บทที่ 11

Pointer and Dynamic Structure

วัตถุประสงค์

1. เพื่อให้นักศึกษารู้และเข้าใจถึงโครงสร้างโปรแกรมที่มีการจองเนื้อที่แบบจลย์
2. เพื่อให้นักศึกษาทราบและเข้าใจตัวแปรชนิดพอยเตอร์
3. เพื่อให้นักศึกษาสามารถประกาศ กำหนด และจัดการกับโปรแกรมที่มีโครงสร้างที่ใช้ตัวแปรชนิดพอยเตอร์ได้
4. ประยุกต์ใช้งานกับโครงสร้างโปรแกรมที่มีการจองเนื้อที่แบบจลย์ได้

ในบทที่แล้ว เราเรียนรู้การสร้าง Template cass ในการพัฒนาโครงสร้างโปรแกรมโดยใช้โครงสร้างข้อมูลชนิดต่างๆไม่ว่าเป็นอาเรย์ชนิดแถว 1 มิติ หรือสแตก หรือ คิว ในบทนี้เราจะมาเรียนรู้การพัฒนาโปรแกรมเพื่อใช้งานอีกแนวทางหนึ่งที่ ผู้ใช้ไม่ต้องมีการจองเนื้อที่ไว้ก่อนเป็นผืนใหญ่เพราะถ้าเราใช้งานไม่หมดจะทำให้สูญเสียเนื้อที่ใช้งานโดยเปล่าประโยชน์ ทำให้โปรแกรมที่พัฒนาขึ้นมีการใช้ทรัพยากรได้อย่างไม่มีประสิทธิภาพ โปรแกรมที่พัฒนาไม่มีคุณภาพเท่าที่ควร การเรียนรู้ข้อมูลชนิดพอยเตอร์จะแก้ปัญหาในการทำงานในบางลักษณะที่ดีมาก โดยเหมาะสมสำหรับระบบที่ต้องการประหยัดเนื้อที่ใช้งานในหน่วยความจำ และเหมาะสำหรับการนำข้อมูลเข้าและนำข้อมูลออกโดยไม่ต้องมีการปรับเปลี่ยนหรือเคลื่อนย้ายข้อมูล ทำให้การปฏิบัติงานมีความเร็วและคล่องตัวมากขึ้น

11.1 พอยเตอร์ (pointer)

พอยเตอร์(pointer) เป็นชนิดของข้อมูลหนึ่งที่เก็บตำแหน่งที่อยู่ของข้อมูล การประกาศชนิดของข้อมูลที่ได้กล่าวๆมาตั้งแต่ต้นนั้นเป็นการประกาศตัวแปรเพื่อจองเนื้อที่สำหรับใช้เก็บข้อมูล เช่น

```
int  A, B;
float  c;
```

โดยการจัดสรรเนื้อที่นี้จะมีขนาดแตกต่างกันไปขึ้นอยู่กับชนิดของข้อมูลที่เก็บถ้าเรากำหนดให้

```
A = 3;
```

จะเป็นการนำเลขจำนวนเต็ม 3 ไปเก็บในตัวแปร A ณ ตำแหน่งที่ระบบจัดสรรให้ โดยเราไม่ทราบว่าอยู่ที่ใดตำแหน่งไหนในหน่วยความจำ แต่ในการแก้ปัญหาบางลักษณะนั้นมีการเคลื่อนย้ายข้อมูลโดยใช้เวลามากทำให้โปรแกรมมีประสิทธิภาพไม่ดีเท่าที่ควร ดังนั้นตัวแปรชนิดพอยเตอร์จึงเป็นทางเลือกอีกหนทางหนึ่งที่ถูกนำมาใช้ในการแก้ปัญหาโดยขจัดการเคลื่อนย้ายข้อมูลให้น้อยที่สุดแต่สามารถทำงานตามที่ต้องการได้ เราใช้วิธีการอ้างถึงตำแหน่งที่อยู่แทน

รูปแบบการประกาศตัวแปรชนิดพอยเตอร์

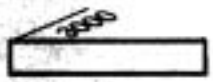
```
type * Variable;
```

```
float *B;
```

เป็นการประกาศตัวแปรชนิดพอยเตอร์ชื่อ B ให้ตัวแปลภาษาได้รับรู้ โดยปีนตัวแปรนี้จะเก็บตำแหน่งที่อยู่ของข้อมูลที่เป็นเลขทศนิยม โดยยังไม่จัดสรรเนื้อที่ในหน่วยความจำแต่อย่างใด

```
B = 3000;
```

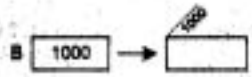
คำสั่งนี้เป็นการนำค่า 3000 ไปเก็บในตัวแปรพอยเตอร์ชื่อ B โดยค่าที่เก็บเป็นตำแหน่งที่อยู่โดยยังไม่มีการจัดสรรเนื้อที่เพื่อเก็บเลขทศนิยมใดๆ



การจองเนื้อที่ให้กับตัวแปรพอยเตอร์ เป็นการจองแบบจลย์(Dynamic) โดยเราจะจองเมื่อโปรแกรมทำงานหรือต้องการเก็บข้อมูลจริงๆ
คำสั่งมีรูปแบบดังนี้

```
new   type;
```

ดังนั้นถ้าเรากำหนดคำสั่งดังนี้



B = new float; เป็นการจองเนื้อที่ใหม่เพื่อเก็บเลขทศนิยม 1 จำนวน โดยให้ตัวแปรพอยเตอร์ B ชี้ที่เนื้อที่ใหม่นี้ สมมุติว่าระบบจัดสรรเนื้อที่ว่างให้ ณ ตำแหน่งที่ 1000 ตัวแปรพอยเตอร์ B จะเก็บค่าตำแหน่งที่อยู่ 1000 ไว้โดยอัตโนมัติ ถ้าเราต้องการปฏิบัติการกับตัวแปรพอยเตอร์ เช่นการนำค่าไปจัดเก็บ หรือการนำข้อมูลที่จัดเก็บในตำแหน่งที่อยู่ที่ต้องการมาใช้งาน เราใช้สัญลักษณ์ * หรือ Asterisk หรือที่นิยมว่า indirection operator ใช้อ้างถึงค่าของข้อมูลที่ตัวแปรพอยเตอร์ ชี้อยู่จาก

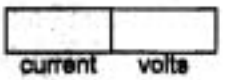
```
float *B;
B = new float;
*B = 15.5;
```

ซึ่งเป็นการนำค่าเลขทศนิยม 15.5 ไปจัดเก็บ ณ ตำแหน่งที่ตัวแปรพอยเตอร์ B ชี้อยู่

11.2 Pointer to Structs

ในการแก้ปัญหาโปรแกรมโดยใช้ตัวแปรพอยเตอร์นั้นส่วนมากจะใช้กับโครงสร้างข้อมูลชนิด struct เพราะการปฏิบัติงานจะกระทำกับรายการข้อมูลมากกว่า 1 ชนิดที่มีความสัมพันธ์กัน เช่น ระเบียนข้อมูลชื่อ electric ซึ่งประกอบด้วยฟิลด์ current และ ฟิลด์ volts โดยมีการประกาศโครงสร้างตามรูปแบบของภาษาดังนี้

```
struct electric
{
    string current;
    int volts;
};
```



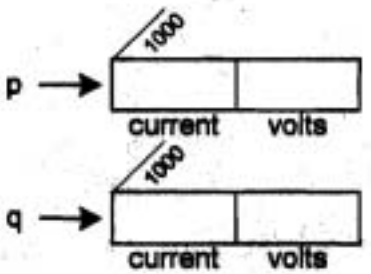
เราสามารถกำหนดตัวแปรพอยเตอร์ชื่อ p และ q ให้เก็บตำแหน่งที่อยู่ของโครงสร้างข้อมูลชนิด electric ใดๆได้ดังนี้

```
electric  *p, *q;
```

การจัดสรรเนื้อที่สำหรับเก็บข้อมูลนั้นเราต้องใช้คำสั่ง new ดังนี้

```
p = new electric;
q = new electric;
```

การทำงานของคำสั่งจะจัดสรรเนื้อที่ว่างเป็นโครงสร้างของ electric โดยให้ตัวแปรพอยเตอร์ p และ q ชี้อยู่ โดยในที่นี้สมมุติว่า p ชี้ที่ตำแหน่ง 1000 ส่วน q ชี้ที่ตำแหน่ง 10000



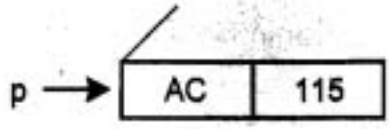
การนำค่าไปเก็บในโครงสร้างที่จัดสรรไว้เราสามารถใช้คำสั่งดังนี้

```
(*p).current = "AC";
(*p).volts = 115;
```

หรือ

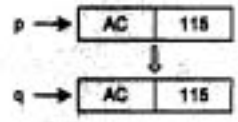
```
p -> current = "AC";
p -> volts = 115;
```

โดยผลของการทำงานจะได้ผลลัพธ์เหมือนกัน โดยนำค่า "AC"ไปจัดเก็บในฟิลด์ current ณ ตำแหน่งที่ตัวแปรพอยเตอร์ p ชี้อยู่ และนำค่า 115 ไปจัดเก็บในฟิลด์ volts เช่นเดียวกัน ดังรูป



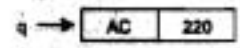
ต้องการคัดลอกข้อมูลทั้งหมดจากที่ตัวแปรพอยเตอร์ p ชี้อยู่ไปให้กับโครงสร้างข้อมูลชนิดเดียวกันที่ตัวแปรพอยเตอร์ q ชี้อยู่ เราใช้คำสั่ง

```
*q = *p;
```



ถ้าต้องการปรับเปลี่ยนข้อมูลเราสามารถอ้างถึงได้ดังนี้

```
q -> volts = 220;
```



การจัดสรรเนื้อที่เพื่อใช้งานนั้นโดยใช้คำสั่ง new นั้นระบบจะถือว่าเนื้อที่นี้ถูกใช้งานอยู่ จะนำไปใช้ประโยชน์ในเรื่องอื่นๆไม่ได้ ถ้าในระบบที่มีหน่วยความจำน้อยต้องการนำเนื้อที่นี้ไปใช้ประโยชน์อย่างอื่นเราต้อง บอกให้ตัวแปรภาษาทราบ โดยปล่อยเนื้อที่ส่วนนี้ไป ในภาษา C++ มีคำสั่งที่ปลดปล่อยเนื้อที่ให้เป็นอิสระจากคำสั่งดังต่อไปนี้

การลบหรือปล่อยเนื้อที่ของตัวแปรพอยเตอร์

รูปแบบ delete Variable;

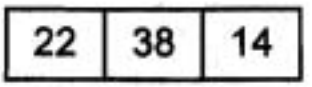
ตัวอย่าง

delete q; จาก 

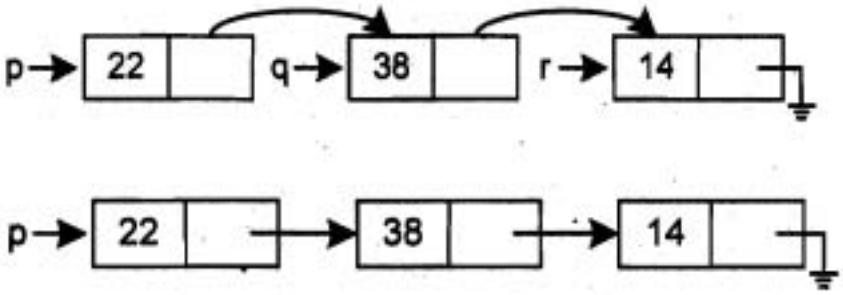
11.3 Singly linked list

เป็นการนำข้อมูลชนิดพอยเตอร์มาใช้สำหรับเชื่อมต่อข้อมูลเป็นลิสต์เชื่อมโยง โดยในหัวข้อนี้เป็นการเชื่อมโยงทางเดียว เพื่อให้เข้าใจได้ดีขึ้นถ้าเราต้องการเก็บข้อมูล 22 38 14 ในหน่วยความจำหลักเราสามารถนำข้อมูลนี้ไปจัดเก็บได้หลายลักษณะ

1. จัดเก็บในลักษณะของอาเรย์ 1 มิติ ข้อมูลจะถูกนำมาเก็บต่อๆกันไปดังรูป

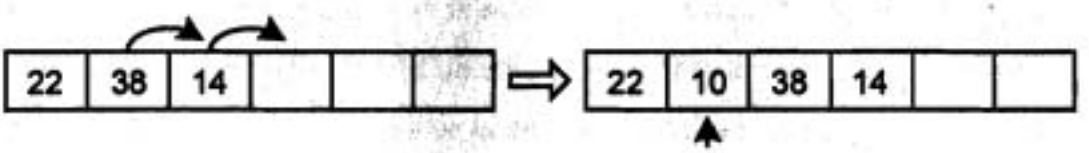


2. จัดเก็บในลักษณะของลิสต์เชื่อมโยงทางเดียว เป็นการนำตัวแปรพอยเตอร์มาเก็บตำแหน่งของข้อมูลตัวถัดไป ดังรูป

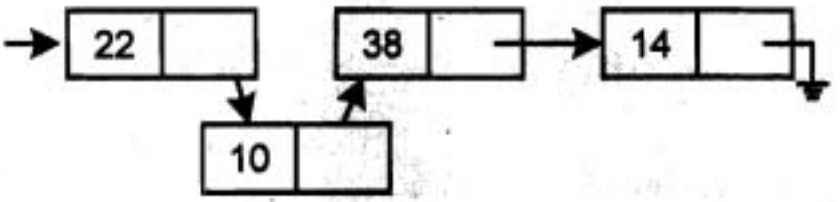


การจัดเก็บใน 2 ลักษณะนี้จะแตกต่างกันในเรื่องของการปฏิบัติการกับข้อมูลไม่ว่าเป็นการจัดเก็บ การแทรกข้อมูล การลบข้อมูล การเข้าถึงข้อมูล ซึ่งการทำงานต่างๆเหล่านี้ส่งผลต่อประสิทธิภาพในการทำงานของโปรแกรมทั้งสิ้นดังนั้นเรามาเรียนรู้ถึงการปฏิบัติการต่างๆเหล่านั้นกันดีกว่า สำหรับโครงสร้างข้อมูลชนิดอาเรย์นั้นการเข้าถึงข้อมูลในอาเรย์จะเป็นลักษณะของเรียงลำดับโดยเริ่มจากข้อมูลในช่องแรกเรื่อยไปจนถึงข้อมูลในช่องสุดท้ายในการเพิ่มข้อมูลแทรก

