Skip List: Linked List ที่มีทางด่วนพิเศษ

อ. เนรมิต ชุมสาย ณ อยุธยา*

Abstract

Skip List is a probabilistic alternative to Balanced Trees, or in short, another linked list which behave like a self-balanced binary search trees. Skip List was created by Professor William Pugh at the University of Maryland in 1990. It has an average running time of $\log_2 n$ and a linear running time at the worst case. This article will show readers what Skip List is and how it can be created using Java.

คำสำคัญ

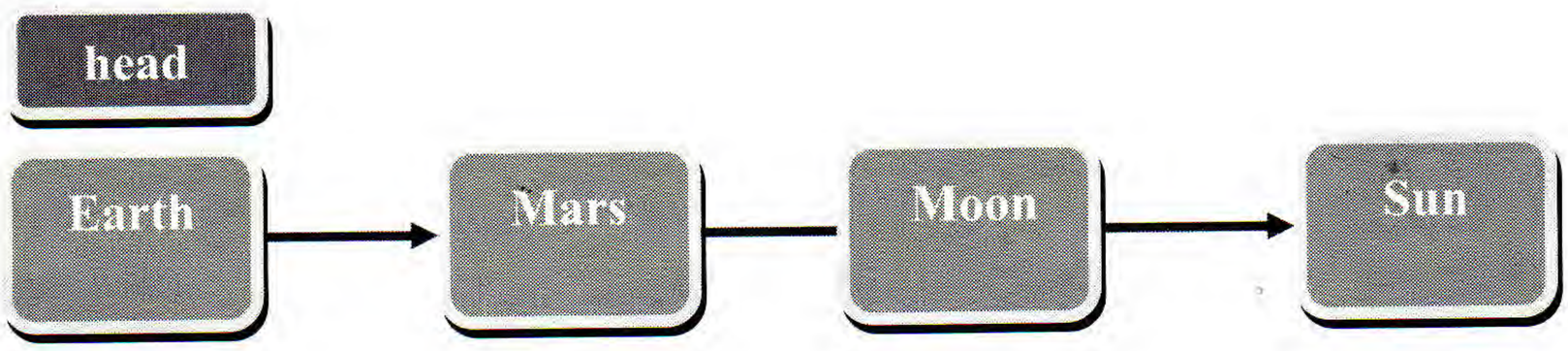
Linked List

บทนำ

Skip List เป็นโครงสร้างที่พัฒนามาจาก Linked List โดยศาสตราจารย์ William Pugh เมื่อปี ค.ศ. 1990 ด้วยการกำหนดให้ node ทุกตัวที่อยู่ใน list มีความสูงที่แตกต่างกันเพื่อให้การเข้าหาทำได้ง่ายขึ้น บทความนี้จะแนะนำถึงโครงสร้างของ Skip List และจะอธิบายถึง code ที่ใช้ในการสร้าง Skip List ในภาษา Java

ก่อนจะถึง Skip List

ในการเรียนเรื่องโครงสร้างข้อมูลและอัลกอริธึมนั้น หนึ่งในโครงสร้างที่เราขาดไม่ได้คือ Linked List ซึ่งเป็นโครงสร้างพื้นฐานตัวหนึ่งที่อาจเรียกอีกแบบได้ว่า Unary Tree กล่าวคือในทุก ๆ ส่วน (node) ของ Linked List จะมีส่วนที่ใช้เป็นที่เก็บข้อมูล (data) และอีกส่วนหนึ่งที่ใช้เป็นตัวเชื่อม (link) ไปยัง node ที่อยู่ติดกันดังที่แสดงให้ดูในรูปที่ 1 เราจะเห็น node 4 node ที่เชื่อมต่อกันคล้ายกับโบกี้ของรถไฟ

