Linked Lists

Week 8

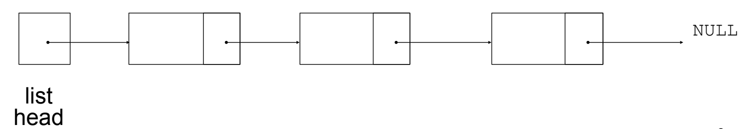
Gaddis: Chapter 17

CS 5301
Fall 2014

Jill Seaman

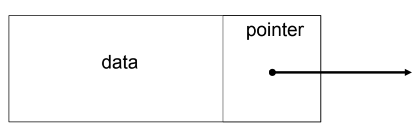
Introduction to Linked Lists

- A data structure representing a list
- A series of **dynamically allocated** 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)



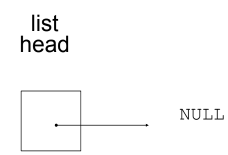
Node Organization

- Each node contains:
  - 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 Node data type

- Use a struct for the node type

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

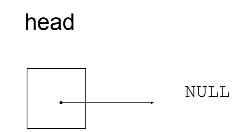
- (this is just a data type, no variables declared)
- `next` can hold the address of a `ListNode`.
  - it can also be NULL
  - “self-referential data structure”

Defining the Linked List variable

- Define a pointer for the head of the list:

```
ListNode *head = NULL;
```

- 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
- Use ONE of the following for NULL:

```
#include <iostream>
#include <cstddef>
```

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

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

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

- in C++11 you may use `nullptr`

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
  - **copy** constructor

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
         struct ListNode *next;  // ptr to next node
      };
      ListNode *head;    // the list head

   public:
      NumberList();
      NumberList(const NumberList & src);
      ~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

When defining list operations, always consider special cases:
- Empty list
- First element, front of the list (when head pointer is involved)

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++

```
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;
   }
}
```

in NumberList.cpp

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

```
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;
}
```

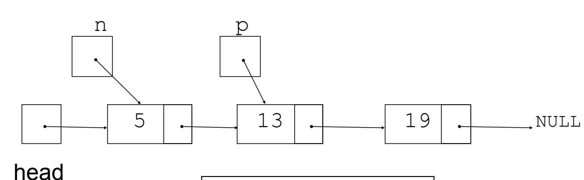
in NumberList.cpp

Or the short version:

```
void NumberList::displayList() {
   ListNode *nodePtr;
   for (nodePtr = head; nodePtr; nodePtr = nodePtr->next)
      cout << nodePtr->value << endl;
}
```


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 to be deleted
  - one to point to the node <u>before</u> the node to be deleted.

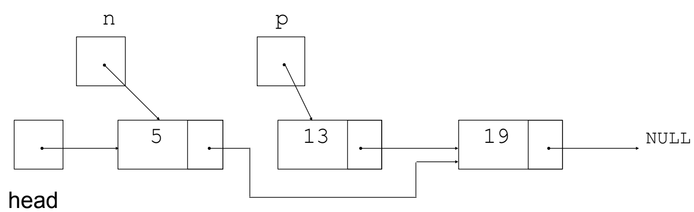


Deleting 13 from the list

Deleting a node

- Change the pointer of the previous node to point to the node after the one to be deleted.

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

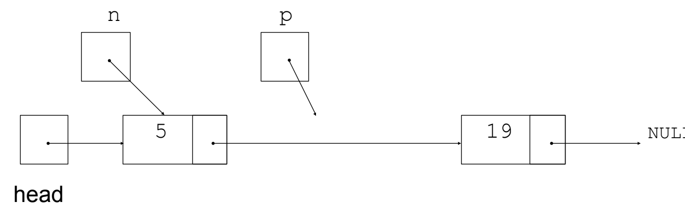


- Now just "delete" the p node

Deleting a node

- After the node is deleted:

```
delete p;
```



Delete Node Algorithm

- Delete the node containing num

use p to traverse the list, until it points to num or NULL
--as p is advancing, make n point to the node before it

```
if (p is not NULL)  //found!
  if (p==head)      //it's the first node, and n is garbage
     make head point to the second element
     delete p's node (the first node)
  else
     make n's node point to what p's node points to
     delete p's node
else: . . . p is NULL, not found do nothing
```


