Chapter 8
Data Structures

8.1 Introduction

The goals of this chapter are to present to the reader a collection of flexible data structure tools and to explain the C++ templates used to implement them. We will develop C++ classes for lists, stacks, queues, binary trees, and directed graphs. The material here is especially elegant and follows the style developed by Stroustrup[] for creating *generic* data structures. The power of C++ templates is that they can be instantiated with the particular records that we want to store and process in a data structure, and then the compiler will generate C++ object code for the data structures of these record types. It is code reuse at its best. The price is that we have to put creativity and time into the design of the template base classes. Well designed templates in data structures use derivation and this derivation has to be designed well to work at all. A less than optimal design in non-template work can be programmed around, but the precision tolerance for templates is much tighter. A well designed template is arguably one of the two most powerful paradigms of C++; the other being inheritance. When we instantiate a template with a data type, the operators and other member functions of the data type automatically are available for use in the member functions of the template class. Directed graphs are the climax of this chapter; a template for them became a must because of their use to model problems in diverse areas of science and engineering.

8.2 Pointers to Data versus Data

First we will develop the data structures for lists and their related stacks and queues. For sake of concreteness, let us assume that we have a user defined type, **record**, with the following data:

```
class   record {
    char* name;
    unsigned int age;
    ...
};
```

This class is an example of what will be referred to as a *data type* in the following. Any other data type with any other name would suffice; none of the following depends upon the data type. Also, the actual elements stored in the data structure can be pointers to objects of a data type; in fact the test programs for the implementations in this chapter store **record** pointers, not **records**. For maximum flexibility of the codes presented here, the classes have been designed to store either pointers for a data type or actual objects of a data type.

There is a fundamental difference in having the stored element being a pointer to data versus the data. Don't get confused if the actual data type also has pointers like **record** above; we are discussing the difference of storing **records** versus **record** pointers. There are two positive reasons for storing a pointer to a **record** instead of the actual **record**. The first reason is that a pointer is stored in only four bytes of memory and to *put it on a link* or *pull it off a link* does not require the call of a copy-initializer, but only requires us to pass four bytes as an argument or return four bytes as a return value. The second reason is not apparent until you start writing the data structure **iterator** classes. The null pointer is useful as a return value; it signifies that the function was not *successful* in finding what was requested of it, and as a signal to this it returns a null pointer (all 4 bytes zero). If, on the other hand, the function were successful, it would return a non-null address where a valid element is stored. Success is italicized because it is fine to have an empty list or an iteration with no more elements to process.The calling function has to check a returned pointer against the null pointer, and then have this logical equality validation determine the course of the future code execution. The conclusion is that the null pointer paradigm provides a simple method of exception handling that is invaluable for signalling that an iteration through a data structure is complete, an element was not found on a list, etc. A solution to allow for storage of both pointer and non-pointer data in a null pointer paradigm setting is to use addresses of links or data or both. If we are returning the address of pointer data, we are in effect returning a double pointer. In designing these template data structure classes, we want the design to allow three things:

(1) instantiate the template with non-pointer types
(2) instantiate the template with pointer types
(3) use the null pointer for signalling information

To accomplish (3) above, we will have the relevant functions return pointers to links, pointers to pointers to links, and pointers to the data objects stored in the data structure. This means that if the data type being stored is a pointer, then we will be returning pointers to pointers when accessing the data on the structure.

If we did not return pointers, then, for example, we would have to check for an empty list before attempting to get data off of it. If all classes had one special object to indicate an *exception*, then we could return this special object as an indication that the function was not successful. Keep in mind, an object with all bytes zero may not exist in a certain class, and even if it did, it may be a valid object with no special significance. An all zero element most likely could not do the job of the null pointer. Also, in general we do not want to add an extra exception data field to a record. Keep in mind, this crude germ of an idea for exception may be useful sometime; but here we don't want to add complexity to an already complex and intricate subject. Data structures are not as inherently easy as vectors and matrices to implement. In what follows, the symbol **T** will be used to indicate the actual type of object that will be stored on the link. **T** may be of a pointer type associated with a data type or it may be the data type. It is essential to understand that the object of type **T** on a link does refer to some piece of data, either indirectly via a pointer or directly as the data record. In fact, the object of type **T** may actually be another data structure object or pointer to such. For example, **T** could represent a pointer to an **ordered_list** object.

In what follows, sometimes, but not always, we will return the actual address of the data part (of type **T**) of the link. This address will be returned in a pointer variable of type **T***. If we are storing pointers to data in our structures, then we will still return a pointer to what is being stored. In this case, the returned variable of type **T*** will actually be a double pointer. To get the actual data referred to by the data structure, we will have to dereference the double pointer to get the single pointer (of type **T**) that is stored on the link. Then to get the underlying record (not stored actually in the data structure), we dereference this single pointer. This may seem complicated, but it actually works smoothly. Returning a pointer allows for exception handling to take place via the null pointer. This is good, and quite essential. Keep in mind that sometimes we will return the variable of type **T** stored on the link, not its address. This will always be made clear in the following. Furthermore, in what follows we are not assuming pointers are the data part of the link. If this were the case, the implementations would be written to take full advantage of null pointers. If the data were assumed to be pointers we could simply return the data on the link, and not a pointer to the data. If we returned a null pointer, then that would signal that the list was empty or a search was not successful. To get this capability with non-pointer data, we must return the address of where the data is stored on the link. This will be discussed again for the *remove* functions of lists, stacks, and queues.

8.3 Separating the Link and Stored Element

The actual structure of lists, stacks, queues, trees, and directed graphs does not depend upon the data stored in them or associated with a visual representation of them. This can be seen by inspecting any book with diagrams on these structures; the goal then is to produce, in C++, skeleton classes that correspond to these generic diagrams, and then somehow add in the application specific records via the C++ template paradigm, instantiating the template with that **record** or **record** pointer type. We first will create the skeleton base classes whose data members (not data records of type **T**) basically consists of pointers to data structure objects, and not to any real world record type objects. These base classes do not depend on **T** in any way. Then off of these base classes (not dependent on T), we will derive template classes with parameter T. The derived template classes will basically serve one of two purposes. In the case of a link type class, a data member of type **T** will be added to the derived class. In the other type classes, such as an iterator class, the functionality of the derived class will convert base class link type pointers to derived class link pointers. We put derived class link pointer objects dependent on the type **T** in base class link pointer variables in order for the skeleton classes to *move* them around, and then when it comes time to use them, we want them converted back to derived class pointers. This approach is thoroughly discussed in Stroustrup[] with many code segments illustrating these ideas.

First, we will present the classes for building lists, stacks, and queues. Our list will consist of a collection of individual links connected to one another via pointers; the list is *doubly linked* in that each link contains both a pointer to a previous link and a pointer to the next link. No matter what we want to store in a list we will need a link pointer called **previous** and a link pointer called **next**. This common functionality will be factored into a common base class, called **list_link**, and the derived template class will be called **T_list_link**. Their class definitions follow:

```
struct list_link {
    list_link* next;
    list_link* previous;
    list_link() { previous = next = 0;}
};

template<class T> class T_list_link: public list_link    {
    T  data;
public:
    T_list_link(const  T& a) :  list_link(), data(a)  {};
    T get_data() {return data};
    T* get_ptr_data() {return &data};
};
```

We use the constructor in the above template to build the **T_list_link** *holding* an object **a** of type **T**. Notice that all data members and functions of **list_link** are **public**; by definition, all members of a **struct** are **public**. Furthermore, the member **data** of the **template** class is constructed by a copy-initializer if **T** is a user defined class and if there is no copy-initializer available (as in the case of pointers) a bitwise copy is made. The users of these data structure codes are responsible for how copies of objects of type **T** are made. The most flexibility will be obtained if **T** represents pointers to some class or if **T** is a class using reference counting. Then any changes made to the **records** stored directly or indirectly (via a pointer) in the data structure will be made on the original.

It will not be necessary to write a destructor for **list_link** or **T_list_link**. The compiler will automatically generate a destructor for **list_link** that will only free the shallow memory of **list_link**. We do not want the **list_link** destructor to invoke **delete** on pointer data members of **list_link**. This would wreck havoc through our linked list. Since **T_list_link** is a **list_link** with an added data member of type **T**, the automatically generated destructor for **T_list_link** will be perfect. This default destructor calls the destructor for its **T** data member and then the destructor for the base **list_link**. Realize that if **T** is a pointer type, the destructor call on **T** is vacuous.

We should categorize the three main types of **T** data members that we should store on a **T_list_link**. The categories are (1) pointers, (2) types with deep memory and reference counting to manage it, and (3) types with no management of deep memory.

If **T** is a pointer, then any implementation of the destructor for the **T_list_link** should not have the statement **delete data**, and thus the object that is pointed to will not be destroyed by the **T_list_link** destructor. If the value of this pointer has been saved somewhere, we will still have access to the actual data record after we **delete** a pointer to the **T_list_link**. Also, any changes to this pointed to object while it was on the list will remain in effect.

If **T** is not a pointer type, then the **T** destructor will automatically be called if we **delete** a pointer to a **T_list_link**. This is a property of composition. If **T** has reference counting, the reference count will be decremented and if still positive a copy of **data** with any changes will exist off the **T_list_link**. In other words, deleting a pointer to a **T_list_link** does not necessarily destroy the data if **T** has reference counting. In fact, while the link is in existence the reference count should be at least one because when the link was constructed the reference count was incremented.

If **T** is a type that does not require memory management, then a destructor for **T** does not have to be written. In this case, the most common form for **T** would be basic types such as **int**, **long**, etc. **T** could include types whose data members were basic types that require no memory management. In this case, when a **T_list_link** goes out of scope, so does all reference to the **data** stored on it. This may be fine in many circumstances.

Another possibility is to have a type **T** with management of deep memory that does not use reference counting (thus a non-trivial destructor). In this case, deleting a **T_list_link** will invoke the **T** destructor on the **data** in a non-passive manner; make sure the **T** destructor is compatible for this action and what you want with your underlying data. If you are a beginner, you could have problems in this case.

It should be noted that if the data stored were pointers, then we could iterate through the data structure, accessing and calling **delete** on each pointer. Keep in mind this would still leave the data structure intact. This method will actually be employed, but the destruction of pointer data will not be a member function of the data structure class or any of its iterator classes. It is appropriate to have an application class that contains a data member that is an instantiated data structure and an associated application member function that performs the destruction of data pointed to by pointers stored on the data structure.

8.4 The Base List Class

The base list class will be independent of the underlying record type that will be associated with the **T_list_links**. We will then derive a list template class off of the base class that will take into account the actual type **T** on the **T_list_link**. For convenience, our base list, called **common_list** will contain exactly two data members, representing the head and tail of the list. An abbreviated class definition (no code bodies) for **common_list** is:

```
class common_list {
protected:
    list_link* head;
    list_link*   tail;
    void add_internal_link(list_link* ptr, list_link** llpptr); //  needed for ordered lists
public:
    friend list_iterator;
    common_list() { head = tail = 0; }
    common_list(list_link* ptr) {  head = tail = ptr; }
    void  add_to_tail(list_link* ptr);
    void  add_to_head(list_link*  ptr);
    list_link* remove_head();
    list_link* remove_tail();
    int empty()  { return (head == 0 ); }
};
```

The functionality has been designed in order that an empty list has both **head** and **tail** set to null. The functionality of the base class is obviously built around**list_link** objects; we can add **list_links** to the list and remove **list_links**. This can be done only at the **head** and **tail** with the given **head** and **tail** functionality. The function **add_internal_link** is used in the **ordered_list** class to insert a link internally to a list for the purpose of maintaining order. Now, the question arises of how we store data on the base list. The secret lies in that each **list_link** has pointers to **previous** and **next list_links**, and that in these pointers we can put addresses of derived class objects, **T_list_links**. The techniques in this chapter depend on the C++ paradigm of storing derived class pointers in base class pointer variables. If we are factoring out common structure with the data stripped away, we must use a pointer to this common structure. In this manner, we can use the functionality of the base class to process pointers to derived class objects.

It should be noted that the base class list here does not assume any responsibility for creating or destroying links. Its functionality takes a pointer to a link and puts it on the list and disconnects a link from the list, returning a pointer to it. It is true that we could write a **common_list** member function that would iterate through the list and destroy the links by calling **delete** on the link pointers, but it makes more sense to do this on the derived class **T_list_links**.

Destructors for these links should not call **delete** on any data members that are pointers to other links; if you **delete** a link pointer, you want that **delete** call to free up the shallow memory of the link. For that reason, the automatically generated default destructor will be fine for **list_link** and **T_list_link**. The **T data** member, if it is not a pointer or basic C type, will have its destructor automatically called if you delete a pointer to a **T_list_link**. If we are using reference counting on **T**, this destruction of the **T data** still leaves the original object, not on the link, intact; it will have to be accessed via a variable other than the **T_list_link** object.

8.5 The Derived List Class

The derived class template, **template<class T> list** serves to create a class to which we can attach a record to the **list_link**. Its definition is:

```
template<class T> class list :  private common_list   {
    void  add_to_head(const T&  info);
    void  add_to_tail(const T&  info);
    T  get_head_data();
    T  get_tail_data();
    void destroy_list();
    int empty() {return common_list::empty();}
    friend  class  T_list_iterator<T>;
};
```

The responsibility of managing the record type falls to the derived class. Notice that there is no new data at all, and the functionality appears at a quick glance to be a redo of the base class. There is no constructor given, so by default the only constructor called is the default constructor of the **common_list,** which is fine. It will set **head** and **tail** pointers to null. On closer inspection, we see the **add_to** member functions have a **T** reference parameter instead of a **list_link** pointer, and the **get** member functions, corresponding to the base class **remove** functions, return objects of type **T** instead of **list_link** pointers.

