資料結構

17
CHAPTER

17.1 串列資料結構

串列資料結構（linked list）是一種動態的線性資料結構。所謂**「動態結構」**表示它是機動性的增加或減少記憶體空間以配合資料的新增或刪除，所謂**「線性結構」**表示它是以類似一維陣列的排列方式來管理資料。

固定長度陣列是「靜態結構」，通常程式設計人員預設最大可能使用的長度給固定長度陣列，但實際使用時可能只使用 30% 的空間，這就造成 70% 記憶體空間的浪費。**串列資料結構是「動態結構」**，設計程式時不必設定元素的最大長度，只需宣告元素的結構，而在程式執行時，配合資料的增減才配置或釋放記憶體空間，如此可以讓記憶體的使用發揮到最大功效。

17.1.1 插入串列資料

圖 17.1 是插入新資料到串列第一項的圖解。首先將新資料的 next 指標插入串列的第一項指標（newPtr->next = firstPtr），再移動第一項指標到新資料（firstPtr = newPtr）使新資料成為串列的第一項資料。

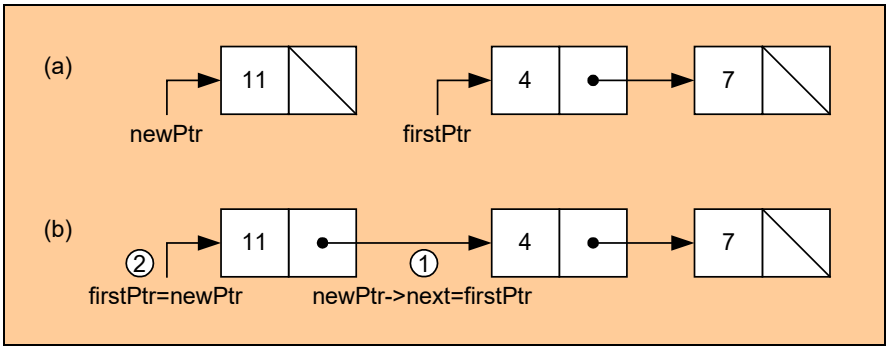

圖 17.1 插入第一項

下面範例是插入串列資料的第一項：首先定義類別型態的 Student 資料與結構型態的 link 資料，在 link 資料結構中包含 Student 型態變數 data 與 link 型態指標 *next。然後定義 linklist 類別，類別的資料成員包含 link 型態指標 *firstPtr 與 *lastPtr，而類別的成員函數包含 linklist 建立者與 addFront 成員函數。

在 addFront 函數中，先定義一個新的 link 指標 *newPtr，並將參數 obj 存入 newPtr 的 data 欄位，然後判斷串列中是否含有資料。若串列中沒有資料則令第一項指標 firstPtr 與最後一項指標 lastPtr 都指向新指標 newPtr，且令 lastPtr 的 next 指向 NULL。若串列中已經含有資料，則令新指標 newPtr 的 next 指標等於前一次的 firstPtr，再令第一項指標 firstPtr 等於新指標 newPtr。

```
class Student {                                  //定義 Student 類別資料
   int student_id;
   char student_name[40];
};

struct link {                                    //定義 link 結構資料
   Student data;
   link *next;
};

class linklist {                                 //定義串列資料類別
   link *firstPtr;
   link *lastPtr;
public:
   linklist() { firstPtr = lastPtr = NULL; }
   void addFront (Student obj)                   //插入第一項資料到串列中
   {
      link *newPtr = new link;                   //①newPtr 為新資料指標
```

```
        newPtr->data = obj;                    //②資料存入新指標的緩衝區
        if (firstPtr == NULL) {                //③若串列中沒有資料
           firstPtr = lastPtr = newPtr;        //④令頭尾指標指向新資料
           lastPtr->next = NULL;               //⑤最後項的 next 指標=0
        }                                      //   NULL 為結束識別碼
        else {                                 //⑥若串列中已有資料
           newPtr->next = firstPtr;            //⑦新資料 next 指向串列的頭
           firstPtr = newPtr;                  //⑧第一項指標等於新資料指標
        }
    }
};
```

圖 17.2 是插入新資料到串列最後項的圖解。首先將新資料指標插入串列的最後項的 next 指標（lastPtr->next = newPtr），再移動最後項指標到新資料（lastPtr = newPtr）使新資料成為串列的最後一項資料。

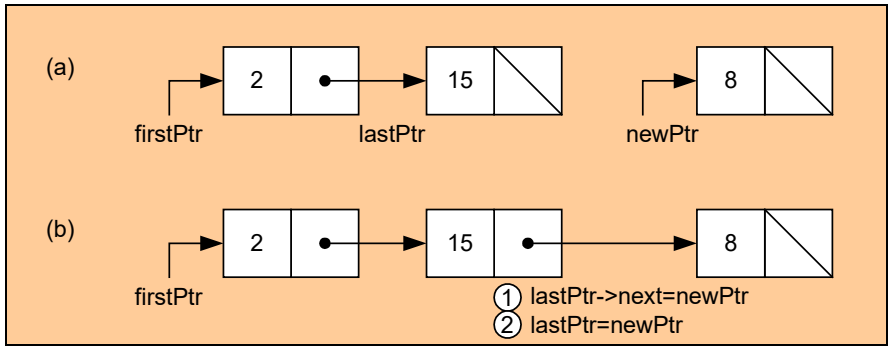

圖 17.2 插入最後一項

下面範例是插入串列資料的最後項：首先定義類別型態的 Student 資料與結構型態的 link 資料，在 link 資料結構中包含 Student 型態變數 data 與 link 型態指標 *next。然後定義 linklist 類別，類別的資料成員包含 link 型態指標 *firstPtr 與 *lastPtr，而類別的成員函數包含 linklist 建立者與 addBack 成員函數。

在 addBack 函數中，先定義一個新的 link 指標 *newPtr，並將參數 obj 存入 newPtr 的 data 欄位，然後判斷串列中是否含有資料。若串列中沒有資料則令第一項指標 firstPtr 與最後一項指標 lastPtr 都指向新指標 newPtr，且令 lastPtr 的 next 指向 NULL。若串列中已經含有資料，則令前一次的 lastPtr 的 next 指標與最後項指標 lastPtr 等於新指標 newPtr，再令 lastPtr 的 next 指標等於 NULL。

```
class Student {                          //自定 Student 資料型態
   int student_id;
   char student_name[40];
};

struct link {                            //定義 link 資料結構
   Student data;
   link *next;
};

class linklist {                         //定義串列資料類別
   link *firstPtr;
   link *lastPtr;
public:
   linklist() { firstPtr = lastPtr = NULL; }
   void addBack (Student obj)            //插入最後一項資料到串列中
   {
      link *newPtr = new link;           //①newPtr 為新資料指標
      newPtr->data = obj;                //②資料存入新指標的緩衝區
      if (firstPtr == NULL) {            //③若串列中沒有資料
         firstPtr = lastPtr = newPtr;    //④令頭尾指標指向新資料
         lastPtr->next = NULL;           //⑤最後項的 next 指標=0
      }
      else {                             //⑥若串列中已有資料
         lastPtr->next = newPtr;         //⑦最後項的 next 指向新資料
         lastPtr = newPtr;               //⑧最後項指標等於新資料指標
         lastPtr->next = NULL;           //⑨最後項的 next 指標=0
      }                                  //   NULL 為結束識別碼
   }
};
```

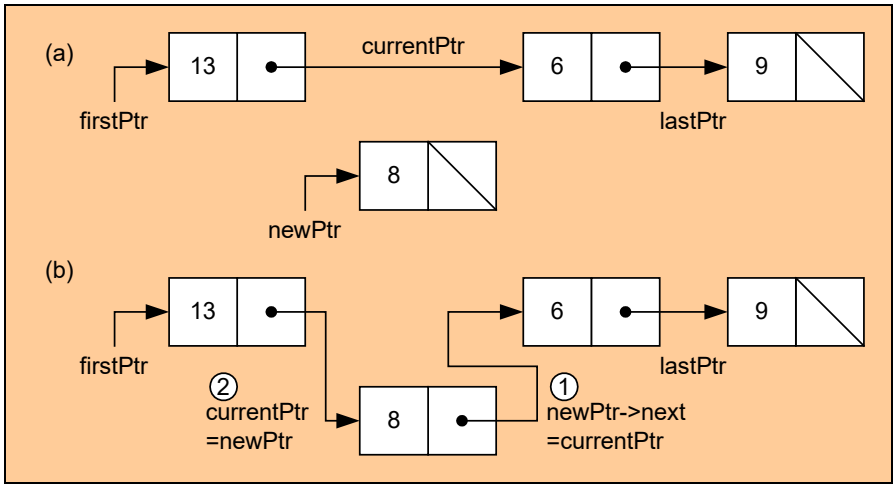

圖 17.3 插入中間項

圖 17.3 是插入新資料到串列中間項的圖解，首先必須先排序資料，再找到要插入的位置指標 currentPtr，然後令新資料的 next 指標指向 currentPtr，再令 currentPtr 等於新資料的指標。

上述插入中間項的方法比較複雜，因此可考慮先將資料附加到第一項或最後項，再利用排序功能來排列資料項。

程式 17-01：插入節點練習

```
1.  //檔案名稱：d:\C++17\C1701.cpp
2.  #include <iostream>
3.  using namespace std;
4.
5.  class Student                                  //自定 Student 資料
6.  {
7.     int student_id;
8.     char student_name[40];
9.  public:
10.    friend istream& operator >> (istream& in, Student& obj) {
11.       in >> obj.student_id >> obj.student_name;
12.       return in;
13.    }
14.    friend ostream& operator << (ostream& out, Student& obj) {
15.       out << obj.student_id << '\t' << obj.student_name;
16.       return out;
17.    }
18. };
19.
20. struct link //定義 link 資料結構
21. {
22.    Student data;                               //Student 型態資料
23.    link *next;                                 //link 型態指標
24. };
25.
26. class linklist                                 //串列資料類別
27. {
28.    link *firstPtr;                             //串列起始指標
29.    link *lastPtr;                              //串列結束指標
30. public:
31.    linklist() { firstPtr = lastPtr = NULL; }  //建立者
32.    void addFront (Student obj);                //宣告插入第一項原型
33.    void addBack (Student obj);                 //宣告插入最後項原型
34.    void showItem();                            //宣告顯示串列原型
35. };
36.
37. void linklist::addFront (Student obj)        //定義插入第一項函數
38. {
39.    link *newPtr = new link;
```

```
40.     newPtr->data = obj;
41.     if (firstPtr == NULL) {
42.        firstPtr = lastPtr = newPtr;
43.        lastPtr->next = NULL;
44.     }
45.     else {
46.        newPtr->next = firstPtr;
47.        firstPtr = newPtr;
48.     }
49.  }
50.
51.  void linklist::addBack (Student obj)       //定義插入最後項函數
52.  {
53.     link *newPtr = new link;
54.     newPtr->data = obj;
55.     if (firstPtr == NULL) {
56.        firstPtr = lastPtr = newPtr;
57.        lastPtr->next = NULL;
58.     }
59.     else {
60.        lastPtr->next = newPtr;
61.        lastPtr = newPtr;
62.        lastPtr->next = NULL;
63.     }
64.  }
65.
66.  void linklist::showItem()                  //定義顯示串列資料函數
67.  {
68.     link *currentPtr = firstPtr;
69.     while( currentPtr != NULL) {
70.        cout << currentPtr->data << endl;
71.        currentPtr = currentPtr->next;
72.     }
73.  }
74.
75.  int main(int argc, char** argv)
76.  {
77.     Student studata;                        //定義 Student 物件
78.     linklist ls;                            //定義 linklist 物件
79.     char n;
80.
81.     while(1) {
82.        cout << "1.插入第一項  2.插入最後項  0.結束  請選擇(1,2 或 0): ";
83.        cin >> n;
84.        switch (n) {
85.           case '1':
86.              cout << "請輸入學號與姓名：" ;
87.              cin >> studata;
88.              ls.addFront(studata);
89.              ls.showItem();
90.              break;
91.           case '2':
92.              cout << "請輸入學號與姓名：" ;
```

```
93.              cin >> studata;
94.              ls.addBack(studata);
95.              ls.showItem();
96.              break;
97.           case '0':
98.              return 0;
99.        }
100.       cout << endl;
101.    }
102.    return 0;
103. }
```

