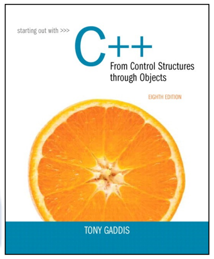
Linked Lists

Ch 17 (Gaddis*)

CS 3358
Spring 2015

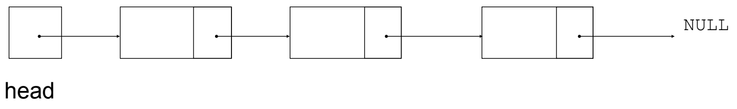
Jill Seaman



*Tony Gaddis, Starting out with C++: From Control Structures through Objects, Addison-Wesley, 8th Edition, 2014. ISBN: 0133769399

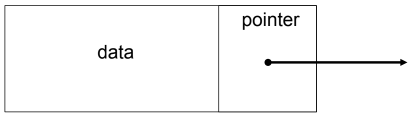
Introduction to Linked Lists

- 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)



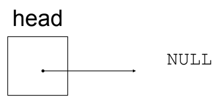
Node Organization

- 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;
};
```

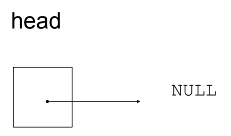
- (this is just a data type, no variables declared)
- `next` can hold the address of a `ListNode`.
  - it can also be NULL
  - “self-referential data structure”

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 (and **operator=**)

Linked List class declaration

- See NumberList.h for the NumberList class decl.
  - contains definition of ListNode and head pointer
  - contains prototypes for all operations on previous slide.
- For each function/operation (to implement it):
  - draw pictures of applying operation to sample lists, look for special cases, etc.
  - write an algorithm in English based on pictures
  - translate the algorithm to code

Operation: **Create** the empty list

- Constructor: sets up empty list
- Add code to NumberList() in NumberList.cpp

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++

appendNode implementation:

- Add code to appendNode() in NumberList.cpp, based on last 2 slides

Traversing a Linked List

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

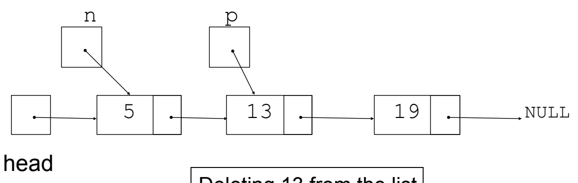
```
set a pointer to point to what head points to
while pointer is not NULL
  process data of current node
  go to the next node by setting the pointer to
     the pointer field of the current node
end while
```

Operation: **display** the list

- Add code to displayList() in NumberList.cpp, based on previous slide
- Then use ListDriver.cpp to test + demo the list.

Operation: **delete** a node from the list

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

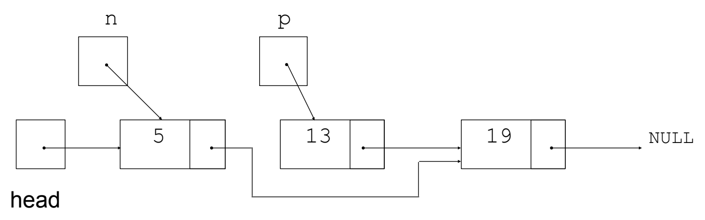


Deleting 13 from the list

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;
```

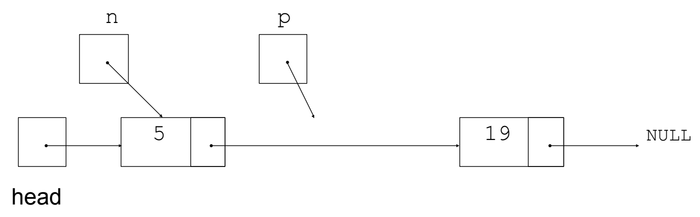


- Now just "delete" the p node

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

- Add code to deleteNode() in NumberList.cpp, based on previous slide
- Then use ListDriver.cpp to test + demo the list.

