Linked List

1. Linked List

A linked list is a series of connected nodes, where each node is a data structure that contains at least one data field and a pointer for linking or pointing to the next node in the list. The nodes are dynamically allocated, used, and deleted, allowing the linked list to grow or shrink in size as the program runs.

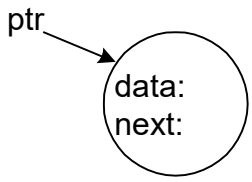
The following shows a structure for a list node containing an integer data field and a pointer to a list node.

```
// data structure for a linked list node
struct ListNode {
    int data;            // data field
    ListNode *next;      // pointer for pointing to the next node in the list
};
```

In the following, memory is allocated for storing one list node, and the starting address of this chunk of memory is assigned to the pointer variable ptr.

```
    ListNode *ptr;       // pointer for pointing to a ListNode
    // create a new node
    ptr = new(ListNode); // allocate memory for a new ListNode
```

Graphically this is what it looks like after executing the above two statements



To access the fields in a struct or class with a pointer, we use the `->` symbol. To access the *data* and *next* fields we do

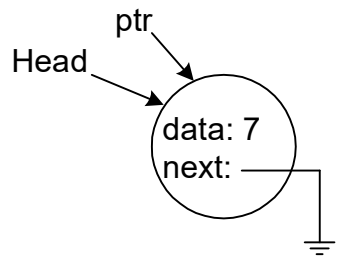
```
    ptr->data = 7;
    ptr->next = NULL;    // assign the NULL value to the next pointer
```


2. The Basic (non-Class) Version

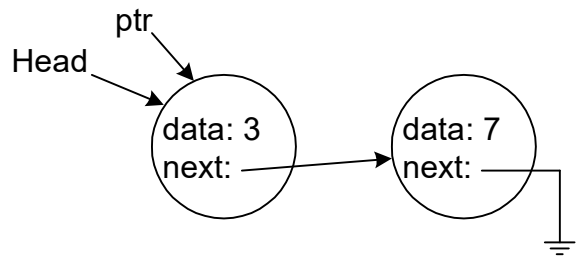
The main program.

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

// data structure for a linked list node
struct ListNode {
    int data;            // data field
    ListNode *next;      // pointer for pointing to the next node in the list
};

int main() {
    ListNode *Head;      // create a pointer to the head of the linked list
    Head = NULL;         // initially the list is empty so the Head points to null

    ListNode *ptr;       // a temporary pointer for pointing to a ListNode

    // create first new node
    ptr = new(ListNode); // allocate memory for a new ListNode
    ptr->data = 7;       // put data in new node
    ptr->next = NULL;    // new node is at the end of the list
    Head = ptr;          // set head to point to this new node

    // create second new node
    ptr = new(ListNode); // allocate memory for a new ListNode
    ptr->data = 3;       // put data in new node
    ptr->next = Head;    // new node is at the end of the list
    Head = ptr;          // set head to point to this new node

    // traverse the linked list and print out the data of each node in the list
    ptr = Head;
    cout << "Here's the linked list..." << endl;
    while (ptr != NULL) {
        cout << ptr->data << endl;
        ptr = ptr->next;
    }

    return 0;
}
```

This is what it looks like after creating the first node and the links



This is what it looks like after creating the second node and the links



Sample output:

```
Here's the linked list...
3
7
```

The use of -> vs . symbols

The use of the symbol -> as in

```
    ptr->data = 7;
```

is similar to the use of the dot notation .

```
    coordinate.longitude = 123.45;
```

In both cases we are accessing a data member field inside a structure. We use -> when using it with a pointer, whereas we use the dot notation when using it with a regular variable. For example,

```
struct Coordinate {
    float longitude;           // data member field
    float latitude;            // data member field
};

int main() {
   Coordinate coor;     // create a variable of type Coordinate
   Coordinate *ptr;     // create a pointer of type Coordinate
   ptr = new Coordinate;  // allocate memory for a new Coordinate

   coor.longitude = 123.45;
   ptr->longitude = 456.78;

   cout << coor.longitude << endl;
   cout << ptr->longitude << endl;
}
```


3. Class Version

The linked list header file LinkedList.h.

```
/*
 *  LinkedList.h
 *
 *  Copyright 2010-02-01 Enoch Hwang
 */
#ifndef __LINKEDLIST__
#define __LINKEDLIST__
#include <iostream>
using namespace std;

struct ListNode {
    int data;
    ListNode* next;
};

class LinkedList {
private:
    ListNode* Head;

public:
    LinkedList();
    void InsertNode(int d);
    ListNode* SearchNode(int n);
    ListNode* SearchNode(int n, ListNode*& pptr);
    bool ChangeNode(int oldN, int newN);
    bool DeleteNode(int d);
    friend ostream& operator<<(ostream& out, const LinkedList& ll);

};
#endif
```

The linked list definition file LinkedList.cpp.