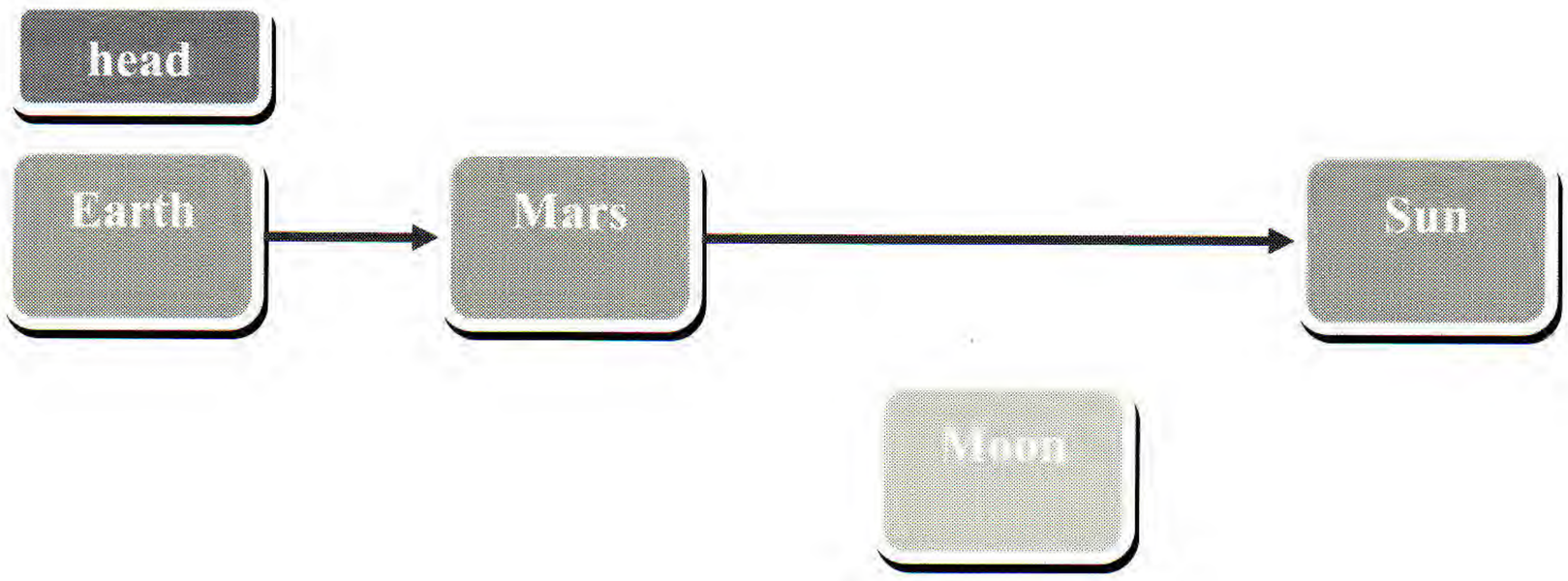


รูปที่ 1. Linked List

* คณบดีคณะวิทยาศาสตร์และเทคโนโลยี มหาวิทยาลัยฟาร์อีสเทอร์น

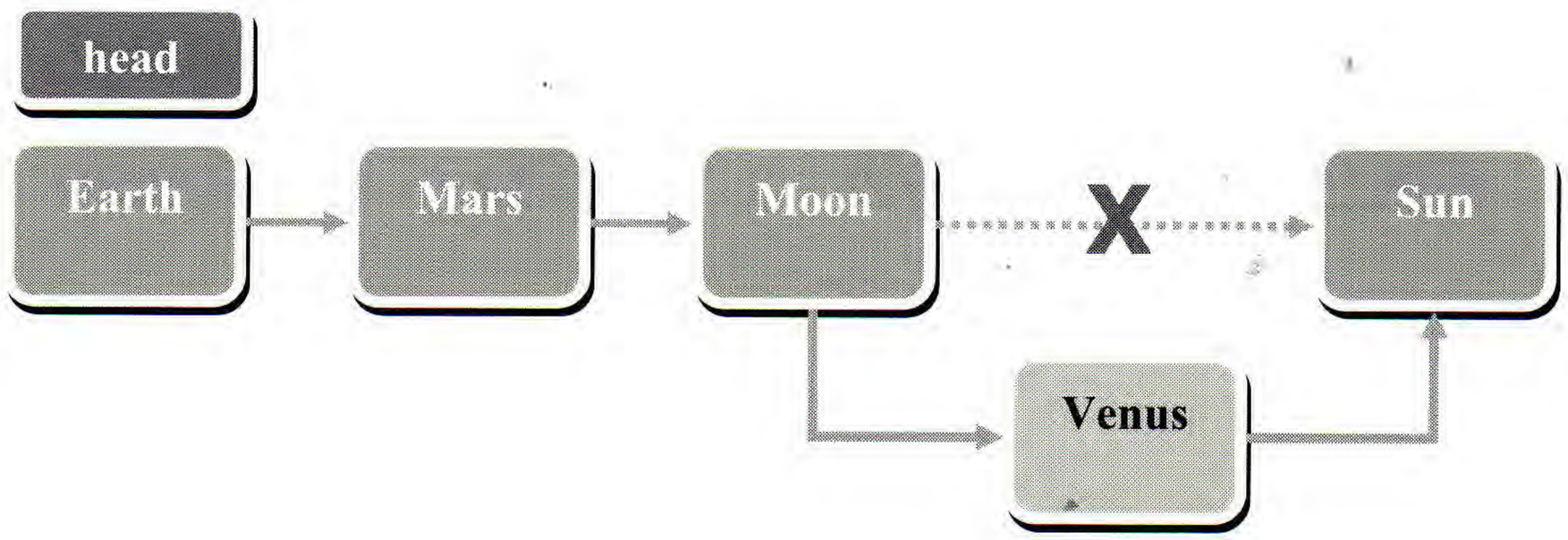
การเข้าหา Linked List นั้นเราต้องทำผ่านตัวอ้างอิง (reference) ของ Linked List ซึ่งโดยทั่วไปเราเรียกว่า head เราไม่จำเป็นต้องเก็บ address ของทุก node ที่อยู่ใน Linked List ทั้งนี้ก็เพราะว่าทุกๆ node ที่อยู่ใน Linked List เก็บข้อมูลเหล่านี้ไว้แล้ว

