

# فصل بیست و یکم

## ساختمان‌های داده

### اهداف

- فرم‌دهی ساختمان‌های داده متصل به هم با استفاده از مراجعه‌ها، کلاس‌های خود ارجاعی و بازگشتی.
- ایجاد و کار با ساختمان‌های داده دینامیکی، همانند لینک لیست‌ها، صف‌ها، پشته‌ها و درخت‌های باینری.
- آشنایی با نحوه عملکرد برنامه‌های ساختمان داده.
- آشنایی با نحوه ایجاد ساختمان‌های داده با قابلیت استفاده مجدد با بکارگیری کلاس‌ها، توارث و ترکیب.

رئوس مطالب

21-1 مقدمه

21-2 کلاس‌های خود ارجاعی



21-3 اخذ دینامیکی حافظه و ساختمان داده

21-4 لیست‌های پیوندی

21-5 پشته‌ها

21-6 صف‌ها

21-7 درخت‌ها

**21-1 مقدمه**

در مورد ساختمان‌های داده با سبایز ثابت همانند آرایه‌های یک بعدی و آرایه‌های دو بعدی مطالبی بیان کردیم. در این فصل در مورد ساختمان‌های داده دینامیکی که در زمان اجرا تغییر سبایز می‌دهند صحبت خواهیم کرد. لیست‌های پیوندی (link list) مجموعه‌ای از عناصر داده هستند که عمل درج و حذف در هر جای لیست پیوندی ممکن است. پشته‌ها (stack) جزء ساختارهای مهم در کامپایلرها و سیستم‌های عامل هستند که عمل درج و حذف فقط از بالای (ابتدای) آن ممکن است. صف‌ها (queue) به مانند صف‌های انتظار می‌باشند. عمل درج یا اضافه کردن به صف از انتهای آن (tail) و عمل حذف از ابتدای (head) آن صورت می‌گیرد. درخت‌های باینری (binary trees)، جستجوی بسیار سریع، ذخیره‌سازی داده‌ها و کامپایل عبارات به زبان ماشین را تسهیل می‌کنند. البته این ساختمان‌های داده در کاربردهای متفاوت دیگری نیز به کار گرفته می‌شوند.

در این فصل در مورد هر کدام یک از این ساختمان‌های داده مطالبی بیان خواهیم و برنامه‌های برای ایجاد و نگهداری این ساختمان داده‌ها خواهیم نوشت. از کلاس‌ها، توارث و ترکیب برای ایجاد بسته‌ها (package) و ساختمان‌های داده برای افزایش قابلیت برنامه‌ها استفاده خواهیم کرد.

**21-2 کلاس‌های خود ارجاعی**

یک کلاس خود ارجاعی، حاوی بخشی است که اشاره به یک شی کلاس، از همان نوع کلاس دارد. برای مثال، کلاس تعریف شده در زیر

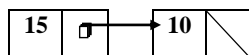
```
class Node
{
public:
    Node( int ); // constructor
    void setData( int ); // set data member
    int getData() const; // get data member
    void setNextPtr( Node * ); // set pointer to next Node
    Node *getNextPtr() const; // get pointer to next Node
private:
    int data; // data stored in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



## ساختمان‌های داده فصل بیست و یکم 1053

تعریف کننده نوع **Node** است. این نوع دارای دو عضو داده **private** به نام‌های **data** از نوع عدد صحیح و **nextPtr** است که به **Node** اشاره دارد. عضو **nextPtr** اشاره به شی از نوع **Node** دارد، همان نوع بعنوان کلاس جاری به عضو **nextPtr** بعنوان یک لینک (پیوند) نگاه می‌شود (به این معنی که **nextPtr** می‌تواند برای گره زدن یک شی از نوع **Node** به شی دیگری از همان نوع بکار گرفته شود). همچنین نوع **Node** دارای پنج تابع عضو است: یک سازنده که یک مقدار صحیح برای مقدار دهی اولیه عضو **data** دریافت می‌کند، یک تابع **setData** برای تنظیم مقدار عضو **data**، یک تابع **getData** برای برگشت دادن مقدار از عضو **data**، تابع **setNextPtr** برای تنظیم مقدار عضو **nextPtr** و تابع **getNextPtr** برای برگشت دادن مقدار از عضو **nextPtr**.

شی‌های خود ارجاعی می‌توانند به یکدیگر متصل شده و ساختمان‌های داده مناسب‌تری همانند لیست‌ها، صف‌ها، پشته‌ها و درخت‌ها ایجاد کنند. شکل 1-21 تصویر دو شی خود ارجاعی متصل به هم را که تشکیل یک لیست را داده نشان می‌دهد. یک کاراکتر **|** که در دومین شی خود ارجاعی جای گرفته نشان می‌دهد که لینک به شی دیگری اشاره نمی‌کند. معمولاً از **null** برای تعریف انتهای یک ساختمان داده استفاده می‌شود.



شکل 1-21 | شی‌های کلاس خود ارجاعی که به هم متصل شده‌اند.

### خطای برنامه‌نویسی

در صورتیکه آخرین گره در یک لیست (یا دیگر ساختمان‌های داده خطی) با **null** تنظیم نشود با خطای منطقی مواجه خواهید شد.



## 3-21 اخذ دینامیکی حافظه و ساختمان داده

ایجاد و نگهداری ساختمان‌های داده دینامیکی مستلزم اخذ دینامیکی حافظه است، قابلیت بدست آوردن حافظه بیشتر (برای نگهداری متغیرهای جدید) و آزاد کردن حافظه‌ای که در زمان اجرای برنامه دیگر به آن نیازی نیست. اخذ دینامیکی حافظه محدود به حافظه فیزیکی موجود در کامپیوتر است (و مقدار فضای موجود بر روی دیسک در حافظه مجازی سیستم). در بیشتر موارد، حافظه موجود در کامپیوتر بایستی مابین برنامه‌های دیگر به اشتراک گذاشته شود.

عملگر **new** برای اخذ حافظه دینامیکی لازم است. کلمه کلیدی **new** نام کلاسی را بعنوان یک آرگومان دریافت می‌کند. سپس بصورت دینامیکی، حافظه را برای شی جدید اخذ کرده، و اشاره‌گری به شی جدیداً ایجاد شده برگشت می‌دهد. برای مثال، عبارت:



## 1054 فصل بیست و یکم ساختمان‌های داده

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

مقدار حافظه متناسب را برای ذخیره سازی Node به اجرا گذاشته، سازنده Node اجرا شده و آدرس شی جدید Node به newPtr تخصیص داده می‌شود. اگر حافظه در دسترس نباشد، new یک استثناء از نوع bad\_alloc به راه می‌اندازد. مقدار 10 به سازنده Node ارسال شده و عضو داده Node با 10 مقداردهی اولیه می‌شود.

عملگر delete مبادرت به اجرای نابود کننده Node کرده و حافظه اخذ شده با new را بازپس می‌گیرد. حافظه به سیستم برگردانده می‌شود. برای آزاد کردن حافظه‌ای که به روش دینامیکی در عبارت فوق اخذ شده است، از عبارت زیر استفاده می‌شود

