บทที่ 12
ลิสต์เชื่อมโยงและต้นไม้ทวิภาค
(Linked List and Binary Trees)

ปัญหาของการใช้โครงสร้างข้อมูลแถวลำดับจัดเก็บข้อมูลประเภทต่าง ๆ ก็คือ ไม่สามารถจะเพิ่มหรือลดขนาดของแถวลำดับขณะที่กำลังกระทำการโปรแกรมได้ ทั้งนี้เพราะว่าตัวแปรแถวลำดับนั้นจะถูกจัดสรรเนื้อที่เมื่อโปรแกรมถูกแปลให้เท่ากับจำนวนช่องที่ถูกกำหนดไว้ในคำสั่งประกาศตัวแปรแถวลำดับ นอกจากปัญหาการจัดสรรเนื้อที่แบบสถิตย์ (static) ดังกล่าวแล้ว ยังมีความยุ่งยากในการจัดการกับโครงสร้างข้อมูลแถวลำดับเมื่อต้องการเพิ่มข้อมูลเข้าไปในแถวลำดับที่มีการเรียงลำดับข้อมูล นักเขียนโปรแกรมจะต้องเขียนขั้นตอนวิธีในการค้นหาตำแหน่งที่เหมาะสมในตาราง หลังจากนั้นจะต้องเลื่อนข้อมูลที่อยู่ ณ ตำแหน่งนั้น (สมมติว่าตำแหน่งนั้นมีข้อมูลเก็บอยู่) และข้อมูลอื่น ๆ ที่อยู่ถัดไปให้เลื่อนไปทางขวา 1 ช่องสำหรับข้อมูลทุกตัว เพื่อให้สามารถเพิ่มข้อมูลใหม่ในตำแหน่งที่เหมาะสมได้

การลบข้อมูลออกจากตารางก็จะมีปัญหาคล้ายกันกับการเพิ่มข้อมูล นั่นคือ จะต้องมีการเลื่อนข้อมูลที่อยู่ติดกันทางขวา มาแทนที่ข้อมูลที่ลบออก แล้วเลื่อนตัวที่อยู่ถัดไปมาแทนตำแหน่งที่ว่าง แล้วดำเนินการเช่นนี้ต่อไปเรื่อย ๆ จนกว่าข้อมูลทุกตัวจะถูกเก็บเรียงลำดับติดต่อกัน

12.1 การจัดสรรเนื้อที่แบบพลวัต

เพื่อแก้ปัญหาการจัดสรรเนื้อที่แบบสถิตย์ดังกล่าว ภาษา C ได้มีการจัดสรรเนื้อที่แบบพลวัต (dynamic memory allocation) ซึ่งกำหนดฟังก์ชันในคลังมาตรฐานให้ผู้ใช้สามารถเรียกใช้เพื่อขอเนื้อที่ขณะที่กำลังกระทำการโปรแกรม และขอคืนเนื้อที่ที่ไม่ต้องการใช้ เพื่อให้ตัวแปรอื่นสามารถนำไปใช้ได้ขณะที่กระทำการโปรแกรมเช่นกัน ฟังก์ชัน 2 ฟังก์ชันดังกล่าว คือ malloc( ) กับ free( )

12.1.1 ฟังก์ชัน malloc( )

ฟังก์ชันนี้จะจัดสรรเนื้อที่ให้กับตัวแปรด้วยขนาดที่เหมาะสมกับตัวแปรนั้น โดยที่ตัวแปรแต่ละประเภทจะใช้เนื้อที่ไม่เท่ากัน เช่น จำนวนเต็ม (int) จะถูกจัดสรรเนื้อที่ให้ 2 ไบต์ เป็นต้น

เมื่อฟังก์ชัน malloc( ) ถูกเรียกใช้ จะคืนค่าเป็นตัวชี้ที่ชี้ไปยังที่อยู่ที่จะจัดเก็บค่าของตัวแปรนั้น ถ้าไม่มีที่ว่างในหน่วยความจำพอตามที่ถูกขอ ฟังก์ชันจะคืนค่าเป็น null

รูปแบบ การเรียกใช้ฟังก์ชัน malloc( ) คือ

```
p   =   malloc  (size of (int));
```

ในที่นี้ p คือ ตัวชี้ที่ชี้ไปยังที่อยู่ในหน่วยความจำที่ถูกจัดสรรด้วยฟังก์ชัน malloc( ) และมีขนาดเท่ากับขนาดของข้อมูลประเภท int คือ 2 ไบต์

12.1.2 ฟังก์ชัน free( )

เมื่อตัวแปรไม่ต้องการใช้เนื้อที่ที่ถูกจัดสรรให้ด้วยฟังก์ชัน malloc() แล้ว สามารถจะคืนเนื้อที่กลับไปให้ระบบคอมพิวเตอร์เพื่อนำไปใช้งานอย่างอื่นได้ ด้วยการเรียกใช้ฟังก์ชัน free() ซึ่งมีรูปแบบดังนี้

```
free (p);
```

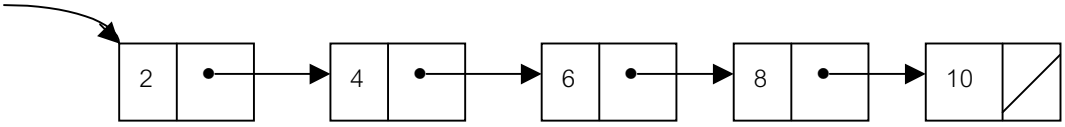
คือคำสั่งที่ขอคืนเนื้อที่ที่ถูกชี้โดยตัวชี้ p

นอกจากจะจัดเตรียมการจัดสรรเนื้อที่แบบพลวัตดังกล่าวแล้ว ภาษา C ยังได้กำหนดโครงสร้างข้อมูลที่จะทำการขอเนื้อที่ในหน่วยความจำเมื่อต้องการเพิ่มสมาชิกและคืนเนื้อที่เมื่อลบสมาชิกออกจากโครงสร้าง ซึ่งโครงสร้างข้อมูลที่มีคุณลักษณะดังกล่าว คือลิสต์เชื่อมโยงและต้นไม้

12.2 ลิสต์เชื่อมโยง

ลิสต์เชื่อมโยง(linked lists) เป็นโครงสร้างข้อมูลที่ซับซ้อน เพราะโครงสร้างข้อมูลลิสต์เชื่อมโยงจะประกอบด้วยกลุ่มของข้อมูลชนิดโครงสร้างที่มีการอ้างอิงตัวเอง (self-reference structure) เราเรียกโครงสร้างหรือสมาชิกของลิสต์เชื่อมโยงว่าบัพ(node) โดยแต่ละบัพจะต้องประกอบด้วยหนึ่งเขตข้อมูลที่เก็บตัวชี้ที่ชี้ไปยังบัพอื่นที่อยู่ในตำแหน่งลำดับถัดไปในลิสต์เชื่อมโยงนั้น

