Chapter 13 — Heaps, Priority Queues, and Graphs

Introduction

We have just explored the idea of a heap and how a heap can be used to create a fast sorting algorithm, the Heapsort. Heaps have many other uses, so next let's formalize the heap methods and utilize them to create a priority queue. In Beginning Data Structures, you may have designed a priority queue while doing one of the Stop Exercises. In a priority queue, the client's items are stored with some kind of retrieval priority. For example, a veterinarian could use a queue to schedule incoming patients normally on a first come-first handled basis. However, emergency cases can arrive which require immediate care — a "go to the head of the queue" type operation. Thus, when a new patient arrives at the veterinarian's clinic, they are added to a priority queue based upon their level of urgency. We will implement a priority queue class by using a heap approach.

The last, and most important topic of this chapter are that of graphs. Probably, when you hear the word "graphs" you immediately think of a two-dimensional graph with an x and y axis showing a plot of some kind. In data structures, a graph has a different meaning. Recall that in a binary tree, any given node has only one other node pointing to it, its parent. Further, any given node can point to, at most, two child nodes below it. However, we have seen that the general tree removes this last requirement; in a general tree, a node can have as many children as needed. Finally, if we also remove the idea that any given node can have only one parent pointing to it, we have the **graph data structure**.

In a graph, the nodes are called **vertices**. And the lines that connect the vertices (nodes) are called either **edges** or **arcs**. Further, the edges or lines connecting the vertices (nodes) can have some kind of weight or importance or significance attached to them. Additionally, each edge can have a direction associated with it. For example, examine an airline's flying schedule between cities. Between any given two cities, flights may go in both directions or maybe only from city A to city B and not from city B back to city A. This shows the idea of direction associated with the edges. The weight is likely the air distance between the connected cities (vertices). And this gets us to the importance of the graph data structure. Now we can answer questions such as "Can I fly from city A to city B?" "Among all the flights, what is the shortest route to take to get from city A to city B?"

Here is another example. Suppose that you want to plan a vacation from Peoria. You've decided to visit ten parks scattered around the country. In what order do you visit the parks? If you just travel from park to park in a random fashion, you may end up spending the whole vacation driving from place to place, from one side of the country to the other, back and forth. So you might wish to determine the order of visiting based on the least amount of driving time. Here the parks and Peoria represent the vertices (nodes), the routes between them are the edges, and the weight of each edge is the number of miles separating the two places. This is a graph. And we can write a simple program that outputs the order that we need to take to visit the park that involves the least number of miles driven.

Heaps

Let's review what we know about heaps from the Trees chapter and last chapter's Heapsort discussion. A **heap** is a complete binary tree in which the data stored in its nodes is arranged such that no child has a larger value than its parent. A complete binary tree is either full or full down to the next-to-the-last level with all of the leaves of the last level as far to the left as possible. Figure 13.1 shows a heap tree. Notice that it is a binary tree but not a searchable binary tree since all of the right child nodes are not less than the parent node and all of the left child nodes are not greater than the parent.

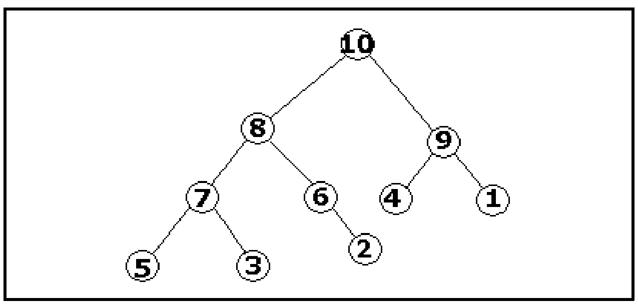


Figure 13.1 A Heap

Technically, a heap is a binary tree which is a complete binary tree and for every node in the heap, the value stored in that node is greater than or equal to the values stored in its children. This gives us the very useful property that the largest value is always stored in the root node!

Very often, an algorithm desires the maximum value. Here it is at a known location, the root node. So if we remove that maximum value, we are then left with a hole at the top of the tree. And the heap must be rebuilt. We have already seen how that can be done with Heapsort.

Recall that the Heapsort pretends the original array is a b-tree that is out of order. Figure 13.2 shows the initial array as if it were a heap tree. Of course, in the unsorted array, the nodes are out of order. That is, all values on the right side of a node are not all greater than that node's value while all nodes on the left side of a node are not all less than that node's value. Heapsort must then perform a "rebuild the heap downward" action to get the nodes in their proper order. The proper order is dictated by **array[node] >= array[node*2+1]** for the left side and **array[node] >= array[node*2+2]** for the right side. The action consists of going down each node and moving the elements around such that all the nodes on the right side of a given node are greater than the node and similarly with the left side. It begins at the top node and works its way to the bottom.

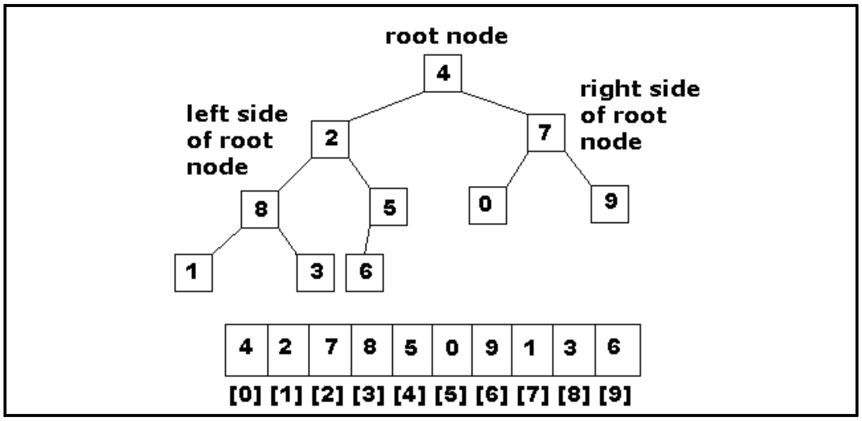


Figure 13.2 The Original Array to Be Sorted Viewed as a b-tree

Returning to the situation when the application has removed the maximum value, which is the root, we have a hole at the top as shown in Figure 13.3.

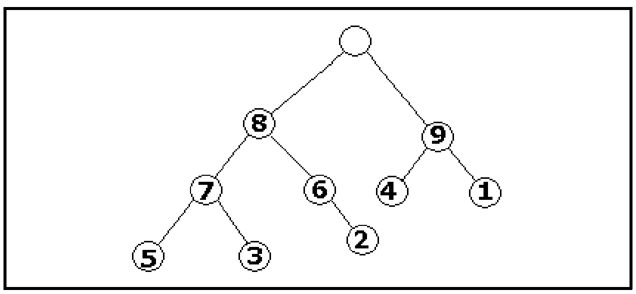


Figure 13.3 The Heap with Root Node Removed

Next, the heap must be restructured as a binary tree. To replace the value in the empty root node, remove and use the value in the rightmost node at the lowest depth or height of the tree. In this case, it is the node containing the value of 2 since it is the rightmost node of nodes 5, 3, and 2. This yields the tree shown in Figure 13.4.

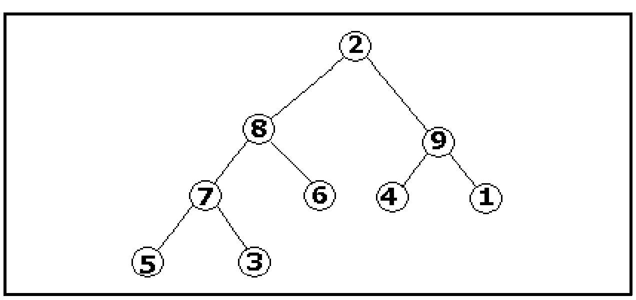


Figure 13.4 The Heap Tree with the New Root

Of course, the tree now does not satisfy the requirement that no child has a value greater than its parent. Obviously the two nodes containing the 8 and 9 are greater than the parent root node. Now a process called **Reheapify** or **RebuildHeapDownwards** must be done. This process consists of starting at the root and repeatedly exchanging its value with the larger value of its children until no more exchanges are possible. The reheapify process first compares the 2 to its two children, the 8 and 9, choosing the larger value, the 9. The 9 replaces the 2 and we get the results shown in Figure 13.5

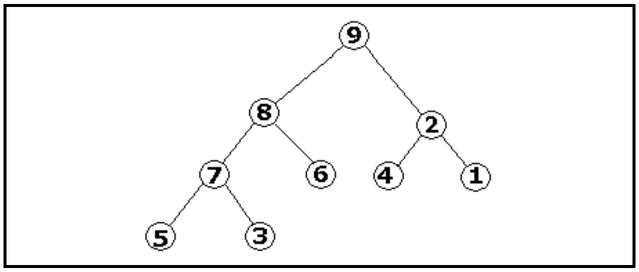


Figure 13.5 The Heap Tree After One Swap

We saw that the process must be recursive since now the node containing the value of 2 is not proper for a heap. So beginning with the new node containing the value 2, we find which of its children contain the larger value and swap once more. This yields the final reheapified tree shown in Figure 13.6.

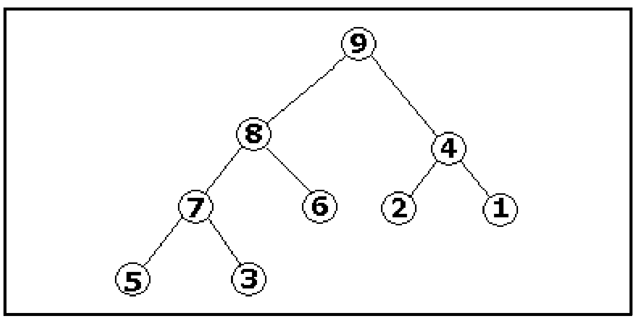


Figure 13.6 The Heap Tree After Reheapify

The **RebuildHeapDownwards** function from the Heapsort is passed the current root node. It compares the two leaves below it to find which one is the greater value and whose index is then stored in **maxChild**. If that found largest value is greater than the root's value, it swaps that **maxChild**'s value with the root's value. Then, it recursively calls itself using the index of **maxChild** as the next downward node.

```
void RebuildHeapDownward (int array[], long root, long bottom) {
 int temp;
 long maxChild;
 long leftChild  = root * 2 + 1;
 long rightChild = root * 2 + 2;
 if (leftChild <= bottom) {
  if (leftChild == bottom)
   maxChild = leftChild;
  else {
   if (array[leftChild] <= array[rightChild])
    maxChild = rightChild;
```

```
   else
    maxChild = leftChild;
  }
  if (array[root] < array[maxChild]) {
   temp = array[root];
   array[root] = array[maxChild];
   array[maxChild] = temp;
   RebuildHeapDownward (array, maxChild, bottom);
  }
 }
}
```

However, rebuilding the heap downward is only one half of the general problem. The other situation we must handle is how to insert a new item into the heap. Of course, this does not occur when sorting. If we want to add a new item to the heap, where do we place it? Because the tree must be a complete tree, we have no choice but to add that item at the bottom rightmost location in the tree. Remember that a complete binary tree is either full or full down to the next-to-the-last level with all of the leaves of the last level as far to the left as possible.

Suppose that we wish to add item 10 back into the heap. Figure 13.7 shows where we must insert it.

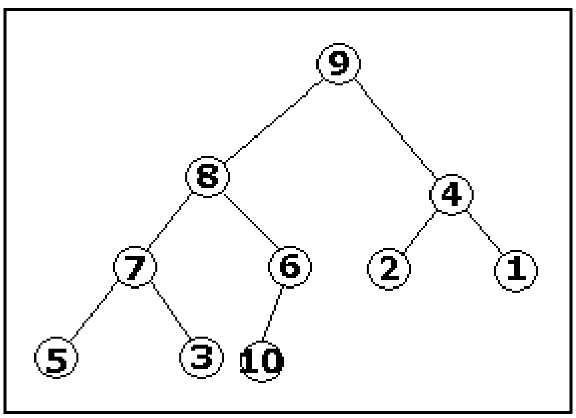


Figure 13.7 Inserting Item 10 into the Heap

Now the heap meets the first criteria, a complete binary tree, but it fails the second: for every node in the heap, the value stored in that node is greater than or equal to the values stored in its children. The 6 is not greater than the 10. Now we must rebuild the heap upwards to get the 10 where it belongs, at the root. The function is much simpler than the downward operation. First, for the node we are at, we must find our parent in order to compare our value to our parents and swap them if needed. Again the function is passed the root and the bottom indexes. The parent is given by (bottom – 1) /2.

```
void RebuildHeapUpward (int array[], long root, long bottom) {
 int temp;
 long parentNode;
 if (bottom <= root) return;
```

```
 parent = (bottom - 1) / 2;
 if (array[parent] < array[bottom]) {
  temp = array[parent];
  array[parent] = array[bottom];
  array[bottom] = temp;
  RebuildHeapUpward (array, root, parent);
 }
}
```

Implementation of a Heap

Thus, a heap has these two basic operations, rebuilding upwards or downwards. Now the question becomes how do we implement a heap in general? Do we make it a class or leave it as stand alone functions? How do we deal with the array of items? The last question is more readily answered. In the Heapsort, we just passed the array of integers to be sorted and the number in that array. Certainly, we must generalize this approach.

Could we pass a **void*** array of items? Yes, we could, but if we did so, we would force users to have to provide a callback function to perform the comparisons. Further, we must swap items in the array. Thus, we must also be passing the total size of the items so that we could dynamically allocate the temp area and use the **memcopy** function to perform the actual movement of data. If this is beginning to sound complicated to you, it should. We have reached a threshold of complexity at which storing generic **void*** to the user's data is no longer viable. The heap coding must know the data type of the user's data. Here is the first time that using templates really offers us great value. Our heap solution must be a template operation.

Do we make this a class or leave it as stand alone functions, perhaps as part of a structure? If we make it into a class, then other ADTs can derive from us and inherit our methods. However, if we do so, we must consider what additional operations a user might desire in their derived classes and provide virtual functions for them. If we fail to do so, then the client programs cannot use a base class pointer to invoke derived class functions. These considerations are best summarized by saying that the heap is really a fundamental building block for other ADTs and not really a stand-alone entity in and of itself. Thus, some designers choose to implement the heap as a structure which contains the dynamically allocated array of user items and the number of elements in that array along with the two heap operation functions. Functions can be members of a structure. All structure member functions have public access to all other structure members. And all members, whether data or methods, have public access to clients.

Thus, there is a strong argument for implementing our heap as a structure with the two heap functions as structure member methods. Other ADTs would then create an instance of the heap structure as one of its data members and directly manipulate and invoke the heap methods.

However, from an educational viewpoint, I think that illustrating how a **Heap** template class can be written is also useful, particularly later on when other ADTs wish to make use of it by creating instances of the **Heap** class or deriving from it. So here, we will embark on the construction of a **Heap** template class.

Let’s assume that the user data is to be called **type T** as is usual with templates. The **Heap** class would contain then a dynamically allocated array of **type T** and a count of the number of elements in that array. Notice that it is not containing pointers to the user’s data of **type T** but an actual instance of that type. This removes the burden on the user from allocating and deleting these instances.

But normally, the heap views the items as being in an array. Thus, we can go two ways. One is to begin with an empty array and provide functions to grow the array — that is, take the growable array approach. When there are many items to be added, this can be time consuming unless we store pointers to the user’s data. A more restrictive approach is to have the constructor be passed the maximum array size and pre build the array that size but set the number of items in the heap to 0. Then, let the user add items to the heap, incrementing the count until the maximum array size is reached. Let’s use this more restrictive approach because it is much easier to implement.

Next, consider how the user is to access items in the heap array itself. If we make the actual array protected, then we must also provide the requisite access functions and so on. With this particular class, it is going to be more difficult to predict the demands that client programs are going to make of it. If we cannot foresee what our client’s will likely need in the way of access operations, later revisions of the class are inherent. Thus, here is a situation in which giving the array of items and the number of items currently in use public access. This way, the clients can access the array directly. Our heap class will make no attempt to maintain the heap order at all times. That is, if the user adds a new item to the heap, it is their responsibility to call the reheap building functions. This “gets us off the hook” so to speak. We construct and destroy the actual array and, when called, rebuild the heap. But the clients must handle inserting and removing elements from the array, subject to their verifying that they are not exceeding the maximum array size. The Heap class is then a skeletal class only.

It should have a constructor, but I default the maximum size so that the function can serve as the default ctor as well. The destructor is virtual in case of derivations. I provide simple access functions for the number of elements and current array size strictly for the convenience of the user. We need the two rebuild functions. But then I added some extra functions.

SortHeap will sort an unsorted array. **GrowHeapBy** dynamically allocates a larger heap and copies existing items onto the new heap before deleting the original heap. And I added support for deep copies by providing the copy ctor and assignment operator.

What kind of user items can be placed in this **Heap**? Any item can be used as long as it provides support for two operators: the assignment operator and the less than relational operator. Since **Heap** is not storing pointers to the user's objects, it must have a way to assign them. Further, the rebuild heap functions require the ability for a less-than comparison operator.

Here is the **Heap** template class. Notice how simple it is to implement this template class.

```
+))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Heap Template Class                                                      *
/)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
*  #ifndef HEAP_H                                                          *
*  #define HEAP_H                                                          *
*                                                                          *
*  /*************************************************************/*
*  /*                                                            */*
*  /* Heap: a class to encapsulate a heap                        */*
*  /*                                                            */*
*  /* user class must provide operator = and < and <=            */*
*  /*                                                            */*
*  /*************************************************************/*
*                                                                          *
*  template<class UserData>                                                *
*  class Heap {                                                            *
*  public:                                                                 *
*   UserData*   array;        // the array of user items                  *
*   long numElements;  // current number of user items                    *
*   long maxSize;      // maximum size of the array                       *
*                                                                          *
*           Heap (long max = 100);                                         *
*  virtual ~Heap ();                                                       *
*  bool     IsArrayFull () const;     // true if numElements=maxSize*
*  bool     IsArrayEmpty () const;    // true if numElements=0      *
*                                                                          *
*  long     GetNumElements () const;                                       *
*  long     GetMaxHeapSize () const;                                       *
*                                                                          *
*  void     EmptyHeap ();           // sets number of elements to 0 *
*                                                                          *
*  void     RebuildHeapUpward (long root, long bottom);                    *
*  void     RebuildHeapDownward (long root, long bottom);                  *
*                                                                          *
*  void     SortHeap ();              // sorts the heap                    *
*  bool     GrowHeapBy (long growby); // grow the array size               *
*                                                                          *
*           Heap (const Heap<UserData>& h);                                *
*  virtual Heap<UserData>& operator= (const Heap<UserData>& h);            *
*  protected:                                                              *
*  void     Copy (const Heap<UserData>& h); // make a duplicate Heap*
*  };                                                                      *
*                                                                          *
*  /*************************************************************/*
*  /*                                                            */*
```

```
/* Heap: allocate the empty max sized array of user items      */
/*                                                              */
/***************************************************************/

template<class UserData>
Heap<UserData>::Heap (long max) {
 numElements = 0;
 maxSize = max > 0 ? max : 100;
 try {
  array = new UserData [maxSize];
 }
 catch (std::bad_alloc e) {
  // check if out of memory
  cerr << "Error: out of memory\n";
 }
}

/***************************************************************/
/*                                                              */
/* ~Heap: delete the array of user items                        */
/*                                                              */
/***************************************************************/

template<class UserData>
Heap<UserData>::~Heap () {
 delete [] array;
}

/***************************************************************/
/*                                                              */
/* IsArrayFull: returns true if the array is full               */
/*                                                              */
/***************************************************************/

template<class UserData>
bool Heap<UserData>::IsArrayFull () const {
 return numElements == maxSize;
}

/***************************************************************/
/*                                                              */
/* IsArrayEmpty: returns true if the array is empty             */
/*                                                              */
/***************************************************************/

template<class UserData>
bool Heap<UserData>::IsArrayEmpty () const {
 return numElements == 0;
}

/***************************************************************/
/*                                                              */
```

```
/* EmptyHeap: empties heap by resetting numElements to 0       */*
/*                                                              */*
/****************************************************************/*
                                                                  *
template<class UserData>                                          *
void Heap<UserData>::EmptyHeap () {                               *
 numElements = 0;                                                 *
}                                                                 *
                                                                  *
/****************************************************************/*
/*                                                              */*
/* GetNumElements: returns numElements                          */*
/*                                                              */*
/****************************************************************/*
                                                                  *
template<class UserData>                                          *
long Heap<UserData>::GetNumElements () const {                    *
 return numElements;                                              *
}                                                                 *
                                                                  *
/****************************************************************/*
/*                                                              */*
/* GetMaxHeapSize: returns the max array size                   */*
/*                                                              */*
/****************************************************************/*
                                                                  *
template<class UserData>                                          *
long Heap<UserData>::GetMaxHeapSize () const {                    *
 return maxSize;                                                  *
}                                                                 *
                                                                  *
/****************************************************************/*
/*                                                              */*
/* RebuildHeapDownward: rebuilds heap when top is bad           */*
/*                                                              */*
/****************************************************************/*
                                                                  *
template<class UserData>                                          *
void Heap<UserData>::RebuildHeapDownward (long root,              *
                                          long bottom) {          *
 UserData temp;                                                   *
 long maxChild;                                                   *
 long leftChild  = root * 2 + 1;                                  *
 long rightChild = root * 2 + 2;                                  *
 if (leftChild <= bottom) {                                       *
  if (leftChild == bottom)                                        *
   maxChild = leftChild;                                          *
  else {                                                          *
   if (array[leftChild] <= array[rightChild])                     *
    maxChild = rightChild;                                        *
   else                                                           *
    maxChild = leftChild;                                         *
```

```
  }
  if (array[root] < array[maxChild]) {
   temp = array[root];
   array[root] = array[maxChild];
   array[maxChild] = temp;
   RebuildHeapDownward (maxChild, bottom);
  }
 }
}

/***************************************************************/
/*                                                             */
/* RebuildHeapUpward: rebuilds heap when new item added at bot */
/*                                                             */
/***************************************************************/

template<class UserData>
void Heap<UserData>::RebuildHeapUpward (long root, long bottom) {
 if (bottom <= root) return;
 UserData temp;
 long parentNode;
 parentNode = (bottom - 1) / 2;
 if (array[parentNode] < array[bottom]) {
  temp = array[parentNode];
  array[parentNode] = array[bottom];
  array[bottom] = temp;
  RebuildHeapUpward (root, parentNode);
 }
}

/***************************************************************/
/*                                                             */
/* SortHeap: sort the heap into numerical order                */
/*                                                             */
/***************************************************************/

template<class UserData>
void Heap<UserData>::SortHeap () {
 long i;
 for (i=numElements/2 - 1; i>=0; i--) {
  RebuildHeapDownward (i, numElements-1);
 }

 for (i=numElements-1; i>=1; i--) {
  UserData temp = array[0];
  array[0] = array[i];
  array[i] = temp;
  RebuildHeapDownward (0, i-1);
 }
}

/***************************************************************/
```

```
/*                                                              */
/* GrowHeapBy: enlarge max size of array,copying existing items*/
/*                                                              */
/***************************************************************/

template<class UserData>
bool Heap<UserData>::GrowHeapBy (long growby) {
 if (growby <= 0) return false;
 UserData* newarray;
 try {
  newarray = new UserData [maxSize + growby];
 }
 catch (std::bad_alloc e) {
  // check if out of memory
  cerr << "Error: out of memory\n";
  return false;
 }
 for (long i=0; i<numElements; i++) {
  newarray[i] = array[i];
 }
 delete [] array;
 array = newarray;
 maxSize += growby;
 return true;
}

/***************************************************************/
/*                                                              */
/* Heap: copy ctor - make a duplicate copy of passed Heap      */
/*                                                              */
/***************************************************************/

template<class UserData>
Heap<UserData>::Heap (const Heap<UserData>& h) {
 Copy (h);
}

/***************************************************************/
/*                                                              */
/* operator= make us a duplicate of passed Heap object        */
/*                                                              */
/***************************************************************/

template<class UserData>
Heap<UserData>& Heap<UserData>::operator= (
                                       const Heap<UserData>& h) {
 if (this == &h) return *this;
 delete [] array;
 Copy (h);
 return *this;
}

```

```
/*****************************************************************/
/*                                                               */
/* Copy: make a duplicate of passed Heap object                  */
/*                                                               */
/*****************************************************************/

template<class UserData>
void Heap<UserData>::Copy (const Heap<UserData>& h){
 numElements = h.numElements;
 maxSize = h.maxSize;
 try {
  array = new UserData [maxSize];
 }
 catch (std::bad_alloc e) { // check if out of memory
  cerr << "Error: out of memory\n";
  array = 0;
  numElements = maxSize = 0;
  return;
 }
 for (long i=0; i<numElements; i++) {
  array[i] = h.array[i];
 }
}

#endif
```

A Priority Queue Based Upon a Heap

Next, let’s see how we can implement a priority queue based upon our **Heap** class. Recall with a priority queue, when items are enqueued, they must be placed into order based on some kind of priority scheme. Then the dequeue operation is simple, it just gets the item at the head of the queue and rebuilds the heap downward. So all the modifications apply to the enqueue operation which must ensure that the item with the highest priority is at the front of the queue.

