21

21 Collections of data

Standard "collection classes"

As illustrated in the "Air Traffic Controller" example in Chapter 20, arrays can be used to store collections of data. But arrays do have limitations (e.g. you have to specify a maximum size), and they don't provide any useful support function that might help manage a data collection. Over the years, many different structures have been devised for holding collections of data items. These different structures are adapted to meet specific needs; each has associated functionality to help manage and use the collection.

Queue

For example, a simple Queue provides a structure where new data items can be added "at the rear"; the data item "at the front" of the Queue can be removed when it is time for it to be dealt with. You probably studied queues already today while you stood in line to pay at the college cafeteria.

Priority Queue

A "Priority Queue" has different rules. Queued items have associated priorities. When an item is added to the collection, its place is determined by its priority. A real world example is the queue in the casualty department of a busy hospital. Car crash and gun shot patients are priority 1, they go straight to the front of the queue, patients complaining of "heavy" pains in their chests - mild heart attacks – are priority 2 etc.

Dynamic arrays and lists

"Dynamic Arrays" and "Lists" provide different solutions to the problem of keeping a collection of items where there are no particular access rules like those of Queue. Items can be added at the front or rear of a list, and items can be removed from the middle of lists. Often, lists get searched to find whether they contain a specific item; the search is "linear" (start at the front and keep checking items until either you find the one that is wanted or you get to the end of the list).

Binary tree

A "Binary Tree" is more elaborate. It provides a means for storing a collection of data items that each include "key" values. A binary tree provides a fast search system so you can easily find a specific data element if you know its "key".

Data storage

Sections 21.1 and 21.2 illustrate a simple Queue and a "Priority Queue". These use fixed size arrays to store (pointers to) data elements. You have to be able to plan for a maximum number of queued items. With the "Dynamic Array", 21.3, things get a bit more complex. A "Dynamic Array" supports "add item", "search for item", and "remove item" operations. Like a simple Queue, the Dynamic Array uses an array of pointers to the items for which it is responsible. If a Dynamic Array gets an "add item"

request when its array is full, it arranges to get additional memory. So it starts with an initial array of some default size, but this array can grow (and later shrink) as needed. With Lists, 21.4, you abandon arrays altogether. Instead, a collection is made up using auxiliary data structures ("list cells" or "list nodes"). These list cells hold the pointers to the data items and they can be linked together to form a representation of the list. Binary trees, 21.5, also utilize auxiliary structures ("tree nodes") to build up a representation of an overall data structure.

Data items?

What data items get stored in these lists, queues and trees?

Ideally, it should be any data item that a programmer wants to store in them. After all, these Queue, List, and Binary Tree structures represent general concepts (in the preferred jargon, they are "abstract data types"). You want to be able to use code implementing these types. You don't want to have to take outline code and re-edit it for each specific application.

"Generic code"

There are at least three ways in C++ in which you can get some degree of generality in your code (code that is independent of the type of data manipulated is "generic"). The examples in this chapter use the crudest of these three approaches. Later examples will illustrate more sophisticated techniques such as the use of (multiple) inheritance and the use of templates. The approach used here is an older, less sophisticated (and slightly more error prone) technique but it is also the simplest to understand.

The data items that are to be in the Queues, Lists, and Dynamic Arrays are going to be structures that have been created in the heap. They will be accessed via pointers. We can simply use `void*` pointers (pointers to data items of unspecified type). Code written to use `void*` pointers is completely independent of the data items accessed via those pointers.

The Priority Queue and Binary Tree both depend on the data items having associated "key values" (the "keys" of the Binary Tree, the priorities of the Priority Queue). These key values will be long integers. The interface for these classes require the user to provide the integer key and a pointer to the associated data object.

21.1 CLASS QUEUE

When on earth would you want to model a "queue" in a computer program?

Well, it is not very often! Most of the places where queues turn up tend to be in the underlying operating system rather than in typical applications programs. Operating systems are full of queues of many varied kinds.

Queues in operating systems

If the operating system is for a shared computer (rather than a personal computer) it will have queues for jobs waiting to run. The computer might have one CPU and one hundred terminals; at any particular time, there may be "requests" for CPU cycles from many of the terminals. The operating system maintains a queue of requests and gets the CPU to process each in turn.

Elsewhere in the operating system, there will be queues of requests for data transfers to and from disk. There will also be queues of characters waiting to be transmitted

down modem lines to remote terminals or other machines, and queues of characters already typed but not yet processed.

Outside of operating systems, queues most frequently turn up in simulations. The real world has an awful lot of queues, and real world objects spend a lot of time standing in queues. Naturally, such queues appear in computer simulations.

A queue owns …

What does a queue own? A queue owns "something" that lets it keep pointers to the queued items; these are kept in "first come, first served" order. (The "something" can take different forms.)

A queue does …

A queue can respond to the following requests:

- First
  Remove the front element from the queue and return it.
- Add (preferred name Append)
  Add another element at the rear of the queue.
- Length
  Report how many items are queued.
- Full
  If the queue only has a finite amount of storage, it may get to be full, a further Add operation would then cause some error. A queue should have a "Full" member function that returns true if the queue is full.
- Empty
  Returns true if there are no data elements queued; an error would occur if a First operation was performed on an empty queue.

Representation: "circular buffer"

Several different data structures can be used to represent a queue in a program. This example uses a structure known as a "circular buffer". This version of a queue is most commonly used for tasks such as queuing characters for output devices.

The principal of a circular buffer is illustrated in Figure 21.1. A fixed sized array, the "buffer", is used to store the data values. In this implementation, the array stores `void*` pointers to other data structures that have been allocated in the heap; but if a circular buffer were being used to queue characters going to a modem, the array would hold the actual characters. (In Figure 21.1, the data items are shown as characters). As well as the array, there are two integers used to index into the array; these will be referred to as the "get-index" and the "put-index". Finally, there is a separate integer that maintains a count of the number of items currently stored.

An Append ("put") operation puts a data item at "the end of the queue". Actually the Append operation puts it in the array at the index position defined by `fp` (the put index) and then increments the `fp` index and the counter (`fcount`). So, successive characters fill in array elements 0, 1, 2, …. In the example shown in Figure 21.1, the array `fdata` has six elements. The `fp` index is incremented modulo the length of the array. So, the next element after `fdata[5]` is `fdata[0]` (this makes the index "circle round" and hence the name "circular buffer").

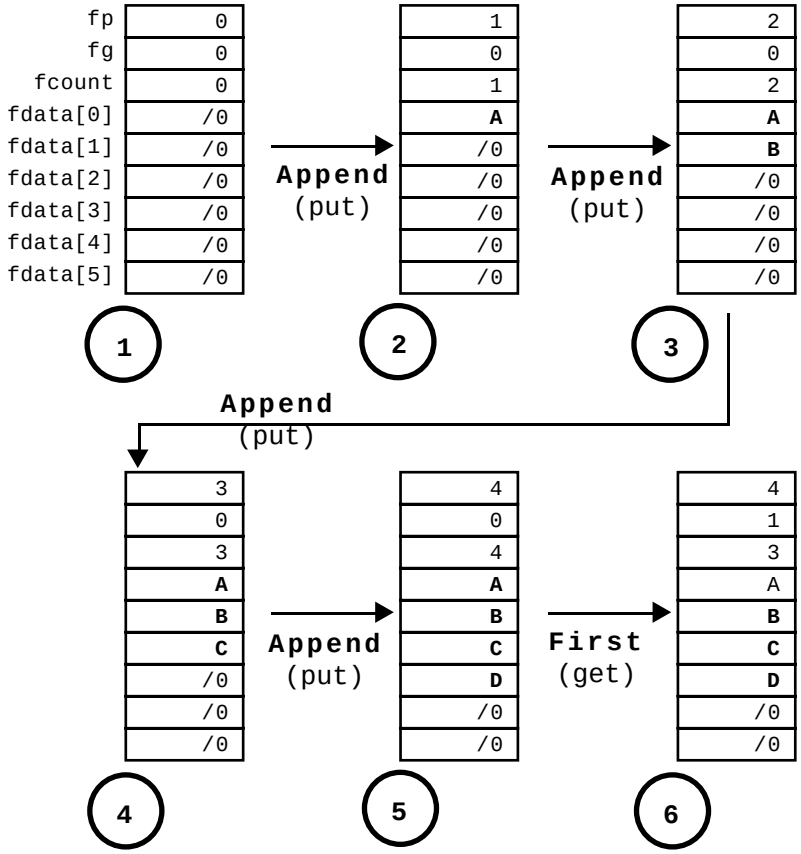

Figure 21.1 Queue represented as a "circular buffer" and illustrative Append (put) and First (get) actions.

The First (get) operation that removes the item from "the front of the queue" works in a similar fashion. It uses the "get index" (`fg`). "First" takes the data item from `fdata[fg]`, updates the value of `fg` (modulo the array length so it too "circles around"), and reduces the count of items stored. (The data value can be cleared from the array, but this is not essential. The data value will be overwritten by some subsequent Append operation.)

Error conditions

Obviously there are a couple of problem areas. A program might attempt a First (get) operation on an empty queue, or might attempt to add more data to a queue that is already full. Strictly a program using the queue should check (using the "Empty" and "Full" operations) before performing a First or Append operation. But the code implementing the queue still has to deal with error conditions.

Exceptions

When you are programming professionally, you will use the "Exception" mechanism (described in Chapter 26) to deal with such problems. The queue code can "throw an

exception"; this action passes information back to the caller identifying the kind of error that has occurred. But simpler mechanisms can be used for these introductory examples; for example, the program can be terminated with an error message if an "Append" operation is performed on a full queue.

Class declaration

The queue will be represented by a class. The declaration for this class (in a header file Q.h) specifies how this queue can be used:

```
#ifndef __MYQ__
#define __MYQ__

#define QSIZE 6

class Queue {
public:
    Queue();

    void   Append(void* newitem);
    void   *First(void);

    int    Length(void) const;
    int    Full(void) const;
    int    Empty(void) const;

private:
    void   *fdata[QSIZE];
    int    fp;
    int    fg;
    int    fcount;
};

inline int Queue::Length(void) const { return fcount; }
inline int Queue::Full(void) const { return fcount == QSIZE; }
inline int Queue::Empty(void) const { return fcount == 0; }
#endif
```

public

The public interface specifies a constructor (initializor), the two main operations (`Append()` and `First()`) and the three functions that merely ask questions about the state of the queue.

private

The private part of the class declaration defines the data structures used to represent a queue. A program can have any number of `Queue` objects; each has its own array, count, and array indices.

inline functions

The three member functions that ask about the queue status are all simple, and are good candidates for being inline functions. Their definitions belong in the header file with the rest of the class declaration. These are `const` functions; they don't change the state of the queue.

The other member functions would be defined in a separate Q.cp file:

```
#include <iostream.h>
```

```
#include <stdlib.h>
#include "Q.h"
```

Handling errors

```
void QueueStuffed()
{
    cout << "Queue structure corrupted" << endl;
    cout << "Read instructions next time" << endl;
    exit(1);
}
```

The `QueueStuffed()` function deals with cases where a program tries to put too many data items in the queue or tries to take things from an empty queue. (Error messages like these should really be sent to `cerr` rather than `cout`; but on many IDEs, `cout` and `cerr` are the same thing.)

Constructor

```
Queue::Queue()
{
    fp = fg = fcount = 0;
}
```

The constructor has to zero the count and the pointers. There is no need to clear the array; any random bits there will get overwritten by "append" operations.

Append

```
void Queue::Append(void* newitem)
{
    if(Full())
           QueueStuffed();
    fdata[fp] = newitem;
    fp++;
    if(fp == QSIZE)
           fp = 0;
    fcount++;
    return;
}
```

The `Append()` function first checks that the operation is legal, using the `QueueStuffed()` function to terminate execution if the queue is full. If there is an empty slot in the array, it gets filled in with the address of the data item that is being queued. Then the `fcount` counter is incremented and the `fp` index is incremented modulo the length of the array. The code incrementing `fp` could have been written:

```
fp++;
fp = fp % QSIZE;
```

The code with the `if()` is slightly more efficient because it avoids the divide operation needed for the modulo (`%`) operator (divides are relatively slow instructions).

```
void *Queue::First(void)
```

```
{
    if(Empty())
           QueueStuffed();
    void* temp = fdata[fg];
           fg++;
    if(fg == QSIZE)
           fg = 0;
    fcount--;
    return temp;
}
```

The First() (get) operation is very similar.

Test program

The test program is part of the package!

Naturally, a small test program must be provided along with a class. As explained in the context of class `Bitmap`, the test program is part of the package. You develop a class for other programmers to use. They may need to make changes or extensions. They need to be able to retest the code. You, as a class implementor, are responsible for providing this testing code.

These test programs need not be anything substantial. You just have to use all the member functions of the class, and have some simple inputs and outputs that make it easy to check what is going on. Usually, these test programs are interactive. The tester is given a repertoire of commands; each command invokes one of the member functions of the class under test. Here we need "add", "get", and "length" commands; these commands result in calls to the corresponding member functions. The implementation of "add" and "get" can exercise the `Full()` and `Empty()` functions.

We need data that can be put into the queue, and then later get removed from the queue so that they can be checked. Often, it is necessary to have the tester enter data; but here we can make the test program generate the data objects automatically.

The actual test program creates "Job" objects (the program defines a tiny class Job) and adds these to, then removes these from the queue.

Include the standard header

```
#include <stdlib.h>
#include <iostream.h>
#include "Q.h"
```

Class Job: just for testing the queue

```
class Job {
public:
    Job(int num, char sym);
    void PrintOn(ostream& out);
private:
    int fN;
    char fC;
};
```

```
Job::Job(int num, char sym)
{
    fN = num; fC = sym;
}

void Job::PrintOn(ostream& out)
{
    out << "Job #" << fN << ", activity " << fC << endl;
}
```

The test program

```
int main()
{
    int n = 0;
    Job *j = NULL;
    Job *current = NULL;
```

Make a Queue object

```
    Queue theQ;
```

Loop until a quit command entered

```
    for(int done = 0; !done ;) {
           char command;
           cout << ">";
           cin >> command;
           switch(command) {
case 'q':          done = 1;
                   break;
```

Test Length()

```
case 'l' :          cout << "Queue length now " <<
                           theQ.Length() << endl;
                   break;
```

Test Full() and Append()

```
case 'a' :
                   j = new Job(++n, (rand() % 25) + 'A');
                   cout << "Made new job "; j->PrintOn(cout);
                   if(theQ.Full()) {
                          cout << "But couldn't add it to queue"
                                 " so got rid of it";
                          cout << endl;
                          delete j;
                          }
                   else {
                          theQ.Append(j);
                          cout << "Queued" << endl;
                          }
                   j = NULL;
                   break;
```

Test Empty() and First()

```
case 'g':
                   if(theQ.Empty())
                          cout << "Silly, the queue is empty"
                                 << endl;
                   else {
                          if(current != NULL)
                                 delete current;
                          current = (Job*) theQ.First();
                          cout << "Got ";
```

```
                                current->PrintOn(cout);
                        }
                    break;
case 'c':
                    if(current != NULL) {
                           cout << "Current job is ";
                           current->PrintOn(cout);
                           }
                    else cout << "No current job" << endl;
                    break;
case '?':
                    cout << "Commands are:" << endl;
                    cout << "\tq Quit\n\ta Add to queue\t" << endl;
                    cout << "\tc show Current job\n"
                           \tg Get job at front of queue" << endl;
                    cout << "\tl Length of queue\n" << endl;
                    break;
default:
                    ;
                    }
            }
    return EXIT_SUCCESS;
}
```

A quick test produced the following:

Recording of test run

```
>a
Made new job Job #1, activity T
Queued
>a
Made new job Job #2, activity N
Queued
>c
No current job
>g
Got Job #1, activity T
>l
Queue length now 1
>a
Made new job Job #3, activity G
Queued
```

21.2 CLASS PRIORITYQUEUE

Priority queues are almost as rare as simple queues. They turn up in similar applications – simulations, low-level operating system's code etc. Fortuitously, they have some secondary uses. The priority queue structure can be used as the basis of a sorting mechanism that is quite efficient; it is a reasonable alternative to the Quicksort

function discussed in Chapter 13. Priority queues also appear in the implementation of some "graph" algorithms such as one that finds the shortest path between two points in a network (such graph algorithms are outside the scope of this text, you may meet them in more advanced courses on data structures and algorithms).

