# 資料結構

# 20

*Much that I bound, I could not free; Much that I freed returned to me.*
—Lee Wilson Dodd
*'Will you walk a little faster?' said a whiting to a snail, 'There's a porpoise close behind us, and he's treading on my tail.'*
—Lewis Carroll
*There is always room at the top.*
—Daniel Webster
*Push on—keep moving.*
—Thomas Morton
*I'll turn over a new leaf.*
—Miguel de Cervantes

## 學習目標

在本章中，你將學到：

- 使用指標、自我參照類別和遞迴建立鏈結資料結構。
- 建立與操作動態資料結構，如鏈結串列、佇列、堆疊與二元樹。
- 使用二元搜尋樹進行高速搜尋與排序。
- 了解鏈結資料結構的各種重要應用。
- 了解如何使用類別樣板、繼承與組合，建立可重複使用的資料結構。

## 本章綱要



## 20.1 簡介

我們已學過固定大小的**資料結構 (data structures)**，如一維陣列和二維陣列。本章介紹執行期間會擴大和縮小的**動態資料結構 (dynamic data structures)**。**鏈結串列 (Linked lists)** 是在邏輯上「排成一列」的資料項集合，我們可在鏈結串列中的任何位置加入和移除資料項目。在編譯器和作業系統中，**堆疊 (stacks)** 是很重要的。我們只能在堆疊的某一端 (也就是堆疊的**頂端，top**) 插入和移除資料項目。**佇列 (Queues)** 代表等待區；我們可以在佇列的後端 (也就是**尾端，tail**) 加入資料項目，然後從佇列的前端 (也就是**頭端，head**) 移除資料項目。**二元樹 (Binary trees)** 可用來進行高速搜尋和排序資料、有效率的消除重複資料項目、顯示檔案系統目錄，以及將程式碼的運算式編譯成機器語言。這些資料結構有許多其它有趣的應用。

我們會探討數種常見、重要的資料結構，並實作程式來建立、使用它們。我們使用類別、類別樣板、繼承和組合來建立這些資料結構，並將它們包裝起來，以便再利用和維護。

本章可爲 22 章「標準樣板函式庫 (STL)」奠定堅實基礎。STL 是 C++ 標準函式庫的主要部分。STL 提供容器、走訪這些容器的循環器、以及處理這些容器內元素的演算法。STL 含有本章討論的每一種資料結構，並將它們包裝成樣板類別。STL 程式碼寫得很棒，可攜性、效率和擴充性極佳。當您了解資料結構的原理與建構方式後，便能妥善利用 STL 內預先包裝的資料結構、循環器與演算法，徹底發揮這個世界級再利用元件的潛力。

本章範例相當實際，可應用在進階課程或業界。這些程式採用大量的指標操作。習題包括許多實際的應用。

我們鼓勵您嘗試特別小節「建立自己的編譯器」中的主要專案。我們已經會用 C++ 編譯器將程式轉譯成機器語言，以在電腦上執行。在本專案中，您將親手打造自己的編譯器。它會讀取由敘述組成的檔案，此檔案由簡單且強大的高階語言所撰寫，類似早期常見的 BASIC 語言。此編譯器會將這些敘述轉譯成 Simpletron 機器語言 (Simpletron Machine Language，SML) 指令組成的檔案。SML 就是第 8 章的特別小節「建立自己的電腦」所學到的語言。然後，此 Simpletron 模擬器程式就會執行本編譯器產生的 SML 程式！特別小節會仔細探討這些高階語言的規格，並說明將每種高階語言敘述轉換成機器語言指令所需的演算法。本章習題中還有許多編譯器和 Simpletron 模擬器的強化部分。

## 20.2　自我參照類別

**自我參照類別 (self-referential class)** 包含一個指標成員，這個指標會指向相同類別型別的類別物件。例如，以下的定義

```
class Node
{
public:
   Node( int ); // constructor
   void setData( int ); // set data member
   int getData() const; // get data member
   void setNextPtr( Node * ); // set pointer to next Node
   Node *getNextPtr() const; // get pointer to next Node
private:
   int data; // data stored in this Node
   Node *nextPtr; // pointer to another object of same type
}; // end class Node
```

會定義一個型別 Node。型別 Node 有兩個 private 資料成員，也就是整數成員 data 和指標成員 nextPtr。成員 nextPtr 指向型別 Node 的物件，這個物件的型別和此處宣告的物件相同，因此稱爲「自我參照類別」。成員 nextPtr 可視爲一種**鏈結 (link)**，也就是說，nextPtr 可將型別 Node 的物件和另一個相同型別的物件「鏈結」起來。Node 型別亦有五個成員函式：一個建構子，它接收一個整數以初始化 data 成員；一個 setData 函式以設定 data 成員的值；一個 getData 函式以傳回 data 成員的值；一個 setNextPtr 函式以設定 nextPtr 成員的值；以及一個 getNextPtr 函式以傳回 nextPtr 成員的值。

自我參照類別物件能彼此鏈結，形成有用的資料結構，如串列、佇列、堆疊和樹。圖 20.1 說明二個自我參照的類別物件鏈結成一個串列。請注意，圖中的斜線表示空指標 (0)，也就是放在第二個自我參照類別物件的鏈結成員，表示該鏈結不會再指向另一個物件。此斜線只是說明之用，不是對應到 C++中的反斜線字元。空指標通常代表資料結構的結束，就像空字元 ('\0') 代表字串結束。

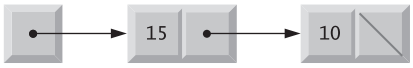


圖 20.1　二個連在一起的自我參照類別物件

**常見的程式設計錯誤 20.1**

沒有將串列資料結構中最後一個節點的鏈結設成空指標 (0)，是一種邏輯錯誤 (甚至可能是致命的邏輯錯誤)。

## 20.3 動態記憶體配置與資料結構

要建立和維護動態資料結構，需使用動態記憶體配置 (dynamic memory allocation)，就是讓程式在執行期取得更多記憶體存放新節點。當程式不再需要記憶體時就予以釋放，以便未來重複使用，配置給其它物件。動態記憶體配置的大小限制爲電腦可用的實體記憶體，或虛擬記憶體系統中的可用虛擬記憶體大小。因爲可用記憶體由許多程式共享，所以可用上限通常會小很多。

運算子 new 接受的引數型別，就是要進行動態配置的物件型別，它會傳回指向該型別物件的指標。例如，以下的敘述

```
Node *newPtr = new Node( 10 ); // create Node with data 10
```

會配置 sizeof (Node) 位元組的空間、執行 Node 建構子、並將新 Node 物件的位址指派給 newPtr。如果沒有可配置的記憶體，new 就會拋出 bad_alloc 例外。10 會傳給 Node 的建構子，將 Node 的 data 成員初始化成 10。

運算子 delete 會執行 Node 解構子，並回收 new 所配置的記憶體交還給系統，以便稍後再行配置。要釋放先前 new 動態配置的記憶體，可使用以下敘述

```
delete newPtr;
```

請注意，newPtr 本身並沒有被刪除；而是 `newPtr` 指向的記憶體空間被刪除了。若指標 `newPtr` 的值是 0 (空指標)，上面的敘述便沒有作用。`delete` 空指標並不會出錯。

以下幾個小節會討論串列、堆疊、佇列和樹。本章介紹的資料結構，均以動態記憶體配置、自我參照類別、類別樣板與函式樣板進行建立與維護。

## 20.4　鏈結串列

鏈結串列是自我參照類別物件的線性集合，每個物件稱爲**節點 (node)**，它們透過**指標鏈結 (pointer links)** 串起來，因此稱爲「鏈結」串列。我們可用指向串列第一個節點的指標來存取鏈結串列。至於後續節點，則透過前一個節點內的鏈結指標成員來存取。習慣上，最後一個節點的鏈結指標會設爲空指標 (0)，代表串列結束。資料會動態儲存在鏈結串列中，依需要建立每個節點。節點可包含任意型別的資料，包括其它類別的物件。若節點包含基本類別的指標，而這個指標指向基本類別和衍生類別物件 (透過繼承)，則我們可建立這類節點的鏈結串列，並使用 `virtual` 函式呼叫，以多型方式處理這些物件。堆疊和佇列也是**線性資料結構 (linear data structures)**，稍後我們將看到，它可算是鏈結串列的一種特殊化版本。樹是**非線性的資料結構 (nonlinear data structures)**。

資料串列可儲存在陣列中，但鏈結串列有幾項優點。無法預知資料結構中有多少資料項目時，鏈結串列便很適用。鏈結串列是動態的，所以串列長度能依需要增加或減少。但「傳統」C++陣列大小是無法改變的，因爲陣列大小在編譯時期就定死了。「傳統」陣列會塞滿。但唯有系統記憶體不足，無法滿足動態記憶體配置要求時，鏈結串列才會塞滿。

**增進效能的小技巧 20.1**

陣列可宣告比預期項目更多的元素，但會浪費記憶體。此時，鏈結串列更能有效使用記憶體。鏈結串列可讓程式在執行期間調整記憶體用量。類別樣板 `vector` (於第 7.11 節介紹) 實作了動態的陣列資料結構，可重新調整大小。

我們可將新元素插入串列的適當位置，以維持鏈結串列的排序。不需移動現有串列元素。只需要將指標更新，指向正確的節點。

**增進效能的小技巧 20.2**

在排序過的陣列中加入和刪除元素都很耗時間，因爲排在加入和刪除元素之後的所有元素都要移動。鏈結串列可以在任意的地方插入新節點，非常快速。

**增進效能的小技巧 20.3**

陣列所有元素都連續地儲存在記憶體中。因爲任何陣列元素的位址，都可依據它相對於陣列的起始位置直接算出來，所以程式可直接存取任何陣列元素。鏈結串列就無法立即「直接存取」元素。所以，在鏈結串列中存取單一元素的成本，就比陣列高得多。我們要考量程式特定操作的效能，以及資料結構的資料項目規模，來選用資料結構。例如，在排序過的鏈結串列中插入項目，通常就比排序過的陣列要快。

鏈結串列節點通常不是連續地儲存在記憶體中。但在邏輯上，鏈結串列節點看來是連續的。圖 20.2 是有數個節點的鏈結串列。

**增進效能的小技巧 20.4**

若資料結構在執行期會增大和縮小，採用動態記憶體配置 (而不是固定大小的陣列)，可節省記憶體空間。但請記住，指標也會佔空間，且動態記憶體配置會造成函式呼叫的額外負擔。

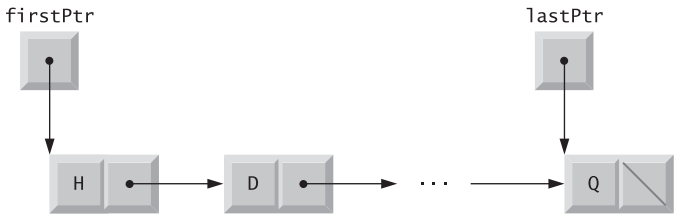


圖 20.2 串列圖示

## 鏈結串列實作

圖 20.3-20.5 的程式使用 `List` 類別樣板 (請參閱第 14 章得知類別樣板的相關資訊) 操作一個整數串列，和一個浮點數串列。測試程式 (圖 20.5) 提供五個選項：1) 在串列起始位置插入一個數值，2) 在串列結束位置插入一個數值，3) 從串列起始位置移除一個數值，4) 從串列結束位置移除一個數值，5) 結束串列處理。後面會詳細討論本程式。習題 20.20 要求您實作一個遞迴函式，以反方向印出鏈結串列的內容，而習題 20.21 要求您實作一個遞迴函式，搜尋鏈結串列中特定的資料項目。

程式使用類別樣板 `ListNode` (圖 20.3) 和 `List` (圖 20.4)。每個 `List` 物件封裝一個由 `ListNode` 物件組成的鏈結串列。`ListNode` 類別樣板 (圖 20.3) 包含 `private` 成員 `data` 和 `nextPtr` (第 19-20 行)、一個初始化這些成員的建構子、以及傳回節點資料的 `getData` 函式。成員 `data` 儲存一個 `NODETYPE` 型別的數值，`NODETYPE` 是傳給

類別樣板的型別參數。成員 nextPtr 則儲存一個指標，指向鏈結串列的下一個 ListNode 物件。ListNode 類別樣板定義的第 13 行將 List <NODETYPE> 類別宣告爲 friend。如此可讓 List 特殊化類別樣板的所有成員函式，成爲相對應 ListNode 特殊化類別樣板的夥伴，好讓它們存取該型別 ListNode 物件的 private 成員。因爲 ListNode 的樣板參數 NODETYPE 是 List 的樣板引數，所以特定型別的特殊化 ListNode，只能由相同型別的特殊化 List 處理 (例如，int 型別的 List 才能管理 int 數值的 ListNode 物件)。

```
1  // Fig. 20.3: ListNode.h
2  // Template ListNode class definition.
3  #ifndef LISTNODE_H
4  #define LISTNODE_H
5
6  // forward declaration of class List required to announce that class
7  // List exists so it can be used in the friend declaration at line 13
8  template< typename NODETYPE > class List;
9
10 template< typename NODETYPE >
11 class ListNode
12 {
13    friend class List< NODETYPE >; // make List a friend
14
15 public:
16    ListNode( const NODETYPE & ); // constructor
17    NODETYPE getData() const; // return data in node
18 private:
19    NODETYPE data; // data
20    ListNode< NODETYPE > *nextPtr; // next node in list
21 }; // end class ListNode
22
23 // constructor
24 template< typename NODETYPE>
25 ListNode< NODETYPE >::ListNode( const NODETYPE &info )
26    : data( info ), nextPtr( 0 )
27 {
28    // empty body
29 } // end ListNode constructor
30
31 // return copy of data in node
32 template< typename NODETYPE >
33 NODETYPE ListNode< NODETYPE >::getData() const
34 {
35    return data;
36 } // end function getData
37
38 #endif
```