เวลา (running time) ที่ต้องใช้ในการค้นหา node ใน Linked List เป็นแบบที่เราเรียกว่า linear running time เช่นเดียวกันกับการค้นหาข้อมูลใน array นั่นก็คือ ถ้าเราต้องการหา node: Mars เราต้องเริ่มต้นที่ node: Earth ทำการตรวจสอบไปทีละ node จนกว่าจะเจอ เราไม่สามารถกระโดดข้ามจาก node หนึ่งไปยังอีก node หนึ่งได้ การลบ node ออกจาก Linked List ก็ใช้เวลาที่เป็น linear running time เช่นเดียวกันเราต้องทำการค้นหา node ที่ต้องการให้เจอก่อน แล้วจึงทำการโยกย้ายตัวอ้างอิงที่เชื่อมต่อไปยัง node ก่อนหน้าและ node ที่ตามมา รูปที่ 2 แสดงการลบ node: Mars ออกจาก Linked List



รูปที่ 2. การลบ node: Mars ออกจาก Linked List

ส่วนเวลาที่ใช้ในการนำ node เข้าสู่ Linked List นั้นขึ้นอยู่กับว่า node ทั้งหมดที่อยู่ใน Linked List มีการจัดเรียงก่อนแล้วหรือไม่ ซึ่งถ้าไม่มีการจัดเรียงการนำ node เข้าก็ใช้เวลาที่เราเรียกว่า constant time ทั้งนี้ก็เพราะว่าเรานำ node เข้าสู่ Linked List ทางด้านหน้า ซึ่งก็ทำได้ด้วยการสร้าง node ใหม่พร้อมกับกำหนดให้ตัวอ้างอิงชี้ไปยัง node ตัวแรกที่อยู่ใน Linked List หลังจากนั้นก็ทำการย้าย head มาชี้ที่ node ใหม่ที่สร้างขึ้นแต่ถ้าการนำ node เข้าต้องมีการจัดเรียงเราต้องหาตำแน่งให้เหมาะสมสำหรับ node นั้นๆ ซึ่งเป็นไปได้อยู่สามกรณีคือ 1) node ที่นำเข้าอยู่ด้านหน้า 2) node ที่นำเข้าอยู่ด้านหลัง และ 3) node ที่นำเข้าอยู่ระหว่าง node ซึ่งการค้นหาจะใช้เวลามาก (กว่าการนำเข้าแบบไม่จัดเรียง) เช่นดังที่แสดงในรูปที่ 3