Notice in the above class definition that **class list** is followed by the modifier **private common_list**. This is an example of *private inheritance*, sometimes referred to as *implementation inheritance*. In private inheritance we want to reuse the functionality of the base class, but we don't consider a derived class object *to be* an example of a base class object. Public inheritance is used when the derived class object *is* a base class object. There is a gray area in making the decision between private inheritance and public inheritance. For these data structure classes, the base classes were designed for the sole purpose of allowing derived classes to inherit the base functionality that processes pointers. This is clearly an example of implementation inheritance. For more discussion on the choice between private and public inheritance see Coplien[].

Let's take a look at the **add_to_head** function:

```
template<class T> void   list<T>::add_to_head(const T& info)  {
    common_list::add_to_head(new T_list_link<T> (info) );
}
```

As can be seen, this function creates the **T_list_link** on which a copy of the **info** data will reside. This copy of **info** data is made by the **T** copy-initializer that is present on the initializer list of the **T_list_link** constructor. Then a pointer to this derived class **T_list_link** is passed on to the corresponding base class **add_to_head** which among other things stores this derived class pointer in a base class pointer object.

Let's take a look at the **get_head_data** function:

```
template<class T> T list<T>::get_head_data() {
     T_list_link<T>*   llink = (T_list_link<T>*)common_list::remove_head();
     T  info = llink->data;
     delete  llink;  //  does  not  destroy  T  part  of  llink
     return   info;
}
```

First of all, **get_head_data** should not be called for a list that is empty; if the list is empty, then llink will be assigned the null pointer. It makes no sense to then access **llink->data** and construct an object **info**. The function **list<T>**::**empty(),** which merely calls c**ommon_class::empty()**, can be used to check for this prior to a call of **get_head_data**.

The central action that these access derived template functions must perform is to recast a base class link pointer to a derived class link pointer; after all, that was what was actually stored in the base link variable. After this recast, we can access the **data** information stored on the **T_list_link**. This data is copied by the copy-initializer for **T**, which may be a default bit-wise copy depending on the implementation of **T**, into a temporary **result** of type **T** which we will return. Then **delete** is called on the pointer to the **T_list_link**. This will make the link copy of **data** go out of scope (if **T** has a destructor it will be called automatically) and free the shallow memory of the link. Remember the **T_list_link** destructor will also call the **list_link** base destructor. Notice that the destruction of the link is the *inverse* of its construction by **list<T>::add_to_head(T&)**. Keep in mind **get_head_data** is destructive in that it destroys the link on which object of type **T** resided.

If you were going to exclusively use pointers for **T**, you could rewrite these implementations to obtain superior error handling. As it is, if you don't know the empty status of the list, you must check it with **empty()**. Because of the possibility of an empty list, you must program your calling function for the possibility that no links exists on the list by putting e**mpty()** in an **if** clause.

If **T** were guaranteed to be a pointer, then we could check **llink** for a null value and immediately return a null pointer to signify that we could not get the piece of data stored on the head link because the head link did not exist. Storing pointers to data on a link in place of non-pointer data would offer this convenience. In order to make this code as flexible as possible, no assumption was made that **T** is a pointer. Stroustrup [] actually creates two sets of data structures: (1) one for **T** when it is a non-pointer type and (2) one for **T** when it is a pointer type. This is actually preferred, but data structures is not the main topic of this book.

If we wanted to access the object of type **T** on a **T_list_link** without destroying the link, we could return the address of the object on the link in a type **T*** variable. This would allow for the return of a null pointer signalling a list did not have a link with the object on it. This is not feasible for **get_head_data** since it destroys the **T_list_link**. We have *iterators*, discussed in the next section, to access elements off of the list in a nondestructive manner. For **iterators**, a pointer to the object on the link is returned. Iteration via **iterator** objects is non-destructive.

8.6 Iterators for List Classes

An iterator is a class that provides us with the functionality to iterate through a data structure, returning the object stored on each link. Like in the above classes, a base class iterator function will return a link pointer while the corresponding derived class function will take that link pointer and return a pointer to the data stored on that link. Since a completed iteration is not the same as a list being empty, the **empty()** function is not used to signal that an iteration has processed all links on a list. We need to be able to determine when the iteration through the list has been completed; the solution is to return a pointer to an object stored on a link with a returned null pointer indicating completion of the list iteration. This is possible since iteration does not destroy links. The base class iterator implementation is the following:

```
class list_iterator {
    list_link* current;
    common_list* listptr;
public:
    list_iterator(common_list&  cl)   {
        listptr =  &cl;
        current = listptr->head;
    }
    list_link*  operator() ()  {
        list_link*  result =  current;
        if  (current)   current   =  current->next;
        return  result;
    }
};
```

An iterator contains a pointer to the list that it is iterating through and a pointer to the **current** link that acts as a window into the data structure, exposing a single link. An **evaluation operator** for the iterator class returns a pointer to the current link immediately after advancing the window to what will be the current link in the next evaluation. If **a** were an iterator and **ptr** a **list_link** pointer then the **evaluation operator** is called by **ptr = a()**. **a** is what is referred to as a functionoid. In this style, **a()** looks like a function with identifier **a** being invoked. Actually **a** is an object and the **()** symbol indicates an **evaluation operator** was invoked. We could have written a member function that was not an **operator** to do this also, and given it some name. When we evaluate at the tail link, current will be assigned the value of **tail->next,** which is zero, and the pointer to the **tail** link will be returned. In the next call to the **evaluation operator**, a null pointer will be returned.

The **evaluation operator** of the derived class iterator takes the link pointer returned by the base class iterator, recasts the pointer to a **T_list_link** pointer, and returns a pointer (of type **T***) to the actual object of type **T** stored on the **T_list_link**. If the returned link pointer were null, then a null **T** pointer would be returned. This would indicate that the iteration was complete. The iteration loop that is invoking the evaluation operator should test for a returned null pointer of type **T***, and suspend looping when it finds one.

The code for the derived class template evaluation operator is:

```
template<class T> T* T_list_iterator<T>::operator() () {
    list_link* ptr = list_iterator::operator() ();
    if (ptr)  return  ( ( T_list_link<T>*)ptr)->get_ptr_data();
    else  return  0;
};
```

8.7 Stacks and Queues

Actually all the functionality needed for stacks and queues has been written in the above list implementation. An elegant approach would be to do a template derivation off of **common_list** for both stacks and queues. A stack adds data to the head and removes data from the head. This is called **push** and **pop** respectively; we still need to check for an empty stack before we pop though since this implementation is written to return an object of type **T** that may or may not be a pointer. A queue (FIFO) on the other hand adds data to the tail and removes it from the head. The functionality of the derived classes, **stack<T>** and **queue<T>**, are renames of parts of the functionality of the class **list<T>**. For an application of stacks and queues, see the implementations of the binary tree iterators presented later in this chapter. These applications use **empty()** before removing from the stack or queue.

8.8 Binary Tree Data Structures

In the above, we did not require any ordering relation on the data being stored. We could have inserted items into a list in such a manner to maintain an ascending or descending order if we were given a function that defined a total ordering on the data (such as <). Binary trees are often used to provide an implicit ordering of data, and the ordering relation on the type **T** objects determines how they are built. If a well balanced binary tree holds **n** pieces of data, we can find a piece of data (or determine that it is not on the tree) in no more than log **n** (base 2) number of accesses, not including the root. Some binary trees really don't represent ordered data, such as an arithmetic expression tree.

8.9 The Tree Links

As with lists, we will have base classes that will only depend upon the base link type, and then our derived class templates will insert and remove objects of type **T** on and from the derived class links respectively, and recast base link pointers to derived link pointers. The class definitions for the tree links are:

```
struct tree_link {
    tree_link* leftptr;
    tree_link*  rightptr;
    tree_link(tree_link*  l,  tree_link*  r)  {  leftptr  =  l;   rightptr  =  r;  }
    tree_link()  {  leftptr  = rightptr  =  0; }
};

template<class  T>  class T_tree_link  :  public  tree_link  {
    T  data;
public:
    T  tree_link();
    T_tree_link(T_tree_link* ,   T_tree_link*,  const  T&);
    T_tree_link(const  T&  a)
    T* get_ptr_data() {return &data); }
    friend class T_tree<T>;
    friend class BFS_T<T>;
    friend class DFS_T<T>;
    friend class inorder_T<T>;
};
```

The key to the use of the left and right children of a parent link is that the data associated with the left child is *less than* the data for the parent and the data associated with the right child is *greater than* that of the parent. The portion of the data that we order on is called the *key* field. In a simple implementation of a binary tree, we might assume that two pieces of data with the same value for their key fields are identical, and thus there is no need to insert a repeat. In a more complex case where identical keys do not imply *equality* of two data instances and we want to insert a link with the same key as that of a link already on the tree, we must decide what we want to do. This will be discussed in the following sections. Almost all the comments for **list_link** and its derived **template** are appropriate for the tree links, and will not be repeated.

8.10 The Tree Classes

The base tree class encapsulates a pointer to the **root** link of a tree; its class definition with declared iterator friends is:

```
class base_tree  {
protected:
    tree_link* root;
public:
    base_tree(tree_link*  ptr )  {  root  = ptr; }
    friend  BFS_base;
    friend  DFS_base;
    friend  inorder_base;
};
```

BFS_base is a breadth first tree iterator, **DFS_base** is a depth first iterator, and **inorder_base** is an inorder iterator for **base_trees**. Note that in the actual header file (section 8.15.3.1) these iterator classes must be declared forward of the **base_tree** definition. Then later on (after **base_tree** is defined) we define the iterators. We must continually make these forward declarations while defining related data structure classes.

template<class T> class BFS_T is the syntax for a forward declaration for a derived iterator class. The class definition for the derived class template is:

```
template<class T> class T_tree : private base_tree {
private:
    void  add_to_tree(T_tree_link<T>**  ptr,  T* pkeyed_rec);
    T_tree_link<T>** aux_search_insertion_ptr(T_tree_link<T>** tpptr, T* pkeyed_rec);
    void destroy_subtree(T_tree_link<T>* ptr);


public:
    T_tree(T_tree_link<T>*  ptr  =  0)  :  base_tree(ptr)   {}
    void insert(T* pkeyed_rec)   { add_to_tree( (T_tree_link<T>**)(&root),  pkeyed_rec);  }
    T_tree_link<T>** search_for_insertion_ptr(T *pkeyed_rec);
    T_tree_link<T>* search_insert(T* pkeyed_rec, int& match);
    void destroy_T_tree() { destroy_subtree((T_tree_link<T>*)root); }
    int empty() { return (!root); }
    friend  class  BFS_T<T>;
    friend  class  DFS_T<T>;
    friend  class  inorder_T<T>;
};
```

The functionality for **T_tree** provides for two types of inserts; one does not allow for modification of tree data in the case of duplicate key matches and the other does. The functions **inser**t and **add_to_tree** alone are used for the case when no changes are made upon a key match. For the case of modification upon key match, we use additionally use **search_for_insertion_ptr** and **aux_search_insertion_ptr.**

The function **void insert(T* pkeyed_rec)** takes a pointer to an object of type **T** and inserts a copy of the object, by dereferencing the pointer, in the correct position based on an ordering for objects of type **T** on the tree. Keep in mind, **T** may be a pointer type. For this implementation, a binary function **compare** is required to be defined by you for the type **T**; also if one uses **insert** to put onto the tree an object with a key that already is present on the tree, the **insert** function terminates without adding the object. **a** and **b** of type **T** are considered to be *equal* if **compare(a,b)** returns **0.** In the particular application that was used to test the trees, the type **T** was a **record*** type. In order to handle this, a friend function **compare(record*, record*)** was defined that used the **C** library function **int strcmp(char*, char*)** to compare the **name** field of two **records**. Remember that we cannot define member functions for **ints**, **floats**, **char***, **record***, and other basic **C** intrinsic types. For this reason, a non-member **compare** function had to be written instead of using **operator<**, which is not valid for **T** of a pointer type. In other words, we can order pointers based on an order defined on what they point to, but we cannot do it with a member function. These are the design trade-offs that occur because **T** is allowed to be a pointer type. If we had an implementation that required **T** to be of a pointer type, than we could dereference variables of type **T**, assume that ***T** variables had an **operator<** member function defined and use it in **add_to_tree**. To quote Stroustrup[], there is no silver bullet. One should also read about **comparator** classes in Stroustrup[]. They can provide more elegance, but at a higher level of sophistication.

Keep in mind as you read the following that the data stored on a **T_tree_link** refers in some way to an actual **record**. If we wanted to allow *repeat data* to be inserted on the tree upon key matches, then the **insert** function would not provide the necessary functionality. The function **insert(T* pkeyed_rec)** is used as the public interface function to add a piece of data; the function that does the labor of traversing the tree to find if a piece of data is represented on the tree or not, and to insert it if it is not, is **template<class T> void T_tree<T>::add_to_tree(T_tree_link<T>** ptr, T* pkeyed_rec). add_to_tree** is first called by **insert** and then **add_to_tree** recursively calls itself until it finds a null pointer (the dereferenced double **ptr** argument of **add_to_tree** is null)) or it finds data on a link with the same key value as that associated with its other argument, **pkeyed_rec**. If it finds *like (*a key match*)* data, all the function calls backing up to and including **insert** are exited with no change in the tree. If a key match is not present on the tree, then at some time **add_to_tree** will be called with an argument that dereferences to a null link pointer. When **add_to_tree** is called with an argument **ptr** that dereferences to a null link pointer, then a new link is created with a copy of the dereferenced **pkeyed_rec** on it, and then the address of this link is assigned to ***ptr**, the dereferenced double pointer that contained null that was passed in on the last call of **add_to_tree**. In doing tree work, when we come to a null pointer, this is an indication that our object that we are searching for and want to add is not already on the tree. It should be noted that the recursive call of **T_tree<T>::add_to_tree()** is with the same **T_tree<T>** object. We are not visiting the various **T_tree_links** and constructing trees with each of these links as the root. Instead, we pass the address containing the address of the link that is the graphical root of the subtree (not a declared **T_tree**) that we must search. If this were a C program, the original root of the whole tree would not be passed along in the recursive function calls. Here it is passed along via the **this** pointer to the T_tree, but not used as we bury ourselves deeper in recursive calls.