เราสามารถสร้างแผนภาพแสดงลิสต์เชื่อมโยงที่มีสมาชิก 5 บัพ โดยแต่ละบัพจะจัดเก็บจำนวนเต็มเรียงตามลำดับได้ดังรูปที่ 12.1 ต่อไปนี้



รูปที่ 12.1 ลิสต์เชื่อมโยงของจำนวนเต็ม

ในรูปที่ 12.1 นี้ ใช้ลูกศรแทนตัวชี้จากบัพที่อยู่ก่อนไปยังบัพถัดไป เช่น จากบัพที่เก็บจำนวน ‘2’ ไปยังบัพที่เก็บจำนวน ‘4’ และจากบัพที่เก็บจำนวน ‘4’ ไปยังบัพที่เก็บจำนวน ‘6’ แล้วต่อไปเรื่อย ๆ จนถึงบัพสุดท้ายที่เก็บจำนวน ‘10’ ซึ่งไม่ได้ชี้ไปที่บัพอื่นอีกแล้ว จึงกำหนดให้ค่าตัวชี้เป็น null ซึ่งแทนในแผนภาพด้วย ⊘

นอกจากนี้จะสังเกตเห็นว่าที่บัพแรก คือ บัพที่เก็บจำนวนเต็ม ‘2’ นั้นจะถูกชี้ด้วยตัวชี้อีกตัวหนึ่ง ซึ่งในการจัดการโครงสร้างข้อมูลลิสต์เชื่อมโยงนั้น จะต้องมีตัวชี้ไปยังบัพแรกเสมอ และนักเขียนโปรแกรมจะท่องเข้าไปในโครงสร้างลิสต์เชื่อมโยงได้โดยเริ่มต้นจากตัวชี้ไปที่บัพแรกหรือหัวลิสต์ แล้วท่องตามตัวชี้ไปยังบัพถัดไป ไปเรื่อย ๆ จนกว่าจะถึงบัพสุดท้าย

12.2.1 การประกาศตัวแปรลิสต์เชื่อมโยง

เนื่องจากบัพแต่ละบัพของลิสต์ในรูปที่ 12.1 ประกอบด้วย 2 เขตข้อมูล จึงต้องกำหนดให้แต่ละบัพเป็นประเภทข้อมูลโครงสร้าง ดังนี้

```
struct node {
        int             value;
        struct node     *next;          / * ตัวชี้ไปยังบัพถัดไป*/
        };
```

จากการประกาศข้างต้นได้กำหนดให้ node เป็นประเภทข้อมูลโครงสร้าง ซึ่งประกอบด้วย 2 เขตข้อมูล คือ

1. value เป็นเขตข้อมูลที่จัดเก็บข้อมูลชนิดจำนวนเต็ม
2. next เป็นเขตข้อมูลที่จัดเก็บข้อมูลชนิดตัวชี้ ไปยังโครงสร้าง node

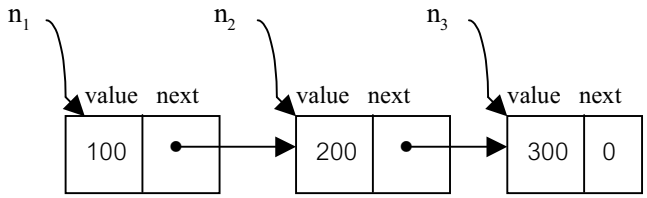
ตัวอย่างที่ 12.1 แสดงลิสต์เชื่อมโยงที่มีสมาชิก 3 บัพ

```
/* Llist.c        Program to illustrates linked lists */
#include <stdio.h>
struct list {
        int     value;
        struct list *next;
        };
main()
{
        struct list     n1, n2, n3;
        n1.value= 100;
        n2.value= 200;
        n3.value= 300;
        n1.next = &n2;
        n2.next= &n3;
        n3.next= 0;
        printf("%d\n", n2.next->value);
        return 0;
}
```

ผลลัพธ์ที่ได้

```
300
```

เราสามารถจำลองลิสต์ในตัวอย่างที่ 12.1 ด้วยแผนภาพในรูปที่ 12.2 ดังนี้



รูปที่ 12.2 แผนภาพลิสต์

ในที่นี้บัพที่อยู่ต่อจากบัพ n_2 คือ บัพ n_3 ตามคำสั่ง (n_2.next) = &n_3; ดังนั้น ผลลัพธ์ของโปรแกรมที่สั่งให้พิมพ์ค่าข้อมูลที่เก็บอยู่ที่เขตข้อมูล 'value' ของบัพ n_3 คือ

นิพจน์ (n_2.next ->value) จึงได้ผลลัพธ์เป็น '300'

ตัวอย่างที่ 12.2 โปรแกรม TravList.c ต่อไปนี้แสดงการท่องเข้าไปในลิสต์ที่มีตัวชี้ list_pointer ชี้ไปยังหัวลิสต์ โปรแกรมจะพิมพ์ค่าข้อมูลที่เก็บอยู่ที่เขตข้อมูล value ในแต่ละบัพ แล้วท่องไปยังบัพถัดไปตามตัวชี้ next จนกระทั่งถึงบัพสุดท้าย

```
/* TravLlist.c   Program to illustrates linked lists */
#include <stdio.h>
struct list {
        int       value;
        struct list *next;
        };

main()
{
        struct list       n1, n2, n3, n4;
        struct list       *list_pointer    = &n1;
        n1.value= 100;
        n1.next           = &n2;
        n2.value          = 200;
        n2.next           = &n3;
        n3.value          = 300;
        n3.next           = &n4;
        n4.value          = 400;
        n4.next           = 0;

        while ( list_pointer      != 0 )
        {
                printf("%d\n", list_pointer->value);
                list_pointer      = list_pointer->next;
        }
        return 0;
}
```

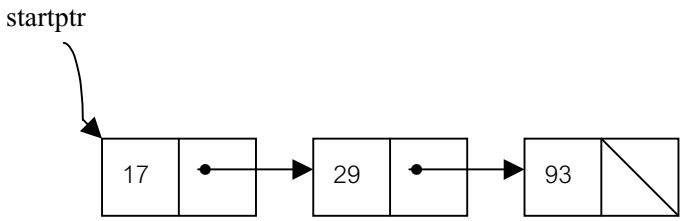
ผลลัพธ์ที่ได้ คือ

```
100
200
300
400
```