圖 20.3　ListNode 類別樣板定義

類別樣板 List 的第 23-24 行 (圖 20.4) 宣告 private 資料成員 firstPtr (指向 List 第一個 ListNode 的指標) 和 lastPtr (指向 List 最後一個 ListNode 的指標)。預設建構子 (第 31–36 行) 會將兩個指標都初始化成 0 (null)。清除 List 物件時，解構子 (第 39–59 行) 會確保已將 List 物件中的所有 ListNode 物件清除。List 的主要函式有 insertAtFront (第 62–74 行)、insertAtBack (第 77–89 行)、removeFromFront (第 92–110 行) 和 removeFromBack (第 113–140 行)。

函式 isEmpty (第 143–147 行) 稱爲判斷函式 (predicate function)，它不會變更 List，只是判斷 List 是否爲空的 (也就是指向 List 第一個節點的指標是否爲 null)。如果 List 是空的，程式就傳回 true；否則就傳回 false。函式 print (158–178 行) 會顯示 List 的內容。工具函式 getNewNode (150–155 行) 會傳回一個動態配置的 ListNode 物件。insertAtFront 和 insertAtBack 函式會呼叫此函式。

**測試和除錯的小技巧 20.1**

將新節點的鏈結成員設爲空指標 (0)。使用指標前，應設定其初始值。

```
1  // Fig. 20.4: List.h
2  // Template List class definition.
3  #ifndef LIST_H
4  #define LIST_H
5
6  #include <iostream>
7  #include "ListNode.h" // ListNode class definition
8  using namespace std;
9
10 template< typename NODETYPE >
11 class List
12 {
13 public:
14    List(); // constructor
15    ~List(); // destructor
16    void insertAtFront( const NODETYPE & );
17    void insertAtBack( const NODETYPE & );
18    bool removeFromFront( NODETYPE & );
19    bool removeFromBack( NODETYPE & );
20    bool isEmpty() const;
21    void print() const;
22 private:
23    ListNode< NODETYPE > *firstPtr; // pointer to first node
24    ListNode< NODETYPE > *lastPtr; // pointer to last node
25
26    // utility function to allocate new node
27    ListNode< NODETYPE > *getNewNode( const NODETYPE & );
```

圖 20.4 List 類別樣板定義

```
28 }; // end class List
29
30 // default constructor
31 template< typename NODETYPE >
32 List< NODETYPE >::List()
33    : firstPtr( 0 ), lastPtr( 0 )
34 {
35    // empty body
36 } // end List constructor
37
38 // destructor
39 template< typename NODETYPE >
40 List< NODETYPE >::~List()
41 {
42    if ( !isEmpty() ) // List is not empty
43    {
44       cout << "Destroying nodes ...\n";
45
46       ListNode< NODETYPE > *currentPtr = firstPtr;
47       ListNode< NODETYPE > *tempPtr;
48
49       while ( currentPtr != 0 ) // delete remaining nodes
50       {
51          tempPtr = currentPtr;
52          cout << tempPtr->data << '\n';
53          currentPtr = currentPtr->nextPtr;
54          delete tempPtr;
55       } // end while
56    } // end if
57
58    cout << "All nodes destroyed\n\n";
59 } // end List destructor
60
61 // insert node at front of list
62 template< typename NODETYPE >
63 void List< NODETYPE >::insertAtFront( const NODETYPE &value )
64 {
65    ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node
66
67    if ( isEmpty() ) // List is empty
68       firstPtr = lastPtr = newPtr; // new list has only one node
69    else // List is not empty
70    {
71       newPtr->nextPtr = firstPtr; // point new node to previous 1st node
72       firstPtr = newPtr; // aim firstPtr at new node
73    } // end else
74 } // end function insertAtFront
75
76 // insert node at back of list
77 template< typename NODETYPE >
78 void List< NODETYPE >::insertAtBack( const NODETYPE &value )
79 {
80    ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node
```

圖 20.4　`List` 類別樣板定義 (續 1)

```
81
82       if ( isEmpty() ) // List is empty
83          firstPtr = lastPtr = newPtr; // new list has only one node
84       else // List is not empty
85       {
86          lastPtr->nextPtr = newPtr; // update previous last node
87          lastPtr = newPtr; // new last node
88       } // end else
89    } // end function insertAtBack
90
91    // delete node from front of list
92    template< typename NODETYPE >
93    bool List< NODETYPE >::removeFromFront( NODETYPE &value )
94    {
95       if ( isEmpty() ) // List is empty
96          return false; // delete unsuccessful
97       else
98       {
99          ListNode< NODETYPE > *tempPtr = firstPtr; // hold tempPtr to delete
100
101         if ( firstPtr == lastPtr )
102            firstPtr = lastPtr = 0; // no nodes remain after removal
103         else
104            firstPtr = firstPtr->nextPtr; // point to previous 2nd node
105
106         value = tempPtr->data; // return data being removed
107         delete tempPtr; // reclaim previous front node
108         return true; // delete successful
109      } // end else
110   } // end function removeFromFront
111
112   // delete node from back of list
113   template< typename NODETYPE >
114   bool List< NODETYPE >::removeFromBack( NODETYPE &value )
115   {
116      if ( isEmpty() ) // List is empty
117         return false; // delete unsuccessful
118      else
119      {
120         ListNode< NODETYPE > *tempPtr = lastPtr; // hold tempPtr to delete
121
122         if ( firstPtr == lastPtr ) // List has one element
123            firstPtr = lastPtr = 0; // no nodes remain after removal
124         else
125         {
126            ListNode< NODETYPE > *currentPtr = firstPtr;
127
128            // locate second-to-last element
129            while ( currentPtr->nextPtr != lastPtr )
130               currentPtr = currentPtr->nextPtr; // move to next node
131
132            lastPtr = currentPtr; // remove last node
133            currentPtr->nextPtr = 0; // this is now the last node
```

圖 20.4　`List` 類別樣板定義 (續 2)

```
134         } // end else
135
136         value = tempPtr->data; // return value from old last node
137         delete tempPtr; // reclaim former last node
138         return true; // delete successful
139      } // end else
140   } // end function removeFromBack
141
142   // is List empty?
143   template< typename NODETYPE >
144   bool List< NODETYPE >::isEmpty() const
145   {
146      return firstPtr == 0;
147   } // end function isEmpty
148
149   // return pointer to newly allocated node
150   template< typename NODETYPE >
151   ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
152      const NODETYPE &value )
153   {
154      return new ListNode< NODETYPE >( value );
155   } // end function getNewNode
156
157   // display contents of List
158   template< typename NODETYPE >
159   void List< NODETYPE >::print() const
160   {
161      if ( isEmpty() ) // List is empty
162      {
163         cout << "The list is empty\n\n";
164         return;
165      } // end if
166
167      ListNode< NODETYPE > *currentPtr = firstPtr;
168
169      cout << "The list is: ";
170
171      while ( currentPtr != 0 ) // get element data
172      {
173         cout << currentPtr->data << ' ';
174         currentPtr = currentPtr->nextPtr;
175      } // end while
176
177      cout << "\n\n";
178   } // end function print
179
180   #endif
```

圖 20.4　`List` 類別樣板定義 (續 3)

圖 20.5 的第 69 行和 73 行會分別建立 `int` 和 `double` 型別的 `List` 物件。第 70 行和 74 行會在這些 `List` 物件上呼叫 `testList` 函式樣板。

```
1  // Fig. 20.5: Fig21_05.cpp
2  // List class test program.
3  #include <iostream>
4  #include <string>
5  #include "List.h" // List class definition
6  using namespace std;
7
8  // display program instructions to user
9  void instructions()
10 {
11    cout << "Enter one of the following:\n"
12       << "  1 to insert at beginning of list\n"
13       << "  2 to insert at end of list\n"
14       << "  3 to delete from beginning of list\n"
15       << "  4 to delete from end of list\n"
16       << "  5 to end list processing\n";
17 } // end function instructions
18
19 // function to test a List
20 template< typename T >
21 void testList( List< T > &listObject, const string &typeName )
22 {
23    cout << "Testing a List of " << typeName << " values\n";
24    instructions(); // display instructions
25
26    int choice; // store user choice
27    T value; // store input value
28
29    do // perform user-selected actions
30    {
31       cout << "? ";
32       cin >> choice;
33
34       switch ( choice )
35       {
36          case 1: // insert at beginning
37             cout << "Enter " << typeName << ": ";
38             cin >> value;
39             listObject.insertAtFront( value );
40             listObject.print();
41             break;
42          case 2: // insert at end
43             cout << "Enter " << typeName << ": ";
44             cin >> value;
45             listObject.insertAtBack( value );
46             listObject.print();
47             break;
48          case 3: // remove from beginning
49             if ( listObject.removeFromFront( value ) )
50                cout << value << " removed from list\n";
51
52             listObject.print();
```

圖 20.5　操作鏈結串列

```
53             break;
54          case 4: // remove from end
55             if ( listObject.removeFromBack( value ) )
56                cout << value << " removed from list\n";
57
58             listObject.print();
59             break;
60       } // end switch
61    } while ( choice < 5 ); // end do...while
62
63    cout << "End list test\n\n";
64 } // end function testList
65
66 int main()
67 {
68    // test List of int values
69    List< int > integerList;
70    testList( integerList, "integer" );
71
72    // test List of double values
73    List< double > doubleList;
74    testList( doubleList, "double" );
75 } // end main
```

```
Testing a List of integer values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter integer: 1
The list is: 1

? 1
Enter integer: 2
The list is: 2 1

? 2
Enter integer: 3
The list is: 2 1 3
```

```
? 2
Enter integer: 4
The list is: 2 1 3 4

? 3
2 removed from list
The list is: 1 3 4

? 3
1 removed from list
The list is: 3 4
```

圖 20.5　操作鏈結串列 (續 1)

```
? 4
4 removed from list
The list is: 3

? 4
3 removed from list
The list is empty

? 5
End list test

Testing a List of double values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter double: 1.1
The list is: 1.1

? 1
Enter double: 2.2
The list is: 2.2 1.1

? 2
Enter double: 3.3
The list is: 2.2 1.1 3.3

? 2
Enter double: 4.4
The list is: 2.2 1.1 3.3 4.4

? 3
2.2 removed from list
The list is: 1.1 3.3 4.4

? 3
1.1 removed from list
The list is: 3.3 4.4

? 4
4.4 removed from list
The list is: 3.3
```

```
? 4
3.3 removed from list
The list is empty

? 5
End list test

All nodes destroyed

All nodes destroyed
```

圖 20.5　操作鏈結串列 (續 2)

### 成員函式 `insertAtFront`

接下來，我們會詳細討論 `List` 類別的每一個成員函式。函式 `insertAtFront` (圖 20.4 的第 62–74 行) 會在串列前端加入新節點。此函式包含幾個步驟：

1. 呼叫函式 `getNewNode` (第 65 行)，將 `value` 傳給此函式，`value` 是一個常數參照，指向要插入的節點數值。
2. 函式 `getNewNode` (第 150–155 行) 使用運算子 `new` 建立一個新的串列節點，並傳回指向此新配置節點的指標，該指標會指定給 `insertAtFront` 的 `newPtr` (第 65 行)。
3. 如果串列是空的 (第 67 行)，則 `firstPtr` 和 `lastPtr` 都會設定成 `newPtr` (第 68 行)，換句話說，第一個和最後一個節點是同一個節點。
4. 如果串列不是空的 (第 69 行)，則將 `firstPtr` 複製到 `newPtr->nextPtr` (第 71 行)，以將 `newPtr` 指向的節點加入串列中，因此新節點就會指到原本的第一個節點，然後將 `newPtr` 複製到 `firstPtr` (第 72 行)，讓 `firstPtr` 指向新的第一個節點。

圖 20.6 說明函式 `insertAtFront`。(a) 部分顯示呼叫 `insertAtFront` 之前的串列和新節點。(b) 部分的虛線表示 `insertAtFront` 操作的步驟 4，讓內容爲 12 的節點成爲新的串列前端。