The member functions **insert** and **add_to_tree** discussed above provide an amount of functionality that does the bare minimum: if an object of type **T** does not have a key match on the tree, they put the object on a link and insert the link on the tree. Also, they give no signal an insert was performed. These functions also do not directly provide for a simple lookup to see if a key is represented on the tree. The additional member functions used for a more advanced binary tree are:

(1) **T_tree_link<T>** search_for_insertion_ptr(T* pkeyed_rec);**
(2) **T_tree_link<T>** aux_search_insertion_ptr(T_tree_link<T>** tpptr,
T* pkeyed_rec);**
(3) **T_tree_link<T>* search_insert(T* pkeyed_rec, int& match);**

If we want to search solely for a key match, then the two member functions **search_for_insertion_ptr(T* pkeyed_rec)** and **aux_search_insertion_ptr(T_tree_link<T>** tpptr, T* pkeyed_rec)** will suffice. **search_for_insertion_ptr** is the public interface function that returns a variable of type **T_tree_link<T>**** and **aux_search_insertion_ptr** is its private recursive helper function that traverses the tree looking for a key match**.** To search for a key match, we produce a record of type **T**, call it **T_object** (this may be a pointer to another object), and instantiate **T_object** with the key value. Then we invoke **search_for_insertion_ptr(& T_object);** this invocation returns a double pointer, call it **insert_location**, which must be evaluated. If **insert_location** is null, this indicates that the tree has no link address assigned to **root** and thus the tree is considered empty with an implied no key match. If **insert_location** is not null, then we dereference it. If ***insert_location** is not null, then ***insert_location** is the address of the link that contains the record with a key match. **insert_location** is the address of the location where our tree is storing this link address. If ***insert_location** is null, then there is no key match, but **insert_location** is the address where we would store the address of a link that would be a key match. If we wanted to insert a link onto to the tree with this key we would put the link address in the memory stored at **insert_location**. In summary, **insert_location** provides access into the tree where a link with a key match should be or actually is stored.

The function **search_insert** allows one (1) to insert a record if its key is not already present on the binary tree or (2) to get the address of the tree link where a match has been found. With knowledge of a match and access to the data via the link pointer, we can modify the data associated with the tree link. How does one use these functions? One must construct an object of type **T**, call it **T_object**, with the underlying key field instantiated; nothing else needs be done to the **T_object** as far as the tree functionality is concerned. One also needs a local variable, call it **match_local**, that will indicate whether a match was found on the tree. Assume the pointer to the object **T_object** being inserted is called **pkeyed_rec (pkeyed_rec == &T_object)** and that **pptr** is a local variable of type **T_tree_link<T>***. Then the statement **pptr = search_insert(pkeyed_rec, match_local)** will in all cases return a pointer to a tree link that has the same key as the associated key of **pkeyed_rec**. If there is indeed a key match, **search_insert** does nothing to the data on the matched link. If there is no key match, **search_insert** creates and inserts a new link with a copy of **T_object** on it. In both cases, a pointer to the link is returned. **match_local** is passed by reference, and it will be set to **1** if the key was present on the tree and **0** if not. With this knowledge and a pointer to the link that leads to the actual data being stored, one can modify the data on the link accordingly. It is the responsibility of the function invoking **search_insert** to make these modifications to the copy of **T_object** stored on the link pointed to by the pointer returned by **search_insert**. Note that **T_tree_link::get_ptr_data()** returns the address of the data stored on a tree link; we can call this function with the pointer returned by **search_insert**.

search_insert calls the function **search_for_insertion_ptr(T* pkeyed_rec)** to determine whether the tree is empty and if not whether the key is already present. If the tree is not empty, **search_for_insertion_ptr** calls the recursive function **aux_search_insertion_ptr** to return the address of where the address of the link with the matched **key** value is or will be stored. This return value is of type **T_tree_link<T>****. **search_insert** takes this double pointer and, if null, creates a tree with root consisting of the link with the **T_object** on it; otherwise it either uses **add_to_tree** to insert a link in its proper location (determined by **search_for_insertion_ptr**) or sets match equal to **1**, indicating the **key** is already present. Again, after **search_insert** does its job, it is the responsibility of the calling function to modify the data referred to by the returned link pointer in whatever way is appropriate. The value of **match_local** will inform the function calling **search_insert** how to treat the data on the selected link.

The implementation presented here may not be what you want in all cases, but it illustrated a viable approach. In what was presented above, it would help to imagine that the data stored on the tree link is a pointer to a record that contains a key field as well as a list of some type of data. When we get a match of key fields, we will modify the linked list data member of the imagined record in some way. If we don't get a match, then we put a pointer to the new record, made up of a key field and associated list, on a link, and then insert the link on the tree in its proper location, based on the appropriate compare function. In the functionality presented, no assumption is made on the data type actually being stored other than that there is a **compare** function available. Again, imagining the idea of a linked list stored on the link of a binary tree will be helpful. The linked list can keep track of the various pieces of data with the same key.

8.11 The Tree Iterator Classes

These binary trees can be traversed in many different ways. In this implementation, we present functionality to traverse them in three ways: (1) breadth-first traversal, (2) depth-first traversal, and (3) inorder traversal. The functionality that performs these traversal is called *iterators*, like that for linked lists. Here though, the code is much more complex. As in all classes in this data structure chapter, the intricacies of iterators are independent of the actual data being stored. Thus, the template paradigm is again appropriate.

Breadth-first iteration visits the links that are on the same level of a tree, and proceeds to visit all the links of the next level, and so forth. Let's take a look at the base class for breadth-first iteration:

```
class BFS_base {
private:
    tree_link*    current;
    queue<tree_link*> bfs_queue;
public:
    BFS_base(base_tree&);
    tree_link* operator() ();
};
```

The data members consist of a pointer to the current link and an auxiliary queue that is needed to hold **tree_link*** values. The constructor **BFS_base (base_tree&)** takes the root of the tree argument and puts it on the auxiliary queue. The **operator()()** will then remove the link at the head of the queue, assign it to **current**, then take **current**'s left and right children (if they exist), add them to the tail of the queue, and then return **current**. In this way, we iterate through the tree in a breadth-first manner, returning pointers to the links. Before we remove a **link***, we must check that the queue is not empty; as previously discussed, our template implementation for lists is designed to store pointer as well as non-pointer objects, and because of this our implementation does not permit removal from an empty queue. When the queue is empty, the iteration evaluator returns a null pointer. This is our signal that the iteration is complete.

The depth-first and inorder iterators are more subtle, and they use stacks instead of queues. You should examine the implementations of these in section 8.15.3.1, and a post-order iteration is assigned as an exercise.

We will present the class template for the breadth-first iterator and the implementation for its evaluation operator:

```
    template<class T> class BFS_T :  private  BFS_base  {
    public:
        BFS_T(T_tree<T>& tr)  :   BFS_base(tr)  {}
        T*   operator()();
    };

    template<class T>    T*   BFS_T<T>::operator()()    {
        T_tree_link<T>*   tlink;
        tlink =  (T_tree_link<T>*)(BFS_base::operator()());
        if  (tlink)  return  &(tlink->data);
        else  return  0;
    }
```

The template iterator class, **BFS_T**, is simply the privately inherited base class iterator, BFS_base, with an evaluation operator parameterized with **T**. The derived class evaluation operator invokes the base class iterator, converts the returned **tree_link*** to a **T_tree_link***, and then returns a pointer to the data stored on the **T_tree_link**. When the iteration is complete, a null pointer of type **T*** is returned.

8.11.1 Compiler Switches for Lists and Trees

In order to successfully compile templates and the source files that use their instances with the Borland compiler, we make use of compiler switches inserted in the source code. One should consult your compiler manuals for a full explanation.

The entire style in this book is to put class definitions in a header file, implementations of the member functions (those not defined in the class definition) in the associated implementation file, and a **main** function with other functionality in a third file. Both the **main** file and the implementation file include the header file, and each of these two files is the target of a separate compilation.

The Borland compiler switches that we use are from the **-Jg** family of switches. At the end of the implementation file, we want to generate instances of the **template** for types we want to parameterize with. If we wanted to use the list functionality for a pointer type, **record***, the following code inserted at the end of the implementation file for lists would do it:

```
#pragma   option   -Jgd
class record;
typedef  list<record*> fake_record_ptr_list;
typedef  T_list_iterator<record*> fake_record_ptr_list;
```

If we wanted to store **record***s on a tree, then we must instantiate the stack and queue classes for **tree link***s. The following would do that:

```
#pragma option -Jgd
class tree_link;
typedef stack<tree_link*> fake_tree_link_ptr_stack;
typedef queue<tree_link*> fake_tree_link_ptr_queue;
```

It might be more safe to include a file with the appropriate **template** instances to be generated, and then modify this file when different instantiations of the **template** are desired. In other words, put the above blocks of code in an **include** file and in place of the above block in the implementation file put a **#include** statement. In this manner, we would edit the **include** file and not the implementation file, which is good software engineering. Above we have the pragma option **-Jgd**; this is the appropriate compiler switch to generate the necessary object code. We only use this switch in one file, the class implementation file, for the class **template**.

The include file for the pragma options discussed above is:

```
// file prglist1.h
#define TREE_LINK_PTR
#define RECORD_PTR
#pragma option -Jgd

#ifdef TREE_LINK_PTR
     class tree_link;
     typedef  list<tree_link*> fake_tree_link_ptr_list;
     typedef  stack<tree_link*> fake_tree_link_ptr_stack;
     typedef  queue<tree_link*> fake_tree_link_ptr_queue;
#endif

#ifdef RECORD_PTR
     class record;
     typedef  list<record*> fake_record_ptr_list;
     typedef T_list_iterator<record*> fake_record_ptr_list_itr;
#endif
// end of prglist1.h
```

See sections 8.15.1.2 and 8.15.1.3 for the use of the pragma file in a template list Such a file makes the template implementation more elegant and safe. We can edit the **prglist1.h** file with new types and use the comment sign, //, to turn off an instantiation. This allows for change without editing the implementation file. **prglist1.h** was the pragma file that is used to instantiate the list functionality. For trees, we need another file, **prgtree2.h** to include at the end of the tree implementation file (sections 8.15.3.2 and 8.15.2.3).

```
// file prgtree2.h
#define RECORD_PTR
//  #define RECORD_MAJITEM_PTR
#pragma option -Jgd
#ifdef RECORD_PTR
     #include "record.h"
     typedef   T_tree_link<record*> fake_record_ptr_T_tree_link;
     typedef   T_tree<record*> fake_record_ptr_T_tree;
     typedef   inorder_T<record*> fake_record_ptr_inorder;
     typedef   BFS_T<record*> fake_record_ptr_BFS;
     typedef   DFS_T<record*> fake_record_ptr_DFS;
#endif

#ifdef  RECORD_MAJITEM_PTR
     #include "recmaitm.h"
     typedef  T_tree_link<record_majitem*>  fake_rec_maitm_ptr_T_tree_link;
     typedef   T_tree<record_majitem*> fake_rec_maitm_ptr_T_tree;
     typedef   inorder_T<record_majitem*> fake_rec_maitm_ptr_inorder;
     typedef   BFS_T<record_majitem*> fake_rec_maitm_ptr_BFS;
     typedef   DFS_T<record_majitem*> fake_rec_maitm_ptr_DFS;
#endif
```

In all other source files, in particular the **main** file, in which we will use instances of these templates, we must use the compiler switch **#pragma option -Jgx** at the top of the file. This will compile the file with *external references* to the template instances; this means that the template instances will not be generated for these files during their compilation. They will be generated in the one file with the **-Jgd** option, and then these external references will be resolved at link time.

8.12 Ordered Lists

The class **ordered_list** adds order to the linked list implementation. **ordered_list** is a privately derived class of **common_list** and it uses the template **T_list_link<T>** class for its links. Its definition and corresponding iterator definition are:

```
template <class T> class ordered_list : private common_list {
private:
     T_list_link<T>** search_for_insertion_ptr(T* key, int&  match);
     T_list_link<T>** aux_search_insertion_ptr(T_list_link<T>** lpptr, T* key,
          int&  match);
     void  add(T_list_link<T>** lpptr, T* key);
public:
     ordered_list()  :  common_list() {};
     int empty() { return common_list::empty(): }
     void destroy_ordered_list();
     T_list_link<T>* search_insert(T* key, int& match);
     friend class ordered_list_iterator<T>;
};

template <class T> class ordered_list_iterator : private list_iterator {
public:
     ordered_list_iterator(ordered_list<T>& tl)  : list_iterator(t1) {}
     T* operator()();
};
```

The functions **search_insert**, **search_for_insertion_ptr**, **aux_search_insertion_ptr**, and **add** correspond in functionality to their namesakes in the binary tree implementation of the previous section. **add_to_tree** was named as such in the previous section instead of merely **add** because of a linker ambiguity problem on a particular UNIX based compiler. The Borland C++ compiler had no such ambiguity problem with two overloaded **add** functions between **ordered_lists** and **T_trees**. Sometimes one has to coerce templates into success.

A **compare** function is necessary for the data type **T** as in the tree implementation; remember **T** can be a pointer type and **compare** will have to dereference the pointers in order to compare the underlying data. The **add** function uses **base_list::add_internal_link** to insert a **T_list_link** into the non-boundary part of the **ordered_list**.

8.13 Directed Graphs

8.13.1 Introduction

A directed graph is a structure with vertices joined by edges that have a direction associated with them. In what follows the vertices will be labelled with the natural numbers, beginning with zero. An edge will not be labelled with a number, but will be identified with the vertices to which it is attached. Figure 8.13 illustrates a typical directed graph, complicated enough to be an illustrative example.