ระหว่างต้องเสียเวลาในการเลื่อนข้อมูลตัวหลังๆให้เลื่อนไปยังช่องถัดไป หรือการลบก็เช่นเดียวกัน จะต้องเลื่อนข้อมูลที่อยู่ในช่องหลังตัวที่ต้องการลบโดยเลื่อนให้มาแทนที่ในช่องแรก 1 ช่อง ซึ่งถ้าข้อมูลที่เก็บมีเป็นจำนวนมากจะทำให้เสียเวลาในการทำงานมาก



แต่ถ้าเป็นโครงสร้างลิสต์แบบเชื่อมโยงทางเดียวนั้น จะขจัดปัญหาในการเคลื่อนย้ายข้อมูล โดยเราเพียงแต่เปลี่ยนตำแหน่งที่อยู่ของข้อมูลให้เก็บข้อมูลตัวถัดไปให้ถูกต้องเท่านั้น เราก็สามารถที่จะเพิ่มหรือลบข้อมูลได้อย่างง่ายดาย

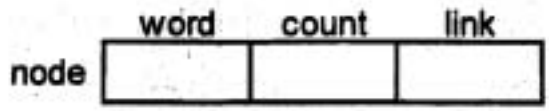


สิ่งที่แตกต่างกันระหว่างโครงสร้างข้อมูลที่เป็นอาเรย์ 1 มิติกับลิสต์เชื่อมโยงทางเดียวที่เห็นชัดที่สุดคือการจัดเก็บข้อมูลของลิสต์เชื่อมโยงทางเดียวที่ต้องมีการเก็บตำแหน่งที่อยู่ของข้อมูลตัวถัดไป ควบคู่กับข้อมูลที่ต้องการจัดเก็บจริงๆ เราจะรวมเรียกข้อมูลทั้งหมดนี้ว่าโหนด (node)

ลักษณะของ node ข้อมูล จึงเป็นโครงสร้างชนิดเรคอร์ดที่ประกอบด้วยฟิลด์ใหญ่ๆ 2 ฟิลด์ด้วยกันคือ

1. Information field เป็นรายการข้อมูลจริงที่เราต้องการจัดเก็บ
2. link field เป็นรายการข้อมูลที่เป็นตัวแปรชนิดพอยเตอร์ ที่เก็บตำแหน่งของโหนดตัวถัดไป ในกรณีที่ไม่มีข้อมูลตัวถัดไปเราจะชี้ไปที่ null ซึ่งเป็นจุดจบของลิสต์ข้อมูล

สมมุติว่าเราต้องการสร้างโหนดข้อมูลประกอบด้วย 3 ฟิลด์คือฟิลด์ word , ฟิลด์ count และ ฟิลด์ link ดังรูป



ก่อนอื่นต้องการกำหนดโครงสร้างของโหนดเสียก่อนในที่นี้ information field มีด้วยกัน 2 ฟิลด์คือ word และ count ส่วน ฟิลด์ link เป็นฟิลด์ที่เก็บตำแหน่งที่อยู่ของโหนดถัดไป เราสามารถกำหนดตามรูปแบบของโครงสร้างข้อมูลชนิด struct ได้ดังนี้

```
struct node
{
        string word;
        int count;
        node *link;
};
```

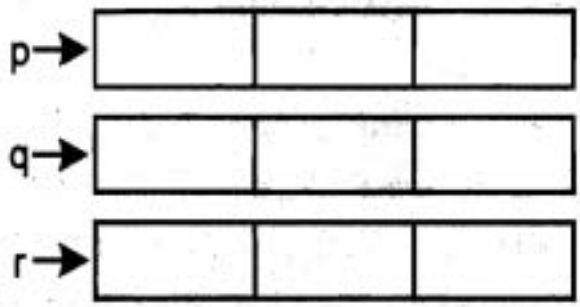
การกำหนดตัวแปรชนิดพอยเตอร์ที่เก็บตำแหน่งที่อยู่ของโหนดได้ดังนี้

```
node *p, *q, *r;
```

เราสามารถจองเนื้อที่ว่าง 1 โหนดเพื่อใช้งานได้ดังนี้

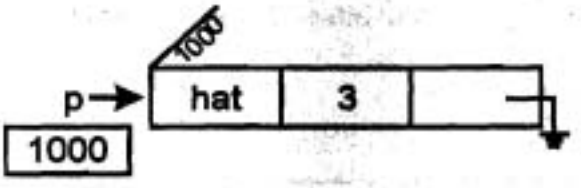
```
p = new node;
q = new node;
r = new node;
```

ในที่นี้ระบบจะจองเนื้อที่ว่างถึง 3 โหนดด้วยกันโดยให้ตัวแปรพอยเตอร์ p ,q ,r ชี้อยู่ดังรูป

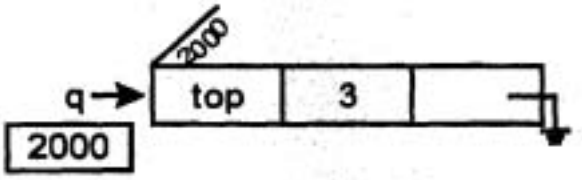


ในการนำข้อมูลไปใส่ในโครงสร้างที่เราจัดสรรเนื้อที่สามารถกระทำได้ดังนี้

```
p -> word = "hat";
p -> count = 3;
p -> link = null;
```

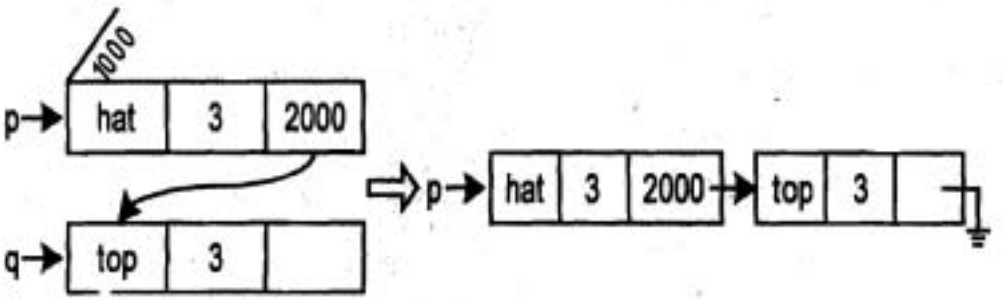



```
q -> word = "top";
q -> count = 3;
q -> link = null,
```



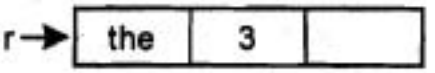
เรานำโหนดที่ตัวแปรพอยเตอร์ p ชี้อยู่การเชื่อมโยงกับโหนดที่ตัวแปรพอยเตอร์ q ชี้ได้ดังนี้

```
p -> link = q;
```

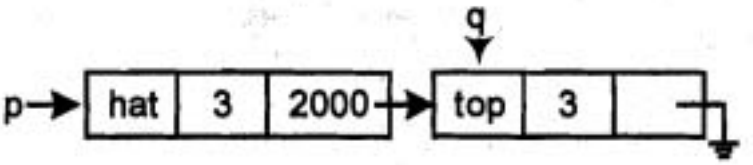


ถ้าเราต้องการแทรกโหนดใหม่โดยสร้างโหนดใหม่ขึ้นมา และให้ตัวแปรพอยเตอร์ r ชี้อยู่ดังรูป

```
r -> word = "the";
r -> count = 3;
```

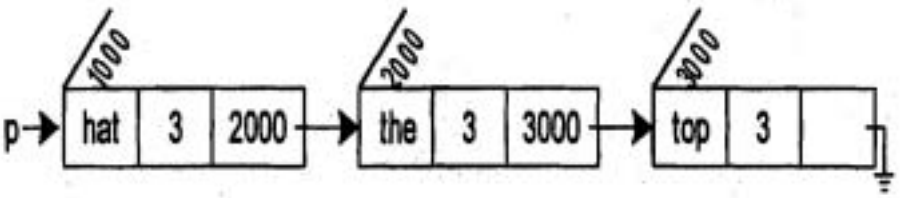


สมมุติว่าลิสต์เชื่อมโยงของเราเป็นดังรูปข้างล่างนี้ โดยตัวแปรพอยเตอร์ p ชี้อยู่ที่ข้อมูลโหนดแรก



เรานำข้อมูลโหนดที่ตัวแปรพอยเตอร์ r ชี้อยู่แทรกระหว่างโหนดข้อมูลที่ p และ q เราสามารถกระทำได้โดยใช้คำสั่งดังนี้

```
p -> link = r;
r -> link = q;
```

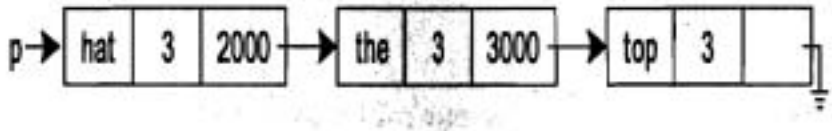


จากรูป

list reference	ผลลัพธ์
p->word	hat
p->link	link field of first node
p->link->word	the
p->link->link	link field of the second node
p->link->link->count	3

ในการแทรกโหนดใหม่ในลิสต์เชื่อมโยงนั้นสามารถกระทำได้ทุกๆส่วนถ้าเราต้องการการแทรกโหนดใหม่เป็นโหนดแรกของลิสต์นั้นสามารถกระทำได้ดังนี้

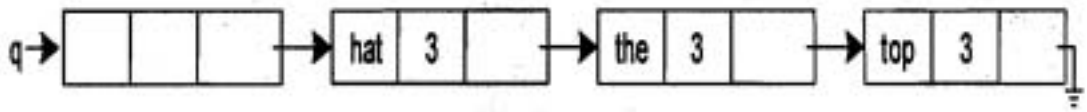
สมมุติว่าข้อมูลในลิสต์เชื่อมโยงมีข้อมูลดังรูปข้างล่างนี้



จากรูปเราให้ตัวแปรพอยเตอร์ p ชี้อยู่ที่โหนดแรก เราจะสร้างโหนดข้อมูลใหม่ขึ้นมาโดยให้ตัวแปรพอยเตอร์ q ชี้อยู่และต้องการแทรกโหนดใหม่นี้ให้เป็นโหนดแรกของลิสต์จากคำสั่งดังต่อไปนี้

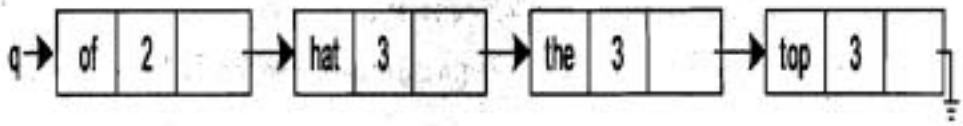
```
q = new node;
q -> link = p;
```

ผลจากการทำงานจะเป็นดังนี้



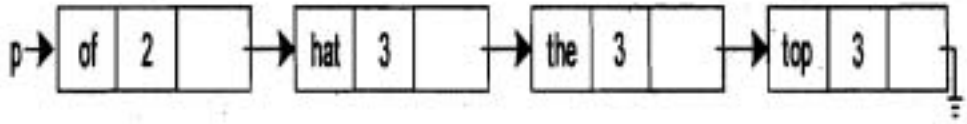
จะเห็นได้ว่าข้อมูลในโหนดแรกที่ตัวแปรพอยเตอร์ q ชี้อยู่ยังไม่มีข้อมูลอยู่เลย เราสามารถให้ค่าข้อมูลในโหนดแรกของลิสต์ได้ดังนี้

```
q -> word = "of";
q -> count = 2;
```

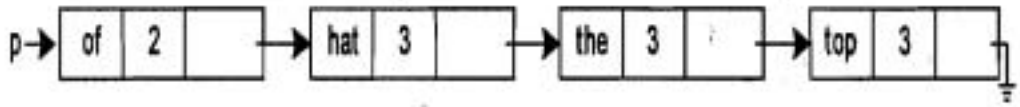


เราสามารถเปลี่ยนตัวแปรพอยเตอร์ให้ p ชี้อยู่ที่โหนดแรกของลิสต์ได้จากคำสั่ง

```
p = q;
```



ในกรณีที่เราต้องการแทรกโหนดใหม่เป็นข้อมูลในโหนดสุดท้ายของลิสต์ โดยพิจารณาจากลิสต์เชื่อมโยงที่มีตัวแปรพอยเตอร์ p ชี้อยู่



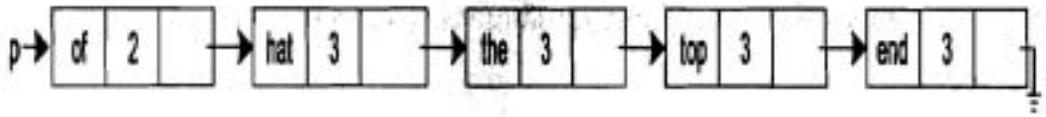
เรากำหนดโหนดใหม่ขึ้นมา 1 โหนดโดยให้ตัวแปรพอยเตอร์ q ชี้อยู่ และกำหนดค่าของข้อมูลให้แก่โหนดใหม่นี้ดังนี้

```
q = new node;
q -> word = "end";
q -> count = 3;
q -> link = null;
```