Destroying a Linked List

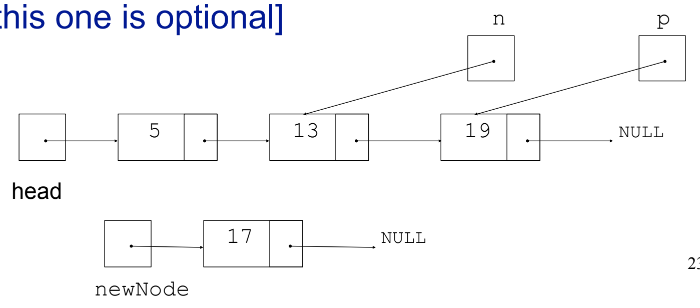
- The destructor must “delete” (deallocate) all nodes used in the list
- To do this, use list traversal to visit each node
- For each node,
  - save the address of the next node in a pointer
  - delete the node
  - advance the ptr

Linked List functions: destructor

- Add code to ~NumberList() in NumberList.cpp, based on previous slide
  - copy paste from displayList()
- Then use ListDriver.cpp to test + demo the list.
  - for testing, add cout before deleting p

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

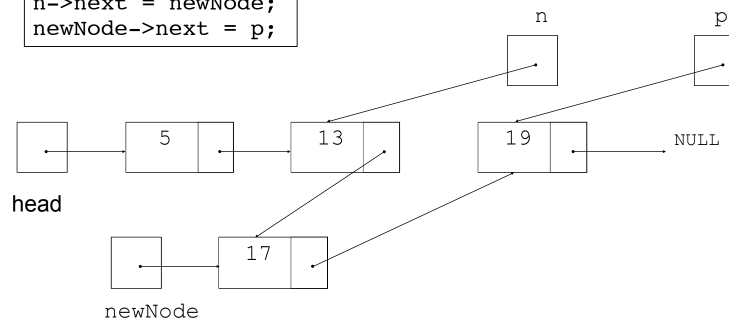
- Inserts a new node into the middle of a list.
- Uses two extra pointers:
  - one to point to node before the insertion point
  - one to point to the node after the insertion point [this one is optional]



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 implementation

- Add code to insertNode() in NumberList.cpp, based on previous slide.
- Then use ListDriver.cpp to test + demo the list.
  - inserting into the middle of the list (general case only)

Operation: **copy constructor**

- Pointers + dynamic allocation => deep copy
- Don't copy any pointers (allocate new memory)
- Iterate over src list, append nodes to new list.

```
Initialize head to NULL
For each item in the src list (in order)
  append item.value to this list
```

copy constructor: implementation

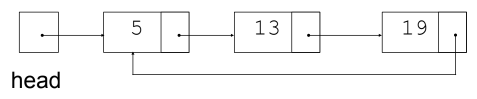
- Add code to copy constructor in NumberList.cpp, based on previous slide
- Then use ListDriver.cpp to test + demo the list.
- Try operator= ?
- All the code for the NumberList demo will be on the class website.

Chapter 17 in Weiss book

- Elegant implementation of linked lists
- It uses a “header node”
  - empty node, immediately before the first node in the list, not visible to users of the class
  - eliminates need for most special cases
  - internal traversals must skip that node
- It uses three separate classes (w/ friend access)
- It implements an iterator
- It uses templates
- We’ll look at it after we cover these new topics.

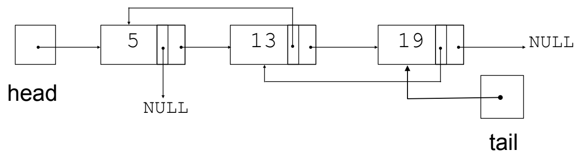
Linked List variations

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



Linked List variations

- Doubly linked list
  - each node has **two** pointers, one to the next node (next) and one to the previous node (prev)
  - 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`.



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).