CHAPTER 6

LINKED LISTS

Linked lists are versatile data structures that feature fast insertion and deletion.

In this chapter we'll learn
- The advantages and disadvantages of linked lists
- How linked lists works
- How to create a linked list in C++

It is well-known that arrays have certain disadvantages as data storage structures. In an unordered array, searching is slow, whereas in an ordered array, insertion is slow. In both kinds of arrays deletion is slow.

Also, the size of an array can't be changed after it's created (although it can be changed in a vector).

The linked list solves some of these problems. Linked lists are probably the second most commonly used general-purpose storage structures after arrays.

The linked list is a versatile mechanism suitable for use in many kinds of general-purpose data-storage applications. It can also replace an array as the basis for other storage structures such as stacks and queues. In fact, you can use a linked list in many cases where you use an array. By doing so you can greatly improve insertion and deletion performance.

Understanding Links

In a linked list, each data item is embedded in a link. A ***link*** is an object of a class called something like Link. Because there are many similar links in a list, it makes sense to use a separate class for them, distinct from the linked list itself. Each link object contains a pointer (which we'll call pNext) to the next link in the list. A data member in the list itself contains a pointer to the first link. This is shown in the following figure.

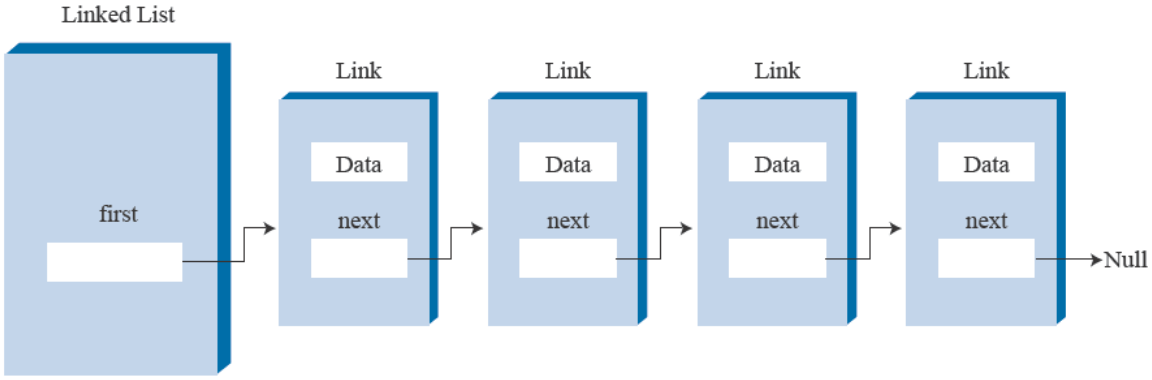

Figure: Links in a list.

Here's part of the definition of a class Link. It contains some data and a pointer to the next link:

```
class Link
{
   public:
   int iData; //data
   double dData; //data
   Link* pNext; //pointer to next link
};
```

This kind of class definition is sometimes called ***self-referential*** because it contains a data member -pNext in this case- which is a pointer to the same type as itself.

We show only two data items in the link: an int and a double. In a typical application there would be many more. A personnel record, for example, might have name, address, Social Security Number, title, salary, and many other fields. Often a pointer to an object that contains this data is used instead of the data items themselves:

```
class Link
{
   public:
   inventoryItem* pItem; //pointer to object holding data
   Link* pNext; //pointer to next link
};
```

Implementing a Simple Linked List

Our first sample program, linkedList1.cpp, demonstrates a simple linked list. The only operations allowed in this version of a list are

- Inserting an item at the beginning of the list
- Removing the item at the beginning of the list
- Iterating through the list to display its contents

These operations are fairly easy to carry out, so we'll start with them. (these operations are also all you need to use a linked list as the basis for a stack.)

Before we get to the complete linkedList.cpp program, we'll look at some important parts of the Link and LinkList classes.

The Link Class

You've already seen the data part of the Link class. Below you see the complete class definition:

```
class Link
{
    public:
    int iData; //data item
    double dData; //data item
    Link* pNext; //ptr to next link in list

    Link(int id, double dd) : //constructor
        iData(id), dData(dd), pNext(NULL){ }
```

```
    void displayLink()          //display ourself {22, 2.99}
    {
        cout << "{" << iData << ", " << dData << "} ";
    }

}; //end class Link
```

Analysis

In addition to the data, there's a constructor and a member function, displayLink(), that displays the link's data in the format {22, 2.99}.