```
delete newPtr;
```

دقت کنید که خود newPtr حذف نمی‌شود، بجای آن فضایی که newPtr به آن اشاره می‌کند حذف می‌گردد. اگر اشاره گر newPtr دارای اشاره گر null با مقدار صفر باشد، عبارت فوق تاثیری نخواهد داشت. حذف اشاره گر null خطا نیست.

در بخش‌های بعدی به توضیح لیست‌ها، پشته‌ها، صف‌ها و درخت‌ها می‌پردازیم. این ساختمان‌های داده با اخذ حافظه دینامیکی و کلاس‌های خود ارجاعی ایجاد و نگهداری می‌شوند.

### 21-4 لیست‌های پیوندی

یک لیست پیوندی (Linked list) مجموعه خطی، از شی‌های کلاس خود ارجاعی به نام گره است که توسط لینک‌های اشاره گر به یکدیگر متصل شده‌اند. یک لیست پیوندی از طریق اشاره گر به اولین گره لیست قابل دسترس است. گره‌های بعدی از طریق قسمت لینک اشاره گر که در هر گره ذخیره شده‌اند، قابل دسترس‌اند. طبق قاعده لینک اشاره گر آخرین گره لیست با null تنظیم می‌شود که انتهای لیست را نشان می‌دهد. داده‌ها در لیست پیوندی بصورت دینامیکی ذخیره می‌شوند و هر گره زمانی ایجاد می‌شود که به آن نیاز است. یک گره می‌تواند شامل هر نوع داده‌ای از جمله شی‌هایی از سایر کلاس‌ها باشد. پشته‌ها و صف‌ها جزء ساختمان داده‌های خطی هستند و همانطوری که خواهید دید، آنها نوعی لیست پیوندی می‌باشند، در حالیکه درخت‌ها جزء داده‌های غیرخطی هستند.

اگر چه داده‌ها را می‌توان در آرایه‌ها ذخیره کرد، اما لیست‌های پیوندی مزیت‌های نسبت به آرایه‌ها دارند، از جمله هنگامی که تعداد عناصر داده مشخص نباشد، کاربرد لیست‌ها بسیار مناسب است. لیست‌های پیوندی حالت دینامیکی دارند و از اینرو طول لیست می‌تواند در صورت نیاز افزایش یا کاهش



## ساختمان‌های داده فصل بیست و یکم 1055

یابد. در حالیکه در آرایه‌ها با سائز ثابت چنین کاری نمی‌توان انجام داد چرا که سائز آرایه در هنگام ایجاد تعیین می‌شود. آرایه می‌تواند پر شود اما لیست‌های پیوندی فقط در زمانیکه سیستم با کمبود حافظه مواجه شود و نتواند حافظه‌ای در اختیار لیست قرار دهد، پر می‌شوند. برنامه‌نویسان می‌توانند با وارد کردن هر عنصر جدید در مکان مناسب در لیست، همواره لیست پیوندی را بصورت مرتب شده نگهداری کنند. اگر چه انجام اینکار زمانبر است اما نیازی به جابجا کردن عناصر لیست ندارد.

### کارآیی



عمل درج و حذف در یک آرایه مرتب شده زمانگیر است چرا که تمام عضوهای موجود در آرایه بایستی بصورت صحیح جابجا شوند.

### کارآیی

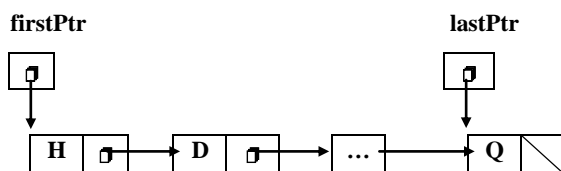


عناصر آرایه بصورت به هم پیوسته در حافظه ذخیره می‌شوند. از اینرو می‌توان بلافاصله به عنصر مورد نظر در آرایه با توجه به آدرس شروع آرایه دسترسی پیدا کرد، در حالیکه در لیست‌های پیوندی نمی‌توان چنین دسترسی داشت.

معمولاً گره‌های لیست پیوندی در حافظه پشت سرهم و متصل به یکدیگر ذخیره نمی‌شود. در عوض گره‌ها بصورت منطقی در پشت سرهم قرار دارند. شکل 2-21 نشاندهنده یک لیست پیوندی متشکل از چند گره است.

### پیاده‌سازی لیست پیوندی

در برنامه‌های 3-21 الی 5-21 از یک الگوی کلاس List برای کار با لیستی از مقادیر صحیح و لیستی از مقادیر اعشاری است. برنامه راه‌انداز (شکل 5-21) پنج گزینه دارد: (1) مقداری به ابتدای لیست اضافه می‌کند، (2) مقداری به انتهای لیست اضافه می‌کند، (3) مقداری را از ابتدای لیست حذف می‌کند، (4) مقداری را از انتهای لیست حذف می‌کند و (5) پردازش لیست را پایان می‌دهد.



شکل 2-21 | نمایش گرافیکی از یک لیست پیوندی.

برنامه از الگوهای کلاس ListNode (شکل 3-21) و List (شکل 4-21) استفاده می‌کند. کپسوله کردن در هر شی List، یک لیست پیوندی از شی‌های ListNode است. کلاس ListNode (شکل 3-21)



## 1056 فصل بیست و یکم ساختمان‌های داده

حاوی دو عضو داده **data** و **nextPtr** (خطوط 19-20)، یک سازنده برای مقداردهی این اعضا و تابع **getData** برای برگشت دادن داده در گره است. عضو **data** مقداری از نوع **NODETYPE**، نوع پارامتر ارسالی به الگوی کلاس را ذخیره می‌کند. عضو **nextPtr** مبادرت به ذخیره کردن اشاره‌گر به شی بعدی **ListNode** در لیست پیوندی می‌کند.

خطوط 24-25 از کلاس **List**، شکل 4-21، متشکل از اعضای داده **firstPtr** (اشاره‌کننده به اولین **ListNode** در یک **List**) و **lastNode** (اشاره‌کننده به آخرین **ListNode** در یک **List**) است. سازنده پیش‌فرض (خطوط 32-37) مبادرت به مقداردهی اولیه هر دو اشاره‌گر با **null** می‌کند. نبود کننده در خطوط 40-60 ما را مطمئن می‌کند که تمام شی‌های **ListNode** در یک شی **List** زمانیکه شی **List** حذف می‌شود، نابود می‌شوند. توابع اصلی **List** عبارتند از **insertAtfront** (خطوط 63-75)، **insertAtBack** (خطوط 78-90)، **removeFromFront** (خطوط 93-111) و **removeFromBack** (خطوط 114-141).

```
1 // Fig. 21.3: 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
```

شکل 3-21 | تعریف کلاس ListNode.

```
1 // Fig. 21.4: List.h
2 // Template List class definition.
3 #ifndef LIST_H
4 #define LIST_H
5
6 #include <iostream>
7 using std::cout;
8
9 #include "ListNode.h" // ListNode class definition
10
11 template< typename NODETYPE >
12 class List
13 {
14 public:
15     List(); // constructor
16     ~List(); // destructor
17     void insertAtFront( const NODETYPE & );
18     void insertAtBack( const NODETYPE & );
19     bool removeFromFront( NODETYPE & );
20     bool removeFromBack( NODETYPE & );
21     bool isEmpty() const;
22     void print() const;
23 private:
24     ListNode< NODETYPE > *firstPtr; // pointer to first node
25     ListNode< NODETYPE > *lastPtr; // pointer to last node
26
27     // utility function to allocate new node
28     ListNode< NODETYPE > *getNewNode( const NODETYPE & );
29 }; // end class List
30
31 // default constructor
32 template< typename NODETYPE >
33 List< NODETYPE >::List()
34     : firstPtr( 0 ), lastPtr( 0 )
35 {
36     // empty body
37 } // end List constructor
38
39 // destructor
40 template< typename NODETYPE >
41 List< NODETYPE >::~~List()
42 {
43     if ( !isEmpty() ) // List is not empty
44     {
45         cout << "Destroying nodes ...\n";
46
47         ListNode< NODETYPE > *currentPtr = firstPtr;
48         ListNode< NODETYPE > *tempPtr;
49
50         while ( currentPtr != 0 ) // delete remaining nodes
51         {
52             tempPtr = currentPtr;
53             cout << tempPtr->data << '\n';
54             currentPtr = currentPtr->nextPtr;
55             delete tempPtr;
56         } // end while
57     } // end if
```



```
58
59     cout << "All nodes destroyed\n\n";
60 } // end List destructor
61
62 // insert node at front of list
63 template< typename NODETYPE >
64 void List< NODETYPE >::insertAtFront( const NODETYPE &value )
65 {
66     ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node
67
68     if ( isEmpty() ) // List is empty
69         firstPtr = lastPtr = newPtr; // new list has only one node
70     else // List is not empty
71     {
72         newPtr->nextPtr = firstPtr; // point new node to previous 1st node
73         firstPtr = newPtr; // aim firstPtr at new node
74     } // end else
75 } // end function insertAtFront
76
77 // insert node at back of list
78 template< typename NODETYPE >
79 void List< NODETYPE >::insertAtBack( const NODETYPE &value )
80 {
81     ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node
82
83     if ( isEmpty() ) // List is empty
84         firstPtr = lastPtr = newPtr; // new list has only one node
85     else // List is not empty
86     {
87         lastPtr->nextPtr = newPtr; // update previous last node
88         lastPtr = newPtr; // new last node
89     } // end else
90 } // end function insertAtBack
91
92 // delete node from front of list
93 template< typename NODETYPE >
94 bool List< NODETYPE >::removeFromFront( NODETYPE &value )
95 {
96     if ( isEmpty() ) // List is empty
97         return false; // delete unsuccessful
98     else
99     {
100         ListNode<NODETYPE> *tempPtr = firstPtr; // hold tempPtr to delete
101
102         if ( firstPtr == lastPtr )
103             firstPtr = lastPtr = 0; // no nodes remain after removal
104         else
105             firstPtr = firstPtr->nextPtr; // point to previous 2nd node
106
107         value = tempPtr->data; // return data being removed
108         delete tempPtr; // reclaim previous front node
109         return true; // delete successful
110     } // end else
111 } // end function removeFromFront
112
113 // delete node from back of list
114 template< typename NODETYPE >
115 bool List< NODETYPE >::removeFromBack( NODETYPE &value )
116 {
117     if ( isEmpty() ) // List is empty
118         return false; // delete unsuccessful
119     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
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 >::getNode(
152     const NODETYPE &value )
153 {
154     return new ListNode< NODETYPE >( value );
155 } // end function getNode
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

```