As suggested in the introduction to this chapter, a priority queue would almost certainly get used in a simulation of a hospital's casualty department. Such simulations are often done. They allow administrators to try "What if …?" experiments; e.g. "What if we changed the staffing so there is only one doctor from 8am to 2pm, and two doctors from 2pm till 9pm?". A simulation program would be set up to represent this situation. Then a component of the program would generate "incoming patients" with different problems and priorities. The simulation would model their movements through casualty from admission through queues, until treatment. These "incoming patients" could be based on data in the hospital's actual records. The simulation would show how long the queues grew and so help determine whether a particular administrative policy is appropriate. (The example in Chapter 27 is a vaguely similar simulation, though not one that needs a priority queue.)

The objects being queued are again going to be "jobs", but they differ slightly from the last example. These jobs are defined by a structure:

```
struct Job {
    long   prio;
    char   name[30];
};
```

(this simple struct is just to illustrate the working of the program, the real thing would be much larger with many more data fields). The `prio` data member represents the priority; the code here will use small numbers to indicate higher priority.

A over simple priority queue!

You can implement a very simple form of priority queue. For example:

```
class TrivialPQ {
public:
    TrivialPQ();
    Job    First();
    void   Insert(const Job& j);
private:
    Job    fJobs[kMAXSIZE];
    int    fcount;
};
```

A `TrivialPQ` uses an array to store the queued jobs. The entries in this array are kept ordered so that the highest priority job is in element 0 of the array.

If the entries are to be kept ordered, the `Insert()` function has to find the right place for a new entry and move all less urgent jobs out the way:

```
void TrivialPQ::Insert(const Job& j)
{
```

```
    int    pos = 0;
    while((pos < fcount) && (fJobs[pos].prio < j.prio)) pos++;
    // j should go at pos
    // less urgent jobs move up to make room
    for(int i = fcount; i> pos; i--)
           fJobs[i] = fJobs[i-1];
    fJobs[pos] = j;
    fcount++;
}
```

Similarly, the `First()` function can shuffle all the lower priority items up one slot after the top priority item has been removed:

```
Job TrivialPQ::First()
{
    Job j = fJobs[0];
    for(int i = 1; i < fcount; i++)
           fJobs[i-1] = fJobs[i];
    fcount--;
    return j;
}
```

Cost is O(N)

The class `TrivialPQ` does define a workable implementation of a priority queue. The trouble is the code is rather inefficient. The cost is O(N) where N is the number of jobs queued (i.e. the cost of operations like `First()` and `Insert()` is directly proportional to N). This is obvious by inspection of `First()`; it has to move the N-1 remaining items down one slot when the front element is removed. On average, a new Job being inserted will go somewhere in the middle. This means the code does about N/2 comparisons while finding the place, then N/2 move operations to move the others out of the way.

You could obviously reduce the cost of searching to find the right place. Since Jobs in the fJobs array are in order, a binary search mechanism could be used. (Do you still remember binary search from Chapter 13?) This would reduce the search costs to O(lgN) but the data shuffling steps would still be O(N).

If you could make the data shuffling steps as efficient as the search, then the overall cost would be introduced to lgN. It doesn't make that much difference if the queues are short and the frequency of calls is low (when N≈10, then O(lgN)/O(N) ≈ 3/10.) If the queues are long, or operations on the queue are very frequent, then changing to a O(lgN) algorithm gets to be important (when N≈1000, then O(lgN)/O(N) ≈ 10/1000.).

An alternative O(lgN) algorithm

There is an algorithm that makes both searching and data shuffling operations have costs that are O(lgN). The array elements, the Jobs, are kept "partially ordered". The highest priority job will be in the first array element; the next two most urgent jobs will be in the second and third locations, but they won't necessarily be in order. When the most urgent job is removed, the next most urgent pops up from the second or the third location; in turn, it is replaced by some other less urgent job. Once all the changes have been made, the partial ordering condition will still hold. The most urgent of the

remaining jobs will be in the first array element, with the next two most urgent jobs in the successive locations.

The algorithm is quite smart in that in manages to keep this partial ordering of the data with a minimum number of comparison and data shuffling steps. Figures 21.2 illustrates the model used to keep the data "partially ordered".

Trees, roots, and tree-nodes

You should imagine the data values placed in storage elements that are arranged in a branching "tree"-like manner. Like most of the "trees" that appear in illustrations of data structures, this tree "grows" downwards. The topmost "node" (branch point) is the "root". It contains the data value with the most urgent priority (smallest "prio" value).

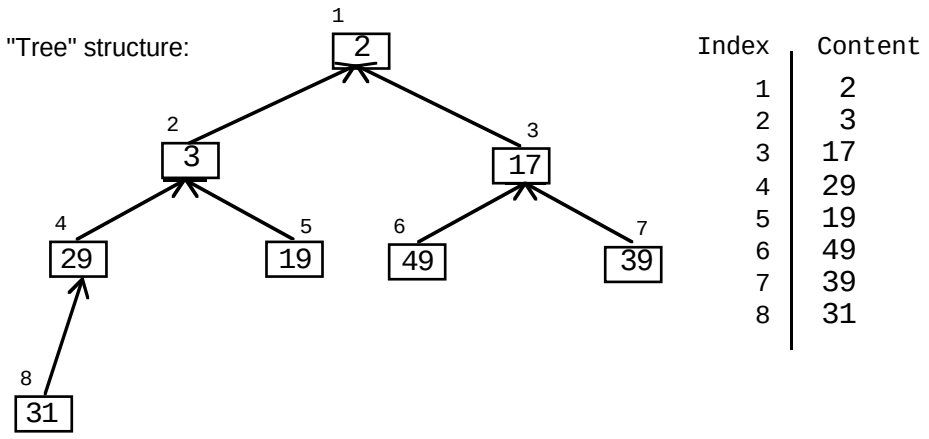

Figure 21.2 "Tree" of partially ordered values and their mapping into an array.

Child nodes and leaves

Each "node" in the tree can have up to two nodes in the level below it. (These are referred to as its "child nodes"; the terminology is very mixed up!) A node with no children is a "leaf".

At every level, the data value stored in a node is smaller than the data values stored in its children. As shown in Figure 21.2, you have the first node holding the value 2, the second and third nodes hold the values 3 and 17. The second node, the one with the 3 has children that hold the values 29 and 19 and so on.

This particular tree isn't allowed to grow arbitrarily. The fourth and fifth nodes added to the tree become children of the second node; the sixth and seventh additions are children of the third node. The eight to fifteenth nodes inclusive form the next level of this tree; with the 8th and 9th "children" of node 4 and so on.

"Tree" structure can be stored in an array

Most of the "trees" that you will meet as data structures are built by allocating separating node structs (using the new operator) and linking them with pointers. This particular form of tree is exceptional. The restrictions on its shape mean that it can be mapped onto successive array elements, as also suggested in Figure 21.2

The arrangement of data values in successive array entries is:

```
(#1,  2) (#2,  3) (#3,  17) (#4, 29) (#5, 19) (#6, 49) (#7, 39)
(#8, 31)
```

The values are "partially sorted", the small (high priority) items are near the top of the array while the large values are at the bottom. But the values are certainly not in sorted order (e.g. 19 comes after 29).

The trick about this arrangement is that when a new data value is added at the end, it doesn't take very much work to rearrange data values to restore the partial ordering. This is illustrated in Figure 21.3 and 21.4.

Figure 21.3 illustrates the situation immediately after a new data value, with priority 5, has been added to the collection. The collection is no longer partially ordered. The data value in node 4, 29, is no longer smaller than the values held by its children.

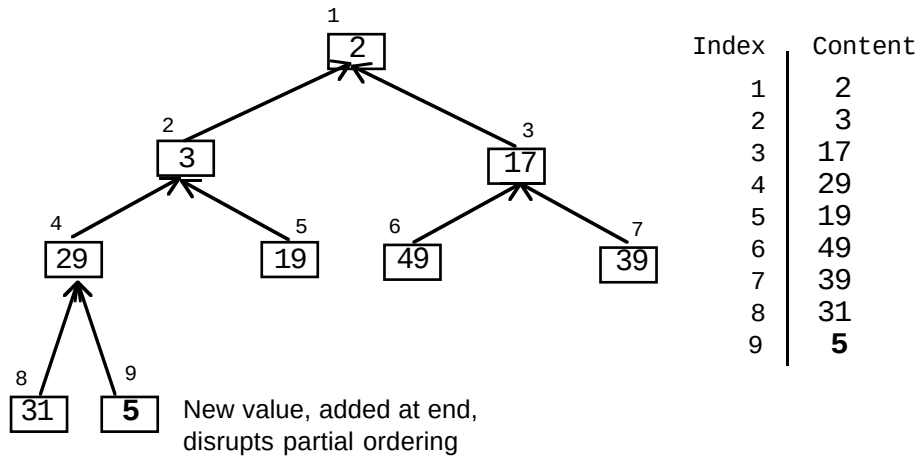

Figure 21.3 Addition of data element disrupts the partial ordering.

The partial ordering condition can be restored by letting the new data value work its way up the branches that link its starting position to the root. If the new value is smaller than the value in its "parent node", the two values are exchanged.

In this case, the new value in node 9 is compared with the value in its parent node, node 4. As the value 5 is smaller than 29, they swap. The process continues. The value 5 in node 4 is compared with the value in its parent node (node 2). Here the comparison is between 3 and 5 and the smaller value is already in the lower node, so no changes are necessary and the checking procedure terminates. Figure 21.4 illustrates the tree and array with the partial ordering reestablished.

Note that the newly added value doesn't get compared with all the other stored data values. It only gets compared with those along its "path to root".

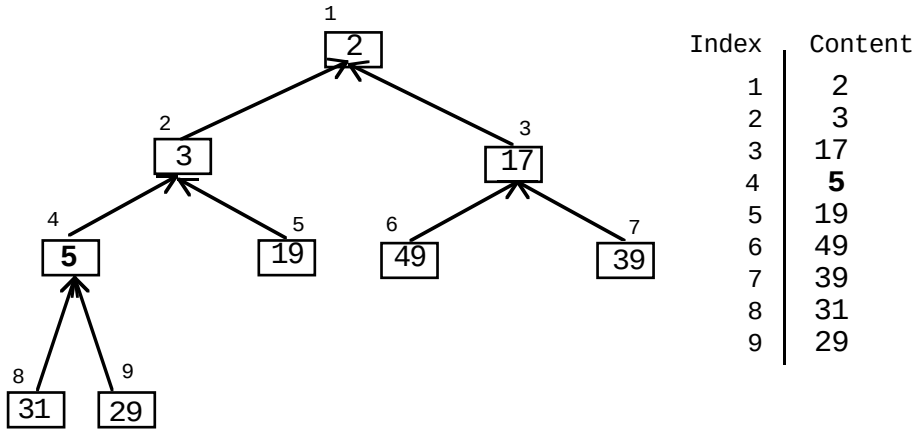

Figure 21.4 Restoration of the partial ordering.

If the newly added value was smaller than any already in the storage structure, e.g. value 1, it would work its way all the way up until it occupied the root-node in array element 1. In this "worst case", the number of comparison and swap steps would then be the same as the number of links from the root to the leaf.

O(lgN) algorithm

Using arguments similar to those presented in Chapter 13 (when discussing binary search and Quicksort), you can show that the maximum number of links from root to leaf will be proportional to lgN where N is the number of data values stored in the structure. So, the Insert operation has a O(lgN) time behaviour.

Removing the top priority item

When the "first" (remove) operation is performed, the top priority item gets removed. One of the other stored data values has to replace the data value removed from the root node. As illustrated in Figure 21.5; the restoration scheme starts by "promoting" the data value from the last occupied node.

Of course, this destroys the partial ordering. The value 29 now in the root node (array element 1) is no longer smaller than the values in both its children.

Tidying up after removing the first element

So, once again a process of rearrangement takes place. Starting at the root node, the data value in a node is compared with those in its children. The data values in parent node and a child node are switched as needed. So, in this case, the value 29 in node 1 switches places with the 3 in node 2. The process is then repeated. The value 29 now in node 2 is switched for the value 5 in node 4. The switching process can then terminate because partial ordering has again been restored.

As in the case of insert, this tidying up after a removal only compares values down from root to leaf. Again its cost is O(lgN).

Although a little more involved than the algorithms for TrivialPQ, the functions needed are not particularly complex. The improved O(lgN) (as opposed to O(N)) performance makes it worth putting in the extra coding effort.

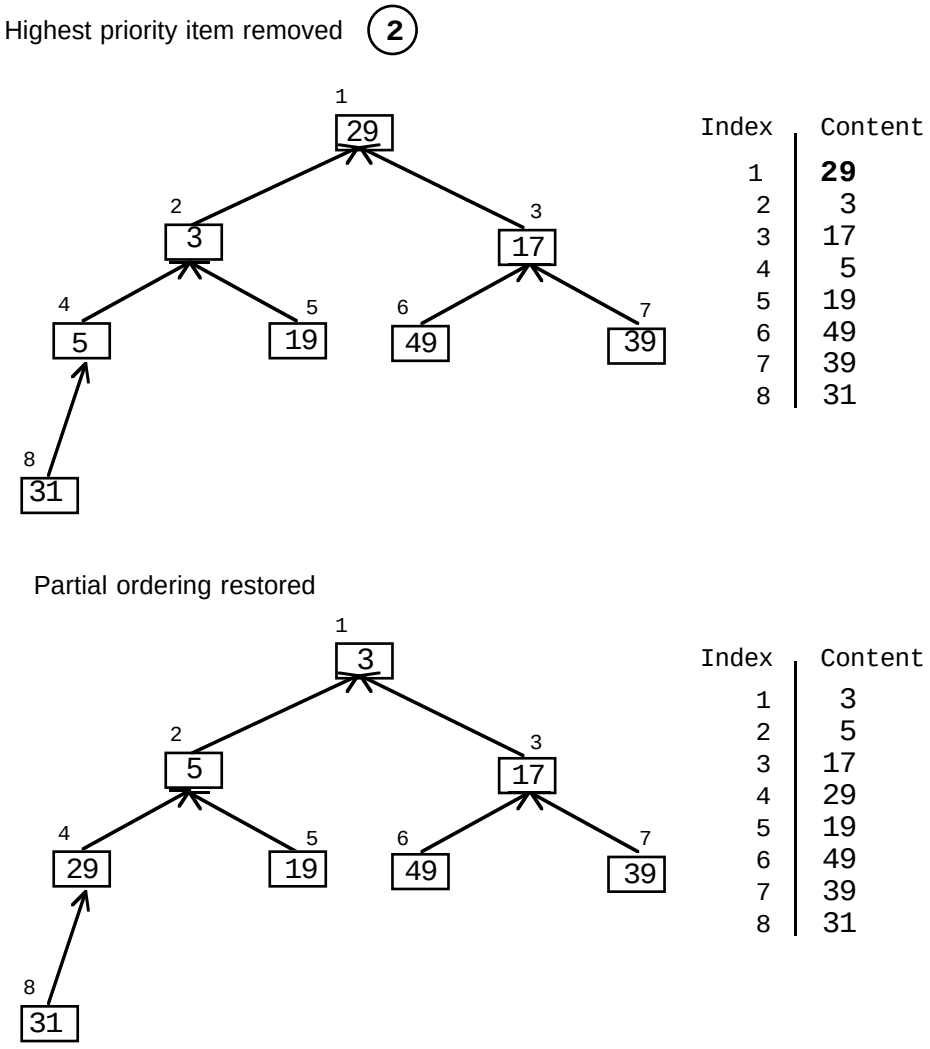

Figure 21.5 Removal of top priority item followed by restoration of partial ordering.

