนบ้าง ถ้ามีลูกจะเรียกใช้ฟังก์ชันนี้กับลูกเช่นเดียวกับ node แล้วทำการเพิ่มค่าไปใน List ที่ได้รับมาจาก parameter แล้วคืนค่าเป็น List ซึ่งจะได้ค่า Big O จากการอัปเดตค่านี้เป็น **O(n)**

4.การค้นหาหมายเลขห้อง

การค้นหาห้องในโรงแรมด้วยหมายเลขห้องนั้นจะเรียกใช้ผ่านฟังก์ชัน search() ตามรูปด้านล่าง

```
#For requirement 7
@track
def search(self, room_number):
    result = self.tree.search(room_number)
    if result:
        op = f"{result}"
    else:
        op = "Room Not found"
    print(f'\n{op}')
```

```
def search(self, data):
    return self._search(self.root, data)

def _search(self, node, data):
    if node == None:
        return None

    if node.data == data:
        return node
    if node.data > data:
        return self._search(node.left, data)
    else:
        return self._search(node.right, data)
```

Function search() จะรับ parameter data เป็นเลขห้องแล้วเรียกใช้ _search() ซึ่งจะทำการ Search ใน AVL Tree แบบ Recursive และเนื่องจากการเก็บข้อมูลใน AVL Tree มีการ rebalance node ทำให้ Big O มีค่าเป็น $O(\log n)$

ภาคผนวก

Source code : <https://github.com/PluemDontKnowToCode/MaanMaiRood>

Hibert Hotel: https://en.wikipedia.org/wiki/Hilbert%27s_paradox_of_the_Grand_Hotel
<https://www.youtube.com/watch?v=HLTjDXT9SqQ>

การ update ห้อง

ในส่วนของการ update() มี parameter รับค่า value เป็น int โดยจะเรียกใช้ function _update() ที่รับค่าเป็น root ที่เป็น Node และ value ซึ่งเป็นค่า int

```
def update(self):
    self._update(self.root, 0)

def _update(self, focus, value):
    if focus is None:
        return None
    if focus.left is not None:
        self._update(focus.left, value)
    focus.data.number = Formula.triangular_accumulate(focus.data.number, value)
    if focus.right is not None:
        self._update(focus.right, value)
```

ใน function _update จะทำการเช็คค่า node นั้นมีลูกทางไหนบ้าง ถ้ามีลูกจะเรียกใช้ฟังก์ชันนี้กับลูกเช่นเดียวกับ node แล้วทำการเพิ่มค่าไปในและทำการ เปลี่ยนเลขห้องของ node นั้น โดยเลขห้องใหม่จะนำไปเข้าสมการ cantor pairing function (Triangular Function) ที่รับ parameter เป็น number (ลำดับของคน) และ value (ลำดับของรถ)