12.2.1 การใช้งานในลิสต์เชื่อมโยง

โดยปกติแล้ว นักพัฒนาโปรแกรมจะใช้โครงสร้างลิสต์เชื่อมโยง ก็ต่อเมื่อจำนวนข้อมูลที่ต้องใช้ในการทำงานไม่แน่นอน อาจจะมีการเพิ่มข้อมูลเข้าหรือนำข้อมูลออกจากลิสต์ เพราะว่าลิสต์เป็นโครงสร้างแบบพลวัต ดังนั้น ความยาวของลิสต์หรือจำนวนสมาชิกในลิสต์จะเพิ่มขึ้นหรือลดลงตามความจำเป็นในการใช้งานจริงเท่านั้น นอกจากนี้เรายังสามารถจัดการให้บัพในลิสต์มีการเรียงลำดับตามค่าข้อมูลในเขตข้อมูลใดเขตข้อมูลหนึ่ง เช่น เขตข้อมูล 'value' ของลิสต์ในรูปที่ 12.2

จากที่กล่าวมาแล้วว่าแถวลำดับที่ข้อมูลต้องเรียงลำดับนั้นจะเกิดความยุ่งยากในการจัดการเมื่อต้องการเพิ่มหรือลดข้อมูล แต่ข้อดีของแถวลำดับ ก็คือ ข้อมูลจะถูกจัดเก็บในเนื้อที่ที่วางเรียงติดต่อกัน เมื่อเป็นเช่นนี้ทำให้การเข้าถึงสมาชิกของแถวลำดับสามารถทำได้ทันที เพราะเราสามารถคำนวณตำแหน่งที่อยู่ของสมาชิกแถวลำดับได้จากดัชนีที่บอกเลขที่แถวและเลขที่สดมภ์ได้อย่างรวดเร็ว แต่สำหรับลิสต์เชื่อมโยงนั้นบัพสองบัพที่จัดเก็บค่าข้อมูลที่อยู่ต่อเนื่องกัน อาจจะถูกจัดเก็บในหน่วยความจำ ณ ตำแหน่งเลขที่อยู่ที่ไม่ต่อเนื่องกันก็ได้ เช่น จากรูปที่ 12.3 จะเห็นว่าบัพที่เก็บค่าข้อมูล 29 เป็นสมาชิกที่อยู่ต่อจากบัพที่เก็บค่าข้อมูล 17 แต่บัพ 2 บัพนี้อาจจะถูกจัดเก็บ ณ ตำแหน่งที่ห่างกัน โดยมีตัวเชื่อมโยงที่เป็นตัวบอกว่าบัพที่อยู่ต่อจากบัพ 17 นั้นคือบัพ 29



รูปที่ 12.3 ลิสต์เชื่อมโยง

ตัวอย่างที่ 12.3 ต่อไปนี้เป็นโปรแกรมแสดงการจัดการกับโครงสร้างลิสต์แบบเชื่อมโยงที่จัดเก็บข้อมูลตัวอักขระ ดังนั้น การประกาศบัพแต่ละบัพเป็นดังนี้

```
struct node {
        char        data;
        struct node *nextptr;
        };
```

ในที่นี้ nextptr เป็นตัวชี้ที่ชี้ไปยังบัพถัดไปเพื่อความสะดวกในการใช้งาน เราจะกำหนดให้ 'LISTNODEPTR' เป็นประเภทข้อมูลชนิดใหม่ ซึ่งมีลักษณะเป็นตัวชี้ไปยังโครงสร้าง node ที่ประกอบด้วย 2 เขตข้อมูลดังกล่าวโดยใช้คำสั่ง 'typedef' ดังนี้

```
struct node {
        char    data;
        struct node  *nextptr;
        };
typedef struct node    LISTNODE;
typedef LISTNODE    *LISTNODEPTR;
```

คำสั่ง typedef จะคล้ายกันกับคำสั่ง #define คือ ประกาศให้ตัวระบุมีค่าหรือมีคุณลักษณะตามที่เราต้องการ นั่นคือ เราสามารถนำส่วนประกาศ typedef นี้ไปไว้ในแฟ้มที่มีส่วนขยายเป็น '.h' ได้ แล้วใช้คำสั่ง #include <filename> เพื่อให้สามารถนำแฟ้มข้อมูลนี้ไปใช้งานในโปรแกรมต่าง ๆ ได้ แต่ข้อแตกต่างระหว่าง 2 คำสั่งนี้คือ typedef ใช้สำหรับกำหนดประเภทข้อมูลชนิดใหม่ และจะถูกกระทำการโดยคอมไพเลอร์ แต่ #define นั้นจะถูกดำเนินการโดยตัวประมวลผลก่อน C

```
/* Llist3.c
 *              Operating and maintaining a list
 */

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

/* definition of a node */
struct node {
        char    data;
        struct node *nextptr;
        };

typedef struct node    LISTNODE;
typedef LISTNODE    *LISTNODEPTR;

void    insert( LISTNODEPTR *, char );
char    delete( LISTNODEPTR *, char );
int     isempty( LISTNODEPTR );
void    print_list( LISTNODEPTR);
void    instructions(void);
```

```
main()
{
        LISTNODEPTR         startptr        = NULL;
        int     choice = 0;
        char    item;
        instructions();
        while (choice != 3)
        {
                printf("? ");
                scanf("%d", &choice);
                switch (choice)
                {
                        case 1:
                                printf("Enter a character: ");
                                scanf("\n%c", &item);
                                insert(&startptr, item);
                                print_list(startptr);
                                break;
                        case 2:
                                if (!isempty(startptr))
                                {
                                        printf("Enter character to be deleted: ");
                                        scanf("\n%c", &item);
                                if ( delete( &startptr, item ))
                                        {
                                                printf("\'%c\' deleted.\n", item);
                                                print_list(startptr);
                                        }
                                        else
                                                printf("\'%c\' not found.\n\n");
                                }
                                else
                                        printf("List is empty.\n\n");
                                break;
                }
        }
        printf("End of run. \n");
        return 0;
}
```

```
/* print the instructions */
void instructions(void)
{
        printf("%s\n %s\n %s\n %s\n",
                "Enter your choice:",
                "       1       to insert an element into the list.",
                "       2       to delete an element from the list.",
                "       3       to end.");
}

/* insert a new value into the list in sorted order */
void insert( LISTNODEPTR  *sptr, char value )
{
        LISTNODEPTR    newptr, previousptr, currentptr;
        newptr = malloc( sizeof( LISTNODE ) );

        if (newptr)
        {
                newptr->data    = value;
                newptr->nextptr         = NULL;
                previousptr             = NULL;
                currentptr              = *sptr;
                while ( currentptr != NULL  && value > currentptr->data )
                {
                        previousptr     = currentptr;
                        currentptr      = (currentptr)->nextptr;
                }

                if ( previousptr == NULL )
                {
                        newptr->nextptr         = *sptr;
                        *sptr                   = newptr;
                }
                else
                {
                        previousptr->nextptr    = newptr;
                        newptr->nextptr                         = currentptr;
                }
        }
        else
                printf("%c not inserted. No memory available.\n", value);
}
```

```
/* delete a list element */
char delete ( LISTNODEPTR *sptr, char value )
{
        LISTNODEPTR                 previousptr, currentptr, tempptr;
        if ( value == (*sptr)->data )
        {
                tempptr         = *sptr;
                *sptr   = (*sptr)->nextptr;
                free(tempptr);
                return value;
        }
        else
        {
                previousptr     = *sptr;
                currentptr      = (*sptr)->nextptr;
                while ( currentptr != NULL && currentptr->data != value )
                {
                        previousptr     = currentptr;
                        currentptr      = currentptr->nextptr;
                }
                if (currentptr)
                {
                        tempptr                         = currentptr;
                        previousptr->nextptr   = currentptr->nextptr;
                        free(tempptr);
                        return( value );
                }
        }
        return '\0';
}
```

```
/* return 1 if the list is empty, 0 otherwise */
int  isempty(LISTNODEPTR sptr)
{
        return !sptr;
}

/* print the list */
void print_list( LISTNODEPTR currentptr)
{
        if (!currentptr)
                printf( "List is empty.\n\n");
        else
        {
                printf("The list is:\n");

                while (currentptr)
                {
                        printf("%c --> ", currentptr->data);
                        currentptr      = currentptr->nextptr;
                }
                printf("NULL\n\n");
        }
}
```