程式輸出

```
1.插入第一項  2.插入最後項  0.結束  請選擇(1,2 或 0): 1 Enter
請輸入學號與姓名：200 Carol Enter
200     Carol

1.插入第一項  2.插入最後項  0.結束  請選擇(1,2 或 0): 2 Enter
請輸入學號與姓名：300 David Enter
200     Carol
300     David

1.插入第一項  2.插入最後項  0.結束  請選擇(1,2 或 0): 1 Enter
請輸入學號與姓名：100 Arther Enter
100     Arther
200     Carol
300     David

1.插入第一項  2.插入最後項  0.結束  請選擇(1,2 或 0): 0 Enter
```

17.1.2 刪除串列資料

圖 17.4 是移除串列第一項資料的圖解。首先保存第一項指標（tempPtr = firstPtr），再移動第一項指標到下一項（firstPtr = firstPtr->next）使第二項成為串列的第一項，然後再刪除原來的第一項（delete tempPtr）。

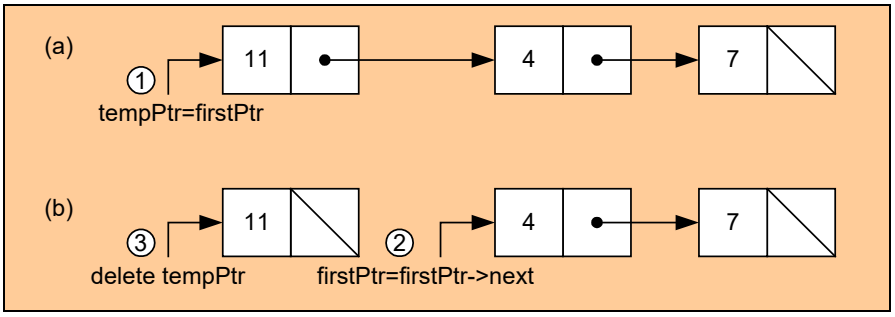

圖 17.4 刪除第一項

下面範例是刪除串列資料的第一項：首先定義類別型態的 Student 資料與結構型態的 link 資料，在 link 資料結構中包含 Student 型態變數 data 與 link 型態指標 *next。然後定義 linklist 類別，類別的資料成員包含 link 型態指標 *firstPtr 與 *lastPtr，而類別的成員函數包含 linklist 建立者與 delFront 成員函數。

在 delFront 函數中，先定義一個暫存的 link 指標 *tempPtr，然後將 firstPtr 指標存入 tempPtr 指標。接著將第二項指標 firstPtr->next 存入 firstPtr 指標，此時前一次的第二項已成為目前的第一項。最後刪除 tempPtr 指標。

```
class Student {                          //自定 Student 資料
   int student_id;
   char student_name[40];
};

struct link {                            //定義 link 資料結構
   Student data;
   link *next;
};

class linklist {                         //定義串列資料類別
   link *firstPtr;
   link *lastPtr;
public:
   linklist() { firstPtr = lastPtr = NULL; }
   void delFront()                       //定義刪除第一項函數
   {
      link *tempPtr = firstPtr;          //①保存第一項的指標
      firstPtr = firstPtr->next;         //②第一項指標向後移
      delete tempPtr;                    //③刪除原來第一項指標與資料
   }
};
```

圖 17.5 是移除串列最後項資料的圖解。首先保存最後項指標（tempPtr = lastPtr），利用迴圈找尋前一項的指標（currentPtr->next = lastPtr），再移動最後項指標到前一項（lastPtr = currentPtr），然後再刪除原來的最後項（delete tempPtr）。

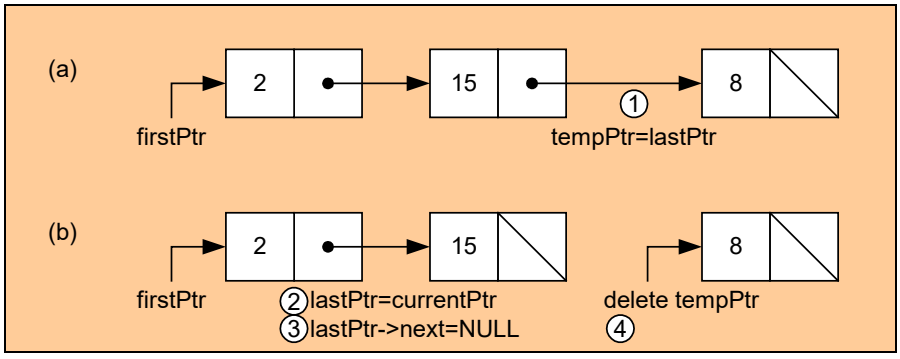

圖 17.5 刪除最後一項

下面範例是刪除串列資料的最後項：首先定義類別型態的 Student 資料與結構型態的 link 資料，在 link 資料結構中包含 Student 型態變數 data 與 link 型態指標 *next。然後定義 linklist 類別，類別的資料成員包含 link 型態指標 *firstPtr 與 *lastPtr，而類別的成員函數包含 linklist 建立者與 delBack 成員函數。

在 delBack 函數中，先定義一個暫存的 link 指標 *tempPtr，然後將 lastPtr 指標存入 tempPtr 指標。接著判斷串列中有沒有資料，若串列中沒有資料則令第一項指標 firstPtr 與最後項指標 lastPtr 皆等於 NULL。若串列中有資料則建立 currentPtr 指標存放 firstPtr 指標，然後從第一項開始找尋 lastPtr 指標，找到後存入 currentPtr->next，最後令 lastPtr 指標等於 currentPtr 指標，而 lastPtr->next 等於 NULL。

```
class Student {                                    //自定 Student 資料
    int student_id;
    char student_name[40];
};

struct link {                                      //定義 link 資料結構
    Student data;
    link *next;
};

class linklist {                                   //定義串列資料類別
```

```
    link *firstPtr;
    link *lastPtr;
public:
    linklist() { firstPtr = lastPtr = NULL; }
    void delBack()                              //定義刪除最後項函數
    {
       link *tempPtr = lastPtr;                 //①保存最後項的指標
       if (firstPtr == lastPtr)                 //②若串列中沒有資料
          firstPtr = lastPtr = NULL;            //③令頭尾指標指向新資料
       else {                                   //④若串列中已有資料
          link *currentPtr = firstPtr;          //⑤令前一項指標=第一項指標
          while(currentPtr->next != lastPtr)    //⑥找尋前一項指標迴圈
             currentPtr = currentPtr->next;
          lastPtr = currentPtr;                 //⑦最後項指標移到前一項
          lastPtr->next = NULL;                 //⑧最後項的 next 指標=0
       }                                        //   NULL 為結束識別碼
       delete tempPtr;                          //⑨刪除原來的最後項指標
    }
};
```

圖 17.6 是刪除串列中間項資料的圖解，首先先找到要刪除的位置指標並保存該指標（tempPtr = currentPtr），然後令指標等於下一指標（currentPtr = currentPtr->next），再刪除原來的中間項指標（delete tempPtr）。

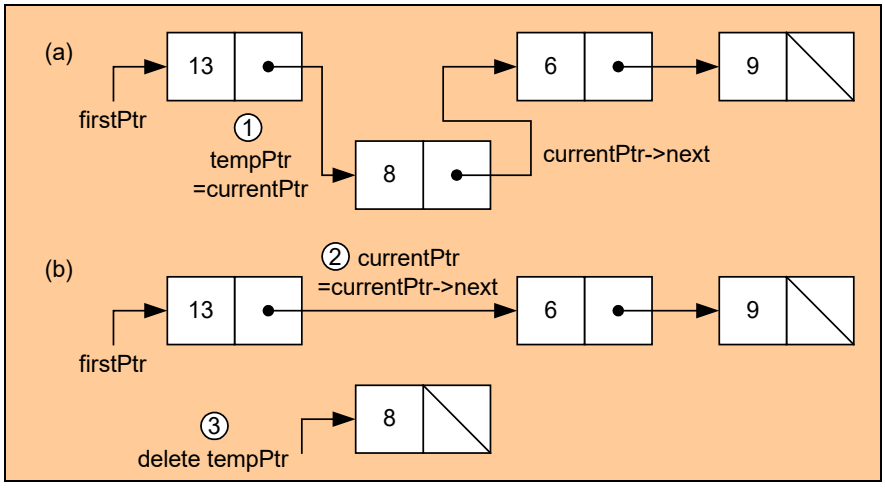

圖 17.6 刪除中間項

程式 17-02：刪除節點練習

```
1.  //檔案名稱：d:\C++17\C1702.cpp
2.  #include <iostream>
3.  using namespace std;
4.
5.  class Student                                  // 自定 Student 資料類別
6.  {
7.     int student_id;
8.     char student_name[40];
9.  public:
10.    friend istream& operator >> (istream& in, Student& obj)
11.    {
12.       in >> obj.student_id >> obj.student_name;
13.       return in;
14.    }
15.    friend ostream& operator << (ostream& out, Student& obj)
16.    {
17.       out << obj.student_id << '\t' << obj.student_name;
18.       return out;
19.    }
20. };
21.
22. struct link //定義 link 資料結構
23. {
24.    Student data;                               //Student 型態資料
25.    link *next;                                 //link 型態指標
26. };
27.
28. class linklist                                 //定義串列資料類別
29. {
30.    link *firstPtr;                             //串列起始指標
31.    link *lastPtr;                              //串列結束指標
32. public:
33.    linklist() { firstPtr = lastPtr = NULL; }  //建立者
34.    void addFront (Student obj);                //宣告插入第一項原型
35.    void addBack (Student obj);                 //宣告插入最後項原型
36.    void delFront();                            //宣告刪除第一項原型
37.    void delBack();                             //宣告刪除最後項原型
38.    void showItem();                            //宣告顯示串列原型
39. };
40.
41. void linklist::addFront (Student obj)         //定義插入第一項函數
42. {
43.    link *newPtr = new link;
44.    newPtr->data = obj;
45.    if (firstPtr == NULL) {
46.       firstPtr = lastPtr = newPtr;
47.       lastPtr->next = NULL;
48.    }
49.    else {
```

```
50.        newPtr->next = firstPtr;
51.        firstPtr = newPtr;
52.     }
53.  }
54.
55.  void linklist::addBack (Student obj)       //定義插入最後項函數
56.  {
57.     link *newPtr = new link;
58.     newPtr->data = obj;
59.     if (firstPtr == NULL) {
60.        firstPtr = lastPtr = newPtr;
61.        lastPtr->next = NULL;
62.     }
63.     else {
64.        lastPtr->next = newPtr;
65.        lastPtr = newPtr;
66.        lastPtr->next = NULL;
67.     }
68.  }
69.
70.  void linklist::delFront()                  //定義刪除第一項函數
71.  {
72.     link *tempPtr = firstPtr;
73.     if (firstPtr == NULL)
74.        return;
75.     else {
76.        firstPtr = firstPtr->next;
77.        delete tempPtr;
78.     }
79.  }
80.
81.  void linklist::delBack()                   //定義刪除最後項函數
82.  {
83.     link *tempPtr = lastPtr;
84.     if (firstPtr == NULL)
85.        return;
86.     else {
87.        if (firstPtr == lastPtr)
88.           firstPtr = lastPtr = NULL;
89.        else {
90.           link *currentPtr = firstPtr;
91.           while(currentPtr->next != lastPtr)
92.              currentPtr = currentPtr->next;
93.           lastPtr = currentPtr;
94.           lastPtr->next = NULL;
95.        }
96.     }
97.     delete tempPtr;
98.  }
99.
100. void linklist::showItem()                 //定義顯示串列資料函數
101. {
102.    link *currentPtr = firstPtr;
```

```
103.    while( currentPtr != NULL)
104.    {
105.       cout << currentPtr->data << endl;
106.       currentPtr = currentPtr->next;
107.    }
108. }
109.
110. int main(int argc, char** argv)
111. {
112.    Student studata;                          //定義 Student 物件
113.    linklist ls;                              //定義 linklist 物件
114.    char n;
115.
116.    cout << "1.插入第一項\n2.插入最後項\n"
117.        << "3.刪除第一項\n4.刪除最後項\n"
118.        << "0.結束\n";
119.    while(1) {
120.       cout << "請選擇(1-4 或 0): ";
121.       cin >> n;
122.       switch (n) {
123.          case '1':
124.             cout << "請輸入學號與姓名：" ;
125.             cin >> studata;
126.             ls.addFront(studata);
127.             ls.showItem();
128.             break;
129.          case '2':
130.             cout << "請輸入學號與姓名：" ;
131.             cin >> studata;
132.             ls.addBack(studata);
133.             ls.showItem();
134.             break;
135.          case '3':
136.             ls.delFront();
137.             ls.showItem();
138.             break;
139.          case '4':
140.             ls.delBack();
141.             ls.showItem();
142.             break;
143.          case '0':
144.             return 0;
145.       }
146.       cout << endl;
147.    }
148.    return 0;
149.  }
```