"Heapsort"

Values get entered in random order, but they will be removed in increasing order. You can see that the priority queue could be used to sort data. You load up the priority queue with all the data elements (using for "priorities" the key values on which you want the data elements sorted). Once they are all loaded, you keep removing the first element in the priority queue until it becomes empty. The values come out sorted.

This particular sorting algorithm is called "heapsort". Its efficiency is O(NlgN) (roughly, you are doing O(lgN) insert and remove operations for each of N data elements). Heapsort and Quicksort have similar performances; different degrees of

ordering in the initial data may slightly favour one algorithm relative to the other. Of course, heapsort has the extra space overhead of the array used to represent the queue.

The name "heap" in heapsort is somewhat unfortunate. It has nothing to do with "the Heap" used for free storage allocation.

Design and implementation of class PriorityQ

A priority queue owns …

What does a priority queue own? This priority queue owns an array in which it keeps pointers to the queued items along with details of their priorities; there will also be an integer data member whose value defines the number of items queued.

A priority queue does …

A priority queue can respond to the following requests:

- First
  Remove the front element from the queue and return it.
- Add (preferred name Insert)
  Insert another item into the queue at a position determined by its priority
- Length
  Report how many items are queued.
- Full
  As the queue only has a finite amount of storage, it may get to be full, a further Insert operation would then cause some error. The queue has to have a "Full" member function which returns true it is full.
- Empty
  Returns true if there are no data elements queued; an error would occur if a First operation was performed on an empty queue.

and supplementary debugging functions

In cases like this where the algorithms are getting a little more complex, it often helps to have some extra functions to support debugging. When building any kind of tree or graph, you will find it useful to have some form of "print" function that can provide structural information. As this is not part of the interface that would normally be needed by clients, it should be implemented as conditionally compiled code.

Auxiliary private member functions

The `Insert()` and `First()` member functions make initial changes to the data in the array, then the "partial ordering" has to be reestablished. With `Insert()`, a data element gets added at the bottom of the array and has to find its way up through the conceptual tree structure until it gets in place. With `First()` a data element gets put in at the root and has to drop down until it is in place. These rearrangements are best handled by auxiliary private member functions (they could be done in `Insert()` and `First()` but it is always better to split things up into smaller functions that are easier to analyze and understand).

A struct to store details of queued objects and priorities

The storage array used by a `PriorityQ` will be an array of simple structs that incorporate a `void*` pointer to the queued object and a long integer priority. This can mean that the priority value gets duplicated (because it may also occur as an actual data member within the queued object). Often though, the priority really is something that

is only meaningful while the object is associated with the priority queue; after all, casualty patients cease to have priorities if they get to be admitted as ward patients.

The type of struct used to store these data is only used within the priority queue code. Since nothing else in the program need know about these structs, the struct declaration can be hidden inside the declaration of class `PriorityQ`.

The "pq.h" header file with the class declaration is:

```
#ifndef __MYPQ__
#define __MYPQ__

#define DEBUG

#define k_PQ_SIZE 50
class PriorityQ {                                          Class declaration
public:
    PriorityQ();

    void   Insert(void* newitem, long priority);
    int    Length(void) const;
    int    Full(void) const;
    int    Empty(void) const;

    void   *First(void);
#ifdef DEBUG
    void   PrintOn(ostream& out) const;
#endif
private:
    void   TidyUp(void);
    void   TidyDown(void);

    struct keyeddata { long fK; void *fd; };
    keyeddata     fQueue[k_PQ_SIZE+1];
    int           fcount;
};

inline int PriorityQ::Length(void) const { return fcount; }
inline int PriorityQ::Full(void) const
    { return fcount == k_PQ_SIZE; }
inline int PriorityQ::Empty(void) const { return fcount == 0; }

#endif
```

Here, the token `DEBUG` is defined so the `PrintOn()` function will be included in the generated code. (Coding is slightly easier if element zero of the array is unused; to allow for this, the array size is `k_PQ_SIZE+1`.)

As in the case of class `Queue`, the simple member functions like `Length()` can be defined as "inlines" and included in the header file.

The constructor has to zero out the count; it isn't really necessary to clear out the array but this is done anyway: ***Constructor***

```
PriorityQ::PriorityQ()
{
    for(int i=0; i<= k_PQ_SIZE; i++) {
           fQueue[i].fd = NULL;
           fQueue[i].fK = 0;
           }
    fcount = 0;
}
```

The `Insert()` function increments the count and adds the new queued item by filling in the next element in the `fQueue` array. Function `TidyUp()` is then called to restore partial ordering. `First()` works in an analogous way using `TidyDown()`:. Note that neither includes any checks on the validity of the operations; it would be wiser to include `Full()` and `Empty()` checks with calls to an error function as was done for class `Queue`.

Insert

```
void PriorityQ::Insert(void* newitem, long priority)
{
    fcount++;
    fQueue[fcount].fd = newitem;
    fQueue[fcount].fK = priority;
    TidyUp();
}
```

First

```
void *PriorityQ::First(void)
{
    void *chosen = fQueue[1].fd;
    fQueue[1] = fQueue[fcount];
    fcount--;
    TidyDown();
    return chosen;
}
```

The `PrintOn()` routine has to be in "conditionally compiled" brackets. It simply runs through the array identifying the priority of the data object in each position. This sort of information can help when checking out the implementation:

```
#ifdef DEBUG
void PriorityQ::PrintOn(ostream& out) const
{
    if(fcount == 0) {
           cout << "Queue is empty" << endl;
           return;
           }

    cout << "Position      Item Priority" << endl;
    for(int i = 1; i<= fcount; i++)
        cout << i << "         " << fQueue[i].fK << endl;
```

```
}
#endif
```

The `TidyUp()` function implements that chase up through the "branches of the tree" comparing the value of the data item in a node with the value of the item in its parent node. The loop stops if a data item gets moved up to the first node, or if the parent has a data item with a lower `fK` value (higher priority value). Note that you find the array index of a node's parent by halving its own index; e.g. the node at array element 9 is at 9/2 or 4 (integer division) which is as shown in the Figures 21.3 etc.

Restoring partial ordering

```
void PriorityQ::TidyUp(void)
{
    int           k = fcount;
    keyeddata     v = fQueue[k];
    for(;;) {
           if(k==1) break;
           int nk = k / 2;
           if(fQueue[nk].fK < v.fK) break;
           fQueue[k] = fQueue[nk];
           k = nk;
           }
    fQueue[k] = v;
}
```

The code for `TidyDown()` is generally similar, but it has to check both children (assuming that there are two child nodes) because either one of them could hold a data item with a priority value smaller than that associated with the item "dropping down" through levels of the "tree". (The code picks the child with the data item with the smaller priority key and pushes the descending item down that branch.)

```
void PriorityQ::TidyDown(void)
{
    int           Nelems = fcount;
    int           k = 1;
    keyeddata     v = fQueue[k];
    for(;;) {
           int nk;
           if(k>(Nelems/ 2)) break;
           nk = k + k;
           if((nk <= Nelems-1) &&
                  (fQueue[nk].fK > fQueue[nk+1].fK)) nk++;

           if(v.fK <= fQueue[nk].fK) break;
           fQueue[k] = fQueue[nk];
           k = nk;
           }
    fQueue[k] = v;
}
```

Test program

Once again, a small test program must be provided along with the class. This test program first checks out whether "heapsort" works with this implementation of the priority queue. The queue is loaded up with some "random data" entered by the user, then the data values are removed one by one to check that they come out in order. The second part of the test is similar to the test program for the ordinary queue. It uses an interactive routine with the user entering commands that add things and remove things from the queue.

The test program starts by including appropriate header files and defining a `Job` struct; the priority queue will store `Jobs` (for this test, a `Job` has just a single character array data member):

```
#include <stdlib.h>
#include <iostream.h>
#include "pq.h"

struct Job {
    char   name[30];
};

Job* GetJob()
{
    cout << "job name> ";
    Job *j = new Job;
    cin >> j->name;
    return j;
}

main()
{
    PriorityQ     thePQ;
    int    i;

/*
Code doing a "heapsort"
*/
cout << "enter data for heap sort test; -ve prio to end data"
           << endl;
for(i = 0; ; i++)
{
    int prio;
    cout << "job priority ";
    cin >> prio;
    if(prio < 0)
           break;
    Job* j = GetJob();
    thePQ.Insert(j, prio);
```

Load the queue with some data

```
    if(thePQ.Full())
           break;
}

while(!thePQ.Empty()) {                                    Pull the data off the
    Job *j = (Job*) thePQ.First();                         queue, hope it comes
    cout <<  j->name << endl;                              out sorted
    delete j;
    }

    for(int done = 0; !done ;) {                           Interactive part
           Job *j;                                         adding, removing
           char ch;                                        items etc
           cout << ">";
           cin >> ch;

           switch(ch) {
case 'q':  done = 1;
           break;
case 'l' : cout << "Queue length now " << thePQ.Length()
                        << endl;
           break;
case 'a':
           if(thePQ.Full()) cout << "Queue is full!" << endl;
           else {
                  int prio;
                  cout << "priority "; cin >> prio;
                  j = GetJob();
                  thePQ.Insert(j,prio);
                  }
           break;
case 's':
           thePQ.PrintOn(cout);;
           break;
case 'g':
           if(thePQ.Empty()) cout << "Its empty!" << endl;
           else {
                  j = (Job*) thePQ.First();                Type case from void*
                  cout << "Removed " << j->name << endl;   to Job*
                  delete j;
                  }
           break;
case '?':
           cout << "Commands are:" << endl;
           cout << "\tq Quit\n\ta Add to queue\t" << endl;
           cout << "\ts show queue status\n\tg"
                  "Get job at front of queue" << endl;
           cout << "\tl Length of queue\n" << endl;
           break;
default:
           ;
           }
```

```
    }

    return 0;
}
```

Note the `(Job*)` type casts. We know that we put pointers to `Jobs` into this priority queue but when they come back they are `void*` pointers. The type cast is necessary, you can't do anything with a `void*`. (The type cast is also safe here; nothing else is going to get into this queue). The code carefully deletes `Jobs` as they are removed from the priority queue; but it doesn't clean up completely (there could be `Jobs` still in the queue when the program exits).

Test output

```
...
job priority 5
job name> data5
job priority 107
job name> data107
job priority -1
data5
data11
data17
data35
data55
data92
data107
data108
>a
priority 18
job name> drunk
>a
priority 7
job name> spider-venom
>a
priority 2
job name> pain-in-chest
>a
priority 30
job name> pain-in-a---
>a
priority 1
job name> gunshot
>g
Removed gunshot
>g
Removed pain-in-chest
>g
Removed spider-venom
>g
Removed drunk
>q
```

21.3 CLASS DYNAMICARRAY

It is surprising how rarely dynamic arrays feature in books on data structures. A dynamic array looks after a variable size collection of data items; thus, it performs much the same sort of role as does a List. In many situations, dynamic arrays perform better than lists because they incur less overhead costs.

Both "list" and "dynamic array" allow "client programs" (i.e. programs that use instances of these classes) to:

- add data objects to the collection (as in the rest of this chapter, the data objects are identified by their addresses and these are handled as `void*` pointers);
- ask for the number of objects stored;
- ask whether a particular object is in the collection;
- remove chosen objects from the collection.

In these two kinds of collection there is an implied ordering of the stored data items. There is a "first item", a "second item" and so forth. In the simple versions considered here, this ordering doesn't relate to any attributes of the data items themselves; the ordering is determined solely by the order of addition. As the items are in sequence, these collections also allow client programs to ask for the first, second, …, *n* -th item.

Figure 21.6 illustrates a dynamic array. A dynamic array object has a main data block which contains information like the number of items stored, and a separate array of pointers. The addresses of the individual data items stored in the collection are held in this array. This array is always allocated in the heap (or simplicity, the header and trailer information added by the run-time memory manager are not shown in illustrations). The basic data block with the other information may be in the heap, or may be an automatic in the stack, or may be a static variable.

As shown in Figure 21.6, the array of pointers can vary in size. It is this variation in size that makes the array "dynamic".

When a dynamic array is created it gets allocated an array with some default number of entries; the example in Figure 21.6 shows a default array size of ten entries. As data objects get added to the collection, the elements of the array get filled in with data addresses. Eventually, the array will become full.

"Growing the array"

When it does become full, the `DynamicArray` object arranges for another larger array to be allocated in the heap (using operator `new []`). All the pointers to stored data items are copied form the existing array to the new larger array, and then the existing array is freed (using operator `delete []`). You can vary the amount of growth of the array, for example you could make it increase by 25% each time. The simplest implementations of a dynamic array grow the array by a specified amount, for example the array might start with ten pointers and grow by five each time it gets filled.

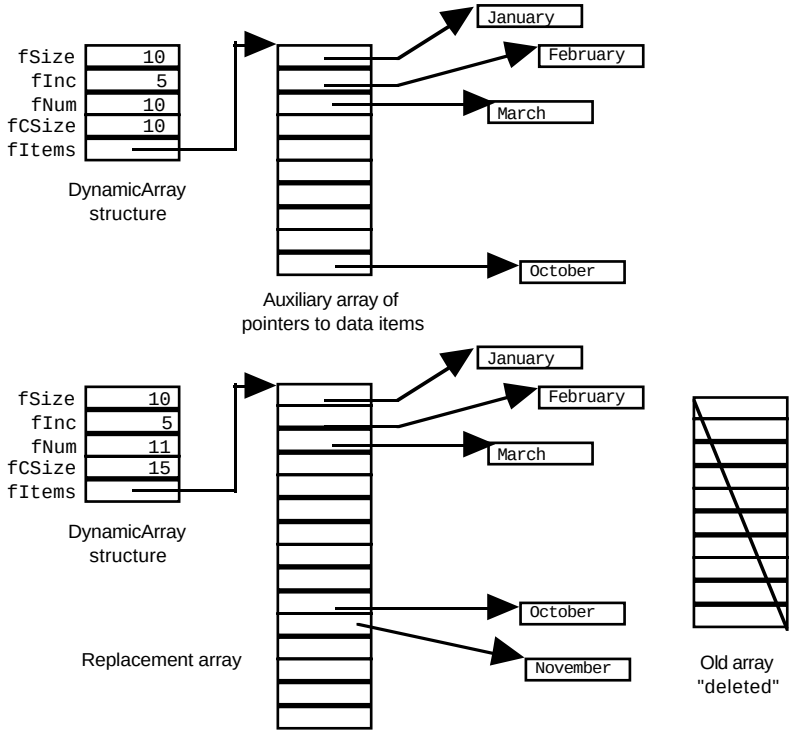

Figure 21.6 A "Dynamic Array".

Removing items and "Shrinking" the array

Stored items do get removed from the array. It isn't like a queue where only the first item can be removed, a "remove" request will identify the item to be removed. When an item is removed, the "gap" in the array is filled up by moving up the data items from the later array elements. So, if the array had had seven elements, removal of the fifth would result in the pointers in the sixth and seventh elements each moving up one place. If lots of items are removed, it may be worth shrinking the array. "Shrinking" is similar to "growing". The `DynamicArray` object again creates a new array (this time smaller) in the heap and copies existing data into the new array. The old large array is deleted. Typically, the array is only shrunk when a remove operation leaves a significant number of unused elements. The array is never made smaller than the initially allocated size.

What does a dynamic array own?

The main thing a `DynamicArray` object owns is the array of pointers to the data items. In addition, it needs to have integer data members that specify: initial size (this is also the minimum size), the size of the increment, the current array size, and the number of data elements currently stored.

A DynamicArray does …

You can vary the behaviours of DyanamicArrays and Lists a little. These classes have to have functions to support at least some version of the following behaviours:

- Length
  Report how many items are stored in the collection.
- Add (preferred name Append)
  Add another item to the collection, placing it "after" all existing items.
- Position
  Find where an item is in the collection (returning its sequence number); this function returns an error indicator if the item is not present. (This function may be replaced by a Member function that simply returns a true or false indication of whether an item is present).
- Nth
  Get the item at a specified position in the collection (the item is not removed from the collection by this operation).
- Remove
  Removes a specified item, or removes the item at a specified position in the collection.