The constructor initializes the data. There's no need to initialize the pNext data member because it's automatically set to NULL when it's created. The NULL value means it doesn't refer to anything, which is the situation until the link is connected to other links.

The LinkList Class

```
class LinkList
{
  private:
    Link* pFirst; //ptr to first link on list

  public:
    LinkList() : pFirst(NULL) //constructor
      { } //(no links on list yet)

    bool isEmpty() //true if list is empty
      { return pFirst==NULL; }

    ... //other methods go here

}; //end class LinkList
```

The LinkList class contains only one data item: ***a pointer*** to the first link on the list. This pointer is called pFirst. It's the only permanent information the list maintains about the location of any of the links. It finds the other links by following the chain of pointers from pFirst, using each link's pNext data member.

Analysis

The constructor for LinkList sets pFirst to NULL. When pFirst has the value NULL, we know there are no items on the list. If there were any items, pFirst would contain a pointer to the first one. As we'll see, the isEmpty() member function uses this fact to determine if the list is empty.

The insertFirst() Member Function

The insertFirst() member function of LinkList inserts a new link at the beginning of the list. This is the easiest place to insert a link because pFirst already points to the first link. To insert the new link, we need only set the pNext data member in the newly created link to point to the old first link, and then change pFirst so it points to the newly created link. This is shown in the following.