The implementation of **Dequeue** is simple if we maintain a heap. That is, the highest priority item is at the front of the queue in element 0. This we must remove it or rather copy it out of the array. Element 0 is then replaced by the last item in the queue and the number of elements decremented. By calling **RebuildHeapDownward** we guarantee that the next item of highest priority is now at element 0.

The **Enqueue** operation is actually even easier because of the Heap. Since the heap is maintained as a heap, the new item is added at the end of the array, which will be the lowest level and rightmost leaf. Then, by calling **RebuildHeapUpwards**, the item with the highest priority is placed at the top once again.

What restrictions are placed on user objects that can be in the priority queue? Only those required by the **Heap** class, operators = and <. Note the vital importance that operator< now takes on — what must be compared in the user items is the priority of each item!

The only remaining question is whether the **PriorityQueue** class should derive from Heap or use an instance of **Heap** as its data member? It can be done either way. However, I choose derivation to illustrate inheritance. Here is the **PriorityQueue** class as a template class derived from the **Heap** class.

PriorityQueue Template Class

```
#ifndef PRIORITYQUEUE_H
#define PRIORITYQUEUE_H

#include "Heap.h"

/***************************************************************/
/*                                                             */
/* PriorityQueue: a class to encapsulate a priority queue      */
/*                                                             */
/* user class must provide operator = and < and <=             */
/* operators < <= used to determine the priority of this item  */
/*                                                             */
/* Vital: if two items have the same priority, then            */
/*        if those items must remain in FIFO order, these      */
/*        operator functions must account for their order      */
/*                                                             */
/***************************************************************/

template<class UserData>
class PriorityQueue : public Heap<UserData> {
public:

     PriorityQueue (long max = 100);
    ~PriorityQueue () {}

// Dequeue returns true and fills userdata with the item
bool Dequeue (UserData& userdata);

// Enqueue a copy of the user's data
bool Enqueue (const UserData& data);
};

/***************************************************************/
/*                                                             */
/* Heap: allocate the empty max sized array of user items      */
/*                                                             */
/***************************************************************/

template<class UserData>
```

```
* PriorityQueue<UserData>::PriorityQueue (long max)                 *
*                        : Heap<UserData> (max) {}                  *
*                                                                   *
* /*****************************************************************/*
* /*                                                               */*
* /* Dequeue: returns userdata filled with next item and true      */*
* /*          or returns false is queue is empty                   */*
* /*                                                               */*
* /*****************************************************************/*
*                                                                   *
* template<class UserData>                                          *
* bool PriorityQueue<UserData>::Dequeue (UserData& userdata) {      *
*  if (IsArrayEmpty())                                              *
*   return false;                                                   *
*  userdata = array[0];                                             *
*  array[0] = array[numElements - 1];                               *
*  numElements--;                                                   *
*  if (numElements)                                                 *
*   RebuildHeapDownward (0, numElements - 1);                       *
*  return true;                                                     *
* }                                                                 *
*                                                                   *
* /*****************************************************************/*
* /*                                                               */*
* /* Enqueue: if no more room in array, it grows the array         */*
* /*          then enqueues a copy of the user's data              */*
* /* Note: UserData's op< is called to determine item priority     */*
* /*                                                               */*
* /*****************************************************************/*
*                                                                   *
* template<class UserData>                                          *
* bool PriorityQueue<UserData>::Enqueue (const UserData& userdata){*
*  if (IsArrayFull())                                               *
*   if (!GrowHeapBy (100)) return false;                            *
*  array[numElements] = userdata;                                   *
*  numElements++;                                                   *
*  RebuildHeapUpward (0, numElements - 1);                          *
*  return true;                                                     *
* }                                                                 *
*                                                                   *
* #endif                                                            *
.))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))-
```

There is only one problem with this priority queue class being based upon a heap class. If two or more user items have the same priority, then ideally a queue should keep those items in the order that they were enqueued, first in-first out. However, neither the queue nor heap rebuilding functions can recall the actual original order of the items. Thus, as it is now implemented, items with the same priority are going to lose their basic FIFO nature. However, if the items themselves can maintain an indication of their enqueue order, then the operators < and <= functions can deal with this situation, providing the correct order between items of the same priority.

Pgm13a tests both of these new classes, the **Heap** and the **PriorityQueue**. It illustrates how the user item can maintain the FIFO nature of items with the same priority. It begins by making a heap of a series of eleven integers, sorts them and displays the heap.

To show the **PriorityQueue** in operation, **Pgm13a** next simulates a veterinarian's patient queue. At a clinic, as people arrive with their pets, they are serviced in a FIFO manner. However, emergency cases can arrive and are handled ahead of the non-emergency pets. File **patients.txt** simulates a few hours of a day at the clinic.

```
+))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* The Patients.txt File                                                   *
/)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
*  1 A Samuel Spade        Fido                 0                         *
*  2 A John Jones          Rover                0                         *
*  3 A Betsy Ann Smithville Jenny               0                         *
*  4 H                                                                    *
*  5 H                                                                    *
*  6 A Lou Ann deVille     Kitty                1                         *
*  7 H                                                                    *
*  8 A Tom Smythe          Fifi                 0                         *
*  9 A Marie Longfellow    Jack                 1                         *
* 10 A Alicia J. Jammissons Pretty Little Kitten 0                        *
* 11 A Harry Thumbs        Buster Brown         1                         *
* 12 H                                                                    *
* 13 H                                                                    *
* 14 H                                                                    *
* 15 H                                                                    *
* 16 H                                                                    *
.)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))-
```

The first character of each line contains A for the arrival of a new patient or H for handle the next patient. The remainder of the arrival lines contains the owner's name and the pet's name followed by a priority code. A code of 1 indicates an emergency case, while a code of 0 is represents a routine visit.

The output of the program is shown below. Notice the order of handling that occurs when an emergency case becomes enqueued.

```
+))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Output of Pgm13a Tester Program                                         *
/)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
*  1 0                                                                    *
*  2 1                                                                    *
*  3 2                                                                    *
*  4 3                                                                    *
*  5 4                                                                    *
*  6 5                                                                    *
*  7 6                                                                    *
*  8 7                                                                    *
*  9 8                                                                    *
* 10 9                                                                    *
```

```


Handling the Patient Queue

Added:   Samuel Spade          Fido                    0
Added:   John Jones            Rover                   0
Added:   Betsy Ann Smithville  Jenny                   0
Treated: Samuel Spade          Fido                    0
Treated: John Jones            Rover                   0
Added:   Lou Ann deVille       Kitty                   1
Treated: Lou Ann deVille       Kitty                   1
Added:   Tom Smythe            Fifi                    0
Added:   Marie Longfellow      Jack                    1
Added:   Alicia J. Jammissons  Pretty Little Kitten  0
Added:   Harry Thumbs          Buster Brown            1
Treated: Marie Longfellow      Jack                    1
Treated: Harry Thumbs          Buster Brown            1
Treated: Betsy Ann Smithville  Jenny                   0
Treated: Tom Smythe            Fifi                    0
Treated: Alicia J. Jammissons  Pretty Little Kitten  0

File processing is complete
No memory leaks.
```

Pgm13a also illustrates some additional features of structures. A structure can have member functions, just as a class can. However, those member functions are always public in nature, as are all of the structure data members. This is vital because I am implementing the user data as a Patient structure which must therefore implement the two comparison operator functions. The syntax of structure member functions is exactly the same as that of a class.

The only tricky aspect of the program and the operator functions is the need to maintain the FIFO order for all patients with the same priority code. To do that, I added an additional member to the structure, **order**. As each new item is added into the queue, I increment the order number. Thus, each item has a different order number increasing in size with each new addition to the queue. Hence, the two operator functions can then correctly maintain the FIFO order of items whose priority values are the same.

Pgm13a Tester of Heap and PriorityQueue Classes

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cctype>
#include <crtdbg.h>

#include "Heap.h"
#include "PriorityQueue.h"

```

```
using namespace std;

const int MAXLEN = 21;

/***************************************************************/
/*                                                             */
/* Patient: defines a patient for the priority queue operation */
/*                                                             */
/* must implement ops < and <= for Heap operations             */
/*                                                             */
/***************************************************************/

struct Patient {
 char ownerName[MAXLEN];
 char petName[MAXLEN];
 int  priority;
 int  order;
 bool operator< (const Patient& p) const;
 bool operator<= (const Patient& p) const;
};

/***************************************************************/
/*                                                             */
/* operator<: if this priority is < p's return true            */
/*            however, if they have the same priority,         */
/*            we must keep the original queue order intact,    */
/*         so check on reverse on incremental order of arrival */
/*                                                             */
/***************************************************************/

bool Patient::operator< (const Patient& p) const {
 if (priority < p.priority)
  return true;
 else if (priority == p.priority && order > p.order)
  return true;
 return false;
}

/***************************************************************/
/*                                                             */
/* operator<=: if this priority is < p's return true           */
/*            however, if they have the same priority,         */
/*            we must keep the original queue order intact,    */
/*         so check on reverse on incremental order of arrival */
/*                                                             */
/***************************************************************/

bool Patient::operator<= (const Patient& p) const {
 if (priority < p.priority)
  return true;
 if (priority == p.priority && order > p.order)
```

```
  return true;
 return false;
}

/***************************************************************/
/*                                                             */
/* Pgm12a: tests the Heap and PriorityQueue classes            */
/*                                                             */
/***************************************************************/

int main () {
 {
  // test the Heap class by inserting & sorting some integers
  Heap<int> heap;
  int i;
  for (i=0; i<10; i++) {
   if (!heap.IsArrayFull()) {
    heap.array[i] = i;
    heap.numElements++;
   }
  }

  heap.SortHeap ();

  for (i=0; i<heap.GetNumElements(); i++) {
   cout << heap.array[i] << endl;
  }

  // now test the priority queue
  PriorityQueue<Patient> queue;
  ifstream infile ("patients.txt");
  if (!infile) {
   cerr << "Error: cannot open patients.txt\n";
   return 1;
  }
  int line = 1;
  char type;
  Patient p;
  cout << "\n\nHandling the Patient Queue\n\n";
  while (infile >> type) {
   type = (char) toupper (type);
   if (type == 'A') {
    infile.get (type);
    infile.get (p.ownerName, sizeof (p.ownerName));
    infile.get (type);
    infile.get (p.petName, sizeof (p.petName));
    infile >> p.priority;
    if (!infile) {
     cerr << "Error: bad data on line: " << line << endl;
     infile.close ();
     return 2;
    }
```

```
    p.order = line;
    if (!queue.Enqueue (p)) {
     infile.close ();
     exit (1);
    }
    cout << "Added:   " << p.ownerName << "  " << p.petName
         << setw (3) << p.priority << endl;
   }
   else if (type == 'H') {
    if (queue.Dequeue (p)) {
     cout << "Treated: " << p.ownerName << "  " << p.petName
          << setw (3) << p.priority << endl;
    }
    else {
     cout << "Error: no more patients in queue\n";
    }
   }
   else {
    cerr << "Error: bad type code in patients.txt file on line: "
         << line << endl;
    infile.close ();
    return 3;
   }
   line++;
  }
  infile.close ();
  cout << "\nFile processing is complete\n";
 }

 // check for memory leaks
 if (_CrtDumpMemoryLeaks())
  cerr << "Memory leaks occurred!\n";
 else
  cerr << "No memory leaks.\n";

 return 0;
}
```

Now that we have these two classes operational, we can turn to the more complex graph situation which will make use of these classes.

Graphs

Basic Graph Terminology

The **graph data structure** is a tree in which any given node can have more than one parent node pointing to it and it can point to many child nodes. The nodes are called **vertices**. The lines that connect the vertices (nodes) are called either **edges** or **arcs**. Further, the edges or lines connecting the vertices (nodes) often have some kind of **weight** or importance or significance attached to them. Additionally, each edge can have a **direction** associated with it. For example, examine an airline's flying schedule between cities. Between any given two cities, flights may go in both directions or maybe only from city A to city B and not from city B back to city A. The air distance between the connected cities (vertices) is often the weight.

Graphs are frequently used to solve routing type problems. Airline routes, mass transportation routes, even Internet routing can make use of graphs. Graphs are used to answer two key questions: "Can I get from A to B?" and "Among all the routes between A and B, what is the shortest route to take?"

If a graph has direction associated with its lines or edges, it is called a **directed graph** or **digraph** for short. If none of a graph's lines or edges has any direction arrows on them, it is an **undirected graph**. Figure 13.8 shows an example of each.

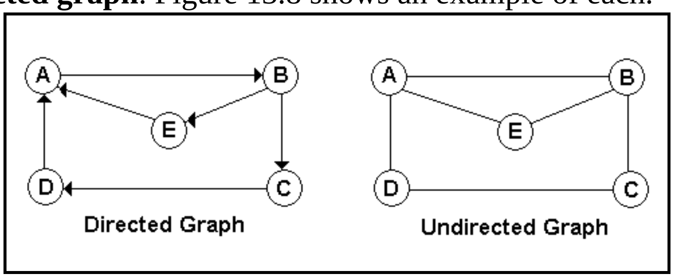


Figure 13.8 Directed and Undirected Graphs

Two vertices are said to be **adjacent vertices** if an edge or line directly connects them. Sometimes adjacent vertices are called **neighbors**. If the above Figure 13.8, A and B are adjacent vertices while A and C are not.

The sequence of vertices where each vertex is adjacent to the next one is called a **path**. A path can only follow the direction of travel along the edge or arc or line. In Figure 13.8 above, one path is {A, B, C, D} and another is {A, B, E} for example. In a digraph, travel is restricted to the direction of the arrows while in an undirected graph, travel can be in both directions.

A **cycle** is a path with at least three vertices that starts and ends on the same vertex. In the above figure and in both graphs, A, B, C, D, A is a cycle as is A, B, E, A. However, A, D, C, B, A is a cycle in the undirected graph but not in the digraph because a path must follow the direction arrows in the digraph. One special case of a cycle is known as a **loop**; in a loop there is one vertex and the line goes out from it and then back into that same vertex, rather like driving out of a city and then coming right back into that same city.

Another property known as **connected** applies to two vertices. If, ignoring direction, there is a path between two vertices, they are said to be **connected**. Further, in a digraph, there are three qualifiers to the connected property. If there is a path from each vertex to every other vertex, it is said to be **strongly connected**. However, if at least two vertices are not connected, the digraph is said to be **weakly connected**. The connected property only applies to digraphs because all undirected graphs would be strongly connected since there is no direction of travel to consider. A graph is **disjoint** is not connected in some manner. In Figure 13.8 above, the digraph is strongly connected because there is a path from every vertex to every other vertex. If the edge from E to A were removed, then the digraph would be weakly connected because there would then be no path from E to any other vertex. If we considered both of the graphs in Figure 13.8 to be one graph, it would be a disjointed graph because there is no way to get from the right portion to the left portion.

The final property of a graph is the **degree** of a vertex which is the total number of lines or edges into or out of it. The **outdegree** of a vertex is the total number of lines leaving that vertex while the **indegree** of a vertex is the total number of lines entering that vertex. In the digraph in Figure 13.8 above, the degree of vertex A is 3 while its outdegree is 1 and its indegree is 2. The degree of vertex E is 2 and its indegree and outdegree are both 1. The degree of vertex B in the digraph is also 3 but its outdegree is 2 while its indegree is 1.

A **network** is a graph whose edges or lines are weighted in some manner. For example, consider an airline's routes between cities. The weight would likely be the frequent traveler's miles between the two cities. Alternatively, the weight could be the price of the ticket or time of day travel or even the dates of travel. The nature of this weight information is unknown to a graph and it stored in an Edge structure provided by the client program. If the client program implements a few Edge structure operations, such as operator<, then the graph itself can find the minimum weighted route between two vertices.

I frequently fly out to Burbank, California (close to Los Angeles), to visit my young nephews. Figure 13.18 shows an airline's flight network from Peoria, Illinois to the Los Angeles area. I also inserted a flight from New York as well. The weight of each edge is the flight miles between those cities.

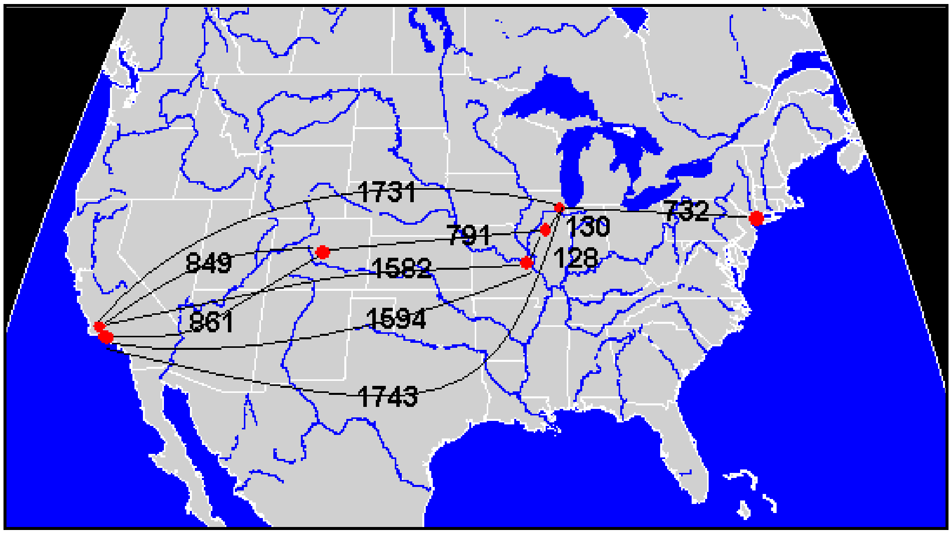


Figure 13.18 A Airline Flight Network

From a graph representing the data shown in Figure 13.18, we can as the graph if there is a flight from Peoria to Burbank. We can also ask what is the shortest path between Peoria and Burbank. "Is there a path?" and "What is the shortest path?" are two vital uses of a graph.

A **spanning tree** is a tree that contains all of the vertices in a graph. A **minimum spanning tree** of a network is a spanning tree in which the sum of its weights are the minium. So if a graph has weighted edges, then we can construct its minimum spanning tree. If the graph represented a computer network (the workstations are the vertices and the cables are the edges), then its minimum spanning tree would tell us how to connect all these computers to the network using the minimum amount of cabling. Of course, if two or more edges have the same weight, there can be more than one such minimum spanning tree.

Graph Basic Operations

Seven basic operations must be supported by a graph. These include Add a Vertex, Add an Edge, Delete a Vertex, Delete an Edge, Find a Vertex, Find an Edge, and Traverse the Graph. Then, other additional specialized processing functions can be added. Let's examine these basic functions in detail before we consider how to implement them.

Add a Vertex inserts a new vertex into the graph. It is always disjointed when it is added because no edges (lines connecting the new vertex to any other vertex) have yet been added. So normal operation usually involves a call to Add a Vertex followed by one or more calls to Add an Edge. This is shown in Figure 13.9 in which vertex E has just been added.

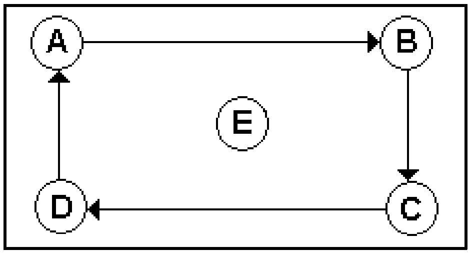


Figure 13.9 Add Vertex E

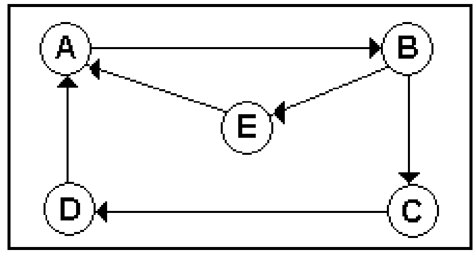


Figure 13.10 Add Edge E->A;B->E

Add an Edge connects a vertex to another vertex. In Figure 13.10, two calls to Add an Edge have been made, adding directed edge E->A and B->E. Further, if the graph is a digraph, then one vertex must be specified as the source and one is the destination.

Delete a Vertex deletes a vertex from the graph. It also deletes all edges or lines that connect to it. If we begin with the graph in Figure 13.10 and delete vertex E, then the resultant digraph is shown in Figure 13.11.

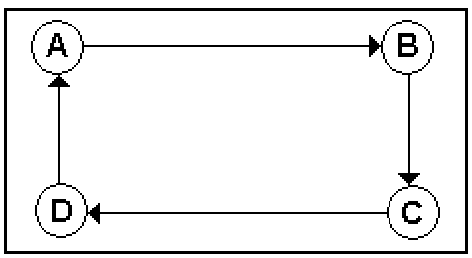


Figure 13.11 The Result of Deleting Vertex E from Figure 13.10

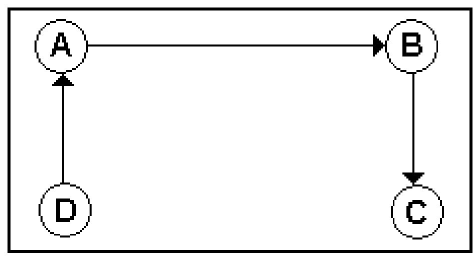


Figure 13.12 Deletion of Edge C-D

Delete an Edge removes one edge or line that connects two vertices. Figure 13.12 shows what results if we delete the edge from C to D.

Traverse Graph permits the client to visit all of the vertices in the graph. But a traversal of a graph is a bit more complex. Since any given vertex can have many different parents, there are going to be multiple ways to get to any specific vertex. How can we tell if we have already visited a given vertex? The usual method is to maintain a **visited** indicator. Initially as the traversal begins, all flags are cleared or set to **0**. Then, as a vertex is visited or processed, its **visited** indicator is set to a non-zero value.

Recall that with trees, there were several different ways the tree nodes and leaves could be visited. In what order do we visit the vertices? There are two usual methods. The first is a **depth-first traversal** in which we process all of the descendants of a node or vertex before we move to an adjacent vertex. If we were processing airline travel routes, a depth-first approach would yield the routing which had the most connections. This is usually considered undesirable by passengers. The other method is **a breath-first traversal** in which we visit all adjacent vertices before we visit descendants. In the airline travel example, a breadth-first traversal would yield the nonstop flights before those with many connections. These traversals parallel those of a tree and are more easily seen if you view the graph as a tree.

The **depth-first** process begins by visiting the first vertex. Next, we choose any one of its descendants and visit it and then one of its descendants. When we finally encounter a vertex with no more descendants (parallel to reaching a leaf in a tree), we back track to its parent and choose the next descendant and follow it down. This backtracking immediately tells us that a stack is needed to handle the processing. However, we must avoid revisiting vertices. Consider the undirected graph shown below in Figure 13.13. Let's assume that the first vertex is A.

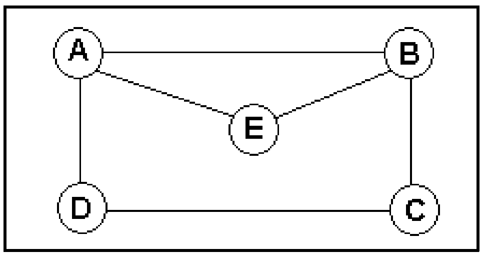


Figure 13.13 Undirected Graph

We begin by pushing A onto the stack. The main loop then operates while there is still another vertex on the stack. We pop A off of the stack, process A, and push all of its descendant vertices onto the stack: E, D, and B in this case. Now we repeat the main loop and pop off B. We process B, but when we go to push the descendants of B, notice that those would be A and C and E. We have already processed A and E is on the stack to be done later on. Here is where we must know additional facts or we end up with an infinite loop forever pushing the same vertices onto the stack.

We actually need to know two key items: has this vertex been processed and has this vertex been pushed onto the stack? This is accomplished by creating a visited array of integers values. Initially, all are set to 0. When we push a vertex onto the stack, we mark it as having been visited by changing its indicator to 1. When we actually process a vertex, we can mark it with a 2, for example. Thus, initially A is so marked. When we push its descendants E, D and B onto the stack, we mark their visited indicators to 1. Thus, when we pop and actually process vertex B and are ready to push its descendants onto the stack, we can avoid pushing vertices A and E because A has already been visited and E is on the stack to be visited. Thus, when processing vertex B, only vertex C is pushed onto the stack. Figure 13.14 shows the sequence of vertices that are processed and the stack as its descendants are pushed.

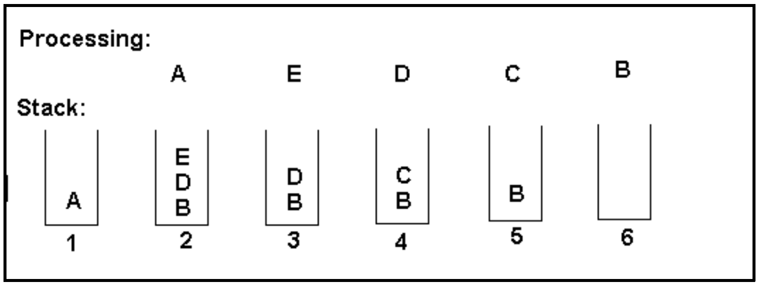


Figure 13.14 Depth Traversal Steps

In the **breath-first traversal** method, we visit all adjacent vertices before visiting any descendants. This means that we must queue up the sequence of vertices to be visited. So a queue structure is used, not a stack. Again referring to Figure 13.13 above, initially, we set our graph vertex pointer to that of the first one, vertex A. The main outer loop runs as long as there remains vertices in the graph. If this current vertex has not yet been processed or enqueued, we perform all of the following steps. If this one has not been enqueued, it is enqueued and marked as enqueued. In all cases, we must now process all items currently in the queue as these represent all of a vertex's descendants. So until the queue is empty, we dequeue a vertex and process it and mark it as having been processed. Next, we enqueue all of this vertex's descendants and mark each as having been enqueued. Finally, at the bottom of the main loop, we move onto the next vertex in the graph. As shown in Figure 13.15 below, we would process the vertices in this order: A, B, D, E, C.

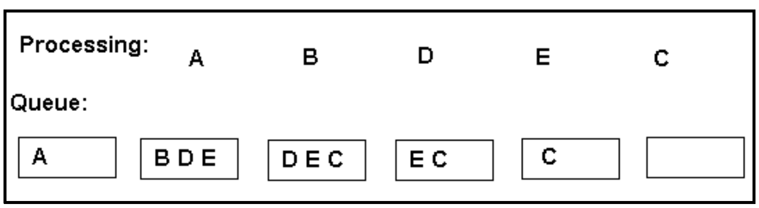


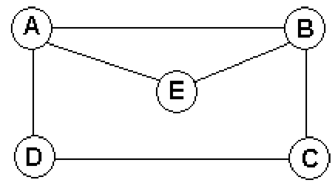
Figure 13.15 Breadth Traversal Using a Queue

Data Structures to Represent Graphs

The graph data is composed of the vertices and all of the edges. Two methods to store these data immediately suggest themselves: two arrays or two linked lists.

The array method is simpler to implement but costly in terms of wasted space, and, unless one uses growable arrays, the number of vertices and edges must be known ahead of time. The linked list method minimizes the amount of memory required; the number of vertices and edges does not need to be known before hand.

Referring again to the undirected graph in Figure 13.13 above, let's see how the data could be stored using arrays. First, one would define an array of five Vertex structures or class instances. Each element in the **Vertex** array represents one vertex in Figure 13.13. In the general case, any one vertex could be connected to all of the other vertices, the second array of **Edge** structures or classes would have to be a two-dimensional array. Figuratively, the rows represent the “from” vertices while the columns of a row represent the vertices that are connected “to” that from vertex. If no weights were needed, then this two-dimensional array could be of type **bool**, where a **true** indicates that there is a connection between this row’s vertex and this column’s vertex. This is illustrated in Figure 13.16 where I used 1's and 0's to indicate **true** and **false**.



Vertex	Edges A	B	C	D	E
A	0	1	0	1	1
B	1	0	1	0	1
C	0	1	0	1	0
D	1	0	1	0	0
E	1	1	0	0	0

Figure 13.16 Using Arrays to Define an Undirected Graph Structure

For example, for vertex A, the **Edge** array says it is not connected to itself or C but vertex A is connected to B, D, and E. The **Edge** array can also indicate any direction of the connection. For example, if A was connected to B but B was not connected to A, then in the second row (the B “from” row), the A column would contain a 0 or **false**.

If we use single linked lists, we gain far more flexibility in the design. A vertex’s list would hold all of the vertex data. Each of these vertex nodes would contain a head pointer to a list of edge nodes to which this vertex was connected. Then, for each vertex in the list, we build a linked list of edge nodes which can contain the weight of that edge as well as a pointer to the vertex of the connection. This is shown in Figure 13.17 below.

The **Vertex** structure or class contains basic data about the vertex itself. The **Edge** structure or class contains the weight of the edge or similar information. If there were no weight or properties associated with an edge, then this structure is not needed or can be a dummy place holder.

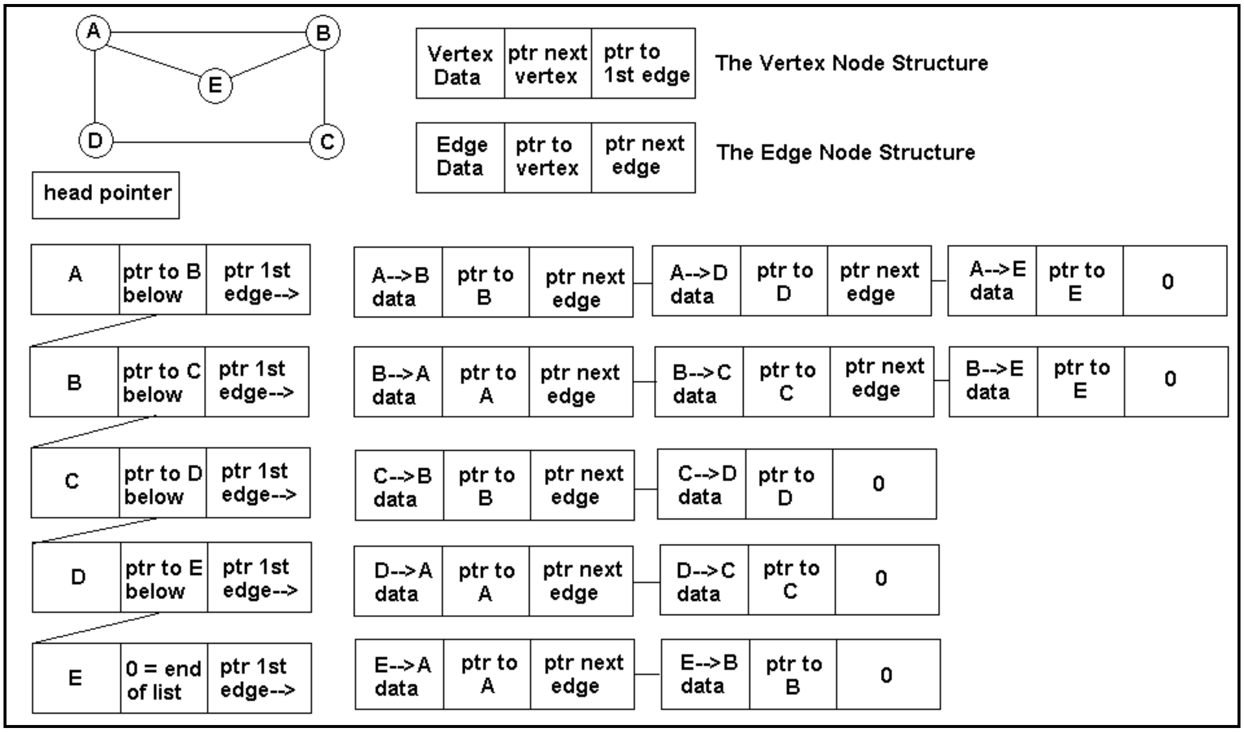


Figure 13.17 Using Lists to Define the Graph

The linked list is the method that I use implement to the graph data structure. The starting point is to implement the set of basic functions as outlined above. However, I will add one more function to that group, **DisplayTree**. The **DisplayTree** function displays each vertex in the vertex list followed by all of the vertices that are connected to it. This is useful to visually verify we have constructed the graph correctly.

Caution. The graph data structure utilizes nearly everything you have learned about data structures to this point. The complete implementation makes use of single and double linked lists, stacks, queues and priority queues.

How Do We Design a Graph Data Structure?

The first design consideration is, do we store **void** pointers to the user’s data items or use a template class or perhaps use some kind of derived class? This time, the answer is nearly forced upon us.

From the above discussion and from the priority queue based upon the Heap class, the user’s data must be stored in either structures or classes because those structures or classes must implement the basic comparison operators, such as < or <= or ==, for example. This tells us at once that we cannot store **void** pointers to the user’s data because we would be unable to invoke these operator functions when we need them within the graph functions.

The graph structure as depicted in Figure 13.17 above is not closely related to any other data structure from which we could derive a graph structure. So using a template class might be the next suggestion as a viable method of writing a generic graph container class.

But there is a serious design consideration that must be met. We cannot know in advance exactly what the user data will be. So let’s say that the user always provides his data in a pair of structures or classes called **Vertex** and **Edge**. These contain the vertex and edge **application-specific** data. Now before we get into the complexities, let’s make this more real by seeing just what that means.

The sample application **Pgm13b** is airline routes across the country. Each **Vertex** is a city that the airline services. The client program can store any number of properties in this **Vertex** structure, but for simplicity, I am storing only the city name. The **Edge** contains the distance in air miles between a pair of cities. Here are the application’s definition and implementation of the needed comparison operators. I call them **Vertex** and **Edge**; both are structures with public, operator overloaded member functions for the comparisons.

Here is the **VertexNode.h** application file. In the sample program, I split the function bodies off into the **VertexNode.cpp** file.