Note that there is neither a "Full" nor an "Empty" function. DynamicArrays and Lists are never full, they can always grow. If a client program needs to check for an "empty" collection, it can just get the length and check whether this is zero.

You have to chose whether the sequence numbers for stored items start with 0 or 1. Starting with 0 is more consistent with C/C++ arrays, but usually a 1-base is more convenient for client applications. So, if there are *N* elements in the collection, the "Position" function will return values in the range 1…N and functions like "Nth", and "Remove" will require arguments in the range 1…N.

The remove function should return a pointer to the item "removed" from the collection.

If desired, a debugging printout function can be included. This would print details like the current array size and number of elements. It is also possible for the function to print out details of the addresses of stored data items, but this information is rarely helpful.

A declaration for a simple version of the `DynamicArray` class is:

File "D.h" with the declaration of class DynamicArray

```
#ifndef __MYDYNAM__
#define __MYDYNAM__

class DynamicArray {
public:
    DynamicArray(int size = 10, int inc = 5);

    int    Length(void) const;
    int    Position(void *item) const;
    void   *Nth(int n) const;

    void   Append(void *item);

    void   *Remove(void *item);
```

Default parameters

```
    void    *Remove(int itempos);

private:
    void    Grow(int amount);

    int     fNum;
    int     fSize;
    int     fCSize;
    int     fInc;
    void    **fItems;
};

inline int DynamicArray::Length(void) const { return fNum; }

#endif
```

Auxiliary "Grow" function as private member

The "pointer to pointer"

For a class like this, it is reasonable for the class declaration to provide default initial values for parameters like the initial array size and the increment, so these appear in the declaration for the constructor.

The next group of functions are all const functions; they don't change the contents of a `DynamicArray`. Once again, the `Length()` function is simple and can be included in the header file as an inline function.

Along with the `Append()` function, there are two overloaded versions of `Remove()`; one removes an item at a specified position in the collection, the other finds an item and (if successful) removes it. The `Append()` and `Remove()` functions require auxiliary "grow" and "shrink" functions. Actually, a single `Grow()` function will suffice, it can be called with a positive increment to expand the array or a negative increment to contract the array. The `Grow()` function is obviously private as it is merely an implementation detail.

*void** ! ?*

Most of the data members are simple integers. However, the `fItems` data member that holds the address of the array of pointers is of type `void**`. This seemingly odd type requires a little thought. Remember, a pointer like `char*` can hold the address of a `char` variable or the address of the start of an array of `chars`; similarly an `int*` holds the address of an `int` variable or the address of an array of `int`s. A `void*` holds an address of something; it could be interpreted as being the address of the start of "an array of `voids`" but you can't have such a thing (while there are `void*` variables you can't have `void` variables). Here we need the address of the start of an array of `void*` variables (or a single `void*` variable if the array size is 1). Since `fItems` holds the address of a `void*` variable, its own type has to be `void**` (pointer to a pointer to something).

The code implementing the class member functions would be in a separate "D.cp" file. The constructor fills in the integer data members and allocates the array:

```
#include <stdlib.h>
#include <assert.h>
#include "D.h"
```

```
const int kLARGE = 10000;
const int kINCL = 5000;

DynamicArray::DynamicArray(int size, int inc)                    Constructor
{
    assert((size > 1) && (size < kLARGE));
    assert((inc > 1) && (inc < kINCL));

    fNum = 0;
    fCSize = fSize = size;
    fInc = inc;

    fItems = new void* [size];
}
```

Note that the definition does not repeat the default values for the arguments; these only appear in the initial declaration. (The `assert()` checks at the start protect against silly requests to create arrays with -1 or 1000000 elements.) The `new []` operator is used to create the array with the specified number of pointers.

Function `Length()` was defined as an inline. The other two const access functions involve loops or conditional tests and so are less suited to being inline.

```
int DynamicArray::Position(void *item) const
{
    for(int i = 0; i< fNum; i++)
           if(fItems[i] == item) return i+1;
    return 0;
}

void *DynamicArray::Nth(int n) const
{
    if((n < 1) || (n > fNum))
           return NULL;
    n--;
    return fItems[n];
}
```

The class interface defines usage in terms of indices 1…N. Of course, the implementation uses array elements 0…N-1. These functions fix up the differences (e.g. `Nth()` decrements its argument `n` before using it to access the array).

Both these functions have to deal with erroneous argument data. Function `Position()` can return zero to indicate that the requested argument was not in the collection. Function `Nth()` can return `NULL` if the index is out of range. Function `Position()` works by comparing the address of the item with the addresses held in each pointer in the array; if the addresses match, the item is in the array.

The `Append()` function starts by checking whether the current array is full; if it is, a call is made to the auxiliary `Grow()` function to enlarge the array. The new item can

then be added to the array (or the new enlarged array) and the count of stored items gets incremented:

```
void DynamicArray::Append(void *item)
{
    if(fNum == fCSize)
           Grow(fInc);
    fItems[fNum] = item;
    fNum++;
}
```

The `Grow()` function itself is simple. The new array of pointers is created and the record of the array size is updated. The addresses are copied from the pointers in the current array to the pointers in the new array. The old array is freed and the address of the new array is recorded.

"Growing" the array

```
void DynamicArray::Grow(int delta)
{
    int newsize = fCSize + delta;
    void **temp = new void* [newsize];
    fCSize = newsize;
    for(int i=0;i < fNum; i++)
           temp[i] = fItems[i];
    delete [] fItems;
    fItems = temp;
}
```

The `Remove(void *item)` function has to find the data item and then remove it. But function `Position()` already exists to find an item, and `Remove(int)` will remove an item at a known position. The first `Remove()` function can rely on these other functions:

```
void *DynamicArray::Remove(void *item)
{
    int where = Position(item);
    return Remove(where);
}
```

The `Remove(int)` function verifies its argument is in range, returning `NULL` if the element is not in the array. If the required element is present, its address is copied into a temporary variable and then the array entries are "closed up" to remove the gap that would otherwise be left.

The scheme for shrinking the array is fairly conservative. The array is only shrunk if remove operations have left it less than half full and even then it is only shrunk a little.

```
void *DynamicArray::Remove(int itempos)
{
```

```
    if((itempos < 1) || (itempos > fNum))
            return NULL;
    itempos--;
    void *tmp = fItems[itempos];
    for(int i = itempos + 1; i < fNum; i++)
            fItems[i-1] = fItems[i];
    fNum--;

    if((fNum > fSize) && (fNum < (fCSize / 2)))
            Grow(-fInc);
    return tmp;
}
```

Test program

The test program for this class creates "book" objects and adds them to the collection. For simplicity, "books" are just character arrays allocated in the heap. Rather than force the tester to type in lots of data, the program has predefined data strings that can be used to initialize the "books" that get created.

```
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include "D.h"

typedef char *Book;
Book BookStore[] = {
    "The C++ Programming Language",
    "Developing C++ Software",
    "Algorithms in C++",
    …
    "Effective C++",
    "Object Models, Strategies, Patterns, and Applications"
};

int Numbooks = sizeof(BookStore) / sizeof(char*);

Book PickABook(void)
{
    int num;
    cout << "Pick a book, any book (#1 … "
            << Numbooks << "): " << endl;
    cin >> num;
    while((num < 1) || (num > Numbooks)) {
            cout << "Try again, enter number ";
            cin.clear();
            cin.ignore(100,'\n');
            cin >> num;
            }
```

Predefined data to reduce amount of data entry in test program

Function that creates a "Book" on the heap

```
    num--;
    cout << "You picked : " << BookStore[num] << endl;
    Book ptr;
    ptr = new char[strlen(BookStore[num]) + 1];
    strcpy(ptr, BookStore[num]);
    return ptr;
}
```

Function `PickABook()` allows the tester to simply enter a number; this selects a string from the predefined array and uses this string to initialize a dynamically created Book.

Function `GetPos()` is another auxiliary function used to allow the tester to enter the sequence number of an item in the collection:

```
int GetPos(int max)
{
    if(max < 1) {
           cout << "The collection is empty!" << endl;
           return 0;
           }
    cout << "Which item from collection? ";
    int temp;
    cin >> temp;
    while((temp < 1) || (temp > max)) {
           cout << "Position must be in range 1..." <<
                   max << endl;
           cin.clear();
           cin.ignore(100,'\n');
           cin >> temp;
           }
    return temp;
}
```

Like most such test programs, this one consists of a loop in which the user is prompted for commands. These commands exercise the different member functions of the class:

```
int main()
{
    Book lastadded = NULL;
    DynamicArray c1;

    for(int done = 0; !done ;) {
           char command;
           Book ptr;
           int pos;
           cout << ">";
           cin >> command;
           switch(command) {
case 'q':  done = 1; break;
```

```
case 'l' :                                                    Check Length()
            cout << "Collection now contains "
                  << c1.Length()
                        << " data items" << endl;
            break;
case 'a' :
            ptr = PickABook();
            if(c1.Position(ptr)) {
                  // Note this will never happen,
                  // see discussion in text
                  cout << "You've already got that one!" << endl;
                  delete ptr;
                  }
            else {
                  c1.Append(ptr);                             Check Append()
                  cout << "Added" << endl;
                  lastadded = ptr;
                  }
            break;
case 'f':
            pos = GetPos(c1.Length());
            if(pos > 0) {
                  ptr = (char*) c1.Nth(pos);                  Check Nth()
                  cout << pos << " : " << ptr << endl;
                  }
            break;
case 'F':
            if(lastadded == NULL)
                  cout << "Got to add something first" << endl;
            else {
                  cout << "Last book added was "
                        << lastadded << endl;
                  pos = c1.Position(lastadded);               Check Position()
                  if(pos)
                        cout << "That's still in the collection"
                              " at position "
                                    << pos << endl;
                  else
                        cout << "You seem to have got rid"
                              " of it." << endl;
                  }
            break;
case 'r':
            pos = GetPos(c1.Length());
            if(pos > 0) {
                  ptr = (Book) c1.Remove(pos);                Check Remove(int)
                  cout << "Removed " << ptr << " from collection"
                        << endl;
                  cout << "Collection now has " <<
                        c1.Length() << " items" << endl;
                  delete ptr;
                  }
```

```
            break;

    case '?':
            cout << "Commands are:" << endl;
            cout << "\tq Quit\n\ta Add to collection\t" << endl;
            cout << "\tf Find item in collection (use number)"
                    << endl;
            cout << "\tF Find last added item in collection"
                    " (use pointer)" << endl;
            cout << "\tr Remove item from collection"
                    " (use number)" << endl;
            cout << "\tl Length (size) of collection\n" << endl;
            break;
    default:
            ;
            }
        }
        return EXIT_SUCCESS;
    }
```

Most of the testing code should be easy to understand.

Note that function `PickABook()` creates a new data structure in the heap for each call. So, if you choose book 1 ("C++ Programming Language") twice, you will get two separate data structures in the heap, both of which contain this string. These structures are at different addresses. The check "`c1.Position(ptr)`" in `case 'a':` will always fail (get a zero result) because the newly allocated book isn't in the collection (even though the collection may already contain a book with similar content). In contrast, the test `c1.Position(lastadded);` in `case 'F':` may succeed (it will succeed if the last book added is still in the collection because then the address in `lastadded` will be the same as the address in the last of the used pointers in the array.

Function `Position(void*)` implements an identity check. It determines whether the specific item identified by the argument is in the array. It does not check for an equal item. In this context, an identity test is what is required, but sometimes it is useful to have an equality test as well . An equality test would allow you to find whether the array contained an item similar to the one referred to in the query.

In this simplified implementation of class `DynamicArray`, it is not possible to have an equality test. You can only test two data objects for equality if you know their structure or know of a routine for comparing them. You never know anything about a data item pointed to by a `void*` so you can't do such tests.

On the whole, a `DynamicArray` is pretty efficient at its job. Remove operations do result in data shuffling. Some implementations cut down on data shuffling by having an auxiliary array of booleans; the true or false setting of a boolean indicates whether a data element referenced by a pointer is still in the collection. When an item is removed, the corresponding boolean is set to false. Data are only reshuffled when the array has a significant number of "removed" items. This speeds up removals, but makes operations

like `Nth()` more costly (as it must then involve a loop that finds the n-th item among those not yet removed).

21.4 CLASS LIST

A list is the hardy perennial that appears in every data structures book. It comes second in popularity only to the "stack". (The "stack" of the data structures books is covered in an exercise. It is not the stack used to organize stackframes and function calls; it is an impoverished simplified variant.)

There are several slightly different forms of list, and numerous specialized list classes that have extra functionality required for specific applications. But a basic list supports the same functionality as does a dynamic array:

- Length
- Append
- Position
- Nth
- Remove

The data storage, and the mechanisms used to implement the functions are however quite different.

List cells, next pointers, and the head pointer

The basics of operations on lists were explained in Section 20.6 and illustrated in Figure 20.7. The list structure is made up from "list cells". These are small structs, allocated on the heap. Each "list cell" holds a pointer to a stored data element and a pointer to the next list cell. The list structure itself holds a pointer to the first list cell; this is the "head" pointer for the list.

```
struct ListCell { void *fData; ListCell *fNext; };

class List {
…
private:
    ListCell      *fHead;
    …
};
```

This arrangement of a "head pointer" and the "next" links makes it easy to get at the first entry in the list and to work along the list accessing each stored element in turn.

A "tail" pointer

An "append" operation adds to the end of the list. The address of the added list cell has to be stored in the "next" link of the cell that was previously at the end of the list. If you only have a head pointer and next links, then each "append" operation involves searching along the list until the end is found. When the list is long, this operation gets to be a little slow. Most implementations of the list abstract data type avoid this cost by having a "tail" pointer in addition to the head pointer. The head pointer holds the

address of the first list cell, the tail pointer holds the address of the las list cell; they will hold the same address if the list only has one member. Maintaining both head and tail pointers means that there is a little more "housekeeping" work to be done in functions that add and remove items but the improved handling of append operations makes this extra housekeeping worthwhile.

"Previous" links in the list cells

Sometimes it is worth having two list cell pointers in each list cell:

```
struct D_ListCell {
    void          *fData;
    D_ListCell    *fNext;
    D_ListCell    *fPrev;
};
```

These "previous" links make it as easy to move backwards toward the start of the list as it is to move forwards toward the end of the list. In some applications where lists are used it is necessary to move backwards and forwards within the same list. Such applications use "doubly linked" lists. Here, the examples will consider just the singly linked form.

The class declaration for a simple singly linked list is:

File "List.h" with class declaration

```
#ifndef __MYLIST__
#define __MYLIST__

class List {
public:
    List();

    int    Length(void) const;
    int    Position(void *item) const;

    void   *Nth(int n) const;

    void   Append(void *item);

    void   *Remove(void *item);
    void   *Remove(int itempos);

private:
    struct ListCell { void *fData; ListCell *fNext; };
    int           fNum;
    ListCell      *fHead;
    ListCell      *fTail;
};

inline int List::Length(void) const { return fNum; }

#endif
```

Apart from the constructor, the public interface for this class is identical to that of class DynamicArray.

The struct ListCell is declared within the private part of class List. ListCells are an implementation detail of the class; they are needed only within the List code and shouldn't be used elsewhere.

The data members consist of the two ListCell* pointers for head and tail and fNum the integer count of the number of stored items. You could miss out the fNum data member and work out the length each time it is needed, but having a counter is more convenient. The Length() function, which returns the value in this fNum field, can once again be an inline function defined in the header file.

The remaining member functions would be defined in a separate List.cp implementation file.

The constructor is simple, just zero out the count and set both head and tail pointers to NULL:

```
#include <stdlib.h>
#include <assert.h>
#include "List.h"

List::List()
{
    fNum = 0;
    fHead = fTail = NULL;
}
```

Figure 21.7 illustrates how the first "append" operation would change a List object. The List object could be a static, an automatic, or a dynamic variable; that doesn't matter. The data item whose address is to be stored is assumed to be a dynamic (heap based) variable. Initially, as shown in part 1 of Figure 21.7, the List object's fNum field would be zero, and the two pointers would be NULL. Part 2 of the figure illustrates the creation of a new ListCell struct in the heap; its fNext member has to be set to NULL and its fData pointer has to hold the address of the data item.

