node.h

```
#ifndef NODE_CLASS
#define NODE_CLASS

template <class T>
class Node
{
  private:
     Node<T> *next;
  public:
     T data;

    // constructor
    Node(const T& item, Node<T>* ptrnext=NULL);

    // list modification methods
    void InsertAfter(Node<T> *p);
    Node<T> *DeleteAfter(void);

    // obtain the address of the next node
    Node<T> *NextNode(void) const;
};
. . .
#endif   // NODE_CLASS
```

1

```
// constructor. initialize data and pointer members
template <class T>
Node<T>::Node(const T& item, Node<T>*
  ptrnext)
      : data(item), next(ptrnext)
{ }

// return value of private member next
template <class T>
Node<T> *Node<T>::NextNode(void) const
{
   return next;
}

// insert a node p after the current one
template <class T>
void Node<T>::InsertAfter(Node<T> *p)
{
   // p points to successor of the current node,
   // and current node points to p.
   p->next = next;
   next = p;
}
```

2

```
// delete the node following current and return its
   address
template <class T>
Node<T> *Node<T>::DeleteAfter(void)
{
   // save address of node to be deleted
   Node<T> *tempPtr = next;

   // if there isn't a successor, return NULL
   if (next == NULL)
      return NULL;

   // current node points to successor of tempPtr.
   next = tempPtr->next;

   // return the pointer to the unlinked node
   return tempPtr;
}
```

3

nodelib.h

```
#ifndef NODE_LIBRARY
#define NODE_LIBRARY

#include <iostream.h>
#include <stdlib.h>
// suppress incorrect warning message in
   InsertFront
#pragma warn -par

#include "node.h"

// allocate a node with data member item and
   pointer nextPtr
template <class T>
Node<T> *GetNode(const T& item, Node<T>
   *nextPtr = NULL)
{
   Node<T>  *newNode;

   // allocate memory while passing item and
    NextPtr to
   // constructor. terminate program if
    allocation fails
   newNode = new Node<T>(item, nextPtr);
   if (newNode == NULL)
```

4

Linked List Class

```
#ifndef LINKEDLIST_CLASS
#define LINKEDLIST_CLASS

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

#ifndef NULL
const int NULL = 0;
#endif  // NULL

#include "node.h"

template <class T>
class SeqListIterator;
```

5

```
template <class T>
class LinkedList
{
  private:
    Node<T> *front, *rear;
    Node<T> *prevPtr, *currPtr;
    int size;

    int position;
    Node<T> *GetNode(const T& item,
                     Node<T> *ptrNext=NULL);
    void FreeNode(Node<T> *p);

    void CopyList(const LinkedList<T>& L);
```