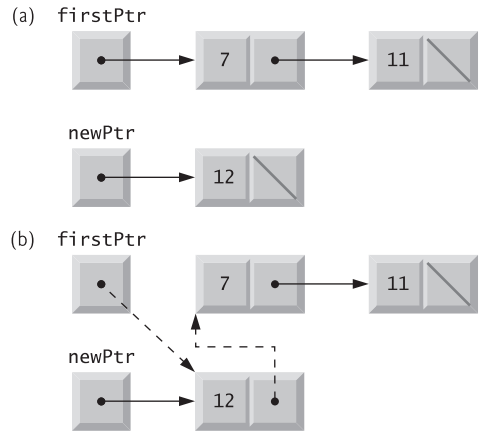


圖 20.6　`insertAtFront` 操作的圖示

### 成員函式 `insertAtBack`

函式 `insertAtBack` (圖 20.4 的第 77–89 行) 會在串列末端加入新節點。此函式包含幾個步驟：

1. 呼叫函式 `getNewNode` (第 80 行)，將 `value` 傳給此函式，`value` 是一個常數參照，指向要插入的節點數值。
2. 函式 `getNewNode` (第 150–155 行) 使用運算子 `new` 建立一個新的串列節點，並傳回指向此新配置節點的指標，該指標會指定給 `insertAtBack` 的 `newPtr` (第 80 行)。
3. 如果串列是空的 (第 82 行)，則 `firstPtr` 和 `lastPtr` 都會設定成 `newPtr` (第 83 行)。
4. 如果串列不是空的 (第 84 行)，則將 `newPtr` 複製到 `lastPtr->nextPtr` (第 86 行)，以將 `newPtr` 指向的節點加入串列中，因此原本的最後一個節點就會指到新節點，然後將 `newPtr` 複製到 `lastPtr` (第 87 行)，讓 `lastPtr` 指向新的最後一個節點。

圖 20.7 說明 `insertAtBack` 操作。圖中 (a) 部分顯示 `insertAtBack` 操作之前的串列和新節點。(b) 部分的虛線表示函式 `insertAtBack` 的步驟 4，可將新節點加入非空串列的末端。

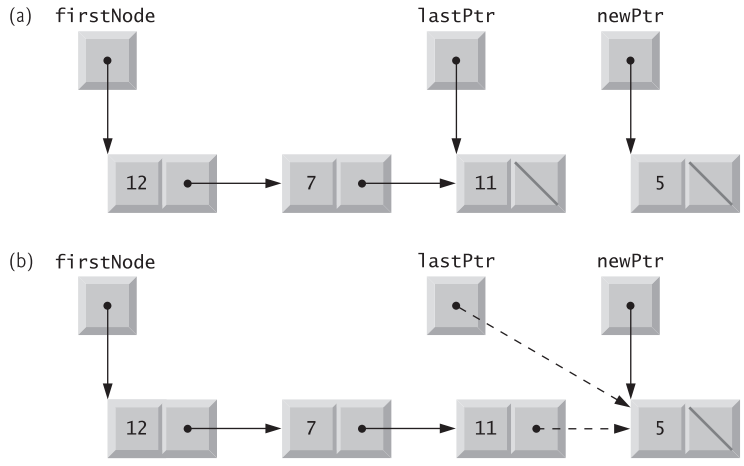


圖 20.7 `insertAtBack` 操作的圖示

### 成員函式 `removeFromFront`

函式 `removeFromFront` (圖 20.4 的第 92–110 行) 會將串列前面的節點移除，並將該節點的數值複製到參照參數。若從空串列移除一個節點，則函式會傳回 `false` (第 95–96 行)，若移除成功，函式就傳回 `true`。此函式包含幾個步驟：

1. 將 `firstPtr` 所指向的位址指定給 `tempPtr` (第 99 行)。最後會用 `tempPtr` 刪掉要移除的節點。
2. 如果 `firstPtr` 等於 `lastPtr` (第 101 行)，也就是移除節點之前，串列只有一個元素，則將 `firstPtr` 和 `lastPtr` 設爲零 (第 102 行)，以從串列中移除此節點 (留下空串列)。
3. 若移除之前，串列有超過一個的節點，則 `lastPtr` 保留不變，而將 `firstPtr` 設爲 `firstPtr->nextPtr` (第 104 行)，也就是修改 `firstPtr`，讓它指向移除節點之前的第二個節點 (現在是新的第一節點)。
4. 在這些指標操作完成之後，將被移除節點的 `data` 成員複製給參照參數 value (第 106 行)。
5. 現在用 `delete` 將 `tempPtr` 指向的節點刪除 (第 107 行)。
6. 傳回 `true`，表示移除成功 (第 108 行)。

圖 20.8 說明函式 `removeFromFront`。(a) 部分說明進行移除操作之前的串列。(b) 部分顯示實際的指標操作，可從非空串列中移除前端節點。

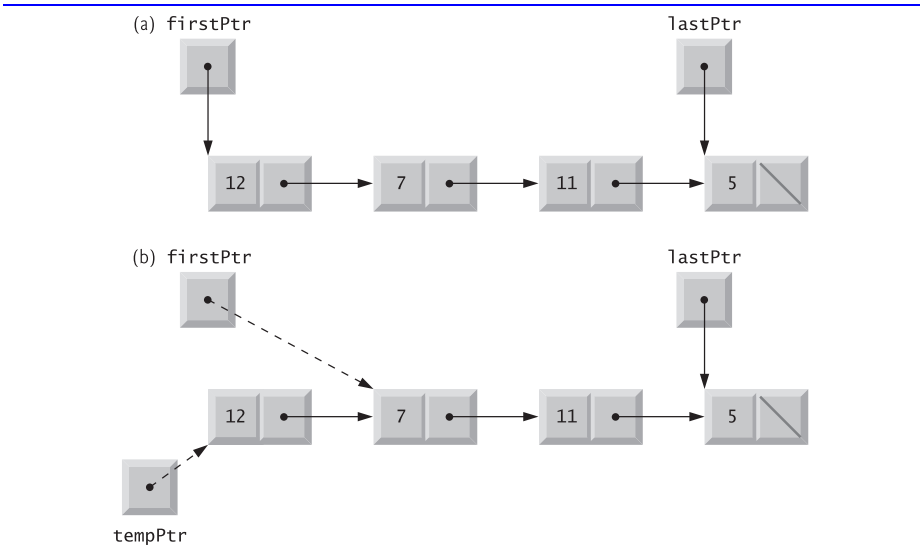


圖 20.8　`removeFromFront` 操作的圖示

### 成員函式 removeFromBack

函式 removeFromBack (圖 20.4 的第 113–140 行) 會將串列後面的節點移除，並將該節點的數值複製到參照參數。若從空串列移除一個節點，則函式會傳回 false (第 116–117 行)，若移除成功，函式就傳回 true。此函式包含幾個步驟：

1. 將 lastPtr 所指向的位址指定給 tempPtr (第 120 行)。最後會用 tempPtr 刪掉要移除的節點。
2. 如果 firstPtr 等於 lastPtr (第 122 行)，也就是移除節點之前，串列只有一個元素，則將 firstPtr 和 lastPtr 設爲零 (第 123 行)，以從串列中移除此節點 (留下空串列)。
3. 若移除之前，串列有兩個以上的節點，就將 currentPtr 設爲 firstPtr 指向的位址 (第 126 行)，準備「走訪串列」。
4. 現在用 currentPtr「走訪串列」，直到它指向倒數第二個節點。移除操作完成後，此節點就變成最後一個節點。這裡使用 while 迴圈 (第 129–130 行) 完成這項工作，當 currentPtr->nextPtr 不是 lastPtr 時，迴圈會重複將 currentPtr 取代成 currentPtr－>nextPtr。
5. 將 lastPtr 設爲 currentPtr 指向的位址 (第 132 行)，以將最末端節點從串列移除。
6. 將新末端節點的 currentPtr->nextPtr 設爲零 (133 行)。
7. 在這些指標操作完成之後，將被移除節點的 data 成員複製給參照參數 value (第 136 行)。
8. 現在用 delete 將 tempPtr 指向的節點刪除 (第 137 行)。
9. 傳回 true (第 138 行)，表示移除成功。

圖 20.9 說明 removeFromBack。(a) 部分說明進行移除操作之前的串列。(b) 部分顯示實際的指標操作。

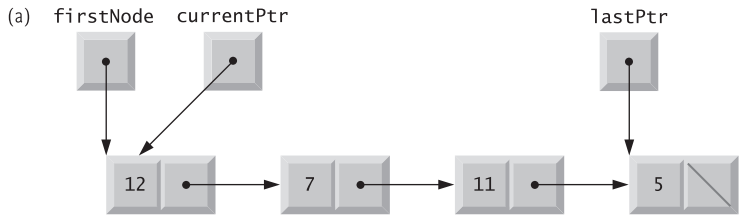


圖 20.9 removeFromBack 操作的圖示

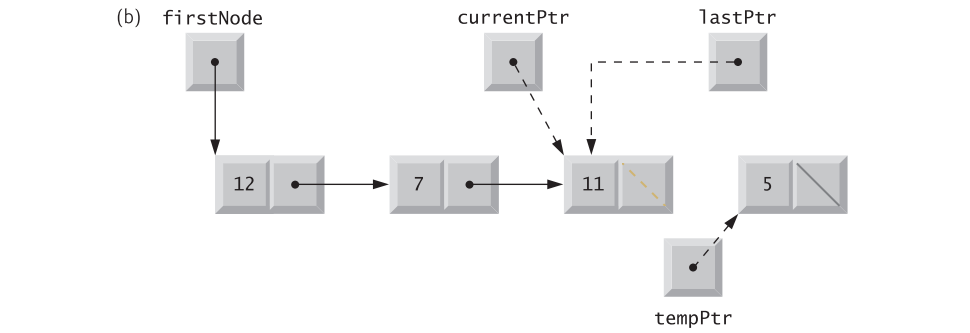


圖 20.9　`removeFromBack` 操作的圖示 (續)

### 成員函式 `print`

`print` 函式 (第 158–178 行) 會先判斷串列是不是空的 (第 161 行)。若是空的，就印出「`The list is empty`」並傳回 (第 163–164 行)。否則便走訪此串列，印出每個節點內容。函式會將 `currentPtr` 的初始值設爲 `firstPtr` 的副本 (第 167 行)，然後印出字串「`The list is:`」(第 169 行)。當 `currentPtr` 不是 null 時 (第 171 行)，就會印出 `currentPtr->data` (第 173 行)，並將 `currentPtr` 設爲 `currentPtr->nextPtr` (第 174 行)。請注意，若串列最後一個節點的 `nextPtr` 不是 null，列印演算法就會出錯，把串列末端之後的資料也一併列印出來。鏈結串列、堆疊、佇列的列印演算法完全相同 (因爲都採用相同的鏈結串列基礎結構)。

### 環狀鏈結串列與雙重鏈結串列

前面討論過的鏈結串列都是**單一鏈結串列 (singly linked list)**，也就是說，串列以一個指向第一節點的指標開始，每個節點都包含一個指標，指向串列的下一個節點。末端節點的指標成員值爲 0，串列也到此結束。單一鏈結串列只能以單向走訪。

**環狀單一鏈結串列** (**circular, singly linked list**，圖 20.10) 是以指向第一節點的指標開始，每個節點包含的指標都指向下一個節點。「最後節點」包含的並不是 null 指標，反之，該指標會指回第一個節點，因此串列會形成封閉的「環狀」。

**雙重鏈結串列** (**doubly linked list**，圖 20.11) 可正向、反向走訪串列。這種串列通常實作成有兩個「起始指標」，一個指向串列的第一個元素，以從前往後走訪串列，另一個指摽則指向串列的最後一個元素，以從後往前走訪串列。每個節點都有一個順向指標 (指到串列中順向的下一個節點) 和逆向指標 (指向串列中逆向的下一個節點)。假設

串列包含一個以字母順序排列的電話目錄，若要搜尋某人的名字，而此名字的第一個字母是英文的前幾個字母，則最好從串列前端開始搜尋。若某人名字的第一個字母是英文的後幾個字母，則最好從串列末端開始搜尋。

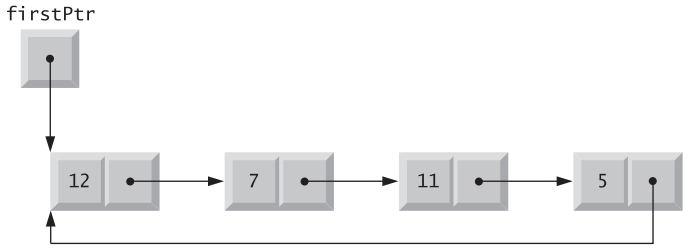


圖 20.10　環狀單一鏈結串列

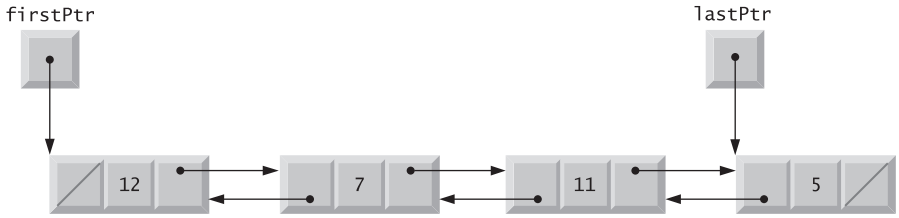


圖 20.11　雙重鏈結串列

在**環狀雙重鏈結串列** (**circular, doubly linked list**，圖 20.12) 中，最後節點的順向指標會指到第一個節點，而第一個節點的逆向指標則指向最後一個節點，因此串列會形成封閉的「環狀」。