```
Pgm13b's Airline Travel Vertex and Edge Structures

#ifndef VERTEXEDGE_H
#define VERTEXEDGE_H
#include <string.h>

/***********************************************************/
/*                                                         */
/* Vertex: this structure contains the city name of the    */
/*         airport - it could contain other application    */
/*         specific information as needed                  */
/* It must implement the comparison operators for graph's */
/* internal usage.                                         */
/*                                                         */
/* It can be a structure or a class                        */
/*                                                         */
/***********************************************************/

const int CITYLEN = 51;

struct Vertex {
 char city[CITYLEN]; // the city containing the airport

 bool operator> (const Vertex& v2) const;
```

```
*  bool operator!= (const Vertex& v2) const;                            *
*  bool operator>= (const Vertex& v2) const;                            *
*  bool operator< (const Vertex& v2) const;                             *
*  bool operator<= (const Vertex& v2) const;                            *
*  bool operator== (const Vertex& v2) const;                            *
* };                                                                    *
*                                                                       *
* /********************************************************/            *
* /*                                                      */            *
* /* Edge: This structure contains edge specific information*/          *
* /*       and most often is the weight assigned to this    */          *
* /*       edge. With airplane travel, it is the distance   */          *
* /*       between the from city and to city.               */          *
* /*                                                      */            *
* /* If there is no weights needed in the graph, the Edge   */          *
* /* still must be provided, however, it can be dummied     */          *
* /* that is, one could store a single dummy char member so */          *
* /* that the structure exists as far as Graph is concerned */          *
* /*                                                      */            *
* /********************************************************/            *
*                                                                       *
* struct Edge {                                                         *
*  double distance;                                                     *
*  bool operator< (const Edge& e2) const;                               *
*  bool operator== (const Edge& e2) const;                              *
*  Edge operator+ (const Edge& e2) const;                               *
* };                                                                    *
*                                                                       *
* /********************************************************/            *
* /*                                                      */            *
* /* The Edge comparison functions that are required by     */          *
* /*        Graph - these must be provided, even if dummied */          *
* /*                                                      */            *
* /********************************************************/            *
*                                                                       *
* Edge Edge::operator+ (const Edge& e2) const {                         *
*  Edge res = *this;                                                    *
*  res.distance += e2.distance;                                         *
*  return res;                                                          *
* }                                                                     *
*                                                                       *
* bool Edge::operator< (const Edge& e2) const {                         *
*  if (distance < e2.distance) return true;                             *
*  return false;                                                        *
* }                                                                     *
*                                                                       *
* bool Edge::operator== (const Edge& e2) const {                        *
*  return distance == e2.distance ? true : false;                       *
* }                                                                     *
*                                                                       *
* /********************************************************/            *
* /*                                                      */            *
```

```
/* The Vertex comparison operators                         */
/*                                                         */
/* These must be implemented in terms of the actual data  */
/* contained in the Vertex - here the city's name          */
/*                                                         */
/***********************************************************/

bool Vertex::operator> (const Vertex& v2) const {
 return strcmp (city, v2.city) > 0 ? true : false;
}

bool Vertex::operator!= (const Vertex& v2) const {
 return strcmp (city, v2.city) != 0 ? true : false;
}

bool Vertex::operator>= (const Vertex& v2) const {
 return strcmp (city, v2.city) >= 0 ? true : false;
}

bool Vertex::operator< (const Vertex& v2) const {
 return strcmp (city, v2.city) < 0 ? true : false;
}

bool Vertex::operator<= (const Vertex& v2) const {
 return strcmp (city, v2.city) <= 0 ? true : false;
}

bool Vertex::operator== (const Vertex& v2) const {
 return strcmp (city, v2.city) == 0 ? true : false;
}

#endif
```

Notice that the **Edge** structure must also implement the **operator+** function. This feature is needed when finding the shortest path between two vertices as we will soon examine.

Next, what data does the **Graph** class need? For each vertex, we must store traversal data in addition to the user's **Vertex** instance. We should store the indegree and outdegree counts. For each edge, besides storing the user's **Edge** information, we need to store the "to" vertex pointer along with a pointer to the next edge. In other words, the **Graph** class must store some additional information for both the vertices and edges. So we wind up defining a **VertexNode** and an **EdgeNode** as follows.

```
// the Visited Enum Flag Definition for Traversal Usage
enum VisitedFlag {NotVisited, Visited, Processed};

struct VertexNode {
 Vertex      vertexData;     // the user's actual data
 VisitedFlag visitedFlag;    // used by traversals
```

```
 long        inDegree;       // num edges coming into this vertex
 long        outDegree;      // num edges going out from this one
 bool        inShortestPath; // used by FindShortestPath
 VertexNode* ptrFwd;         // fwd ptr to next VertexNode
 EdgeNode*   ptrEdgeHead;    // head ptr for list of EdgeNodes
};

struct EdgeNode {
 Edge        edgeData;        // the user's actual weighted edge
 bool        inShortestPath;  // used by FindShortestPath
 VertexNode* ptrToVertex;     // the vertex this one goes to
 EdgeNode*   ptrFwd;          // fwd ptr to next edge in list
};
```

If we choose to go the template route, then each of these structures becomes a template.

```
template<classVertex, class Edge>
struct VertexNode {
 Vertex      vertexData;
 VisitedFlag visitedFlag;
 long        inDegree;
 long        outDegree;
 bool        inShortestPath;
 VertexNode<Vertex, Edge>* ptrFwd;
 EdgeNode<Vertex, Edge>*   ptrEdgeHead;
};
```

If we used the template definitions for the two nodes, then the **Graph** is a template also based on **Vertex** and **Edge**. And herein lies the problem. Several member functions of **Graph** are going to use the **Stack**, **Queue**, **DoubleLinkedList**, and the **PriorityQueue** classes to carry out their tasks. These four classes are now template classes. And we now cannot construct specific instances of these four containers by coding the following.

```
Stack<VertexNode<Vertex, Edge>*> stack;
```

To create instances of the **Stack** template class, we must use a known at compile-time data type.

We cannot get around this problem by rewriting the four template classes to use **void** pointers to the user's data instead of being template classes. If we did so, then we would not be able to invoke the required operator comparison functions of the user's data. A **void** pointer cannot be used to invoke a function unless it is typecast to the type of data to which it is really pointing, which is the user's data — of which we know nothing. It is rather a catch-22 situation with the **void** pointers.

We could create specific instances of the container classes by having the type of data be a **void***.

```
Stack<void*> stack;
```

However, two new problems arise. We get back **void** pointers which must be typecasted back to the type of data to which they really are pointing.

```
VertexNode<Vertex, Edge>* ptrvertex =
                          (VertexNode<Vertex, Edge>*) stack.Pop();
```

This is cumbersome at best, though doable. However, if the four container classes are storing **void** pointers, then the **PriorityQueue** is in trouble because its operation requires user operator comparison functions to determine the largest priority item during the heap rebuilding operations. Again, we cannot do so with a **void** pointer. So the approach of using templates for these **Graph** node structures is not going to work.

Does this mean that we must forsake our overall design guidelines of writing reusable container classes and write something totally specific to the air travel problem? No. There is another approach we can take that still retains a generalized nature.

Notice that in every graph situation, the user must be specifying their vertex data and edge information, even if the edge information is just a placeholder because there is no weight associated with edges. What if we **force** the user to provide a header file that must define **Vertex** and **Edge** as either structures or classes along with the required operator comparison functions? If we can include this file in **Graph.h**, then we know at compile-time what the actual items are going to be. We do not need to "template-ize" our two node structures that wrap around the user's data. Thus, **Graph** does not need to be a template class. Hence, **Graph** functions can then actually create specific instances of the template containers this way.

```
Stack<VertexNode> stack;
```

Here, the compiler knows exactly what a **VertexNode** is at compile-time.

This is the approach that I am taking. Force the user to provide a header file called **VertexEdge.h** in which they define **Vertex** and **Edge** as structures or classes and provide the implementation of the needed comparison functions.

With my design, the **Graph** can be a weighted (network) graph or not. If there is no weight to an edge, the edge structure or class must still be provided, but it can be dummied out, say containing a single char item that is never really used. The edge instance is used to indicate that there is a connection from a vertex to another vertex. If there is an actual weight to an edge, then the contents of the edge instance can be used to find the shortest path and so on. By designing the **Graph** class this way, we can write one class that can handle any **Graph** situation. Specifically, we do not need a separate class to handle a weighted or network graph. By using the linked list of connected vertices approach, the single **Graph** class can handle undirected graphs as well as digraphs. The only drawback is the user can only have one kind of **Graph** per application and they must provide the needed header file of that precise name with those precise class or structure names.

Next, let's examine the overall **Graph** class definition.

```
+))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Graph Class Definition                                                        *
/)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
```

```
*    #ifndef GRAPH_H                                                   *
*    #define GRAPH_H                                                   *
*                                                                      *
*    #include "VertexEdge.h" // !!!! user must supply this file !!!!!!*
*                                                                      *
*    #include "Stack.h"                                                *
*    #include "PriorityQueue.h"                                        *
*    #include "Queue.h"                                                *
*    #include "DoubleLinkedList.h"                                     *
*    using namespace std;                                              *
*                                                                      *
*    struct EdgeNode;             // forward reference                 *
*                                                                      *
*    // the Visited Enum Flag Definition for Traversal Usage           *
*    enum VisitedFlag {NotVisited, Visited, Processed};                *
*                                                                      *
*                                                                      *
*    /***************************************************************/*
*    /*                                                             */*
*    /* VertexNode: stores user Vertex info along with Graph data   */*
*    /*                                                             */*
*    /***************************************************************/*
*                                                                      *
*    struct VertexNode {                                               *
*     Vertex      vertexData;     // the user's actual data            *
*     VisitedFlag visitedFlag;    // used by traversals                *
*     long        inDegree;       // num edges coming into this vertex*
*     long        outDegree;      // num edges going out from this one*
*     bool        inShortestPath; // used by FindShortestPath          *
*     VertexNode* ptrFwd;         // fwd ptr to next VertexNode        *
*     EdgeNode*   ptrEdgeHead;    // head ptr for list of EdgeNodes    *
*    };                                                                *
*                                                                      *
*                                                                      *
*    /***************************************************************/*
*    /*                                                             */*
*    /* EdgeNode: stores user's Edge info along with Graph's info   */*
*    /*                                                             */*
*    /***************************************************************/*
*                                                                      *
*    struct EdgeNode {                                                 *
*     Edge        edgeData;        // the user's actual weighted edge *
*     bool        inShortestPath;  // used by FindShortestPath         *
*     VertexNode* ptrToVertex;     // the vertex this one goes to      *
*     EdgeNode*   ptrFwd;          // fwd ptr to next edge in list     *
*    };                                                                *
*                                                                      *
*                                                                      *
*    /***************************************************************/*
*    /*                                                             */*
*    /* Graph: a class to encapsulate any kind of graph data str    */*
*    /*                                                             */*
```

```
/***************************************************************/

class Graph {
protected:
 VertexNode* ptrHead; // ptr to the list of vertices

public:
     Graph ();
    ~Graph ();
void EmptyGraph ();

bool AddVertex (const Vertex& vert);
int  AddEdge   (const Vertex& fromVert, const Vertex& toVert,
                const Edge& edge);

VertexNode* FindThisVertex (const Vertex& v) const;
VertexNode* FindThisVertex (const Vertex& v,
                            VertexNode*& ptrprev) const;
EdgeNode*   FindThisEdge   (const Vertex& from,
                            const Vertex& to) const;

bool DeleteVertex (const Vertex& hasIdToDel);
int  DeleteEdge (const Vertex& fromVert, const Vertex& toVert);

void DisplayTree (void (*ShowTree) (Vertex& v,
                                    bool isConnectedVertex));

void ClearProcessedFlags ();
void DepthFirstTraversal (void (*Process) (Vertex& v));
void BreadthFirstTraversal (void (*Process) (Vertex& v));


bool DoesPathExistBetween_DepthFirst   (const Vertex& from,
                                        const Vertex& to);
bool DoesPathExistBetween_BreadthFirst (const Vertex& from,
                                        const Vertex& to);

void ClearInShortestTreeFlags ();
void BuildMinimumSpanningTree (Edge& maxValue);
void ShowMinimumSpanningTree (
       void (*DisplayEdge) (const Vertex& from, const Vertex& to,
                            const Edge& edge));

bool FindShortestPath (const Vertex& from, const Vertex& to,
                      const Edge& minDist, bool showOnlyShortest,
                      bool smallestIsHighest,
                 void (*DisplayShortestPath) (const Vertex& from,
                        const Vertex& to, const Edge& distance));
};

#endif
```

The Graph Basic Functions' Implementation

Let's begin by examining the simpler functions. The only data member is the pointer to the head of the single linked list of vertices which is initialized to 0 in the constructor. The user would next call **AddVertex**.

AddVertex begins by allocating a new **VertexNode** and copying the user's passed vertex data into the new structure instance. After then setting the other members to zeros, this new node must be added to the linked list. There are two cases. If the list is empty, then this one goes at the head of the list. If there are vertices in the list, we must insert this one in sorted order. The sorted order is dictated by the user's **operator**> function.

