Linked Lists

Unit 5

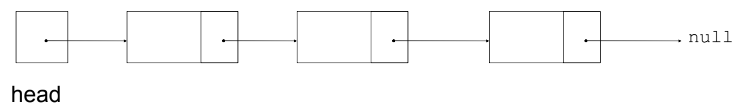
Chapter 17

CS 2308
Fall 2016

Jill Seaman

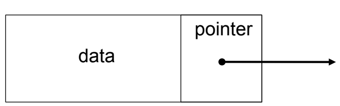
17.1 Introduction to Linked Lists

- 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 null (address 0)



Node Organization

- Each node contains:
  - data members – contain the elements' values.
  - 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

Using `NULL` (or `nullptr`)

- Equivalent to address 0
- Used to specify end of the list
- In C++11, you can use `nullptr` instead of `NULL`
- `NULL` is defined in the cstddef header.
- to test a pointer p for NULL, these are equivalent:

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

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

- Note: Do NOT dereference a NULL ptr!

```
ListNode *p = NULL;
cout << p->value;    //crash! null pointer exception
```

Linked Lists: Tasks

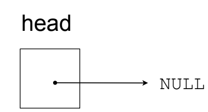
We will implement the following tasks on a linked list:

- T1: Create an empty list
- T2: Create a new node
- T3: Add a new node to front of list (given newNode)
- T4: Traverse the list (and output)
- T5: Find the last node (of a non-empty list)
- T6: Find the node containing a certain value
- T7: Find a node AND it's previous neighbor.
- T8: Append to the end of a non-empty list
- T9: Delete the first node
- T10: Delete an element, given p and n
- T11: Insert a new element, given p and n

T1:Create an empty list

- let's make the empty list

```
struct ListNode      // the node data type
{
   double value;     // data
   ListNode *next;   // ptr to next node
};

int main() {

   ListNode *head = NULL;      // the empty list

}
```

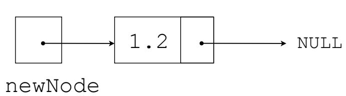


T2:Create a new node:

- let's make a new node:

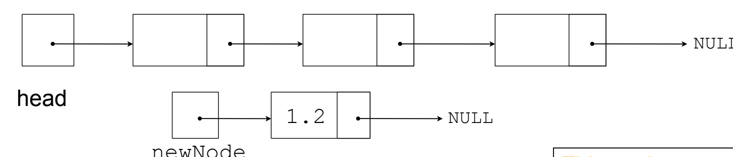
```
ListNode *newNode = new ListNode();
newNode->value = 1.2;
newNode->next = NULL;
```

- It's not attached to the list yet.



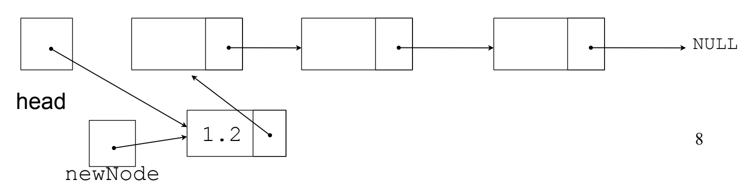
T3:Add new node to front of list:

- make newNode's next point to the first element.
- then make head point to newNode.




```
newNode->next = head;
head = newNode;
```

This works even if head is NULL, try it.



T4:Traverse the list

(and output all the elements)

- let's output a list of two elements:

```
cout << head->value << " " << head->next->value << endl;
```

- now using a temporary pointer to point to each node:

```
ListNode *p;   //temporary pointer (don't use head for this)
p = head;      //p points to the first node
cout << p->value << " ";
p = p->next;   //makes p point to the 2nd node (draw it!)
cout << p->value << endl;
p = p->next;   //what does p point to now?
```

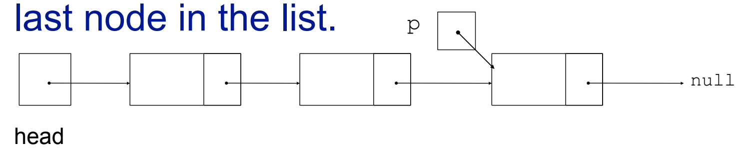
- now let's rewrite that as a loop:

```
ListNode *p;   //temporary pointer (don't use head for this)
p = head;      //p points to the first node

while (p!=NULL) {
  cout << p->value << " ";
  p = p->next;   //makes p point to the next node
}
```


T5:Find the last node

(of a non-empty list)

- Goal: make a temporary pointer, p, point to the last node in the list.