程式輸出

```
1.插入第一項
2.插入最後項
3.刪除第一項
4.刪除最後項
0.結束
請選擇(1-4 或 0): 1 Enter
請輸入學號與姓名：200 Frank Enter
200     Frank

請選擇(1-4 或 0): 2 Enter
請輸入學號與姓名：300 Carol Enter
200     Frank
300     Carol

請選擇(1-4 或 0): 1 Enter
請輸入學號與姓名：100 Arther Enter
100     Arther
200     Frank
300     Carol

請選擇(1-4 或 0): 3 Enter
200     Frank
300     Carol

請選擇(1-4 或 0): 3 Enter
300     Carol

請選擇(1-4 或 0): 4 Enter

請選擇(1-4 或 0): 0 Enter
```

17.2 堆疊與佇列

堆疊（stack）是一種**後進先出（last-in-first-out；LIFO）**的資料結構。**佇列（queue）**則是一種**先進先出（first-in-first-out；FIFO）**的資料結構。

17.2.1 堆疊資料結構

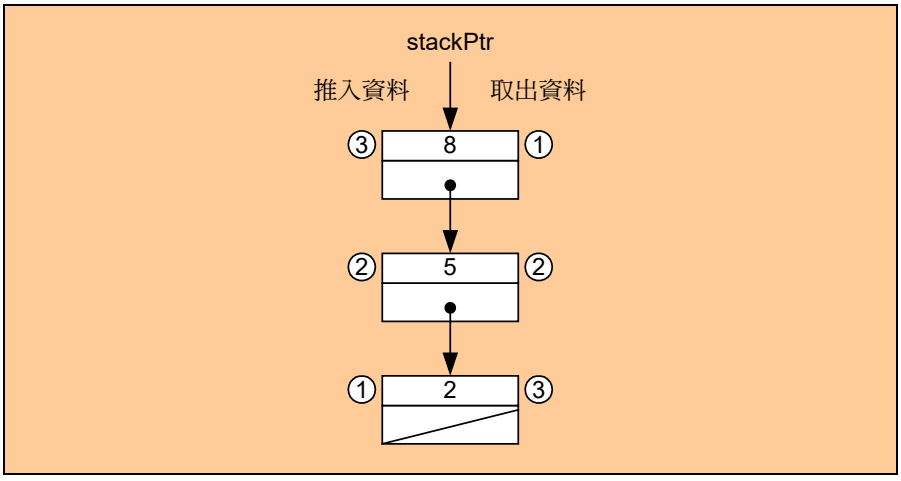

圖 17.7 堆疊資料結構

堆疊（stack）者堆碟也，也就是類似堆盤子啦！資料依序由下向上堆，取出時反序由上向下取，也就是後進先出（last-in-first-out）的意思。從指標的用法來看，堆疊資料結構類似單一指標的串列資料結構，資料永遠是由堆疊頂端（串列第一項）推入，而且永遠是由堆疊頂端（串列第一項）取出。

下面範例是堆入與取出堆疊資料：首先定義類別型態的 Student 資料與結構型態的 link 資料，在 link 資料結構中包含 Student 型態變數 data 與 link 型態指標 *next。然後定義 linklist 類別，類別的資料成員包含 link 型態指標 *stackPtr，而類別的成員函數包含 linklist 建立者、push 與 pop 成員函數。

在 push 函數中，先定義一個 link 指標 *newPtr，然後將參數 obj 存入 newPtr->data。接著將堆疊指標 stackPtr 存入 newPtr->next，此時新指標的下一項為 NULL，最後將新指標 newPtr 存入 stackPtr 成為堆疊的新指標。

在 pop 函數中，先定義一個暫存的 link 指標 *tempPtr，然後將 stackPtr 指標存入 tempPtr 指標。接著判斷堆疊中有沒有資料，若堆疊中沒有資料則顯示 “堆疊空了” 訊息並結束 pop，若堆疊中有資料則將 tempPtr->data

存入 tempData 中，且令 stackPtr 等於 stackPtr->next，也就是指標向下移動，最後刪除暫存指標 tempPtr。

```
class Student                                          //自定 Student 資料類別
{
   int student_id;
   char student_name[40];
};

struct link                                            //定義 link 資料結構
{
   Student data;
   link *next;
};

class Stack                                            //定義堆疊資料類別
{
   link *stackPtr;                                     //堆疊指標
public:
   Stack() { stackPtr = NULL; }                        //建立者
   void push (Student obj) {                           //定義推入資料函數
      link *newPtr = new link;
      newPtr->data = obj;
      newPtr->next = stackPtr;
      stackPtr = newPtr;
   }
   Student pop() {                                     //定義取回資料函數
      link *tempPtr = stackPtr;
      Student tempData;
      if (stackPtr == NULL) {
         cout << "堆疊空了！" << endl;
         exit(0);
      } else {
         tempData = tempPtr->data;
         stackPtr = stackPtr->next;
         delete tempPtr;
      }
      return tempData;
   }
};
```

程式 17-03：堆疊資料結構練習

```
1.   //檔案名稱：d:\C++17\C1703.cpp
2.   #include <iostream>
3.   #include <string>
4.   using namespace std;
5.
6.   class Student                                     //自定 Student 資料類別
7.   {
```

```
8.      int student_id;
9.      char student_name[40];
10.  public:
11.     friend istream& operator >> (istream& in, Student& obj) {
12.        in >> obj.student_id >> obj.student_name;
13.        return in;
14.     }
15.     friend ostream& operator << (ostream& out, Student& obj) {
16.        out << obj.student_id << '\t' << obj.student_name;
17.        return out;
18.     }
19.  };
20.
21.  struct link                                    //定義 link 資料結構
22.  {
23.     Student data;
24.     link *next;
25.  };
26.
27.  class Stack                                    //定義堆疊資料類別
28.  {
29.     link *stackPtr;                             //堆疊指標
30.  public:
31.     Stack() { stackPtr = NULL; }                //建立者
32.     void push (Student obj);                    //宣告推入資料原型
33.     Student pop();                              //宣告取回資料原型
34.  };
35.
36.  void Stack::push (Student obj)                 //定義推入資料函數
37.  {
38.     link *newPtr = new link;
39.     newPtr->data = obj;
40.     newPtr->next = stackPtr;
41.     stackPtr = newPtr;
42.  }
43.
44.  Student Stack::pop()                           //定義取回資料函數
45.  {
46.     link *tempPtr = stackPtr;
47.     Student tempData;
48.     string error = "堆疊空了！\n";
49.     if (stackPtr == NULL) {
50.        throw error;
51.     }
52.     else {
53.        tempData = tempPtr->data;
54.        stackPtr = stackPtr->next;
55.        delete tempPtr;
56.     }
57.     return tempData;
58.  }
```

```
59.
60.  int main(int argc, char** argv)
61.  {
62.     Student studata;                          //定義 Student 物件
63.     Stack ls;                                 //定義 Stack 物件
64.     char n;
65.
66.     try {
67.        while(1) {
68.           cout << "1.推入資料項  2.取出資料項  0.結束  請選擇(1,2 或 0): ";
69.           cin >> n;
70.           switch (n) {
71.              case '1':
72.              cout << "請輸入學號與姓名：" ;
73.              cin >> studata;
74.              ls.push(studata);
75.              break;
76.              case '2':
77.              studata = ls.pop();
78.              cout << studata << endl;
79.              break;
80.              case '0':
81.              exit(0);
82.           }
83.           cout << endl;
84.        }
85.     } catch (string error) {
86.        cout << error;
87.     }
88.     return 0;
89.  }
```

程式輸出

```
1.推入資料項  2.取出資料項  0.結束  請選擇(1,2 或 0): 1 Enter
請輸入學號與姓名：100 Ken Enter

1.推入資料項  2.取出資料項  0.結束  請選擇(1,2 或 0): 1 Enter
請輸入學號與姓名：200 Shanon Enter

1.推入資料項  2.取出資料項  0.結束  請選擇(1,2 或 0): 1 Enter
請輸入學號與姓名：300 Bill Enter

1.推入資料項  2.取出資料項  0.結束  請選擇(1,2 或 0): 2 Enter
300     Bill

1.推入資料項  2.取出資料項  0.結束  請選擇(1,2 或 0): 2 Enter
200     Shanon
```

```
1.推入資料項  2.取出資料項  0.結束  請選擇(1,2 或 0): 2 Enter
100     Ken

1.推入資料項  2.取出資料項  0.結束  請選擇(1,2 或 0): 2 Enter
堆疊空了！
```

17.2.2 佇列資料結構

佇列（queue）則是先進先出（first-in-first-out）的觀念，資料依序由前端堆入佇列，然後由佇列後端取出。從指標的用法來看，佇列資料結構類似單向雙指標的串列資料結構，資料永遠是由佇列前端（串列第一項）推入，而且永遠是由佇列後端（串列最後項）取出。

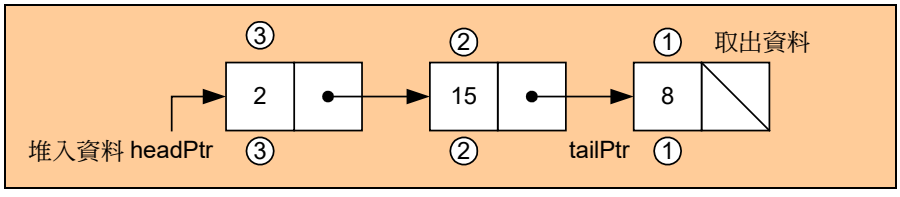

圖 17.8 佇列資料結構

下面範例是推入與取出佇列資料：首先定義類別型態的 Student 資料與結構型態的 link 資料，在 link 資料結構中包含 Student 型態變數 data 與 link 型態指標 *next。然後定義 linklist 類別，類別的資料成員包含 link 型態指標 *headPtr 與 *tailPtr，而類別的成員函數包含 linklist 建立者、push 與 pop 成員函數。