```
     VertexNode* ptrthis = ptrHead;
     VertexNode* ptrprev = 0;
     while (ptrthis && vertNew > ptrthis->vertexData) {
      ptrprev = ptrthis;
      ptrthis = ptrthis->ptrFwd;
     }
```

Once we have found where this one goes (at the head or after another vertex), we use the normal list insertion coding.

Adding an edge along with some other functions are going to need a **FindThisVertex** function. One version simply returns a pointer to the found **VertexNode**. The overloaded version also fills in the passed reference to the previous **VertexNode** for ease of list insertion. To find out if this **VertexNode** matched the user's requested **Vertex**, I call the user's **operator!=** function.

```
VertexNode* Graph::FindThisVertex (const Vertex& v,
                                   VertexNode*& ptrprev) const {
 ptrprev = 0;
 VertexNode* ptrfind = ptrHead;
 while (ptrfind && v != ptrfind->vertexData) {
  ptrprev = ptrfind;
  ptrfind = ptrfind->ptrFwd;
 }
 return ptrfind;
}
```

Again, there is really nothing new in this routine either.

After adding some vertices, the user is likely to add one or more edges connecting pairs of vertices. This time, three errors are possible: out of memory, unable to find the "from" vertex, and unable to find the "to" vertex. The **AddEdge** function must therefore return an integer indicating success (0) or one of the three failures. Thus, the function begins by attempting to find the two needed vertices. If both are found, then a new **EdgeNode** is allocated and filled with the user's data and the graph members initialized. The **indegree** and **outdegree** members of the two found vertex nodes are also incremented.

```
int Graph::AddEdge (const Vertex& fromVert, const Vertex& toVert,
                    const Edge& edge) {
```

```
 VertexNode* ptrFrom = FindThisVertex (fromVert);
 if (!ptrFrom) return -2;

 VertexNode* ptrTo = FindThisVertex (toVert);
 if (!ptrTo) return -3;

 EdgeNode* ptrnew = new EdgeNode; // allcoate a new edge node
 if (!ptrnew) return -1;

 ptrnew->edgeData = edge;
 ptrnew->inShortestPath = false;
 ptrFrom->outDegree++;
 ptrTo->inDegree++;
 ptrnew->ptrToVertex = ptrTo;
```

Examine Figure 13.17 once more. Ths new **EdgeNode** must be chained into the single linked list of edges attached to the “from” **VertexNode**. If this is the first edge, then it is added at the head of the edge list in the “from” **VertexNode**. If not, the **EdgeNode** list must be searched to find the insertion point. Again, the edges are stored in sorted order. The user’s **Edge operator**>= is called to determine the insertion point. Notice that as I traverse the list of **EdgeNode**s, I am saving the previous **EdgeNode**’s address to be used in the next insertion.

```
 if (!ptrFrom->ptrEdgeHead) {    // no edges from this one yet
  ptrFrom->ptrEdgeHead = ptrnew; // add at head
  ptrnew->ptrFwd = 0;
  return 0;
 }

 EdgeNode* ptrEdge = ptrFrom->ptrEdgeHead;
 EdgeNode* ptrprev = 0;
 while (ptrEdge &&
        ptrTo->vertexData >= ptrEdge->ptrToVertex->vertexData) {
  ptrprev = ptrEdge;
  ptrEdge = ptrEdge->ptrFwd;
 }

 if (!ptrprev) { // add at head
  ptrnew->ptrFwd = ptrFrom->ptrEdgeHead;
  ptrFrom->ptrEdgeHead = ptrnew;
 }
 else { // add in middle
  ptrprev->ptrFwd = ptrnew;
  ptrnew->ptrFwd = ptrEdge;
 }
 return 0;
}
```

The **DeleteVertex** and **DeleteEdge** functions are straightforward, single linked list operations once the vertex or edge is located. Don't forget to decrement the **indegree** and **outdegree** members of the vertices. The destructor calls **EmptyGraph** which is a very simple function that traverses the list of vertices and for each vertex, deletes all of its edges and then that vertex.

The **DisplayTree** function walks down the list of vertices and for each vertex, displays the edges connected to it. The user must provide a callback function that actually displays each vertex. The callback function is passed a **bool** which indicates whether or not this vertex is in the edges list of a given vertex. The user function is here called **ShowTree**.

```
void Graph::DisplayTree (
          void (*ShowTree) (Vertex& v, bool isConnectedVertex)) {
 if (!ptrHead) return;         // empty graph, so nothing to do
 VertexNode* ptrthis = ptrHead;
 while (ptrthis) {                       // for each VertexNode,
  ShowTree (ptrthis->vertexData, false); // display it
  EdgeNode* ptre = ptrthis->ptrEdgeHead;
  while (ptre) {                         // for each of its edges
   ShowTree (ptre->ptrToVertex->vertexData, true); // display it
   ptre = ptre->ptrFwd;
  }
  ptrthis = ptrthis->ptrFwd;
 }
}
```

Now examine the two traversal methods, **DepthFirstTraversal** and **BreadthFirstTraversal**. Both begin by clearing the visited flags. With the depth first form, the initial vertex is examined first. The main loop continues until all have been visited. What happens when a vertex is actually visited? That is entirely up to the user. The user provides a call-back function, **Process**, that is given each vertex to be actually processed. If a vertex has not been even visited, it is pushed onto the stack. Next, all of the other descendants of this current vertex are popped from the stack and actually processed and all of its descendants or edges that have not yet been visited are pushed onto the stack. Only then do we go on down the actual list of vertices. Thus, we are traversing depth first.

```
void Graph::DepthFirstTraversal (void (*Process) (Vertex& v)){
 if (!ptrHead) return;     // nothing to do
 ClearProcessedFlags ();   // set all flags to NotVisited yet

 Stack<VertexNode> stack;  // create a stack of VertexNode ptrs
 VertexNode* ptrthis = ptrHead;
 while (ptrthis) {
  if (ptrthis->visitedFlag < Processed) {
   if (ptrthis->visitedFlag < Visited) {
    stack.Push (ptrthis);  // push each not yet visited nodes
    ptrthis->visitedFlag = Visited; // but mark them as visited
                                    // as they are pushed
```

```
   }
  }

  // process descendants of this vertex at the top of stack
  while (!stack.IsEmpty()) {
   VertexNode* ptrnode = stack.Pop (); // get most recent Vertex
   Process (ptrnode->vertexData);      // let user process it
   ptrnode->visitedFlag = Processed;   // and mark it processed
   // now traverse all edges of this vertex
   EdgeNode* ptrthisedge = ptrnode->ptrEdgeHead;
   while (ptrthisedge) {
    VertexNode* ptrv = ptrthisedge->ptrToVertex;
    if (ptrv->visitedFlag == NotVisited) {// if this one is not
     stack.Push (ptrv);                   // yet visited, push it
     ptrv->visitedFlag = Visited;
    }
    ptrthisedge = ptrthisedge->ptrFwd;
   }
  }

  // now move on down the VertexNode list to the next Vertex
  ptrthis = ptrthis->ptrFwd;
 }
}
```

In contrast, the **BreadthFirstTraversal** uses a queue to store the **VertexNodes**, so that all of a given vertex's siblings are examined before going on down the edge chain. The coding is very parallel.

```
void Graph::BreadthFirstTraversal (void (*Process) (Vertex& v)) {
 if (!ptrHead) return;     // here, nothing to do
 ClearProcessedFlags ();   // set all flags to NotVisited yet

 Queue<VertexNode> queue;  // our queue of nodes visited FIFO
 VertexNode* ptrthis = ptrHead;
 while (ptrthis) {         // for each VertexNode,
  if (ptrthis->visitedFlag < Processed) {
   if (ptrthis->visitedFlag < Visited) {
    queue.Enqueue (ptrthis); // enqueue NotVisited Vertex and
    ptrthis->visitedFlag = Visited; // mark it as now Visited
   }
  }

  // for each remaining Vertex in the queue, process it
  while (!queue.IsEmpty()) {
   VertexNode* ptrv = queue.Dequeue (); // get next Vertex
   Process (ptrv->vertexData);          // let user process it
   ptrv->visitedFlag = Processed;       // and mark as processed
   EdgeNode* ptre = ptrv->ptrEdgeHead;
```

```
   while (ptre) {                          // for all of its edges,
    VertexNode* ptrve = ptre->ptrToVertex;
    if (ptrve->visitedFlag == NotVisited) {// if it's not visited
     queue.Enqueue (ptrve);                // enqueue this node and
     ptrve->visitedFlag = Visited;         // mark it as now Visited
    }
    ptre = ptre->ptrFwd;
   }
  }

  // move on to the next VertexNode in the list
  ptrthis = ptrthis->ptrFwd;
 }
}
```

Now we have a basic graph class. But as yet, it does not do much for the user. We need some powerhouse advanced functions to make this class fully operational. Here is the first part of the **Graph.cpp** file covering the functions so far discussed.

```
+))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Graph Class Implementation                                           *
/)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
#include "Graph.h"

/***************************************************************/
/*                                                             */
/* Graph: set head pointer to 0                                */
/*                                                             */
/***************************************************************/

Graph::Graph () : ptrHead (0) {}

/***************************************************************/
/*                                                             */
/* AddVertex: add a new vertex to the chain of vertices        */
/*              returns false if out of memory                 */
/*  The vertices list is sorted into increasing user Vertex    */
/*  order - that is, we maintain it as a sorted list           */
/*                                                             */
/***************************************************************/

bool Graph::AddVertex (const Vertex& vertNew) {
 VertexNode* ptrnew = new VertexNode; // allocate a new node

 // fill up node with default values and the user's data
 ptrnew->vertexData = vertNew;
 ptrnew->visitedFlag = NotVisited;
 ptrnew->ptrEdgeHead = 0;
 ptrnew->inShortestPath = false;
 ptrnew->inDegree = ptrnew->outDegree = 0;

```

```
*  // chain into the list of VertexNodes                           *
*  if (!ptrHead) {       // is there anything in the list yet?     *
*   ptrnew->ptrFwd = 0;  // no, so add at head                     *
*   ptrHead = ptrnew;                                              *
*   return true;                                                   *
*  }                                                               *
*                                                                  *
*  // here try to find where in the list this vertex should go     *
*  // by calling user's operator> to find its position             *
*  VertexNode* ptrthis = ptrHead;                                  *
*  VertexNode* ptrprev = 0;                                        *
*  while (ptrthis && vertNew > ptrthis->vertexData) {              *
*   ptrprev = ptrthis;                                             *
*   ptrthis = ptrthis->ptrFwd;                                     *
*  }                                                               *
*                                                                  *
*  // sort out the two cases - at the head or after another vertex *
*  if (!ptrprev) {            // it goes at the very head          *
*   ptrnew->ptrFwd = ptrHead; // us points fwd to the next node    *
*   ptrHead = ptrnew;         // head is now us                    *
*   return true;                                                   *
*  }                                                               *
*                                                                  *
*  // here ptrprev points to the one to insert after               *
*  ptrnew->ptrFwd = ptrprev->ptrFwd; // us points prev's fwd       *
*  ptrprev->ptrFwd = ptrnew;  // previous one points to us         *
*  return true;                                                    *
* }                                                                *
*                                                                  *
* /***************************************************************/*
* /*                                                             */*
* /* FindThisVertex: Given a Vertex, find it in the list         */*
* /*                                                             */*
* /***************************************************************/*
*                                                                  *
* VertexNode* Graph::FindThisVertex (const Vertex& v) const {      *
*  VertexNode* ptrfind = ptrHead;                                  *
*  while (ptrfind && v != ptrfind->vertexData)                     *
*   ptrfind = ptrfind->ptrFwd;                                     *
*  return ptrfind;                                                 *
* }                                                                *
*                                                                  *
* /***************************************************************/*
* /*                                                             */*
* /* FindThisVertex: Given a Vertex, find it in the list         */*
* /*                but also return the previous item in the list*/*
* /*                                                             */*
* /***************************************************************/*
*                                                                  *
* VertexNode* Graph::FindThisVertex (const Vertex& v,              *
*                                   VertexNode*& ptrprev) const { *
*  ptrprev = 0;                                                    *
```

```
 VertexNode* ptrfind = ptrHead;
 while (ptrfind && v != ptrfind->vertexData) {
  ptrprev = ptrfind;
  ptrfind = ptrfind->ptrFwd;
 }
 return ptrfind;
}

/***************************************************************/
/*                                                             */
/* AddEdge: add an edge - vertices must be already added       */
/*                                                             */
/* returns 0 if added, -1 if out of memory, -2 if it cannot    */
/*         find the from vert, -3 if it cannot find to vert    */
/*                                                             */
/***************************************************************/

int Graph::AddEdge (const Vertex& fromVert, const Vertex& toVert,
                    const Edge& edge) {
 // find the from and to vertices in the VertexNode list
 VertexNode* ptrFrom = FindThisVertex (fromVert);
 if (!ptrFrom) return -2;

 VertexNode* ptrTo = FindThisVertex (toVert);
 if (!ptrTo) return -3;

 EdgeNode* ptrnew = new EdgeNode; // allocate a new edge node

 // fill with user's data and the default values
 ptrnew->edgeData = edge;
 ptrnew->inShortestPath = false;


 // increment the vertices degrees and set the EdgeNode's to vert
 ptrFrom->outDegree++;
 ptrTo->inDegree++;
 ptrnew->ptrToVertex = ptrTo;

 // see if this from node has any edges in its list as yet
 if (!ptrFrom->ptrEdgeHead) {    // no edges from this one yet
  ptrFrom->ptrEdgeHead = ptrnew; // add at head
  ptrnew->ptrFwd = 0;
  return 0;
 }

 // here from vert edges exist, so find the insertion point
 // again, calling the user's operator>= to find its location
 EdgeNode* ptrEdge = ptrFrom->ptrEdgeHead;
 EdgeNode* ptrprev = 0;
 while (ptrEdge &&
       ptrTo->vertexData >= ptrEdge->ptrToVertex->vertexData) {
  ptrprev = ptrEdge;
```

```
  ptrEdge = ptrEdge->ptrFwd;                                          *
 }                                                                    *
                                                                      *
 // sort out the two cases - at head and after another EdgeNode      *
 if (!ptrprev) { // add at head                                       *
  ptrnew->ptrFwd = ptrFrom->ptrEdgeHead;                              *
  ptrFrom->ptrEdgeHead = ptrnew;                                      *
 }                                                                    *
 else { // add in middle                                              *
  ptrprev->ptrFwd = ptrnew;                                           *
  ptrnew->ptrFwd = ptrEdge;                                           *
 }                                                                    *
 return 0;                                                            *
}                                                                     *
                                                                      *
/*******************************************************************/*
/*                                                                  */*
/* DeleteVertex: deletes the vertex requested                       */*
/*               returns false if not found or still has edges      */*
/*                                                                  */*
/*******************************************************************/*
                                                                      *
bool Graph::DeleteVertex (const Vertex& hasIdToDel) {                 *
 // find the vertex which has an id key given by hasIdToDel           *
 VertexNode* ptrprev = 0;                                             *
 VertexNode* ptrthis = FindThisVertex (hasIdToDel, ptrprev);          *
                                                                      *
 // quit if Vertex is not found or VertexNode still has Edges         *
 if (!ptrthis) return false;                                          *
 if (ptrthis->ptrEdgeHead) return false;                              *
                                                                      *
 // here it has no EdgeNodes, so it is safe to delete it              *
 if (!ptrprev) // this is the first one                               *
  ptrHead = ptrthis->ptrFwd;                                          *
 else                                                                 *
  ptrprev->ptrFwd = ptrthis->ptrFwd;                                  *
 delete ptrthis;                                                      *
 return true;                                                         *
}                                                                     *
                                                                      *
/*******************************************************************/*
/*                                                                  */*
/* DeleteEdge: deletes the requested EdgeNode                       */*
/*             returns 0 if it is successful                        */*
/*                    -1 if there are no vertices at all            */*
/*                    -2 if vertex is not found                     */*
/*                    -3 if edge is not found                       */*
/*                                                                  */*
/*******************************************************************/*
                                                                      *
int Graph::DeleteEdge (const Vertex& fromVert,                        *
                       const Vertex& toVert) {                        *
```

```
 if (!ptrHead) return -1;

 // find the Edge given by the id keys in the two vertices
 VertexNode* ptrFrom = FindThisVertex (fromVert);
 if (!ptrFrom) return -2;

 // now find the to edge in this list
 EdgeNode* ptrprev = 0;
 EdgeNode* ptredge = ptrFrom->ptrEdgeHead;
 if (!ptredge) return -3; // no edges left!

 // find the required edge by using user's != operator function
 while (ptredge && ptredge->ptrToVertex->vertexData != toVert) {
  ptrprev = ptredge;
  ptredge = ptredge->ptrFwd;
 }
 if (!ptredge) return -3; // did not find the required edge

 // here we found the edge, so decrement vertices' degrees
 ptrFrom->outDegree--;
 ptredge->ptrToVertex->inDegree--;

 if (!ptrprev) // deleting the first one?
  ptrFrom->ptrEdgeHead = ptredge->ptrFwd;
 else
  ptrprev->ptrFwd = ptredge->ptrFwd;
 delete ptredge;
 return 0;     // succesful
}

/***************************************************************/
/*                                                             */
/* ~Graph: remove dynamically allocated memory                 */
/*                                                             */
/***************************************************************/

Graph::~Graph () {
 EmptyGraph ();
}

/***************************************************************/
/*                                                             */
/* EmptyGraph: delete all items so graph can be reused         */
/*                                                             */
/***************************************************************/

void Graph::EmptyGraph () {
 if (!ptrHead) return;          // nothing to do
 VertexNode* ptrthis = ptrHead;
 VertexNode* ptrnext = 0;

 // loop through all vertices
```

```
 while (ptrthis) {
  EdgeNode* ptrthisedge = ptrthis->ptrEdgeHead;
  while (ptrthisedge) {   // delete all edge nodes of this vertex
   EdgeNode* ptrnextedge = ptrthisedge->ptrFwd;
   delete ptrthisedge;
   ptrthisedge = ptrnextedge;
  }
  ptrnext = ptrthis->ptrFwd; // point to next vertex node
  delete ptrthis;            // and delete this vertex node
  ptrthis = ptrnext;
 }
}

/***************************************************************/
/*                                                             */
/* ClearProcessedFlags: set all visited flags to NotVisited    */
/*                                                             */
/***************************************************************/

void Graph::ClearProcessedFlags () {
 if (!ptrHead) return;
 VertexNode* ptrthis = ptrHead;
 while (ptrthis) {
  ptrthis->visitedFlag = NotVisited;
  ptrthis = ptrthis->ptrFwd;
 }
}

/***************************************************************/
/*                                                             */
/* ClearInShortestTreeFlags: clear all inShortestTree flags    */
/*                                                             */
/***************************************************************/

void Graph::ClearInShortestTreeFlags () {
 if (!ptrHead) return;
 VertexNode* ptrthis = ptrHead;
 while (ptrthis) {                    // for each VertexNode,
  ptrthis->inShortestPath = false;    // clear its flag
  EdgeNode* ptre = ptrthis->ptrEdgeHead;
  while (ptre) {                      // for all of its EdgeNodes
   ptre->inShortestPath = false;      // clear its flag
   ptre = ptre->ptrFwd;
  }
  ptrthis = ptrthis->ptrFwd;
 }
}

/***************************************************************/
/*                                                             */
/* DisplayTree: Display a representation of the tree for debug */
/*  requires user callback function to do actual displaying    */
```

```
/*                                                                */
/******************************************************************/

void Graph::DisplayTree (
          void (*ShowTree) (Vertex& v, bool isConnectedVertex)) {
 if (!ptrHead) return;          // empty graph, so nothing to do
 VertexNode* ptrthis = ptrHead;
 while (ptrthis) {                           // for each VertexNode,
  ShowTree (ptrthis->vertexData, false); // display it
  EdgeNode* ptre = ptrthis->ptrEdgeHead;
  while (ptre) {                             // for each of its edges
   ShowTree (ptre->ptrToVertex->vertexData, true); // display it
   ptre = ptre->ptrFwd;
  }
  ptrthis = ptrthis->ptrFwd;
 }
}

/******************************************************************/
/*                                                                */
/* DepthFirstTraversal: perfom a depth first traversal            */
/*   requires a user callback function to perform any desired     */
/*   work on each node found                                      */
/*                                                                */
/******************************************************************/

void Graph::DepthFirstTraversal (void (*Process) (Vertex& v)){
 if (!ptrHead) return;     // nothing to do
 ClearProcessedFlags ();   // set all flags to NotVisited yet

 Stack<VertexNode> stack;  // create a stack of VertexNode ptrs
 VertexNode* ptrthis = ptrHead;
 while (ptrthis) {
  if (ptrthis->visitedFlag < Processed) {
   if (ptrthis->visitedFlag < Visited) {
    stack.Push (ptrthis);  // push each not yet visited nodes
    ptrthis->visitedFlag = Visited; // but mark them as visited
                                    // as they are pushed
   }
  }

  // process descendants of this vertex at the top of stack
  while (!stack.IsEmpty()) {
   VertexNode* ptrnode = stack.Pop (); // get most recent Vertex
   Process (ptrnode->vertexData);      // let user process it
   ptrnode->visitedFlag = Processed;   // and mark it processed
   // now traverse all edges of this vertex
   EdgeNode* ptrthisedge = ptrnode->ptrEdgeHead;
   while (ptrthisedge) {
    VertexNode* ptrv = ptrthisedge->ptrToVertex;
    if (ptrv->visitedFlag == NotVisited) {// if this one is not
     stack.Push (ptrv);                  // yet visited, push it
```

```
*     ptrv->visitedFlag = Visited;                                    *
*    }                                                                *
*    ptrthisedge = ptrthisedge->ptrFwd;                               *
*   }                                                                 *
*  }                                                                  *
*                                                                     *
*  // now move on down the VertexNode list to the next Vertex         *
*  ptrthis = ptrthis->ptrFwd;                                         *
* }                                                                   *
*}                                                                    *
*                                                                     *
*/***************************************************************/*
*/*                                                             */*
*/* BreadthFirstTraversal: perform a breadth first traversal    */*
*/*    requires a user callback function to process each vertex */*
*/*                                                             */*
*/***************************************************************/*
*                                                                     *
*void Graph::BreadthFirstTraversal (void (*Process) (Vertex& v)) {*
* if (!ptrHead) return;     // here, nothing to do                    *
* ClearProcessedFlags ();   // set all flags to NotVisited yet        *
*                                                                     *
* Queue<VertexNode> queue;  // our queue of nodes visited FIFO        *
* VertexNode* ptrthis = ptrHead;                                      *
* while (ptrthis) {         // for each VertexNode,                   *
*  if (ptrthis->visitedFlag < Processed) {                            *
*   if (ptrthis->visitedFlag < Visited) {                             *
*    queue.Enqueue (ptrthis); // enqueue NotVisited Vertex and        *
*    ptrthis->visitedFlag = Visited; // mark it as now Visited        *
*   }                                                                 *
*  }                                                                  *
*                                                                     *
*  // for each remaining Vertex in the queue, process it              *
*  while (!queue.IsEmpty()) {                                         *
*   VertexNode* ptrv = queue.Dequeue (); // get next Vertex           *
*   Process (ptrv->vertexData);          // let user process it       *
*   ptrv->visitedFlag = Processed;       // and mark as processed *
*   EdgeNode* ptre = ptrv->ptrEdgeHead;                               *
*   while (ptre) {                       // for all of its edges, *
*    VertexNode* ptrve = ptre->ptrToVertex;                           *
*    if (ptrve->visitedFlag == NotVisited) {// if it's not visited*
*     queue.Enqueue (ptrve);             // enqueue this node and *
*     ptrve->visitedFlag = Visited;      // mark it as now Visited*
*    }                                                                *
*    ptre = ptre->ptrFwd;                                             *
*   }                                                                 *
*  }                                                                  *
*                                                                     *
*  // move on to the next VertexNode in the list                      *
*  ptrthis = ptrthis->ptrFwd;                                         *
* }                                                                   *
*}                                                                    *
```

Advanced Graph Functions

The first question a client program might have is, “Does a path exist between two vertices?” For illustration’s sake, I implement two versions of this operation based upon the two types of **Graph** traversals. The first, **DoesPathExistBetween_DepthFirst**, uses a depth first approach.

It is given the “from” and “to” vertices and returns either **true**, a path exists, or **false**. It begins by finding the “from” vertex in the list of vertices. First, the “from” vertex is pushed onto the stack. The entire process is repeated until the stack is empty (not found) or we find the “to” vertex that is connected directly or via other vertices to the “from” vertex. The process begins by popping the next vertex to try off of the stack. Then, the user’s **Vertex** structure’s **operator==** is called to see if this is the “to” **Vertex**. If it is, we return **true** and are done. If not, then we check to see if this vertex has already been visited. If it has, it is, of course, skipped. If it has not yet been visited, then all of its **EdgeNode**s are enqueued into a queue. Once the queue is built, then each vertex in the queue is examined to see if it has been visited. If not, it is pushed onto the stack to be tried. And the process is repeated using the next vertex in the stack.