Finally, as shown in part 3, both head and tail pointers of the List object are changed to hold the address of this new ListCell. Both pointers refer to it because as it is the only ListCell it is both the first and the last ListCell. The count field of the List object is incremented.

The code implementing Append() is:

```
void List::Append(void *item)
{
    ListCell *lc = new ListCell;                    Create new ListCell
    lc->fData = item;                               and link to data item
    lc->fNext = NULL;
```

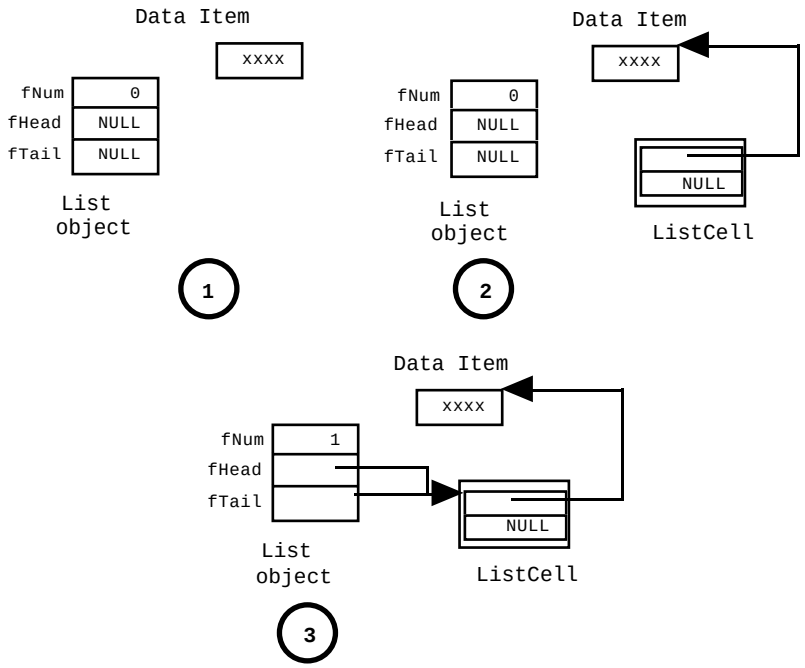

Figure 21.7 The first append operation changing a List object.

Add to an empty list

```
        if(fHead == NULL)
                fHead = fTail = lc;
```

Add to end of an existing list

```
        else {
                fTail->fNext = lc;
                fTail = lc;
                }
```

Update member count

```
        fNum++;
}
```

Figure 21.8 illustrates the steps involved when adding an extra item to the end of an existing list.

Part 1 of Figure 21.8 shows a `List` that already has two ListCells; the `fTail` pointer holds the address of the second cell.

An additional `ListCell` would be created and linked to the data. Then the second cell is linked to this new cell (as shown in Part 2):

```
fTail->fNext = lc;
```

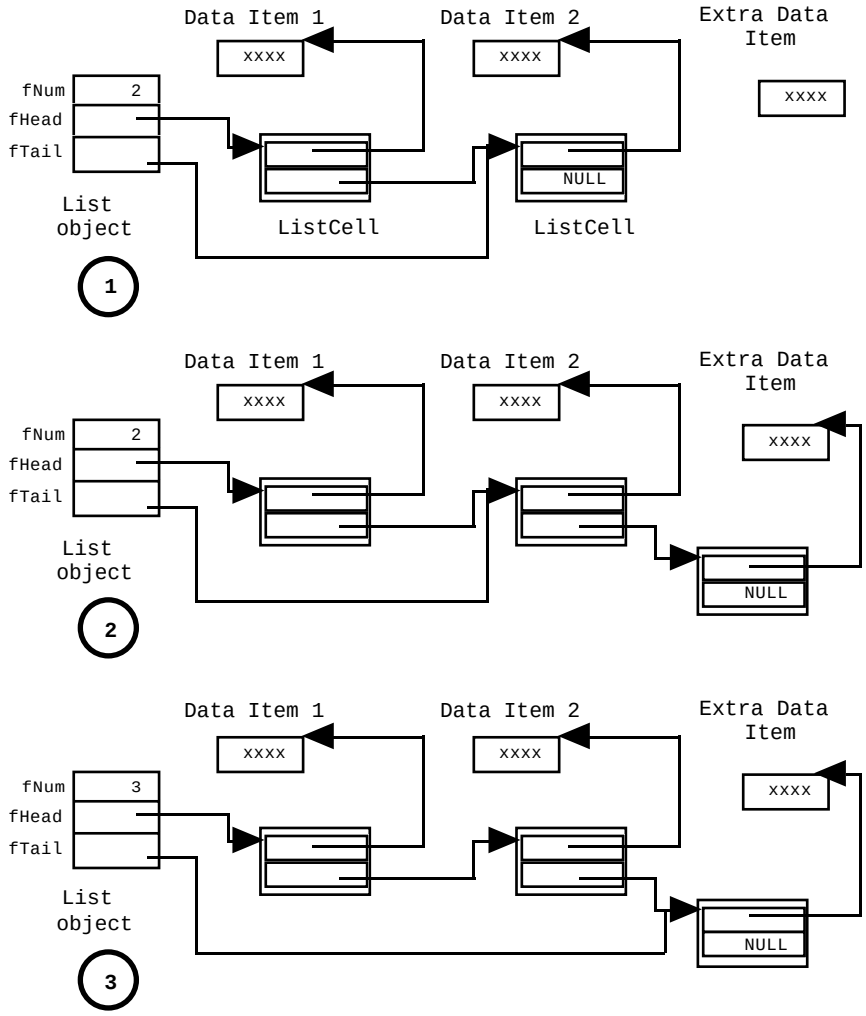

Figure 21.8 Adding an element at the tail of an existing list.

Finally, the `fTail` pointer and `fNum` count are updated as shown in Part 3 of the figure.

The `Position()` function, that finds where a specific data element is stored, involves "walking along the list" starting at the cell identified by the head pointer. The `ListCell*` variable `ptr` is initialized with the address of the first list cell and the position counter `pos` is intialized to 1. The item has been found if its address matches the value in the `fData` member of a `ListCell`; when found, the value for `pos` is returned. If the data address in the current list cell does not match, `ptr` is changed so

that it holds the address of the next list cell. The item will either be found, or the end of the list will be reached. The last list cell in the list will have the value `NULL` in its `fNext` link, this gets assigned to `ptr`. The test in the `while` clause will then terminate the loop. As with the `DynamicArray`, a return value of 0 indicates that the sought item is not present in the list.

```
int List::Position(void *item) const
{
    ListCell *ptr = fHead;
    int pos = 1;
    while(ptr != NULL) {
           if(ptr->fData == item) return pos;
           pos++;
           ptr = ptr->fNext;
           }
    return 0;
}
```

The function `Nth()` is left as an exercise. It also has a `while` loop that involves walking along the list via the `fNext` links. This `while` loop is controlled by a counter, and terminates when the desired position is reached.

Removing an item from the middle of a list

The basic principles for removing an item from the list are illustrated in Figure 21.9. The `ListCell` that is associated with the data item has first to be found. This is done by walking along the list, in much the same way as was done the `Position()` function. A `ListCell*` pointer, `tmp`, is set to point to this `ListCell`.

There is a slight complication. You need a pointer to the previous `ListCell` as well. If you are using a "doubly linked list" where the `ListCells` have "previous" as well as "next" links then getting the previous `ListCell` is easy. Once you have found the `ListCell` that is to be unhooked from the list, you use its "previous" link to get the `ListCell` before it. If you only have a singly linked list, then its slightly more complex. The loop that walks along the list must update a "previous" pointer as well as a pointer to the "current" `ListCell`

Once you have got pointer `tmp` to point to the `ListCell` that is to be removed, and prev to point to the previous `ListCell`, then unhooking the `ListCell` from the list is easy, you simply update `prev`'s `fNext` link:

```
prev->fNext = tmp->fNext;
```

Parts 1 and 2 of Figure 21.9 shows the situation before and after the resetting of `prev`'s `fNext` link.

When you have finished unhooking the `ListCell` that is to be removed, you still have to tidy up. The unhooked `ListCell` has to be deleted and a pointer to the associated data item has to be returned as the result of the function.

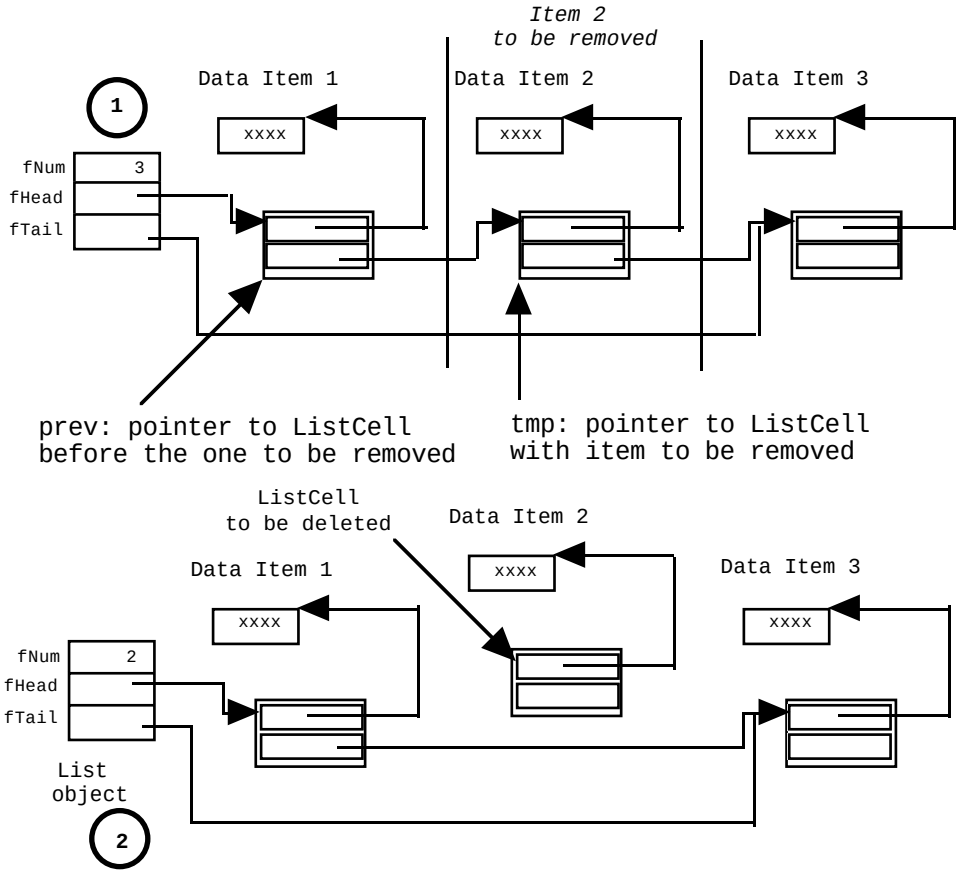

Figure 21.9 Removing an item from the middle of a list.

Complications at the beginning and end of a list

A `Remove()` function has to check for special cases where the `ListCell` that gets removed is the first, or the last (or the only) `ListCell` in the list. Such cases are special because they make it necessary to update the `List`'s `fHead` and/or `fTail` pointers.

Removing the first entry

If the `ListCell` is the first in the list (`fHead == tmp`) then rather than "unhooking" it, the `fHead` pointer can just be set to hold the address of the next `ListCell`:

```
if(fHead == tmp)
    fHead = tmp->fNext;
```

Removing the last entry

If the `ListCell` should happen to be the last one in the current list (`fTail == tmp`), the `List`'s `fTail` pointer needs to be moved back to point to the previous entry:

```
if(fTail == tmp)
```

```
    fTail = prev;
```

The two versions of `Remove()` are similar. The functions start with the loop that searches for the appropriate `ListCell` (and deal with the `ListCell` not being present, a `List` may get asked to remove an item that isn't present). Then there is code for unhooking the `ListCell` and resetting `fHead` and `fTail` pointers as required. Finally, there is code that tidies up by deleting the discarded `ListCell` and returning a pointer to the associated data item.

The following code illustrates the version that finds an item by matching its address. The version removing the item at a specified position is left as another exercise.

```
void *List::Remove(void *item)
{
    ListCell *tmp = fHead;
    ListCell *prev = NULL;

    while(tmp != NULL) {
           if(tmp->fData == item) break;
           prev = tmp;
           tmp = tmp->fNext;
           }

    if(tmp == NULL)
           return NULL;

    if(fHead == tmp)
           fHead = tmp->fNext;

    else prev->fNext = tmp->fNext;

    if(fTail == tmp)
           fTail = prev;

    fNum--;
    void *dataptr = tmp->fData;
    delete tmp;

    return dataptr;
}
```

Loop to find the right ListCell

Item to be removed was not present anyway

ListCell is first

ListCell in middle

Fix up if last ListCell

Tidy up

Go through the code and convince yourself that it does deal correctly with the case of a `List` that only has one `ListCell`.

Test program

You've already seen the test program! Apart from two words, it is identical to the test program for class `DynamicArray`. One word that gets changed is the definition of the datatype for the collection. This has to be changed from

```
DynamicArray      c1;
```

to

```
List              c1;
```

The other word to change is the name of the header file that has to be #included; this changes from "D.h" to "List.h".

The test program has to be the same. Lists and dynamic arrays are simply different implementations of the same "collection" abstraction.

Overheads with the `List` implementation are significantly higher. If you just look at the code, the overheads may not be obvious. The overheads are the extra space required for the `ListCells` and the time required for calls to `new` and `delete`.

With the `ListCells`, a `List` is in effect using two pointers for each data item while the `DynamicArray` managed with one. Further, each `ListCell` incurs the space overhead of the storage manager's header and trailer records. In the `DynamicArray`, the array allocated in the heap has a header an trailer, but the cost of these is spread out over the ten or more pointers in that array.

The two implementations have similar costs for their `Position()` functions. The `Append()` for a `List` always involves a call to `new`, whereas this is relatively rare for a `DynamicArray`. A `DynamicArray` has to reshuffle more data during a `Remove()`; but the loop moving pointers up one place in an array is not particularly costly. Further, every (successful) `Remove()` operation on a `List` involves a call to `delete` while `delete` operations are rare for a `DynamicArray`. The difference in the pattern of calls to `new` and `delete` will mean that, in most cases, processing as well as space costs favour the `DynamicArray` implementation of a collection.

Although `DynamicArrays` have some advantages, `Lists` are more generally used and there are numerous special forms of list for specific applications. You had better get used to working with them.

21.5 CLASS BINARYTREE

Just as there are many different kinds of specialized list, there are many "binary trees". The binary tree illustrated in this section is one of the more common, and also one of the simplest. It is a "binary search tree".

When to use a binary search tree

A "binary search tree" is useful when:

- you have data items (structs or instances of some class) that are characterized by unique "key" values, e.g. the data item is a "driver licence record" with the key being the driver licence number;

- you need to maintain a collection of these items, the size of the collection can not be readily fixed in advanced and may be quite large (but not so large that the collection can't fit in main memory);
- items are frequently added to the collection;
- items are frequently removed from the collection;
- the collection is even more frequently searched to find an item corresponding to a given key.

The "keys" are most often just integer values, but they can be things like strings. The requirement that keys be unique can be relaxed, but this complicates searches and removal operations. If the keys are not unique, the search mechanism has to be made more elaborate so that it can deal in turn with each data item whose key matches a given search key. For simplicity, the rest of this section assumes that the keys are just long integers and that they will be unique.

Alternatives: a simple array? Bit slow.