在 push 函數中，先定義一個 link 指標 *newPtr，然後將參數 obj 存入 newPtr->data。接著判斷佇列中是否有資料，若佇列中沒有資料則令 headPtr=tailPtr=newPtr，且 tailPtr->next 等於 NULL。若佇列中有資料則先將 headerPtr 存入 newPtr->next 指標，再將新指標 newPtr 存入 headPtr 指標。

在 pop 函數中，先定義一個暫存的 link 指標 *tempPtr，然後將 tailPtr 指標存入 tempPtr 指標。接著判斷佇列中有沒有資料，若佇列中沒有資料則顯示 “佇列空了！”訊息。若佇列中有資料且 headPtr=tailPtr，則令 headPtr=tailPtr 等於 NULL，若堆疊中有資料而 headPtr!=tailPtr，則將

headPtr 存入 currentPtr 指標中，然後從第一項開始找尋最後一項指標 tailPtr，找到後將 currentPtr 存入 tailPtr，再令 tailPtr->next 等於 NULL。最後將 tempPtr->data 資料存入 tempData 緩衝器中，並傳回給呼叫敘述。

```
class Student {                                    //自定 Student 資料類別
   int student_id;
   char student_name[40];
};

struct link {                                      //定義 link 資料結構
   Student data;
   link *next;
};

class Queue {                                      //定義佇列資料類別
   link *headPtr;                                  //佇列起始指標
   link *tailPtr;                                  //佇列結束指標
public:
   Queue() { headPtr = tailPtr = NULL; }           //建立者
   void push (Student obj) {                       //宣告推入資料原型
      link *newPtr = new link;
      newPtr->data = obj;
      if (headPtr == NULL) {
         headPtr = tailPtr = newPtr;
         tailPtr->next = NULL;
      }
      else {
         newPtr->next = headPtr;
         headPtr = newPtr;
      }
   }
   Student pop() {                                 //宣告取回資料原型
      link *tempPtr = tailPtr;
      Student tempData;
      if (headPtr == NULL) {
         cout << "佇列空了！" << endl;
         exit(0);
      }
      else {
         if (headPtr == tailPtr)
            headPtr = tailPtr = NULL;
         else {
            link *currentPtr = headPtr;
            while(currentPtr->next != tailPtr)
               currentPtr = currentPtr->next;
            tailPtr = currentPtr;
            tailPtr->next = NULL;
         }
         Student tempData = tempPtr->data;
         delete tempPtr;
```

```
      }
      return tempData;
   }
};
```

程式 17-04：佇列資料結構練習

```
1.  //檔案名稱：d:\C++17\C1704.cpp
2.  #include <iostream>
3.  #include <string>
4.  using namespace std;
5.
6.  class Student                                  //自定 Student 資料類別
7.  {
8.     int student_id;
9.     char student_name[40];
10. public:
11.    friend istream &operator>> (istream &in, Student &obj) {
12.       in >> obj.student_id >> obj.student_name;
13.       return in;
14.    }
15.    friend ostream &operator<< (ostream &out, Student &obj) {
16.       out << obj.student_id << '\t' << obj.student_name;
17.       return out;
18.    }
19. };
20.
21. struct link                                    //定義 link 資料結構
22. {
23.    Student data;
24.    link *next;
25. };
26.
27. class Queue                                    //定義佇列資料類別
28. {
29.    link *headPtr;                             //佇列起始指標
30.    link *tailPtr;                             //佇列結束指標
31. public:
32.    Queue() { headPtr = tailPtr = NULL; }      //建立者
33.    void push (Student obj);                   //宣告推入資料原型
34.    Student pop();                             //宣告取回資料原型
35. };
36.
37. void Queue::push (Student obj)                //定義推入資料原型
38. {
39.    link *newPtr = new link;
40.    newPtr->data = obj;
41.    if (headPtr == NULL) {
42.       headPtr = tailPtr = newPtr;
43.       tailPtr->next = NULL;
```

```
44.     }
45.     else {
46.        newPtr->next = headPtr;
47.           headPtr = newPtr;
48.        }
49.  }
50.
51.  Student Queue::pop()                          //定義取回資料原型
52.  {
53.     link *tempPtr = tailPtr;
54.     Student tempData;
55.     string error = "佇列空了！\n";
56.     if (headPtr == NULL) {
57.        throw error;
58.     }
59.     else {
60.        if (headPtr == tailPtr)
61.           headPtr = tailPtr = NULL;
62.        else {
63.           link *currentPtr = headPtr;
64.           while(currentPtr->next != tailPtr)
65.              currentPtr = currentPtr->next;
66.           tailPtr = currentPtr;
67.           tailPtr->next = NULL;
68.        }
69.        tempData = tempPtr->data;
70.        delete tempPtr;
71.     }
72.     return tempData;
73.  }
74.
75.  int main(int argc, char** argv)
76.  {
77.     Student studata;                           //定義 Student 物件
78.     Queue ls;                                  //定義 Queue 物件
79.     char n;
80.
81.     try {
82.        while(true) {
83.           cout << "1.推入資料項  2.取出資料項  0.結束  請選擇(1,2 或 0): ";
84.           cin >> n;
85.           switch (n) {
86.              case '1':
87.              cout << "請輸入學號與姓名：" ;
88.              cin >> studata;
89.              ls.push(studata);
90.              break;
91.              case '2':
92.              studata = ls.pop();
93.              cout << studata << endl;
94.              break;
```

```
95.              case '0':
96.              exit(1);
97.           }
98.           cout << endl;
99.        }
100.    } catch (string error) {
101.       cout << error;
102.    }
103.    return 0;
104. }
```

程式輸出

```
1.推入資料項  2.取出資料項  0.結束  請選擇(1,2 或 0): 1 Enter
請輸入學號與姓名：100 Frank Enter

1.推入資料項  2.取出資料項  0.結束  請選擇(1,2 或 0): 1 Enter
請輸入學號與姓名：200 Jesse Enter

1.推入資料項  2.取出資料項  0.結束  請選擇(1,2 或 0): 1 Enter
請輸入學號與姓名：300 Carol Enter

1.推入資料項  2.取出資料項  0.結束  請選擇(1,2 或 0): 2 Enter
100     Frank

1.推入資料項  2.取出資料項  0.結束  請選擇(1,2 或 0): 2 Enter
200     Jesse

1.推入資料項  2.取出資料項  0.結束  請選擇(1,2 或 0): 2 Enter
300     Carol

1.推入資料項  2.取出資料項  0.結束  請選擇(1,2 或 0): 2 Enter
佇列空了！
```

17.3 二元樹

二元樹（binary tree）是非線性的連結串列，它的每一個節點都可以連結到左右二個子節點。在龐大資料的排序與搜尋中，使用二元樹資料結構比使用線性資料結構快很多。

17.3.1 二元樹定義

前二節所討論的都是線性的資料結構（linear data structure），而且每個節點都只連結到另一個節點。但是二元樹資料結構則是非線性的資料結構（non-linear data structure），它的每一個節點都可以連結到左右二個子節點。

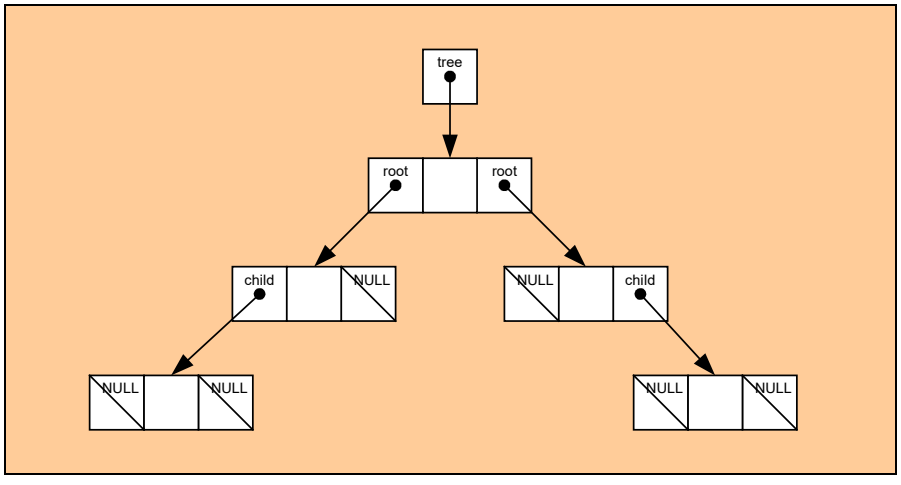

圖 17.9 二元樹結構

如圖 17.9 所示，二元樹的結構像一個上下反轉的樹狀，樹根在頂端然後向下分支發展。二元樹的樹根位置稱為**根節點（root node）**，而根節點有二個指標分別指向二個**子節點（child node）**，每個子節點又有二個指標且分別指向他們的子節點。但不是每個節點都指向二個子節點，有些節點指向二個子節點，有些節點指向一個子節點而另一個指標則設為

NULL，而有些節點的二個指標皆設為 NULL。二個指標皆設為 NULL 的節點稱為**葉節點（leaf node）**。

二元樹可以被分成**副分支（subtree）**，副分支則是從樹根分出的整個分支，如圖 17.10 虛線範圍內的整個分支稱為左副分支（left subtree）。

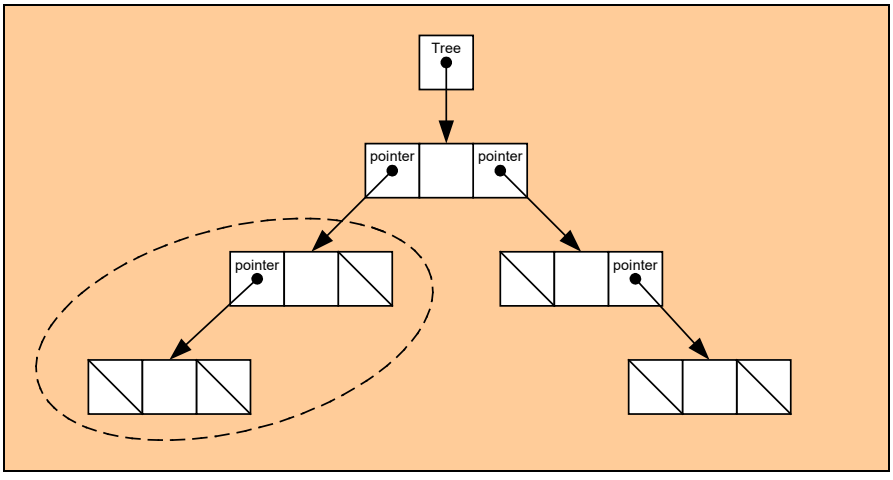

圖 17.10 根節點下的左副分支結構

17.3.2 二元樹應用

因為線性資料結構是以循序方式處理資料，所以當資料很多時利用線性資料結構來搜尋資料是很慢的。二元樹資料結構不是以循序方式處理資料，因此非常適合處理大批資料的搜尋，而使用二元樹搜尋資料稱為**二元搜尋樹（binary search tree）**。

如圖 17.11 二元搜尋樹的每個節點都儲存一個字母，且每一個節點內的字母大於左邊子節點內的字母，但小於右邊子節點內的字母。如下圖根節點儲存 L，它的左邊子節點儲存 F，它的右邊子節點儲存 O。F 的左邊子節點儲存 E，F 的右邊子節點為 NULL。O 的左邊子節點為 NULL，O 的右邊子節點儲存 T。