ผลลัพธ์ที่ได้ คือ

```
Enter your choice:
        1       to insert an element into the list.
        2       to delete an element from the list.
        3       to end.
? 1
Enter a character: A
The list is:
A --> NULL

? 1
Enter a character: B
The list is:
A --> B --> NULL

? 2
Enter character to be deleted: B
'B' deleted.
The list is:
A --> NULL

? 2
Enter character to be deleted: A
'A' deleted.
List is empty.

? 3
End of run.
```

โปรแกรมในตัวอย่างที่ 12.3 ทำหน้าที่ที่สำคัญ 2 อย่างคือ เพิ่มข้อมูลใหม่และลบข้อมูลออกจากลิสต์ โดยตัวอักขระในลิสต์จะถูกเรียงลำดับจากน้อยไปหามาก

12.2.2 การเพิ่มข้อมูลใหม่

โปรแกรมได้จัดเตรียมฟังก์ชัน 'insert' เพื่อเพิ่มตัวอักขระเข้าไปในลิสต์ ซึ่งขั้นตอนการดำเนินงานตามลำดับเป็นดังนี้

(1) สร้างบัพใหม่ โดยเรียกใช้ฟังก์ชัน malloc ซึ่งฟังก์ชันนี้จะจัดสรรเนื้อที่ให้ 1 บัพ และคืนค่าเป็นตัวชี้ไปยังบัพใหม่นั้น เก็บไว้ที่ตัวแปร 'newptr' แล้วกำหนดให้เขตข้อมูล 'data' เก็บตัวอักขระใหม่ด้วยคำสั่ง

```
newptr -> data  =  value;
```

และกำหนดเขตข้อมูล 'nextptr' ซึ่งเก็บค่าตัวชี้ไปยังบัพถัดไปเป็น NULL ดังนี้

```
newptr -> newptr  =  null;
```

(2) กำหนดค่าเริ่มต้นของตัวชี้ previousptr = null และค่าตัวชี้ currentptr = *sptr (ซึ่งเป็นตัวชี้ไปยังหัวลิสต์) โดยตัวชี้ previousptr กับ currentptr นี้เป็นตัวชี้ที่ใช้เพื่อเก็บตำแหน่งของบัพก่อนหน้า และ บัพที่อยู่จากบัพใหม่ที่จะเพิ่มเข้าไปในลิสต์

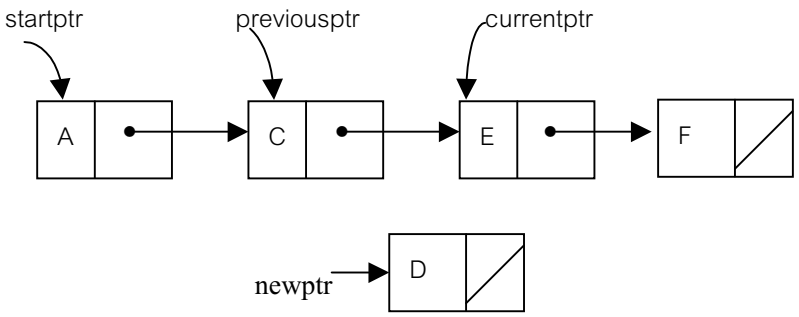
(3) ในขณะที่ currentptr != null และค่าข้อมูลที่จะเพิ่มเข้าไปมีค่ามากกว่า currentptr - > data (ตัวอักขระที่เก็บอยู่ที่บัพที่ถูกชี้โดย currentptr) โปรแกรมจะทำการปรับเลื่อนตัวชี้ currentptr กับ previousptr ไปยังบัพถัดไปทางขวาเรื่อย ๆ จนกระทั่งพบตำแหน่งที่จะเพิ่มบัพใหม่จึงหยุด

(4) ถ้า previousptr == null แสดงว่าลิสต์ว่างและบัพนี้เป็นบัพแรกที่จะเพิ่มเข้าไปที่หัวลิสต์จึงกำหนดให้ newptr -> nextptr = *sptr; (บัพใหม่ถูกโยงไปยังบัพแรกในลิสต์) และให้ *sptr = newptr; (*sptr ชี้ไปยังบัพใหม่)