```
1 // Fig. 21.5: Fig21_05.cpp
2 // List class test program.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9 using std::string;
10
11 #include "List.h" // List class definition
12
13 // function to test a List
14 template< typename T >
15 void testList( List< T > &listObject, const string &typeName )
16 {
17     cout << "Testing a List of " << typeName << " values\n";
18     instructions(); // display instructions
19
20     int choice; // store user choice
21     T value; // store input value
22
23     do // perform user-selected actions
24     {
25         cout << "? ";
26         cin >> choice;
27
28         switch ( choice )
29         {
30             case 1: // insert at beginning
31                 cout << "Enter " << typeName << ": ";
32                 cin >> value;
33                 listObject.insertAtFront( value );
34                 listObject.print();
35                 break;
36             case 2: // insert at end
37                 cout << "Enter " << typeName << ": ";
38                 cin >> value;
39                 listObject.insertAtBack( value );
40                 listObject.print();
41                 break;
42             case 3: // remove from beginning
43                 if ( listObject.removeFromFront( value ) )
44                     cout << value << " removed from list\n";
45
46                 listObject.print();
47                 break;
48             case 4: // remove from end
49                 if ( listObject.removeFromBack( value ) )
50                     cout << value << " removed from list\n";
51
52                 listObject.print();
53                 break;
54         } // end switch
55     } while ( choice != 5 ); // end do...while
56
57     cout << "End list test\n\n";
58 } // end function testList
59
60 // display program instructions to user
```



```
61 void instructions()
62 {
63     cout << "Enter one of the following:\n"
64         << " 1 to insert at beginning of list\n"
65         << " 2 to insert at end of list\n"
66         << " 3 to delete from beginning of list\n"
67         << " 4 to delete from end of list\n"
68         << " 5 to end list processing\n";
69 } // end function instructions
70
71 int main()
72 {
73     // test List of int values
74     List< int > integerList;
75     testList( integerList, "integer" );
76
77     // test List of double values
78     List< double > doubleList;
79     testList( doubleList, "double" );
80     return 0;
81 } // 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 remove from list
The list is: 1 3 4