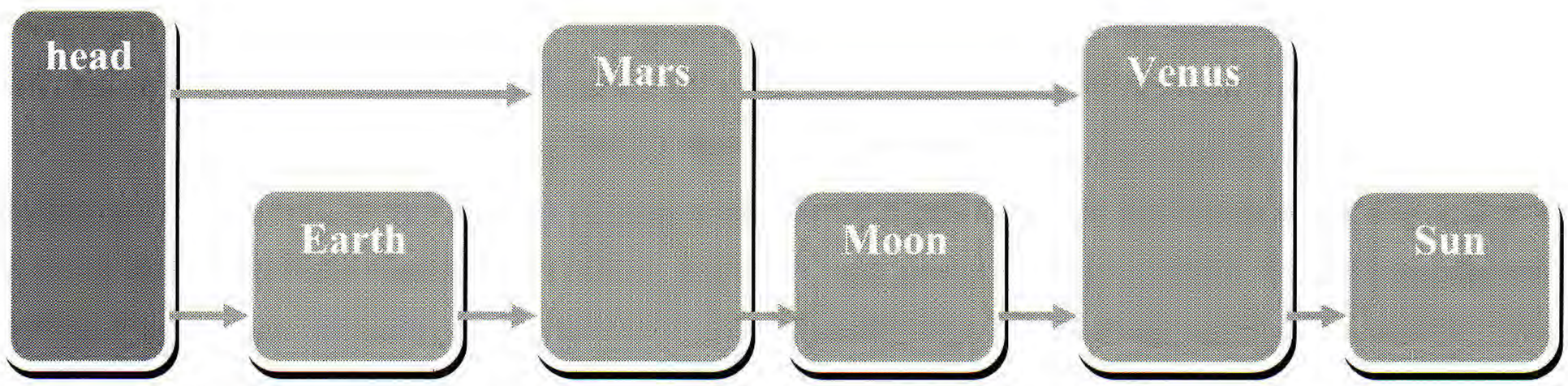


รูปที่ 3. การนำ node เข้าสู่ Linked List แบบจัดเรียง

เราต้องหา node ก่อนหน้า node ที่ต้องนำเข้าให้เจอ เช่นจะนำ node: Venus เข้าเราต้องเดินเข้าหา node: Earth, node: Mars, และ node: Moon ทำการเชื่อม node: Moon กับเข้า node: Venus หลังจากนั้นก็เชื่อม node: Venus เข้ากับ node: Sun ผู้อ่านจะเห็นว่าเราไม่สามารถกระโดดข้าม node อื่นๆ ไปยังตำแหน่งที่ต้องการได้เหมือนที่เราทำได้กับ array อย่างไรก็ตาม Linked List เอื้อความสะดวกในเรื่องของการนำเข้าและดึงออกที่ไม่ต้องวุ่นวายถึงขนาดของ Linked List ในการนำข้อมูลเข้าสู่ array นั้นถ้าความจุสูงสุดของ array มีการใช้เต็มที่แล้ว การนำข้อมูลเข้าใหม่ต้องขยายขนาดของ array และทำการโยกย้ายข้อมูลในตำแหน่งต่างๆ ให้สอดคล้องกันตามลำดับก่อนหลังก่อนและหลังการนำข้อมูลเข้า [2] ดังนั้นถ้าเรารู้ว่าจำนวนของข้อมูลสูงสุดที่เราต้องดูแลเป็นจำนวนเท่าใดการใช้ array ก็เป็นสิ่งเหมาะสมที่สุดแต่ถ้าเราไม่รู้ Linked List ก็คงต้องเป็นโครงสร้างที่เราต้องใช้

Skip List

ในปี ค.ศ. 1990 ศาสตราจารย์ William Pugh แห่งมหาวิทยาลัย Maryland ได้คิดถึงวิธีการที่จะทำให้การค้นหาข้อมูลใน Linked List ที่มีการจัดเรียงข้อมูลอยู่แล้วใช้เวลาได้น้อยลง วิธีการที่ท่านได้ทดลองทำคือ กำหนดให้ node แต่ละ node มีตัวเชื่อมไปยัง node ที่อยู่ห่างออกไปสอง node เพิ่มขึ้นมาอีกตัวหนึ่ง ดังที่แสดงในรูปที่ 4

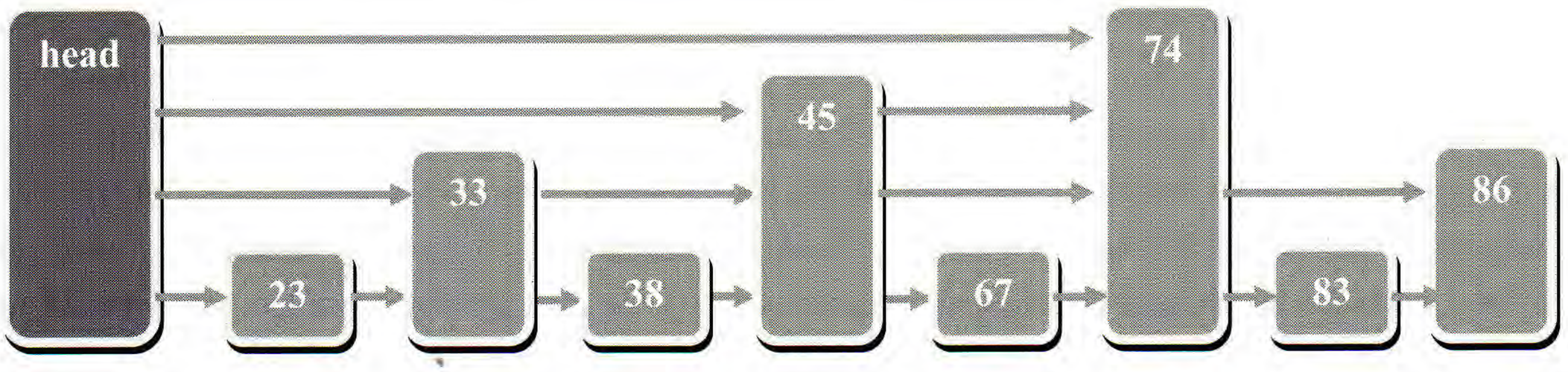


รูปที่ 4. Skip List

ในการเข้าหา node ใน Linked List นั้นเราต้องใช้เวลาในการตรวจสอบทุก node จนกว่าจะเจอ node ที่เราต้องการ แต่ถ้าเรากำหนดดังที่แสดงในรูปที่ 4 เราจะเห็นว่าการค้นหา node จะใช้เวลาน้อยลงครึ่งหนึ่ง เช่น ในการค้นหา node: Venus แทนที่จะต้องผ่าน node: Earth, node: Mars, และ node: Moon เราก็เพียงแค่ผ่าน node: Mars เพียง node เดียว กระบวนการค้นหาจะเริ่มต้นที่ node: head ซึ่งเป็น node ที่ไม่มีการเก็บข้อมูลและมีความสูงเท่ากับจำนวนสูงสุดของระดับที่มีอยู่ใน list โดยเราจะเริ่มที่ระดับสูงสุดก่อน (เพราะทำให้เราสามารถข้าม node ที่อยู่ต่ำกว่าได้) เมื่อเราตามตัวเชื่อมมาถึง node: Mars เราก็ถามตัวเองว่า node: Mars อยู่ก่อนหรือหลัง node: Venus ถ้าอยู่ก่อนเราก็รู้ว่า node: Venus ต้องอยู่ทางด้านขวาของ node: Mars เราก็ตามตัวเชื่อมไปถัดไปซึ่งในที่นี้ (บังเอิญ) ก็คือ node: Venus เราก็ยุติการค้นหา

