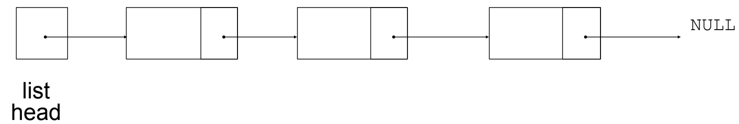
Ch. 17: Linked Lists

CS 2308
Spring 2015

Jill Seaman

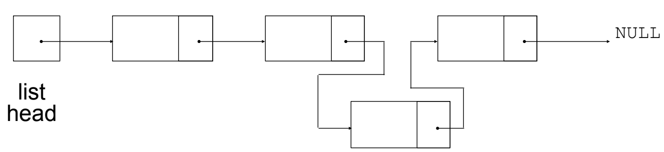
17.1 Introduction to Linked Lists

- A data structure representing a list
- A series of nodes chained together in sequence
  - Each node points to one other node.
- A separate pointer (the head) points to the first item in the list.
- The last element points to nothing (NULL)



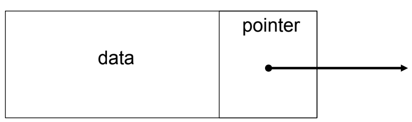
Introduction to Linked Lists

- The nodes are **dynamically allocated**
  - The list grows and shrinks as nodes are added/ removed.
- Linked lists can easily insert a node between other nodes
- Linked lists can easily delete a node from between other nodes



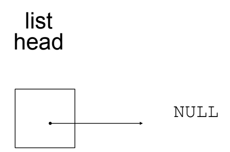
Node Organization

- The node is usually implemented as a struct
- Each node contains:
  - a data field – may be organized as a structure, an object, etc.
  - a pointer – that can point to another node



Empty List

- An empty list contains 0 nodes.
- The list head points to NULL (address 0)
- (There are no nodes, it's empty)



Declaring the Linked List data type

- We will be defining a class to represent a linked list data type that can store values of type double.
- This data type will describe the values (the lists) and operations over those values.
- In order to define the values we must:
  - define a (nested) data type for the nodes
  - define a pointer variable (head) that points to the first node in the list.

Declaring the Node data type

- Use a struct for the node type

```
struct ListNode {
   double value;
   ListNode *next;
};
```

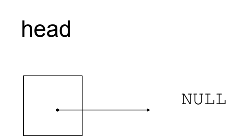
- (this is just a data type, no variables declared)
- next can hold the address of a ListNode.
  - it can also be NULL
  - "self-referential data structure"

Defining the Linked List member variable

- Define a pointer for the head of the list:

```
ListNode *head;
```

- It must be initialized to NULL to signify the end of the list.
- Now we have an empty linked list:



Using NULL

- Equivalent to address 0
- Used to specify end of the list
- In C++11, you can use `nullptr` instead of `NULL`
- `NULL` is defined in the cstddef header:

```
#include <cstddef>
```

- to test a pointer for NULL (these are equivalent):

```
while (p) ...  <==>  while (p != NULL) ...

if (!p) ...  <==>  if (p == NULL) ...
```


17.2 Linked List operations

- Basic operations:
  - **create** a new, empty list
  - **append** a node to the end of the list
  - **insert** a node within the list
  - **delete** a node
  - **display** the linked list
  - **delete/destroy** the list

Linked List class declaration

NumberList.h

```
#include <cstddef>   // for NULL
using namespace std;

class NumberList
{
   private:
      struct ListNode      // the node data type
      {
         double value;     // data
         ListNode *next;   // ptr to next node
      };
      ListNode *head;      // the list head

   public:
      NumberList();        // creates an empty list
      ~NumberList();

      void appendNode(double);
      void insertNode(double);
      void deleteNode(double);
      void displayList();
};
```


Operation: **Create** the empty list

- Constructor: sets up empty list

NumberList.cpp

```
#include "NumberList.h"


NumberList::NumberList()
{
   head = NULL;
}
```

Operation: **append** node to end of list

- appendNode: adds new node to end of list
- Algorithm:

Create a new node and store the data in it
If the list has no nodes (it's empty)
   Make head point to the new node.
Else
   Find the last node in the list
   Make the last node point to the new node

appendNode: find last elem

- How to find the last node in the list?
- Algorithm:

Make a pointer p point to the first element
while (the node p points to) is not pointing to NULL
   make p point to (the node p points to) is pointing to

- In C++:

```
ListNode *p = head;
while ((*p).next != NULL)
   p = (*p).next;
```

<==>

```
ListNode *p = head;
while (p->next)
   p = p->next;
```

p=p->next is like i++

in NumberList.cpp

```
void NumberList::appendNode(double num) {

   ListNode *newNode;  // To point to the new node

   // Create a new node and store the data in it
   newNode = new ListNode;
   newNode->value = num;
   newNode->next = NULL;

   // If empty, make head point to new node
   if (head==NULL)
      head = newNode;

   else {
      ListNode *p;  // To move through the list
      p = head;     // initialize to start of list

      // traverse list to find last node
      while (p->next)          //it's not last
         p = p->next;          //make it pt to next

      // now p pts to last node
      // make last node point to newNode
      p->next = newNode;
   }
}
```