? 3
1 remove from list
The list is: 3 4

? 4
4 remove from list
The list is: 4

? 4
3 remove 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
```

شکل 5-21 | کار با لیست پیوندی.

تابع `isEmpty` (خطوط 144-148) یک تابع پیشگو است که تعیین می‌کند آیا لیست تهی است یا خیر. اگر لیست تهی باشد، تابع `isEmpty` مقدار `true` و در غیر اینصورت `false` باز می‌گرداند. تابع `print` (خطوط 159-179) محتویات لیست را به نمایش در می‌آورد. تابع کمکی `getNode` در خطوط 151-156 یک شی `ListNode` اخذ شده به روش دینامیکی را برگشت می‌دهد. این تابع از سوی توابع `insertAtFront` و `insertAtBack` فراخوانی می‌شود.



## ساختمان‌های داده فصل بیست و یکم 1063

برنامه راه‌انداز (شکل 5-21) از تابع `testList` برای اینکه کاربر بتواند با شی‌های کلاس `List` کار کند استفاده کرده است. خطوط 74 و 78 شی‌های لیست را برای نوع‌های `int` و `double` ایجاد می‌کنند. خطوط 75 و 79 تابع `testList` را به همراه این شی‌های `List` فراخوانی می‌کنند.

### تابع عضو `insertAtFront`

در چند صفحه بعدی در مورد عملکرد هر کدامیک از توابع عضو کلاس `List` توضیح خواهیم داد. تابع `insertAtFront` (شکل 4-21، خطوط 63-75) یک گره جدید در ابتدای لیست قرار می‌دهد. این تابع در چند مرحله عمل می‌کند که بصورت زیر می‌توان آنرا توضیح داد (به شکل 6-21 توجه کنید):

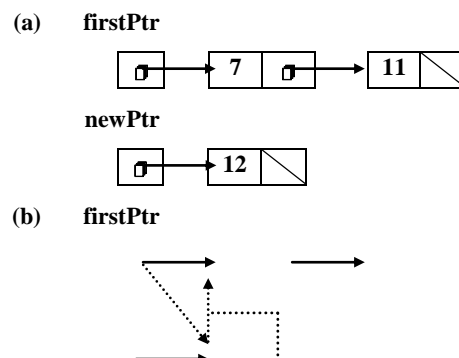
1- فراخوانی تابع `getNodeNew`، ارسال مقدار به آن، که یک مراجعه ثابت به مقدار گره‌ای است که درج می‌شود (خط 66).

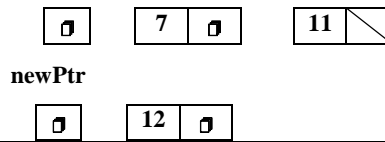
2- تابع `getNodeNew` در خطوط 151-156 از عملگر `new` برای ایجاد یک گره جدید و برگشت دادن یک اشاره‌گر به گره جدیداً اخذ شده استفاده می‌کند، که به `newPtr` در `insertAtFront` تخصیص داده می‌شود (خط 66).

3- اگر لیست تهی باشد، هم `firstPtr` و هم `lastPtr` با `newPtr` تنظیم خواهند شد (خط 69).

4- اگر لیست خالی نباشد (خط 70)، گره مورد اشاره توسط `newPtr` به لیست با کپی `firstPtr` به `newPtr->nextPtr` متصل می‌شود (خط 72)، از اینرو است که گره جدید به اولین گره لیست اشاره می‌کند، و با کپی کردن `newPtr` به `firstPtr` (خط 73) است که `firstPtr` به اولین گره جدید در لیست اشاره می‌کند.

شکل 6-21 نحوه عملکرد تابع `insertAtFront` را نشان می‌دهد. بخش (a) در حال نمایش لیست و گره جدیدی است که در عملیات `insertAtFront` بوجود آمده و قبل از وصل کردن گره جدید (حاوی مقدار 12) به لیست است. فلش‌های نقطه‌چین در بخش (b) مراحل عملکرد `insertAtFront` را که در مرحله 4 شرح داده شده است و در تبدیل به گره اول در لیست می‌شود را نشان می‌دهند.



شکل 21-6 | نمایش گرافیکی از عملکرد `insertAtFront`.تابع عضو `insertAtBack`

تابع `insertAtBack` (شکل 21-4، خطوط 78-90) یک گره جدید در انتهای لیست قرار می‌دهد. عملکرد این تابع در چهار مرحله است (شکل 21-7):

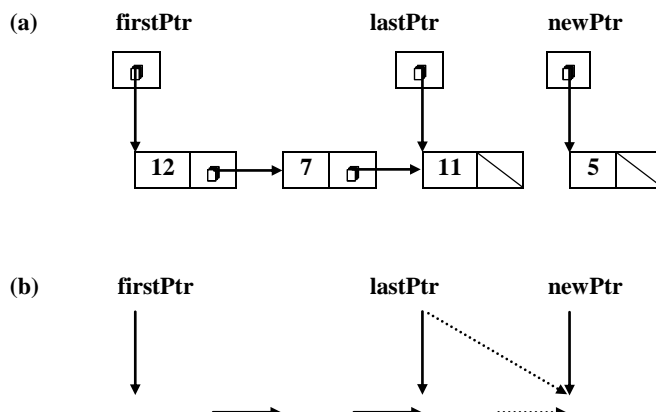
1- فراخوانی تابع `getNode`، ارسال مقدار به آن، که یک مراجعه ثابت به مقدار گره‌ای است که درج می‌شود (خط 81).

2- تابع `getNode` در خطوط 151-156 از عملگر `new` برای ایجاد یک گره جدید و برگشت دادن یک اشاره گر به گره جدیداً اخذ شده استفاده می‌کند، که به `newPtr` در `insertAtBack` تخصیص داده می‌شود (خط 81).

3- اگر لیست تهی باشد، هم `firstPtr` و هم `lastPtr` با `newPtr` تنظیم خواهند شد (خط 83).

4- اگر لیست خالی نباشد (خط 85)، گره مورد اشاره توسط `newPtr` به لیست با کپی `newPtr` به `lastPtr->nextPtr` متصل می‌شود (خط 87)، از اینرو است که گره جدید به آخرین گره لیست اشاره می‌کند، و با کپی کردن `newPtr` به `lastPtr` (خط 88) است که `lastPtr` به آخرین گره جدید در لیست اشاره می‌کند.

شکل 21-7 مراحل عملکرد تابع `insertAtBack` را نشان می‌دهد. بخش (a) در حال نمایش لیست و گره جدید (با مقدار 5) بوجود آمده در عملیات `insertBack` و قبل از متصل شدن گره به لیست است. فلش‌های نقطه‌چین در بخش (b) مراحل که تابع `insertAtBack` طی می‌کند تا گره جدید به انتهای لیست افزوده شود، را نشان می‌دهند.





شکل 7-21 | نمایش گرافیکی از عملکرد `insertAtBack`.

تابع `removeFromFront`

تابع `removeFromFront` (شکل 4-21، خطوط 93-111) مبادرت به حذف گره ابتدایی از لیست کرده و اشاره‌گری به داده حذف شده باز می‌گرداند. اگر برنامه سعی در حذف گره از یک لیست تهی نماید، تابع مقدار `false` و در غیر اینصورت مقدار `true` برگشت می‌دهد (خطوط 96-97). این تابع در شش مرحله وظیفه خود را به انجام می‌رساند:

1- تخصیص آدرس `tempPtr` به جای که `firstPtr` اشاره می‌کند (خط 100). سرانجام، `tempPtr` برای حذف گره بکار گرفته می‌شود.

2- اگر `firstPtr` و `lastPtr` یکی باشند (خط 102) پس لیست حاوی یک عنصر است. در این حالت، تابع مبادرت به تنظیم `firstPtr` و `lastPtr` با 0 (خط 103) می‌کند، تا گره از لیست حذف شود (لیست تهی می‌شود).

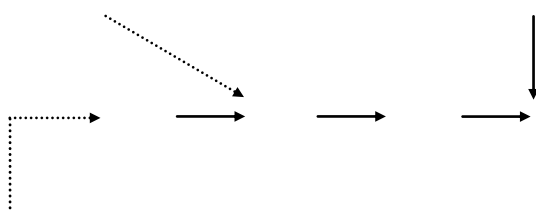
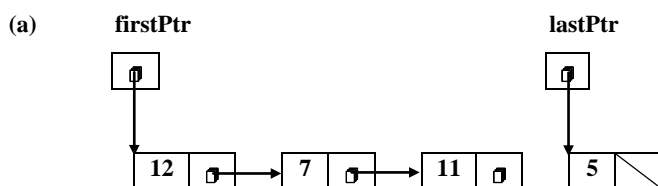
3- اگر لیست حاوی بیش از یک گره باشد، گره اولی حذف خواهد شد، در اینحالت اشاره‌گر `lastPtr` در جای خود باقی می‌ماند و فقط `firstPtr` با `firstPtr->nextPtr` تنظیم می‌شود (خط 105). از اینرو، `firstPtr` به گره دوم قبل از فراخوانی تابع `removeFromFront` اشاره می‌کند.

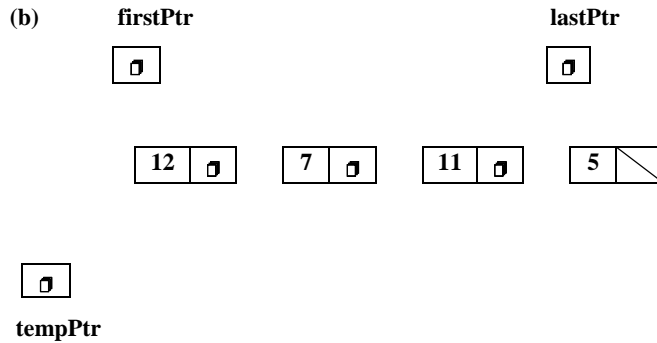
4- پس از اینکه کار تمام این اشاره‌گرها تمام شد، مقدار عضو `data` متعلق به گره‌ای که حذف خواهد شد، به `value` تخصیص داده می‌شود (خط 107).

5- حال گره مورد اشاره توسط `tempPtr` حذف می‌شود (خط 108).

6- برگشت `true`، نشان می‌دهد که حذف با موفقیت صورت گرفته است (خط 109).

شکل 8-21 | مراحل عملکرد تابع `removeFromFront` را نشان می‌دهد. بخش (a) در حال نمایش لیست قبل از انجام عمل حذف است. بخش (b) نحوه جابجایی اشاره‌گر را نشان می‌دهد.





شکل 8-21 | نمایش گرافیکی عملکرد `removeFromFront`.

تابع عضو `removeFromBack`

تابع `removeFromBack` (شکل 4-21، خطوط 114-141) آخرین گره از لیست را حذف و مراجعه‌ای به داده حذف شده باز می‌گرداند. اگر برنامه مبادرت به حذف گره از یک لیست تهی نماید، تابع مقدار `false` (خطوط 117-118) و در غیر اینصورت مقدار `true` برگشت خواهد داد. عملکرد این تابع در نه مرحله صورت می‌گیرد (شکل 9-21):

1- تخصیص آدرس `tempPtr` به جای که `lastPtr` اشاره می‌کند (خط 121). سرانجام، `tempPtr` برای حذف گره بکار گرفته می‌شود..

2- اگر `firstPtr` و `lastPtr` یکی باشند (خط 123) پس لیست حاوی یک عنصر است. در این حالت، تابع مبادرت به تنظیم `firstPtr` و `lastPtr` با 0 (خط 124) می‌کند، تا گره از لیست حذف شود (لیست تهی می‌شود).

3- اگر لیست حاوی بیش از یک گره باشد، اشاره‌گر `currentPtr` ایجاد و به `firstPtr` تخصیص داده می‌شود (خط 127).

4- از `currentPtr` برای پیمایش لیست تا رسیدن آن به گره ماقبل آخر استفاده می‌شود. حلقه `while` (خطوط 130-131) تا مادامیکه `currentPtr->nextPtr` برابر `lastPtr` نشده، آنرا به `currentPtr` اشاره می‌دهد.

5- پس از یافتن گره ما قبل آخر، `currentPtr` به `lastPtr` تخصیص یافته (خط 133) و موجب می‌شود تا گره آخر از لیست جدا شود.

6- تنظیم `currentPtr->nextPtr` گره جدید آخر لیست با 0 (خط 134).





7- پس از اینکه کار تمام این اشاره‌گرها تمام شد، مقدار عضو **data** متعلق به گره‌ای که حذف خواهد شد، به **value** تخصیص داده می‌شود (خط 137).

8- حال گره مورد اشاره توسط **tempPtr** حذف می‌شود (خط 138).

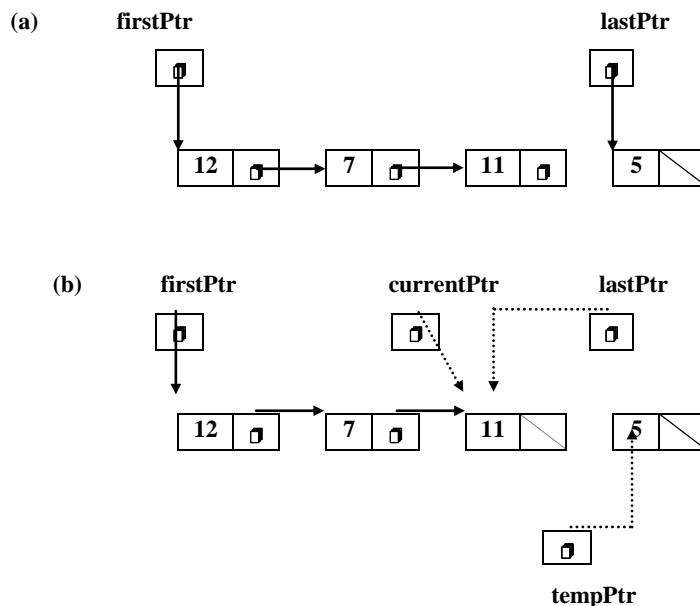
9- برگشت **true** نشان می‌دهد که حذف با موفقیت صورت گرفته است (خط 139).

شکل 9-21 مراحل عملکرد تابع **removeFromBack** را نشان می‌دهد. بخش (a) در حال نمایش لیست قبل از انجام عمل حذف است. بخش (b) نحوه جابجایی اشاره‌گر را نشان می‌دهد.

## 5-23 پشته‌ها

پشته‌ها فضاهای برای متغیرهای محلی به هنگام فراخوانی روال ایجاد می‌کنند. هنگامی که روال به روال فراخوان خود باز می‌گردد، فضای اخذ شده برای متغیرها از پشته حذف می‌شود و دیگر متغیرها در برنامه شناخته نمی‌شوند. همچنین از پشته‌ها برای ارزیابی عبارات محاسباتی در کامپایلرها و ایجاد کد زبان ماشین برای این عبارات استفاده می‌شود.

با استفاده از مزایای لیست‌ها و پشته‌ها، کلاس پشته مورد نظر خود را پیاده‌سازی می‌کنیم (با استفاده مجدد از کلاس لیست). به توضیح دو نوع متفاوت از بکارگیری استفاده مجدد می‌پردازیم. ابتدا کلاس پشته را با ارث‌بری از کلاس **List** پیاده‌سازی می‌کنیم. سپس همان کلاس پشته را از طریق ترکیب با بکارگیری یک شی **List** بعنوان یک عضو **private** در کلاس پشته پیاده‌سازی خواهیم کرد.



شکل 9-21 | نمایش گرافیکی عملکرد `removeFromBack`.

برنامه‌های شکل 13-21 و 14-21 یک کلاس پشته (شکل 13-21) را از طریق ارث‌بری (خط 9) از کلاس `List` موجود در برنامه 4-21 ایجاد می‌کنند. می‌خواهیم که پشته دارای توابع `push` (خطوط 13-16)، `pop` (خطوط 19-22)، `isEmpty` (خطوط 25-28) و `printStack` (خطوط 31-34) باشد. ضرورتاً، آنها توابع `insertAtFront`، `removeFromFront`، `isEmpty` و `print` از کلاس `List` خواهند بود. کلاس `List` حاوی کلاس‌های دیگری همانند `insertAtBack` و `removeFromBack` است که مایل نیستیم از طریق واسط `public` پشته در دسترس قرار گیرند.

به هنگام پیاده‌سازی تابعهای پشته، تابعهای متناسب در `List` فراخوانی خواهند شد، تابع `push` تابع `insertAtFront` (خط 15)، تابع `pop` تابع `removeFromFront` (خط 21) و تابع `isEmpty` تابع `isEmpty` (خط 27) و تابع `printStack` تابع `print` (خط 33) را فراخوانی می‌کند.