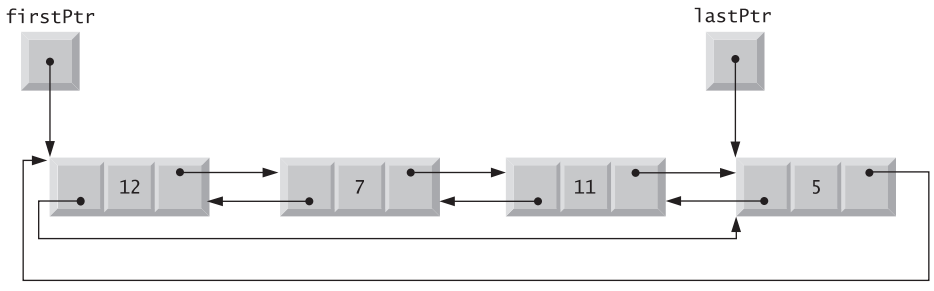


圖 20.12　環狀雙重鏈結串列

## 20.5　堆疊

在第 14 章「樣板」中，我們以陣列實作來解釋堆疊類別樣板的觀念。本節則使用指標型的鏈結串列實作。我們也會在第 22 章「標準樣板函式庫」(STL) 中討論堆疊。

堆疊資料結構可讓節點加入堆疊頂端，且只能夠從堆疊頂端移除節點。因此堆疊稱爲後進先出 (LIFO) 的資料結構。實作堆疊的方式之一，就是將它實作成特殊化的鏈結串列。在此實作中，堆疊最後一個節點的鏈結成員會設定成 null (零)，表示堆疊底部。

操作堆疊的主要成員函式是 `push` 和 `pop`。`push` 函式會將新節點插入堆疊頂端。`pop` 函式會從堆疊頂端取出一個節點，並將取出的數值儲存在呼叫函式傳入的參照變數中，若 `pop` 操作成功，就傳回 `true` (否則傳回 `false`)。

堆疊有許多有趣的應用。例如，呼叫某個函式時，被呼叫的函式得知道如何回應呼叫者，所以傳回位址會推入堆疊。如果連續呼叫函式，則一連串傳回值會以「後進先出」的順序推入堆疊中，所以每個函式就能傳回給它的呼叫者。堆疊支援以傳統非遞迴的函式呼叫方式，來進行遞迴的函式呼叫。第 6.11 節詳細討論了函式呼叫堆疊。

堆疊提供每次呼叫函式時自動變數所需的記憶體空間，以儲存自動變數的數值。當函式返回它的呼叫者或拋出例外時，就會呼叫每個區域物件的解構子 (如果有的話)，該函式的自動變數記憶體空間會從堆疊取出，程式便無法再使用這些變數。

編譯器在解析運算式和產生機器語言程式碼的過程中，就會使用堆疊。本章習題會探索堆疊的幾種應用，包括使用堆疊開發一個完整可運作的編譯器。

我們會利用串列和堆疊的密切關係，重複使用串列類別來實作堆疊類別。首先，我們 `private` 繼承串列類別來實作堆疊類別。然後透過組合方式，將一個串列物件當成堆疊類別的 `private` 成員，以實作出操作方式完全相同的堆疊類別。當然，本章所有資料結構 (包括這二種堆疊類別) 都實作成樣板，以提升再利用性。

圖 20.13-20.14 的程式 `private` 繼承 (第 9 行) 圖 20.4 的 `List` 類別樣板，以建立 `Stack` 類別樣板 (圖 20.13)。我們希望 `Stack` 具有成員函式 `push` (第 13-16 行)、`pop` (第 19-22 行)、`isStackEmpty` (第 25-28 行) 和 `printStack` (第 31-34 行)。請注意，這些基本上就是 `List` 類別樣板的 `insertAtFront`、`removeFromFront`、`isEmpty` 和 `print` 函式。當然，`List` 類別樣板包含其它成員函式 (也就是 `insertAtBack` 和 `removeFromBack`)，但我們 `Stack` 類別的 `public` 介面不想提供這些函式。所以當 `Stack` 類別樣板繼承 `List` 類別樣板時，我們使用 `private` 繼承。這可讓所有 `List` 類別樣板的成員函式，在 `Stack` 類別樣板中都是 `private` 的。當我們實作 `Stack` 的

成員函式時，就能讓每個函式呼叫 `List` 類別的適當成員函式，`push` 會呼叫 `insertAtFront`（第 15 行），`pop` 會呼叫 `removeFromFront`（第 21 行），`isStackEmpty` 會呼叫 `isEmpty`（第 27 行），而 `printStack` 會呼叫 `print`（第 33 行）。這稱作**委派 (delegation)**。

```
1  // Fig. 20.13: Stack.h
2  // Template Stack class definition derived from class List.
3  #ifndef STACK_H
4  #define STACK_H
5
6  #include "List.h" // List class definition
7
8  template< typename STACKTYPE >
9  class Stack : private List< STACKTYPE >
10 {
11 public:
12    // push calls the List function insertAtFront
13    void push( const STACKTYPE &data )
14    {
15       insertAtFront( data );
16    } // end function push
17
18    // pop calls the List function removeFromFront
19    bool pop( STACKTYPE &data )
20    {
21       return removeFromFront( data );
22    } // end function pop
23
24    // isStackEmpty calls the List function isEmpty
25    bool isStackEmpty() const
26    {
27       return this->isEmpty();
28    } // end function isStackEmpty
29
30    // printStack calls the List function print
31    void printStack() const
32    {
33       this->print();
34    } // end function print
35 }; // end class Stack
36
37 #endif
```

圖 20.13 `Stack` 類別樣板定義

在第 27 行和 33 行中我們明確地使用 `this` 指標，這是必要的，這樣編譯器才能正確地解析樣板定義中的識別字。**依賴名稱 (dependent name)** 是依賴於樣板參數的識別字。舉例來說，`removeFromFront` 呼叫 (第 21 行) 依賴於引數 `data`，其型別依賴

於樣板參數 STACKTYPE。依賴名稱的解析發生在樣板實體化的時候。相反地，沒有引數的函式識別字 (例如父類別 List 中的 isEmpty 或 print) 爲**非依賴名稱 (non-dependent name)**。這種識別字的解析通常發生在樣板定義的時候。假如樣板還沒有實體化，非依賴名稱的函式程式碼還不存在，某些編譯器會產生編譯錯誤。在第 27 行和 33 行明確使用 this->可以讓基本類別的成員函式呼叫依賴於樣板參數，讓程式碼能夠順利編譯。

在 main 函式 (圖 20.14) 中，程式使用堆疊類別樣板產生型別 Stack <int> 的整數堆疊 intStack (第 9 行)。整數 0 到 2 會推入 intStack (第 14-18 行)，然後再從 intStack 取出 (第 23-28 行)。程式使用 Stack 類別樣板建立型別 Stack <double> 的 doubleStack (第 30 行)。數值 1.1、2.2、3.3 會推入 doubleStack (第 36-41 行)，然後彈出 doubleStack (第 46-51 行)。

```
1  // Fig. 20.14: Fig21_14.cpp
2  // Template Stack class test program.
3  #include <iostream>
4  #include "Stack.h" // Stack class definition
5  using namespace std;
6
7  int main()
8  {
9     Stack< int > intStack; // create Stack of ints
10
11    cout << "processing an integer Stack" << endl;
12
13    // push integers onto intStack
14    for ( int i = 0; i < 3; i++ )
15    {
16       intStack.push( i );
17       intStack.printStack();
18    } // end for
19
20    int popInteger; // store int popped from stack
21
22    // pop integers from intStack
23    while ( !intStack.isStackEmpty() )
24    {
25       intStack.pop( popInteger );
26       cout << popInteger << " popped from stack" << endl;
27       intStack.printStack();
28    } // end while
29
30    Stack< double > doubleStack; // create Stack of doubles
31    double value = 1.1;
32
```

圖 20.14　簡單的堆疊程式

```
33      cout << "processing a double Stack" << endl;
34
35      // push floating-point values onto doubleStack
36      for ( int j = 0; j < 3; j++ )
37      {
38         doubleStack.push( value );
39         doubleStack.printStack();
40         value += 1.1;
41      } // end for
42
43      double popDouble; // store double popped from stack
44
45      // pop floating-point values from doubleStack
46      while ( !doubleStack.isStackEmpty() )
47      {
48         doubleStack.pop( popDouble );
49         cout << popDouble << " popped from stack" << endl;
50         doubleStack.printStack();
51      } // end while
52   } // end main
```

```
processing an integer Stack
The list is: 0

The list is: 1 0

The list is: 2 1 0

2 popped from stack
The list is: 1 0

1 popped from stack
The list is: 0

0 popped from stack
The list is empty

processing a double Stack
The list is: 1.1

The list is: 2.2 1.1
```

```
The list is: 3.3 2.2 1.1

3.3 popped from stack
The list is: 2.2 1.1

2.2 popped from stack
The list is: 1.1

1.1 popped from stack
The list is empty

All nodes destroyed

All nodes destroyed
```

圖 20.14　簡單的堆疊程式 (續)

另外一個實作 Stack 類別樣板的方式，就是以「組合」再利用 List 類別樣板。圖 20.15 是新的 Stack 類別樣板實作，它包含一個稱爲 stackList (第 38 行) 的 List <STACKTYPE> 物件。此版 Stack 類別樣板使用圖 20.4 的 List 類別。我們使用圖 20.14 的測試程式測試此類別，但在第 6 行改用新的 Stackcomposition.h 標頭檔。兩個版本的 Stack 類別輸出是相同的。

```
1  // Fig. 20.15: Stackcomposition.h
2  // Template Stack class definition with composed List object.
3  #ifndef STACKCOMPOSITION_H
4  #define STACKCOMPOSITION_H
5
6  #include "List.h" // List class definition
7
8  template< typename STACKTYPE >
9  class Stack
10 {
11 public:
12    // no constructor; List constructor does initialization
13
14    // push calls stackList object's insertAtFront member function
15    void push( const STACKTYPE &data )
16    {
17       stackList.insertAtFront( data );
18    } // end function push
19
20    // pop calls stackList object's removeFromFront member function
21    bool pop( STACKTYPE &data )
22    {
23       return stackList.removeFromFront( data );
24    } // end function pop
25
26    // isStackEmpty calls stackList object's isEmpty member function
27    bool isStackEmpty() const
28    {
29       return stackList.isEmpty();
30    } // end function isStackEmpty
31
32    // printStack calls stackList object's print member function
33    void printStack() const
34    {
35       stackList.print();
36    } // end function printStack
37 private:
38    List< STACKTYPE > stackList; // composed List object
39 }; // end class Stack
40
41 #endif
```

圖 20.15　Stack 類別樣板，組合了 List 物件

## 20.6 佇列

**佇列 (queue)** 類似超市的結帳隊伍，櫃臺會先服務第一位顧客，其他顧客就加入隊伍末端等候結帳。佇列節點只能從佇列前端移除，且只能從佇列尾端加入。因此佇列稱爲先進先出 (FIFO) 的資料結構。插入和移除操作稱爲 **enqueue** 和 **dequeue**。

佇列在電腦系統中有許多應用。單一處理器電腦一次只能服務一位使用者。其它使用者個體會放到佇列。當前面使用者接受服務後，佇列中的個體會逐漸移到前面。當個體到達佇列最前面時，就成爲下一個服務對象。

佇列也可支援**列印排存 (print spooling)**。例如，網路使用者會共用一台印表機。每個使用者都可送列印工作給印表機，即使印表機持續忙碌中。這些列印工作會放在佇列中，直到印表機可以列印爲止。一個稱作**排存器 (spooler)** 的程式會管理佇列，確保每個列印工作完成後，下一個列印工作就會送給印表機。

資料封包也會在電腦網路的佇列中等候。每當封包到達某個網路節點時，它一定會沿著封包最後目的地的路徑，路由到網路的下一個節點。路由節點每次只能路由一個封包，所以多的封包會先推入佇列中，直到路由器能處理它們爲止。

電腦網路上的檔案伺服器，必須處理眾多用戶端的存取需求。伺服器容量有限，但必須應付許多用戶的服務需求。超過容量限制時，用戶便只能在佇列等待。

圖 20.16-20.17 的程式 `private` 繼承 (第 9 行) 圖 20.4 的 `List` 類別樣板，以建立 `Queue` 類別樣板 (圖 20.16)。`Queue` 具有成員函式 `enqueue` (第 13-16 行)、`dequeue` (第 19-22 行)、`isQueueEmpty` (第 25-28 行) 和 `printQueue` (第 31-34 行)。這些基本上就是 `List` 類別樣板的 `insertAtFront`、`removeFromFront`、`isEmpty` 和 `print` 函式。當然，`List` 類別樣板包含其它成員函式，但 `Queue` 類別的 `public` 介面不想提供這些函式。所以當 `Queue` 類別樣板繼承 `List` 類別樣板時，我們使用 `private` 繼承。這可讓所有 `List` 類別樣板的成員函式，在 `Queue` 類別樣板中都是 `private` 的。當我們實作 `Queue` 的成員函式時，就能讓每個函式呼叫 `List` 類別的適當成員函式，`enqueue` 會呼叫 `insertAtBack` (第 15 行)，`dequeue` 會呼叫 `removeFromFront` (第 21 行)，`isQueueEmpty` 會呼叫 `isEmpty` (第 27 行)，而 `printQueue` 會呼叫 `print` (第 33 行)。如同圖 20.13 的 `Stack` 範例，`isQueueEmpty` 和 `printQueue` 中需要明確地使用 `this` 指標以避免編譯錯誤。

```
1  // Fig. 20.16: Queue.h
2  // Template Queue class definition derived from class List.
3  #ifndef QUEUE_H
4  #define QUEUE_H
5
6  #include "List.h" // List class definition
7
8  template< typename QUEUETYPE >
9  class Queue : private List< QUEUETYPE >
10 {
11 public:
12    // enqueue calls List member function insertAtBack
13    void enqueue( const QUEUETYPE &data )
14    {
15       insertAtBack( data );
16    } // end function enqueue
17
18    // dequeue calls List member function removeFromFront
19    bool dequeue( QUEUETYPE &data )
20    {
21       return removeFromFront( data );
22    } // end function dequeue
23
24    // isQueueEmpty calls List member function isEmpty
25    bool isQueueEmpty() const
26    {
27       return this->isEmpty();
28    } // end function isQueueEmpty
29
30    // printQueue calls List member function print
31    void printQueue() const
32    {
33       this->print();
34    } // end function printQueue
35 }; // end class Queue
36
37 #endif
```