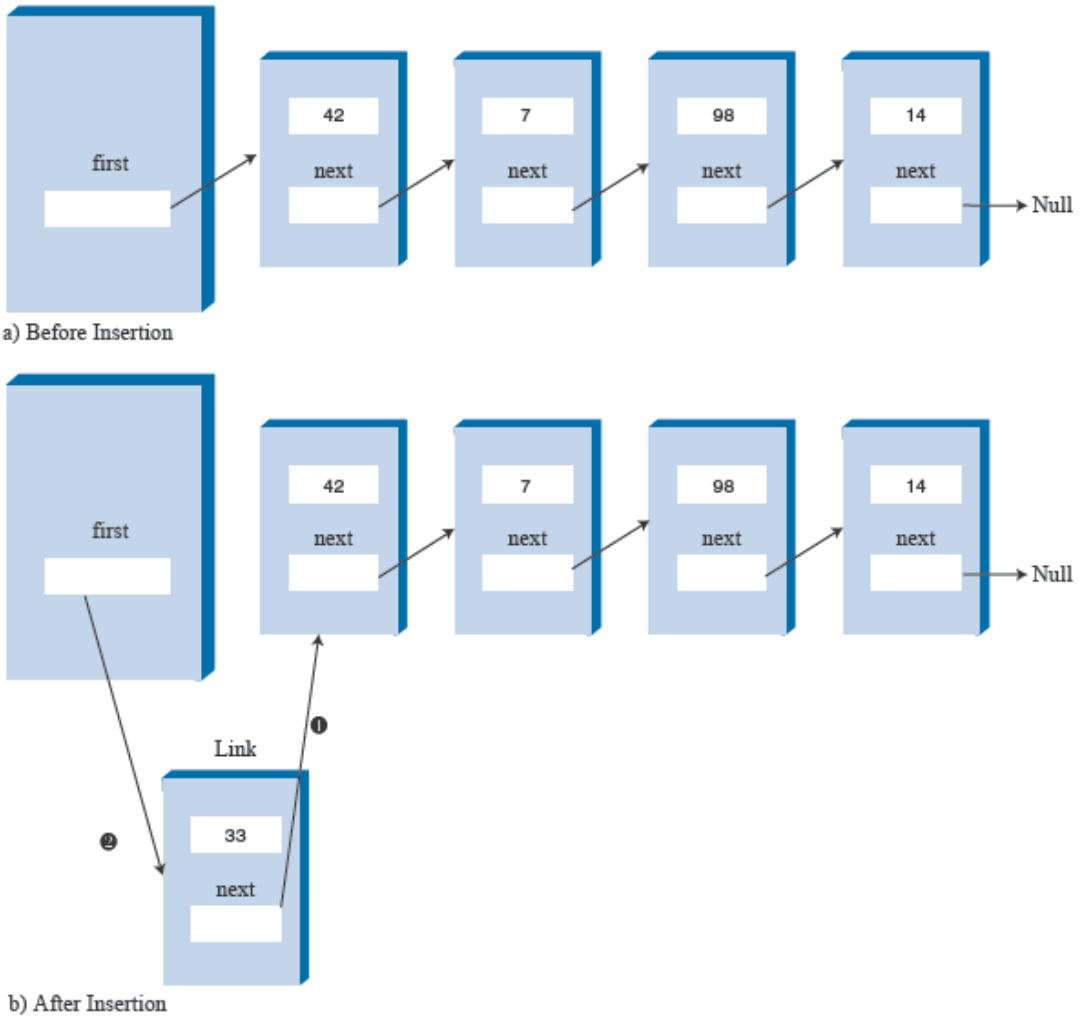

Figure: Inserting a new link.

In insertFirst() we begin by creating the new link using the data passed as arguments. Then we change the link pointers as we just noted.

```
//insert at start of list
void insertFirst(int id, double dd)
{ //make new link
    Link* pNewLink = new Link(id, dd);
    pNewLink->pNext = pFirst; //newLink-->old first
    pFirst = pNewLink; //first-->newLink
}
```

Analysis

The arrows --> in the comments in the last two statements mean that a link (or the pFirst data member) connects to the next (***downstream***) link. (In ***doubly linked lists*** there are ***upstream*** connections as well, symbolized by <-- arrows.) Compare these two statements with the above figure. Make sure you understand how the statements cause the links to be changed, as shown in the figure. This kind of pointer-manipulation is the heart of linked list algorithms.

Notice that we've created a new link with new. This implies we'll need to delete the link from memory if at some point we remove it from the list.

The removeFirst() Member Function

In C++, delete is an operator used to remove an object from memory. In listings we'll use the term "remove" to refer to a link being taken out of a list.

The removeFirst() member function is the reverse of insertFirst(). It disconnects the first link by rerouting pFirst to point to the second link. This second link is found by looking at the pNext data member in the first link.

```
void removeFirst() //delete first link
{ //(assumes list not empty)
    Link* pTemp = pFirst; //save first
    pFirst = pFirst->pNext; //unlink it: first-->old next
    delete pTemp; //delete old first
}
```

Analysis

The second statement is all you need to remove the first link from the list. The following figure shows how pFirst is rerouted to delete the object. We also need to delete the removed link from memory, so we save a pointer to it in pTemp, and then delete it in the last statement. This prevents a "memory leak" caused by an accumulation of links deleted from the list but still taking up space in memory. (We'll examine memory leaks again in the next program.)

Notice that the removeFirst() member function assumes the list is not empty. Before calling it, your program should verify this with the isEmpty() member function.