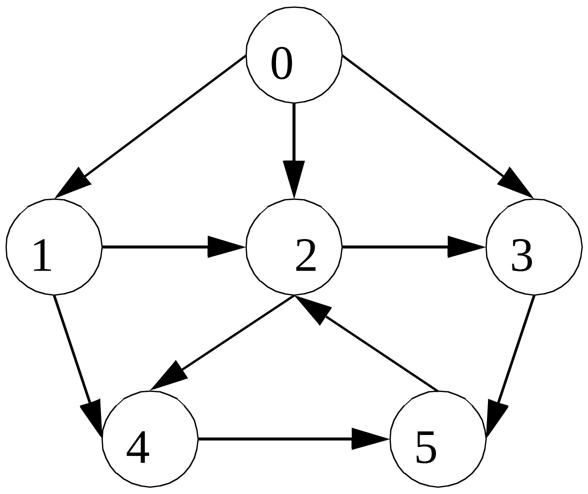

FIGURE 8.13

The edges of this graph are identified with the pairs (0,1), (0,2), (0,3), (1,2), (1,4), (2,3), (2,4), (3,5), (4,5), and (5,2), with the order in a pair indicating the **start** and **end** vertex respectively. For each vertex, there is a set of edges entering the vertex and a set of edges leaving the vertex. Each edge belongs both to the set of **out** edges for its **start** vertex and to the set of **in** edges for its **end** vertex. The following table lists the set of **in** edges and **out** edges for the six vertices in Figure 8.13.

Table 1: 8.1

VER #	0 out	0 in	1 out	1 in	2 out	2 in	3 out	3 in	4 out	4 in	5 out	5 in
	(0,1)		(1,2)	(0,1)	(2,3)	(0,2)	(3,5)	(0,3)	(4,5)	(1,4)	(5,2)	(3,5)
	(0,2)		(1,4)		(2,4)	(1,2)		(2,3)		(2,4)		(4,5)
	(0,3)					(5,2)						

8.13.2 Classes to Model a Directed Graph

The approach to model a directed graph will be similar to that for linked lists and binary trees. Base classes will handle the pointer work while the derived template classes will add a generic data member and be responsible for converting base class pointers to the actual derived class pointers stored in them. The three related base classes are **dg_vertex**, **dg_edge**, and **di_graph**. The data member parts of their definitions are given by:

```
struct dg_vertex {
    int id;
    dg_edge* outedges;
    dg_edge* inedges;
    ...
};

struct dg_edge {
    dg_vertex* from_ptr;
    dg_vertex* to_ptr;
    dg_edge* next_inedge;
    dg_edge* next_outedge;
    ...
};
struct di_graph {
    dg_vertex** npp;
    int size_ptr_array;
    int num_edges;
    int num_vertices;
    ...
};
```

8.13.3 Internal Structure of Directed Graphs

The foundation for the directed graph implementation is an array of **dg_vextex** pointers pointed to by the data member **npp** of type **dg_vertex**** in the class **di_graph**; there is a 1-1 onto map between the array of these pointers and the set of vertices of the directed graph. An alternative would be to have a linked list of **dg_vertex** pointers as the key data member of **di_graph**. This would be a more flexible approach that would allow us to dynamically change the directed graph by deleting vertices and edges. Unfortunately, this would make programming much more complex. There are many applied problems in which the directed graph initially constructed does not change during the solution of the problem; we are interested in these kinds of problems. It should be noted that this approach allows us to add as many vertices and edges as we want, but we cannot delete an arbitrary vertex. This does not imply that we cannot have a directed graph destructor deallocate edges, vertices, and the array of vertex pointers; but the array of vertex pointers is not amenable to have one of its pointers **deleted**.

.

8.13.4 Directed Graph Constructor

Let's take a look at the constructor for a directed graph:

```
di_graph::di_graph(int size) {
    size_ptr_array = size;
    npp = new dg_vertex*[size];
    num_edges = 0;
    num_vertices = 0;
};
```

The sole role of the **di_graph** constructor is to allocate memory for the array of **dg_vertex** pointers. Notice that the **di_graph** constructor adds no vertices or edges to the graph; it only sets the maximum number, **size**, of vertices for the directed graph and allocates memory for the pointer array. It should be mentioned that with a little extra functionality, **size** and the array of pointers can be increased after they are initially constructed. One would need to allocate a larger array of pointers, copy the old array of pointers into the top part of the new array of pointers, assign the starting address of the new array to **npp**, and then **delete** the memory holding the old array. Obviously, do not **delete** the **dg_vertex** pointers. To access a vertex **i**, we will access its pointer with identifier **npp[i]** or with **(vertex_ptr_array())[i]**, a **di_graph** member function returning the double pointer **npp**. Once a vertex is obtained, we need the functionality to access the **in** edges and **out** edges for that vertex. We will then have the basic functionality to inspect and work with a directed graph.

8.13.5 Constructors for the Edges and Vertices

The constructor for **dg_vertex** and **dg_edge** are:

```
dg_vertex::dg_vertex(int vid) {
    id = vid;
    inedges = 0;
    outedges = 0;
}

dg_edge::dg_edge(dg_vertex* f_ptr, dg_vertex* t_ptr) {
    from_ptr =f_ptr;
    to_ptr = t_ptr;
    next_inedge = t_ptr->inedges;
    next_out_edge = f_ptr->outedges;
}
```

Inspecting the above code bodies dictates it is time to discuss the way in which the edges are linked to each other. Each **dg_edge** contains two pointers, possibly null, to other **dg_edges**: (1) **next_inedge** is a pointer to another **dg_edge** that is entering the same vertex as the current edge and (2) **next_outedge** is a pointer to another **dg_edge** that is leaving the same vertex as the current edge. In this way, each **dg_edge** is on two linked lists and **dg_edge** itself provides storage to access the links. It was possible to use the previously developed linked list classes and to associate them with each vertex, but it turned out to be a real mess and difficult to read. There was too much base class conversion of pointers to derived template class pointers with these linked lists, and it was a syntactic nightmare. Sometimes it is better to not reuse existing code, especially if you have to explain the manner in which you awkwardly forced it to work.

The lists of edges under the row of vertices in Table 1:8.1 were not randomly entered; they were entered by adding the edges in the order: (**0**,**1**), (**0**,**2**), (**0**,**3**), (**1**,**2**), (**1**,**4**), (**2**,**3**), (**2**,**4**), (**3**,**5**), (**4**,**5**), and (**5**,**2**). The table, another visual representation of a directed graph, was built in the same order as a directed graph would be built in code by the given graph functionality. Each of the twelve lists of edges in Table 1:8:1 is a singly linked list with a head. When we add an edge to the directed graph, we must add it to the heads of both the appropriate *in* and *out* lists. As a list is being built, the current edge at the head of the list is the one at the bottom of the column. For example, after the graph in Figure 8.13 is completely built, the edge (**0**,**3**) appears at the bottom of the list associated with vertex **#0**, viewed as an **out** vertex. This edge (**0**,**3**) is at the head of the list of **out** edges for vertex **#0**. Edge (**0**,**3**) has its data member **next_outedge** pointing to the **dg_edge** representing (**0**,**2**) while (**0**,**2**) has its **next_outedge** pointing to the **dg_edge** representing (**0**,**1**). The edge (**0**,**1**) is at the tail of the list and its **next_outedge** value is a null pointer. The next question is where do we put a pointer to the head of the list. The data member **outedges** of the **dg_vertex** object representing vertex **#0** will store a pointer to the edge (**0**,**3**). It is through **outedges** of a vertex that we can iterate through the list of **dg_edges** leaving that vertex. Inspecting Table 1:8.1, we see that edge (**0**,**3**) also appears in the column for **in** edges for vertex **#3**. Notice that edge (**0**,**3**) is not at the head of this list, but rather (**2**,**3**). For this column, the **inedges** data member of the **dg_vertex** object representing vertex **#3** points to the edge (**2**,**3**), and the **next_inedge** data member of (**2**,**3**) points to edge (**0**,**3**). The **next_inedge** data member for edge (**0**,**3**) has the null pointer stored in it. It should be pointed out that the storage representing edge (**0**,**3**) on both lists is the same storage in memory. To sum it up, each edge is linked to two lists; there are pointers everywhere in this implementation. If one were to write a destructor or function to deallocate memory for edges of a directed graph, they would have to remove the edge from both lists before they deallocated the edge memory. Also, if one iterated through the **inedges** list of one vertex to free the edges on this list, one would have to search the proper **out** list for each individual edge also before deleting it. This is doable, and will be left as an exercise.

Notice that in the **dg_edge** constructor the **next_inedge** and **next_outedge** pointers are assigned the old values from the heads of the respective **inedges** and **outedges** lists. This coincides with the above explanation. The question is where do we update the new values for **inedges** and **outedges**. It is not done in either the edge or vertex constructors. It will be done by **add_edge,** a member function of the derived template class for **di_graph**.

8.13.6 Template Classes for the Directed Graph Implementation

The derived template classes for **T_dg_edge< T>** and **V_dg_vertex< V>** add the data member slots **edata** of type **T** and **vdata** of type **V** respectively. The initialization lists for the constructors simply contain calls to the base class constructors and initializations of the **edata** and **vdata** members. The types **T** and **V** may or may not be pointers. In any case, no memory management has to be specifically undertaken for **edata** or **vdata**. If they are pointers, they are the responsibility of the function that created the pointer values initially before assignment to the edges or vertices. If they are non-pointer objects, the copy initializers and destructors for **T** and **V** automatically provide the default memory management of data members of **T_dg_edge** and **V_dg_vertex** objects respectively. No explicit destructors have to be written for the edges or vertices; merely **delete** pointers to their memory in a yet to be implemented **di_graph** member function. This has to be done in an intelligent manner because of all the cross linking of edges.

The final responsibility of the data lies outside of the scope of the directed graph functionality. The application program must deal with this; if the data is pointers, then iterating through vertices and edges of a directed graph provides a handle for the deletion of data. The application program should destroy the original copies of the data, and when or before it goes out of scope, destroy the directed graph, automatically eliminating any **T** and **V** copies.

The template class, **TV_di_graph<T,V>**, adds no new data to the base class. Its purpose is to provide functionality to build the graph through two functions: (1) **void add_edge(int i, int j, T data)** and (2) **void add_vertex(int i, V data)**. Before we add an edge to the directed graph, we must add the vertices of the edge to the graph if they have not already been added. The base **di_graph** object keeps a count on the number of vertices constructed for the graph, and one can check before attempting to add a vertex if the vertex number of a desired addition is less than the current number of vertices; if it is do not add it. See the **add_vertex** function implementation for this. Let’s look at the function **void TV_di_graph<T,V>::add_edge(int i, int j, T data)**:

```
template<class T, class V> void TV_di_graph<T,V>::add_edge(int i, int j, T data) {
    if ( ( i >= number_of_vertices() ) || (j>= number_of_vertices() ) )
        error(“adding arc with vertices not present”);
    dg_vertex** dgnpp = vertex_ptr_array();
    T_dg_edge<T>* tedge = new T_dg_edge<T>(data,dgnpp[i], dgnpp[j] );
    dgnpp[i]->outedges = tedge;
    dgnpp[i]->inedges = tedge;
    num_edges++;
}
```

It is in this function where the pointers to the heads of the **inedges** list and **outedges** list are updated when an edge is added.

8.13.7 Iterator Classes for the Directed Graph Class

We need to be able to find our way from one vertex to another along a path in the directed graph. It is easy to iterate through the vertices since this implementation stores vertex pointers in an array. Simply write a **for** loop and index yourself through the array of pointers returning pointers to the vertices. You need the functionality to pick off any information that you may want from a vertex; the obvious information is a pointer to **vdata**. This is easy and left as an exercise.

Iterating through the edges is obviously a little more involved, but follows the same line of thinking as for the earlier linked list classes. We have base class iterators and the derived **template** class iterators. Let’s look at the **out_edge** iterator base class:

```
class out_edge iterator {
    di_graph* dgptr;
    dg_edge* current;
    int vertex_number;
public:
    out_edge_iterator(di_graph& dg, int id) {
        dgptr = &dg;
        vertex_number = id;
        current = ((dgptr->vertex_ptr_array())[id])->first_outedge();
        // gets head of list
    }
    dg_edge* operator() () {
        dg_edge* result = current;
        if (current) current = current->next_outedge;
        return result;
    }
};
```

As can be seen, the base constructor attaches the iterator to the directed graph and then initializes the edge pointer to the first edge pointer in the linked list of **out** edges associated with that vertex. The **operator()** returns the current edge pointer in the form of a **dg_edge** pointer. We need a pointer that includes the actual data; that will be the function of the following derived **template** class iterator:

```
template<class T, class V> class TV_out_edge_iterator : private out_edge_iterator {
public:
    TV_out_edge_iterator(TV_di_graph<T,V>& dg, int id) : out_edge_iterator(dg,
    id) {}
    T_dg_edge<T>* operator() () {
        T_dg_edge<T>* tedge;
        tedge = (T_dg_edge<T>*) (out_edge_iterator::operator() () );
        return tedge;
    }
};
```

The evaluation operator of the derived class simply converts the **dg_edge** pointer to a **T_dg_edge** pointer, allowing us to access the **edata** part of the **T_dg_edge**.