```
bool Graph::DoesPathExistBetween_DepthFirst
                         (const Vertex& from, const Vertex& to) {
 if (!ptrHead) return false; // an empty graph

 // try to find the from vertex
 VertexNode* ptrfrom = FindThisVertex (from);
 if (!ptrfrom) return false;

 // from Vertex is found, so now try to find a path to "to" vert.
 Stack<VertexNode> stack;    // stack of vertices to try
 ClearProcessedFlags ();     // set all flags to NotVisited
 Queue<VertexNode> queue;    // queue of vertices to try next
 bool found = false;         // found is true when a path exists
 stack.Push (ptrfrom);       // store initial from vertex
 VertexNode* ptrthis;
 do {
  ptrthis = stack.Pop ();    // pop next vertex to try
  // call user's operator= function to look for the "to" vertex
  if (ptrthis->vertexData == to) {
   found = true;             // it was found, so we are done
   break;
  }

  // this vertex is not it, so if it has not yet been visited,
  // enqueue all of its edges and try them
  if (ptrthis->visitedFlag == NotVisited) {
   ptrthis->visitedFlag = Visited;
   EdgeNode* ptre = ptrthis->ptrEdgeHead;
   while (ptre) { // enqueues all of this vertex's edges
    queue.Enqueue (ptre->ptrToVertex);
```

```
    ptre = ptre->ptrFwd;
   }
   // now try each edge. If an edge has not yet been visited,
   // push that vertex onto the stack to be tried later on
   while (!queue.IsEmpty()) {
    VertexNode* ptrv = queue.Dequeue ();
    if (ptrv->visitedFlag == NotVisited)
     stack.Push (ptrv);
   }
  }
 } while (!stack.IsEmpty() && !found); // repeat for all vertices
 return found;
}
```

In the **DoesPathExistBetween_BreathFirst** function, a queue replaces the stack, since we wish to test all of the siblings before we go deeper into the tree. Its coding is parallel to what we have seen before.

```
bool Graph::DoesPathExistBetween_BreadthFirst
                         (const Vertex& from, const Vertex& to) {
 if (!ptrHead) return false; // no vertices in the graph

 // try to find the from vertex, returning false if not found
 VertexNode* ptrfrom = FindThisVertex (from);
 if (!ptrfrom) return false;

 ClearProcessedFlags ();  // set all flags as NotVisited yet
 bool found = false;      // true when we have found the "to"

 Queue<VertexNode> queue1; // the main queue to check
 queue1.Enqueue (ptrfrom); // store the first vertex

 Queue<VertexNode> queue2; // secondary to try queue
 VertexNode* ptrthis;
 do {
  ptrthis = queue1.Dequeue (); // retrieve next vertex to try
  // call the user's operator== function to see it this is it
  if (ptrthis->vertexData == to) {
   found = true;           // we have found the "to" vertex!
   break;
  }

  // this one is not it, if this vertex has not yet been visited
  if (ptrthis->visitedFlag == NotVisited) { // then visit it
   ptrthis->visitedFlag = Visited;
   EdgeNode* ptre = ptrthis->ptrEdgeHead;

   // enqueue all of this vertex's edges
```

```
   while (ptre) {
    queue2.Enqueue (ptre->ptrToVertex);
    ptre = ptre->ptrFwd;
   }

   // now check all of this vertex's edges - if any are not yet
   // visited, then add them to the main queue to be visited
   while (!queue2.IsEmpty()) {
    VertexNode* ptrv = queue2.Dequeue ();
    if (ptrv->visitedFlag == NotVisited)
     queue1.Enqueue (ptrv);
   }
  }
 } while (!queue1.IsEmpty() && !found); // repeat for all vertex
 return found;
}
```

A client program may wish to create a minimum spanning tree. Recall that this can only be done if the edges have a weight associated with them. The minimum spanning tree is a network such that all of its edge weights are guaranteed to be the minimum value. Remember that one use for this spanning tree is to find the shortest cabling required to tie a series of networked computers together.

The general process is: from all of the vertices in the tree, select the edge with the minium distance to a vertex not currently in the tree and add it (flag it) to the minimal tree. This process is illustrated in the next series of figures. Consider the network shown in Figure 16.19 below.

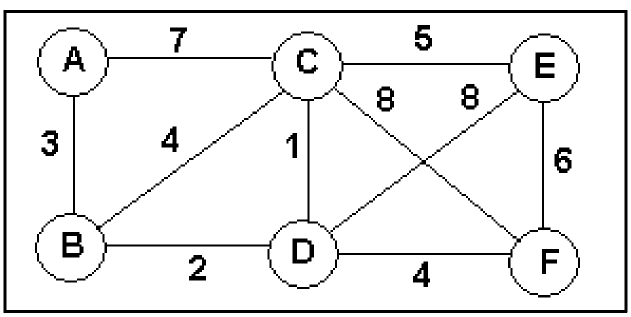


Figure 13.19 A Network with Weighted Edges

We start with the initial vertex A and find the shortest path to vertex B. Then we find the shortest path to C which goes through D. Part way through the process, we now have the following nodes added to the minimal spanning tree.

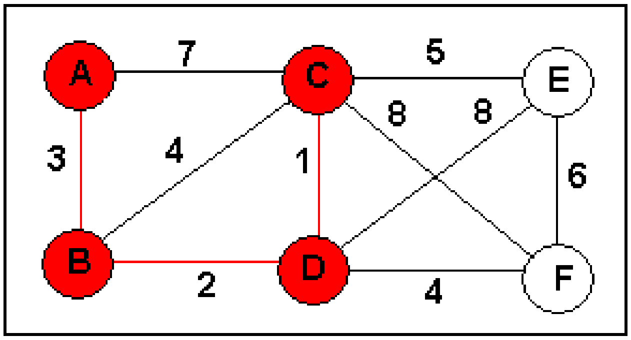


Figure 13.20 The First Four Minimum Nodes

We continue with the process. The shortest distance to vertex E is from C and to F is from D. Thus, we end up with the following minimum spanning tree shown in red below in Figure 13.21.

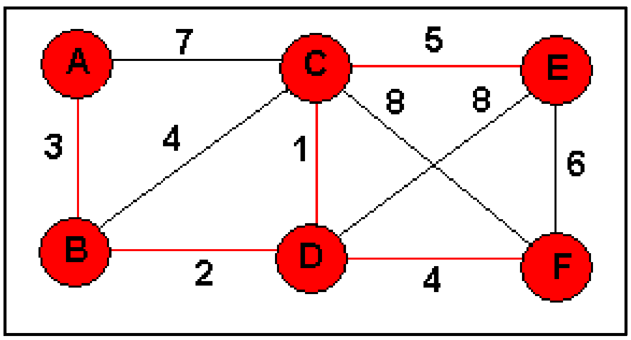


Figure 13.21 The Minimum Spanning Tree

The implementation is in two parts. The first step is to build the minimum spanning tree and the second is to display it in some manner. Have you spotted the one piece of information that the graph functions cannot possibly know? If we are to find the minimum weight, what kind of data is that weight? And what is the largest value that kind of data can have? Ok, if the distance was a **double** that represents miles, then what is the largest value it can have, since we need to find distances that are less than this? Thus, we must have the user provide the build function with an **Edge** structure that contains the largest possible weight value in this situation.

Unlike the traversal methods that need a “visited” flag for the duration of the traversal, here we need to retain the state of being in the minimum spanning tree until the user is finished using the graph. Thus, I chose to have another member of our nodes keep track of whether or not this item is in the minimum spanning tree. It is the **bool inShortestPath** found in both the **VertexNode** and **EdgeNode** structure.

The **BuildMinimumSpanningTree** function is passed an **Edge** that contains the maximum distance. The function first clears all of the **inShortestPath bool**s. The process begins with the head vertex of the list and processes all of the vertices. Within the outer loop, I define a minimum Edge instance as containing the maximum Edge value. Now, we look at all of the edges connected to this vertex and find the minimum distance edge for any one that is not already in the shortest path. If we find one that is smaller than the currently smallest one, I save a pointer to the found one and adjust the minium distance downward. If one is found, then I set both the “from” and “to” vertices’ **inShortestPath** members to **true**. Notice that I must rely on the user’s **Edge operator**< function.

```
void Graph::BuildMinimumSpanningTree (Edge& maxEdgeValue) {
 ClearInShortestTreeFlags ();    // clear all span flags
 if (!ptrHead) return;           // here there is nothing to do
 VertexNode* ptrthis = ptrHead;  // begin with the first vertex
 ptrthis->inShortestPath = true; // set in shortest path
 bool treeComplete = false;
 while (!treeComplete) {         // repeat until tree is done
  // assume it's done unless we find another one
  treeComplete = true;
  VertexNode* ptrcheck = ptrthis; // check this one out
  EdgeNode* ptrMinEdge = 0;
  Edge minEdge = maxEdgeValue;    // set to smallest value
  while (ptrcheck) {
   // if this one is in the shortest path and has edges, then
   if (ptrcheck->inShortestPath && ptrcheck->outDegree > 0) {
    EdgeNode* ptre = ptrcheck->ptrEdgeHead; // process all edges
    while (ptre) {
     if (!ptre->ptrToVertex->inShortestPath) { // if it is not,
      treeComplete = false;       // then we must check it out
      // call user's op< function to check if this edge is < min
      if (ptre->edgeData < minEdge) {
       minEdge = ptre->edgeData;   // it is, so update the min
       ptrMinEdge = ptre;
      }
     }
     ptre = ptre->ptrFwd;          // repeat for all edges
    }
   }
   ptrcheck = ptrcheck->ptrFwd;    // repeat for all verts
  }
  if (ptrMinEdge) {                // if we found one,
   ptrMinEdge->inShortestPath = true; // flag being in shortest
   ptrMinEdge->ptrToVertex->inShortestPath = true; // path
  }
 }
}
```

With the minimum spanning tree build, **ShowMinimumSpanningTree** can be used to display the resultant tree. The caller provides a callback function that is passed a pair of pair of minimum spanning vertices and the distance between them.

```
void Graph::ShowMinimumSpanningTree (
       void (*DisplayEdge) (const Vertex& from, const Vertex& to,
```

```
                                   const Edge& edge)) {
 if (!ptrHead) return; // an empty graph

 VertexNode* ptrthis = ptrHead; // loop through all vertices
 while (ptrthis) {
  EdgeNode* ptre = ptrthis->ptrEdgeHead; // loop thru all edges
  while (ptre) {
   if (ptre->inShortestPath)  // if in shortest path, display it
    DisplayEdge (ptrthis->vertexData,
                ptre->ptrToVertex->vertexData, ptre->edgeData);
   ptre = ptre->ptrFwd;
  }
  ptrthis = ptrthis->ptrFwd;
 }
}
```

The next likely question we will be asked is “What is the shortest path between two vertices?” The caller passes our function a “from” and “to” **Vertex** instances; we must find the minimum path between them. Finding the minimum path between two vertices is much like the other two traversal methods. However, the stack and queue which were used before are now replaced by a priority queue. That is, when we must order the vertices to search by priority based upon the smallest weight. In other words, when we queue up vertices to try, we always want that vertex with the smallest distance from the current one to be at the front of the queue to try next.

Our priority queue is derived from the **Heap** class and this poses a new problem. When the heap is rebuilt, it places the largest value item at element 0. So if we blindly check is any item is less than another item, we will have the heap backwards! On the other hand, sometimes, the minimum value, from the user’s position is actually the larger value. Rather than locking our solution into either “smallest value is largest” or “largest value is smallest,” I let the user notify us of the situation via a **bool**, **smallestIsHighest**. Then, if I relay that state to all of the items, then the comparison operators can return the proper result no matter which way the user desires. Again, this makes a more generalized solution.

Typically, these shortest path algorithms, display all possible shortest paths from a given vertex. While this extra information is sometimes useful, normally, the user wants to just see that shortest path from A to B. Hence, the function is passed another **bool**, **showOnlyShortest**, which is **true** if the user only wants to see the actual shortest path from A to B.

The caller must also provide a callback function to receive pairs of **Vertex** structures and the minimum distance between them. The final item that is required is the smallest value that the user’s distance item can hold. Since we do not know its data type, the caller passes an **Edge** structure that contains the smallest distance value. Notice that this is the opposite of the previous function which required the largest value.

In order to handle this process, we need a helper structure to organize the results. I call it **ItemNode**. It contains the "from" and "to" **VertexNode** pointers along with the **Edge** distance between these and the order long which the priority queue uses when two items have equal priority and the **smallestIsHighest** flag. Notice how I have implemented the two different sets of comparison operator results, depending on the setting of **smallestIsHighest**.

```
/***************************************************************/
/*                                                             */
/* ItemNode: helper struct for finding shortest distances      */
/*                                                             */
/* because of heap, the largest value is at top - so we must   */
/* reverse test results is smallest is the highest value       */
/*                                                             */
/***************************************************************/

struct ItemNode {
 VertexNode* ptrFromVertex;
 VertexNode* ptrToVertex;
 bool        smallestIsHighest;
 long        order;
 Edge        distance;
 bool operator< (const ItemNode& i2) const;
 bool operator<= (const ItemNode& i2) const;
};

bool ItemNode::operator< (const ItemNode& i2) const {
 if (smallestIsHighest) {
  if (distance < i2.distance) return false;
  if (distance == i2.distance)
   return order < i2.order ? false : true;
  return true;
 }
 else {
  if (distance < i2.distance) return true;
  if (distance == i2.distance)
   return order < i2.order ? true : false;
  return false;
 }
}

bool ItemNode::operator<= (const ItemNode& i2) const {
 if (smallestIsHighest) {
  if (distance < i2.distance) return false;
  if (distance == i2.distance)
   return order < i2.order ? false : true;
  return true;
 }
 else {
```

```
  if (distance < i2.distance) return true;
  if (distance == i2.distance)
   return order < i2.order ? true : false;
  return false;
 }
}
```

This **FindShortestPath** function is the longest function in the class. It is composed of two sections: finding all of the shortest paths from a given vertex and then finding and showing just the path desired in the correct order of vertices from the "from" vertex to the "to" vertex.

The finding the shortest path uses a **PriorityQueue** in place of the stack and uses a queue to store the ones to try next just as in the previous examples. However, when an item is found to be in the shortest path sequence, it is copied and placed into a linked list of answers for use in the second half of the function. Thus, when the first half of the processing is finished, the **answers** list contains a collection of **ItemNode** structures each with a "from" and "to" set of nodes and the accumulated distance between the original "from" vertex and the current to vertex.

FindShortestPath begins by finding the "from" vertex in the list of vertices. If it is found, then all of the visited flags are cleared. The **order long** is initialized to 1, in case there are duplicate distances to be priority enqueued.

```
bool Graph::FindShortestPath (const Vertex& from,
                              const Vertex& to,
                              const Edge& minDist,
                              bool showOnlyShortest,
                              bool smallestIsHighest,
 void (*DisplayShortestPath) (const Vertex& from,
                              const Vertex& to,
                              const Edge& distance) ) {
 if (!ptrHead) return false; // nothing to do

 // find the from vertex
 VertexNode* ptrfrom = FindThisVertex (from);
 if (!ptrfrom) return false; // nothing to do

 ClearProcessedFlags ();
 // order is required in case queue items have same priority
 long order = 1;
```

Next, an **ItemNode** instance, called **item**, is initialized to the starting node. A minimum distance Edge structure is initialized to the minimum value a user's distance can have. And an instance of the **PriorityQueue**, **Queue**, and **DoubleLinkedList** classes are created and this original **item** is priority enqueued.

```
 ItemNode item; // setup the initial beginning node
 item.smallestIsHighest = smallestIsHighest;
 item.ptrFromVertex = ptrfrom;
```

```
item.ptrToVertex = ptrfrom;
item.distance = minDist;
item.order = order++;
Edge minimumDistance = minDist; // set min dist to default min
PriorityQueue<ItemNode> pqueue;
Queue<VertexNode> queue;
DoubleLinkedList<ItemNode> answers;

pqueue.Enqueue (item);// put this first item into priority queue
bool failed = false;  // set to true if we encounter an internal
                      // error
```

The main loop dequeues the highest priority item. If it is not yet visited, it is handled as follows. It is marked as visited and a new **ItemNode** structure is allocated and the current **item** instance is copied into it and it is added to the tail of the **answers** list. Note that this first answer contains a “from” and “to” vertex which are the same value, the “from” vertex and the accumulated distance is the minimum value an **Edge** can have, usually 0. With this node saved in the answer list, I now change the “from” destination of the **item** to be the “to” vertex and set the currently found **minimumDistance** to the currently found **item distance**. Since I made a copy of the original state of **item**, the answer **ItemNode** does not get altered by this process.

Next, I do a normal enqueue of all of the edges of this current vertex.

```
do {
 pqueue.Dequeue (item); // get highest priotity vertex to check
 // if it is not yet visited, handle it
 if (item.ptrToVertex->visitedFlag == NotVisited) {
  item.ptrToVertex->visitedFlag = Visited;
  ItemNode* ptrqi = new ItemNode; // copy current item node and
  *ptrqi = item;                  // add it to the answers list
  answers.AddAtTail (ptrqi);
  item.ptrFromVertex = item.ptrToVertex; // reset from vertex
  minimumDistance = item.distance;       // store new min dist
  // now queue up all of its edges
  EdgeNode* ptre = item.ptrFromVertex->ptrEdgeHead;
  while (ptre) {
   queue.Enqueue (ptre->ptrToVertex);
   ptre = ptre->ptrFwd;
  }
```

Now, we must examine all edges in turn that are queued up. If any are not yet visited, I must find that node’s list of **EdgeNodes** to check. Here, I used a helper function, **FindThisEdge** which returns a pointer to the found **EdgeNode** structure. **FindThisEdge** is given the two vertices and it then finds the corresponding **EdgeNode** between them by finding the “from” vertex in the main list of vertices and then searches its list of **EdgeNode** structures looking for a match.

```
  while (!queue.IsEmpty ()) {
```

```
    VertexNode* ptrthis = queue.Dequeue ();
    if (ptrthis->visitedFlag == NotVisited) {
     item.ptrToVertex = ptrthis;
     EdgeNode* ptree = FindThisEdge (
                                 item.ptrFromVertex->vertexData,
                                 ptrthis->vertexData);
     if (!ptree) { // here we cannot find the requested edge
      failed = true;
      break;
     }
```

Having found the edge between these two, we add in the edge's distance into the **item**'s accumulated **distance**. After incrementing the count, this item is then priority enqueued.

```
     // add in the distance to this edge
     item.distance = minimumDistance + ptree->edgeData;
     item.order = order++;
     pqueue.Enqueue (item); // add this one to the priority queue
    }
   }
  }
 } while (!failed && !pqueue.IsArrayEmpty ());
```

When all items have been processed, the **answers** list contains a series of all possible minimum distances from the original "from" vertex. To understand what is in the list of **answers**, refer to Figure 13.21 The Minimum Spanning Tree above. The red lines represent the minimum paths. Suppose that we called **FindShortestPath** passing it vertex A and E. The list of items in the **answers** linked list for this graph would be as follows.

A A 0
A B 3
B D 5
D C 6
C E 11 <---
D F 9

Sometimes, the user wants all of this information. But usually, they desire only the shortest path, in this case from A to E.

The second half of the function either displays all of the results or finds only the shortest path and then shows it. The algorithm to find the shortest path out of this set of results is simple. Search the list of items looking for the "to" vertex in the "to" column. I indicated that one with an arrow above. Push that item onto a stack. Now, we got to E by "from" vertex C, so look from the beginning of the list for a "to" vertex of C. Push that one onto the stack. We got there using a "from"vertex of D, so look from the beginning for a "to" vertex of D and push that one onto the stack. The process is repeated until we push onto the stack an **ItemNode** whose "from" vertex is the original "from" vertex, here A. Finally, to display the path, just pop each item off in turn and display it. Using the above we would produce the following.

A B 3
B D 5
D C 6
C E 11
And this is the shortest path from A to E. And it is presented to the user in a manner that they can effectively use.

```
 ItemNode* ptri;
 if (!failed) {
  answers.ResetToHead ();
  ptri = answers.GetCurrentNode ();
  if (showOnlyShortest) { // if we want to show only the shortest
   Vertex findThisOne = to; // path, then begin by finding the to
   Stack<ItemNode> path;    // node and push it on the stack
   answers.ResetToHead ();  // then find how we got to it
   ptri = answers.GetCurrentNode (); // and so on til we get to
   while (ptri) {           // the from vertex
    if (ptri->ptrToVertex->vertexData == findThisOne) {
     path.Push (ptri);
     if (ptri->ptrFromVertex->vertexData == from)
      break;
     findThisOne = ptri->ptrFromVertex->vertexData;
     answers.ResetToHead ();
    }
    else
     answers.Next();
    ptri = answers.GetCurrentNode ();
   }
   // now poping off the vertices shows the path from-to
   ptri = path.Pop ();
   while (ptri) {
    DisplayShortestPath (ptri->ptrFromVertex->vertexData,
                  ptri->ptrToVertex->vertexData, ptri->distance);
    ptri = path.Pop ();
   }
  }
  // otherwise, user wants all shortest paths found
  else {
   while (ptri) {
    DisplayShortestPath (ptri->ptrFromVertex->vertexData,
                  ptri->ptrToVertex->vertexData, ptri->distance);
    answers.Next ();
    ptri = answers.GetCurrentNode ();
   }
  }
 }
 return true;
}
```

Here is the remainder of the **Graph.cpp** file with the advanced functions. For completeness, I also show the other template container classes that are used.