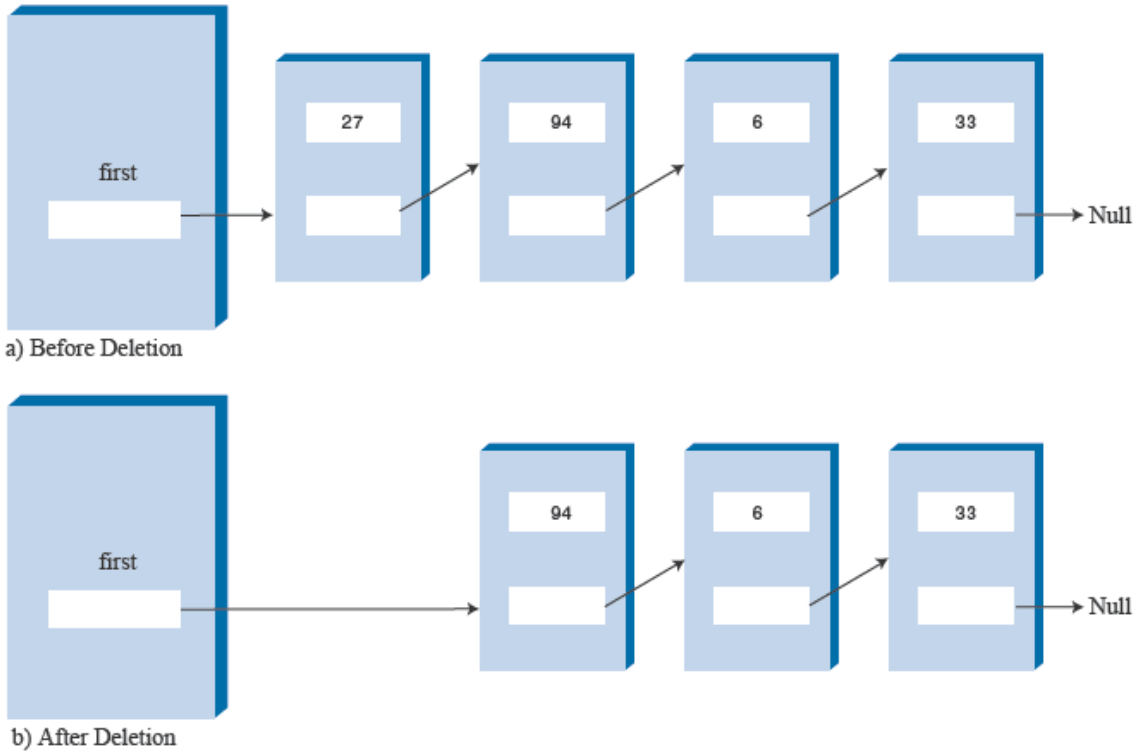

Figure: Deleting a link.

The displayList() Member Function

To display the list, you start at pFirst and follow the chain of pointers from link to link. A variable pCurrent points to each link in turn. It starts off pointing to pFirst, which holds a

pointer to the first link. The following statement changes pCurrent to point to the next link because that's what's in the pNext data member in each link:

```
pCurrent = pCurrent->pNext;
```

Here's the entire displayList() member function:

```
void displayList(){
    cout << "List (first-->last): ";
    Link* pCurrent = pFirst; //start at beginning of list

    while(pCurrent != NULL) //until end of list,
    {
        pCurrent->displayLink(); //print data
        pCurrent = pCurrent->pNext; //move to next link
    }
    cout << endl;
}
```

Analysis

The end of the list is indicated by the pNext data member in the last link pointing to NULL rather than another link. How did this data member get to be NULL? It started that way when the link was constructed and was never given any other value because it was always at the end of the list. The while loop uses this condition to terminate itself when it reaches the end of the list. Following figure shows how pCurrent steps along the list.

At each link, the displayList() member function calls the displayLink() member function to display the data in the link.

