Implementing Collections with Linked Lists

- So far, we have used array-based implementations of the Stack, Queue, and SeqList classes.
- Now, we are going to develop linked-list-based implementations of the same collections.
  - – more efficient that array-based impls.

Linked Queues

- By using composition, a LinkedList object is used as a flexible storage structure for a list of items.
- The LinkedList object performs the Queue operations by executing the equivalent LinkedList operations:
  - – QInsert: InsertRear
  - – ODelete: DeleteFront
  - – QFront: Reset
  - – QLength: ListSize
  - – QEmpty: ListEmpty

```
#ifndef QUEUE_CLASS
#define QUEUE_CLASS

#include <iostream.h>
#include <stdlib.h>

#include "link.h"

template <class T>
class Queue
{
   . . .
}
#endif  // QUEUE_CLASS
```

```
template <class T>
class Queue
{
   private:
    // a linked list object to hold the queue items
      LinkedList<T> queueList;

   public:
      // constructor
      Queue(void);

      // queue access methods
      void QInsert(const T& elt);
      T QDelete(void);

      // queue access
      T QFront(void);

      // queue test methods
      int QLength(void) const;
      int QEmpty(void) const;
      void QClear(void);
};
```

```
// constructor
template <class T>
Queue<T>::Queue(void)
{}

// LinkedList method ListSize returns length of list
template <class T>
int Queue<T>::QLength(void) const
{
   return queueList.ListSize();
}

// LinkedList method ListEmpty tests for empty queue
template <class T>
int Queue<T>::QEmpty(void) const
{
   return queueList.ListEmpty();
}

// LinkedList method ClearList clears the queue
template <class T>
void Queue<T>::QClear(void)
{
   queueList.ClearList();
}
```

```
// LinkedList method InsertRear inserts item at rear
template <class T>
void Queue<T>::QInsert(const T& elt)
{
   queueList.InsertRear(elt);
}

// LinkedList method DeleteFront removes item from front
template <class T>
T Queue<T>::QDelete(void)
{
    // test for an empty queue and terminate if true
   if (queueList.ListEmpty())
   {
      cerr << "Calling QDelete for an empty queue!"
           << endl;
      exit(1);
   }
   return queueList.DeleteFront();
}
```

```
// return the data value from the first item in the queue
template <class T>
T Queue<T>::QFront(void)
{
    // test for an empty queue and terminate if true
   if (queueList.ListEmpty())
   {
      cerr << "Calling QFront for an empty queue!"
           << endl;
      exit(1);
   }

   // reset to front of the queue and return data
   queueList.Reset();
   return queueList.Data();
}
```

Linked SeqList Class

- The SeqList class defines a restricted storage structure that allows items to be inserted only at the rear of the list and deletes only the first item in the list or an item that matches a key.
- The client is permitted to access data in the list using the Find method or by using a position index to read the data value in a node.
- We can use a LinkedList object to hold the data when implementing the SeqList class.