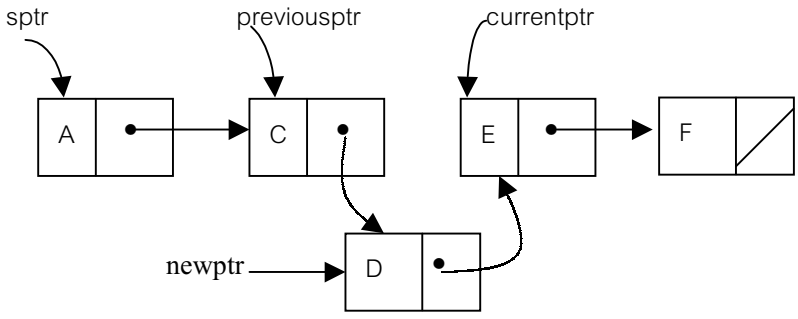
ถ้า previousptr != null แสดงว่าบัพใหม่จะถูกเพิ่มเข้าไปตรงกลางลิสต์ ซึ่งในกรณีนี้จะมีการปรับตัวชี้ 2 ตัว ดังนี้

```
previousptr -> nextptr = newptr;
newptr -> nextptr      = currentptr;
```

จากรูปที่ 12.4 ต่อไปนี้ แสดงการปรับตัวชี้ เมื่อต้องการเพิ่ม 'D' เข้าไปในลิสต์



(a) ลิสต์ก่อนการเพิ่มบัพ



(b) แสดงการเพิ่มบัพ D เข้าไประหว่างบัพ C กับ E และมีการปรับเส้นโยง 2 เส้น คือ จากบัพ C ไปยังบัพ D และจากบัพ D ไปยังบัพ E

รูปที่ 12.4 แสดงการเพิ่มบัพในลิสต์เชื่อมโยง

12.2.3 การลบบัพ

ฟังก์ชัน 'delete' ในโปรแกรมจะรับค่าพารามิเตอร์ 2 ตัว คือ ตัวชี้ไปยังหัวของลิสต์ และตัวอักขระที่ต้องการจะลบ ซึ่งขั้นตอนในการลบบัพเป็นดังนี้

(1) ถ้าตัวอักขระที่ต้องการจะลบอยู่ที่บัพแรกในลิสต์นั้น โปรแกรมดำเนินการดังนี้

```
tempptr      = *sptr;      /* ให้ tempptr ชี้ไปยังบัพที่จะคืนเนื้อที่ */
*sptr = (*sptr) -> nextptr;  /* *sptr ชี้ไปยังบัพที่สองในลิสต์ */
free (tempptr);            /* คืนเนื้อที่ไม่ใช้ */
```

(2) ถ้าตัวอักขระที่ต้องการจะลบไม่ได้อยู่ที่บัพแรกจะดำเนินการดังนี้

```
previousptr = *sptr;
currentptr  = (*sptr) -> nextptr;
```

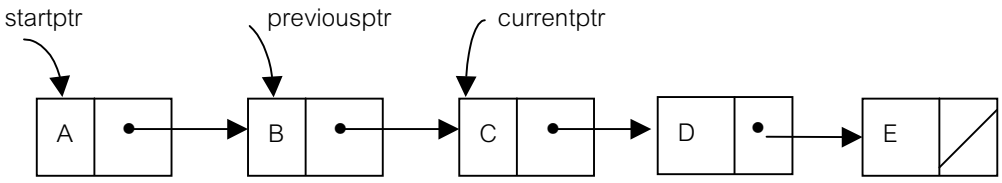
(3) ขณะที่ currentptr != null และตัวอักขระต้องการจะลบไม่อยู่ในบัพที่ถูกชี้โดย currentptr (currentptr ->data != value)

ให้ปรับค่าของ currentptr กับ previousptr ไปเรื่อย ๆ โดยกำหนดให้

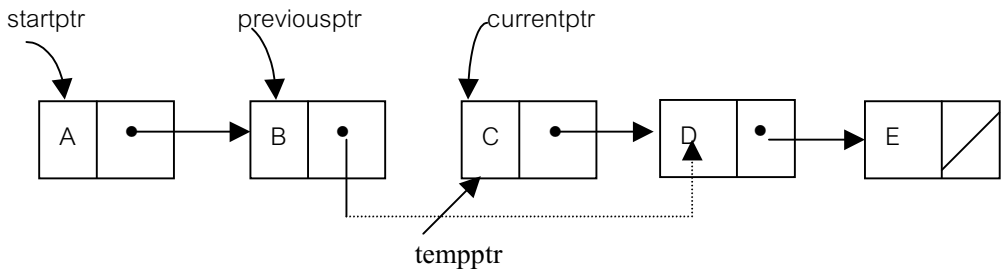
```
previousptr  =  currentptr
currentptr   =  currentptr -> nextptr;
```

(4) ถ้า currentptr != null แสดงว่าตัวอักขระที่จะลบอยู่ที่บัพที่ currentptr ชี้อยู่ โปรแกรมดำเนินการดังนี้

```
tempptr                  = currentptr;
previousptr -> nextptr = currentptr -> nextptr;
free(tempptr);
```



(a) ลิสต์ก่อนการลบบัพ



(b) ลิสต์หลังการปรับเปลี่ยนตัวชี้

รูปที่ 12.5 แสดงการลบบัพ ‘ C’ ซึ่งอยู่ตรงกลางออกจากลิสต์

นอกจากฟังก์ชัน insert กับ delete ดังกล่าวแล้วยังมีฟังก์ชัน print_list ซึ่งใช้แสดงสมาชิกของลิสต์ที่เรียงตามลำดับจากหัวลิสต์จนถึงบัพสุดท้าย โดยใช้ currentptr เป็นตัวชี้บัพที่กำลังถูกแวะผ่านอยู่ ซึ่งถ้า currentptr != null โปรแกรมจะพิมพ์ตัวอักขระในบัพนั้นออกมา แล้วปรับให้ currentptr ชี้ไปยังบัพถัดไปด้วยคำสั่ง

```
currentptr = currentptr -> nextptr;
```

12.3 ต้นไม้ทวิภาค

ต้นไม้ทวิภาค (binary trees) เป็นโครงสร้างข้อมูลชนิดพิเศษของโครงสร้างลิสต์เชื่อมโยง โดยบัพแต่ละบัพของต้นไม้ทวิภาคจะมีส่วนประกอบ 3 ส่วน หรือ 3 เขตข้อมูล คือ