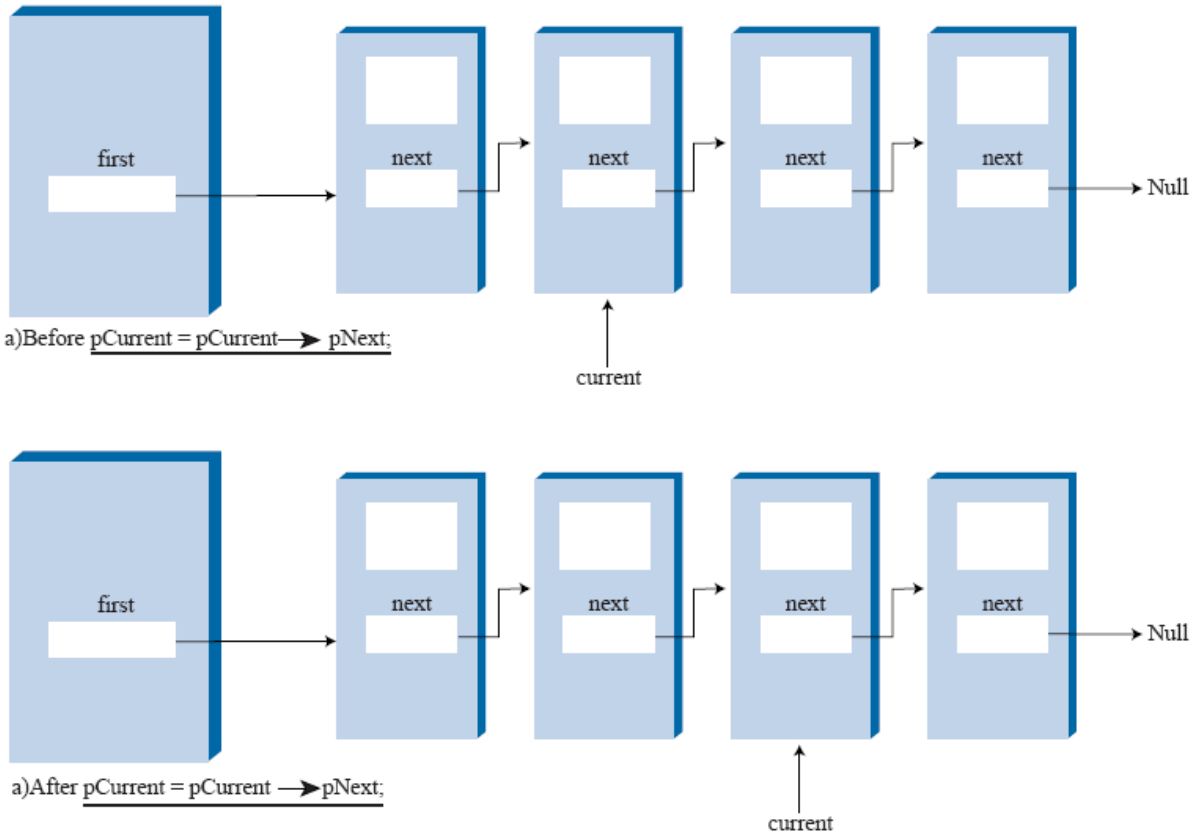

Figure: Stepping along the list.

The linkList1.cpp Program

The follwing is the the complete linkList.cpp program. You've already seen all the components except the main() routine.

```
1   //Program linkList1.cpp
2   //demonstrates linked list
3   #include <iostream>
4   using namespace std;
5   ////////////////////////////////////////////////////////////////
6   class Link
7   {
8       public:
9       int iData; //data item
10      double dData; //data item
11      Link* pNext; //ptr to next link in list
12      //-------------------------------------------------------------
13      Link(int id, double dd) : //constructor
14      iData(id), dData(dd), pNext(NULL)
15      { }
16      //-------------------------------------------------------------
17      void displayLink() //display ourself {22, 2.99}
18      {
19          cout << "{" << iData << ", " << dData << "} ";
20      }
21      //-------------------------------------------------------------
22  }; //end class Link
23  ////////////////////////////////////////////////////////////////
24  class LinkList
25  {
26      private:
27          Link* pFirst; //ptr to first link on list
28      public:
29          //-------------------------------------------------------------
30          LinkList() : pFirst(NULL) //constructor
31          { } //(no links on list yet)
32          //-------------------------------------------------------------
33          bool isEmpty() //true if list is empty
34          {
35              return pFirst==NULL;
36          }
37          //-------------------------------------------------------------
38          //insert at start of list
39          void insertFirst(int id, double dd)
40          { //make new link
41              Link* pNewLink = new Link(id, dd);
42              pNewLink->pNext = pFirst; //newLink-->old first
43              pFirst = pNewLink; //first-->newLink
44          }
45
```

```
46          //--------------------------------------------------------------
47          Link* getFirst() //return first link
48          {
49              return pFirst;
50          }
51          //--------------------------------------------------------------
52          void removeFirst() //delete first link
53          { //(assumes list not empty)
54              Link* pTemp = pFirst; //save first
55              pFirst = pFirst->pNext; //unlink it: first-->old next
56              delete pTemp; //delete old first
57          }
58          //--------------------------------------------------------------
59          void displayList()
60          {
61              cout << "List (first-->last): ";
62              Link* pCurrent = pFirst; //start at beginning of list
63              while(pCurrent != NULL) //until end of list,
64              {
65                  pCurrent->displayLink(); //print data
66                  pCurrent = pCurrent->pNext; //move to next link
67              }
68              cout << endl;
69          }
70      }; //end class LinkList
71      ////////////////////////////////////////////////////////////////
72      int main()
73      {
74          LinkList theList; //make new list
75          theList.insertFirst(22, 2.99); //insert four items
76          theList.insertFirst(44, 4.99);
77          theList.insertFirst(66, 6.99);
78          theList.insertFirst(88, 8.99);
79          theList.displayList(); //display list
80          while( !theList.isEmpty() ) //until it's empty,
81          {
82              Link* pTemp = theList.getFirst(); //get first link
83              //display its key
84              cout << "Removing link with key " << pTemp->iData << endl;
85              theList.removeFirst(); //remove it
86          }
87          theList.displayList(); //display empty list
88          getchar();
89          return 0;
90      } //end main( )
```

Here's the output from linkList1.cpp:

```
List (first-->last): {88, 8.99} {66, 6.99} {44, 4.99} {22, 2.99}
Removing link with key 88
Removing link with key 66
Removing link with key 44
Removing link with key 22
List (first-->last):
```

Analysis

In main() we create a new list, insert four new links into it with insertFirst(), and display it. Then, in the while loop, we repeatedly display the first item with getFirst() and remove it with removeFirst() until the list is empty. The empty list is then displayed.