8.13.8 A Sample main() Program

Let's take a look at a program that constructs the sample directed graph of Figure 8.13. The directed graph is templatized in terms of **vertex_record** and **edge_record** pointers defined by the following classes:

```
struct vertex_record {
    int cost;
    vertex_record(int c) {
        cost = c;
    }
    void print() {
        cout << "vertex cost is " << cost;
    }
```

```
};

struct edge_record {
     int cost;
     edge_record( int c) {
          cost = c;
     }
     void print() {
          cout << “\n” << “edge cost is “ << cost;
     }
};



#pragma option -Jgx
#include “record.h”
#include “newdg.h”

void main() {
     vertex_record* pv[6];
     edge_record* pe[10];
     TV_di_graph<edge_record*, vertex_record*> DG(6);
     pv[0]  =  new vertex_record(100);
     pv[1]  =  new vertex_record(200);
     pv[2]  =  new vertex_record(300);
     pv[3]  =  new vertex_record(400);
     pv[4]  =  new vertex_record(500);
     pv[5]  =  new vertex_record(600);
     cout << “\n allocated size is “ << DG.allocated_size();
     DG.add_vertex(0, pv[0]);
     DG.add_vertex(1, pv[1]);
     DG.add_vertex(2, pv[2]);
     DG.add_vertex(3, pv[3]);
     DG.add_vertex(4, pv[4]);
     DG.add_vertex(5, pv[5]);
     pe[0]  =  new edge_record(10);
     DG.add_edge(0,1, pe[0]);
     pe[1]  =  new edge_record(40);
     DG.add_edge(0,2, pe[1]);
     pe[2] = new edge_record(30);
     DG.add_edge(0,3, pe[2]);
     pe[3] = new edge_record(50);
     DG.add_edge(1,2, pe[3] );
     pe[4] = new edge_record(60);
     DG.add_edge(1,4, pe[4]);
     pe[5] = new edge_record(20);
     DG.add_edge(2,3, pe[5]);
     pe[6] = new edge_record(70);
     DG.add_edge(2,4, pe[6]);
     pe[7] = new edge_record(80);
```

```
    DG.add_edge(3,5, pe[7]);
    pe[8] = new edge_record(90);
    DG.add_edge(4,5, pe[8]);
    pe[9] = new edge_record(100);
    DG.add_edge(5,2, pe[9]);
    cout << “\n all edges added “;
    for (int k  =  0; k  <  DG.allocated_size();  k++)  {
        TV_in_edge_iterator<edge_record*,vertex_record*> test_it_in(DG, k);
        TV_out_edge_iterator<edge_record*,vertex_record*> test_it_out(DG, k);
        T_dg_edge<edge_record*>* eptr;
        edge_record** dptr;
        while ( eptr = test_it_out() ) {
            dptr = eptr->get_data();
            int source = eptr->get_from_vertex();
            int sink = eptr->get_to_vertex();
            cout << “\nsource is “ << source << “, sink is “ << sink;
            *dptr)->print();
        }
        while ( eptr = test_it_in() )  {
            dptr = eptr->get_data();
            int source = eptr->get_from_vertex();
            int sink = eptr->get_to_vertex();
            cout << "\nsource is " << source << ", sink is  " << sink;
            (*dptr)->print();
        }
    }
}
```

In the above, neither the directed graph storage nor the edge and vertex record storage areas were deallocated. To be complete, they should be.

8.14 Finite State Machine

8.14.1 Introduction

A beautiful application for a directed graph is in the modelling of a finite state machine. A finite state machine is an algorithm that defines a set of strings of tokens and validates the membership of a string in this set. Often the strings of tokens are strings of like format of ASCII characters. Examples include the following string formats: (1) integer, (2) floating point, (3) dollars and cents, and (4) polynomial.

Figure 8.14 contains the directed graph that represents the FSM that will validate dollars and cents format. A string of characters is processed from left to right, and after processing a character we enter a new state. In each state, there is an acceptable set of characters in which the next character must belong. If and when the string is completely processed, we accept it as valid if we are in an accepting state. A double circled vertex indicates an accepting state. **0** is the starting state and **6** is an accepting state. To validate the string "**$781.65**", we will traverse the path represented by the hyphenated list: **0-1-2-2-2-4-5-6**. For further reference, consult a discrete math or automata book that discusses FSMs.

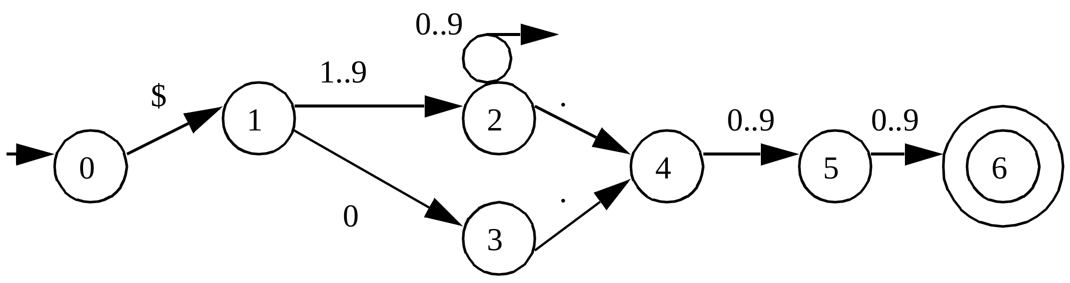

FIGURE 8.14.1

```
#ifndef DOLFSM_REC_H
#define DOLFSM_REC_H

#include <iostream.h>
#include "mstring.h"

struct fsm_vertex_record {
     int accepting;
     fsm_vertex_record(int a) { accepting = a; }
     void print() {
          cout << "\nvertex  is " <<  (accepting ? "accepting" : "non_accepting");
     }
     int accept() { return accepting; }
};

struct fsm_edge_record {
      String transition_chars;
      fsm_edge_record(char* ch_ptr) : transition_chars(ch_ptr) {};
      void print() {
            cout << "\n" << "the transition chars for this edge are "<<
            transition_chars;}
};

#endif DOLFSM_REC_H


//  testfsm.cpp
//  test file for fsm for dollars and cents
#pragma option -Jgx

#include "fsmrec.h"
```

```
#include "newdg.h"

void main() {
   TV_di_graph<fsm_edge_record*, fsm_vertex_record*> DG(7);
   fsm_vertex_record* pv[7];
   fsm_edge_record* pe[7];
   pv[0] = new fsm_vertex_record(0);
   pv[1] = new fsm_vertex_record(0);
   pv[2] = new fsm_vertex_record(0);
   pv[3] = new fsm_vertex_record(0);
   pv[4] = new fsm_vertex_record(0);
   pv[5] = new fsm_vertex_record(0);
   pv[6] = new fsm_vertex_record(1);
   cout<<"\nallocated size is "<<DG.allocated_size();
   DG.add_vertex(0, pv[0]);
   DG.add_vertex(1, pv[1]);
   DG.add_vertex(2, pv[2]);
   DG.add_vertex(3, pv[3]);
   DG.add_vertex(4, pv[4]);
   DG.add_vertex(5, pv[5]);
   DG.add_vertex(6, pv[6]);
   pe[0] = new fsm_edge_record("$");
   DG.add_edge(0, 1, pe[0]);
   pe[1] =new fsm_edge_record ("123456789");
   DG.add_edge(1, 2, pe[1]);
   pe[2] =new fsm_edge_record ("0123456789");
   DG.add_edge(2, 2, pe[2]);
   pe[3] =new fsm_edge_record (".");
   DG.add_edge(2, 4, pe[3]);
   pe[4]=new fsm_edge_record ("0");
   DG.add_edge(1, 3, pe[4]);
   pe[5]=new fsm_edge_record (".");
   DG.add_edge(3, 4, pe[5]);
   pe[6]=new fsm_edge_record ("0123456789");
   DG.add_edge(4, 5, pe[6]);
   pe[7]=new fsm_edge_record ("0123456789");
   DG.add_edge(5, 6, pe[7]);
   T_dg_edge<fsm_edge_record*> *eptr;
   for (int k = 0; k <  DG.allocated_size(); k++) {
      TV_in_edge_iterator<fsm_edge_record*,fsm_vertex_record*> test_it_in(DG, k);
      TV_out_edge_iterator<fsm_edge_record*, fsm_vertex_record*> test_it_out(DG,k);
      T_dg_edge<fsm_edge_record*> *eptr;
      fsm_edge_record **dptr;
      cout << "\nout edges for vertex " <<  k << " are : ";
      while (eptr=test_it_out()) {
         dptr=eptr->get_data();
         int source=eptr->get_from_vertex();
         int sink=eptr->get_to_vertex();
         cout<<"\nsource is "<<source<<", sink is "<<sink;
         (*dptr)->print();
```

```
            cout << "\n";
        }
        cout << "\nin edges for vertex " << k  <<"  are : ";
        while (eptr=test_it_in()) {
            dptr=eptr->get_data();
            int source=eptr->get_from_vertex();
            int sink=eptr->get_to_vertex();
            cout << "\nsource is "<< source<< ", sink is " << sink;
            (*dptr)->print();
              cout << "\n\n";
        }
    }
    //  test the string "$781.65" for validity
    String Dollars("$781.65");
    char current_char;
    int vertex_index, found, index_to_string;
    for (vertex_index = 0, index_to_string = 0, found = 0, current_char ='\0';
        current_char = Dollars.get_char(index_to_string); index_to_string++) {
        cout << current_char;
        found = 0;
        TV_out_edge_iterator<fsm_edge_record*,
            fsm_vertex_record*> search_edges(DG,vertex_index);
        while (eptr = search_edges()) {
            if (strchr(((*(eptr->get_data()))->transition_chars).string(),
                current_char)) {
                    found = 1;
                    break;
                }
        }
        if (found) vertex_index = eptr->get_to_vertex();
        else break;
    }
    if (current_char) cout << "\n" << Dollars << " is not valid";
    else {
        //  check last vertex for accepting
        if ((*(((DG.vertex_ptr_array())[vertex_index])->get_data()))->accept())
            cout << "\n" << Dollars << " is  valid";
        else   cout << "\n" << Dollars << " is not valid";
    }
}
```

8.15 C++ Files

8.15.1 Lists, Stacks, and Queues

8.15.1.1 File *lsq.h*

```
#ifndef TEMPLATE_LISTS
```

Chapter 8
Data Structures

```
#define TEMPLATE_LISTS
// this is the header file for lists,  stacks, and  queues
#include <stddef.h>
class common_list;
class list_iterator;

struct list_link {
     list_link *next;
     list_link *previous;
     list_link() { previous=next=0;}
      // friend common_list;
      // friend list_iterator;
};

template<class T> class T_list_link : public list_link {
     T data;
public:
     T* get_ptr_data() {return &data;}  // ptr to data on list returned
     T get_data() {return data;}   // copy will be made by copy-init of T
     T_list_link(const T& a) : list_link(), data(a) {  }; //  copy-initializer
      //  data will be copied from a by copy-initializer  for data type T
};

class common_list {
protected:
     list_link *head;
     list_link *tail;
     void add_internal_link(list_link* ptr, list_link** llpptr) {
          list_link* temp_ptr = *llpptr;
          *llpptr = ptr;
          ptr->next = temp_ptr;
          ptr->previous = temp_ptr->previous;
          temp_ptr->previous = ptr;
     }
public:
     friend list_iterator;
     void add_to_tail(list_link *ptr) {
          if (tail) {
               tail->next=ptr;
               ptr->previous=tail;
          } else head=ptr;
          tail=ptr;
     }
     void add_to_head(list_link *ptr) {
          if (head) {
               ptr->next=head;
               head->previous=ptr;
          } else tail=ptr;
          head=ptr;
     }
```

```
    list_link *remove_head() {
           // this disconnects the head link from the list and returns a
           // pointer to the disconnected link - it does not destroy
           // the link or its data
         list_link *result=head;
         if (head==tail)head=tail=0;
         else head=head->next;
         return result;
    }
    list_link *remove_tail() {
           // this disconnects the tail link from the list and returns a
           // pointer to the disconnected link - it does not destroy
           // the link or its data
         list_link *result=tail;
         if (head==tail)head=tail=0;
         else tail=tail->previous;
         return result;
    }
    common_list() {
         head=0;
         tail=0;
    }
    common_list(list_link *ptr) {
         head=ptr;
         tail=ptr;
    }
    int empty() { return(head==0); }
};

template<class T>  class T_list_iterator;
template<class T>  class list : private common_list {
public:
    void add_to_head(const T& info);
    void add_to_tail(const T& info);
    T get_head_data();
    T get_tail_data();
    void destroy_list(); // does not call delete on pointer data
  int empty() {return common_list::empty();}
    friend class T_list_iterator<T>;
};

class list_iterator {
    list_link *current;
    common_list *listptr;
public:
    list_iterator(common_list& cl) {
           listptr=&cl;
           current=listptr->head;
    }
    list_link *operator()() {
        list_link *result=current;
        if (current) current=current->next;
        return result;
          // if current == 0, we leave it at this value to indica
          // the end of iteration
    }
};

template<class T>  class T_list_iterator : private list_iterator {
public:
    T_list_iterator(list<T>& tl) : list_iterator(tl) {}
    T *operator()();
};

template<class T>  class stack : private common_list {
public:
    void push(const T& info);
    T pop();
    int empty() { return common_list::empty(); }
};

template<class T>  class queue : private common_list {
public:
    void add(const T& info);
    T remove();
    int empty() {return common_list::empty(); }
};

#endif
```