圖 20.16　Queue 類別樣板定義

圖 20.17 使用 Queue 類別樣板來產生型別 Queue <int> (第 9 行) 的整數佇列 intQueue。整數 0 到 2 會先推入佇列 intQueue (第 14–18 行)，然後按先進先出的順序 (第 23–28 行) 從 intQueue 取出數值。接下來，程式會建立 Queue <double> 型別的 doubleQueue 佇列 (第 30 行)。數值 1.1、2.2、3.3 會加入 doubleQueue (第 36–41 行)，然後按先進先出的順序從 doubleQueue 中取出數值 (第 46–51 行)。

```
// Fig. 20.17: Fig21_17.cpp
// Template Queue class test program.
#include <iostream>
#include "Queue.h" // Queue class definition
using namespace std;

int main()
{
   Queue< int > intQueue; // create Queue of integers

   cout << "processing an integer Queue" << endl;

   // enqueue integers onto intQueue
   for ( int i = 0; i < 3; i++ )
   {
      intQueue.enqueue( i );
      intQueue.printQueue();
   } // end for

   int dequeueInteger; // store dequeued integer

   // dequeue integers from intQueue
   while ( !intQueue.isQueueEmpty() )
   {
      intQueue.dequeue( dequeueInteger );
      cout << dequeueInteger << " dequeued" << endl;
      intQueue.printQueue();
   } // end while

   Queue< double > doubleQueue; // create Queue of doubles
   double value = 1.1;

   cout << "processing a double Queue" << endl;

   // enqueue floating-point values onto doubleQueue
   for ( int j = 0; j < 3; j++ )
   {
      doubleQueue.enqueue( value );
      doubleQueue.printQueue();
      value += 1.1;
   } // end for

   double dequeueDouble; // store dequeued double

   // dequeue floating-point values from doubleQueue
   while ( !doubleQueue.isQueueEmpty() )
   {
      doubleQueue.dequeue( dequeueDouble );
      cout << dequeueDouble << " dequeued" << endl;
      doubleQueue.printQueue();
   } // end while
} // end main
```

圖 20.17　佇列處理程式

```
processing an integer Queue
The list is: 0

The list is: 0 1

The list is: 0 1 2

0 dequeued
The list is: 1 2

1 dequeued
The list is: 2

2 dequeued
The list is empty

processing a double Queue
The list is: 1.1

The list is: 1.1 2.2

The list is: 1.1 2.2 3.3

1.1 dequeued
The list is: 2.2 3.3

2.2 dequeued
The list is: 3.3

3.3 dequeued
The list is empty

All nodes destroyed

All nodes destroyed
```

圖 20.17　佇列處理程式 (續)

## 20.7　樹

鏈結串列、堆疊和佇列都是線性資料結構。樹則是非線性、二維的資料結構。樹的每個節點都包含二個以上的鏈結。本節將討論**二元樹** (**binary trees**，圖 20.18)，就是節點都只含二個鏈結的樹 (可能兩個都是 null、其中一個是 null、或都不是 null)。

### 基本術語

爲方便討論，我們將圖 20.18 的節點稱作 A、B、C、D。**根節點** (**root node**，就是 B) 是樹的第一個節點。根節點的每個鏈結叫做**子節點** (**child**，就是 A 和 D)。**左子節點** (**left child**，節點 A) 是**左子樹** (**left subtree**，只有節點 A) 的根節點，**右子節點** (**right**

**child**，節點 D) 則是**右子樹** (**right subtree**，包含節點 D 和節點 C) 的根節點。某節點的各子節點，稱爲**兄弟節點** (**sibling**，例如節點 A 和 D 就是兄弟節點)。沒有子節點的節點稱爲**葉節點** (**leaf node**，例如 A 和 C 便是葉節點)。電腦科學家通常從根節點往下畫出樹的結構，與自然界中樹的生長方向相反。

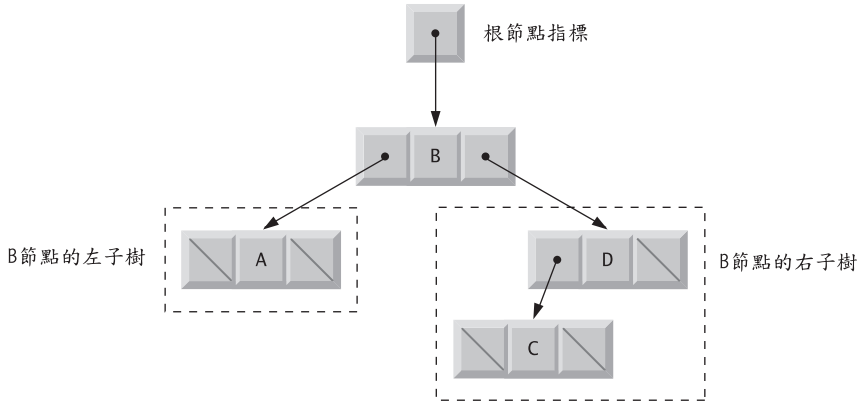


圖 20.18　二元樹圖示

## 二元搜尋樹

**二元搜尋樹** (**binary search tree**，節點的數值沒有重複) 的特性，就是左子樹中任何子節點的數值，均小於其**父節點** (**parent node**) 的數值，而右子樹中任何子節點的數值，均大於其父節點的數值。圖 20.19 是擁有 9 個數值的二元搜尋樹。請注意，同一組資料的二元搜尋樹圖形可能會有所不同，數值加入二元樹的順序會決定其形狀。

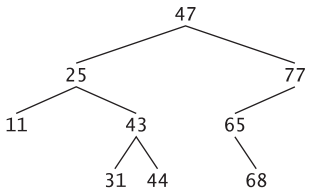


圖 20.19　二元搜尋樹

## 實作二元搜尋樹程式

圖 20.20-20.22 的程式建立一棵二元搜尋樹，並用三種方式走訪 (就是走過所有節點)；遞迴的**中序 (inorder)**、**前序 (preorder)**、**後序走訪 (postorder traversal)**。稍後會解釋這些走訪演算法。

我們先討論測試程式 (圖 20.22)，然後討論 `TreeNode` 類別 (圖 20.20) 和 `Tree` 類別 (圖 20.21) 的實作。`main` 函式 (圖 20.22) 首先會產生型別 `Tree <int>` 的整數樹 `intTree` (第 10 行)。程式提示輸入 10 個整數，然後呼叫函式 `insertNode` (第 19 行) 將每個整數加入二元樹。然後，程式會開始執行 `intTree` 的前序、中序和後序走訪 (我們很快會加以說明，分別是第 23、26 和 29 行)。然後程式會產生型別 `Tree <double>` 的浮點數樹 `doubleTree` 物件 (第 31 行)。程式提示輸入 10 個 `double` 值，然後呼叫函式 `insertNode` (第 41 行) 將每個整數加入二元樹。然後程式會執行 `doubleTree` 的前序、中序和後序走訪 (分別是第 45、48 和 51 行)。

```
1  // Fig. 20.20: TreeNode.h
2  // Template TreeNode class definition.
3  #ifndef TREENODE_H
4  #define TREENODE_H
5
6  // forward declaration of class Tree
7  template< typename NODETYPE > class Tree;
8
9  // TreeNode class-template definition
10 template< typename NODETYPE >
11 class TreeNode
12 {
13    friend class Tree< NODETYPE >;
14 public:
15    // constructor
16    TreeNode( const NODETYPE &d )
17       : leftPtr( 0 ), // pointer to left subtree
18         data( d ), // tree node data
19         rightPtr( 0 ) // pointer to right substree
20    {
21       // empty body
22    } // end TreeNode constructor
23
24    // return copy of node's data
25    NODETYPE getData() const
26    {
27       return data;
28    } // end getData function
29 private:
30    TreeNode< NODETYPE > *leftPtr; // pointer to left subtree
```

圖 20.20　`TreeNode` 類別樣板定義

```
31      NODETYPE data;
32      TreeNode< NODETYPE > *rightPtr; // pointer to right subtree
33   }; // end class TreeNode
34
35   #endif
```

圖 20.20 TreeNode 類別樣板定義 (續)

```
1    // Fig. 20.21: Tree.h
2    // Template Tree class definition.
3    #ifndef TREE_H
4    #define TREE_H
5
6    #include <iostream>
7    #include "TreeNode.h"
8    using namespace std;
9
10   // Tree class-template definition
11   template< typename NODETYPE > class Tree
12   {
13   public:
14      Tree(); // constructor
15      void insertNode( const NODETYPE & );
16      void preOrderTraversal() const;
17      void inOrderTraversal() const;
18      void postOrderTraversal() const;
19   private:
20      TreeNode< NODETYPE > *rootPtr;
21
22      // utility functions
23      void insertNodeHelper( TreeNode< NODETYPE > **, const NODETYPE & );
24      void preOrderHelper( TreeNode< NODETYPE > * ) const;
25      void inOrderHelper( TreeNode< NODETYPE > * ) const;
26      void postOrderHelper( TreeNode< NODETYPE > * ) const;
27   }; // end class Tree
28
29   // constructor
30   template< typename NODETYPE >
31   Tree< NODETYPE >::Tree()
32   {
33      rootPtr = 0; // indicate tree is initially empty
34   } // end Tree constructor
35
36   // insert node in Tree
37   template< typename NODETYPE >
38   void Tree< NODETYPE >::insertNode( const NODETYPE &value )
39   {
40      insertNodeHelper( &rootPtr, value );
41   } // end function insertNode
42
43   // utility function called by insertNode; receives a pointer
44   // to a pointer so that the function can modify pointer's value
45   template< typename NODETYPE >
```

圖 20.21 Tree 類別樣板定義

```
46  void Tree< NODETYPE >::insertNodeHelper(
47     TreeNode< NODETYPE > **ptr, const NODETYPE &value )
48  {
49     // subtree is empty; create new TreeNode containing value
50     if ( *ptr == 0 )
51        *ptr = new TreeNode< NODETYPE >( value );
52     else // subtree is not empty
53     {
54        // data to insert is less than data in current node
55        if ( value < ( *ptr )->data )
56           insertNodeHelper( &( ( *ptr )->leftPtr ), value );
57        else
58        {
59           // data to insert is greater than data in current node
60           if ( value > ( *ptr )->data )
61              insertNodeHelper( &( ( *ptr )->rightPtr ), value );
62           else // duplicate data value ignored
63              cout << value << " dup" << endl;
64        } // end else
65     } // end else
66  } // end function insertNodeHelper
67
68  // begin preorder traversal of Tree
69  template< typename NODETYPE >
70  void Tree< NODETYPE >::preOrderTraversal() const
71  {
72     preOrderHelper( rootPtr );
73  } // end function preOrderTraversal
74
75  // utility function to perform preorder traversal of Tree
76  template< typename NODETYPE >
77  void Tree< NODETYPE >::preOrderHelper( TreeNode< NODETYPE > *ptr ) const
78  {
79     if ( ptr != 0 )
80     {
81        cout << ptr->data << ' '; // process node
82        preOrderHelper( ptr->leftPtr ); // traverse left subtree
83        preOrderHelper( ptr->rightPtr ); // traverse right subtree
84     } // end if
85  } // end function preOrderHelper
86
87  // begin inorder traversal of Tree
88  template< typename NODETYPE >
89  void Tree< NODETYPE >::inOrderTraversal() const
90  {
91     inOrderHelper( rootPtr );
92  } // end function inOrderTraversal
93
94  // utility function to perform inorder traversal of Tree
95  template< typename NODETYPE >
96  void Tree< NODETYPE >::inOrderHelper( TreeNode< NODETYPE > *ptr ) const
97  {
```

圖 20.21　`Tree` 類別樣板定義 (續 1)

```
98       if ( ptr != 0 )
99       {
100         inOrderHelper( ptr->leftPtr ); // traverse left subtree
101         cout << ptr->data << ' '; // process node
102         inOrderHelper( ptr->rightPtr ); // traverse right subtree
103      } // end if
104   } // end function inOrderHelper
105
106   // begin postorder traversal of Tree
107   template< typename NODETYPE >
108   void Tree< NODETYPE >::postOrderTraversal() const
109   {
110      postOrderHelper( rootPtr );
111   } // end function postOrderTraversal
112
113   // utility function to perform postorder traversal of Tree
114   template< typename NODETYPE >
115   void Tree< NODETYPE >::postOrderHelper(
116      TreeNode< NODETYPE > *ptr ) const
117   {
118      if ( ptr != 0 )
119      {
120         postOrderHelper( ptr->leftPtr ); // traverse left subtree
121         postOrderHelper( ptr->rightPtr ); // traverse right subtree
122         cout << ptr->data << ' '; // process node
123      } // end if
124   } // end function postOrderHelper
125
126   #endif
```