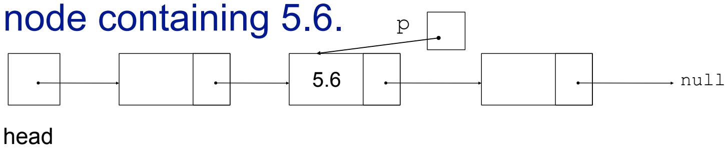


- Make p point to the first node. Then:
  - do p=p->next until p points to the last node.
  - in the last node, next is null.
  - so stop when p->next is null.

```
ListNode *p=head;      //p points to what head points to

while (p->next!=NULL) {
  p = p->next;         //makes p point to the next node
}
```


T6:Find the node containing a certain value

- Goal: make a temporary pointer, p, point to the node containing 5.6.

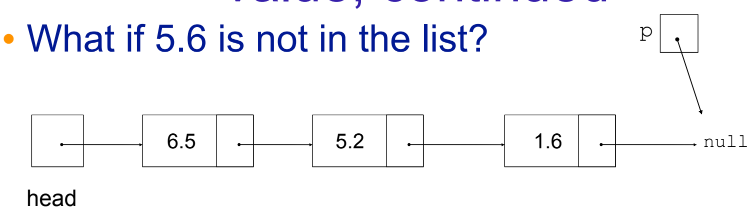


- Make p point to the first node. Then:
  - do p=p->next until p points to the node with 5.6.
  - so stop when p->value is 5.6.

```
ListNode *p=head;      //p points to what head points to

while (p->value!=5.6) {
  p = p->next;         //makes p point to the next node
}
```


Find the node containing a certain value, continued

- What if 5.6 is not in the list?

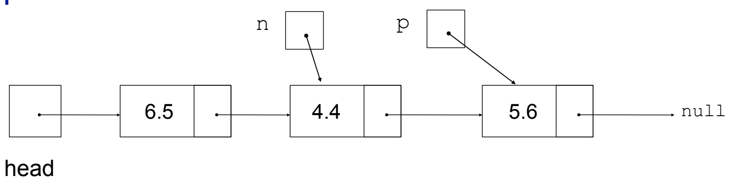


- If 5.6 is not in the list, the loop will not stop. p will eventually be NULL, and evaluating p->value in the condition will crash.
- So let's make the loop stop if p becomes NULL.

```
ListNode *p=head;      //p points to what head points to

while (p!=NULL && p->value!=5.6) {
  p = p->next;         //makes p point to the next node
}
```

T7:Find a node AND it's previous neighbor.

- sometimes we need to track the current **and** the previous node:



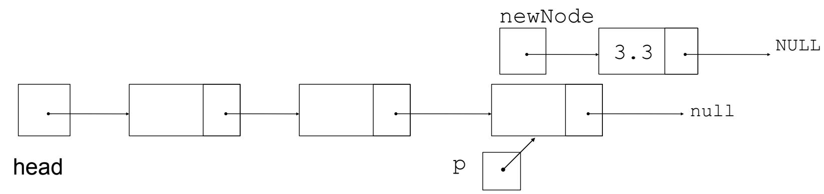

```
ListNode *p = head;  //current node, set to first node
ListNode *n = NULL;  //previous node, none yet
while (p!=NULL && p->value!=5.6) {
   n = p;          //save current node address
   p = p->next;    //advance current node to next one
}
```


T8:Append to the end of a non-empty list

- Create a new node, and find the last node:

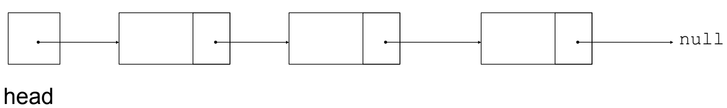
```
ListNode *newNode = new ListNode();
newNode->value = 3.3;
newNode->next = NULL;

ListNode *p=head;
while (p->next!=NULL) {
  p = p->next;
}
```

We've done this already.