8.15.1.2 File *lsq.cpp*

```
// implementation file liststqu.cpp
#include "lsq.h"

template<class T> void list<T>::add_to_head(const T& info) {
    common_list::add_to_head(new T_list_link<T> (info));
}

template<class T> void list<T>::add_to_tail(const T& info) {
    common_list::add_to_tail(new T_list_link<T> (info));
}

template<class T> T list<T>::get_head_data() {
    T_list_link<T> *llink=(T_list_link<T>*)common_list::remove_head();
    T info = llink->get_data();    //  done by copy-init of class T
    delete llink;            //  this will destroy the T member
     //  of the link and free the link memory
    return info;
}

template<class T> T list<T>::get_tail_data() {
     T_list_link<T> *llink=(T_list_link<T>*)common_list::remove_tail();
     T info = llink->get_data();    //  done by copy-init of class T
     delete llink;            //  this  will  destroy the  T member
      //  of  the  link and  free the  link memory
     return info;
}

template<class T>  void list<T>::destroy_list() {
     while (!empty()) {
          T_list_link<T> *llink=(T_list_link<T>*)common_list::remove_tail();
          delete llink;
     }
}

template<class T> T* T_list_iterator<T>::operator()() {
      /* returns a ptr to actual data on link */
     list_link *ptr = list_iterator::operator()();
     if (ptr)  return ((T_list_link<T>*)ptr)->get_ptr_data();
     else return 0;
};

template<class T> void stack<T>::push(const T& info) {
     common_list::add_to_head(new T_list_link<T> (info));
}

template<class T> T stack<T>::pop() {
     T_list_link<T> *llink=(T_list_link<T>*)common_list::remove_head();
     T info = llink->get_data();    //  done by copy-init of class T
     delete llink;            //  this  will  destroy the  T member
      //  of  the  link and  free the  link memory
     return info;
}

template<class T> void queue<T>::add(const T& info) {
     common_list::add_to_tail(new T_list_link<T> (info));
}

template<class T> T queue<T> ::remove() {
     T_list_link<T> *llink=(T_list_link<T>*)common_list::remove_head();
     T info = llink->get_data();    //  done by copy-init of class T
     delete llink;            //  this  will  destroy the  T member
      //  of  the  link and  free the  link memory
     return info;
}

/* the following file will have the pragma options to instantiate the templates
     for the needed data types */
```

```
#include "prglist1.h"
```

8.15.1.3 File *prglist1.h*

```
//  this is include file prglist1.h
//  it is used to include the pragma options for liststqu.cpp

#define TREE_LINK_PTR
#define RECORD_PTR
#pragma option -Jgd

#ifdef TREE_LINK_PTR
      class tree_link;
      typedef list<tree_link*> fake_tree_link_ptr_list;
      typedef stack<tree_link*> fake_tree_link_ptr_stack;
      typedef queue<tree_link*> fake_tree_link_ptr_queue;
#endif

#ifdef RECORD_PTR
      class record;
      typedef list<record*> fake_record_ptr_list;
      typedef T_list_iterator<record*> fake_record_ptr_list_itr;
#endif
```

8.15.1.4 File *record.h*

```
#ifndef RECORD_H
#define RECORD_H
#include <iostream.h>

class record  {
      char *name;
      unsigned age;
public:
      record();
      record(char *, unsigned int);
      record(const record&);
      ~record();
      friend ostream& operator<<(ostream&, record*);
      friend int less_than(record*, record*);
   friend int compare(record*, record*);
};

#endif
```

8.15.1.5 File *record.cpp*

```
// record.cpp - sample file for data structures
#include "record.h"
#include <string.h>

record::record() {
     name=new char [1];
     name[0]='\0';
     age=0;
}

record::record(char *xname, unsigned xage) {
     name=new char [strlen(xname)+1];
     strcpy(name, xname);
     age=xage;
}

record::record(const record& original) {
     name=new char [strlen(original.name)+1];
     strcpy(name, original.name);
     age=original.age;
}

record::~record(void) {
     delete[] name;
}

int less_than(record *ptra, record *ptrb) {
     return(strcmp(ptra->name, ptrb->name)<0);
}

int compare(record* ptra, record *ptrb) {
     return(strcmp(ptra->name, ptrb->name));
}

ostream& operator<<(ostream& out, record *printout) {
     out<<"Name: "<<printout->name<<"--Age: "<<printout->age<<"\n";
     return out;
}
```

8.15.6 File *testlist.cpp*

```
#pragma option -Jgx

#include "record.h"
#include "lsq.h"

void main() {
    record *p[14];
    p[0]=new record ("AL", 28);
    p[1]=new record ("JOHN", 60);
    p[2]=new record ("MARY", 30);
    p[3]=new record ("LARRY", 21);
    p[4]=new record ("MARGARET", 16);
    p[5]=new record ("BUD", 42);
    p[6]=new record ("BILL", 18);
    p[7]=new record ("GORDON", 23);
    p[8]=new record ("DEBBIE", 40);
    p[9]=new record ("JANET", 70);
    p[10]=new record ("JOE", 17);
    p[11]=new record ("IRENE", 28);
    p[12]=new record ("ABE", 2);
    p[13]=new record ("AARON", 4);
    list<record*> record_list;
    for (int ii=0; ii<14; ii++) record_list.add_to_head(p[ii]);
    T_list_iterator<record*> record_iterator(record_list);
    record **dptr;
    while (dptr=record_iterator()) {
        cout<< *dptr;
        delete (*dptr);
    }
    record_list.destroy_list();
}
```

8.15.2 Ordered Lists

8.15.2.1File *olimp.h*

```
#ifndef ORDER_LIST_H
#define ORDER_LIST_H

#include "lsq.h"

/* the list will be ordered in ascending order from head to tail */
template<class T> class ordered_list_iterator;

template<class T> class ordered_list : private common_list {
private:
    T_list_link<T>** search_for_insertion_ptr(T* key, int&);
    T_list_link<T>** aux_search_insertion_ptr(T_list_link<T>** lpptr, T* key,
        int& match);
  void add(T_list_link<T>** lpptr, T* key);
public:
    ordered_list() : common_list() {};
    int empty() {return common_list::empty();}
    void destroy_ordered_list();
    T_list_link<T>* search_insert(T* key, int& match);
    friend class ordered_list_iterator<T>;
};

template<class T>  class ordered_list_iterator : private list_iterator {
public:
    ordered_list_iterator(ordered_list<T>& tl) : list_iterator(tl) {}
    T *operator()();
};
#endif
```

8.15.2.2 File *olimp.cpp*

```
// olimp.cpp

#include "olimp.h"

template<class T> T_list_link<T>** ordered_list<T>::search_for_insertion_ptr(
    T* key, int& match) {
    if (empty()) {
        match = 0;
        return (T_list_link<T>**)0;
    }
    if (compare(*key, ((T_list_link<T>*)head)->get_data()) < 0 ) {
        match = 0;
        return (T_list_link<T>**)(&head);
    }
    else {
        if (compare(((T_list_link<T>*)head)->get_data(),*key)<0) {
            match = 0;
            return aux_search_insertion_ptr(((T_list_link<T>**)&(head->next)),
                key, match);
        }
        else {
            match = 1;
            return (T_list_link<T>**)(&head);
        }
    }
}

template<class T> T_list_link<T>** ordered_list<T>::aux_search_insertion_ptr(
    T_list_link<T>** lpptr, T* key, int& match) {
    /* check for null contents of lpptr */
    if (*lpptr == 0) {
        match = 0;
        return lpptr;  // should insert at this address
    }
    else {
        if (compare((*lpptr)->get_data(),*key) < 0) {
            match = 0;
            return aux_search_insertion_ptr(((T_list_link<T>**)&((*lpptr)->next)),
                key, match);
        }
        else  {
            if (compare(*key,(*lpptr)->get_data())<0) {
                match = 0;
                return (T_list_link<T>**)(&((*lpptr)->previous->next));
            } else {
                match = 1;
                return (T_list_link<T>**)(&((*lpptr)->previous->next));
            }
        }
    }
}

template<class T> void ordered_list<T>::add(T_list_link<T>** lpptr, T* key) {
 if (empty()) add_to_head(new T_list_link<T>(*key));
            // this checks for lpptr == 0
    else { // check for head insertion
        if (lpptr == (T_list_link<T>**)&head)
            add_to_head(new T_list_link<T>(*key));
        else {
            if (lpptr == (T_list_link<T>**)&(tail->next))
                add_to_tail(new T_list_link<T>(*key));
            else add_internal_link(new T_list_link<T>(*key),(list_link**) lpptr);
        }
    }
}

template<class T> T_list_link<T>* ordered_list<T>::search_insert(T* key,
    int& match) {
    T_list_link<T>** insert_location;
    insert_location = search_for_insertion_ptr(key, match);
    if (insert_location == 0) {
        add_to_head(new T_list_link<T>(*key));
        insert_location = (T_list_link<T>**)(&head);
    }
    else if (match == 0) add(insert_location, key);
    return *insert_location;
}

template<class T> T* ordered_list_iterator<T>::operator()() {
     /* returns a ptr to actual data on link */
    list_link *ptr = list_iterator::operator()();
    if (ptr)  return ((T_list_link<T>*)ptr)->get_ptr_data();
    else  return 0;
};
```

```
template<class T>  void ordered_list<T>::destroy_ordered_list() {
    while (!empty()) {
        T_list_link<T> *llink=(T_list_link<T>*)remove_tail();
        delete llink;
    }
}

#include "prgorli2.h"
```

8.15.2.3 File *prgorli2.h*

```
// file "prgorli2.h"
#define RECORD_PTR
// #define RECORD_MODITEM_PTR
#pragma option -Jgd
#ifdef RECORD_PTR
    #include "record.h"
    typedef ordered_list<record*> fake_record_ptr_ordered_list;
    typedef ordered_list_iterator<record*> fake_record_ptr_ordered_list_itr;
#endif

#ifdef RECORD_MODITEM_PTR
    #include "recmoitm.h"
    typedef ordered_list<record_moditem*>  fake_rec_moitm_ptr_ordered_list;
    typedef ordered_list_iterator<record_moditem*> fake_rec_moitm_ptr_ordered_list_itr;
#endif
```

8.15.2.4 File *orlitest.cpp*

```
// this tests an ordered list implementation

#include "record.h"
#include "lsq.h"
#include "olimp.h"

#pragma option -Jgx

void main() {
    record *p[14];
    p[0]=new record ("AL", 28);
    p[1]=new record ("JOHN", 60);
    p[2]=new record ("MARY", 30);
    p[3]=new record ("LARRY", 21);
    p[4]=new record ("MARGARET", 16);
    p[5]=new record ("BUD", 42);
    p[6]=new record ("BILL", 18);
    p[7]=new record ("GORDON", 23);
    p[8]=new record ("DEBBIE", 40);
    p[9]=new record ("JANET", 70);
    p[10]=new record ("JOE", 17);
    p[11]=new record ("IRENE", 28);
    p[12]=new record ("ABE", 2);
    p[13]=new record ("AARON", 4);
    record **dptr;
    int match;
    match = 0;
    ordered_list<record*> or_record_list;
    for (int ii= 0; ii < 14; ii++)
        or_record_list.search_insert(&(p[ii]), match);
    ordered_list_iterator<record*> or_record_iterator(or_record_list);
    while (dptr=or_record_iterator()) {
        cout << *dptr;
        delete (*dptr);     // destroy the data
    }
    or_record_list.destroy_ordered_list();  // destroy the list links
}
```

8.15.3 Trees

8.15.3.1 File *treeimp.h*

```
// this is the header file for a tree class and its iterators
#ifndef TREE_ITERATOR_H
#define TREE_ITERATOR_H

#include "lsq.h"
#include <iostream.h>

class BFS_base;
class DFS_base;
class inorder_base;

struct tree_link  {
    tree_link *leftptr;
    tree_link *rightptr;
    tree_link(tree_link *l, tree_link *r) {
        leftptr=l;
        rightptr=r;
    }
    tree_link() {leftptr=rightptr=0;}
};

template<class T>  class T_tree;
template<class T>  class BFS_T;
```

```
template<class T>  class DFS_T;
template<class T>  class inorder_T;

template<class T>  class T_tree_link : public tree_link {
     T data;
public:
     T_tree_link();
     T_tree_link(T_tree_link<T>*, T_tree_link<T>*, const T&);
     T_tree_link(const T& a);
     T* get_ptr_data() { return &data; }
     friend class T_tree<T>;
     friend class BFS_T<T> ;
     friend class DFS_T<T> ;
     friend class inorder_T<T> ;
};

class base_tree  {
protected:
     tree_link *root;
public:
     base_tree(tree_link* ptr) { root = ptr; }
     friend BFS_base;
     friend DFS_base;
     friend inorder_base;
};

template<class T>  class T_tree : private base_tree {
private:
     void destroy_subtree(T_tree_link<T>* ptr);
     T_tree_link<T>** aux_search_insertion_ptr(T_tree_link<T>** tpptr, T *key);
     void add_to_tree(T_tree_link<T>**, T*);
     T_tree_link<T>** search_for_insertion_ptr(T *key);
public:
     T_tree(T_tree_link<T>* ptr = 0) : base_tree(ptr) { }
     int empty() { return (!root);}
     void insert(T *pkey) {
                    add_to_tree((T_tree_link<T>**)(&root), pkey);
        }
     void destroy_T_tree(){destroy_subtree((T_tree_link<T>*)root);}
     T_tree_link<T>* search_insert(T* key, int& match);
     friend class BFS_T<T> ;
     friend class DFS_T<T> ;
     friend class inorder_T<T>;
};

class BFS_base  {
private:
     tree_link *current;
     queue<tree_link*> bfs_queue;
public:
     BFS_base(base_tree&);
     tree_link *operator()();
};

template<class T>  class BFS_T : private BFS_base {
public:
     BFS_T(T_tree<T>& tr) : BFS_base(tr) {}
     T *operator()();
};

class DFS_base  {
private:
     tree_link *current;
     stack<tree_link*> dfs_stack;
public : DFS_base(base_tree&);
     tree_link *operator()();
};

template<class T>  class DFS_T : private DFS_base {
public:
     DFS_T(T_tree<T>& tr) : DFS_base(tr) {}
     T *operator()();
};

class inorder_base  {
private:
     tree_link *current;
     stack<tree_link*> io_stack;
public:
     inorder_base(base_tree&);
     tree_link *operator()();
};

template<class T>  class inorder_T : private inorder_base {
public:
     inorder_T(T_tree<T>& tr) : inorder_base(tr) {  }
   T *operator()();
};

#endif
```