```
/***************************************************************/*
/*                                                             */*
/* DoesPathExistBetween_DepthFirst: returns true if a path     */*
/*                   exists between the two indicated Vertices */*
/*                                                             */*
/***************************************************************/*
                                                                 *
bool Graph::DoesPathExistBetween_DepthFirst                      *
                         (const Vertex& from, const Vertex& to) {*
 if (!ptrHead) return false; // an empty graph                   *
                                                                 *
 // try to find the from vertex                                  *
 VertexNode* ptrfrom = FindThisVertex (from);                    *
 if (!ptrfrom) return false;                                     *
                                                                 *
 // from Vertex is found, so now try to find a path to "to" vert.*
 Stack<VertexNode> stack;    // stack of vertices to try         *
 ClearProcessedFlags ();     // set all flags to NotVisited      *
 Queue<VertexNode> queue;    // queue of vertices to try next    *
 bool found = false;         // found is true when a path exists *
 stack.Push (ptrfrom);       // store initial from vertex        *
 VertexNode* ptrthis;                                            *
 do {                                                            *
  ptrthis = stack.Pop ();    // pop next vertex to try           *
  // call user's operator= function to look for the "to" vertex  *
  if (ptrthis->vertexData == to) {                               *
   found = true;             // it was found, so we are done     *
   break;                                                        *
  }                                                              *
                                                                 *
  // this vertex is not it, so if it has not yet been visited,   *
  // enqueue all of its edges and try them                       *
  if (ptrthis->visitedFlag == NotVisited) {                      *
   ptrthis->visitedFlag = Visited;                               *
   EdgeNode* ptre = ptrthis->ptrEdgeHead;                        *
   while (ptre) { // enqueues all of this vertex's edges         *
    queue.Enqueue (ptre->ptrToVertex);                           *
    ptre = ptre->ptrFwd;                                         *
   }                                                             *
   // now try each edge. If an edge has not yet been visited,    *
   // push that vertex onto the stack to be tried later on       *
   while (!queue.IsEmpty()) {                                    *
    VertexNode* ptrv = queue.Dequeue ();                         *
    if (ptrv->visitedFlag == NotVisited)                         *
     stack.Push (ptrv);                                          *
   }                                                             *
  }                                                              *
 } while (!stack.IsEmpty() && !found); // repeat for all vertices*
 return found;                                                   *
```

```
}                                                                *
                                                                 *
/***************************************************************/*
/*                                                             */*
/* DoesPathExistBetween_BreadthFirst: returns true if a path   */*
/*                   exists between the two indicated Vertices */*
/*                                                             */*
/***************************************************************/*
                                                                 *
bool Graph::DoesPathExistBetween_BreadthFirst                    *
                         (const Vertex& from, const Vertex& to) {*
 if (!ptrHead) return false; // no vertices in the graph         *
                                                                 *
 // try to find the from vertex, returning false if not found    *
 VertexNode* ptrfrom = FindThisVertex (from);                    *
 if (!ptrfrom) return false;                                     *
                                                                 *
 ClearProcessedFlags ();  // set all flags as NotVisited yet     *
 bool found = false;      // true when we have found the "to"    *
                                                                 *
 Queue<VertexNode> queue1; // the main queue to check            *
 queue1.Enqueue (ptrfrom); // store the first vertex             *
                                                                 *
 Queue<VertexNode> queue2; // secondary to try queue             *
 VertexNode* ptrthis;                                            *
 do {                                                            *
  ptrthis = queue1.Dequeue (); // retrieve next vertex to try    *
  // call the user's operator== function to see it this is it    *
  if (ptrthis->vertexData == to) {                               *
   found = true;           // we have found the "to" vertex!     *
   break;                                                        *
  }                                                              *
                                                                 *
  // this one is not it, if this vertex has not yet been visited *
  if (ptrthis->visitedFlag == NotVisited) { // then visit it     *
   ptrthis->visitedFlag = Visited;                               *
   EdgeNode* ptre = ptrthis->ptrEdgeHead;                        *
                                                                 *
   // enqueue all of this vertex's edges                         *
   while (ptre) {                                                *
    queue2.Enqueue (ptre->ptrToVertex);                          *
    ptre = ptre->ptrFwd;                                         *
   }                                                             *
                                                                 *
   // now check all of this vertex's edges - if any are not yet  *
   // visited, then add them to the main queue to be visited     *
   while (!queue2.IsEmpty()) {                                   *
    VertexNode* ptrv = queue2.Dequeue ();                        *
    if (ptrv->visitedFlag == NotVisited)                         *
     queue1.Enqueue (ptrv);                                      *
   }                                                             *
  }                                                              *
```

```
 } while (!queue1.IsEmpty() && !found); // repeat for all vertex *
 return found;                                                   *
}                                                                *
                                                                 *
/***************************************************************/*
/*                                                             */*
/* BuildMinimumSpanningTree: construct a min span tree         */*
/*                                                             */*
/***************************************************************/*
                                                                 *
void Graph::BuildMinimumSpanningTree (Edge& maxEdgeValue) {      *
 ClearInShortestTreeFlags ();    // clear all span flags         *
 if (!ptrHead) return;           // here there is nothing to do  *
 VertexNode* ptrthis = ptrHead;  // begin with the first vertex  *
 ptrthis->inShortestPath = true; // set in shortest path         *
 bool treeComplete = false;                                      *
 while (!treeComplete) {         // repeat until tree is done    *
  // assume it's done unless we find another one                 *
  treeComplete = true;                                           *
  VertexNode* ptrcheck = ptrthis; // check this one out          *
  EdgeNode* ptrMinEdge = 0;                                      *
  Edge minEdge = maxEdgeValue;    // set to smallest value       *
  while (ptrcheck) {                                             *
   // if this one is in the shortest path and has edges, then    *
   if (ptrcheck->inShortestPath && ptrcheck->outDegree > 0) {    *
    EdgeNode* ptre = ptrcheck->ptrEdgeHead; // process all edges *
    while (ptre) {                                               *
     if (!ptre->ptrToVertex->inShortestPath) { // if it is not,  *
      treeComplete = false;        // then we must check it out  *
      // call user's op< function to check if this edge is < min *
      if (ptre->edgeData < minEdge) {                            *
       minEdge = ptre->edgeData;   // it is, so update the min   *
       ptrMinEdge = ptre;                                        *
      }                                                          *
     }                                                           *
     ptre = ptre->ptrFwd;          // repeat for all edges       *
    }                                                            *
   }                                                             *
   ptrcheck = ptrcheck->ptrFwd;    // repeat for all verts       *
  }                                                              *
  if (ptrMinEdge) {                // if we found one,           *
   ptrMinEdge->inShortestPath = true; // flag being in shortest  *
   ptrMinEdge->ptrToVertex->inShortestPath = true; // path       *
  }                                                              *
 }                                                               *
}                                                                *
                                                                 *
/***************************************************************/*
/*                                                             */*
/* ShowMinimumSpanningTree: display the resultant min span tree*/*
/*                                                             */*
/***************************************************************/*
```

```

void Graph::ShowMinimumSpanningTree (
       void (*DisplayEdge) (const Vertex& from, const Vertex& to,
                            const Edge& edge)) {
 if (!ptrHead) return; // an empty graph

 VertexNode* ptrthis = ptrHead; // loop through all vertices
 while (ptrthis) {
  EdgeNode* ptre = ptrthis->ptrEdgeHead; // loop thru all edges
  while (ptre) {
   if (ptre->inShortestPath)  // if in shortest path, display it
    DisplayEdge (ptrthis->vertexData,
                 ptre->ptrToVertex->vertexData, ptre->edgeData);
   ptre = ptre->ptrFwd;
  }
  ptrthis = ptrthis->ptrFwd;
 }
}

/***************************************************************/
/*                                                             */
/* ItemNode: helper struct for finding shortest distances      */
/*                                                             */
/* because of heap, the largest value is at top - so we must   */
/* reverse test results is smallest is the highest value       */
/*                                                             */
/***************************************************************/

struct ItemNode {
 VertexNode* ptrFromVertex;
 VertexNode* ptrToVertex;
 bool        smallestIsHighest;
 long        order;
 Edge        distance;
 bool operator< (const ItemNode& i2) const;
 bool operator<= (const ItemNode& i2) const;
};

bool ItemNode::operator< (const ItemNode& i2) const {
 if (smallestIsHighest) {
  if (distance < i2.distance) return false;
  if (distance == i2.distance)
   return order < i2.order ? false : true;
  return true;
 }
 else {
  if (distance < i2.distance) return true;
  if (distance == i2.distance)
   return order < i2.order ? true : false;
  return false;
 }
}
```

```
bool ItemNode::operator<= (const ItemNode& i2) const {
 if (smallestIsHighest) {
  if (distance < i2.distance) return false;
  if (distance == i2.distance)
   return order < i2.order ? false : true;
  return true;
 }
 else {
  if (distance < i2.distance) return true;
  if (distance == i2.distance)
   return order < i2.order ? true : false;
  return false;
 }
}

/***************************************************************/
/*                                                             */
/* FindShortestPath: calcs the shortest path from - to verts   */
/*                                                             */
/* Caller provides an Edge that is storing the minimum value   */
/* that that data type can hold                                */
/*                                                             */
/* if showOnlyShortest, then only display that path            */
/* otherwise, show all the shortest paths for all from "from"  */
/*                                                             */
/* if smallestIsHighest, we must reverse the comparison op's   */
/* results so that the "highest" is in heap element [0]       */
/*                                                             */
/***************************************************************/

bool Graph::FindShortestPath (const Vertex& from,
                              const Vertex& to,
                              const Edge& minDist,
                              bool showOnlyShortest,
                              bool smallestIsHighest,
 void (*DisplayShortestPath) (const Vertex& from,
                              const Vertex& to,
                              const Edge& distance) ) {
 if (!ptrHead) return false; // nothing to do

 // find the from vertex
 VertexNode* ptrfrom = FindThisVertex (from);
 if (!ptrfrom) return false; // nothing to do

 ClearProcessedFlags ();
 // order is required in case queue items have same priority
 long order = 1;

 ItemNode item; // setup the initial beginning node
 item.smallestIsHighest = smallestIsHighest;
 item.ptrFromVertex = ptrfrom;
```

```
item.ptrToVertex = ptrfrom;
item.distance = minDist;
item.order = order++;
Edge minimumDistance = minDist; // set min dist to default min
PriorityQueue<ItemNode> pqueue;
Queue<VertexNode> queue;
DoubleLinkedList<ItemNode> answers;

pqueue.Enqueue (item);// put this first item into priority queue
bool failed = false;  // set to true if we encounter an internal
                      // error
do {
 pqueue.Dequeue (item); // get highest priotity vertex to check
 // if it is not yet visited, handle it
 if (item.ptrToVertex->visitedFlag == NotVisited) {
  item.ptrToVertex->visitedFlag = Visited;
  ItemNode* ptrqi = new ItemNode; // copy current item node and
  *ptrqi = item;                  // add it to the answers list
  answers.AddAtTail (ptrqi);
  item.ptrFromVertex = item.ptrToVertex; // reset from vertex
  minimumDistance = item.distance;       // store new min dist
  // now queue up all of its edges
  EdgeNode* ptre = item.ptrFromVertex->ptrEdgeHead;
  while (ptre) {
   queue.Enqueue (ptre->ptrToVertex);
   ptre = ptre->ptrFwd;
  }
  // now examine all edges
  while (!queue.IsEmpty ()) {
   VertexNode* ptrthis = queue.Dequeue ();
   if (ptrthis->visitedFlag == NotVisited) {
    item.ptrToVertex = ptrthis;
    EdgeNode* ptree = FindThisEdge (
                             item.ptrFromVertex->vertexData,
                             ptrthis->vertexData);
    if (!ptree) { // here we cannot find the requested edge
     failed = true;
     break;
    }
    // add in the distance to this edge
    item.distance = minimumDistance + ptree->edgeData;
    item.order = order++;
    pqueue.Enqueue (item); // add this one to the priority queue
   }
  }
 }
} while (!failed && !pqueue.IsArrayEmpty ());

// now examine the answer list and display just that part of the
// result the user requires
ItemNode* ptri;
if (!failed) {
```

```
  answers.ResetToHead ();
  ptri = answers.GetCurrentNode ();
  if (showOnlyShortest) { // if we want to show only the shortest
   Vertex findThisOne = to; // path, then begin by finding the to
   Stack<ItemNode> path;    // node and push it on the stack
   answers.ResetToHead ();  // then find how we got to it
   ptri = answers.GetCurrentNode (); // and so on til we get to
   while (ptri) {           // the from vertex
    if (ptri->ptrToVertex->vertexData == findThisOne) {
     path.Push (ptri);
     if (ptri->ptrFromVertex->vertexData == from)
      break;
     findThisOne = ptri->ptrFromVertex->vertexData;
     answers.ResetToHead ();
    }
    else
     answers.Next();
    ptri = answers.GetCurrentNode ();
   }
   // now poping off the vertices shows the path from-to
   ptri = path.Pop ();
   while (ptri) {
    DisplayShortestPath (ptri->ptrFromVertex->vertexData,
                  ptri->ptrToVertex->vertexData, ptri->distance);
    ptri = path.Pop ();
   }
  }
  // otherwise, user wants all shortest paths found
  else {
   while (ptri) {
    DisplayShortestPath (ptri->ptrFromVertex->vertexData,
                  ptri->ptrToVertex->vertexData, ptri->distance);
    answers.Next ();
    ptri = answers.GetCurrentNode ();
   }
  }
 }
 return true;
}

/***************************************************************/
/*                                                             */
/* FindThisEdge: given two vertices, find corresponding edge   */
/*                                                             */
/***************************************************************/

EdgeNode* Graph::FindThisEdge (const Vertex& from,
                               const Vertex& to) const {
 if (!ptrHead) return 0; // nothing to find

 // find the from vertex in the vertex list
 VertexNode* ptrfrom = FindThisVertex (from);
```

```
*      if (!ptrfrom) return 0; // from vertex not in the list         *
*                                                                     *
*      // find the to vertex in the from's edge list                  *
*      EdgeNode* ptre = ptrfrom->ptrEdgeHead;                         *
*      while (ptre) {                                                 *
*       if (ptre->ptrToVertex->vertexData == to)                      *
*        return ptre; // found it, so return this edge                *
*       ptre = ptre->ptrFwd;                                          *
*      }                                                              *
*                                                                     *
*      return 0; // return not found                                  *
*     }                                                               *
.))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))-
+)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Stack Class Template                                                *
/))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
*     #ifndef STACKH                                                  *
*     #define STACKH                                                  *
*     #include <iostream>                                             *
*     using namespace std;                                            *
*                                                                     *
*     /***************************************************************/*
*     /*                                                             */*
*     /* StackNode: contains the forward pointer and this item data  */*
*     /*                                                             */*
*     /***************************************************************/*
*     template<class UserData>                                        *
*     struct StackNode {                                              *
*      StackNode* fwdptr;                                             *
*      UserData*  dataptr;                                            *
*     };                                                              *
*                                                                     *
*     /***************************************************************/*
*     /*                                                             */*
*     /* Stack: a generic stack class                                */*
*     /*                                                             */*
*     /***************************************************************/*
*                                                                     *
*     template<class UserData>                                        *
*     class Stack {                                                   *
*     protected:                                                      *
*      StackNode<UserData>* headptr; // the top of the stack pointer  *
*      long                 count;   // number of items in the stack  *
*                                                                     *
*     public:                                                         *
*      Stack ();                     // construct an empty stack      *
*      Stack (const Stack<UserData>& s); // the copy constructor      *
*      Stack& operator= (const Stack<UserData>& s); // assignment op  *
*                                                                     *
*      ~Stack ();                    // delete the stack              *
*                                                                     *
*      void Push (UserData* ptrdata); // store new node on the stack  *
```

```
*  UserData* Pop ();           // removes top node from the stack  *
*  UserData* GetCurrentData () const; // returns user's data on top*
*  long GetCount () const; // returns the number of nodes in stack *
*                                                                   *
*  bool IsEmpty () const;  // returns true if stack is empty       *
*  void RemoveAll ();      // removes all nodes in the stack       *
*                                                                   *
* protected:                                                        *
*  // helper function to duplicate the stack                       *
*  void CopyStack (const Stack& s);                                *
* };                                                                *
*                                                                   *
* /****************************************************************/*
* /*                                                              */*
* /* Stack: create an empty stack                                 */*
* /*                                                              */*
* /****************************************************************/*
*                                                                   *
* template<class UserData>                                          *
* Stack<UserData>::Stack<UserData> () {                             *
*  headptr = 0;                                                     *
*  count = 0;                                                       *
* }                                                                 *
*                                                                   *
* /****************************************************************/*
* /*                                                              */*
* /* ~Stack: deletes the stack                                    */*
* /*                                                              */*
* /****************************************************************/*
*                                                                   *
* template<class UserData>                                          *
* Stack<UserData>::~Stack () {                                      *
*  RemoveAll ();                                                    *
* }                                                                 *
*                                                                   *
* /****************************************************************/*
* /*                                                              */*
* /* RemoveAll: deletes all nodes of the stack                    */*
* /*                                                              */*
* /****************************************************************/*
*                                                                   *
* template<class UserData>                                          *
* void Stack<UserData>::RemoveAll () {                              *
*  while (!IsEmpty()) Pop ();                                       *
* }                                                                 *
*                                                                   *
* /****************************************************************/*
* /*                                                              */*
* /* Push: store new node on the top of the stack                 */*
* /*                                                              */*
* /****************************************************************/*
*                                                                   *
```

```
template<class UserData>                                            *
void Stack<UserData>::Push (UserData* ptrdata) {                    *
 StackNode<UserData>* ptrnew = new StackNode<UserData>;             *
 ptrnew->dataptr = ptrdata;                                         *
 ptrnew->fwdptr = headptr;                                          *
 headptr = ptrnew;                                                  *
 count++;                                                           *
}                                                                   *
                                                                    *
/****************************************************************/*
/*                                                              */*
/* Pop: remove the top item from the stack                      */*
/*                                                              */*
/****************************************************************/*
                                                                    *
template<class UserData>                                            *
UserData* Stack<UserData>::Pop () {                                 *
 if (!headptr) return 0;                                            *
 StackNode<UserData>* ptrtodel = headptr;                           *
 headptr = headptr->fwdptr;                                         *
 UserData* ptrdata = ptrtodel->dataptr;                             *
 delete ptrtodel;                                                   *
 count--;                                                           *
 return ptrdata;                                                    *
}                                                                   *
                                                                    *
/****************************************************************/*
/*                                                              */*
/* GetCurrentData: returns a pointer to user data on the top    */*
/*                 of the stack or 0 if it is empty             */*
/*                                                              */*
/****************************************************************/*
                                                                    *
template<class UserData>                                            *
UserData* Stack<UserData>::GetCurrentData () const {                *
 return !IsEmpty () ? headptr->dataptr : 0;                         *
}                                                                   *
                                                                    *
/****************************************************************/*
/*                                                              */*
/* IsEmpty: returns true when there are no items on the stack   */*
/*                                                              */*
/****************************************************************/*
                                                                    *
template<class UserData>                                            *
bool Stack<UserData>::IsEmpty () const{                             *
 return headptr ? false : true;                                     *
}                                                                   *
                                                                    *
/****************************************************************/*
/*                                                              */*
/* GetCount: returns the number of nodes on the stack           */*
```

```
/*                                                              */*
/****************************************************************/*
                                                                  *
template<class UserData>                                          *
long Stack<UserData>::GetCount () const {                         *
 return count;                                                    *
}                                                                 *
                                                                  *
/****************************************************************/*
/*                                                              */*
/* Stack: copy constructor - make a duplicate copy of passed s  */*
/*                                                              */*
/****************************************************************/*
                                                                  *
template<class UserData>                                          *
Stack<UserData>::Stack<UserData> (const Stack<UserData>& s) {     *
 CopyStack (s);                                                   *
}                                                                 *
                                                                  *
/****************************************************************/*
/*                                                              */*
/* operator=: assignment op - makes a copy of passed stack      */*
/*                                                              */*
/****************************************************************/*
                                                                  *
template<class UserData>                                          *
Stack<UserData>& Stack<UserData>::operator= (                     *
                                      const Stack<UserData>& s) {*
 if (this == &s) return *this; // avoid a = a; situation          *
 RemoveAll ();  // remove all items in this stack                 *
 CopyStack (s);  // make a copy of stack s                        *
 return *this;                                                    *
}                                                                 *
                                                                  *
/****************************************************************/*
/*                                                              */*
/* CopyStack: helper that makes a duplicate copy                */*
/*                                                              */*
/****************************************************************/*
                                                                  *
template<class UserData>                                          *
void Stack<UserData>::CopyStack (const Stack<UserData>& s) {      *
 if (!s.headptr) { // handle stack s being empty                  *
  headptr = 0;                                                    *
  count = 0;                                                      *
  return;                                                         *
 }                                                                *
 count = s.count;                                                 *
 StackNode<UserData>* ptrScurrent = s.headptr;                    *
 // previousptr tracks our prior node so we can set its           *
 // forward pointer to the next new one                           *
 StackNode<UserData>* previousptr = 0;                            *
```

```
*  // prime the loop so headptr can be set one time                 *
*  StackNode<UserData>* currentptr = new StackNode<UserData>;       *
*  headptr = currentptr;  // assign this one to the headptr         *
*                                                                   *
* // traverse s stack's nodes                                       *
*  while (ptrScurrent) {                                            *
*   // copy node of s into our new node                             *
*   currentptr->dataptr = ptrScurrent->dataptr;                     *
*   currentptr->fwdptr = 0;  // set our forward ptr to 0            *
*   // if previous node exists, set its forward ptr to the new one *
*   if (previousptr)                                                *
*    previousptr->fwdptr = currentptr;                              *
*   // save this node as the prevous node                           *
*   previousptr = currentptr;                                       *
*   // and get a new node for the next iteration                    *
*   currentptr = new StackNode;                                     *
*   // move to s's next node                                        *
*   ptrScurrent = ptrScurrent->fwdptr;                              *
*  }                                                                *
*  delete currentptr; // delete the extra unneeded node             *
* }                                                                 *
*                                                                   *
*                                                                   *
* #endif                                                            *
.)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))-
+))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Queue Class Template                                              *
/)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
* #ifndef QUEUEH                                                    *
* #define QUEUEH                                                    *
* using namespace std;                                              *
*                                                                   *
* /***************************************************************/*
* /*                                                             */*
* /* QueueNode: stores the double linked list's fwd/back ptrs    */*
* /*              and the user's data ptr                        */*
* /*                                                             */*
* /***************************************************************/*
*                                                                   *
* template<class UserData>                                          *
* struct QueueNode {   // a double linked list                      *
*  QueueNode* fwdptr;                                               *
*  QueueNode* backptr;                                              *
*  UserData*  dataptr; // the user's object being stored            *
* };                                                                *
*                                                                   *
* /***************************************************************/*
* /*                                                             */*
* /* Queue Container Class                                       */*
* /*                                                             */*
* /* stores void pointers to user's objects                      */*
* /* before deleting an instance, the user MUST traverse and     */*
```

```
* /* delete any objects whose pointers are being stored          */*
* /*                                                             */*
* /**************************************************************/*
*                                                                  *
* template<class UserData>                                         *
* class Queue {                                                    *
*                                                                  *
* public:                                                          *
*  Queue ();                          // makes an empty queue      *
*  Queue (const Queue<UserData>& q);  // copy constructor          *
*  Queue<UserData>& operator= (const Queue<UserData>& q);          *
*  // VITAL NOTE: when a copy is made, the copy contains the SAME  *
*  // pointers as the original Queue - be careful not to delete    *
*  // them twice                                                   *
*                                                                  *
*  ~Queue ();          // the destructor removes the queue         *
*  void  RemoveAll (); // but not the user objects being stored!   *
*                                                                  *
*  void  Enqueue (UserData* ptrdata);// add an object to the queue *
*  UserData* Dequeue ();             // ret and remove current node*
*  long  GetSize () const;           // returns size of the queue  *
*  bool  IsEmpty () const;                                         *
*                                                                  *
*  void  ResetToHead (); // reset current to the start of the queue*
*  UserData* GetNext (); // returns next user object or 0 when at  *
*                        // the end of the queue                   *
*  // for cleanup operations, traverse the queue and delete the    *
*  // objects the queue is saving for you before you destroy or    *
*  // empty the queue                                              *
*                                                                  *
* /**************************************************************/*
* /*                                                             */*
* /* for Queue's internal use only                               */*
* /* Queue uses a double linked list                             */*
* /*                                                             */*
* /**************************************************************/*
*                                                                  *
* private:                                                         *
*  QueueNode<UserData>* headptr;    // pointer to first node       *
*  QueueNode<UserData>* tailptr;    // pointer to last node        *
*  QueueNode<UserData>* currentptr; // current node when traversing*
*  long       count;      // the number of nodes                   *
*                                                                  *
*  // helper function to copy a Queue                              *
*  void  CopyQueue (const Queue<UserData>& q);                     *
* };                                                               *
*                                                                  *
* /**************************************************************/*
* /*                                                             */*
* /* Queue: construct an empty queue                             */*
* /*                                                             */*
* /**************************************************************/*
```