- Now make the last node's next point to newNode. `p->next = newNode;`

T9:Delete the first node

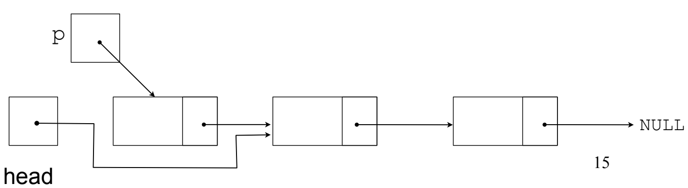


- delete the first element of a non-empty list

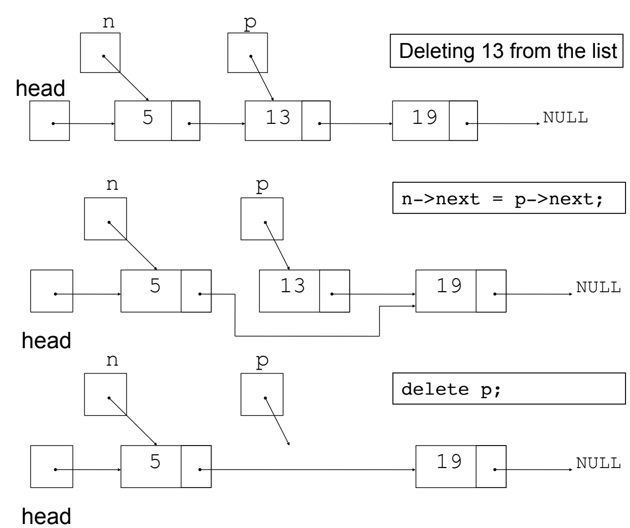
```
head = head->next;
```

- what about deallocating the first node? Oops.

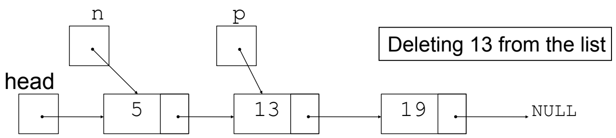
```
ListNode *p = head;
head = head->next;
delete p;
```



T10: Delete an element, given p and n



T10: Delete an element, given p and n



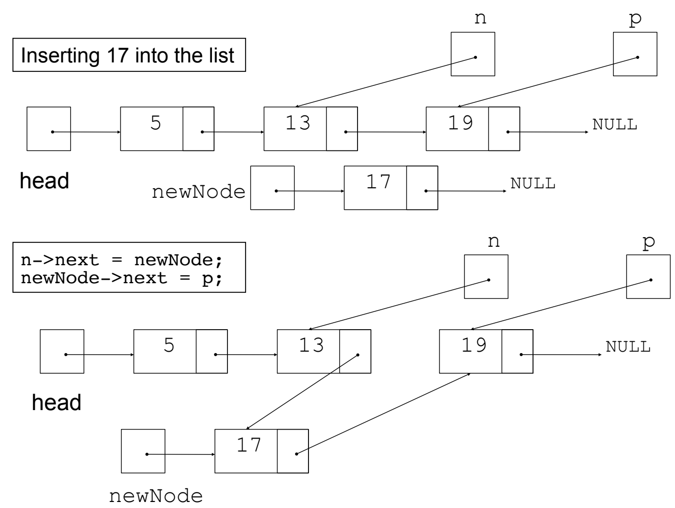
- We know how to set up p and n, see T7.
- Now just reset a link, and deallocate p:

```
n->next = p->next; //make 5 point to 19
delete p;
```

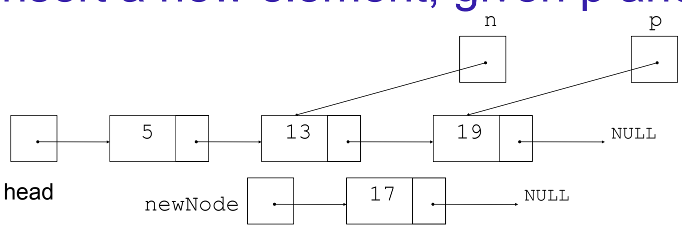
- But we should make sure p and n are not NULL:

```
if (p!==NULL) {          // p is pointing to something!
  if (n==NULL)           // p must be pointing to first node
     head = head->next;
  else                   // p and n are not NULL
     n->next = p->next;
  delete p;              // since p wasn't NULL, deallocate
}            //there is no else, if p was NULL, nothing to remove
```


T11:Insert a new element, given p and n



T11:Insert a new element, given p and n



- We know how to set up p and n, see T7.
- We know how to create a new node, see T2.
- Now reset some links (consider if p and n are null):

```
  if (n==NULL) {         // p must be pointing to first node
     head = newNode;
     newNode->next = p;
  } else  {              // n is not NULL
     n->next = newNode;
     newNode->next = p;
  }
  //if p is null, n is pointing to the last node, and it works.
```


