Linked Lists

Ch 17 (Gaddis)

Week 8

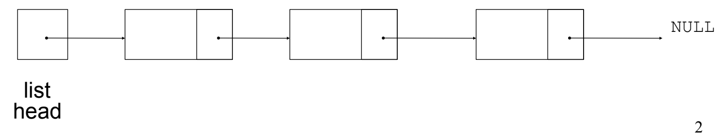
CS 5301

Summer I 2012

Jill Seaman

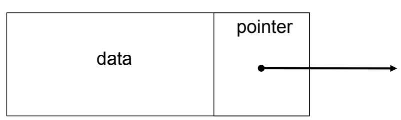
Introduction to Linked Lists

- 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)



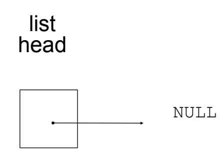
Node Organization

- 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;
};
```

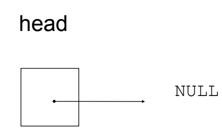
- (this is just a data type, no variables declared)
- `next` can hold the address of a `ListNode`.
  - it can also be NULL
  - “self-referential data structure”

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) ...
```


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

```
// file 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();
};
```


Linked List functions: constructor

- Constructor: sets up empty list

```
// file NumberList.cpp

#include "NumberList.h"


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


Linked List functions: appendNode

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

  Create a new node and store the data in it
  If the list is 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)

Linked List functions: appendNode

- 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++

Linked List functions: appendNode

```
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)
      head = newNode;

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

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

      // now nodePtr pts to last node
      // make last node point to newNode
      nodePtr->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 pointer field of the current node
end while
```


Linked List functions: displayList

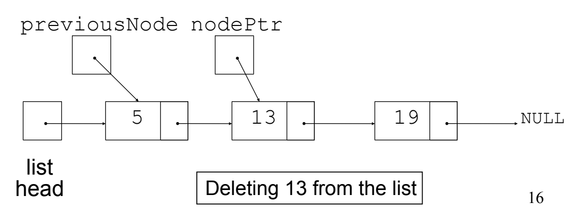
```
void NumberList::displayList() {
   ListNode *nodePtr;  //ptr to traverse the list

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

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

      //Move to the next node
      nodePtr = nodePtr->next;
   }
}
```

Or the short version:

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


Deleting a Node from a Linked List

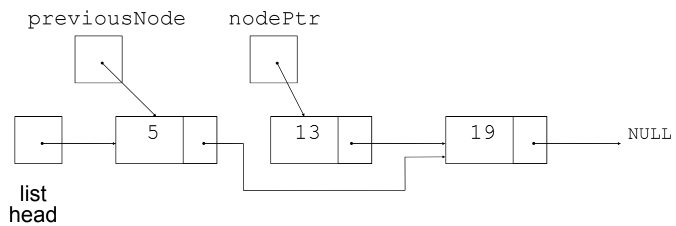
- deleteNode: removes node from list, and deletes (deallocates) the removed node.
- Requires two pointers:
  - one to point to the node to be deleted
  - one to point to the node <u>before</u> the node to be deleted.



Deleting a node

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

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

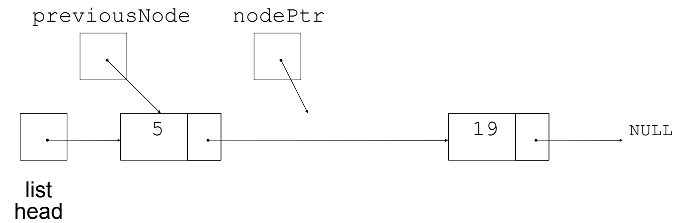


- Now just “delete” the nodePtr node

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 contains 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 it
  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

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

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

   ListNode *nodePtr;           // to traverse the list
   if (head->value == num) {   // if first node contains num
      nodePtr = head;
      head = nodePtr->next;
      delete nodePtr;
   }
   else {
      ListNode *previousNode;  // trailing node pointer
      nodePtr = head;       // traversal ptr, set to first node

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

      if (nodePtr) {       // num is found, set links + delete
         previousNode->next = nodePtr->next;
         delete nodePtr;
      }
      // else: end of list, num not found in list, do nothing
   }
}
```

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 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
   }

}
```

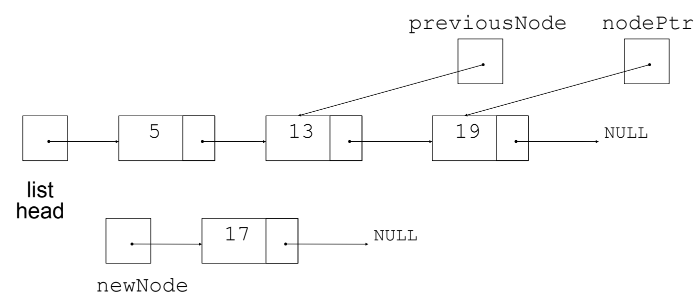

Inserting a Node into a Linked List

- Requires two pointers:
  - 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

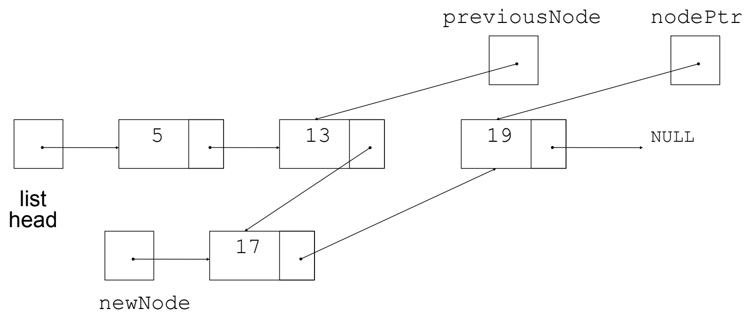
Inserting a Node into a Linked List

- 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

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

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

      // skip all nodes less than num (list is sorted)
      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;
      }
   }
}
```

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

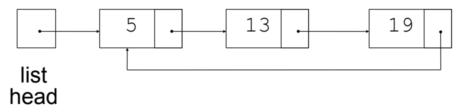
Linked List functions: copy constructor

- Can't copy src.head to head (share same nodes)

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

    head = NULL;          // initialize empty list

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

    for (nodePtr=src.head; nodePtr; nodePtr=nodePtr->next)
    {
        appendNode(nodePtr->value);
    }

}
```

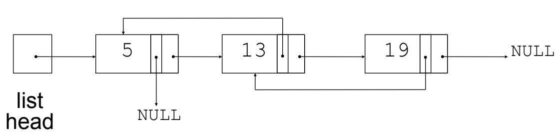

Linked List variations

- 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 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`.



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).