6

```
public:
    LinkedList(void);
    LinkedList(const LinkedList<T>& L);

    ~LinkedList(void);

   LinkedList<T>& operator= (
                          const LinkedList<T>& L);

    // methods to check list status
    int ListSize(void) const;
    int ListEmpty(void) const;

    // Traversal methods
    void Reset(int pos = 0);
    void Next(void);
    int EndOfList(void) const;
    int CurrentPosition(void) const;
```



```
    // Insertion methods
    void InsertFront(const T& item);
    void InsertRear(const T& item);
    void InsertAt(const T& item);
    void InsertAfter(const T& item);

    // Deletion methods
    T DeleteFront(void);
    void DeleteAt(void);

    // Data retrieval/modification
    T& Data(void);

    // method to clear the list
    void ClearList(void);

    // this class (Ch. 12) needs access to front
    friend class SeqListIterator<T>;
};
```



```
template <class T>
Node<T> *LinkedList<T>::GetNode(
                    const T& item,
                    Node<T>* ptrNext)
{
  Node<T> *p;

  p = new Node<T>(item,ptrNext);
  if (p == NULL)
  {
    cout << "Memory allocation failure!\n";
    exit(1);
  }
  return p;
}

template <class T>
void LinkedList<T>::FreeNode(Node<T> *p)
{
  delete p;
}
```



```
// copy L to the current list, which is assumed to be empty
template <class T>
void LinkedList<T>::CopyList(const LinkedList<T>& L)
{
  // use p to chain through L
  Node<T> *p = L.front;
  int pos;

  // insert each element in L at the rear of current object
  while (p != NULL)  {
    InsertRear(p->data);
    p = p->NextNode();
  }
  // if list is empty return
  if (position == -1)
    return;

  // reset prevPtr and currPtr in the new list
  prevPtr = NULL;
  currPtr = front;
  for (pos = 0; pos != position; pos++)
  {
    prevPtr = currPtr;
    currPtr = currPtr->NextNode();
  }
}
```



```
// create empty list by setting pointers to NULL,
// size to 0 and list position to -1
template <class T>
LinkedList<T>::LinkedList(void): front(NULL),
    rear(NULL),
     prevPtr(NULL),currPtr(NULL), size(0),
    position(-1)
{}

template <class T>
LinkedList<T>::LinkedList(const LinkedList<T>&
    L)
{
  front = rear = NULL;
  prevPtr = currPtr = NULL;
  size = 0;
  position = -1;
  CopyList(L);
}
```



```
template <class T>
void LinkedList<T>::ClearList(void)
{
  Node<T> *currPosition, *nextPosition;

  currPosition = front;
  while(currPosition != NULL)
  {
    // get address of next node and delete current node
    nextPosition = currPosition->NextNode();
    FreeNode(currPosition);
    currPosition = nextPosition;  // Move to next node
  }
  front = rear = NULL;
  prevPtr = currPtr = NULL;
  size = 0;
  position = -1;
}
```

```
template <class T>
LinkedList<T>::~LinkedList(void)
{
  ClearList();
}

template <class T>
LinkedList<T>& LinkedList<T>::operator=
            (const LinkedList<T>& L)
{
  if (this == &L)      // Can't assign list to itself
    return *this;

  ClearList();
  CopyList(L);
  return *this;
}

template <class T>
int LinkedList<T>::ListSize(void) const
{
  return size;
}
```



```
template <class T>
int LinkedList<T>::ListEmpty(void) const
{
  return size == 0;
}

// move prevPtr and currPtr forward one node
template <class T>
void LinkedList<T>::Next(void)
{
  // if traversal has reached the end of the list or
  // the list is empty, just return
  if (currPtr != NULL)
  {
    // advance the two pointers one node forward
    prevPtr = currPtr;
    currPtr = currPtr->NextNode();
    position++;
  }
}
```



```
// True if the client has traversed the list
template <class T>
int LinkedList<T>::EndOfList(void) const
{
  return currPtr == NULL;
}

// return the position of the current node
template <class T>
int LinkedList<T>::CurrentPosition(void) const
{
  return position;
}
```



```
// reset the list position to pos
template <class T>
void LinkedList<T>::Reset(int pos)
{
  int startPos;

  // if the list is empty, return
  if (front == NULL)
    return;

  // if the position is invalid, terminate the
   program
  if (pos < 0 || pos > size-1)
  {
    cerr << "Reset: Invalid list position: " << pos
         << endl;
    return;
  }
```



```
// move list traversal mechanism to node pos
  if(pos == 0)
  {
     // reset to front of the list
    prevPtr = NULL;
    currPtr = front;
    position = 0;
  }
  else
  // reset currPtr, prevPtr, and position
  {
     currPtr = front->NextNode();
     prevPtr = front;
     startPos = 1;
      // move right until position == pos
     for(position=startPos; position != pos;
                                  position++)
     {
        // move both traversal pointers forward
       prevPtr = currPtr;
       currPtr = currPtr->NextNode();
     }
  }
}
```



```
// return a reference to the data value in the
   current node
template <class T>
T& LinkedList<T>::Data(void)
{
  // error if list is empty or traversal completed
  if (size == 0 || currPtr == NULL)
  {
    cerr << "Data: invalid reference!" << endl;
    exit(1);
  }
  return currPtr->data;
}

// Insert item at front of list
template <class T>
void LinkedList<T>::InsertFront(const T& item)
{
  // call Reset if the list is not empty
  if (front != NULL)
    Reset();
  InsertAt(item);       // inserts at front
}
```

```
// Insert item at rear of list
template <class T>
void LinkedList<T>::InsertRear(const T& item)
{
  Node<T> *newNode;

  prevPtr = rear;
  newNode = GetNode(item);  // create the
   new node
  if (rear == NULL)
        // if list empty, insert at front
    front = rear = newNode;
  else
  {
    rear->InsertAfter(newNode);
    rear = newNode;
  }
  currPtr = rear;
  position = size;
  size++;
}
```

