Chapter 41 – Linked Lists

1. Introducing Linked Lists

Dynamic arrays were introduced earlier as a dynamic data structure. Versatile with randomly accessible elements, dynamic arrays have one substantial disadvantage. Dynamic arrays can only be created from and returned to the free store of memory as a complete unit. Sub-units cannot be removed, nor can they be added when needed.

A linked list is a data structure that can function as a repository for lists and can be resized one element at a time. It can be traversed and single elements can be inserted, moved or deleted without moving any other element. Linked lists have one basic disadvantage. Elements in a linked list cannot be accessed randomly as in an array because linked lists do not have indices.

2. What are Linked Lists?

Linked lists are built from elements that are created from unnamed struct or class objects created with new. Each element (called a node) in a linked list must have a pointer variable (called a link) that points to the next object in the list. Here is a visual example:

```
Node data pointer Node data pointer Node
```

Here is an example of a "hand made" linked list and the struct that its nodes are created from:

```
#include <iostream>
using namespace std;

struct daytemps {  // struct to hold daily hi and low temperatures
    int day, hi, low; // data members
    daytemps *next; // pointer to an object of struct type temp
};

void main( ) {
    daytemps *first, *tp; // pointers to an object of struct type daytemps

    tp = new daytemps( ); // create an unnamed object and assign the address to // the pointer tp
    tp -> day = 1;       // put data in unnamed object
    tp -> hi = 98;
    tp -> low = 73;

    first = tp;         // assign the address of the unnamed object to the // pointer first
```
tp = new daytemps(); // create an unnamed object and assign the address to
    // the pointer tp
tp -> day = 2;     // put data in unnamed object
tp -> hi = 95;
tp -> low = 78;

    // link the unnamed object to the object pointed to by
    // first
first = tp;        // assign the address of the unnamed object pointed
    // to by tp to the pointer first; first should always
    // point to the head of the list

    // ********** output the data stored in the first element in the list ********
cout << first->day << "---" << first->hi << "---" << first->low << endl;

    // ********** output the data stored in the second element in the list *****
cout << first->next->day << "---" << first->next->hi << "---" << first->next->low << endl;

delete first->next; // return the memory for the last object in the list to
    // the free store of memory for reuse
delete first;      // return the memory for the first object in the list to
    // the free store of memory for reuse
}

This example creates the following list from the free store of memory:

```
first -> 2,95,78 -> 1,98,73
```

Before the program ends, the memory used to make this list is then returned to the free
store of memory for reuse.

**Programming Exercise 41.1**
Enter, run and debug the previous example program.

**Programming Exercise 41.2**
Add code to add a third object to the front of the linked list (have first point to it) of
41.1 and code to output the entire list. Remember to delete all nodes in the list.

**Programming Exercise 41.3**
Add a forth object to the back (at the right most side) of the linked list of 41.2 and
code to output the entire list. Remember to all nodes in the list.
3. Basic Algorithms on Linked Lists

Like any other data structure, linked lists are manipulated by a set of algorithms. These are insertion of a new node, location of a node, access to the data of a node, deletion of a node and traversal of the linked list. The member functions of a class based on a linked list data structure are derived from these algorithms.

For any of these algorithms to work, the next pointer of the last node of a linked list must point to a value that is an impossible address and which can uniformly be tested for its presence. This is the purpose of the value NULL, which can be assigned to a pointer of any type. Having NULL as the value of the last pointer of the last node in a linked list relieves the programmer from having to keep track of the number of nodes in the linked list.

Traversal of a linked list is the most basic of the above listed algorithms. Starting at the node pointed to by first, a simple loop with a peculiar version of a counter is sufficient to traverse the list and visit each node. With each node visited, the pointer being used to point to the current node has its value updated to whatever the next pointer of the current node contains. The loop continues until NULL is encountered as a pointer value. The traversal algorithm is as follows:

```c
pointertype *current = first;
while (current != NULL) {
    ****** visit the node ******
    current = current->next;
}
```

Any algorithm that produces a pointer to a node can access the data values stored in that node by pointing to the desired data member. For instance:

```c
current->dataField
```

Visiting a node usually involves accessing one or more members in this fashion.