ในการค้นหา node: Moon นั้นเราก็เริ่มที่ node: Mars เหมือนเดิมถามว่า node: Mars กับ node: Moon ใครอยู่ก่อนใคร ในที่นี้เรารู้ว่าว่า node: Moon อยู่ทางขวาของ node: Mars ดังนั้นเราก็ตามตัวเชื่อมไปเจอ node: Venus ทำให้เรารู้ว่า node: Moon ต้องอยู่ระหว่าง node: Mars และ node: Venus เราจึงต้องลดระดับลงมาอีกระดับหนึ่ง แล้วก็เริ่มกระบวนการค้นหาต่อ ซึ่งในที่สุดก็เจอ node: Moon

ศาสตราจารย์ Pugh เห็นว่าเวลาถึงแม้จะลดลงแล้วแต่ก็ยังไม่ดีมากเท่าใด (สิ่งที่มีอยู่ในใจคือ $\log_2 n$) ดังนั้นจึงทำการทดลองด้วยการเพิ่มความสูงของระดับตามจำนวนของ node ที่มีอยู่ใน List เพื่อให้ได้ running time ของการค้นหาเป็น $\log_2 n$ เช่น ถ้าจำนวนของ node เป็น 8 ความสูงก็ต้องเป็น 3 ($\log_2 8 = 3$) ถ้าจำนวนของ node เป็น 16 ความสูงก็เป็น 4 เป็นต้น ดูรูปที่ 5



รูปที่ 5. Skip List ที่มีความสูงต่างๆ กันแต่สูงสุดเท่ากับ 4

จากรูปที่ 5 ผู้อ่านจะเห็นถึงความสูงของ node ที่ต่างกัน เช่น node: 74 มีความสูงเท่ากับ 4, node: 45 มีความสูงเท่ากับ 3, node: 33 และ 86 มีความสูงเท่ากับ 2, ส่วน node: 23, 38, 67 และ 83 มีความสูงเท่ากับ 1 ซึ่งถ้าดูแล้วก็สามารถสรุปได้ว่าทุกๆ node ที่อยู่ในระดับ 2^{ith} จะเชื่อมอยู่กับ 2^{i} node ที่อยู่ถัดไป เช่น node: 23 เชื่อมอยู่กับ 20 node คือ node: 33, node: 33 เชื่อมอยู่กับ 2^{1} node ถัดไปคือ node: 38 และ node: 45, node: 45 เชื่อมอยู่กับ 2^{2} node ถัดไปคือ node: 67, 74, 83, และ node: 86 การกำหนดแบบนี้ทำให้การค้นหาเป็นไปอย่างรวดเร็ว อย่างไรก็ตามวิธีการนี้ทำเสียเวลามากในการนำข้อมูลเข้าและการดึงข้อมูลออก เพราะต้องมีการปรับตัวเชื่อมและสร้างระดับใหม่ให้เหมาะสมตามตำแหน่งที่อยู่

จากสาเหตุดังกล่าวศาสตราจารย์ Pugh จึงเลือกใช้การกำหนดความสูงของ node ด้วยการสุ่มแทนการกำหนดเฉพาะที่ (เช่นในรูป) ก็เพราะว่าสามารถสร้างได้ง่าย และยังมี running time ที่เป็น $\log_2 n$ ทั้งการนำข้อมูลเข้า การดึงข้อมูลออก และการค้นหาข้อมูล ซึ่งในการสุ่มค่าความสูงนั้นศาสตราจารย์ Pugh ใช้อัลกอริธึมที่นำเอาการกระจาย (probability distribution) มาช่วย คือใช้ 50% ที่ความสูงในระดับ 1, 25% ในความสูงระดับ 2, 12.5% ในความสูงระดับ 3 เป็นต้น แต่เพื่อให้เป็นการง่ายอัลกอริธึมที่นำมาใช้จะใช้ค่าความเป็นไปได้ p ที่ 0.5 และสร้างค่าสุ่มจาก random number generator ในการหาระดับความสูงของแต่ละ node ในlist (เมื่อมีการนำข้อมูลเข้า) เช่นที่เห็นใน code นี้

```
private int randomLevel() {
    int level = 0;

    while(level < maxLevel && Math.random() < prob)
        level++;

    return level;
}
```