圖 20.21 `Tree` 類別樣板定義 (續 2)

```
1   // Fig. 20.22: Fig21_22.cpp
2   // Tree class test program.
3   #include <iostream>
4   #include <iomanip>
5   #include "Tree.h" // Tree class definition
6   using namespace std;
7
8   int main()
9   {
10     Tree< int > intTree; // create Tree of int values
11     int intValue;
12
13     cout << "Enter 10 integer values:\n";
14
15     // insert 10 integers to intTree
16     for ( int i = 0; i < 10; i++ )
17     {
18        cin >> intValue;
19        intTree.insertNode( intValue );
20     } // end for
21
```

圖 20.22 建立並走訪二元樹

```
22      cout << "\nPreorder traversal\n";
23      intTree.preOrderTraversal();
24
25      cout << "\nInorder traversal\n";
26      intTree.inOrderTraversal();
27
28      cout << "\nPostorder traversal\n";
29      intTree.postOrderTraversal();
30
31      Tree< double > doubleTree; // create Tree of double values
32      double doubleValue;
33
34      cout << fixed << setprecision( 1 )
35         << "\n\n\nEnter 10 double values:\n";
36
37      // insert 10 doubles to doubleTree
38      for ( int j = 0; j < 10; j++ )
39      {
40         cin >> doubleValue;
41         doubleTree.insertNode( doubleValue );
42      } // end for
43
44      cout << "\nPreorder traversal\n";
45      doubleTree.preOrderTraversal();
46
47      cout << "\nInorder traversal\n";
48      doubleTree.inOrderTraversal();
49
50      cout << "\nPostorder traversal\n";
51      doubleTree.postOrderTraversal();
52      cout << endl;
53   } // end main
```

```
Enter 10 integer values:
50 25 75 12 33 67 88 6 13 68

Preorder traversal
50 25 12 6 13 33 75 67 68 88
Inorder traversal
6 12 13 25 33 50 67 68 75 88
Postorder traversal
6 13 12 33 25 68 67 88 75 50


Enter 10 double values:
39.2 16.5 82.7 3.3 65.2 90.8 1.1 4.4 89.5 92.5

Preorder traversal
39.2 16.5 3.3 1.1 4.4 82.7 65.2 90.8 89.5 92.5
Inorder traversal
1.1 3.3 4.4 16.5 39.2 65.2 82.7 89.5 90.8 92.5
Postorder traversal
1.1 4.4 3.3 16.5 65.2 89.5 92.5 90.8 82.7 39.2
```

圖 20.22　建立並走訪二元樹 (續)

我們從 TreeNode 類別樣板 (圖 20.20) 定義開始，它將 Tree <NODETYPE> 宣告成它的 friend (第 13 行)。如此可讓 Tree (圖 20.21) 的特殊化類別樣板的所有成員函式，成爲相對應 TreeNode 的特殊化類別樣板的夥伴，好讓它們存取該型別 TreeNode 物件的 private 成員。由於在此 friend 宣告中，TreeNode 的樣板參數 NODETYPE 是 Tree 的樣板引數，所以特定型別的特殊化 TreeNode，只能由相同型別的特殊化 Tree 處理 (例如，int 型別的 Tree 才能管理 int 數值的 TreeNode 物件)。

第 30-32 行宣告 TreeNode 的 private 資料，也就是節點的 data、指標 leftPtr (指向節點的左子樹) 和 rightPtr (指向節點的右子樹)。建構子 (第 16-22 行) 會將接收的引數設給 data，並將指標 leftPtr 和 rightPtr 設爲零 (因此讓此節點成爲葉節點)。成員函式 getData (第 25-28 行) 會傳回 data 的數值。

Tree 類別樣板 (圖 20.21) 包含 private 資料 rootPtr (第 20 行) ，這是一個指向樹的根節點的指標。類別樣板的第 15-18 行也宣告了 public 成員函式 insertNode (在樹中加入新節點)，以及 preOrderTraversal 、 inOrderTraversal 和 postOrderTraversal，分別以指定方式走訪整顆樹。這些成員函式會呼叫各自的遞迴工具函式，以在樹的內部結構上執行適當操作，所以程式不須存取底層 private 資料，就能執行這些函式。記住，遞迴要我們傳入一個指標，代表下一個要處理的子樹。Tree 的建構子會將 rootPtr 初始爲零，表示樹一開始是空的。

insertNode (第 37–41 行) 會呼叫 Tree 類別的工具函式 insertNodeHelper (第 45–66 行)，遞迴地將節點加入樹中。節點只能當作「葉節點」加入二元搜尋樹。若樹是空的，程式就會建立新的 TreeNode，設定其初始值並將它加入樹中 (第 51–52 行)。

如果樹不是空的，程式會將要加入的數值與根節點的 data 值比較。如果加入的值比較小 (第 55 行)，程式就會遞迴呼叫 insertNodeHelper 函式 (第 56 行) 將數值加到左子樹。如果加入的值比較大 (第 60 行)，程式就會遞迴呼叫 insertNodeHelper 函式 (第 61 行) 將數值加到右子樹。如果要加入的數值等於根節點的 data，程式就會印出訊息"dup" (第 63 行) 然後傳回，不會將重複的值加入樹中。請注意，insertNode 會將 rootPtr 的位址傳給 insertNodeHelper (第 40 行)，所以它可以修改 rootPtr 的數值 (也就是根節點的位址)。爲了接收一個指向 rootPtr (它本身也是一個指標) 的指標，insertNodeHelper 的第一個引數會宣告成「指向 TreeNode 指標」的指標。

成員函式 inOrderTraversal (第 88–92 行)，preOrderTraversal (第 69– 73 行) 和 postOrderTraversal (第 107–111 行) 會走訪整顆樹，並印出節點數值。在接下來的討論中，我們會使用圖 20.23 的二元搜尋樹。

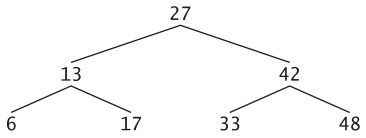


圖 20.23　二元搜尋樹

### 中序走訪演算法

函式 `inOrderTraversal` 會呼叫 `inOrderHelper` 工具函式來執行二元樹的中序走訪。中序走訪的步驟爲：

1. 以中序走訪左子樹。(由第 100 行的 `inOrderHelper` 呼叫達成)。
2. 處理節點中的數值，也就是印出節點的數值 (第 101 行)。
3. 以中序走訪右子樹。(由第 102 行的 `inOrderHelper` 呼叫達成)。

除非左子樹的數值都處理完畢，否則不會處理節點中的數值，因爲每個 `inOrderHelper` 呼叫都會立刻以「指向左子樹」的指標再次呼叫 `inOrderHelper`。圖 20.23 的樹的中序走訪就是：

```
6 13 17 27 33 42 48
```

請注意，二元搜尋樹的中序走訪會以遞增順序印出節點的數值。實際上，建立二元搜尋樹的過程會將資料排序，因此這個過程也稱爲**二元樹排序 (binary tree sort)**。

### 前序走訪演算法

函式 `preOrderTraversal` 會呼叫 `preOrderHelper` 工具函式來執行二元樹的前序走訪。前序走訪的步驟爲：

1. 處理節點中的數值。(第 81 行)
2. 以前序走訪左子樹。(由第 82 行的 `preOrderHelper` 呼叫達成)
3. 以前序走訪右子樹。(由第 83 行的 `preOrderHelper` 呼叫達成)

走到某個節點時，就會處理該節點的數值。程式會先處理節點中的數值，再處理左子樹的值。然後再處理右子樹的值。圖 20.23 的樹的前序走訪就是：

```
27 13 6 17 42 33 48
```

### 後序走訪演算法

函式 `postOrderTraversal` 會呼叫 `postOrderHelper` 工具函式來執行二元樹的後序走訪。後序走訪的步驟爲：

1. 以後序走訪左子樹。(由第 120 行的 `postOrderHelper` 呼叫達成)
2. 以後序走訪右子樹。(由第 121 行的 `postOrderHelper` 呼叫達成)
3. 處理節點中的數值。(第 122 行)

程式會先印出所有子節點的數值，然後才印出該節點的值。圖 20.23 的樹的後序走訪就是：

```
      6 17 13 33 48 42 27
```

### 消除重複數值

二元搜尋樹可以**消除重複數值 (duplicate elimination)**。建立樹的時候會抓出重複的數值，因爲重複數值也跟原來數值加入時一樣，每次比較就要決定「往左」還是「往右」。因此，重複數值最後一定會碰到含有相同數值的節點。這時就會把重複的數值丟掉。

在二元樹內搜尋某個鍵值也很快。若樹長得夠平衡，則每個分枝大約包含一半的節點數。每次將節點與搜尋鍵值比較時，就會砍掉一半節點。這稱作 O(log n) 演算法 (Big O 記號於第 19 章討論)。所以，n 個元素的二元搜尋樹最多只要比 $\log_2 n$ 次，就能找出鍵值或回報找不到。以 1000 個元素的二元搜尋樹爲例 (假設它夠平衡)，最多只要比 10 次，因爲 $2^{10}$>1000。搜尋 1,000,000 個元素的 (平衡) 二元搜尋樹時，最多只要比 20 次，因爲 $2^{20}$ >1,000,000。

### 二元樹習題概觀

本章習題會提出其它幾種二元樹操作演算法，例如刪除二元樹的一個項目、以二維空間的樹狀格式印出一棵二元樹，以及對二元樹進行**階層順序走訪 (level-order traversal)**。按照階層順序走訪二元樹，會採逐列的方式走訪每個節點，程式會先從根節點的階層開始。在樹的每個階層中，節點都是從左到右走訪。其它二元樹的習題，包括讓二元搜尋樹含有重複的數值、在二元樹中加入字串、以及判斷二元樹有多少層。

## 20.8　總結

在本章中，你學到了鏈結串列是「排成一列」的資料項集合。你也學到了程式可以在鏈結串列的任意位置加入／刪除資料 (但我們的版本只能在鏈結串列的前端和後端加入和刪除項目)。我們示範了堆疊和佇列資料結構都是串列的特殊版本。你只能在堆疊的頂端插入和刪除項目，只能從佇列的尾端插入項目，頭端刪除項目。我們也示範了二元樹資料結構，並使用二元搜尋樹進行高速搜尋、排序，以及刪除重複資料。你學到了如何以可重複使用 (如樣板) 和易於管理的方式建立這些資料結構。在下一章，我們將介紹 `struct` (與類別相似)，並討論位元、字元和 C 風格字串的處理。

## 摘要

### 20.1 簡介

- 動態資料結構會在執行期間擴大和縮小。
- 鏈結串列是在邏輯上「排成一列」的資料項集合，我們可在鏈結串列中的任何位置加入和移除資料項目。
- 在編譯器和作業系統中，堆疊是很重要的。我們只能在堆疊的某一端 (也就是堆疊的頂端) 插入和移除資料項目。
- 佇列代表等待區；我們可以在佇列的後端 (也就是尾端) 加入資料項目，然後從佇列的前端 (也就是頭端) 移除資料項目。
- 二元樹可用來進行高速搜尋和排序資料、有效率的消除重複資料項目、顯示檔案系統目錄，以及將程式碼的運算式編譯成機器語言。

### 20.2 自我參照類別

- 自我參照類別包含一個指標成員，這個指標會指向相同類別型別的類別物件。
- 自我參照類別物件能彼此鏈結，形成有用的資料結構，如串列、佇列、堆疊和樹。

### 20.3 動態記憶體配置與資料結構

- 動態記憶體配置的大小限制爲電腦可用的實體記憶體，或虛擬記憶體系統中的可用虛擬記憶體大小。

### 20.4 鏈結串列

- 鏈結串列是自我參照類別物件的線性集合，每個物件稱爲節點，它們透過指標鏈結串起來，因此稱爲「鏈結」串列。

- 我們可用指向串列第一個節點的指標來存取鏈結串列。至於後續節點，則透過前一個節點內的鏈結指標成員來存取。
- 鏈結串列、堆疊和佇列都是線性資料結構。樹是非線性的資料結構。
- 無法預知資料結構中有多少資料項目時，鏈結串列便很適用。
- 鏈結串列是動態的，所以串列長度能依需要增加或減少。
- 單一鏈結串列是以指向第一節點的指標開始，每個節點包含的指標都「循序」指向下一個節點。
- 環狀單一鏈結串列是以指向第一節點的指標開始，每個節點包含的指標都指向下一個節點。「最後節點」包含的並不是 null 指標，反之，該指標會指回第一個節點，因此串列會形成封閉的「環狀」。
- 雙重鏈結串列可正向、反向走訪串列。
- 雙重鏈結串列通常實作成有兩個「起始指標」，一個指向串列的第一個元素，以從前往後走訪串列，另一個指摽則指向串列的最後一個元素，以從後往前走訪串列。每個節點都擁有指向下一個節點和前一個節點的指標。
- 在環狀雙重鏈結串列中，最後節點的順向指標會指到第一個節點，而第一個節點的逆向指標則指向最後一個節點，因此串列會形成封閉的「環狀」。