```
1 // Fig. 21.13: 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     // isEmpty calls the List function isEmpty
25     bool isEmpty() const
26     {
27         return isEmpty();
28     } // end function isEmpty
29
30     // printStack calls the List function print
31     void printStack() const
32     {
33         print();
34     } // end function print
35 }; // end class Stack
36
37 #endif
```



شکل 21-13 | تعریف کلاس پشته.

کلاس پشته در **main** (شکل 21-14) برای نمونه‌سازی پشته صحیح **intStack** از نوع **Stack<int>** بکار گرفته شده است (خط 11). مقادیر صحیح 0 تا 2 وارد پشته **intStack** شده (خطوط 16-20)، سپس از آن خارج می‌شوند (خطوط 25-30). برنامه از کلاس پشته برای ایجاد یک **doubleStack** از نوع **Stack<double>** استفاده کرده است (خط 32). مقادیر 1.1، 2.2 و 3.3 وارد **doubleStack** شده (خطوط 38-43) و سپس از آن خارج شده‌اند (خطوط 48-53).

```
1 // Fig. 21.14: Fig21_14.cpp
2 // Template Stack class test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Stack.h" // Stack class definition
8
9 int main()
10 {
11     Stack< int > intStack; // create Stack of ints
12
13     cout << "processing an integer Stack" << endl;
14
15     // push integers onto intStack
16     for ( int i = 0; i < 3; i++ )
17     {
18         intStack.push( i );
19         intStack.printStack();
20     } // end for
21
22     int popInteger; // store int popped from stack
23
24     // pop integers from intStack
25     while ( !intStack.isStackEmpty() )
26     {
27         intStack.pop( popInteger );
28         cout << popInteger << " popped from stack" << endl;
29         intStack.printStack();
30     } // end while
31
32     Stack< double > doubleStack; // create Stack of doubles
33     double value = 1.1;
34
35     cout << "processing a double Stack" << endl;
36
37     // push floating-point values onto doubleStack
38     for ( int j = 0; j < 3; j++ )
39     {
40         doubleStack.push( value );
41         doubleStack.printStack();
42         value += 1.1;
43     } // end for
44
45     double popDouble; // store double popped from stack
46
47     // pop floating-point values from doubleStack
48     while ( !doubleStack.isStackEmpty() )
49     {
```



```
50     doubleStack.pop( popDouble );
51     cout << popDouble << " popped from stack" << endl;
52     doubleStack.printStack();
53 } // end while
54
55 return 0;
56 } // 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
```

شکل 21-14 | برنامه ساده پشته.

روش دیگر در پیاده‌سازی یک کلاس پشته از طریق استفاده مجدد از کلاس لیست و به کمک ترکیب است. کلاس موجود در برنامه شکل 21-15 پیاده‌سازی جدیدی از کلاس پشته است که حاوی یک شی `List<STACKTYPE>` بنام `stackList` می‌باشد (خط 38). این نسخه از کلاس پشته از کلاس `List` شکل 21-4 استفاده می‌کند. برای تست این کلاس، از برنامه راه‌انداز در شکل 21-14 استفاده شده است، اما حاوی فایل سرآیند `Stackcomposition.h` در خط 6 است. خروجی برنامه با هر دو نسخه کلاس پشته یکسان است.

```
1 // Fig. 21.15: 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     // isEmpty calls stackList object's isEmpty member function
27     bool isEmpty() const
28     {
29         return stackList.isEmpty();
30     } // end function isEmpty
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
```

شکل 15-21 | کلاس پشته با ترکیب شی لیست.

## 21-6 صف‌ها

نوع دیگری از ساختمان داده‌ها، صف (*Queue*) است. صف شبیه به یک صف انتظار (نوبت) در یک فروشگاه است، که در آن ابتدا به اولین شخص در سر صف، سرویس داده می‌شود و مشتری‌های بعدی باید در انتظار رسیدن نوبت خود باقی بمانند. در صف گره‌ها از ابتدا یا سر صف (*head*) حذف می‌شوند و اضافه کردن گره به صف از انتهای (*tail*) آن صورت می‌گیرد، به همین دلیل به ساختمان داده صف، اولین ورودی-اولین خروجی (*FIFO*) گفته می‌شود. به عمل افزودن گره به صف *enqueue* و حذف گره از صف *dequeue* گفته می‌شود.

در سیستم‌های کامپیوتری صف‌ها کاربردهای متفاوتی دارند. بیشتر کامپیوترها دارای یک پردازنده هستند، بنابراین فقط یک کاربرد در هر زمان می‌تواند سرویس بگیرد و تقاضای سایر کاربران در صف قرار می‌گیرد. همچنین از صف‌ها در فرآیند چاپ هم استفاده می‌شود. در یک محیط چند کاربره ممکن است، فقط یک چاپگر وجود داشته باشد و کاربران هم برای ارسال مستندات خود به چاپگر نیاز داشته



## 1072 فصل بیست و یکم ساختمان‌های داده

باشند. اگر چاپگر مشغول باشد، ممکن است خروجی‌های دیگری در حال تولید باشند و از اینرو یک صف انتظار برای دسترسی به چاپگر تشکیل می‌شود.

در شبکه‌های کامپیوتری بسته‌های اطلاعاتی در صف‌های انتظار قرار می‌گیرند و در هر زمان یک بسته به عنوان گره به شبکه ارسال می‌شود و اگر نیاز باشد تا رسیدن به مقصد این بسته جابجا می‌شود.

برنامه‌های شکل 21-16 و 21-17 کلاس صف (Queue) را از طریق ارث‌بری (خط 9) از کلاس List ایجاد می‌کنند (شکل 21-4). می‌خواهیم کلاس Queue حاوی توابع enqueue (خطوط 13-16)، dequeue (خطوط 19-22)، isEmpty (خطوط 25-28) و printQueue (خطوط 31-34) باشد. دقت کنید که این توابع ضرورتاً تابعهای insertAtBack، removeFromFront، isEmpty و print از کلاس List هستند.

```
1 // Fig. 21.16: 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     // isEmpty calls List member function isEmpty
25     bool isEmpty() const
26     {
27         return isEmpty();
28     } // end function isEmpty
29
30     // printQueue calls List member function print
31     void printQueue() const
32     {
33         print();
34     } // end function printQueue
35 }; // end class Queue
36
37 #endif
```

شکل 21-16 | تعریف کلاس صف.



به هنگام پیاده‌سازی توابع صف، توابع متناسب در **List** فراخوانی خواهند شد، تابع **enqueue** تابع **insertAtBack** (خط 15)، تابع **dequeue** تابع **removeFromFront** (خط 21) و تابع **isEmpty** تابع **isQueueEmpty** (خط 27) و تابع **printQueue** تابع **print** (خط 33) را فراخوانی می‌کند.

کلاس صف در شکل 21-17 برای نمونه‌سازی صف صحیح **intQueue** از نوع **Queue<int>** بکار گرفته شده است (خط 11). مقادیر صحیح 0 تا 2 وارد صف **intQueue** شده (خطوط 20-16)، سپس از آن خارج می‌شوند (خطوط 25-30). برنامه از کلاس صف برای ایجاد یک **doubleQueue** از نوع **Queue<double>** استفاده کرده است (خط 32). مقادیر 1.1، 2.2 و 3.3 وارد **doubleQueue** شده (خطوط 38-43) و سپس از آن خارج شده‌اند (خطوط 48-53).

```
1 // Fig. 21.17: Fig21_17.cpp
2 // Template Queue class test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Queue.h" // Queue class definition
8
9 int main()
10 {
11     Queue< int > intQueue; // create Queue of integers
12
13     cout << "processing an integer Queue" << endl;
14
15     // enqueue integers onto intQueue
16     for ( int i = 0; i < 3; i++ )
17     {
18         intQueue.enqueue( i );
19         intQueue.printQueue();
20     } // end for
21
22     int dequeueInteger; // store dequeued integer
23
24     // dequeue integers from intQueue
25     while ( !intQueue.isQueueEmpty() )
26     {
27         intQueue.dequeue( dequeueInteger );
28         cout << dequeueInteger << " dequeued" << endl;
29         intQueue.printQueue();
30     } // end while
31
32     Queue< double > doubleQueue; // create Queue of doubles
33     double value = 1.1;
34
35     cout << "processing a double Queue" << endl;
36
37     // enqueue floating-point values onto doubleQueue
38     for ( int j = 0; j < 3; j++ )
39     {
40         doubleQueue.enqueue( value );
41         doubleQueue.printQueue();
42         value += 1.1;
43     } // end for
```



```
44
45     double dequeueDouble; // store dequeued double
46
47     // dequeue floating-point values from doubleQueue
48     while ( !doubleQueue.isEmpty() )
49     {
50         doubleQueue.dequeue( dequeueDouble );
51         cout << dequeueDouble << " dequeued" << endl;
52         doubleQueue.printQueue();
53     } // end while
54
55     return 0;
56 } // end main
```