```
#include "LinkedList.h"

LinkedList::LinkedList() {
    Head = NULL;
}

// insert new node at the head of the list
void LinkedList::InsertNode(int d) {
    ListNode* ptr = new(ListNode);
    ptr->data = d;
    ptr->next = Head;
    Head = ptr;
}

// search for node in list
// returns pointer to node if found
// else returns NULL
ListNode* LinkedList::SearchNode(int n) {
    ListNode* ptr;
    ptr = Head;
    while (ptr != NULL) {        // traverse list
        if (ptr->data == n) {      // found node
            return ptr;              // return pointer to found node
        }
        ptr = ptr->next;            // go to next node
    }
    return NULL;
}

// search for node in list
// returns pointer to node if found
// else returns NULL
// also returns the previous pointer in argument list
ListNode* LinkedList::SearchNode(int n, ListNode*& pptr) {
    ListNode* ptr = Head;
    pptr = NULL;
    while (ptr != NULL) {        // traverse list
        if (ptr->data == n) {      // found node
            return ptr;              // return pointer to found node
        }
        pptr = ptr;                // update previous pointer
        ptr = ptr->next;            // go to next node
    }
    return NULL;
}

bool LinkedList::ChangeNode(int oldN, int newN) {
    ListNode* ptr = SearchNode(oldN);  // search for node
    if (ptr) {                         // found node to change
        ptr->data = newN;
        return true;
    }
    else {
        return false;
    }
}
```



```
bool LinkedList::DeleteNode(int d) {
    ListNode* ptr;
    ListNode* pptr;                  // previous pointer
    ptr = SearchNode(d, pptr);  // search for node
    if (ptr != NULL) {          // found node to delete
        if (ptr == Head) {        // need to treat the first node differently
            Head = Head->next;      // delete the head node
        }
        else {                  // delete a non-head node
            pptr->next = ptr->next;
        }
        delete ptr;               // release memory
        return true;
    }
    else {
        return false;
    }
}

ostream& operator<<(ostream& out, const LinkedList& ll) {
    ListNode* ptr;
    ptr = ll.Head;
    while (ptr != NULL) {
        cout << ptr->data << ", ";
        ptr = ptr->next;
    }
    cout << endl;
    return out;
}
```

The main program.

```
/*
 *  main.cpp
 *
 *  Copyright 2010-02-01 Enoch Hwang
 */
#include <iostream>
#include "LinkedList.h"
using namespace std;


int main() {
    LinkedList L;

    // insert 6 nodes
    for (int i = 6; i > 0; i--) {
        L.InsertNode(i);
    }
    cout << "Here's the linked list after inserting 6 nodes..." << endl;
    cout << L;

    // change node 5
    L.ChangeNode(5, 123);
    cout << "Here's the linked list after changing node 5 to 123..." << endl;
    cout << L;
```



```
    // delete node 4
    L.DeleteNode(4);
    cout << "Here's the linked list after deleting node 4..." << endl;
    cout << L;

    // delete node 1
    L.DeleteNode(1);
    cout << "Here's the linked list after deleting node 1..." << endl;
    cout << L;

    return 0;
}
```

Sample output:

```
Here's the linked list after inserting 6 nodes...
1, 2, 3, 4, 5, 6,
Here's the linked list after changing node 5 to 123...
1, 2, 3, 4, 123, 6,
Here's the linked list after deleting node 4...
1, 2, 3, 123, 6,
Here's the linked list after deleting node 1...
2, 3, 123, 6,
```


4. *Exercises* (Problems with an asterisk are more difficult)

Write a class for the following functions, and test out your class in main.

1. **Insert**. Create a linked list with 10 nodes. The data in each node is an automatically generated random number between 1 and 100. Each new node is added to the beginning of the linked list.

2. **Traverse**. Traverse the linked list created in question 1 and print out the numbers from all the nodes.

3. Repeat question 1 but add each new node to the end of the linked list.

4. **Search**. Have the user enter a number, then search through the list created in question 1 Print out the appropriate message from the search. The Search function itself does not do any I/Os. See sample in section 3.

5. *** Change**. Have the user enter a number, then search through the list created in question 1. If the number is found then ask the user to enter another number. Replace the original number with the new number in that node. The Change function itself does not do any I/Os. See sample in section 3.

6. *** Delete**. Have the user enter a number, then search through the list created in question 1. If the number is found then ask the user if s/he wants to delete that node from the list. If yes, then delete the node from the list. The Delete function itself does not do any I/Os. See sample in section 3.

Questions 7 to 10 are exactly like questions 1 to 6 except using a temperature as the data value instead of an integer. If you have overloaded all the needed operators in the temperature class then all you needed to do is to replace all the references to the integer data type with the temperature type and everything should work.

7. *** Insert**. Create a linked list with 5 nodes. The data in each node is a temperature value. Use the Temperature class that you have created. As you add each node, ask the user to input a temperature value for the node. You'll need to use the >> operator from the Temperature class. Add each new node to the beginning of the linked list.

8. *** Traverse**. Traverse the linked list created in question 7 to print out the temperatures from the nodes. You'll need to use the << operator from the Temperature class.

9. *** Change**. Have the user enter a temperature, then search through the list created in question 7. If the temperature is found then ask the user to enter another temperature. Replace the original temperature with the new temperature in that node.

10. *** Delete**. Have the user enter a temperature, then search through the list created in question 7. If the temperature is found then ask the user if s/he wants to delete that node from the list. If yes, then delete the node from the list.