Driver to demo NumberList

- ListDriver.cpp version 1 (no output)

ListDriver.cpp

```
#include "NumberList.h"

int main() {

   // Define the list
   NumberList list;

   // Append some values to the list
   list.appendNode(2.5);
   list.appendNode(7.9);
   list.appendNode(12.6);

}
```

Traversing a Linked List

- Visit each node in a linked list, to
  - display contents, sum data, test data, etc.
- Basic process:

```
set a pointer to point to what head points to
while pointer is not NULL
  process data of current node
  go to the next node by setting the pointer to
     the pointer field of the current node
end while
```


Operation: **display** the list

in NumberList.cpp

```
void NumberList::displayList() {

   ListNode *p;  //ptr to traverse the list

   // start p at the head of the list
   p = head;

   // while p pts to something (not NULL), continue
   while (p)
   {
      //Display the value in the current node
      cout << p->value << " ";

      //Move to the next node
      p = p->next;
   }
   cout << endl;
}
```


Driver to demo NumberList

- ListDriver.cpp version 2

ListDriver.cpp

```
#include "NumberList.h"

int main() {

   // Define the list
   NumberList list;

   // Append some values to the list
   list.appendNode(2.5);
   list.appendNode(7.9);
   list.appendNode(12.6);

   // Display the values in the list.
   list.displayList();
}
```

Output:
2.5 7.9 12.6

Operation: destroy a List

- The destructor must “delete” (deallocate) all nodes used in the list
- To do this, use list traversal to visit each node
- ~NumberList: what’s wrong with this definition?

```
NumberList::~NumberList() {

   ListNode *p;   // traversal ptr
   p = head;      //start at head of list

   while (p) {

      delete p;     // delete current
      p = p->next;  // advance ptr
   }
}
```

destructor

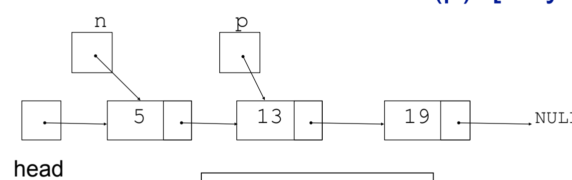
- You need to save p->next before deleting p:

in NumberList.cpp

```
NumberList::~NumberList() {

   ListNode *p;    // traversal ptr
   ListNode *n;    // saves the next node

   p = head;       //start at head of list

   while (p) {

      n = p->next;   // save the next
      delete p;      // delete current
      p = n;         // advance ptr
   }
}
```


Operation:
delete a node from the list

- deleteNode: removes node from list, and deletes (deallocates) the removed node.
- Requires two extra pointers:
  - one to point to the node <u>before</u> the node to be deleted. (n) [why?]
  - one to point to the node to be deleted (p) [why?]

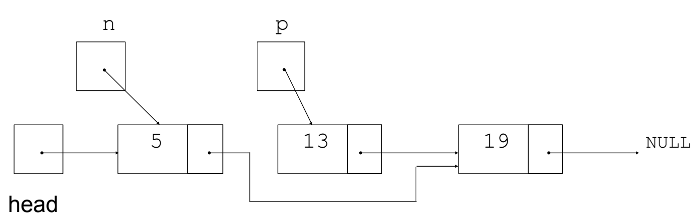


Deleting 13 from the list

Deleting a node

- Change the pointer of the previous node to point to the node <u>after</u> the one to be deleted.

```
n->next = p->next;
```

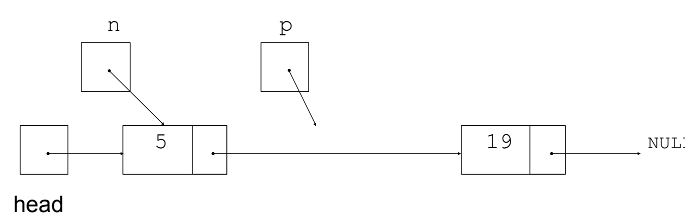


- Now just “delete” the p node

Deleting a node

- After the node is deleted:

```
delete p;
```



deleteNode code

in NumberList.cpp

```
void NumberList::deleteNode(double num) {

    ListNode *p = head;    // to traverse the list
    ListNode *n;           // trailing node pointer

    // skip nodes not equal to num, stop at last
    while (p && p->value!=num) {
       n = p;          // save it!
       p = p->next;  // advance it
    }

    // p not null: num was found, set links + delete
    if (p) {
        if (p==head) {   // p points to the first elem.
            head = p->next;
            delete p;
        } else {         // n points to the predecessor
            n->next = p->next;
            delete p;
       }
    }
```