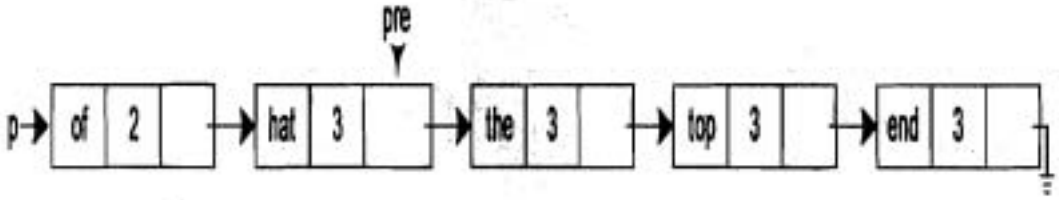
การจะแทรกโหนดใหม่ ณ ตำแหน่งใดนั้น ต้องหาตำแหน่งที่อยู่ในการแทรกให้ได้เสียก่อน ซึ่งเราไม่ทราบว่าข้อมูลตัวสุดท้าย ณ ปัจจุบันอยู่ที่ใด แต่ทราบแต่เพียงว่าโหนดสุดท้ายฟิลด์ของ link ต้องมีค่าเท่ากับ null เราจะใช้คำสั่งวนรอบในที่นี้คือคำสั่ง while เพื่อทำการค้นหาโหนดสุดท้ายโดยการเริ่มจากข้อมูลโหนดแรกเพราะเราทราบว่าตัวแปรพอยเตอร์ p ชี้อยู่ที่โหนดแรก ต่อจากนั้นจะทำการเลื่อนข้อมูลจากโหนดแรกไปยังตำแหน่งต่อๆไปจนพบโหนดสุดท้ายในลิสต์เชื่อมโยงนี้ ในที่นี้ให้ตัวแปรพอยเตอร์ r เป็นตัวแปรที่เก็บตำแหน่งที่อยู่ของโหนดโดยจะมีค่าเปลี่ยนแปลงไปเมื่อมีการเลื่อนโหนดของข้อมูลไปยังตัวถัดๆไป ส่วนตำแหน่งที่อยู่ของตัวแปรพอยเตอร์ p เราจะให้คงเดิมเพราะต้องเก็บตำแหน่งที่อยู่ของโหนดแรกของลิสต์ไว้เสมอ ห้ามเปลี่ยนแปลง

```
r = p;
while (r -> link != null)     //จะตรวจสอบจนกว่า r -> link = null
        r = r -> link;        //เลื่อนไปชี้ที่ node ถัดไป
r -> link = q;                //เชื่อมกับ q เป็น node สุดท้าย
```

ผลการทำงานจากคำสั่งข้างต้นนั้น การหลุดออกจากลูปคำสั่ง while นั้นก็ต่อเมื่อตัวแปรพอยเตอร์ r ชี้อยู่ที่โหนดสุดท้ายของลิสต์เชื่อมโยงนี้ ซึ่งจะมีข้อมูลกี่โหนดก็ตาม หลังจากนั้นเรานำโหนดสุดท้ายไปเชื่อมโยงกับโหนดใหม่ ผลรองลิสต์เชื่อมโยงนี้เป็นดังนี้



สำหรับการลบโหนดข้อมูลที่ไม่ต้องการนั้น สามารถกระทำได้หลายจุดเช่นเดียวกันไม่ว่าจะลบข้อมูลเป็นโหนดแรก หรือเป็นโหนดระหว่างลิสต์เชื่อมโยง หรือลบเป็นโหนดสุดท้าย ซึ่งการลบโหนดจะทำให้เกิดการเปลี่ยนแปลงกับตำแหน่งที่อยู่ของข้อมูลที่เกี่ยวข้องทั้งสิ้น ดังนี้ สมมุติว่าลิสต์เชื่อมโยงในปัจจุบันที่ตัวแปรพอยเตอร์ p ชี้อยู่เป็นดังนี้



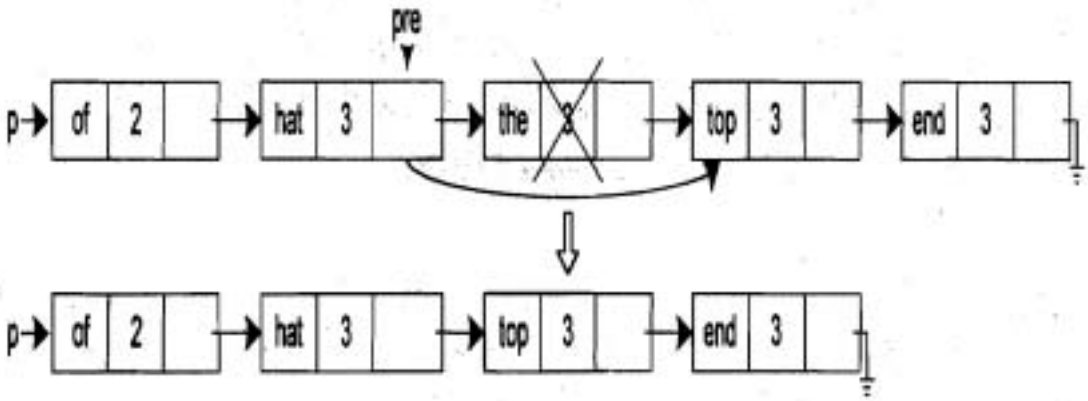
สมมุติว่าเราต้องการลบโหนดข้อมูลที่มีคำว่า the เราจะต้องกระทำอย่างไร วิธีการนั้นเราต้องหาให้ได้ว่าโหนดก่อนหน้าโหนดที่ต้องการลบอยู่ที่ไหนเพื่อเปลี่ยนให้ฟิลด์ของโหนดก่อนหน้าไปชี้หรือเก็บตำแหน่งที่อยู่ของโหนดถัดไปของโหนดที่ต้องการลบแทน คือปัจจุบันชี้ที่โหนดข้อมูลที่มีคำว่า the ก็ไปชี้ที่โหนดข้อมูลที่มีคำว่า top แทนนั่นเอง ในการค้นหาตำแหน่งที่อยู่ของโหนดก่อนหน้าที่ต้องการลบนั้น เราต้องกำหนดตัวแปรพอยเตอร์ขึ้นมาเพื่อทำหน้าที่นี้ในที่นี้ชื่อว่า pre และมีตัวแปรพอยเตอร์อีกตัวหนึ่งชื่อ r เป็นตัวแปรที่เลื่อนตำแหน่งข้อมูลไปยังตำแหน่งข้อมูลที่ต้องการลบดังคำสั่งต่อไปนี้

```
pre = null;
r = p;
while (r -> word != "the")
{
        pre = r;
```

```
        r = r -> link;
}
```

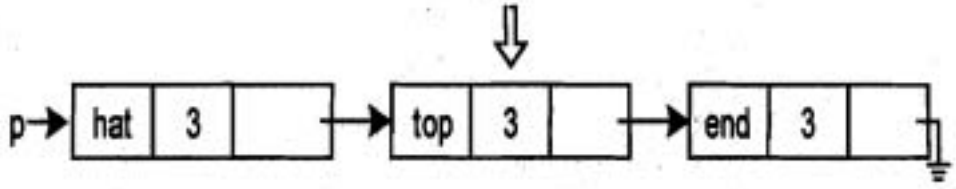
ผลจากการทำงานของคำสั่งนี้นั้นจะหยุดการทำงานวนรอบก็ต่อเมื่อตัวแปรพอยเตอร์ r ชี้อยู่ที่โหนดข้อมูลที่มีคำว่า the และส่งผลให้ตัวแปรพอยเตอร์ pre ชี้อยู่ที่โหนดก่อนหน้า เราสามารถลบโหนดข้อมูลที่มีคำว่า the ได้โดยปรับเปลี่ยนค่าของฟิลด์ link ของโหนดที่ตัวแปรพอยเตอร์ pre ชี้อยู่ได้ดังนี้

```
pre -> link = r -> link;
delete r;
```

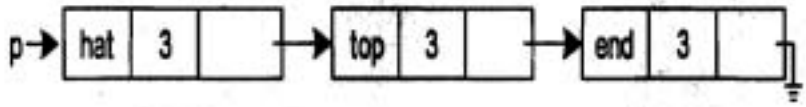


เมื่อมีการเปลี่ยนตำแหน่งที่อยู่ตัวถัดไปของโหนดข้อมูลที่ตัวแปรพอยเตอร์ pre ชี้อยู่แล้ว เราควรทำการลบหรือคืนเนื้อที่โหนดข้อมูลที่มีคำว่า the ให้แก่หน่วยความจำส่วนกลางที่จะนำไปใช้ประโยชน์ในด้านอื่นๆต่อไปโดยใช้คำสั่ง delete ด้วย สำหรับการลบโหนดข้อมูลเป็นโหนดแรกนั้นเราสามารถทราบได้ถ้า pre มีค่าเท่ากับ null ต่อจากนั้นทำการเลื่อนให้ตัวแปรพอยเตอร์ไปชี้ที่โหนดถัดไปแทน ดังคำสั่งต่อไปนี้

```
if (pre = = null)
{
        p = p -> link;
        delete r;
}
```



สำหรับการท่องโหนด หรือการเข้าถึงข้อมูลจากโหนดแรกจนถึงโหนดสุดท้ายอย่างเป็นลำดับ เราสามารถกระทำได้ไม่ยาก ถ้าเราทราบตำแหน่งที่อยู่ของโหนดแรกของข้อมูล เพราะเราสามารถเลื่อนไปยังข้อมูลของโหนดถัดไปได้ และก็ทราบด้วยว่าโหนดข้อมูลตัวสุดท้ายนั้นมีฟิลด์ link เท่ากับ null นั่นเอง



ในที่นี้เราจะออกแบบเป็นฟังก์ชันชื่อ printlist โดยมีการส่งผ่านตำแหน่งที่อยู่ของข้อมูลโหนดแรกไปให้แก่ฟังก์ชันโดยมีการเรียกใช้ดังนี้

```
printlist (p);
```

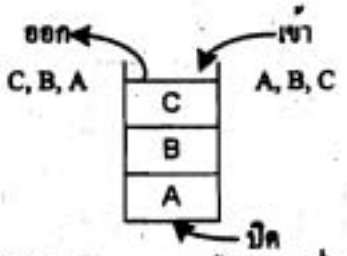
สำหรับ formal parameter ที่ใช้สำหรับรับค่านั้นต้องเป็นตัวแปรชนิดพอยเตอร์ด้วยดังนี้

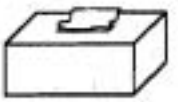
```
//file: printlist.cpp
//display the list pointed to by head
//pre: head pointed to list whose last node has a pointer  member of null
//post: the word and cout members of each list node
// are displayed and the last value of head is null

void printlist  (listnode *head)  //IN:pointer to list to be printed
{
        while (head != null)
                {
                        //no prior value of head was null.
                        cout  << head->word << " " << head-link<<endl;
                        head = head->link;      //advance to next list node.
                }
}   //end printlist
```

11.4 Stacked as linked list

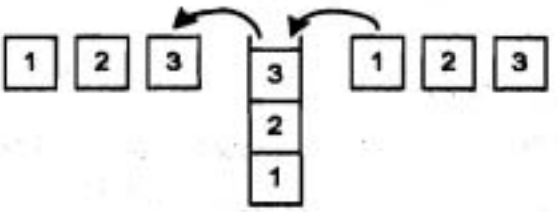
สแต็ก(stack) เป็นโครงสร้างข้อมูลที่เป็นลักษณะ LI-FO (last in - first out) หมายถึง ข้อมูลนำเข้าตัวสุดท้ายจะได้รับการบริการก่อน โดยข้อมูลที่มีการนำเข้า และนำออกได้เพียงทางเดียว



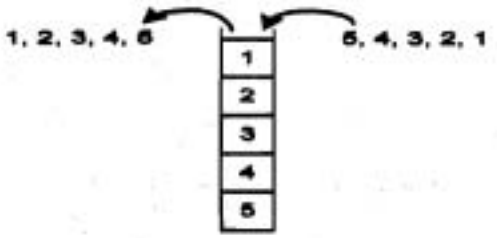


กล่องกระดาษทิชชู

ตัวอย่าง การใส่ถ่านในกระบอกไฟฉาย ซึ่งกระบอกไฟฉายจะมีทางให้เราใส่ถ่านเพียงด้านเดียว อีกด้านหนึ่งเป็นปลายปิด ถ่านก้อนแรกจะถูกใสเข้าไปโดยอยู่ล่างสุด ถ่านก้อนต่อๆไปจะถูกใส่เข้าไป จะเห็นได้ว่าก้อนสุดท้ายจะอยู่บนสุดของกระบอกไฟฉาย เมื่อนำก้อนถ่านที่หมดอายุออกมา ตัวบนสุดจะถูกนำออกก่อนเป็นลำดับแรก และถ่านก้อนล่างสุดจะถูกนำออกเป็นลำดับสุดท้าย



ตัวอย่าง การเขียนโปรแกรมแบบเรียกตัวเอง เช่นการหาค่าของแฟคตอเรียลในที่นี้สมมุติว่าfac(5) เมื่อมีการเรียกใช้จะมีการเรียกการทำงานในลำดับที่แล้วทำงาน โดยจะเรียกลำดับที่แล้วเพื่อทำงานวนรอบไปเรื่อยๆ fac(4) , fac(3) , fac(2) , ... จนกระทั่งถึงเงื่อนไขในการหยุดการเรียกตัวเอง fac(1) จะเห็นได้ว่าฟังก์ชั่นใดที่เรียกเป็นลำดับสุดท้ายจะมีการ return ค่ากลับก่อน ฟังก์ชันที่เรียกเป็นลำดับแรก จะ return ค่ากลับเป็นลำดับสุดท้าย