We've examined a barebones linked list program. Now let's look at a program with some additional features.

Finding and Removing Specified Links

Our next sample program adds methods to search a linked list for a data item with a specified key value, and to remove an item with a specified key value. The complete **linkList2.cpp** program is shown below:

```
1   //Program linkList2.cpp
2   //demonstrates linked list
3   #include <iostream>
4   using namespace std;
5   ////////////////////////////////////////////////////////////////
6   class Link
7   {
8       public:
9           int iData; //data item (key)
10          double dData; //data item
11          Link* pNext; //next link in list
12          //-------------------------------------------------------------
13          Link(int id, double dd) : //constructor
14          iData(id), dData(dd), pNext(NULL)
15          { }
16          //-------------------------------------------------------------
17          void displayLink() //display ourself: {22, 2.99}
18          {
19              cout << "{" << iData << ", " << dData << "} ";
20          }
21  }; //end class Link
22  ////////////////////////////////////////////////////////////////
23  class LinkList
24  {
25      private:
26          Link* pFirst; //ptr to first link on list
27      public:
28          //-------------------------------------------------------------
29          LinkList() : pFirst(NULL) //constructor
30          { } //(no links on list yet)
31          //-------------------------------------------------------------
```

```
32          ~LinkList() //destructor (deletes links)
33          {
34              Link* pCurrent = pFirst; //start at beginning of list
35              while(pCurrent != NULL) //until end of list,
36              {
37                  Link* pOldCur = pCurrent; //save current link
38                  pCurrent = pCurrent->pNext; //move to next link
39                  delete pOldCur; //delete old current
40              }
41          }
42          //--------------------------------------------------------------
43          void insertFirst(int id, double dd)
44          { //make new link
45              Link* pNewLink = new Link(id, dd);
46              pNewLink->pNext = pFirst; //it points to old first link
47              pFirst = pNewLink; //now first points to this
48          }
49          //--------------------------------------------------------------
50          Link* find(int key) //find link with given key
51          { //(assumes non-empty list)
52              Link* pCurrent = pFirst; //start at 'first'
53              while(pCurrent->iData != key) //while no match,
54              {
55                  if(pCurrent->pNext == NULL) //if end of list,
56                  return NULL; //didn't find it
57                  else //not end of list,
58                  pCurrent = pCurrent->pNext; //go to next link
59              }
60              return pCurrent; //found it
61          }
62          //--------------------------------------------------------------
63          bool remove(int key) //remove link with given key
64          { //(assumes non-empty list)
65              Link* pCurrent = pFirst; //search for link
66              Link* pPrevious = pFirst;
67              while(pCurrent->iData != key)
68              {
69                  if(pCurrent->pNext == NULL)
70                      return false; //didn't find it
71                  else
72                  {
73                      pPrevious = pCurrent; //go to next link
74                      pCurrent = pCurrent->pNext;
75                  }
76              } //found it
```

```
77              if(pCurrent == pFirst) //if first link,
78                  pFirst = pFirst->pNext; //change first
79              else //otherwise,
80                  pPrevious->pNext = pCurrent->pNext; //bypass it
81
82              delete pCurrent; //delete link
83              return true; //successful removal
84          }
85          //-------------------------------------------------------------
86          void displayList() //display the list
87          {
88               cout << "List (first-->last): ";
89               Link* pCurrent = pFirst; //start at beginning of list
90               while(pCurrent != NULL) //until end of list,
91               {
92                    pCurrent->displayLink(); //print data
93                    pCurrent = pCurrent->pNext; //move to next link
94               }
95                    cout << endl;
96           }
97    //-------------------------------------------------------------
98    }; //end class LinkList
99
100   ////////////////////////////////////////////////////////////////
101   int main()
102   {
103       LinkList theList; //make list
104       theList.insertFirst(22, 2.99); //insert 4 items
105       theList.insertFirst(44, 4.99);
106       theList.insertFirst(66, 6.99);
107       theList.insertFirst(88, 8.99);
108       theList.displayList(); //display list
109       int findKey = 44; //find item
110       Link* pFind = theList.find(findKey);
111
112       if( pFind != NULL)
113           cout << "Found link with key " << pFind->iData << endl;
114       else
115           cout << "Can't find link" << endl;
116
117       int remKey = 66; //remove item
118       bool remOK = theList.remove(remKey);
119
120       if( remOK )
121           cout << "Removed link with key " << remKey << endl;
122       else
123           cout << "Can't remove link" << endl;
124
125       theList.displayList(); //display list
126
```

```
127        getchar();
128           return 0;
129        } //end main()
```

Analysis

The main() routine makes a list, inserts four items, and displays the resulting list. It then searches for the item with key 44, removes the item with key 66, and displays the list again.