19

```
// Insert item at the current list position
template <class T>
void LinkedList<T>::InsertAt(const T& item)
{
  Node<T> *newNode;

  // two cases: inserting at the front or inside the list
  if (prevPtr == NULL) {
    // inserting at the front of the list. also //places node into
    //an empty list
    newNode = GetNode(item,front);
    front = newNode;
  }
  else { // inserting inside the list. place node after prevPtr
    newNode = GetNode(item);
    prevPtr->InsertAfter(newNode);
  }
  // if prevPtr == rear, we are inserting into empty list
  //or at rear of non-empty list; update rear and position
  if (prevPtr == rear) {
    rear = newNode;
    position = size;
  }
  // update currPtr and increment the list size
  currPtr = newNode;
  size++;          // increment list size
}
```

20

```
// Insert item after the current list position
template <class T>
void LinkedList<T>::InsertAfter(const T& item)
{
  Node<T> *p;

  p = GetNode(item);
  if (front == NULL) {  //inserting into an empty list
    front = currPtr = rear = p;
    position = 0;
  }
  else  {   // inserting after last node of list
    if (currPtr == NULL)
      currPtr = prevPtr;
    currPtr->InsertAfter(p);
    if (currPtr == rear) {
      rear = p;
      position = size;
    }
    else
      position++;
    prevPtr = currPtr;
    currPtr = p;
  }
  size++;          // increment list size
}
```

21

```
// Delete the node at the front of list
template <class T>
T LinkedList<T>::DeleteFront(void)
{
  T item;

  Reset();
  if (front == NULL)
  {
    cerr << "Invalid deletion!" << endl;
    exit(1);
  }
  item = currPtr->data;
  DeleteAt();
  return item;
}
```

22

```
// Delete the node at the current list position
template <class T>
void LinkedList<T>::DeleteAt(void)
{
  Node<T> *p;

  // error if empty list or at end of list
  if (currPtr == NULL)
  {
    cerr << "Invalid deletion!" << endl;
    exit(1);
  }

  // deletion must occur at front node or
   inside the list
  if (prevPtr == NULL)
  {
    // save address of front and unlink it. if
   this
    // is the last node, front becomes NULL
    p = front;
    front = front->NextNode();
  }
  else
    // unlink interior node after currPtr, save
```

23

Program 9.1

```
#include <iostream.h>
#include "node.h"
#include "nodelib.h"
#include "random.h"

void main(void)
{
  // set list head to NULL
  Node<int> *head = NULL, *currPtr;

  int i, key, count = 0;
  RandomNumber rnd;

  // insert 10 random integers at front of list
  for (i=0;i < 10;i++)
    InsertFront(head, int(1+rnd.Random(10)));

  // print the original list
  cout << "List: ";
  PrintList(head,noNewline);
  cout << endl;
```

24

```
    // prompt user for an integer key
    cout << "Enter a key: ";
    cin >> key;

    // cycle through the list
    currPtr = head;
    while (currPtr != NULL)
    {
        // if data matches key, increment count
        if (currPtr->data == key)
            count++;

        currPtr = currPtr->NextNode();
    }

    cout << "The data value " << key << " occurs "
     << count << " times in the list" << endl;
}

/*
<Run of Program 9.1>
List: 3  6  5  7  5  2  4  5  9  10
Enter a key: 5
The data value 5 occurs 3 times in the list
*/
```


Program 9.2

```
#include <iostream.h>
#include "random.h"
#include "strclass.h"
#include "nodelib.h"

void main(void)
{

    // node list to hold jumbled characters
    Node<char> *jumbleword = NULL;

    // input string, random number generator,
     counters
    String  s;
    RandomNumber rnd;
    int i, j;
```



```
    // input four strings
    for (i = 0; i < 4; i++)
    {
        cin >> s;
        for (j = 0; j < s.Length(); j++)
            if (rnd.Random(2))
                InsertRear(jumbleword, s[j]);
            else
                InsertFront(jumbleword, s[j]);

        // print input string and its jumbled variation
        cout << "String/Jumble: " << s << "    ";
        PrintList(jumbleword);
        cout << endl << endl;
        ClearList(jumbleword);
    }
}
```



```
/*
<Run of Program 9.2>

pepper
String/Jumble: pepper   r  p  p  e  p  e

hawaii
String/Jumble: hawaii   i  i  h  a  w  a

jumble
String/Jumble: jumble   e  b  m  j  u  l

C++
String/Jumble: C++   +  C  +
*/
```