## 20.5 堆疊

- 堆疊資料結構可讓節點只能從頂端加入堆疊，且只能夠從堆疊頂端移除節點。
- 堆疊稱爲後進先出 (LIFO) 的資料結構。
- 操作堆疊的主要成員函式是 `push` 和 `pop`。`push` 函式會將新節點插入堆疊頂端。`pop` 函式會將節點從堆疊頂端移除。
- **依賴名稱 (dependent name)** 是依賴於樣板參數的識別字。依賴名稱的解析發生在樣板實體化的時候。
- 非依賴名稱的解析通常發生在樣板定義的時候。

## 20.6 佇列

- 佇列類似超市的結帳隊伍，櫃臺會先服務第一位顧客，其他顧客就加入隊伍末端等候結帳。
- 佇列節點只能從佇列前端移除，且只能從佇列尾端加入。
- 佇列稱爲先進先出 (FIFO) 的資料結構。插入和移除操作稱爲 `enqueue` 和 `dequeue`。

## 20.7 樹

- 二元樹就是節點都只含二個鏈結的樹 (可能兩個都是 null、其中一個是 null、或都不是 null) 。
- 根節點是樹的第一個節點。
- 根節點的每個鏈結叫做子節點。左子節點是左子樹的根節點，右子節點則是右子樹的根節點。
- 某個節點的子節點彼此互爲兄弟節點。沒有子節點的節點稱爲葉節點。
- 二元搜尋樹 (節點的數值沒有重複) 的特性，就是左子樹中任何子節點的數值，均小於其父節點的數值，而右子樹中任何子節點的數值，均大於其父節點的數值。
- 節點只能當作「葉節點」加入二元搜尋樹。
- 二元樹的中序走訪會先走訪左子樹，然後處理根節點的值，再走訪右子樹。除非左子樹的數值都處理完畢，否則不會處理節點中的數值。
- 前序走訪會先處理根節點的數值、然後走訪左子樹、再走訪右子樹。走到某個節點時，就會先處理該節點的數值。
- 後序走訪會先走訪左子樹，然後走訪右子樹，再處理根節點中的數值。除非左右子樹的數值都處理完畢，否則不會處理節點中的數值。
- 二元搜尋樹可以消除重複數值。建立樹的時候會抓出重複的數值，並把重複的數值丟掉。
- 按照階層順序走訪二元樹，會採逐列的方式走訪每個節點，程式會先從根節點的階層開始。在樹的每個階層中，節點都是從左到右走訪。

## 術語

二元搜尋樹 (binary search tree)
二元樹排序 (binary tree sort)
二元樹 (binary tree)
子節點 (child node)
環狀雙重鏈結串列 (circular, doubly linked list)
環狀單一鏈結串列 (circular, singly linked list)
資料結構 (data structure)
委派 (delegation)
依賴名稱 (dependent name)
`dequeue`
雙重鏈結串列 (doubly linked list)
消除重複數值 (duplicate elimination)
動態資料結構 (dynamic data structure)
`enqueue`
先進先出 (first-in-first-out，FIFO)
佇列的頭端 (head of a queue)
中序表示法 (infix notation)
二元樹的中序走訪 (inorder traversal of a binary tree)
插入節點 (inserting a node) 後進先出 (last-in-first-out，LIFO)
葉節點 (leaf node)
左子節點 (left child)
左子樹 (left subtree)
階層順序走訪 (level-order traversal)
線性資料結構 (linear data structure)

鏈結 (link)
鏈結串列 (linked list)
節點 (node)
非依賴名稱 (non-dependent name)
非線性資料結構 (nonlinear data structure)
父節點 (parent node)
指標鏈結 (pointer link)
`pop`
後序表示法 (postfix notation)
二元樹的後序走訪 (postorder traversal of a binary tree)
二元樹的前序走訪 (preorder traversal of a binary tree)
列印排存 (print spooling)
`push`
佇列 (queue)
右子節點 (right child)
右子樹 (right subtree)
根節點 (root node)
自我參照類別 (self-referential class)
兄弟節點 (sibling node)
單一鏈結串列 (singly linked list)
排存器 (spooler)
堆疊 (stack)
佇列的尾端 (tail of a queue)
堆疊頂端 (top of a stack)

## 自我測驗

**20.1** 請填入以下題目的空格：

a) 自我_____類別可用來建立動態資料結構，這些資料結構可在執行時間增加或縮減。
b) _____運算子可動態配置記憶體和建構物件；此運算子會傳回一個指向物件的指標。
c) _____是特殊化的鏈結串列，其節點只能夠在此串列的起始位置進行加入和刪除操作，且節點數值以後進先出的順序傳回。
d) 若函式不會變動鏈結串列，只是判斷串列是不是空的，此種函式便是一種_____函式的範例。
e) 佇列是一種_____資料結構，因爲第一加入的節點就是第一個移除的節點。
f) 在鏈結串列中，指向下一個節點的指標就稱爲_____。
g) _____運算子可摧毀物件，並釋放動態配置的記憶體。
h) _____是特殊化的鏈結串列，節點只能在鏈結串列的尾端加入，而從串列的前端刪除。
i)_____是一種非線性、二維的資料結構，包含的節點可擁有兩個或更多的鏈結。
j) 堆疊是一種_____資料結構，因爲最後加入的節點就是第一個移除的節點。
k) _____樹的節點包含兩個鏈結成員。
l) 樹的第一個節點就是_____節點。
m) 樹節點的每個鏈結都指到該節點的_____或_____。
n) 沒有子節點的樹節點，就稱爲_____節點。
o) 本章提到二元搜尋樹的四種走訪演算法，就是_____、_____、_____和_____。

**20.2** 鏈結串列和堆疊有何差異？

**20.3** 堆疊和佇列有何差異？

**20.4** 也許我們可以爲本章取一個更好的標題：「可再利用的資料結構」。請說明下列各項如何促進資料結構的可再利用性。

a) 類別

b) 類別樣板

c) 繼承

d) `private` 繼承

e) 組合

**20.5** 請寫出圖 20.24 的二元搜尋樹之中序、前序和後序走訪結果。

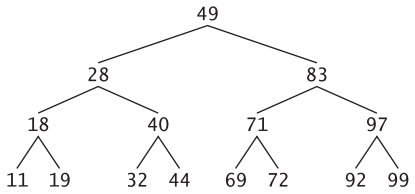


圖 20.24　15 個節點的二元搜尋樹

## 自我測驗解答

**20.1** a) 參照。b) `new`。c) 堆疊。d) 判斷。e) 先進先出 (FIFO)。f) 鏈結。g) `delete`。h) 佇列。i) 樹。j) 後進先出 (LIFO)。k) 二元。l) 根。m) 子節點或子樹。n) 葉。o) 中序，前序，後序，按階層順序。

**20.2** 我們可在鏈結串列的任何位置加入或移除節點。但我們只能在堆疊的頂端加入和移除節點。

**20.3** 佇列資料結構只能從佇列頭端移除節點，且只能從佇列尾端加入節點。佇列稱爲先進先出 (FIFO) 的資料結構。堆疊資料結構可讓節點加入堆疊頂端，且只能夠從堆疊頂端移除節點。堆疊稱爲後進先出 (LIFO) 的資料結構。

**20.4** a) 類別允許我們產生許多特定型別 (也就是類別) 的資料結構物件。

b) 類別樣板可讓我們以不同的型別參數產生相關類別。接著，我們便可依需要產生這些樣板類別的物件。

c) 繼承可讓我們在衍生類別中再利用基本類別的程式碼，所以衍生類別資料結構也是一種基本類別資料結構 (透過 `public` 繼承)。

d) private 繼承可讓我們再利用基本類別的部分程式碼，以形成衍生類別的資料結構；因爲繼承是 private 的，所以所有基本類別的 public 成員函式都會成爲衍生類別的 private 成員。可防止衍生類別資料結構的使用者，存取衍生類別不應該使用的基本類別成員函式。

e) 「組合」可讓一個類別物件資料結構成爲另一個類別的成員，以再利用程式碼；若讓類別物件成爲另一個類別的 private 成員，則類別物件的 public 成員函式就無法透過組合類別物件的介面取用。

**20.5** 中序走訪的結果是

```
11 18 19 28 32 40 44 49 69 71 72 83 92 97 99
```

前序走訪的結果是

```
49 28 18 11 19 40 32 44 83 71 69 72 97 92 99
```

後序走訪的結果是

```
11 19 18 32 44 40 28 69 72 71 92 99 97 83 49
```

## 習題

**20.6 (串接串列)** 撰寫一個串接兩個字元鏈結串列的程式。此程式應有一個 concatenate 函式，該函式會以這兩個串列物件的參照爲引數，然後將第二個串列串接到第一個串列。

**20.7 (合併有序串列)** 撰寫一個，將兩個有序的整數串列，合併成一個有序的整數串列的程式。函式 merge 的引數，應該接收這兩個要合併的串列物件的參照，以及另一個用來存放合併後元素的串列物件的參照。

**20.8 (計算串列元素的總和和平均)** 撰寫一個將 25 個介於 0 到 100 之間的隨機整數依次加入鏈結串列物件的程式。程式應計算所有元素的總和，以及所有元素的浮點數平均值。

**20.9 (反向複製串列)** 撰寫一個程式，建立具有 10 個字元的鏈結串列，然後以相反順序排列此串列，並放到第二個串列中。

**20.10 (反向列印堆疊中的句子)** 撰寫一個可輸入一行文字，然後利用堆疊將這些文字以相反的順序顯示出來的程式。

**20.11 (用堆疊測試迴文)** 撰寫一個利用堆疊來判斷某個字串是不是迴文 (亦即此字串從前面和從後面的拼法是相同的) 的程式。程式必須忽略空白和標點符號。

**20.12 (中序表示法轉換成後序表示法)** 編譯器在解析運算式和產生機器語言程式碼的過程中，就會使用堆疊。本題與下一題中，我們探討編譯器如何計算只由常數、運算子和小括號組成的算術運算式。

人類寫 3 + 4 和 7 / 9 這種算式的時候，都把運算子 (+或/) 放在運算元中間，這叫做中序表示法 (infix notation)。但電腦「喜歡」用後序表示法 (postfix notation)，就是把運算子寫在兩個運算元右邊。前述算式的後序表示法分別是 3 4 +和 7 9 /。

若要計算複雜的中序運算式，編譯器會先把它轉換成後序運算式，再計算後序運算式的結果。這些演算法都只要從左到右掃一遍運算式即可。每個演算法均使用一個堆疊支援其操作，但這些演算法的堆疊用途不同。

本題要您寫個 C++版的中序轉後序演算法。下一題就要您寫 C++版的後序運算式計算演算法。在本章後面，您會發現本題撰寫的程式碼，可幫助您實作一個完整的編譯器。請撰寫一個只有一位整數的中序算術運算式 (假設輸入的都是有效運算式) 程式，例如

```
(6 + 2) * 5 - 8 / 4
```

轉換成後序運算式。前述運算式的後序表示法如下

```
6 2 + 5 * 8 4 / -
```

程式應將運算式讀入字串 `infix`，並修改本章的堆疊函式實作，在字串 `postfix` 建立後序運算式。建立後序運算式的演算法如下：

1) 將左小括號「`(`」推入堆疊。
2) 將右小括號「`)`」加入 `infix` 末端。
3) 當堆疊不是空的時候，從左到右讀取 `infix` 並執行下列動作：
   若 `infix` 目前字元是數字，就將它複製到 `postfix` 下一個元素。
   若 `infix` 目前字元是左小括號，就把它推入堆疊。
   若 `infix` 目前字元是個運算子，
      若如果堆疊頂端有運算子，且堆疊頂端運算子的優先權大於或等於目前的運算子，就取出堆疊頂端運算子，
      將 `infix` 目前字元推入堆疊。
   若 `infix` 目前字元是右小括號。
      將堆疊頂端的運算子一直取出來放進 `postfix`，直到堆疊頂端的運算子是左小括號爲止。
      將左小括號從堆疊取出，然後丟掉。

運算式可用以下算術操作：

`+` 加法
`-` 減法
`*` 乘法
`/` 除法
`^` 次方
`%` 模數

[請注意：本習題假設所有運算子都是從左到右結合。] 此堆疊應以堆疊節點維護，每個節點含一個資料成員，以及一個指向下一個堆疊節點的指標。

您可提供以下函式功能：

a) `convertToPostfix` 函式，將中序運算式轉成後序運算式

b) `isOperator` 函式，判斷 `c` 是不是運算子

c) `precedence` 函式，判斷 `operator1` 的優先權是否大於等於 `operator2`，假如是，傳回 `true`。

d) `push` 函式，將數值推入堆疊

e) `pop` 函式，從堆疊取出數值

f) `stackTop` 函式，傳回堆疊頂端的數值，但不會把它從堆疊取出

g) `isEmpty` 函式，判斷堆疊是不是空的

h) `printStack` 函式，印出此堆疊

**20.13 (計算後序運算式)** 撰寫一個程式，計算後序運算式 (假設是有效算式)，例如

```
6 2 + 5 * 8 4 / -
```

此程式應將一段由數字和運算子組成的後序運算式讀入字串。本程式應修改本章的堆疊函式實作，掃瞄此運算式並計算結果。演算法如下：

1) 還沒遇到字串尾端時，就從左往右讀入運算式。

若目前字元是個數字，

將此整數值推入堆疊 (數字字元的整數值，便是它在電腦字元集中的數值，減掉'0'在電腦字元集中的數值)。

要不然，若目前字元是個運算子，

將堆疊頂端的兩個元素取出，放入變數 x 和 y。

計算 y operator x 的值。

將計算結果再推入堆疊。

2) 當碰到字串尾端時，就將堆疊頂端的數值取出。這就是後序運算式的結果。

[請注意：在步驟 2 中，假設運算子是'/'，堆疊頂端是 2，下一個是 8，就取出 2 放到 x，取出 8 放到 y，計算 8 / 2，再將結果 4 推入堆疊。此規則也適用於'-'運算子。] 運算式可採用以下算術操作