8.15.3.2 File *treeimp.cpp*

```
// this is the implementation file for iterators on a binary tree
#include <string.h>
#include "treeimp.h"

template<class T> T_tree_link<T> ::T_tree_link() : tree_link(), data() {}
```

```
template<class T> T_tree_link<T> ::T_tree_link(T_tree_link<T> *leftchild,
    T_tree_link<T> *rightchild, const T& r) : tree_link(leftchild,
    rightchild), data(r) {
}

BFS_base::BFS_base(base_tree& tr) {
    bfs_queue.add(tr.root);
}

DFS_base::DFS_base(base_tree& tr) {
    current=tr.root;
}

inorder_base::inorder_base(base_tree& tr){
    current=tr.root;
}

tree_link *BFS_base::operator()() {
    if (bfs_queue.empty())
        return 0;
    else {
        current=bfs_queue.remove();
        if (current->leftptr) bfs_queue.add(current->leftptr);
        if (current->rightptr) bfs_queue.add(current->rightptr);
        return current;
    }
}

tree_link *DFS_base::operator()() {
    tree_link *result;
    if (current) {
        result=current;
        if (current->rightptr) dfs_stack.push(current->rightptr);
        if (current->leftptr) current=current->leftptr;
        else {
            if (dfs_stack.empty()) current=0;
            else  current=dfs_stack.pop();
        }
        return result;
    } else return 0;
}

tree_link *inorder_base::operator()() {
    tree_link *result;
    while (current) {
        io_stack.push(current);
        current=current->leftptr;
    }
    if (io_stack.empty()) return 0;
    else {
          current=io_stack.pop();
          result=current;
          current=current->rightptr;
          return result;
     }
}

template<class T>  T_tree_link<T>::T_tree_link(const T& a) : tree_link(),
     data(a) {
}

template<class T> T *DFS_T<T>::operator()() {
     T_tree_link<T> *tlink;
     tlink=(T_tree_link<T>*)(DFS_base::operator()());
     if (tlink) return &(tlink->data);
     else return 0;
}

template<class T> T *BFS_T<T>::operator()() {
     T_tree_link<T> *tlink;
     tlink=(T_tree_link<T>*)(BFS_base::operator()());
     if (tlink) return &(tlink->data);
     else  return 0;
}

template<class T> T *inorder_T<T>::operator()() {
     T_tree_link<T> *tlink;
     tlink=(T_tree_link<T>*)(inorder_base::operator()());
     if (tlink) return &(tlink->data);
     else  return 0;
}

template<class T> void T_tree<T>::add_to_tree(T_tree_link<T>** ptr, T *key) {
     int decide;
     if (*ptr == 0)  *ptr = new T_tree_link<T>(0, 0, *key);
     else  {
          if ((decide = compare(*key, (*ptr)->data)) < 0)
                add_to_tree((T_tree_link<T>**)&((*ptr)->leftptr) , key);
          else if (decide > 0)
                add_to_tree((T_tree_link<T>**)&((*ptr)->rightptr), key);
     }
}


template<class T> void T_tree<T>::destroy_subtree(T_tree_link<T>* ptr) {
     if (ptr->leftptr) destroy_subtree((T_tree_link<T> *)(ptr->leftptr));
     if (ptr->rightptr) destroy_subtree((T_tree_link<T> *)(ptr->rightptr));
     delete ptr;
}
```

```
template<class T> T_tree_link<T>** T_tree<T>::search_for_insertion_ptr(T *key) {
     int decide;
     if ((T_tree_link<T>*)root == 0)  return (T_tree_link<T>**)0;
     if ((decide = compare(*key, ((T_tree_link<T>*)root)->data)) <0)
          return aux_search_insertion_ptr(((T_tree_link<T>**)&(root->leftptr)), key);
     else if (decide > 0) return aux_search_insertion_ptr(
                 ((T_tree_link<T>**)&(root->rightptr)), key);
            else  return ((T_tree_link<T>**)&root);
      // if we return a null pointer, then the root node is a match  or null
}
template<class T> T_tree_link<T>** T_tree<T>::aux_search_insertion_ptr(
     T_tree_link<T>** tpptr, T *key) {
     int decide;
     if (*tpptr == 0)  return tpptr;
     if ((decide = compare(*key, (*tpptr)->data))<0)
          return aux_search_insertion_ptr(((T_tree_link<T>**)&((*tpptr)->leftptr)), key);
     else if (decide > 0) return aux_search_insertion_ptr(
                 ((T_tree_link<T>**)&((*tpptr)->rightptr)), key);
            else  return tpptr;
}

template<class T> T_tree_link<T>* T_tree<T>::search_insert(T* key,
     int& match) {
     T_tree_link<T>** insert_location;
     insert_location = search_for_insertion_ptr(key);
     if (insert_location == 0) { // this signals no root node on tree yet
          // create the root node and assign its address to "root"
          insert(key);
          match = 0;
          insert_location = (T_tree_link<T>**)(&root);
     }
     else {
          if (*insert_location == 0)  /* add it */ {
                add_to_tree(insert_location, key);
                match = 0;
          }
          else { /* here there is a collision */
                match = 1;
          }
     }
     return *insert_location;
}

/* the following file will have the pragma options to instantiate the templates
     for the needed data types */

#include "prgtree2.h"
```

8.15.3.3 File *prgtree2.h*

```
// file prgtree2.h

#define RECORD_PTR
// #define RECORD_MAJITEM_PTR
// #define RECORD_MODITEM_PTR
#pragma option -Jgd

#ifdef RECORD_PTR
    #include "record.h"
    typedef T_tree_link<record*> fake_record_ptr_T_tree_link;
    typedef T_tree<record*> fake_record_ptr_T_tree;
    typedef inorder_T<record*> fake_record_ptr_inorder;
    typedef BFS_T<record*> fake_record_ptr_BFS;
    typedef DFS_T<record*> fake_record_ptr_DFS;

#endif

#ifdef RECORD_MAJITEM_PTR
    #include "recmaitm.h"
    typedef T_tree_link<record_majitem*> fake_rec_maitm_ptr_T_tree_link;
    typedef T_tree<record_majitem*> fake_rec_maitm_ptr_T_tree;
    typedef inorder_T<record_majitem*> fake_rec_maitm_ptr_inorder;
    typedef BFS_T<record_majitem*> fake_rec_maitm_ptr_BFS;
    typedef DFS_T<record_majitem*> fake_rec_maitm_ptr_DFS;

#endif

#ifdef RECORD_MODITEM_PTR
    #include "recmoitm.h"
    typedef T_tree_link<record_moditem*> fake_rec_moitm_ptr_T_tree_link;
    typedef T_tree<record_moditem*> fake_rec_moitm_ptr_T_tree;
    typedef inorder_T<record_moditem*> fake_rec_moitm_ptr_inorder;
    typedef BFS_T<record_moditem*> fake_rec_moitm_ptr_BFS;
    typedef DFS_T<record_moditem*> fake_rec_moitm_ptr_DFS;

#endif
```

8.15.3.4 File *testtree.cpp*

```
// testtree.cpp
#pragma option -Jgx
#include "record.h"
#include "lsq.h"
#include "treeimp.h"

void main() {
    record *p[14];
```

```
T_tree<record*> test_tree;   // this assigns 0  to  root
record **ttptr;
ttptr=&(p[0]=new record ("AL", 28));
test_tree.insert(ttptr);
ttptr=&(p[1]=new record ("JOHN", 60));
test_tree.insert(ttptr);
ttptr=&(p[2]=new record ("MARY", 30));
test_tree.insert(ttptr);
ttptr=&(p[3]=new record ("LARRY", 21));
test_tree.insert(ttptr);
ttptr=&(p[4]=new record ("MARGARET", 16));
test_tree.insert(ttptr);
ttptr=&(p[5]=new record ("BUD", 42));
test_tree.insert(ttptr);
ttptr=&(p[6]=new record ("BILL", 18));
test_tree.insert(ttptr);
ttptr=&(p[7]=new record ("GORDON", 23));
test_tree.insert(ttptr);
ttptr=&(p[8]=new record ("DEBBIE", 40));
test_tree.insert(ttptr);
ttptr=&(p[9]=new record ("JANET", 70));
test_tree.insert(ttptr);
ttptr=&(p[10]=new record ("JOE", 17));
test_tree.insert(ttptr);
ttptr=&(p[11]=new record ("IRENE", 28));
test_tree.insert(ttptr);
ttptr=&(p[12]=new record ("ABE", 2));
test_tree.insert(ttptr);
ttptr=&(p[13]=new record ("AARON", 4));
test_tree.insert(ttptr);
int match;
for (int k = 0; k < 14; k++) {
    match = 0;
    test_tree.search_insert(&p[k], match);
}
BFS_T<record*> breadth_first(test_tree);
DFS_T<record*> depth_first(test_tree);
inorder_T<record*> in_order(test_tree);
record **ptr_tree_data;
// PERFORM BREADTH-FIRST TRAVERSAL
cout << "\nBFS traversal \n";
while (ptr_tree_data = breadth_first()) cout << *ptr_tree_data;
cout << "success of BFS\n";
// PERFORM DEPTH-FIRST TRAVERSAL
cout<<"\nDFS traversal \n";
while (ptr_tree_data = depth_first()) cout << *ptr_tree_data;
cout << "success  for depth\n";
//  PERFORM INORDER TRAVERSAL
cout<<"\ninorder traversal \n";
while (ptr_tree_data = in_order())  cout<< *ptr_tree_data;
```

```
     cout <<"successful term  of inorder\n";
     //  iterate inorder through the ptr_tree_data destroying the pointer data
     inorder_T<record*> link_destroyer(test_tree);
     while (ptr_tree_data = link_destroyer()) delete (*ptr_tree_data);
     cout << "the pointer data has  been destroyed\n";
     // now destroy the tree - the links
     test_tree.destroy_T_tree();
     cout << "the tree links have been destroyed\n";
}
```

8.15.4 Directed Graphs

8.15.4.1 File *dgrecord.h*

```
#ifndef DGRECORD_H
#define DGRECORD_H

#include <iostream.h>
struct vertex_record  {
    int cost;
    vertex_record(int c) {
        cost=c; }
    void print() {
        cout<<"vertex cost is "<<cost; }
};

struct edge_record  {
    int cost;
    edge_record(int c) {
        cost=c; }
    void print() {
        cout<<"\n"<<"edge cost is "<<cost; }
};

#endif DGRECORD_H
```

8.15.4.2 File *newdg.h*

```
//  newdg.h  header file for directed graphs - aug 27, 1993
#ifndef NEWDG_H
#define NEWDG_H

#include <iostream.h>
#include <stdlib.h>
struct di_graph;
struct dg_edge;

struct dg_vertex  {
     int id;
     dg_edge *inedges;
     dg_edge *outedges;
     dg_edge *first_inedge() { return inedges; }
     dg_edge *first_outedge() { return outedges; }
     dg_vertex(int vid) {
          id=vid;
          inedges=0;
          outedges=0;
     }
};

struct dg_edge  {
     dg_vertex *from_ptr;
     dg_vertex *to_ptr;
     dg_edge *next_inedge;
     dg_edge *next_outedge;
     dg_edge(dg_vertex *f_ptr, dg_vertex *t_ptr) {
          from_ptr=f_ptr;
          to_ptr=t_ptr;
          next_inedge=t_ptr->inedges;
          next_outedge=f_ptr->outedges;
     }
};

struct di_graph  {
     dg_vertex **npp;
     int size_ptr_array;
     int num_edges;
     int num_vertices;
     di_graph(int size) {
          size_ptr_array=size;
          npp=new dg_vertex *[size];
          num_edges=0;
          num_vertices=0;
     }
     dg_vertex **vertex_ptr_array() { return npp; }
     int number_of_edges() { return num_edges; }
     int number_of_vertices() { return num_vertices; }
     int allocated_size() { return size_ptr_array; }
     void error(char *message) {
            cerr<<"\n"<<message<<"\n";
            exit(1);
     }
};

template<class V>  struct V_dg_vertex : public dg_vertex {
    V vdata;
    V_dg_vertex(const V& a, int vid) : dg_vertex(vid), vdata(a) {  }
      V *get_data() { return &vdata; }
};

template<class T>  struct T_dg_edge : public dg_edge {
    T edata;
    T_dg_edge(const T& a, dg_vertex *fp, dg_vertex *tp) : dg_edge(fp, tp),
        edata(a) {  }
    T *get_data() { return &edata; }
    int get_from_vertex() { return from_ptr->id; }
    int get_to_vertex() { return to_ptr->id; }
};

template<class T, class V>  class TV_di_graph : public di_graph {
public:
     TV_di_graph(int size) : di_graph(size) {  }
     void add_edge(int i, int j, T data);
     void add_vertex(int i, V data);
     V_dg_vertex<V>** vertex_ptr_array() {
          return (V_dg_vertex<V>**)(di_graph::vertex_ptr_array()); }
};

class in_edge_iterator  {
    di_graph *dgptr;
    dg_edge *current;
      int vertex_number;
public:
    in_edge_iterator(di_graph& dg, int id) {
        dgptr=&dg;
        vertex_number=id;
        current=((dgptr->vertex_ptr_array())[id])->first_inedge();
    }
    dg_edge *operator()() {
        dg_edge *result=current;
        if (current)
        current=current->next_inedge;
        return result;
    }
};

class out_edge_iterator  {
    di_graph *dgptr;
    dg_edge *current;
    int vertex_number;
public:
    out_edge_iterator(di_graph& dg, int id) {
        dgptr=&dg;
        vertex_number=id;
        current=((dgptr->vertex_ptr_array())[id])->first_outedge();
    }
    dg_edge *operator()() {
        dg_edge *result=current;
        if (current)
        current=current->next_outedge;
        return result;
    }
};


template<class T, class V>
class TV_in_edge_iterator : private in_edge_iterator {
public:
    TV_in_edge_iterator(TV_di_graph<T, V>& dg,
                        int id) : in_edge_iterator(dg, id) {  }
    T_dg_edge<T> *operator()() {
        T_dg_edge<T> *tedge;
        tedge=(T_dg_edge<T>*)(in_edge_iterator::operator()());
        return tedge;
    }
};

template<class T, class V>
class TV_out_edge_iterator : private out_edge_iterator {
public:
    TV_out_edge_iterator(TV_di_graph<T, V>& dg,
                         int id) : out_edge_iterator(dg, id) {  }
    T_dg_edge<T> *operator()() {
        T_dg_edge<T> *tedge;
        tedge=(T_dg_edge<T>*)(out_edge_iterator::operator()());
        return tedge;
    }
};

#endif NEWWDG_H
```