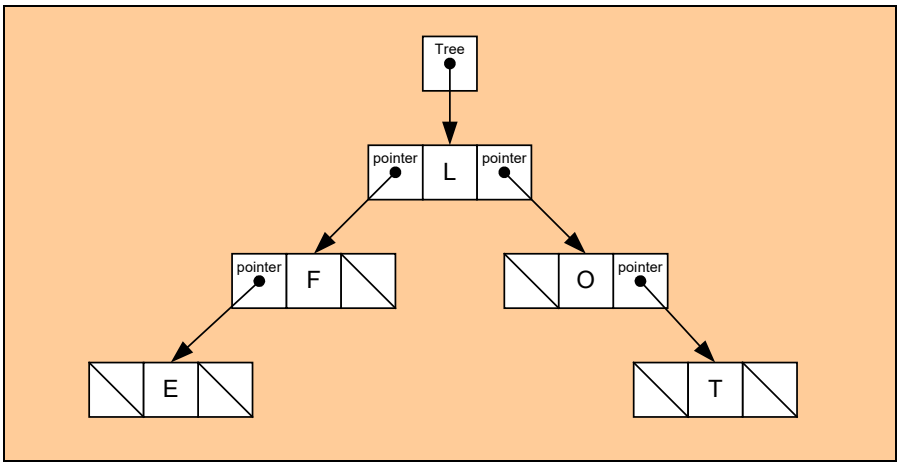

圖 17.11 二元搜尋樹的結構

17.4 二元樹運算

二元樹（binary tree）資料結構與線性資料結構都是用來管理資料庫的資料，例如插入資料到資料庫、排列資料庫的資料、在資料庫中搜尋資料、與刪除資料庫的資料等等。

17.4.1 建立二元樹

下面範例是建立二元樹資料結構與類別：首先定義結構型態的 TreeNode 資料，在 TreeNode 資料結構中包含 int 型態變數 data 與 TreeNode 型態指標 *left 與 *right。然後定義 BinaryTree 類別，類別的資料成員包含 TreeNode 型態指標 *root，類別成員函數原型包含 BinaryTree 建立者，showInOrder、showPreOrder、showPostOrder、deleteNode 等私用成員函數原型，insertNode、removeNode 等公用函數原型，與 coutInOrder、coutPreOrder、coutPostOrder、searchNode 等公用成員函數。17.4.2 至 17.4.5 節將逐步實現（implement）上述公用函數。

因為 showInOrder、showPreOrder、showPostOrder、deleteNode 等為遞迴函數，而且必須傳遞 TreeNode 指標，所以定義成 private 成員函數，然後再利用公用成員函數 coutInOrder、coutPreOrder、coutPostOrder、removeNode 呼叫這些遞迴函數。

```
struct TreeNode {                                  //定義 TreeNode 結構資料
   int data;
   TreeNode *left;
   TreeNode *right;
};

class BinaryTree {                                 //定義二元樹資料類別
   TreeNode *root;
   void showInOrder(TreeNode *);                    //顯示大小排序後資料原型
   void showPreOrder(TreeNode *);                   //顯示二元排列前資料原型
   void showPostOrder(TreeNode *);                  //顯示二元排列後資料原型
   void deleteNode(int num, TreeNode *&nodePtr);   //刪除節點函數原型
public:
   BinaryTree() { root = NULL; }
   void insertNode(int);                            //插入節點函數原型
   void coutInOrder() { showInOrder(root); } //呼叫大小排序後資料
   void coutPreOrder() { showPreOrder(root); } //呼叫二元排列前資料
   void coutPostOrder() { showPostOrder(root); } //呼叫二元排列後資料
   bool searchNode(int num);                        //搜尋資料函數原型
   void removeNode(int num) {deleteNode(num, root);} //呼叫刪除節點函數
};
```

17.4.2 插入節點

下面範例是實現（implement）17.4.1 節的 insertNode 函數。插入新節點之前必須先建立新節點位置，並將參數存入新節點的資料欄位，然後將新節點的左右子節點設成 NULL，因為新加入的節點必須是葉節點（leaf node）。

接下來則決定插入新節點的位置：首先判斷根節點是否為葉節點，若是則將新節點插入到根節點位置，若不是則必須逐一比對新資料與各節點資料，找出新節點的插入位置。例如若新資料小於根節點資料則逐一比對左邊分支的各節點並找出插入點，若新資料大於根節點資料則逐一比對右邊分支的各節點找出插入點。

```
void BinaryTree::insertNode(int num)        //插入節點函數
{
   TreeNode *newNode = new TreeNode;        //newNode 為新節點指標
   TreeNode *tempNode;                      //tempNode 為暫存節點
```

```
   newNode->left = newNode->right = NULL; //令新節點左右指標=NULL
   newNode->data = num;                   //資料存入新節點的緩衝區
   if(root == NULL) {                     //若二元樹還沒有資料
      root = newNode;                     //令根節點=新節點
   } else {                               //若二元樹中已有資料
      tempNode = root;                    //令 tempNode 指向 root
      while(tempNode) {                   //當 tempNode 不等於 NULL
         if(num < tempNode->data) {       //新資料<tempNode->data
            if(tempNode->left) {          //若 tempNode->left != NULL
               tempNode = tempNode->left; //tempNode->left 為臨時指標
            } else {                      //若 tempNode->left == NULL
               tempNode->left = newNode;  //新節點插入 tempNode->left
               break;
            }
         } else if(num > tempNode->data) { //新資料 > tempNode->data
            if(tempNode->right) {         //若 tempNode->right!=NULL
               tempNode = tempNode->right; //tempNode->right 為臨時指標
            } else {                      //若 tempNode->right==NULL
               tempNode->right = newNode; //新節點插入 tempNode->right
               break;
            }
         } else {                         //新資料 == tempNode->data
            cout << "資料重複！";
            break;
         }
      }
   }
}
```

程式 17-05 於建立二元樹資料結構後，插入 5, 9, 1, 6, 4 五個節點至二元樹結構中。首先因根節點為 NULL，所以第一筆資料 5 被插入根節點位置。第二筆資料 9 大於 5 所以被插入到 5 的右邊子節點。第三筆資料 1 小於 5 所以被插入到 5 的左邊子節點。第四筆資料 6 大於 5，屬於右副分支，但小於 9 所以被插入到 9 的左邊子節點。第五筆資料 4 小於 5，屬於左副分支，但大於 1 所以被插入到 1 的右邊子節點。插入五個節點後的二元樹結構如下圖。

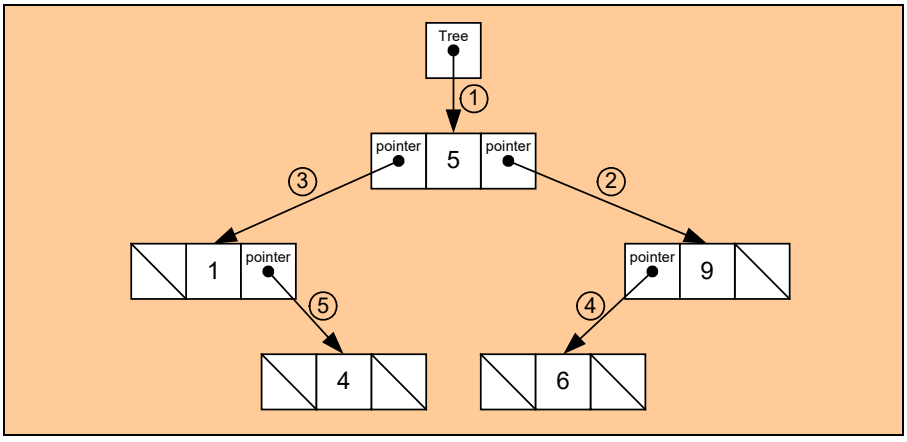

圖 17.12 程式 17-05 插入節點的順序

程式 17-05：建立二元樹並插入節點

```
1.  //檔案名稱：d:\C++17\C1705.cpp
2.  #include <iostream>
3.  using namespace std;
4.
5.  struct TreeNode {                              //定義 TreeNode 結構資料
6.     int data;
7.     TreeNode *left;
8.     TreeNode *right;
9.  };
10.
11. class BinaryTree {                             //定義二元樹資料類別
12.    TreeNode *root;
13. public:
14.    BinaryTree() { root = NULL; }
15.    void insertNode(int);                       //插入節點函數原型
16. };
17.
18. void BinaryTree::insertNode(int num)           //插入節點函數
19. {
20.    TreeNode *newNode = new TreeNode;           //newNode 為新節點指標
21.    TreeNode *tempNode;                         //tempNode 為暫存節點
22.    newNode->left = newNode->right = NULL;      //令新節點左右指標=NULL
23.    newNode->data = num;                        //資料存入新節點的緩衝區
24.    if(root == NULL) {                          //若二元樹還沒有資料
25.       root = newNode;                          //令根節點=新節點
26.    } else {                                    //若二元樹中已有資料
27.       tempNode = root;                         //令 tempNode 指向 root
28.       while(tempNode) {                  //當 tempNode 不等於 NULL
29.          if(num < tempNode->data) {   //新資料 < tempNode->data
```

```
30.              if(tempNode->left) {      //若 tempNode->left != NULL
31.                  tempNode = tempNode->left; //tempNode->left 為臨時指標
32.              } else {                  //若 tempNode->left == NULL
33.                  tempNode->left = newNode; //新節點插入 tempNode->left
34.                  break;
35.              }
36.           } else if(num > tempNode->data) { //新資料 > tempNode->data
37.              if(tempNode->right) {     //若 tempNode->right!=NULL
38.                  tempNode = tempNode->right; //tempNode->right 為臨時指標
39.              } else {                   //若 tempNode->right==NULL
40.                  tempNode->right = newNode; //新節點插入 tempNode->right
41.                  break;
42.              }
43.           } else {                       //新資料 == tempNode->data
44.              cout << "資料重複！";
45.              break;
46.           }
47.        }
48.     }
49.  }
50.
51.  int main(int argc, char** argv)
52.  {
53.        BinaryTree intTree;
54.     cout << "插入節點...";
55.     intTree.insertNode(5);                  //呼叫 insertNode 函數
56.     intTree.insertNode(9);                  //呼叫 insertNode 函數
57.     intTree.insertNode(1);                  //呼叫 insertNode 函數
58.     intTree.insertNode(6);                  //呼叫 insertNode 函數
59.     intTree.insertNode(4);                  //呼叫 insertNode 函數
60.     cout << "完成\n";
61.     return 0;
62.  }
```

程式輸出

```
插入節點...完成
```

17.4.3 顯示二元樹資料

二元樹的 inorder **遍歷（traversal）**是先**越過（traverse）**左邊副分支，再處理根節點的資料，最後在越過右邊副分支。inorder 先處理每一節點的左邊子節點資料後，再處理該節點的資料，最後處理該節點右邊子節點的資料如下。下面範例是實現 17.4.1 節的 showInOrder 函數，而圖 17.13 則

顯示 inorder 越過二元樹的順序。**註：遍歷（traversal）與越過（traverse）更白話的說明是逐項比對。**

1. 以遞迴呼叫 showInOrder 函數，處理左副分支的節點。
2. 處理節點。
3. 以遞迴呼叫 showInOrder 函數，處理右副分支的節點。

```
void BinaryTree::showInOrder(TreeNode *nodePtr) //顯示 InOrder 資料函數
{
   if(nodePtr)                                  //若 nodePtr 不等於 NULL
   {
      showInOrder(nodePtr->left);               //遞迴呼叫，越過左邊分支
      cout << nodePtr->data << ' ';             //處理節點資料
      showInOrder(nodePtr->right);              //遞迴呼叫，越過右邊分支
   }
}
```