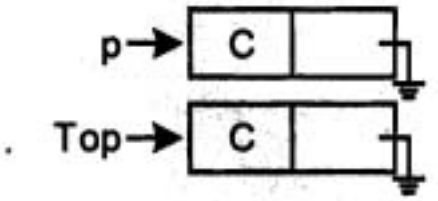
ตัวอย่าง การนำสแต็กมาใช้ในการเก็บนิพจน์คณิตศาสตร์เช่นต้องการเก็บ C + 2 + 3 – 5 โดยต้องการจัดเก็บในลักษณะของลิสต์เชื่อมโยง เราต้องมีการกำหนดตัวแปรพอยเตอร์ชื่อ Top ให้ชี้ที่โหนดข้อมูลแรก ซึ่งในกรณีที่ลิสต์ว่างหรือไม่มีข้อมูล Top มีค่าเท่ากับ null การนำสมาชิกเข้าในลิสต์เชื่อมโยงจะกระทำที่ปลายด้านใดด้านหนึ่งเราเรียกการปฏิบัติการนี้ว่า push และการนำสมาชิกออกจากลิสต์จะกระทำที่ปลายด้านเดียวกันเราเรียกการปฏิบัติการนี้ว่า pop โดยตัวแปรพอยเตอร์ Top จะชี้ที่ตำแหน่งโหนดข้อมูลที่กระทำ

จากตัวอย่างนี้มีการนำนิพจน์ทางคณิตศาตร์มาเก็บในโครงสร้างของลิสต์สแตกก่อนอื่น

```
Top = null;
```

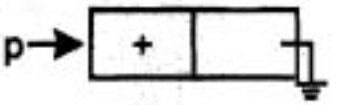
นำข้อมูลตัวแรกใส่เข้าลิสต์สแตก โดยสร้างโหนดข้อมูลใหม่ให้ตัวแปรพอยเตอร์ p ชี้ที่โหนดใหม่นี้

```
push ('C');
```



ต่อจากนั้นให้ตัวแปรพอยเตอร์ Top ชี้ที่ข้อมูลโหนดใหม่นี้ สำหรับข้อมูลตัวที่เหลือเราจะทำการ push ตามลำดับดังนี้

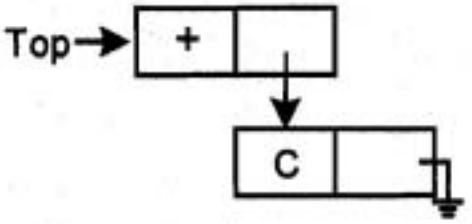
```
push ('+');
```



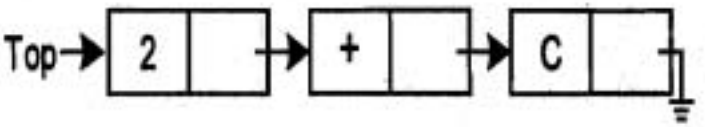
ทำนองเดียวกันต้องมีการสร้างโหนดใหม่โดยให้ตัวแปรพอยเตอร์ p ชี้อยู่นำข้อมูลที่ต้องการให้ค่าแก่โหนดข้อมูลนี้ ดังรูป ต่อจากนั้นให้ฟิลด์ link ของโหนดใหม่เก็บตำแหน่งที่อยู่ของโหนดที่ Top ชี้อยู่ ซึ่งการ push นี้เปรียบได้กับการแทรกข้อมูลใหม่เป็นโหนดแรกของลิสต์นั่นเอง

```
p -> link = Top;
Top = p;
```

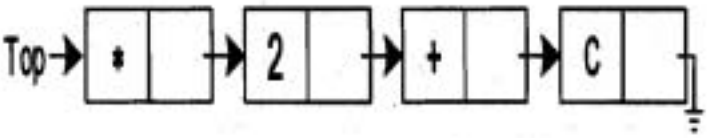
ต่อจากนั้นเปลี่ยนให้ตัวแปรพอยเตอร์ Top ไปชี้ที่โหนดใหม่ซึ่งทำให้ Top ชี้ที่ข้อมูลโหนดแรกของลิสต์นั่นเองดังรูป



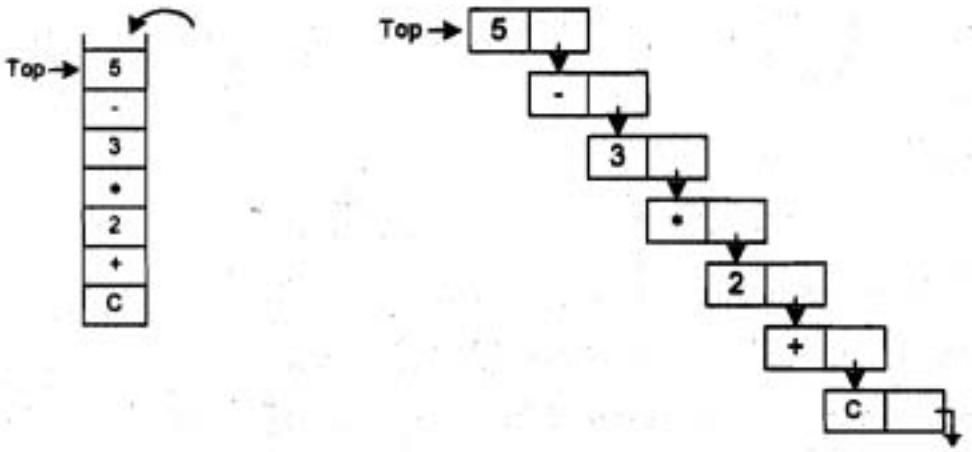
push ('2');



push ('*');

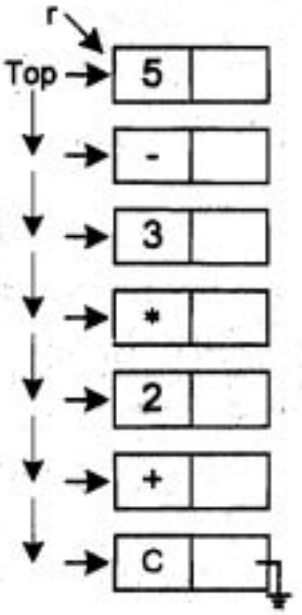


จากตัวอย่าง จะได้เห็นว่า Top จะชี้ที่โหนดบนสุดของลิสต์สแตกนั่นเอง



สำหรับการนำข้อมูลออกจากลิสต์สแตกนั้น เราจะทำการ pop ออกมาโดยใช้คำสั่ง pop (item); การนำข้อมูลตัวที่ Top ชี้อยู่ออกมา และเลื่อนให้ตัวแปรพอยเตอร์ Top ไปชี้ที่โหนดข้อมูลตำแหน่งที่อยู่ตัวถัดไป และทำการลบโหนดข้อมูลที่นำออกคืนให้แก่หน่วยความจำกลางด้วย

```
item = Top -> data;
r = Top;
Top = Top -> link;
delete r;
```



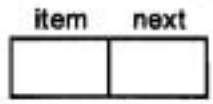
สำหรับการนำข้อมูลออกจากลิสต์สแตกนั้นต้องมีข้อแม้ว่า Top ต้องมีค่าไม่เท่ากับ null สำหรับจากคำสั่งข้างต้นจะเห็นว่า เรานำข้อมูล ณ ตำแหน่งที่ Top ชี้อยู่นำออกมาให้ค่าแก่ตัวแปร item ต่อจากนั้นเราจะลบโหนดข้อมูลนี้ ต้องให้ตัวแปรพอยเตอร์ r ชี้เอาไว้ก่อน แล้วเลื่อนตัวแปรพอยเตอร์ Top ให้ชี้ไปยังโหนดถัดไป แล้วจึงทำการลบโหนดข้อมูลที่ตัวแปรพอยเตอร์ r ชี้อยู่

ตัวอย่างที่ 11.2 Class stacklist

จากการอธิบายการทำงานของลิสต์สแตกในตอนต้นนี้เราจะนำมาออกแบบโครงสร้างของโปรแกรมโดยสร้างเป็นTemplate Class ที่ชื่อว่า stacklist โดยการปฏิบัติการกับข้อมูลในคลาสนี้ผู้ใช้สามารถกำหนดชนิดของข้อมูลที่กระทำได้จากภายนอกได้ทำให้เกิดความหยืดหยุ่นในการแก้ปัญหา เราสามารถระบุรายละเอียดของคลาสดังนี้

คุณลักษณะ (Attribute)

```
struct stackNode
{
        stackelement item;
        stackNode *next;
};
stackNode *top;
```



เรามีการกำหนดโครงสร้างของโหนดข้อมูลให้ชื่อว่า stackNode โดยโหนดข้อมูลประกอบด้วย 2 ฟิลด์คือ ฟิลด์ item จะเก็บข้อมูลได้หลายๆชนิดแตกต่างกันขึ้นอยู่กับการกำหนดของผู้ใช้ โดย stackelement คือชนิดของข้อมูลที่ผู้ใช้กำหนดจากภายนอก สำหรับฟิลด์ next เป็นตัวแปรพอยเตอร์ที่เก็บตำแหน่งที่อยู่หรือชี้ที่โหนดข้อมูลตัวถัดไป ตัวแปร top เป็นตัวแปรชนิดพอยเตอร์ที่ชี้หรือเก็บตำแหน่งของโหนดข้อมูลแรกของลิสต์สแตก

Method : **ฟังก์ชันที่กระทำกับข้อมูล**

1. stackList ( ) เป็น Constructor โดยการปฏิบัติการในฟังก์ชันจะกำหนดค่าเริ่มต้นของลิสต์ให้มีค่าเท่ากับ null คือ stack ว่างนั่นเอง
2. bool push (const stackelement &) เป็นการส่งผ่านค่าที่ส่งมาจากผู้ใช้เพื่อนำไปเก็บใน stack พร้อมตรวจสอบความสำเร็จในการปฏิบัติงาน ในกรณีที่ไม่สามารถนำไปเก็บได้ ผลลัพธ์ของฟังก์ชันมีค่าเท่ากับ false แต่ถ้าสามารถปฏิบัติงานได้สำเร็จจะมีการส่งผ่านค่า true กลับมา
3. bool pop (stackelement &) ทำหน้าที่นำข้อมูลจากลิสต์สแตกออกมาโดย formal parameter ทำหน้าที่เป็น output นอกจากนี้มีการตรวจสอบความสำเร็จของการปฏิบัติงานว่าสามารถกระทำได้หรือไม่ ในกรณีที่สามารถนำออกได้สำเร็จจะมีการส่งผ่านค่า true กลับมาให้แก่ผู้ใช้ ในกรณีที่ไม่สำเร็จกล่าวคือในกรณีที่สแตกว่าง จะส่งผ่านค่า false กลับมาให้แก่ผู้ใช้

4. bool peek (stackelement & x) const เป็นฟังก์ชันที่ทำหน้าที่นำค่าของข้อมูลที่อยู่ตัวบนสุดของสิงค์สแตกออกมาปฏิบัติงาน โดยไม่มีการเปลี่ยนแปลงค่าใดๆในลิสต์สแตก
5. bool isempty() const เป็นฟังก์ชันในการตรวจสอบว่าสิงค์สแตกนี้ว่างไม่หรือไม่ถ้าว่างจะส่งผ่านค่า true กลับมายังผู้ใช้แต่ถ้าไม่ว่างจะส่งค่า false กลับมาให้กับผู้ใช้
6. bool isFull() const เป็นฟังก์ชันในการตรวจสอบว่าสิงค์สแตกนี้เต็มไหมไหม ในกรณีที่เต็มจะส่งค่า true กลับมายังผู้ใช้ แต่ถ้าไม่เต็มจะส่งค่า false กลับมาให้ผู้ใช้

```
//definition of a template class stacklist using a linked list
#ifndef STACK_LIST_H
#define STACK_LIST_H

template <class stackElement>
class stackliist
{
      public:
        //member function.....
        //constructor to create an empty stack
        stacklist();
        //push an element into the stack
        bool push  (const stackelement& x );  // IN:item pushed onto stack
        //pop an element off the stack
        bool  pop   (stackelement& x);                    //OUT: element poped from
stack
        //access top element of stack whitout poping
        bool  peek  (stackelement & x) const; //OUT: value returned  from top of stack
        //test to see if stack is empty
        bool is empty() const;
```

```
        //test to see if stack is full
        bool isFull() const;
    private:
        struct stacknode
            {
                stackelement item;      //storage for the node data
                stacknode*next;         //link to next node
            };
            //data member
            stacknode*top;   //pointer to node at top of stack
};
#endif  // STACK_LIST_H
```

เราสามารถสร้างต้นแบบของ stacklist โดยสร้างเป็นแฟ้มชื่อ stacklist.h และผู้ใช้ภายนอกสามารถเรียกใช้งานได้ดังนี้

```
#include "stacklist.h"
#include <string>
#include <iostream>
using namespace std;
int main ( )
{
        stacklist <string> S;                   ..... 1
        stacklist <int> I;                      .....2
        bool B;                                 .....3
        string item1;
        int item2;
        B = S.push ("ABC");                     ..... 4
        if (B)
                cout <<"success";
```

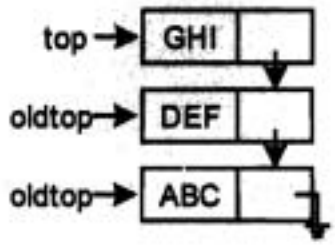
```
        else
                cout <<"not success";
        B = S.push ("DEF");                    ..... 5
        B = S.push ("GHI");                    ..... 6
        B = I.push (13);
        B = I.push (14);
        B = I.push (91);                       .....7
        B = S.pop (item1);                     ..... 8
        if (B)
                cout <<item1;
        B = I.pop (item2);
        if (B)
                cout <<item2;
        return 0;
}
```