Locating a node is just a modification of the traversal algorithm to halt when a certain key value is located in a node. Once the node is located, its values or address can be returned, or the node itself copied for use elsewhere. The node not being found presents the only difficulty. In the following outline, the address of the node is returned. NULL is returned if the node was not found.

```c
nodeType *find(keyType item) {
    nodeType *current = first;
    while (current != NULL && current->key != item)
        current = current->next
    return current;
}
```
If the list is unordered, insertion of a new node is very simple, as it is placed at the start of the list. To insert at the start of the list, the new node is created and supplied with values. Whatever \textit{first} points to is the value given to the \textit{next} pointer of the new node. \textit{first} is then set to point to the new node.

\begin{verbatim}
nodeType *t = new nodeType();  // create node t

/****** assign data into node t ******/
 t->dataField1 = data;
 t->dataFiled2 = data;
 :  
 t->dataFiledn = data;
/****** insert the node t into the linked list ******/
 t->next = first;   // make the pointer in the node
                  // t point to the first node in the list

first = t;    // make t the first node
\end{verbatim}

Deleting a node is the most difficult operation involved with unordered linked lists. This is because the regular \textit{find} algorithm does not provide enough information to safely delete a node. The \textit{find} algorithm only returns the address of the node, because that is all it knows. To delete a node, the address of the previous node must also be known. The \textit{next} pointer of the previous node must have its value adjusted so that it contains the value of the \textit{next} pointer in the node to be deleted. If this does not occur, all nodes of the list after the node that was deleted will be inaccessible and their values lost.

In order to delete a node safely, the \textit{find} algorithm must be altered to maintain the address of the node previous to the current node. If the node to be deleted is found and the previous pointer is NULL, the node to be deleted is the first node and the value of \textit{first} will have to be updated. If the node to be deleted is found and the previous node is not NULL, the value of the \textit{next} pointer indicated by the \textit{previous} pointer must be updated.

\begin{verbatim}
nodeType *previous, *current;
current = first;
previous = NULL;  // since there is no previous node
                  // when current points to first, previous must
                  // be set to NULL at the beginning

// ****** find the node and the previous node ******
while (current != NULL &\& current->key != item) {
    previous = current;
    current = current->next;
}
// ****** delete the node and update the correct pointer ******
if (current != NULL) {
    if (previous == NULL) first = first->next;
    else previous->next = current->next;
    delete current;
}
\end{verbatim}
4. A Linked List Class

The following example class contains a linked list based on the struct `daytemps` from the previous example.

```cpp
#include <iostream>
using namespace std;

struct daytemps {   // struct to hold daily hi and low temperatures
    int day, hi, low;  // data members
    daytemps *next;  // pointer to an object of struct type temp
};

class temperatures {
    public:
        temperatures( );  // create class object with empty list
        ~temperatures( );  // return the list to memory

        void insertDay(int day, int high, int low); // *** insert data for a new day in new node ***

        int returnHi(int day);   // *** functions for accessing data ***
        int returnLow(int day);

        void deleteDay(int day); // delete a node

        void listAll( ); // *** functions based on traversing the list ***

    private:
        daytemps * find(int day);   // find a day based on traversal of list
        daytemps *first;  // pointer to the first node in the list
};

NULL is a special value that can be applied to any pointer. When a pointer is assigned the value of NULL, it is pointing to a testable value, but one that points nowhere. The constructor for class `temperatures` must do only one thing. Since the linked list is yet to be constructed, it must initialize the pointer `first` to NULL.