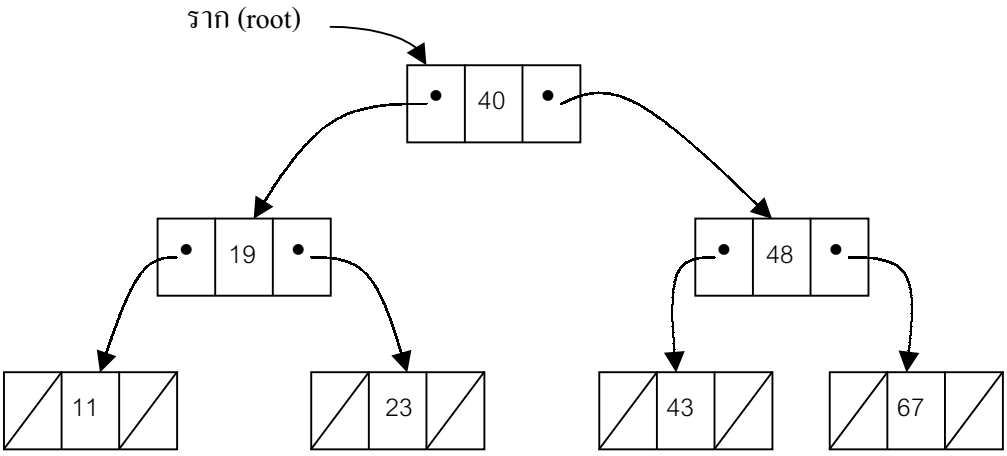
1. ส่วนที่ใช้เก็บข้อมูล
2. ตัวชี้ที่ชี้ไปที่บัพที่อยู่ก่อนหน้าหรือบัพที่อยู่ทางซ้าย
3. ตัวชี้ที่ชี้ไปยังบัพที่อยู่ถัดไปหรือบัพที่อยู่ไปทางขวา

เนื่องจากต้นไม้ทวิภาคอาจจะถูกเพิ่มบัพใหม่เข้าไปทางซ้ายหรือทางขวาก็ได้ ดังนั้น บัพแรกสุดจะถูกเรียกว่า ราก(root) และการท่องเข้าไปในโครงสร้างต้นไม้ทวิภาคนี้จะเริ่มต้นเข้าจากรากเสมอ หลังจากนั้นก็จะท่องตามตัวชี้ที่ชี้ไปยังบัพอื่นซึ่งอาจจะเป็นทางซ้ายหรือขวาก็ได้

คำศัพท์เฉพาะของต้นไม้ทวิภาค มีดังนี้

1. บัพที่อยู่ทางซ้ายของบัพ A ใด ๆ จะถูกเรียกว่า ลูกทางซ้าย (left son)
2. บัพที่อยู่ทางขวาของบัพ A ใด ๆ จะถูกเรียกว่าลูกทางขวา (right son)
3. บัพใด ๆ ที่ไม่มีลูกทั้งทางซ้ายและทางขวาจะเรียกว่าใบ (leaf)
4. ต้นไม้ย่อยทางซ้าย (left subtree) และต้นไม้ย่อยทางขวา (right subtree) ของบัพ A คือต้นไม้ที่มีลูกทางซ้ายและลูกทางขวาของบัพ A เป็นราก

ตัวอย่างต้นไม้ทวิภาคปรากฏในรูปที่ 12.6



รูปที่ 12.6 ต้นไม้ทวิภาค

จากรูปที่ 12.6 จะได้ว่า
รากของต้นไม้ คือ บัพ 40
ใบของต้นไม้ คือ บัพ 11, 23, 43 และ 67
ลูกทางซ้ายของบัพ 19 คือ บัพ 11
ลูกทางขวาของบัพ 19 คือ บัพ 23
ต้นไม้ย่อยทางขวาของ 40 คือ ต้นไม้ที่มีบัพ 48 เป็นราก

ต้นไม้ในรูปที่ 12.6 เป็นตัวอย่างต้นไม้ทวิภาคที่มีประโยชน์มากประเภทหนึ่ง ซึ่งเรียกว่า ต้นไม้ค้นหาทวิภาค (binary search tree) ต้นไม้ค้นหาทวิภาค เป็นต้นไม้ที่มีคุณลักษณะพิเศษ คือ ค่าของข้อมูลทุก ๆ ค่าข้อมูลในทุกบัพของต้นไม้ย่อยทางซ้ายของบัพ A จะน้อยกว่าค่าข้อมูลที่บัพ A และค่าข้อมูลในทุก ๆ บัพของต้นไม้ย่อยทางขวาของบัพ A จะมากกว่าค่าข้อมูลที่บัพ A

การประกาศโครงสร้างต้นไม้ทวิภาค จะคล้าย ๆ กับการประกาศโครงสร้างบัพของลิสต์ต่างกันตรงที่แต่ละบัพของต้นไม้จะประกอบด้วย 3 เขตข้อมูลดังนี้

เขตข้อมูลที่ 1 เก็บตัวชี้ที่ชี้ไปยังต้นไม้ย่อยทางซ้าย
เขตข้อมูลที่ 2 เก็บค่าข้อมูล
เขตข้อมูลที่ 3 เก็บตัวชี้ที่ชี้ไปยังต้นไม้ย่อยทางขวา

ตัวอย่างการประกาศโครงสร้างบัพของต้นไม้ที่ใช้เก็บค่าของจำนวนเต็มเป็นดังนี้

```
struct treenode {
        struct treenode *leftptr;
        int data;
        struct treenode *rightptr;
};
```

leftptr และ rightptr เป็นตัวชี้ที่ชี้ไปยังบัพที่เป็นลูกทางซ้ายและทางขวาตามลำดับ และเพื่อความสะดวกในการใช้งาน เราอาจใช้คำสั่ง typedef กำหนดประเภทข้อมูลใหม่ดังนี้

```
typedef struct treenode     TREENODE;
typedef TREENODE            *TREENODEPTR;
```

ในที่นี้ได้ประกาศให้ *TREENODEPTR* เป็นประเภทข้อมูลใหม่ที่เป็นตัวชี้ที่ชี้ไปยังบัพของต้นไม้ที่ประกอบด้วย 3 เขตข้อมูล

12.4 การท่องเข้าไปในต้นไม้ทวิภาค

วิธีการท่องเข้าไปในต้นไม้ทวิภาค(binary tree traversal) มี 3 แบบ คือ

1. แบบตามลำดับ (inorder)
2. แบบก่อนลำดับ (preorder)
3. แบบหลังลำดับ (postorder)

12.4.1 การท่องแบบตามลำดับ

(1) ท่องเข้าไปในต้นไม้ย่อยทางซ้าย
(2) แวะเยี่ยมบัพที่เป็นราก
(3) ท่องเข้าไปในต้นไม้ย่อยทางขวา

12.4.2 การท่องแบบก่อนลำดับ

(1) แวะเยี่ยมบัพที่เป็นราก
(2) ท่องเข้าไปในต้นไม้ย่อยทางซ้าย
(3) ท่องเข้าไปในต้นไม้ย่อยทางขวา