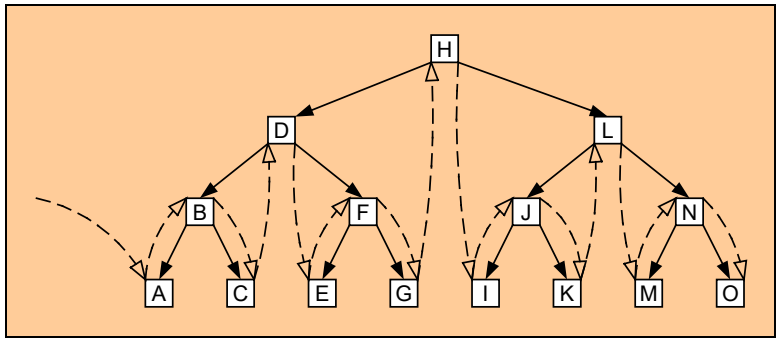

圖 17.13 inorder 越過二元樹的順序

二元樹的 preorder 遍歷是先處理根節點的資料，然後越過左邊副分支，最後後越過右邊副分支。Preorder 處理資料的順序是先處理節點資料，在處理左邊子節點與右邊子節點的資料如下。下面範例是實現（implement）17.4.1 節的 showPreOrder 函數，而圖 17.14 則顯示 preorder 越過二元樹的順序。

1. 處理節點。
2. 以遞迴呼叫 showPreOrder 函數，處理左副分支的節點。
3. 以遞迴呼叫 showPreOrder 函數，處理右副分支的節點。

```
void BinaryTree::showPreOrder(TreeNode *nodePtr) //顯示 PreOrder 資料函數
{
   if(nodePtr)                              //若 nodePtr 不等於 NULL
   {
      cout << nodePtr->data << ' ';         //處理節點資料
      showPreOrder(nodePtr->left);          //遞迴呼叫，越過左邊分支
      showPreOrder(nodePtr->right);         //遞迴呼叫，越過右邊分支
   }
}
```

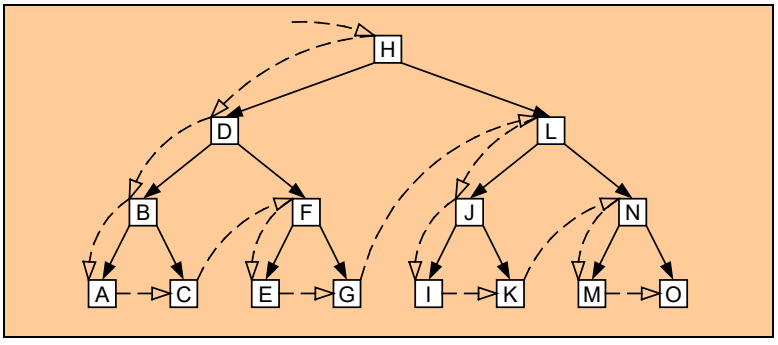

圖 17.14 preorder 越過二元樹的順序

二元樹的 postorder 遍歷是先越過左邊副分支，然後再越過右邊副分支，最後是處理根節點的資料。postorder 先處理每一節點的左邊子節點與右邊子節點的資料，再處理該節點的資料如下。下面範例是實現（implement）17.4.1 節的 showPostOrder 函數，而 postorder 越過二元樹的順序。

1. 以遞迴呼叫 showPostOrder 函數，處理左副分支的節點。
2. 以遞迴呼叫 showPostOrder 函數，處理右副分支的節點。
3. 處理節點。

```
void BinaryTree::showPostOrder(TreeNode *nodePtr) //顯示 PostOrder 函數
{
   if(nodePtr)                              //若 nodePtr 不等於 NULL
   {
      showPostOrder(nodePtr->left);         //遞迴呼叫，越過左邊分支
      showPostOrder(nodePtr->right);        //遞迴呼叫，越過右邊分支
      cout << nodePtr->data << ' ';         //處理節點資料
   }
}
```

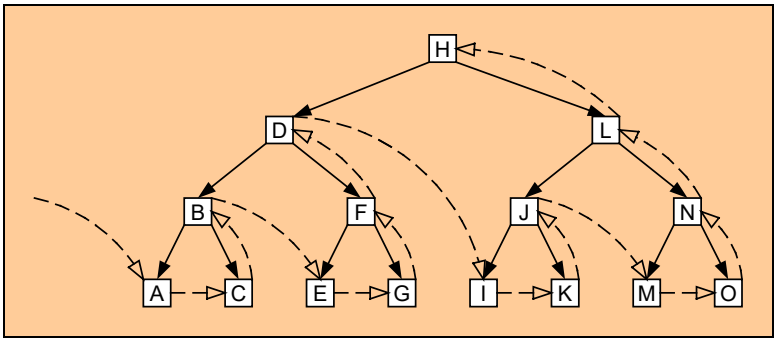

圖 17.15 postorder 越過二元樹的順序

程式 17-06：顯示二元樹節點資料

```
1.  //檔案名稱：d:\C++17\C1706.cpp
2.  #include <iostream>
3.  using namespace std;
4.
5.  struct TreeNode {                              //定義 TreeNode 結構資料
6.     int data;
7.     TreeNode *left;
8.     TreeNode *right;
9.  };
10.
11. class BinaryTree {                             //定義二元樹資料類別
12.    TreeNode *root;
13.    void showInOrder(TreeNode *);               //顯示 InOrder 資料原型
14.    void showPreOrder(TreeNode *);              //顯示 PreOrder 資料原型
15.    void showPostOrder(TreeNode *);             //顯示 PostOrder 原型
16. public:
17.    BinaryTree() { root = NULL; }
18.    void insertNode(int);                       //插入節點函數原型
19.    void coutInOrder() { showInOrder(root); } //呼叫 InOrder 資料函數
20.    void coutPreOrder() { showPreOrder(root); } //呼叫 PreOrder 資料函數
21.    void coutPostOrder() { showPostOrder(root); } //呼叫 PostOrder 函數
22. };
23.
24. void BinaryTree::insertNode(int num)         //插入節點函數
25. {
26.    TreeNode *newNode = new TreeNode;           //newNode 為新節點指標
27.    TreeNode *tempNode;                         //tempNode 為暫存節點
28.    newNode->left = newNode->right = NULL;  //令新節點左右指標=NULL
29.    newNode->data = num;                        //資料存入新節點的緩衝區
30.    if(root == NULL) {                          //若二元樹還沒有資料
31.       root = newNode;                          //令根節點=新節點
32.    } else {                                    //若二元樹中已有資料
```

```
33.        tempNode = root;                  //令 tempNode 指向 root
34.        while(tempNode) {                 //當 tempNode 不等於 NULL
35.           if(num < tempNode->data) {     //新資料 < tempNode->data
36.              if(tempNode->left) {        //若 tempNode->left != NULL
37.                  tempNode = tempNode->left;  //tempNode->left 為臨時指標
38.              } else {                      //若 tempNode->left == NULL
39.                  tempNode->left = newNode;  //新節點插入 tempNode->left
40.                  break;
41.              }
42.           } else if(num > tempNode->data) {   //新資料 > tempNode->data
43.              if(tempNode->right) {            //若 tempNode->right!=NULL
44.                  tempNode = tempNode->right; //tempNode->right 為臨時指標
45.              } else {                     //若 tempNode->right==NULL
46.                  tempNode->right = newNode;  //新節點插入 tempNode->right
47.                  break;
48.              }
49.           } else {                         //新資料 == tempNode->data
50.              cout << "資料重複！";
51.              break;
52.           }
53.        }
54.     }
55.  }
56.
57.  void BinaryTree::showInOrder(TreeNode *nodePtr)//顯示 InOrder 資料函數
58.  {
59.     if(nodePtr)                              //若 nodePtr 不等於 NULL
60.     {
61.        showInOrder(nodePtr->left);           //遞迴呼叫，越過左邊分支
62.        cout << nodePtr->data << ' ';         //處理節點資料
63.        showInOrder(nodePtr->right);          //遞迴呼叫，越過右邊分支
64.     }
65.  }
66.
67.  void BinaryTree::showPreOrder(TreeNode *nodePtr) //顯示 PreOrder 資料函數
68.  {
69.     if(nodePtr)                              //若 nodePtr 不等於 NULL
70.     {
71.        cout << nodePtr->data << ' ';    //處理節點資料
72.        showPreOrder(nodePtr->left);     //遞迴呼叫，越過左邊分支
73.        showPreOrder(nodePtr->right);    //遞迴呼叫，越過右邊分支
74.     }
75.  }
76.
77.  void BinaryTree::showPostOrder(TreeNode *nodePtr) //顯示 PostOrder 函數
78.  {
79.     if(nodePtr)                           //若 nodePtr 不等於 NULL
80.     {
```

```
81.         showPostOrder(nodePtr->left);    //遞迴呼叫，越過左邊分支
82.         showPostOrder(nodePtr->right);   //遞迴呼叫，越過右邊分支
83.         cout << nodePtr->data << ' ';    //處理節點資料
84.     }
85.  }
86.
87.  int main(int argc, char** argv)
88.  {
89.     BinaryTree intTree;
90.     cout << "插入節點...";
91.     intTree.insertNode(5);               //呼叫 insertNode 函數
92.     intTree.insertNode(9);               //呼叫 insertNode 函數
93.     intTree.insertNode(1);               //呼叫 insertNode 函數
94.     intTree.insertNode(6);               //呼叫 insertNode 函數
95.     intTree.insertNode(4);               //呼叫 insertNode 函數
96.     cout << "完成";
97.     cout << "\n 顯示 PreOrder 資料：";
98.     intTree.coutPreOrder();              //呼叫 coutPreNode 函數
99.     cout << "\n 顯示 PostOrder 資料：";
100.    intTree.coutPostOrder();             //呼叫 coutPostNode 函數
101.    cout << "\n 顯示 InOrder 資料：";
102.    intTree.coutInOrder();               //呼叫 coutInNode 函數
103.    cout << endl;
104.    return 0;
105. }
```

程式輸出

```
插入節點...完成
顯示 PreOrder 資料：5 1 4 9 6
顯示 PostOrder 資料：4 1 6 9 5
顯示 InOrder 資料：1 4 5 6 9
```

17.4.4 二元樹搜尋

下面範例是實現（implement）17.4.1 節的 searchNode 函數：首先定義搜尋節點指標，並令搜尋節點指標的起始位置等於根節點位置。然後比對搜尋資料與根節點資料是否相符，若相符則傳回 true。若搜尋資料小於根節點資料，則令搜尋指標等於左邊子節點指標，並繼續搜尋左邊副分支。若搜尋資料大於根節點資料，則令搜尋指標等於右邊子節點指標，並繼續搜尋右邊副分支。若全部都不相符則傳回 false。

```
bool BinaryTree::searchNode(int num)            //搜尋二元樹資料函數
{
   TreeNode *nodePtr = root;                    //令 nodePtr 等於 root
   while(nodePtr)                               //當 nodePtr 不等於 NULL
   {
      if(num == nodePtr->data)                  //若搜尋資料==節點資料
         return true;
      else if(num < nodePtr->data)              //若搜尋資料<節點資料
         nodePtr = nodePtr->left;
      else if(num > nodePtr->data)              //若搜尋資料>節點資料
         nodePtr = nodePtr->right;
   }
   return false;
}
```