```cpp
temperatures::temperatures( ) {
    first = NULL;
}
```
The \textit{insertDay} member function may only insert the data for a given day if the day in question does not exist. The private function \textit{find} is called to discover if the given day is already in the list. If it is not (\textit{find} returns \texttt{NULL}), a new node ($t$) is must be created, the data assigned to it and, finally, the node inserted into the linked list.

$$
\begin{verbatim}
void temperatures::insertDay(int day, int high, int low) {
    if (find(day) == NULL) {
        daytemps *t = new daytemps(); // create node $t$

        /* assign data into node $t$ */
        t->day = day;
        t->hi = high;
        t->low = low;

        /* insert the node $t$ into the linked list */
        t->next = first; // make the pointer in the node
        // $t$ point to the first node in
        // the list

        first = t; // make $t$ the first node
    }
}
\end{verbatim}
$$

The member function \textit{returnHi} receives a day, then uses the private member function \textit{find} to attempt to locate the node for that day. If the day is found ($t$ is not assigned the value \texttt{NULL}), $t$ points to the node with the data for the desired day and the \textit{Hi} for the day is returned. If the day is not found ($t$ is assigned the value \texttt{NULL}), the value -500 (a value below absolute zero in degrees F) is returned. The member function \textit{returnLow} works the same way, but returns the low temperature for the day if the day exists in the linked list.

$$
\begin{verbatim}
int temperatures::returnHi(int day) {
    daytemps *t = find(day);
    if (t != NULL) return t->hi;
    else return -500;
}

int temperatures::returnLow(int day) {
    daytemps *t = find(day);
    if (t != NULL) return t->low;
    else return -500;
}
\end{verbatim}
$$
The function \textit{find} is created as a private function because it returns a pointer to a node in the linked list. Since the class \textit{temperatures} contains or encapsulates the linked list as part of itself, placing member function \textit{find} in the private section preserves the encapsulation of the data as part of the class.

The function \textit{find} traverses the list in search of the given day until the pointer \textit{current} becomes NULL (when the desired day is not found in the list) or the day is found. In either event, \textit{find} returns the value in \textit{current}. If the day was not found, the value of \textit{current} returned is NULL. If the day was found, the value of \textit{current} returned is the address of the node containing the data of the given day. \textit{Note: The local pointer} current is used instead of first in order to preserve the value stored in first.

```cpp
daytemps * temperatures::find(int day) {
    daytemps *current = first;
    while (current != NULL && current->day != day)
        current = current->next;  // acts as an index counter
        // would when using an array
    return current;
}
```

The member function \textit{deleteDay} deletes the node of the given day from the linked list. It works by traversing the linked list in search of a given day. Unlike the other functions that must know the address or existence of a node, \textit{deleteDay} cannot use the function \textit{find} to locate the node to delete. This is because deleting a node from a linked list has the potential of losing part of the list (the part after the deleted node). The node deleted contains the only link to the next node in the list. In order to avoid losing the address in this link, the function \textit{deleteDay} must assign the value in the pointer \textit{next} of the node to be deleted to the pointer \textit{next} of the node previous to the node to be deleted. In order to do this, \textit{deleteDay} must keep track of the address of the node previous to the current node as it traverses the linked list searching for the given day. Here is portion of the function \textit{deleteDay} that locates the desired node such that pointers previous and current contain the desired information.

```cpp
void temperatures::deleteDay(int day) {
    daytemps *previous, *current;
    current = first;
    previous = NULL;  // since there is no previous node
    // when current points to first, previous must
    // be set to NULL at the beginning
    while (current != NULL && current->day != day) {
        previous = current;
        current = current->next;
    }
```
In \textit{deleteDay}, if the node of the given day is found (pointer \textit{current} is not NULL), there is a node to be deleted. If pointer \textit{previous} is NULL, the node to be delete is the first node in the linked list and the value pointer \textit{first} must be adjusted to point to the second node in the linked list before the current node is deleted. If the pointer \textit{previous} is not NULL, the node to be deleted is past the first node in the linked list and the value of \textit{previous}$\rightarrow$n\textit{ext} must be adjusted to point to the node indicated by \textit{current}$\rightarrow$n\textit{ext}.

\begin{verbatim}
if (current != NULL) {
  if (previous == NULL) first = first->next;
  else previous->next = current->next;
  delete current;
}
/*** end of deleteDay ***/
\end{verbatim}

The function \textit{listAll} traverses the linked list and outputs the day, high temperature and low temperature found in each node in the list.

\begin{verbatim}
void temperatures::listAll( ) {
  cout << "Day\tHigh\tLow\n";
  cout << "= = =\t= = = =\t= = =\n";
  daytemps *current = first;
  while (current != NULL) {
    cout << current->day << '\t' << current->hi << '\t'
      << current->low << endl;
    current = current->next;
  }
}
\end{verbatim}

The destructor function \textit{~temperatures} traverses the linked list, deleting any remaining nodes. Like all destructors, \textit{~temperatures} is automatically activated when a program that uses a temperature object moves outside of the scope of the object.

\begin{verbatim}
temperatures::~temperatures( ) {
  daytemps * t;
  while (first != NULL) {
    t = first;
    first = first->next;
    delete t;
  }
}
\end{verbatim}
Here is the complete listing of struct `daytemps` and class `temperatures`:

```cpp
#include <iostream>
using namespace std;

struct daytemps {
    int day, hi, low;
    daytemps *next;
};

class temperatures {
public:
    temperatures( );
    ~temperatures( );
    void insertDay(int day, int high, int low);
    int returnHi(int day);
    int returnLow(int day);
    void deleteDay(int day);
    void listAll( );
private:
    daytemps * find(int day);
    daytemps *first;
};

temperatures::temperatures( ) {
    first = NULL;
}

temperatures::~temperatures( ) {
    daytemps *t;
    while (first != NULL) {
        t = first;
        first = first->next;
        delete t;
    }
}
```
void temperatures::insertDay(int day, int high, int low) {
    if (find(day) == NULL) {
        daytemps *t = new daytemps();
        t->day = day;
        t->hi = high;
        t->low = low;
        t->next = first;
        first = t;
    }
}

int temperatures::returnHi(int day) {
    daytemps *t = find(day);
    if (t != NULL) return t->hi;
    else return -500;
}

int temperatures::returnLow(int day) {
    daytemps *t = find(day);
    if (t != NULL) return t->low;
    else return -500;
}

daytemps * temperatures::find(int day) {
    daytemps *current = first;
    while (current != NULL && current->day != day)
        current = current->next;
    return current;
}

void temperatures::deleteDay(int day) {
    daytemps *previous, *current;
    current = first;
    previous = NULL;
    while (current != NULL && current->day != day) {
        previous = current;
        current = current->next;
    }
    if (current != NULL) {
        if (previous == NULL) first = first->next;
        else previous->next = current->next;
        delete current;
    }
}
void temperatures::listAll( ) {
    cout << "Day\tHigh\tLow\n";
    cout << "= = =\t= = =\t= = =\n";
    daytemps *current = first;
    while (current != NULL) {
        cout << current->day << 't' << current->hi << 't'
             << current->low << endl;
        current = current->next;
    }
}

Programming Exercise 41.4
Create class temperatures, then using the following main, test the class temperatures.
Note: Change the name of the ".h" file in the second include statement to match the path and name of your header file.

#include <iostream>
#include "temperatures.h"
using namespace std;

int main( ) {
    temperatures t;
    t.insertDay(1,10,5);
    t.insertDay(2,15,3);
    t.insertDay(3,37,33);
    t.insertDay(3,0,0);
    t.insertDay(4,56,23);
    t.insertDay(5,2,1);
    t.listAll( );
    cout << "Average of day 2 is " << (t.returnHi(2)+t.returnLow(2))/2
         << endl;
    t.deleteDay(3);
    t.listAll( );
    return 0;
}
**Programming Exercise 41.5**
Add the following public member functions that depend on the traversal algorithm to class *temperatures*. Add calls to them in the *main* function to test them. Also include output of the returned values.

