Linked Lists

Dynamic variables combined with structs or classes can be linked together to form dynamic lists or other structures. We define a record (called a node) that has at least two members: next (a pointer to the next node in the list) and component (the type of the items on the list). For example, let's assume that our list is a list of integers.

```
class Node
{
  private:
    int num;                    // Some numeric value for this node
    Node  *next;                // Pointer to a NodeType
  public:
    Node();
    Node(int num);
    void setNum(int n);
    int getNum();
    void setNext(Node *next);
    Node* getNext();
};
Node::Node()
{
        num = 0;
        next = NULL;
}
Node::Node(int n) : num(n), next(NULL)
{ }
// You should be able to implement setNum,getNum,setNext,getNext

Node  *headPtr = NULL;          // Pointer to the first thing in the list
Node  *newNodePtr = NULL;       // extra pointer
```

To form dynamic lists, we link variables of Node together to form a chain using the next member. We get the first dynamic list node and save its address in the variable headPtr. This pointer is to the first node in the list. We then get another node and have it accessible via Node:

```
headPtr = new Node();
newNodePtr = new Node();
```

Next, let's store some data into the node pointers. To access the structure, we have to first de-reference the pointer (using *) and then we need to use the dot notation to access the member of the structure:

```
(*headPtr).setNum(55);
(*headPtr).setNext(NULL);
```

Instead of using the * and the . separately, C++ supports a special operator to do both simultaneously. This operator is the arrow: -> and it is identical to dereferencing a pointer and then accessing a structure:

```
newNodePtr->setNum(55);
newNodePtr->setNext(NULL);
```

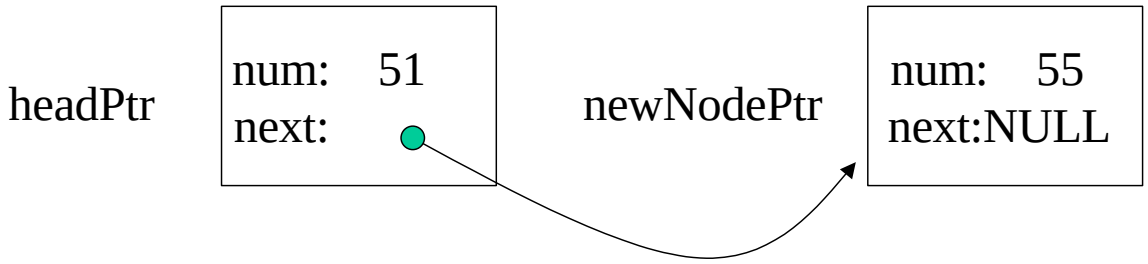
is identical to

```
(*newNodePtr).setNum(55);
(*newNodePtr).setNext(NULL);
```

Right now we have two separate Nodes. We can link them together to form a linked list by having the next field of one pointing the address of the next node:

```
headPtr->setNext(newNodePtr);
```

We now have a picture that looks like:



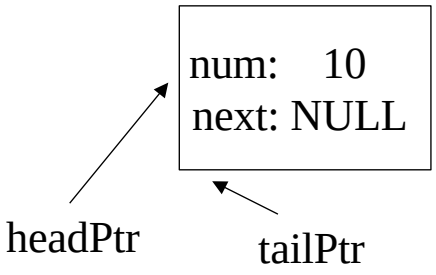
We just built a linked list consisting of two elements! The end of the list is signified by the *next* field holding NULL.

We can get a third node and store its address in the next member of the second node. This process continues until the list is complete. The following code fragment reads and stores integer numbers into a list until the input is –1:

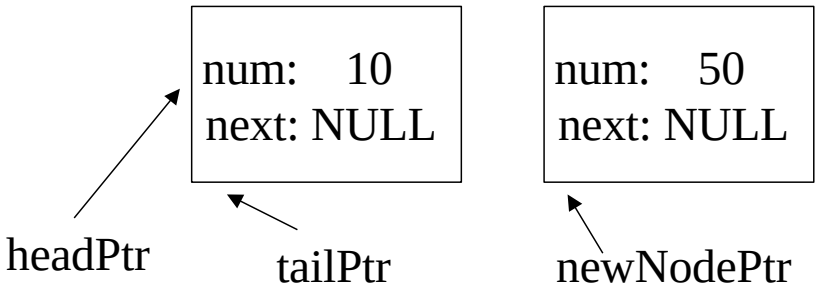
```
int main()
{
 Node *headPtr = NULL, *newNodePtr = NULL,
        *tailPtr = NULL, *tempPtr = NULL;
 int temp;

 headPtr = new Node();
 tailPtr = headPtr;                  // Points to the end of the list
 cout << "Enter value for first node" << endl;
 cin >> temp;
 headPtr->setNum(num);          // Require at least one value

 cout << "Enter values for remaining nodes, with -1 to stop." << endl;
 cin >> temp;
 while (temp!=-1) {
```

```
        // First fill in the new node
        newNodePtr = new Node();
        newNodePtr->setNum(temp);   // next set to NULL in constructor
        // Now link it to the end of the list
        tailPtr->setNext(newNodePtr);
        // Set tail to the new tail
        tailPtr = newNodePtr;
        // Get next value
        cin >> temp;
    }
```

