Linked Lists

Week 8

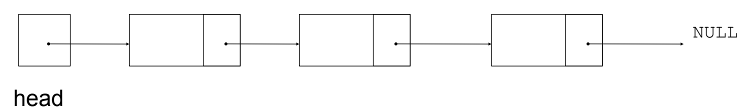
Gaddis: Chapter 17

CS 5301
Fall 2016

Jill Seaman

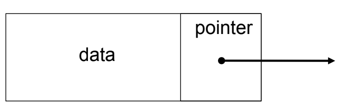
Introduction to Linked Lists

- 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 a structure, an object, etc.
  - a pointer – that can point to another node



- We use a struct to define the node type:

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

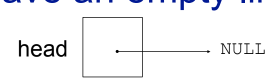
- `next` can hold the address of a `ListNode`.
  - it can also be NULL

Defining the Linked List variable

- Define a pointer for the head of the list:

```
ListNode *head = NULL;   //NULL specifies end of list
```

- Now we have an empty linked list:



- NULL is equivalent to address 0
- to test a pointer for NULL (these are equivalent):

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

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

Note: If you try to dereference a pointer whose value is NULL, you will get a runtime error.   For example: `head->next`   Check before you do this.

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() = { head = NULL; } //create empty list
     ~NumberList();

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


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 has a NEXT node
     make p point to that node (the NEXT node of the node p currently points 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;
   }
}
```


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 next 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;
}
```

Or the short version:

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


Destroying a Linked List: destructor

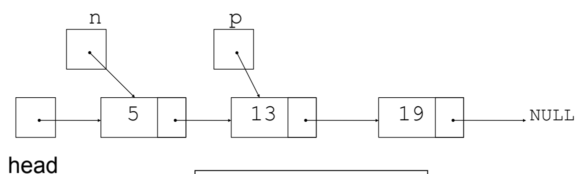
- The destructor must "delete" (deallocate) all nodes used in the list
- To do this, use list traversal to visit each node:
  - save the address of the next node in a pointer
  - delete the node

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

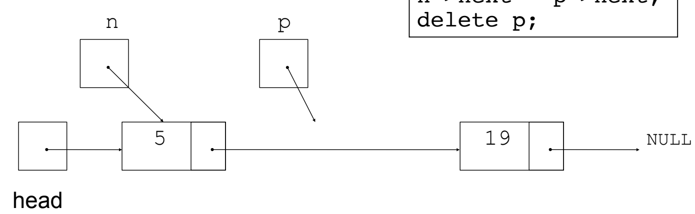


Deleting 13 from the list

Deleting a node

- Change the pointer of the previous node to point to the node <u>after</u> the one to be deleted.
- Then just "delete" the p node

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

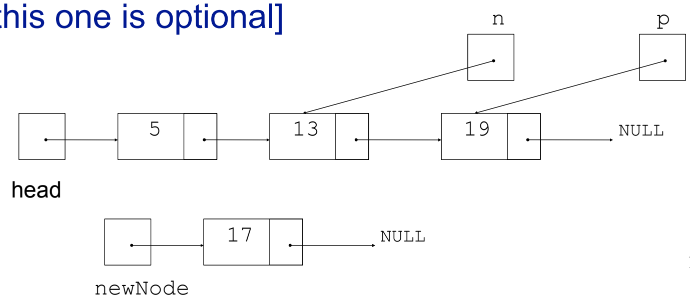

Linked List functions: deleteNode

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;
        }
    }
}
```

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

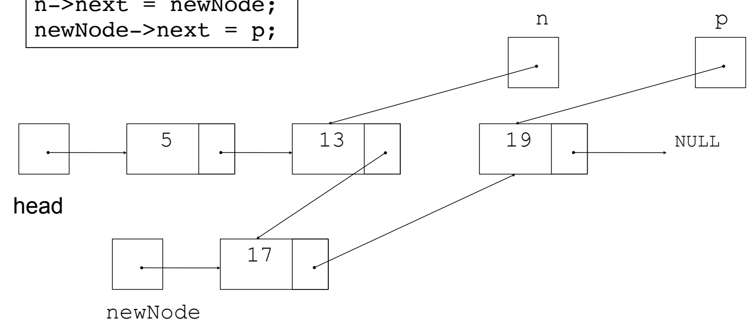
- Inserts a new node into the middle of a list.
- Uses two extra pointers:
  - one to point to node before the insertion point
  - one to point to the node after the insertion point [this one is optional]



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

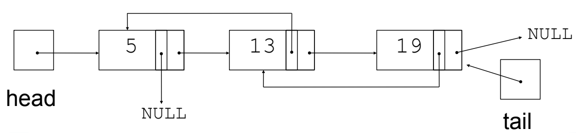
```
void NumberList::insertNode(double num) {
   ListNode *newNode;   // ptr to new node
   ListNode *p;         // ptr to traverse list
   ListNode *n;         // node previous to p

   //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, or empty list
     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?

Linked List variations

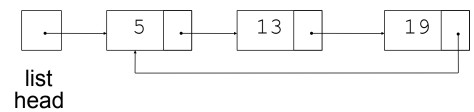
- Doubly linked list
  - each node has two pointers, one to the next node and one to the previous node
  - head points to first element, tail points to last.

```
private:
    // Structure for nodes
    struct Node {
        int value;      // Value in the node
        Node *prev;     // Pointer to the previous node
        Node *next;     // Pointer to the next node
    };
    Node *head;        // Pointer to the first element
    Node *tail;         // Pointer to the last element
```



Linked List variations

- Circular linked list
  - last cell's next pointer points to the first element.
  - no null pointers
  - every node has a successor



Linked lists vs Arrays
(pros and cons)

- A <u>linked list</u> can easily grow or shrink in size.
  - No maximum capacity required
  - No need to resize+copy when list reaches max size.
- When a value is inserted into or deleted from a <u>linked list</u>, no other nodes have to be moved.
- <u>Arrays</u> allow random access to elements: array[i] (linked lists require traversal to get i'th element).
- <u>Arrays</u> do not require extra storage for "links" (linked lists are impractical when the pointer value is bigger than data value).