```
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 an 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
```

شکل 21-17 | برنامه پردازش صف.

## 21-7 درخت‌ها

لیست‌های پیوندی، پشته‌ها و صف‌ها جزء ساختمان داده‌های خطی هستند (متوالی). در حالیکه درخت (tree) یک ساختمان داده خطی نیست و یک ساختمان داده دو بعدی با خصوصیات ویژه خود است. گره‌های درخت دارای دو یا بیشتر از دو لینک هستند. در این بخش در مورد درخت‌های باینری (دودویی)





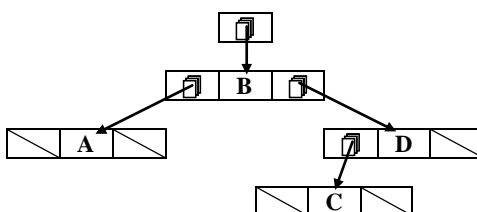
## ساختمان‌های داده فصل بیست و یکم 1075

صحبت خواهیم کرد (شکل 18-21). در این نوع از درخت‌ها همه گره‌ها دارای دو لینک هستند (یکی یا هر دو می‌تواند برابر **null** باشد یا هیچ کدام).

گره ریشه اولین گره در درخت است. هر لینک گره ریشه، به یک فرزند اشاره می‌کند. فرزند چپ، اولین گره در زیر درخت چپ و فرزند راست اولین گره در زیر درخت راست است. به فرزندان یک گره **sibling** و به گره بدون فرزند گره برگ (**leaf node**) گفته می‌شود. دانشمندان کامپیوتر معمولاً درخت‌ها را از ریشه به پایین ترسیم می‌کنند که برخلاف رشد طبیعی درخت در طبیعت است.

### درخت جستجوی باینری

در این بخش یک درخت باینری ویژه بنام درخت جستجوی باینری (*binary search tree*) ایجاد خواهیم کرد. یک درخت جستجوی باینری (با مقادیر غیر تکرار شونده در گره‌ها) دارای خصوصیتی به شرح زیر است: مقادیر موجود در هر زیر درخت چپ کمتر از مقدار گره والد خود است و مقادیر موجود در هر زیر درخت راست بزرگتر از مقدار والد خود می‌باشد. شکل 19-21 یک درخت جستجوی باینری با 12 مقدار صحیح است.



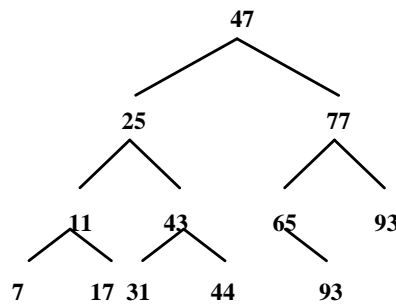
شکل 18-21 | نمایش گرافیکی از یک درخت باینری.

### پیاده‌سازی درخت جستجوی باینری

برنامه شکل‌های 20-21 الی 21-22 یک درخت جستجوی باینری ایجاد و آن را به سه روش بازگشتی *preorder*، *inorder* و *postorder* پیمایش می‌کنند. برنامه مبادرت به ایجاد 10 عدد تصادفی کرده و هر یک را وارد درخت می‌کند. در شکل 17-23، کلاس **Tree** در فضای نامی **BinaryTreeLibrary** تعریف شده است. شکل 18-23 تعریف کننده کلاس **TreeTest** است که توصیف کننده قابلیت‌های **Tree** می‌باشد. تابع **Main** از مدول **TreeTest** یک شی تھی **Tree** نمونه‌سازی کرده، 10 مقدار صحیح تصادفی ایجاد و هر مقدار را در درخت باینری و با استفاده از تابع **InsertNode** وارد می‌کند. سپس برنامه پیمایش‌های *preorder*، *inorder* و *postorder* را بر روی درخت به انجام می‌رساند. در مورد هر کدامیک از این پیمایش‌ها توضیح خواهیم داد.



بحث خود را با برنامه راه‌انداز شکل 21-22 شروع کرده، سپس با پیاده‌سازی کلاس‌های `TreeNode` (شکل 21-20) و `Tree` (شکل 21-21) ادامه می‌دهیم. تابع `main` (شکل 21-22) با نمونه‌سازی درخت `intTree` از نوع `Tree<int>` شروع می‌شود (خط 15). برنامه ده مقدار صحیح درخواست کرده، هر یک را با فراخوانی تابع `insertNode` وارد درخت باینری می‌کند (خط 24). سپس برنامه پیمایش‌های `preorder`, `inorder` و `postorder` را بر روی `intTree` انجام می‌دهد (خطوط 28, 31 و 34). در ادامه، برنامه مبادرت به نمونه‌سازی درخت `doubleTree` از نوع `Tree<double>` می‌کند (خط 36). برنامه ده مقدار `double` درخواست کرده، هر یک را با فراخوانی تابع `insertNode` وارد درخت باینری می‌کند (خط 46). سپس برنامه پیمایش‌های `preorder`, `inorder` و `postorder` را بر روی `doubleTree` انجام می‌دهد (خطوط 53, 56 و 50).



شکل 21-19 | درخت جستجوی باینری.

```
1 // Fig. 21.20: 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 subtree
20     {
21         // empty body
22     } // end TreeNode constructor
23
24     // return copy of node's data
25     NODETYPE getData() const
```



## ساختمان‌های داده فصل بیست و یکم 1077

```
26 {
27     return data;
28 } // end getData function
29 private:
30     TreeNode< NODETYPE > *leftPtr; // pointer to left subtree
31     NODETYPE data;
32     TreeNode< NODETYPE > *rightPtr; // pointer to right subtree
33 }; // end class TreeNode
34
35 #endif
```

شکل 20-21 | تعریف کلاس `TreeNode`.

```
1 // Fig. 21.21: Tree.h
2 // Template Tree class definition.
3 #ifndef TREE_H
4 #define TREE_H
5
6 #include <iostream>
7 using std::cout;
8 using std::endl;
9
10 #include <new>
11 #include "Treenode.h"
12
13 // Tree class-template definition
14 template< typename NODETYPE > class Tree
15 {
16 public:
17     Tree(); // constructor
18     void insertNode( const NODETYPE & );
19     void preOrderTraversal() const;
20     void inOrderTraversal() const;
21     void postOrderTraversal() const;
22 private:
23     TreeNode< NODETYPE > *rootPtr;
24
25     // utility functions
26     void insertNodeHelper(TreeNode< NODETYPE > **, const NODETYPE & );
27     void preOrderHelper( TreeNode< NODETYPE > * ) const;
28     void inOrderHelper( TreeNode< NODETYPE > * ) const;
29     void postOrderHelper( TreeNode< NODETYPE > * ) const;
30 }; // end class Tree
31
32 // constructor
33 template< typename NODETYPE >
34 Tree< NODETYPE >::Tree()
35 {
36     rootPtr = 0; // indicate tree is initially empty
37 } // end Tree constructor
38
39 // insert node in Tree
40 template< typename NODETYPE >
41 void Tree< NODETYPE >::insertNode( const NODETYPE &value )
42 {
43     insertNodeHelper( &rootPtr, value );
44 } // end function insertNode
45
46 // utility function called by insertNode; receives a pointer
47 // to a pointer so that the function can modify pointer's value
48 template< typename NODETYPE >
49 void Tree< NODETYPE >::insertNodeHelper(
```



```
50     TreeNode< NODETYPE > **ptr, const NODETYPE &value )
51 {
52     // subtree is empty; create new TreeNode containing value
53     if ( *ptr == 0 )
54         *ptr = new TreeNode< NODETYPE >( value );
55     else // subtree is not empty
56     {
57         // data to insert is less than data in current node
58         if ( value < ( *ptr )->data )
59             insertNodeHelper( &( ( *ptr )->leftPtr ), value );
60         else
61         {
62             // data to insert is greater than data in current node
63             if ( value > ( *ptr )->data )
64                 insertNodeHelper( &( ( *ptr )->rightPtr ), value );
65             else // duplicate data value ignored
66                 cout << value << " dup" << endl;
67         } // end else
68     } // end else
69 } // end function insertNodeHelper
70
71 // begin preorder traversal of Tree
72 template< typename NODETYPE >
73 void Tree< NODETYPE >::preOrderTraversal() const
74 {
75     preOrderHelper( rootPtr );
76 } // end function preOrderTraversal
77
78 // utility function to perform preorder traversal of Tree
79 template< typename NODETYPE >
80 void Tree<NODETYPE>::preOrderHelper(TreeNode<NODETYPE> *ptr ) const
81 {
82     if ( ptr != 0 )
83     {
84         cout << ptr->data << ' '; // process node
85         preOrderHelper( ptr->leftPtr ); // traverse left subtree
86         preOrderHelper( ptr->rightPtr ); // traverse right subtree
87     } // end if
88 } // end function preOrderHelper
89
90 // begin inorder traversal of Tree
91 template< typename NODETYPE >
92 void Tree< NODETYPE >::inOrderTraversal() const
93 {
94     inOrderHelper( rootPtr );
95 } // end function inOrderTraversal
96
97 // utility function to perform inorder traversal of Tree
98 template< typename NODETYPE >
99 void Tree<NODETYPE>::inOrderHelper(TreeNode<NODETYPE> *ptr ) const
100 {
101     if ( ptr != 0 )
102     {
103         inOrderHelper( ptr->leftPtr ); // traverse left subtree
104         cout << ptr->data << ' '; // process node
105         inOrderHelper( ptr->rightPtr ); // traverse right subtree
106     } // end if
107 } // end function inOrderHelper
108
109 // begin postorder traversal of Tree
110 template< typename NODETYPE >
111 void Tree< NODETYPE >::postOrderTraversal() const
```