```
template<class UserData>
Queue<UserData>::Queue () {
 headptr = currentptr = tailptr = 0;
 count = 0;
}

/***************************************************************/
/*                                                             */
/* ~Queue: remove all QueueNode objects - but does not delete  */
/*         any user objects                                    */
/*                                                             */
/***************************************************************/

template<class UserData>
Queue<UserData>::~Queue () {
 RemoveAll ();
}

/***************************************************************/
/*                                                             */
/* RemoveAll: removes all QueueNode Objects, leaving the queue */
/*            in an empty but valid state                      */
/*                                                             */
/***************************************************************/

template<class UserData>
void Queue<UserData>::RemoveAll () {
 if (!headptr) return;          // nothing to do case
 QueueNode<UserData>* ptrnext = headptr; // start at the front
 QueueNode<UserData>* ptrdel;
 while (ptrnext) {          // for all QueueNodes,
  ptrdel = ptrnext;         // save its pointer for later deletion
  ptrnext = ptrnext->fwdptr;// set for next node in the queue
  delete ptrdel;            // remove this node
 }
 // leave queue in a default, valid , empty state
 currentptr = tailptr = headptr = 0;
 count = 0;
}

/***************************************************************/
/*                                                             */
/* Queue Copy Constructor: duplicate the passed Queue object   */
/*                                                             */
/* VITAL: we will not duplicate the user's actual data         */
/*                                                             */
/***************************************************************/

template<class UserData>
Queue<UserData>::Queue (const Queue<UserData>& q) {
 CopyQueue (q);             // call helper function to do the work
```

```
}                                                                 *
                                                                  *
/***************************************************************/*
/*                                                             */*
/* Operator= - Assignment operator: duplicate this Queue object*/*
/*                                                             */*
/* VITAL: we will not duplicate the user's actual data         */*
/*                                                             */*
/***************************************************************/*
                                                                  *
template<class UserData>                                          *
Queue<UserData>& Queue<UserData>::operator= (                     *
                                      const Queue<UserData>& q) {*
 if (&q == this) return *this; // avoid silly case of x = x;     *
 if (count != 0) RemoveAll (); // if we are not empty, empty us  *
 CopyQueue (q);                // call helper function to do it  *
 return *this;                 // return us so user can chain    *
}                                                                 *
                                                                  *
/***************************************************************/*
/*                                                             */*
/* CopyQueue: make a shallow copy of the passed queue          */*
/*                                                             */*
/* VITAL: we will not duplicate the user's actual data         */*
/*                                                             */*
/***************************************************************/*
                                                                  *
template<class UserData>                                          *
void  Queue<UserData>::CopyQueue (const Queue<UserData>& q) {    *
 // initialize queue so that we can use Enqueue to add the nodes *
 currentptr = tailptr = headptr = 0;                              *
 count = 0;                                                       *
 if (!q.count) // if there are none, queue is now initialized    *
  return;                                                         *
 // point to their head                                           *
 QueueNode<UserData>* ptrqcurrent = q.headptr;                    *
 while (ptrqcurrent) {              // while there's another node*
  Enqueue (ptrqcurrent->dataptr);   // add it to our queue       *
  ptrqcurrent = ptrqcurrent->fwdptr;// point to next one to copy *
 }                                                                *
}                                                                 *
                                                                  *
/***************************************************************/*
/*                                                             */*
/* IsEmpty: returns true if queue is empty                     */*
/*                                                             */*
/***************************************************************/*
                                                                  *
template<class UserData>                                          *
bool Queue<UserData>::IsEmpty () const {                          *
 return count == 0 ? true : false;                                *
}                                                                 *
```

```
                                                                     *
/*******************************************************************/*
/*                                                                 */*
/* Enqueue: Add a new node to the tail of the queue                */*
/*                                                                 */*
/*******************************************************************/*
                                                                     *
template<class UserData>                                             *
void  Queue<UserData>::Enqueue (UserData* ptrdata) {                 *
 QueueNode<UserData>* ptrnew = new QueueNode<UserData>;              *
 ptrnew->dataptr = ptrdata;     // insert user's object              *
 count++;                       // increment number of nodes         *
 if (tailptr) {                 // if there are other nodes,         *
  tailptr->fwdptr = ptrnew;     // last one now points to us         *
  ptrnew->backptr = tailptr;    // us points to previous last one*
  ptrnew->fwdptr = 0;           // us points to none                 *
  tailptr = currentptr = ptrnew;// reset tail to us                  *
 }                                                                   *
 else { // queue is currently empty, so just add us                  *
  headptr = tailptr = currentptr = ptrnew;                           *
  ptrnew->fwdptr = ptrnew->backptr = 0;                              *
 }                                                                   *
}                                                                    *
                                                                     *
/*******************************************************************/*
/*                                                                 */*
/* Dequeue: return object at the head and delete that node         */*
/*                                                                 */*
/*******************************************************************/*
                                                                     *
template<class UserData>                                             *
UserData* Queue<UserData>::Dequeue () { // remove at head            *
 if (!headptr) return 0;       // we are empty, so do nothing        *
 currentptr = headptr;         // reset to the head object           *
 if (headptr->fwdptr)          // is there more than one node?       *
  headptr->fwdptr->backptr = 0;// yes,set next one's back to none*
 headptr = headptr->fwdptr;    // reset head ptr to the next one *
 count--;                      // decrement count of nodes           *
 if (count == 0) tailptr = 0;  // reset tailptr if queue is empty*
 UserData* retval = currentptr->dataptr;// save object to be ret *
 delete currentptr;            // remove previous head object        *
 currentptr = headptr;         // reset the current node ptr         *
 return retval;                // give the user their object         *
}                                                                    *
                                                                     *
/*******************************************************************/*
/*                                                                 */*
/* GetSize: returns the number of items in the queue               */*
/*                                                                 */*
/*******************************************************************/*
                                                                     *
template<class UserData>                                             *
```

```
* long  Queue<UserData>::GetSize () const {                          *
*  return count;                                                     *
* }                                                                  *
*                                                                    *
* /***************************************************************/*
* /*                                                             */*
* /* ResetToHead: reset currentptr to head pointer for queue     */*
* /*              traversal operations                           */*
* /*                                                             */*
* /***************************************************************/*
*                                                                    *
* template<class UserData>                                           *
* void  Queue<UserData>::ResetToHead () {                            *
*  currentptr = headptr;                                             *
* }                                                                  *
*                                                                    *
* /***************************************************************/*
* /*                                                             */*
* /* GetNext: returns next user object & sets currentptr for next*/*
* /*                                                             */*
* /***************************************************************/*
*                                                                    *
* template<class UserData>                                           *
* UserData* Queue<UserData>::GetNext () {                            *
*  if (!currentptr) return 0;      // queue is empty, so do nothing*
*  UserData* retval = currentptr->dataptr;// save object to be ret *
*  currentptr = currentptr->fwdptr;// set currentptr to next in one*
*  return retval;                  // give the user the current obj*
* }                                                                  *
*                                                                    *
* #endif                                                             *
.)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))-
```

Pgm13b — A Sample Application — Airline Routes

I often spend my Christmas holidays with my two young nephews who live near Burbank, California. I am in Peoria, Illinois. Thus, I fly. But which airline and which route should I take? From Peoria, one can fly there several ways, including via Chicago, St. Louis, and Denver, for example. **Pgm13b** builds a graph of a number of airline flights around the country. The **Vertex** contains the city name. The **Edge** contains the air distance between the two cities. See Figure 13.18 once more. And then reexamine the listing for **VertexNode.h** and **VertexNode.cpp** just below that figure.

I created a **Routes.txt** file that contains a number of direct flights. Here is a sample line from the file.

```
From "Peoria, IL" To "Chicago, IL" is 130
```

Pgm13b inputs this routes file and builds a graph from the data. For each line, after inputting the “from” vertex and “to” vertex, **FindThisVertex** is called to verify each vertex is already in the graph. If a vertex is not in the graph, **AddVertex** is called. Once both vertices are present, the **AddEdge** is called to store the distance between the cities. In this manner, a graph can easily be loaded with user information.

However, in order to write a reasonable client program that utilizes these vertices, they need to be stored in an array of **Vertex** structures. I chose to make that an **Array** template class. Then, when the user needs to pick a “from” or “to” city, I can retrieve the vertices from the array and display the city names as well as pass the requested **Vertex** to the various **Graph** functions.

First, let’s see what the output of this simple program looks like. After loading the file of vertices, the basic form of the graph is displayed on lines 1 through 40. Then, the two graph traversal functions are called whose output is shown on lines 42 through 59. A minimum spanning tree is built next and shown on lines 62 through 68. Finally, the remainder illustrates a simple user application of determining whether or not a flight exists between two cities and/or the shortest path.

```
+))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Output from Pgm13b Airline Flight Picker Program                    *
/)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
*  1 A Display of the Graph Tree - Vertex with its Edges              *
*  2                                                                  *
*  3 From: Burbank, CA                                                *
*  4   To: Chicago, IL                                                *
*  5   To: Denver, CO                                                 *
*  6   To: St. Louis, MO                                              *
*  7                                                                  *
*  8 From: Chicago, IL                                                *
*  9   To: Burbank, CA                                                *
* 10   To: Denver, CO                                                 *
* 11   To: Los Angeles, CA                                            *
* 12   To: New York, NY                                               *
* 13   To: Peoria, IL                                                 *
* 14                                                                  *
* 15 From: Denver, CO                                                 *
* 16   To: Burbank, CA                                                *
* 17   To: Chicago, IL                                                *
* 18   To: Los Angeles, CA                                            *
* 19   To: Peoria, IL                                                 *
* 20   To: St. Louis, MO                                              *
* 21                                                                  *
* 22 From: Los Angeles, CA                                            *
* 23   To: Chicago, IL                                                *
* 24   To: Denver, CO                                                 *
* 25   To: St. Louis, MO                                              *
* 26                                                                  *
* 27 From: New York, NY                                               *
* 28   To: Chicago, IL                                                *
```

```
*                                                                    *
* From: Peoria, IL                                                   *
*   To: Chicago, IL                                                  *
*   To: Denver, CO                                                   *
*   To: St. Louis, MO                                                *
*                                                                    *
* From: St. Louis, MO                                                *
*   To: Burbank, CA                                                  *
*   To: Denver, CO                                                   *
*   To: Los Angeles, CA                                              *
*   To: Peoria, IL                                                   *
*                                                                    *
*                                                                    *
* Depth First Traversal:                                             *
* Burbank, CA                                                        *
* St. Louis, MO                                                      *
* Peoria, IL                                                         *
* Los Angeles, CA                                                    *
* Denver, CO                                                         *
* Chicago, IL                                                        *
* New York, NY                                                       *
*                                                                    *
*                                                                    *
* Breadth First Traversal:                                           *
* Burbank, CA                                                        *
* Chicago, IL                                                        *
* Denver, CO                                                         *
* St. Louis, MO                                                      *
* Los Angeles, CA                                                    *
* New York, NY                                                       *
* Peoria, IL                                                         *
*                                                                    *
*                                                                    *
* The Minimum Spanning Tree                                          *
* From Vertex: Burbank, CA     To Vertex: Denver, CO         849   *
* From Vertex: Chicago, IL     To Vertex: New York, NY       732   *
* From Vertex: Denver, CO      To Vertex: Los Angeles, CA    861   *
* From Vertex: Denver, CO      To Vertex: Peoria, IL         791   *
* From Vertex: Peoria, IL      To Vertex: Chicago, IL        130   *
* From Vertex: Peoria, IL      To Vertex: St. Louis, MO      128   *
*                                                                    *
*                                                                    *
*                                                                    *
*         Vic's Airplane Flight Checker                              *
*                                                                    *
*         1. Does a flight exist (depth first)?                      *
*         2. Does a flight exist (breadth first)?                    *
*         3. What is the shortest route? (Show only shortest)        *
*         4. What is the shortest route? (Show all)                  *
*         5. Quit                                                    *
*                                                                    *
* Enter the number of your choice: 3                                 *
```

```
*                                                                   *
*                                                                   *
*                                                                   *
*            Pick the "From" city                                   *
*            1. Peoria, IL                                          *
*            2. Chicago, IL                                         *
*            3. Denver, CO                                          *
*            4. St. Louis, MO                                       *
*            5. Los Angeles, CA                                     *
*            6. Burbank, CA                                         *
*            7. New York, NY                                        *
*            8. Abort this action                                   *
*                                                                   *
*                                                                   *
*    Enter the number of your choice: 1                             *
*                                                                   *
*                                                                   *
*                                                                   *
*            Pick the "To" city                                     *
*            1. Peoria, IL                                          *
*            2. Chicago, IL                                         *
*            3. Denver, CO                                          *
*            4. St. Louis, MO                                       *
*            5. Los Angeles, CA                                     *
*            6. Burbank, CA                                         *
*            7. New York, NY                                        *
*            8. Abort this action                                   *
*                                                                   *
*                                                                   *
*    Enter the number of your choice: 6                             *
*                                                                   *
*                                                                   *
*    Shortest path from Peoria, IL to Burbank, CA                   *
*       From City           To City                 Total Miles     *
*       Peoria, IL          Denver, CO                  791         *
*       Denver, CO          Burbank, CA                1640         *
*    Enter C to continue c                                          *
*                                                                   *
*                                                                   *
*                                                                   *
*            Vic's Airplane Flight Checker                          *
*                                                                   *
*            1. Does a flight exist (depth first)?                  *
*            2. Does a flight exist (breadth first)?                *
*            3. What is the shortest route? (Show only shortest)    *
*            4. What is the shortest route? (Show all)              *
*            5. Quit                                                *
*                                                                   *
*    Enter the number of your choice: 1                             *
*                                                                   *
*                                                                   *
*                                                                   *
```

```
*133         Pick the "From" city                                          *
*134         1. Peoria, IL                                                 *
*135         2. Chicago, IL                                                *
*136         3. Denver, CO                                                 *
*137         4. St. Louis, MO                                              *
*138         5. Los Angeles, CA                                            *
*139         6. Burbank, CA                                                *
*140         7. New York, NY                                               *
*141         8. Abort this action                                          *
*142                                                                       *
*143                                                                       *
*144 Enter the number of your choice: 1                                    *
*145                                                                       *
*146                                                                       *
*147                                                                       *
*148         Pick the "To" city                                            *
*149         1. Peoria, IL                                                 *
*150         2. Chicago, IL                                                *
*151         3. Denver, CO                                                 *
*152         4. St. Louis, MO                                              *
*153         5. Los Angeles, CA                                            *
*154         6. Burbank, CA                                                *
*155         7. New York, NY                                               *
*156         8. Abort this action                                          *
*157                                                                       *
*158                                                                       *
*159 Enter the number of your choice: 6                                    *
*160 A path exists between Peoria, IL and Burbank, CA                      *
*161 Enter C to continue                                                   *
*162 c                                                                     *
*163                                                                       *
*164                                                                       *
*165                                                                       *
*166         Vic's Airplane Flight Checker                                 *
*167                                                                       *
*168         1. Does a flight exist (depth first)?                         *
*169         2. Does a flight exist (breadth first)?                       *
*170         3. What is the shortest route? (Show only shortest)           *
*171         4. What is the shortest route? (Show all)                     *
*172         5. Quit                                                       *
*173                                                                       *
*174 Enter the number of your choice: 2                                    *
*175                                                                       *
*176                                                                       *
*177                                                                       *
*178         Pick the "From" city                                          *
*179         1. Peoria, IL                                                 *
*180         2. Chicago, IL                                                *
*181         3. Denver, CO                                                 *
*182         4. St. Louis, MO                                              *
*183         5. Los Angeles, CA                                            *
*184         6. Burbank, CA                                                *
```

```
        7. New York, NY
        8. Abort this action
Enter the number of your choice: 1
        Pick the "To" city
        1. Peoria, IL
        2. Chicago, IL
        3. Denver, CO
        4. St. Louis, MO
        5. Los Angeles, CA
        6. Burbank, CA
        7. New York, NY
        8. Abort this action
Enter the number of your choice: 6
A path exists between Peoria, IL and Burbank, CA
Enter C to continue c
        Vic's Airplane Flight Checker
        1. Does a flight exist (depth first)?
        2. Does a flight exist (breadth first)?
        3. What is the shortest route? (Show only shortest)
        4. What is the shortest route? (Show all)
        5. Quit
Enter the number of your choice: 4
        Pick the "From" city
        1. Peoria, IL
        2. Chicago, IL
        3. Denver, CO
        4. St. Louis, MO
        5. Los Angeles, CA
        6. Burbank, CA
        7. New York, NY
        8. Abort this action
Enter the number of your choice: 1
```

```
*         Pick the "To" city                                          *
*         1. Peoria, IL                                               *
*         2. Chicago, IL                                              *
*         3. Denver, CO                                               *
*         4. St. Louis, MO                                            *
*         5. Los Angeles, CA                                          *
*         6. Burbank, CA                                              *
*         7. New York, NY                                             *
*         8. Abort this action                                        *
*                                                                     *
*                                                                     *
* Enter the number of your choice: 6                                  *
*                                                                     *
*                                                                     *
* Shortest path from Peoria, IL to Burbank, CA                        *
*    From City           To City                  Total Miles         *
*    Peoria, IL          Peoria, IL                     0             *
*    Peoria, IL          St. Louis, MO                128             *
*    Peoria, IL          Chicago, IL                  130             *
*    Peoria, IL          Denver, CO                   791             *
*    Chicago, IL         New York, NY                 862             *
*    Denver, CO          Burbank, CA                 1640             *
*    Denver, CO          Los Angeles, CA             1652             *
* Enter C to continue c                                               *
*                                                                     *
*                                                                     *
*                                                                     *
*         Vic's Airplane Flight Checker                               *
*                                                                     *
*         1. Does a flight exist (depth first)?                       *
*         2. Does a flight exist (breadth first)?                     *
*         3. What is the shortest route? (Show only shortest)         *
*         4. What is the shortest route? (Show all)                   *
*         5. Quit                                                     *
*                                                                     *
* Enter the number of your choice: 5                                  *
* No memory leaks                                                     *
.))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))-
```

The most important aspect of **Pgm13b** is how the graph is actually built from the user's data file. Notice I separated the lower level action of inputting the line of data into a separate function, **InputLine**, which leaves **LoadGraph** to concentrate only on building the graph. The basic idea is as follows. If a vertex does not exist, then add it. Once both vertices have been added or exist, then add in the edges. In this case, I assume that one can fly both ways — a digraph. You could easily modify this procedure to implement direction as well by changing how the edges are added.

```
 Vertex from;
 Vertex to;
 Edge edge;
 while (InputLine (infile, from, to, edge)) {
```

```
  if (!g.FindThisVertex (from)) {
   g.AddVertex (from);
   array.Add (from);
  }
  if (!g.FindThisVertex (to)) {
   g.AddVertex (to);
   array.Add (to);
  }
  g.AddEdge (from, to, edge);
  g.AddEdge (to, from, edge);
 }
```

Notice that if a **Vertex** is not found in the graph, it is added to the graph and to my array of vertices.

Here is the complete **Pgm13b** coding.

```
+))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Pgm13b Airline Flight Picker Program                                      *
/)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
* #include <iostream>                                                      *
* #include <iomanip>                                                       *
* #include <cstring>                                                       *
* #include <crtdbg.h>                                                      *
* #include <fstream>                                                       *
*                                                                          *
* /***************************************************************/       *
* /*                                                             */       *
* /* Pgm13b: a Simple Graph Class Tester Program                 */       *
* /*                                                             */       *
* /***************************************************************/       *
*                                                                          *
* using namespace std;                                                     *
*                                                                          *
* #include "Graph.h"                                                       *
* #include "Array.h"                                                       *
* #include "Heap.h"                                                        *
* #include "PriorityQueue.h"                                               *
* #include "VertexEdge.h"                                                  *
*                                                                          *
* /***************************************************************/       *
* /*                                                             */       *
* /* Needed Graph Callback Functions                             */       *
* /*                                                             */       *
* /***************************************************************/       *
*                                                                          *
* void Display (Vertex& v);                                                *
* void DisplayTree (Vertex& v, bool isConnectedVertex);                    *
* void DisplayShortestPath (const Vertex& from, const Vertex& to,          *
*                           const Edge& distance);                         *
* void DisplaySpanningTree (const Vertex& from, const Vertex& to,          *
*                           const Edge& edge);                             *
```

```
*                                                                  *
* /******************************************************/        *
* /*                                                    */        *
* /* Pgm13b: functions                                  */        *
* /*                                                    */        *
* /******************************************************/        *
*                                                                  *
* void     LoadGraph (Graph& g, Array<Vertex>& list);              *
* istream& InputLine (istream& is, Vertex& from, Vertex& to,       *
*                     Edge& e);                                    *
*                                                                  *
* enum MainMenuChoice {ExistDepth = 1, ExistBreadth, Shortest,     *
*                      ShortestAll, Quit};                         *
* MainMenuChoice GetValidMainMenuChoice ();                        *
* void DisplayMainMenu ();                                         *
*                                                                  *
* int GetValidCityChoice (Array<Vertex>& array, const char* title);*
* void DisplayCityPicker (Array<Vertex>& array, const char* title);*
*                                                                  *
* int main () {                                                    *
* {                                                                *
*  cin.sync_with_stdio ();                                         *
*  cout.setf (ios::fixed, ios::floatfield);                        *
*  Graph g;                                                        *
*                                                                  *
*  // illustrate how a graph can be loaded from a file             *
*  Array<Vertex> array;                                            *
*  LoadGraph (g, array);                                           *
*                                                                  *
*  // a simple display to verify graph appears correctly loaded    *
*  cout << "A Display of the Graph Tree - Vertex with its Edges\n";*
*  g.DisplayTree (DisplayTree);                                    *
*  cout << endl << endl;                                           *
*                                                                  *
*  // sample traversals                                            *
*  cout << "Depth First Traversal:\n";                             *
*  g.DepthFirstTraversal (Display);                                *
*                                                                  *
*  cout << "\n\nBreadth First Traversal:\n";                       *
*  g.BreadthFirstTraversal (Display);                              *
*                                                                  *
*  // find the minimum spanning tree                               *
*  Edge max;                                                       *
*  max.distance = 1e10;                                            *
*  g.BuildMinimumSpanningTree (max);                               *
*  cout << "\n\nThe Minimum Spanning Tree\n";                      *
*  g.ShowMinimumSpanningTree (DisplaySpanningTree);                *
*                                                                  *
*  // illustrate using the graph to find the shortest paths        *
*  MainMenuChoice choice = GetValidMainMenuChoice ();              *
*  while (choice != Quit) {                                        *
*   // next pick the from and to cities                            *
```

```
  int from = GetValidCityChoice(array, "Pick the \"From\" city");
  if (from == array.GetSize() || from == -1) break;
  int to = GetValidCityChoice (array, "Pick the \"To\" city");
  if (to == array.GetSize() || from == -1) break;