คำอธิบาย

1 - 2 เป็นการประกาศออบเจ็กต์ชื่อ S และ I ให้เป็นสมาชิกของ stackclass นั่นคือสามารถเรียกใช้ฟังก์ชันที่เป็น public ได้ แต่สิ่งที่แตกต่างกันคือออบเจ็กต์ S จะมีการปฏิบัติงานกับข้อมูลที่เป็น string ส่วนออบเจ็กต์ I จะปฏิบัติงานกับข้อมูลที่เป็นเลขจำนวนเต็ม โดยค่าที่เก็บใน Top ของออบเจ็กต์ทั้งสองตัวมีค่าเริ่มต้นเท่ากับ null

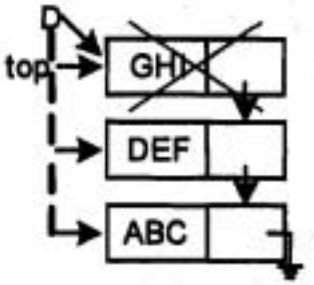
3 ตัวแปร B กำหนดขึ้นมาเพื่อตรวจสอบการปฏิบัติงานกับข้อมูลว่ามีการปฏิบัติงานได้สำเร็จหรือไม่ ถ้าปฏิบัติงานได้สำเร็จค่าของตัวแปรนี้มีค่าเท่ากับ true แต่ถ้าไม่สำเร็จจะมีค่าเท่ากับ false

4 – 6 ทำการนำข้อมูลที่เป็น string ใส่เข้าไปใน ลิสต์สแตกสำหรับออบเจ็กต์ S



7 ทำการนำข้อมูลที่เป็นเลขจำนวนเต็มใส่เข้าไปในลิสต์สแตกสำหรับออบเจ็กต์ I

8 นำข้อมูลออกจากลิสต์สแตกของออบเจ็กต์ S ให้ค่าแก่ตัวแปร Item1



สำหรับการปฏิบัติงานของฟังก์ชันภายในคลาสนั้นเราจะสร้างในชื่อ stacklist.cpp โดยต้องสร้างตามรูปแบบของ Template Class ดังนี้

```
//file: stacklist.cpp
//implementation of template class stack as a linked list
#include "stacklist.h"
#include <cstdlib>
using namespace std;
//member function....
//constructor to create a empty stack
template <class stackelement>
stacklist<stackelement>::stacklist()
{
   top = null;
} // end stacklist
//push  an element in to the stack
//pre: the element x is define.
//    onto the stack and true is returned. Otherwise, the template <class stackelement>
template <class stackelement>
```

```
bool stacklist <stackelement>:: push
    (const stackelement& x ) // IN: element pushed on to stack
{
    //local data
    stacknode *oldtop;
    bool success;       //program flag – indicates
                        // success or failure
   oldtop = top;        // save old top
   top = new stacknode;  //allocate new node at top of stack
   if (top = = null)    //check to see if new was successful
   {
      top = oldtop ;    //if not, restore top
      success = false;          //indicate push failed
  }
else
{
   top->nex = oldtop;           //link new node to old stack
   top->item = x;       //store x in new node
success = true ;        //indicate success
}
return success;
} //end push
//pop element off the stack
//pre: none
//post: if the stack is not empty , the value at the top
//      off the stack is remove, its value is placed in
///       x,and true is return .if the stack is empty ,x is not define and false is return.
```

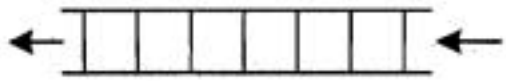
```
template <class stackelement>
bool stacklist<stackelement>::pop
    (stackelement & x)   //out: element poped from stack
{
    //local data
    stackNode*oldtop;
    bool success;   //program flag-indenticates
                    //success or failure
    if  (top = = null)
        success = flase;
    else
    {
     x = top->item;             //copy top of stack into x
     oldTop =  top;             //save old top of stack
     top = oldTop-> next;       //reset top of stack
     delete oldTop ;            //return top node to the heap
     success = true;            //indicate success
    }
    return success;
}  //end pop
//access top element of stack without poping
//pre: none
//post: if the stack is not empty, the value at the top is copied into x and true is return
//    if the stack is empty, x is not defined and false is return ;
//    either case, the stack is not change.
template <class stackElement >
bool stacklist <stackElement> ::peek
```

```
    (stackElement & x) const    //out: value return from stack
{
   //Local data
   bool success;     //program flag – Indicate
                     //   success or failure
   if  (top = = null)
     success = false;
   else
   {
     x = top->item;
     success = true;
   }
   return success;
}  //end peek
//test to see if stack is empty
//pre: none
//post: return true if the stack is empty; otherwise,
//    return false.
template <class stackElement>
bool stackList<stackElement>::isempty() const
{
   return top = = null;
}  //end isempty
//test to see if stack is full
//per: none
//post: return false. List stacks are never full.(dose not check heap availability.)
template <class stackElement>
```

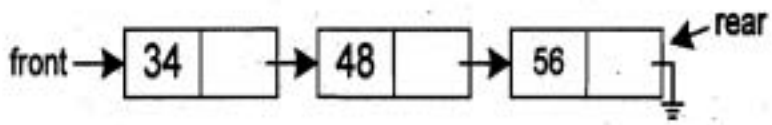
```
bool stackList<stackElement>::isFull() const
{
    return false;
}  //end isfull
```

11.5 Queue as linked list

เป็นการนำลิสต์เชื่อมโยงมาประยุกต์โดยให้มีลักษณะการทำงานในลักษณะของ FI-FO (first in – first out) โดยการปฏิบัติงานกับข้อมูลในโครงสร้างนี้จะมีการกระทำแตกต่างจากลิสต์สแตก โดยการนำสมาชิกเข้าไปเก็บในลิสต์คิวนั้นจะเรียกว่าการ insert โดยกระทำที่ปลายด้านหนึ่งที่มีตัวแปรพอยเตอร์ชื่อ rear ชี้อยู่ที่ข้อมูลตัวสุดท้ายของลิสต์ และการนำข้อมูลออกจากลิสต์คิวจะกระทำที่ปลายอีกด้านหนึ่งเรียกว่าการ remove โดยกระทำที่ปลายอีกด้านหนึ่งในที่นี้ให้ตัวแปรพอยเตอร์ front ชี้ที่ปลายด้านนี้ ตัวอย่างการปฏิบัติงานในลักษณะของคิวได้แก่ การรอคอยซื้ออาหารโดยเข้าคิว , การฝาก-ถอนเงินโดยเข้าคิว โดยผู้ที่อยู่หัวแถวจะได้รับบริการก่อนเป็นลำดับ



การเก็บข้อมูลใน Linked Queue นั้นจะมีการสร้างโหนดข้อมูลขึ้นมามีลักษณะเหมือนกับ Linked Stack โดยมีการเก็บเชื่อมโยงกันมีตัวแปรพอยเตอร์ front เก็บตำแหน่งที่อยู่ของข้อมูลที่โหนดแรก และตัวแปรพอยเตอร์ rear เก็บตำแหน่งที่อยู่ของข้อมูลโหนดสุดท้ายของลิสต์ และตัวเชื่อมโยงของโหนดสุดท้ายต้องมีค่าเท่ากับ null ดังรูป



ถ้าเราต้องการ insert ข้อมูลใหม่เข้าไปในลิสต์คิว ในที่นี้ต้องการนำข้อมูล 78 ใส่เข้าไป

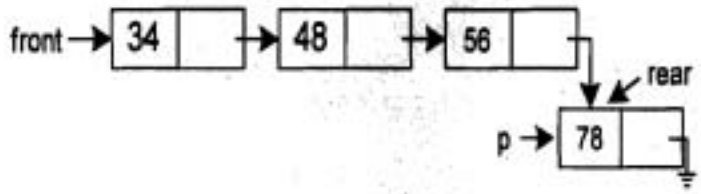
```
insert (78);
```

เราต้องจองเนื้อที่ใหม่ 1 โหนดสำหรับเก็บข้อมูล ในที่นี้ให้ตัวแปรพอยเตอร์ p ชี้ที่โหนดใหม่นี้ และนำข้อมูลไปเก็บในโหนด

```
p = new  queueNode ;
p -> item = 78;
```

สำหรับการเชื่อมโยงเราจะให้ตัวเชื่อมโยงของโหนดสุดท้ายไปชี้ที่โหนดใหม่ และเปลี่ยนให้ตัวแปรพอยเตอร์ rear ไปชี้ที่โหนดใหม่แทน

```
rear -> next = p;
rear = p;
```



สำหรับการนำข้อมูลออกจากลิสต์คิวจะกระทำในลักษณะเดียวกับลิสต์สแตกแต่เปลี่ยนตัวแปรพอยเตอร์จาก Top เป็น front แทน

จากการทำงานของลิสต์คิวเราสามารถนำมาสร้างเป็น Template class ชื่อ queue ได้ดังนี้

```
//file: queue.h
//definition of a template class queue using linked list
#ifndef QUEUE_H
#define QUEUE_H
//specification of the class queue<queueElement>
//element:A queue consists of a collection of element that are all of the
sametype,queueElement.
//structure :  The element of a queue are ordered according to time of arrival . the
element was
//    first inserted into the queue is the only one that may be remove or examined .
Element are
//    remove from the front of the queue and inserted at the rear of the queue.
```

```
template<class queueElement>
class queue
{
    public:
     //member function ...
     //constructor – create an empty queue
     queue();
     //insert an element into the queue
     bool insert
       (const queueElement & x);    //in:Element to insert
     //remove an element from the queue
     bool remove
       (queueElement & x);     // out:element remove
     //test for empty queue
     bool isEmpty();
     //Get queue size
     int getSize();
    private:
     //data member...
     {
       queueElement item;
       queueNode*next;
     };
     queueNode*front;      //the front of the queue
     queueNode*rear;       //the back of the queue
     int numItems;         //the number of item currently in the queue
};
```

```
#endif   //QUEUE_H
```

โครงสร้างของโหนดของลิสต์คิวนั้นเหมือนกับลิสต์ลแตกทุกประการ แต่ตัวชี้โหนดจะมี 2 ตัวคือ front เก็บตำแหน่งที่อยู่ของโหนดแรกของลิสต์คิวและ rear จะเก็บตำแหน่งที่อยู่ของโหนดสุดท้ายของคิว ในกรณีที่คิวว่าง ทั้ง front และ rear มีค่าเท่ากับ null ถ้ามีการนำข้อมูลเก็บในลิสต์คิวเป็นโหนดแรกจะทำให้ตัวแปรพอยเตอร์ทั้ง front และ rear ชี้ที่โหนดใหม่นี้ เรามาดูการเรียกใช้ Template class queue จากโปรแกรมต่อไปนี้

```
//File : test.cpp
#include "queue.h"
#include <iostream>
using namespace std;
int main ()
{
        queue <string> S;                    .....1
        queue <int> I;                       .....2
        bool B;
        string item1;
        int item2;
        B = I.insert (34);                   .....3
        B = I.insert (56);                   .....4
        B = I.insert (39);                   .....5
        B = I.remove (item2);                .....6
        If (B)
                cout <<item2 <<endl;
        B = S.insert ("CT212");              .....7
        B = S.insert ("CT484");              .....8
        B = S.insert ("CT479");              .....9
        B = S.remove (item1);                .....10
```

```
    if (B)
        cout <<item1 <<endl;
    return 0;
}
```

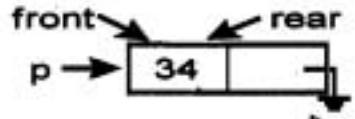
คำอธิบาย

1-2 เป็นการประกาศออบเจ็กต์ชื่อ S และ I ให้เป็นสมาชิกของคลาส queue โดยข้อมูลในการ

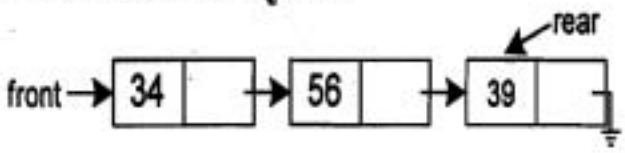
ปฏิบัติงานกับออบเจ็กต์ S เป็นชนิด string แต่ออบเจ็กต์ชื่อ I กระทำกับข้อมูลที่เป็นเลขจำนวนเต็ม

3-5 เป็นการนำสมาชิกเข้าไปเก็บในออบเจ็กต์ชื่อ I

B = I.insert (34);

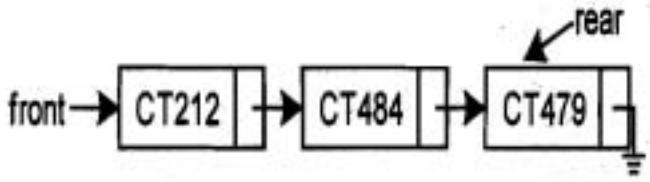


ผลของการทำงานได้ข้อมูลดังนี้



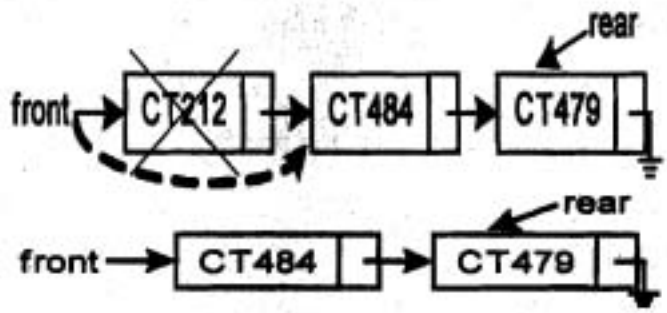
6 เป็นการนำข้อมูลออกจากลิสต์คิวของออบเจ็กต์ I ส่งผลให้ front เลื่อนไปชี้ที่โหนดถัดไปในที่นี้ โหนดที่มีข้อมูล 34 จะถูกลบออกจากลิสต์นี้

7-9 เป็นการนำข้อมูลเข้าลิสต์คิวของออบเจ็กต์ S ดังรูป



10 เป็นคำสั่งในการนำข้อมูลออกจากลิสต์คิวของออบเจ็กต์ S ส่งผลให้ item1 มีค่าเท่ากับ "CT212"