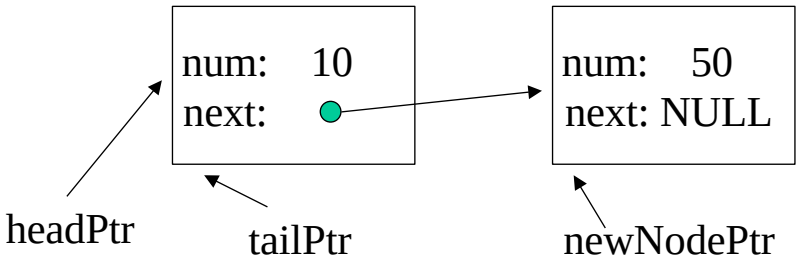
This program (it is incomplete, we'll finish it below) first allocates memory for headPtr and inputs a value into it. It then sets tailPtr equal to headPtr. tailPtr will be used to track the end of the list while headPtr will be used to track the beginning of the list. For example, let's say that initially we enter the value 10:



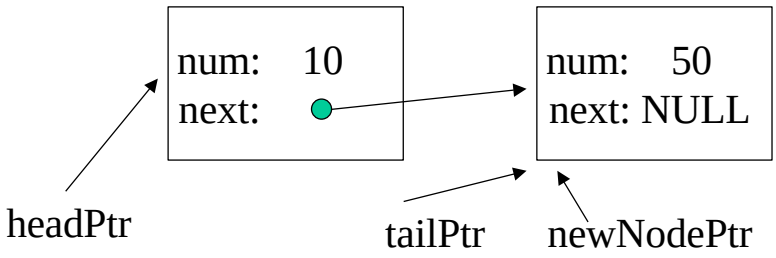
Upon entering the loop, let's say that we enter 50 which is stored into temp. First we create a new node, pointed to by newNodePtr, and store data into it:



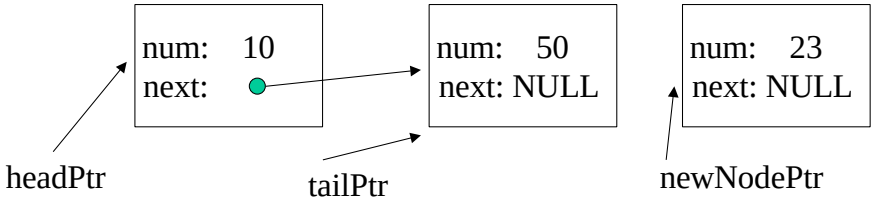
Then we link tailPtr->next to newNodePtr:



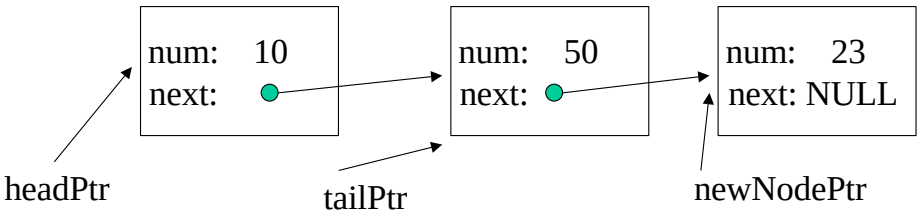
Finally we update tailPtr to point to newNodePtr since this has become the new end of the list:



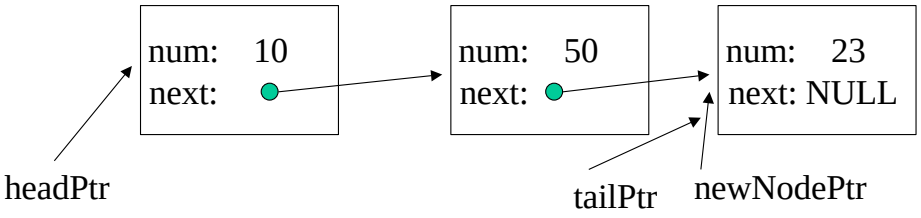
Let's say that the next number we enter is 23. We will repeat the process, first allocated a new node pointed to by newNodePtr, and filling in its values:



Then we link tailPtr to newNodePtr:



Finally we update tailPtr to point to the new end of the list, newNodePtr:



The process shown above continues until the user enters –1. Note that this allows us to enter an arbitrary number of elements, up until we run out of memory! This overcomes limitations with arrays where we need to pre-allocate a certain amount of memory (that may turn out later to be too small).