程式 17-07：搜尋二元樹節點資料

```
1.  //檔案名稱：d:\C++17\C1707.cpp
2.  #include <iostream>
3.  using namespace std;
4.
5.  struct TreeNode {                           //定義 TreeNode 結構資料
6.     int data;
7.     TreeNode *left;
8.     TreeNode *right;
9.  };
10.
11. class BinaryTree {                          //定義二元樹資料類別
12.    TreeNode *root;
13.    void showInOrder(TreeNode *);            //顯示 InOrder 資料原型
14.    void showPreOrder(TreeNode *);           //顯示 PreOrder 資料原型
15.    void showPostOrder(TreeNode *);          //顯示 PostOrder 原型
16. public:
17.    BinaryTree() { root = NULL; }
18.    void insertNode(int);                    //插入節點函數原型
19.    void coutInOrder() { showInOrder(root); } //呼叫 InOrder 資料函數
20.    void coutPreOrder() { showPreOrder(root); } //呼叫 PreOrder 資料函數
21.    void coutPostOrder() { showPostOrder(root); }   //呼叫 PostOrder 函數
22.    bool searchNode(int num);                //搜尋資料函數原型
23. };
24.
25. void BinaryTree::insertNode(int num)       //插入節點函數
26. {
27.    TreeNode *newNode = new TreeNode;        //newNode 為新節點指標
28.    TreeNode *tempNode;                      //tempNode 為暫存節點
29.    newNode->left = newNode->right = NULL;   //令新節點左右指標=NULL
30.    newNode->data = num;                     //資料存入新節點的緩衝區
31.    if(root == NULL) {                       //若二元樹還沒有資料
```

```
32.        root = newNode;                          //令根節點=新節點
33.     } else {                                    //若二元樹中已有資料
34.        tempNode = root;                         //令 tempNode 指向 root
35.        while(tempNode) {                 //當 tempNode 不等於 NULL
36.           if(num < tempNode->data) {   //新資料 < tempNode->data
37.              if(tempNode->left) {      //若 tempNode->left != NULL
38.                  tempNode = tempNode->left; //tempNode->left 為臨時指標
39.              } else {                   //若 tempNode->left == NULL
40.                  tempNode->left = newNode; //新節點插入 tempNode->left
41.                  break;
42.              }
43.           } else if(num > tempNode->data) { //新資料 > tempNode->data
44.              if(tempNode->right) {   //若 tempNode->right!=NULL
45.                  tempNode = tempNode->right; //tempNode->right 為臨時指標
46.              } else {                     //若 tempNode->right==NULL
47.                  tempNode->right = newNode;  //新節點插入 tempNode->right
48.                  break;
49.              }
50.           } else {                         //新資料 == tempNode->data
51.              cout << "資料重複！";
52.              break;
53.           }
54.        }
55.     }
56.  }
57.
58.  void BinaryTree::showInOrder(TreeNode *nodePtr) //顯示 InOrder 資料函數
59.  {
60.     if(nodePtr)                              //若 nodePtr 不等於 NULL
61.     {
62.        showInOrder(nodePtr->left);           //遞迴呼叫，越過左邊分支
63.        cout << nodePtr->data << ' ';         //處理節點資料
64.        showInOrder(nodePtr->right);          //遞迴呼叫，越過右邊分支
65.     }
66.  }
67.
68.  void BinaryTree::showPreOrder(TreeNode *nodePtr) //顯示 PreOrder 資料函數
69.  {
70.     if(nodePtr)                              //若 nodePtr 不等於 NULL
71.     {
72.        cout << nodePtr->data << ' ';         //處理節點資料
73.        showPreOrder(nodePtr->left);          //遞迴呼叫，越過左邊分支
74.        showPreOrder(nodePtr->right);         //遞迴呼叫，越過右邊分支
75.     }
76.  }
77.
78.  void BinaryTree::showPostOrder(TreeNode *nodePtr)  //顯示 PostOrder 函數
79.  {
80.     if(nodePtr)                              //若 nodePtr 不等於 NULL
```

```
81.     {
82.        showPostOrder(nodePtr->left);     //遞迴呼叫，越過左邊分支
83.        showPostOrder(nodePtr->right);    //遞迴呼叫，越過右邊分支
84.        cout << nodePtr->data << ' ';     //處理節點資料
85.     }
86.  }
87.
88.  bool BinaryTree::searchNode(int num)    //搜尋二元樹資料函數
89.  {
90.     TreeNode *nodePtr = root;            //令 nodePtr 等於 root
91.     while(nodePtr)                       //當 nodePtr 不等於 NULL
92.     {
93.        if(num == nodePtr->data)          //若搜尋資料==節點資料
94.           return true;
95.        else if(num < nodePtr->data)      //若搜尋資料<節點資料
96.           nodePtr = nodePtr->left;
97.        else if(num > nodePtr->data)      //若搜尋資料>節點資料
98.           nodePtr = nodePtr->right;
99.     }
100.    return false;
101. }
102.
103. int main(int argc, char** argv)
104. {
105.    BinaryTree intTree;
106.    cout << "插入節點...";
107.    intTree.insertNode(5);               //呼叫 insertNode 函數
108.    intTree.insertNode(9);               //呼叫 insertNode 函數
109.    intTree.insertNode(1);               //呼叫 insertNode 函數
110.    intTree.insertNode(6);               //呼叫 insertNode 函數
111.    intTree.insertNode(4);               //呼叫 insertNode 函數
112.    cout << "完成";
113.
114.    cout << "\n 顯示 PreOrder 資料：";
115.    intTree.coutPreOrder();              //呼叫 coutPreNode 函數
116.    cout << "\n 顯示 PostOrder 資料：";
117.    intTree.coutPostOrder();             //呼叫 coutPostNode 函數
118.    cout << "\n 顯示 InOrder 資料：";
119.    intTree.coutInOrder();               //呼叫 coutInNode 函數
120.
121.    if(intTree.searchNode(3))            //呼叫 searchNode 函數
122.       cout << "\n 在二元樹資料結構中找到 3\n";
123.    else
124.       cout << "\n 在二元樹資料結構中找不到 3\n";
125.    return 0;
126. }
```

程式輸出

```
插入節點...完成
顯示 PreOrder 資料：5 1 4 9 6
顯示 PostOrder 資料：4 1 6 9 5
顯示 InOrder 資料：1 4 5 6 9
在二元樹資料結構中找不到 3
```

17.4.5 刪除節點

如果要刪除的節點是葉節點（leaf node），只要找到該葉節點的父節點（parent node），再將指向葉節點的指標設成 NULL，最後釋放被刪除的葉節點記憶體即可。

但如果要刪除的節點不是葉節點，則在刪除節點之前，必須先連結要刪除節點的父節點（parent node）與要刪除節點的子節點（child node），然後才可刪除該節點並釋放被刪除節點的記憶體。

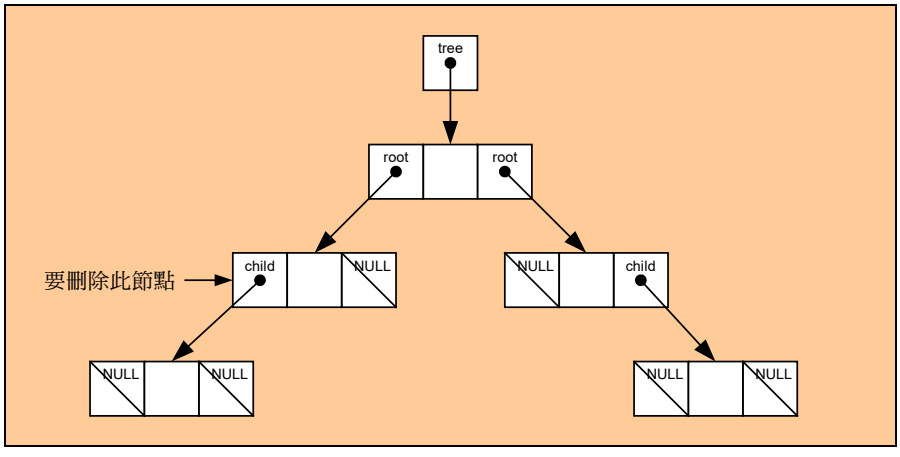

圖 17.16 刪除節點前

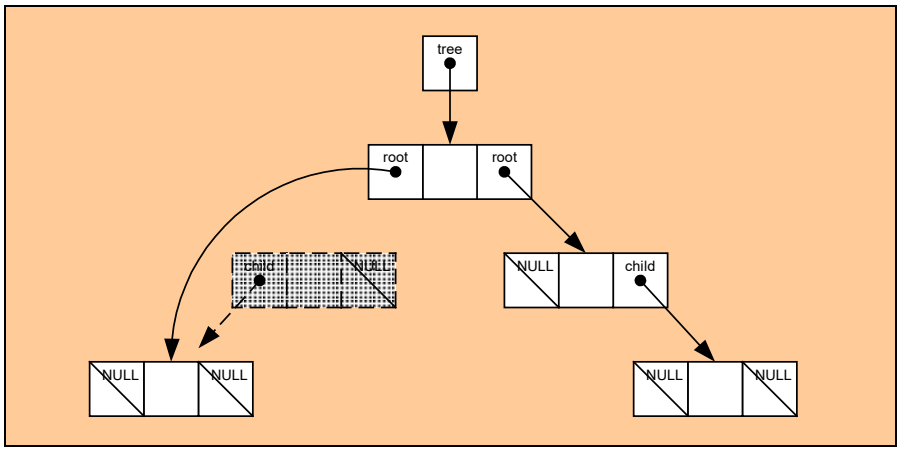

圖 17.17 連結後並刪除節點

圖 17.16 與圖 17.17 顯示要被刪除的節點只有一個子節點的情況。先連結要刪除節點的父節點與要被刪除節點的子節點後，即可刪除該節點並釋放該節點佔用的記憶體。

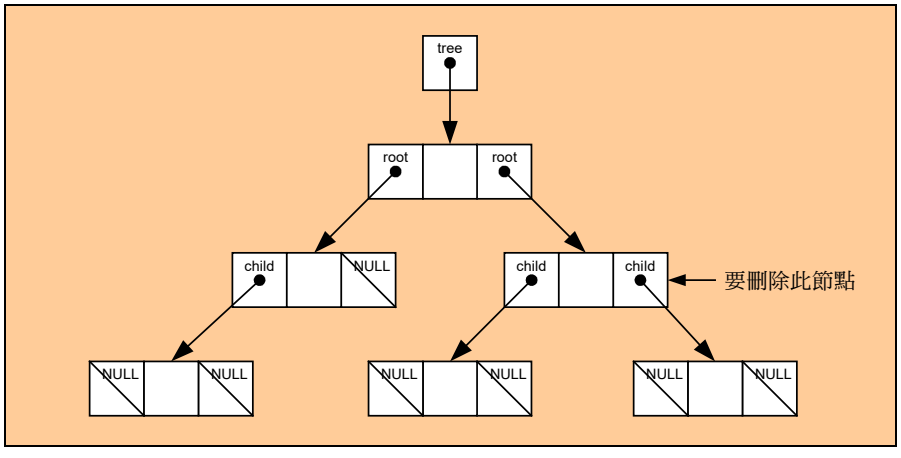

圖 17.18 刪除節點前

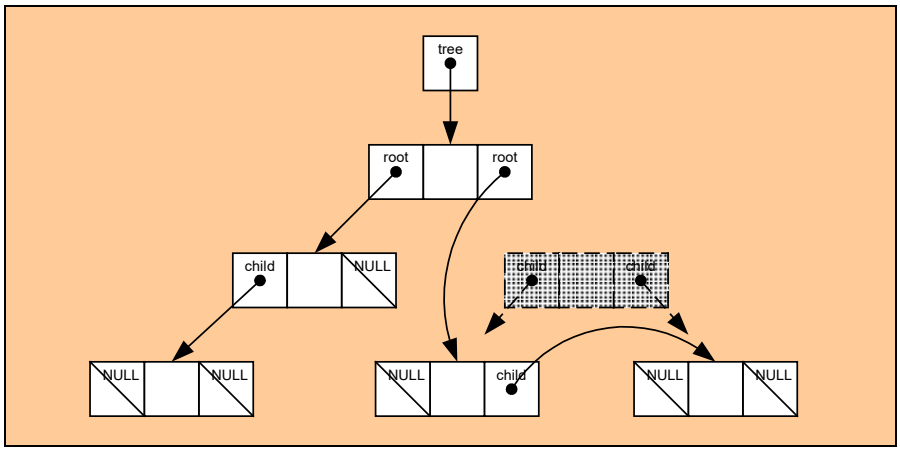

圖 17.19 連結後並刪除節點

圖 17.18 與圖 17.19 顯示要被刪除的節點有二個子節點。若要刪除的節點在右副分支（right subtree），則以要刪除節點的左邊子節點取代要刪除節點的位置，也就是先連結要刪除節點的父節點與要刪除節點的左邊子節點，再連結要刪除節點的左邊子節點到右邊子節點，最後即可刪除該節點並釋放該節點佔用的記憶體。