```
112 {
113     postOrderHelper( rootPtr );
114 } // end function postOrderTraversal
115
116 // utility function to perform postorder traversal of Tree
117 template< typename NODETYPE >
118 void Tree< NODETYPE >::postOrderHelper(
119     TreeNode< NODETYPE > *ptr ) const
120 {
121     if ( ptr != 0 )
122     {
123         postOrderHelper( ptr->leftPtr ); // traverse left subtree
124         postOrderHelper( ptr->rightPtr ); // traverse right subtree
125         cout << ptr->data << ' '; // process node
126     } // end if
127 } // end function postOrderHelper
128
129 #endif
```

#### شکل 21-21 | تعریف کلاس Tree.

بحث را با تعریف الگوی کلاس **TreeNode** آغاز می‌کنیم (شکل 21-20) که **Tree<NODETYPE>** را بعنوان **friend** اعلان کرده است (خط 13). با اینکار تمام توابع عضو که از کلاس الگوی **Tree** بدست می‌آیند (شکل 21-21) دوستان متناظر کلاس الگوی **TreeNode** می‌شوند، از اینرو است که می‌توانند به اعضای **private** شی‌های **TreeNode** از آن نوع دسترسی پیدا کنند. بدلیل اینکه از پارامتر **NODETYPE** بعنوان آرگومان **Tree** در اعلان **friend** استفاده شده است، **TreeNode** می‌تواند فقط از طریق یک **Tree** با همان نوع پردازش شود.

```
1 // Fig. 21.22: Fig21_22.cpp
2 // Tree class test program.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include "Tree.h" // Tree class definition
12
13 int main()
14 {
15     Tree< int > intTree; // create Tree of int values
16     int intValue;
17
18     cout << "Enter 10 integer values:\n";
19
20     // insert 10 integers to intTree
21     for ( int i = 0; i < 10; i++ )
22     {
23         cin >> intValue;
24         intTree.insertNode( intValue );
25     } // end for
26
27     cout << "\nPreorder traversal\n";
```