You could just use an array to store such data items (or a dynamic array so as to avoid problems associated with not knowing the required array size). Addition of items would be easy, but searches for items and tidying up operations after removal of items would be "slow" (run times would be O(N) with N the number of items in the collection). The search would be O(N) because the items wouldn't be added in any particular order and so to find the one with a particular key you would have to start with the first and check each in turn. (You could make the search O(lgN) by keeping the data items sorted by their keys, but that would make your insertion costs go up because of the need to shuffle the data.) Removal of an item would require the movement of all subsequent pointers up one place in the array, and again would have a run time O(N).

Alternatives: a hash table?? Removals too hard.

If you simply needed to add items, and then search for them, you could use a hash table. (Do you still remember hash tables from Chapter 18?) A "search" of a hash table is fast, ideally it is a single operation because the key determines the location of the data item in the table (so, ideally, search is O(1)). The keys associated with the data items would be reduced modulo the size of the hash table to get their insertion point. As explained in Chapter 18, key collisions would mean that some data items would have to be inserted at places other than the location determined simply by their key. It is this factor that makes it difficult to deal with removals. A data item that gets removed may have been one that had caused a key collision and a consequent shift of some other data item. It is rather easy to lose items if you try doing removals on a hash table.

Binary search tree performance

Binary trees offer an alternative where insertions, searches, and removals are all practical and all are relatively low cost. On a good day, a binary tree will give you O(lgN) performance for all operations; the performance does deteriorate to O(N) in adverse circumstances.

A binary search tree gets its O(lgN) performance in much the same way as we got O(lgN) performance with binary search in a sorted array (Chapter 13). The data are going to be organized so that testing a given search key against a value from one of the

data items is going to let you restrict subsequent search steps to either one of two subtrees (subsets of the collection). One subtree will have all the larger keys, the other subtree will have all the smaller keys. If your search key was larger than the key just tested you search the subtree with the large keys, if your key is smaller you search the other subtree. (If your search key was equal to the value just checked, then you've found the data item that you wanted.)

If each such test succeeded in splitting the collection into equal sized subtree , then the search costs would be O(lgN). In most cases, the split results in uneven sized subtree and so performance deteriorates.

Tree structure

Simple "tree-nodes"

A binary search tree is built up from "tree-node" structures. In this section, "tree-nodes" (tn) are defined roughly as follows:

```
struct tn {
    long   key;
    void   *datalink;
    tn     *leftlink;
    tn     *rightlink;
};
```

The tree-node contains a key and a pointer to the associated data record (as in the rest of this chapter, it is assumed that the data records are all dynamic structures located in the heap). Having the key in the tree-node may duplicate information in the associated data record; but it makes coding these initial examples a little simpler. The other two pointers are links to "the subtree with smaller keys" (leftlink) and "the subtree with larger keys" (rightlink). (The implementation code shown later uses a slightly more sophisticated version where the equivalent of a "tree-node" is defined by a class that has member functions and private data members.)

BinaryTree object

The actual binary tree object is responsible for using these `tn` structures to build up an overall structure for the data collection. It also organizes operations like searches. It keeps a pointer, the "root" pointer, to the "root node" of the tree.

Figure 21.10 illustrates a simple binary search tree. The complete tree start starts with a pointer to the "root" tree-node record. The rest of the structure is made up from separate tree-nodes, linked via their "left link" and "right link" pointers.

The key in the root node is essentially arbitrary; it will start by being the key associated with the first data record entered; in the example, the key in the root node is 1745.

All records associated with keys less than 1745 will be in the subtree attached via the root node's "left link". In Figure 21.10, there is only one record associated with a smaller key (key 1642). The tree node with key 1642 has no subtrees, so both its left link and right link are `NULL`.

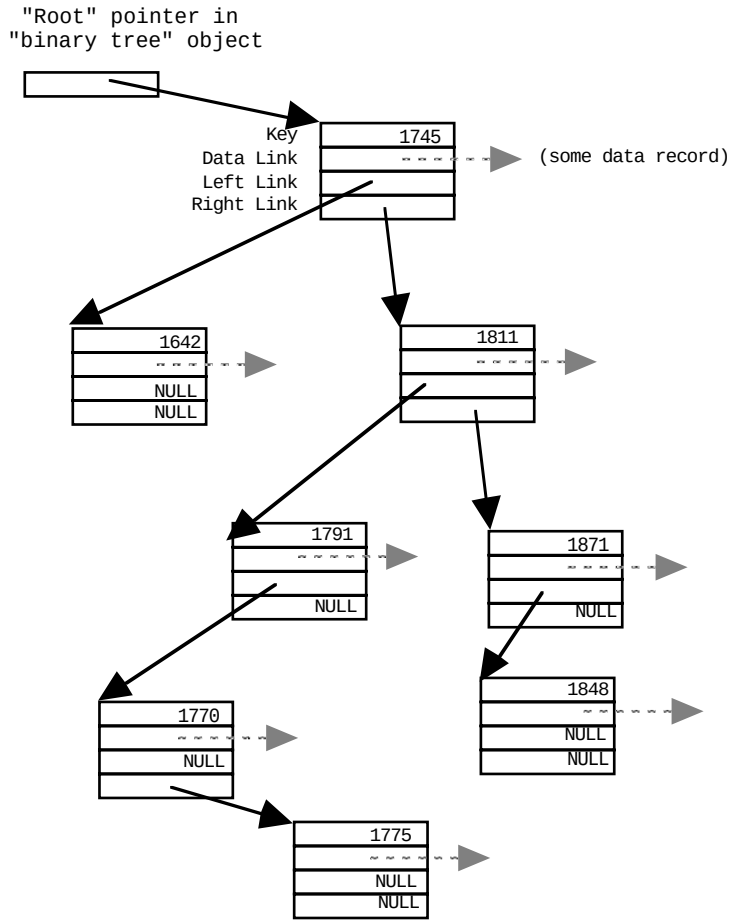

Figure 21.10 A "binary search tree".

All records having keys greater than 1745 must be in the right subtree. The first entered might have been a record with key 1811. The left subtree of the tree node associated with key 1811 will hold all records that are less than 1811 but greater than 1745. Its right subtree will hold all records associated with keys greater than 1811.

The same structuring principles apply "recursively" at every level in the tree.

The shape of the tree is determined mainly by the order in which data records are added. If the data records are added in some "random" order, the tree will be evenly balanced. However, it is more usual for the addition process to be somewhat orderly, for example records entered later might tend to have larger keys than those entered earlier. Any degree of order in the sequence of addition results in an unbalanced tree.

Thus, if the data records added later do tend to have larger keys, then the "right subtrees" will tend to be larger than the left subtrees at every level.

Searching

You can find a data record associated with a given "search key" (or determine that there is no record with that key) by a simple recursive procedure. This procedure takes as arguments a pointer to a subtree and the desired key value:

Pseudo-code of a recursive search function

```
recursive_search(tn *sub_tree_ptr,  Key search_key)
    if sub_tree_ptr is NULL
           return NULL;

    compare_result  = Compare search_key and sub_tree_ptr->Key

    if(compare_result is EQUAL)
           return sub_tree_ptr->Data;

    else
    if(compare_result is LESS)
           return recursive_search(
                  sub_tree_ptr->left_link, search_key)
    else
           return recursive_search(
                  sub_tree_ptr->right_link, search_key)
```

This function would be called with the "root" pointer and the key for the desired record.

Termination test of recursive search function

Like all recursive functions, its code needs to start by checking for termination conditions. One possible terminating condition is that there is no subtree to check! The function is meant to chase down the left/right links until it finds the specified key. But if the key is not present, the function will eventually try to go down a `NULL` link. It is easiest to check this at the start of the function (i.e. at the next level of recursion). If the subtree is `NULL`, the key is not present so the function returns `NULL`.

Comparison of keys

If there is a subtree to check, the search key should be compared with the key in the tree-node record at the "root of the current subtree". If these keys are equal, the desired record has been found and can be returned as the result of the function.

Recursive call to search either the left or the right subtree

Otherwise the function has to be called recursively to search either the left or the right subtree as shown.

Two example searches work as follows:

Program call

Execution at first level of recursion

```
recursive_search(root, 1770)    recursive_search(root, 1795)

sub_tree_ptr != NULL            sub_tree_ptr != NULL
compare 1770, 1745              compare 1795, 1745
result is GREATER               result is GREATER
recursive_search(               recursive_search(
```

```
                    right_subtree, 1770)          right_subtree, 1795)
```

Execution at second level of recursion

```
sub_tree_ptr != NULL          sub_tree_ptr != NULL
compare 1770, 1811            compare 1795, 1811
result is LESS                result is LESS
recursive_search(             recursive_search(
       left_subtree, 1770)           left_subtree, 1795)
```

Execution at third level of recursion

```
sub_tree_ptr != NULL          sub_tree_ptr != NULL
compare 1770, 1791            compare 1795, 1791
result is LESS                result is GREATER
recursive_search(             recursive_search(
       left_subtree, 1770)           right_subtree, 1795)
```

Execution at fourth level of recursion

```
sub_tree_ptr != NULL          sub_tree_ptr == NULL
compare 1770, 1770            return NULL
result is EQUALS
return data for key 1770
```

Addition of a record

The function that adds a new record is somewhat similar to the search function. After all, it has to "search" for the point in the tree structure where the new record can be attached.

Of course, there is a difference. The addition process has to change the tree structure. It actually has to replace one of the `tn*` (pointers to treenode) pointers.

Changing an existing pointer to refer to a new treenode

A pointer has to be changed to hold the address of a new tree node associated with the new data record. It could be the root pointer itself (this would be the case when the first record is added to the tree). More usually it would be either the left link pointer, or the right link pointer of an existing tree-node structure that is already part of the tree.

The recursive function that implements the addition process had better be passed a reference to a pointer. This will allow the pointer to be changed.

The addition process is illustrated in Figure 21.11. The figure illustrates the addition of a new treenode associated with key 1606. The first level of recursion would have the addition function given the root pointer. The key in the record identified by this pointer, 1745, is larger than the key of the record to be inserted (1606), so a recursive call is made passing the left link of the first tree node.

The treenode identified by this pointer has key 1642, again this is larger than the key to be inserted. So a further recursive call is made, this time passing the left link of the treenode with key 1642.

The value in this pointer is `NULL`. The current tree does not contain any records with keys less than 1642. So, this is the pointer whose value needs to be changed. It is to point to the subtree with keys less than 1642, and there is now going to be one – the new treenode associated with key 1606.

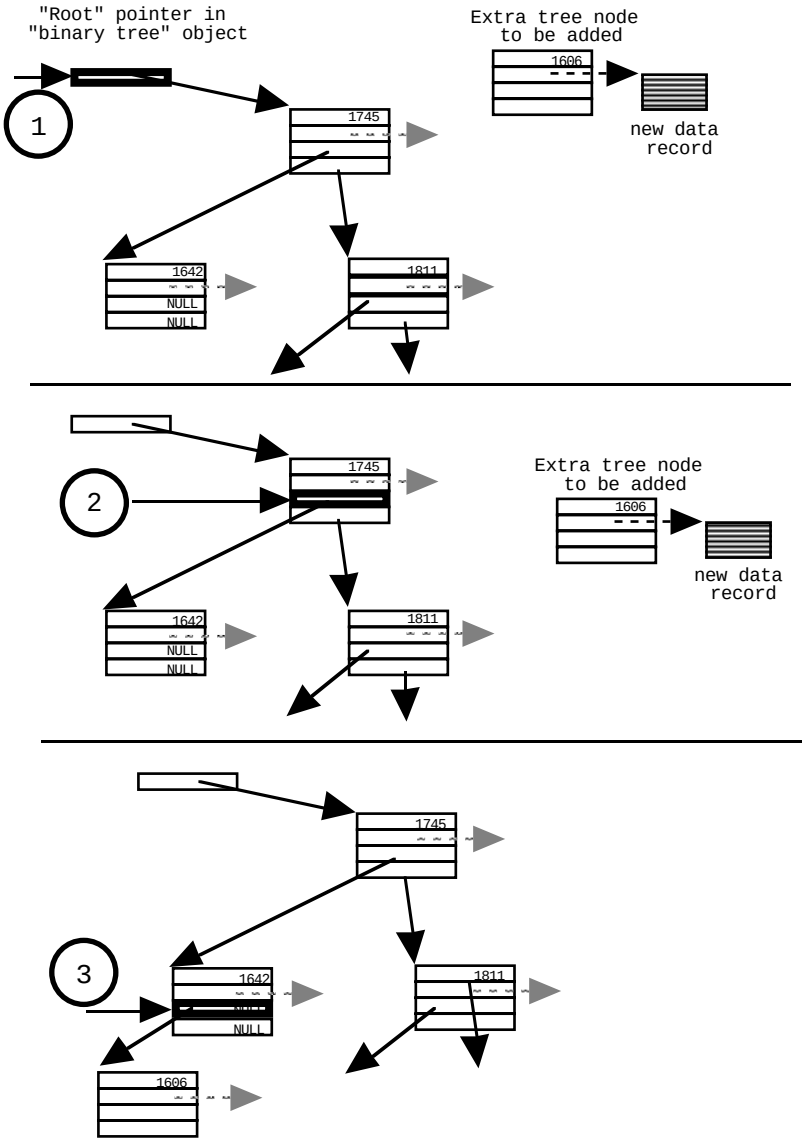

Figure 21.11 Inserting a new record.

A pseudo-code outline for the addition function is:

Pseudo-code of a recursive addition function

```
recursive_add(a treenode pointer passed by reference ('c'),
                    pointer to new data item, key)
    if c contains NULL
```

```
            make a new treenode and fill in its
                   pointer to data and its key
            change contents of pointer 'c' to hold the address
                   of this new treenode
            return

      compare_result  = Compare key and c->Key

      if(compare_result is EQUAL)
            warn user that duplicate keys are not allowed
            return without making any changes

      else
      if(compare_result is LESS)
            return recursive_add(
                   sub_tree_ptr->left_link, data item, key)
      else
            return recursive_add(
                   sub_tree_ptr->right_link, data item, key)
```

Recursion terminates if the treenode pointer argument contains `NULL`. A `NULL` pointer is the one that must be changed; so the function creates the new treenode, fills in its data members, changes the pointer, and returns.

If the pointer argument identifies an existing treenode structure, then its keys must be compared with the one that is to be inserted. If they are equal, you have a "duplicate key"; this should cause some warning message (or maybe should throw an exception). The addition operation should be abandoned.

In other cases, the result of comparing the keys determines whether the new record should be inserted in the left subtree or the right subtree. Appropriate recursive calls are made passing either the left link or the right link as the argument for the next level of recursion.

Removal of a record

Removing a record can be trickier. As shown in Figures 21.12 and 21.13, some cases are relatively easy.

If the record that is to be removed is associated with a "leaf node" in the tree structure, then that leaf can simply be cut away (see Figure 21.12).

If the record is on a vine (only one subtree below its associated treenode) then its treenode can be cut out. The treenode's only subtree can be reattached at the point where the removed node used to be attached (Figure 21.13).

Things get difficult with "internal nodes" that have both left and right subtrees. In the example tree, the treenodes associated with keys 1645 and 1811 are both "internal" nodes having left and right subtrees. Such nodes cannot be cut from the tree because this would have the effect of splitting the tree into two separate parts.