The find() Member Function

The find() member function works much like the displayList() member function seen in the linkList1.cpp program. The pointer pCurrent initially points to pFirst, and then steps its way along the links by setting itself repeatedly to pCurrent->pNext. At each link, find() checks whether that link's key is the one it's looking for. If it is, it returns with a pointer to that link. If it reaches the end of the list without finding the desired link,
it returns NULL.

The remove() Member Function

The remove() member function is similar to find() in the way it searches for the link to be removed. However, it needs to maintain a pointer not only to the current link (pCurrent), but to the link preceding the current link (pPrevious). This is because if it removes the current link, it must connect the preceding link to the following link, as shown in the following figure. The only way to remember the location of the preceding link is to maintain a pointer to it.

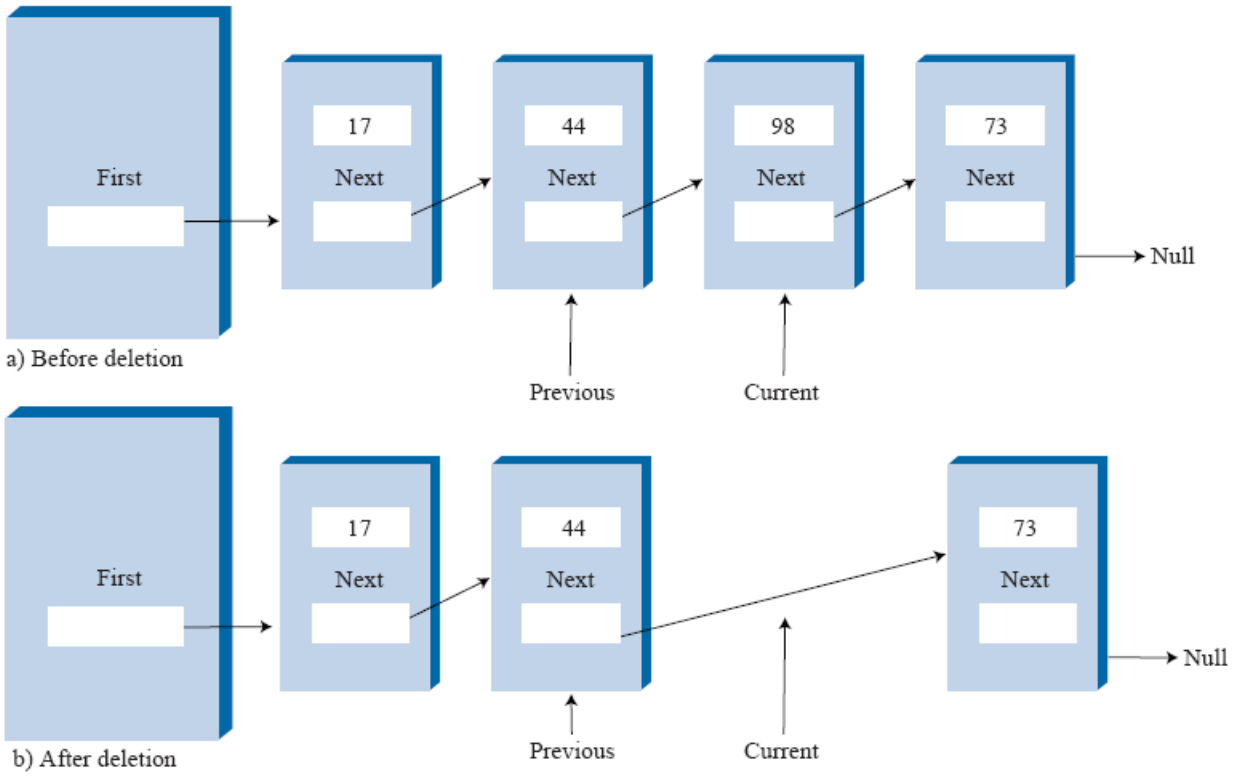

Figure: Removing a specified link.