Driver to demo NumberList

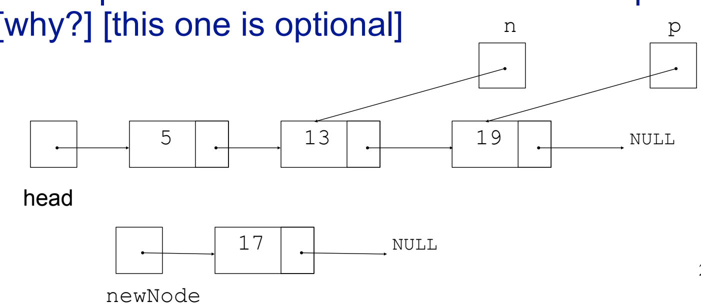
in ListDriver.cpp

```
// set up the list
NumberList list;
list.appendNode(2.5);
list.appendNode(7.9);
list.appendNode(12.6);
list.displayList();

cout << endl << "remove 7.9:" << endl;
list.deleteNode(7.9);
list.displayList();

cout << endl << "remove 8.9: " << endl;
list.deleteNode(8.9);
list.displayList();

cout << endl << "remove 2.5: " << endl;
list.deleteNode(2.5);
list.displayList();
```

```
Output:
2.5   7.9   12.6

remove 7.9:
2.5   12.6

remove 8.9:
2.5   12.6

remove 2.5:
12.6
```


Operation: **insert** a node into a linked list

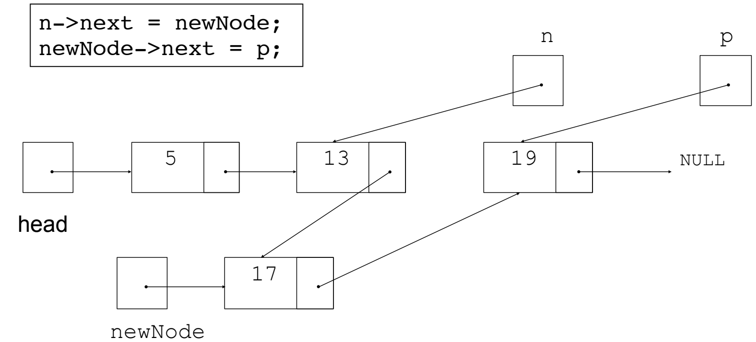
- Inserts a new node into the middle of a list.
- Uses two extra pointers:
  - one to point to node before the insertion point [why?]
  - one to point to the node after the insertion point [why?] [this one is optional]



Inserting a Node into a Linked List

- Insertion completed:

```
n->next = newNode;
newNode->next = p;
```



Insert Node Algorithm

How do you determine the insertion point?

- Maintain sorted list: the insertion point is immediately before the first node in the list that has a value greater than the value being inserted. **We do this.**
- Insert by position: InsertNode takes a second argument that is the index of a node. Insert new value before (or after) that node.
- Use a cursor: The list class has a member variable that is a pointer to a "current node", insert new value before (or after) the current node.

insertNode code

in NumberList.cpp

```
void NumberList::insertNode(double num) {
   ListNode *newNode;   // ptr to new node
   ListNode *p;         // ptr to traverse list
   ListNode *n;         // node previous to p

   //allocate new node
   newNode = new ListNode;
   newNode->value = num;

   // skip all nodes less than num
   p = head;
   while (p && p->value < num) {
     n = p;         // save
     p = p->next;   // advance
   }

   if (p == head) {       //insert before first
     head = newNode;
     newNode->next = p;
   }
   else {                 //insert after n
     n->next = newNode;
     newNode->next = p;
   }
}
```


Driver to demo NumberList

in ListDriver.cpp

```
int main() {

   // set up the list
   NumberList list;
   list.appendNode(2.5);
   list.appendNode(7.9);
   list.appendNode(12.6);
   list.displayList();

   list.insertNode (8.5);
   list.displayList();

   list.insertNode (1.5);
   list.displayList();

   list.insertNode (21.5);
   list.displayList();

}
```

```
Output:
2.5   7.9   12.6
2.5   7.9   8.5   12.6
1.5   2.5   7.9   8.5   12.6
1.5   2.5   7.9   8.5   12.6   21.5
```


Advantages of linked lists over arrays

- A linked list can easily grow or shrink in size.
  - Nodes are created in memory as they are needed.
  - The programmer doesn't need to predict how many elements will be in the list.
- The amount of memory used to store the list is always proportional to the number of elements in the list.
  - For arrays, the amount of memory used is often much more than is required by the actual elements in the list.
- When a node is inserted into or deleted from a linked list, none of the other nodes have to be moved.

Advantages of arrays over linked lists

- Arrays allow random access to elements: array[i]
  - linked lists allow only sequential access to elements (must traverse list to get to i'th element).
- Arrays do not require extra storage for "links"
  - linked lists are impractical for lists of characters or booleans (pointer value is bigger than data value).

Exercise: find four errors

```
int main() {
   struct node {
      int data;
      node * next;
   }

   // create empty list
   node * list;

   // insert six nodes at front of list
   node *n;
   for (int i=0;i<=5;i++) {
      n = new node;
      n->data = i;
      n->next = list;
   }

   // print list
   n = list;
   while (!n) {
      cout << n->data << "  ";
      n = n->next;
   }
   cout << endl;
}
```