การสร้าง Skip List ใน Java

เราจะเริ่มด้วยการสร้าง class SkipNode สำหรับเก็บข้อมูลและตัวเชื่อมดังนี้

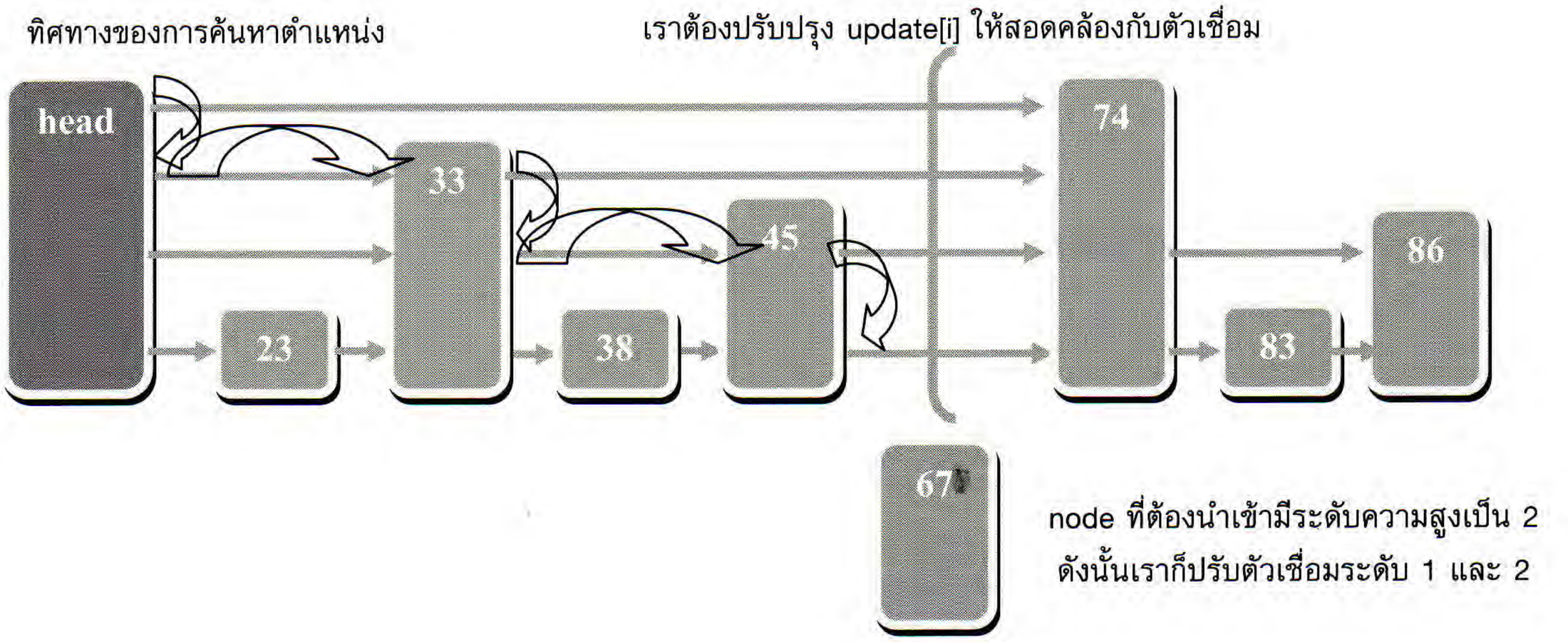
```
public class SkipNode {
    int key;                    //key to insert
    SkipNode []forward;         //links

    public SkipNode(int level, int key) {
        this.key = key;
        forward = new SkipNode[level+1];
    }
}
```

เพื่อให้เป็นการง่ายต่อการทำความเข้าใจ node ที่เราสร้างขึ้นจะเก็บเฉพาะข้อมูลที่เป็น int เท่านั้น ในการเชื่อมต่อกับ node อื่นๆ ใน list เราจะใช้ตัวแปร forward ที่เป็น array ที่ใช้เก็บตัวเชื่อมในระดับต่างๆ ขั้นตอนต่อไปก็คือการสร้าง class SkipList

Class SkipList จะประกอบไปด้วย method หลัก ๆ อยู่สามตัวคือ method สำหรับการนำข้อมูลเข้า (insert(key)), method สำหรับการดึงข้อมูลออก (delete(key)), method สำหรับการค้นหาข้อมูล (search(key))