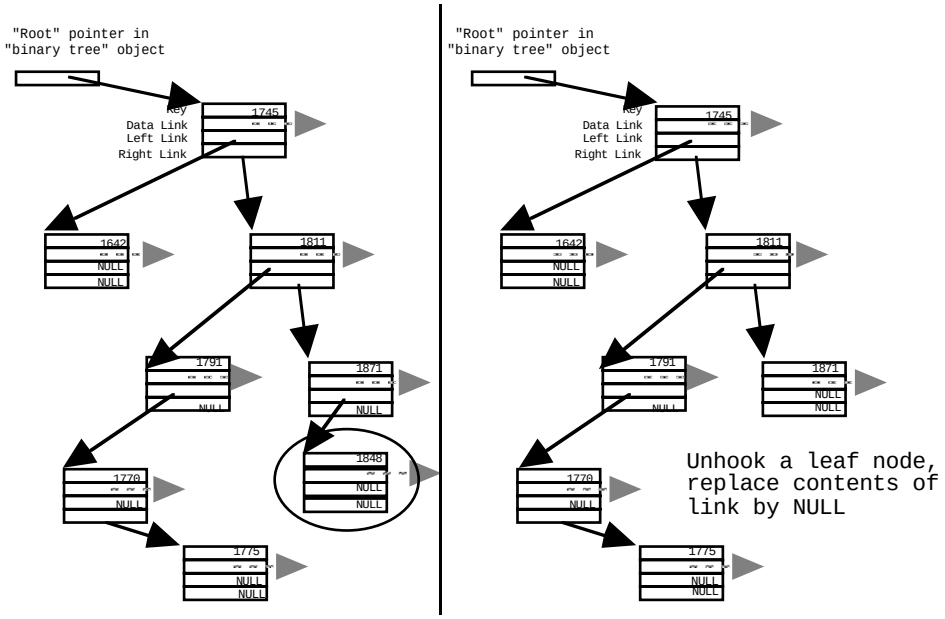

Figure 21.12 Removing a "leaf node".

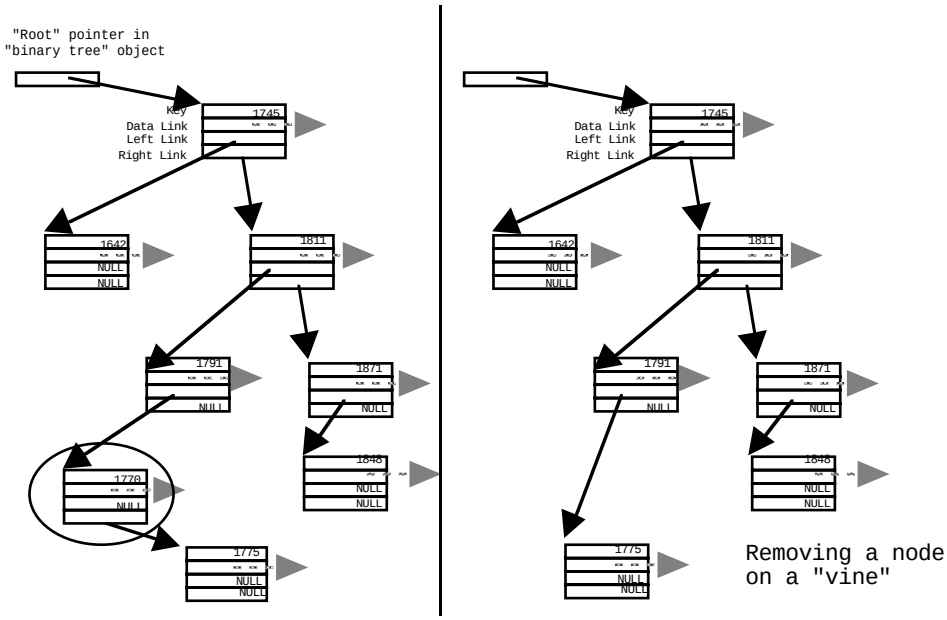

Figure 21.13 Removing a "vine" node.

The key, and pointer to data record, can be removed from a treenode provided that they are replaced by other appropriate data. Only the key and the pointer to data record get changed; the existing left link and right link from the treenode are not to be changed. The replacement key and pointer have to be data that already exist somewhere in the tree. They have to be chosen so that the tree properties are maintained. The replacement key must be such that it is larger than all the keys in the subtree defined by the existing left link, and smaller than all the keys in the subtree defined by the existing right link.

Promoting a "successor"

These requirements implicitly define the key (and associated pointer to data record) that must be used to replace the data that are being removed. They must correspond to that data record's "successor". The successor will be the (key, data record) combination that has the smallest key that is larger than the key being deleted from an internal node.

Find the (key, data) combination to be deleted

Figure 21.14 illustrates the concept of deletion with promotion of a successor. The (key, data) combination that is to be deleted is the entry at the root, the one with key 1745. The key has first to be found; this would be done in the first step of the process.

Find the successor

The next stage involves finding the successor – the (key, data) combination with the smallest key greater than 1745. This will be the combination with key 1770. You find it by starting down the right subtree (going to the treenode with key 1811) and then heading down left links for as far as you can go.

Promoting the successor

The key and pointer to data record must then be copied from the successor treenode. They overwrite the values in the starting treenode, replacing the key and data pointer that are supposed to get deleted. Thus, in the example in Figure 21.14, the "root" treenode gets to hold the key 1770 and a pointer to the data item associated with this key.

Removal of duplicate entry

Of course, key 1770 and its associated data record are now in the tree twice! They exist in the treenode where the deleted (key, data record) combination used to be, and they are still present in their original position. The treenode where they originally occurred must be cut from the tree. If, as in the case shown in Figure 21.14, there is a right subtree, this replaces the "successor treenode". This is shown in Figure 21.14, where the treenode with key 1775 becomes the new "left subtree" of the treenode with key 1791.

If you had to invent such an algorithm for yourself, it might be hard. But these algorithms have been known for the last forty years (and for most of those forty years, increasing numbers of computing science students have been given the task of coding the algorithms again). Coding is not hard, provided the process is broken down into separate functions.

One function will find the (key, data record) combination that is to be deleted; the algorithm used will be similar to that used for both the search and insert operations. Once the (key, data) combination has been found, a second function can sort out whether the deletion involves cutting off a leaf, excising a node on a vine, or replacing the (key, data record) by a successor. Cutting off a leaf and excising a vine node are both straightforward operations that don't require further subroutines. If a successor has to be promoted, it must be found and removed from its original position in the tree.

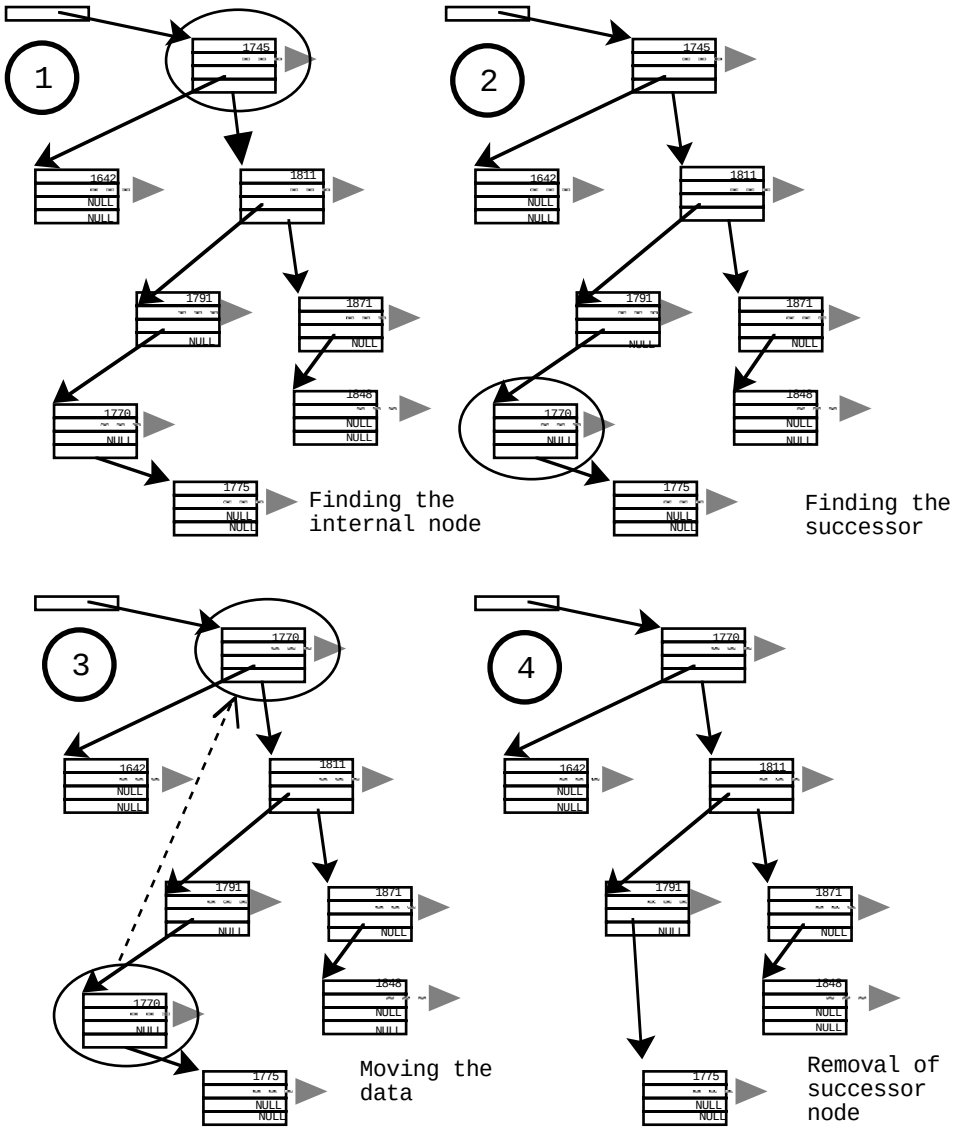

Figure 21.14 Removing a "internal" node and promoting a successor.

Pseudo-code for the initial search step is as follows:

```
recursive_remove(a treenode pointer passed by reference ('c'), key)
    if c contains NULL
            return NULL
```

Pseudo-code of a recursive removal function

```
compare_result  = Compare key and c->Key

if(compare_result is EQUAL)
       return result of Delete function applied 'c'

else
if(compare_result is LESS)
       return recursive_remove(
              sub_tree_ptr->left_link, data item, key)
else
       return recursive_remove(
       sub_tree_ptr->right_link, data item, key)
```

The function should return a pointer to the data record associated with the key that is to be removed (or `NULL` if the key is not found). Like the search function, the remove function has to check whether it has run off the end of the tree looking for a key that isn't present; this is done by the first conditional test.

Passing a pointer by reference

The rest of code is similar to the recursive-add function. Once again, a pointer has to be passed by reference. The actual pointer variable passed as a reference argument could be the root pointer for the tree (if we are deleting the root node of a tree with one or two records); more usually, it will be either the "left link" or "right link" data member of another treenode. The contents of this pointer may get changed. At the stage where the search process is complete, the argument pointer will hold the address of the treenode with the key that is to be removed. If the treenode with the "bad" key is a leaf, the pointer should be changed to `NULL`. If the treenode with the bad key is a vine node, the pointer should be changed to hold the address of that treenode's only child. The actual changes are made in the auxiliary function Delete. But the pointer argument has to be passed by reference to permit changes to its value.

Determining the deletion action

Pseudo code for the Delete function is as follows:

```
delete(a treenode pointer passed by reference ('c'))
    save pointer to data record associated with bad key

    if treenode pointed to by c is a leaf (both subtree
           links being NULL)
                  delete that treenode,
                  set c to NULL
    else
    if treenode pointed to by c is a vine node with only a
           right subtree
                  set c to hold address of right subtree
                  delete discarded treenode;
    else
    similar code for c being a vine node with only a
           left subtree
    else
           get successor (removing it from tree when found)
           replace key and data pointers with values from
```

```
                successor
            delete the successor treenode

    return pointer to data record associated with bad key
```

Most of the code of Delete is made up by the conditional construct that checks for, and deals with the different possible situations.

A pseudo code outline for the function to find the successor treenode and remove it from its current position in the tree is:

```
successor(a treenode pointer passed by reference ('c'))
    if c->treenode has a subtree attached to its left link
            call successor recursively passing left link
    else
            have found successor treenode, unhook it
            c changed to point to treenode's right subtree
            return treenode
```

The BinaryTree class

More detailed outlines of the search, add, and remove operations can be found in numerous text books; many provide Pascal or C implementations. Such implementations have the functions as separate, independent global functions. We need these functions packaged together as part of a BinaryTree class.

What does a BinaryTree object own?

A BinaryTree object doesn't have much data, after all most of the structure representing the tree is made up from treenodes. A BinaryTree object will own a pointer to the "root" treenode; it is useful if it also owns a count of the number of (key, data record) combinations currently stored in the tree.

What does a BinaryTree object do?

A BinaryTree should do the following:

- "Num items" (equivalent to Length of lists etc)
  Report how many items are stored in the collection.
- Add
  Add another (key, data record) combination to the tree, placing it at the appropriate position (the Add operation should return a success/failure code, giving a failure return for a duplicate key).
- Find
  Find the data record associated with a given key, returning a pointer to this record (or `NULL` if the specified key is not present).
- Remove
  Removes the (key, data record) combination associated with a specified key, returning a pointer to the data record (or `NULL` if the specified key is not present).

Because the structure is moderately complex, and keeps getting rearranged, it would be worth including a debugging output function that prints a representation of the structure of the tree.

The discussion of operations like remove identified several additional functions, like the one to find a treenode's successor. These auxiliary functions would be private member functions of the class.

The declaration for the class would be in a header file:

File "biny.h"

```
#ifndef __MYBINY__
#define __MYBINY__

#define DEBUG

class TreeNode;

class BinaryTree
{
public:
    BinaryTree();

    int    NumItems() const;

    int    Add(void* nItem, long key);
    void   *Find(long key);
    void   *Remove(long key);
#ifdef DEBUG
    void   PrintOn(ostream& out) const;
#endif
private:
    int    AuxAdd(TreeNode*& c, void* nItem, long key);
    void   *AuxFind(TreeNode* c, long key);
    void   *AuxRemove(TreeNode*& c, long key);

    void   *Delete(TreeNode*&c);
    TreeNode *Successor(TreeNode*& c);

#ifdef DEBUG
    void   AuxPrint(TreeNode* c, ostream& out, int depth)
const;
#endif
    TreeNode      *fRoot;
    int           fNum;
};

inline int BinaryTree::NumItems(void) const { return fNum; }

#endif
```

Note the declaration:

```
class TreeNode;
```

This needs to appear before class `BinaryTree` to allow specification of function arguments and variables of type `TreeNode*` (pointer to `TreeNode`). The full declaration of class `TreeNode` does not need to appear in the header file. Client's using class `BinaryTree` will know that the tree is built up using auxiliary `TreeNode` objects. However, clients themselves never need to use `TreeNodes`, so the declaration of the `TreeNode` class can be "hidden" in the implementation file of class `BinaryTree`.

Those TreeNode*& arguments

Some of the functions have odd looking arguments:

```
void        *Delete(TreeNode*&c);
```

An expression like `TreeNode*&` is a little unpronounceable, but that really is the hardest part about it. Reference arguments, like `int&`, were illustrated in many earlier examples (starting with the example in section 12.8.3). We use references when we need to pass a variable to a function that may need to change the value in the variable. In this case the variable is of type "pointer to `TreeNode`" or `TreeNode*`. Hence, a "pointer to a `TreeNode` being passed by reference" is `TreeNode*&`.

As is illustrated below in the implementation, the public interface functions like `Find()` do very little work. Almost all the work is done by an associated private member function (e.g. `AuxFind()`). The public function just sets up the initial call, passing the root pointer to the recursive auxiliary function.

As usual, trivial member functions like NumItems() can be defined as inlines. Their definitions go at the end of the header file.

Implementation

The implementation code would be in a separate file, biny.cp.

After the usual #includes, this file would start with the declaration of the auxiliary `TreeNode` class and the definition of its member functions.

```
class TreeNode {
public:
    TreeNode(long k, void *d);
    TreeNode*& LeftLink(void);
    TreeNode*& RightLink(void);
    long Key(void) const;
    void *Data(void) const;
    void Replace(long key, void *d);
private:
    long          fKey;
    void          *fData;
    TreeNode      *fLeft;
    TreeNode      *fRight;
};
```

Class TreeNode is simply a slightly fancy version of the tn treenode struct illustrated earlier. It packages the key, data pointer, and links and provides functions that the BinaryTree code will need to access a TreeNode.

The function definitions would specify most (or all) as "inline":

```
TreeNode::TreeNode(long k,void *d)
{
    fKey = k;
    fData = d;
    fLeft = NULL;
    fRight = NULL;
}

inline long TreeNode::Key(void) const { return fKey; }
inline void *TreeNode::Data(void) const { return fData; }
inline void TreeNode::Replace(long key, void *d)
    { fKey = key; fData = d; }
```

When TreeNodes are created, the key and data link values will be known so these can be set by the constructor. TreeNodes always start as leaves so the left and right links can be set to NULL.