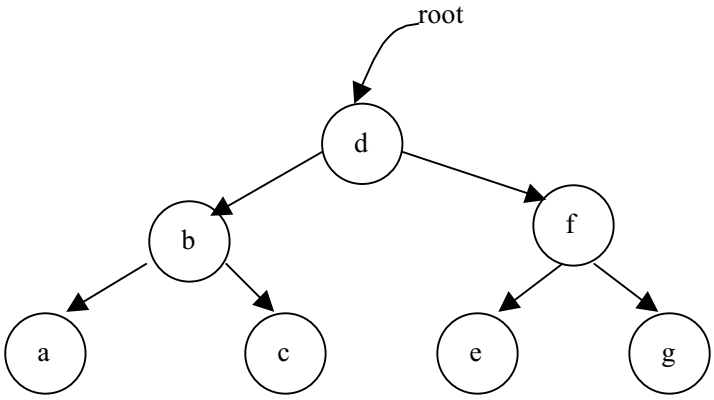
12.4.3 การท่องแบบหลังลำดับ

(1) ท่องเข้าไปในต้นไม้ย่อยทางซ้าย

(2) ท่องเข้าไปในต้นไม้ย่อยทางขวา

(3) แวะเยี่ยมบัพที่เป็นราก

ตัวอย่างที่ 12.4 กำหนดต้นไม้ทวิภาคดังรูปที่ 12.7



รูปที่ 12.7 ต้นไม้ทวิภาค

เมื่อท่องเข้าไปในต้นไม้ในรูปที่ 12.7 แบบต่าง ๆ แล้วพิมพ์ค่าข้อมูล ณ บัพนั้น จะได้ผลลัพธ์ดังนี้

วิธีการท่อง	**ผลลัพธ์**
• แบบตามลำดับ	a, b, c, d, e, f, g
• แบบก่อนลำดับ	d, b, a, c, f, e, g
• แบบหลังลำดับ	a, c, b, e, g, f, d

ตัวอย่างที่ 12.5 แสดงการสร้างต้นไม้ทวิภาค และ การท่องเข้าไปในต้นไม้

```
/* BinaryTree.c
 *              Create a binary tree and traverse it
 *              preordere, inorder, and postorder
 */
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
/* definition of a tree node */
struct treenode {
        struct treenode *leftptr;
        int     data;
        struct treenode *rightptr;
        };
typedef struct treenode         TREENODE;
typedef TREENODE    *TREENODEPTR;
void    insert_node( TREENODEPTR*, int );
void inorder(TREENODEPTR);
void    preorder( TREENODEPTR );
void    postorder( TREENODEPTR );

main()
{
        TREENODEPTR         rootptr= NULL;
        int i,item;
        /* insert random values between 1 and 15 in the tree */
        printf("\nThe number being placed in the tree are:\n");

        for (i=1; i <= 10; i++ )
        {
                item = rand() % 15;
                printf("%3d", item);
                insert_node(&rootptr, item);
        }
        printf("\n\nThe preorder traversal is:\n");
        preorder(rootptr);

        /* traverse the tree inorder */
        printf("\n\nThe inorder traversal is:\n");
        inorder(rootptr);

          /* traverse the tree postorder */
        printf("\n\nThe postorder traversal is:\n");
        postorder(rootptr);
        return 0;
}
```

```
void insert_node( TREENODEPTR *treeptr, int value)
{
        if  ( !*treeptr )
        {                                       /* treeptr is NULL */
                *treeptr        = malloc( sizeof( TREENODE ) );
        if ( *treeptr )
                {
                        ( *treeptr)->data       = value;
                        ( *treeptr)->leftptr    = NULL;
                        ( *treeptr)->rightptr   = NULL;
                        printf(" inserted.\n");
                }

                else
                        printf("%d not inserted.  No memory available.\n", value
);
                return;
        }
        else
                if ( value < ( *treeptr)->data )
                        insert_node( &((*treeptr)->leftptr), value);
                else
                        if ( value > ( *treeptr)->data )
                                insert_node( &((*treeptr)->rightptr), value);
                        else
                        {
                                printf("duplicated. \n");
                                return;
        }
}
void inorder(TREENODEPTR treeptr)
{

        if (treeptr)
        {
                inorder(treeptr->leftptr);
                printf("%3d", treeptr->data);
                inorder(treeptr->rightptr);
        }
}
```

```
void preorder(TREENODEPTR treeptr)
{
        if(treeptr)
        {
                printf("%3d",treeptr -> data);
                preorder(treeptr -> leftptr);
                preorder(treeptr -> rightptr);
        }
}
void postorder(TREENODEPTR treeptr)
{
        if (treeptr)
        {
                postorder(treeptr -> leftptr);
                postorder(treeptr -> rightptr);
                printf("%3d", treeptr -> data);
        }
}
```

ผลลัพธ์ที่ได้ คือ

```
The number being placed in the tree are:
  1 inserted.
 10 inserted.
  2 inserted.
 10duplicated.
  1duplicated.
  7 inserted.
  0 inserted.
 10duplicated.
 13 inserted.
  1duplicated.


The preorder traversal is:
  1  0 10  2  7 13

The inorder traversal is:
  0  1  2  7 10 13

The postorder traversal is:
  0  7  2 13 10  1
```

โปรแกรมในตัวอย่างที่ 12.5 เป็นโปรแกรมที่เพิ่มบัพของจำนวนเต็มเข้าในต้นไม้ค้นหาทวิภาค โดยใช้ฟังก์ชัน insert_node ซึ่งรับอาร์กิวเมนต์ 2 ตัว คือ ตัวชี้ไปยังรากของต้นไม้ และจำนวนเต็มที่ต้องการเก็บไว้ในโครงสร้างต้นไม้นั้น และ การเพิ่มบัพใหม่จะเพิ่มเฉพาะส่วนของใบเท่านั้น ขั้นตอนการเพิ่มบัพใหม่เป็นดังนี้

(1) ถ้า *treeptr == null ให้สร้างบัพใหม่โดยขอเนื้อที่จากการเรียกใช้ฟังก์ชัน malloc ซึ่งฟังก์ชันจะคืนค่าเป็นตัวชี้ไปให้ตัวแปร *treeptr แล้วนำข้อมูลจำนวนไปเก็บ แล้วกำหนดค่าตัวชี้ทั้งทางซ้ายและขวาเป็น null ดังคำสั่งต่อไปนี้

```
(*treeptr) -> data     =  value;
(*treeptr) -> leftptr  =  null;
(*treeptr) -> rightptr =  null;
```