ผลจากการทำงานลิสต์คิวของออบเจ็กต์ S จะเก็บข้อมูลดังนี้



สำหรับฟังก์ชันการทำงานภายในคลาสเราจะสร้างในแฟ้มที่ชื่อเดียวกับชื่อคลาสแต่มีนามสกุลเป็น .cpp ดังนี้

```
// File: queue.cpp
// Implementation of template class queue
#include "queue.h"
#include  <cstdlib>                    // for null
using namespace std ;

// Member functions
// constructor - create an empty queue
template<class queueElement>
queue<queueElement> : : queue()
{
        numItems = 0 ;
       front = null ;
       rear = null ;
}

// Insert an element into the queue
```

```
// Pre  : none
// Post : If heap space is available, the value x is inserted at the rear of the queue and
true is
// returned  Otherwise , the queue is not changed and false is    returned .
template<class queueElement>
bool queue<queueElement> : : insert
        (const queueElement& x)          // IN: Element to insert
{
    if (numItems = = 0)                  // Test for empty queue
    {
        rear = new queueNode ;           // Allocate first queue node
        if (rear = = NULL)               // Check to see if allocated
            return false ;
        else
            front = rear ;               // queue with one element
    }
      else
    {
        rear -> next = new queueNode ;   // Connect new last node
        if (rear -> next = = NULL)
            return false ;               // no node connected
        else
            rear = rear -> next ;        // Point rear to last node
    }
      rear -> item = x ;                 // Store data in last node
      numItems++ ;
      return true ;
```

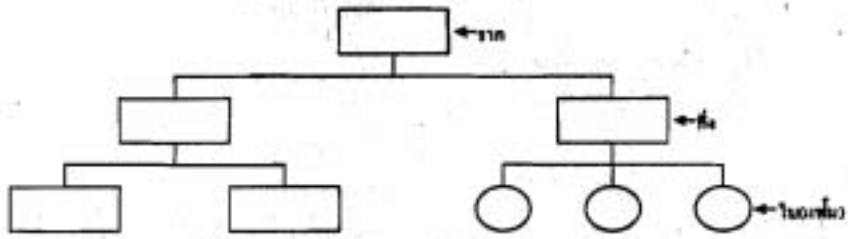
```
}   // end insert
// Remove an element from the queue
// Pre  : none
// Post : If the queue is not empty, the value at the front of    the queue is removed, its
value is
//  placed in x, and   true is returned. If the queue is empty, x is not  defined and false is
returned .
template<class queueElement>
bool queue<queueElement> : : remove
        (queueElement& x)   // OUT : element removed
{
    // Local data
    queueNode* oldFront ;
    if (numItems = = 0)                    // Test for  empty queue
      {
        return false ;                     // queue was empty
      }
      else                                 // Remove first node
      {
        oldFront = front ;                 // Point oldFront to first node
        x = front -> item ;                // Retrieve its data
        front = front -> next ;            // Bypass old first note
        oldfront -> next = null ;          // Disconnect old first node
        delete old front ;                 // Return its storage
        numItems-- ;                       // Decrement queue size
        return true ;
      }
```

```
}          // end remove
// test whether queue is empty
template<class queueElement>
bool queue<queueElement> : : isEmpty( )
{
        return (numItems = = 0) ;
}

// Return queue size
templete<class queueElement>
bool queue<queueElement> : : getSize ( )
{
        return numItems ;
}
```

11.6 Tree Structure

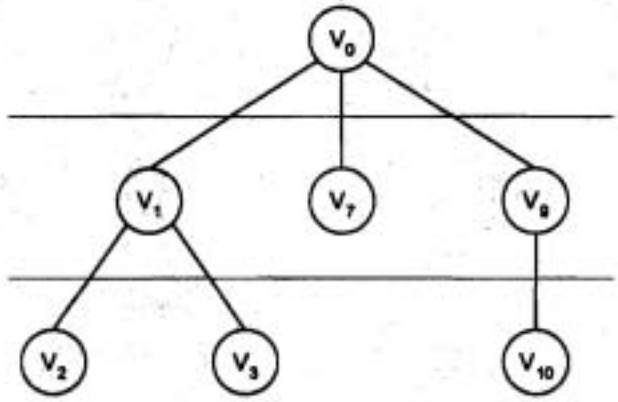
Tree Structure หรือโครงสร้างต้นไม้เป็นการนำตัวแปรชนิดพอยเตอร์มาใช้งานในอีกลักษณะหนึ่งโดยเก็บโหนดของข้อมูลแต่ไม่เป็นเชิงเส้น (non-linear list) แต่การจัดเก็บเป็นลำดับชั้น (hierarchical) ในลักษณะที่ลำดับชั้นบนสุดจะทำหน้าที่คล้ายรากของต้นไม้ และลำดับชั้นถัดๆไปเปรียบเสมือนกิ่งของต้นไม้ โดยสามารถแตกแยกย่อยเป็นกิ่งย่อยๆได้ในลำดับชั้นถัดๆไปจนกระทั่งเป็นใบ เปรียบเสมือนแฟ้มข้อมูลหนึ่งๆที่ไม่สามารถแตกออกไปได้อีก ตัวอย่างของการนำข้อมูลมาเก็บในโครงสร้างของต้นไม้ที่เรารู้จักกันดีคือ การจัดเก็บสารสนเทศของแฟ้มข้อมูลหรือแฟ้มโปรแกรมต่างๆในสื่อกลางภายนอก เช่น ดิสก์ , แผ่น CD-ROM ซึ่งเราจะมีการเก็บแฟ้มชนิดเดียวกันไว้ในโฟลเดอร์เดียวกัน มีการออกแบบในการจัดเก็บแฟ้มเป็นลำดับชั้น การเข้าถึงแฟ้มที่ต้องการก็ต้องมีการอ้างทางเดิน(path) ให้ถูกต้อง ไปตามลำดับชั้นและกิ่งต่างๆจนถึงแฟ้มที่ต้องการ



โครงสร้างต้นไม้หรือTree นี้ ประกอบด้วย เซตของโหนดข้อมูล โดยโหนดข้อมูลจะมีลักษณะดังนี้

1. โหนดพิเศษ เรียกว่า root (โหนดเริ่มต้น อยู่บนสุดของลำดับชั้น)
2. โหนดอื่นๆ จะถูกแบ่งเป็นเซตย่อยๆ จะเรียกว่า sub tree

ตัวอย่าง



จะเห็นว่า V_0 เป็น root

Tree = $\{V_0, V_1, V_2, V_3, V_7, V_9, V_{10}\}$

Sub tree ของ V_0 = $(\{V_1, V_2, V_3\}, \{V_7\}, \{V_9, V_{10}\})$

Degree ของ Node หมายถึง จำนวนต้นไม้ย่อยของแต่ละโหนด

V_0 มี degree = 3

V_1 มี degree = 2

V_7 มี degree = 0

V_9 มี degree = 1

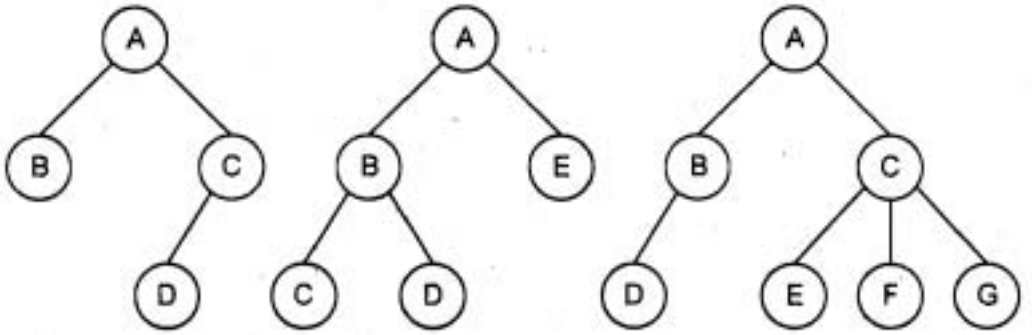
โหนดใดที่มี degree = 0 ได้แก่ V_2, V_3, V_7, V_{10}

เราเรียกโหนดเหล่านี้ว่า Leaf node (โหนดที่เป็นใบ)

Binary Tree

เป็นต้นไม้ที่มีกิ่งแยกออกมา (out node) ของทุกๆ โหนดน้อยกว่าหรือเท่ากับ 2 ทุกๆ โหนดแต่ถ้าทุกโหนดมี out node เท่ากับ 2 หรือ 0 โดยไม่มี out node เท่ากับ 1 เลย เราเรียกว่าต้นไม้นี้ว่า Full Binary Tree

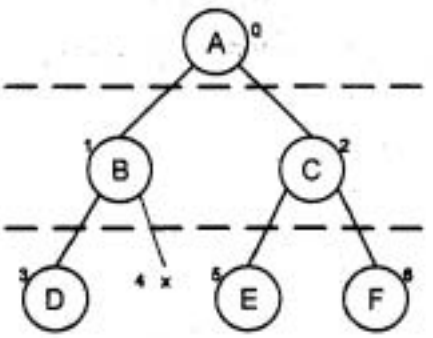
พิจารณาจากต้นไม้แบบไบนารีต่อไปนี้



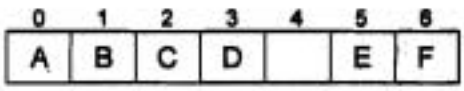
Binary Tree | Full Binary Tree (เต็มต้น) | ไม่ใช่ Binary Tree

ทุกโหนดมี out node = 2 หรือ 0 ทุกโหนด

เราสามารถแทนความหมายของต้นไม้แบบไบนารีในหน่วยความจำโดยกระทำได้หลายลักษณะ ตามมุมิร้อมูลในต้นไม้แบบไบนารีเป็นดังภาพต่อไปนี้

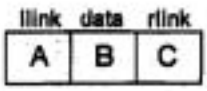


1. Sequential link เป็นการแทนต้นไม้เป็นชนิด Full Binary Tree กำหนดโดย จองอาเรย์1 มิติให้มีขนาดเท่ากับต้นไม้แบบเต็มต้นโดยเก็บ รากที่ตำแหน่งแรกในที่นี้คือ 0 ดังนี้



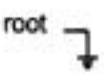
การจัดเก็บโครงสร้างแบบนี้มักใช้กับภาษาโปรแกรมที่ไม่ได้รอบรับตัวแปรชนิดพอยเตอร์ไว้ให้เราสามารถจัดเก็บในลักษณะนี้ได้

2. Link allocation เป็นการเก็บโดยใช้ลิสต์เชื่อมโยง(linked list) แต่ข้อมูลแต่ละโหนดจะมีตัวเชื่อมโยง 2 ตัว โดยใช้ตัวแปรพอยเตอร์ตัวแรกจะเก็บตำแหน่งที่อยู่ของโหนดที่เป็นลูกทางซ้าย และตัวแปรพอยเตอร์อีกตัวหนึ่งจะเชื่อมโยงโดยเก็บตำแหน่งที่อยู่ของโหนดที่เป็นลูกทางขวา



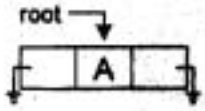
โดย llink เป็นตัวแปร พอยเตอร์ที่ ชี้ไปยัง left sub tree
data เป็นตัวแปรที่ใช้สำหรับ เก็บข้อมูล
rlink เป็นตัวแปร พอยเตอร์ ชี้ไปยัง right sub tree

การนำข้อมูลที่ต้องการเก็บในโครงสร้างต้นไม้นั้น เราเพียงแต่ให้ตัวแปรพอยเตอร์ชี้ที่ราก หรือ root ซึ่งเป็นโหนดข้อมูลในระดับบนสุด ในที่นี้กำหนดตัวแปรพอยเตอร์ root ขึ้นมาในกรณีที่ต้นไม้ว่าง ไม่มีข้อมูลใดๆเก็บอยู่ root จะมีค่าเท่ากับ null



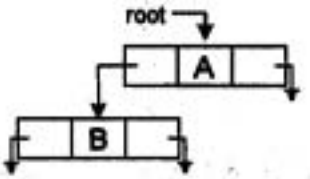
ถ้าเราต้องการนำข้อมูลค่าแรกเก็บในโครงสร้างต้นไม้ ตัวแปรพอยเตอร์ root จะชี้ที่โหนดข้อมูลนี้

insert ('A');

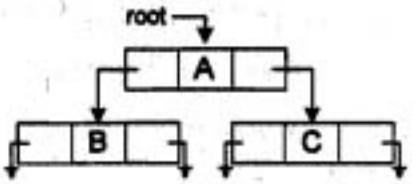


ในกรณีที่นำข้อมูลตัวต่อๆไปเก็บในโครงสร้างต้นไม้ ข้อมูลตัวใหม่จะถูกจัดเก็บในโหนดใหม่ และนำโหนดใหม่นี้ไปเชื่อมโยงกับโหนด root ในลำดับขั้นถัดไป

Insert ('B');



Insert ('C');



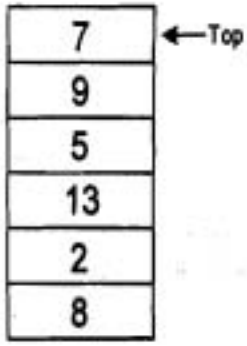
สำหรับข้อมูลตัวถัดๆไปจะถูกนำไปจัดเก็บในระดับขั้นถัดไปอีก ซึ่งการจัดเก็บนั้นมีกฎเกณฑ์แตกต่างกันขึ้นอยู่กับชนิดของต้นไม้ ซึ่งมีอยู่มากมายหลายประเภท สำหรับการเรียนการสอนในวิชานี้มุ่งเน้นให้ทราบถึงโครงสร้างต้นไม้เบื้องต้น รวมทั้งการปฏิบัติงานกับโครงสร้างต้นไม้ไม่ว่าจะเป็นการนำข้อมูลไปเก็บในโครงสร้างต้นไม้ การลบข้อมูลในต้นไม้ การท่องต้นไม้ เป็นต้น