+ 加法

- 減法

* 乘法

/ 除法

^ 次方

% 模數

[請注意：本習題假設所有運算子都是從左到右結合。] 此堆疊應以堆疊節點維護，每個節點包含一個 `int` 資料成員，以及一個指向下一個堆疊節點的指標。您可提供以下函式功能：

a) `evaluatePostfixExpression` 函式，計算後序運算式

b) `calculate` 函式，計算運算式 `op1 operator op2`

c) `push` 函式，將數值推入堆疊

d) `pop` 函式，從堆疊取出數值

e) `isEmpty` 函式，判斷堆疊是不是空的

f) `printStack` 函式，印出此堆疊

**20.14 (加強後序計算程式)** 修改習題 20.13 的後序計算程式，讓它處理比 9 大的整數運算元。

**20.15 (超市模擬)** 撰寫一個程式，模擬超市的結帳隊伍。此隊伍是個佇列物件。顧客 (也就是顧客物件) 到達的時間，是 1-4 分鐘之間的隨機整數。而每位顧客的結帳時間 (被服務的時間) 也是 1-4 分鐘之間的隨機整數。當然，這些時間必須平衡。若平均到達時間比平均服務時間來得快，此佇列便會無限拉長。就算它是「平衡」的，隨機結果也可能讓隊伍拖得很長。請撰寫一個超市模擬程式，使用以下演算法進行 12 小時 (720 分鐘) 的模擬：

1) 隨機選擇 1 到 4 之間的一個整數，決定第一位顧客到達的時間。

2) 在第一位顧客到達時：決定顧客的服務時間 (1 到 4 之間的隨機整數)；排定下一位顧客的到達時間 (將目前時間加上 1 到 4 之間的隨機整數)。

3) 在每一分鐘：

　若下一位顧客來了，

　　那麼，

　　將此顧客排入佇列；

　　排定下一位顧客的到達時間；

　若上一位顧客服務完了；

　　那麼

　　從佇列取出下一位顧客爲他服務

　　決定顧客的服務完成時間

　　(將目前時間加上 1 到 4 之間的隨機整數)。

將此模擬執行 720 分鐘，並回答下列問題：

a) 在這段時間內，佇列中最多有幾位顧客？

b) 顧客最長等待多久時間？

c) 若將到達時間從 1-4 分鐘改成 1-3 分鐘會怎麼樣？

**20.16 (讓二元樹可包含重複元素)** 修改圖 20.20-20.22 的程式，讓二元樹物件可包含重複元素。

**20.17 (字串二元樹)** 請以圖 20.20-20.22 爲基礎寫個程式，輸入一行文字、將這段句子切成多個單字 (可用 `istringstream` 函式庫類別)、將單字插入二元搜尋樹，然後以中序、前序、後序走訪印出這棵樹。請使用 OOP 方法。

**20.18 (消除重複數值)** 本章說明建立二元搜尋樹時，可輕易消除重複數值。請說明如何只用一維陣列消除重複數值。請比較陣列和二元搜尋樹消除重複數值的效能。

**20.19 (二元樹的深度)** 撰寫一個 `depth` 函式，它會接收一棵二元樹，並算出該樹的深度 (有幾層)。

**20.20 (遞迴逆向列印一個串列)** 撰寫一個成員函式 `printListBackward`，以相反順序遞迴輸出鏈結串列物件中的項目。撰寫一個測試程式，建立一個排序過的整數串列，並以相反順序列印該串列。

**20.21 (遞迴搜尋串列)** 撰寫一個成員函式 `searchList`，遞迴搜尋一個鏈結串列物件中某個指定的數值。若找到數值，此函式就傳回一個指向此數值的指標；否則就傳回 null。請在測試程式中建立一個整數串列，以測試本函式。程式應提示使用者，輸入要在串列中尋找的數值。

**20.22 (二元樹的刪除)** 刪除演算法並不像插入演算法這麼簡單。刪除項目時，會遇到三種情況：該項目位於葉節點中 (就是沒有子節點)、該項目位於有一個子節點的節點中、以及該項目位於有兩個子節點的節點中。

若要刪除的項目位於葉節點，就刪除該節點，其父節點的指標就設爲 null。

若要刪除的項目位於有一個子節點的節點，其父節點的指標就指到其子節點，然後刪除該項目節點。這麼一來，其子節點就會頂替被刪除節點的位置。

最後一個情況最複雜。要刪除有兩個子節點的節點時，必須拿樹中的另一個節點頂替它的位置。但我們不能把父節點內的指標，直接指向被刪節點的子節點之一。若這麼做，在大部分情況下，就會違反下列二元搜尋樹 (無重複元素) 的特性：所有左子樹的節點數值必須小於父節點的數值，所有右子樹的節點數值必須大於父節點的數值。

那麼該用哪個節點頂替，才能維持此特性呢？答案有兩個：比被刪節點小的子樹中，數值最大的節點；或是比被刪節點大的子樹中，數值最小的節點。我們採用數值最大的節點。在二元搜尋樹中，小於父節點的最大數值位於左子樹中，且一定位在這棵子樹的最右邊節點上。只要在左子樹中一直往右走訪，直到目前節點的右節點指標是 null 爲止。這就是拿來頂替的節點了，它要不是葉節點，就是只有左子節點的節點。若此節點是葉節點，則刪除步驟如下：

1) 將被刪節點的指標存入一個暫存指標變數中 (本指標用來刪除動態配置的記憶體)。
2) 將被刪節的父節點中的指標，指向該替代節點。
3) 將該替代節點的父節點中的指標設爲 null。

4) 將替代節點中指向左子樹的指標，指向被刪節點的左子樹。將替代節點中指向右子樹的指標，指向被刪節點的右子樹。
5) 將暫存指標變數所指的節點刪除。

若替代節點擁有左子節點，其步驟也類似，但演算法必須將左子節點搬到替代節點的位置上。若此節點是擁有左子節點的節點，則刪除步驟如下：

1) 將被刪節點的指標存入一個暫存指標變數中。
2) 將被刪節的父節點中的指標，指向該替代節點。
3) 將替代節點的父節點中的指標，指向替代節點的左子節點。
4) 將替代節點中指向左子樹的指標，指向被刪節點的左子樹。將替代節點中指向右子樹的指標，指向被刪節點的右子樹。
5) 將暫存指標變數所指的節點刪除。

撰寫一個成員函式 `deleteNode`，它有兩個引數，一個是指向根節點的指標，另一個是要刪除的數值。此函式要找出被刪節點的位置，並以上述演算法刪除節點。此函式要顯示訊息，表示節點是否成功刪除。修改圖 20.20–20.22，使用這個函式。刪除項目後，呼叫 `inOrder`、`preOrder` 和 `postOrder` 走訪函式，確認刪除作業是否正確執行。

**20.23 (二元樹搜尋)** 撰寫一個成員函式 `binaryTreeSearch`，以在二元搜尋樹物件中搜尋指定的值。此函式應接受兩個引數，一個是指向二元樹根節點的指標,，另一個是搜尋鍵值。若找到含有搜尋鍵值的節點，函式應傳回指向該節點的指標；否則函式就傳回 null 指標。

**20.24 (階層走訪二元樹)** 圖 20.20-20.22 介紹三種走訪二元樹的遞迴方法：中序、前序、後序走訪。本習題將介紹二元樹的階層順序走訪 (level order traversal)，在此方法中，節點數值會由根節點開始一層一層地顯示出來。每個階層的節點是從左往右列印。階層走訪並不是遞迴演算法。它使用佇列來控制輸出的節點。演算法如下：

1) 將根節點加入佇列。
2) 當佇列中還有節點時，

    取得佇列的下一個節點
    列印節點的值
    若此節點的左指標不爲 null
        將左子節點加入佇列
    若此節點的右指標不爲 null
        將右子節點加入佇列

撰寫一個成員函式 `levelOrder`，執行二元樹的階層順序走訪。修改圖 20.20–20.22，使用這個函式。[請注意：本程式中，您也需修改並使用圖 20.16 的佇列處理函式。]

**20.25 (列印樹)** 撰寫一個遞迴成員函式 `outputTree`，顯示一棵二元樹。本函式必須一列一列顯示此樹，頂部節點靠螢幕左方，底部節點靠螢幕右方。每一列則垂直列印。例如，圖 20.24 二元樹的輸出應爲圖 20.25 所示的輸出。注意，最右邊的葉節點應顯示在最右一列的最上方，根節點則顯示在最左方。每一行離前一行五個空格。函式 `outputTree` 應接收一個引數 `totalSpaces`，表示輸出數值前面要有幾個空格 (此值應從零開始，所以根節點會顯示在最左邊)。本函式修改了中序走訪以印出此樹：它從最右邊節點開始走，一步步走回左邊。演算法如下：

```
            99
        97
            92
    83
            72
        71
            69
49
            44
        40
            32
    28
            19
        18
            11
```

圖 20.25　列印二元樹

當目前節點的指標不是 null 時
　遞迴呼叫 `outputTree`，引數是目前節點的右子樹以及 `totalSpaces + 5`
　使用 `for` 結構從 1 算到 `totalSpaces` 並印出空格
　印出目前節點的數值
　將指向目前節點的指標，指到目前節點的左子樹
　將 `totalSpaces` 加 5。

**20.26 (在鏈結串列的任意位置加入／刪除資料)** 我們的鏈結串列類別樣板，只允許在鏈結串列的前端和後端加入和刪除項目。只要重複使用串列類別樣板，以 `private` 繼承與組合方式，就能以最少的程式碼建立堆疊類別樣板和佇列類別樣板。事實上，鏈結串列的功能比我們提供的更廣。請修改本章的鏈結串列類別，以在串列的任意位置加入和刪除項目。

**20.27 (沒有尾端指標的串列和佇列)** 我們的鏈結串列實作 (圖 20.3-20.5) 使用了 `firstPtr` 和 `lastPtr`。對於 `List` 類別的成員函式 `insertAtBack` 和 `removeFromBack` 而言，指標 `lastPtr` 是很有用的。函式 `insertAtBack` 對應於 `Queue` 類別的 `enqueue` 成員函式。請重新撰寫 `List` 類別，讓它不再使用 `lastPtr`。因此，任何在串列尾端執行的操作，就必須從串列的前端開始搜尋。這會影響到我們實作的 `Queue` 類別嗎 (圖 20.16)？

**20.28** 使用組合版本的堆疊程式 (圖 20.15) 建立一個完整的堆疊程式。修改此程式，將成員函式行內化 (inline)。比較二種方法的優缺點。摘要出成員函式行內化的優缺點。

**20.29 (二元樹排序和搜尋效率)** 二元樹排序的一個問題，就是資料加入的順序會影響樹的形狀，相同資料以不同的順序加入樹中，就會產生不同形狀的二元樹。二元樹排序和搜尋演算法的效率，會與二元樹形狀有很大的關係。如果二元樹的資料以遞增順序加入，其形狀爲何？若以遞減順序加入，其形狀又爲何？

**20.30 (具有索引的串列)** 如本書所述，鏈結串列必須循序搜尋。若串列很大，效能會很差。有一種增進串列搜尋效能的常見技術，就是在串列中建立和維護索引。索引就是一組指標，它會指向串列中各個關鍵位置。例如，若程式要搜尋一大串姓名串列，可建立 26 個進入點索引 (每個對應到一個字母) 以提升效能。若要搜尋一個以「Y」開始的姓名時，可先搜尋索引，判斷「Y」開始的位置，然後直接「跳到」該位置，並循序搜尋，直到發現要找的姓名爲止。這種方式會比從頭開始搜尋快很多。請用圖 20.3-20.5 的 `List` 類別爲基礎，撰寫 `IndexedList` 類別。撰寫一個程式，說明索引串列的操作。請確定程式具有 `insertInIndexedList`、`seachIndexedList` 和 `deleteFromIndexedList` 成員函式。

## 特別小節：建立自己的編譯器

習題 8.18、8.19 和 8.20 介紹過 `Simpletron Machine Language` (SML)，我們也實作了一個 `Simpletron` 電腦模擬器，以執行 SML 撰寫的程式。習題 20.31-20.35 將建置一個編譯器，可將高階語言撰寫的程式轉換成 SML。本節貫穿了整個程式設計程序。我們會用此高階語言撰寫程式、用您建立的編譯器編譯程式，然後用習題 8.19 建立的模擬器執行程式。您應完全採用物件導向的方式實作此編譯器。[請注意：由於習題 20.31-20.35 的篇幅較長，我們將其置於 `www.deitel.com/books/cpphtp7/`的 `pdf` 檔中。]