Linked List functions: deleteNode

in NumberList.cpp

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

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

    // 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 is found, set links + delete
    if (p) {
        if (p==head) {    // p points to the first elem, n is garb
            head = p->next;
            delete p;
        } else {          // n points to the predecessor
            n->next = p->next;
            delete p;
        }
    }
}
```

Destroying a Linked List

- The destructor must “delete” (deallocate) all nodes used in the list
- To do this, use list traversal to visit each node
- For each node,
  - save the address of the next node in a pointer
  - delete the node

destructor

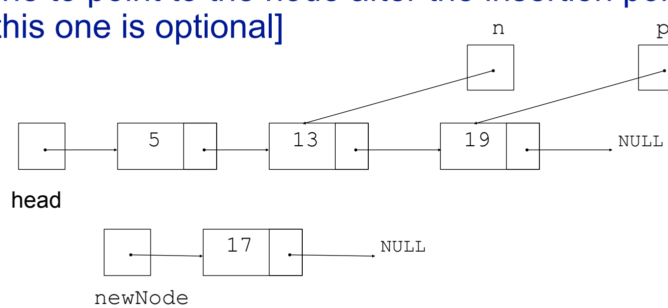
- ~NumberList: deallocates all the nodes

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

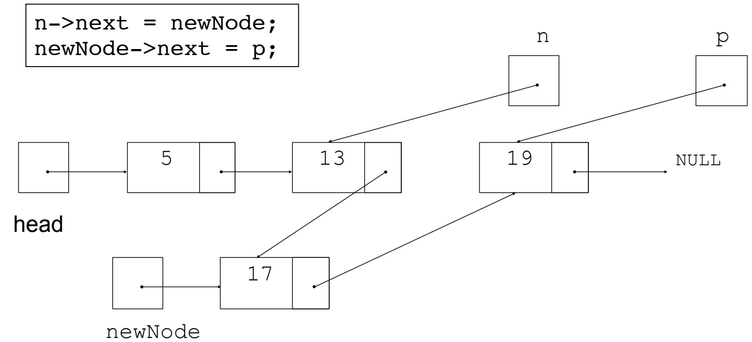
- Inserts a new node into the middle of a list.
- Uses two extra pointers:
  - one to point to node before the insertion point
  - one to point to the node after the insertion point [this one is optional]



Inserting a Node into a Linked List

- Insertion completed:

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



Insert Node Algorithm

- Insert node in a certain position

Create the new node, store the data in it
Use pointer p to traverse the list,
    until it points to: node after insertion point or NULL
    --as p is advancing, make n point to the node before
  if p points to first node (p is head, n was not set)
    make head point to new node
    make new node point to p's node
  else
    make n's node point to new node
    make new node point to p's node

Note: we will assume our list is sorted, so the insertion point is immediately before the first node that is larger than the number being inserted.

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;
   }
}
```

What if num is bigger than all items in the list?

Operation: **copy constructor**

- Can't copy src.head to head (then the lists would share same nodes)

in NumberList.cpp

```
NumberList::NumberList(const NumberList & src) {

    head = NULL;          // initialize empty list

    // traverse src list, append its values to end of this list
    ListNode *p;

    for (p=src.head; p; p=p->next)
    {
        appendNode(p->value);     //calls the member function
    }

}
```


Driver to demo NumberList

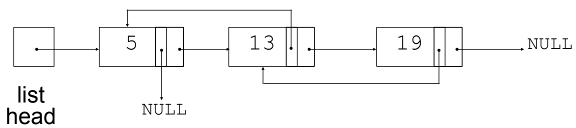
in ListDriver.cpp

```
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.deleteNode (2.5);
   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   7.9   8.5   12.6
```

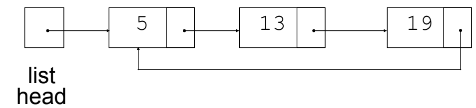
Linked List variations

- Doubly linked list
  - each node has two pointers, one to the next node and one to the previous node
  - head points to first element, tail points to last.
  - can traverse list in reverse direction by starting at the tail and using `p=p->prev`.



Linked List variations

- Circular linked list
  - last cell's next pointer points to the first element.



Advantages of linked lists
(over arrays)

- A linked list can easily grow or shrink in size.
  - The programmer doesn't need to predict how many values could be in the list.
  - The programmer doesn't need to resize (copy) the list when it reaches a certain capacity.
- When a value 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).