การนำข้อมูลเข้าและการดึงข้อมูลออก

กระบวนในการนำข้อมูลเข้าเริ่มต้นด้วยการค้นหาตำแหน่งที่เหมาะสมของ node ใหม่ ทำการสร้าง node ใหม่ด้วยค่าความสูงที่เกิดจากการสุ่ม แล้วจึงทำการปรับปรุงตัวเชื่อม เราใช้ตัวแปร update เป็นที่เก็บตัวเชื่อมต่าง ๆ ซึ่งเมื่อเราหาตำแหน่งเจอแล้ว update[i] จะมีตัวเชื่อมที่ชี้ไปยัง node ที่อยู่ทางด้านขวาของระดับความสูง i หรือสูงกว่าของ node ทางด้านซ้ายของตำแหน่งที่ต้องนำ node ใหม่เข้า และถ้าหากว่าระดับความสูงของ node ใหม่ที่เราสร้างขึ้นสูงกว่าค่าความสูงของ node เก่าที่อยู่ในตำแหน่งนี้ เราก็ทำการปรับปรุงความสูงของ list พร้อมกับปรับค่าของ update ในส่วนต่างๆ ให้สอดคล้องกับสิ่งที่เกิดขึ้น (ดูรูปที่ 6)



รูปที่ 6. การนำข้อมูลเข้าสู่ Skip List

ในการสร้าง Skip List นั้นเรายอมให้ผู้ใช้สามารถเลือกความสูงและค่าความเป็นไปได้ผ่านทาง constructor ของ SkipList

```
public SkipList(double prob, int maxLevel) {
    this.level = 0;
    this.prob = prob;
    this.maxLevel = maxLevel-1;

    //header node with min value
    header = new SkipNode(this.maxLevel, MIN_VALUE);
    //nil node with max value
    nil = new SkipNode(this.maxLevel, MAX_VALUE);

    //attach nil to header
    for(int i = 0; i <= this.maxLevel; i++)
        header.forward[i] = nil;
}
```

ในการ implement Skip List นั้นเราต้องมี node พิเศษ 2 ตัวคือ header และ nil ซึ่งทำหน้าที่เป็นตัวเริ่มและตัวจบ (ตามลำดับ) ของ Skip List ผู้อ่านจะเห็นว่าใน header เราจะกำหนดค่าให้เป็น MIN__VALUE และใน nil เรากำหนดให้เป็น MAX__VALUE ทั้งนี้ก็เพราะว่าเราใช้ข้อมูลที่เป็น int ดังนั้นค่าที่อยู่ทาง header ต้องมีค่าน้อยกว่าทาง nil (ข้อมูลจะเรียงจากน้อยไปหามาก) ก่อนนำข้อมูลเข้า header และ nil มีหน้าตาดังนี้

รูปที่ 7. node: head และ node: nil ก่อนการนำข้อมูลเข้า

ส่วน code ของการนำข้อมูลเข้าก็มีดังนี้

```
public void insert(int key) {
    //update stores pointers to keys on each level
    SkipNode []update = new SkipNode[maxLevel+1];
    SkipNode x = header;

    //locate a place to insert
    for(int i = level; i >= 0; i--) {
        while(x.forward[i].key < key)
                x = x.forward[i];
        update[i] = x;
    }
    x = x.forward[0]; //reset level to the lowest

    //insert key
    if(x.key == key)
        ; //no duplicates allowed
    else {
        //flip the coin
        int newLevel = randomLevel();
        //new key has more level than this one, update list
        if(newLevel > level) {
                for(int i = level+1; i <= newLevel; i++)
                        update[i] = header;
                //current level seen so far
                level = newLevel;
        }
        //create new node with key and update links
        x = new SkipNode(newLevel, key);
        for(int i = 0; i <= newLevel; i++) {
                x.forward[i] = update[i].forward[i];
                update[i].forward[i] = x;
        }
    }
}
```

ในส่วนของการดึงข้อมูลออกก็คล้ายๆ กับการนำข้อมูลเข้า เราต้องหาข้อมูลที่ต้องการลบออกให้เจอ และเมื่อหาเจอแล้วเราก็ทำการปรับปรุงตัวเชื่อมในระดับต่าง ๆ ของ node ที่อยู่ทั้งทางด้านซ้ายและ node ที่อยู่ทางด้านขวาให้เหมาะสม ซึ่งถ้าบังเอิญว่า node ที่เราดึงออกนั้นเป็น node ที่มีระดับความสูง สูงที่สุดใน list เราก็ต้องลดระดับความสูงของ list ลง ดังที่แสดงให้ดูจาก code นี้

```
public void delete(int key) {
    SkipNode[] update = new SkipNode[maxLevel+1];
    SkipNode x = header;

    //search for key in list
    for(int i = level; i >= 0; i--) {
        while(x.forward[i].key < key)
                x = x.forward[i];
        update[i] = x;
    }
    x = x.forward[0]; // reset level to the lowest

    //found key to delete, rebuild list
    if(x.key == key) {
        for(int i = 0; i <= level; i++) {
                if(update[i].forward[i] == x)
                        update[i].forward[i] = x.forward[i];
        }
        x = null; //destroy this node
        //reset level
        while(level > 0 && header.forward[level].key == MAX_VALUE)
                level--;
    }
}
```