(2) ถ้า treeptr != null และจำนวนเต็มที่ต้องการเพิ่มเข้ามาใหม่น้อยกว่าจำนวนที่เก็บอยู่ที่บัพ ณ ขณะนั้น (value < (*treeptr) -> data) โปรแกรมจะเรียกซ้ำฟังก์ชัน insert_node โดยส่งพารามิเตอร์เป็นตัวชี้ที่ชี้ไปที่ลูกทางซ้ายของบัพ ณ ขณะนั้น พร้อมกับจำนวนเต็มที่ต้องการเพิ่ม แต่ถ้าจำนวนเต็มที่ต้องการเพิ่มเข้ามาใหม่มากกว่าจำนวนที่เก็บอยู่ที่บัพ ณ ขณะนั้น โปรแกรมจะเรียกซ้ำฟังก์ชัน insert_node ด้วยพารามิเตอร์ที่เป็นตัวชี้ไปที่ลูกทางขวา พร้อมทั้งจำนวนที่ต้องการเพิ่ม

กระบวนการการเรียกซ้ำนี้กระทำไปเรื่อย ๆ จนกระทั่งพบว่า *treeptr เป็น null แล้วกลับไปทำงานที่ข้อ 1 แล้วทำการเพิ่มบัพใหม่

แบบฝึกหัด

1. จงตอบคำถามต่อไปนี้

(1) ฟังก์ชันที่ถูกเรียกใช้ เพื่อจัดสรรเนื้อที่หน่วยความจำให้กับตัวแปร คือ ฟังก์ชันใด

(2) ฟังก์ชันที่ถูกเรียกใช้ เพื่อคืนเนื้อที่ไม่ต้องการใช้กลับคืน คือ ฟังก์ชันใด

(3) บัพแรกของต้นไม้เรียกว่าอะไร

(4) บัพของต้นไม้ที่ไม่มีลูกทั้งทางซ้ายและขวาเรียกว่าอย่างไร

(5) ขั้นตอนวิธีในการท่องเข้าไปต้นไม้มีกี่แบบ อะไรบ้าง

2. ถ้ากำหนดโครงสร้าง gradenode ดังนี้

```
struct gradenode {
            char lastname[20];
            float grade;
            struct gradenode  *nextptr;
      };
typedef struct gradenode GRADENODE;
typedef GRADENODE * GRADENODEPTR
```

(1) สร้างตัวชี้ startptr ไปยังหัวลิสต์ และลิสต์เป็นลิสต์ว่าง

(2) สร้างบัพใหม่ซึ่งมีโครงสร้างเป็น GRADENODE และบัพนี้ถูกชี้โดยตัวชี้ newptr ซึ่งมีประเภทข้อมูลเป็น GRADENODEPTR หลังจากนั้นให้กำหนดค่าในเขตข้อมูล lastname เป็น "Jones" และเขตข้อมูล grade เป็น 91.5

(3) สมมติว่า ณ ขณะนี้ลิสต์ซึ่งถูกชี้โดย startptr ประกอบสมาชิก 2 บัพที่จัดเก็บข้อมูลของนักเรียน 2 คน ที่มีค่าในเขตข้อมูล lastname เป็น "Jones" กับ "Smith" และให้บัพเหล่านี้จัดเรียงตามลำดับตัวอักษรของนามสกุล

จงเขียนคำสั่งในการเพิ่มบัพใหม่เข้าไปในตำแหน่งที่เหมาะสมเมื่อกำหนด 'lastname' และ 'grade' ดังนี้

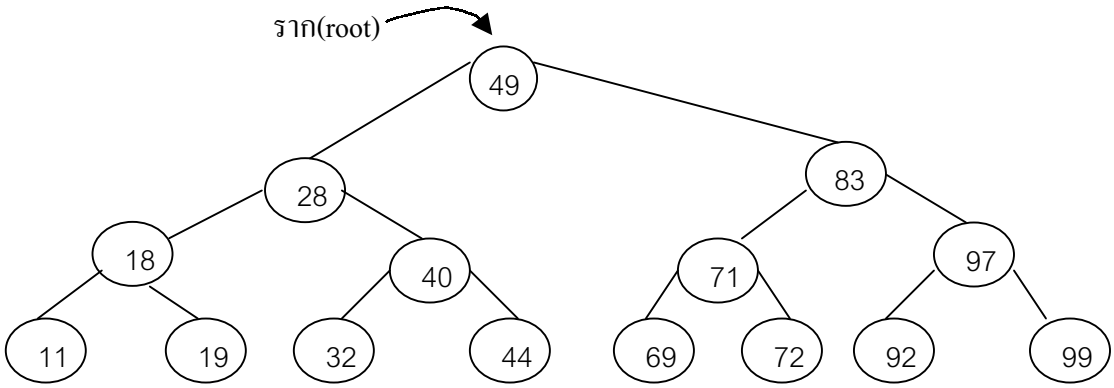
Lastname	grade
"Adams"	85.0
"Thompson"	73.5
"Pritchard"	66.5

ให้ใช้ตัวชี้ previousptr, currentptr และ newptr ชี้ไปยังบัพในลิสต์ในลักษณะเช่นเดียวกันกับในโปรแกรม ตัวอย่างที่ 12.5

(4)ใช้คำสั่งวงวน 'while' เพื่อเพิ่มข้อมูลในแต่ละบัพ โดยใช้ตัวชี้ currentptr เพื่อท่องเข้าไปในลิสต์

(5) ใช้คำสั่งวงวน 'while' เพื่อลบทุก ๆ บัพออกจากลิสต์โดยใช้ตัวชี้ currentptr กับ tempptr ในการท่องเข้าไปในลิสต์ และการคืนเนื้อที่

3. กำหนดต้นไม้ค้นหาทวิภาคดังรูปที่ 12.8 ให้เขียนผลลัพธ์ที่ได้จากการท่องเข้าไปในต้นไม้นี้แบบตามลำดับ ก่อนลำดับ และหลังลำดับ



รูปที่ 12.8 ต้นไม้ค้นหาทวิภาค

4. จงเขียนโปรแกรมที่จะเพิ่มจำนวนเต็ม 25 จำนวนโดยแต่ละจำนวนได้มาจากฟังก์ชันเลขสุ่ม (rand) และตัวเลขที่สุ่มได้นี้ต้องอยู่ระหว่าง 0-100 เข้าไปในลิสต์แบบตามลำดับ และเมื่อโปรแกรมทำงานเสร็จสิ้นให้แสดงผลบวกของจำนวนทั้งหมดที่มีอยู่ในลิสต์ด้วย

5. จงปรับโปรแกรมในตัวอย่างที่ 12.5 เพื่อให้ต้นไม้ค้นหาทวิภาค สามารถรับค่าข้อมูลที่ซ้ำกันกับค่าที่ถูกเก็บอยู่ในบัพของต้นไม้นั้นได้