<u>Binary Search Tree</u> เป็นต้นไม้แบบ Binary ชนิดหนึ่งที่มีโหนดที่สามารถเชื่อมต่อกับโหนดอื่นๆได้ไม่เกิน 2 โหนดหรือมี out node ไม่เกิน 2 ต้นไม้ชนิดนี้มีกฎเกณฑ์ในการสร้างดังนี้

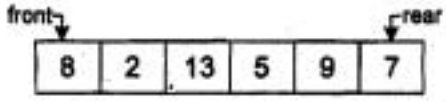
1. ค่าที่เก็บที่ root ต้องมีค่ามากกว่าข้อมูล Sub Tree ทางซ้ายทุกโหนด
2. ค่าที่เก็บใน Sub Tree ทางขวา ต้องมากกว่า หรือเท่ากับ root ทุกๆ โหนด

<u>ตัวอย่าง</u> **ต้องการเก็บข้อมูลดังนี้** 8 2 13 5 9 7

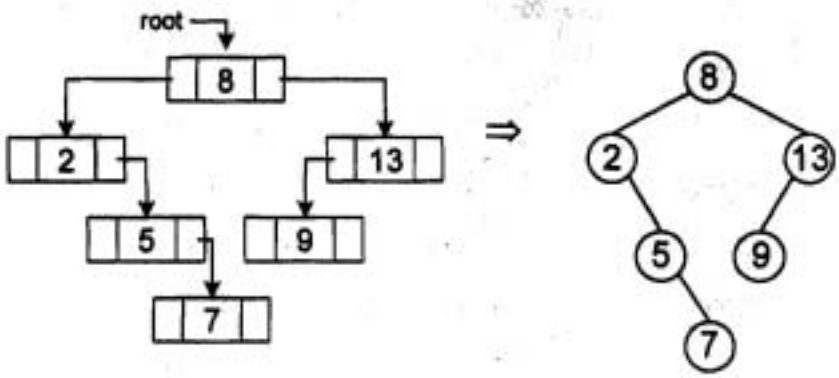
ถ้าเราต้องการนำข้อมูลเก็บแบบ Stack จะเก็บดังนี้



เก็บแบบ Queue จะเก็บดังนี้

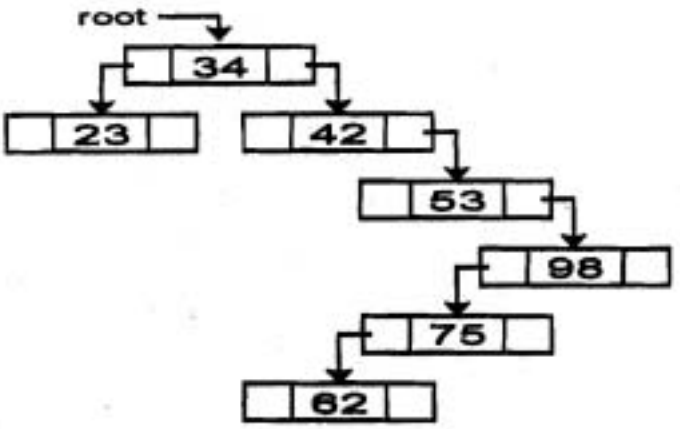


เก็บแบบ Binary Search Tree จะเก็บดังนี้



ตัวอย่าง สร้างต้นไม้ Binary Search Tree จากข้อมูลต่อไปนี้

34 42 53 98 75 62 23



จากคุณลักษณะของ Binary Search Tree ดังกล่าวข้างต้นเราจะนำมาสร้างเป็น Template class ที่ชื่อว่า binaryTree โดยระบุรายละเอียดของคลาสดังนี้

Attributes for Binary Tree Class

root	// a pointer to the tree root

Member Functions for Binary Tree Class

binarytree	// a constructor
insert	// inserts an item into the tree
retrieve	// retrieves all the data for a given key
search	// locates the node for a given key
display	// displays a binary tree

จะเห็นได้ว่าจะมีตัวแปรพอยเตอร์ root ซึ่งเก็บตำแหน่งที่อยู่ของรากต้นไม้ที่อยู่ในระดับบนสุด ส่วนการปฏิบัติการกับต้นไม้นี้จะเป็นการนำข้อมูลใหม่ไปเก็บในโครงสร้างต้นไม้ การนำข้อมูลที่ต้องการออกมาเพื่อปฏิบัติงาน การค้นหาตำแหน่งที่อยู่ของโหนดที่เก็บข้อมูลที่ต้องการ รวมทั้งการท่องต้นไม้เป็นการนำข้อมูลทั้งหมดที่เก็บในโครงสร้างต้นไม้มาพิมพ์ออกทางจอภาพ

```
// File: binarytree.h
// Definition of template class binary search tree
```

```
#include <cstdlib>              // for null
using namespace std ;

#ifndef  BINARY_TREE_H
#define  BINARY_TREE_H

// Specification of the class binarytree <treeElement>
// Elements :    A tree consists of a collection of elements
//               that are all of the same type, treeElement .
// Atructure :   Each node of a binary search tree has zero, one,
//               or two subtrees connected to it. The key value in
//               each node of a binary search tree is larger than
//               all key values on its right subtree .
//               than all key values in its right subtree .

template <class treeElement>
class binarytree
{
    public :
        // Member functions    . . .
        // constructor – create an empty tree
        binarytree( ) ;

        // Insert an element into the tree
        bool insert
           (const treeElement& el );          // IN: Element to insert
```

```
    // Retrieve an element from the tree
    bool retrieve
       (const treeElement& el) const;     // OUT: element retrieved
    // Search for an element in the tree
    bool search
       (const treeElement& el) const ;    // IN: element being searched
    // Display a tree
    void display( ) const ;
private :
    // Data type  . . .
    struct treeNode
    {
          treeElement info ;                    // the node data
          treeNode *left ;                      // pointer to left-subtree
          treeNode *right ;                     // pointer to right-subtree
    } ;
    // Data member
    treeNode* root ;                            // the root of the tree

    // Private member function  . . .
    // searches a subtree for a key
    bool search (treeNode*,                             // root of a subtree
                 const treeElement&) const ;    // item being inserted
    // Inserts an item in a subtree
    bool insert (treeNode*&,                    // root of  a subtree
                 const treeElement&) ;          // item being inserted
    // Retrieves an item in a subtree
```

```
        bool retrieve (treeNode*,                    // root of a subtree
                       treeElement&) const ;         // item to retrieve
        // Displays a subtree
        void display (treeNode*) const ;
} ;
#endif   // BINARY_TREE_H
```

สำหรับการออกแบบคลาสนี้ผู้ใช้จะไม่ทราบว่าต้นไม้มีการเก็บอย่างไร เพียงแต่ส่งผ่านตำแหน่งที่อยู่ของโหนด root เพื่อปฏิบัติงาน ระบบจะกระทำงานตามต้องการได้ทันที แต่การสร้างฟังก์ชันที่กระทำนั้น จากตัวอย่างนี้มีการสร้างฟังก์ชันที่เป็น private และมีชื่อเหมือนกันกับฟังก์ชันที่เป็น public สิ่งที่แตกต่างกันคือการทำงานและ formal parameter ที่แตกต่างกัน

```
// File: binarytree.cpp
// Implementation of template class binary search tree

#include  "binarytree.h"
#include <iostream>
using namespace std ;

      // member functions  . . .
      // constructor – create an empty tree
      template <class treeElement>
      binarytree <treeElement> : : binarytree ( )
      {
         root = null ;
    }

    // Search for the item with same key as el
```

```
// in a binary search tree . (public)
// Pre  : el is difined .
// Returns true if el 's key is located ,
// otherwise, return false .
template<class treeElemente>
bool binarytree <treeElement> : : search
    (const treeTree& el) const              // IN: Element to  search for
    {
        return search (root , el) ;                 // Start search at tree root .
    }       // search

    // (private) Searches for the item with same key as el in the
    // subtree pointed to by aRoot. Called by public search .
    // Pre : el and aRoot are defined .
    // Returns true if el 's key is located ,
    // otherwise, returns false.
    template<class treeElement>
    bool binarytree<treeElement> : : search
        (treeNode *aRoot,                       // IN: Subtree to search
        const treeElement& el) const            // IN: Element to search for
    {
        if (aRoot = = null)
            return false ;                              // Tree is empty .
        else if (el <=  aRoot -> info)
            return  true ;                      // Target is found .
        else if (el <=  aRoot -> info)
            return search (aRoot -> left, el) ;  // Search left .
```

```
            else
                return search (aRoot -> right, el) ;          // Search right .
        }       // search

// Insearch itemp el into a binary search tree . (public)
// Pre  : el is defined .
// Post : Insearch el if el is not the tree .
//        Returns true if the insertion is performed .
//        If there is a node with the same key value as el ,
//        returns false .
template<class treeElement>
bool binarytree<treeElement> : : insert
    (const treeElement& el)          // IN – item to insert
{
    return insert (root , el) ;
}   // insert

// Inserts item el in the tree pointed to by aRoot . (private)
// Called by public insert .
// Pre  : aRoot and el are defined .
// Post : If  a node with same key as el is found,
//        returns false . If an empty tree is reached,
//        inserts el as a leaf node and returns true .
template<class treeElement>
bool binarytree<treeElement> : : insert
    (treeNode*& aRoot,                          // INOUT  : Insertion subtree
     const treeElement& el)                     // IN     : Element to insert
```

```
{
    // Check for empty tree .
    if (aRoot = = null)
    {   // Attach new node
        aRoot = new treeNode ;                    // const aRoot to new node .
        aRoot -> left = null ;                    // Make new node a leaf .
        aRoot -> right = null ;
        aRoot -> info = el ;                      // Place el in new node
        return true ;
    }
    else if (el = = aRoot -> info)
        return false ;                            // duplicate key found .
    else if (el <= aRoot -> info)
        return insert (aRoot -> left,  el) ;      // insert left .
    else
        return insert (aRoot -> right,  el) ;     // insert right .
}   // insert

// Displays a binary search tree in key order . (public)
// Pre  : none
// Post : Each element of the tree is displayed .
//        Elements are displayed in key order .
template<class treeElement>
void binarytree<treeElement> : : display( ) const
{
    display(root) ;
}   // display
```

```
// (private) displays the binary seach treepointed to
// by aRoot in key order . Called by display .
// Pre  : aRoot is defined .
// Post : displays each node in key order .
template<class treeElement>
void binarytree<treeElement> : : display
     (treeNode *aRoot) const                    // IN : subtree to display
{

     if (aRoot != null)

     {  // recursive step

        display (aRoot -> left) ;               // display left subtree .

        cout << aRoot  -> info << endl ;        // display root item .

        display (aRoot -> right) ;              // display right subtree .

     }

}    // display

// Insert public and private member function retrieve
```

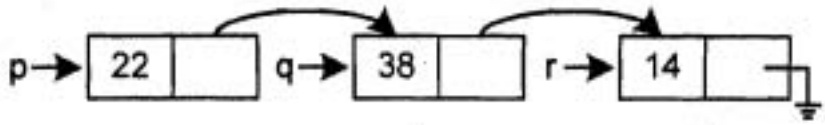
สรุป

1. พอยเตอร์(pointer) เป็นชนิดของข้อมูลหนึ่งที่เก็บตำแหน่งที่อยู่ของข้อมูล

   รูปแบบการประกาศตัวแปรชนิดพอยเตอร์

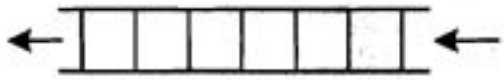
   Type * Variable;

2. การจองเนื้อที่ให้กับตัวแปรพอยเตอร์ เป็นการจองแบบจลย์(Dynamic) โดยเราจะจองเมื่อโปรแกรมทำงานหรือต้องการเก็บข้อมูลจริงๆ
   คำสั่งมีรูปแบบดังนี้ new type;
3. การนำค่าไปจัดเก็บ หรือการนำข้อมูลที่จัดเก็บในตำแหน่งที่อยู่ที่ต้องการมาใช้งาน เราใช้สัญลักษณ์ * หรือ Asterisk หรือที่เรียกว่า Indirection operator ใช้อ้างถึงค่าของข้อมูลที่ตัวแปรพอยเตอร์ ชี้อยู่
4. การลบหรือปล่อยเนื้อที่ของตัวแปรพอยเตอร์ รูปแบบ delete Variable;
5. Singly linked list เป็นจัดเก็บในลักษณะของลิสต์เชื่อมโยงทางเดียว เป็นการนำตัวแปรพอยเตอร์มาเก็บตำแหน่งของข้อมูลตัวถัดไป ดังรูป

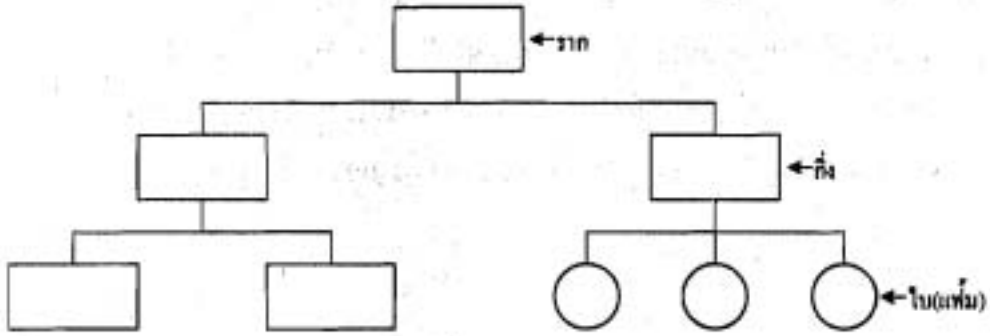


6. ลักษณะของ node ข้อมูลในSingly linked list เป็นโครงสร้างชนิดเรคอร์ดที่ประกอบด้วยฟิลด์ใหญ่ๆ 2 ฟิลด์ด้วยกันคือ
   - Information field เป็นรายการข้อมูลจริงที่เราต้องการจัดเก็บ
   - link field เป็นรายการข้อมูลที่เป็นตัวแปรชนิดพอยเตอร์ ที่เก็บตำแหน่งของโหนดตัวถัดไป ในกรณีที่ไม่มีข้อมูลตัวถัดไปเราจะชี้ไปที่ null ซึ่งเป็นจุดจบของลิสต์ข้อมูล
7. สแต็ก(stack) เป็นโครงสร้างข้อมูลที่เป็นลักษณะ LI-FO (last in - first out) หมายถึง ข้อมูลนำเข้าตัวสุดท้ายจะได้รับการบริการก่อน โดยข้อมูลที่มีการนำเข้า และนำออกได้เพียงทางเดียว