SeqList Class

```
#ifndef SEQLIST_CLASS
#define SEQLIST_CLASS

#include <iostream.h>
#include <stdlib.h>

#include "link.h"

template <class T>
class SeqList
{
   . . .
}
#endif  // SEQLIST_CLASS
```

```
template <class T>
class SeqList
{
     private:
          // linked list object
          LinkedList<T> llist;

     public:
          // constructor
          SeqList(void);

          // list access methods
          int ListSize(void) const;
          int ListEmpty(void) const;
          int Find (T& item);
          T GetData(int pos);

          // list modification methods
          void Insert(const T& item);
          void Delete(const T& item);
          T DeleteFront(void);
          void ClearList(void);
};
```

```
// default constructor. it has nothing to do.
template <class T>
SeqList<T>::SeqList(void)
{}

// use method ListSize to return number elements in list
template <class T>
int SeqList<T>::ListSize(void) const
{
   return llist.ListSize();
}

// use method ListEmpty to test for an empty list
template <class T>
int SeqList<T>::ListEmpty(void) const
{
   return llist.ListEmpty();
}

// use method ClearList to clear the linked list
template <class T>
void SeqList<T>::ClearList(void)
{
   llist.ClearList();
}
```

```
// use method InsertRear to add item at the rear of the list
template <class T>
void SeqList<T>::Insert(const T& item)
{
    llist.InsertRear(item);
}

// use method DeleteFront to remove first item from the list
template <class T>
T SeqList<T>::DeleteFront(void)
{
    return llist.DeleteFront();
}
```

```
// delete node whose data value matches item
template <class T>
void SeqList<T>::Delete(const T& item)
{
    int result = 0;

    // search for item in list. if found, set result to True
    for(llist.Reset();!llist.EndOfList();llist.Next(
    ))
          if (item == llist.Data())
          {
                    result++;
                    break;
          }

    // if item is found, delete it; otherwise return
    if (result)
          llist.DeleteAt();
}
```

```
// return the data value of item at position pos
template <class T>
T SeqList<T>::GetData(int pos)
{
    // check for a valid position
    if (pos < 0 || pos >= llist.ListSize())
    {
        cerr << "pos is out of range!" << endl;
        exit(1);
    }

    // set current linked list position to pos and return data
    llist.Reset(pos);
    return llist.Data();
}
```

```
// take item as the key and search the list. return True if item
// is in the list and False otherwise. if found,
// assign the list element to the reference parameter item
template <class T>
int SeqList<T>::Find (T& item)
{
    int result = 0;

    // search for item in list. if found, set result to True
    for(llist.Reset();!llist.EndOfList();llist.Next())
    if (item == llist.Data())
    {
        result++;
        break;
    }

    // if result is True, update item and return True;
    // otherwise, return False
    if (result)
        item = llist.Data();
    return result;
}
```

9.7a) List Class – Array Implementation

```
#include <iostream.h>
typedef int DataType;
// include the array-based SeqList class
#include "aseqlist.h"

void main(void)
{
    // a list with capacity 500 integers
    SeqList L;
    long  i;

    // initialize the list with values 0 .. 499
    for (i = 0; i < 500; i++)
        L.Insert(int(i));

    // exercise the delete/insert operations 50000 times
    cout << "Program begin!" << endl;
    for (i = 1; i <= 50000L; i++)
    {
        L.DeleteFront();
        L.Insert(0);
    }
    cout << "Program done!" << endl;
}
```

9.7b) List Class Linked List Implementation

```
#include <iostream.h>
// include the linked list implementation of the SeqList
#include "seqlist1.h"

void main(void)
{
    // define an integer list
    SeqList<int> L;
    long  i;

    // initialize the list with values 0 .. 499
    for (i = 0; i < 500; i++)
        L.Insert(int(i));

    // exercise the delete/insert operations 50000 times
    cout << "Program begin!" << endl;
    for (i = 1; i <= 50000L; i++)
    {
        L.DeleteFront();
        L.Insert(0);
    }
    cout << "Program done!" << endl;
}
```

```
/*
<Run of Program 9.7a>

Program begin!
Program done!       // 55 seconds

*/

/*
<Run of Program 9.7b>

Program begin!
Program done!   // 4 seconds
*/
```