```cpp
int returnAverageTemp( ); // returns the average temperature over all days
int returnAverageHigh( );  // returns the average high temperature over all days
int returnAverageLow( );    // returns the average low temperature over all days
int returnMaxHigh();        // returns the highest temperature over all days
int returnMinLow();         // returns the lowest temperature over all days
```

**Programming Exercise 41.6**
Alter the class *temperatures* so that the list is kept in ascending order based on day. Use the *insertion sort* algorithm to do this. This will require changes to the member functions *insertDay*, *find* and *deleteDay*. 
Programming Exercise 41.7: First Fit

In most operating system environments, more than one program is running at a time. Even the operating system of a personal computer system with one user must allocate resources such as memory and processor time between several programs running at one time. A user working on a single document in a word processor may think that the computer is only serving that function, but it may be doing many other things. For example, the clock display, the mouse driver, the word processor, the window the word processor where document is displayed, background printing, virus checking, and so on may be in process.

First fit is the name of a strategy that is often used by operating systems when allocating space in memory or on a drive to a running program. When a program (often called a job or process) tries to start, the operating system must find enough memory for it to run. The list of available places in memory is searched until one that is equal of greater in size to the requested space is found. The memory requested by the starting job is removed from the available list and placed in the in-use list. The portion of memory that was found that was not used by the job is left in the available list. If no large enough area of memory is available, the job can not run. When a job terminates, the memory it was using is removed from the in-use list and returned to the available list.

Linked lists are generally used to represent the available and in-use memory in a first fit scheme.

The in-use linked list starts off empty. New jobs are added to the front of the list, terminated jobs are searched for and deleted from the list. The node for the in-use list consists of the job number, the starting address of the memory allocated to the job, the amount of memory the job uses and the pointer to the next node.

A node for the available memory linked list must contain the starting address of the memory segment it represents and the amount of memory available at that address. The available memory linked list starts off with one node. This node contains all memory that is available for jobs to request. When a job requests memory, the first node in the available list that can provide the memory is called upon to do so, hence the name first fit. The memory is subtracted from the amount of memory that the node in the available list contains and its address is adjusted. If all of the memory that a node represents is used, the node is deleted.

The difficulty for the programmer in managing an available list is when a terminated job's memory must be returned. If the starting address and ending address of the returned memory is not adjacent to the memory of any existing node in the available list, a new node is created and inserted in address order. If the starting or ending address of the memory returned is adjacent to the address of an existing node in the available list, the memory is simply added to that node and that node's address and / or length is changed.

The problem comes when the beginning address of the returned memory and the ending address of the returned memory are both adjacent to the memory of existing nodes. This makes it necessary for the returned memory to be added to first of the existing nodes and for the memory of the second existing node to be added to that of the first existing node as well. The second existing node is then no longer needed and can then be deleted.
The following is an example memory being used for various jobs. At the beginning, all memory is available and none is in-use. Available memory will be represented by nodes of the address of the beginning of the memory block, the length of the block and the pointer to the next block. In-use memory will be represented by nodes of the job number, address of the memory block, the length of the block and a pointer to the next block.

Available: fa\[0\] 100\[\rightarrow\] NULL
In-use: fi\[\rightarrow\]NULL

Job #1 is run and requests 10 units of memory. This is taken from the available list and recorded in the in-use list.

Available: fa\[10\] 90\[\rightarrow\] NULL
In-use: fi\[\#1\] 0\[\rightarrow\] 10\[\rightarrow\]NULL

Job #2 is run and requests 15 units of memory. This is taken from the available list and recorded in the in-use list.

Available: fa\[25\] 75\[\rightarrow\] NULL
In-use: fi\[\#2\] 10\[\rightarrow\] 15\[\rightarrow\] #1\[\rightarrow\] 0\[\rightarrow\] 10\[\rightarrow\] NULL

Job #3 is run and requests 10 units of memory. This is taken from the available list and recorded in the in-use list.

Available: fa\[35\] 65\[\rightarrow\] NULL
In-use: fi\[\#3\] 25\[\rightarrow\] 10\[\rightarrow\] #2\[\rightarrow\] 10\[\rightarrow\] 15\[\rightarrow\] #1\[\rightarrow\] 0\[\rightarrow\] 10\[\rightarrow\] NULL

Job #2 now terminates. The memory that it used is removed from the in-use list and returned to the available list. Since this memory block is not adjacent to any other available memory block, it is recorded in a new node.

Available: fa\[10\] 15\[\rightarrow\] 35\[\rightarrow\] 65\[\rightarrow\] NULL
In-use: fi\[\#3\] 25\[\rightarrow\] 10\[\rightarrow\] #1\[\rightarrow\] 0\[\rightarrow\] 10\[\rightarrow\] NULL

Job #4 is run and requests 20 units of memory. This amount of memory is not available in the first node of the available list, so it is taken from the second node and recorded in the in-use list.

Available: fa\[10\] 15\[\rightarrow\] 55\[\rightarrow\] 45\[\rightarrow\] NULL
In-use: fi\[\#4\] 35\[\rightarrow\] 20\[\rightarrow\] #3\[\rightarrow\] 25\[\rightarrow\] 10\[\rightarrow\] #1\[\rightarrow\] 0\[\rightarrow\] 10\[\rightarrow\] NULL
Job #1 now terminates. Its memory is removed from the in-use list and returned to the available list. Since the memory block that job #1 was using is adjacent to the memory block in the first node of the available list, the memory from job #1 is added to the memory recorded in the first node and its starting address is adjusted.

Available: fa→0 25 →55 45 →NULL
In-use: fi→#4 35 20 →#3 25 10 →NULL

Job #4 now terminates. Its memory is removed from the in-use list and returned to the available list. Since the memory block that job #4 was using is adjacent to the memory block in the second node of the available list, the memory from job #4 is added to the memory recorded in the second node and its starting address is adjusted.

Available: fa→0 25 →35 65 →NULL
In-use: fi→#3 25 10 →NULL

Job #3 now terminates. Its memory is removed from the in-use list and returned to the available list. Since the memory block that job #3 was using is adjacent to the memory block in the first node of the available list, the memory from job #3 is added to the memory recorded in the first node, but its starting address does not have to be adjusted. This leaves two nodes in the available list. The memory blocks of these two nodes are adjacent. The memory of the second of the adjacent nodes is added to the first and the second node is removed from the list.

Available: fa→0 100 →NULL
In-use: fi→NULL

Create a class that uses a linked list to represent the available list of memory. Create a class that uses a linked list to represent the in-use list of memory. Use these classes to create a program that simulates first fit memory allocation by an operating system. The user should be able to:

- specify the total amount of memory available to the system at startup
- enter new jobs by specifying the amount of memory it will need
- terminate running jobs
- view the available and in-use memory lists
**Programming Exercise 41.8: Best Fit**

The first fit algorithm of 41.7 has a flaw. Since the job requesting memory gets the first available space, large blocks of available memory tend to be broken up into smaller blocks. If a program that needs a large amount of memory comes along, a single block of memory that size may not be available, event though the total of all memory available could handle the job's request.

For example, jobs #1, #2, and #3 request 25, 20 and 40 units of memory for a total of 85 units out of 100 available units of memory. After a while, job #1 terminates, leaving the following available memory:

```
Available: fa       0     25  85   15            NULL
```

Then, job #4 starts up and requests 15 units of memory, leaving the following available memory:

```
Available: fa       15    10  85   15            NULL
```

Then, job #5 tries to start up and requests 20 units of memory. Even though there is enough total memory available to run job #5, there is not enough in one block of available memory. Job #5 cannot run.

*Best fit* is a variation of the first fit allocation algorithm that can solve the dilemma illustrated by the previous example. In best fit, available memory is kept sorted in order of the smallest available memory block to largest, not on memory block address. If we revisit the previous example of memory allocation, but this time apply the best first idea to available memory, job #5 will find enough memory to run:

```
Available: fa       85   15             0     25            NULL
```

Then, job #4 starts up and requests 15 units of memory, leaving the following available memory:

```
Available: fa       0     25          NULL
```

Now, when job #5 tries to start up and requests 20 units of memory, there is a block large enough to grant the 20 units of memory and job #5 can run!

From this example, it is easy see how best fit got its name. Since the available memory is kept in ascending order on size, the first node that contains enough memory to satisfy a job's request for memory is the closest in size to the memory needed, hence it is the *best fit*. 
As advantageous as best fit is, it does create additional complexity when returning memory to the available list. Since memory blocks are not longer kept in order of address, the entire list must be searched for adjacent blocks of memory before returning block can be inserted into the available list.

Copy the classes and program of 41.7. Rewrite the available memory list class so that it uses the best fit algorithm instead of the first fit.

**Programming Exercise 41.9: Doubly Linked Lists**

Linked lists can be *doubly linked*. That is, each node in the list not only points to the next node in the list but also can point to the previous node in the list.

```
previous node ← data → next node
```

Using a doubly linked list allows searching in either direction. Doubly linked lists are used in situations where a node to be sought is likely to be near the last node sought. A current pointer is always preserved so that the next search of the list can begin their instead of a the first node in the list.

Using a doubly linked list adds complexity not only to searching, but to insertion and deletion, since two links must be manipulated on each node.

Devise a doubly linked list demonstration class and use it in a program.

**Programming Exercise 41.10**

Linked lists normally terminate with the value NULL in the *next* pointer of the last node in the list. Instead of NULL, the *next* pointer of the last node can point to the first node in the list. (In a doubly linked list, the *previous* pointer of the first node would also be set to point to the last node.) This type of link value is called a thread (not to be confused with threads in the Java programming language) and, in effect, creates a loop.

The problem that linking the last *next* pointer to the first node of a linked list introduces is this: When has the entire list been traversed in a search or other algorithm? There are two possible answers. First, the staring pointer value could be preserved. When the current node pointer equals the starting pointer, the traversal can stop. Second, the current node pointer can always be checked against the pointer to the first node. When they are equal, the traversal can stop. Which one is appropriate depends on where the traversal starts, the ordering of the list and other features of the data being represented.

Change the program of 41.5 or 41.6 to use a threaded link to the first node from the last node in the list.