At each cycle through the while loop, just before pCurrent is set to pCurrent->pNext, pPrevious is set to pCurrent. This keeps it pointing at the link preceding pCurrent. To remove the current link after it's found, the pNext data member of the previous link is set to the next link. A special case arises if the current link is the first link because the first link is pointed to by the LinkList's pFirst data member and not by another link. In this case the link

is removed by changing pFirst to point to pFirst->pNext, as we saw in the linkList2.cpp program with the removeFirst() member function. Here's the code that covers these two possibilities:

```
if(pCurrent == pFirst) //if first link,
      pFirst = pFirst->pNext; // change first
else //otherwise,
      pPrevious->pNext = pCurrent->pNext; // bypass it
```

Avoiding Memory Leaks

To create a linked list, the LinkList program creates a LinkList object. During the operation of the linked list, the insertFirst() member function of this object creates Link objects using the C++ new operator. When the LinkList object is destroyed (in this program because it goes out of scope when main() terminates), the Link objects that have been created will remain in memory unless we take steps to delete them. Accordingly, in the destructor for the LinkList class, we install code to step through the list and apply the delete operator to each link. For the same reason, the remove() member function must delete a link after it has been removed from the list.

Note:

We've seen methods to insert and remove items at the start of a list, and to find a specified item and remove a specified item. You can imagine other useful list methods. For example, an insertAfter() member function could find a link with a specified key value and insert a new link following it.

The Efficiency of Linked Lists

Insertion and deletion at the beginning of a linked list are very fast. They involve changing only one or two pointers, which takes O(1) time.

Finding or deleting a specified item requires searching through, on the average, half the items in the list. This requires O(N) comparisons. An array is also O(N) for these operations, but the linked list is nevertheless faster because nothing needs to be moved when an item is inserted or removed. The increased efficiency can be significant, especially if a copy takes much longer than a comparison.

Of course, another important advantage of linked lists over arrays is that the linked list uses exactly as much memory as it needs, and can expand to fill all available memory. The size of an array is fixed when it's created; this usually leads to inefficiency because the array is too large, or to running out of room because the array is too small. Vectors, which are expandable arrays, might solve this problem to some extent, but they usually expand in fixed-sized increments (such as doubling the size of the array whenever it's about to overflow). This use of memory is still not as efficient as a linked list.

Summary

In this chapter, you've learned the following:

- A linked list consists of one linkedList object and a number of link objects.
- The linkedList object contains a pointer, often called pFirst, to the first link in the list.
- Each link object contains data and a pointer, often called pNext, to the next link in the list.
- A pNext value of NULL signals that a link is the last one on the list.

- Inserting an item at the beginning of a linked list involves setting the new link's pNext data member to point to the old first link, and changing pFirst to point to the new link.
- Deleting a link at the beginning of a list involves setting pFirst to point to pFirst->pNext.
- When a link is removed from a list it must also be deleted from memory to avoid a memory leak.
- To traverse a linked list, you start at pFirst; then go from link to link, using each link's pNext data member to find the next link.
- A link with a specified key value can be found by traversing the list. After it is found, an item can be displayed, removed, or operated on in other ways.
- A new link can be inserted before or after a link with a specified key value, following a traversal to find this link.

Q&A

Q How do I know when to use a linked list instead of an array?

A You should consider a linked list when there will be lots of insertions of new data items or deletions of existing items.

Q When shouldn't I use a linked list?

A Don't use a linked list if you need frequent access to data items with a specified key, or to arbitrary items in the list (such as the access provided by array indices).

Quiz

1. What one piece of data must be included in a link class?
2. What one piece of data must be included in a linked list class?
3. Deleting a link from a linked list involves only one change in the list's structure. What is it?
4. How do you get from the current link to the next link?
5. What task must be carried out by both the find(int key) and remove(int key) member functions?
6. How many objects of the linked list class are normally used to implement a linked list?
7. What task should be carried out by the destructor of a linked list class in a C++ program?

Answers

1. A pointer to the next link.
2. A pointer to the first link.
3. Changing the pointer in the preceding link so it points to the link that follows the one being deleted.
4. Go to the link pointed to by pNext in the current link.
5. They must both search for a given key.
6. One.
7. It should delete any links currently in the list. Failure to do this might cause memory to fill up with unused links.