Ch. 17: Linked Lists
Part 2

CS 2308
Fall 2011

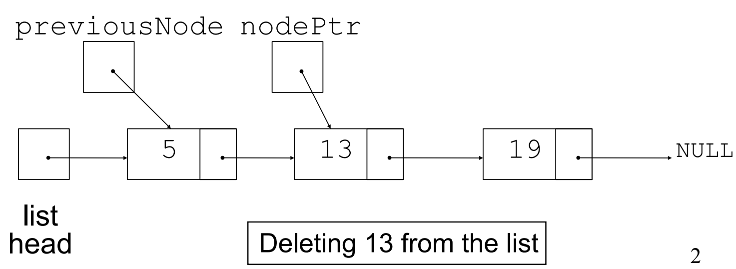
Jill Seaman

Lecture 17

Using content from textbook slides: Starting Out with C++, Gaddis, Pearson/Addison-Wesley

Deleting a Node from a Linked 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.

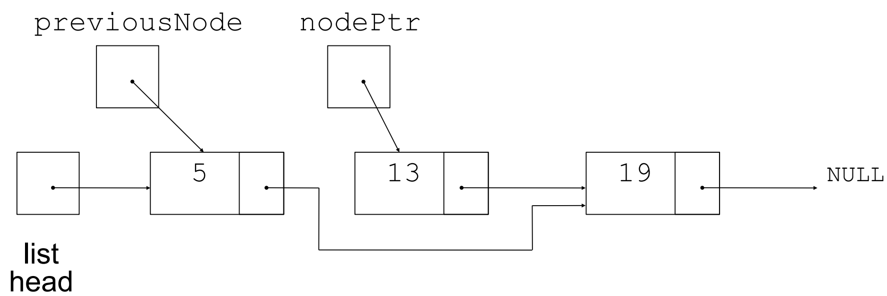


Deleting 13 from the list

Deleting a node

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

```
previousNode->next = nodePtr->next;
```

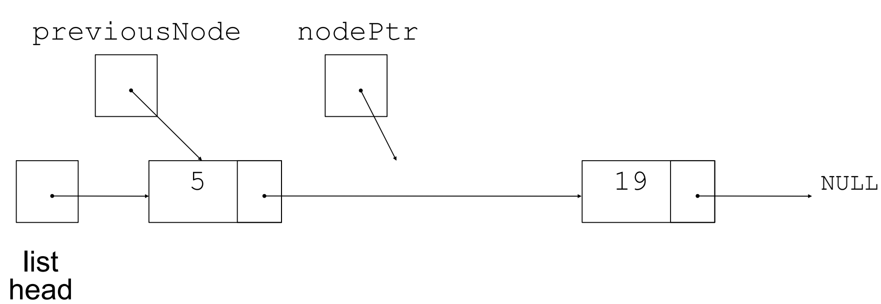


- Now just "delete" the nodePtr node

Deleting a node

- After the node is deleted:

```
delete nodePtr;
```



Delete Node Algorithm

- Delete the node containing num

If list is empty, exit
If first node is num
  make p point to first node
  make head point to second node
  delete p
else
  use p to traverse the list, until it points to num or NULL
  --as p is advancing, make n point to the node before
  if (p is not NULL)
    make n's node point to what p's node points to
    delete p's node

Linked List functions: deleteNode

- deleteNode: removes num from list

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

   // empty list, exit
   if (!head)
      return;

   ListNode *nodePtr;      // to traverse the list

   // check if first node is num
   if (head->value == num) {
      nodePtr = head;
      head = nodePtr->next;
      delete nodePtr;
   }

   // else is continued on next slide
```

Linked List functions: deleteNode

- deleteNode: cont.

```
else {
   ListNode *previousNode;  // trailing node pointer

   // initialize traversal ptr to first node
   nodePtr = head;

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

   // nodePtr not null: num is found, set links + delete
   if (nodePtr) {
      previousNode->next = nodePtr->next;
      delete nodePtr;
   }
   // else: end of list, num not found in list
}
}
```


Driver to demo NumberList

- 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
```

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

Linked List functions: destructor

- ~NumberList: deallocates all the remaining nodes

```
NumberList::~NumberList() {

   ListNode *nodePtr;   // traversal ptr
   ListNode *nextNode;  // saves the next node

   nodePtr = head;      //start at head of list

   while (nodePtr) {

      nextNode = nodePtr->next;  // save the next
      delete nodePtr;            // delete current
      nodePtr = nextNode;        // advance ptr
   }

}
```