31

Program 9.3

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <iomanip.h>

#include "node.h"
#include "nodelib.h"
#include "studinfo.h"

void main(void)
{
   Node<StudentRecord> *graduateList=NULL,
                       *currPtr, *prevPtr,
                       *deletedNodePtr;
   StudentRecord srec;
   ifstream fin;

   fin.open("studrecs",ios::in | ios::nocreate);
   if (!fin)
   {
      cerr << "Cannot open file studrecs." << endl;
      exit(1);
   }
```

32

```
cout.setf(ios::fixed);
cout.precision(1);
cout.setf(ios::showpoint);

while(fin >> srec)
{
   // insert srec at the head of the list
   InsertFront(graduateList,srec);
}
```

<Run of Program 9.3>

```
Students attending
    graduation:
Shannon Johnson  3.8
Rebecca Neeson  4.0
Nancy Barnes  2.5
Harold Nelson  2.9
```

```
/*
<File "studrecs">
Julie Bailey
1.5
Harold Nelson
2.9
Thomas Frazer
3.5
Bailey Harnes
1.7
Sara Miller
3.9
```

```
Nancy Barnes
2.5
Rebecca Neeson
4.0
Shannon Johnson
3.8

<File "noattend">
Thomas Frazer
Sara Miller
```

33

```
prevPtr = NULL;
currPtr = graduateList;
while (currPtr != NULL)
{
   if (currPtr->data.gpa < 2.0) {
      if (prevPtr == NULL) {
         graduateList = currPtr->NextNode();
         deletedNodePtr = currPtr;
         currPtr = graduateList;
      }
      else    // delete node inside the list
      {
         currPtr = currPtr->NextNode();
         deletedNodePtr =prevPtr->DeleteAfter();
      }
      delete deletedNodePtr;
   }
   else
   {
      // no deletion. move on down list
      prevPtr = currPtr;
      currPtr = currPtr->NextNode();
   }
}
fin.close();
```

34

```
fin.open("noattend",ios::in | ios::nocreate);
  if (!fin)
  {
     cerr << "Cannot open file noattend." << endl;
     exit(1);
  }

  while(srec.name.ReadString(fin) != -1)
     Delete(graduateList,srec);

  cout << "Students attending graduation:" << endl;
  PrintList(graduateList,addNewline);
}
```

35

Program 9.4

```
#include <iostream.h>
#include "node.h"
#include "nodelib.h"

template <class T>
void LinkSort(T a[], int n)
{
   Node<T> *ordlist = NULL, *currPtr;
   int i;

   for (i=0;i < n;i++) InsertOrder(ordlist, a[i]);

   // scan the list and copy the data values back to the array
   currPtr = ordlist;
   i = 0;
   while(currPtr != NULL)
   {
      a[i++] = currPtr->data;
      currPtr = currPtr->NextNode();
   }
   // delete all the nodes created for the ordered list
   ClearList(ordlist);
}
```

36

```
// 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};

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

/*
<Run of Program 9.4>

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


Program 9.5

```
#include <iostream.h>

#include "link.h"  // include the linked list class
#include "random.h"


// print list L
template <class T>
void PrintList(LinkedList<T>& L)
{
    for(L.Reset(); !L.EndOfList(); L.Next())
        cout << L.Data() << "  ";
}
```



```
// position the list L at its maximum element
template <class T>
void FindMax(LinkedList<T> &L)
{
   if (L.ListEmpty())
   {
      cerr << "FindMax: list empty!" << endl;
      return;
   }

   // reset to start of the list
   L.Reset();

   T max = L.Data();
   int maxLoc = 0;

   for (L.Next(); !L.EndOfList(); L.Next())
      if (L.Data() > max)
      {
         max = L.Data();
         maxLoc = L.CurrentPosition();
      }

   // reset the list back to the maximum value
   L.Reset(maxLoc);
}
```



```
void main(void)
{
   // list L is placed in sorted order in list K
   LinkedList<int> L, K;
   RandomNumber rnd;
   int i;

   // L is a list of 10 random integers in range 0-99
   for(i=0; i < 10; i++)
      L.InsertRear(rnd.Random(100));
   cout << "Original list: ";
   PrintList(L);
   cout << endl;

   // delete data from L until it is empty, inserting into K
   while (!L.ListEmpty()) {
      FindMax(L);

      K.InsertFront(L.Data());
      L.DeleteAt();
   }
   cout << "Sorted list:   ";
   PrintList(K);
   cout << endl;
}
```

```
/*
<Run of Program 9.5>

Original list: 82  72  62  3  85  33  58  50  91  26
Sorted list:   3  26  33  50  58  62  72  82  85  91
*/
```

43