8. Linked queue เป็นการนำลิสต์เชื่อมโยงมาประยุกต์โดยให้มีลักษณะการทำงานในลักษณะของ FI-FO (first in – first out) โดยการนำสมาชิกเข้าไปเก็บในลิสต์คิวนั้นจะเรียกว่าการ insert โดยกระทำที่ปลายด้านหนึ่งที่มีตัวแปรพอยเตอร์ชื่อ rear ชี้อยู่ที่ข้อมูลตัวสุดท้ายของลิสต์ และการนำข้อมูลออกจากลิสต์คิวจะกระทำที่ปลายอีกด้านหนึ่งเรียกว่าการ remove โดยกระทำที่ปลายอีกด้านหนึ่งในที่นี้ให้ตัวแปรพอยเตอร์ front ชี้ที่ปลายด้านนี้



9. Tree Structure หรือโครงสร้างต้นไม้เป็นการนำตัวแปรชนิดพอยเตอร์มาใช้งานในอีกลักษณะหนึ่งโดยเก็บโหนดของข้อมูลแต่ไม่เป็นเชิงเส้น (non-linear list) แต่การจัดเก็บเป็นลำดับชั้น (hierarchical) ในลักษณะที่ลำดับชั้นบนสุดจะทำหน้าที่คล้ายรากของต้นไม้ และลำดับชั้นถัดๆไปเปรียบเสมือนกิ่งของต้นไม้



10. โครงสร้างต้นไม้หรือTree นี้ ประกอบด้วย เซตของโหนดข้อมูล โดยโหนดข้อมูลจะมีลักษณะดังนี้

    10.1โหนดพิเศษ เรียกว่า root (โหนดเริ่มต้น อยู่บนสุดของลำดับชั้น)

    10.2โหนดอื่นๆ จะถูกแบ่งเป็นเซตย่อยๆ จะเรียกว่า sub tree

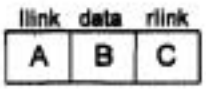
11. Degree ของ Node หมายถึง จำนวนต้นไม้ย่อยของแต่ละโหนด

    โหนดใดที่มี degree = 0 เราเรียกโหนดเหล่านี้ว่า Leaf node (โหนดที่เป็นใบ)

12. Binary Tree เป็นต้นไม้ที่มีกิ่งแยกออกมา (out node) ของทุกๆ โหนดน้อยกว่าหรือเท่า

กับ 2 ทุกๆโหนดแต่ถ้าทุกโหนดมี out node เท่ากับ 2 หรือ 0 โดยไม่มี out node เท่ากับ 1 เราเรียกว่าต้นไม้นี้ว่า Full Binary Tree

13. Link allocation เป็นการเก็บโดยใช้ลิสต์เชื่อมโยง(linked list) แต่ข้อมูลแต่ละโหนดจะมีตัวเชื่อมโยง 2 ตัว โดยใช้ตัวแปรพอยเตอร์ตัวแรกจะเก็บตำแหน่งที่อยู่ของโหนดที่เป็นลูกทางซ้ายและตัวแปรพอยเตอร์อีกตัวหนึ่งจะเชื่อมโยงโดยเก็บตำแหน่งที่อยู่ของโหนดที่เป็นลูกทางขวา



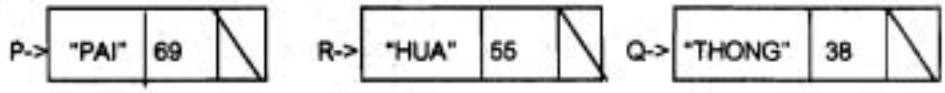
โดย llink เป็นตัวแปร พอยเตอร์ที่ ชี้ไปยัง left sub tree
data เป็นตัวแปรที่ใช้สำหรับ เก็บข้อมูล
rlink เป็นตัวแปร พอยเตอร์ ชี้ไปยัง right sub tree

14. Binary Search Tree เป็นต้นไม้แบบ Binary ชนิดหนึ่งที่มีโหนดที่สามารถเชื่อมต่อกับโหนดอื่นๆได้ไม่เกิน 2 โหนดหรือมี out node ไม่เกิน 2 ต้นไม้ชนิดนี้มีกฎเกณฑ์ในการสร้างดังนี้ค่าที่เก็บที่ root ต้องมีค่ามากกว่าข้อมูล Sub Tree ทางซ้ายทุกโหนด ค่าที่เก็บใน Sub Tree ทางขวา ต้องมากกว่า หรือเท่ากับ root ทุกๆ โหนด

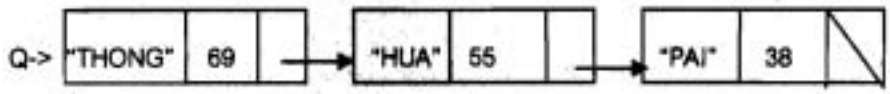
แบบฝึกหัด

1. พิจารณาโครงสร้างข้อมูลของนักศึกษาและรูปภาพต่อไปนี้แล้วตอบคำถาม

```
struct STU
{
    string   name;
    double  score;
    STU    *link;
}
STU  *P, *Q, *R ,*Head;
```

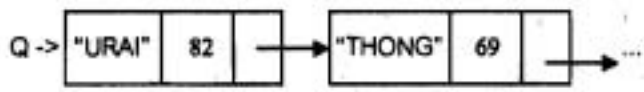


คำถามข้อ 1.1 จงเขียนส่วนของคำสั่งในการเชื่อมต่อโหนดทั้งสามเข้าด้วยกันให้ผลลัพธ์สุดท้ายเป็นดังนี้ตามภาพ



คำถามข้อ 1.2 จงเขียนคำสั่งในการสร้างโหนดใหม่โดยเก็บ ข้อมูล "URAI" และ 82 ในฟิลด์ต่างๆตามลำดับ

คำถามข้อ 1.3 จงเขียนคำสั่งแทรกโหนดใหม่ที่สร้างในคำถามข้อ 2.2 ที่ต้นลิสต์เชื่อมโยง โดยให้ตัวแปรพอยเตอร์ Q ชี้ดังภาพ



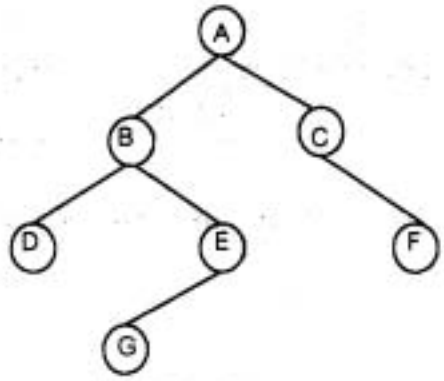
คำถามข้อ 1.4 จงเขียนคำสั่งในการลบโหนดแรกของลิสต์

2. จงอธิบายความแตกต่างของโครงสร้างสแตก(stack) และ คิว(queue) ของการนำสมาชิกเพิ่มและนำออกจากโครงสร้างทั้งสองชนิด โดยกำหนดให้ข้อมูลที่ต้องการประมวลผลเป็นเลขจำนวนเต็มดังนี้

52 62 78 12 23 15 89 56 (5 คะแนน)

3. พิจารณาข้อมูลต่อไปนี้ 45 85 95 23 42 52 84 12 35 24
จงวาดรูป Binary Search Tree และเขียนฟังก์ชันในการท่องต้นไม้นี้พิมพ์ข้อมูลเฉพาะที่มากกว่า 50 ออกมาทางจอภาพ

4. พิจารณาต้นไม้ต่อไปนี้เพื่อตอบคำถาม



ก. จงแสดงการแทนความหมายของ Binary tree ในหน่วยความจำแบบที่ใช้ sequential link

ข. จงประกาศโครงสร้างข้อมูลของ Binary tree แบบ link allocation และอธิบายด้วยว่า field ต่างๆหมายถึงอะไร

5. จงเขียนฟังก์ชันที่ชื่อ Is_Empty(P) เพื่อตรวจสอบว่า ลิสต์ ที่มีตัวแปรพอยเตอร์ P เก็บตำแหน่งเริ่มต้นของ list เป็นลิสต์ว่างหรือไม่

6. จงเขียนฟังก์ชั่น find_previous(X, P) เพื่อหาตำแหน่งของข้อมูลก่อนหน้า กำหนดให้ X เป็นข้อมูลที่ต้องการ P เก็บตำแหน่งเริ่มต้นในลิสต์

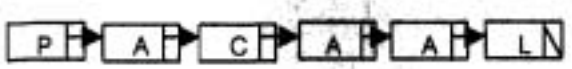
7. จงเขียนฟังก์ชัน numberoNode(L) เพื่อนับจำนวนของโหนดทั้งหมด ที่อยู่ในลิสต์ที่ L ชี้อยู่ที่โหนดแรก

8. จงเขียนฟังก์ชั่นเหมือนข้อ 7 แต่เขียนในลักษณะแบบเรียกตัวเอง

9. จงเขียนฟังก์ชันเพื่อเชื่อมโยงข้อมูลในลิสต์ 2 ลิสต์ใดๆ ชื่อ Concate(a, b, c)

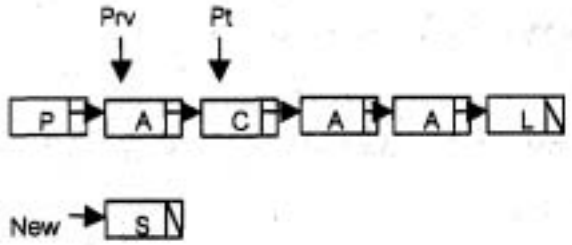
   a. กำหนดให้ a และ b เป็นตัวชี้โหนดแรกของ 2 ลิสต์ใดๆ ผลจากการทำงานจะนำลิสต์ b เชื่อมต่อกับ ลิสต์ a โดยให้ c ชี้ที่โหนดใหม่

10. ต้องการเก็บคำศัพท์ในภาษาอังกฤษโดยเก็บเป็น singly linked list ด้วยโครงสร้างข้อมูลแบบ struct เช่น คำว่า PACAAL จะถูกจัดเก็บลักษณะดังนี้

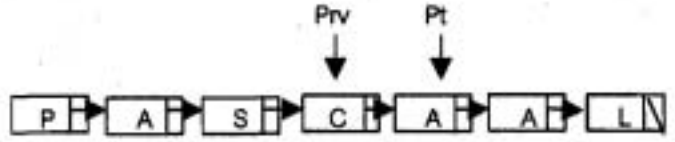


   ก. จงประกาศโครงสร้างข้อมูล ของ Node type ใน singly linked list

   ข. จงเขียนอัลกอริทึมในการแทรกโหนดใหม่เข้าไปใน Singly linked list จากเดิม PACAAL เป็นคำว่า PASCAAL
   กำหนดให้ New , Prv , Pt เป็นตัวแปร pointer ที่ชี้ไปยัง โครงสร้างของ node ที่ประกาศไว้ในข้อ ก. และชี้อยู่ในตำแหน่งดังภาพต่อไปนี้

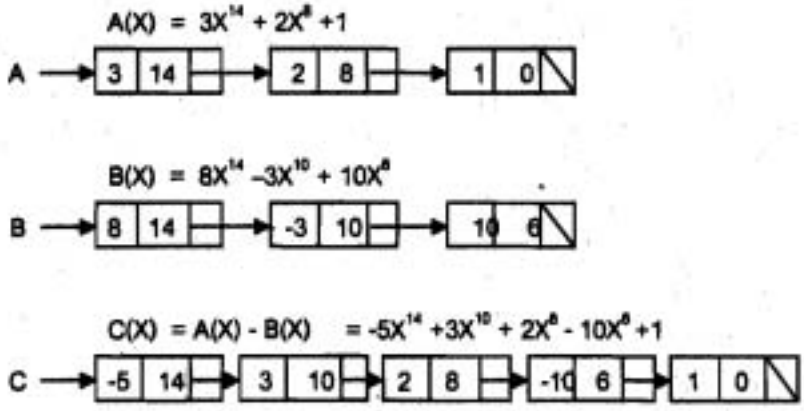


   ค. จงเขียนอัลกอริทึมในการลบโหนดใน Singly linked list
   จากเดิม PASCAAL เป็น PASCAL



   ง. จงเขียน Procedure Lqinsert (item : char ; var front ,rear : Nodeptr);
   เพื่อนำข้อมูลในที่นี้คือตัวอักขระ 1 ตัวอักษร เก็บใน Linked queue

11. กำหนด singly linked list ซึ่งนำมาประยุกต์ใช้แทน polynomial functions ดังต่อไปนี้



ก. จงประกาศโครงสร้างข้อมูลของ node ใน singly linked lists ซึ่งใช้แทน polynomial functions และอธิบายด้วยว่า field ต่างๆหมายถึงอะไร

ข. จงอธิบายถึงแนวความคิดในการลบ polynomial functions โดยใช้ singly linked lists โดยอธิบายมาให้เข้าใจอย่างเป็นขั้นตอน

ค. จงสร้างคลาสที่แทนการทำงานของสมการโพลิโนเมียล และสามารถปฏิบัติงานกับฟังก์ชันต่างๆของคลาสนี้ได้