ในส่วนของการค้นหาข้อมูลนั้นเราก็ใช้วิธีการคล้ายๆ กับที่เราได้ใช้ในการค้นหาตำแหน่งที่ต้องใส่ข้อมูลหรือดึงข้อมูลออก

```
public boolean search(int key) {
    SkipNode x = header;

    for(int i = level; i >= 0; i--) {
        while(x.forward[i].key < key)
                x = x.forward[i];
    }
    x = x.forward[0];

    return x.key == key ? true : false;
}
```

ตัวอย่างผลลัพธ์ของการ run โปรแกรมทดสอบ Skip List ด้วยค่าความเป็นไปได้ที่ 0.5 และระดับความสูงสูงสุดที่ 5 จำนวนของข้อมูลที่นำเข้าเท่ากับ 15 ตัวโดยกำหนดให้ 14 ตัวแรกเกิดจากการสุ่ม หลังจากนั้นก็นำ 43 เข้า (และลบออก) ตัวเลขที่อยู่ในวงเล็บหลัง keys: คือจำนวนของ node ที่มีอยู่ในระดับนั้นๆ

```
level[4]keys:(00):
level[3]keys:(03):   -   -   -   -   -   -  40  43  45
level[2]keys:(04):  12   -  15   -   -  35  40  43  45   -   -   -   -  94
level[1]keys:(03):  12  14  15   -   -  35  40  43  45   -  61  75   -  94
level[0]keys:(05):  12  14  15  24  29  35  40  43  45  59  61  75  80  94  97

level[4]keys:(00):
level[3]keys:(02):   -   -   -   -   -   -  40  45
level[2]keys:(04):  12   -  15   -   -  35  40  45   -   -   -   -  94
level[1]keys:(03):  12  14  15   -   -  35  40  45   -  61  75   -  94
level[0]keys:(05):  12  14  15  24  29  35  40  45  59  61  75  80  94  97
```

เนื่องจากเรากำหนดให้ความสูงของแต่ละ node เกิดจากค่าสุ่มดังนั้นก็อาจเป็นไปได้ว่า จะสุ่มได้ค่าเดียวกันทุกครั้ง (ซึ่งเป็นไปค่อนข้างยาก แต่ก็อาจเป็นได้) และถ้าสุ่มได้ค่า 1 ก็จะทำให้ Skip List กลายเป็น Linked List ทันที

ศาสตราจารย์ Pugh ได้แสดงการวิเคราะห์ Skip List ไว้ในบทความที่ท่านได้เขียนไว้อย่างละเอียด [1] โดยคร่าวๆ แล้วท่านได้บอกว่าค่าเฉลี่ยของ running time ของ Skip List เท่ากับ $\log_2 n$ ส่วนในกรณีที่แย่ที่สุดที่อาจเกิดขึ้นค่า running time จะเป็น linear time ซึ่งเป็นกรณีที่เกิดขึ้นไม่ง่ายนัก ผู้อ่านอย่าลืมว่าเราใช้ probability ในการหาค่าระดับความสูงดังนั้นโอกาสที่ความสูงของแต่ละ node จะมีค่าเป็น 1 ทุกตัวนั้นแทบเป็นไปไม่ได้เลย (ผู้อ่านที่สนใจก็สามารถอ่านได้จากบทความดังกล่าว)

สรุป

Skip List เป็นโครงสร้างอีกชนิดหนึ่งที่มีการจัดเก็บข้อมูลในรูปแบบของการจัดเรียง เป็นโครงสร้างที่พัฒนามาจาก Linked List โดยศาสตราจารย์ William Pugh ด้วยการกำหนดให้แต่ละ node ที่ประกอบกันเป็น list มีความสูงที่เกิดจากการสุ่มในเวลาที่นำเข้า ซึ่งทำให้การค้นหา การนำข้อมูลเข้า และการดึงข้อมูลออกใช้เวลาที่สั้นลงเมื่อเปรียบเทียบกับ Linked List ทั่วๆ ไป เป็นโครงสร้างที่มี running time อยู่ในกลุ่มของ self balanced binary search tree [2] กล่าวคือเวลาที่ใช้ไปมีค่าเป็น O(log n)

เอกสารอ้างอิง

Pugh, William. (1990). **Skip Lists: A Probabilistic Alternative to Balanced Trees.**

Retrieved November 2007, from: ftp://ftp.cs.umd.edu /pub/skipLists/ skiplists.pdf

เนรมิต ชุมสาย ณ อยุธยา. (2550) **เรียนรู้โครงสร้างข้อมูลด้วย Java**. (ฉบับปรับปรุง). กรุงเทพฯ: ซีเอ็ดยูเคชั่น.

ข้อมูลอื่นๆ

1. Skip List in Action จาก http://iamwww.unibe.ch/~wenger/DA/SkipList/ เป็นโปรแกรมที่แสดงถึงขั้นตอนการทำงานของ Skip List ที่เขียนโดย Thomas Wenger จากมหาวิทยาลัย Berne, Switzerland
2. Source code จากหนังสือในเอกสารอ้างอิง [2] ที่ http://sci.feu.ac.th/faa/dsa/dsa.html