若要刪除的節點在左副分支（left subtree），則以要刪除節點的右邊子節點取代要刪除節點的位置，也就是先連結要刪除節點的父節點與要刪除節點的右邊子節點，再連結要刪除節點的右邊子節點到左邊子節點，最後刪除該節點並釋放記憶體即可。

下面範例是實現（implement）17.4.1 節的 deleteNode 函數，它是先搜尋要刪除的節點，找到後再依據上述刪除節點的原理，連結相關的節點後刪除該節點。

```
void BinaryTree::deleteNode(int num, TreeNode *&nodePtr)
{                                                    //刪除二元樹節點函數
   if(num < nodePtr->data)                           //若搜尋資料<節點資料
      deleteNode(num, nodePtr->left);                //遞迴呼叫 deleteNode
   else if(num > nodePtr->data)                      //若搜尋資料>節點資料
      deleteNode(num, nodePtr->right);               //遞迴呼叫 deleteNode
   else {                                            //若搜尋資料==節點資料
      TreeNode *tempNode;                            //建立臨時節點
      if(nodePtr == NULL)                            //若 nodePtr 等於 NULL
```

```
         cout << "不能刪除空節點！";
      else if(nodePtr->right == NULL)   //若 nodePtr->right 等於 NULL
      {
         tempNode = nodePtr;            //令 tempNode 要刪除的節點
         nodePtr = nodePtr->left;       //令 nodePtr = nodePtr->left
         delete tempNode;               //刪除 tempNode 節點
      }
      else if(nodePtr->left == NULL)    //若 nodePtr->left 等於 NULL
      {
         tempNode = nodePtr;            //令 tempNode 要刪除的節點
         nodePtr = nodePtr->right;      //令 nodePtr = nodePtr->right
         delete tempNode;               //刪除 tempNode 節點
      }
      else                              //以上皆非
      {
         tempNode = nodePtr->right;     //令 tempNode=nodePtr->right
         while(tempNode->left)          //若 tempNode->left 不等於 NULL
            tempNode = tempNode->left;  //令 tempNode=tempNode->left
         tempNode->left = nodePtr->left;
         tempNode = nodePtr;            //令 tempNode 要刪除的節點
         nodePtr = nodePtr->right;      //令 nodePtr = nodePtr->right
         delete tempNode;               //刪除 tempNode 節點
      }
   }
}
```

程式 17-08：刪除二元樹節點資料

```
1.  //檔案名稱：d:\C++17\C1708.cpp
2.  #include <iostream>
3.  using namespace std;
4.
5.  struct TreeNode {                           //定義 TreeNode 結構資料
6.     int data;
7.     TreeNode *left;
8.     TreeNode *right;
9.  };
10.
11. class BinaryTree {                          //定義二元樹資料類別
12.    TreeNode *root;
13.    void showInOrder(TreeNode *);            //顯示 InOrder 資料原型
14.    void showPreOrder(TreeNode *);           //顯示 PreOrder 資料原型
15.    void showPostOrder(TreeNode *);          //顯示 PostOrder 原型
16.    void deleteNode(int num, TreeNode *&nodePtr); //刪除節點函數原型
17. public:
18.    BinaryTree() { root = NULL; }
19.    void insertNode(int);                    //插入節點函數原型
20.    void coutInOrder() { showInOrder(root); }//呼叫 InOrder 資料函數
```

```
21.     void coutPreOrder() { showPreOrder(root); }  //呼叫 PreOrder 資料函數
22.     void coutPostOrder() { showPostOrder(root); }    //呼叫 PostOrder 函數
23.     bool searchNode(int num);                  //搜尋資料函數原型
24.     void removeNode(int num) { deleteNode(num, root); } //呼叫刪除節點原型
25.  };
26.
27.  void BinaryTree::insertNode(int num) //插入節點函數
28.  {
29.     TreeNode *newNode = new TreeNode; //newNode 為新節點指標
30.     TreeNode *tempNode;               //tempNode 為暫存節點
31.     newNode->left = newNode->right = NULL; //令新節點左右指標=NULL
32.     newNode->data = num;              //資料存入新節點的緩衝區
33.     if(root == NULL) {                //若二元樹還沒有資料
34.        root = newNode;                //令根節點=新節點
35.     } else {                          //若二元樹中已有資料
36.        tempNode = root;               //令 tempNode 指向 root
37.        while(tempNode) {              //當 tempNode 不等於 NULL
38.           if(num < tempNode->data) {  //新資料 < tempNode->data
39.              if(tempNode->left) {     //若 tempNode->left!=NULL
40.                 tempNode = tempNode->left; //tempNode->left 為臨時指標
41.              } else {                 //若 tempNode->left==NULL
42.                 tempNode->left = newNode; //新節點插入 tempNode->left
43.                 break;
44.              }
45.           } else if(num > tempNode->data) { //新資料 > tempNode->data
46.              if(tempNode->right) {         //若 tempNode->right!=NULL
47.                 tempNode = tempNode->right; //tempNode->right 為臨時指標
48.              } else {                      //若 tempNode->right==NULL
49.                 tempNode->right = newNode; //新節點插入 tempNode->right
50.                 break;
51.              }
52.           } else {                     //新資料 == tempNode->data
53.              cout << "資料重複！";
54.              break;
55.           }
56.        }
57.     }
58.  }
59.
60.  void BinaryTree::showInOrder(TreeNode *nodePtr) //顯示 InOrder 資料函數
61.  {
62.     if(nodePtr)                       //若 nodePtr 不等於 NULL
63.     {
64.        showInOrder(nodePtr->left);    //遞迴呼叫，越過左邊分支
65.        cout << nodePtr->data << ' ';  //處理節點資料
66.        showInOrder(nodePtr->right);   //遞迴呼叫，越過右邊分支
67.     }
68.  }
```

```
69.
70.  void BinaryTree::showPreOrder(TreeNode *nodePtr) //顯示 PreOrder 資料函數
71.  {
72.     if(nodePtr)                          //若 nodePtr 不等於 NULL
73.     {
74.        cout << nodePtr->data << ' ';  //處理節點資料
75.        showPreOrder(nodePtr->left);   //遞迴呼叫，越過左邊分支
76.        showPreOrder(nodePtr->right);  //遞迴呼叫，越過右邊分支
77.     }
78.  }
79.
80.  void BinaryTree::showPostOrder(TreeNode *nodePtr)    //顯示 PostOrder 函數
81.  {
82.     if(nodePtr)                          //若 nodePtr 不等於 NULL
83.     {
84.        showPostOrder(nodePtr->left);  //遞迴呼叫，越過左邊分支
85.        showPostOrder(nodePtr->right); //遞迴呼叫，越過右邊分支
86.        cout << nodePtr->data << ' ';  //處理節點資料
87.     }
88.  }
89.
90.  bool BinaryTree::searchNode(int num) //搜尋二元樹資料函數
91.  {
92.     TreeNode *nodePtr = root;         //令 nodePtr 等於 root
93.     while(nodePtr)                    //當 nodePtr 不等於 NULL
94.     {
95.        if(num == nodePtr->data)       //若搜尋資料==節點資料
96.           return true;
97.        else if(num < nodePtr->data)   //若搜尋資料<節點資料
98.           nodePtr = nodePtr->left;
99.        else if(num > nodePtr->data)   //若搜尋資料>節點資料
100.          nodePtr = nodePtr->right;
101.    }
102.    return false;
103. }
104.
105. void BinaryTree::deleteNode(int num, TreeNode *&nodePtr)
106. {                                     //刪除二元樹節點函數
107.    if(num < nodePtr->data)            //若搜尋資料<節點資料
108.       deleteNode(num, nodePtr->left);  //遞迴呼叫 deleteNode
109.    else if(num > nodePtr->data)       //若搜尋資料>節點資料
110.       deleteNode(num, nodePtr->right); //遞迴呼叫 deleteNode
111.    else {                             //若搜尋資料==節點資料
112.       TreeNode *tempNode;             //建立臨時節點
113.       if(nodePtr == NULL)             //若 nodePtr 等於 NULL
114.          cout << "不能刪除空節點！";
115.       else if(nodePtr->right == NULL) //若 nodePtr->right 等於 NULL
116.       {
```

```
117.          tempNode = nodePtr;          //令 tempNode 要刪除的節點
118.          nodePtr = nodePtr->left;     //令 nodePtr = nodePtr->left
119.          delete tempNode;             //刪除 tempNode 節點
120.       }
121.       else if(nodePtr->left == NULL) //若 nodePtr->left 等於 NULL
122.       {
123.          tempNode = nodePtr;          //令 tempNode 要刪除的節點
124.          nodePtr = nodePtr->right;    //令 nodePtr=nodePtr->right
125.          delete tempNode;             //刪除 tempNode 節點
126.       }
127.       else  //以上皆非
128.       {
129.          tempNode = nodePtr->right;   //令 tempNode=nodePtr->right
130.          while(tempNode->left)        //tempNode->left 不等於 NULL
131.             tempNode = tempNode->left;    //令 tempNode=左邊子節點
132.          tempNode->left = nodePtr->left;
133.          tempNode = nodePtr;          //tempNode 為要刪除節點
134.          nodePtr = nodePtr->right;    //令 nodePtr=nodePtr->right
135.          delete tempNode;             //刪除 tempNode 節點
136.       }
137.    }
138. }
139.
140. int main(int argc, char** argv)
141. {
142.    BinaryTree intTree;
143.    cout << "插入節點...";
144.    intTree.insertNode(5);            //呼叫 insertNode 函數
145.    intTree.insertNode(9);            //呼叫 insertNode 函數
146.    intTree.insertNode(1);            //呼叫 insertNode 函數
147.    intTree.insertNode(6);            //呼叫 insertNode 函數
148.    intTree.insertNode(4);            //呼叫 insertNode 函數
149.    cout << "完成";
150.    cout << "\n 顯示 PreOrder 資料：";
151.    intTree.coutPreOrder();           //呼叫 coutPreNode 函數
152.    cout << "\n 顯示 PostOrder 資料：";
153.    intTree.coutPostOrder();          //呼叫 coutPostNode 函數
154.    cout << "\n 顯示 InOrder 資料：";
155.    intTree.coutInOrder();            //呼叫 coutInNode 函數
156.
157.    if(intTree.searchNode(3))         //呼叫 searchNode 函數
158.       cout << "\n 在二元樹資料結構中找到 3\n";
159.    else
160.       cout << "\n 在二元樹資料結構中找不到 3\n";
161.
162.    cout << "\n 刪除節點 9";
163.    intTree.removeNode(9);            //呼叫 removeNode 函數
```

```
164.    cout << "\n 刪除節點 1";
165.    intTree.removeNode(1);              //呼叫 removeNode 函數
166.    cout << "\n 顯示 InOrder 資料：";
167.    intTree.coutInOrder();              //呼叫 coutInNode 函數
168.    cout << endl;
169.    return 0;
170. }
```

程式輸出

```
插入節點...完成
顯示 PreOrder 資料：5 1 4 9 6
顯示 PostOrder 資料：4 1 6 9 5
顯示 InOrder 資料：1 4 5 6 9
在二元樹資料結構中找不到 3

刪除節點 9
刪除節點 1
顯示 InOrder 資料：4 5 6
```

17.5 習題

實作題

1. 利用函數範本，寫一個二分搜尋的程式，此程式可以處理整數（int）與小數（float）的搜尋。本程式包含輸入、排序、搜尋與輸出等功能。
2. 利用串列資料（linked list），寫一程式包含 length(字串) 函數，模擬 strlen(字串) 計算字串長度的功能。也就是呼叫 length(字串) 函數時，將計算字串長度。