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;
}
```

The (Abstract) List Type

- A List is an ordered collection of items of some type T:
  - each element has a position in the list
  - duplicate elements are allowed
- List is not a C++ data type. It is abstract/ conceptual. It can be implemented in various ways (using arrays or linked lists or…)
- We will first implement the list using a linked list
- Later we'll consider how to use an array to implement the list.

17.2 List operations

- Basic operations over a list:
  - **create** a new, empty list
  - **append** a value to the end of the list
  - **insert** a value within the list
  - **delete** a value (remove it from the list)
  - **display** the values in the list
  - **delete/destroy** the list (if it was dynamically allocated)

Declaring the List data type

- We will be defining a class called NumberList to represent a List data type.
  - ours will store values of type double, using a linked list.
- The class will implement the basic operations over lists on the previous slide.
- In the private section of the class we will:
  - define a struct data type for the nodes
  - define a pointer variable (head) that points to the first node in the list.

NumberList 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();        // creates an empty list
      ~NumberList();

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

Operation: **Create** the empty list

- Constructor: sets up empty list (This is T1, create an 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 (T2)
If the list is empty,
  Make head point to the new node. (T3)
Else (T8)
  Find the last node in the list
  Make the last node point to the new node

in NumberList.cpp

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

   // Create a new node and store the data in it (T2)
   ListNode *newNode = new ListNode;
   newNode->value = num;
   newNode->next = NULL;

   // If empty, make head point to new node (T3)
   if (head==NULL)
      head = newNode;

   else {
      // Append to end of non-empty list (T8)
      ListNode *p = head;  // p points to first element

      // 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;
   }
}
```


Driver to demo NumberList

- ListDriver.cpp version 1 (no output)

ListDriver.cpp

```
#include "NumberList.h"

int main() {

   // Define the list
   NumberList list;

   // Append some values to the list
   list.appendNode(2.5);
   list.appendNode(7.9);
   list.appendNode(12.6);

}
```

Traversing a Linked List

- Visit each node in a linked list, to
  - display contents, sum data, test data, etc.
- Basic process (this is T4):

```
set a pointer to point to what head points to
while the 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 = head;  //start p at the head of the list

   // 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;
}
```


Driver to demo NumberList

- ListDriver.cpp version 2

ListDriver.cpp

```
#include "NumberList.h"

int main() {

   // Define the list
   NumberList list;

   // Append some values to the list
   list.appendNode(2.5);
   list.appendNode(7.9);
   list.appendNode(12.6);

   // Display the values in the list.
   list.displayList();
}
```

Output:
2.5 7.9 12.6

Operation: destroy a List

- The destructor must “delete” (deallocate) all nodes used in the list
- To do this, use list traversal to visit each node
- ~NumberList: what’s wrong with this definition?

```
NumberList::~NumberList() {

   ListNode *p;    // traversal ptr
   p = head;       //start at head of list

   while (p) {

      delete p;     // delete current
      p = p->next;  // advance ptr
   }
}
```

destructor

- You need to save p->next before deleting p:

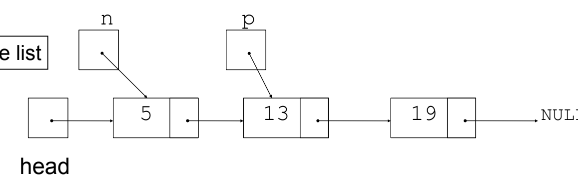
in NumberList.cpp

```
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.
- This is T7 and T10:
  - T7: Find a node AND it's previous neighbor (p&n)
  - then do T10: Delete an element, given p and n

Deleting 13 from the list



deleteNode code

in NumberList.cpp

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

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

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


Driver to demo NumberList

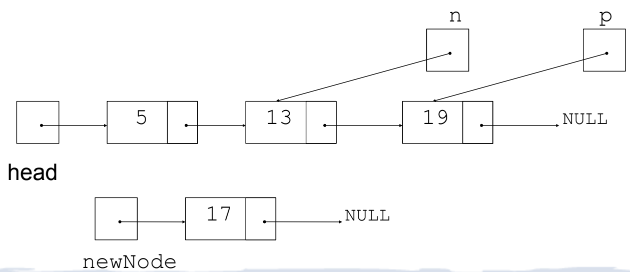
in ListDriver.cpp

```
// 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
```

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

- Inserts a new node into the middle of a list.
- This is T7 and T11:
  - T7: Find a node AND it's previous neighbor (p&n) we will make p point to the first element > 17
  - then do T11: Insert a new element, given p and n



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;
   }
}
```


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.insertNode (21.5);
   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  2.5  7.9  8.5  12.6  21.5
```


List operations, array implementation

- What if we use an array instead of a linked list? How would these operations be implemented?
  `double a[100]; int count;`
  - **create** a new, empty list: `count=0;`
  - **append** a value to the end of the list
    `a[count]=v; count++;`
  - **insert** a value within the list `shift!`
  - **delete** a value (remove it from the list) `shift!`
  - **display** the values in the list `for loop`
  - **delete/destroy** the list `unnecessary`

Advantages of linked lists over arrays

- A linked list can easily grow or shrink in size.
  - Nodes are created in memory as they are needed.
  - The programmer doesn't need to predict how many elements will be in the list.
- The amount of memory used to store the list is always proportional to the number of elements in the list.
  - For arrays, the amount of memory used is often much more than is required by the actual elements in the list.
- 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]
  - 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).