Lists of dynamic variables are traversed (nodes accessed one by one) by beginning with the first node and accessing each node until the next member of a node is NULL. The following code fragment prints out the values in the list.

```
cout << "Printing out the list" << endl;
```

```
tempPtr = headPtr;
while (tempPtr!=NULL) {
      cout << tempPtr->getNum() << endl;
      tempPtr=tempPtr->getNext();
}
```

tempPtr is initialized to headPtr, the first node. If tempPtr is NULL, the list is empty and the loop is not entered. If there is at least one node in the list, we enter the loop, print the member component of tempPtr, and update tempPtr to point to the next node in the list. tempPtr is NULL when the last number has been printed, and we exit the loop.

Once we have printed out the data, we're not done! Before we exit the program, we should make certain to free up the memory we allocated to prevent memory leaks. We can do so in a loop similar to the one we used to print out the list:

```
// Now free the dynamically allocated memory to prevent memory leaks
 while (headPtr!=NULL) {
      tempPtr=headPtr;
      headPtr=headPtr->getNext();
      delete tempPtr;
 }
}        // End program (piecing together all of the above)
```

This loop goes through and frees each node until we reach the end.

Note that we used two pointers above, tempPtr and headPtr. What is wrong with the following?

```
while (headPtr!=NULL) {
      delete headPtr;
      headPtr=headPtr->getNext();
}
```

Because the types of a list node can be any data type, we can create an infinite variety of lists. Pointers also can be used to create very complex data structures such as stacks, queues, and trees that are the subject of more advanced computer science courses.

Here is a complete example that uses a linked list to store strings:

```
class Node
{
  private:
        string name;
        Node* next;
  public:
        Node();
        Node(string theName, Node* nextNode);
        string getName();
        void setName(string newName);
        Node* getNext();
        void setNext(Node *newNext);
};
Node::Node() : name(""), next(NULL)
{
}
Node::Node(string theName, Node* nextNode) :
        name(theName), next(nextNode)
{
}
string Node::getName()
{
        return name;
}
void Node::setName(string newName)
{
        name = newName;
}
Node* Node::getNext()
{
        return next;
}
void Node::setNext(Node *newNext)
{
        next = newNext;
}
```

```
class LinkedList
{
  private:
        Node* head;             // Points to the first thing in the list
  public:
        LinkedList();
        ~LinkedList();
        void add(string s); // Adds to front of list
        void remove(string s); // Deletes first occurrence of s in the list
        void output();  // Output everything in the list
};
LinkedList::LinkedList()
{
        head = nullptr;
}
LinkedList::~LinkedList()
{
        // TO DO: Free up the memory allocated to this linked list
}

// Adds a new node with this string to the front of the linked list
void LinkedList::add(string s)
{
        if (head == nullptr)
        {
                // New list, make head point to it
                head = new Node(s, nullptr);
        }
        else
        {
                // Make a new node that points to head
                Node *temp = new Node(s, head);
                // Set head to the new node
                head = temp;
        }
}

// Deletes first occurrence of s in the list
void LinkedList::remove(string s)
{
        Node *temp = head;
              Node *prev = nullptr;
        while (temp != nullptr)
        {
                if (temp->getName() == s)
                {
                        if (prev == nullptr)
                                        {
                                        // We're deleting the head of the list
                                               head = head->getNext();
                                               delete temp;
                                        }
                                        else
                                        {
                                               // We're deleting past the head
                                               prev->setNext(temp->getNext());
                                               delete temp;
```

```
                                        }
                        break;
                }
                        prev = temp;
                temp = temp->getNext();
        }
}

// Output everything in the list
void LinkedList::output()
{
        Node *temp = head;
        while (temp != nullptr)
        {
                cout << temp->getName() << endl;
                temp = temp->getNext();
        }
        cout << endl;
}

int main()
{
  LinkedList mylist;

  mylist.add("Andy");
  mylist.add("Bobo");
  mylist.add("Chuck");
  mylist.add("Drogo");
  mylist.add("Erin");
  mylist.add("Frodo");

  cout << "Output entire list" << endl;
  mylist.output();

  cout << "Output after removing Chuck" << endl;
  mylist.remove("Chuck");
  mylist.output();
  cout << "Output after removing Frodo" << endl;
  mylist.remove("Frodo");
  mylist.output();

  cout << "Output after removing Andy" << endl;
  mylist.remove("Andy");
  mylist.output();

  system("pause");
  return 0;
}
```

Things to add:

1) Write an “InsertBefore” method that inserts a node before another specified node. For example, mylist.insertBefore(“Bilbo”,“Frodo”) would insert “Bilbo” in front of “Frodo”, if found. If not found, just insert on the front.
2) Convert the node and class into a doubly-linked list rather than a singly linked list.