  Vertex fromV = *(array.GetAt (from));
  Vertex toV = *(array.GetAt (to));
  Edge es;
  es.distance = 0;

  switch (choice) {
   case ExistDepth:
    if (g.DoesPathExistBetween_DepthFirst (fromV, toV))
     cout << "A path exists between " << array.GetAt(from)->city
          << " and " << array.GetAt(to)->city << endl;
    else
     cout << "A path does not exist between "
          << array.GetAt(from)->city
          << " and " << array.GetAt(to)->city << endl;
    break;
   case ExistBreadth:
    if (g.DoesPathExistBetween_BreadthFirst (fromV, toV))
     cout << "A path exists between " << array.GetAt(from)->city
          << " and " << array.GetAt(to)->city << endl;
    else
     cout << "A path does not exist between "
          << array.GetAt(from)->city
          << " and " << array.GetAt(to)->city << endl;
    break;
   case Shortest:
     cout << "\n\nShortest path from " << array.GetAt(from)->city
          << " to " << array.GetAt(to)->city << endl
          <<
 "   From City           To City                  Total Miles\n";
     g.FindShortestPath (fromV, toV, es, true, true,
                         DisplayShortestPath);
    break;
   case ShortestAll:
    cout << "\n\nShortest path from " << array.GetAt(from)->city
         << " to " << array.GetAt(to)->city << endl
         <<
 "   From City           To City                  Total Miles\n";
    g.FindShortestPath (fromV, toV, es, false, true,
                        DisplayShortestPath);
    break;
  }
  char c;
  cout << "Enter C to continue ";
  cin >> c;
  choice = GetValidMainMenuChoice ();
 }
}
```

```
 // check for memory leaks
 if (_CrtDumpMemoryLeaks())
  cerr << "Memory leaks occurred!\n";
 else
  cerr << "No memory leaks.\n";
 return 0;
}

/***********************************************************/
/*                                                         */
/* LoadGraph: illustrates how to load a graph from a file */
/*                                                         */
/***********************************************************/

void LoadGraph (Graph& g, Array<Vertex>& array) {
 ifstream infile ("Routes.txt");
 if (!infile) {
  cerr << "Error: cannot open Routes.txt\n";
  exit (2);
 }
 Vertex from;
 Vertex to;
 Edge edge;
 while (InputLine (infile, from, to, edge)) {
  if (!g.FindThisVertex (from)) {
   g.AddVertex (from);
   array.Add (from);
  }
  if (!g.FindThisVertex (to)) {
   g.AddVertex (to);
   array.Add (to);
  }
  g.AddEdge (from, to, edge);
  g.AddEdge (to, from, edge);
 }
 if (!infile.eof()) {
  infile.close ();
  exit (1);
 }
 infile.close ();
}

/*********************************************************/
/*                                                       */
/* InputLine: inputs a single data line                  */
/*                                                       */
/*********************************************************/

istream& InputLine (istream& is, Vertex& from, Vertex& to,
                    Edge& e) {
 char str[80];
 is >> str;
```

```
 if (!is) return is;
 if (stricmp (str, "From") != 0) {
  cerr << "Error: bad data - expected From but found " << str
       << endl;
  is.clear (ios::failbit);
  return is;
 }
 char c;
 is >> c;
 if (c != '\"') {
  cerr << "Error: expected a \" before From city\n";
  is.clear (ios::failbit);
  return is;
 }
 is.getline (from.city, sizeof (from.city), '\"');
 is >> str;
 if (!is || stricmp (str, "To") != 0) {
  cerr << "Error: expected To but found " << str << endl;
  is.clear (ios::failbit);
  return is;
 }
 is >> c;
 if (c != '\"') {
  cerr << "Error: expected a \" before To city\n";
  is.clear (ios::failbit);
  return is;
 }
 is.getline (to.city, sizeof (to.city), '\"');
 is >> str;
 if (!is || stricmp (str, "is") != 0) {
  cerr << "Error: expected is but found " << str << endl;
  is.clear (ios::failbit);
  return is;
 }
 is >> e.distance;
 return is;
}

/********************************************************/
/*                                                      */
/* Display: a callback function to display a single vert*/
/*                                                      */
/********************************************************/

void Display (Vertex& v) {
 cout << v.city << endl;
}

/********************************************************/
/*                                                      */
/* DisplayTree: a callback function to show a vertex    */
/*                                                      */
```

```
/*******************************************************/

void DisplayTree (Vertex& v, bool isConnectedVertex) {
 if (!isConnectedVertex)
  cout << "\nFrom: " << v.city << endl;
 else
  cout << "  To: " << v.city << endl;
}

/*******************************************************/
/*                                                     */
/* DisplayShortestPath: a callback function to show path*/
/*                                                     */
/*******************************************************/

void DisplayShortestPath (const Vertex& from, const Vertex& to,
                          const Edge& edge) {
 cout.setf (ios::left, ios::adjustfield);
 cout << "   " << setw (20) << from.city << setw(25)
      << to.city;
 cout.setf (ios::right, ios::adjustfield);
 cout << setw(8) << setprecision (0) << edge.distance << endl;
}

/*******************************************************/
/*                                                     */
/* DisplaySpanningTree: callback to display span tree   */
/*                                                     */
/*******************************************************/

void DisplaySpanningTree (const Vertex& from, const Vertex& to,
                          const Edge& edge) {
 cout.setf (ios::left, ios::adjustfield);
 cout << "From Vertex: " << setw (20) << from.city << "  "
      << "To Vertex: " << setw (20) << to.city << "  ";
 cout.setf (ios::right, ios::adjustfield);
 cout << setprecision (0) << setw (5) << edge.distance << endl;
}

/*******************************************************/
/*                                                     */
/* GetValidMainMenuChoice and DisplayMainMenu:          */
/*                                                     */
/*******************************************************/

MainMenuChoice GetValidMainMenuChoice () {
 int choice = 6;
 while (choice < 1 || choice > 5) {
  DisplayMainMenu ();
  cin >> choice;
  if (!cin) return Quit;
 }
```

```
 return (MainMenuChoice) choice;
}

void DisplayMainMenu () {
 cout << "\n\n\n\tVic's Airplane Flight Checker\n\n"
      << "\t1. Does a flight exist (depth first)?\n"
      << "\t2. Does a flight exist (breadth first)?\n"
      << "\t3. What is the shortest route? (Show only shortest)\n"
      << "\t4. What is the shortest route? (Show all)\n"
      << "\t5. Quit\n\n"
      << "Enter the number of your choice: ";
}

/********************************************************/
/*                                                      */
/* DisplayCityPicker and GetValidCityChoice:            */
/*                                                      */
/********************************************************/

void DisplayCityPicker (Array<Vertex>& array, const char* title){
 cout << "\n\n\n\t" << title << endl;
 for (int i=0; i<array.GetSize(); i++) {
  cout << "\t" << i+1 << ". " << array.GetAt(i)->city << endl;
 }
 cout << "\t" << array.GetSize()+1 << ". Abort this action\n\n";
 cout << "\nEnter the number of your choice: ";
}

int GetValidCityChoice (Array<Vertex>& array, const char* title){
 int choice = array.GetSize()+2;
 while (choice < 1 || choice > array.GetSize()+1) {
  DisplayCityPicker (array, title);
  cin >> choice;
  if (!cin) return array.GetSize();
 }
 return choice -1;
}
```

Finally, here is the coding for the **Array** class.

```
The Array Template Class

#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>
using namespace std;

/********************************************************/
/*                                                      */
```

```
*  /* Array: container for growable array of user items     */      *
*  /*        it can store a variable number of elements     */      *
*  /*                                                        */      *
*  /*        elements stored are copies of the original     */      *
*  /*                                                        */      *
*  /*  errors are logged to cerr device                     */      *
*  /*                                                        */      *
*  /*********************************************************/      *
*                                                                    *
*  template<class UserData>                                          *
*  class Array {                                                     *
*                                                                    *
*   /********************************************************/      *
*   /*                                                       */      *
*   /* class data                                            */      *
*   /*                                                       */      *
*   /********************************************************/      *
*                                                                    *
*  protected:                                                        *
*   UserData* array;                                                 *
*   long      numElements;                                           *
*                                                                    *
*   /********************************************************/      *
*   /*                                                       */      *
*   /* class functions                                       */      *
*   /*                                                       */      *
*   /********************************************************/      *
*                                                                    *
*  public:                                                           *
*   Array ();    // default constructor - makes an empty array      *
*   ~Array ();   // deletes the array                               *
*                                                                    *
*   bool Add (const UserData& newElement); // add an element        *
*   bool InsertAt (long i, const UserData& newElement);             *
*        // adds this element at subscript i                        *
*        // if i < 0, it is added at the front                      *
*        // if i >= numElements, it is added at the end             *
*        // otherwise, it is added at the ith position              *
*        // returns true if successful                              *
*                                                                    *
*   UserData* GetAt (long i) const; // rets element at the ith pos  *
*                    // If i is out of range, it returns 0          *
*                                                                    *
*   bool RemoveAt (long i); // removes the element at subscript i   *
*             // if i is out of range, an error is displayed on cerr*
*             // returns true if successful                         *
*                                                                    *
*   void RemoveAll (); // removes all elements                      *
*                                                                    *
*   long GetSize () const; // rets num elements in array            *
*                                                                    *
*   // copy ctor and assignment operator                            *
```

```
 Array (const Array<UserData>& a);
 Array<UserData>& operator= (const Array<UserData>& a);

protected:
 void Copy (const Array<UserData>& a); // performs the copy
};

/***************************************************************/
/*                                                             */
/* Array: constructs an empty array                            */
/*                                                             */
/***************************************************************/

template<class UserData>
Array<UserData>::Array () {
 numElements = 0;
 array = 0;
}

/***************************************************************/
/*                                                             */
/* ~Array: deletes dynamically allocated memory                */
/*                                                             */
/***************************************************************/

template<class UserData>
Array<UserData>::~Array () {
 RemoveAll ();
}

/***************************************************************/
/*                                                             */
/* Add: Adds this new element to the end of the array          */
/*      if out of memory, displays error message to cerr       */
/*                                                             */
/***************************************************************/

template<class UserData>
bool Array<UserData>::Add (const UserData& newElement) {
 // allocate new temporary array one element larger
 UserData* temp = new UserData [numElements + 1];

 // check for out of memory
 if (!temp) {
  cerr << "Array: Add Error - out of memory\n";
  return false;
 }

 // copy all existing elements into the new temp array
 for (long i=0; i<numElements; i++) {
  temp[i] = array[i];
 }
```

```

 // copy in the new element to be added
 temp[numElements] = newElement;

 numElements++;  // increment the number of elements in the array
 if (array) delete [] array; // delete the old array
 array = temp;   // point out array to the new array
 return true;
}

/*******************************************************/
/*                                                     */
/* InsertAt: adds the new element to the array at index i*/
/*  if i is in range, it is inserted at subscript i     */
/*  if i is negative, it is inserted at the front       */
/*  if i is greater than or equal to the number of      */
/*     elements, then it is added at the end of the array*/
/*                                                     */
/*  if there is insufficient memory, an error message   */
/*     is displayed to cerr                            */
/*                                                     */
/*******************************************************/

template<class UserData>
bool Array<UserData>::InsertAt (long i,
                                const UserData& newElement) {
 UserData* temp;
 long j;
 // allocate a new array one element larger
 temp = new UserData [numElements + 1];

 // check if out of memory
 if (!temp) {
  cerr << "Array: InsertAt - Error out of memory\n";
  return false;
 }

 // this case handles an insertion that is within range
 if (i < numElements && i >= 0) {
  for (j=0; j<i; j++) { // copy all elements below insertion
   temp[j] = array[j];  // point
  }
  temp[i] = newElement; // insert new element
  for (j=i; j<numElements; j++) { // copy remaining elements
   temp[j+1] = array[j];
  }
 }

 // this case handles an insertion when the index is too large
 else if (i >= numElements) {
  for (j=0; j<numElements; j++) { // copy all existing elements
   temp[j] = array[j];
```

```
  }
  temp[numElements] = newElement; // add new one at end
 }

 // this case handles an insertion when the index is too small
 else {
  temp[0] = newElement;        // insert new on at front
  for (j=0; j<numElements; j++) { // copy all others after it
   temp[j+1] = array[j];
  }
 }

 // for all cases, delete current array, assign new one and
 // increment the number of elements in the array
 if (array) delete [] array;
 array = temp;
 numElements++;
 return true;
}

/*******************************************************/
/*                                                     */
/* GetAt: returns the element at index i               */
/*        if i is out of range, returns 0              */
/*                                                     */
/*******************************************************/

template<class UserData>
UserData* Array<UserData>::GetAt (long i) const {
 if (i < numElements && i >=0)
  return &array[i];
 else
  return 0;
}

/*******************************************************/
/*                                                     */
/* RemoveAt: removes the element at subscript i        */
/*                                                     */
/* If i is out of range, an error is displayed on cerr */
/*                                                     */
/* Note that what the element actually points to is not */
/*      deleted                                        */
/*                                                     */
/*******************************************************/

template<class UserData>
bool Array<UserData>::RemoveAt (long i) {
 UserData* temp;
 if (numElements > 1) {
  if (i >= 0 && i < numElements) {  // if the index is in range,
   temp = new UserData [numElements - 1]; // alloc smaller array
```

```
   long j;
   for (j=0; j<i; j++) {           // copy all elements up to
    temp[j] = array[j];            // the desired one to be
   }                               // removed
   for (j=i+1; j<numElements; j++) {// then copy all the elements
    temp[j-1] = array[j];          // that remain
   }
   numElements--;              // decrement number of elements
   if (array) delete [] array; // delete the old array
   array = temp;               // and assign the new one
   return true;
  }
 }
 cerr << "Array: RemoveAt Error - element out of range\n"
      << "       It was " << i << " and numElements is "
      << numElements << endl;
 return false;
}

/*********************************************************/
/*                                                       */
/* RemoveAll: empties the entire array, resetting it to  */
/*            an empty state ready for reuse             */
/*                                                       */
/*********************************************************/

template<class UserData>
void Array<UserData>::RemoveAll () {
 if (array) delete [] array; // remove all elements
 numElements = 0;            // reset number of elements
 array = 0;                  // and reset array to 0
}

/*********************************************************/
/*                                                       */
/* GetNumberOfElements: returns the number of elements   */
/*                      currently in the array           */
/*                                                       */
/*********************************************************/

template<class UserData>
long Array<UserData>::GetSize () const {
 return numElements;
}

/*********************************************************/
/*                                                       */
/* Array: copy constructor, makes a duplicate copy of a  */
/*                                                       */
/* Note: what the elements actually point to are not     */
/* duplicated only our pointers are duplicated           */
/*                                                       */
```

```
/*******************************************************/

template<class UserData>
Array<UserData>::Array (const Array<UserData>& a) {
 Copy (a);
}

/*******************************************************/
/*                                                     */
/* operator=: makes a duplicate array of passed array a  */
/*                                                     */
/* Note: what the elements actually point to are not     */
/* duplicated only our pointers are duplicated           */
/*                                                     */
/*******************************************************/

template<class UserData>
Array<UserData>& Array<UserData>::operator= (
                                      const Array<UserData>& a) {
 if (this == &a) // avoids silly a = a assignemnts
  return *this;
 delete [] array; // remove existing array
 Copy (a);        // duplicate array a
 return *this;    // return us for chaining assignments
}

/*******************************************************/
/*                                                     */
/* Copy: helper function to actual perform the copy      */
/*                                                     */
/*******************************************************/

template<class UserData>
void Array<UserData>::Copy (const Array<UserData>& a) {
 if (a.numElements) { // be sure array a is not empty
  numElements = a.numElements;
  // allocate a new array the size of a
  array = new void* [numElements];

  // check for out of memory condition
  if (!array) {
   cerr << "Array: Copy function - Error out of memory\n";
   numElements = 0;
   return;
  }

  // copy all of a's pointers into our array
  for (long i=0; i<numElements; i++) {
   array[i] = a.array[i];
  }
 }
 else { // a is empty, so make ours empty too
```

```
  numElements = 0;
  array = 0;
 }
}

#endif
```

Please notice that for simplicity, I have ignored all out of memory situations with dynamic memory.

Review Questions

1. Describe three different types of user application programs for which the graph would be an ideal data structure. Be sure to explain why the graph is well suited for each of these.

2. Make a diagram showing how the two different graph traversal methods actually work. Under what kind of circumstances would a depth traversal be more desirable than a breadth traversal?

3. Explain why the largest value item must be stored in element 0 of the Heap implementation. How can the user do this when the most important item is the lesser value item? Explain in detail the difference.

4. Diagram how a priority queue could be used to hold a series of dictionary words for a spelling checker program.

5. Draw a diagram illustrating how the shortest path algorithm works using the **Pgm13b** graph when flying from Peoria to Burbank.

6. Draw an example of a digraph and an undirected graph. Show an example of a network graph.

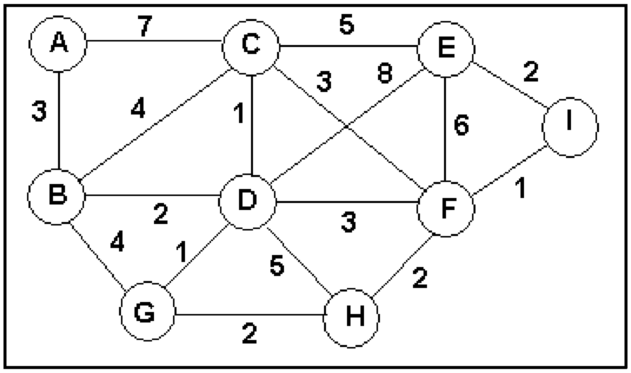


Figure 13.22 A Network

7. Using the network in Figure 13.22, draw a minimum spanning tree.

8. Using the network in Figure 13.22, what is the shortest path from A to I? From A to H?

9. Draw the vertices and edges if a graph were constructed to hold the data shown in Figure 13.22 as linked lists.

10. Draw the vertices and edges if a graph were constructed to hold the data shown in Figure 13.22 as arrays instead.

Stop! Do These Exercises Before Programming

There is another type of graph situation that is commonly needed, **topological sorting** in which a vertex B appears after vertex A, if there is a path from A to B. Of course, the graph cannot have a path from B to A if there is one from A to B or it would be circular.

Suppose that we needed to store a list of college courses. However, we must list all prerequisite courses before the course that needs those prerequisite courses. Thus, an edge from A to B means that course A must be taken prior to taking B.

Another powerful use is in the construction industry. Building a new road has many separate actions that must be done. Before certain ones can be undertaken, others must have already been completed. For example, the purchase of the right-of-ways must occur before you do the initial grading, which, in turn, must be done before the road bed can be formed, which must be done before the concrete can be poured, and so on. Here, **critical path analysis** is used to determine the scheduling of all of the tasks to get a project completed. The amount of time to accomplish a task is stored in the vertices which represent the tasks. The edges only indicate that a task must be completed before the next one can begin. With a critical path analysis, the two key questions are: What is the earliest completion date and what portions can be delayed a bit without impacting the completion date?

This series of exercises attempts to solve the following problem. Acme Construction wishes to make a bid on a construction project of a new building. They need first to know the minimum time it will take them to build the building. Secondarily, when troubles occur, they need to know which activities can be delayed without impacting that minimum completion time.

A Vertex node contains the string description of the task and its length to perform in days. If another task B depends upon this task A being finished first, then there is an Edge structure between them from A to B only but not from B to A.

1. Our programmer has devised the following pseudo coding to accomplish the task of performing a topological sort based upon depth first. The method is that each vertex must ahead of all other vertices that are its successors in the directed graph. Thus, we begin by finding a vertex that has no successors. It is then placed last into the ordered list. Then, recursively place all of the successors into the ordered list and finally place this one into the list.

```
void Graph::TopologicalSort (List& order)
    Clear the visited flags
    for all vertices v from the beginning to the end
        If ( ! Visited) RecursiveSort (v, order)

void Graph::RecursiveSort (Vertex& v, List& order)
    Mark v as visited
    for all of its edges
        If that edge's vertex is not yet visited
            RecursiveSort (that not yet visited one, order)
    end for
    insert this vertex v into the order at element 0
```

Convert this pseudo coding into working functions as part of our **Graph** template class. Note that you will need to create a different **Vertex** and **Edge** structure definitions from that used in **Pgm13b**.

2. Check out the solution by using the following input file that defines a construction project. The last two numbers represent the months to complete each of the two actions on that line, respectively.

```
From "Plans Drawn" To "Survey" is 2 1
From "Plans Drawn" To "Land Acquisition" is 2 3
From "Survey" To "Initial Grading" is 1 2
From "Land Acquisition" To "Initial Grading" is 3 2
From "Initial Grading" To "Fine Grading" is 2 1
```

From "Initial Grading" To "Bed Preparations" is 2 2
From "Fine Grading" To "Lay Road Bed" is 1 3
From "Bed Preparations" To "Lay Road Bed" is 2 3
From "Lay Road Bed" To "Final Landscaping" is 3 2

The resultant ordered list should contain the following.
Plans Drawn 2
Survey 1
Land Acquisition 3
Initial Grading 2
Fine Grading 1
Bed Preparations 2
Lay Road Bed 3
Final Landscaping 2

3. Next, devise an algorithm to display the critical path through the ordered list. The “Plans Drawn” vertex has two edges. The critical one is that edge which takes the longest time to accomplish. Thus, the “Land Acquisition” becomes the determining task before “Initial Grading” can occur. The routine should display the critical path and the accumulated total time through the project. The results should be something like the following.

```
The Critical Path to Follow
Plans Drawn        2
Land Acquisition   5
Initial Grading    7
Bed Preparations   9
Lay Road Bed      12
Final Landscaping 14
```

Programming Problems

Problem Pgm13-1 — The Grand Vacation

You have decided to take the summer off and visit a large number of US National Parks and Forests out west. The order that they are visited is important because you only have two summer months for the trip. You cannot visit them in a random order because of the excessive driving time. For example, it would not be wise to visit Glacier National Park in Montana, then drive to the Grand Canyon and then back up to the Tetons. So a minimum spanning tree would be helpful.

First, examine an US map and pick out 20 national parks, forest and lake resorts located in the western states. Assume that you are leaving from Denver, Colorado on your trip and that you intend to end up in Denver when you are finished.

Create an input file similar to that used in **Pgm13b** for the key routes among all of the parks and Denver. Now write an application program that loads in the graph and displays an optimum sequence of the visitation of these parks. It should also display the total miles traveled.

Problem Pgm13-2 — Cabling the Company's New Network

Assume that Figure 13.22 above represents your company's new proposed network of computers, where each vertex represents a computer installation. Assume that a weight of 1 shown in the figure represents 10 feet of cable. Your job is to cable those computers using the smallest amount of cabling. Write a program that determines how these computers should be cabled and the total amount of cable required.

Problem Pgm13-3 — Delivery Routes

Assume that Figure 13.22 above represents your company's delivery routes. Each vertex represents a city that you service. A distance unit of 1 represents 10 miles; a weight of 1 means 10 miles. Assume that any single service truck can cover 60 miles one way a day. Further, node D is the city from which your company operates.

Write a program that determines how the minimum number of service trucks your company will need to properly service each city in the network.

Problem Pgm13-4 — Airline Routes with Departure/Arrival Times

Of course, **Pgm13b** is overly simplistic. We know that departure/arrival times are vital to making travel plans. For example, the shortest route from A to B might require spending the night at C because of the arrival and departure times. Most travelers wish to avoid sleeping in an airport. So we really must add in support for timings.

For each Edge, which represents a flight from A to B, add two more variables: the float time of day that the flight leaves from the "from" city, or A, and the float time of day the flight arrives in the "to" city, B. Notice that if there are several flights from A to B throughout the day, there can be several Edge instances for these two cities.

Modify the input file for **Pgm13b** to handle a number of daily flights from each of the cities. Specifically, add in four flights scattered throughout the day from Peoria to Chicago and St. Louis. Add in only 1 morning flight from New York to Chicago. Add in a morning and evening flight from Chicago to the two West Coast cities and Denver. Similarly, add in two flights from St. Louis to the West Coast cities.

Now modify the shortest path functions of **Pgm13b** to handle this new consideration that for a connection to exist, any arrival to a node must occur before any departure from that city. That is, if going from A to B to C, then you must arrive at B before B's flight to C takes off.