Doubly Linked Lists

```
#ifndef DOUBLY_LINKED_NODE_CLASS
#define DOUBLY_LINKED_NODE_CLASS

template <class T>
class DNode
{
   private:
      // circular links to the left and right
      DNode<T> *left;
      DNode<T> *right;
   public:
      T data;

      // constructors
      DNode(void);
      DNode (const T& item);

      // list modification methods
      void InsertRight(DNode<T> *p);
      void InsertLeft(DNode<T> *p);
      DNode<T> *DeleteNode(void);

      // obtain address of the next node to the left or right
      DNode<T> *NextNodeRight(void) const;
      DNode<T> *NextNodeLeft(void) const;
};
```

```
// constructor that creates an empty list and
// leaves the data uninitialized. use for header
template <class T>
DNode<T>::DNode(void)
{
    // initialize the node so it points to itself
   left = right = this;
}

// constructor that creates an empty list and initializes data
template <class T>
DNode<T>::DNode(const T& item)
{
    // set node to point to itself and initialize data
   left = right = this;
   data = item;
}
```

```
// insert a node p to the right of current node
template <class T>
void DNode<T>::InsertRight(DNode<T> *p)
{
    // link p to its successor on the right
   p->right = right;
    right->left = p;

    // link p to the current node on its left
   p->left = this;
   right = p;
}

// insert a node p to the left of current node
template <class T>
void DNode<T>::InsertLeft(DNode<T> *p)
{
    // link p to its successor on the left
   p->left = left;
   left->right = p;

    // link p to the current node on its right
   p->right = this;
   left = p;
}
```

```
// unlink the current node from the list and return its address
template <class T>
DNode<T> *DNode<T>::DeleteNode(void)
{
    // node to the left must be linked to current node's right
   left->right = right;

    // node to the right must be linked to current node's left
   right->left = left;

   // return the address of the current node
   return this;
}

// return pointer to the next node on the right
template <class T>
DNode<T> *DNode<T>::NextNodeRight(void) const
{
   return right;
}

// return pointer to the next node on the left
template <class T>
DNode<T> *DNode<T>::NextNodeLeft(void) const
{
   return left;
}
```

Application: Doubly Linked List Sort

```
include <iostream.h>

#include "dnode.h"

template <class T>
void InsertLower(DNode<T> *dheader,
    DNode<T>* &currPtr, T item)
{
   DNode<T> *newNode= new DNode<T>(item),
    *p;

   // look for the insertion point
   p = currPtr;
   while (p != dheader && item < p->data)
       p = p->NextNodeLeft();

   // insert the item
   p->InsertRight(newNode);

   // reset currPtr to the new node
   currPtr = newNode;
}
```

```
template <class T>
void InsertHigher(DNode<T>* dheader,
    DNode<T>* & currPtr, T item)
{
   DNode<T> *newNode= new DNode<T>(item), *p;

   // look for the insertion point
   p = currPtr;
   while (p != dheader && p->data < item)
       p = p->NextNodeRight();

   // insert the item
   p->InsertLeft(newNode);

   // reset currPtr to the new node
   currPtr = newNode;
}
```

```
template <class T>
void DLinkSort(T a[], int n)
{   // set up the doubly linked list to hold array items
    DNode<T> dheader, *currPtr;
    int i;
    // insert the first element in dlist
    DNode<T>  *newNode = new DNode<T>(a[0]);
    dheader.InsertRight(newNode);
    currPtr = newNode;
     for (i=1;i < n;i++) // insert the remaining elements in dlist
        if (a[i] < currPtr->data)
            InsertLower(&dheader,currPtr,a[i]);
        else
            InsertHigher(&dheader,currPtr,a[i]);
    // scan the list and copy the data values back to the array
    currPtr = dheader.NextNodeRight();
    i = 0;
    while(currPtr != &dheader) {
        a[i++] = currPtr->data;
        currPtr = currPtr->NextNodeRight();
    }
   // delete all nodes in the list
    while(dheader.NextNodeRight() != &dheader) {
    currPtr = (dheader.NextNodeRight())->DeleteNode();
      delete currPtr;
    }
}
```

```
// scan the array and print its elements
void PrintArray(int a[], int n)
{
    for(int i=0;i < n;i++)
        cout << a[i] << "  ";
}

void main(void)
{
    // initialized array with 10 integer values
    int A[10] = {82,65,74,95,60,28,5,3,33,55};

    DLinkSort(A,10);                    // sort the array
    cout << "Sorted array:     ";
    PrintArray(A,10);                   // print the array
    cout << endl;
}

/*
<Run of Program 9.10>

Sorted array:     3  5  28  33  55  60  65  74
    82  95
*/
```