8.15.4.3 File *newdg.cpp*

```
// implementation file for directed graphs

#include "newdg.h"
template<class T, class V> void TV_di_graph<T, V>::add_vertex(int i, V data) {
     if (i<number_of_vertices()) error("vertex already exists");
     if (i>=allocated_size()) error("not enough vertex memory allocated");
     (vertex_ptr_array())[i] = new V_dg_vertex <V>(data, i);
     num_vertices++;
}

template<class T, class V>  void TV_di_graph<T, V>::add_edge(int i,
     int j, T data) {
     if ((i>=number_of_vertices())||(j>=number_of_vertices()))
         error("adding arc with vertices not present");
     dg_vertex** dgnpp= di_graph::vertex_ptr_array();
     T_dg_edge<T> *tedge=new T_dg_edge <T>(data, dgnpp[i], dgnpp[j]);
     dgnpp[i]->outedges=tedge;
     dgnpp[j]->inedges=tedge;
     num_edges++;
}

/* the following file will have the pragma options to instantiate the templates
     for the needed data types */

#include "prgdg2.h"
```

8.15.4.4 File *prgdg2.h*

```
// this is the pragma option file prgdg2.h for directed graphs

#define EDGE_RECORD_PTR
#define VERTEX_RECORD_PTR

#define FSM_EDGE_RECORD_PTR
#define FSM_VERTEX_RECORD_PTR

#pragma option -Jgd

#if defined(EDGE_RECORD_PTR) && defined(VERTEX_RECORD_PTR)
    struct vertex_record;
    struct edge_record;
    typedef V_dg_vertex<vertex_record*> fakevdgvertex;
    typedef T_dg_edge<edge_record*> faketdgedge;
    typedef TV_di_graph<edge_record*, vertex_record*> faketvdgraph;
    typedef TV_out_edge_iterator<edge_record*, vertex_record*> faketvinedit;
    typedef TV_in_edge_iterator<edge_record*, vertex_record*> faketvoutedit;
#endif


#if defined(FSM_EDGE_RECORD_PTR) && defined(FSM_VERTEX_RECORD_PTR)
    struct fsm_vertex_record;
    struct fsm_edge_record;
    typedef V_dg_vertex<fsm_vertex_record*> fakefsmvdgvertex;
    typedef T_dg_edge<fsm_edge_record*> fakefsmtdgedge;
    typedef TV_di_graph<fsm_edge_record*, fsm_vertex_record*> fakefsmtvdgraph;
    typedef TV_out_edge_iterator<fsm_edge_record*, fsm_vertex_record*>
        fakefsmtvinedit;
    typedef TV_in_edge_iterator<fsm_edge_record*, fsm_vertex_record*>
        fakefsmtvoutedit;
#endif
```

8.15.4.5 File *testdg.cpp*

```
#pragma option -Jgx

#include "dgrecord.h"
#include "newdg.h"

void main() {
    vertex_record* pv[6];
    edge_record* pe[10];
    TV_di_graph<edge_record*, vertex_record*> DG(6);
    pv[0] = new vertex_record (100);
    pv[1] = new vertex_record (200);
    pv[2] = new vertex_record (300);
    pv[3] = new vertex_record (400);
    pv[4] = new vertex_record (500);
      pv[5] = new vertex_record (600);
    cout<<"\n allocated size is "<<DG.allocated_size();
    DG.add_vertex(0, pv[0]);
      DG.add_vertex(1, pv[1]);
      DG.add_vertex(2, pv[2]);
      DG.add_vertex(3, pv[3]);
      DG.add_vertex(4, pv[4]);
      DG.add_vertex(5, pv[5]);
      pe[0] = new edge_record(10);
      DG.add_edge(0, 1, pe[0]);
      pe[1] = new edge_record (40);
      DG.add_edge(0, 2, pe[1]);
      pe[2] = new edge_record (30);
      DG.add_edge(0, 3, pe[2]);
      pe[3] = new edge_record (50);
      DG.add_edge(1, 2, pe[3]);
      pe[4] = new edge_record (60);
      DG.add_edge(1, 4, pe[4]);
      pe[5] = new edge_record (20);
      DG.add_edge(2, 3, pe[5]);
      pe[6] = new edge_record (70);
      DG.add_edge(2, 4, pe[6]);
      pe[7] = new edge_record (80);
      DG.add_edge(3, 5, pe[7]);
      pe[8] = new edge_record (90);
      DG.add_edge(4, 5, pe[8]);
      pe[9] = new edge_record (100);
      DG.add_edge(5, 2, pe[9]);
      cout<<"\n all edges added ";
      for (int k = 0; k <  DG.allocated_size(); k++) {
          TV_in_edge_iterator<edge_record*, vertex_record*> test_it_in(DG, k);
          TV_out_edge_iterator<edge_record*, vertex_record*> test_it_out(DG, k);
          T_dg_edge<edge_record*> *eptr;
          edge_record **dptr;
          while (eptr=test_it_out()) {
            dptr=eptr->get_data();
            int source=eptr->get_from_vertex();
            int sink=eptr->get_to_vertex();
            cout<<"\nsource is "<<source<<", sink is "<<sink;
            (*dptr)->print();
          }
          while (eptr=test_it_in()) {
               dptr=eptr->get_data();
               int source=eptr->get_from_vertex();
               int sink=eptr->get_to_vertex();
               cout << "\nsource is "<< source<< ", sink is " << sink;
               (*dptr)->print();
          }
     }
}
```

8.15.5 Finite State Machine

8.15.5.1 File *fsmrec.h*

```
#ifndef DOLFSM_REC_H
#define DOLFSM_REC_H

#include <iostream.h>
#include "mstring.h"

struct fsm_vertex_record {
     int accepting;
     fsm_vertex_record(int a) { accepting = a; }
     void print() {
          cout << "\nvertex  is " <<  (accepting ? "accepting" : "non_accepting");
     }
     int accept() { return accepting; }
};

struct fsm_edge_record {
      String transition_chars;
      fsm_edge_record(char* ch_ptr) : transition_chars(ch_ptr) {};
      void print() {
            cout << "\n" << "the transition chars for this edge are "<<
            transition_chars;}
};

#endif DOLFSM_REC_H
```

8.15.5.2 File *testfsm.cpp*

```
//  testfsm.cpp
//  test file for fsm for dollars and cents
#pragma option -Jgx

#include "fsmrec.h"
#include "newdg.h"

void main() {
   TV_di_graph<fsm_edge_record*, fsm_vertex_record*> DG(7);
   fsm_vertex_record* pv[7];
   fsm_edge_record* pe[7];
   pv[0] = new fsm_vertex_record(0);
   pv[1] = new fsm_vertex_record(0);
   pv[2] = new fsm_vertex_record(0);
   pv[3] = new fsm_vertex_record(0);
   pv[4] = new fsm_vertex_record(0);
   pv[5] = new fsm_vertex_record(0);
   pv[6] = new fsm_vertex_record(1);
   cout<<"\nallocated size is "<<DG.allocated_size();
   DG.add_vertex(0, pv[0]);
   DG.add_vertex(1, pv[1]);
   DG.add_vertex(2, pv[2]);
   DG.add_vertex(3, pv[3]);
   DG.add_vertex(4, pv[4]);
   DG.add_vertex(5, pv[5]);
   DG.add_vertex(6, pv[6]);
   pe[0] = new fsm_edge_record("$");
   DG.add_edge(0, 1, pe[0]);
   pe[1] =new fsm_edge_record ("123456789");
   DG.add_edge(1, 2, pe[1]);
   pe[2] =new fsm_edge_record ("0123456789");
   DG.add_edge(2, 2, pe[2]);
   pe[3] =new fsm_edge_record (".");
   DG.add_edge(2, 4, pe[3]);
   pe[4]=new fsm_edge_record ("0");
   DG.add_edge(1, 3, pe[4]);
   pe[5]=new fsm_edge_record (".");
   DG.add_edge(3, 4, pe[5]);
   pe[6]=new fsm_edge_record ("0123456789");
   DG.add_edge(4, 5, pe[6]);
   pe[7]=new fsm_edge_record ("0123456789");
   DG.add_edge(5, 6, pe[7]);
   T_dg_edge<fsm_edge_record*> *eptr;
   for (int k = 0; k <  DG.allocated_size(); k++) {
      TV_in_edge_iterator<fsm_edge_record*, fsm_vertex_record*> test_it_in(DG, k);
      TV_out_edge_iterator<fsm_edge_record*, fsm_vertex_record*> test_it_out(DG,k);
      T_dg_edge<fsm_edge_record*> *eptr;
      fsm_edge_record **dptr;
      cout << "\nout edges for vertex " <<  k << " are : ";
      while (eptr=test_it_out()) {
         dptr=eptr->get_data();
         int source=eptr->get_from_vertex();
         int sink=eptr->get_to_vertex();
         cout<<"\nsource is "<<source<<", sink is "<<sink;
         (*dptr)->print();
         cout << "\n";
      }
      cout << "\nin edges for vertex " << k  <<"  are : ";
      while (eptr=test_it_in()) {
         dptr=eptr->get_data();
         int source=eptr->get_from_vertex();
         int sink=eptr->get_to_vertex();
         cout << "\nsource is "<< source<< ", sink is " << sink;
         (*dptr)->print();
           cout << "\n\n";
      }
   }
   //  test the string "$781.65" for validity
   String Dollars("$781.65");
   char current_char;
   int vertex_index, found, index_to_string;
   for (vertex_index = 0, index_to_string = 0, found = 0, current_char ='\0';
      current_char = Dollars.get_char(index_to_string); index_to_string++) {
      cout << current_char;
      found = 0;
      TV_out_edge_iterator<fsm_edge_record*,
         fsm_vertex_record*> search_edges(DG,vertex_index);
      while (eptr = search_edges()) {
         if (strchr(((*(eptr->get_data()))->transition_chars).string(),
            current_char)) {
               found = 1;
               break;
            }
      }
      if (found) vertex_index = eptr->get_to_vertex();
      else break;
   }
   if (current_char) cout << "\n" << Dollars << " is not valid";
   else {
      //  check last vertex for accepting
      if ((*(((DG.vertex_ptr_array())[vertex_index])->get_data()))->accept())
         cout << "\n" << Dollars << " is  valid";
      else   cout << "\n" << Dollars << " is not valid";
   }
}
```

Chapter 8
Data Structures

EXERCISES

1. Write a function to deallocate the memory of the links of a list. It should have prototype: **template<class T> void list<T>::destroy_list()**. Do not assume the data type **T** is a pointer and call **delete T** on the node data.

2. How do we handle the destruction of all the memory associated with the **T** data on a list if **T** is a pointer type?

3. Referring to the question in Exercise 2, what about the **T** data if **T** is a class (not a pointer) with a destructor? What if **T** is a type with no deep memory such as a long or a struct with no pointer data members?

4. Postorder traversal of a binary tree is defined recursively by traversing the left subtree, then the right subtree, and finally the root link of the subtree. Use this concept to write a member function: **template<class T> void T_tree::destroy_T_tree()** that will deallocate the link memory of a **T_tree**. You should study the member function **insert** and its recursive helper function **add**.

5. We return now to the problem of determining the inverse of an integer **a** relatively prime to an integer **m**. This problem will be solved by using a stack data structure with the Euclidean algorithm.

6. Use the stack class to calculate the inverse **p** of an integer q relatively prime to an integer **m**. **q** is defined to be relatively prime to **m** if their greatest common divisor is **1**. The method to determine the greatest common divisor of two positive integers and calculate the inverse (if it exists) of the smaller is called the Euclidean algorithm. The following example will illustrate the technique. From this example, determine the data structure needed to instantiate the template stack class. You can determine the formulas iteration formulas from the example.

   Calculate the g.c.d. of 331 and 207:

```
    331  =  207  *   1  +  124
    207  =  124  *   1  +   83
    124  =   83  *   1  +   41
     83  =   41     *     2           +1
     41  =     1  *  4  1     +0
```

   The algorithm iteratively takes the divisor and remainder of one line and makes it the dividend and divisor respectively of the next line. It terminates on a zero remainder. The smallest positive remainder (second line from the bottom) will be the g.c.d. To determine the inverse of 207 with respect to 331, we take the equation **83 = 41 * 2 + 1** and iteratively replace the remainder with its equation, working toward the top equation. Each equation will be stored on the stack, and popped as we produce the following set of equations.

$$
\begin{aligned}
1 &= (-2) * 41 + 1 * 83 \\
1 &= (-2) * (124 + (-1)83) + 1 * 83 \\
1 &= 3 * 83 + (-2)*(124) \\
1 &= 3 * (207 + (-1)124) + (-2)*(124) \\
1 &= (-5) * 124 + 3* 207 \\
1 &= (-5) * (331 + (-1)207) + 3*207 \\
1 &= 8 * 207 + (-5)*331
\end{aligned}
$$

This says 8 is the inverse of 207 mod 331. If the inverse were negative, we simply would add the original dividend to get the positive equivalent. Keep in mind there is a hidden update formula in the above set of equations, and