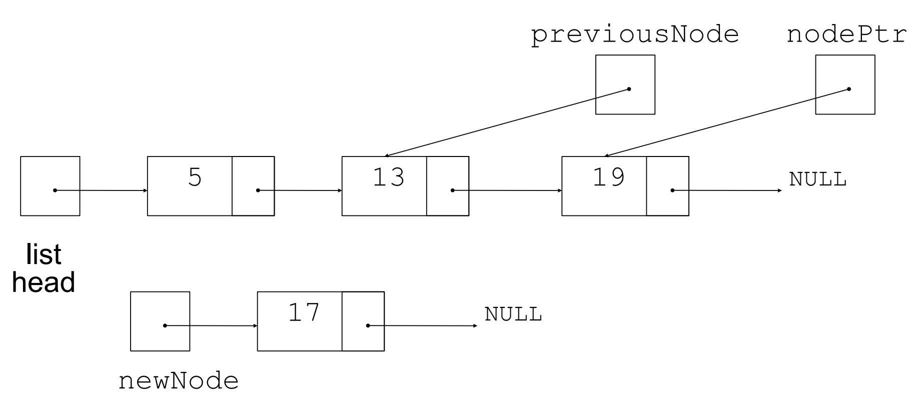
Inserting a Node into a Linked List

- Requires two pointers to traverse the list:
  - pointer to point to the node after the insertion point
  - pointer to point to node before point of insertion
- New node is inserted between the nodes pointed at by these pointers

- The before and after pointers move in tandem as the list is traversed to find the insertion point
  - Like delete

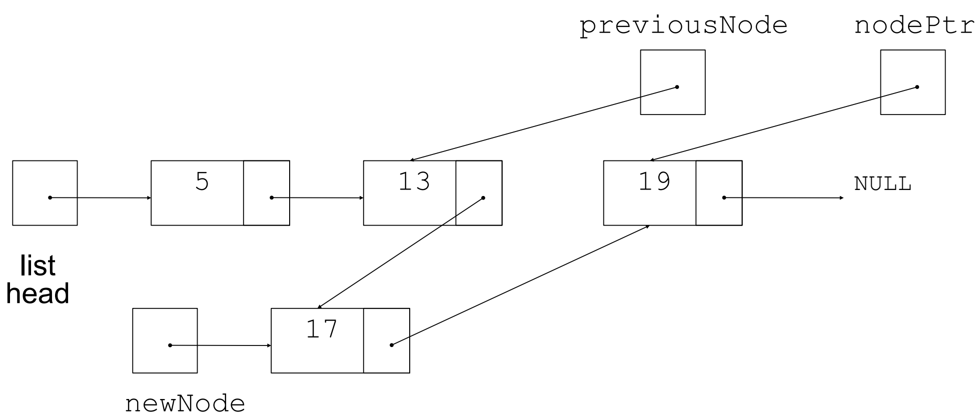
Inserting a Node into a Linked List

- New node created, new position located:



Inserting a Node into a Linked List

- Insertion completed:



Insert Node Algorithm

- Insert node in a certain position

Create the new node, store the data in it
If list is empty,
  make head point to new node, new node to null
else
  use 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 (n is null)
    make head point to new node
    new node to p's node
  else
    make n's node point to new node
    make new node point to p's node

Insert Node Algorithm

- Note that in the insertNode implementation that follows, the insertion point is immediately before the first node in the list that has a value greater than the value being inserted.
- This works very nicely if the list is already sorted and you want to maintain the sort order.
- Another way to specify the insertion point is to have insertNode take a second argument that is the index of the node after the insertion point.
- In this case you can use a count-controlled loop to advance the pointer(s) through the list.

Linked List functions: insertNode

- insertNode: inserts num into middle of list

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

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

   // empty list, insert at front
   if (!head) {
      head = newNode;
      newNode->next = NULL;
   }

   //else is on the next slide . . .
```

Linked List functions: insertNode

- insertNode: inserts num into middle of list

```
    else {
       // initialize the two traversal ptrs
       nodePtr = head;
       previousNode = NULL;

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

       if (previousNode == NULL) { //insert before first
          head = newNode;
          newNode->next = nodePtr;
       }
       else {                      //insert after previousNode
          previousNode->next = newNode;
          newNode->next = nodePtr;
       }
    }
}
```


Driver to demo NumberList

- ListDriver.cpp

```
int main() {

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

   list.insertNode(10.5);

   list.displayList();

   return 0;
}
```

```
Output:
2.5
7.9
10.5
12.6
```

Advantages of linked lists over arrays

- A linked list can easily grow or shrink in size.
  - The programmer doesn't need to know how many nodes will be in the list.
  - Nodes are simply created in memory as they are needed.
- 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], while linked lists allow only sequential access to elements (must traverse list to get to i'th element).
- Another disadvantage of linked lists is the extra storage needed for references. This makes them 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;
   return 0;
}
```