The Key() and Data() functions provide access to the private data. The Replace() function is used when the contents of an existing TreeNode have to be replaced by data values promoted from a successor TreeNode (part of the deletion process explained earlier).

The LeftLink() and RightLink() functions return "references" to the left link and right link pointer data members. They are the first examples that we have had of functions that return a reference data type. Such functions are useful in situations like this where we need in effect to get the address of a data member of a struct or class instance.

```
inline TreeNode*& TreeNode::LeftLink(void) { return fLeft; }
inline TreeNode*& TreeNode::RightLink(void) { return fRight; }
```

The code for class BinaryTree follows the declaration and definition of the auxiliary TreeNode class. The constructor is simple; it has merely to set the root pointer to NULL and zero the count of data records.

```
BinaryTree::BinaryTree()
{
    fRoot = NULL;
    fNum = 0;
}
```

The public functions `Find()`, `Add()`, and `Remove()` simply set up the appropriate recursive mechanism, passing the root pointer to the auxiliary recursive function. Thus, Find() is:

```
void *BinaryTree::Find(long key)
{
    return AuxFind(fRoot, key);
}
```

the other functions are similar in form.

The `AuxFind()` function implements the recursive mechanism exactly as explained earlier:

Search

```
void *BinaryTree::AuxFind(TreeNode* c, long key)
{
    if(c == NULL)
           return NULL;

    int compare = key - c->Key();

    if(compare == 0)
           return c->Data();

    if(compare < 0)
           return AuxFind(c->LeftLink(), key);
    else
           return AuxFind(c->RightLink(), key);
}
```

The `AuxAdd()` function implements the "recursive_add" scheme outlined above, and includes necessary details such as the update of the count of stored data records, and the return of a success or failure indicator:

Addition

```
int BinaryTree::AuxAdd(TreeNode*& c, void* nItem, long key)
{
    if(c==NULL) {
           c = new TreeNode(key, nItem);
           fNum++;
           return 1;
    }

    int compare = key - c->Key();

    if(compare == 0) {
           cout << "Sorry, duplicate keys not allowed" << endl;
           return 0;
           }

    if(compare < 0)
```

```
            return AuxAdd(c->LeftLink(), nItem, key);
        else
            return AuxAdd(c->RightLink(),nItem, key);
    }
```

The following three functions implement the remove mechanism. First there is the `AuxRemove()` function that organizes the search for the `TreeNode` with the "bad" key:

Removal

```
    void *BinaryTree::AuxRemove(TreeNode*& c, long key)
    {
        if(c == NULL)
            return NULL;

        int compare = key - c->Key();

        if(compare == 0)
            return Delete(c);

        if(compare < 0)
            return AuxRemove(c->LeftLink(), key);
        else
            return AuxRemove(c->RightLink(), key);
    }
```

The `Delete()` function identifies the type of deletion action and makes the appropriate rearrangements:

```
    void *BinaryTree::Delete(TreeNode*& c)
    {
        void *deaddata = c->Data();
        if((c->LeftLink() == NULL) && (c->RightLink() == NULL))
                { delete c; c = NULL; }
        else
        if(c->LeftLink() == NULL) {
            TreeNode* temp = c;
            c = c->RightLink();
            delete temp;
            }
        else
        if(c->RightLink() == NULL) {
            TreeNode* temp = c;
            c = c->LeftLink();
            delete temp;
            }
        else {
            TreeNode* temp = Successor(c->RightLink());
            c->Replace(temp->Key(), temp->Data());
            delete temp;
            }
        return deaddata;
```

```
}
```

The `Successor()` function finds and unhooks a successor node:

```
TreeNode *BinaryTree::Successor(TreeNode*& c)
{
    if(c->LeftLink() != NULL)
           return Successor(c->LeftLink());
    else {
           TreeNode *temp = c;
           c = c->RightLink();
           return temp;
    }
}
```

The print function is only intended for debugging purposes. It prints a rough representation of a tree. For example, if given the tree of Figure 21.10, it produces the output:

```
    1848
  1811
    1791
        1775
      1770
1745
  1642
```

With a little imagination, you should be able to see that this does represent the structure of the tree (the less imaginative can check by reference to Figure 21.15).

Given that everything else about this tree structure is recursive, it is inevitable that the printout function is too! The public interface function sets up the recursive process calling the auxiliary print routine to do the work. The root pointer is passed in this initial call.

```
#ifdef DEBUG
void BinaryTree::PrintOn(ostream& out) const
{
    AuxPrint(fRoot, out, 0);
}
```

The (recursive) auxiliary print function takes three arguments – a pointer to treenode to process, an indentation level (and also an output stream on which to print). Like all recursive functions, it starts with a check for a terminating condition; if the function is called with a `NULL` subtree, there is nothing to do and an immediate return can be made.

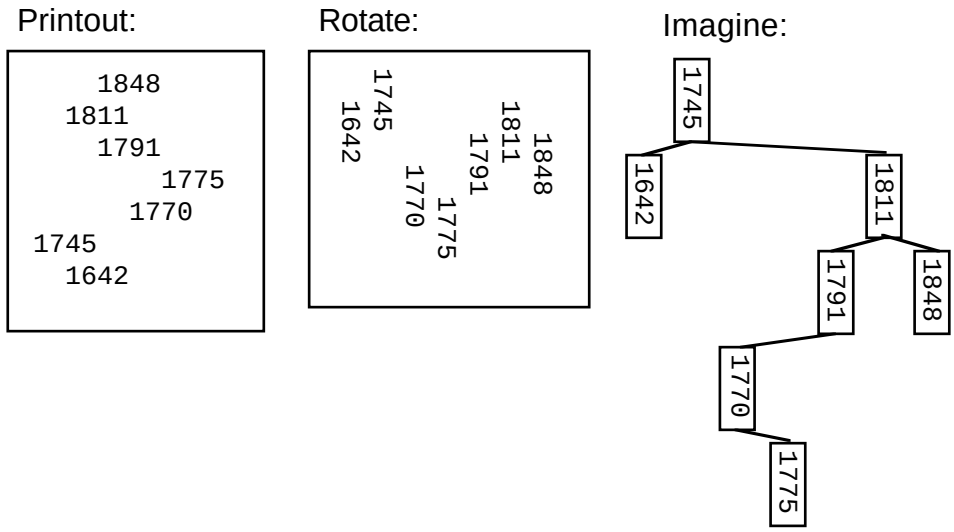

Figure 21.15 Printed representation of binary tree.

```
void BinaryTree::AuxPrint(TreeNode* c, ostream& out,
    int depth) const
{
    if(c == NULL)
           return;

    AuxPrint(c->RightLink(), out, depth + 2);

    for(int i=0;i< depth; i++)
           cout << " ";
    cout << c->Key();
    cout << endl;

    AuxPrint(c->LeftLink(), out, depth + 2);

}
#endif
```

Deal with right subtree

Deal with data in this treenode

Deal with left subtree

If the function has been passed a pointer to a `TreeNode`, processing starts by a recursive call to deal with the *right* subtree (with an increased indentation specified in the `depth` argument). This recursive call will get all higher valued keys printed before the current entry. (The higher valued keys are printed first so that when you rotate the printout they are on the right!) Once the right subtree has been dealt with, the key in the current `TreeNode` is printed (the `for` loop arranges for the appropriate indentation on the line). Finally, the left subtree is processed.

Tree traversal

The recursive `AuxPrint()` function gets to process every node in the tree – it is said to perform a "tree traversal". There are many circumstances where it is useful to visit every node and so traversal functions are generally provided for trees and related data structures. Such functions are described briefly in the section on "Iterators" in Chapter 23.

Test Program

As with the other examples in this chapter, the class is tested using a simple interactive program that allows data records to be added to, searched for, and removed from the tree.

The test program starts with the usual #includes, then defines a trivial `DataItem` class. The tree is going to have to hold some kind of data, class `DataItem` exists just so as to have a simple example. A `DataItem` owns an integer key and a short name string.

```
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include "biny.h"

class DataItem {
public:
    DataItem(long k, char  txt[]);
    void PrintOn(ostream& out) const;
    long K() const;
private:
    long   fK;
    char   fName[20];
};

DataItem::DataItem(long k, char txt[] )
{
    fK = k;
    int len = strlen(txt);
    len = (len > 19) ? 19 : len;
    strncpy(fName, txt, len);
    fName[len] = '\0';
}

void DataItem::PrintOn(ostream& out) const
{
    out << fName << " : " << fK << ";" << endl;
}

inline long DataItem::K() const { return fK; }
```

The actual `BinaryTree` object has been defined as a global. Variable `gTree` is initialized by an implicit call to the constructor, `BinaryTree::BinaryTree()` that gets executed in "preamble" code before entry to `main()`.

```
BinaryTree gTree;
```

The test program uses auxiliary functions to organize insertion and deletion of records, and searches for records with given keys. Insertion involves creation of a new DataItem, initializing it with data values entered by the user. This item is then added to the tree. (Note, the code has a memory leak. Find it and plug it!)

```
void DoInsert()
{
    char buff[100];
    int k;
    cout << "Enter key and name" << endl;
    cin >> k >> buff;
    DataItem *d = new DataItem(k, buff);
    if(gTree.Add(d , k)) cout << "OK" << endl;
    else cout << "Problems" << endl;
}
```

The search function gets a key from the user, and asks the tree to find the associated `DataItem`. If found, a record is asked to print itself:

```
void DoSearch()
{
    cout << "Enter search key : ";
    int k;
    cin >> k;
    DataItem *item = (DataItem*) gTree.Find(k);
    if(item == NULL) cout << "Not found " << endl;
    else {
           cout << "Found record : ";
           item->PrintOn(cout);
           }
}
```

Similarly, the delete function prompts for a key, and asks the tree for the record. If a `DataItem` is returned, it is deleted so recovering the space that it occupied in the heap.

```
void DoDelete()
{
    cout << "Enter key : ";
    int k;
    cin >> k;
    DataItem *item = (DataItem*) gTree.Remove(k);
    if(item == NULL) cout << "Wasn't there" << endl;
```

```
    else {
            cout << "Removed item: " ;
            item->PrintOn(cout);
            cout << "Destroyed" << endl;
            delete item;
            }
}
```

The `main()` function has the usual interactive loop typical of these small test programs:

```
int main()
{
    for(int done = 0; ! done; ) {
            cout << ">";
            char ch;
            cin >> ch;
            switch(ch) {
case 'q':   done = 1; break;
case 's':
                    DoSearch();
                    break;
case 'i':
                    DoInsert();
                    break;
case 'd':
                    DoDelete();
                    break;
case 'p':
                    gTree.PrintOn(cout);
                    break;
case '?':
                    cout << "q to quit, s to Search, i to "
                           "Insert,d to Delete, p Print" << endl;
                    break;
default:
                    ;
                    }
            }
    return EXIT_SUCCESS;
}
```

21.6 COLLECTION CLASS LIBRARIES

You should not need to implement the code for any of the standard collection classes. Your IDE will come with a class library containing most of the more useful collection classes. Actually, it is likely that your IDE comes with several class libraries, more than one of which contains implementations of the frequently used collection classes.

"Standard Template Library"

Your IDE may have these classes coded as "template classes". Templates are introduced in Chapter 25. Templates offer a way of achieving "generic" code that is more sophisticated than the `void*` pointer to data item used in the examples in this chapter. In the long run, templates have advantages; they provide greater flexibility and better type security than the simpler approaches used in this chapter. They have some minor disadvantages; their syntax is more complex and they can complicate the processes of compilation and linking. There is a set of template classes, the "Standard Template Library", that may become part of "standard C++"; these classes would then be available on all implementations. Some of the classes in the Standard Template Library are collection classes.

Your IDE will also provide a "framework class library". A framework class library contains classes that provide prebuilt parts of a "standard Macintosh" program (as in Symantec's TCL2 library) or "standard Windows" program (as with Borland's OWL or Microsoft's MFC). Framework libraries are discussed more in Chapter 31. Your IDE's framework class library contains additional collection classes. These classes will have some restrictions. Normally they can only be used if you are building your entire program on top of the framework.

The IDE may have additional simpler examples of collection classes. They will be similar to, or maybe a little more sophisticated than the examples in this chapter and in Chapter 24. Many other libraries of useful components and collection classes are in the public domain and are available over the Internet.

Use and re-use!

You should not write yet another slightly specialized version of "doubly linked list", or "queue based on list". Instead, when analysing problems you should try to spot opportunities to use existing components.

Read the problem specification. It is going to say either "get and put a file record" (no need for collection classes), or "do something with this block of data" (arrays rather than collection classes), or "get some items from the user and do something with them" (probably a collection class needed).

Once you have established that the problem involves a variable number of data items, then you know that you probably need to use a collection class. Read further. Find how items are added to the collection, and removed. Find whether searches are done on the collections or whether collections of different items are combined in some way.

Once you have identified how the collection is used, you more or less know which collection class you want. If items are frequently added to and removed from the collection and the collection is often searched to find an item, you may be dealing with a case that needs a binary tree. If you are building collections of different items, and then later combining these, then you are probably going to need a list. If much of the processing consists of gathering items then sorting them, you might be able to use a priority queue.

For example, a problem may require you to represent polynomials like $7x^3 + 6x^2 - 4x+5$, and do symbolic arithmetic like multiplying that first polynomial by $4x^2-3x+7$. These polynomials are collections (the items in the collection represent the coefficients

for different powers of x); they vary in size (depending on the person who types in the examples). The collections are combined. This is going to be a case where you need some form of list. You could implement your own; but it is much better to reuse a prebuilt, debugged, possibly optimized version.

Initially, stick to the simple implementations of collection classes like those illustrated in this chapter or the versions in most public domain class libraries. As you gain confidence in programming, start to experiment with the more advanced templates provided by your IDE.

The good programmer is not the one who can rapidly hack out yet another version of a standard bit of code. The good programmer is the one who can build a reliable program by combining small amounts of new application specific code with a variety of prebuilt components.

EXERCISES

1. Implement class Stack.

   A stack owns a collection of data items (void* pointers in this implementation). It supports the following operations:

   Add (preferred name "Push")
   : adds another element to the front of the collection;

   First (preferred name "Pop")
   : removes most recently added element from the collection;

   Length (preferred name "Size")
   : reports number of elements in the collection;

   Full
   : returns true if collection is full and error would result from further Push operation;

   Empty
   : return true if collection is empty and error would result from further Pop operation..

   Your stack can use a fixed size array for storing the void* pointers, or a dynamic array or a list. (Function Full should always return false if a list or dynamic array is used.)

2. Complete the implementation of class List by writing the `Nth()` and `Remove(int pos)` member functions.

3. The "awful warning" exercise.

   Modify the implementation of `List::Remove()` by changing the last part of the code from:

   ```
   fNum--;
   void *dataptr = tmp->fData;
   delete tmp;
   ```

```
    return dataptr;
```

to

```
    fNum--;
    void *dataptr = tmp->fData;
    delete prev;

    return dataptr;
```

Note that this introduces two errors. First, there is a memory leak. The listcell pointed to by `tmp` is being discarded, but as it is not being deleted it remains occupying space in the heap.

There is a much more serious error. The listcell pointed to by `prev` is still in use, it is still part of the list, but this code "deletes it".

Compile and run the modified program. Test it. Are any of the outputs wrong? Does your system "crash" with an address error?

Unless your IDE has an unusually diligent memory manager, the chances are that you will find nothing wrong in your tests of your program. Everything will appear to work.

Things work because the listcells that have been deleted exist as "ghosts" in the heap. Until their space gets reallocated, they still appear to be there. Their space is unlikely to be reallocated during the course of a short test with a few dozen addition, search, and removal operations.

Although you know the code is wrong, you will see it as working.

This should be a warning to all of us who happily say "Ah it's working!" after running a few tests on a piece of code.