```
28     intTree.preOrderTraversal();
29
30     cout << "\nInorder traversal\n";
31     intTree.inOrderTraversal();
32
33     cout << "\nPostorder traversal\n";
34     intTree.postOrderTraversal();
35
36     Tree< double > doubleTree; // create Tree of double values
37     double doubleValue;
38
39     cout << fixed << setprecision( 1 )
40         << "\n\nEnter 10 double values:\n";
41
42     // insert 10 doubles to doubleTree
43     for ( int j = 0; j < 10; j++ )
44     {
45         cin >> doubleValue;
46         doubleTree.insertNode( doubleValue );
47     } // end for
48
49     cout << "\nPreorder traversal\n";
50     doubleTree.preOrderTraversal();
51
52     cout << "\nInorder traversal\n";
53     doubleTree.inOrderTraversal();
54
55     cout << "\nPostorder traversal\n";
56     doubleTree.postOrderTraversal();
57
58     cout << endl;
59     return 0;
60 } // 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
Inoreder 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
```

شکل 21-22 ایجاد و پیمایش درخت باینری.

خطوط 30-32 داده `private` از `TreeNode` را اعلان کرده‌اند- مقدار داده گره، و اشاره گرهای `leftPtr` (برای زیردرخت گره چپ) و `rightPtr` (برای زیردرخت گره راست). سازنده (خطوط 16-22)



مباردت به تنظیم مقدار **data** تدارک دیده شده از سوی آرگومان سازنده و تنظیم اشاره گرهای **leftPtr** و **rightPtr** با صفر کرده است. تابع عضو **getData** (خطوط 25-28) مقدار **data** را برگشت می‌دهد.

کلاس **Tree** (شکل 21-21) حاوی گره ریشه (**rootPtr**) در خط 22 است که به گره ریشه در درخت اشاره دارد. همچنین کلاس دارای تابع **insertNode** (خطوط 17-20) است که گره‌ای را در درخت جای می‌دهد، البته دارای توابع سراسری **preOrderTraversal**، **inOrderTraversal** و **postOrderTraversal** است که وظیفه پیمایش درخت را برعهده دارند. هر تابع پیمایش درخت، یک تابع یوتیلیتی بازگشتی جداگانه را برای انجام عملیات پیمایش بر روی ساختار داخلی درخت فراخوانی می‌کند. سازنده **Tree** مباردت به مقداردهی **rootPtr** با **null** (صفر) می‌کند تا نشان دهد درخت در ابتدای کار تهی است.

تابع **InsertNode** ابتدا تابع کمک **insertNodeHelper** (خطوط 47-68) را فراخوانی می‌کند تا بصورت بازگشتی گره‌ای را وارد درخت کند. در یک درخت جستجوی باینری گره‌ها فقط می‌توانند به عنوان گره‌های برگ وارد درخت شوند. اگر درخت تهی باشد، یک **TreeNode** جدید ایجاد، مقداردهی اولیه شده و در درخت درج می‌شود (خطوط 53-54).

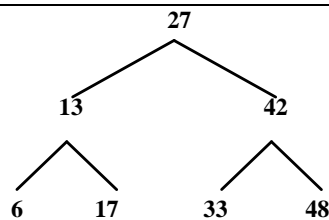
اگر درخت تهی نباشد، برنامه مقدار وارد شده را با مقدار **data** در گره ریشه مقایسه می‌کند. اگر مقدار وارده کمتر از (کوچکتر از) مقدار گره ریشه باشد (خط 57)، برنامه بصورت بازگشتی تابع **insertNodeHelper** (خط 58) را برای درج مقدار در زیر درخت چپ فراخوانی می‌کند. اگر مقدار وارده بزرگتر از مقدار گره ریشه باشد (خط 62)، برنامه بصورت بازگشتی تابع **insertNodeHelper** (خط 64) را برای درج مقدار در زیر درخت راست فراخوانی می‌کند. اگر مقدار وارد شده برابر با مقدار گره ریشه باشد، برنامه پیغام "dup" را چاپ (خط 65) کرده و بدون اینکه مقدار تکراری را وارد درخت کند، باز می‌گردد. دقت کنید که **insertNode** آدرس **rootPtr** را به **insertNodeHelper** ارسال می‌کند (خط 42) از اینرو است که می‌تواند مقدار ذخیره شده در **rootPtr** را اصلاح کند. برای دریافت اشاره گر به **rootPtr** که خود یک اشاره گر است، اولین آرگومان **insertNodeHelper** بعنوان یک اشاره گر به اشاره گر به یک **treeNode** اعلان شده است.

توابع پیمایش **inOrderTraversal** (خطوط 90-94)، **preOrderTraversal** (خطوط 71-75) و **postOrderTraversal** (خطوط 109-113) توابع کمکی **inOrderHelper** (خط 102)، **preOrderHelper** (خط 84) و **postOrderHelper** (خط 123) را به ترتیب فراخوانی می‌کنند تا درخت



## 1082 فصل بیست و یکم ساختمان‌های داده

پیمایش شده و مقادیر هر گره به نمایش در آید. تابعهای کمک کننده در کلاس **Tree** به برنامه‌نویس امکان می‌دهند تا شروع به پیمایش درخت نماید بدون اینکه ابتدا به گره ریشه اشاره کند. برای اینکه بهتر با مبحث پیمایش آشنا شوید از تصویر درخت جستجوی باینری در شکل 21-23 استفاده می‌کنیم.



شکل 21-23 | یک درخت جستجوی باینری.

### الگوریتم پیمایش Inorder

تابع **inOrderHelper** تعریف کننده مراحل پیمایش بفرم **inorder** است. این مراحل عبارتند از:

- 1- اگر آرگومان برابر **null** باشد، بلافاصله برگشت داده می‌شود.
  - 2- با فراخوانی **inOrderHelper** زیر درخت چپ را پیمایش می‌کند (خط 102).
  - 3- مقدار موجود در گره پردازش می‌شود (خط 103).
  - 4- پیمایش زیر درخت راست با فراخوانی **inOrderHelper** (خط 104).
- در پیمایش **inorder** تا زمانی که مقادیر گره‌های زیر درخت چپ پردازش نشده، مقدار موجود در گره پردازش نخواهد شد. پیمایش **inorder** بر روی درخت شکل 21-23 نتیجه زیر را خواهد داشت:

6 13 17 27 33 42 48

دقت کنید که در این نوع پیمایش بر روی درخت جستجوی باینری مقادیر گره‌ها بصورت صعودی بچاپ می‌رسند و اصولاً فرآیند ایجاد یک درخت جستجوی باینری بصورت مرتب شده است.

### الگوریتم پیمایش Preorder

تابع **preOrderHelper** تعریف کننده مراحل پیمایش بفرم **preorder** است. این مراحل عبارتند از:

- 1- اگر آرگومان **null** باشد، بلافاصله برگشت داده می‌شود.
- 2- پردازش مقدار موجود در گره (خط 83).





3- پیمایش زیر درخت چپ با فراخوانی `preOrderHelper` (خط 84).

4- پیمایش زیر درخت راست با فراخوانی `preOrderHelper` (خط 85).

در پیمایش `preorder` مقدار هر گره ملاقات شده پردازش می‌شود. پس از پردازش مقدار گره بدست آمده، پیمایش کار خود را با پردازش مقادیر در زیر درخت چپ ادامه داده و سپس مقادیر زیر درخت راست پردازش می‌شود. نتیجه پیمایش `preorder` بر روی درخت شکل 21-23 بصورت زیر خواهد بود:

27 13 6 17 42 33 48

#### الگوریتم پیمایش `Postorder`

تابع `postOrderHelper` تعریف کننده مراحل پیمایش بفرم `postorder` است. این مراحل عبارتند از:

1- اگر آرگومان `null` باشد، بلافاصله برگشت داده می‌شود.

2- پیمایش زیر درخت چپ با فراخوانی `postOrderHepler` (خط 122).

3- پیمایش زیر درخت راست با فراخوانی `postOrderHelper` (خط 123).

4- پردازش مقدار موجود در گره (خط 124).

نتیجه پیمایش `postroder` بر روی درخت شکل 21-23 بصورت زیر خواهد بود:

6 17 13 33 48 42 27

#### تمرینات

21-1 برنامه‌ای بنویسید که دو لیست پیوندی را به یکدیگر متصل کند.

21-2 برنامه‌ای بنویسید که دو لیست پیوندی مرتب شده با مقادیر صحیح را باهم ادغام کرده و یک لیست مرتب شده ایجاد نماید.

21-3 برنامه‌ای بنویسید که 25 عدد تصادفی از میان اعداد از 0 تا 100 را بصورت مرتب شده در یک لیست پیوندی قرار دهد.

21-4 برنامه‌ای بنویسید که عبارتی دریافت و با استفاده از پشته، آن عبارت را بفرم معکوس به نمایش در آورد.

21-5 برنامه‌ای بنویسید که با استفاده از پشته مشخص کند که آیا رشته‌ای پالندروم است یا خیر.

21-6 روالی بنام `depth` بنویسید که یک درخت باینری دریافت و عمق آنرا بدست آورد.



## 1084 فصل بیست و یکم ساختمان‌های داده

21-7 برنامه‌ای بنویسید که یک لیست پیوندی با 10 کاراکتر ایجاد کرده و سپس لیست دیگری که کپی از لیست اولیه است ایجاد کند اما با ترتیب معکوس.

21-8 همانطوری که می‌دانید پشته‌ها توسط کامپایلرها به منظور کمک در پردازش ارزیابی عبارات و ایجاد کد زبان ماشین بکار گرفته می‌شوند. بطور کلی ما انسان‌ها عبارات را بفرم  $3+4$  یا  $7/9$  می‌نویسیم که عملگر (+ یا /) در بین عملوندها قرار می‌گیرد، که به اینحالت نشانه‌گذاری میانوندی (infix notation) گفته می‌شود. کامپیوترها حالت نشانه‌گذاری پسوندی (postfix notation) را ترجیح می‌دهند، که در آن عملگر در سمت راست دو عملوند نوشته می‌شود (قرار می‌گیرد). در نشانه‌گذاری پسوندی عبارات  $3 + 4$  و  $7/9$  به ترتیب بفرم  $34+$  و  $79/$  نوشته می‌شوند.

برای ارزیابی یک عبارت پیچیده infix، بایستی ابتدا کامپیوتر کل عبارات را بحالت postfix تبدیل کرده و سپس آنرا ارزیابی کند. هر کدام از این الگوریتم‌ها مستلزم یک گذار از چپ به راست بر روی عبارات خواهند داشت.

هر الگوریتم از یک پشته برای انجام عملیات خود استفاده می‌کند و در هر الگوریتم از پشته به منظور متفاوتی استفاده می‌شود. در این تمرین برنامه تبدیل عبارت infix به postfix نوشته خواهد شد و در تمرین بعدی برنامه ارزیابی عبارات postfix نوشته می‌شود.

کلاسی بنام **InfixToPostfixConverter** بنویسید که یک عبارت عادی ریاضی (infix) که فقط متشکل از اعداد صحیح همانند  $8 / 4 - 5 * (6+2)$  است به عبارت postfix تبدیل کند. پس از تبدیل این عبارت، postfix بفرم زیر خواهد بود:

$62+5*84/-$

برنامه باید عبارت وارد شده را بدرون رشته **string Builder** خوانده و از کلاس **StackCompostion** برای ایجاد عبارت postfix، در رشته **stringBuilder** استفاده کند. الگوریتم ایجاد عبارت postfix بصورت زیر است:

- (a) پرانتز سمت چپ '(' در درون پشته قرار می‌گیرد.
- (b) الحاق یک پرانتز راست ')' به انتهای infix.
- (c) تا زمانیکه پشته خالی نشده، infix از چپ به راست خوانده شده و مراحل زیر انجام می‌شود:

- \* اگر کاراکتر جاری در infix یک رقم باشد، آنرا به postfix اضافه می‌کند.
- \* اگر کاراکتر جاری در infix یک پرانتز چپ باشد، آنرا بدرون پشته وارد می‌کند (push).

- اگر کاراکتر جاری در infix یک عملگر باشد:

\* خارج کردن عملگرها (اگر عملگری وجود داشته باشد) از بالای پشته (Pop) تا زمانیکه عملگری دارای تقدم برابر یا بالاتر از عملگر فعلی قرار داشته باشد. عملگرهای خارج شده از پشته به postfix الحاق می‌شوند.



## ساختمان‌های داده فصل بیست و یک 1085

\* وارد شدن کاراکتر جاری در infix بدرون پشته

- اگر کاراکتر فعلی در infix یک پرانتز سمت راست باشد:  
\* عملگرها از بالای پشته خارج شده (Pop) و به postfix الحاق می‌شود و اینکار تا زمانی صورت می‌گیرد که یک پرانتز چپ در بالای پشته مشاهده شود.  
\* پرانتز سمت چپ از پشته خارج می‌شود.  
همچنین عملگرهای ریاضی نیز می‌توانند در عبارات وجود داشته باشند، عملگرهای:

+ جمع

- تفریق

\* ضرب

/ تقسیم

^ توان

% باقیمانده

تعدادی از تابعهایی که ممکن است در این برنامه از آنها استفاده کنید عبارتند از:

- (a) تابع **ConvertToPostfix** برای تبدیل عبارت infix به postfix.  
(b) تابع **IsOperator** برای تعیین اینکه آیا کاراکتر یک عملگر است یا خیر.  
(c) تابع **Precedence** برای تعیین اینکه آیا تقدم عملگر **operator1** (از عبارت infix) کمتر، برابر یا بیشتر از تقدم عملگر **operator2** (از پشته) است یا خیر. اگر تقدم عملگر **operator1** کمتر از عملگر **operator2** باشد، تابع مقدار **True** و در غیر اینصورت مقدار **False** برگشت می‌دهد.  
21-9 کلاسی با نام **PostfixEvaluator** که عبارات postfix را ارزیابی می‌کند، بنویسید. فرض کنید عبارت postfix بفرم زیر باشد:

$$62 + 5 * 84 / -$$

برنامه بایستی عبارت postfix را که متشکل از ارقام و عملگرها می‌باشد بدرون رشته **StringBuilder** وارد کند. برنامه باید عبارت را از ابتدا تا انتها طی کرده و آنرا ارزیابی کند. الگوریتم به شرح زیر عمل می‌کند:  
(a) الحاق پرانتز سمت راست '()' به انتهای عبارت postfix. هنگامی که با پرانتز سمت راست مواجه شود، ادامه پردازش ضروری نیست.

(b) تا زمانی که با پرانتز سمت راست مواجه نشده است، عبارت از سمت چپ به راست خوانده می‌شود.

- اگر کاراکتر جاری یک رقم باشد:

\* دو عنصر از بالای پشته خارج شده (Pop) و در درون متغیرهای **x** و **y** جای داده می‌شوند.  
\* محاسبه مقدار، **x عملگر y**.  
\* وارد کردن نتیجه محاسبه، به درون پشته (Push).



## 1086 فصل بیست و یکم \_\_\_\_\_ ساختمان‌های داده

c) هنگامی که با کاراکتر سمت راست مواجه شود، مقدار بالایی پشته از آن خارج می‌شود (Pop) که نتیجه ارزیابی عبارت postfix است.

عملگرهای ریاضی زیر می‌توانند در عبارات وجود داشته باشند:

+ جمع

- تفریق

\* ضرب

/ تقسیم

^ توان

% باقیمانده

می‌توانید از تابع‌های زیر در برنامه استفاده کنید:

a) تابع **EvaluatePostfixExpression** برای ارزیابی عبارت postfix

b) تابع **Calculate** برای ارزیابی عبارت **Op2** عملگر **Op1**.